YMT219
Veri Yapıları

Doç.Dr. Erkan TANYILDIZI

Ders Kitapları ve Yardımcı Kaynaklar

- Veri Yapıları ve Algoritmalar
  - Dr. Rifat ÇÖLKESEN, Papatya yayıncılık



- Data Structures and Problem Solving Using Java
  - Mark Allen Weiss, Pearson International Edition

- Ahmet Yesevi Üniversitesi Ders Notları

- Ayrıca internet üzerinden çok sayıda kaynağa ulaşabilirsiniz.

Dersin Gereksinimleri

- Bu dersteki öğrencilerin Nesne tabanlı programlama dillerinden birisini(Java, C++, C#) veya yordamsal programlama dillerinden birisini(C, Pascal) bildiği varsayılmıştır.
- Bilinmesi gereken konular:
  - Temel veri türleri (int, float)
  - Kontrol yapısı (if else yapısı)
  - Döngüler
  - Fonksiyonlar(Methods)
  - Giriş çıkış işlemleri
  - Basit düzeyde diziler ve sınıflar

Ders İşleme Kuralları

- Derse devam zorunludur.
  Ders başlangıç saatlerine özen gösteriniz. Derse geç gelen öğrenci ara verilinceye kadar bekleyecektir.
- Her ders imza alınacaktır.
- Ödevler zamanında teslim edilecektir. Verilen tarihten sonra getirilen ödevler kabul edilmeyecektir.
- Ders esnasında lütfen kendi aranızda (veya kendi kendinize) konuşmayın, fısıldaşmayın, mesajlaşmayın v.s
- Dersi anlatan ve dinleyen kişilere lütfen saygı gösterin.
- Anlatılan, anlatılmayan, merak ettiğiniz her konuda soru sor maktan çekinmeyin.
- Cep telefonu v.b kişisel taşınabilir iletişim cihazlarınızı ders süresince mutlaka kapalı tutunuz ve sınav zamanı yanınızda bulundurmayınız.

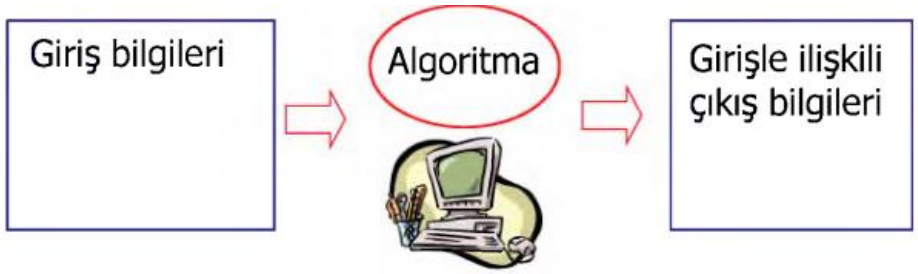
Veri Yapıları ve Modelleri

Bölüm 1

Algoritma

- **Algoritma**, bir problemin çözümünde izlenecek yol anlamına gelir. Algoritma, bir programlama dilinde (Java, C++, C# gibi) ifade edildiğinde **program** adını alır.

- Algoritma, belirli bir problemin sonucunu elde etmek için art arda uygulanacak adımları ve koşulları kesin olarak ortaya koyar. Herhangi bir giriş verisine karşılık, çıkış verisi elde edilmesi gereklidir. Bunun dışındaki durumlar algoritma değildir.



Algoritma

- Bilgisayar uygulamasında, bir yazılım geliştirirken birçok algoritmaya ihtiyaç duyulur. Örneğin,
  - arama algoritması
  - sıralama algoritması
  - matris veya vektörel işlem algoritması
  - graf algoritması
  - bir matematiksel modelin çözülmesi algoritması
- gibi birçok algoritma türü vardır ve uygulama geliştirirken, bunların biri veya birkaçı her zaman kullanılır.

Veri

- **Veri,** algoritmalar tarafından işlenen en temel elemanlardır (sayısal bilgiler, metinsel bilgiler, resimler, sesler ve girdi, çıktı olarak veya ara hesaplamalarda kullanılan diğer bilgiler…).

- Bir algoritmanın etkin, anlaşılır ve doğru olabilmesi için, algoritmanın işleyeceği verilerin düzenlenmesi gerekir.

Veri Yapısı ve Veri Modeli

- **Veri yapısı** (Data Structure) verinin veya bilginin bellekte tutulma şeklini veya düzenini gösterir.
  - Tüm programlama dillerinin, genel olarak, tamsayı, kesirli sayı, karakter ve sözcük saklanması için temel veri yapıları vardır. Bir program değişkeni bile basit bir veri yapısı olarak kabul edilebilir.
- **Veri modeli** (Data Model), verilerin birbirleriyle ilişkisel veya sırasal durumunu gösterir; problemin çözümü için kavramsal bir yaklaşım yöntemidir denilebilir.
  - Bilgisayar ortamında uygulanacak tüm matematik ve mühendislik problemleri bir veri modeline yaklaştırılarak veya yeni veri modelleri tanımlaması yapılarak çözülebilmektedir.

Veri Yapılarına Neden İhtiyaç Vardır?

- Bilgisayar yazılımları gün geçtikçe daha karmaşık bir hal almaktadır. Yazılımların programlanması ve yönetimi zorlaşmaktadır.
  - Örneğin 8 milyar sayfanın indekslenmesi (Google)
- Veri yapıları, daha etkin ve daha doğru program yazmayı sağlar.
- İyi bir yazılım için gereksinimler:
  - Temiz bir tasarım
  - Kolay bakım ve yönetim
  - Güvenilir
  - Kolay kullanımlı
  - Hızlı algoritmalar

Veri Yapılarına Neden İhtiyaç Vardır?

- **Örnek**
- Her biri satır başına ortalama 10 kelimeden ve yine ortalama 20 satırdan oluşan 3000 metin koleksiyonu olduğunu düşünelim.
  - →600,000 kelime
- Bu metinler içinde “dünya” kelimesi ile eşleşecek bütün kelimeleri bulmak isteyelim
- Doğru eşleştirme için yapılacak karşılaştırmanın 1 sn. sürdüğünü varsayalım.
- Ne yapılmalıdır?

Veri Yapılarına Neden İhtiyaç Vardır?

- **Örnek**
- **Çözüm. 1:**
- Sıralı eşleştirme: 1 sn. x 600,000 kelime= 166 saat
- **Çözüm. 2:**
- İkili Arama (Binary searching):
  - - kelimeler sıralanır
  - - sadece tek yarıda arama yapılır
  - toplam adım sayısı $\log_2 N = \log_2 600000$ yaklaşık 20 adım (çevrim) 20 sn.

- 20 saniye veya 166 saat!

Veri Yapılarının Sınıflandırılması

- Veri yapıları,
  - **Temel Veri Yapıları**
  - **Tanımlamalı (Bileşik) Veri Yapıları**
- Temel veri yapıları, daha çok programlama dilleri tarafından doğrudan değişken veya sabit bildirimi yapılarak kullanılır.
- Tanımlamalı veri yapıları, kendisinden önceki tanımlamalı veya temel veri yapıları üzerine kurulurlar; yani, önceden geçerli olan veri yapıları kullanılarak sonradan tanımlanırlar.

Temel Veri Yapıları

- Tüm programlama dillerinin, genel olarak, karakter, tamsayı, kesirli sayı ve sözcük (karakter katarı) saklanması için temel veri yapıları vardır. Veri yapısı, aslında, ham olarak 1 ve 0'lardan oluşan verinin yorumlanmasını belirleyen biçimleme (formating) düzenidir. Örneğin, 62 sayısının ikili tabandaki karşılığı, 111110 olarak bellekte saklanır.
- Temel veri yapıları aşağıdaki gibi sınıflanabilir:

- **Karakter** (Character) - A, B,C, @, ?, >,∞…
- **Tamsayı** (Integer) - 1923, 19, 29, 23, 5, - 153
- **Gerçel Sayı** (Real Number) - 3.14, 2.71, 1.53×10^8…
- **Katar** (String) - "Üniversite", "Oğuzhan", "Can", "A"
- **Dizi/Matris** (Array/Matrix) - [23 4 1923], $\begin{bmatrix} 19 & 23 \\ 29 & 5 \end{bmatrix}$

Tanımlamalı Veri Yapıları

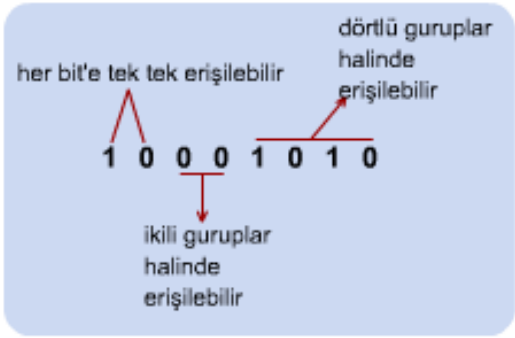
- Tanımlamalı veri yapısı, temel veya daha önceden tanımlanmış veri yapılarının kullanılıp yeni veri yapıları oluşturulmasıdır. Üç değişik şekilde yapılabilir:
  - **Topluluk (Struct) Oluşturma:** Birden çok veri yapısının bir araya getirilip yeni bir veri yapısı ortaya çıkarmaktır. (Java dilinde sınıflar)
  - **Ortaklık (Union) Oluşturma:** Birden çok değişkenin aynı bellek alanını kullanmasını sağlayan veri yapısı tanımlamasıdır. Ortaklıkta en fazla yer işgal eden veri yapısı hangisi ise, ortaklık içerisindeki tüm değişkenler orayı paylaşır.
  - **Bit Düzeyinde Erişim:** Verinin her bir bit'i üzerinde diğerlerinden bağımsız olarak işlem yapılması olanağı sunar.

- Her birinin kullanım amacı farklı farklı olup uygulamaya göre bir tanesi veya hepsi bir arada kullanılabilir. Genel olarak, en çok kullanılanı topluluk oluşturmadır; böylece birden fazla veri yapısı bir araya getirilip/paketlenip yeni bir veri yapısı/türü ortaya çıkarılır.

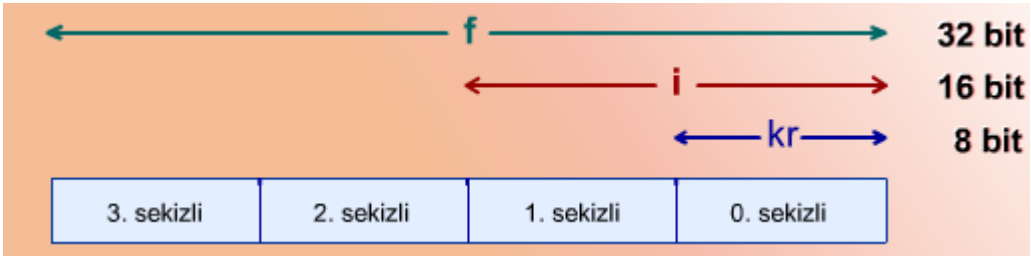
Tanımlamalı Veri Yapıları

- C dilinde tanımlamalı veri yapılarına örnek aşağıda verilmiştir.

```
struct kimlik {
   char ad[15];
   char soyad[20];
   int yas;
   char adres[50];
  };
```

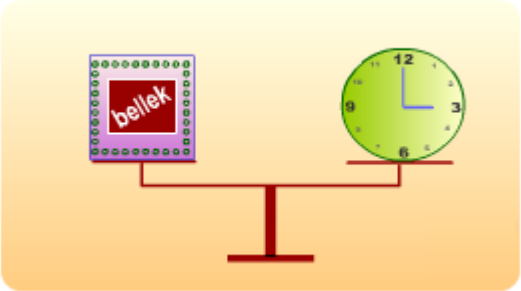



```
union paylas {
   int i;
   flat f;
   char kr;
  };_
```



Veri Modelleri

- Veri modelleri, tasarımı yapılacak programın en uygun ve etkin şekilde olmasını sağlar ve daha baştan programın çalışma hızı ve bellek gereksinimi hakkında bilgi verir. Çoğu zaman, programın çalışma hızıyla bellek gereksinimi miktarı doğru orantılıdır denilebilir.
- Veri modelleri, genel olarak, aşağıdaki gibi verilebilir:
  - **Bağlantılı Liste (Link List)**
  - **Ağaç (Tree)**
  - **Graf (Graph)**
  - **Durum Makinası (State Machine)**
  - **Veritabanı-İlişkisel (Database Relational)**
  - **Ağ Bağlantı (Network Connection)**



Hız ile Bellek Miktarı arasında denge kurulması

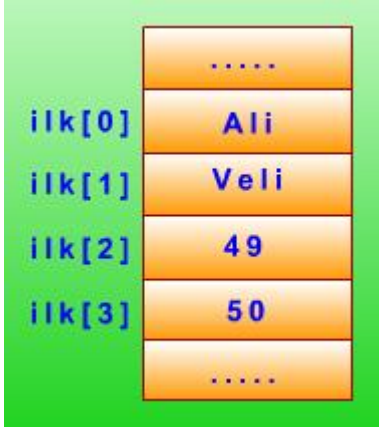
Liste ve Bağlantılı Liste Veri Modeli

- Liste veri modeli, aynı kümeye ait olan verilerin bellekte art arda tutulması ilkesine dayanır. Veriler belirli bir düzen içerisinde (sıralı vs.) olabilir veya olmayabilir; önemli olan tüm verilerin art arda gelen sırada tutulmasıdır.

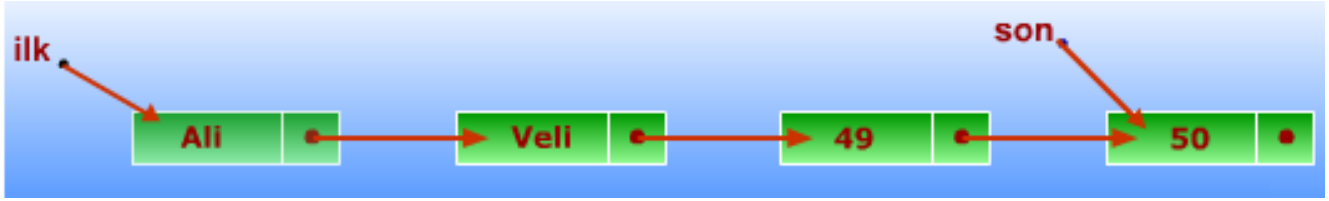


Liste ve Bağlantılı Liste Veri Modeli

- En yalın liste veri modeli bir boyutlu dizi üzerinde tutulanıdır. Böylesi bir listeye eleman ekleme işlemi oldukça kolaydır; genel olarak, yeni gelen elemanlar listenin sonuna eklenir. Yalın listede bir sonraki eleman hemen o elemanın işgal ettiği bellek alanından sonradır.



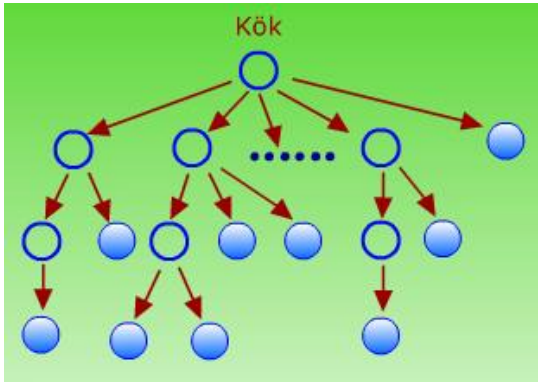
- Bağlantılı liste (link list) ise, elemanların kendi değerlerine ek olarak bir de bağlantı bilgisinin kullanılmasıyla sağlanır; bağlantı bilgisi bir sonraki elemanın adresi niteliğindedir.



Ağaç Veri Modeli

- Ağaç veri modeli, düğümlerden ve dallardan oluşur; düğümlerde verilerin kendileri veya bir kısmı tutulurken, dallar diğer düğümlere olan bağlantı ilişkilerini gösterir. Ağaç veri modeli, özellikle kümenin büyük olduğu ve arama işleminin çok kullanıldığı uygulamalarda etkin bir çözüm sunar.
- En üstteki düğüm kök (root), kendisine alttan hiçbir bağlantının olmadığı düğüm yaprak (leaf), diğerleri de ara düğüm (internal node) olarak adlandırılır. Bir düğüme alttan bağlı düğümlere çocuk (child), üsten bağlı düğüme de o düğümün ailesi (parent) denilir.

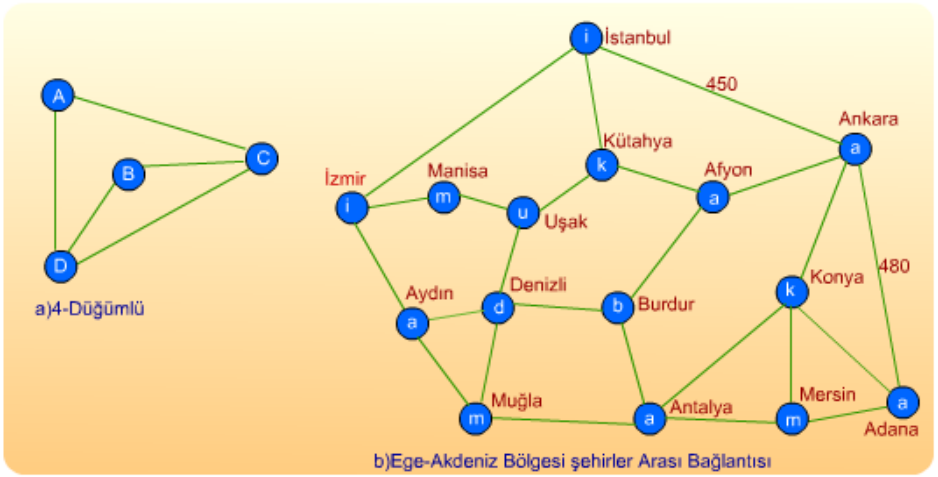
- Hızlı Erişim
- Esnek Bellek Kullanımı
- Arama-Ekleme Maliyetleri
- Tasarım Esnekliği
- Çok Farklı problemlere Model



Graf Veri Modeli

- Graf veri modeli, aynı kümeye ait olan verilerin şekilde görüldüğü gibi düğümler, ayrıtlar (kenarlar) ve bunların birleştirilmesinden oluşur. Düğümler birleşme noktasını ayrıtlar da düğümlerin bağlantı ilişkisini gösterir. Verilerin kendileri veya bir kısmı hem düğümlerde hem de ayrıtların bilgi kısmında tutulabilir.
  - Graflar, yazılım dünyasından önemli bir yere sahiptir. Örneğin, bir şehrin trafik altyapısından en yüksek akışın sağlanması, taşıma şirketinin en verimli taşıma şekli veya network bağlantılarında yüksek başarım elde edilmesi gibi problemler.

- Modelleme Esnekliği
- Üzerine Tasarlanmış Algoritma Çokluğu
- En kısa Yol (Shortest Path)
- Yol Ağacı (Spanning Tree)
- Greddy Yöntemi
- DFS ve BFS Yaklaşımları
- Kuruksal Algoritması



Durum Makinası Veri Modeli

- Durum makinası veri modeli, bir sistemin davranışını tanımlamak ve ortaya çıkarmak için kullanılan bir yaklaşım şeklidir; işletim sistemlerinde, derleyici ve yorumlayıcılarda, kontrol amaçlı yazılımlarda, sistemin davranışını durumlara indirger ve durumlar arası geçiş koşullarıyla sistemi ortaya koyar.
  - Durum makinası, yazılım uygulamasında birçok alanda kullanılabilir. Örneğin bir robot kolunun hareketi, şifre çözme, gerçek zamanlı işletim sistemlerinde proses kontrolü ve genel olarak kontrol alt sistemlerinin yazılımla uygulamayı başarılı bir şekilde sonuçlandırma durumlarında çözüm olur.

Durum Makinası Veri Modeli

- Durum makinası veri modeli şeklen yönlü graflara benzer, ancak, birleşme noktaları graflarda olduğu gibi düğüm olarak değil de **durum**, ayrıtlar da **geçiş eğrileri** olarak adlandırılır. Durumlar arasındaki geçişler, sistemin o ana kadar ki durumlarına ve giriş parametrelerine bağlıdır.



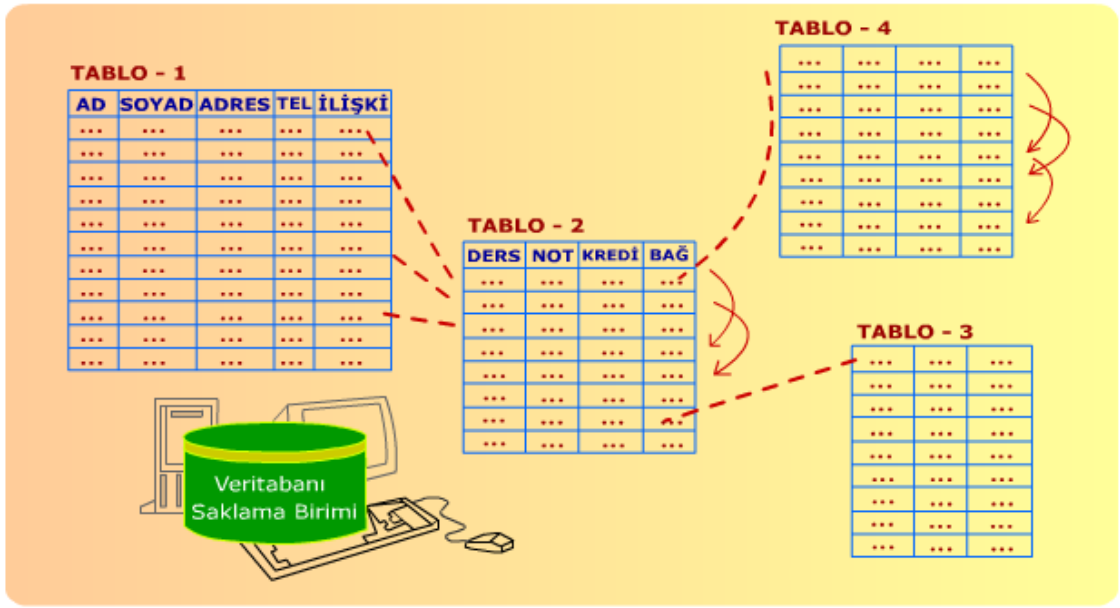


Aşk Sözcüğü İçin Durum Makinası

Veritabanında İlişkisel Veri Modeli

- Veritabanı ilişkisel veri modeli veritabanı uygulamalarında var olan dört beş sınıftan birisidir; veriler şekilde gösterildiği gibi tablolar üzerinden kurulan ilişkilere dayanmaktadır.

- Bilgilerin Düzenli Saklanması
- Hızlı Arama/Sorulama
- Veriler Arasında İlişki Oluşturulması
- İnternet Tarayıcısı Ortamında Bilgi Sorgulama
- Koruma
- Verilerin Arşivlenmesi



Yukarıda tipik olarak veritabanı ilişkisel veri modelindeki tablolar ve alanların durumu görülmektedir.

Veritabanında İlişkisel Veri Modeli

- SQL (Structured Query Language), sorgulama dili kullanılarak veritabanı üzerinde sorgulama yapılabilir. Access, Microsoft SQL, ORACLE, SYBASE, Ingres gibi birçok veritabanı yönetim sistemleri ilişkisel veri modelini desteklemektedir.

- Veritabanı yönetim sistemleri, veritabanı oluşturma, tablo yaratma, alanları tanımlama gibi işlerin başarılı bir şekilde sonuçlandırmasını ve genel olarak veritabanı yönetimini sağlarlar.

Ağ Veri Modeli

- Ağ veri modeli, katmalı ağ mimarilerinde, bilgisayarlar arasında eş katmanlar (peer layers) düzeyinde veri alış-verişini sağlayan dilim (segment), paket (packet) ve çerçeve yapılarını ortaya koyar ve iletişim için gerekli davranışı tanımlar. Veri haberleşmesinde hemen hemen tüm mimariler katmanlı yapıdadır. Tüm mimariler örnek temsil eden OSI 1, başvuru modeli 7, TCP/IP (**T**ransmission **C**ontrol **P**rotocol / **I**nternet **P**rotocol) protokol kümesi 4 katmanlıdır.



Veri Modelleri

- Bu derste aşağıdaki veri modelleri detaylı ele alınacaktır;
- **Liste**
  - Sonlu sayıda elemandan oluşan ve elemanları doğrusal sırada yerleştirilmiş veri modeli. Herhangi bir elemanına erişimde sınırlama yoktur.
- **Yığıt veya Yığın**
  - Elemanlarına erişim sınırlaması olan, liste uyarlı veri modeli (Last In First Out-LIFO listesi).
- **Kuyruk**
  - Elemanlarına erişim sınırlaması olan, liste uyarlı veri modeli. (First In First Out-FIFO listesi).
- **Ağaç**
  - Doğrusal olmayan belirli niteliklere sahip veri modeli
- **Çizge (Graph)**
  - Köşe adı verilen düğümleri ve kenar adı verilen köşeleri birbirine bağlayan bağlantılardan oluşan doğrusal olmayan veri modeli

Veri Modeli	Artıları	Eksileri
Dizi	Hızlı ekleme ve çok hızlı erişim(indis biliniyorsa).	Yavaş arama, yavaş silme ve sabit boyut.
Sıralı Dizi	Sıralanmamış diziye göre daha hızlı arama.	Yavaş silme ve sabit boyut.
Yığın	Son giren, ilk çıkar(last-in, first-out) erişimi sağlar.	Diğer öğelere yavaş erişim.
Kuyruk	İlk giren, ilk çıkar(first-in, first-out) erişimi sağlar.	Diğer öğelere yavaş erişim.
Bağlı Liste	Hızlı ekleme ve silme.	Yavaş arama.
Hash Tablosu	Hızlı ekleme ve anahtar bilindiğinde çok hızlı erişim.	Yavaş silme, anahtar bilinmediğinde yavaş erişim ve verimsiz bellek kullanımı.
Küme(Heap)	Hızlı ekleme ve silme.	Diğer öğelere yavaş erişim. Başta en büyük öğeye erişim.
İkili Ağaç	Hızlı arama, ekleme ve silme(ağaç dengeli kalmışsa).	Silme algoritması karmaşık.
Graf	Gerçek-dünya problemlerini modelleyebilmesi.	Bazı algoritmaları yavaş çalışmakta ve karmaşıklığı yüksek.

Algoritmik Program Tasarımı ve Analizi

Bölüm 2

Algoritmik Program Tasarımı Nedir?

- **Verilen bir problemin bilgisayar ortamında çözülecek biçimde adım adım ortaya koyulması ve herhangi bir programlama aracıyla kodlanması sürecidir.**
- Çözüm için yapılması gereken işlemler hiçbir alternatif yoruma izin vermeksizin sözel olarak ifade edilir.
- Verilerin, bilgisayara hangi çevre biriminden girileceğinin, problemin nasıl çözüleceğinin, hangi basamaklardan geçirilerek sonuç alınacağının, sonucun nasıl ve nereye yazılacağının sözel olarak ifade edilmesi biçiminde de tanımlanabilir.

Algoritmaların Özellikleri

- Bir algoritmanın taşıması gereken beş tane temel özelliği vardır.

- **1. Giriş (Input)**
  - Bir algoritmanın sıfır veya daha fazla giriş değişkeni vardır. Giriş değişkenleri algoritma işlemeye başlamadan önce, algoritmaya verilen değerler kümesidir veya değer kaydetmesi için verilen hafıza bölgesidir.
- **2. Belirlilik (Definiteness)**
  - Bir algoritmanın her adımı için kesin olarak ne iş yapacağı belirlenmelidir ve belirsizlik olmamalıdır. Her durum için hangi işlem gerçekleştirilecekse, o açık olarak tanımlanmalıdır.

Algoritmaların Özellikleri

- **3.Çıkış (Output))**
  - Her algoritmanın bir veya daha fazla çıkış değeri vardır. Çıkış değerleri ile giriş değerleri arasında bağıntılar vardır.
- **4.Etkililik (Efficiency)**
  - Doğal olarak her algoritmanın etkili olarak işlem yapması beklenir. Olabildiğince hızlı çalışmalıdır, olabildiğince az hafıza kullanmalıdır.
  - Bunun anlamı yapılan işlemler yeterince temel işlemler olacak ve sınırlı zaman süresince işleyip bitmelidir.

Algoritmaların Özellikleri

- **5. Sınırlılık (Boundedness)**
  - Her algoritma sınırlı sayıda çalışma adımı sonunda bitmelidir.

- Bir algoritma için sınırlılık çok önemlidir. Aynı işlemi yapan iki algoritmadan biri bir milyar adımda bitiyor olsun ve diğeri de yüz adımda bitiyor olsun. Bu durumda yüz adımda biten algoritma her zaman daha iyidir. Bunun anlamı sınırlılık kavramı ile anlatılmak istenen mümkün olan en az sayıda adım ile işlemin bitirilmesidir. Diğer bazı kriterler ise algoritmanın bilgisayar ortamına aktarılabilme özelliği, basitliği, vb. gibi özelliklerdir.

Algoritma Süreci

- Tasarım (design)
- Doğruluğunu ispat etme (validation)
- Analiz (analysis)
- Uygulama (implementation)
- Test

Kaba-Kod (Pseudo Code)

- Kaba-kod, bir algoritmanın yarı programlama kuralı, yarı konuşma diline dönük olarak ortaya koyulması, tanımlanması, ifade edilmesidir.

- Kaba-kod, çoğunlukla, bir veri yapısına dayandırılmadan algoritmayı genel olarak tasarlamaya yardımcı olur.

Gerçek Kod

- Algoritmanın herhangi bir programlama diliyle, belirli bir veri yapısı üzerinde gerçekleştirilmiş halidir.
- Bir algoritmanın gerçek kodu, yalnızca, tasarlandığı veri yapısı üzerinde çalışır.
- Bir algoritma kaba-kod ile verilirse gerçek kod verilmesin den daha kolay anlaşılır.

Kaba-kod: temel gösterim

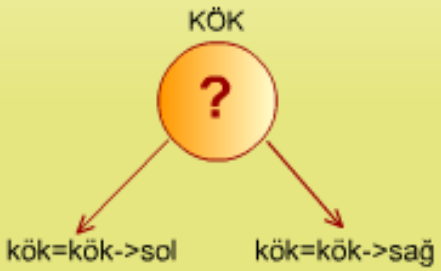
- 1. Bir değer atamak için genellikle " **:=** veya ←" kullanılır, "**=**" işareti ise eşitlik kontrolü için kullanılır.
- 2. Metot, fonksiyon, yordam isimleri:
  Algoritma Adı ({parametre listesi})
- 3. Program yapısı şu şekilde tanımlanır::
  - Karar yapıları: **if ... then ... else ...**
  - while döngüleri: **while ... do {döngü gövdesi}**
  - Tekrar döngüleri: **repeat {döngü gövdesi} until ...**
  - for döngüleri: **for ... do {döngü gövdesi}**
  - Dizi indeksleri: **A[i]**
- 4. Metotların çağrılması: **Metot adı ({değişken listesi})**
- 5. Metotlardan geri dönüş: **return değer**

Algoritmaların kaba-kod olarak ifade edilmesi

- Örnek: Bir dizideki elemanların toplam ve çarpımını hesaplayan algoritmayı kaba-kod kullanarak tanımlayınız.
  - **Toplam ve Çarpım Hesapla** (dizi, toplam, çarpım)
  - **Girdi:** n sayıdan oluşan dizi.
  - **Çıktı:** dizi elemanlarının toplam ve çarpım sonucu
  - **for** i← 1 **to** n **do**
  - toplam← toplam + dizi[i]
  - çarpım← çarpım* dizi[i]
  - **end for**

Kaba-kod ve Gerçek Kod

```
/* İkili arama ağacına düğüm ekleme algoritmasının kaba-kodu */
if (kök boş ağacı gösteriyorsa)
   Düğümü köke ekle;
else {
   if (eklenen kökten küçükse)
    sol altağaca dallan; (kök=kökün sol çocuğu adresi )
    başa dön;
   else
    sağ altağaca dallan; (kök=kökün sağ çocuğu adresi )
    başa dön;
   }
```




```
/* Ikili arama agacina dügüm ekleme fonksiyonu */
void ekle (agacKok, eklenen )
          AGAC2 *agacKok, *eklenen;
{
  if (agacKok==NULL )
     kok=eklenen;
     else {
        if (eklenen->bilgi <= agacKok->bilgi)
          {
           if (agacKok->sol==NULL)
              agacKok->sol=eklenen;
           else ekle (agacKok->sol, eklenen );
          }
        else
          {
             if (agacKok->sag==NULL )
                agacKok->sag=eklenen;
             else ekle (agacKok->sag, eklenen );
          }
     }
}
```

Algoritma Analizi

- Algoritmanın icra edilmesi sırasında duyacağı kaynak miktarının tahmin edilmesine **Algoritma Analizi** denir.
- Kaynak denildiğinde, bellek, iletişim bant genişliği, mantık kapıları akla gelebilir, fakat en önemli kaynak algoritmanın icra edilebilmesi için gerekli olan zamanın tahmin edilmesidir.
- Algoritma analizi, tasarlanan program veya fonksiyonun belirli bir işleme göre matematiksel ifadesini bulmaya dayanır.
- Burada temel hesap birimi seçilir ve programın görevini yerine getirebilmesi için bu işlemden kaç adet yapılması gerektiğini bulmaya yarayan bir bağıntı hesaplanır.
- Eğer bu bağıntı zamanla ilgiliyle çalışma hızını, bellek gereksinimiyle ilgiliyse bellek gereksinimi ortaya koyar.

Algoritma Analizi

- Neden algoritmayı analiz ederiz?
  - Algoritmanın performansını ölçmek için
  - Farklı algoritmalarla karşılaştırmak için
  - Daha iyisi mümkün mü? Olabileceklerin en iyisi mi?
- Algoritmayı nasıl analiz ederiz?
  - Çalışma zamanı (Running Time)-T(n)
  - Karmaşıklık (Complexity) -Notasyonlar

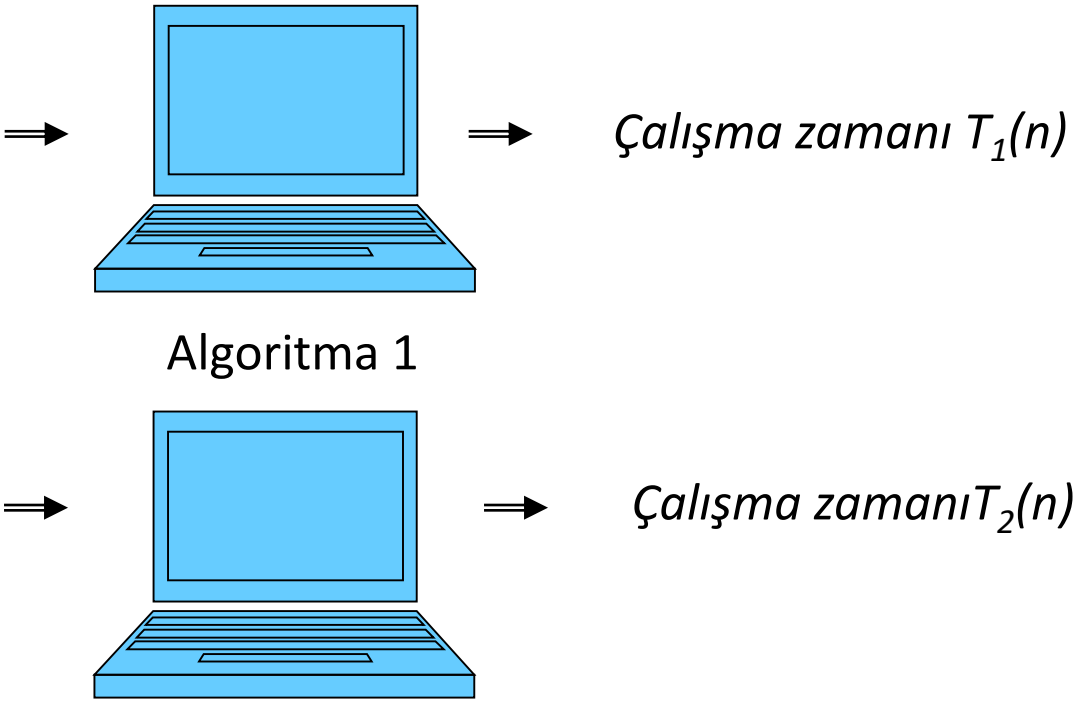
Çalışma Zamanı Analizi (Running Time)

- **Çalışma Zamanı**; Bir programın veya algoritmanın işlevini yerine getirebilmesi için, temel kabul edilen işlevlerden kaç adet yürütülmesini veren bir bağıntıdır ve T(n) ile gösterilir.

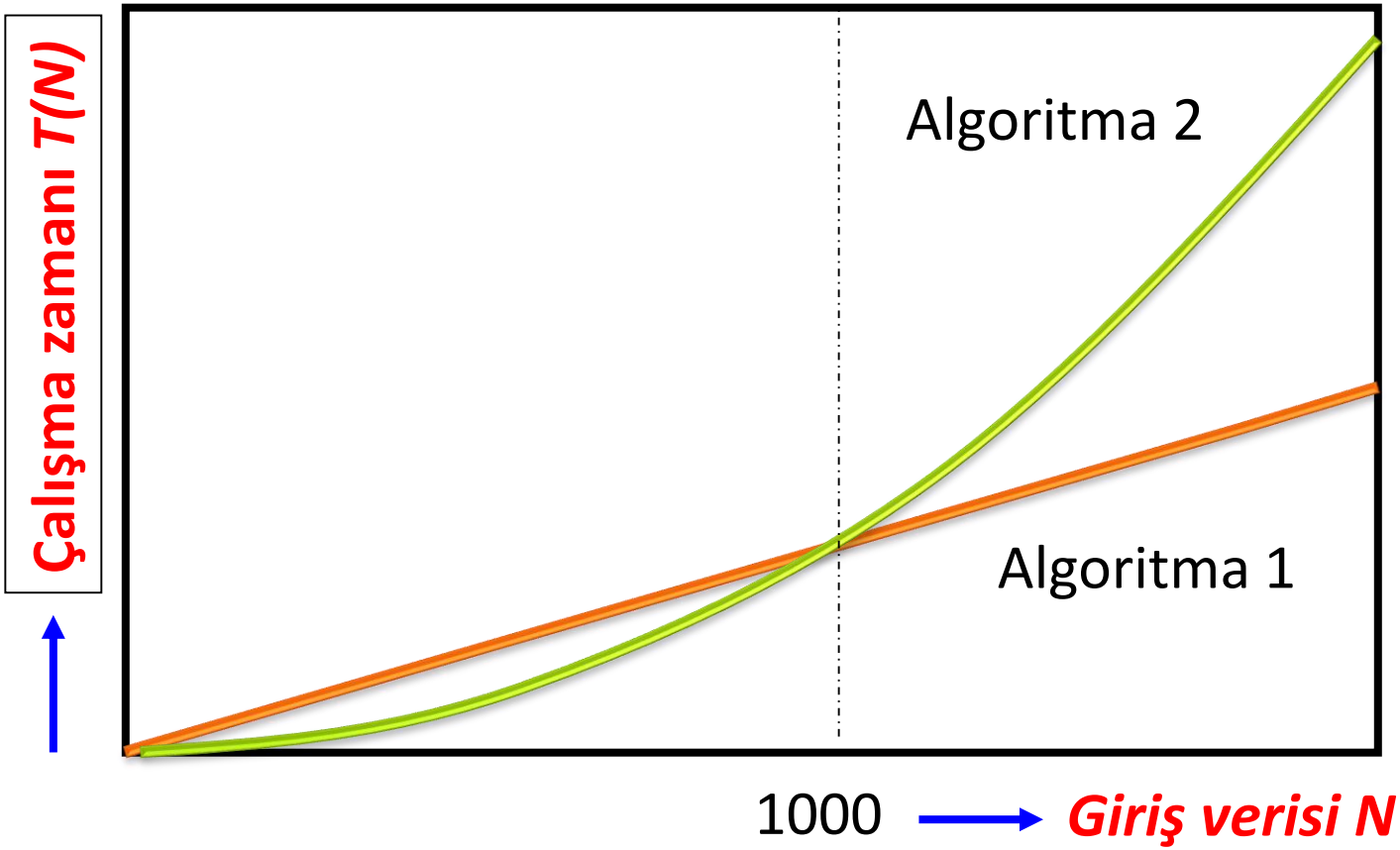
- Temel hesap birimi olarak, programlama dilindeki deyimler seçilebildiği gibi döngü sayısı, toplama işlemi sayısı, atama sayısı, dosyaya erişme sayısı gibi işler de temel hesap birimi olarak seçilebilir.

Çalışma Zamanı Analizi

- Algoritma 1, $T_1(N) = 1000*N$
- Algoritma 2, $T_2(N) = N^2$



Çalışma Zamanı Analizi



Çalışma Zamanlarının Özeti

N	T1	T2
10	10^{-2} sec	10^{-4} sec
100	10^{-1} sec	10^{-2} sec
1000	1 sec	1 sec
10000	10 sec	100 sec
100000	100 sec	10000 sec

N değerinin 1000'den küçük olduğu durumlarda iki algoritma arasındaki çalışma zamanı ihmal edilebilir büyüklüktedir.

Analiz

- Çalışma zamanının kesin olarak belirlenmesi zordur
  - **Giriş verilerine bağlı olan en iyi durum** (best case)
  - **Ortalama durum** (average case); hesaplanması zordur
  - **Diğerlerine göre en kötü durum** (worst case); hesaplanması kolaydır
- Bunun için çeşitli notasyonlardan faydalanılır.

Aritmetik Ortalama için *T(n)* Hesabı

- **Örnek 1:** Aşağıda bir dizinin aritmetik ortalamasını bulan ve sonucu çağırana gönderen **bulOrta()** fonksiyonun kodu verilmiştir. Bu fonksiyonun yürütme zamanını gösteren **T(n)** bağıntısını ayrık C dili deyimlerine göre belirleyiniz.

```
        float bulOrta(float A[], int n) {
        {
        float ortalama, toplam=0;
        int k ;
1-      for(k=0;k<n;k++)
2-              toplam+=A[k];
3-      ortalama=toplam/n
4-      return ortalama;
        }
```

Aritmetik Ortalama için *T(n)* Hesabı

- **Çözüm 1:**

Temel Hesap Birimi	Birim Zaman (Unit Time)	Frekans(Tekrar) (Frequency)	Toplam (Total)
float bulOrta(float A[], int n)	-	-	-
{	-	-	-
float ortalama, toplam=0;	-	-	-
int k ;	-	-	-
1- for(k=0;k<n;k++)	1,1,1	1, (n+1), n	2n+2
2- toplam+=A[k];	1	n	n
3- ortalama=toplam/n	1	1	1
4- return ortalama;	1	1	1
}	-	-	-
			T(n)=3n+4

En Küçük Eleman Bulma için *T(n)* Hesabı

- **Örnek 2:** Aşağıda bir dizi içerisindeki en küçük elamanı bulan **bulEnKucuk()** adlı bir fonksiyonun kodu görülmektedir. Bu fonksiyonun yürütme zamanını gösteren *T(n)* bağıntısı ayrık C dili deyimlerine göre belirleyiniz.

```
        float bulEnKucuk(float  A[])
        {
        float enkucuk;
        int k ;
1-      enkucuk=A[0];
2-      for(k=1;k<n;k++)
3-              if (A[k]<enkucuk)
4-                      enkucuk=A[k];
5-      return enkucuk;
        }
```

En Küçük Eleman Bulma için *T(n)* Hesabı

- **Çözüm 2:**

Temel Hesap Birimi	Birim Zaman (Unit Time)	Frekans(Tekrar) (Frequency)	Toplam (Total)
float bulEnKucuk(float A[]) {	-	-	-
float enkucuk;	-	-	-
int k ;	-	-	-
1- enkucuk=A[0];	1	1	1
2- for(k=1;k<n;k++)	1,1,1	1, n, (n-1)	2n
3- if (A[k]<enkucuk)	1	n-1	n-1
4- enkucuk=A[k];	1	n-1	n-1
5- return enkucuk;	1	1	1
}	-	-	-
			T(n)=4n

Matris Toplama için *T(n)* Hesabı

Temel Hesap Birimi	Birim Zaman (Unit Time)	Frekans(Tekrar) (Frequency)	Toplam (Total)
void toplaMatris (A,B,C) {	-	-	-
int A[n][m], B[n][m], C[n][m];	-	-	-
int i,j ;	-	-	-
1- for(i=0;i<n;i++)	1,1,1	1,(n+1),n	2n+2
2- for(j=0;j<m;j++)	1,1,1	n(1,(m+1),m)	n(2m+2)
3- C[i][j]=A[i][j]+B[i][j];	1	nm	nm
}	-	-	-
			T(n,m)=3nm+4n+2
n=m ise			$T(n)=3n^2+4n+2$

$$T(N)=\sum_{i=1}^{N}\sum_{j=1}^{N}1=\sum_{i=1}^{N}N=N*N=N^2$$

T(n) Genel Hesabı

Temel Hesap Birimi	Cost (Maliyet)	Frekans(Tekrar) (Frequency)
int i, j, x=0;	c1	1
for(i=n; i>=1; i=i-1)	c2	n
for(j=1; j<=i; j=j+1)	c3	n*(n-1)/2
x=i+j;	c4	n*(n-1)/2
T(n)=c1*1+c2*n+(c3+c4)*n*(n-1)/2		
T(n)=c1*1+c2*n+(c3+c4)*(n^2/2-n/2)		
T(n)=c1*1+ (c2-(c3+c4)/2) *n+(c3+c4)*(n^2/2)		

Karmaşıklık (Copmlexity)

- Karmaşıklık; bir algoritmanın çok sayıda parametre karşısındaki değişimini gösteren ifadelerdir. Çalışma (Yürütme) zamanını daha doğru bir şekilde bulmak için kullanılırlar.

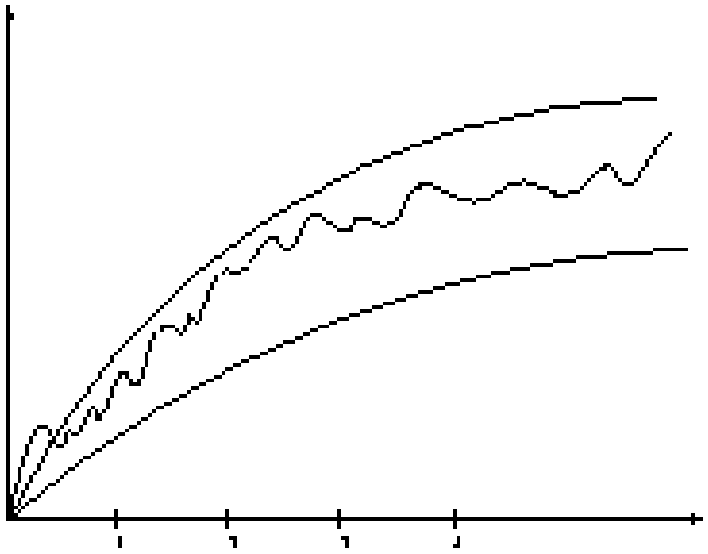
- Genel olarak, az sayıda parametreler için karmaşıklıkla ilgilenilmez; eleman sayısı ***n*** nin sonsuza gitmesi durumunda algoritmanın maliyet hesabının davranışını görmek veya diğer benzer işleri yapan algoritmalarla karşılaştırmak için kullanılır.

- Karmaşıklığı ifade edebilmek için asimtotik ifadeler kullanılmaktadır.

- Bu amaçla O(n) (O notasyonu), Ω(n) (Omega notasyonu), θ(n) (Teta notasyonu) gibi tanımlamalara baş vurulur.

Karmaşıklık (Copmlexity)

- Strateji: Alt ve üst limitlerin bulunması



Üst limit – O(n)

Algoritmanın gerçek fonksiyonu

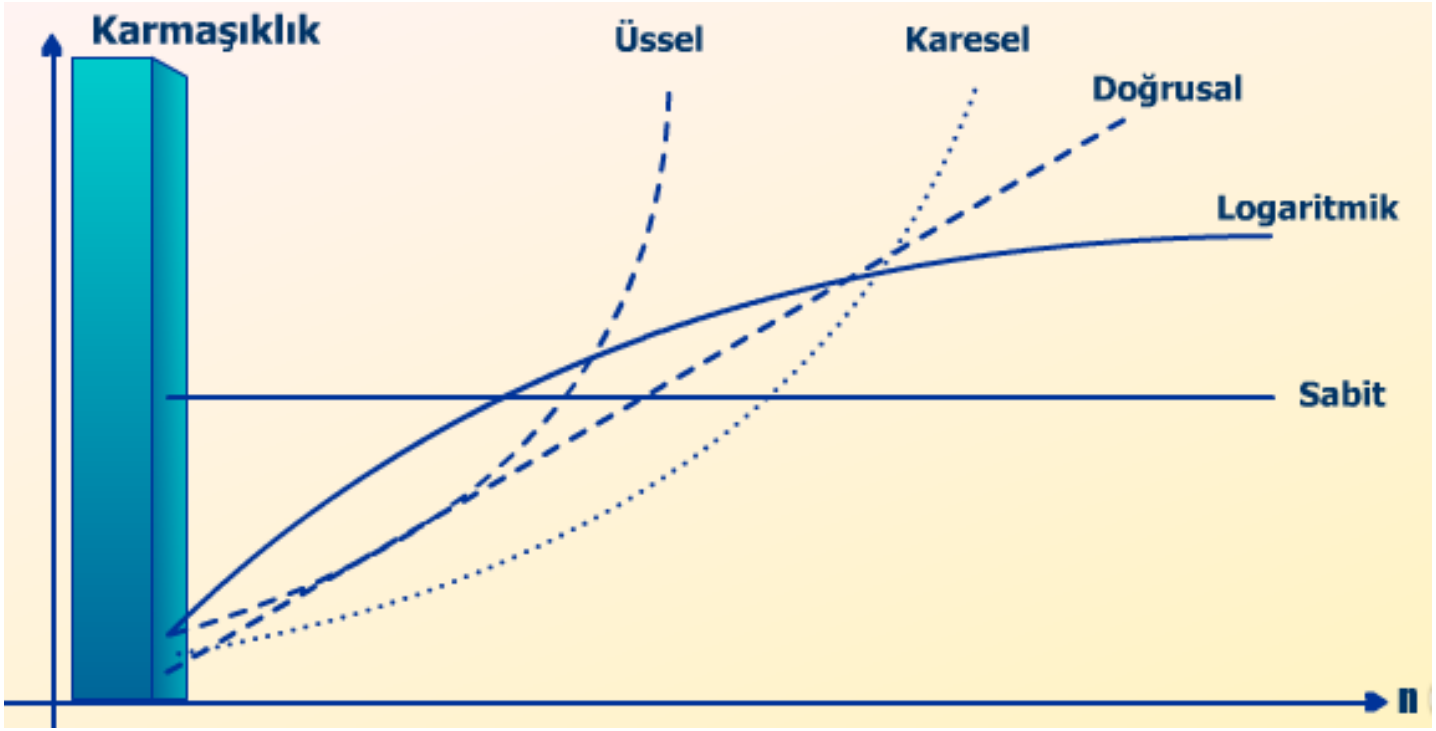
Alt limit- Ω (n)

Karşılaşılan Genel Fonksiyonlar

Maliyet artar

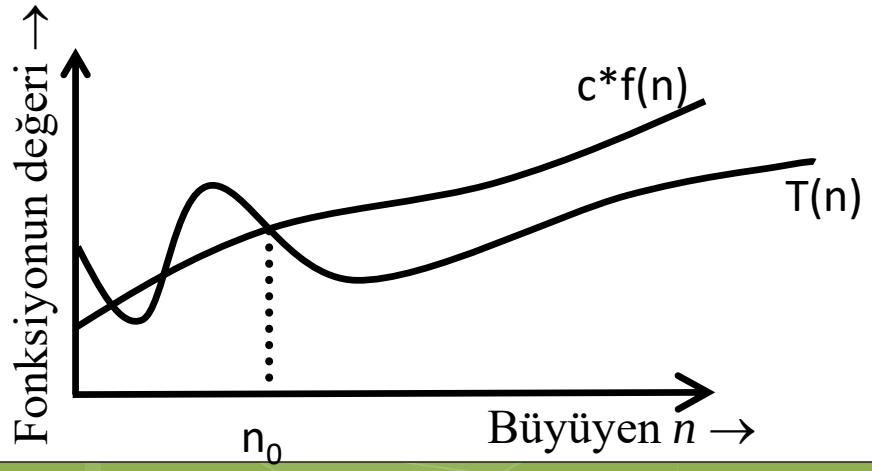
Big-Oh	Değişim Şekli
O(1)	Sabit, komut bir veya birkaç kez çalıştırılır. Yenilmez!
O(log_n)	Logaritmik, Büyük bir problem, her bir adımda sabit kesirler tarafından orijinal problemin daha küçük parçalara ayrılması ile çözülür. İyi hazırlanmış arama algoritmalarının tipik zamanı
O(n)	Doğrusal, Küçük problemlerde her bir eleman için yapılır. Hızlı bir algoritmadır. N tane veriyi girmek için gereken zaman.
O($nlog_n$)	Doğrusal çarpanlı logaritmik. Çoğu sıralama algoritması
O(n^2)	Karasel. Veri miktarı az olduğu zamanlarda uygun (N<1000)
O(n^3)	Kübik. Veri miktarı az olduğu zamanlarda uygun (N<1000)
O(2^n)	İki tabanında üssel.Veri miktarı çok az olduğunda uygun (N<=20)
O(10^n)	On tabanında üssel
O(n!)	Faktöriyel

Karmaşıklık (Copmlexity)



Büyük-Oh(Big-Oh) Notasyonu: Asimptotik Üst Sınır (En kötü durum analizi)

- Bir algoritmanın çalışma süresi,
- T(n)=O(f(n)) **O**, bir fonksiyon değil, sadece gösterimdir.
- ***T(n)*** $\leq$ ***c f(n)*** *ve* ***n*** $\geq$ ***n*$_0$** *koşullarını sağlayan* ***c*** *ve* ***n*$_0$** *değerleri varsa* ***T(n)*** $\leq$ ***c f(n)*** *ifadesi doğrudur.*
- *f(n), T(n)'in asimtotik üst limiti olarak adlandırılır.*
- *T(n)=O(f(n))*



Büyük-Oh(Big-Oh) Notasyonu: Asimptotik Üst Sınır

- Örnek: T(n) = 2n+5 is $O(n^2)$ Neden?
  - $n>=n_0$ şartını sağlayan tüm sayılar için $T(n) = 2n+5 <= c*n^2$ şartını sağlayan c ve n_0 değerlerini arıyoruz.
  - $n>=4$ için $2n+5 <= 1*n^2$
    - c = 1, no = 4
  - $n>=3$ için $2n+5 <= 2*n^2$
    - c = 2, no = 3
  - Diğer c ve n_0 değerleri de bulunabilir.

Büyük-Oh(Big-Oh) Notasyonu: Asimptotik Üst Sınır

- Örnek: T(n) = n(n+1)/2 → O(?)
  - T(n) = $n^2/2 + n/2$ → $O(N^2)$. Neden?
  - n >= 1 iken $n^2/2 + n/2 <= n^2/2 + n^2/2 <= n^2$
  - Böylece, T(n)=n*(n+1)/2 <= 1* n^2 for all n >=1
    - c=1, no=1
  - Not: T(n) ayrıca $O(n^3)$ tür.

O Notasyonunun Önemi

- İki algoritma karşılaştırılırken zaman mertebesinden konuşulur. Mertebesi büyük olanın daha yavaş olduğu kolaylıkla anlaşılır.

- Makineler arasındaki fark, katsayılar olarak düşünüldüğünde **$O(7n^2)$** yerine **$O(n^2)$** ifadesi geçerli olur. Dolayısıyla sabitler ihmal edilebilir ve birimlerden kurtulur

O Notasyonu

- O notasyonunda yazarken en basit şekilde yazarız.
  - Örneğin
    - $3n^2+2n+5 = O(n^2)$
  - Aşağıdaki gösterimlerde doğrudur fakat kullanılmaz.
    - $3n^2+2n+5 = O(3n^2+2n+5)$
    - $3n^2+2n+5 = O(n^2+n)$
    - $3n^2+2n+5 = O(3n^2)$

O Notasyonu-Örnek 1

- $3n^2+2n+5 = O(n^2)$ ifadesinin doğru olup olmadığını ispatlayınız.

$10\ n^2 = 3n^2 + 2n^2 + 5n^2$

$\geq 3n^2 + 2n + 5$ için $n \geq 1$

$c = 10, n_0 = 1$

Çözüm kümesini sağlayan kaç tane n_0 ve c çifti olduğu önemli değildir. Tek bir çift olması notasyonun doğruluğu için yeterlidir.

O Notasyonu-Örnek 2

- $T(n)=O(7n^2+5n+4)$ olarak ifade edilebiliyorsa, T(n) fonksiyonu aşağıdakilerden herhangi biri olabilir.
- $T(n)=n^2$
- $T(n)=4n+7$
- $T(n)=1000n^2+2n+300$
- $T(n)= O(7n^2+5n+4) =O(n^2)$

O notasyonu- Örnek 3

- Fonksiyonların harcadıkları zamanları O notasyonuna göre yazınız.

- $f1(n) = 10\ n + 25\ n^2$
- $f2(n) = 20\ n \log n + 5\ n$
- $f3(n) = 12\ n \log n + 0.05\ n^2$
- $f4(n) = n^{1/2} + 3\ n \log n$

- $O(n^2)$
- $O(n \log n)$
- $O(n^2)$
- $O(n \log n)$

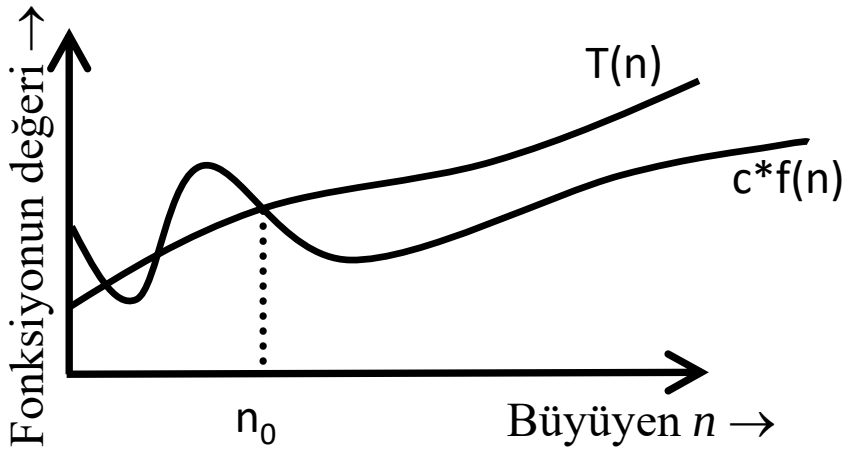
- $2n^2 = O(n^3)$: $2n^2 \leq cn^3 \Rightarrow 2 \leq cn \Rightarrow c = 1$ and $n_0 = 2$
- $n^2 = O(n^2)$: $n^2 \leq cn^2 \Rightarrow c \geq 1 \Rightarrow c = 1$ and $n_0 = 1$
- $1000n^2+1000n \neq O(n^2)$: Şart doğrumu

- $n = O(n^2)$:

$n \leq cn^2 \Rightarrow cn \geq 1 \Rightarrow c = 1$ and $n_0 = 1$

Ω Notasyonu- Asimtotik Alt Limit

- O notasyonun tam tersidir.
- Her durumda **T(n) ≥ c f(n)** ve **n ≥ n_0** koşullarını sağlayan pozitif, sabit **c** ve **n_0** değerleri bulunabiliyorsa **T(n)=Ω(f(n))** ifadesi doğrudur.
- f(n), T(n)'in asimtotik alt limiti olarak adlandırılır.



Ω notasyonu-Örnek

- T(n) = 2n + 5 → Ω(n). Neden?
  - 2n+5 >= 2n, tüm n >= 1 için
- T(n) = 5*n^2 - 3*n → Ω(n^2). Neden?
  - 5*n^2 - 3*n >= 4*n^2, tüm n >= 4 için

- $7n^2+3n+5 = O(n^4)$
- $7n^2+3n+5 = O(n^3)$
- $7n^2+3n+5 = O(n^2)$
- $7n^2+3n+5 = \Omega(n^2)$
- $7n^2+3n+5 = \Omega(n)$
- $7n^2+3n+5 = \Omega(1)$

Examples

- **$5n^2 = \Omega(n)$**

  $\exists\ c, n_0$ such that: $0 \le cn \le 5n^2 \Rightarrow$ $cn \le 5n^2 \Rightarrow c = 1$ and $n_0 = 1$

- **$100n + 5 \neq \Omega(n^2)$**

  $\exists\ c, n_0$ such that: $0 \le cn^2 \le 100n + 5$

  $100n + 5 \le 100n + 5n\ (\forall\ n \ge 1) = 105n$

  $cn^2 \le 105n \Rightarrow n(cn - 105) \le 0$

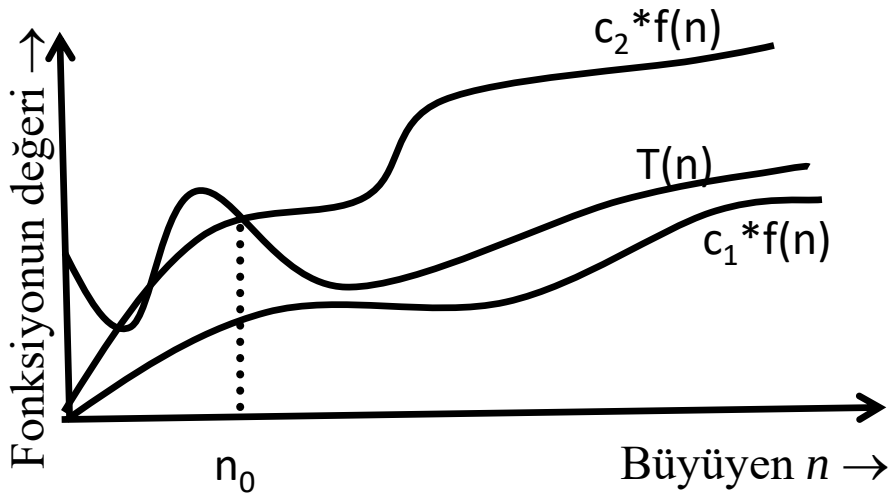
  Since n is positive $\Rightarrow cn - 105 \le 0 \Rightarrow n \le 105/c$

  $\Rightarrow$ contradiction: n cannot be smaller than a constant

- **$n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(\log n)$**

Θ Notasyonu

- Her durumda $c_1 f(n) \geq T(n) \geq c_2 f(n)$ ve $n \geq n_0$ koşullarını sağlayan pozitif, sabit c_1,c_2 ve n_0 değerleri bulunabiliyorsa $T(n) = \Theta(f(n))$ ifadesi doğrudur.



Θ notasyonu- Örnek

- $T(n) = 2n + 5 \rightarrow \Theta(n)$. Neden?

  $2n <= 2n+5 <= 3n$, tüm $n >= 5$ için

- $T(n) = 5*n^2 - 3*n \rightarrow \Theta(n^2)$. Neden?
  - $4*n^2 <= 5*n^2 - 3*n <= 5*n^2$, tüm $n >= 4$ için

Examples

- $n^2/2 - n/2 = \Theta(n^2)$
  - $\frac{1}{2}\, n^2 - \frac{1}{2}\, n \leq \frac{1}{2}\, n^2 \; \forall n \geq 0 \Rightarrow c_2 = \frac{1}{2}$
  - $\frac{1}{2}\, n^2 - \frac{1}{2}\, n \geq \frac{1}{2}\, n^2 - \frac{1}{2}\, n * \frac{1}{2}\, n \; (\forall n \geq 2) = \frac{1}{4}\, n^2 \Rightarrow c_1 = \frac{1}{4}$

- $n \neq \Theta(n^2)$: $c_1\, n^2 \leq n \leq c_2\, n^2 \Rightarrow$ only holds for: $n \leq 1/c_1$
- $6n^3 \neq \Theta(n^2)$: $c_1\, n^2 \leq 6n^3 \leq c_2\, n^2 \Rightarrow$ only holds for:

  $n \leq c_2/6$

N değerini keyfi olarak belirlemek imkansızdır. Çünkü c_2 sabittir.

Another example

- Prove that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
  - **Determine c_1, c_2 and n_0 such that**

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

$$\frac{1}{2} - \frac{3}{n} \le c_2 \rightarrow n \ge 1, c_2 \ge \frac{1}{2}$$

$$c_1 \le \frac{1}{2} - \frac{3}{n} \rightarrow n \ge 7, c_1 \le \frac{1}{14}$$

For any polynomial $p(n) = \sum_{i=0}^{d} a_i n^i$
p(n) = $\Theta(n^d)$

  - c_1=1/14, c_2=1/2, n_0=7

Büyük-Oh, Theta, Omega

- **n**'nin büyük olduğu ve sabitlerin elendiği durumlarda.
- O(g(n)) düşünürsek f(n) ile "eşit veya büyük"
  - Üstten sınır: f(n) ile "aynı hızda veya hızlı büyür"
- Ω(g(n)) düşünürsek f(n) ile "eşit veya küçük"
  - Alttan sınır: f(n) ile "aynı hızda veya yavaş büyür"
- Θ(g(n)) düşünürsek f(n) ile "eşit"
  - Alttan ve Üsten sınır : büyüme oranları eşit
- Her bağıntı için alt ve üst sınırlar aynı zamanda belirlenemiyorsa veya bu sınırlardan sadece biri belirlenebiliyorsa, o sınıra göre notasyon gösterimi yapılır.

Sıkça Yapılan Hatalar

- Karmaşıklığı bulmak için sadece döngüleri saymakla yetinmeyin.
  - 2 içi içe döngünün 1 den n^2 kadar döndüğünü düşünürsek karmaşıklık $O(n^4)$ olur.
- $O(2n^2)$ veya $O(n^2+n)$ gibi ifadeler kullanmayın.
  - Sadece baskın terim kullanılır.
  - Öndeki sabitler kaldırılır.
- İç içe döngüler karmaşıklığı direk etkilerken art arda gelen döngüler karmaşıklığı etkilemez.

En iyi zaman (T_{best})

- Bir algoritma için, yürütme zamanı, maliyet veya karmaşıklık hesaplamalarında en iyi sonucun elde edildiği duruma “en iyi zaman” denir.

- Örneğin bir dizide yapılan aramanın en iyi durumu, aranan elemanın dizinin ilk elemanı olmasıdır.

En iyi zaman (Tbest)

	Birim Zaman (Unit Time)	Frekans (Frequency)	Toplam (Total)
i=1	1	1	1
while (a[i] ≠key) and (i≤ N)	2	1	2
i++	1	0	0
if (a[i]=key) print found(veya return i) else print not found(veya return 0)	2	1	2
	$T_{best}(N) = 5 = \Theta(1)$		

En kötü zaman (T_{worst})

- En kötü zaman, tüm olumsuz koşulların oluşması durumunda algoritmanın çözüm üretmesi için gerekli hesaplama zamanıdır.

- Örneğin bir dizi içinde arama yapılması durumunda en kötü durum aranan elemanın dizide olmamasıdır. Çünkü aranan elemanın dizide olmadığının anlaşılması için bütün elemanlara tek tek bakılması gerekir.

En kötü zaman (T_{worst})

	Birim Zaman (Unit Time)	Frekans (Frequency)	Toplam (Total)
i=1	1	1	1
while (a[i] ≠key) and (i≤ N)	2	N+1	2N+2
i++	1	N	N
if (a[i]=key) print found(veya return i) else print not found(veya return 0)	2	1	2
	$T_{worst}(N) = 3N+5 = \Theta(N)$ $T(N) \neq \Theta(N)$ $T(N) = O(N)$		

Ortalama zaman (Taverage)

- Ortalama zaman, giriş parametrelerin en iyi ve en kötü durum arasında gelmesi ile ortaya çıkan durumda harcanan zamandır.

- Bu işletim süresi, her girdi boyutundaki tüm girdilerin ortalamasıdır. n elemanın her birinin aranma olasılığının eşit olduğu varsayıldığında ve liste dışından bir eleman aranmayacağı varsayıldığında ortalama işletim süresi (n+1)/2'dir. İkinci varsayım kaldırıldığında ortalama işletim süresi [(n+1)/2,n] aralığındadır (aranan elemanların listede olma eğilimine bağlı olarak). Ortalama durum analizi basit varsayımlar yapıldığında bile zordur ve varsayımlar da gerçek performansın iyi tahmin edilememesine neden olabilir.

Ortalama zaman ($T_{average}$)

	Birim Zaman (Unit Time)	Frekans (Frequency)	Toplam (Total)
i=1	1	1	1
while (a[i] ≠key) and (i≤ N)	2	k+1	2k+2
i++	1	k	k
if (a[i]=key) print found(veya return i) else print not found(veya return 0)	2	1	2
			=3k+5

Ortalama zaman ($T_{average}$)

$$T_{average}(N) = \sum_{k=0}^{N-1} \frac{1}{2N}(3k+5) \quad + \quad \frac{1}{2}(3N+5)$$

$$T_{average}(N) = \frac{1}{2N}5 + \frac{1}{2N}(3+5) + \frac{1}{2N}(3.2+5) + \ldots \quad \frac{1}{2N}(3(N-1)+5) + \frac{1}{2}(3N+5)$$

$$T_{average}(N) = \frac{1}{2N}(3\frac{N(N-1)}{2} + 5N) + \frac{3}{2}N + \frac{5}{2}$$

$$T_{average}(N) = \frac{3}{4}N - \frac{3}{4} + \frac{5}{2} + \frac{3}{2}N + \frac{5}{2} \cong 2N + 4 = \theta(N)$$

Algoritma Analizinde Bazı Kurallar

- **For Döngüsü:**
  - Bir **For** döngüsü için yürütme zamanı en çok **For** döngüsünün içindeki (test dahil) deyimlerin yinelenme sayısı kadardır**.**
- **İç içe döngüler (Nested Loops)**
  - İç içe döngülerde grubunun içindeki deyimin toplam yürütme zamanı, deyimlerin yürütme sürelerinin bütün **For** döngülerinin boyutlarının çarpımı kadardır. Bu durumda analiz içten dışa doğru yapılır.

$$T(N) = \sum_{i=1}^{N} \sum_{j=1}^{N} 1 = \sum_{i=1}^{N} N = N * N = N^2$$

Algoritma Analizinde Bazı Kurallar

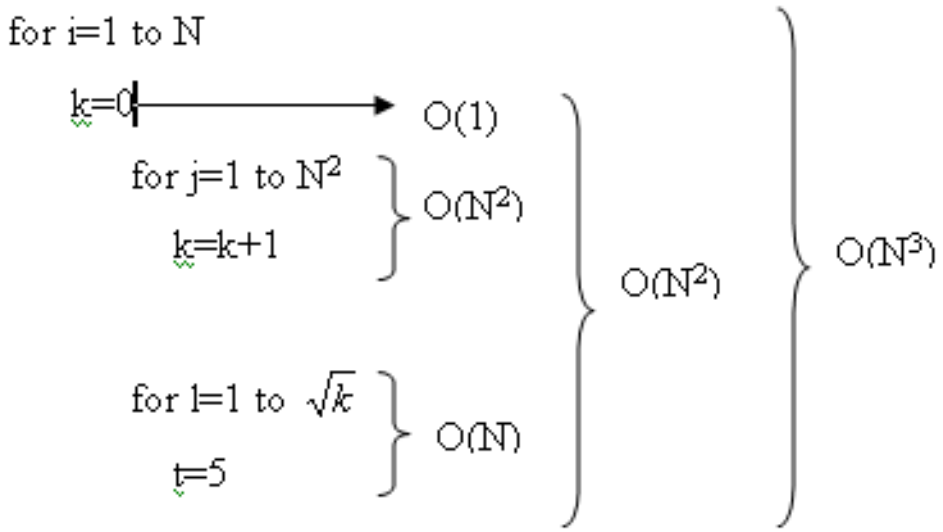
- **For Döngüsü:**

```
For i=1 to N                                              } O(N^3)
        For j=1 to N^2                    } O(N^2)        }
                k=k+1 -> O(1)             }               }
```

$T(n)=c1*N+ (c2+c3) *N^3$

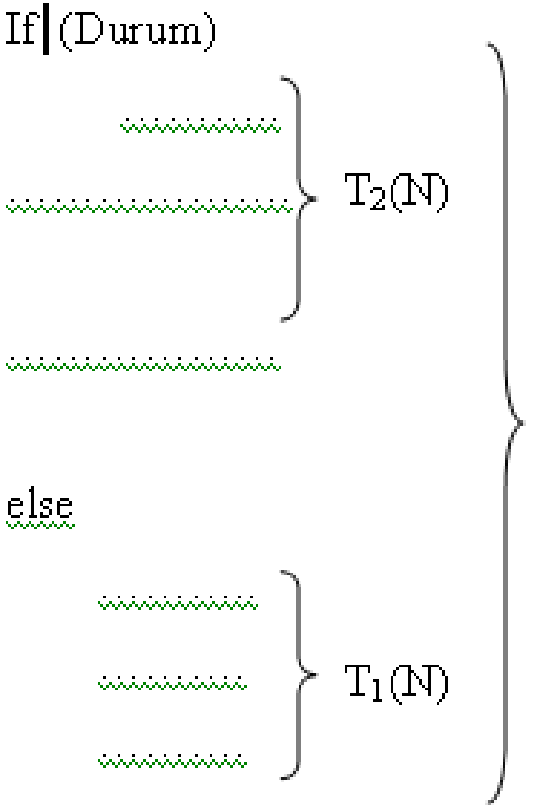
Algoritma Analizinde Bazı Kurallar

- **For Döngüsü:** Ardışık deyimlerin toplam yürütme zamanını bulabilmek için sadece toplama yapılır.



$$T_1 = O(1)$$
$$T_2 = O(N^2)$$
$$T_3 = O(N)$$
$$\Big\} T_1 + T_2 + T_3 = O(N^2)$$

Algoritma Analizinde Bazı Kurallar



Durumun maksimum olduğu değerle durumu hesaplama süresinin toplamıdır.

Dolayısıyla değerlendirme zamanı $T+\max(T_1(N),T_2(N))$

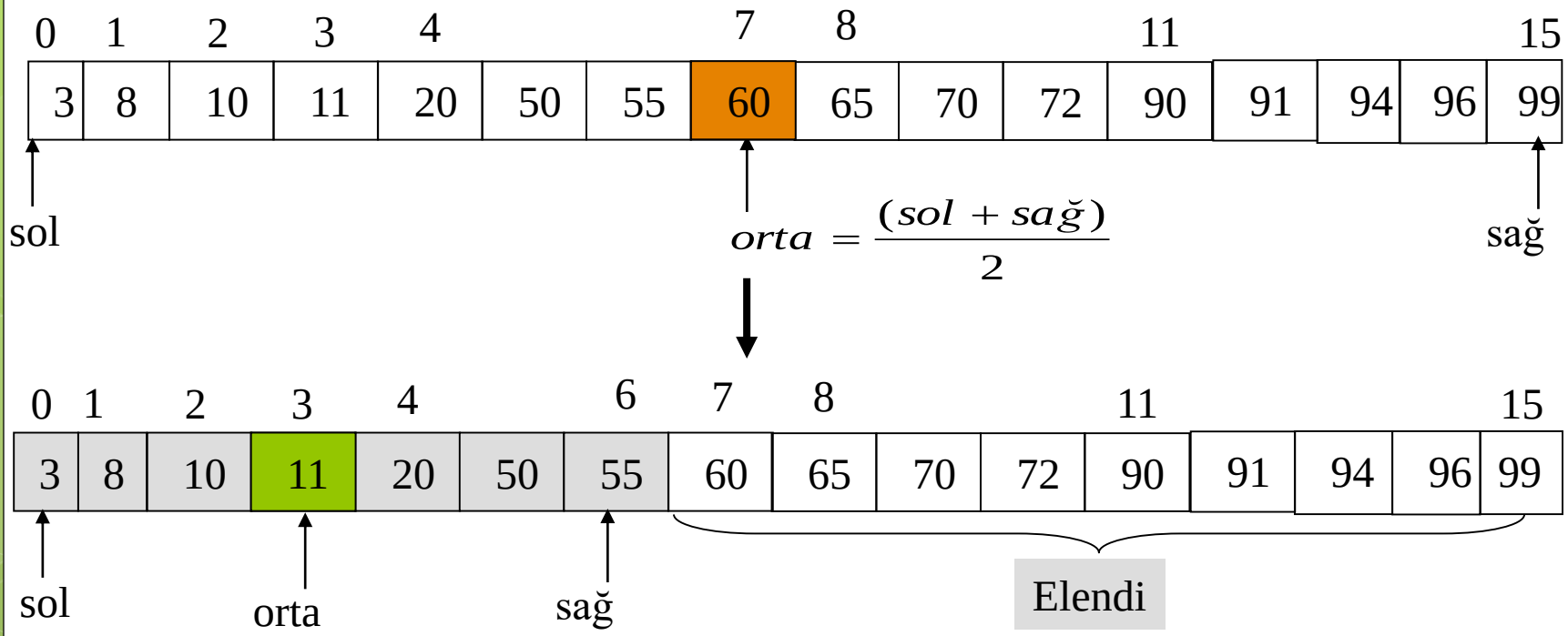
$O(\log N)+\max(O(N),O(N^2)) = O(N^2)$ olur.

Algoritma Analizinde Bir Örnek: İkili Arama Algoritması

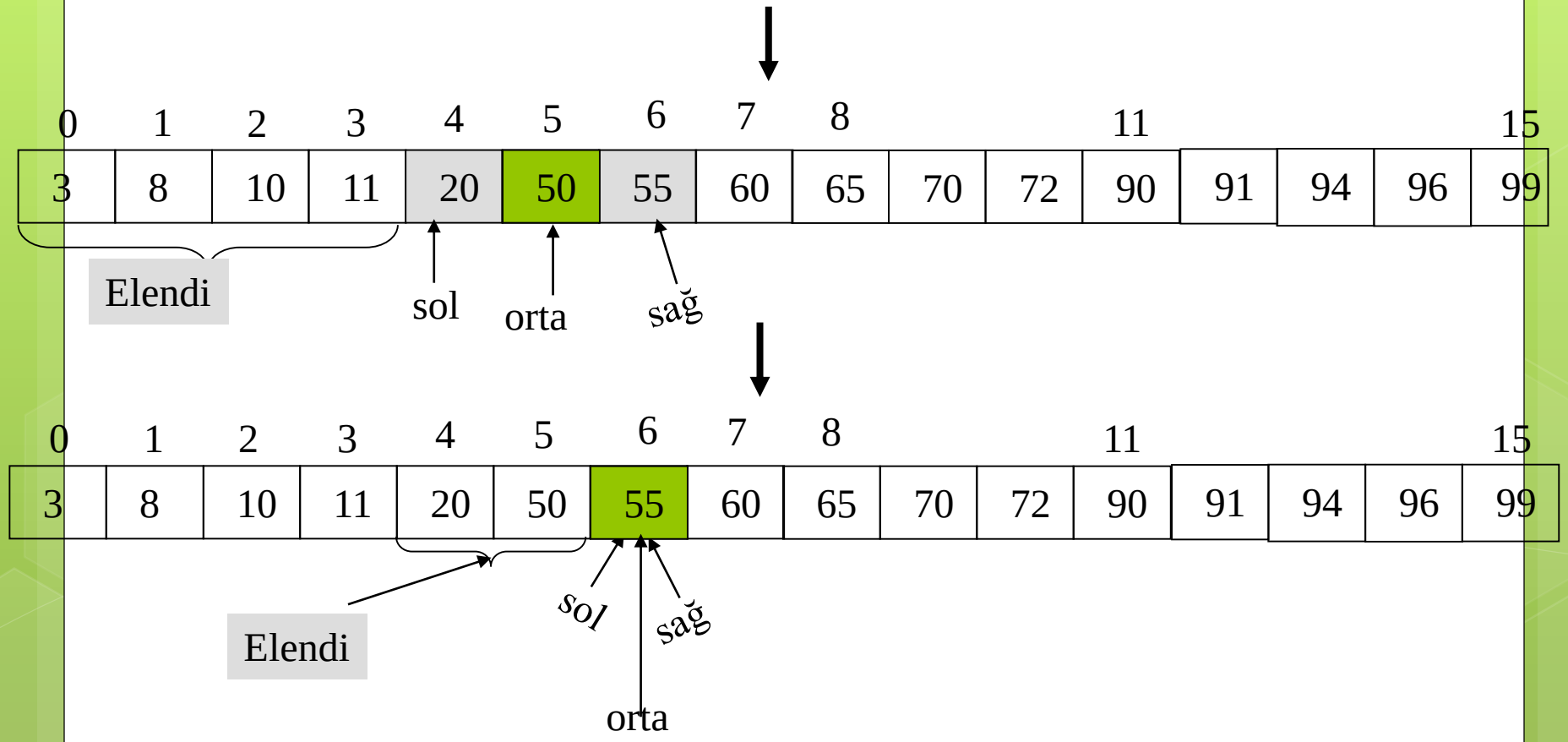
- İkili arama algoritması, sıralı dizilerde kullanılan bir arama metodur. Algoritma iteratif veya tekrarlamalı (recursive) olabilir.
- İterasyon ifadeleri (döngü) belirli bir koşul sağlanana kadar, verilen talimatların çalıştırılmasını sağlar. Belirlenen koşul, “for” döngüsündeki gibi, önceden belirlenebilir veya “while-do” döngüsünde olduğu gibi net olarak belirlenmemiş, uçu açık da olabilir.
- Aşağıda iteratif olarak tasarlanmış ikili arama algoritmasına bir örnek verilmiştir.

Örnek : İkili Arama

- Dizi sıralanmış olduğundan, dizinin ortasında bulunan sayı ile aranan sayıyı karşılaştırarak arama boyutunu yarıya düşürülür ve bu şekilde devam edilir.
- Örnek: 55'i arayalım

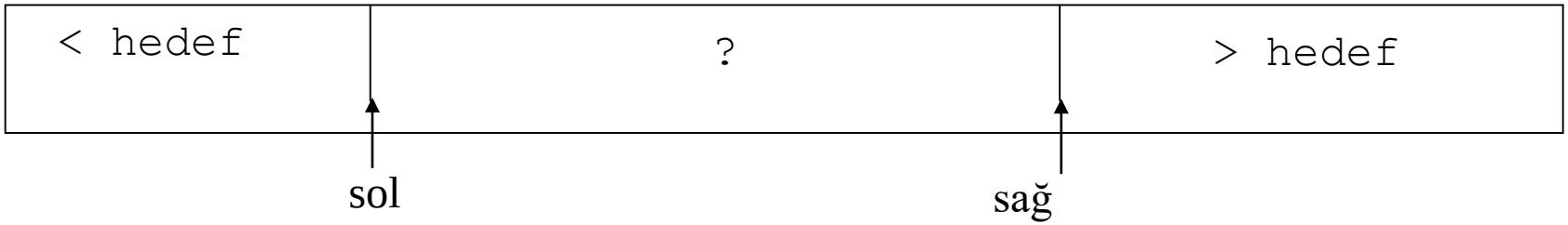


İkili arama (devam)



- 55'i bulduk→ Başarılı arama
- 57'yi aradığımızda, bir sonraki işlemde başarısız bir şekilde sonlanacak.

İkili Arama (devam)



- Hedefi ararken herhangi bir aşamada, arama alanımızı “sağ” ile “sol” arasındaki alana kısıtlamış oluyoruz.
- “sol” ’un solunda kalan alan hedeften küçüktür ve bu alan arama alanından çıkarılır.
- “sağ” ın sagında kalan alan hedeften büyüktür ve bu alan arama alanından çıkarılır.

İkili Arama - Algoritma

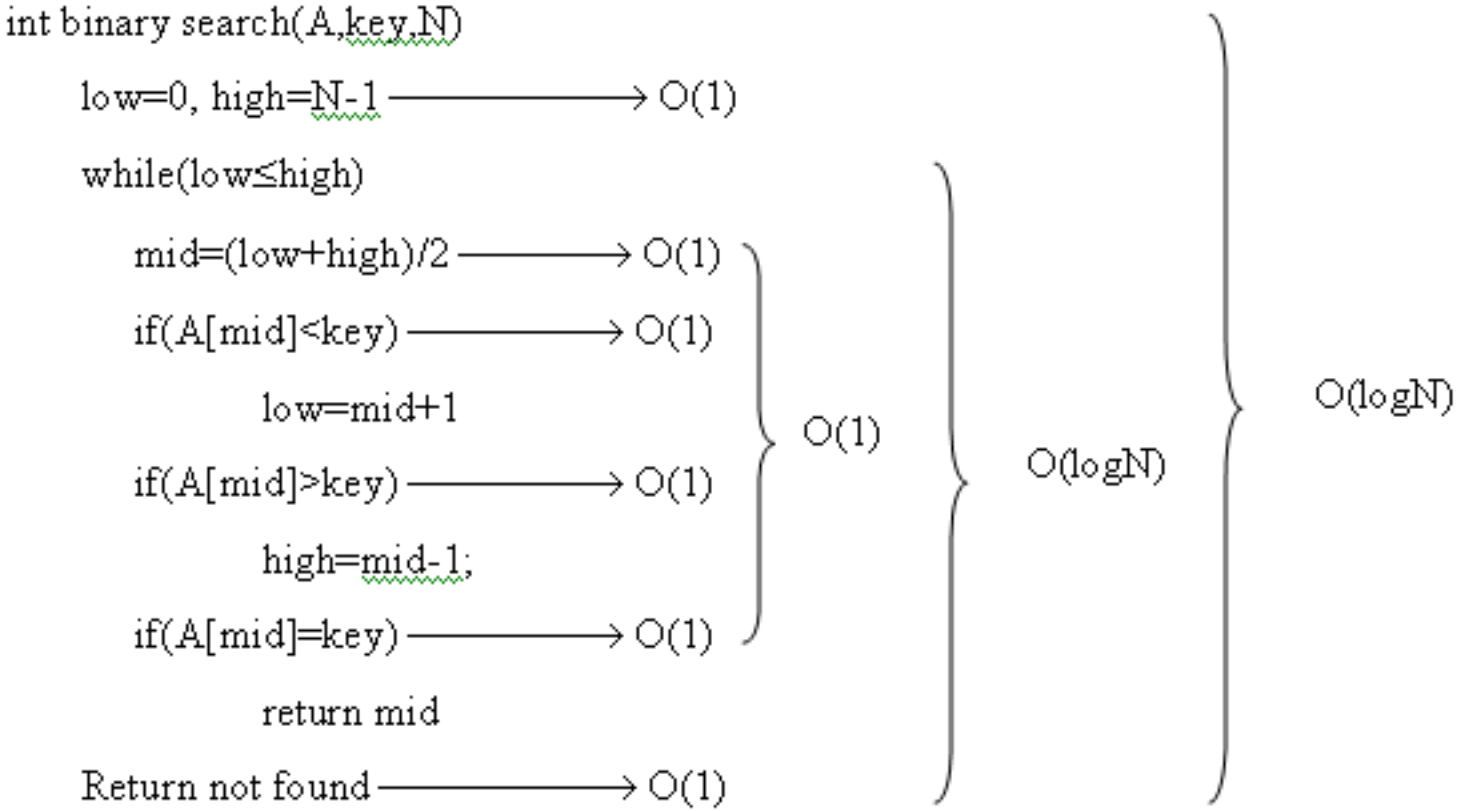
```
// Aranan sayının indeksini döndürür aranan sayı bulunamazsa -1 döndürür.
int ikiliArama(int A[], int N, int sayi){
    sol = 0;
    sag = N-1;

    while (sol <= sag){
        int orta = (sol+sag)/2;              // Test edilecek sayının indeksi
        if (A[orta] == sayi) return orta;  // Aranan sayı bulundu. İndeksi döndür
        else if (sayi < A[orta]) sag = orta – 1;       // Sağ tarafı ele
        else sol = orta+1;                                  // Sol tarafı ele
    } //bitti-while

    return –1;   // Aranan sayı bulunamadı
} //End-ikiliArama
```

- En kötü çalışma zamanı: $T(n) = 3 + 5*\log_2 N$. Neden?

Algoritma Analizinde Bir Örnek: İkili Arama Algoritması



Algoritma Analizinde Bir Örnek: İkili Arama Algoritması

- İlk çalışmada N eleman var.
- 2. de $\frac{N}{2^1} \rightarrow 3.de \frac{N}{2^3} \rightarrow \ldots \frac{N}{2^k} = 1 \rightarrow 0$
- Yukarıdaki ifadeden $N=2^k$ olduğu anlaşılmaktadır. Bu durumda k=logN olarak belirlenir. N=1 için k değeri 0 olacağından,
- Tworst =logN+1 olacaktır.

Ödev

- Diğer notasyonlar hakkında araştırma yapıp, ders notundaki örneklere uygulayınız.
- Farklı algoritmalara tüm notasyonları uygulayıp (T(n) hesabı) sonuçları yorumlayınız.(En az 5 algoritma)
- Ödevler gelecek hafta Word dokümanı, slayt ve çıktı olarak gelecektir. Süre dışında gelen ödevler kabul edilmeyecektir.
- Ödev raporunda hazırlayan kişi, çalışma süresi, bulduğu kaynaklar ve katkısı yazılacaktır.
- Ödev teslimi imza ile alınacaktır.

Veri Yapıları- Bölüm Özeti

- Veri modelleri ve onlara ait veri yapıları yazılım geliştirmenin temel noktalarıdır; problemlerin en etkin şekilde çözülebilmesi için ona algoritmik ifadenin doğasına yakın bulunmasıdır. Kısaca, veri yapıları, verinin saklanma kalıbını, veri modelleri de veriler arasındaki ilişkiyi gösterir.

- Bilinen ve çözümlerde sıkça başvurulan veri modelleri, genel olarak, bağlantılı liste (link list), ağaç (tree), graf (graph), durum makinası (state machine), ağ (network) ve veritabanı-ilişkisel (database-relation) şeklinde verilebilir.

- Her veri modelinin, altında duran veri yapısına bağlı olarak, işlem zaman maliyetleri ve bellek gereksinimleri farklıdır. Program geliştirilirken, zaman ve bellek alanı maliyetlerini dengeleyecek çözüm üretilmeye çalışılır. Genel olarak, RAM türü ardışıl erişimlerin yapılabildiği bellek üzerinde, maliyeti ile bellek gereksinim ters orantılı olduğu söylenebilir.

Örnekler

Örnek I:
Dizideki sayıların toplamını bulma

```
int Topla(int A[], int N)
{
  int toplam = 0;

  for  (i=0; i < N; i++){
    toplam += A[i];
  } //Bitti-for

  return toplam;
} //Bitti-Topla
```

- Bu fonksiyonun yürütme zamanı ne kadardır?

Örnek I:
Dizideki sayıların toplamını bulma

```
int Topla(int A[], int N)
{
  int topla = 0;
  for (i=0; i < N; i++){
    topla += A[i];
  } //Bitti-for

  return topla;
} //Bitti-Topla
```

İşlem sayısı

1

N

N

1

Toplam: 1 + N + N + 1 = 2N + 2

- Çalışma zamanı: T(N) = 2N+2
  - N dizideki sayı sayısı

Örnek II:
Dizideki bir elemanın aranması

```
int Arama(int A[], int N,
                  int sayi) {
  int i = 0;
  while (i < N){
    if (A[i] == sayi) break;
    i++;
  } //bitti-while

  if (i < N) return i;
  else return -1;
} //bitti-Arama
```

İşlem sayısı

1

1<=L<=N

1<=L<=N

0<=L<=N

1

1

Toplam: 1+3*L+1+1 = 3L+3

Örnek II:
Dizideki bir elemanın aranması

- En iyi çalışma zamanı nedir?
  - Döngü sadece bir kez çalıştı=>T(n) = 6
- Ortalama(beklenen) çalışma zamanı nedir?
  - Döngü N/2 kez çalıştı =>T(n)=3*n/2+3 = 1.5n+3
- En kötü çalışma zamanı nedir?
  - Döngü N kez çalıştı =>T(n) = 3n+3

Örnek III: Matris Çarpımı

```
/* İki boyutlu dizi A, B, C.      Hesapla C = A*B    */
for (i=0; i<N; i++) {
   for (j=0; j<N; j++) {
      C[i][j] = 0;
      for (int k=0; k<N; k++){
        C[i][j] += A[i][k]*B[k][j];
      } //bitti-en içteki for
   } //bitti-içteki for
} //bitti-dıştaki for
```

$$T(N) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (1 + \sum_{k=0}^{N-1} 1) = N^3 + N^2$$

Bazı Matematiksel İfadeler

S(N) = 1 + 2 + 3 + 4 + … N = $$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

Karelerin Toplamı: $$\sum_{i=1}^{N} i^2 = \frac{N*(N+1)*(2n+1)}{6} \approx \frac{N^3}{3}$$

Geometrik Seriler: $$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$$ A > 1

$$\sum_{i=0}^{N} A^i = \frac{1-A^{N+1}}{1-A} = \Theta(1)$$ A < 1

Bazı Matematiksel İfadeler

Lineer Geometrik seriler: $$\sum_{i=0}^{n} ix^i = x + 2x^2 + 3x^3 + \ldots + nx^n = \frac{(n-1)x^{(n+1)} - nx^n + x}{(x-1)^2}$$

Harmonik seriler: $$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = (\ln\ n) + O(1)$$

Logaritma:

$$\log\ A^B = B * \log\ A$$

$$\log(\ A * B) = \log\ A + \log\ B$$

$$\log(\ \frac{A}{B}) = \log\ A - \log\ B$$

Bazı Matematiksel İfadeler

- İki sınır arasındaki sayıların toplamı:

$$\sum_{i=a}^{b} f(i) = \sum_{i=0}^{b} f(i) - \sum_{i=0}^{a-1} f(i)$$

$$\sum_{i=1}^{n} (4i^2 - 6i) = 4\sum_{i=1}^{n} i^2 - 6\sum_{i=1}^{n} i$$

Listeler

Yrd. Doç. Dr. Aybars UĞUR,
Yrd.Doç.Dr. M. Ali Akcayol ders notları ve yabancı kaynak derlemelerinden hazırlanmıştır.

LİSTELER

- Günlük hayatta listeler; alışveriş listeleri, davetiye, telefon listeleri vs. kullanılır.
- Programlama açısından liste; aralarında doğrusal ilişki olan veriler topluluğu olarak görülebilir.
- Veri yapılarında değişik biçimlerde listeler kullanılmakta ve üzerlerinde değişik işlemler yapılmaktadır.

Doğrusal Listeler (Diziler)

- Diziler(arrays), doğrusal listeleri oluşturan yapılardır. Bu yapıların özellikleri şöyle sıralanabilir:

- Doğrusal listelerde süreklilik vardır. Dizi veri yapısını ele alırsak bu veri yapısında elemanlar aynı türden olup bellekte art arda saklanırlar.

- Dizi elemanları arasında başka elemanlar bulunamaz. Diziye eleman eklemek gerektiğinde (dizinin sonu hariç) dizi elemanlarının yer değiştirmesi gerekir.

- Dizi program başında tanımlanır ve ayrılacak bellek alanı belirtilir. Program çalışırken eleman sayısı arttırılamaz veya eksiltilemez.

Doğrusal Listeler (Diziler)

- Dizinin boyutu baştan çok büyük tanımlandığında kullanılmayan alanlar oluşabilir.
- Diziye eleman ekleme veya çıkarmada o elemandan sonraki tüm elemanların yerleri değişir. Bu işlem zaman kaybına neden olur.
- Dizi sıralanmak istendiğinde de elemanlar yer değiştireceğinden karmaşıklık artabilir ve çalışma zamanı fazlalaşır.

BAĞLI LİSTELER

- Bellekte elemanları ardışık olarak bulunmayan listelere bağlı liste denir.

- Bağlı listelerde her eleman kendinden sonraki elemanın nerede olduğu bilgisini tutar. İlk elemanın yeri ise yapı türünden bir göstericide tutulur. Böylece bağlı listenin tüm elemanlarına ulaşılabilir.

- Bağlı liste dizisinin her elemanı bir yapı nesnesidir. Bu yapı nesnesinin bazı üyeleri bağlı liste elemanlarının değerlerini veya taşıyacakları diğer bilgileri tutarken, bir üyesi ise kendinden sonraki bağlı liste elemanı olan yapı nesnesinin adres bilgisini tutar.

BAĞLI LİSTELER

- Bağlantılı liste yapıları iki boyutlu dizi yapısına benzemektedir. Aynı zamanda bir boyutlu dizinin özelliklerini de taşımaktadır.
- Bu veri yapısında bir boyutlu dizilerde olduğu gibi silinen veri alanları hala listede yer almakta veri silindiği halde listenin boyu kısalmamaktadır.
- Eleman eklemede de listenin boyutu yetmediğinde kapasiteyi arttırmak gerekmektedir. Bu durumda istenildiği zaman boyutun büyütülebilmesi ve eleman çıkarıldığında listenin boyutunun kendiliğinden küçülmesi için yeni bir veri yapısına ihtiyaç vardır.

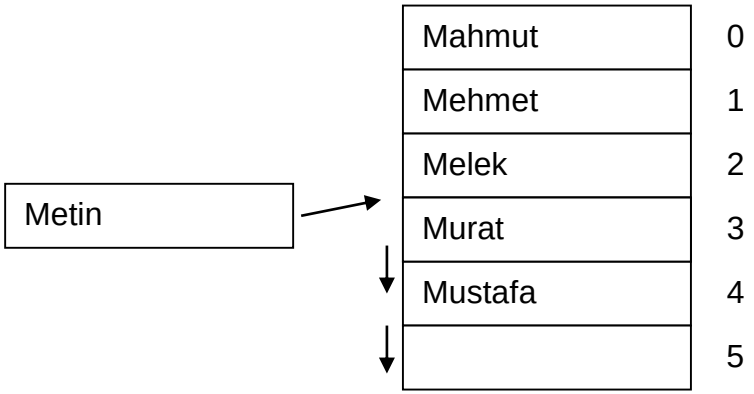
BAĞLI LİSTELER

- Doğrusal veri yapılarında dinamik bir yaklaşım yoktur. İstenildiğinde bellek alanı alınamaz ya da eldeki bellek alanları iade edilemez. Bağlantılı listeler dinamik veri yapılar olup yukarıdaki işlemlerin yapılmasına olanak verir. Bağlantılı listelerde düğüm ismi verilen bellek büyüklükleri kullanılır.

- Bağlantılı listeler çeşitli tiplerde kullanılmaktadır;
  - Tek yönlü doğrusal bağlı liste
  - İki yönlü doğrusal bağlı liste
  - Tek yönlü dairesel bağlı liste
  - İki yönlü dairesel bağlı liste

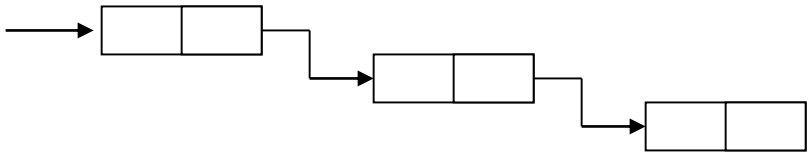
BAĞLI LİSTELER

- **Bağlı Listelerle Dizilerin Karşılaştırılması:**
- Diziler;
  - Boyut değiştirme zordur
  - Yeni bir eleman ekleme zordur
  - Bir elemanı silme zordur
  - Dizinin tüm elemanları için hafızada yer ayrılır
- Bağlı listeler ile bu problemler çözülebilir.



BAĞLI LİSTELER

- Ayrıca, bağlı listelerde
  - Her elaman için ayrı hafıza alanı ayrılır.
  - Bilgi kavramsal olarak sıralıdır ancak hafızada bulunduğu yer sıralı değildir.
  - Her bir eleman (node) bir sonrakini gösterir.



BAĞLI LİSTELER

- **Örnek: C dilinde bağlı liste yapısı**

```
struct ListNode
{ int data;
  struct ListNode *next;
 }
```

- Bağlı listenin ilk elemanının adresi **global** bir göstericide tutulabilir. Son elemanına ilişkin gösterici ise **NULL** adresi olarak bırakılır. Bağlı liste elemanları **malloc** gibi dinamik bellek fonksiyonları ile oluşturulur.

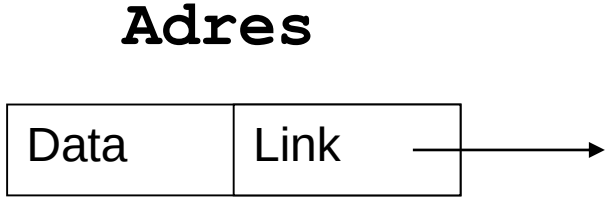
- **Örnek: Java dilinde bağlı liste yapısı**

```
public class ListNode
{
int data;
public ListNode next;
}
```

BAĞLI LİSTELER

- Listeler;
  - Listedeki her bir eleman data (veri) ve link (bağlantı) kısmından oluşur. Data kısmı içerisinde saklanan bilgiyi ifade eder. Link kısmı ise kendisinden sonraki elamanı işaret eder.

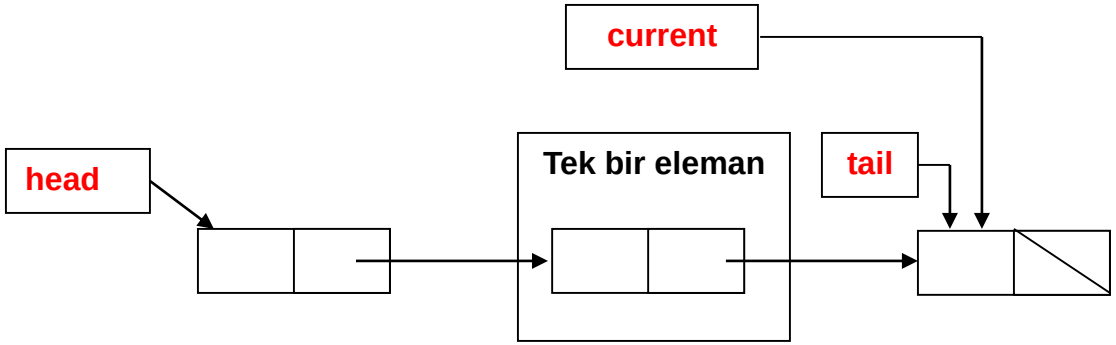
```
public class ListNode
{
int data;
public ListNode sonraki;
}
```



Bir düğümün yapısı

BAĞLI LİSTELER

- Listede bir başlangıç (head) elemanı (listenin başlangıcını gösterir ve diğer düğümlere ulaşmak için bu düğüm ile başlamak mecburidir), birde sonuncu (tail) elamanı vardır.
- Listede aktif (current) eleman şu anda bilgilerine ulaşabileceğimiz elemandır.

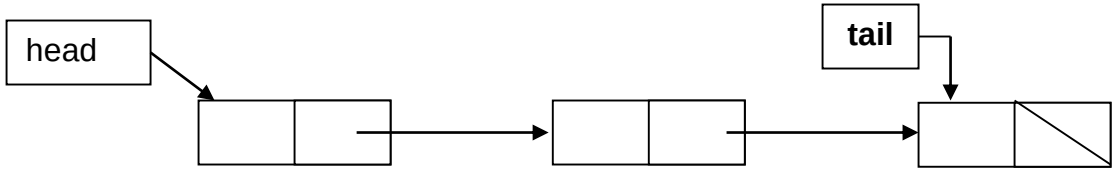


Bağlı Liste İşlemleri

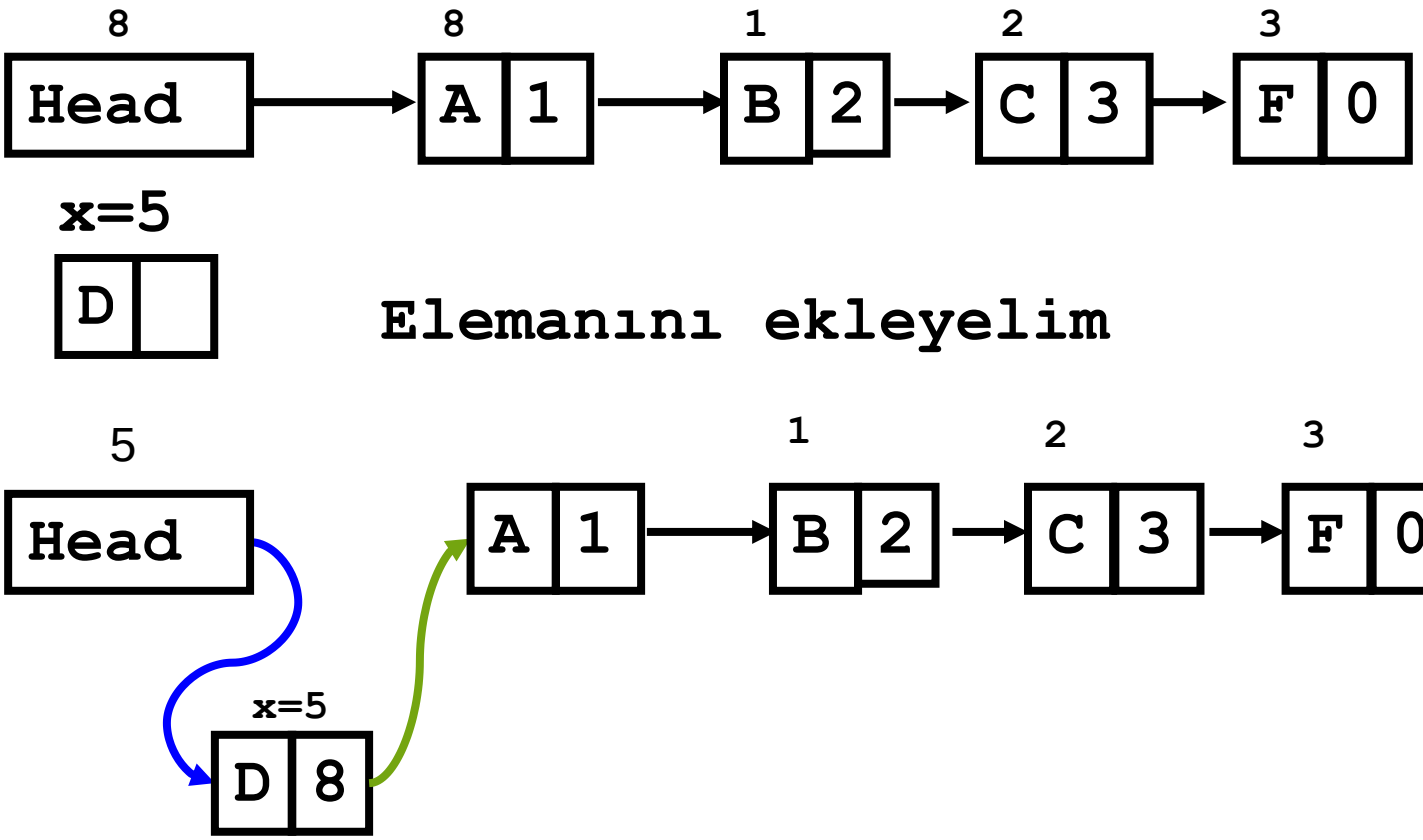
- Listeler ile yapılan işlemler,
  - Listeye eleman ekleme
    - Başa
    - Sona
    - Sıralı
  - Listeden eleman silme
    - Baştan
    - Sondan
    - Tümünü
    - Belirlenen bilgiye sahip elemanı
    - İstenen sıradaki elemanı
  - Arama
  - Listeleme
    - İstenen bilgiye göre
  - Kontrol
    - Boş liste
    - Liste boyutu

Tek Yönlü Bağlı Listeler

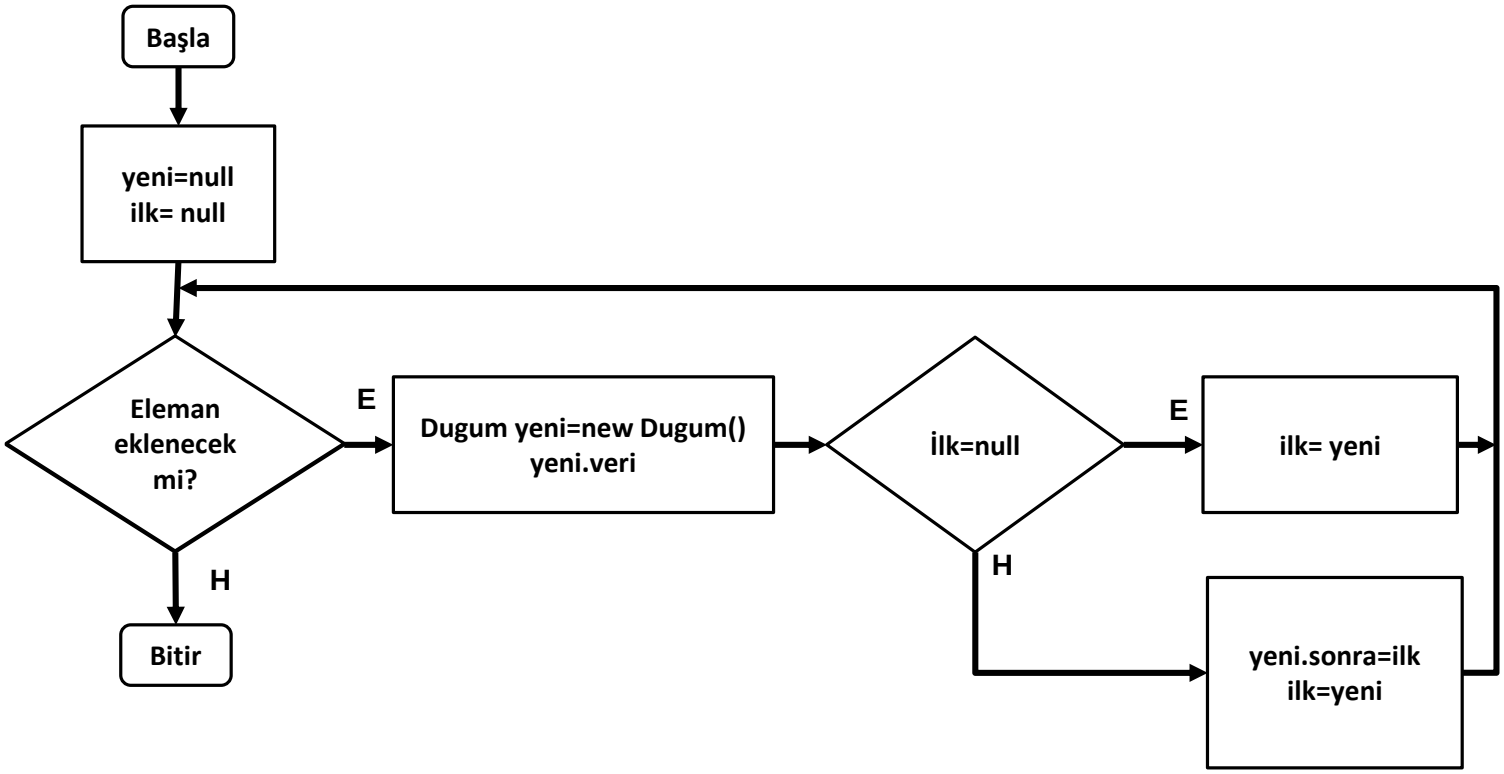
- Listedeki elemanlar arasında sadece tek bir bağ vardır. Bu tür bağlı listelerde hareket yönü sadece listenin başından sonuna doğrudur.



Tek Yönlü Bağlı Listenin Başına Elaman Ekleme



Tek Yönlü Bağlı Liste Başa Ekleme



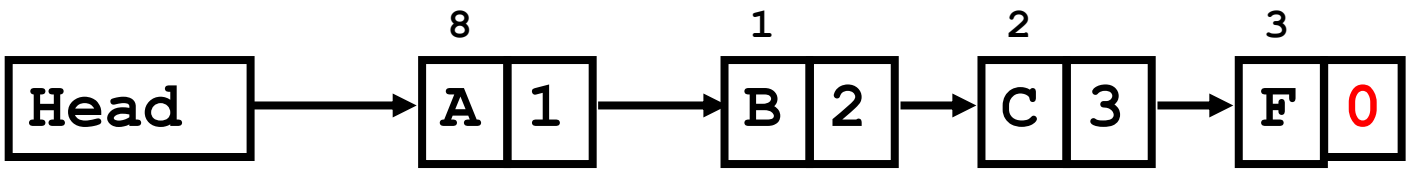
Tek Yönlü Bağlı Listeler

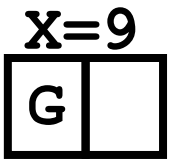
```
class Dugum
{
 public int veri;
// Değişik tiplerde çoğaltılabilir
 public Dugum kuyruk;
// son düğümün adresi
public void yazdir()
{System.out.print(" "+veri);
}
}

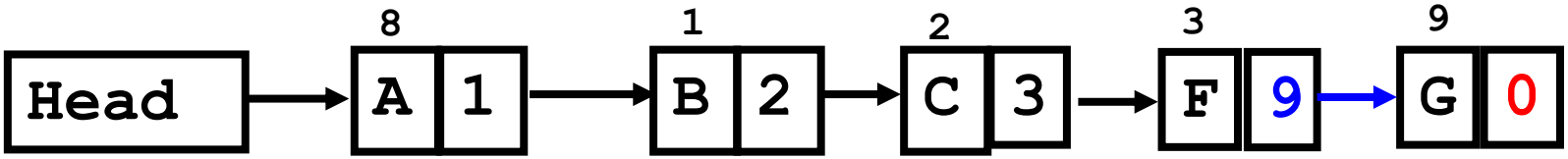
//C# için  Console.WriteLine(" "+veri);
```

```
class BListe
{ private Dugum  ilk,son;
// Listenin ilk düğümünün adresini tutar
 public BListe()
// Bir BListe nesnesi yaratıldığında
 {  ilk = null;
    //boş liste olarak açılır.
 }
public void basaEkle(int data)
// Liste başına eleman ekler
{
  Dugum yeni = new Dugum();
  yeni.veri = data;
  yeni.kuyruk = ilk;
  ilk = yeni;
}
```

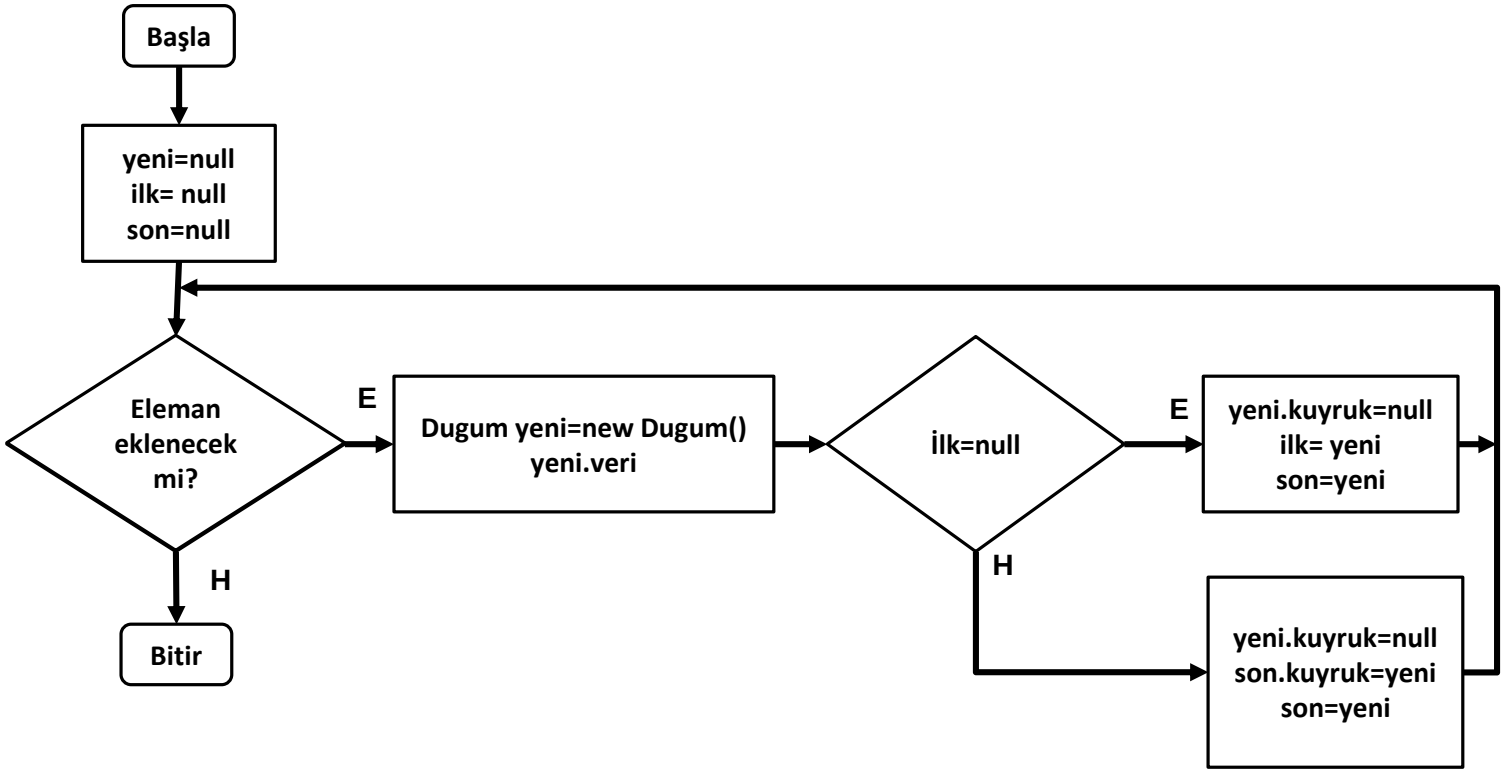
Tek Yönlü Bağlı Listeye Sona Ekleme







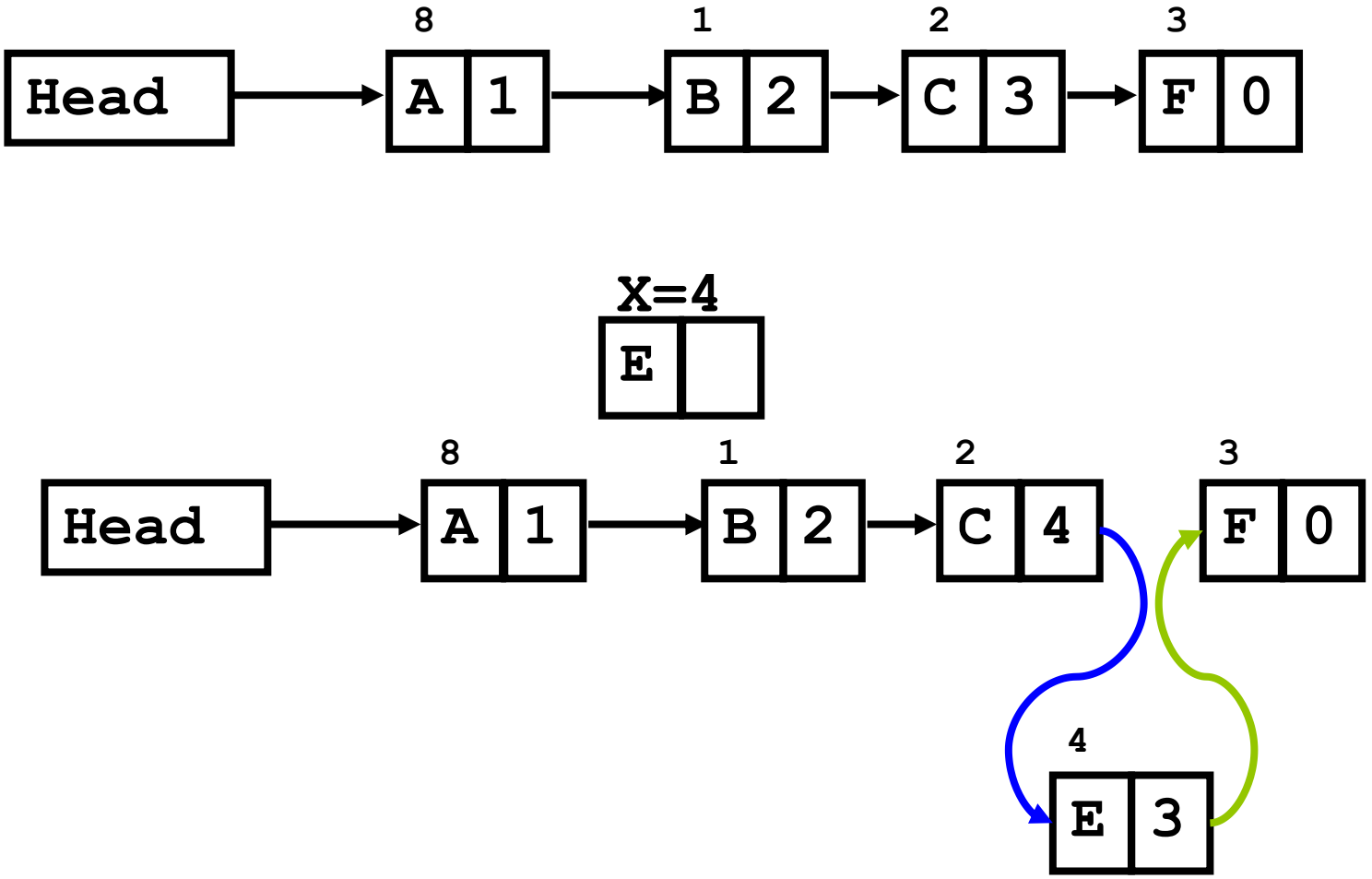
Tek Yönlü Bağlı Liste Sona Ekleme



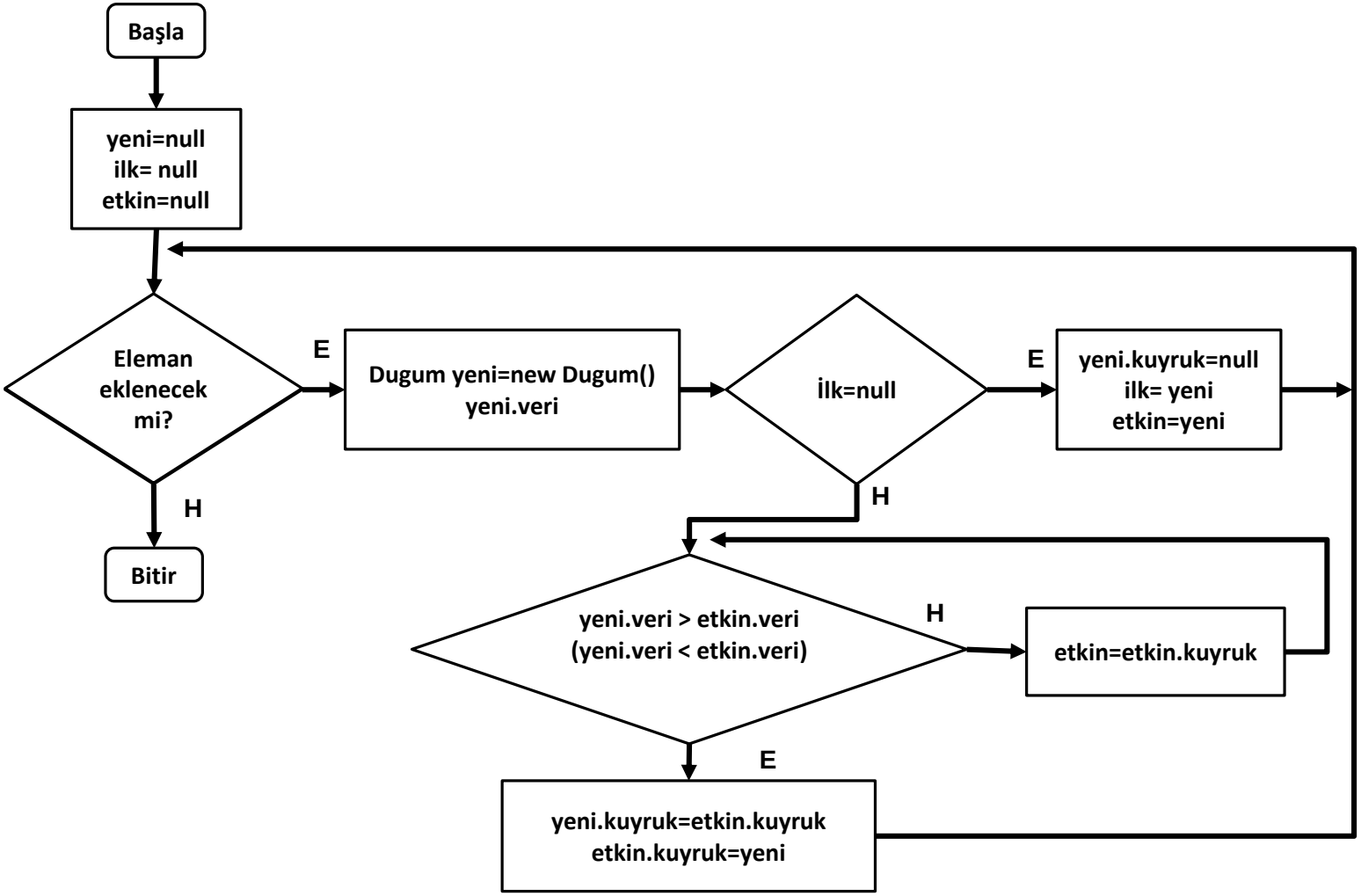
Tek Yönlü Bağlı Listeler

```
public void sonaEkle(int data)
// Liste sonuna eleman ekler
{   Dugum yeni = new Dugum();
    yeni.veri = data;
    yeni.kuyruk = null;
    if (ilk == null)
    {
        ilk = yeni;
        son = yeni;
    }
    else
    {   son.kuyruk=yeni;
        son = yeni;
    }
}
```

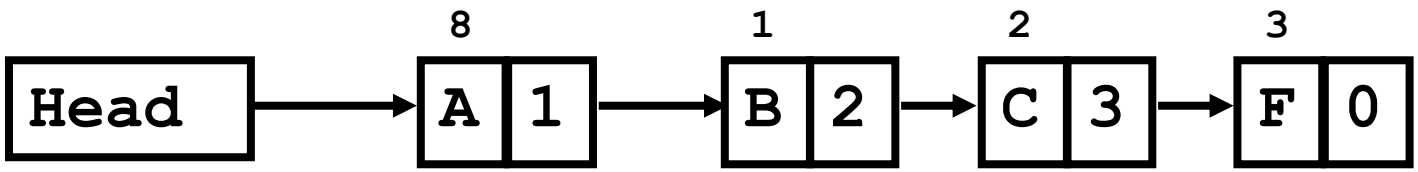
Tek Yönlü Bağlı Liste-Araya Ekleme



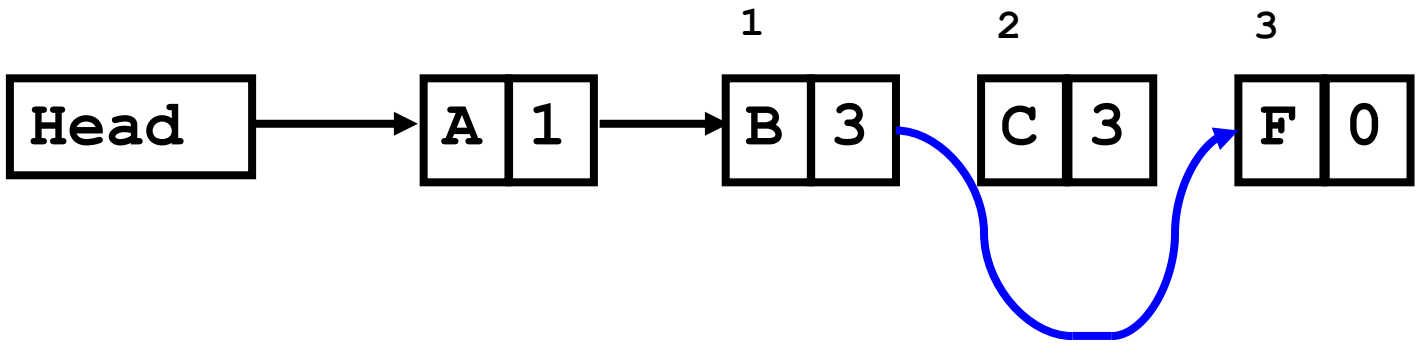
Tek Yönlü Bağlı Liste Araya Ekleme



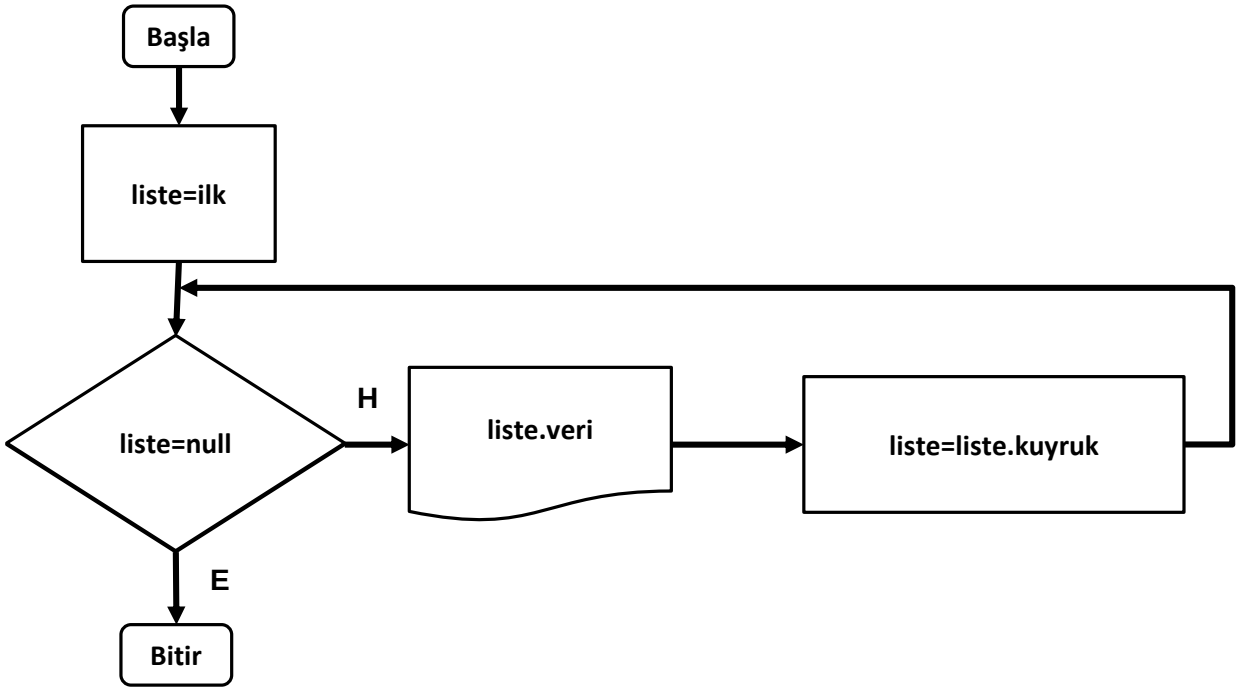
Tek Yönlü Bağlı Listeden Elaman Çıkarma



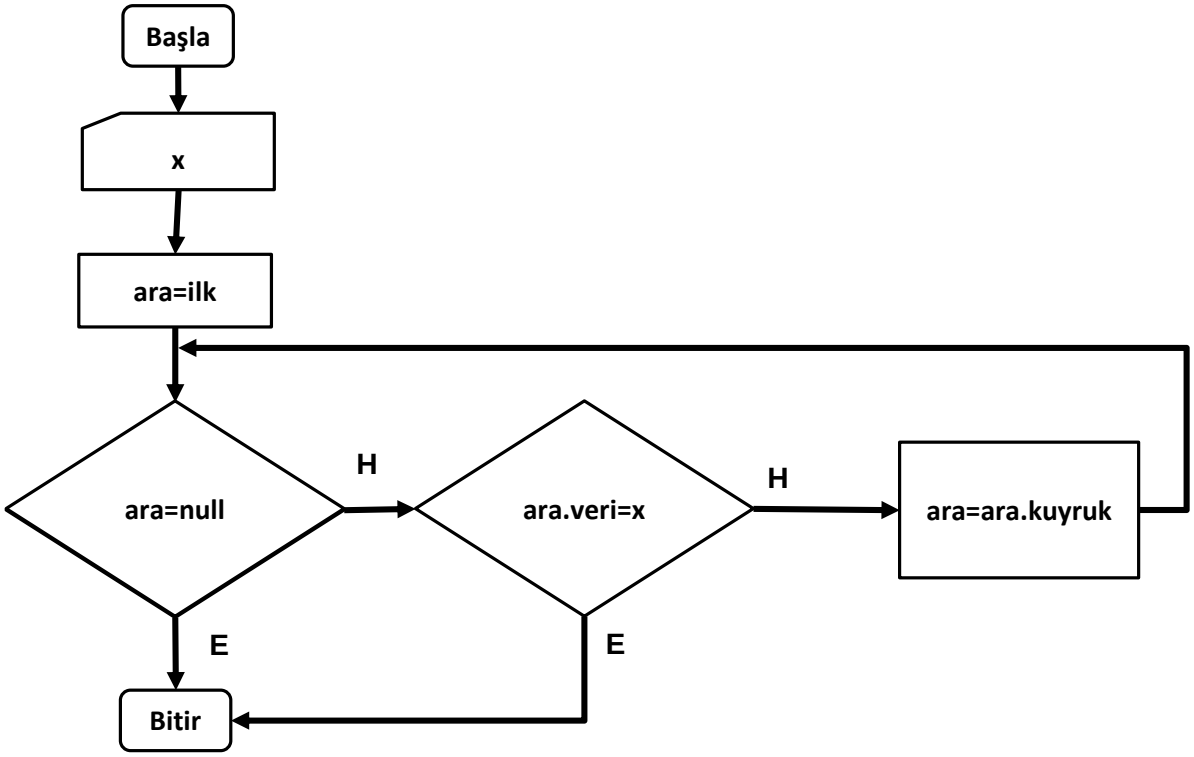
C elamanını çıkarma



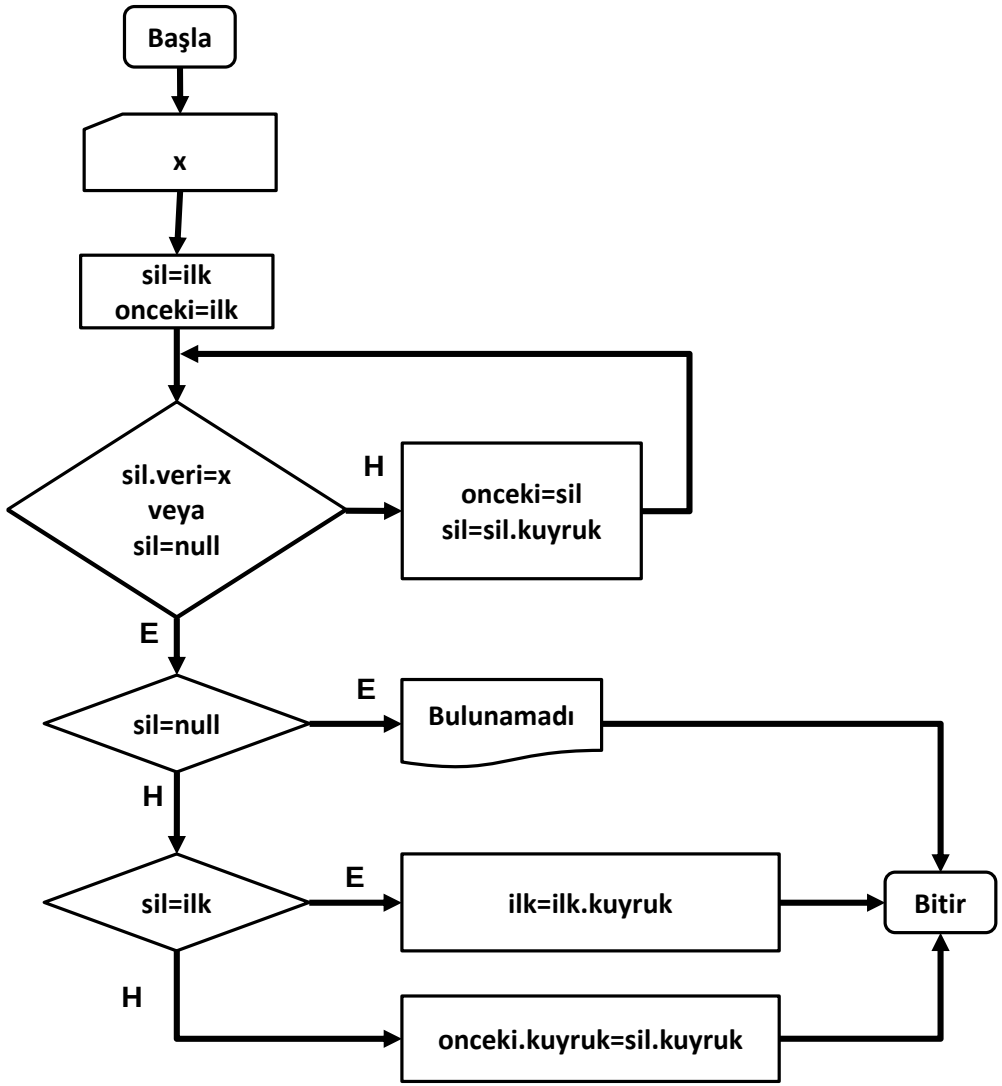
Tek Yönlü Bağlı Liste- Listeleme



Tek Yönlü Bağlı Liste- Arama



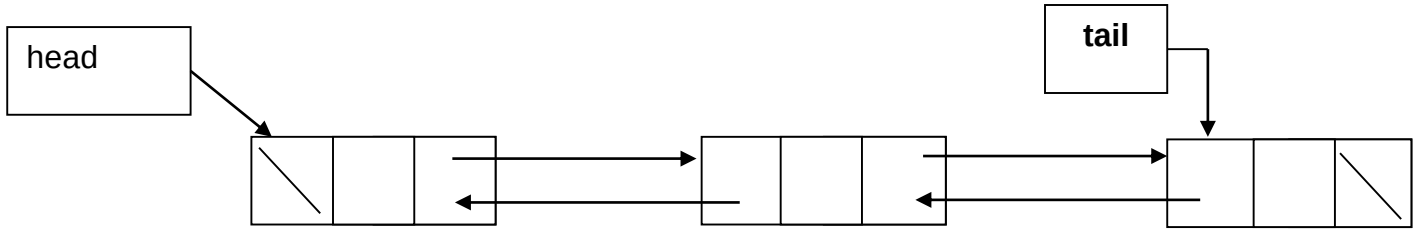
Tek Yönlü Bağlı Liste- Silme



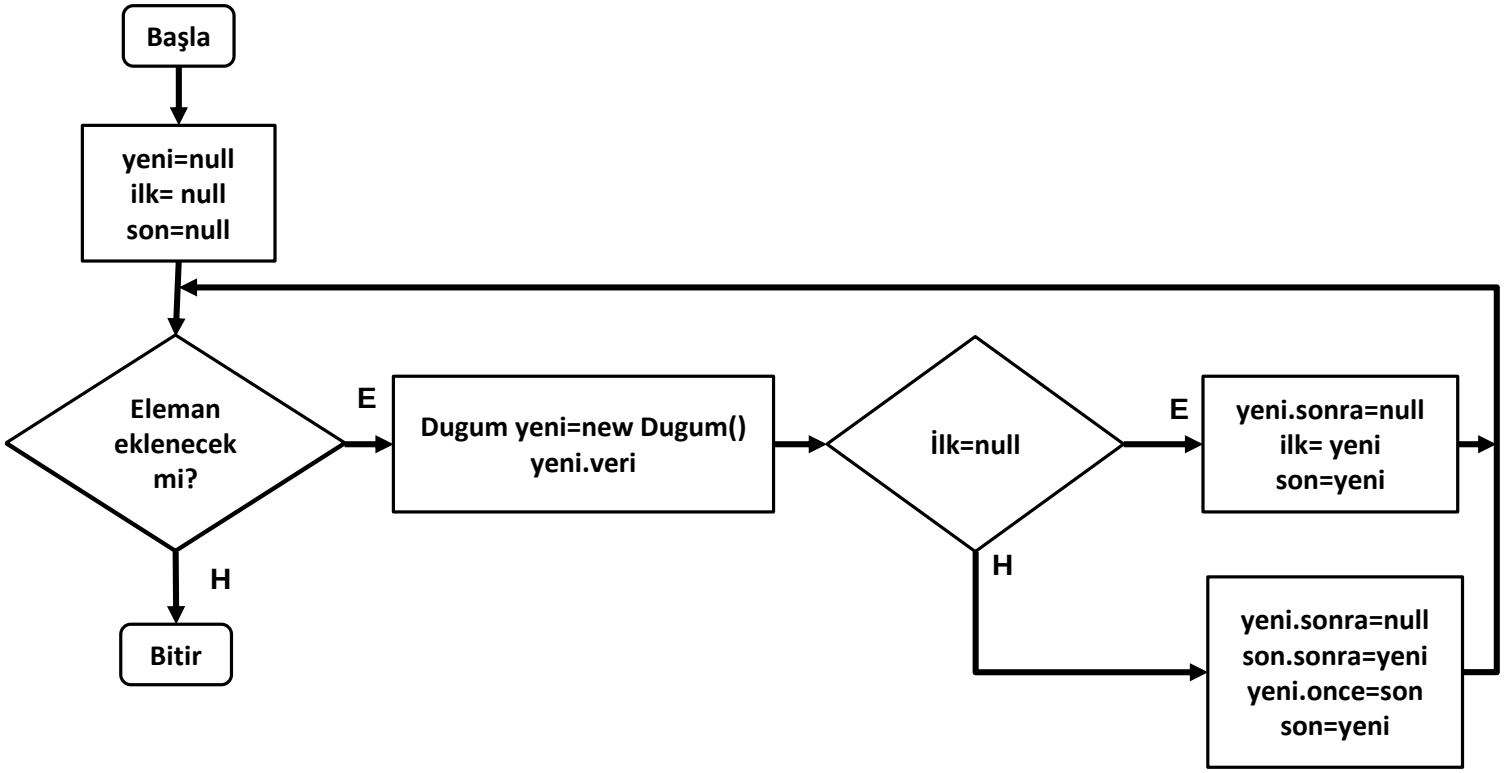
İki Yönlü Bağlı Listeler:

- Listedeki elemanlar arasında iki yönlü bağ vardır. Elemanın bağlantı bilgisi bölümünde iki gösterici bulunur. Bu göstericinin biri kendisinden sonra gelen elemanı diğeri ise kendisinden önce gelen elamanın adres bilgisini tutar.

- Bu sayede listenin hem başından sonuna hem de listenin sonundan başına doğru hareket edilebilir. Bu yöntem daha esnek bir yapıya sahip olduğundan bazı problemlerin çözümünde daha işlevsel olabilmektedir. Örnek:Telefon rehberi, Uzaktan kumanda kanal listesi



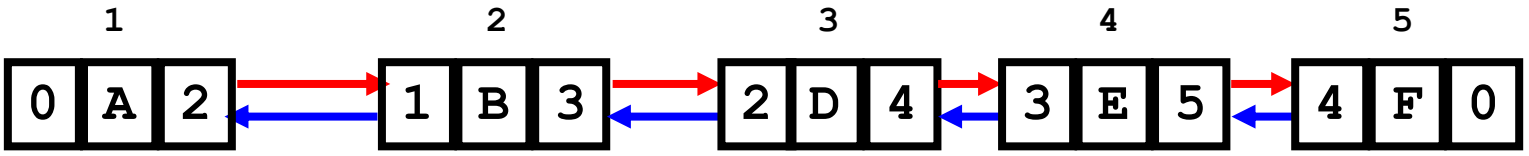
İki Yönlü Bağlı Liste Ekleme



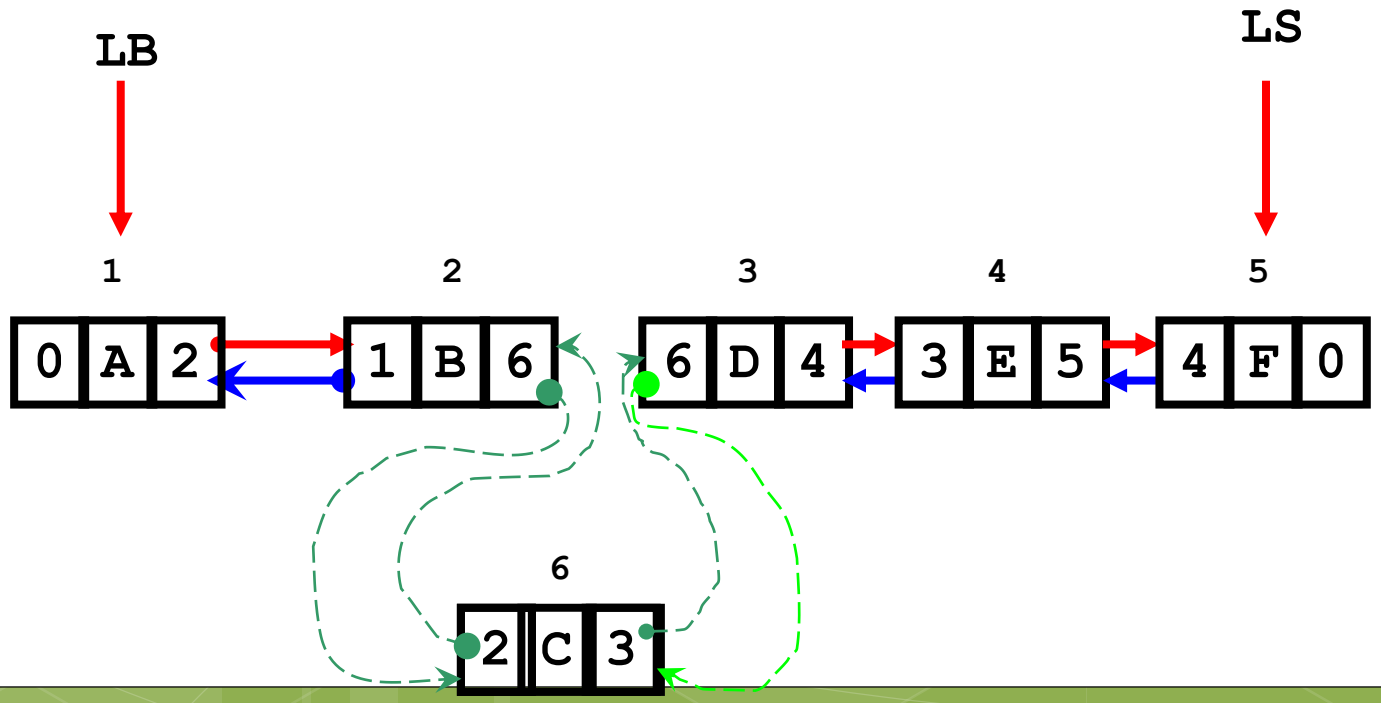
İki Yönlü Bağlı Listeler: Elaman Ekleme

Liste Başı (head)

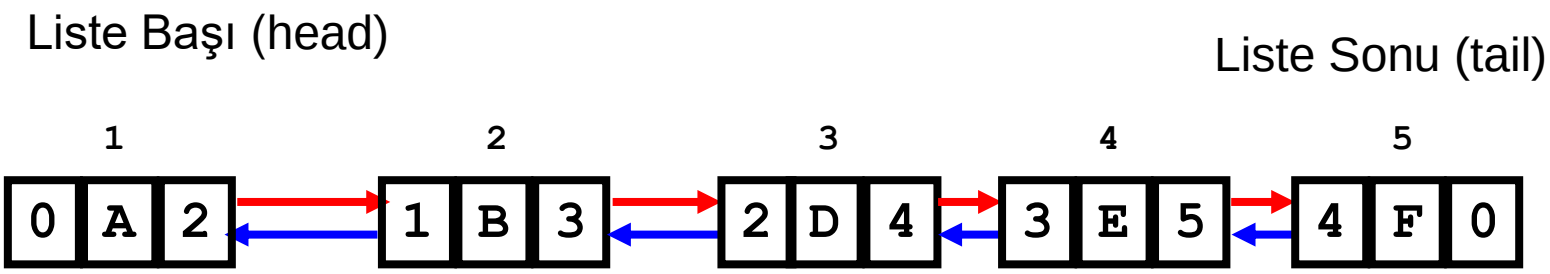
Liste Sonu (tail)



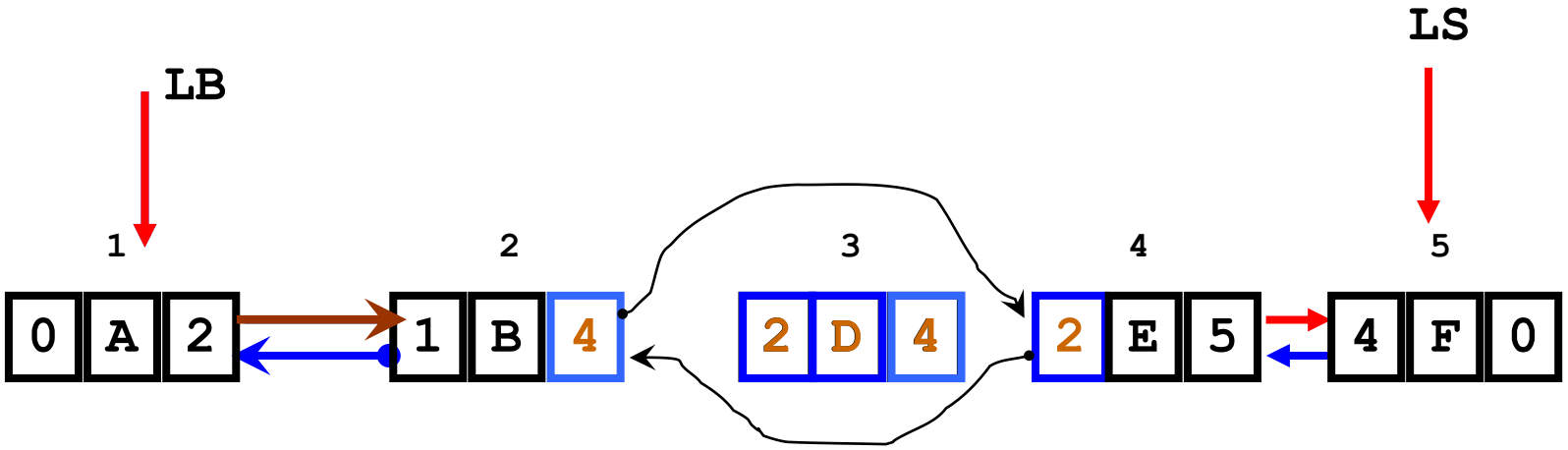
C elemanını ekleme: C'nin ekleneceği yer bulunur ve bir C düğümü oluşturulur.



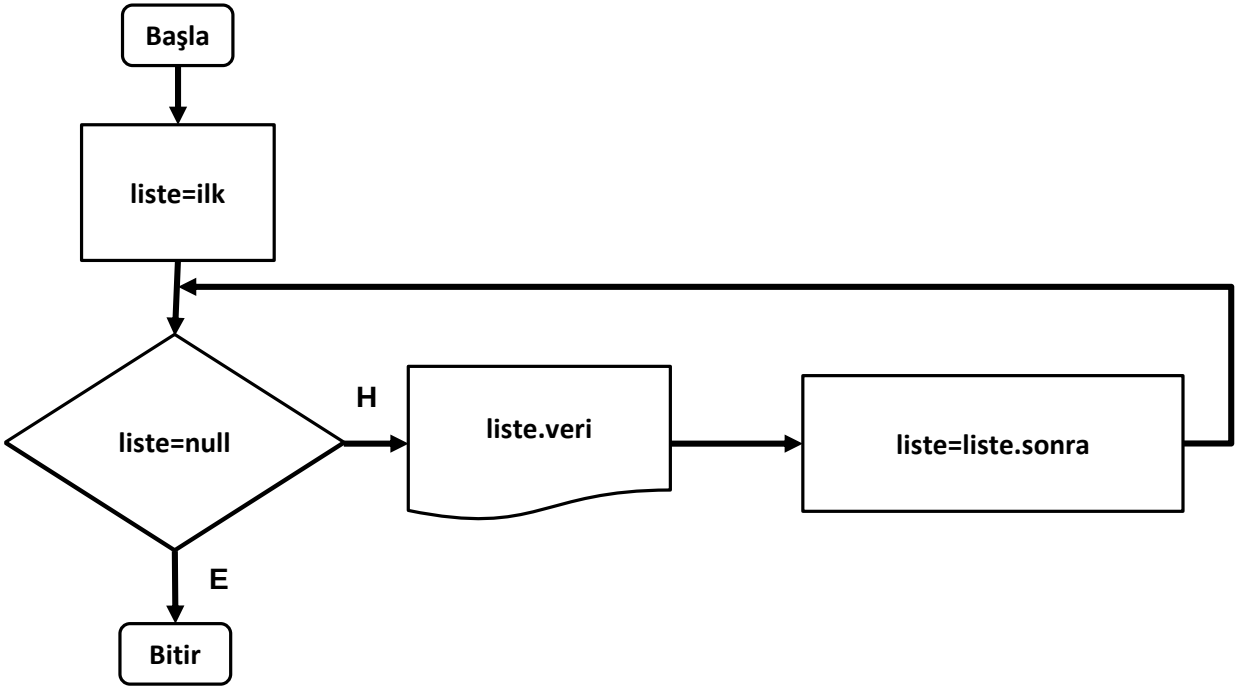
İki Yönlü Bağlı Listeler: Elaman Silme



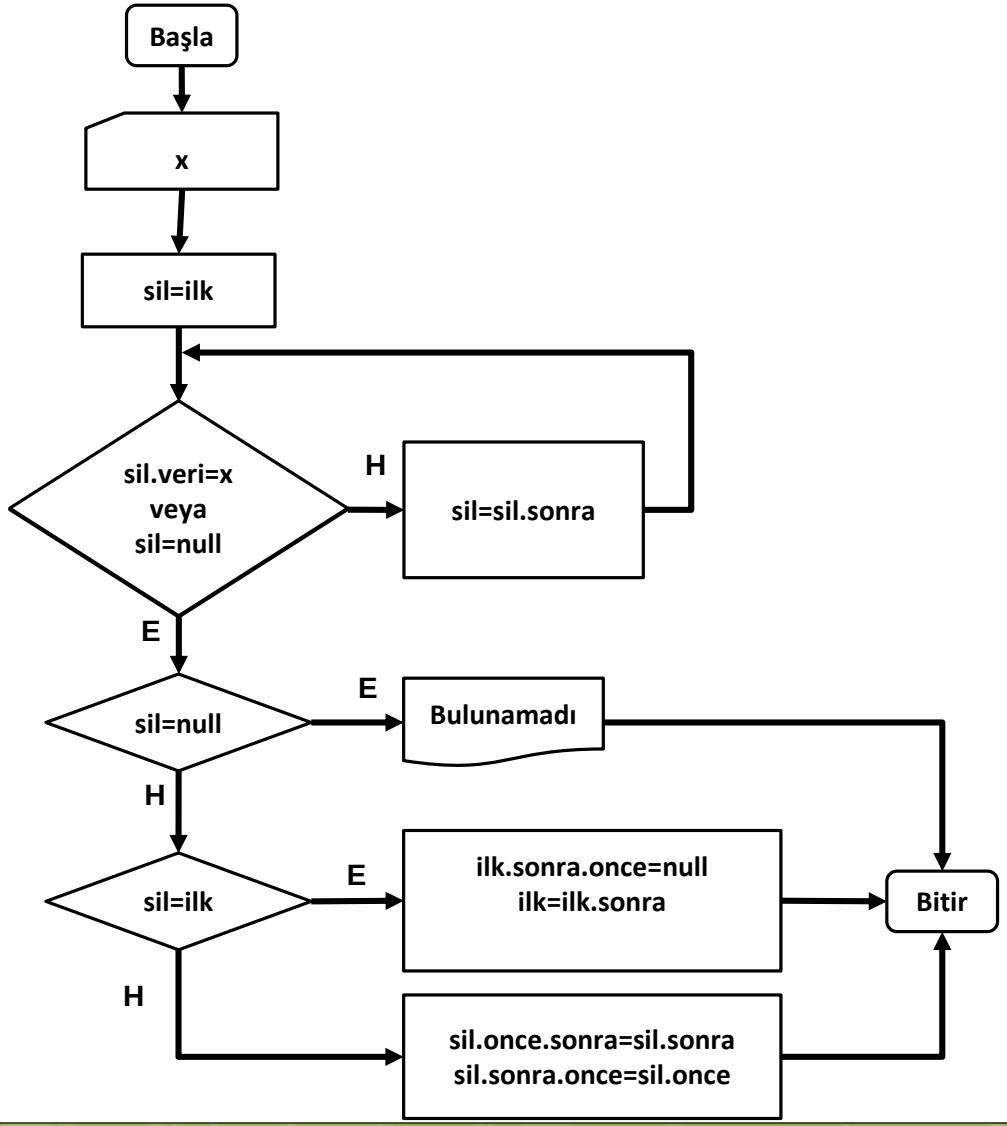
D elemanını silme: D'nin yeri bulunur ve bağ güncellemesi yapılır.



İki Yönlü Bağlı Liste Listeleme

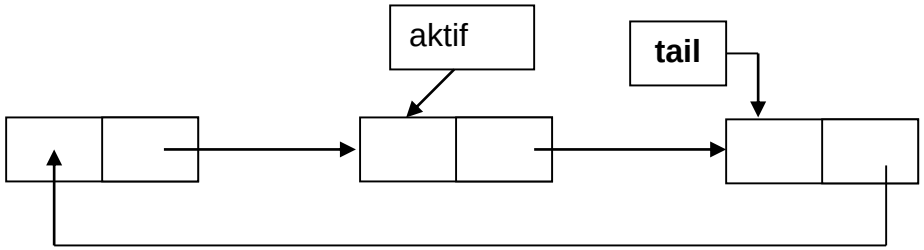


İki Yönlü Bağlı Liste Silme

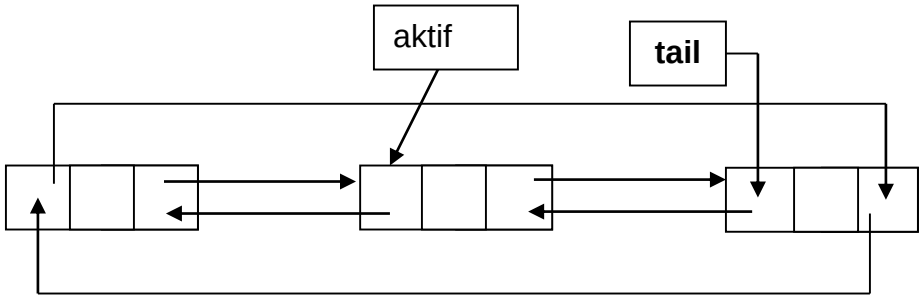


Dairesel Bağlı Listeler:

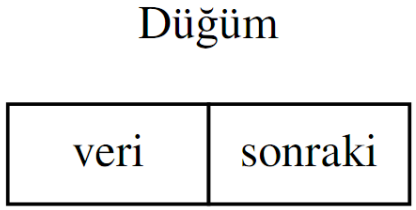
- **Tek Yönlü Dairesel Bağlı Listeler** : Listedeki elemanlar arasında tek yönlü bağ vardır. Tek yönlü bağlı listelerden tek farkı ise son elemanın göstericisi ilk listenin ilk elamanının adresini göstermesidir. Bu sayede eğer listedeki elemanlardan birinin adresini biliyorsak listedeki bütün elemanlara erişebiliriz.



- **İki Yönlü Dairesel Bağlı Listeler** : Hem dairesellik hem de çift bağlılık özelliklerine sahip listelerdir. İlk düğümden önceki düğüm son, son düğümden sonraki düğüm de ilk düğümdür.

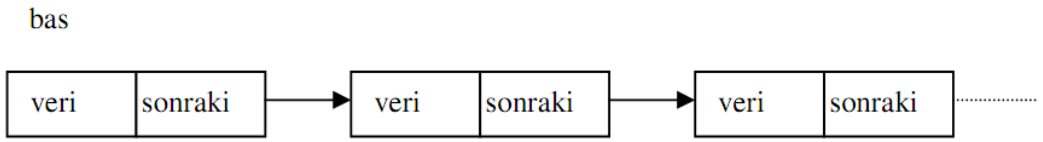


Java Programlama Dilinde Bağlı Liste Örneği: Bağlı Listenin Düğüm Yapısı




```
class Dugum
{
  public int veri;      // Değişik tiplerde çoğaltılabilir
  public Dugum sonraki; // Sonraki düğümün adresi
  public Dugum (int gelenVeri)   // Yapıcı metot
  { veri = gelenVeri; } // Düğüm yaratılırken değerini  aktarır
  public void yazdir() // Düğümün verisini yazdırır
  {   System.out.print(" "+veri); }
}
```

Java-Bağlı Liste ve Bağlı Listede Ekleme Yapısı




```
class BListe
{
private Dugum  bas;      // Listenin ilk düğümünün adresini tutar
public BListe()          // Bir BListe nesnesi yaratıldığında
 {
 bas = null;             // boş liste olarak açılır.
 son=null;
 }

public void basaEkle(int yeniEleman) // Liste bağına eleman ekler
 {
   Dugum yeniDugum = new Dugum(yeniEleman);
   yeniDugum.sonraki = bas;
   bas = yeniDugum;
 }
```

Java-Bağlı Liste ve Bağlı Listede Ekleme Yapısı

```
public void sonaEkle(int yeniEleman) // Liste bağına eleman ekler
 {
 Dugum yeni = new Dugum(yeniEleman);
 yeni. sonraki = null;
   if (ilk == null)
   {
      ilk = yeni;
      son = yeni;
   }
   else
   {  son. sonraki=yeni;
      son = yeni;
   }

 }
```

Java-Bağlı Listeye Arama Yapısı

```
public Dugum bul (int anahtar) {
//Listede anahtar değerini bulur
    Dugum etkin = bas;
    while(etkin.veri != anahtar)
    {
     if(etkin.sonraki==null)
       return null;
     else
       etkin = etkin.sonraki;
    };
    return etkin;   }
```

Java-Bağlı Listede Silme Yapısı

```
public Dugum sil(int anahtar)
 {
  // Verilen anahtar değerindeki düğümü siler
   Dugum etkin = bas;
   Dugum onceki = bas;
   while(etkin.veri!=anahtar)
  {
    if(etkin.sonraki==null)        return null;
    else        { onceki = etkin; etkin = etkin.sonraki; }
  }
  if(etkin==bas)      bas = bas.sonraki;
  else      onceki.sonraki = etkin.sonraki;
   return etkin;
 }
```

Java - Bağlı Listede Listeleme Yapısı

```
public void listele()
 {
System.out.println();
   System.out.print("Bastan Sona Liste : ");
   Dugum etkin = bas;
   while(etkin!=null)
   { etkin.yazdir(); etkin=etkin.sonraki; }
    }
}
```

Java - Bağlı Liste Örneği

- // Bir bağlı liste oluşturarak, Bliste ve Dugum sınıflarını metotlarıyla birlikte test eden sınıf

```
class BListeTest {
public static void main(String args[])  {
 BListe liste = new BListe(); // liste adlı bir bağlı liste nesnesi oluşturur.
 liste.basaEkle(9);
 for(int i=8; i>=1; --i) liste.basaEkle(i);  //liste.sonaEkle(i);
 liste.listele();    int deger = 5;    Dugum d = liste.bul(deger);
  if(d==null)      System.out.println("\n"+deger+" Listede Yok");
  else      System.out.println("\n"+deger+" Bulundu");
  Dugum s = liste.sil(5);
  liste.listele();
 }
}
```

Ekran Çıktısı :
// Bastan Sona Liste : 1 2 3 4 5 6 7 8 9
// 5 Bulundu
// Bastan Sona Liste : 1 2 3 4 6 7 8 9

Java Programlama Dilinde Bağlı Liste Örneği ve Kullanımı

- Ödev:
- Verilen örnekte
- Ekleme,
  - Baştan ekleme
  - İstenilen sırada ekleme
  - Sondan ekleme yapıları
- Silme
  - Baştan silme
  - İstenilen sırada silme
  - Sondan silme
- yapıları sizin tarafınızdan bu örnek üzerinde oluşturulacaktır ve ders hocasına teslim edilecektir.

Ödev

- **2-**
- K adet ders için N adet öğrencinin numara ve harf ortalamalarını bağlı dizilerle gösteriniz.
- Her bir node öğrenci numarası, dersin kodu, dersin harf ortalaması bilgilerini bulunduracak.
- Her öğrencinin numarası headNode içindeki bilgi olacak.
- Her dersin kodu headNode içindeki bilgi olacak.
- Her bir node aynı dersteki bir sonraki öğrenciyi gösterecek.
- Her bir node aynı kişinin diğer dersini gösterecek.
- Program Java veya C# Windows application olarak hazırlanacak ve aşağıdaki işlemleri butonlarla yapacak.
- 1- Bir öğrenciye yeni bir ders ekleme
- 2- Bir derse yeni bir öğrenci ekleme
- 3- Bir öğrencinin bir dersini silme
- 4- Bir dersteki bir öğrenciyi silme
- 5- Bir dersteki tüm öğrencileri numara sırasına göre sıralı listeleme
- 6- Bir öğrencinin aldığı tüm dersleri ders koduna göre sıralı listeleme

Ödev

- **3-** Java veya C# ile dairesel bağlı liste uygulamasını gerçekleştiriniz.
- Ekleme,
  - Baştan ekleme
  - İstenilen sırada ekleme
  - Sondan ekleme yapıları
- Silme
  - Baştan silme
  - İstenilen sırada silme
  - Sondan silme

Yığıt (Stack)

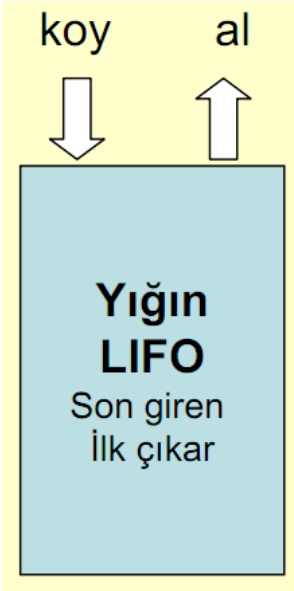
Yığıt/Yığın (Stack)

- Son giren ilk çıkar (Last In First Out-LIFO) veya İlk giren son çıkar (First-in-Last-out FILO) mantığıyla çalışır.
- Eleman ekleme çıkarmaların en üstten (top) yapıldığı veri yapısına yığıt (stack) adı verilir.
- Bir eleman ekleneceğinde yığıtın en üstüne konulur. Bir eleman çıkarılacağı zaman yığıtın en üstündeki eleman çıkarılır.
- Bu eleman da yığıttaki elemanlar içindeki en son eklenen elemandır. Bu nedenle yığıtlara LIFO (Last In First Out Son giren ilk çıkar) listesi de denilir.

Yığıt/Yığın (Stack)

- Yığın yapısını gerçekleştirmek için 2 yol vardır.
  - Dizi kullanmak
  - Bağlantılı liste kullanmak

- **empty stack:** Boş yığıt
- **push (koy):**Yığıta eleman ekleme.
- **pop (al):**Yığıttan eleman çıkarma

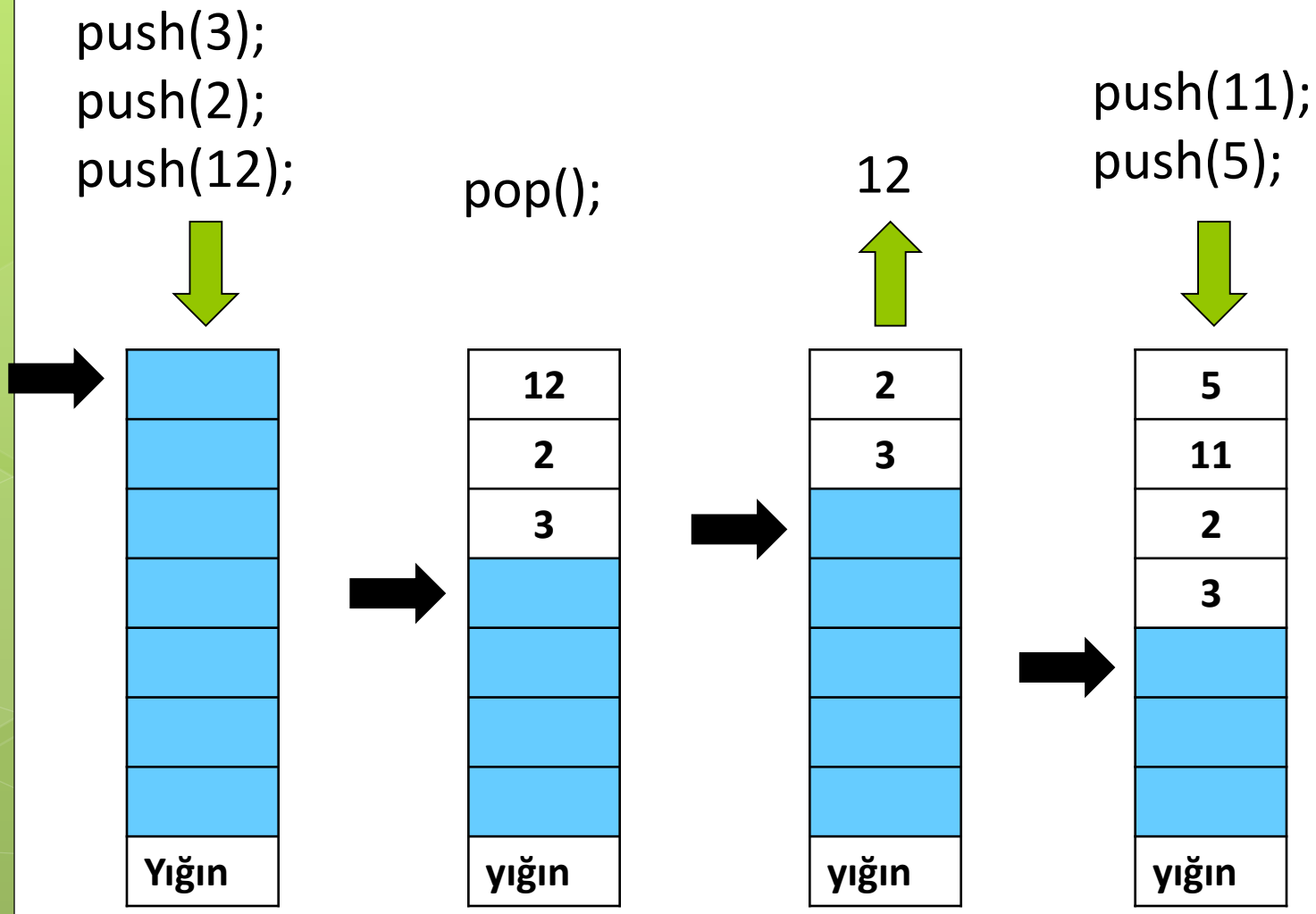


Yığın İşlemleri

- **Ana yığın işlemleri:**
  - **push(nesne):** yeni bir nesne ekler
    - **Girdi**: Nesne **Çıktı**: Yok
  - **pop():** en son eklenen nesneyi çıkarıp geri döndürür.
    - **Girdi**: Yok **Çıktı**: Nesne

- **Yardımcı yığın işlemleri:**
  - **top():** en son eklenen nesneyi çıkarmadan geri döndürür.
    - **Girdi**: Yok **Çıktı**: Nesne
  - **size():** depolanan nesne sayısını geri döndürür.
    - **Girdi**: Yok **Çıktı**: Tamsayı
  - **isEmpty**(): yığında nesne bulunup bulunmadığı bilgisi geri döner.
    - **Girdi**: Yok **Çıktı**: Boolean

Yığın (stack) Yapısı



Yığıt/Yığın (Stack)

- Örnek kullanım yerleri
  - Yazılım uygulamalarındaki Undo işlemleri stack ile yapılır. Undo işlemi için LIFO yapısı kullanılır.
  - Web browser'lardaki Back butonu (önceki sayfaya) stack kullanır. Buradada LIFO yapısı kullanılır.
  - Matematiksel işlemlerdeki operatörler (+,*,/,- gibi) ve operandlar için stack kullanılabilir.
  - Yazım kontrolündeki parantezlerin kontrolünde stack kullanılabilir.

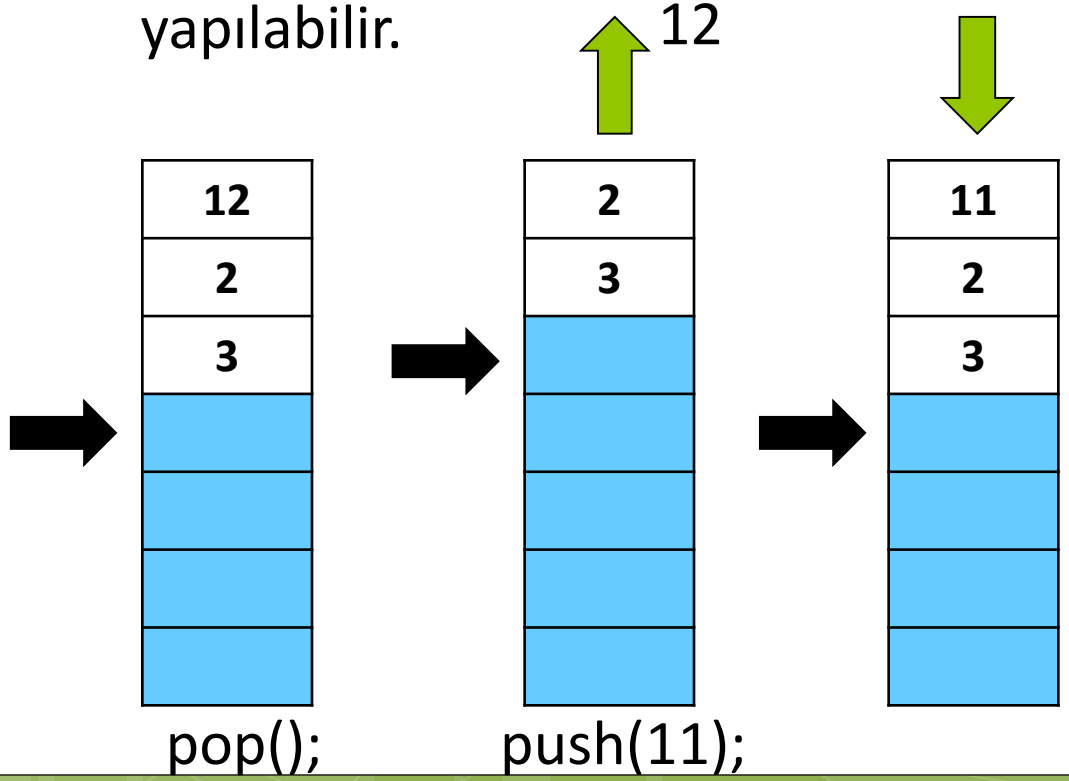
Yığıt/Yığın (Stack)

- **Örnek : Yığına ekleme ve çıkarma**

İşlem	Yığıt (tepe)	Çıktı
push(“M”);	M	
push(“A”);	MA	
push(“L”);	MAL	
push(“T”);	MALT	
pop();	MAL	T
push(“E”);	MALE	T
pop();	MAL	TE
push(“P”);	MALP	TE
pop();	MAL	TEP
push(“E”);	MALE	TEP
pop();	MAL	TEPE

Dizi Tabanlı Yığın

- Bir yığını gerçeklemenin en gerçeklemenin en kolay yolu dizi kullanmaktır. Yığın yapısı dizi üzerinde en fazla N tane eleman tutacak şekilde yapılabilir.



Dizi Tabanlı Yığın

- Nesneleri soldan sağa doğru ekleriz. Bir değişken en üstteki nesnenin index bilgisini izler. Eleman çıkarılırken bu index değeri alınır.

```
Algorithm size()
    return t + 1

Algorithm pop()
    if isEmpty() then
        throw EmptyStackException
    else
        t ← t − 1
        return S[t + 1]
```



Dizi Tabanlı Yığın

- Yığın nesnelerinin saklandığı dizi dolabilir. Bu durumda push işlemi aşağıdaki mesajı verir.
- FullStackException (DoluYığınİstinası)
  - Dizi tabanlı yaklaşımın sınırlamasıdır.

```
Algorithm push(o)
  if t = S.length − 1 then
    throw FullStackException
  else
    t ← t + 1
    S[t] ← o
```



Başarım ve Sınırlamalar

- Başarım
  - n yığındaki nesne sayısı olsun
  - Kullanılan alan O(n)
  - Her bir işlem O(1) zamanda gerçekleşir.
- Sınırlamalar
  - Yığının en üst sayısı önceden tanımlanmalıdır ve değiştirilemez.
  - Dolu bir yığına yeni bir nesne eklemeye çalışmak istisnai durumlara sebep olabilir.

Büyüyebilir Dizi Tabanlı Yığın Yaklaşımı

- **push** işlemi esnasında dizi dolu ise bir istisnai durum bildirimi geri dönmektense yığının tutulduğu dizi daha büyük bir dizi ile yer değiştirilir.
- Yeni dizi ne kadar büyüklükte olmalı?
  - **Artımlı strateji:** yığın büyüklüğü sabit bir c değeri kadar arttırılır.
  - **İkiye katlama stratejisi:** önceki dizi boyutu iki kat arttırılır
- Bağlı liste yapılarını kullanarak yığın işlemini gerçekleştirmek bu tür problemlerin önüne geçmede yararlı olur.

```
Algorithm push(o)
    if t = S.length – 1 then
        A ← new array of
                size …
        for i ← 0 to t do
            A[i] ← S[i]
        S ← A
    t ← t + 1
    S[t] ← o
```

Yığın ve Operasyonları

```
public class Yigin {

  int kapasite=100;  // maksimum eleman sayısı
  int S[]; //Yığın elemanları – pozitif tam sayı
  int p;  // eleman sayısı

  public Yigin(){    // yapıcı yordam
    s[] = new int[kapasite];
    p = 0;
  }

  int koy(int item);
  int al();
  int ust();
  boolean bosmu();
  boolean dolumu();
}
```

Yığın Operasyonları – bosmu, dolumu

```
// yığın boşsa true döndür
public boolean bosmu() {
   if (p < 1) return true;
   else return false;
} //bitti-bosmu


// Yığın doluysa true döndür
public boolean dolumu(){
   if (p == kapasite-1) return true;
   else return false;
} // bitti-dolumu
```

Yığın Operasyonları: koy

```
// Yığının üstüne yine bir eleman koy
// Başarılı ise 0 başarısız ise -1 döndürür.
int koy(int yeni){

  if (dolumu()){
    // Yığın dolu. Yeni eleman eklenemez.
    return -1;
  }

  S[p] = yeni;
  p++;

  return 0;
} /bitti-koy
```

Yığın Operasyonları: ust

```
// Yığının en üstündeki sayıyı döndürür
// Yığın boşsa, -1 döndürür
public int ust(){
  if (bosmu()){
    // Yığın başsa hata dönder
    System.out.println("Stack underflow");
    return -1;
  }

  return S[p-1];
}
```

Stack Operations: al

```
// En üsteki elemanı dönder.
// Yığın boşsa -1 dönder.
public int al(){
  if (bosmu()){
    // Yığın boşsa hata dönder
    System.out.println("Stack underflow");
    return -1;
  }

  int id = p-1; // en üsteki elemanın yeri
  p--;          // elemanı sil

  return S[id];
}
```

Yığın Kullanım Örneği

```
public static void main(String[] args){
  Yigin y = new Yigin();

  if (y.bosmu())
    System.out.println("Yığın boş");

  y.koy(49);   y.koy(23);

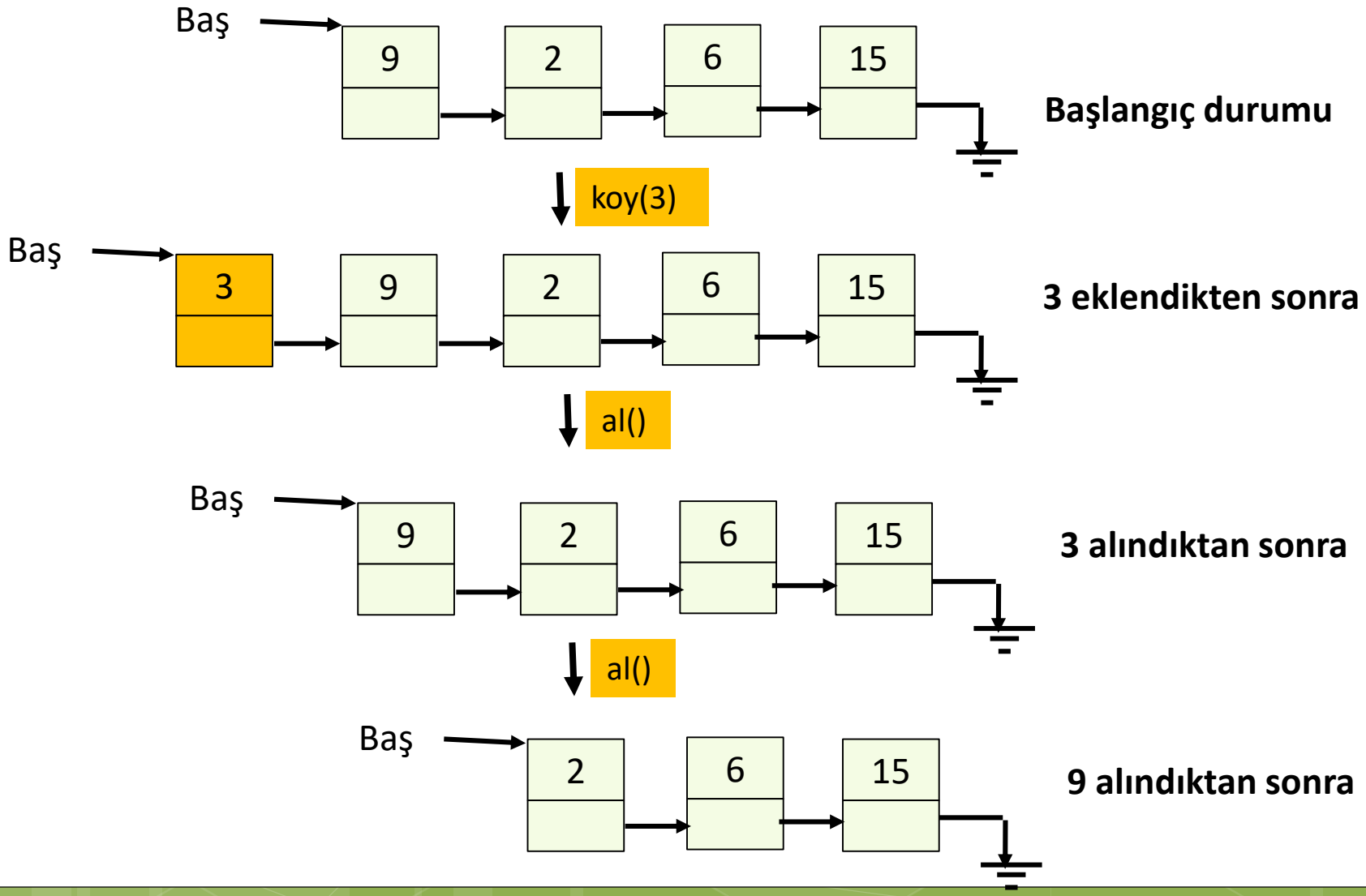
  System.out.println("Yığının ilk elemanı: "+ y.al());

  y.koy(44);   y.koy(22);

  System.out.println("Yığının ilk elemanı: "+ y.al());
  System.out.println("Yığının ilk elemanı: "+ y.al());
  System.out.println("Yığının ilk elemanı: "+ y.ust());
  System.out.println("Yığının ilk elemanı: "+ y.al());.

  if (y.bosmu()) System.out.println("Yığın boş");
}
```

Yığın- Bağlantılı Liste Gerçekleştirimi



Bağlantılı Liste Gerçekleştirimi

```
public class  YiginDugumu {
  int eleman;
  YiginDugumu sonraki;

  YiginDugumu(int e){
    eleman = e; sonraki = NULL;
  }
}
```

```
public class Yigin {
  private YiginDugumu ust;

  public Yigin() {ust = null;}

  void koy(int eleman);
  int al();
  int ust();
  boolean bosmu();
};
```

Yığın Operasyonları: koy, bosmu

```
// Yığına yeni eleman ekle
public void koy(int eleman){
   YiginDugumu x = new YiginDugumu(eleman);
   x.sonraki = ust;
   ust = x;
}

// Yığın boşsa true döndür
public boolean bosmu(){
   if (ust == NULL)
      return true;
   else
      return false;
}
```

Yığın Operasyonları: ust

```
// Yığının ilk elemanını döndür
public int ust(){
  if (bosmu()){
    System.out.println("Stack underflow"); // Boş yığın
    return -1; // Hata
  }

  return ust.eleman;
} //bitti-ust
```

Yığın Operasyonları: Al()

```
// Yığının en üst elemanın siler ve döndürür.
public int Al(){
  if (bosmu()){
    System.out.println("Stack underflow"); // Boş yığın.
    return -1; // Hata
  }

  YiginDugumu temp = ust;

  // Bir sonraki elemana geç
  ust = ust.sonraki;

  return temp.eleman;
} //bitti-al
```

Yığın Kullanım Örneği

```
public static void main(String[] args){
  Yigin y = new Yigin();

  if (y.bosmu())
    System.out.println("Yığın boş");

  y.koy(49);   y.koy(23);

  System.out.println("Yığının ilk elemanı: "+ y.al());

  y.koy(44);   y.koy(22);

  System.out.println("Yığının ilk elemanı: "+ y.al());
  System.out.println("Yığının ilk elemanı: "+ y.al());
  System.out.println("Yığının ilk elemanı: "+ y.ust());
  System.out.println("Yığının ilk elemanı: "+ y.al());.

  if (y.bosmu()) System.out.println("Yığın boş");
}
```

Örnekler –Java

- Java'da hazır Stack (yığıt) sınıfı da bulunmaktadır. Aşağıdaki örnekte String'ler, oluşturulan ***s*** yığıtına yerleştirilerek ters sırada listelenmektedir.

```
import java.util.*;
public class StackTest
{
  public static void main(String args[])
  {  String str[] = { "Bilgisayar", "Dolap", "Masa",        "Sandalye", "Sıra" };
     Stack s = new Stack();
     for(int i=0; i < t.length; ++)      s.push(str[i]);
     while(!s.empty() ) System.out.println(s.pop());
   }
}
```

Uygulama Ödevi

- Derleyici/kelime işlemciler
  - Derleyicileri düşünecek olursak yazdığımız ifadede ki parantezlerin aynı olup olmadığını kontrol ederler.
  - Örneğin: 2*(i + 5*(7 – j / (4 * k)) ifadesinde parantez eksikliği var. ")"
  - Yığın kullanarak ifadedeki parantezlerin eşit sayıda olup olmadığını kontrol eden programı yazınız.



Uygulama Ödevi

- Yığın kullanarak parantez kontrol:
  1) Boş bir yığın oluştur ve sembolleri okumaya başla
  2) Eğer sembol başlangıç sembolü ise (‘(‘, ’[’, ’{’) Yığına koy
  3) Eğer sembol kapanış sembolü ise (‘)’, ’]’, ’}’)
     I. Eğer yığın boşsa hata raporu döndür
     II. Değilse
         Yığından al
         Eğer alınan sembol ile başlangıç sembolü aynı değilse hata gönder
  4) İfade bitti ve yığın dolu ise hata döndür.

ÖRNEKLER

Örnekler- C++

- **Örnek 1: Tek yönlü bağlı liste**
- #include <stdio.h>
- #include <stdlib.h>
- main() {
- struct notlar {
- int vize1;
- struct notlar *bag;
- } *yeniadres, *ilk, *son;
- yeniadres=(struct notlar*) malloc(sizeof(struct notlar));
- printf("Yeniadres:%d\n",yeniadres);
- ilk=yeniadres;
- son=yeniadres;

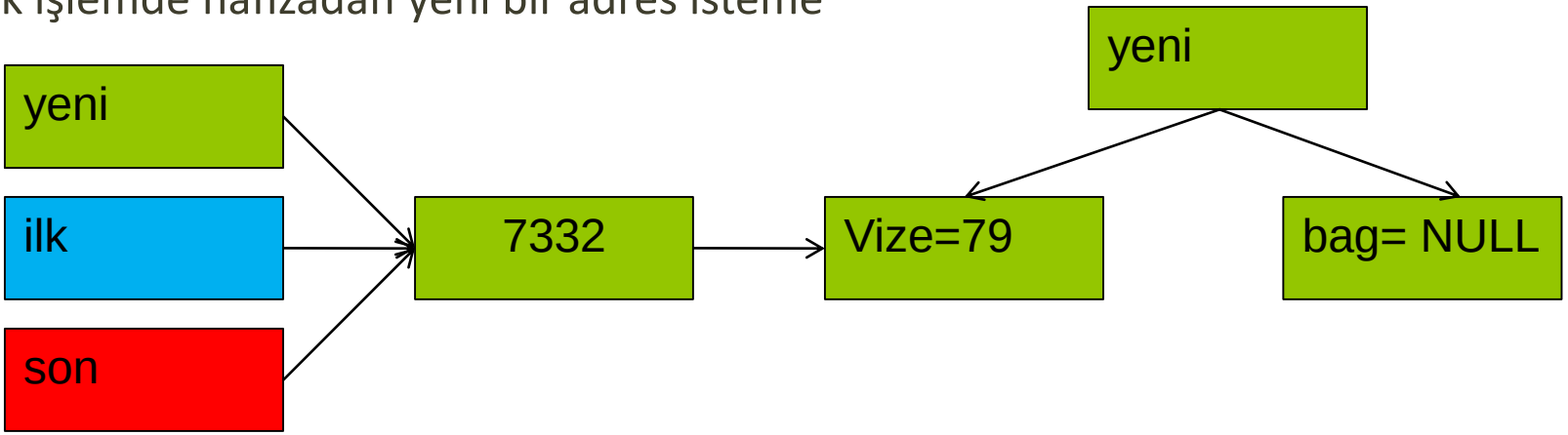
- yeniadres->vize1=79;
- yeniadres->bag=NULL;

C++

- printf("ilk:%d\n",ilk); //İlk elemanın adresini tutar
- printf("son:%d\n",son); //Son elemanın adresini tutar
- printf("T1___yeniadres->vize1:%d yeniadres->bag:%d\n",yeniadres->vize1, yeniadres->bag);
- //Hafızadan tekrar yer iste
- yeniadres=(struct notlar*) malloc(sizeof(struct notlar));
- printf("Yeniadres:%d\n",yeniadres);
- son->bag=yeniadres;
- yeniadres->vize1=95;
- yeniadres->bag=NULL;
- son=yeniadres;
- printf("ilk:%d\n",ilk);
- printf("son:%d\n",son);
- printf("T2___yeniadres->vize1:%d yeniadres->bag:%d\n", yeniadres->vize1, yeniadres->bag);}

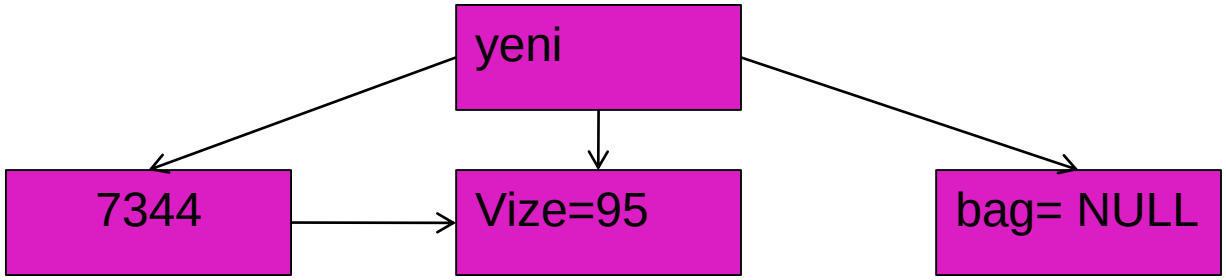
BAĞLI LİSTELER

- İlk işlemde hafızadan yeni bir adres isteme

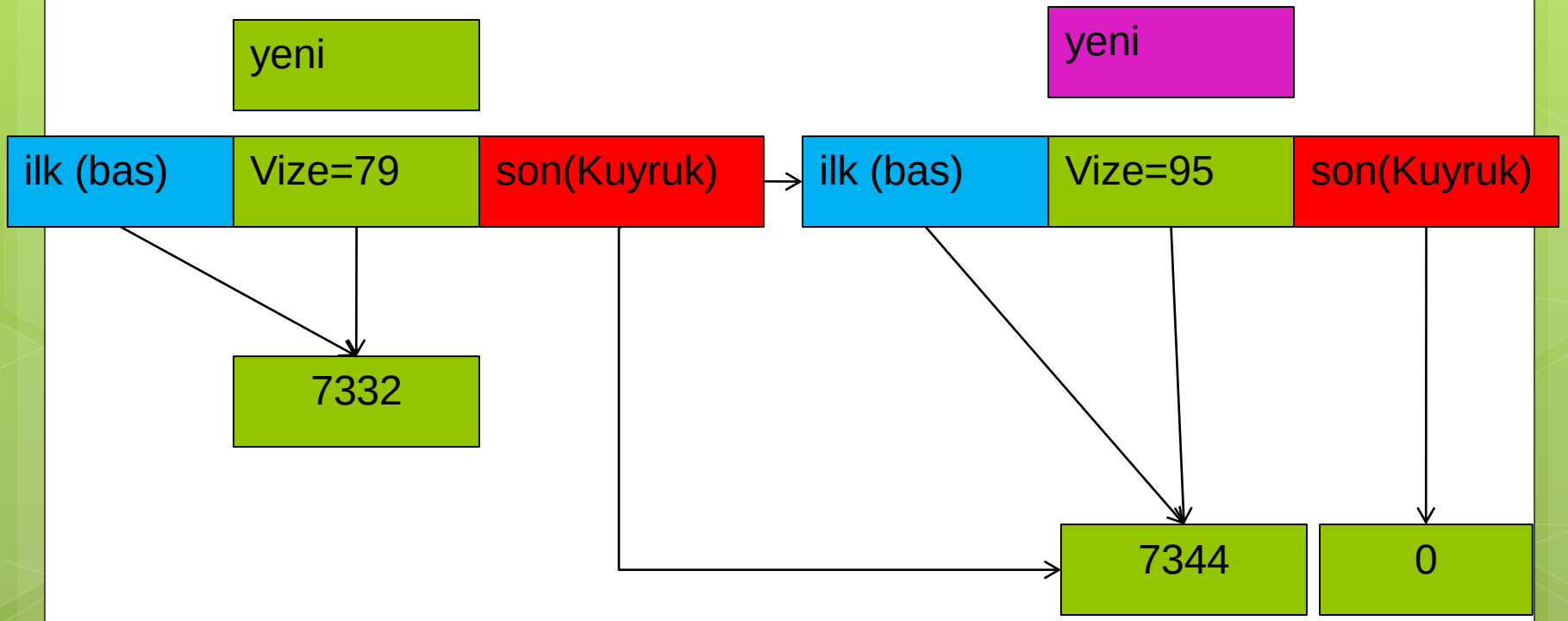


BAĞLI LİSTELER

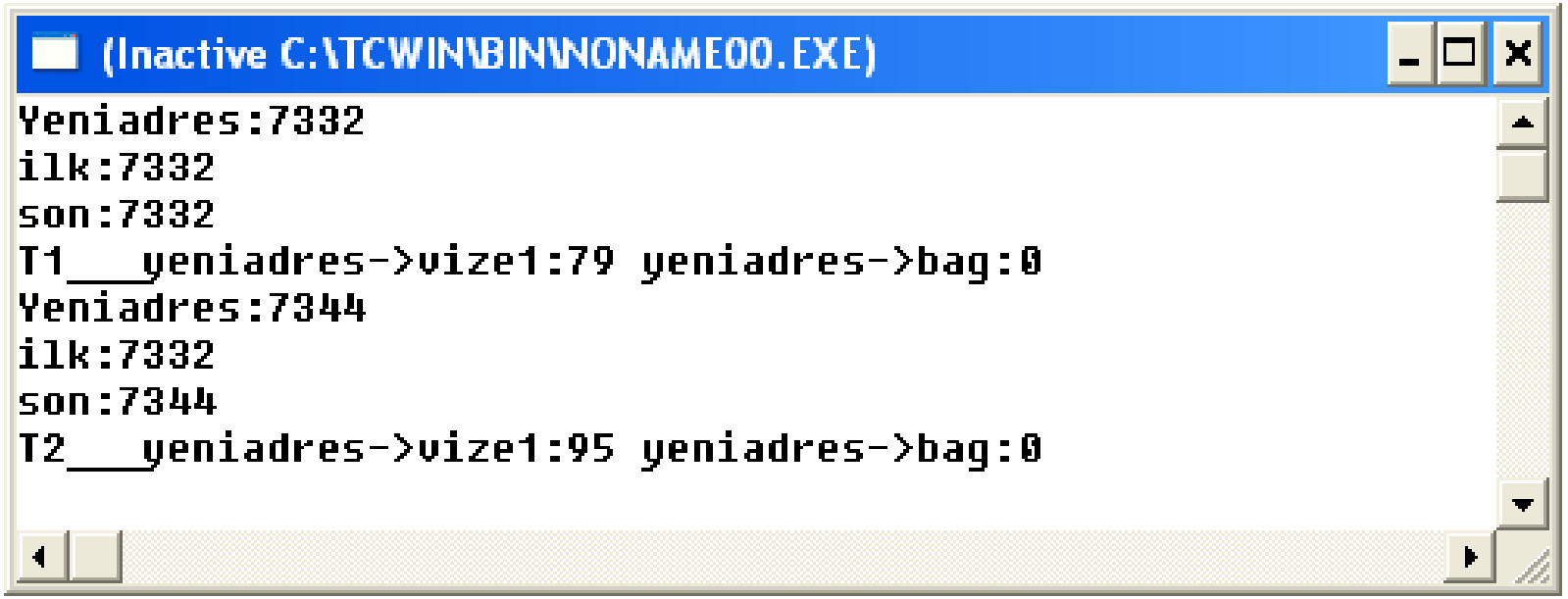
- İkinci işlemde hafızadan yeni bir adres isteme



BAĞLI LİSTELER



C++


(Inactive C:\TCWIN\BIN\NONAME00.EXE)
Yeniadres:7332
ilk:7332
son:7332
T1___yeniadres->vize1:79 yeniadres->bag:0
Yeniadres:7344
ilk:7332
son:7344
T2___yeniadres->vize1:95 yeniadres->bag:0

Örnekler - C++

- **Örnek: 2- Çift yönlü bağlı liste ile filim adlarının saklanması**
- #include <stdio.h>
- #include <stdlib.h>
- #include <string.h>
- #define TSIZE 45
- struct filim {
- char baslik[TSIZE];
- int rating;
- struct filim * sonraki, * onceki; } ;
- int main(void) {
- struct filim * ilk = NULL;
- struct filim * simdiki, *son;
- char input[TSIZE];
- puts("Filimin basligini girin (sonlandirmak icin enter (boşluk) girin):");

C++

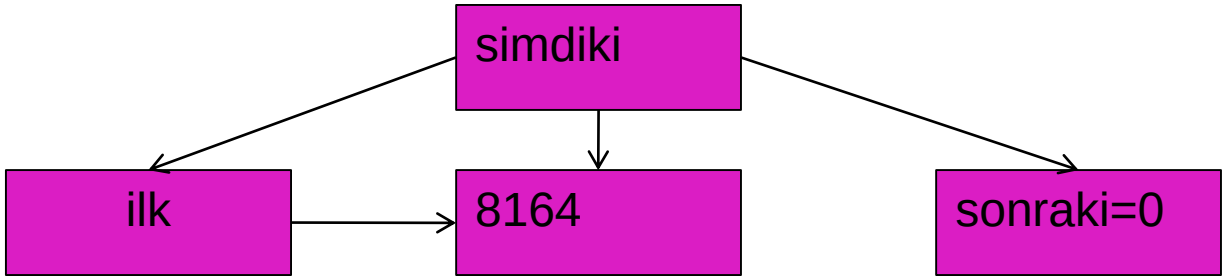
```
while (gets(input) != NULL && input[0] != '\0')    {
      simdiki = (struct filim *) malloc(sizeof(struct filim));
      if (ilk == NULL)     ilk = simdiki;
      else       {son->sonraki = simdiki; şimdiki->önceki=son;}

      simdiki->sonraki = NULL;
      strcpy(simdiki->baslik, input);
      puts("Ratingi girin <0-10>:");
      scanf("%d", &simdiki->rating);
      while(getchar() != '\n')     continue;
      puts("Filimin basligini girin (sonlandirmak icin bosluk girin):");
      son = simdiki;
   }
```

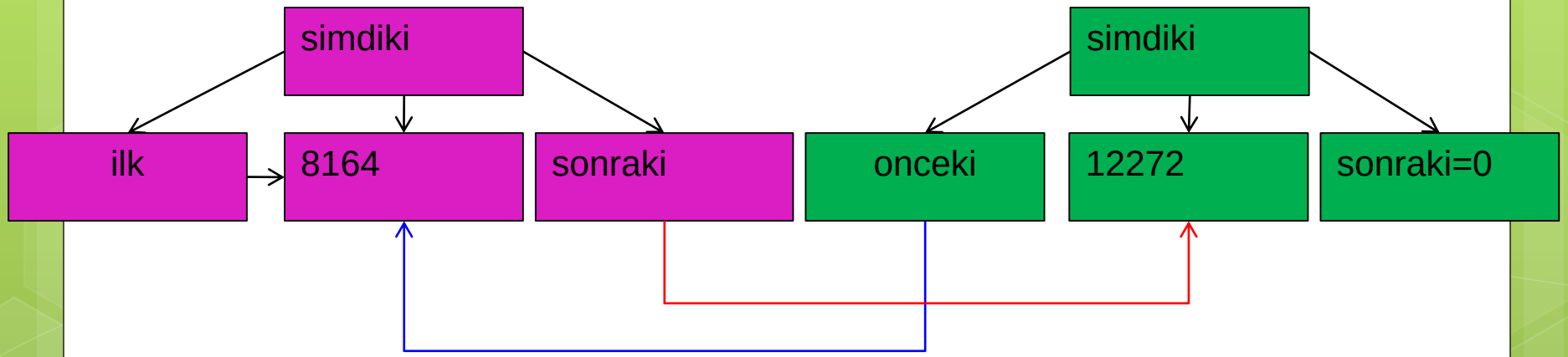
C++

```
if (ilk == NULL)
      printf("Veri girilmemistir. ");
   else
      printf ("Filim Listesi:\n");
   simdiki = ilk;
   while (simdiki != NULL)
   {
  printf("Filim: %s  Rating: %d\n", simdiki->baslik, simdiki->rating);
      simdiki = simdiki->sonraki;
   }
simdiki = ilk;
   }
```

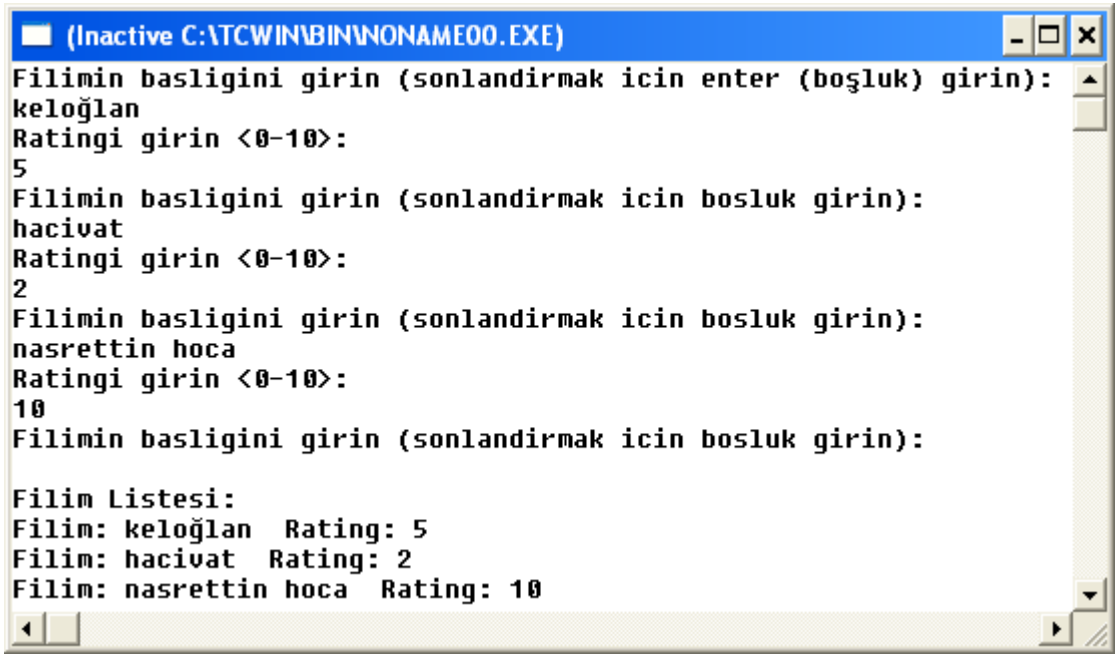
LİSTELER ve BAĞLI LİSTELER



LİSTELER ve BAĞLI LİSTELER



LİSTELER ve BAĞLI LİSTELER
Örnekler



C# Programlama Dilinde Bağlı Liste Örneği ve Kullanımı-TEK YÖNLÜ BAĞLI LİSTE Düğüm Yapısı

```
public class ListNode
{
public string numara, adSoyad;
public double ortalama;
public ListNode sonraki;
public ListNode(string numara, string adSoyad, double ortalama)
{
   this.numara = numara; ;
   this.adSoyad = adSoyad;
   this.ortalama = ortalama;
}
}
```

C# -TEK YÖNLÜ BAĞLI LİSTE Yapısı

- public class LinkedList
- {
- public ListNode headNode, tailNode;
- public LinkedList()
- {
- headNode = new ListNode("head","",o);
- tailNode = new ListNode("tail","",o);
- headNode.sonraki = tailNode;
- tailNode.sonraki = tailNode;
- }
- }

C# -TEK YÖNLÜ BAĞLI LİSTE
Düğüm Ekleme

```
public void Ekle(LinkedList bL, ListNode lN, string yer)
{  ListNode aktif = bL.headNode;
   if (yer == "BASA" )
   { lN.sonraki = aktif.sonraki;
    aktif.sonraki = lN; }
else if (yer == "SIRAYA" )
{
while ((aktif.sonraki!= bL.tailNode) && (string.Compare(aktif.sonraki.numara, lN.numara)<0))
   aktif = aktif.sonraki;
   lN.sonraki = aktif.sonraki;
   aktif.sonraki = lN;
} }
else if (yer == "SONA" )
{ while (aktif.sonraki != bL.tailNode)
    aktif = aktif.sonraki;
lN.sonraki = aktif.sonraki;
aktif.sonraki = lN;
}  }
```

C# -TEK YÖNLÜ BAĞLI LİSTE
Düğüm Silme

- **//Baştan düğüm silme**

- if (rBBas.Checked)
- bagliListe.headNode.sonraki = bagliListe.headNode.sonraki.sonraki;

- **//Sondan Düğüm silme**
- else if (rBSon.Checked)
- {
- ListNode aktif = bagliListe.headNode;
- while (aktif.sonraki.sonraki != bagliListe.tailNode)
- aktif = aktif.sonraki;
- aktif.sonraki = bagliListe.tailNode;
- }

C# -TEK YÖNLÜ BAĞLI LİSTE
Düğüm Silme

```
//Tümünü Silme
else if (rBTumu.Checked)
{
    ListNode aktif = bagliListe.headNode;
    while (aktif.sonraki != bagliListe.tailNode)
      aktif.sonraki = aktif.sonraki.sonraki;
}
//Aranan kişiyi Silme
else if (rBKisi.Checked)
{
ListNode aktif = bagliListe.headNode;
while((aktif.sonraki!=bagliListe.tailNode) &&(aktif.sonraki.numara!=tBNum.Text))
aktif = aktif.sonraki;
aktif.sonraki = aktif.sonraki.sonraki;
}
```

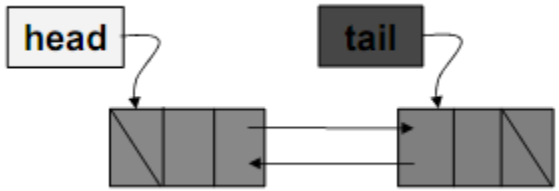
C# -TEK YÖNLÜ BAĞLI LİSTE
Düğüm Silme

```
//İstenilen Sırada Silme
else if (rBSira.Checked)
{   int Sira = Convert.ToInt16(tBSil.Text);       int i = 1;
    if (Sira != 0)
    {    ListNode aktif = bagliListe.headNode;
         while ((aktif.sonraki != bagliListe.tailNode) && (i < Sira))
         {    aktif = aktif.sonraki;     i++;    }

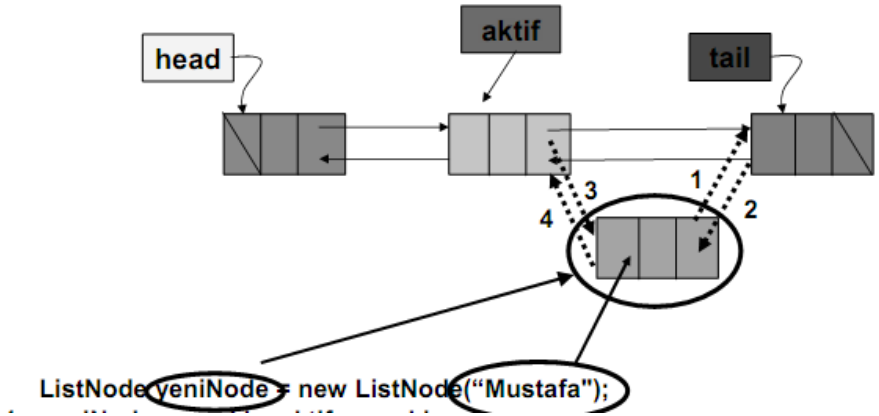
       if ((aktif.sonraki == bagliListe.tailNode) && ((i-1) < Sira))
       MessageBox.Show("Listedeki eleman sayısından büyük değer girildi !" ,
                                    "Hata !", MessageBoxButtons.OK);
        else if (!(aktif.sonraki == bagliListe.tailNode))
       {    aktif.sonraki = aktif.sonraki.sonraki;   }
    }
}
```

C# Programlama Dilinde İKİ YÖNLÜ BAĞLI LİSTE

- **İKİ YÖNLÜ BAĞLI LİSTE**
- public class ListNode {
- public string adSoyad;
- public ListNode onceki, sonraki;
- public ListNode(string adSoyad)
- { this.adSoyad = adSoyad; }
- }
- public class LinkedList {
- public ListNode headNode, tailNode;
- public LinkedList()
- {
- headNode = new ListNode("head");
- tailNode = new ListNode("tail”);
- headNode.onceki = headNode;
- headNode.sonraki = tailNode;
- tailNode.onceki = headNode;
- tailNode.sonraki = tailNode;
- }
- }

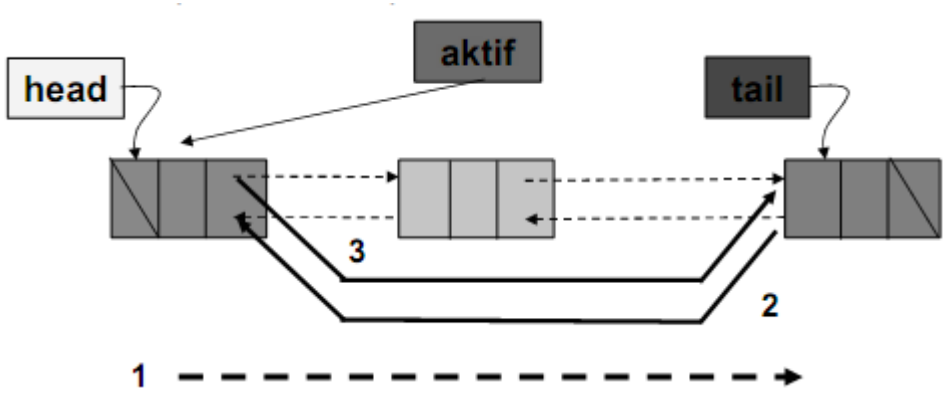


C# Programlama Dilinde Bağlı Liste Örneği ve Kullanımı



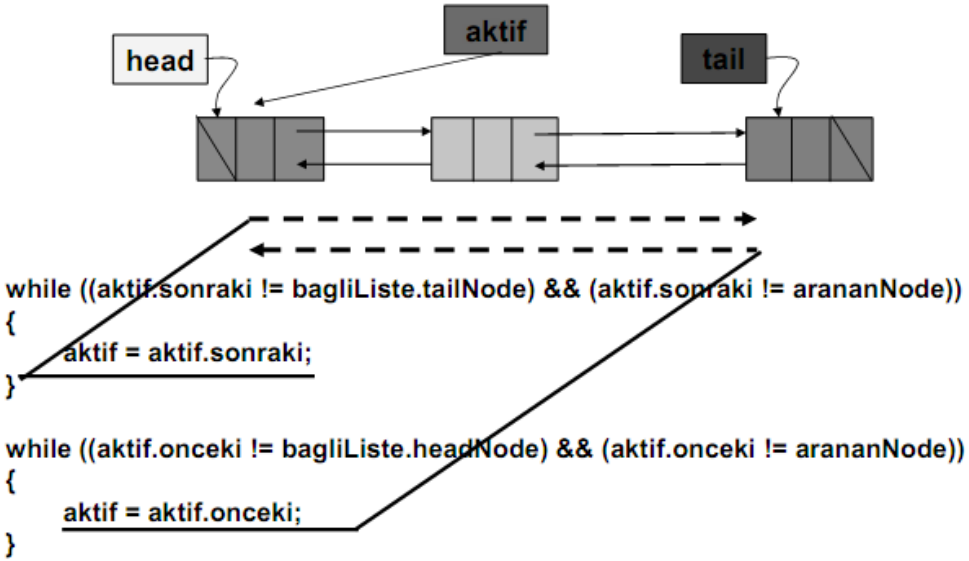
- Eleman ekleme
- ListNode yeniNode = new ListNode("Mustafa");
- yeniNode.sonraki = aktif.sonraki;
- aktif.sonraki.onceki = yeniNode;
- aktif.sonraki = yeniNode;
- yeniNode.onceki = aktif;

C# Programlama Dilinde Bağlı Liste Örneği ve Kullanımı



- Eleman silme
- while ((aktif.sonraki != bagliListe.tailNode) && (aktif.sonraki != silinecekNode))
- {
- aktif = aktif.sonraki;
- }
- aktif.sonraki.sonraki.onceki = aktif;
- aktif.sonraki = aktif.sonraki.sonraki;

C# Programlama Dilinde Bağlı Liste Örneği ve Kullanımı



- Eleman arama
- while ((aktif.sonraki != bagliListe.tailNode) && (aktif.sonraki != arananNode))
- { aktif = aktif.sonraki;}
- while ((aktif.onceki != bagliListe.headNode) && (aktif.onceki != arananNode))
- { aktif = aktif.onceki; }

C# Programlama Dilinde İki Yönlü Bağlı Liste-Kuyruk

```
class Node
{
public object veri; //Düğümde tutulacak veri
public Node onceki, sonraki;//Düğümün adres bilgileri
public Node(object veri)
{
this.veri = veri;
this.onceki = null;
this.sonraki = null;
}
}
public class BList
{
Node ilkNode = new Node("");
Node sonNode = new Node("");
public int s = 0;
public BList()//Bağlı liste yapısı kurulur
{
this.ilkNode.sonraki = this.sonNode;
this.sonNode.onceki = this.ilkNode;
```

C# Programlama Dilinde İki Yönlü Bağlı Liste-yığın

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;
namespace ConsoleApplication1_4.HAFTA
{
class Node
{
public object veri; //Düğümde tutulacak veri
public Node onceki, sonraki;//Düğümün adres bilgileri
public Node(object veri)
{
this.veri = veri;
this.onceki = null;
this.sonraki = null;
}
}
public class BList
{
```

Java Programlama Dilinde İki Yönlü Dairesel Bağlı Liste

```
class Node {

   Object data;
   Node prev;
   Node next;

   public Node( Object data ) {
      this.data = data;
   }

   public Node( Object data, Node prev, Node next ) {
      this.data = data;
      this.prev = prev;
      this.next = next;
   }
}

public class CircularDoublyLinkedList {
```

Örnekler - C++

- **Örnek:- Çift yönlü bağlı liste**
-

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>

typedef struct kisi
{
        char isim[50];
        struct kisi *sonra;
        /*Bu pointera sürekli bir sonraki structı gösterecek olan adres atanacak.
          Bu sayede her struct kendinden sonraki structun bellekte
        nerede olduğunu bilecek*/
        struct kisi *once;
        /*Bu pointera sürekli bir onceki structı gösterecek olan adres atanacak.
          Bu sayede her struct kendinden onceki structun bellekte
        nerede olduğunu bilecek*/
};

struct kisi *yeni , *ilk , *son;/*struct kisi tipinde yeni , ilk ve son isminde tanımlamalar yaptık.*/

void menu();            void dugum_ekleme();        void listele();   void silme();
```

Örnekler - C++

```
int main()
{
   yeni=NULL;    ilk=NULL;    son=NULL;      /*Bütün structları NULL yaptık*/

   menu();/*Menu fonksiyonunu çağırdık*/
   getch();
   return 0;
}
/*****************************/
/*Menu*/
void menu()
{
    system("cls");/*Ekran temizleme fonksiyonu*/
    int cevap;
    printf("\n\n\n\n\t\t1-Kayit ekle\n");
    printf("\t\t2-Kayitlari listele\n");
    printf("\t\t3-Kayit sil\n");
    printf("\t\t4-Cikis\n");
    printf("\t\tLutfen seciminizi yapiniz :");
    scanf("%d",&cevap);
    switch(cevap)
    {
              case 1 : dugum_ekleme(); break;
              case 2 : listele(); break;
              case 3 : silme(); break;
              case 4 : exit(1); break;

    }

}
```

Örnekler - C++

```
/*Dugum Ekleme Fonksiyonu*/
void dugum_ekleme()
{
    if(ilk==NULL)/*Dugumde eleman var mı?*/
    {      /*Dugumde eleman yoksa ilk=NULL olacaktır.*/
       struct kisi *yeni = ((kisi *) malloc(sizeof(kisi)));       /*struct kisi tipinde yeni tanımladık ve bellekten structın boyutu kadar yer aldık*/
       printf("\n\tLutfen kisinin adini giriniz : ");
       scanf("%s",&yeni->isim);/*Yeni icin bilgiler kullanıcıdan alındı*/
       yeni->sonra=NULL;       /*Yeni alınan bilgilerden sonra hiç bi eleman olmadığı için NULL yapıldı*/
       ilk=yeni;       son=yeni;       /*Dügüm için alınan ilk eleman olduğundan yeni ve son ilke eşitlendi*/
    }
    else
    {
       struct kisi *yeni = ((kisi *) malloc(sizeof(kisi)));
       /*struct kisi tipinde yeni tanımladık ve bellekten structın boyutu kadar yer aldık*/
       printf("\n\tLutfen kisinin adini giriniz : ");
       scanf("%s",&yeni->isim);/*Yeni icin bilgiler kullanıcıdan alındı*/
       yeni->sonra=NULL;         /*Yeni alınan bilgilerden sonra hiç bi eleman olmadığı için NULL yapıldı*/
       son->sonra=yeni;       /*Bi önceki structun sonra adindaki göstericisi yeniye eşitlendi*/
       yeni->once=son;  /*Tek yönlü bağlı liste oluşturmadan tek farkı*/
       /*Su an oluşturulmuş olan structun once adindaki gostericisi bir onceki structa eşitlendi*/
       son=yeni;
       /*Son adındaki struct artık yeni structı oldu.       Bunun yapılmasındaki amac       yeni bir dugum ekleneceginde en son hangi structın eklendigini bilmek
       en son eklenen structun sonrasını yeni structına esitlemek       yeni oluşturulan structun once adındaki gostericisini sona eşitlemek.*/
    }
    printf("\n\tAna menuye donmek icin bir tusa basiniz...");
    getch();
    menu();
}
```

Örnekler - C++

```
/*Kayit listeleme*/
void listele()/*Tek yönlü bağlı listeyi listeleme ile aynıdır*/
{
   struct kisi *ara;
   /*struct kisi tipinde ara isminde bir tip tanımladık*/
   printf("\n\n");
   /*ara structunu ilk structa eşitledik ve NULL görene kadar dügümde
ilerledik*/
   for(ara=ilk;ara!=NULL;ara=ara->sonra)
   {
                printf("\t\t->%s \n",ara->isim);

   }
   printf("\n\tAna menuye donmek icin bir tusa basiniz...");
   getch();
   menu();

}
/**********************************/
```

Örnekler - C++

```
/*Kayit silme*/
void silme()
{   char ad[20];     struct kisi *sil;
    printf("\n\tSilmek istediginiz adini giriniz..");     scanf("%s",&ad);
    sil=ilk;
     while(sil)
     {       if(!strcmp(ad,sil->isim))/*Aranan isim dugumde bulunursa*/
             break;
             sil=sil->sonra;
     }
     if(sil==NULL)
    printf("\tBulunamadi.\n");
    else if(sil==ilk)
    {          ilk->sonra->once=NULL;                          ilk=ilk->sonra;
             free(sil);              printf("\tKayit silindi..");
    }
    else if(sil==son)
    {       son=sil->once;                                     sil->once->sonra=NULL;
             free(sil);               printf("\tKayit silindi..");
    }
    else
    {       sil->once->sonra=sil->sonra;
             sil->sonra->once=sil->once;
             free(sil);              printf("\tKayit silindi..");
    }
    printf("\n\tAna menuye donmek icin bir tusa basiniz...");
    getch();       menu();
}
```

C++ Dairesel Bağlı listeler ile oyun

```
#include "stdio.h"
#include "conio.h"
#include "stdlib.h"
#include "string.h"
#include "iostream.h“
//OYUNCULARIN İSİMLERİ----------------------------------------------------------------
char *isimler[10]= {"HACER", "GULTEN","IDRIS", "HASAN","HATİCE",
"CEMIL","BELMA", "CANSU","EBRU","NALAN"};
//-------------------------------------------------------------------------------------
typedef struct veri{
    char adi[20];
    struct veri *arka;
    } BLISTE;
BLISTE *ilk=NULL,*son=NULL;
//-----------------------------------------------
```

Örnekler-Dairesel Bağlı listeler ile oyun

```
int ekle(BLISTE *sayigeldi)
{
  if(ilk==NULL)
   {
     ilk=sayigeldi;
     son=ilk;
     ilk->arka=ilk;
   }
  else
  {
    son->arka=sayigeldi;
    son=sayigeldi;
    son->arka=ilk;
  }
    return 0;
}
//---------------------------------------------------------------------------
```

Örnekler-Dairesel Bağlı listeler ile oyun

- int oyuncular()
- { BLISTE *p; int x=3,y=3,i=0; clrscr();
- gotoxy(x,y);printf("DAIRESEL BAGLI LISTE");y+=2;
- gotoxy(x,y);printf("ODEVIN KONUSU : Dairesel Bagli Liste Yapisini Kullanarak Oyun Tasarlama");
- y+=3; gotoxy(x,y);printf("Oyuncular : "); p=ilk;
- while(p->arka!=ilk) { printf("%s\n ",p->adi); p=p->arka; i++; }
- printf("%s\n ",p->adi);
- y+=13;
- gotoxy(x,y);printf("TOPLAM OYUNCU SAYISI : %d ",i); getch();
- return 0;}
- //---

Örnekler-Dairesel Bağlı listeler ile oyun

```
int oyundan_at(BLISTE *onceki,BLISTE *silinecek) {
 if(ilk->arka==ilk) {      //listede bir eleman varsa
       clrscr();
       printf("Oyunda Su an Sadece 1 kisi var");
       printf("OYUNU KAZANAN = %s",ilk->adi);    }
  else if(onceki->arka==ilk) //silinecek ilk eleman mı?
{     onceki->arka=ilk->arka;    ilk=ilk->arka;
      printf(" Oyundan Cıkıyor....%s",onceki->adi);
      free(silinecek); oyuncular();   }
   else if(silinecek==son)//silinecek son eleman mı?
 {       onceki->arka=ilk;         son=onceki;     free(silinecek); oyuncular();   }
   else //silinecek aradan mı?
   {         onceki->arka=silinecek->arka;   free(silinecek); oyuncular();   }
 return 0;}
//-------------------------------------------------------------------------------------
```

Örnekler-Dairesel Bağlı listeler ile oyun

```
void main(void) {
  clrscr();  randomize();
  BLISTE *yeni;int i,x=3,y=3;
        for( i=0;i<10;i++)   {
        yeni=(BLISTE *)malloc(sizeof(BLISTE));
        if(!yeni)   {        clrscr();
       printf("Oyuncuları Ekleme Yapacak Kadar Bos Alan Yok Kusura Bakmayiniz");
        exit(0);   }
        else   {     strcpy(yeni->adi,isimler[i]);     ekle(yeni);    }
  }
clrscr();x=3;y=3; int soylenen;
oyuncular();
yeni=ilk;     BLISTE *p;    BLISTE *bironceki;
```

Örnekler-Dairesel Bağlı listeler ile oyun

- do {
- printf("\n\nSAYIN :%s ---> Bir Sayi Soyler misiniz ? ",ilk->adi);
- scanf("%d",&soylenen);
- for(i=0;i<soylenen;i++) {
- yeni->adi; bironceki=yeni; yeni=yeni->arka; }
- //bağı koparılacak olan.. adresi=yeni, bir oncekinin adresi bironcekinde, tahmini p ile adreslenen kişi yapacak
- p=yeni->arka;//sayıyı tahmin edecek kişi
- oyundan_at(bironceki,yeni); yeni=p;
- } while(ilk!=son);
- if(ilk==son) { clrscr(); x=5; y=5;
- gotoxy(x,y); printf("Oyunda Su an Sadece 1 kisi var");
- gotoxy(x,y);printf("OYUNU KAZANAN = %s",ilk->adi); y+=2;
- gotoxy(x,y);printf("Oyun bitmistir...programi sonlandırmak ıcın herhangı bır tusa basınız."); y+=2; getch(); exit(0); }
- getch(); }

Örnekler -Java

- Örnek : Java'da dizi kullanarak karakter yığıtı sınıfı oluşturma ve kullanma.
- import java.io.*;
- class StackChar
- {
- private int maxSize; private char[] stackArray; private int top;
- public StackChar(int max)
- { maxSize = max; stackArray = new char[maxSize]; top= -1; }
- public void push(char j) { stackArray[++top] = j; }
- public char pop() { return stackArray[top--]; }
- public boolean isEmpty() { return top==-1; }
- }

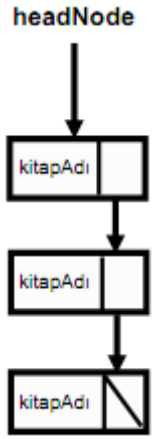
Örnekler –Java

```
class Reverse
{
  public static void main(String args[])
  {
    StackChar y = new StackChar(100);
    String str = "Merhaba";
    for(int  i=0;  i<str.length();  ++i)
     y.push(str.charAt(i));
    while(! y.isEmpty() ) System.out.println(y.pop());
  }
}
```

Örnekler –C#

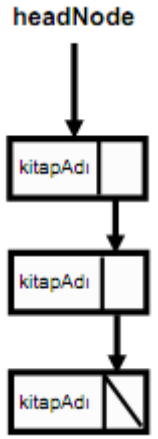
- **Örnek:** Bağlı liste ile yığıt oluşturma
- **class stackNodeC**
- **{**
- **public string kitapAdi; public stackNodeC sonraki;**
- **public stackNodeC(string kitapAdi) { this.kitapAdi = kitapAdi; }**
- **}**

- **class stackC**
- **{**
- **public stackNodeC headNode;**
- **public stackC(string kitapAdi)**
- **{**
- **this.headNode = new stackNodeC(kitapAdi);**
- **this.headNode.sonraki = headNode;**
- **}**
- **}**
- **stackC kitapYigin = new stackC("");**



Örnekler –C#

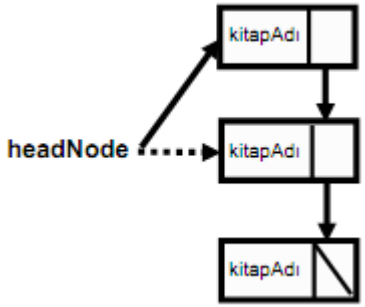
- /* Stack işlemleri :
- boş yığın stackSize() == 0 eleman sayısı= stackSize()
- eleman ekleme= push(kitapAdi) eleman alma= pop() */



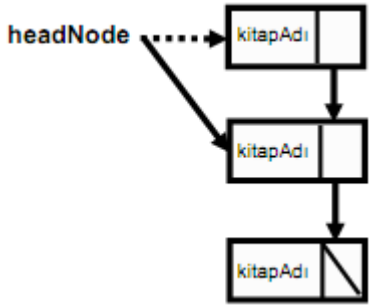

```
public int stackSize()
{
stackNodeC aktif = new stackNodeC("");
aktif = kitapYigin.headNode;
int i = 0;
while (aktif.sonraki != aktif)
{
aktif = aktif.sonraki;   i++;
}
return i;
}
```

Örnekler –C#

- // Stack işlemleri (eleman ekleme)
- public void push(string kitapAdi)
- {
- if (stackSize() >= MaxSize)
- MessageBox.Show ("Yığın maksimum elemana sahip ! Yeni
- eleman eklenemez !", "Dikkat");
- else
- { stackNodeC yeniNode = new stackNodeC(tBKitapAdi.Text);
- yeniNode.sonraki = kitapYigin.headNode;
- kitapYigin.headNode = yeniNode;
- lStackSize.Text = "Stack Size =
- "+Convert.ToString(stackSize());
- }
- }



Örnekler –C#




```
// Stack işlemleri (eleman alma)
public void pop()
{
if (stackSize() == 0)
{
MessageBox.Show("Yığında eleman yok !", "Dikkat");
}
else
{
kitapYigin.headNode = kitapYigin.headNode.sonraki;
}
}
```

Örnekler –C++

- **Örnek:** Yığıt (Stack) kullanarak bağlı liste işlemleri. Girilen cümleyi kelimeleri bozmadan tersten yazdır.
- **#include <stdio.h>**
- **#include <string.h>**
- **#include <conio.h>**
- **#include <stdlib.h>**
- **struct kelimeler**
- **{ char kelime[50];**
- **struct kelimeler *onceki;**
- **} *tepe,*yenieleman,*yedek;**

Örnekler –C++

```
void push(char *gelen) {
  yenieleman =(struct kelimeler *) malloc(sizeof(struct kelimeler));
  strcpy(yenieleman->kelime, gelen);
  yenieleman->onceki = tepe;
  tepe = yenieleman;  }

int pop(char *giden)  {

   if (tepe != NULL) {
     yedek = tepe;
     strcpy(giden, tepe->kelime);
     tepe = tepe->onceki;   free(yedek);    return 0;    }
   else
     return 1; /* yığın boş */
```

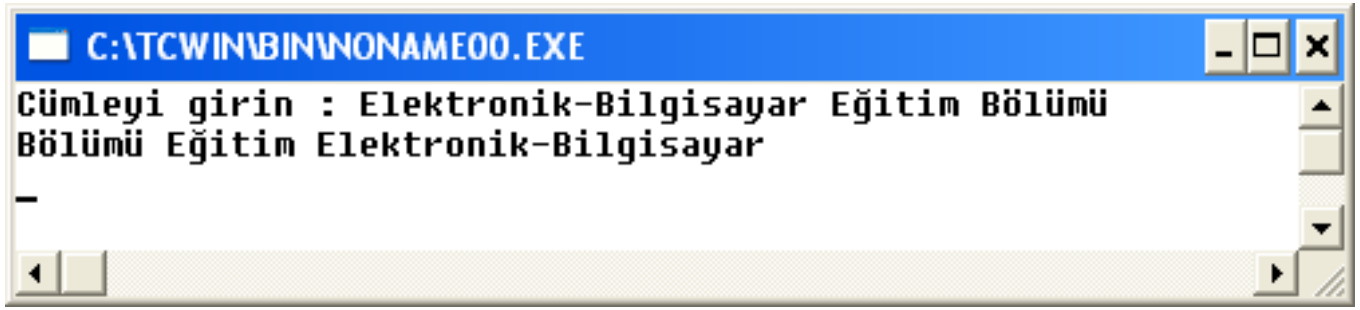
Örnekler –C++

```
main() {
  char *yenikelime, *cumle;
  tepe = NULL;

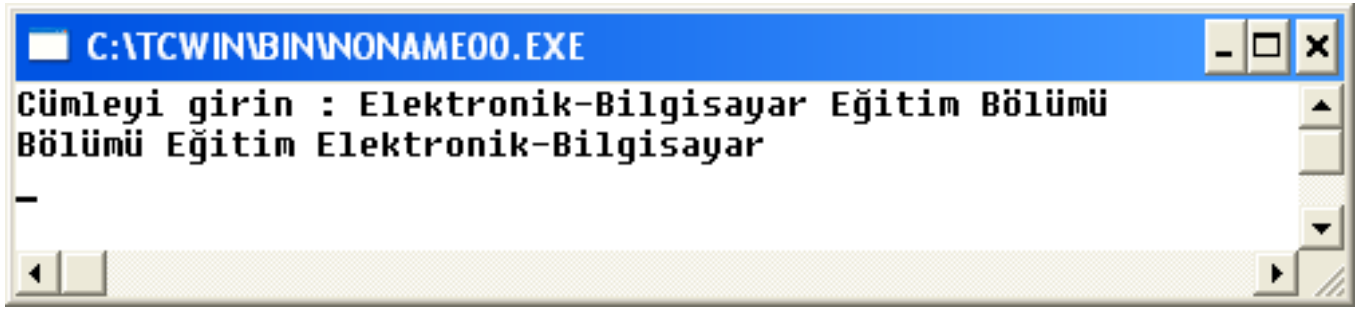
  printf("Cümleyi girin : "); gets(cumle);
  yenikelime = strtok(cumle, " "); // cumle değişkenini boşluğa göre ayırma.

while (yenikelime) {
     push (yenikelime);
     yenikelime = strtok(NULL, " ");
  }
  yenikelime = (char *) malloc(20); /* yenikelime NULL idi, yer açalım*/
  while (!pop(yenikelime))  printf("%s ", yenikelime);
  printf("\n");
  getch();
}
```

Örnekler –C++


C:\TCWIN\BIN\NONAME00.EXE
Cümleyi girin : Elektronik-Bilgisayar Eğitim Bölümü
Bölümü Eğitim Elektronik-Bilgisayar

Örnekler –C++


C:\TCWIN\BIN\NONAME00.EXE
Cümleyi girin : Elektronik-Bilgisayar Eğitim Bölümü
Bölümü Eğitim Elektronik-Bilgisayar

Yığın (Stack) devam

Infix, Postfix, & Prefix Gösterimleri

Yığın Kullanımı - Infix Gösterimi

- Genellikle cebirsel işlemleri şu şekilde ifade ederiz: a + b
- Buna infix gösterim adı verilir, çünkü operatör (“+”) ifade içindedir.
- Problem: Daha karmaşık cebirsel ifadelerde parantezlere ve öncelik kurallarına ihtiyaç duyulması.
- Örneğin:
- a + b * c = (a + b) * c ?
- = a + (b * c) ?

Infix, Postfix, & Prefix Gösterimleri

- Herhangi bir yere operatör koymamamızın önünde bir engel yoktur.
- **Operatör Önde (Prefix) : + a b**
  - Biçim: işlem işlenen işlenen (operator operand operand) şeklindedir: + 2 7
  - İşlem sağdan sola doğru ilerler. Öncelik (parantez) yoktur.
- **Operatör Arada (Infix) : a + b**
  - Biçim: işlenen işlem işlenen (operand operator operand) şeklindedir: 2 + 7
  - İşlem öncelik sırasına göre ve soldan sağa doğru ilerler.
- **Operatör Sonda (Postfix) : a b +**
  - Biçim: işlenen işlenen işlem (operand operand operator) şeklindedir: 2 7 +
  - İşlem soldan sağa doğru ilerler. Öncelik (parantez) yoktur.

Prefix, Postfix : Diğer İsimleri

- Prefix gösterimi Polanyalı (Polish) bir mantıkçı olan Lukasiewicz, tarafından tanıtıldığı için “Polish gösterim” olarak da isimlendirilir.
- Postfix gösterim ise ters Polish gösterim “reverse Polish notation” veya RPN olarak da isimlendirilebilir.

Neden Infix, Prefix, Postfix ?

- Soru: infix gösterimde çalışmayla herşey yolunda iken neden böyle “aykırı”, “doğal olmayan” bir gösterim şekli tercih edilsin.?

- Cevap: postfix and prefix gösterimler ile parantez kullanılmasına gerek yoktur !
- Bir çok compilerlar programlaradaki infix notasyonunda yazili expressionlari postfix notasyonuna çevirerek bellekte saklar.

Infix, Prefix, Postfix İşlemleri

- **İşlem önceliği (büyükten küçüğe)**
  - Parantez
  - Üs Alma
  - Çarpma /Bölme
  - Toplama/Çıkarma

    - Parantezsiz ve aynı önceliğe sahip işlemcilerde soldan sağa doğu yapılır (üs hariç).
    - Üs almada sağdan sola doğrudur. A-B+C'de öncelik (A-B)+C şeklindedir. A^B^C'de ise A^(B^C) şeklindedir. (parantezler öncelik belirtmek için konulmuştur)

Parantez -Infix

- 2+3*5 işlemini gerçekleştiriniz.
- **+** önce ise:
  - (2+3)*5 = 5*5 = 25
- ***** önce ise:
  - 2+(3*5) = 2+15 = 17
- Infix gösterim paranteze ihtiyaç duyar.

Prefix Gösterim

- * + 2 3 5 =
- = * + 2 3 5
- = * 5 5 = 25
- + 2 * 3 5 =
- = + 2 * 3 5
- = + 2 15 = 17
- Paranteze ihtiyaç yok!

Postfix Gösterim

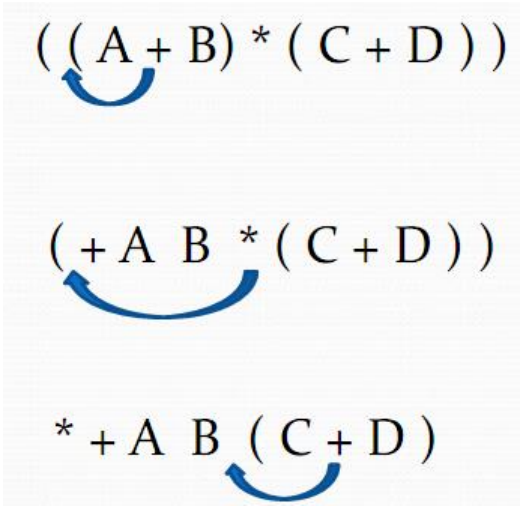
- 2 3 + 5 * =
- = 2 3 + 5 *
- = 5 5 * = 25
- 2 3 5 * + =
- = 2 3 5 * +
- = 2 15 + = 17
- **Sonuç**:
- **Infix işlem sıralarının düzenlenmesi için paranteze ihtiyaç duyan tek gösterim şeklidir**

Tamamen Parantezli Anlatım

- TPA gösterimde her operatör ve işlenenini çevreleyen parantezlerden oluşan tam bir set vardır.
- Hangisi tam parantezli gösterim?
  - (A + B) * C
  - **((A + B) * C)**
  - ((A + B) * (C)) ((A + B) (C))

Infix'ten Prefix'e Dönüşüm

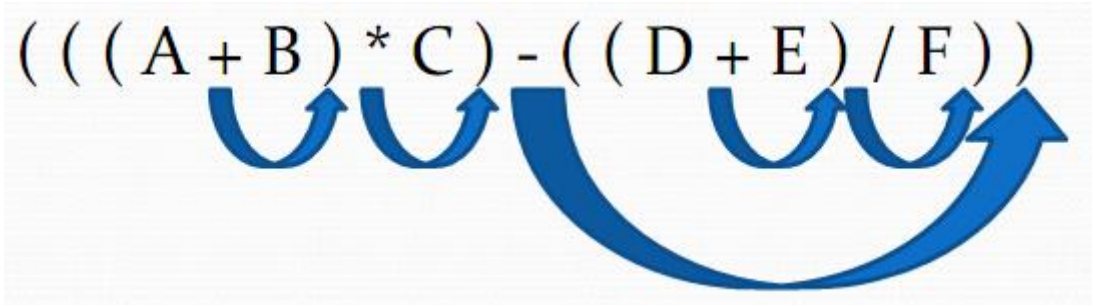
- Her bir operatörü kendi işlenenlerinin soluna taşı ve parantezleri kaldır. :



* + A B + C D

- İşlenenlerin sırasında bir değişiklik olmadı!

Infix'ten Postfix'e Dönüşüm



- ((AB+* C) - ((D + E) / F))
- (AB+C* - ((D + E) / F))
- AB+C* ((D + E) / F)-
- AB+C* (DE+ / F)-
- A B + C * D E + F / -
- İşlenenlerin sırası değişmedi!
- Operatörler değerlendirme sırasına göre!

Infix, Prefix, Postfix

- Aşağıda verilen işlemlerde işleyişe bakınız

Infix	Postfix	Prefix
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD
A^B*C-D+E/F/(G+H)	AB^C*D-EF/GH+/+	+-*^ABCD//EF+GH
((A+B)*C-(D-E))^(F+G)	AB+C*DE—FG+^	^-*+ABC-DE+FG
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Infix, Prefix, Postfix İşlemleri

- **Örnek:** Parantezsiz operatör arada ifadenin operatör sonda hale çevrilmesi :a + b * c – d

Okunan	Yığıt	Çıktı /Operatör sonda ifade
a		a
+	+	a
b	+	a b
*	+ *	a b
c	+ *	a b c
-	+ *	a b c
	+	a b c *
	-	a b c * +
d	-	a b c * + d -

Operatör Sonda (Postfix) İfadenin İşlenişi:

- **Örnek:** a b c * + d - ifadesini a=2 b=3 c=5 d=10 --> 2 3 5 * + 10 -

Okunan	Yığıt	Hesaplanan
2	2	
3	2 3	
5	2 3 5	
*	2	islem=* pop1=5 pop2=3
	2 15	3 * 5=15
+	17	islem=+ pop1=15 pop2=2
		2 + 15 =17
10	17 10	
-	7	islem=- pop1=10 pop2=17
		17 – 10= 7
		a b c * + d -

Infix, Prefix, Postfix İşlemleri

- **Örnek:** Parantezli operatör arada ifadenin operatör sonda hale çevrilmesi. Infix ifade: (2 + 8) / (4 – 3)

Okunan	Yığıt	Hesaplanan
(	(	
2	(	2
+	(+	2
8	(+	2 8
)		2 8 +
/	/	2 8 +
(	/ (	2 8 +
4	/ (	2 8 + 4
-	/ (-	2 8 + 4
3	/ (-	2 8 + 4 3
)	/	2 8 + 4 3 –
		2 8 + 4 3 –
		2 8 + 4 3 – /

Örnek: Java

- postfix olarak yazılmış ifadeyi hesaplayarak sonucu bulan bir program yazınız.
- Örnek olarak "5 3 2 - / 4 7 * + "ifadesinin sonucunu bulunuz.
- Sonuç = 33 olarak bulunacak.

Örnek: Java

```
import java.awt.*;
import java.io.*;
import java.util.*;

public class yigin {
public int top=0;

public void push( int x, int a[], int top)
{   a[top] = x;      ++top;  this.top=top;  }

 public int pop( int a[], int top)
{   --top;         this.top=top;  return a[top]; }
```

Örnek:

```
public static void main(String[] args)  {
yigin metot =new yigin();
String str="";   int x = 0 , a=0,b=0;   int stk[]=new int [120];
Scanner  oku = new Scanner(System.in);
System.out.println("\n Postfix ifadeyi giriniz\n");
str=oku.nextLine();
for(int i=0;i<str.length();i++)
{   if(str.charAt(i) == '+')
     {  b = metot.pop( stk, metot.top);
        a = metot.pop( stk, metot.top);
        metot.push( a + b, stk, metot.top);
        System.out.println("\n a+b ="+ (a+b));
        }
```

Örnek:

```
else if(str.charAt(i) == '-')
    {  b = metot.pop( stk,metot.top);        a = metot.pop( stk, metot.top);
    metot.push( a - b, stk,metot.top);
    System.out.println("\n a-b ="+ (a-b));
    }
  else if(str.charAt(i) == '/')
  { b = metot.pop( stk, metot.top);    a = metot.pop( stk, metot.top);
  metot.push( a / b, stk, metot.top);
  System.out.println("\na/b= "+ a/b);
  }
  else if(str.charAt(i) == '*')
  {  b = metot.pop( stk, metot.top);   a = metot.pop( stk, metot.top);
 metot.push( a * b, stk, metot.top);
 System.out.println("\n a*b= "+ a*b);
 }
```

Örnek:

- **if(str.charAt(i) >= '0' && str.charAt(i) <= '9')**
- **{ x =str.charAt(i)- '0'; metot.push(x, stk, metot.top); }**
- **}**
- **System.out.println("\n Postfix ifadesinin işlem sonucu="+ metot.pop(stk, metot.top));**
- **}**
- Çıktı:
- Postfix ifadeyi giriniz
- 532-/47*+
- a-b =1
- a/b= 5
- a*b= 28
- a+b =33
- Postfix ifadesinin işlem sonucu=33

Infix, Prefix, Postfix İşlemleri

- **Örnek:** Aşağıda boş bırakılan alanları tamamlayınız

Infix	Prefix	Postfix
2x(3+5)-7^(2+1)		
	++x23^-5721	
		235+7-^2x1+
2x3+5-7^2+1		

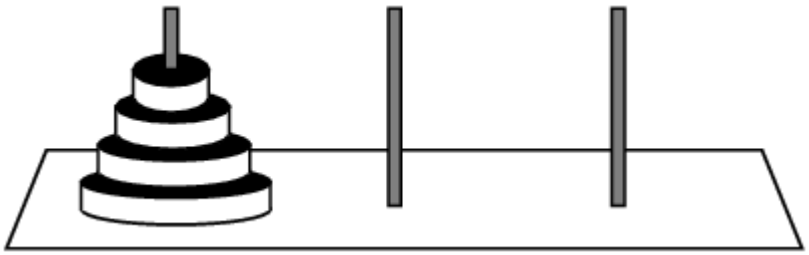
Infix, Prefix, Postfix İşlemleri

- **Cevap**

Infix	**Prefix**	**Postfix**
2x(3+5)-7^(2+1)	**-x2+35^7+21**	**235+x721+^-**
(2x3)+(5-7)^2+1	++x23^-5721	**23x57-2^+1+**
2^ (3+5-7)x2+1	**+x^2-+35721**	235+7-^2x1+
2x3+5-7^2+1	**+-+x235^721**	**23x5+72^-1+**

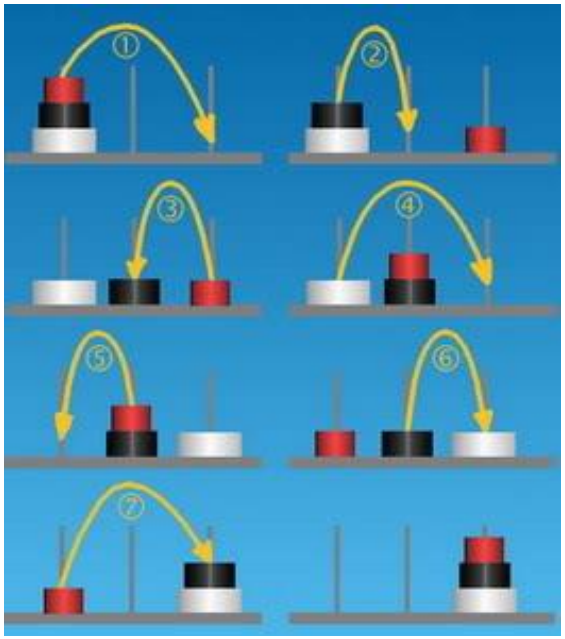
Hanoi Kuleleri Yığın temelli Çözüm

- **Verilen:** üç iğne
  - İlk iğnede en küçük disk en üstte olacak şekilde yerleştirilmiş farklı büyüklükte disk kümesi.
- **Amaç:** diskleri en soldan en sağa taşımak.
- **Şartlar:** aynı anda sadece tek disk taşınabilir.
- Bir disk boş bir iğneye veya daha büyük bir diskin üzerine taşınabilir.



Hanoi Kuleleri Yığın temelli Çözüm

- Problem karmaşıklığı 2^n
- 64 altın disk
- 1 taşıma işlemi 1 saniye sürsün:
- 18446744073709551616 sn
- 593.066.617.596,15 yıl
- Dünyanın sonuna 600.000.000.000 yıl var ☺



Hanoi Kuleleri– Özyinelemeli Çözüm-Java

```
package hanoikuleleri;
import java.util.*;
public class Hanoikuleleri  {

public static void main(String[] args)
{
 System.out.print("n değerini giriniz : ");
 Scanner klavye = new Scanner(System.in);  int n = klavye.nextInt();
 tasi(n, 'A', 'B', 'C');
}
 public static void tasi(int n, char A, char B, char C)
{if(n==1) System.out.println(A + " --> " + B);
 else
 {
 tasi(n-1, A, C, B);      tasi(1, A, B, C);    tasi(n-1, C, B, A); }
 return;
}
```

Ödev

- 1-Infix'ten Prefix ve Postfix'e Çevirin
  - x
  - x + y
  - (x + y) - z
  - w * ((x + y) - z)
  - (2 * a) / ((a + b) * (a - c))
- 2-Postfix'ten Infix'e Çevirin
  - 3 r -
  - 1 3 r - +
  - s t * 1 3 r - + +
  - v w x y z * - + *

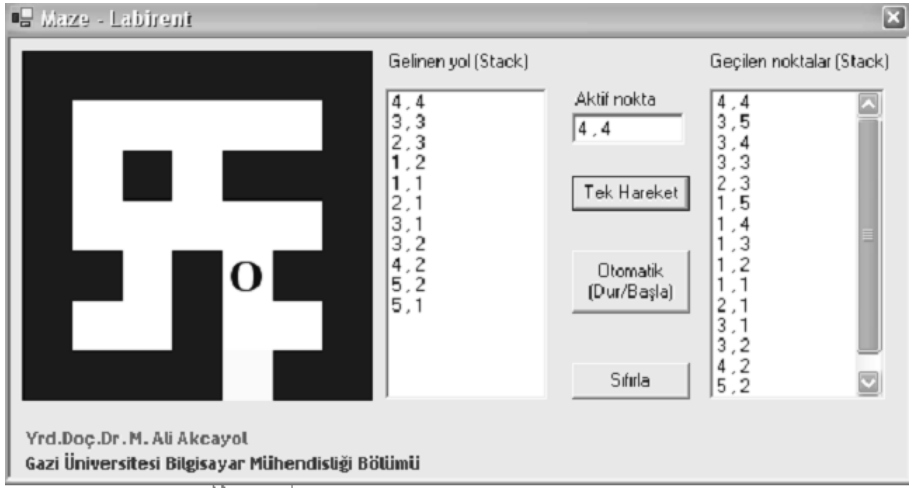
Ödev

- 3- postfix olarak yazılmış ifadeyi hesaplayarak sonucu bulan programı Java/C# bağlı liste yapısı ile yazınız.

- 4-Verilen Maze (Labirent) uygulamasını -başlangıç olarak istediğimiz noktadan başlayarak çıkışa ulaşmasını sağlayınız.



```
1,1,1,1,1,1,1
1,0,0,0,0,0,1
1,0,1,0,1,1,1
1,0,0,0,0,0,1
1,1,0,1,0,1,1
1,0,0,1,0,0,1
1,1,1,1,0,1,1
```



Kuyruk (Queue)

Kuyruk (Queue)



- Kuyruklar, eleman eklemelerinin sondan (rear) ve eleman çıkarmalarının baştan (front) yapıldığı doğrusal veri yapılarıdır.
- Bir eleman ekleneceği zaman kuyruğun sonuna eklenir.
- Bir eleman çıkarılacağı zaman kuyrukta bulunan ilk eleman çıkarılır.
- Bu nedenle kuyruklara FIFO (First In First Out-İlk giren ilk çıkar) veya LILO (Last-in-Last-out-Son giren son çıkar) listeleri de denilmektedir.

Kuyruk (Queue)

- Gerçek yaşamda banklarda, duraklarda, otoyollarda kuyruklar oluşmaktadır. Kuyruğu ilk olarak girenler işlemlerini ilk olarak tamamlayıp kuyruktan çıkarlar.
- İşletim sistemleri bünyesinde öncelik kuyruğu, yazıcı kuyruğu gibi birçok uygulama alanı vardır.
- Kuyruk modeli, program tasarımında birçok yerde kullanılır. Örneğin, iki birimi arasında iletişim kanalı, ara bellek/tampon bellek oluşturmada bu modele başvurulur.

Kuyruk (queue)

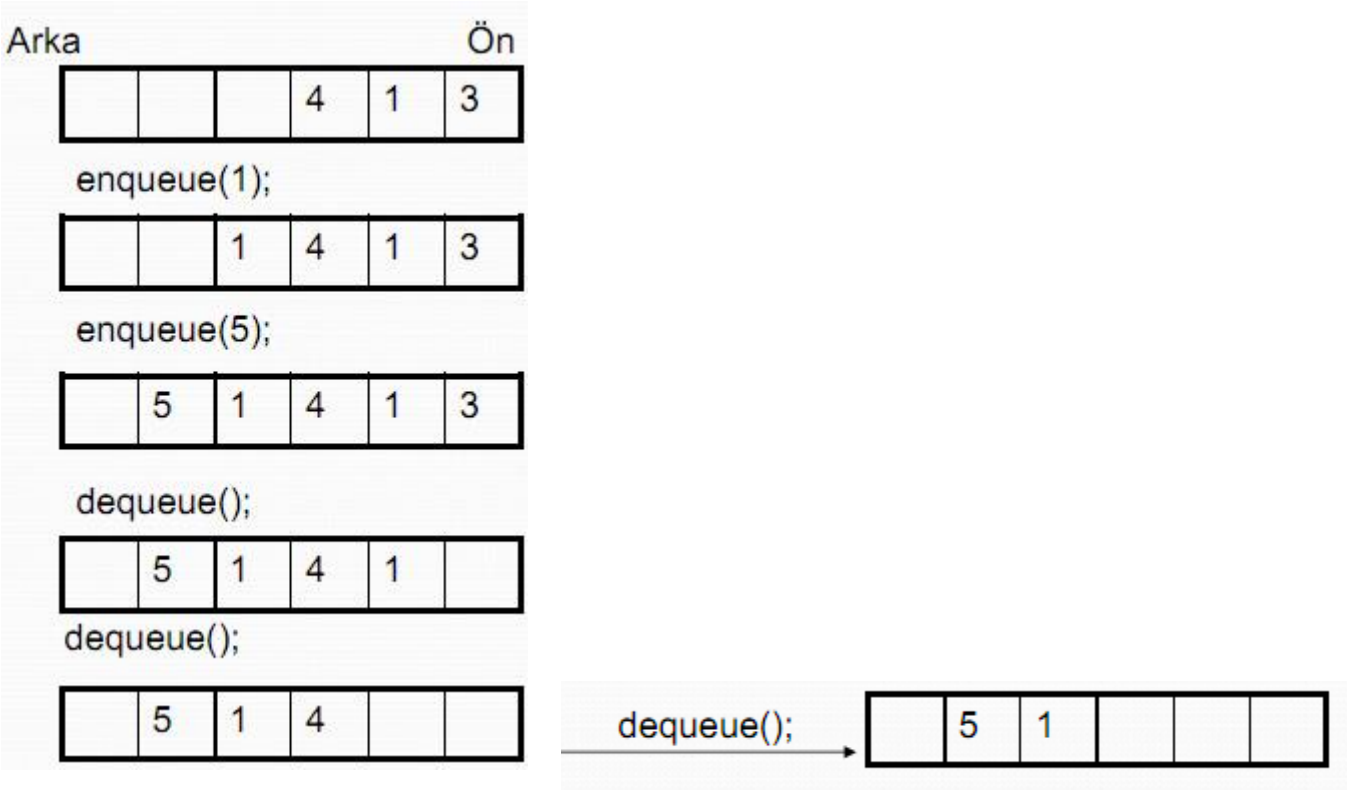
- Ardışıl nesneleri saklarlar
- Ekleme ve silme FIFO'ya göre gerçekleşir.
- Ekleme işlemi kuyruk arkasına yapılırken, çıkarma işlemi ise kuyruk önünden yapılır.
- **Ana Kuyruk İşlemleri:**
- İki tane temel işlem yapılabilir ;
  - **enqueue (object)**, bir nesneyi kuyruğun en sonuna ekler (insert).
  - **object dequeue ()**,
    Kuyruk başındaki nesneyi getirir ve kuyruktan çıkarır (remove veya delete).

Kuyruk Veri Yapısı Bileşenleri

- **Yardımcı kuyruk işlemleri:**
  - object front() (getHead/getFront): kuyruk başındaki nesneyi kuyruktan çıkarmadan geri döndürür.
  - integer size(): kuyrukta saklanan nesne sayısını geri döner
  - boolean isEmpty(): Kuyrukta nesne olup olmadığını kontrol eder.

- **İstisnai Durumlar (Exceptions)**
  - Boş bir kuyruktan çıkarma işlemi yapılmak istendiğinde veya ilk nesne geri döndürülmek istendiğinde EmptyQueueException oluşur.

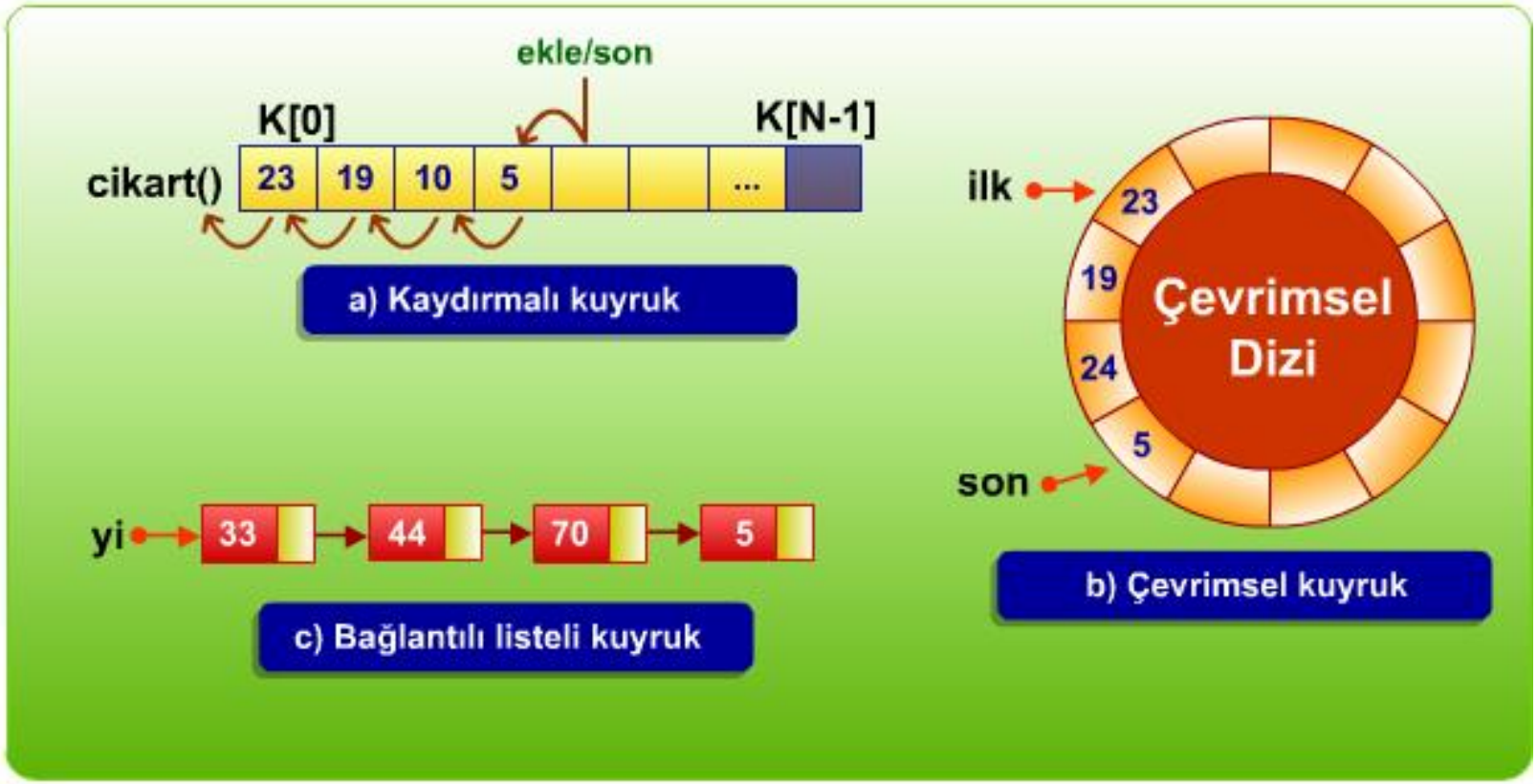
Kuyruk Ekleme/Çıkarma



Kuyruk Tasarımı

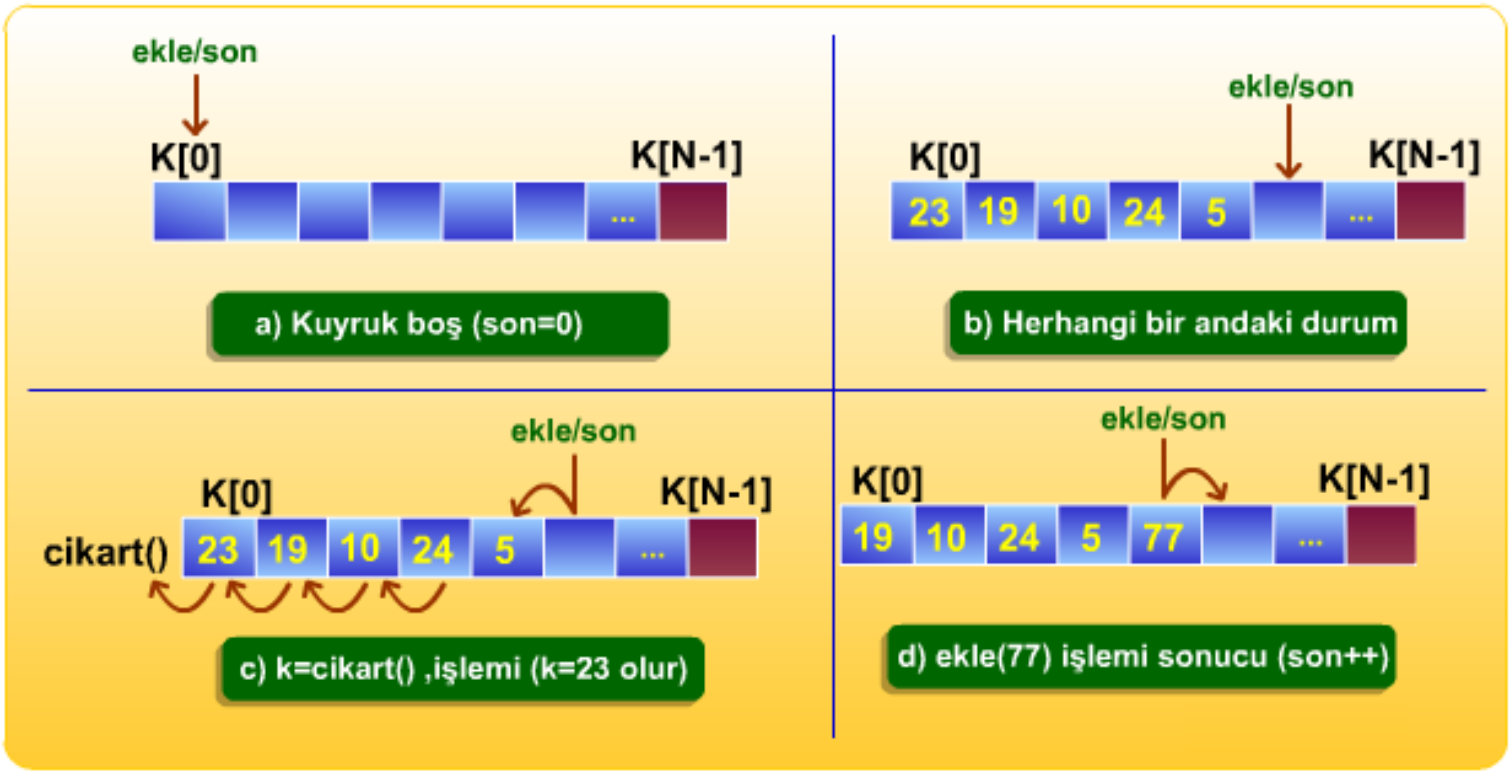
- Kuyruk tasarımı çeşitli şekillerde gerçekleştirilebilir. Biri, belki de en yalın olanı, bir dizi ve bir indis değişkeni kullanılmasıdır.
- Dizi gözlerinde yığına atılan veriler tutulurken indis değişkeni kuyruğa eklenen son veriyi işaret eder.
- Kuyruktan alma/çıkartma işlemi her zaman için dizinin başı olan 0 indisli gözden yapılır.
- Kuyruk tasarımı için, genel olarak, üç değişik çözüm şekli vardır:
  - **Dizi Üzerinde Kaydırmalı (QueueAsArray) (Bir indis Değişkenli)**
  - **Dizi Üzerinde Çevrimsel (QueueAsCircularArray) (İki indis Değişkenli)**
  - **Bağlantılı Liste (QueueAsLinkedList) ile**

Kuyruk Tasarımı



Dizi Üzerinde Kaydırmalı Kuyruk

- *N* uzunluktaki bir dizi üzerinde kaydırmalı kuyruk yapısının davranışı şekilde gösterilmiştir;



Dizi Üzerinde Kaydırmalı Kuyruk

- **a)**'da kuyruğun boş hali,
- **b)**'de ise herhangi bir andaki tipik hali görülmektedir. Kuyruktan çıkartma işlemi,
- **c)**'de görüldüğü gibi dizinin ilk gözünden, yani K[0] elemanı üzerinden yapılır; eğer kuyrukta birden fazla veri varsa, geride olanlarda bir öne kaydırılır.
- Kuyruğa ekleme işlemi çıkartma işleminden daha az maliyetle yapılır; eklenecek veri doğrudan ekle/son adlı indis değişkenin gösterdiği bir sonraki göze yapılır.

Dizi Üzerinde Kaydırmalı Kuyruk Kaba Kodu

Dizi üzerinde kaydırmalı kuyruk-ekleme işlemi kaba-kodu

```
if(Kuyrukta yer yoksa)
        Kuyruk dolu mesajını yaz ve EKES değerini gönder;
else
        son'u bir sonraki göz için arttır;
        Veriyi K kuyruğunda son ile gösterilen göze ekle;
}
```

Dizi üzerinde kaydırmalı kuyruk-çıkartma işlemi kaba-kodu

```
if(Kuyrukta veri yoksa)
    Kuyruk boş mesajını yaz ve EKES değerini gönder;
else
     K kuyruğundaki K[0] verisi al;
     Kuyruktaki verileri öne kaydır, K[0]'ı K[1], K[1]'ı K[2]...K[son-2]'ı K[son-1].
    son'u bir önceki göz için azalt;
    Veriyi gönder;
}
```

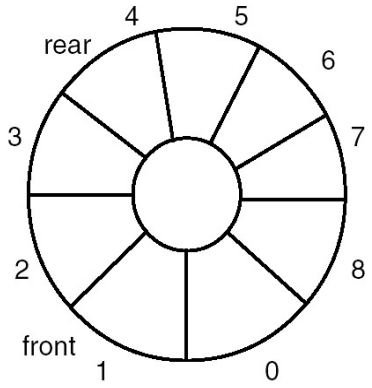
Dizi Üzerinde Çevrimsel Kuyruk

- Bu çözüm şeklinde dizi elemanlarına erişim doğrusal değil de çevrimsel yapılır; yani dizinin son elemanına ulaşıldığında bir sonraki göz dizinin ilk elemanı olacak şekilde indis değeri hesaplanır.
- Böylece kuyruk için tanımlanan dizi üzerinde sanki bir halka bellek varmış gibi dolaşılır.
- Kuyruktaki sıradaki eleman alındığında bu işaretçinin değeri bir sonraki gözü gösterecek biçimde arttırılır.
- Arttırma işleminin de, dizinin sonuna gelindiğinde başına gidecek biçimde çevrimsel olması sağlanır.
- Aşağıda dizi üzerinde çevrimsel kuyruk modeli için ekleme ve çıkartma işlemleri kaba-kodları verilmiştir:

Dizi Üzerinde Çevrimsel Kuyruk

Dizi üzerinde çevrimsel kuyruk - ekleme işlemi kaba-kodu

```
if(Kuyrukta yer yoksa)
        Kuyruk dolu mesajını yaz ve EKES değerini gönder;
else
        son'u çevrimsel şekilde bir sonraki göz için ayarla;
        Veriyi K kuyruğuna ekle;
}
```



- Yukarıda görüldüğü gibi ilk önce ekleme yapılması için kuyrukta yer olup olmadığına bakılmakta, yer varsa son adlı ekleme sayacı çevrimsel erişime imkan verecek şekilde arttırılmakta ve işaret ettiği yere veri koyulmaktadır.

Dizi Üzerinde Çevrimsel Kuyruk

- Aşağıda ise alma işlemi kaba kodu verilmiştir. Görüleceği gibi ilk önce kuyrukta veri olup olmadığı sınanmakta ve veri varsa ilk adlı işaretçinin gösterdiği gözdeki veri alınıyor, ilk adlı işaretçi çevrimsel olarak sıradaki veriyi gösterecek şekilde ayarlanıyor ve veri gönderiliyor.

Dizi üzerinde çevrimsel kuyruk - çıkartma işlemi kaba-kodu

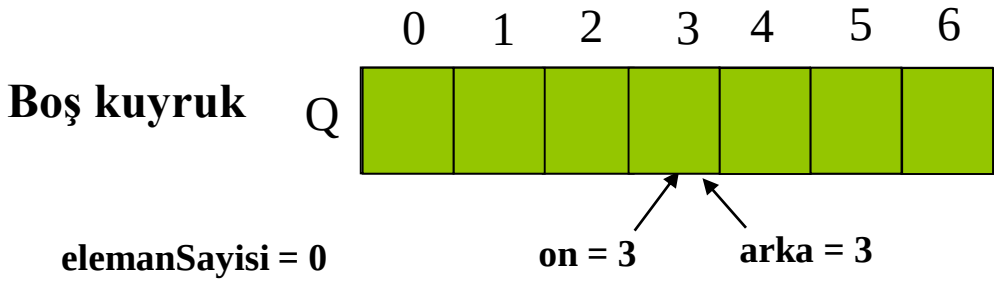
```
if(Kuyrukta yer yoksa)
        Kuyruk boş mesajını yaz ve EKES değerini gönder;
else
        K kuyruğundan ilk indisli yerdeki veriyi al;
        ilk'i çevrimsel şekilde bir sonraki göz için ayarla;
        Kuyruktan alınan veriyi gönder;
}
```

Dizi Kullanılarak Gerçekleştirim

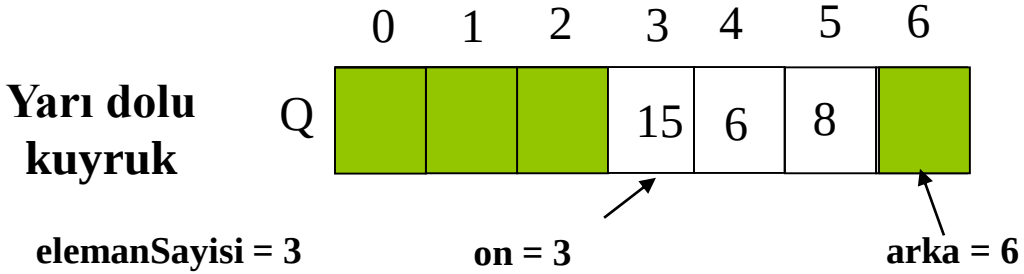
- N boyutlu bir dizi kullanılır.
- ön kuyruğun ilk elemanını tutar. Dizide ilk elemanın kaçıncı indisten başlayacağını belirtir.
- arka kuyrukta son elemandan sonraki ilk boşluğu tutar.
- elemanSayisi kuyruktaki eleman sayısını tutar.
- Boş kuyruk eleman sayısının sıfır olduğu durumdur.
- Dolu kuyruk eleman sayısının N'ye eşit olduğu durumdur.

Dizi Kullanarak Gerçekleştirim

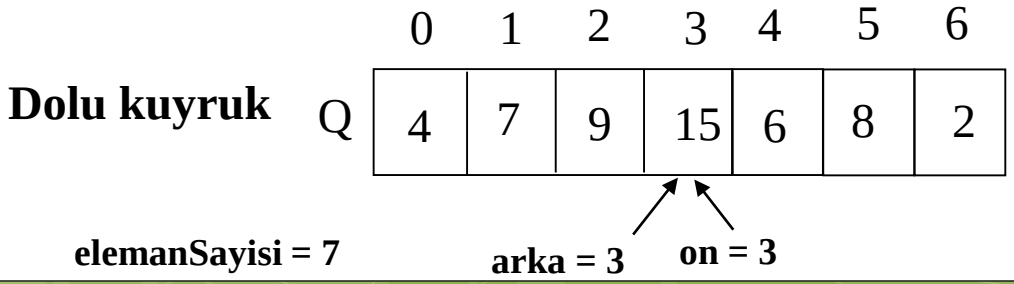
- Kuyruğu N boyutlu bir dizi (int K[N]) ve 3 değişken (int on, int arka, int elemanSayisi) ile gerçekleştirebiliriz.



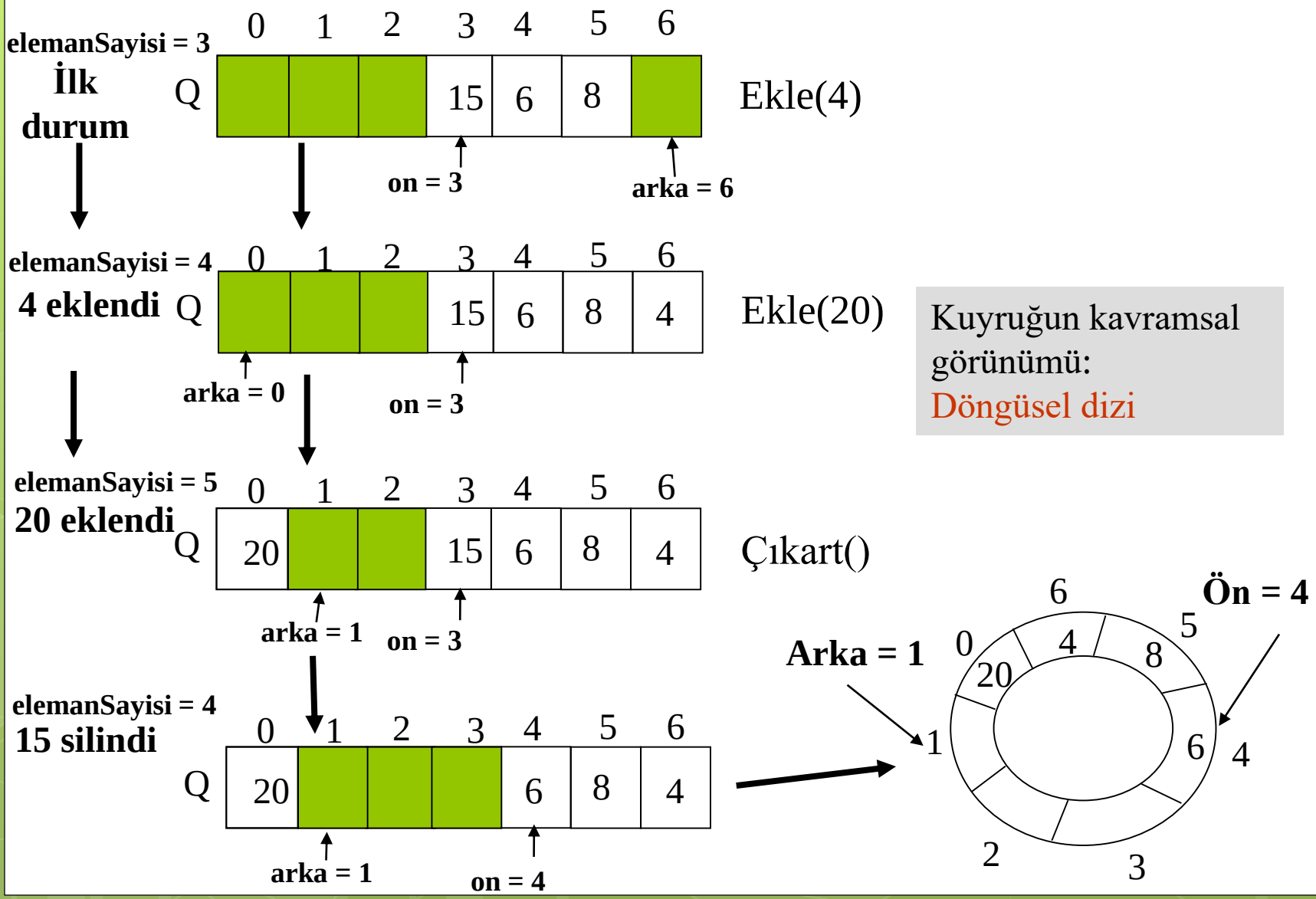
- on ve arka birbirlerine eşit ve elemanSayisi = 0 → Boş kuyruk



- on kuyruktaki ilk elemanı tutar
- arka son elemandan sonraki ilk boş yeri tutar.



- on ve arka birbirlerine eşit ve elemanSayisi=7 → Dolu kuyruk.



Kuyruğun kavramsal görünümü:
Döngüsel dizi

Dizi Üzerinde Çevrimsel Kuyruk- Java

```
public class kuyruk {

  private int K[N];     // kuyruk elemanlarını tutan dizi
  private int on;       // kuyruğun başı
  private int arka;     // kuyruğun sonu
  private int elemanSayisi; // kuyruktaki eleman sayısı


  public kuyruk();

  public boolean bosmu();
  public boolean dolumu();
  public int ekle(int item);
  public int cikart();
};
```

Dizi Üzerinde Çevrimsel Kuyruk- Java

```
// yapıcı yordam
public Kuyruk(){
   on = arka = elemanSayisi = 0;
}

// Kuyruk boşsa true döndür
public boolean bosmu(){
   return elemanSayisi == 0;
}

// Kuyruk doluysa ture döndür
public boolean dolumu(){
   return elemanSayisi == N;
}
```

Dizi Üzerinde Çevrimsel Kuyruk- Java

```
// Kuyruğa yeni bir eleman ekle
// Başarılı olursa 0 başarısız olursa -1 döndür
public int ekle(int yeni){
  if (dolumu()){
    System.out.println("Kuyruk dolu.");
    return -1;
  }

  K[arka] = yeni;   // Yeni elemanı sona koy
  arka++; if (arka == N) arka = 0;
  elemanSayisi++;

  return 0;
}
```

Dizi Üzerinde Çevrimsel Kuyruk- Java

```
// Kuyruğun önündeki elemanı çıkart ve döndür.
// Kuyruk boşsa -1 döndür
public int cikart(){
  int id = -1;

  if (bosmu()){
    System.out.println("Kuyruk boş");
    return -1;
  }

  id = on;   // ilk elemanın olduğu yer
  on++; if (on == N) on = 0;
  elemanSayisi--;

  return K[id];  // elemanı döndür
}
```

```
public static void main(String[] args){
  Kuyruk k;

  if (k.bosmu())
   System.out.println("Kuyruk boş");

  k.ekle(49);
  k.ekle(23);

  System.out.println("Kuyruğun önü: "+ k.cikart());
  k.ekle(44);
  k.ekle(22);

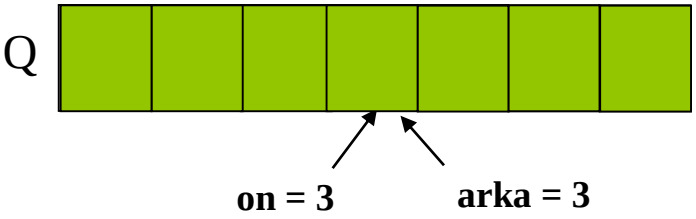
  System.out.println("Kuyruğun ilk elemanı: "+ k.cikart());
  System.out.println("Kuyruğun ilk elemanı: "+ k.cikart());
  System.out.println("Kuyruğun ilk elemanı: "+ k.cikart());
  System.out.println("Kuyruğun ilk elemanı: "+ k.cikart());

  if (k.bosmu())
    System.out.println("Kuyruk boş");
 }
```

Dizi Gerçekleştirimi: Son Söz

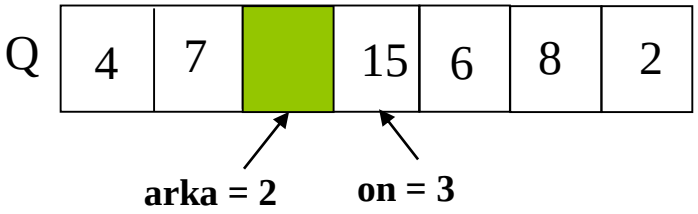
- Kuyruğu N-1 elemanlı bir dizi (int K[N]) ve 2 tane değişken (int on, int arka) ile gerçekleştirebiliriz.
- Aşağıdakileri nasıl tanımlayabileceğimizi düşünün.
  - Boş kuyruk
  - Dolu kuyruk

Boş kuyruk



- On ve arka birbirine eşit ise boş kuyruk

Dolu kuyruk



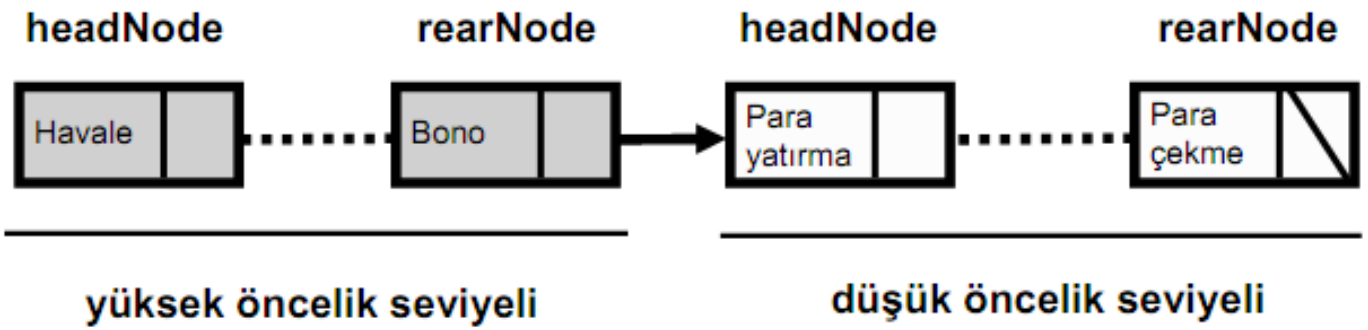
- On ve arka arasında 1 tane boşluk varsa dolu kuyruk

Öncelikli Kuyruklar (Priority Queues)

- Öncelikli kuyruk uygulamasında kuyruğa girecek verilerin veya nesnelerin birer öncelik değeri vardır ve ekleme bu öncelik değerine bakılarak yapılır.
- Eğer eklenecek verinin önceliği en küçük ise, yani en önceliksiz ise, doğrudan kuyruğun sonuna eklenir; diğer durumlarda kuyruk üzerinde arama yapılarak kendi önceliği içerisinde sona eklenir.
- Örneğin bazı çok prosesli işletim sistemlerinde bir proses işleme kuyruğu vardır ve yürütülme için hazır olan proseslerin listesi bir kuyrukta tutulur. Eğer proseslerin birbirlerine göre herhangi bir önceliği yoksa, prosesler kuyruğa ekleniş sırasına göre işlemci üzerinde yürütülürler; ilk gelen proses ilk yürütülür. Ancak, proseslerin birbirlerine göre bir önceliği varsa, yani aynı anda beklemekte olan proseslerden bazıları daha sonra kuyruğa eklenmiş olsalar dahi diğerlerine göre ivedi olarak yürütülmesi gerekebilir.
- Öncelikli kuyruk oluşturmak için bağlantılı listeyle, kümeleme ağacına dayalı ve araya sokma sıralaması yöntemiyle sağlanan çözümler olmaktadır.

Öncelikli Kuyruklar (Priority Queues)

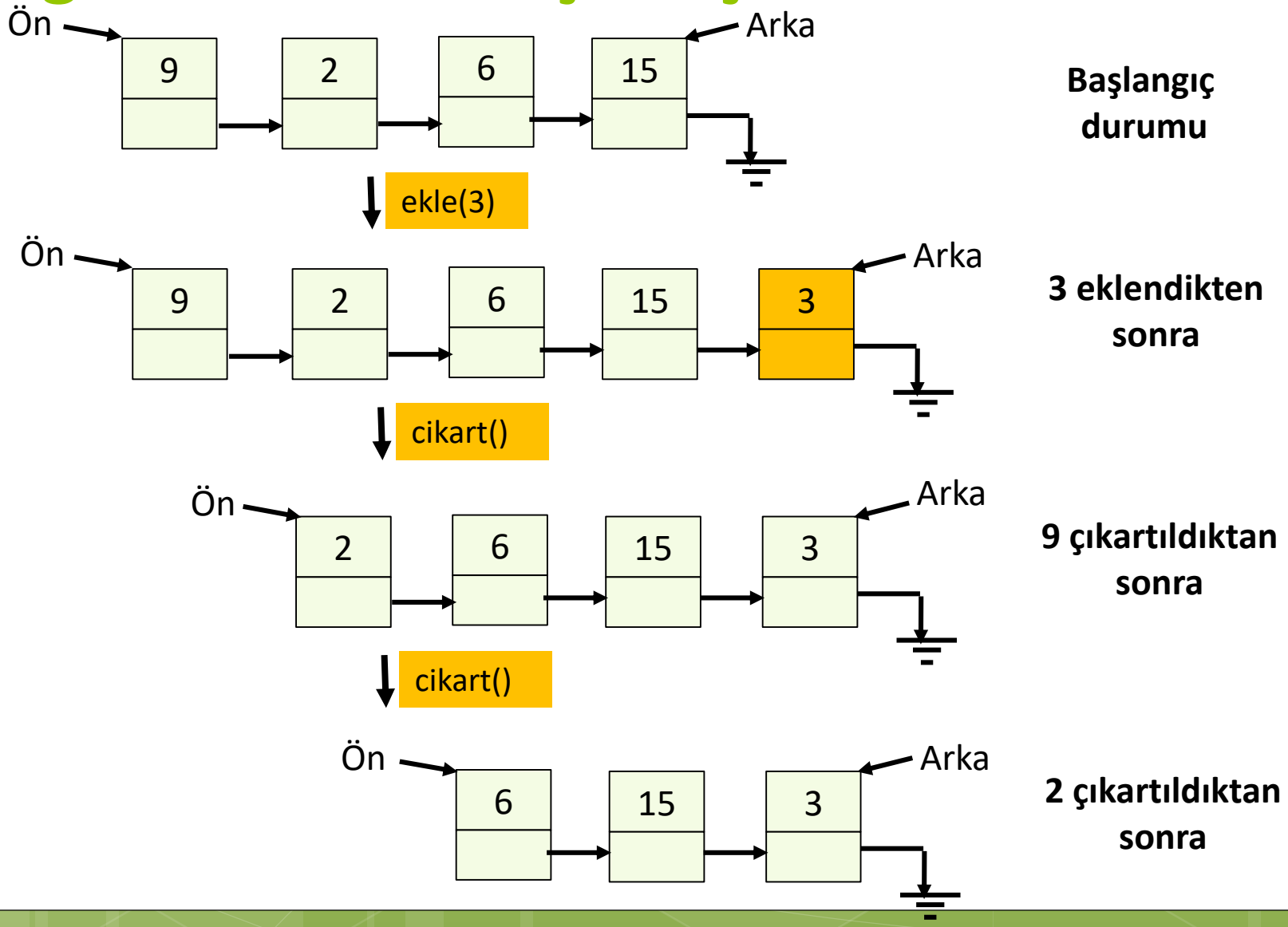
- Kuyruktaki işler kendi arasında önceliklerine göre seviyelendirilebilir.
- Her öncelik seviyesinin headNode ve rearNode'u vardır.
- Elaman alınırken en yüksek seviyeye ait elemanların ilk geleni öncelikli alınır.
- Yüksek öncelik seviyeli grubun son elemanı düşük öncelik seviyeli grubun ilk elemanından daha önceliklidir.



Bağlantılı Liste ile Kuyruk Tasarımı

- Ekleme (add) işlemi, son adresindeki verinin arkasına eklenir.
- Çıkarma (get) işlemi, ilk verinin alınması ve kuyruktan çıkarılması ile yapılır. bellekte yer olduğu sürece kuyruk uzayabilir.

Bağlantılı Liste Gerçekleştirimi



```
public class KuyrukDugumu {
  public int eleman;
  public KuyrukDugumu sonraki;

  public KuyrukDugumu(int e){
    eleman = e; sonraki = null;
  }
}

public class Kuyruk{
  private KuyrukDugumu on; // Kuyruğun önü
  private KuyrukDugumu arka;  // Kuyruğun arkası

  public Kuyruk(){
    on = arka = null;
  }
  public boolean bosmu() { ... }
  public void ekle(int eleman) { ... }
  public int cikart() { ... }
}
```

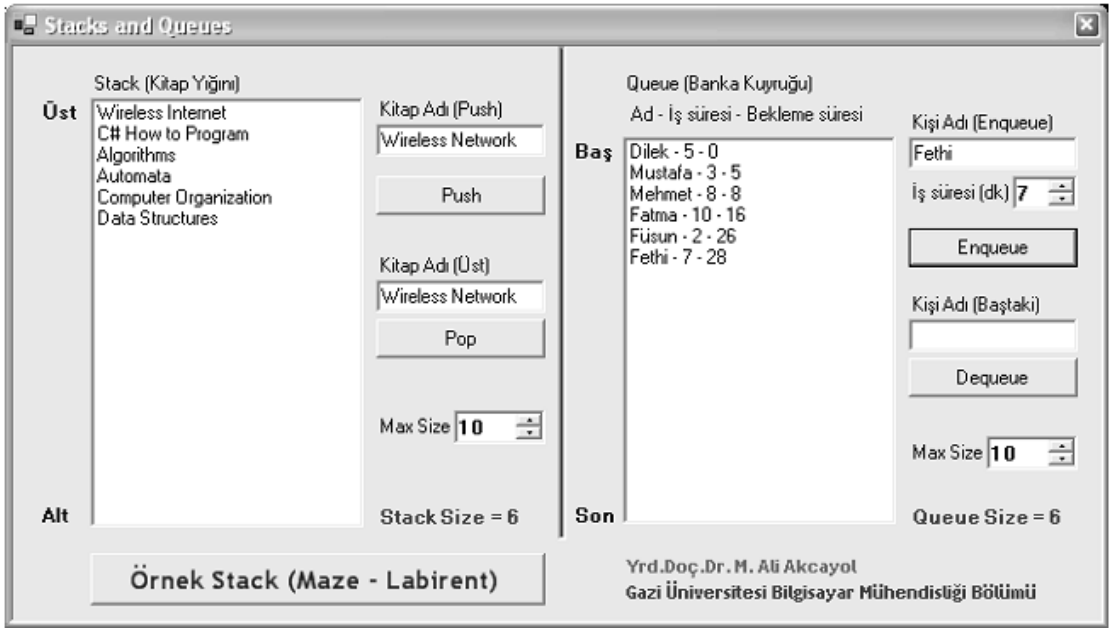
Haftalık Ödev

- 1- Banka kuyruğu örneğini ele alıp Banka işlemlerini kendi içerisinde üç öncelik grubuna ayırınız. Her yeni gelen kişiyi iş seçimine göre otomatik olarak ait olduğu grubun en sonuna ekleyiniz (enqueue). Her eleman alımında (dequeue) ise kuyruktaki en yüksek seviyeli grubun ilk elemanını alınız. Kuyruğu Dairesel bağlı listeler ile gerçekleştiriniz.

Haftalık Ödev

- 2- Aşağıda verilen uygulamayı C# veya Javada yapınız

Uygulama programı (Stacks ve Queues):



Örnek Uygulamalar

Örnek Uygulamalar

Java-Yığıt

Yığıt (stack) yapısının dinamik dizi ile gerçekleştirimi

- Stack.java
- public interface Stack<T>
- { public boolean empty(); // yığıt boş mu test eder
- public T top(); // tepedeki elemanı ver (silme)
- public T pop(); // tepedeki elemanı sil ve ver
- public void push(T item); // item'i yığıtın tepesine ekle
- public void clear(); // Yığıtı boşalt }

Yığıt (stack) yapısının dinamik dizi ile gerçekleştirimi

- ArrayStack.java
- public class ArrayStack<T> implements Stack<T>
- {
- private static final int DEFAULT_SIZE = 10;
- private T[] array; // yığıt elemanlarını tutan dizi
- private int top; // son eklenen elemanın indisi
- public ArrayStack() // kurucu sınıf
- { array = (T[]) new Object[DEFAULT_SIZE]; top = -1; }
- public boolean empty() // yığıt boş mu test eder
- { return (top==-1); }

Yığıt (stack) yapısının dinamik dizi ile gerçekleştirimi

- ArrayStack.java
- public T top() // tepedeki elemanı ver (silme)
- { if (empty()) return null;
- return array[top];
- }
- public T pop() // tepedeki elemanı sil ve ver
- { if (empty()) return null;
- return array[top--];
- }

Yığıt (stack) yapısının dinamik dizi ile gerçekleştirimi

- ArrayStack.java
- public void push(T item) // item'i yığıtın tepesine ekle
- { if (top+1==array.length)
- { T[] newArray = (T[]) new Object[array.length * 2];
- for (int i=0; i<array.length; i++)
- newArray[i] = array[i];
- array = newArray;
- }
- array[++top] = item;
- }
- public void clear() // Yığıtı boşalt
- { top = -1; }
- }

Yığıt (stack) yapısının dinamik dizi ile gerçekleştirimi

- TestArrayStack.java
- public class TestArrayStack
- {
- public static void main(String[] args)
- { Stack<String> yigit = new ArrayStack<String>();
- yigit.push("a"); yigit.push("b");
- System.out.println("Yigittaki son eleman: " + yigit.top());
- yigit.push("c");
- String s = yigit.pop();
- System.out.println("Yigittan silinen eleman: " + s);
- yigit.push("d");
- }
- }

Yığıt (stack) yapısının ArrayList ile gerçekleştirimi

- Stack.java // Daha önce verildi
- ArrayListStack.java
- import java.util.*;
- public class ArrayListStack<T> implements Stack<T>
- { private ArrayList<T> array; // yığıt elemanlarını tutan dizi
- public ArrayListStack() // kurucu sınıf
- { array = new ArrayList<T>(); }
- public boolean empty() // yığıt boş mu test eder
- { return array.size()==0; }
-
- public T top() // tepedeki elemanı ver (silme)
- { if (empty()) return null;
- return array.get(array.size()-1);
- }
-

Yığıt (stack) yapısının ArrayList ile gerçekleştirimi

- ArrayListStack.java
- public T pop() // tepedeki elemanı sil ve ver
- {
- if (empty()) return null;
- return array.remove(array.size()-1);
- }
- public void push(T item) // item'i yığıtın tepesine ekle
- { array.add(item); }
- public void clear() // Yığıtı boşalt
- { array.clear(); }
- }

Yığıt (stack) yapısının ArrayList ile gerçekleştirimi

- TestArrayListStack.java
- public class TestArrayListStack
- {
- public static void main(String[] args)
- {
- Stack<String> yigit = new ArrayStack<String>();
- yigit.push("a");
- yigit.push("b");
- System.out.println("Yigittaki son eleman: " + yigit.top());
- yigit.push("c");
- String s = yigit.pop();
- System.out.println("Yigittan silinen eleman: " + s);
- yigit.push("d");
- }
- }

Yığıt (stack) yapısının bağlantılı liste ile gerçekleştirimi

- Stack.java // Daha önce verildi
- LinkedListStack.java
- public class LinkedListStack<T> implements Stack<T>
- { private Node<T> top = null; // yığıtın tepesindeki eleman
- public boolean empty() // yığıt boş mu test eder
- { return top==null; }
- public T top() // tepedeki elemanı ver (silme)
- { if (empty()) return null;
- return top.data;
- }
- public T pop() // tepedeki elemanı sil ve ver
- { if (empty()) return null;
- T temp = top.data;
- top = top.next;
- return temp;
- }

Yığıt (stack) yapısının bağlantılı liste ile gerçekleştirimi

- LinkedListStack.java
- public void push(T item) // item'i yığıtın tepesine ekle
- { Node<T> newNode = new Node<T>();
- newNode.data = item;
- newNode.next = top;
- top = newNode;
- }
- public void clear() // Yığıtı boşalt
- { top = null; }
- // Yığıt elemanlarını tutan liste elemanı (inner class)
- class Node<T>
- { public T data;
- public Node<T> next;
- }
- }

Yığıt (stack) yapısının bağlantılı liste ile gerçekleştirimi

- TestLinkedListStack.java
- public class TestLinkedListStack
- { public static void main(String[] args)
- { Stack<Character> yigit = new LinkedListStack<Character>();
- yigit.push('a');
- yigit.push('b');
- yigit.push('c');
- System.out.println("Yigittaki elemanlar cikariliyor: ");
- while (!yigit.empty())
- {
- System.out.println(yigit.pop());
- }
- }
- }

Hanoi Kuleleri– Özyinelemeli Çözüm- C#

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Hanoi
{  class Program
   {   static void Main(string[] args)
       {   int x;    char from='A', to='B', help='C';
       do {
            try
            { Console.Write(" input number of disk: "); x = Int32.Parse(Console.ReadLine()); }
            catch (FormatException e) { x = -10; }
           } while(x==-10 || x>10);
```

Hanoi Kuleleri– Özyinelemeli Çözüm- C#

```
Console.WriteLine("\n from = A, to = B, help = C\n");
hanoi(x, from, to, help);     Console.Read();
}
static void hanoi(int x, char from, char to, char help)
{    if (x > 0)
     {   hanoi(x - 1, from, help, to);
         move(x, from, to);
         hanoi(x - 1, help, to, from);
      }
}
static void move(int x, char from, char to)
{ Console.WriteLine(" move disk "+x+" from "+from+" to "+to); }
}
}
```

Örnek Uygulamalar

JAVA-Kuyruk

Kuyruk Tasarımı ve Kullanımı -Java

```
class Kuyruk
{  private int boyut;   private int[ ] kuyrukDizi;
  private int bas;      private int son;        private int elemanSayisi;

public Kuyruk(int s)
  { boyut = s;          kuyrukDizi = new int[boyut];
    bas = 0;            son = -1;                 elemanSayisi = 0;
  }
  public void ekle(int j) // Kuyrugun sonuna eleman ekler
  {
    if (son==boyut-1) son = -1;
    kuyrukDizi[++son] = j;    elemanSayisi++;
  }
```

Kuyruk Tasarımı ve Kullanımı -Java

```
public int cikar()
 {
  int temp = kuyrukDizi[bas++];
  if(bas==boyut) bas=0;
  elemanSayisi--;      return temp;
 }
  public boolean bosMu()  {   return(elemanSayisi==0);  }
}

public class KuyrukTest  {
 public static void main(String args[])
 {   Kuyruk k = new Kuyruk(25);   k.ekle(1);   k.ekle(2);
     System.out.println(k.cikar());  k.ekle(3);
  for(int i=4; i<10; ++i)      k.ekle(i);
  while(!k.bosMu())      System.out.println(k.cikar());
 }
}
```

Kuyruk (queue) yapısının dinamik dizi ile gerçekleştirimi

- Queue.java
- public interface Queue<T>
- {
- public boolean empty(); // kuyruk boş mu test eder
- public T getFront(); // kuyruğun önündeki elemanı ver (silme)
- public T dequeue(); // kuyruğun önündeki elemanı sil ve ver
- public void enqueue(T item); // item'i kuyruğun arkasına ekle
- public void clear(); // kuyruğu boşalt
- }

Kuyruk (queue) yapısının dinamik dizi ile gerçekleştirimi

- ArrayQueue.java
- public class ArrayQueue<T> implements Queue<T>
- { private static final int DEFAULT_SIZE = 10;
- private T[] array; // kuyruk elemanlarını tutan dizi
- private int size; // kuyruktaki eleman sayısı
- private int front; // en önce eklenen elemanın indisi
- private int back; // en son eklenen elemanın indisi

- public ArrayQueue() // kurucu sınıf
- { array = (T[]) new Object[DEFAULT_SIZE]; clear(); }

- public boolean empty() // kuyruk boş mu test eder
- { return size==0; }

Kuyruk (queue) yapısının dinamik dizi ile gerçekleştirimi

- ArrayQueue.java
- public T getFront() // kuyruğun önündeki elemanı ver (silme)
- {
- if (empty()) return null;
- return array[front];
- }
-
- public T dequeue() // kuyruğun önündeki elemanı sil ve ver
- {
- if (empty()) return null;
- size--;
- T temp = array[front];
- front = increment(front);
- return temp;
- }

Kuyruk (queue) yapısının dinamik dizi ile gerçekleştirimi

- ArrayQueue.java
- public void enqueue(T item) // item'i kuyruğun arkasına ekle
- {
- if (size==array.length) // kuyruk dolu
- {
- T[] newArray = (T[]) new Object[array.length * 2];
- for (int i=0; i<size; i++, front=increment(front))
- newArray[i] = array[front];
- array = newArray;
- }
- back = increment(back);
- array[back] = item;
- size++;
- }
-

Kuyruk (queue) yapısının dinamik dizi ile gerçekleştirimi

- ArrayQueue.java
- private int increment(int i)
- {
- i++;
- if (i==array.length) i=0;
- return i;
- }
- public void clear() // kuyrukı boşalt
- {
- size = 0;
- front = 0;
- back = -1;
- }
- }
-

Kuyruk (queue) yapısının dinamik dizi ile gerçekleştirimi

- TestArrayQueue.java
- public class TestArrayQueue
- { public static void main(String[] args)
- { Queue<Integer> kuyruk = new ArrayQueue<Integer>(); int say=1;
- for (int i=1; i<=15; i++)
- kuyruk.enqueue(i);
- for (int i=1; i<=10; i++)
- System.out.println((say++) + ".kuyruk elemani: " + kuyruk.dequeue());
- for (int i=11; i<=25; i++)
- kuyruk.enqueue(i);
- say=1;
- while (!kuyruk.empty())
- System.out.println((say++) + ".kuyruk elemani: " + kuyruk.dequeue());
- }
- }

Kuyruk (queue) yapısının ArrayList ile gerçekleştirimi

- Queue.java // daha önce verilmişti
- ArrayListQueue.java
- import java.util.ArrayList;
- public class ArrayListQueue<T> implements Queue<T>
- { private ArrayList<T> array; // kuyruk elemanlarını tutan dizi
- public ArrayListQueue() // kurucu sınıf
- { array = new ArrayList<T>(); }

- public boolean empty() // kuyruk boş mu test eder
- { return array.size()==0; }
- public T getFront() // kuyruğun önündeki elemanı ver (silme)
- { if (empty()) return null;
- return array.get(0);
- }

Kuyruk (queue) yapısının ArrayList ile gerçekleştirimi

- ArrayListQueue.java
- public T dequeue() // kuyruğun önündeki elemanı sil ve ver
- {
- if (empty()) return null;
- return array.remove(0);
- }
- public void enqueue(T item) // item'i kuyruğun arkasına ekle
- {
- array.add(item);
- }
- public void clear() // kuyrukı boşalt
- {
- array.clear();
- }
- }

Kuyruk (queue) yapısının ArrayList ile gerçekleştirimi

- TestArrayListQueue.java
- public class TestArrayListQueue
- {
- public static void main(String[] args)
- {
- Queue<Integer> kuyruk = new ArrayListQueue<Integer>();
- for (int i=1; i<=25; i++)
- kuyruk.enqueue(i);
- int say=1;
- while (!kuyruk.empty())
- System.out.println((say++) + ".kuyruk elemani: " + kuyruk.dequeue());
- }
- }

Kuyruk (queue) yapısının bağlantılı liste ile gerçekleştirimi

- Queue.java // daha önce verilmişti
- LinkedListQueue.java
- public class LinkedListQueue<T> implements Queue<T>
- {
- private Node<T> front, back; // kuyruk başı ve sonu
- public LinkedListQueue() // kurucu sınıf
- { clear(); }
- public boolean empty() // kuyruk boş mu test eder
- { return front==null; }
- public T getFront() // kuyruğun önündeki elemanı ver (silme)
- { if (empty()) return null;
- return front.data;
- }

Kuyruk (queue) yapısının bağlantılı liste ile gerçekleştirimi

```
public T dequeue()     // kuyruğun önündeki elemanı sil ve ver
{   if (empty()) return null;
    T temp = front.data;          front = front.next;       return temp;
}
    public void enqueue(T item) // item'i kuyruğun arkasına ekle
{   if (empty())       front = back = new Node<T>(item, null);
    else               back = back.next = new Node<T>(item,null);
}
    public void clear() {       front = back = null;    }       // kuyruğu boşalt
// Kuyruk elemanlarını tutan liste elemanı (inner class)
   private class Node<T>
   {    private T data;       private Node<T> next;
        public Node(T data, Node next)    {   this.data = data;          this.next = next;   }
    }
}
```

Kuyruk (queue) yapısının bağlantılı liste ile gerçekleştirimi

TestLinkedListQueue.java

```
public class TestLinkedListQueue
{   public static void main(String[] args)
    {   Queue<Integer> kuyruk = new ArrayListQueue<Integer>();
        for (int j=1, i=1; i<=20; i++)
        {   if ((int)(Math.random()*2) == 1)
            {   System.out.println("Kuyruga ekle: " + j);      kuyruk.enqueue(j++);       }
            else
            {  System.out.println("Kuyrugun onundeki "
                                        + kuyruk.dequeue() + " cikarildi..");
            }
        }
    }
}
```

Bağlı Liste ile Kuyruk gerçekleştirimi C#

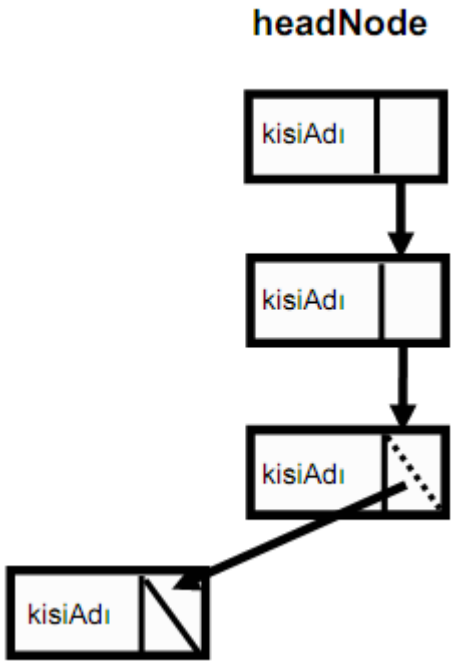
```
// Queue oluşturma :
class queueNodeC {
public string kisiAdi;                public queueNodeC sonraki;
public queueNodeC(string kisiAdi) {          this.kisiAdi = kisiAdi;      }
}
class queueC
{
public int size;                      public queueNodeC headNode;
public queueC(string kisiAdi)
{
this.headNode = new queueNodeC(kisiAdi);
this.headNode.sonraki = headNode;          this.size = 0;
}
}
queueC kisiKuyruk = new queueC("");
```

Bağlı Liste ile Kuyruk gerçekleştirimi C#

- **Örnek (C#):**

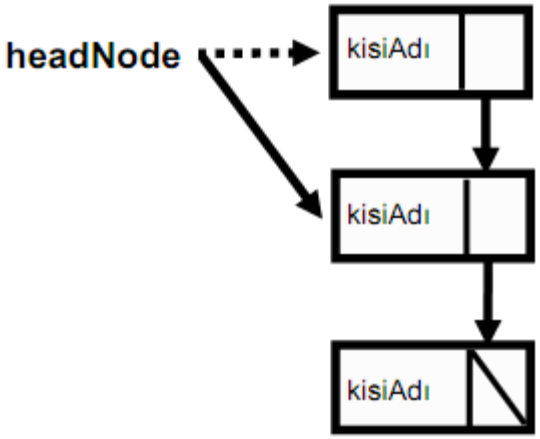
```
/* Queue işlemleri :
boş kuyruk size == 0
eleman sayısı= size
eleman ekleme= enqueue(kisiAdi)
eleman alma= dequeue() */
public void enqueue(kisiAdi)
{
queueNodeC yeniNode = new queueNodeC(kisiAdi);
queueNodeC aktif = kisiKuyruk.headNode;
while (aktif.sonraki != aktif)
{           aktif = aktif.sonraki;          }

aktif.sonraki = yeniNode;
yeniNode.sonraki = yeniNode;
kisiKuyruk.size++;
}
```



Bağlı Liste ile Kuyruk gerçekleştirimi C#

- // Queue işlemleri (eleman alma)
- public void dequeue()
- {
- queueNodeC yeniNode = new queueNodeC(“ ”);
- yeniNode = kisiKuyruk.headNode;
- kisiKuyruk.headNode = kisiKuyruk.headNode.sonı
- kisiKuyruk.size--;
- }



Kuyruk oluşturma,ekleme ve silme C++

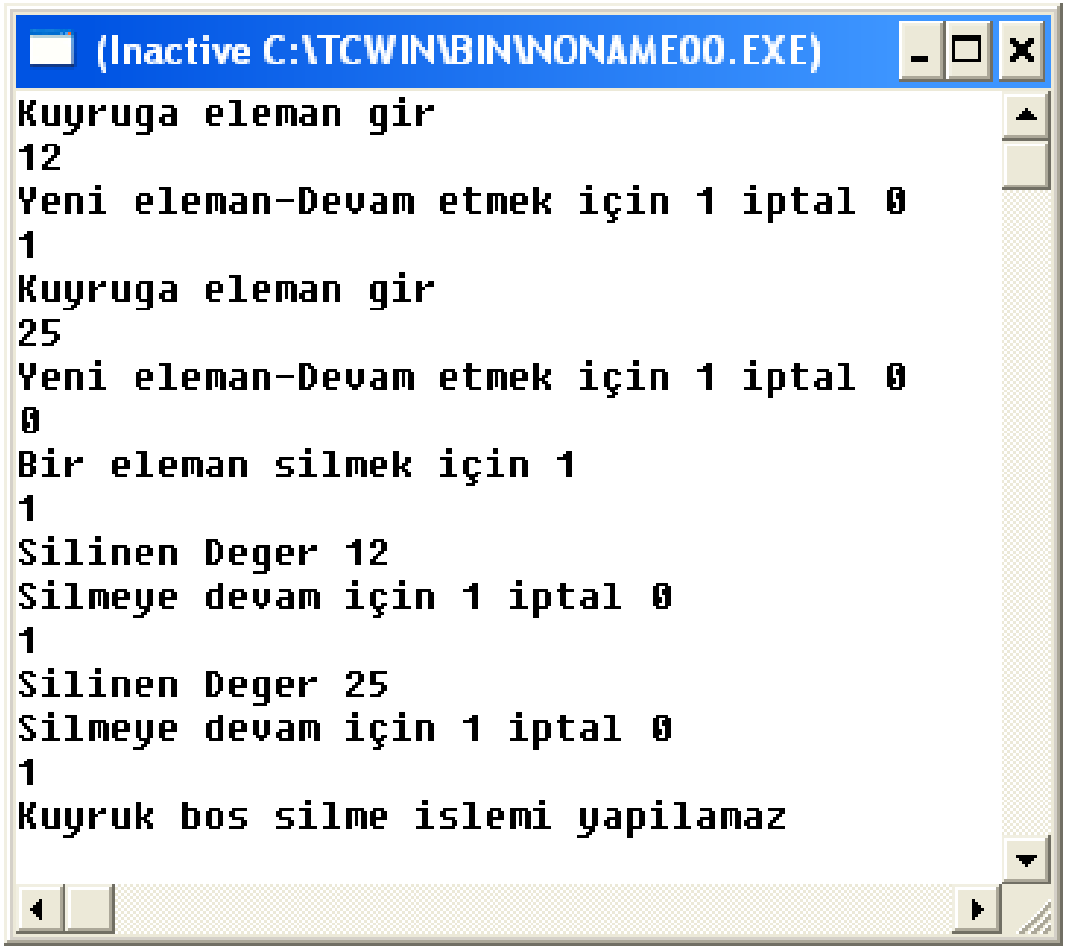
```
#include <stdio.h>
#include <stdlib.h>
#define boyut 10 /*Maksimum kuyruk uzunluğu */

void ekle (int kuyruk[], int *arka, int deger)
{
  if(*arka < boyut-1)   {
    *arka= *arka +1;
    kuyruk[*arka] = deger;   }
  else
     { printf("Kuyruk doldu daha fazla ilave edilemez\n");    }
 }

void sil (int kuyruk[], int * arka, int * deger)
{  int i;
  if(*arka<0)   {printf("Kuyruk bos silme islemi yapilamaz\n");}
  *deger = kuyruk[0];
  for(i=1;i<=*arka;i++)
  {    kuyruk[i-1]=kuyruk[i];    }
  *arka=*arka-1;
}
```

Kuyruk oluşturma,ekleme ve silme C++

```
main() {   int kuyruk[boyut];   int arka, n, deger;   arka=(-1);
do  {
 do {  printf("Kuyruga eleman gir\n");     scanf("%d",&deger);
        ekle(kuyruk,&arka,deger); //arka değeri arttı
        printf("Yeni eleman-Devam etmek için 1 iptal 0 \n");  scanf("%d",&n);
         }  while(n == 1);
    printf("Bir eleman silmek için 1\n");  scanf("%d",&n);
   while( n == 1) {
    sil(kuyruk,&arka,&deger);
    printf("Silinen Deger %d\n",deger); printf("Silmeye devam için 1 iptal 0\n");
    scanf("%d",&n);     }
    printf("Eleman girmek icin 1 iptal icin 0\n");      scanf("%d",&n);
 } while(n == 1);  }
```


(Inactive C:\TCWIN\BIN\NONAME00.EXE)
Kuyruga eleman gir
12
Yeni eleman-Devam etmek için 1 iptal 0
1
Kuyruga eleman gir
25
Yeni eleman-Devam etmek için 1 iptal 0
0
Bir eleman silmek için 1
1
Silinen Deger 12
Silmeye devam için 1 iptal 0
1
Silinen Deger 25
Silmeye devam için 1 iptal 0
1
Kuyruk bos silme islemi yapilamaz

Dizi Üzerinde Çevrimsel Kuyruk- C++

```
#include <stdio.h>
#define N 500 /*Maksimum kuyruk uzunluğu */
#include <stdlib.h>
typedef struct kuyrukyapisi
{   int bas;  int son ; int sayac; int D[N]; }  KUYRUK ;

KUYRUK *M;
void baslangic (KUYRUK *K)
{    K->bas=0;
     K->sayac=0;
     K->son=0;
}
```

Dizi Üzerinde Çevrimsel Kuyruk- C++

```
void ekle(int veri, KUYRUK *K)
{
  if(K->sayac>N-1)  {  printf ("Kuyruk dolu");  }
  /*Doğrusal erişimli bir diziye çevrimsel erişim yapılabilmesi için dizi indisi son gözden  sonra ilk gözü işaret edebilmesi için artık bölme işlemiyle indis=(indis+1)%N yapılır. */
  K->son=(K->son+1)%N;
  K->D[K->son]=veri;
  K->sayac++;
}
void silme(KUYRUK *K,int *deger)
{
  if((K->sayac)<=0)  {printf("Kuyruk bos silme islemi yapilamaz\n");}
  *deger = K->D[K->bas]=0;
   K->bas=(K->bas+1)%N;
   K->sayac--;
}
```

Dizi Üzerinde Çevrimsel Kuyruk- C++

```
main() {
int veri,n,deger;
M=(KUYRUK *)malloc(sizeof(KUYRUK));
 baslangic (M);
   do   {
     do {  printf("Kuyruga eleman gir\n");     scanf("%d",&veri);  ekle (veri,M);
           printf("Yeni eleman-Devam etmek için 1 iptal 0 \n");  scanf("%d",&n);
        }  while(n == 1);
    printf("Bir eleman silmek için 1\n");  scanf("%d",&n);
    while( n == 1) {  silme (M,&deger);
        printf("Silinen Deger %d\n",deger); printf("Silmeye devam için 1 iptal 0\n");
        scanf("%d",&n);      }
     printf("Eleman girmek icin 1 iptal icin 0\n");       scanf("%d",&n);
  } while(n == 1);
}
```

ÖZYİNELEME (RECURSION)

ÖZYİNELEME(RECURSION)

- Özyineleme, bir işlemin sonucu kendine önceki bir veya iki işlem sonucuna bağlı olarak bulunmasıdır.
- Kendi kendisini doğrudan veya dolaylı olarak çağıran fonksiyonlara özyineli (recursive) fonksiyonlar adı verilir.
- Özyineleme bir problemi benzer şekilde olan daha basit alt problemlere bölünerek çözülmesini sağlayan bir tekniktir.
- Alt problemler de kendi içlerinde başka alt problemlere bölünebilir.
- Alt problemler çözülebilecek kadar küçülünce bölme işlemi durur.
- Özyineleme, döngülere (iteration) alternatif olarak kullanılabilir.

ÖZYİNELEME

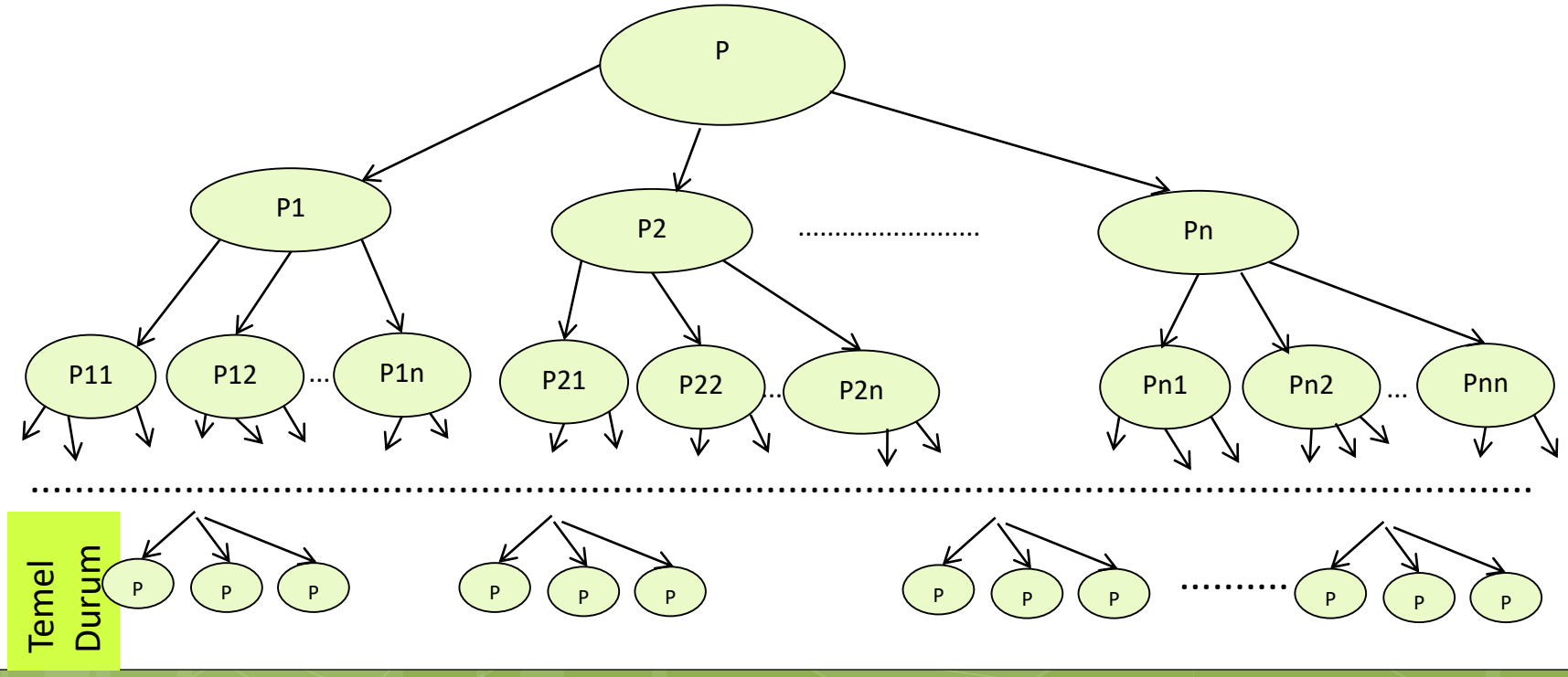
- Bir problem özyineli olmayan basit bir çözüme sahiptir. Problemin diğer durumları özyineleme ile durdurma durumuna (stopping case) indirgenebilir.

- Özyineleme işlemi durdurma durumu sağlanınca sonlandırılır.
- Özyineleme güçlü bir problem çözme mekanizmasıdır.
  - Çoğu algoritma kolayca özyinelemeli şekilde çözülebilir.
  - Fakat sonsuz döngü yapmamaya dikkat edilmeli.

- **Genel yazımı-kaba kod:**
  - *if (durdurma durumu sağlandıysa)*
  - *çözümü yap*
  - *else*
  - *problemi özyineleme kullanarak indirge*

Özyineleme- Böl & Yönet Stratejisi

- Bilgisayar birimlerinde önemli bir yere sahiptir:
  - Problemi küçük parçalara böl
  - Her bir parçayı bağımsız şekilde çöz
  - Parçaları birleştirerek ana problemin çözümüne ulaş



Özyineleme- Böl & Yönet Stratejisi

```
/* P problemini çöz */
Solve(P){
    /* Temel durum(s) */
    if P problemi temel durumda ise
        return çözüm

 /* (n>=2) için P yi P1, P2, ..Pn şeklinde parçalara böl */
 /* Problemleri özyinelemeli şekilde çöz */
    S1 = Solve(P1); /* S1 için P1 problemini çöz */
    S2 = Solve(P2); /* S2 için P2 problemini çöz*/
     …
    Sn = Solve(Pn); /* Sn için Pn problemini çöz */

     /* Çözüm için parçaları birleştir. */
    S = Merge(S1, S2, …, Sn);

     /* Çözümü geri döndür */
     return S;
} //bitti-Solve
```

ÖZYİNELEME

- Faktöryel fonksiyonu: Klasik bir özyineleme örneğidir:
  - **n! = 1 * 2 * 3 * … * (n – 1) * n**
- Faktoriyel işlemini özyineli tanımlamak için küçük sayıların faktöriyeli şeklinde tanımlamak gerekir.
  - **n! = n * (n – 1) !**
  - Durdurma durumu 0 ! = 1 olarak alınır.
- Her çağırmada n değeri bir azaltılarak durdurma durumuna ulaşılır.
- Özyineli tanımlama:

$$f(n) = \begin{cases} n = 1, & \text{if } n = 0 \\ n * f(n-1), & \text{if } n > 0 \end{cases}$$

ÖZYİNELEME

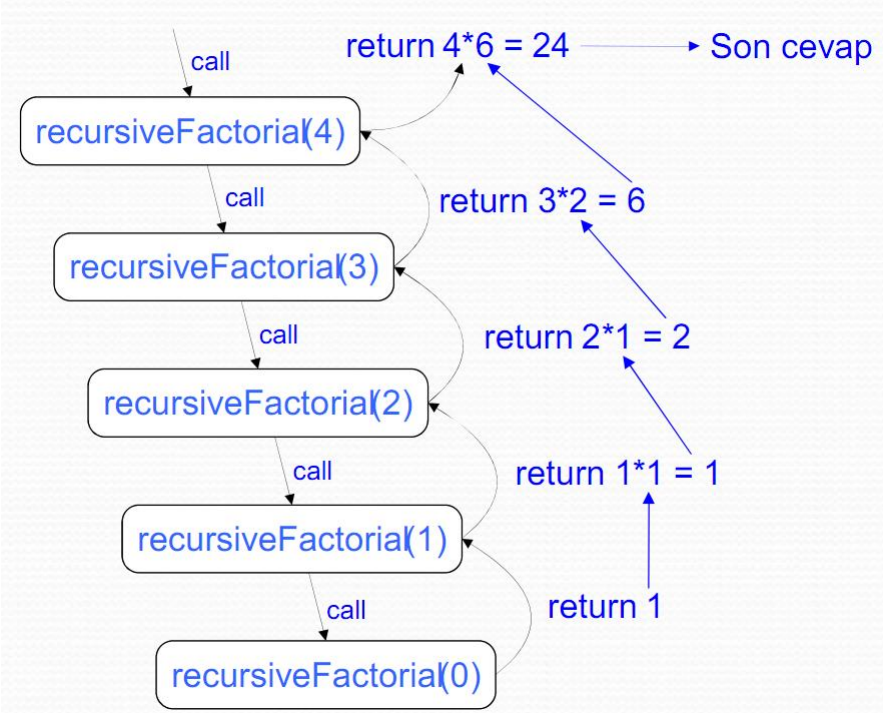
- Aşağıdaki kod çalıştığında n sayısının faktöriyel değerini hesaplar.

```
int recursiveFactorial(int n)
{
    if ( n == 1 )  return( 1 );
    else  return ( n * recursiveFactorial( n - 1 ));
}
```

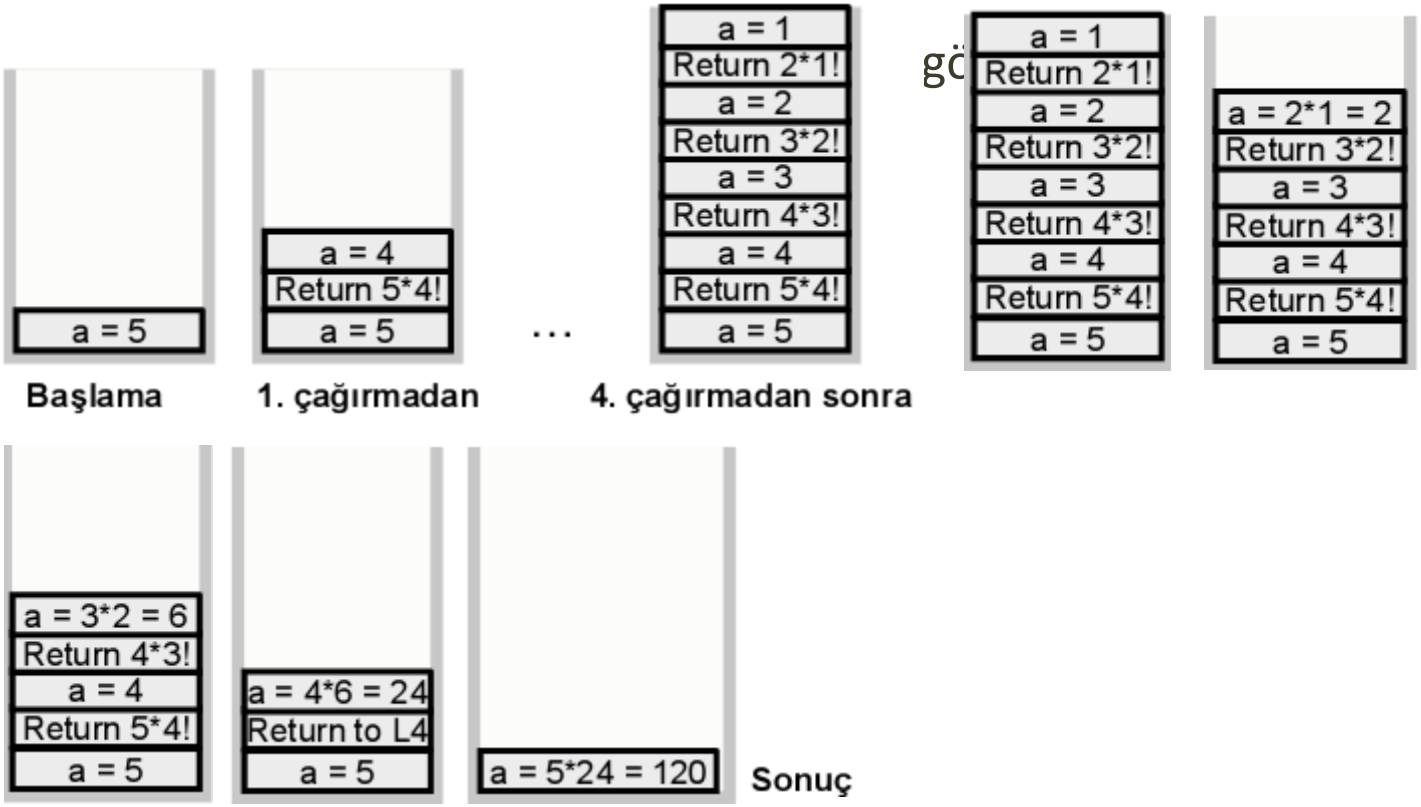
- n! değerini hesaplar ve bulduğu değeri return ile gönderir.

ÖZYİNELEME

- Özyineleme izleme
- Her özyineleme çağrımı için bir kutu
- Her çağırandan çağrılana bir ok
- Her çağrılandan çağırana çizilen ok geri dönüş değerini gösterir.



ÖZYİNELEME


a = 5
a = 4
Return 5*4!
a = 5
...
a = 1
Return 2*1!
a = 2
Return 3*2!
a = 3
Return 4*3!
a = 4
Return 5*4!
a = 5
gö
a = 1
Return 2*1!
a = 2
Return 3*2!
a = 3
Return 4*3!
a = 4
Return 5*4!
a = 5
a = 2*1 = 2
Return 3*2!
a = 3
Return 4*3!
a = 4
Return 5*4!
a = 5
Başlama
1. çağırmadan
4. çağırmadan sonra
a = 3*2 = 6
Return 4*3!
a = 4
Return 5*4!
a = 5
a = 4*6 = 24
Return to L4
a = 5
a = 5*24 = 120
Sonuç

ÖZYİNELEME

- Genellikle iterative fonksiyonlar zaman ve yer bakımından daha etkindirler.
- Iterative algoritma döngü yapısını kullanır.
- Özyineleme algoritması dallanma (branching) algoritmasını kullanır.
- Fonksiyon özyineli olarak her çağrılışında yerel değişkenler ve parametreler için bellekte yer ayrılır.
- Özyineleme problemin çözümünü basitleştirebilir, sonuç genellikle kısadır ve kod kolayca anlaşılabilir.
- Her özyinelemeli olarak tanımlanmış problemin iterative çözümüne geçiş yapılabilir.

ÖZYİNELEME

- **Recursive**
- int recFact(int n)
- {
- if(n ==1) return(1);
- else
- return(n * recFact(n - 1));
- }

- **Iterative**
- int iteFact(int n)
- {
- int araDeger = 1;
- for (int i = n; i > 0; i- -)
- araDeger * = i;
- return araDeger;
- }

FaktoriyelOrnek.java

```
import java.io.*;
 class FaktoryelOrnek
{   static int sayi;
   public static void main(String args[]) throws IOException
  {
    System.out.print("Sayi veriniz :");     System.out.flush();
    sayi=getInt();     int sonuc = factorial(sayi);
    System.out.println(sayi+"! ="+sonuc);
  }
   public static int factorial(int n)
  {
    if(n==0)       return 1;
    else       return(n*factorial(n-1));
  }
```

```
public static String getString() throws IOException
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
  }
   public static int getInt() throws IOException
  { String s = getString();
    return Integer.parseInt(s);
  }
 }
```

N’ye Kadar Olan Sayıların Toplamı

- Problemimizin 1’den n’ye kadar sayıların toplamı olduğunu varsayalım.

- Bu problemi özyinelemeli nasıl düşüneceğiz:
  - Topla(n) = 1+2+..+n ifadesini hesaplamak için
    - Topla(n-1) = 1+2+..+n-1 ifadesini hesapla (aynı türden daha küçük bir problem)
    - Topla(n-1) ifadesine n ekleyerek Topla(n) ifadesi hesaplanır.
    - Topla(n) = Topla(n-1) + n;
  - Temel durumu belirlememiz gerekiyor.
    - Temel durum, (alt problem) problemi bölmeye gerek kalmadan kolayca çözülebilen problemdir.
    - n = 1 ise, Topla(1) = 1;

Topla(4) için Özyineleme Ağacı

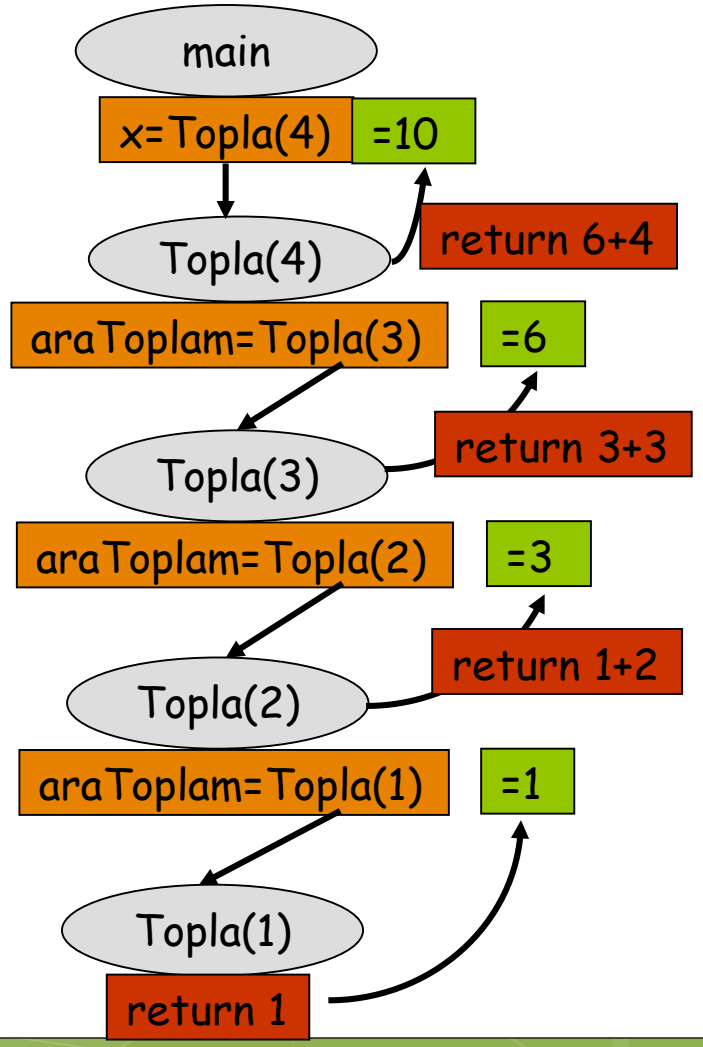
```
/* Topla 1+2+3+…+n */
int Topla(int n){
  int araToplam = 0;

  /* Temel Durum */
  if (n == 1) return 1;

  /* Böl ve Yönet */
  araToplam = Topla(n-1);

  /* Birleştir */
  return araToplam + n;
} /* bitti-Topla */

Public ... main(...){
print("Topla: "+ Topla(4));
} /* bitti-main */
```



Topla(n)'nin çalışma zamanı

```
/* Topla 1+2+3+…+n */
int Topla(int n){
  int araToplam = 0;

  /* Temel durum */
  if (n == 1) return 1;

  /* Böl ve yönet */
  araToplam = Topla(n-1);

  /* Birleştir */
  return araToplam + n;
} /* bitti-araToplam */
```

$$T(n) = \begin{cases} n = 1 \rightarrow 1 \text{ (Temel durum)} \\ n > 1 \rightarrow T(n-1) + 1 \end{cases}$$

a^n İfadesini Hesaplama-Pow(a,n)

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- Böl, yönet & birleştir işlemleri bir ifade ile yapılabilir.

```
/* a^n hesapla */
double Ust(double a, int n){
  double araSonuc;

  /* Temel durum */
  if (n == 0) return 1;
  else if (n == 1) return a;

  /* araSonuc = a^(n-1) */
 araSonuc = Ust(a, n-1);

  /* Birleştir */
  return araSonuc*a;
} /* bitti-Ust */
```

```
/* Hesapla a^n */
double Ust(double a, int n){
  /* Temel durum */
  if (n == 0) return 1;
  else if (n == 1) return a;

  return Ust(a, n-1)*a;
} /* bitti-Ust */
```

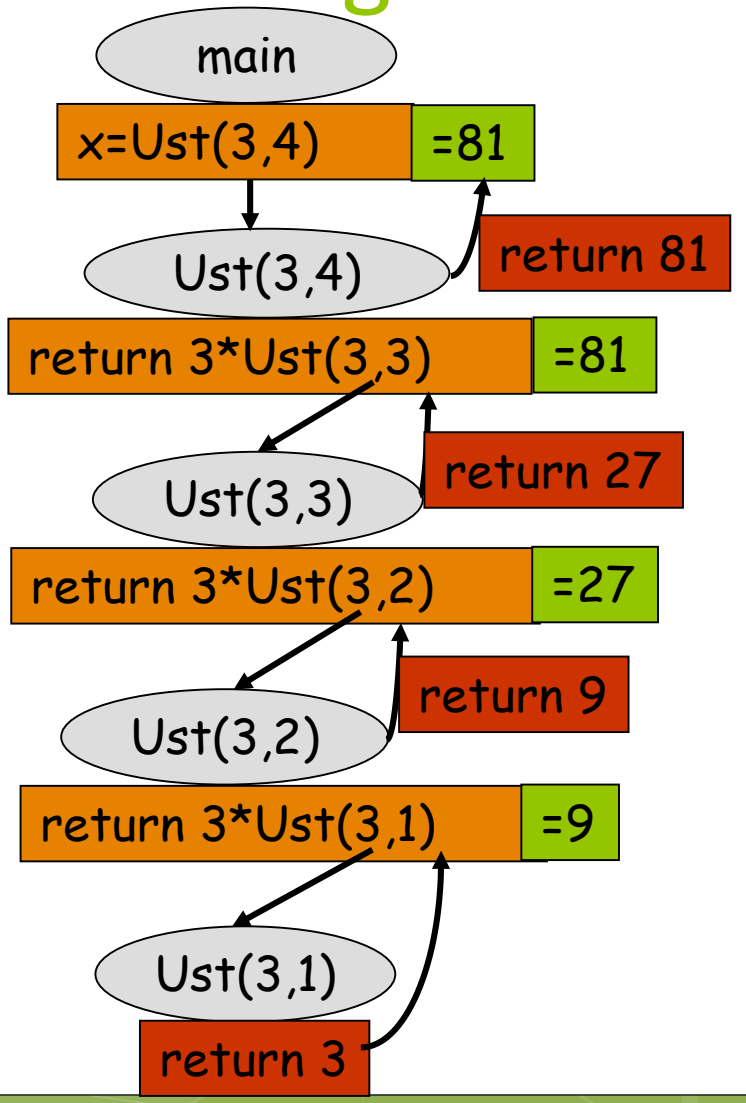
Ust(3, 4) için Özyineleme ağacı

```
/* Hesapla a^n */
double Ust(double a, int n){
  /* Temel durum */
  if (n == 0) return 1;
  else if (n == 1) return a;

  return a * Ust(a, n-1);
} /* bitti-Ust */

Public ... main(...){
  double x;

  x = Ust(3, 4);
} /* bitti-main */
```



Ust(a, n)'nin Çalışma Zamanı

```
/* Hesapla a^n */
double Ust(double a, int n){
  /* temel durum */
  if (n == 0) return 1;
  else if (n == 1) return a;

  return a * Ust(a, n-1);
} /* bitti-Ust */
```

$$T(n) = \begin{cases} n <= 1 \rightarrow 1 \text{ (Temel durum)} \\ N > 1 \rightarrow T(n-1) + 1 \end{cases}$$

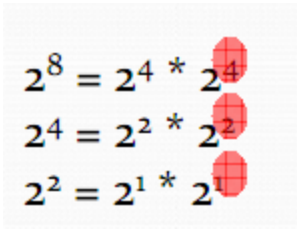
Pow(x, n) ’nin Çalışma Zamanı

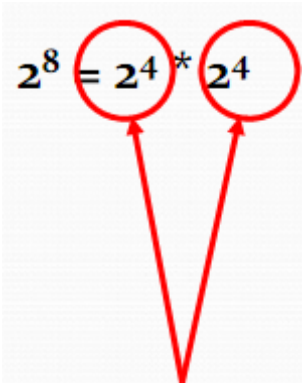
- Bu fonksiyon O(n) zamanında çalışır (n adet özyineli çağrı yapılır)
- Daha iyi bir çözüm var mı?
- Ara sonuçların karesini alarak daha etkin bir doğrusal özyineli algoritma yazabiliriz:

$$p(x,n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

Power(2, 8)

- $2^8=2*2*2*2*2*2*2*2$
- Ancak biz bu çözümü iki eşit parçaya bölerek ifade ede biliriz:
- $2^8=2^4*2^4$
- ve $2^4=2^2*2^2$
- ve $2^2=2^1*2^1$





- herhangi bir sayının 1. kuvveti kendisidir.
- Avantaj...
- İkisi de aynı olduğu için, her ikisini de hesaplamıyoruz! ve sadece 3 adet çarpma işlemi yapıyoruz.

Kuvvet değeri tek sayı ise

- Tek olanları şu şekilde yaparız:
- $2^{tek} = 2 * 2^{(tek-1)}$
- Peki öyleyse, 2^{21} 'i hesaplayalım

$2^{21} = 2 * 2^{20}$ (Tek sayı hilesi)

$2^{20} = 2^{10} * 2^{10}$

$2^{10} = 2^{5} * 2^{5}$

$2^{5} = 2 * 2^{4}$

$2^{4} = 2^{2} * 2^{2}$

$2^{2} = 2^{1} * 2^{1}$

- Görüldüğü gibi 20 defa çarpım yerine sadece 6 kerede sonuca ulaşılıyor

$2^{21} = 2 * 2^{20}$

$2^{20} = 2^{10} * 2^{10}$

$2^{10} = 2^{5} * 2^{5}$

$2^{5} = 2 * 2^{4}$

$2^{4} = 2^{2} * 2^{2}$

$2^{2} = 2^{1} * 2^{1}$

Özyinelemenin mantığı

- Eğer üst çift ise, iki parçaya ayırıp işleriz.
- Eğer üst tek ise, 1 çıkarır daha sonra kalan kısmı ikiye ayırıp işleriz. Ama üstten bir çıkardığımız için en sonunda 1 adet çarpma işlemi daha yapmayı unutmayın.

- İşlemi formülize etmeye başlayalım:
  - e == 0 , Pow(x, e) = 1
  - e == 1 , Pow(x, e) = x
  - e çift ise , Pow(x, e) = Pow(x, e/2) * Pow(x,e/2)
  - e>1 ve tek ise , Pow(x, e) = x * Pow(x, e-1)

Power(x, n) 'nin Çalışma Zamanı

$$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$$
$$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$$
$$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$$
$$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$$

Algorithm Power(x, n):

Input: x sayısı ve n tamsayısı, n >= 0

Output: xn değeri

if n = 0 **then**

return 1

if n is odd **then**

y = Power(x, (n - 1)/ 2)

return x · y ·y

else

y = Power(x, n/ 2)

return y · y

Her özyineli çağırmada n sayısını 2'ye bölüyoruz; dolayısıyla, log n özyineli çağrı yaparız. Bu metod O(log n) zamanına sahiptir.

Burada ara sonucu y değişkeni ile göstermemiz önemli; şayet metod çağırma yazarsak metod 2 defa çağrılmış olur.

Fibonacci Sayıları

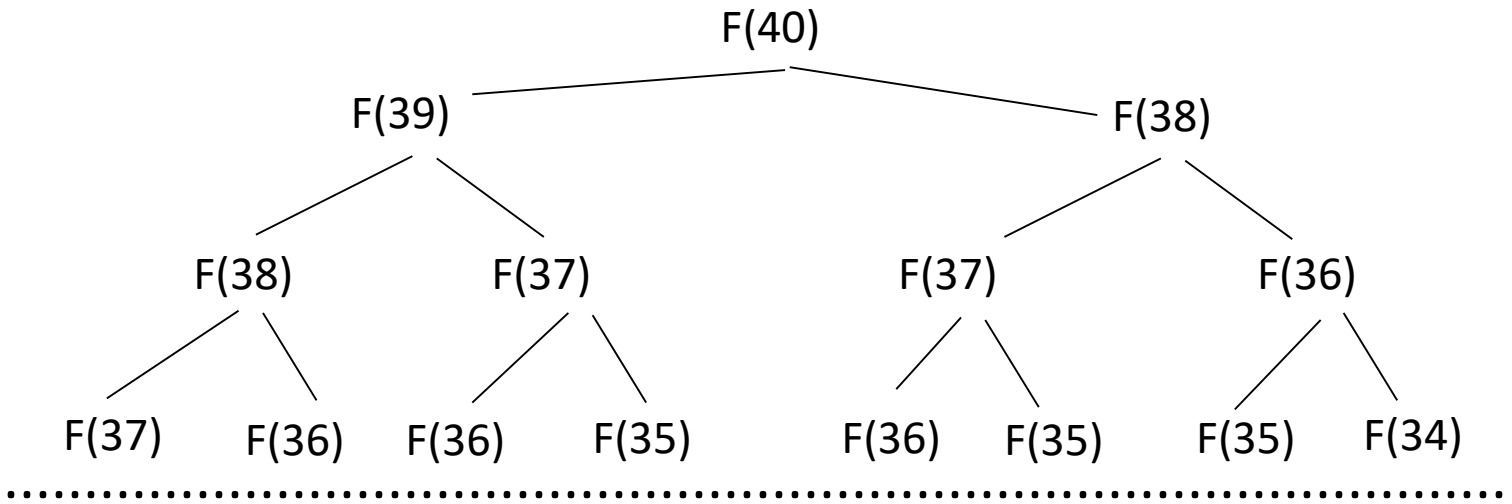
- Fibonacci sayılarını tanımlayacak olursak:
- 1 1 2 3 5 8 13.....
  - F(0) = 0
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2)

```
/* n. Fibonacci sayısını hesaplama*/
int Fibonacci(int n){
  /* Temel durum */
  if (n == 0) return 0;
  if (n == 1) return 1;

  return Fibonacci(n-1) + Fibonacci(n-2);
} /* bitti-Fibonacci */
```

Fibonacci Sayıları

- Fibonacci sayılarının tanımı özyinelemelidir.
- Örneğin 40. fibonacci değerini bulmaya çalışalım.



- F(40) için toplam kaç tane özyinelemeli çağrı yapılır.
- Cevap: 300 000 000 den fazla yordam çağrılır.
- Çalışma zamanı yaklaşık $O(2^n)$ Bellek karmaşıklığı O(derinlik*n)

Fibonacci Sayıları

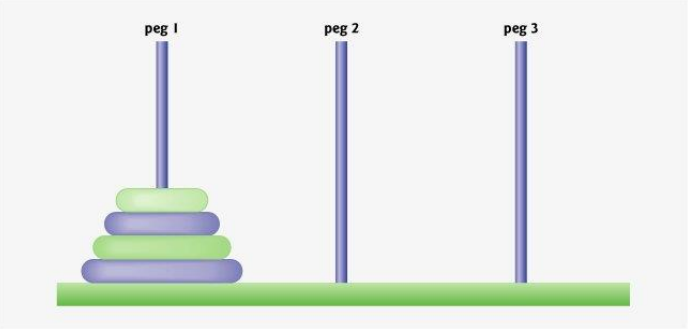
- Basit bir "for" ile çözülebilecek problemler için özyinelemeli algoritmalar kullanılmaz.

```
/* n. Fibonacci sayısını hesaplama*/
public static int fibonacci(int n){
  if(n == 1 || n == 2)  return 1;

  int s1=1,s2=1,sonuc=0;
  for(int i=0; i<n; i++){
    sonuc = s1 + s2;
    s1 = s2;
    s2 = sonuc;
  }
  return sonuc;
}
```

Hanoi Kuleleri

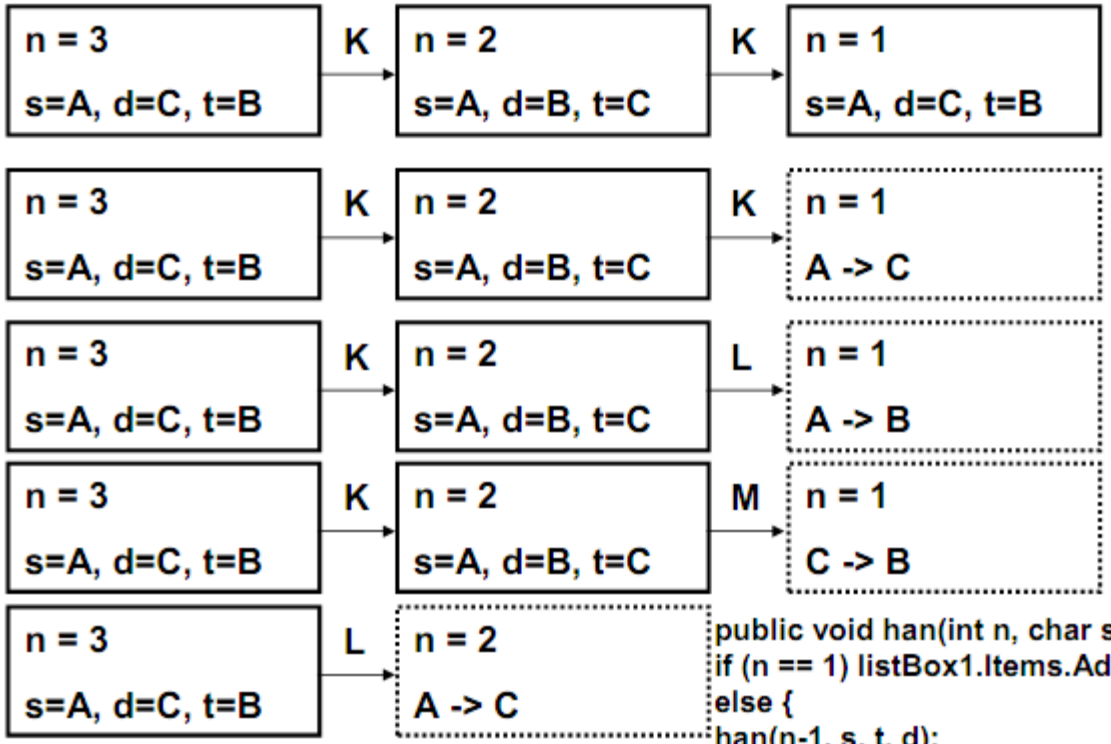
- **Verilen:** üç iğne
  - İlk iğnede en küçük disk en üstte olacak şekilde yerleştirilmiş farklı büyüklükte disk kümesi.
- **Amaç:** diskleri en soldan en sağa taşımak.
- **Şartlar:** aynı anda sadece tek disk taşınabilir.
- Bir disk boş bir iğneye veya daha büyük bir diskin üzerine taşınabilir.



Hanoi Kuleleri– Özyinelemeli Çözüm-Java

```
package hanoikuleleri;
import java.util.*;
public class Hanoikuleleri  {

public static void main(String[] args)
{
 System.out.print("n değerini giriniz : ");
 Scanner klavye = new Scanner(System.in);  int n = klavye.nextInt();
 tasi(n, 'A', 'B', 'C');
}
 public static void tasi(int n, char A, char B, char C)
{if(n==1) System.out.println(A + " --> " + B);
 else
 {
 tasi(n-1, A, C, B);      tasi(1, A, B, C);    tasi(n-1, C, B, A); }
 return;
}
```




```
public void han(int n, char s, char d, char t) {
if (n == 1) listBox1.Items.Add(s + " -> " + d);
else {
han(n-1, s, t, d);          // K
han(1, s, d, t);            // L
han(n-1, t, d, s);          // M
}
```

Kuyruk Özyineleme (Tail Recursion)

- Özyineleme metodunda, özyineli çağrı son adım ise bu durum oluşur. Yineleme çağrısı metodun en sonunda yapılır.
- int recFact(int n)
- {
- if (n<=1) return 1;
- else return n * recFact(n-1);
- }
- void tail() {
- …..
- …..
- tail(); }

Kuyruk Olmayan Özyineleme (nonTail Recursion)

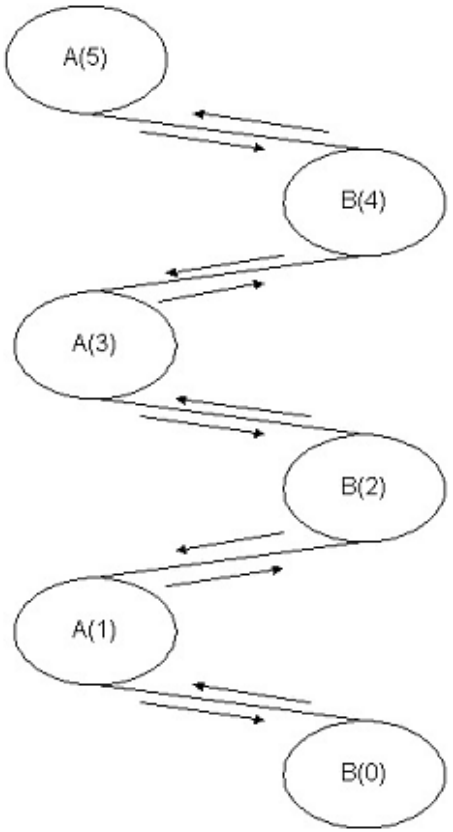
- Özyineleme metodunda, özyineli çağrı son adım değil ise bu durum oluşur. Yineleme çağrısından sonra başka işlemler yapılır (yazdırma v.b.) veya ikili özyineleme olabilir.

- Taban durumu haricinde iki özyinelemeli çağrı yapılıyorsa, ikili özyineleme oluşur.

```
int nontail(int n)
{
  if (n > 0)  {
    .......
    nontail(n-1);
    printf(n);
    .......
   nontail(n-1); }
}
```

Dolaylı Özyineleme (Indirect Recursion)

- Yineleme çağrısı başka bir fonksiyonun içinden yapılır.
- **void A(int n)**
- **{**
  - **if (n <= 0) return 1;**
  - **n- -;**
  - **B(n);**
- **}**
- **void B(int n)**
- **{**
  - **if (n <= 0) return 1;**
  - **n- -;**
  - **A(n);**
- **}**



İç içe Özyineleme (Nested Recursion)

- Yineleme çağrısı içindende yineleme çağrısı yapılır.

```
int A(int n, int m)
{
    if (n <= 0) return 1;
    return A(n-1, A(n-1, m-1));
}
```

Özyineleme mi İterasyon mu?

- Eğer iteratif çözümü tercih ederseniz algoritmanızın karmaşıklığı **O(n)** olur.
- Gerçek dünyada bu yöntemi kullanırsanız kovulursunuz.
- Eğer dinamik programlama tasarım yöntemi ile özyineleme kullanırsanız, algoritmanızın karmaşıklığı $O(\log_2 n)$ olur.
- Bunun için terfi bile alabilirsiniz.

ÖZET

- Özyineleme bütün doğada mevcuttur.
- Formül ezberlemekten daha kolaydır. Her problemin bir formülü olmayabilir, ancak her problem, küçük, tekrarlayan adımlar serisi olarak ifade edilebilir.
- Özyinelemeli bir işlemde her adım küçük ve hesaplanabilir olmalıdır.
- Kesinlikle sonlandırıcı bir şarta sahip olmalı.
- Özyineleme en çok, özyinelemeli olarak tanımlanmış binary tree dediğimiz veri yapılarında kullanılmak üzere yazılan algoritmalar için faydalıdır. Normal iterasyona göre çok daha kolaydır.
- Özyineleme, böl/yönet (divide & conquer) ve dinamik program türündeki algoritmalar için mükemmeldir.

Ödev

- 1- n adet x değeri için standart sapmayı (σ) bulan programı iterasyon ve recursive ile yapınız.

- xm=(1/n) $\sum_{k} x_k$

- (σ) = standart sapma
- V = varyans değeri
- xm = mean değeri

$$\sigma = \sqrt{V}$$

$$V = (1 / (n - 1)) \sum_{k} (x_k - xm)^2$$

Ödev

- 2- Ackerman fonksiyonu aşağıdaki şekilde tanımlanmıştır. Bu fonkisyonu öz yinelemeli olarak gerçekleştiriniz.

$$A(m,n)=\begin{cases} n+1 & ,m=1 \\ A(m-1,1) & ,m>0\ ve\ n=0 \\ A\big(m-1,A(m,n-1)\big) & ,m>0\ ve\ n>0 \end{cases}$$

Hashing

HASHING (Kırpma veya Çırpma)

- Hashing, elimizdeki veriyi kullanarak o veriden elden geldiği kadar benzersiz bir tamsayı elde etme işlemidir.
- Bu elde edilen tamsayı, dizi şeklinde tutulan verilerin indisi gibi kullanılarak verilere tek seferde erişmemizi sağlar.
- Hasing konusunu alt başlıklara ayıracak olursak;
  - **Hashing Fonksiyonu**
  - **Hash Tablosu**
  - **Çakışma (collision)**

HASHING (KIRPMA)

- **Hash Fonksiyonları**
  - Selecting Digits- Rakam seçme
  - Folding (shift folding, boundary folding)-Katlama
  - Division-Bölme
  - Mid-Square-Orta-Kare
  - Extraction-Çıkarım
  - Radix Transformation- Taban Dönüşüm

- **Çakışma (Collision) ve çözümler**
  - Linear Probing- Doğrusal doldurma
  - Double Hashing- Çift Kırpma
  - Quadratic Probing-Kuadratik Doldurma
  - Chaining-Zincirleme

HASHING

- Arama metotlarında temel işlem anahtarları karşılaştırmaktır. Bir anahtarın tablo içerisinde bulunduğu pozisyona ulaşıncaya kadar arama işlemine devam edilir.

- Hash fonksiyonuyla aranan anahtar elemana doğrudan erişilebilmektedir. Hash fonksiyonu bir anahtar bilgisinin tabloda bulunduğu indeksi hesaplamaktadır.

- Open hashing (Açık kırpma): Potansiyel olarak limitsiz alan kullanır.
- Closed hashing (Kapalı kırpma): Bilgi kaydı için sabit alan kullanır.

HASHING (Kırpma veya Çırpma)

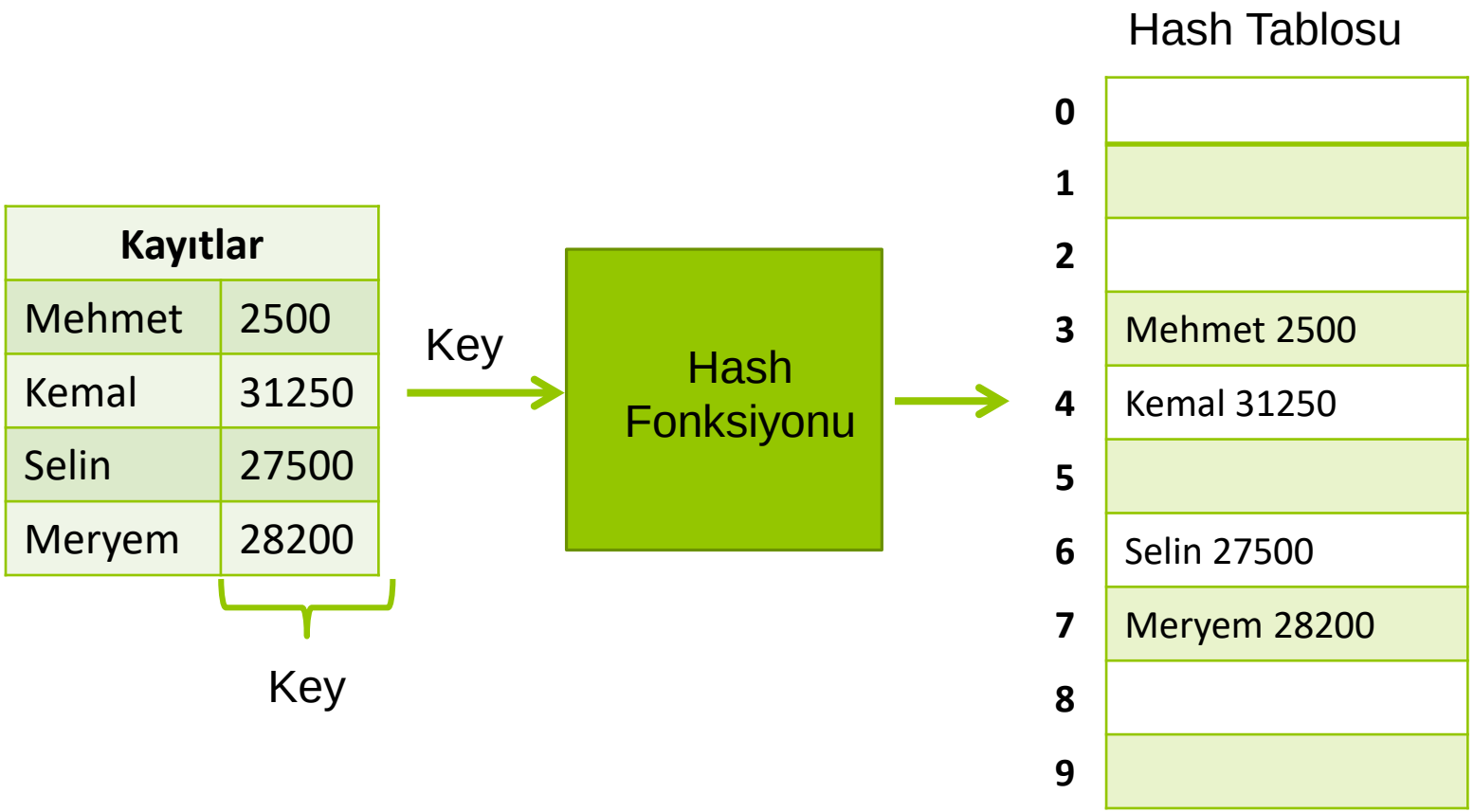
- Boyutu N olan bir tabloda, hash fonksiyonu (h(x)) bir x anahtarını 0 ile N-1 arasında bir değerle eşleştirir.
- Örnek:
- N=15 olan bir tablo için h(x) = x % 15 (% - > modlu bölüm) olarak belirlenebilir. Eğer,

x	25	129	35	2501	47	36
h(x)	10	9	5	11	2	6

- Anahtarların tablo içerisindeki yerleri ise aşağıdaki gibidir:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-	-	47	-	-	35	36	-	-	129	25	2501	-	-	-

HASHING (Kırpma veya Çırpma)



Hash fonksiyonları

- Hash fonksiyonları, bir anahtarın tabloda bulunduğu indeks sırasını verir.
- **Perfect hash fonksiyonu:**
- Her bir anahtara sadece bir pozisyonu eşleştiren fonksiyona denir.
- **Simple perfect hash fonksiyonu:**
- Tablo boyutu ile toplam anahtar sayısı eşit olduğunda (tabloda boş yer yoksa) her bir anahtara sadece bir pozisyonu eşleştiren fonksiyona denir.
- **İyi bir hash fonksiyonu:**
  - kolay ve hızlı hesaplanabilir olmalıdır.
  - tablodaki her bir pozisyon için sadece bir anahtar atamalıdır.

Hash fonksiyonları

- Hash fonksiyonları integer sayılarla işlem yaparlar.
- Integer olmayan anahtarlarda integer değere dönüştürme işlemi yapılır.
- Örneğin kişilere ait sağlık numarası 9635-8904 şeklinde ise aradaki tire işareti kaldırılarak 96358904 olarak alınır.
- Eğer anahtar karakterlerden oluşuyorsa karakterlerin ASCII kodları kullanılır.

Hash fonksiyonları (Selecting Digits – Rakam Seçme)

- Anahtar üzerindeki belirlenmiş bazı haneleri seçip birleştirerek tablodaki pozisyon bulunur.
- 2. ve 5. hanelerin seçimiyle oluşturulan değer aşağıdaki gibidir.
  - h(033475678) = 37
  - h(023455678) = 25
- **Artıları ve Eksileri**
  - Yapısı basittir.
  - Anahtarları tablonun tamamına düzgün bir şekilde dağıtamaz.
  - Çakışma çok sık olabilir.

Hash fonksiyonları (Folding-katlama)

- Anahtar birkaç parçaya bölünür ve bu parçalar kendi arasında toplanarak tablodaki pozisyon bulunur.
- Shift folding metodunda anahtarın her bir parçası değiştirilmeden tablo boyutuna göre mod ile toplanır.
  - Örnek: SSN = 123-45-6789 olarak verilsin. SSN numarası 123, 456, 789 olarak üç parçaya ayrılıp toplandığında 123+456+789 = 1368 olarak pozisyon numarası elde edilir.
- Boundary folding metodunda anahtarın parçalarının sırası değiştirilerek tablo boyutuna göre mod ile toplanır.
  - Örnek: SSN numarası 123, 456, 789 olarak üç parçaya ayrılır. Birinci parça aynı sırada kalır ve ikinci parça ters sıralanır. Daha sonra üçüncü parça aynı sırada alınır ve tablo boyutuna göre mod ile toplanır. (123+654+789 = 1566)

Hash fonksiyonları (Division – Bölme)

- Anahtar değeri tablo boyutuna göre mod ile bölünür.
- Örnek: SSN = 123456789 olarak verilsin. Tablo boyutu 1000 olursa, hash fonksyonu sonucu aşağıdaki gibi bulur;
- hash(h) = 123456789 % 1000 = 789
- Yapısı basittir ancak çakışma olur.

Hash fonksiyonları (Mid-square, Orta kare)

- Anahtar değerin karesi alınır ve sonucun orta kısmı seçilerek tablodaki pozisyon değeri bulunur.
- Örnek: anahtar = 3121 olarak verilsin. Hash fonksyonu sonucu aşağıdaki gibi bulur;
- $3121^2 = 9740641$
- hash(3121) = 406
- Anahtarın karesi binary olarak gösterilebilir.
- $3121^2 = 100101001010000101100001$
- hash(3121) = 0101000010 = 322

Hash fonksiyonları (Extraction)

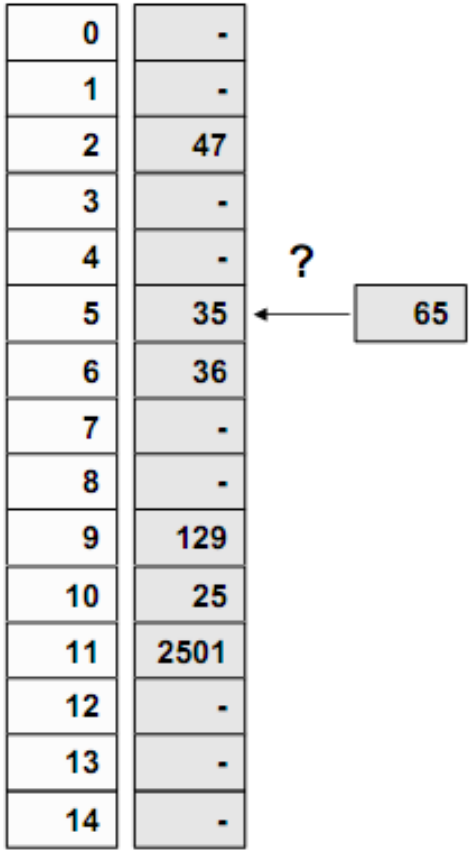
- Anahtar değerinin sadece bazı kısımları seçilerek tablodaki pozisyon değeri bulunur.
- Örnek: anahtar = 123-45-6789 olarak verilsin. Hash fonksiyonu aşağıdakilerden herhangi birisi olabilir;
- hash(123-45-6789) = 123456789 = 1234
- hash(123-45-6789) = 123456789 = 6789
- hash(123-45-6789) = 123456789 = 1289

Hash fonksiyonları (Radix Transformation)

- Anahtar değeri başka bir sayı tabanına dönüştürülür.
- Örnek: anahtar = 1238 olarak verilsin. Tablo boyutu 1000 olarak alındığında,
- $1238_{10} = 2326_{8}$
- Hesaplanan değer tablo boyutuna mod ile bölünerek pozisyon değeri bulunur.
- Hash(1238) = 2326 % 1000 = 326

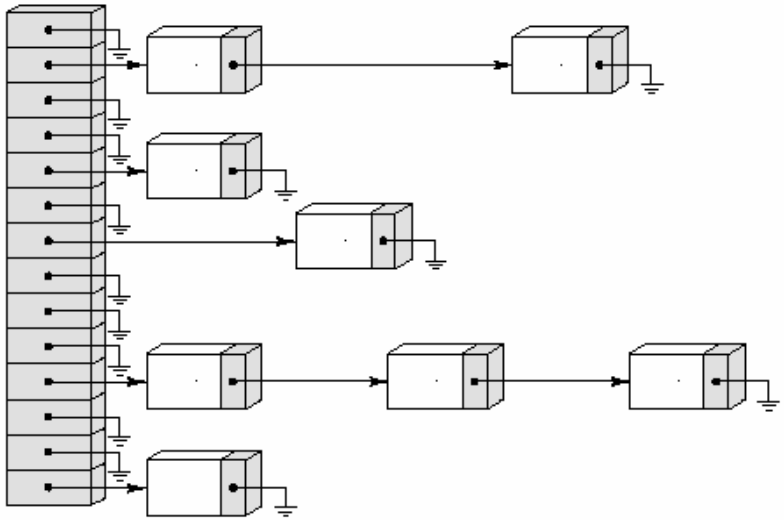
Hash fonksiyonları -Çakışma

- x = 65 değerini aşağıdaki tabloya ekleyim.
- x= 65
- h(x) =65 mod 15 = 5
- Aynı pozisyona birden fazla kayıt gelirse çakışma meydana gelir.



Hash fonksiyonları
Çakışmanın giderilmesi (Chaining)

- Aynı pozisyona gelen kayıtlar bağlı listelerle gösterilir.
- Ekleme: Listenin başına eklenir
- Silme/Erişim: Uygun listede arama yapılır

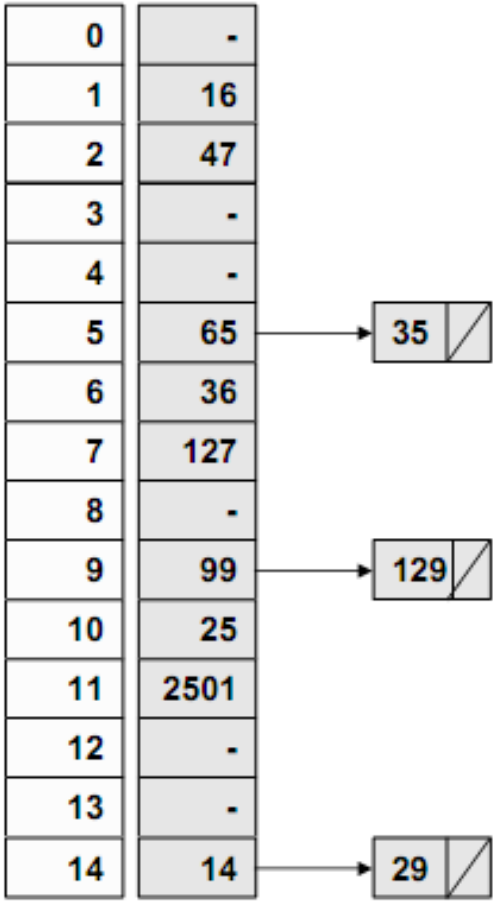


0	-
1	-
2	47
3	-
4	-
5	65 → 35
6	36
7	-
8	-
9	129
10	25
11	2501
12	-
13	-
14	-

Hash fonksiyonları
Çakışmanın giderilmesi (Chaining)

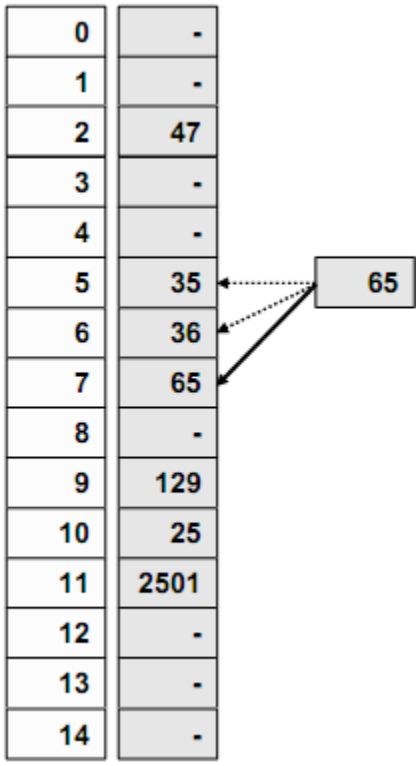
- Örnek:
- 29, 16, 14, 99, 127 sayılarının eklenmesi
- Aynı pozisyana gelen diğer anahtarlar bağlı listenin başına eklenmektedir.

- **Chaining metodunun dezavantajları**
- Tablonun bazı kısımları hiç kullanılmamaktadır.
- Bağlı listeler uzadıkça arama ve silme işlemleri için gereken zaman uzamaktadır.



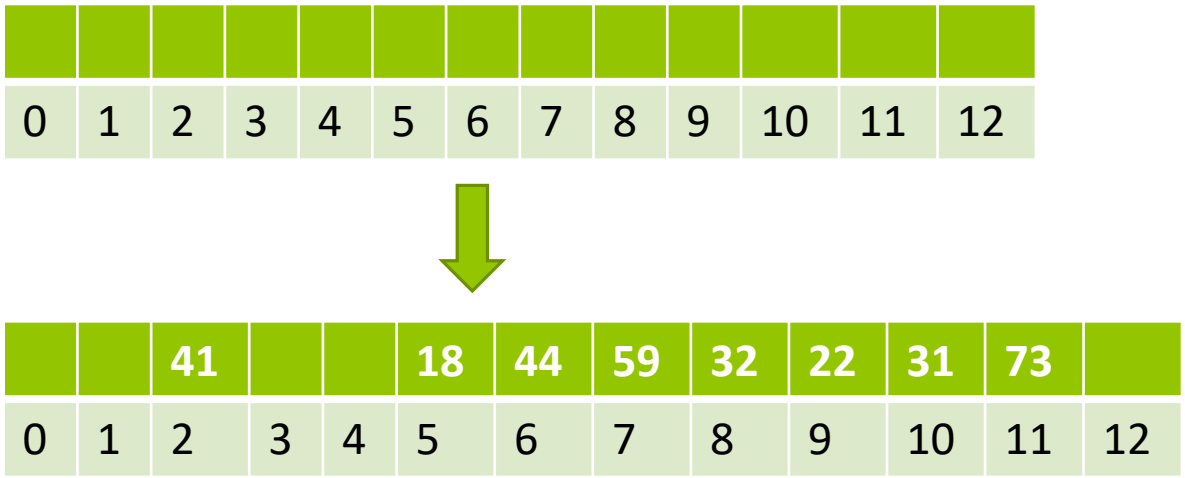
Hash fonksiyonları
Çakışmanın giderilmesi (Linear Probing)

- Aynı pozisyona gelen ikinci kayıt ilgili pozisyondan sonraki ilk boş pozisyona yerleştirilir.
- Ekleme: Boş bir alan bulunarak yapılır.
- Silme/Erişim: İlk boş alan bulunana kadar devam edebilir.
- h(k)=x % m
- h(k,i)=(h(k)'+i)%m
- x:anahtar, m: tablo boyutu



Hash fonksiyonları
Çakışmanın giderilmesi (Linear Probing)

- Örnek : h(x) = x mod 13
- 18, 41, 22, 44, 59, 32, 31, 73 değerlerini verilen sırada giriniz.



Hash fonksiyonları
Çakışmanın giderilmesi (Linear Probing)

- **Linear Probing metodunun avantajları / dezavantajları**

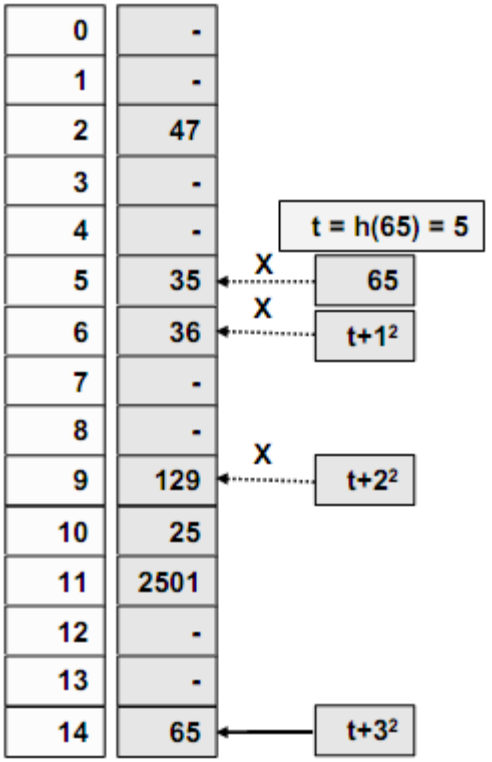
- Bağlı listeler gibi ayrı bir veri yapısına ihtiyaç duyulmaz.

- Kayıtların yığın şeklinde toplanmasına sebep olur.

- Silme ve arama işlemleri için gereken zaman aynı hash değeri sayısı arttıkça artar.

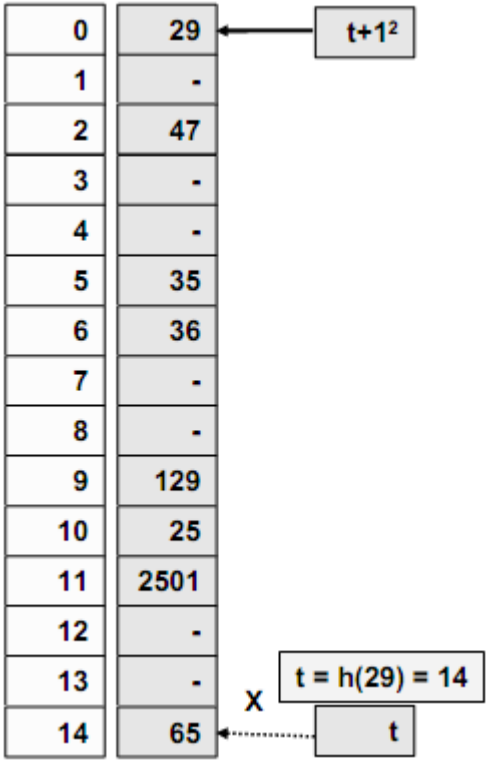
Hash fonksiyonları
Çakışmanın giderilmesi (Quadratic Probing)

- Aynı pozisyona gelen ikinci kayıt Quadratic Fonksiyonla yerleştirilir.
- En çok kullanılan fonksiyon
- ***x:anahtar, m:tablo boyutu***
- ***h(x)=x % m***
- ***h(x,i) = (h(k)'+i²)%m***
- Yeni pozisyon için sırasıyla $(t+1^2)$, $(t+2^2)$, ..., $(t+n^2)$ değerlerine karşılık gelen pozisyonlara bakılır ve ilk boş olana yerleştirilir.



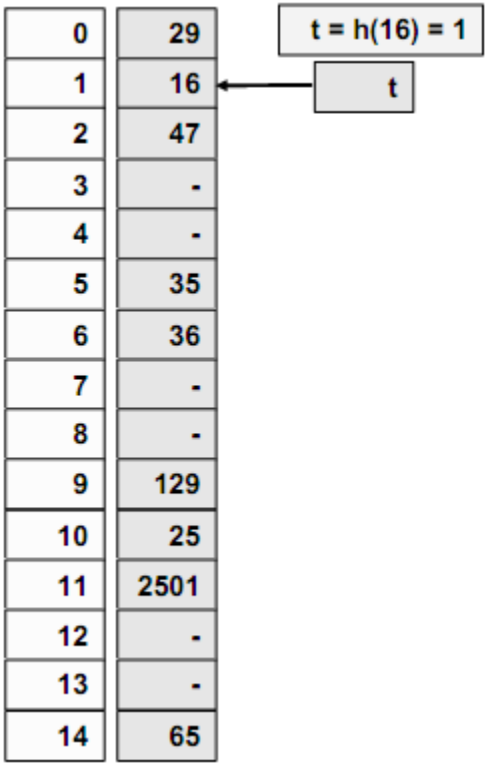
Hash fonksiyonları
Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- h(x) = x **mod** m
- h(29) = 29 % 15=14
- h(29,1)=(h(29)'+1)%15=(14+1)%15=0



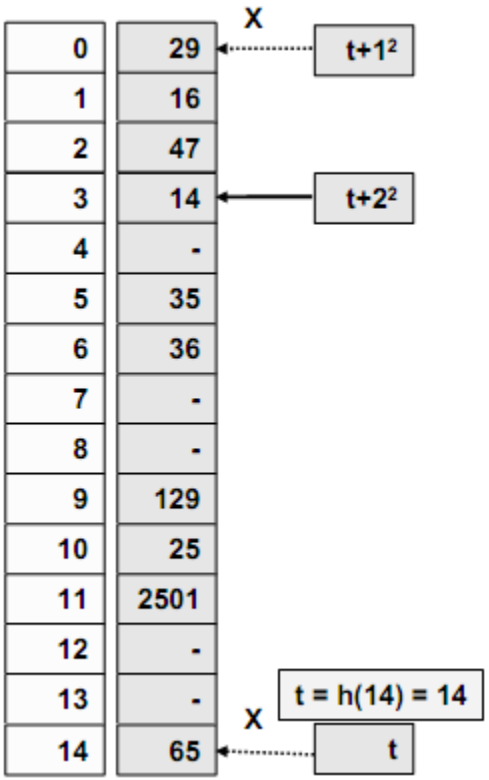
Hash fonksiyonları
Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- h(16) = 16 % 15=1



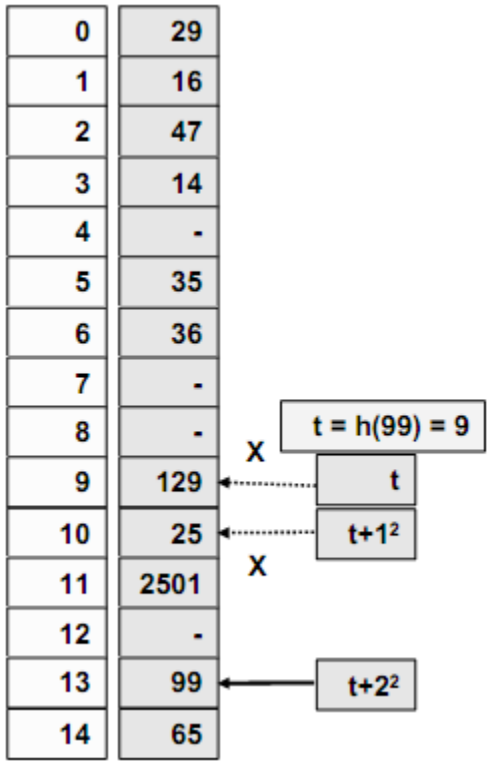
Hash fonksiyonları
Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- h(14) = 14 % 15=14
- h(14,1)=(14+1)%15=0
- h(14,2)=(14+2^2)%15=18%15=3



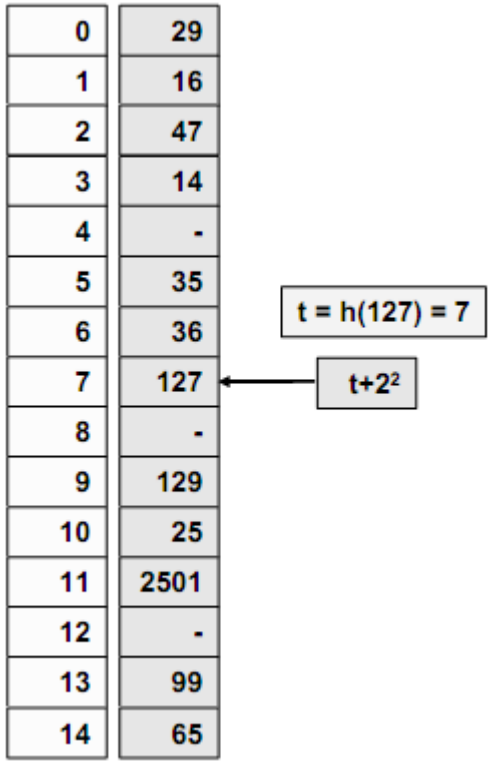
Hash fonksiyonları
Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- h(99) = 99 % 15=9
- h(99,1)=(9+1)%15=0
- h(99,2)=(9+2^2)%15=13%15=13



Hash fonksiyonları
Çakışmanın giderilmesi (Quadratic Probing)

- Örnek: 29, 16, 14, 99, 127 değerlerini hash tablosuna quadratic probing metoduyla sırayla yerleştiriniz.
- h(127) = 127 % 15= 7



Hash fonksiyonları
Çakışmanın giderilmesi (Quadratic Probing)

- **Quadratic Probing metodunun avantajları / dezavantajları**

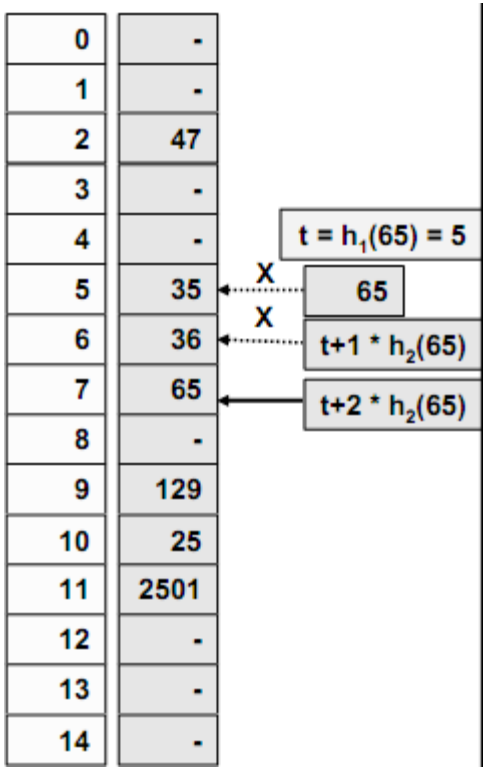
- Anahtar değerlerini linear probing metoduna göre daha düzgün dağıtır.
- Yeni eleman eklemede tablo boyutuna dikkat edilmezse sonsuza kadar çalışma riski vardır.
- Örn.: Boyutu 16 (0-15) olan bir tabloda 0, 1, 4 ve 9 pozisyanlarının dolu olduğu durumda 16 değerini eklemeye çalıştığımız zaman sonsuz döngüye girer.

Hash fonksiyonları
Çakışmanın giderilmesi (Double Hashing)

- Aynı pozisyona gelen ikinci kayıt için ikinci bir hash fonksiyonu kullanılır.
- İkinci hash fonksiyonu 0 değerini alamaz.
- En çok kullanılan fonksiyon:
- **$hash(x) = (hash_1(x) + i * hash_2(x)) \% m$**
- **$hash_2(x) = R - (x \% R)$**, $R < m$, R: asal (veya m ile aralarında asal)
- **$hash(x) = hash_1(x)$**
- **$hash(x) = (hash_1(x) + 1 * hash_2(x)) \% m$**
- **$hash(x) = (hash_1(x) + 2 * hash_2(x)) \% m$**
- **$hash(x) = (hash_1(x) + 3 * hash_2(x)) \% m$**

Hash fonksiyonları
Çakışmanın giderilmesi (Double Hashing)

- Örnek: 65 değerinin eklenmesi
- $hash_1(x)= x \% 15$
- $hash_2(x)=11 - (x \% 11)$
- $hash(65) = hash_1(65)$
- hash(65,0) =5 dolu
- $hash(65,1) = (hash_1(5) + 1 * hash_2(65))\%15$
- hash(65,1) =(5 +(11-10))%15=6
- $hash(65,2) = (hash_1(5) + 2 * hash_2(65))\%15$
- hash(65,2) =(5 +2)%15=7



Hash fonksiyonları
Çakışmanın giderilmesi (Double Hashing)

- **Double Hashing metodunun avantajları / dezavantajları**

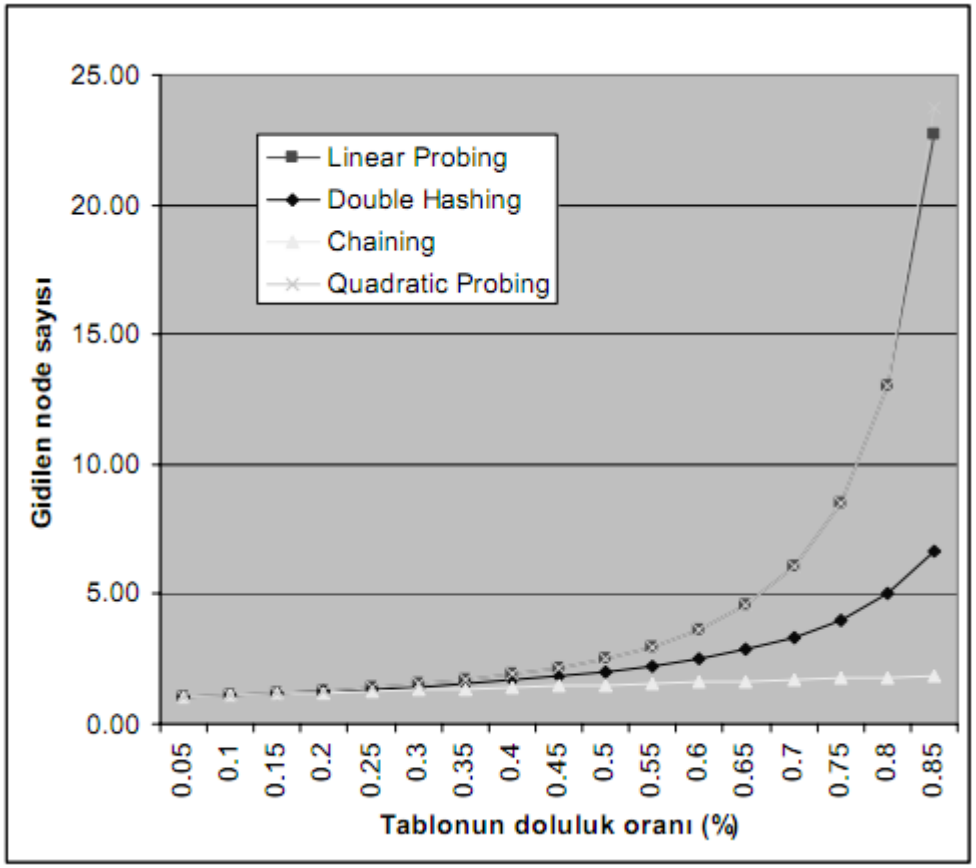
- Anahtar değerlerini linear probing metoduna göre daha düzgün dağıtır ve gruplar oluşmaz.

- Quadratic probing metoduna göre daha yavaştır çünkü ikinci bir hash fonksiyonu hesaplanır.

Hash fonksiyonları

- **Performans**

yapılacak sondalama sayısı:
1/(1-α)
α:tablo dolum oranı
α=n/m
n: girilen eleman sayısı
m: tablo boyutu



Ödev

- 100 tane anahtar değerini rastgele üreten ve bu değerleri boyutu 100 olan bir hash tablosuna yerleştiren programı yazınız.
- Programda hash fonksiyonu olarak division metodunu, çakışma çözümü için chaining, linear probing ile quadratic probing metodlarını ayrı ayrı kullanınız.
- Anahtar değerlerini integer olarak alınız.
- Hash tablosu için dizi yapısını kullanınız.

Chaining java kd

```
public class LinkedHashEntry
  {  private int key;
     private int value;
     private LinkedHashEntry next;
     LinkedHashEntry (int key, int value)
     {  this.key = key;
        this.value = value;
        this.next = null;
     }
     public int getValue()      {   return value;      }
     public void setValue(int value)      {   this.value = value;     }
     public int getKey()      {    return key;      }
     public LinkedHashEntry getNext()  { return next; }
     public void setNext(LinkedHashEntry next) { this.next = next; }
```

Chaining java kd

```
public class HashMap
{   private static int TABLE_SIZE = 128;
    LinkedHashEntry[] table;
    HashMap()
    {   table = new LinkedHashEntry[TABLE_SIZE];
        for (int i = 0; i < TABLE_SIZE; i++)      table[i] = null;
    }
  public int get(int key) {
    int hash = (key % TABLE_SIZE);
    if (table[hash] == null) return -1;
    else {       LinkedHashEntry entry = table[hash];
        while (entry != null && entry.getKey() != key)  entry = entry.getNext();
        if (entry == null)          return -1;
        else      return entry.getValue();
    }
  }
```

Chaining java kd

```
public void put(int key, int value) {
        int hash = (key % TABLE_SIZE);
        if (table[hash] == null)   table[hash] = new LinkedHashEntry(key, value);
        else {  LinkedHashEntry entry = table[hash];
           while (entry.getNext() != null && entry.getKey() != key)
              entry = entry.getNext();
           if (entry.getKey() == key)        entry.setValue(value);
           else      entry.setNext(new LinkedHashEntry(key, value));
        }
    }
```

Chaining java kd

```
public void remove(int key)
    {    int hash = (key % TABLE_SIZE);
         if (table[hash] != null) {
              LinkedHashEntry prevEntry = null;
              LinkedHashEntry entry = table[hash];
              while (entry.getNext() != null && entry.getKey() != key)
              {    prevEntry = entry;              entry = entry.getNext();        }
              if (entry.getKey() == key) {
                   if (prevEntry == null)           table[hash] = entry.getNext();
                   else              prevEntry.setNext(entry.getNext());
              }
         }
     }
 }
```

AĞAÇ (TREE) Veri Modeli

Ağaç Veri Modeli Temel Kavramları

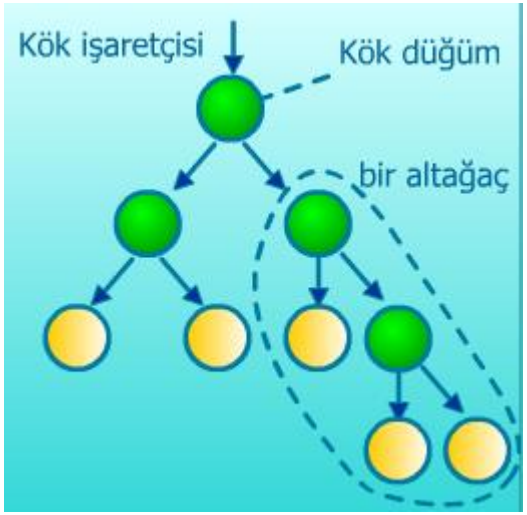
- Ağaç, bir kök işaretçisi, sonlu sayıda düğümleri ve onları birbirine bağlayan dalları olan bir veri modelidir; aynı aile soyağacında olduğu gibi hiyerarşik bir yapısı vardır ve orada geçen birçok kavram buradaki ağaç veri modelinde de tanımlıdır.

- Örneğin çocuk, kardeş düğüm, aile, ata gibi birçok kavram ağaç veri modelinde de kullanılır. Genel olarak, veri, ağacın düğümlerinde tutulur; dallarda ise geçiş koşulları vardır denilebilir.

- Her biri değişik bir uygulamaya doğal çözüm olan ikili ağaç, kodlama ağacı, sözlük ağacı, kümeleme ağacı gibi çeşitli ağaç şekilleri vardır; üstelik uygulamaya yönelik özel ağaç şekilleri de çıkarılabilir.

Ağaç Veri Modeli Temel Kavramları

- Bağlı listeler, yığıtlar ve kuyruklar doğrusal (linear) veri yapılarıdır. Ağaçlar ise doğrusal olmayan belirli niteliklere sahip iki boyutlu veri yapılarıdır.
  - Ağaçlar hiyerarşik ilişkileri göstermek için kullanılır.
  - Her ağaç node'lar ve kenarlardan (edge) oluşur.
  - Herbir node(düğüm) bir nesneyi gösterir.
  - Herbir kenar (bağlantı) iki node arasındaki ilişkiyi gösterir.
  - Arama işlemi bağlı dizilere göre çok hızlı yapılır.

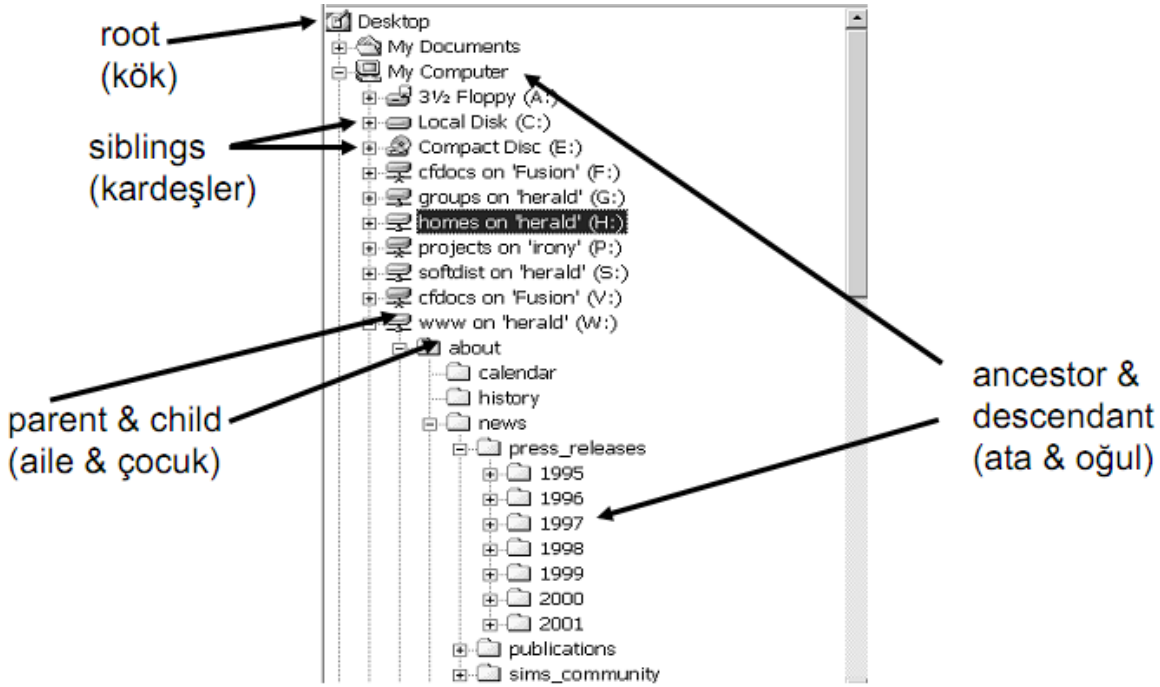


Ağaç Veri Modeli Temel Kavramları

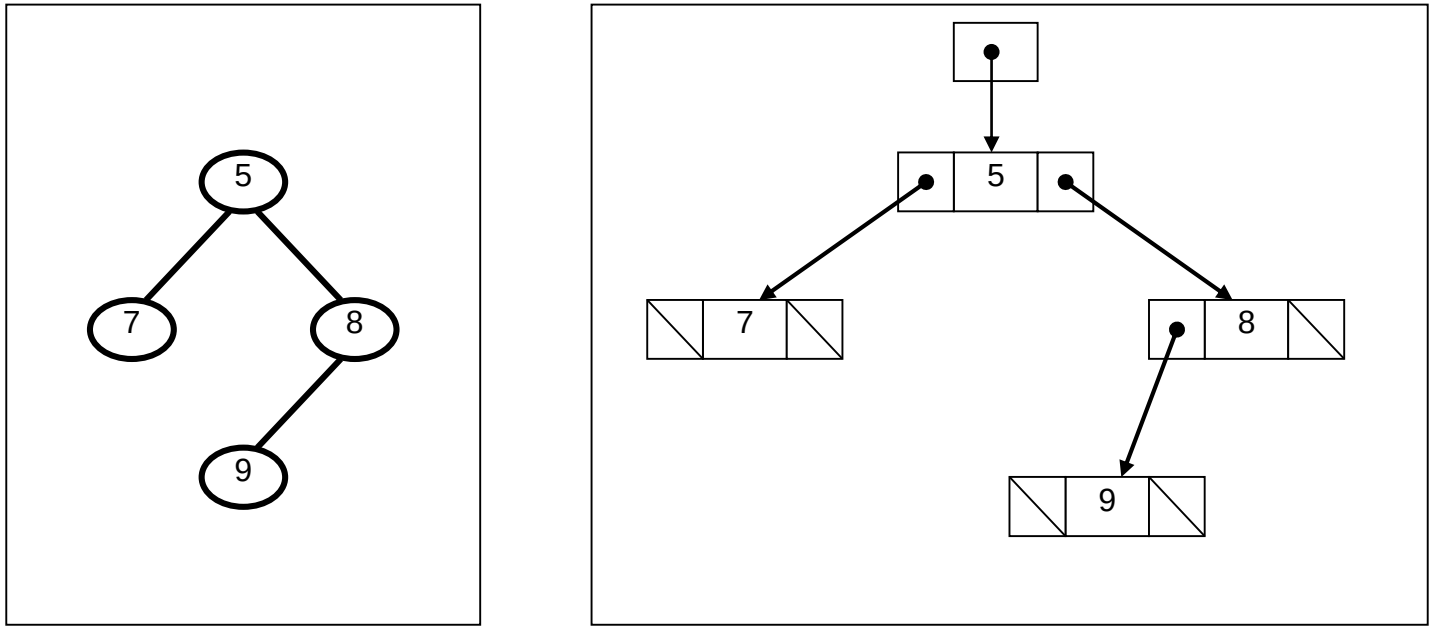
- Ağaçlardaki düğümlerden iki veya daha fazla bağ çıkabilir. İkili ağaçlar (binary trees), düğümlerinde en fazla iki bağ içeren (0,1 veya 2) ağaçlardır. Ağacın en üstteki düğümüne kök (root) adı verilir.
- **Uygulamaları:**
  - Organizasyon şeması
  - Dosya sistemleri
  - Programlama ortamları

Ağaç Veri Modeli Temel Kavramları

- Örnek ağaç yapısı



Ağaç Veri Modeli Temel Kavramları

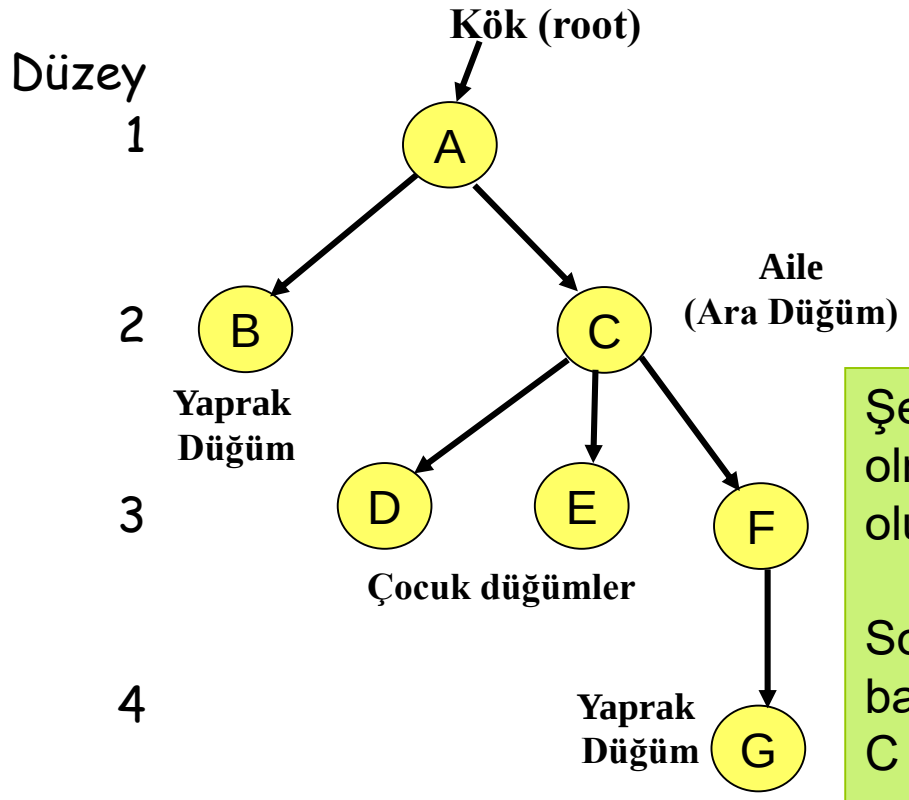


- **Şekil 1:** İkili ağacın grafiksel gösterimleri

Ağaç Veri Modeli Temel Kavramları

- Şekil 1’de görülen ağacın düğümlerindeki bilgiler sayılardan oluşmuştur. Her düğümdeki sol ve sağ bağlar yardımı ile diğer düğümlere ulaşılır. Sol ve sağ bağlar boş ("NULL" = "/" = "\") da olabilir.
- Düğüm yapıları değişik türlerde bilgiler içeren veya birden fazla bilgi içeren ağaçlar da olabilir.
- Doğadaki ağaçlar köklerinden gelişip göğe doğru yükselirken veri yapılarındaki ağaçlar kökü yukarıda yaprakları aşağıda olacak şekilde çizilirler.

Ağaç Veri Modeli Temel Kavramları



- **Şekil :** Ağaçlarda düzeyler

Şekildeki ağaç, A düğümü kök olmak üzere 7 düğümden oluşmaktadır.

Sol alt ağaç B kökü ile başlamakta ve sağ alt ağaç da C kökü ile başlamaktadır.

A'dan solda B'ye giden ve sağda C'ye giden iki dal (branch) çıkmaktadır.

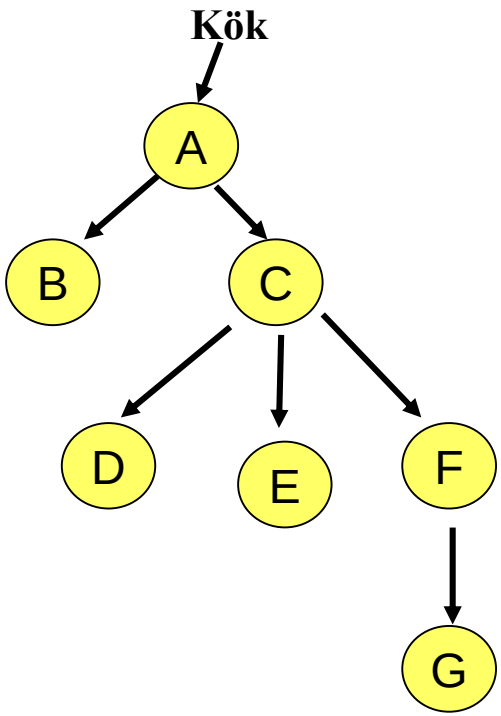
Ağaç Veri Modeline İlişkin Tanımlar

- **Düğüm (Node)**
  - Ağacın her bir elemanına düğüm adı verilir.
- **Kök Düğüm (Root)**
  - Ağacın başlangıç düğümüdür.
- **Çocuk (Child)**
  - Bir düğüme doğrudan bağlı olan düğümlere o çocukları denilir.
- **Kardeş Düğüm (Sibling)**
  - Aynı düğüme bağlı düğümlere kardeş düğüm veya kısaca kardeş denir.
- **Aile (Parent)**
  - Düğümlerin doğrudan bağlı oldukları düğüm aile olarak adlandırılır; diğer bir deyişle aile, kardeşlerin bağlı olduğu düğümdür.
- **Ata (Ancestor) ve Torun (Dedscendant)**
  - Aile düğümünün daha üstünde kalan düğümlere ata denilir; torun, bir düğümün çocuğuna bağlı olan düğümlere denir.

Ağaç Veri Modeline İlişkin Tanımlar

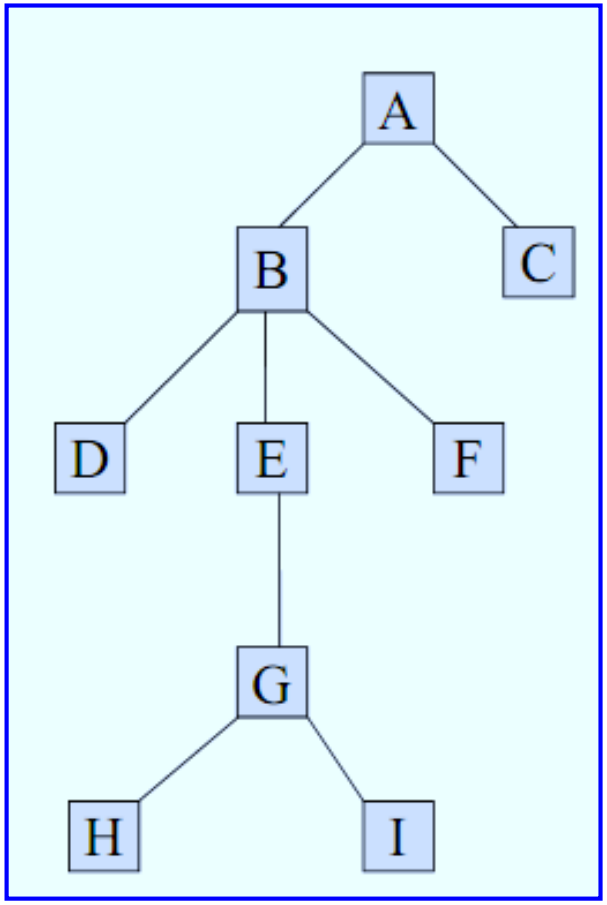
- **Derece (Degree)**
  - Bir düğümden alt hiyerarşiye yapılan bağlantıların sayısıdır; yani çocuk veya alt ağaç sayısıdır.
- **Düzey (Level) ve Derinlik (Depth)**
  - Düzey, iki düğüm arasındaki yolun üzerinde bulunan düğümlerin sayısıdır. Kök düğümün düzeyi 1, doğrudan köke bağlı düğümlerin düzeyi 2'dir. Bir düğümün köke olan uzaklığı ise derinliktir. Kök düğümün derinliği 1 dir.
- **Yaprak (Leaf)**
  - Ağacın en altında bulunan ve çocukları olmayan düğümlerdir.
- **Yükseklik (Height)**
  - Bir düğümün kendi silsilesinden en uzak yaprak düğüme olan uzaklığıdır.(İki düğüm arasındaki kenarların sayısı)
- **Yol(Path)**
  - Bir düğümün aşağıya doğru (çocukları üzerinden) bir başka düğüme gidebilmek için üzerinden geçilmesi gereken düğümlerin listesidir.

Ağaç Veri Modeline İlişkin Tanımlar



Tanım	Kök	B	D
Çocuk/Derece	2	0	0
Kardeş sayısı	1	2	3
Düzey	1	2	3
Aile	yok	kök	C
Ata	yok	yok	Kök
Yol	A	A, B	A,C,D
Derinlik	1	2	3
Yükseklik	3	2	1

Ağaç Veri Modeline İlişkin Tanımlar

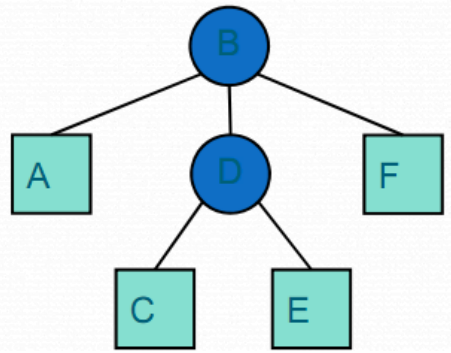


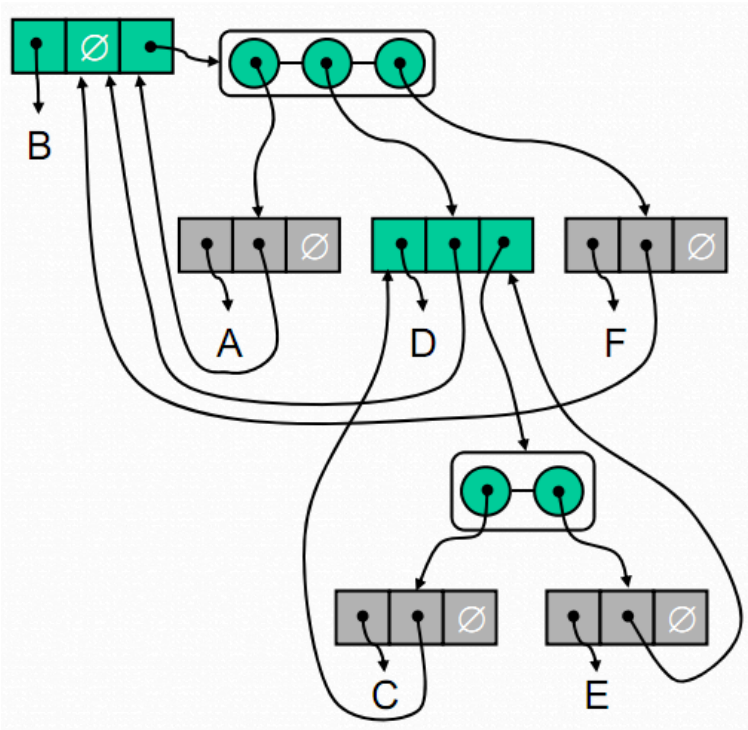
Tanım	Değer
Düğüm sayısı:	9
Yükseklik:	4
Kök düğüm:	A
Yapraklar:	C, D, F, H, I
Düzey sayısı:	5
H'nin ataları:	E, B, A
B'nin torunları:	G, H, I
E'nin kardeşleri:	D, F
Sağ alt ağaç:	Yok
Sol alt ağaç:	B

Ağaçların Bağlı Yapısı

- Bir düğüm çeşitli bilgiler ile ifade edilen bir nesnedir. Her bir bağlantı için birer bağlantı bilgisi tutulur.
  - Nesne/Değer (Element)
  - Ana düğüm (Parent node)
  - Çocuk düğümlerin listesi
- Problem: Bir sonraki elemanın çocuk sayısını bilmiyoruz.
- Daha iyi: Çocuk/Kardeş Gösterimi
  - Her düğümde iki bağlantı bilgisi tutularak hem çocuk hem de yandaki kardeş tutulabilir.
  - İstenildiği kadar çocuk/kardeş olabilir.

```
JAVA Declaration
class AgacDugumu {
    int eleman;
    AgacDugumu ilkCocuk;
    AgacDugumu kardes; }
```





Ağaç İşlemleri

- Genel Yöntemler:
  - integer size()
  - boolean isEmpty()
  - elements()
  - positions()
- Erişim yöntemleri:
  - root()
  - parent(p)
  - children(p)
- Sorgu (Query) Yöntemleri:
  - boolean isInternal(p)
  - boolean isRoot(p)
  - boolean isExternal(p)
  - Güncelleme (Update) Yöntemi:
  - object replace (p, o)
- Ek yöntemler de tanımlanabilir

Ağaç Türleri

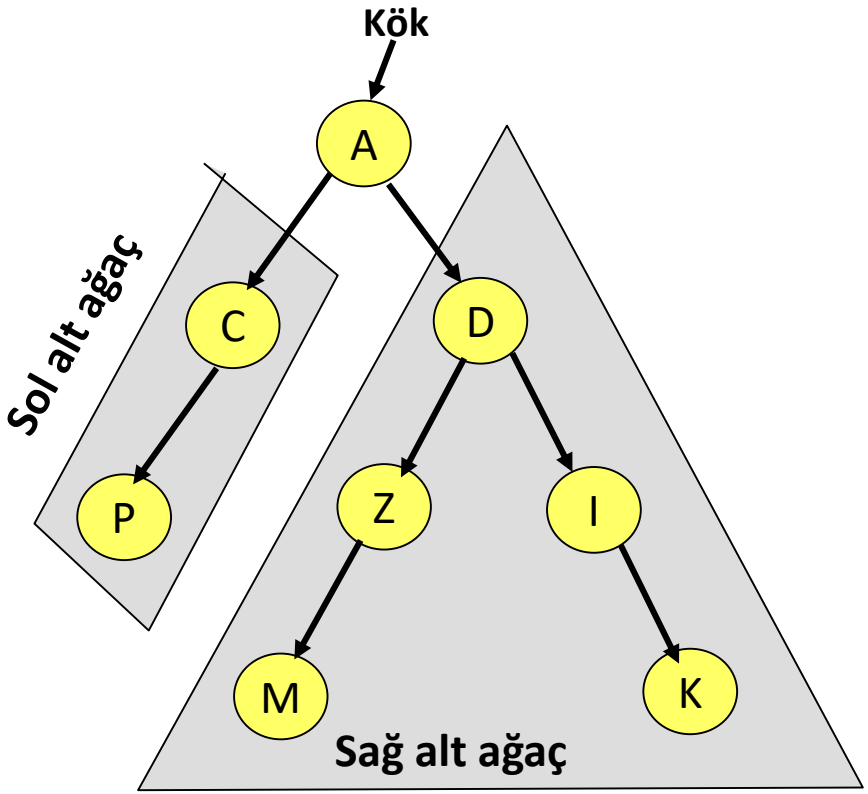
- En çok bilinen ağaç türleri ikili arama ağacı olup kodlama ağacı, sözlük ağacı, kümeleme ağacı gibi birçok ağaç uygulaması vardır.
  - **İkili Arama Ağacı (Binary Search Tree):**
    - İkili arama ağacında bir düğüm en fazla iki tane çocuğa sahip olabilir ve alt/çocuk bağlantıları belirli bir sırada yapılır.
  - **Kodlama Ağacı (Coding Tree):**
    - Bir kümedeki karakterlere kod ataması için kurulan ağaç şeklidir. Bu tür ağaçlarda kökten başlayıp yapraklara kadar olan yol üzerindeki bağlantı değerleri kodu verir.
  - **Sözlük Ağacı(Trie: Keyword Tree):**
    - Bir sözlükte bulunan sözcüklerin tutulması için kurulan bir ağaç şeklidir. Amaç arama işlemini en performanslı bir şekilde yapılması ve belleğin optimum kullanılmasıdır.
  - **Kümeleme Ağacı (Heap Tree):**
    - Bir çeşit sıralama ağacıdır. Çocuk düğümler her zaman aile düğümlerinden daha küçük değerlere sahip olur.

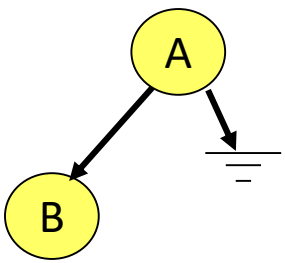
İkili Ağaç (Binary Tree) :

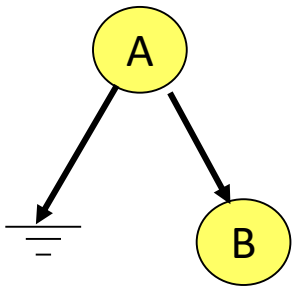
- Sonlu düğümler kümesidir. Bu küme boş bir küme olabilir (empty tree). Boş değilse şu kurallara uyar.
  - Kök olarak adlandırılan özel bir düğüm vardır.
  - Her düğüm en fazla iki düğüme bağlıdır.
    - Left child : Bir node'un sol işaretçisine bağlıdır.
    - Right child : Bir node'un sağ işaretçisine bağlıdır.
  - Kök hariç her düğüm bir daldan gelmektedir.
  - Tüm düğümlerden yukarı doğru çıkıldıkça sonuçta köke ulaşılır.

İkili Ağaç (Binary Tree) :

- Bilgisayar bilimlerinde en yaygın ağaçtır.



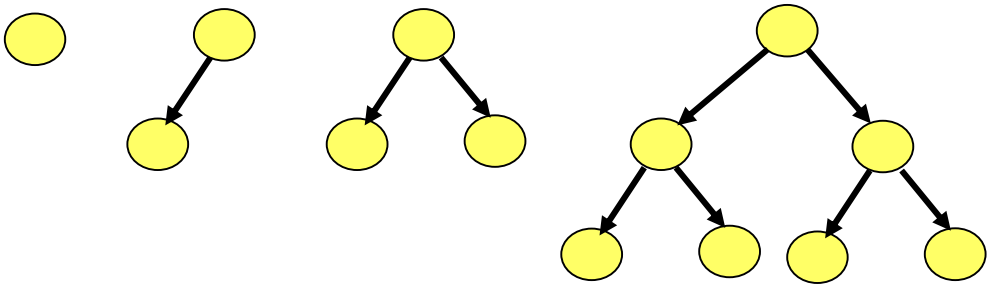




İki farklı ikili ağaç

İkili Ağaç (Binary Tree)

- **n** tane düğüm veriliyor, İkili ağacın minimum derinliği nedir.



Derinlik 1: n = 1, 1 düğüm 2^1 -1

Derinlik 2: n = 2, 3 düğüm, 2^2 -1 düğüm

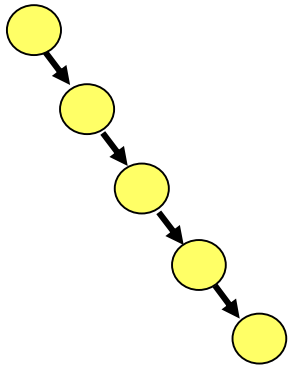
Herhangi bir d derinliğinde, n = ?

Derinlik d: n = 2^d-1 düğüm (tam bir ikili ağaç)

En küçük derinlik: **Θ(log n)**

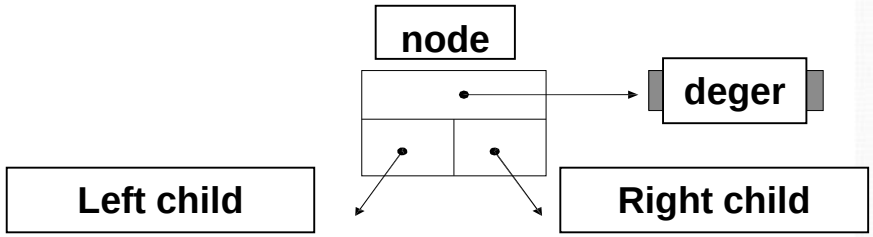
İkili Ağaç

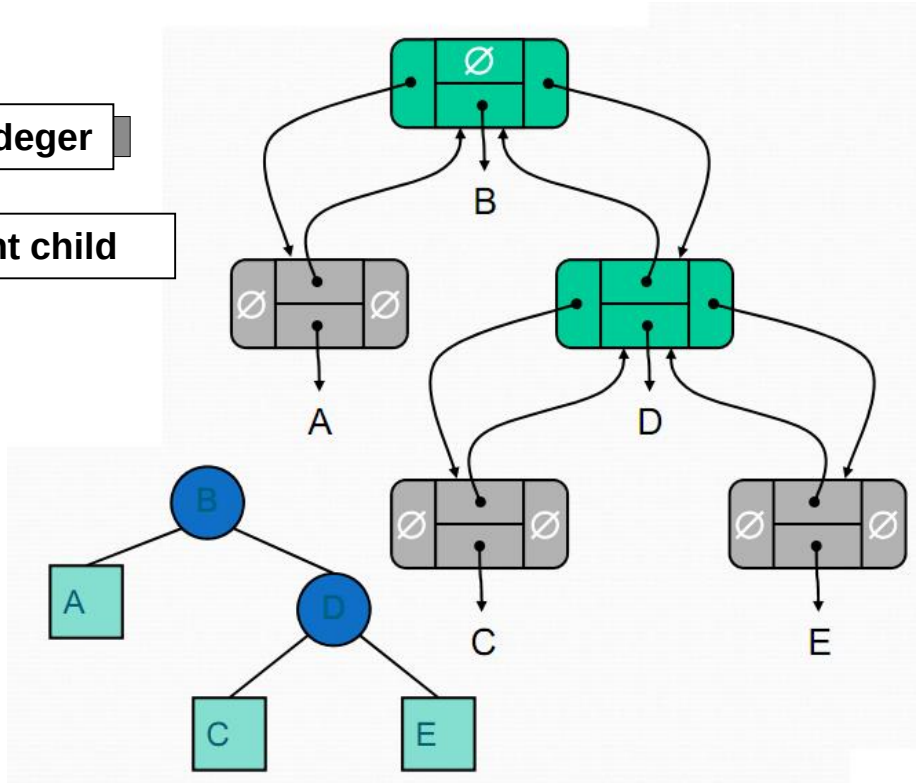
- ***n*** düğümlü ikili ağacın minimum derinliği: **Θ(log n)**
- İkili ağacın maksimum derinliği ne kadardır?
  - Dengesiz ağaç: Ağaç bir bağlantılı liste olursa!
  - Maksimum derinlik = n
- Amaç: Arama gibi operasyonlarda bağlantılı listeden daha iyi performans sağlamak için derinliğin log n de tutulması gerekmektedir.



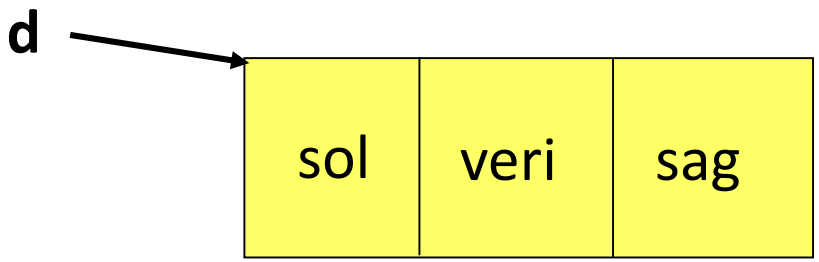
- Bağlantılı liste
- Derinlik = n

İkili Ağaç Bağlı Liste Yapısı

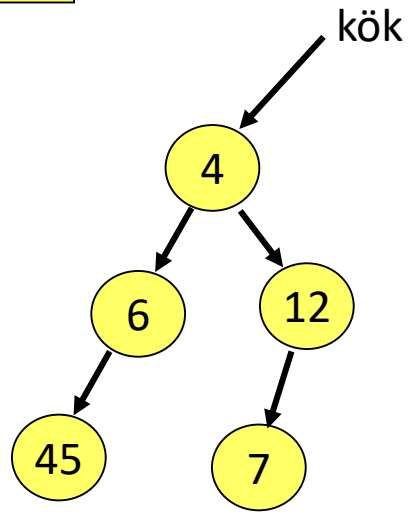




İkili Ağaç Gerçekleştirimi



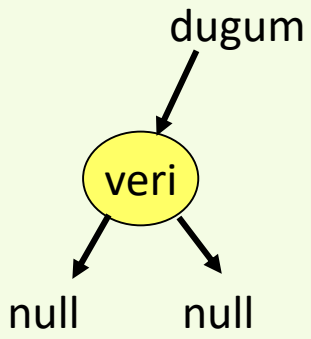

```
public class İkiliAgacDugumu {
    public İkiliAgacDugumu sol;
    public int veri;
    public İkiliAgacDugumu sag;
}
```



İkili Ağaç Gerçekleştirimi

- İkili ağaç için düğüm oluşturma. Her defasında sol ve sağı boş olan bir düğüm oluşturulur.

```
/* İkili ağaç düğümü oluşturur. */
  İkiliAgacDugumu DugumOlustur(int veri){
  İkiliAgacDugumu dugum = new İkiliAgacDugumu();
  dugum.veri = veri;
  dugum.sol = null;
  dugum.sag = null;
  return dugum;
}
```



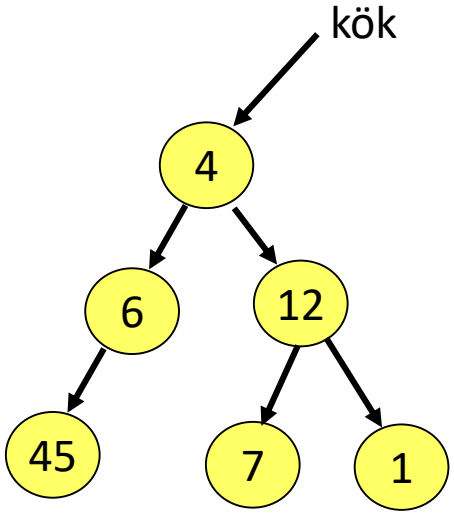
İkili Ağaç Gerçekleştirimi

- İteratif düğüm oluşturma ve ağaca ekleme.

```
İkiliAgacDugumu dugum = null;

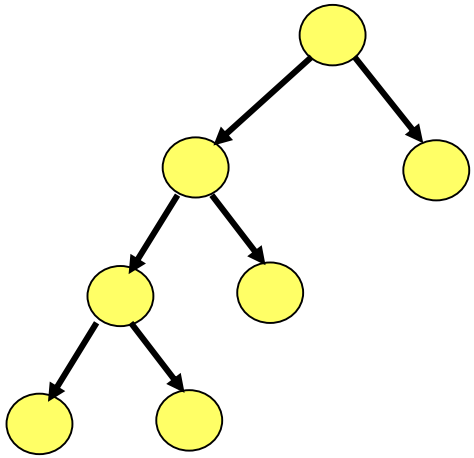
public static void main main(){
   kok = DugumOlustur(4);

   kok.sol = DugumOlustur(6);
   kok.sag = DugumOlustur(12);
   kok.sol.sol = DugumOlustur(45);
   kok.sag.sol = DugumOlustur(7);
   kok.sag.sag = DugumOlustur(1);
} /* main */
```

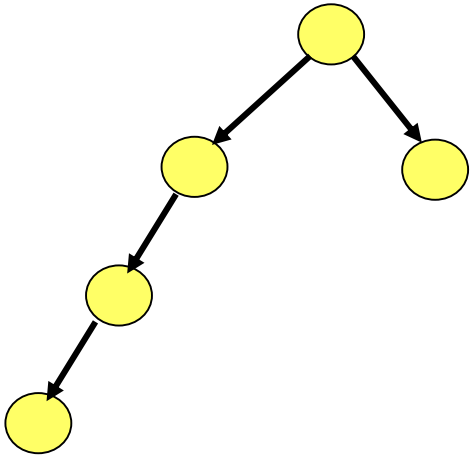


Proper (düzgün) Binary Trees

- Yaprak olmayan düğümlerin tümünün iki çocuğu olan **T** ağacı proper(düzgün) binary tree olarak adlandırılır.



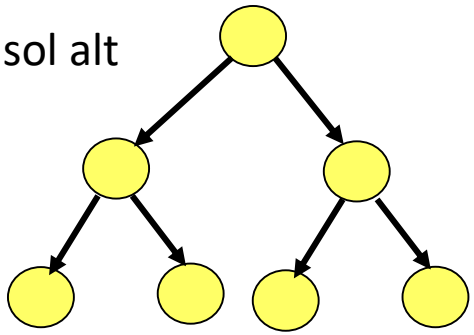
- Proper Tree



ImProper Tree

Full Binary Tree

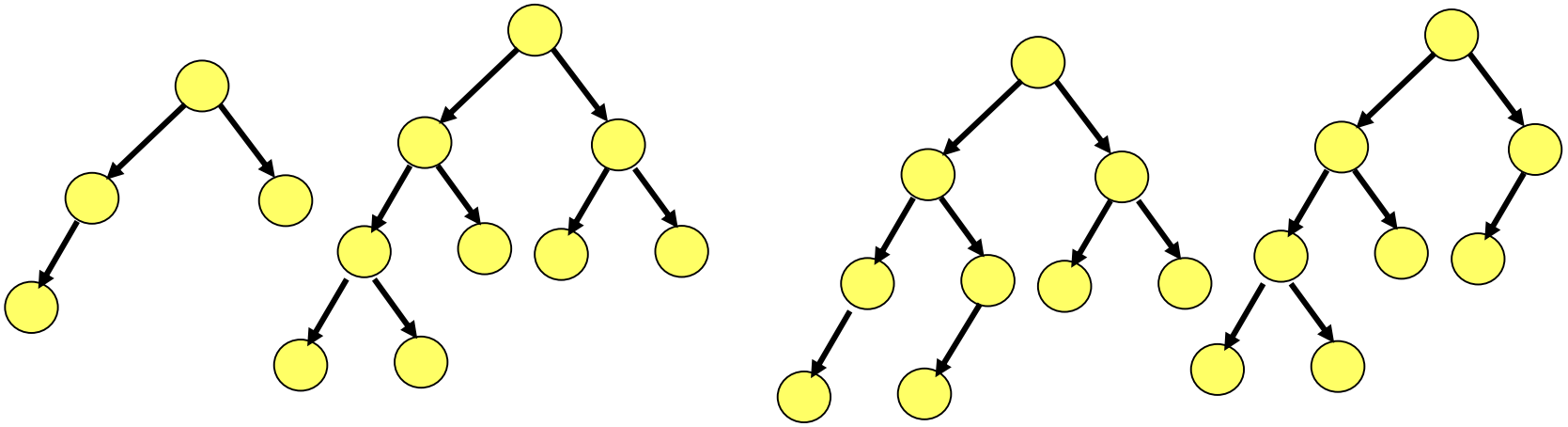
- Full binary tree:
  - **1-** Her yaprağı aynı derinlikte olan
  - **2-** Yaprak olmayan düğümlerin tümünün iki çocuğu olan ağaç **Full Binary Tree**'dir.
- Bir full binary tree'de **m** tane yaprak varsa bu ağaçta toplam **n=2m-1** düğüm vardır. Başka bir şekilde ifade edilirse,
  - Eğer **T** ağacı boş ise, **T** yüksekliği **0** olan bir full binary ağaçtır.
  - **T** ağacının yüksekliği **h** ise ve yüksekliği **h**'den küçük olan tüm node'lar iki child node'a sahipse, **T full binary** tree'dir.
  - Full binary tree'de her node aynı yüksekliğe eşit sağ ve sol alt ağaçlara sahiptir. En iyi bilinen proper tree ağacıdır.



Complete Binary Tree (Tamamlanmış İkili Ağaç)

- Full Binary Tree'de, yeni bir derinliğe soldan sağa doğru düğümler eklendiğinde oluşan ağaçlara **Complete Binary Tree** denilir.
- Böyle bir ağaçta bazı yapraklar diğerlerinden daha derindir. Bu nedenle Full Binary Tree olmayabilirler. En derin düzeyde düğümler olabildiğince soldadır. (Full Binary Tree aynı zamanda Complete binary tree dir)
-
  - **T**, **n** yükseklikte Complete Binary Tree ise, tüm yaprak node'ları n veya **n-1** derinliğindedir ve yeni bir derinliğe soldan sağa doğru ekleme başlanır.
  - Her node iki tane child node'a sahip olmayabilir.

Complete Binary Tree



- **Complete** **incomplete**
  - Her node iki tane child node’a sahip olmayabilir.

Örnek

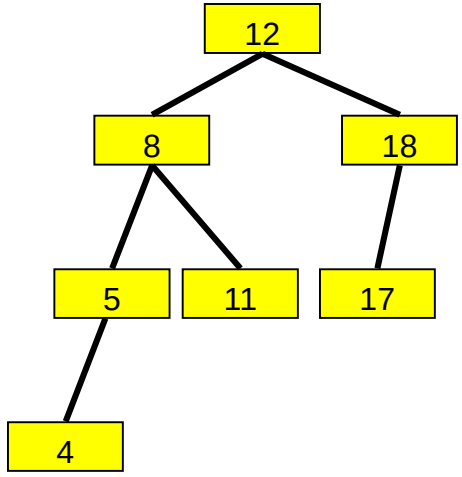
- Yüksekliği (h) 3 olan Tam (Full) ikili ağaçta minimum ve maksimum düğüm sayısı kaçtır?

Örnek

- Yüksekliği (h) 3 olarak verilen bir ağaçta sol alt ağaç ile sağ alt ağaç yükseklik farkı 1 ise Komple (Complete) ikili ağacın maksimum düğüm sayısı kaçtır?

Balanced Binary Trees

- Yüksekliği ayarlanmış ağaçlardır.
- Bütün düğümler için sol altağacın yüksekliği ile sağ altağacın yüksekliği arasında en fazla bir fark varsa balanced binary tree olarak adlandırılır.
- Complete binary tree'ler aynı zamanda balanced binary tree'dir.
- Her balanced binary tree, complete binary tree olmayabilir. (Neden?)

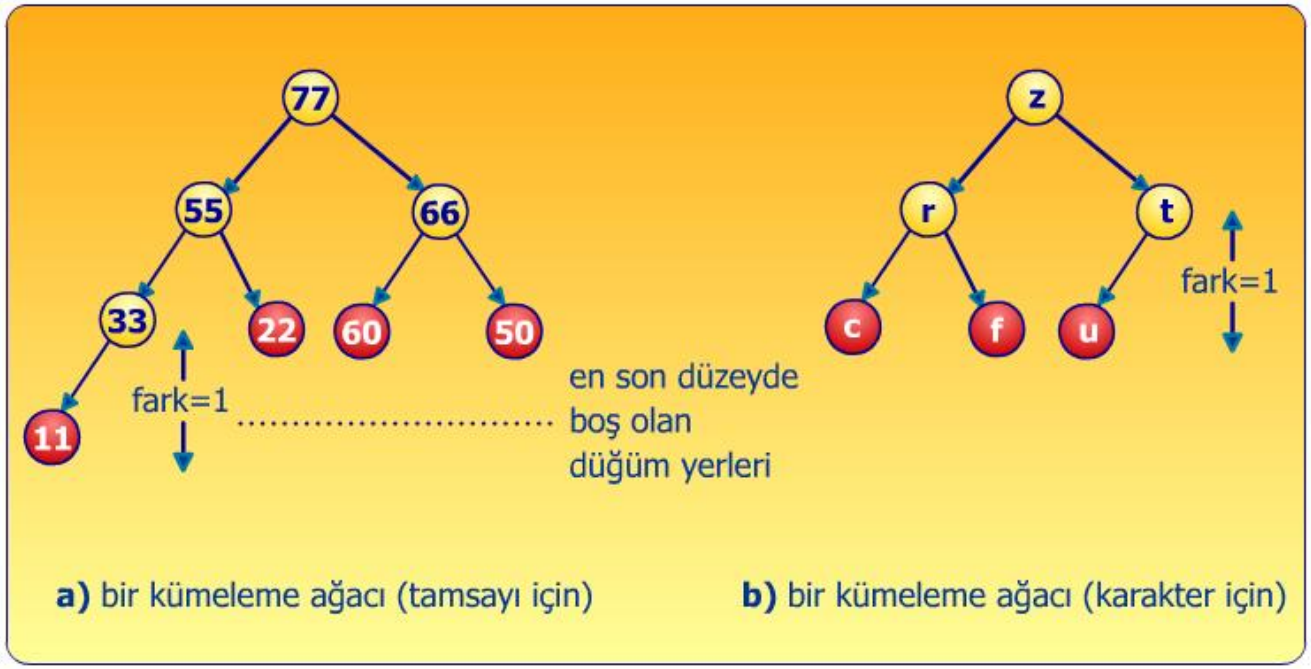


Heap Tree (Kümele Ağacı)

- Kümeleme ağacı, ağaç oluşturulurken değerleri daha büyük olan düğümlerin yukarıya, küçük olanların da aşağıya eklenmesine dayanan tamamlanmış bir ağaçtır.
- Böylece 0. düzeyde, kökte, en büyük değer bulunurken yaprakların olduğu *k*. düzeyde en küçük değerler olur. Yani büyükten küçüğe doğru bir kümeleme vardır; veya tersi olarak küçükten büyüğe kümeleme de yapılabilir.
- Aksi belirtilmediği sürece kümeleme ağacı, sayısal veriler için büyükten küçüğe, sözcükler için alfabetik sıralamaya göre yapılır.
- Kümeleme ağacı bilgisayar biliminde bazı problemlerin çözümü için çok uygundur; hatta birebir uyuşmaktadır denilebilir. Üstelik, hem bellek alanı hem de yürütme zamanı açısından getirisi olur.

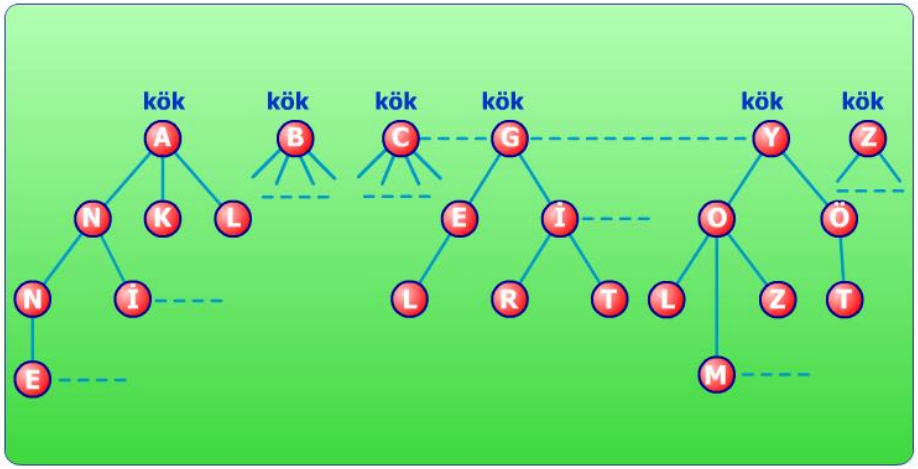
Heap Tree (Kümele Ağacı)

- Bir düğüm her zaman için çocuklarından daha büyük değere sahiptir.
- Yaprak düğümlerin herhangi ikisi arasındaki fark en fazla 1 dir.
- En son düzey hariç tüm düzeyler düğümlerle doludur.
- En son düzeyde oluşan boşluklar sadece sağ taraftadır
- Sıralama işlemlerinde kullanılır.



Trie Ağacı/Sözlük Ağacı

- Sözlük ağacı, bir sözlükte bulunan sözcükleri tutmak ve hızlı arama yapabilmek amacıyla düşünülmüştür; bellek gereksinimi arttırmadan, belki de azaltarak, onbinlerce, yüzbinlerce sözcük bulunan bir sözlükte 10-15 çevrim yapılarak aranan sözcüğün bulunması veya belirli bir karakter kadar uyuşanının bulunması için kullanılmaktadır.
- Sözlük ağacı, sıkıştırma, kodlama gibi sözlük kurulmasına dayalı algoritmalarda ve bir dilin sözlüğünü oluşturmada kullanılmaktadır.



Ödev

- Aşağıda verilen ağaç yapılarını araştırınız. Bu ağaçlara ait bilgileri Word ve Powerpoint ortamında hazırlayıp getiriniz.
  - Sözlük Ağaçları
  - Kodlama Ağaçları
  - Sıkıştırma Ağaçları
  - Bağıntı Ağaçları
  - Küme Ağacı

Geçiş İşlemleri

İkili Ağaçlar Üzerindeki Geçiş İşlemleri

- Geçiş (traverse), ağaç üzerindeki tüm düğümlere uğrayarak gerçekleştirilir. Ağaçlar üzerindeki geçiş işlemleri, ağaçtaki tüm bilgilerin listelenmesi veya başka amaçlarla yapılır.

- Doğrusal veri yapılarında baştan sona doğru dolaşmak kolaydır. Ağaçlar ise düğümleri doğrusal olmayan veri yapılarıdır. Bu nedenle farklı algoritmalar uygulanır.

İkili Ağaçlar Üzerindeki Geçiş İşlemleri

- **Preorder (depth-first order) Dolaşma (Traversal) (Önce Kök)**
  - Köke uğra (visit)
  - Sol alt ağacı preorder olarak dolaş.
  - Sağ alt ağacı preorder olarak dolaş.
- **Inorder (Symmetric order) Dolaşma(Ortada Kök)**
  - Sol alt ağacı inorder'a göre dolaş
  - Köke uğra (visit)
  - Sağ alt ağacı inorder'a göre dolaş.
- **Postorder Dolaşma (Sonra Kök)**
  - Sol alt ağacı postorder'a göre dolaş
  - Sağ alt ağacı postorder'a göre dolaş.
  - Köke uğra (visit)
- **Level order Dolaşma (Genişliğine dolaşma)**
  - Köke uğra
  - Soldan sağa ağacı dolaş

Preorder Geçişi

- Preorder geçişte, bir düğüm onun neslinden önce ziyaret edilir.
- Uygulama: yapılandırılmış bir belgenin yazdırılması

Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)

```
OnceKok(IkiliAgacDugumu kok)
{
    if (kok == null) return;
    System.out.print(kok.veri+" ");
    OnceKok(kok.sol);
    OnceKok(kok.sag);
}
```

Postorder Geçişi

- Postorder geçişte, bir düğüm onun neslinden sonra ziyaret edilir.
- Uygulama: Bir klasör ve onun alt klasörlerindeki dosyaların kullandıkları alanın hesaplanması.

```
Algorithm postOrder(v)
    for each child w of v
        postOrder (w)
    visit(v)
```

```
SonraKok(IkiliAgacDugumu kok)
{
    if (kok == null) return;
    SonraKok(kok.sol);
    SonraKok(kok.sag);
    System.out.print(kok.veri+" ");
}
```

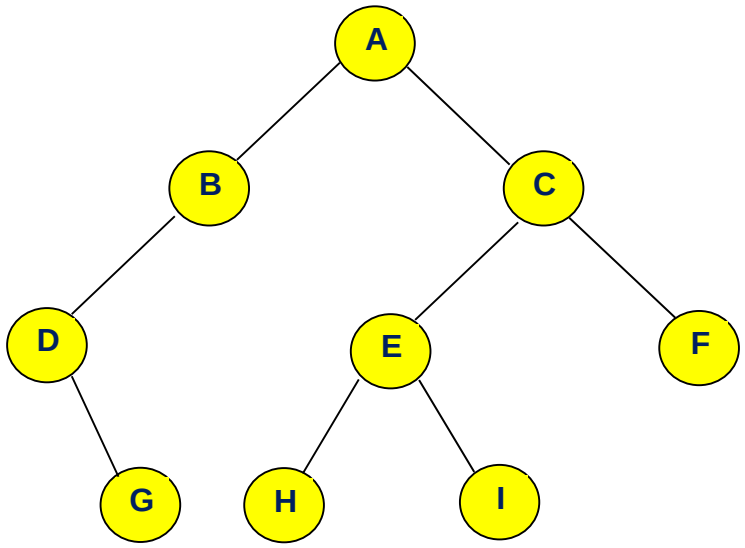
Inorder Geçişi

- Inorder geçişte, bir düğüm sol alt ağaçtan sonra ve sağ alt ağaçtan önce ziyaret edilir.
- Uygulama: ikili ağaç çizmek
- x(v) = v düğümünün inorder sıralaması (rank)
- y(v) = v'nin derinliği




```
OrtadaKok (IkiliAgacDugumu kok)
{
    if (kok == null) return;
    OrtadaKok(kok.sol);
    System.out.print(kok.veri+" ");
    OrtadaKok(kok.sag);
}
```

İkili Ağaçlar Üzerindeki Geçiş İşlemleri



PreOrder : A B D G C E H I F

InOrder : D G B A H E I C F

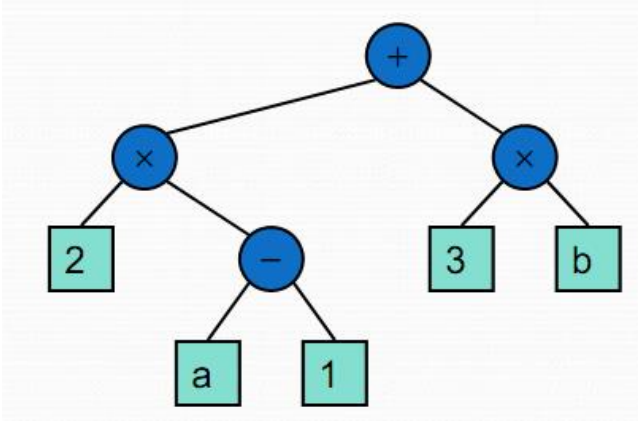
PostOrder : G D B H I E F C A

LevelOrder: A B C D E F G H I

Bağıntı (İfade) Ağaçları (Expression Tree)

- Bağıntı ağaçları bir matematiksel bağıntının ağaç şeklinde tutulması için tanımlanmıştır. Bir aritmetik ifade ile ilişkili ikili ağaçtır.
- Ağacın genel yapısı:
  - Yaprak düğüm = değişken/sabit değer
  - Kök veya ara düğümler = operatörler
  - Birçok derleyicide kullanılır. Parantez gereksinimi yoktur.

- Örnek: (2 × (a − 1) + (3 × b)) aritmetik ifadesi için ağaç yapısı

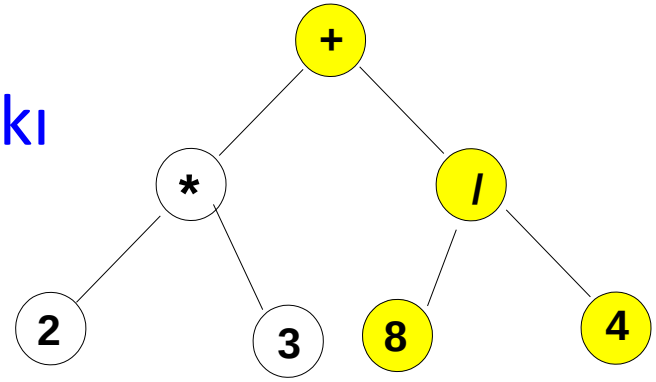


Bağıntı (İfade) Ağaçları (Expression Tree)

- Verilen denklemden bağıntı ağacı kurulması veya verilen ağaçtan denklemin çıkartılması için üç değişik yöntem vardır:
  - İç Ek(Infix): matematikte alışılagelen şekilde, operatörlerin ortada, değişken veya sabit değerlerin operatörün kenarında bulunan yöntemdir.
  - Ön Ek (Prefix): operatörler kendilerine karşı düşen parametrelerin(değişken/sabit) önündedir.
  - Son Ek(Postfix): Polonyalı notasyonu olarak da adlandırılır ve operatörler kendi parametrelerinin arkasından gelir.

Bağıntı Ağaçlarında Geçiş

- preOrder, (prefix)- Ön-takı
  - + * 2 3 / 8 4
- inOrder, (infix)- İç-takı
  - 2 * 3 + 8 / 4
- postOrder, (postfix) Son-takı
  - 2 3 * 8 4 / +
- levelOrder,
  - + * / 2 3 8 4

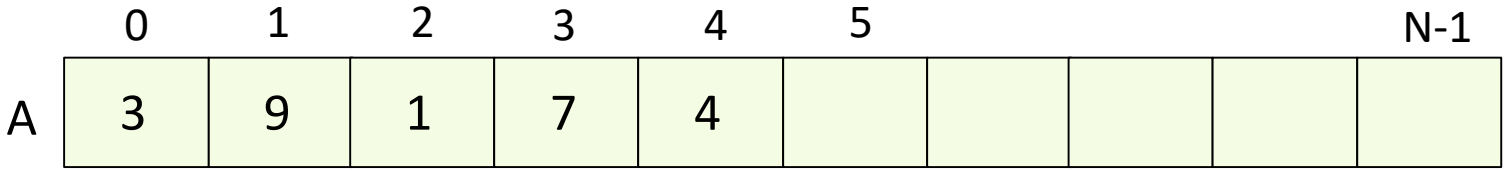


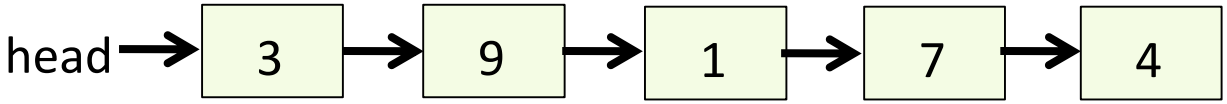
Arama Ağaçları

- Bir veri yapısı içerisine çok sayıda (anahtar, değer) çiftleri saklamak istediğimizi varsayalım.
- Aşağıdaki işlemleri etkili bir şekilde yerine getirebilecek bir veri yapısına ihtiyacımız var.
  - Ekle(anahtar, değer)
  - Sil(anahtar, değer)
  - Bul(anahtar)
  - Min()
  - Max()
- Alternatif veri yapıları?
  - Dizi kullanmak
  - Bağlantılı liste kullanmak

Arama Ağaçları

Örnek: Yandaki değerleri saklayalım: 3, 9, 1, 7, 4





Operasyon	Sırasız Dizi	Sıralı Dizi	Sırasız Liste	Sıralı List
Bul (Arama)	O(N)	O(logN)	O(N)	O(N)
Ekle	O(1)	O(N)	O(1)	O(N)
Sil	O(1)	O(N)	O(1)	O(1)

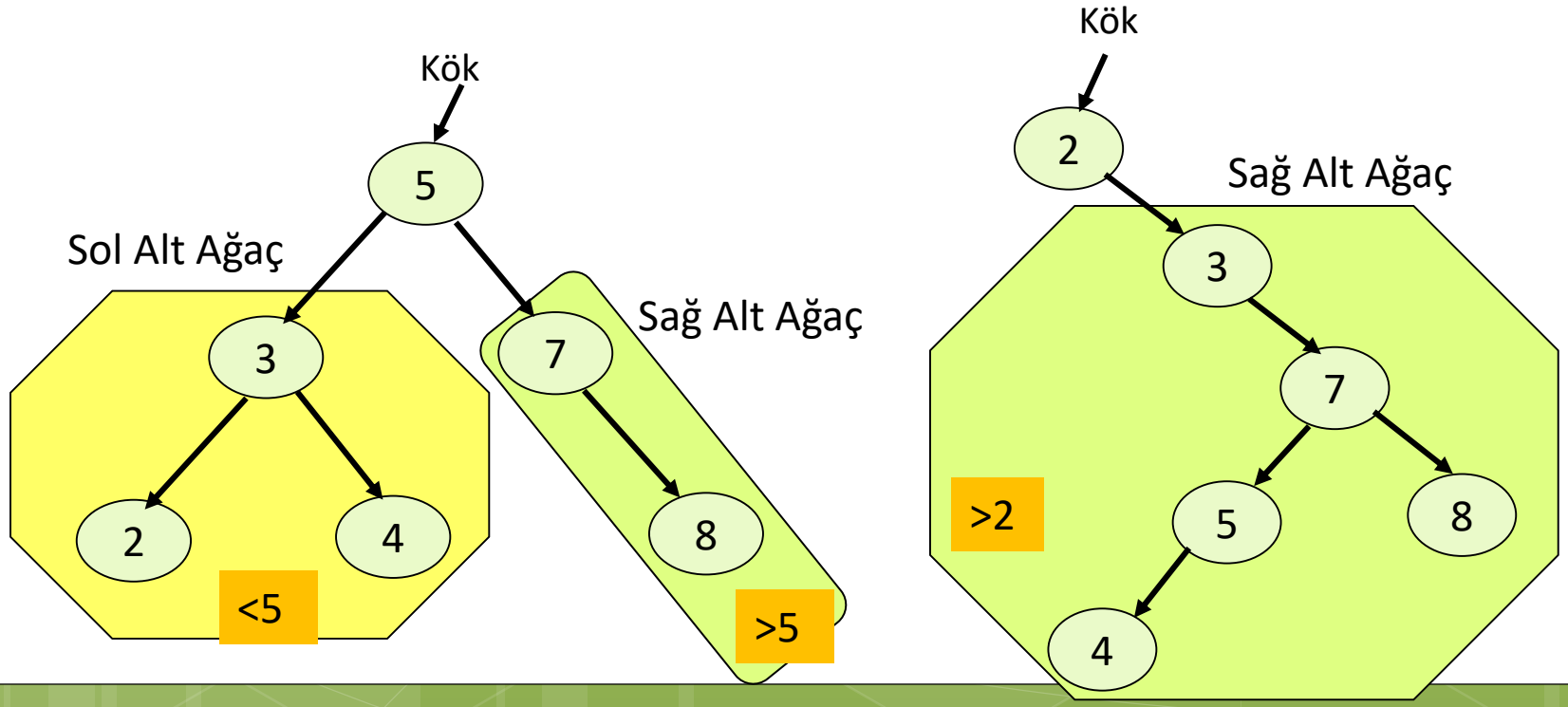
Bul/Ekle/Sil işlemlerinin hepsini O(logN) de yapabilir miyiz?

Kullanılan Verimli Arama Ağaçları

- **Fikir:** Verileri arama ağacı yapısına göre düzenlersek arama işlemi daha verimli olacaktır.
  - İkili Arama Ağacı (**B**inary **S**earch **T**ree (**BST**))
  - AVL Ağacı
  - Splay Ağacı
  - 2-3-4 Ağacı
  - Red-Black Ağacı
  - B Ağacı ve B+ Ağacı

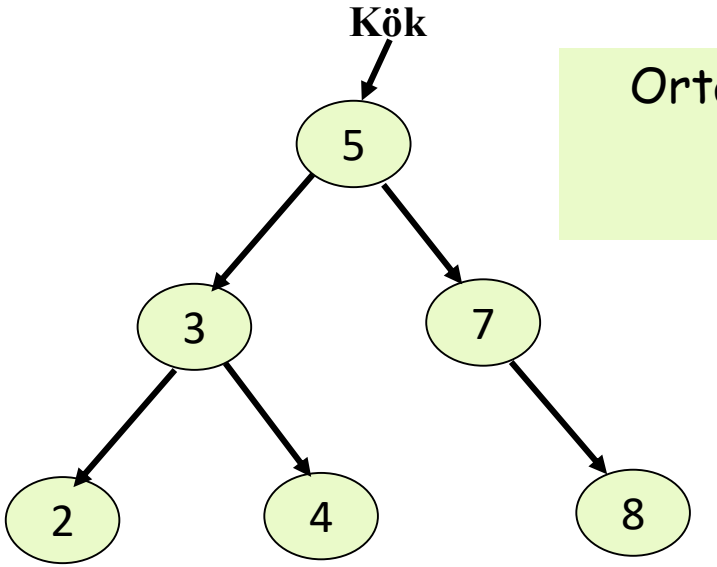
İkili Arama Ağacı (Binary Search Tree)

- İkili Arama Ağacı her bir düğümdeki değerlere göre düzenlenir:
  - Sol alt ağaçtaki tüm değerler kök düğümünden küçüktür.
  - Sağ alt ağaçtaki tüm değerler kök düğümünden büyüktür.



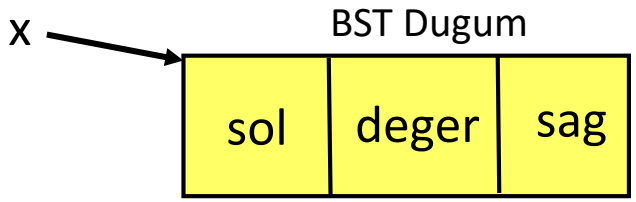
İkili Arama Ağacında Sıralama

- İkili arama ağacı önemli özelliklerinden birisi Ortada-kök (Inorder) dolaşma algoritması ile düğümlere sıralı bir şekilde ulaşılmasını sağlar.



Ortada-kök sonucu
2 3 4 5 7 8

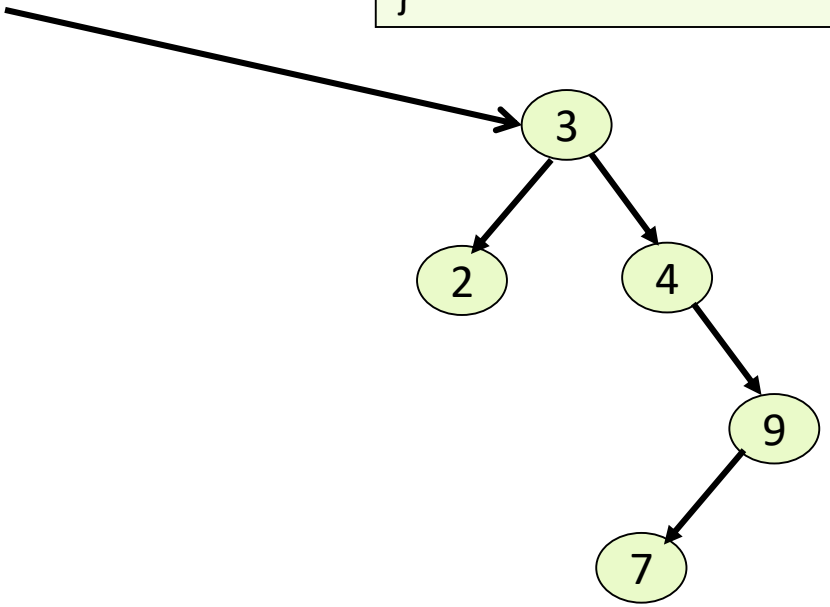
BST İşlemleri- Tanımlama




```
public class BSTDugum
{
  public BSTDugum sol;
  public int deger;
  public BSTDugum sag;
}
```

```
/* İKİLİ ARAMA AĞACI */
public class BST {
  Private BSTDugum kok;

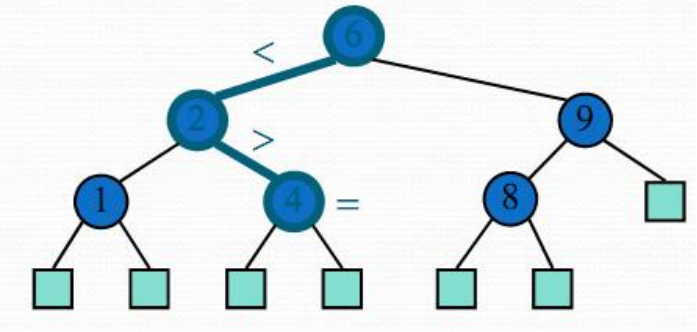
  public BST(){kok=null;}
  public void Ekle(int deger);
  public void Sil(int deger);
  public BSTNode Bul(int key);
  public BSTNode Min();
  public BSTNode Max();
};
```



BST İşlemleri-Arama

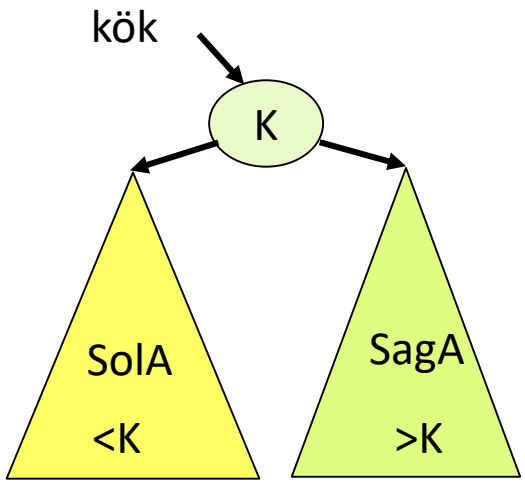
- Bir k anahtarını aramak için, kök düğümden başlayarak aşağı doğru bir yol izlenir.
- Bir sonraki ziyaret edilecek düğüm, k anahtar değerinin geçerli düğümün anahtar değeriyle karşılaştırılması sonucuna bağlıdır.
- Eğer yaprağa ulaşıldıysa anahtar bulunamamıştır ve null değer geri döndürülür.
- Örnek: Bul(4)

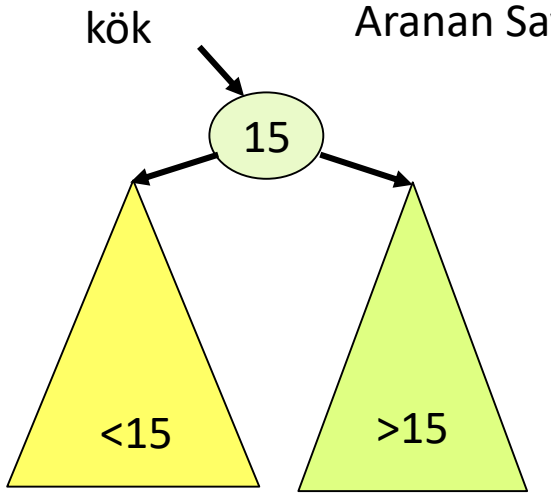
```
Algorithm TreeSearch(k, v)
    if T.isExternal (v)
        return v
    if k < key(v)
        return TreeSearch(k, T.left(v))
    else if k = key(v)
        return v
    else { k > key(v) }
        return TreeSearch(k, T.right(v))
```



BST İşlemleri- Arama

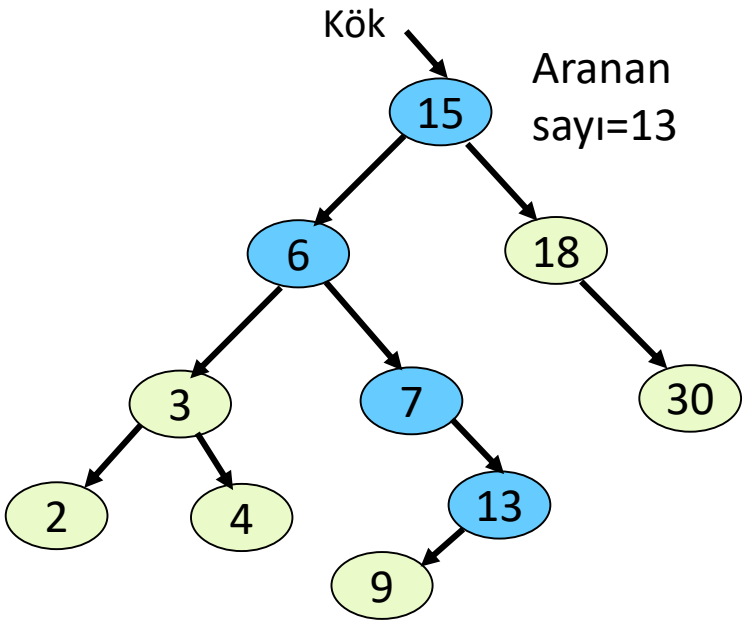
- Değeri içeren düğümü bul ve bu düğümü geri döndür.






```
1.  Arama işlemine kökten başla
2.  if (aranaDeger == kok.deger)      return kok;
3.  if (aranaDeger < kok.deger)       Ara SolAltAğaç
4.  else                              Ara SagAltAğaç
```

BST İşlemleri- Arama




```
public BSTDugum Bul(int deger)
{  return Bul2(kok, deger); }
```

```
public BSTDugum Bul2(BSTDugum kok, int deger)
{
  if (kok == null) return null;
  if (deger == kok.deger)
    return kok;
  else if (deger < kok.deger)
    return Bul2(kok.sol, deger);
  else /* deger > kok.deger */
    return Bul2(kok.sag, deger);
}
```

Mavi renkli düğümler arama sırasında ziyaret edilen düğümlerdir.

Algoritmanın çalışma karmaşıklığı O(d)'dir. (d = ağacın derinliği)

BST İşlemleri-Arama

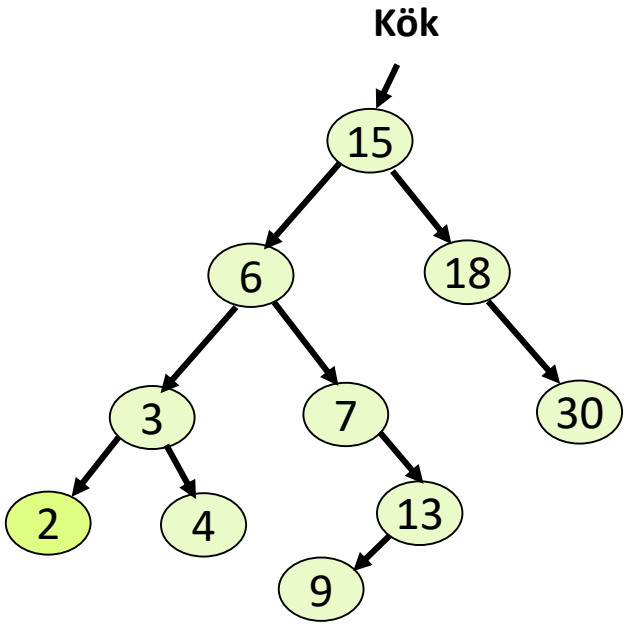
- Aynı algoritma while döngüsü yardımıyla yinelemeli şekilde yazılabilir. Yinelemeli versiyon özyinelemeli versiyona göre daha verimli çalışır.

```
public BSTDugum Bul(int deger){
  BSTDugum p = kok;
  while (p){
    if (deger == p.deger)      return p;
    else if (deger < p.deger)  p = p.sol;
    else /* deger > p.deger */  p = p.sag;
  } /* while-bitti */
  return null;
} //bul-Bitti
```

BST İşlemleri- Min

- Ağaçtaki en küçük elemanı içeren düğümü bulur ve geri döndürür.
  - En küçük elemanı içeren düğüm en soldaki düğümde bulunur.
  - Kökten başlayarak devamlı sola gidilerek bulunur.

```
public BSTDugum Min()
{
  if (kok == null)
    return null;
  BSTDugum p = kok;
  while (p.sol != null){
    p = p.sol;
  }
  return p;
}
```



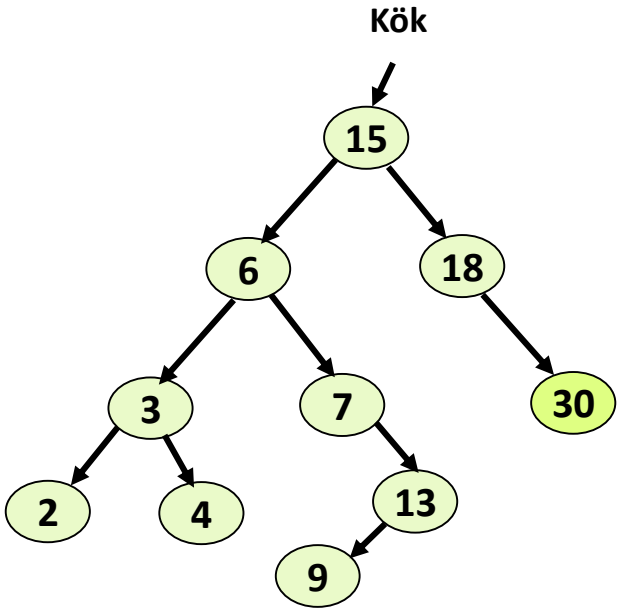
BST İşlemleri-Max

- Ağaçtaki en büyük elemanı içeren düğümü bulur ve geri döndürür.
  - En büyük elemanı içeren düğüm en sağdaki düğümde bulunur.
  - Kökten başlayarak devamlı sağa gidilerek bulunur.

```
public BSTDugum Max(){
  if (kok == null)
    return null;

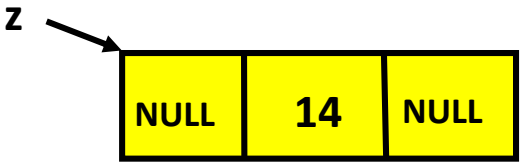
  BSTDugum p = kok;
  while (p.sag != null){
    p = p.sag;
  }

  return p;
}
```

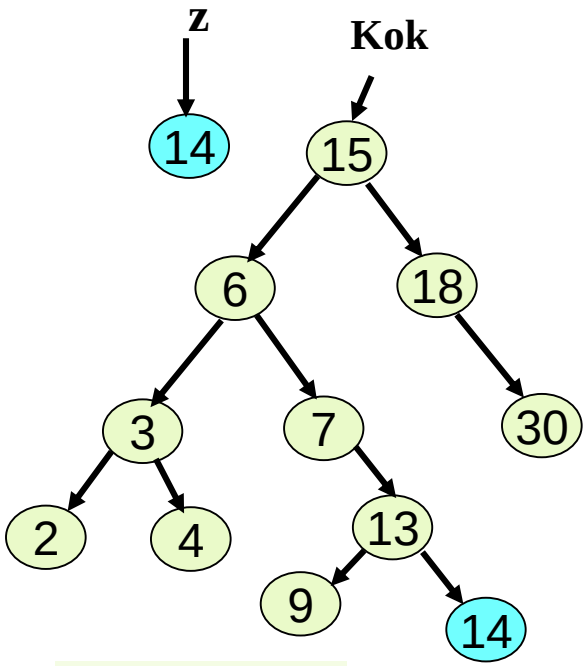


BST İşlemleri– Ekle(int deger)

- Eklenecek değeri içeren “z” isimli yeni bir düğüm oluştur.
- Ö.g.: Ekle 14
- Kökten başlayarak ağaç üzerinde eklenecek sayıyı arıyormuş gibi aşağıya doğru ilerle.
- Yeni düğüm aramanın bittiği düğümün çocuğu olmalıdır.



Eklenecek “z” düğümü.
z.deger = 14



Eklemeden sonra

BST İşlemleri- Ekle(int deger)

```
public void Ekle(int deger) {
  BSTDugum pp = null;  /* pp p'nin ailesi */
  BSTDugum p = kok;   /* Kökten başla ve aşağıya doğru ilerle*/
  while (p) {
    pp = p;
    if (deger == p.deger)      return;  /* Zaten var */
    else if (deger < p.deger)    p = p.sol;
    else /* deger > p.deger */ p = p.sag;
  }
 /* Yeni değeri kaydedeceğimiz düğüm */
  BSTDugum z = new BSTDugum();
  z.deger = deger; z.sol = z.sag = null;
  if (kok == null) kok = z; /* Boş ağaca ekleme */
  else if (deger < pp.deger) pp.sol = z;
  else pp.sag = z;
} // ekleme işlemi bitti.
```

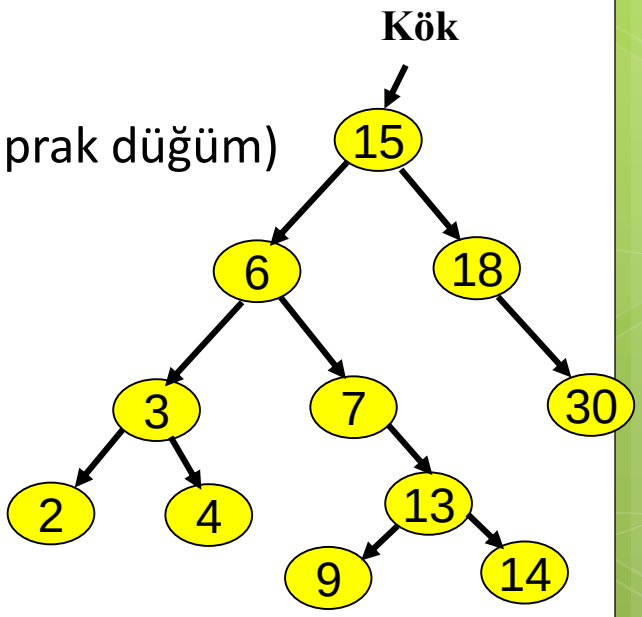
BST-Silme Kaba Kod

- while(silinecek düğümün ve ailesinin adresini bulana kadar) {
- q <- silinecek düğümün, qa<- ailesinin adresi;
- if(silinmek istenen bulunamadı ise)
  - yapacak bir şey yok dön;
- if(silinecek düğümüm iki alt çocuğu da varsa)
  - sol alt ağacın en büyük değerli düğümünü bul;
  - (veya denge bozulmuş ise sağ alt ağacın en küçük değerli düğümünü bul)
  - bu düğümdeki bilgiyi silinmek istenen düğüme aktar;
  - bu aşamada en fazla bir çocuğu olan düğümü sil;
- silinen düğümün işgal ettiği bellek alanını serbest bırak;
  }

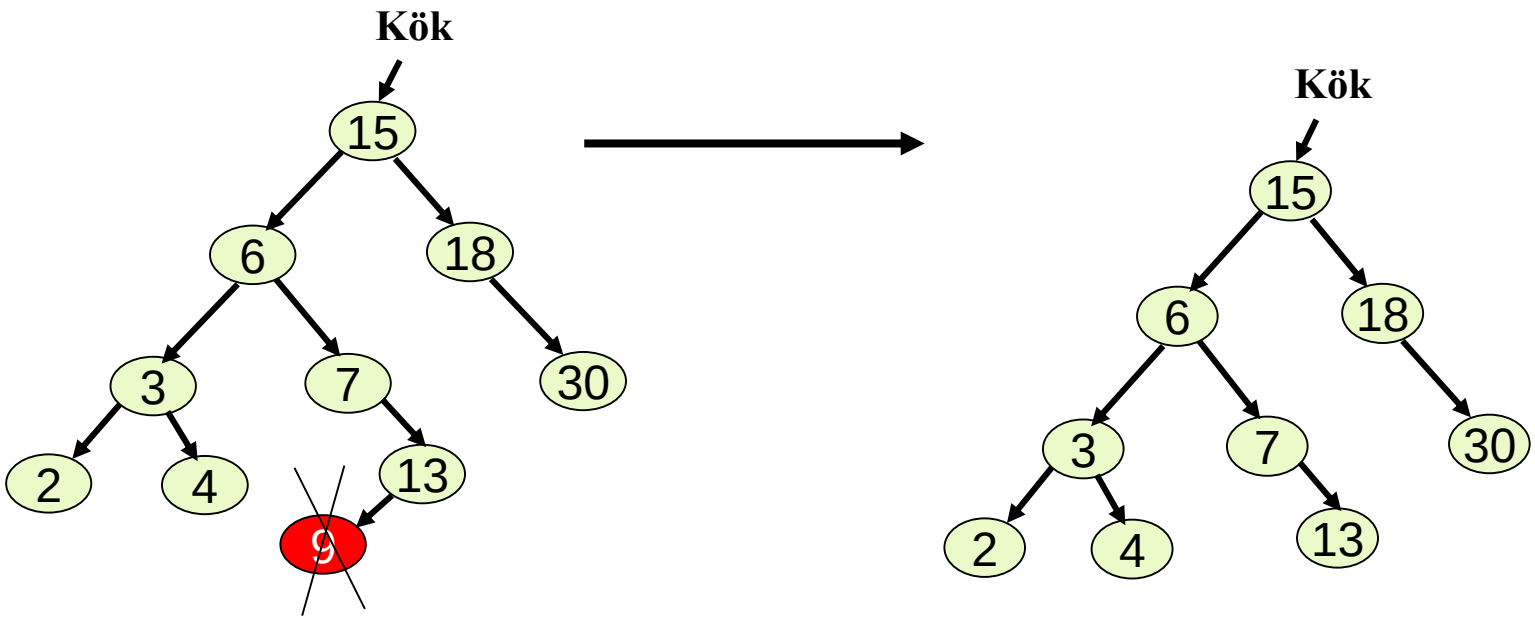
BST İşlemleri- Sil(int deger)

- Silme işlemi biraz karmaşıktır.
- 3 durum var:

1) Silinecek düğümün hiç çocuğu yoksa (yaprak düğüm)
   - Sil 9
2) Silinecek düğümün 1 çocuğu varsa
   - Sil 7
3) Silinecek düğümün 2 çocuğu varsa
   - Sil 6



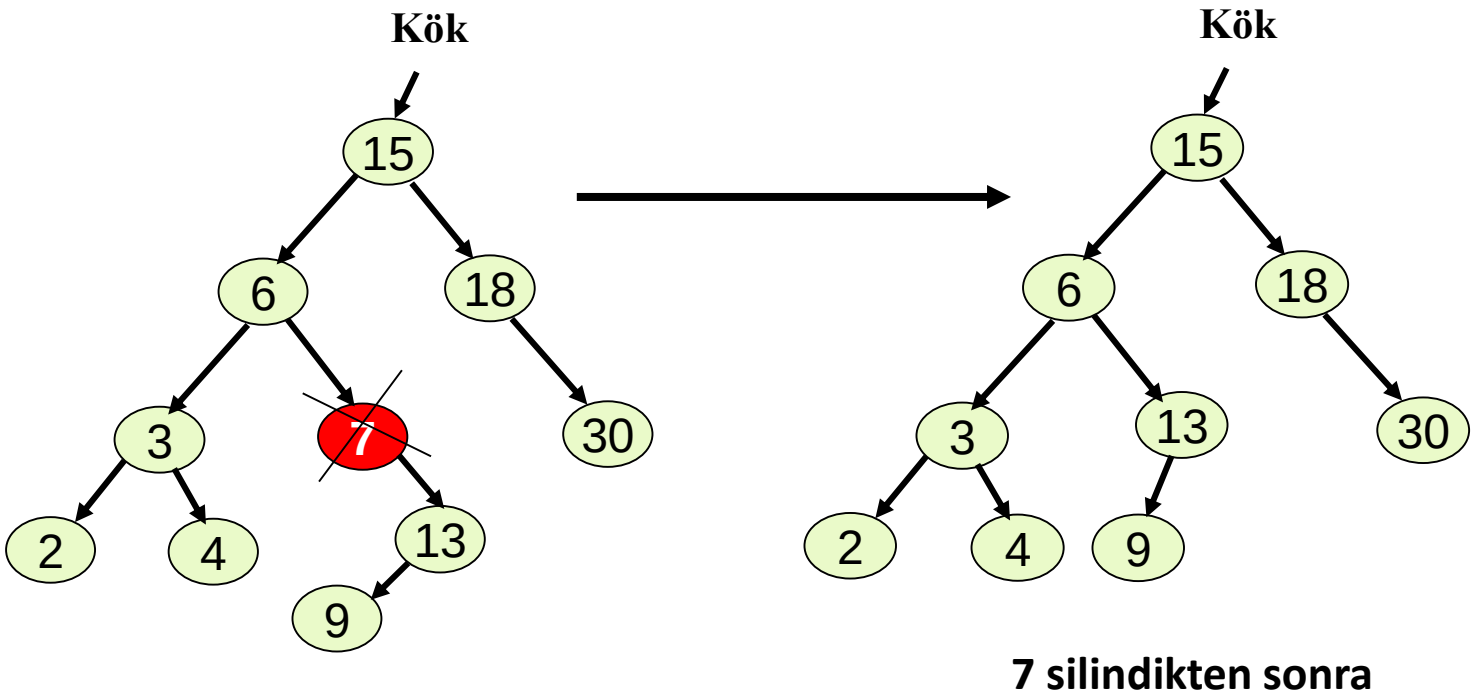
Silme: Durum 1 – Yaprak Düğümü Silme



Sil 9: Düğümü kaldırın ve bağlantı kısmını güncelleyin

9 silindikten sonra

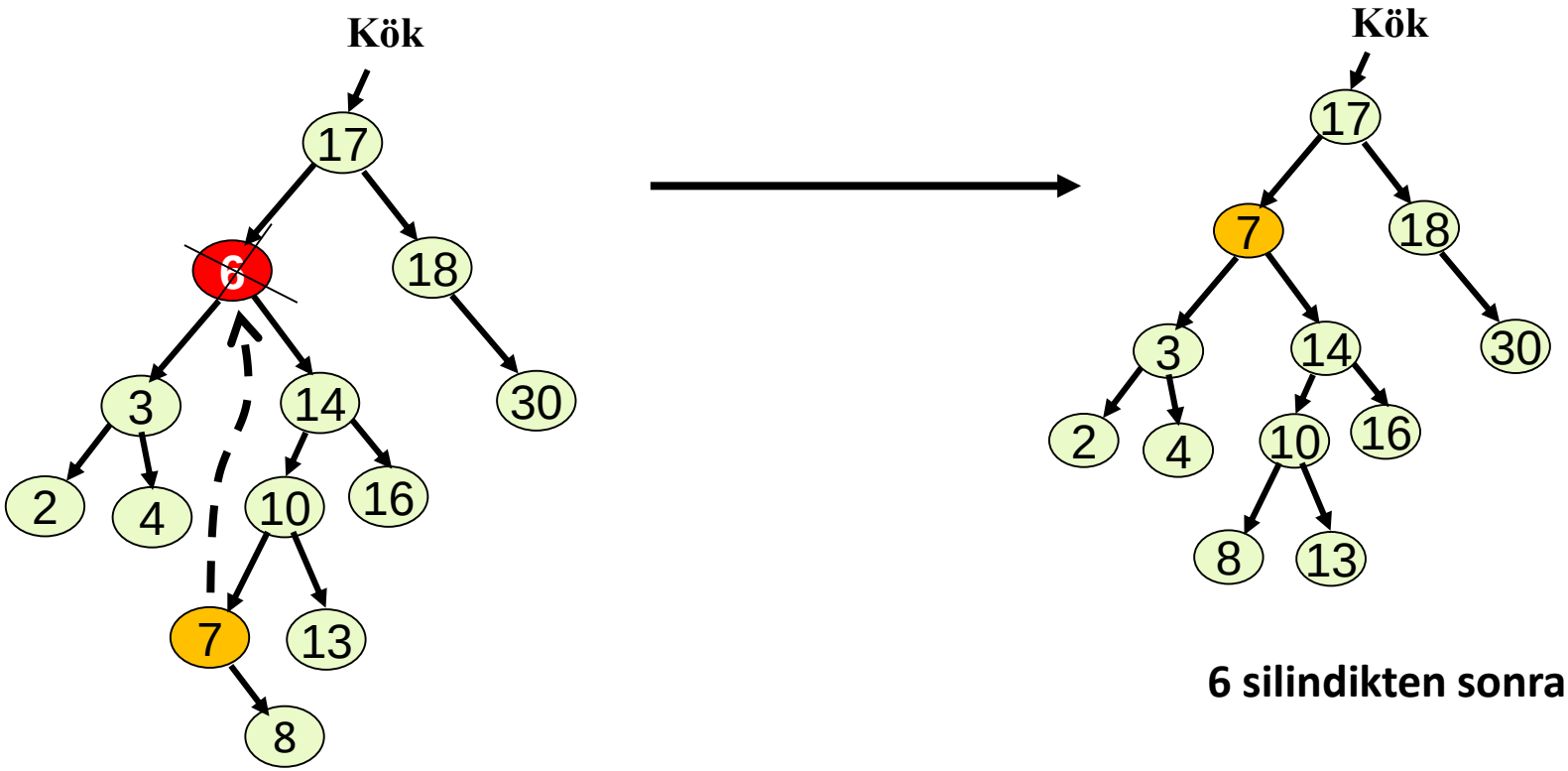
Silme: Durum 2 – 1 Çocuklu Düğüm



7 silindikten sonra

Sil 7: Silinecek düğümün ailesi ve çocuğu arasında bağ kurulur

Silme: Durum 3 – 2 Çocuklu Düğüm



6 silindikten sonra

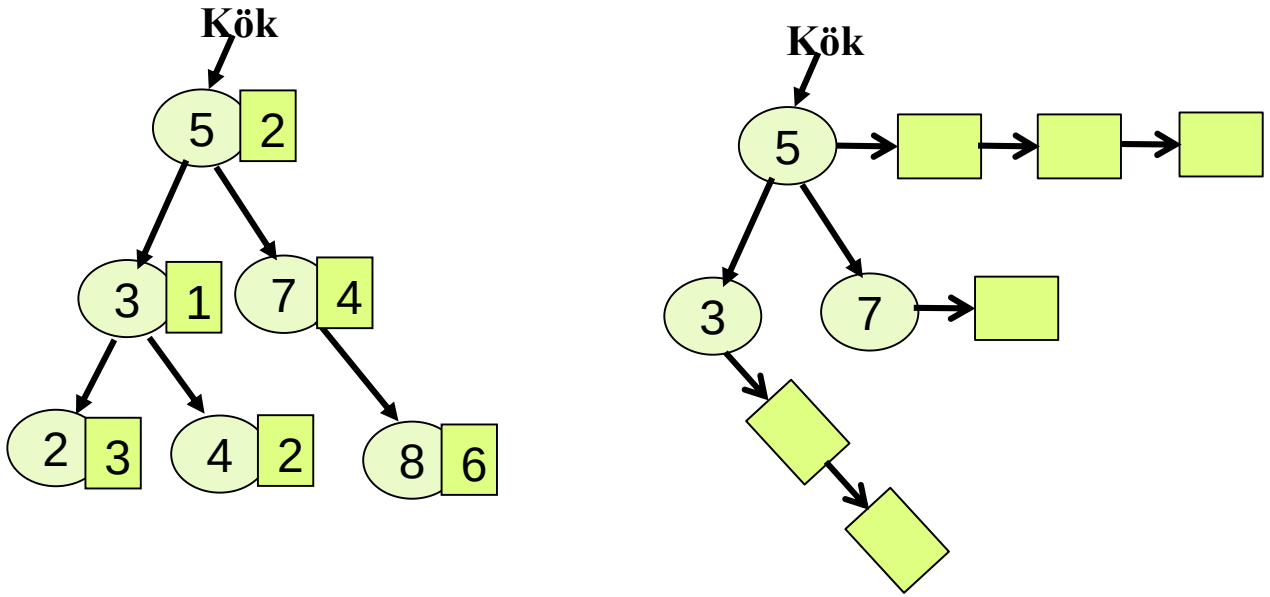
Sil 6:

1) **Sağ alt ağaçtaki en küçük eleman bulunur.(7)**
2) **Bu elemanın sol çocuğu olmayacaktır.**
3) **6 ve 7 içeren düğümlerin içeriklerini değiştirin**
4) **6 nolu eleman 1 çocuğu varmış gibi silinir.**

Not: Sağ alt ağaçtaki en küçük eleman yerine sol alt ağaçtaki en büyük eleman bulunarak aynı işlemler yapılabilir.

BST-Aynı Sayılarla Başa Çıkma

- Ağaç içerisindeki aynı sayılarla aşağıda verilen iki şeklide başa çıkılabilir:
  - Düğümde saklanan bir sayaç değişkeni ile
- veya
  - Düğümde kullanılan bağlantılı liste ile

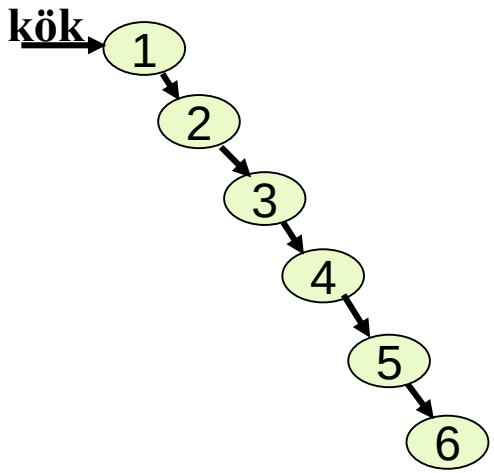


İkili Arama Ağacı Uygulamaları

- İkili arama ağacı harita, sözlük gibi birçok uygulamada kullanılır.
  - İkili arama ağacı (anahtar, değer) çifti şeklinde kullanılacak sistemler için uygundur.
    - Ö.g.: Şehir Bilgi Sistemi
      - Posta kodu veriliyor , şehir ismi döndürülüyor. (posta kodu/ Şehir ismi)
    - Ö.g.: telefon rehberi
      - İsim veriliyor telefon numarası veya adres döndürülüyor. (isim, Adres/Telefon)
    - Ö.g.: Sözlük
      - Kelime veriliyor anlamı döndürülüyor. (kelime, anlam)

İkili Arama Ağacı

- Bul, Min, Max, Ekle, Sil işlemlerinin karmaşıklığı O(d)
- d ağacın derinliğine bağlı.
- Örnek: 1,2,3,4,5,6 sayılarını sıralı bir şekilde ekleyelim.
- Ortaya çıkan ağaç bağlantılı listeye benzemektedir. Dolayısıyla karmaşıklık O(n) şeklinde olacaktır.



- Daha iyisi yapılabilir mi?
- Ağacımızı dengeli yaparsak evet
  1. AVL-ağaçları
  2. Splay ağaçları
  3. 2-3-4 Ağaçları
  4. Red-Black ağaçları
  5. B ağaçları, B+ ağaçları

Ödev

- **1-**İkili arama ağacını Oluşturma-Arama-Ekleme-Silme olaylarını bir bütün olarak gerçekleştiriniz. (Önerilen Ders kitabından yararlanabilirsiniz)
- **2-** İkili binary ağacında minimum ve maksimum değerin bulunduğu düğümün atasını ve kardeşlerini bulan metotları yazınız.
- **3-** İkili binary ağacını level-order dolaşmayı sağlayan metodu yazınız.
- **4-** İkili binary ağacında her düğümün sol ve sağ ağaçları arasındaki denge farklarını bularak bu farkın en çok olduğu düğümü bulunuz.

Örnek Programlar

İkili arama Ağacı–Java

```
class TreeNode  // Düğüm Sınıfı
 {
  public int data;
  public TreeNode leftChild;
  public TreeNode rightChild;
  public void displayNode()  {  System.out.print(" "+data+" ");  }
 }

 // Ağaç Sınıfı
 class Tree
 {
  private TreeNode root;
  public Tree()   { root = null; }

  public TreeNode getRoot()  {  return root;  }
```

İkili arama Ağacı–Java

```
// Ağacın preOrder Dolaşılması
public void preOrder(TreeNode localRoot)
{
 if(localRoot!=null)    {
  localRoot.displayNode();
  preOrder(localRoot.leftChild);
  preOrder(localRoot.rightChild);
 }
}

// Ağacın inOrder Dolaşılması
public void inOrder(TreeNode localRoot)
{
 if(localRoot!=null)
 {
  inOrder(localRoot.leftChild);
  localRoot.displayNode();
  inOrder(localRoot.rightChild);
 }
}
```

İkili arama Ağacı–Java

```
// Ağacın postOrder Dolaşılması
public void postOrder(TreeNode localRoot)
{
  if(localRoot!=null)
  {
    postOrder(localRoot.leftChild);
    postOrder(localRoot.rightChild);
    localRoot.displayNode();
  }
}

// Ağaca bir düğüm eklemeyi sağlayan metot
public void insert(int newdata)
{
  TreeNode newNode = new TreeNode();
  newNode.data = newdata;

if(root==null)
    root = newNode;
```

İkili arama Ağacı–Java

```
else    {
    TreeNode current = root;      TreeNode parent;
    while(true)     {
     parent = current;
     if(newdata<current.data)
     {  current = current.leftChild;
      if(current==null)
      {
       parent.leftChild=newNode;
       return;
      }
     } else
     {
      current = current.rightChild;
      if(current==null)
      {          parent.rightChild=newNode;          return;        }
     }
    }  // end while
   } // end else not root
  } // end insert()
} // class Tree
```

İkili arama Ağacı–Java

```
// BinTree Test sınıfı
class BinTree
{
  public static void main(String args[])
  {
    Tree theTree = new Tree();

    // Ağaca 10 tane sayı yerleştirilmesi
    System.out.println("Sayılar : ");
    for (int i=0;i<10;++i) {
      int sayi = (int) (Math.random()*100);
      System.out.print(sayi+" ");
      theTree.insert(sayi);
    };

    System.out.print("\nAğacın InOrder Dolaşılması   : ");
    theTree.inOrder(theTree.getRoot());
    System.out.print("\nAğacın PreOrder Dolaşılması  : ");
    theTree.preOrder(theTree.getRoot());
    System.out.print("\nAğacın PostOrder Dolaşılması : ");
    theTree.postOrder(theTree.getRoot());
  }
}
```

İkili arama Ağacı– C++

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
/* Ağaca ait düğüm yapısı tanımlanıyor */
struct agacdugum {
  struct agacdugum *soldal;
  int data;
  struct agacdugum *sagdal;
};
// Düğüm yapısı için değişken tanımlarının yapıldığı kısım
typedef struct agacdugum AGACDUGUM;
typedef struct agacdugum * AGACDUGUMPTR;
```

İkili arama Ağacı– C++

```
// Ağaca düğüm eklemeyi saglayan fonksiyon yapisi
AGACDUGUMPTR dugumekle(AGACDUGUMPTR agacptr, int veri) {
/* her defasında tek dallı ağaç oluşturuluyor. Daha sonra ikili arama ağacındaki kurala göre sol veya sağ dala yerleştiriliyor. */

 if(agacptr==NULL)
  {
/*eger agaç işaretçisi boş ise ağaca eklenecek yeni düğüm için hafızada yer ayrılıyor*/

    agacptr =(agacdugum *) malloc(sizeof(agacdugum));
   if (agacptr!=NULL)
   {
  // Düğümler tek hücre, sağ ve sol dalları boş olarak oluşturuluyor
  //  printf("Ağaça veri eklendi\n ");
     agacptr->data = veri;
     agacptr->soldal = NULL;
     agacptr->sagdal= NULL;
   }
   else printf("%d eklenemedi. Bellek yetersiz.\n",veri);
  }
```

İkili arama Ağacı– C++

```
/*Gelen veri değeri daha önce girilen değerler ile karşılaştırılıp
uygun düğümün sol veya sağ dalına yerleştiriliyor.*/
  else

    if(veri<agacptr->data){ printf("Ağaçın soluna veri eklendi\n ");
      agacptr->soldal = dugumekle(agacptr->soldal,veri);}
    else

    if(veri>agacptr->data){printf("Ağaçın sağına veri eklendi\n ");
      agacptr->sagdal = dugumekle(agacptr->sagdal,veri);}

// eğer girilen değer ler daha önce var ise alınmıyor.
    else printf("Eşit olduğu için alınmadı\n ");
    return agacptr;
}
```

İkili arama Ağacı– C++

```
/* Agacın inorder dolaşılması */
void inorder(AGACDUGUMPTR agacptr) {
 if (agacptr != NULL) {
   inorder(agacptr->soldal);
   printf("%3d",agacptr->data);
   inorder(agacptr->sagdal);   }            }
/* Ağacın preorder dolaşılması */
void preorder(AGACDUGUMPTR agacptr) {
 if (agacptr != NULL) {
   printf("%3d",agacptr->data);
   preorder(agacptr->soldal);
   preorder(agacptr->sagdal);  }            }
/* Ağacın postorder dolaşılması */
void postorder(AGACDUGUMPTR agacptr) {
 if (agacptr != NULL) {
   postorder(agacptr->soldal);
   postorder(agacptr->sagdal);
   printf("%3d",agacptr->data); }            }
```

İkili arama Ağacı– C++

```
void main() {
    int i, dugum;
    AGACDUGUMPTR agacptr = NULL;
    for(i=0; i<12; ++i)
     { /* Ağaca yerleştirilecek sayılar */
       scanf("%d",&dugum);  printf("\n");
    // girilen değeri düğüm ekleme fonksiyonuna gönderiyoruz.
       agacptr = dugumekle(agacptr, dugum);
    }  printf("\n");

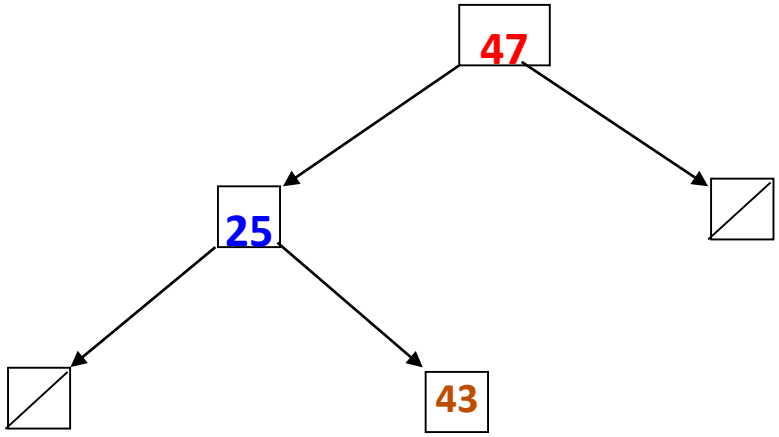
     printf("Ağacın preorder dolaşılması  :\n");
      preorder(agacptr); printf("\n");

      printf("Ağacın inorder dolaşılması   :\n");
      inorder(agacptr); printf("\n");

      printf("Ağacın postorder dolaşılması :\n");
      postorder(agacptr); printf("\n");
    }
```

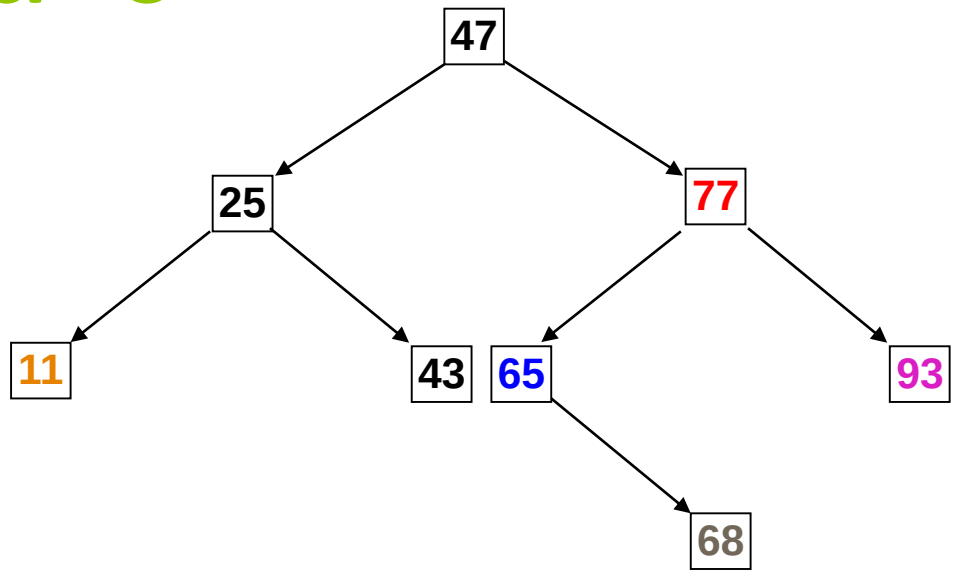
İkili arama Ağacı– C++

- 47
- 25
- Ağaçın soluna veri eklendi
- 43
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi



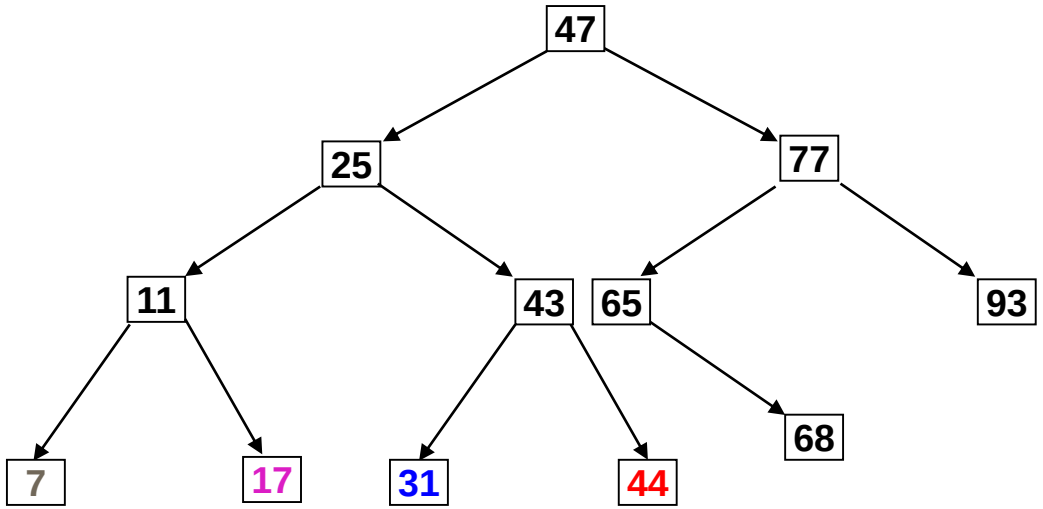
İkili arama Ağacı– C++

- 77
- Ağaçın sağına veri eklendi
- 65
- Ağaçın sağına veri eklendi
- Ağaçın soluna veri eklendi
- 68
- Ağaçın sağına veri eklendi
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi
- 93
- Ağaçın sağına veri eklendi
- Ağaçın sağına veri eklendi
- 11
- Ağaçın soluna veri eklendi
- Ağaçın soluna veri eklendi



İkili arama Ağacı– C++

- 17
- Ağaçın soluna veri eklendi
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi
- 44
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi
- Ağaçın sağına veri eklendi
- 31
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi
- Ağaçın soluna veri eklendi
- 7



İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
#include <stdio.h>
#include <stdlib.h>
struct tnode {
 int data;
 struct tnode *lchild, *rchild; };
/* Dugum degeri verilen dugumun kok pointer degerini elde eden bir foksiyondur*/
struct tnode *getptr(struct tnode *p, int key, struct tnode **y) {
printf("kok pointer\n");//silme anında devreye giriyor
 struct tnode *temp;
   if( p == NULL)      return(NULL);
   temp = p;    *y = NULL;
    while( temp != NULL)     {
      if(temp->data == key)   return(temp);
      else     {
         *y = temp; /* bu pointer root (kok) olarak depolanir */
   if(temp->data > key)
     temp = temp->lchild;
   else
     temp = temp->rchild;      }     }
    return(NULL);  }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* Veri degeri verilen dugumu silmek icin bir fonksiyon */
struct tnode *delete1(struct tnode *p,int val)  {
   struct tnode *x, *y, *temp;
   x = getptr(p,val,&y);
   if( x == NULL)   {     printf("Dugum mevcut degil\n");     return(p);    }
   else
   {
   /* bu kod kok (root) dugumu silmek icindir*/
   if( x == p)     {
     printf("kok dugum siliniyor\n");
      temp = x->lchild;
      y = x->rchild;
      p = temp;
      while(temp->rchild != NULL)     temp = temp->rchild;
    temp->rchild=y;
    free(x);
    return(p);
   }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* bu kod dugumun sahip oldugu cocukları silmek icindir.*/
 if( x->lchild != NULL && x->rchild != NULL)    {
      if(y->lchild == x)      {
temp = x->lchild;
y->lchild = x->lchild;
while(temp->rchild != NULL)    temp = temp->rchild;

temp->rchild=x->rchild;
x->lchild=NULL;
x->rchild=NULL;
      }
      else       {
        temp = x->rchild;
        y->rchild = x->rchild;
         while(temp->lchild != NULL)
     temp = temp->lchild;
   temp->lchild=x->lchild;
x->lchild=NULL;
 x->rchild=NULL;
       }
free(x);
   return(p);   }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* bu kod cocukları ile birlikte bir dugum siliyor*/
  if(x->lchild == NULL && x->rchild !=NULL)
    {
      if(y->lchild == x)
   y->lchild = x->rchild;
        else
          y->rchild = x->rchild;
        x->rchild=NULL;
        free(x);
        return(p);
      }
      if( x->lchild != NULL && x->rchild == NULL)
       {
        if(y->lchild == x)
          y->lchild = x->lchild ;
        else
          y->rchild = x->lchild;
        x->lchild = NULL;
        free(x);
        return(p);
            }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* bu kod cocukları olmadan bir dugumu siliyor*/
      if(x->lchild == NULL && x->rchild == NULL)
       {
         if(y->lchild == x)
           y->lchild = NULL ;
         else
           y->rchild = NULL;
         free(x);
         return(p);
       }
   }
}
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/*inorder binary agacını tekrarlamalı olarak yazdıran bir fonksiyon*/
void inorder1(struct tnode *p) {
 struct tnode *stack[100]; //yigin
 int top;
 top = -1;
if(p != NULL)   {
   top++;
   stack[top] = p;
   p = p->lchild;
   while(top >= 0)       {
        while ( p!= NULL)/* sol cocuk yigindan (stack dizisinden) cikariliyor*/
         {     top++;
              stack[top] =p;
              p = p->lchild;   }
    p = stack[top];
    top--;
    printf("%d\t",p->data);
    p = p->rchild;
    if ( p != NULL) /* sag cocuk cikariliyor*/
      {     top++;
      stack[top] = p;
         p = p->lchild;    }   }    }   }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* Olusturulan agaca yeni bir dugum ilave etmek icin olusturulan bir fonksiyon*/

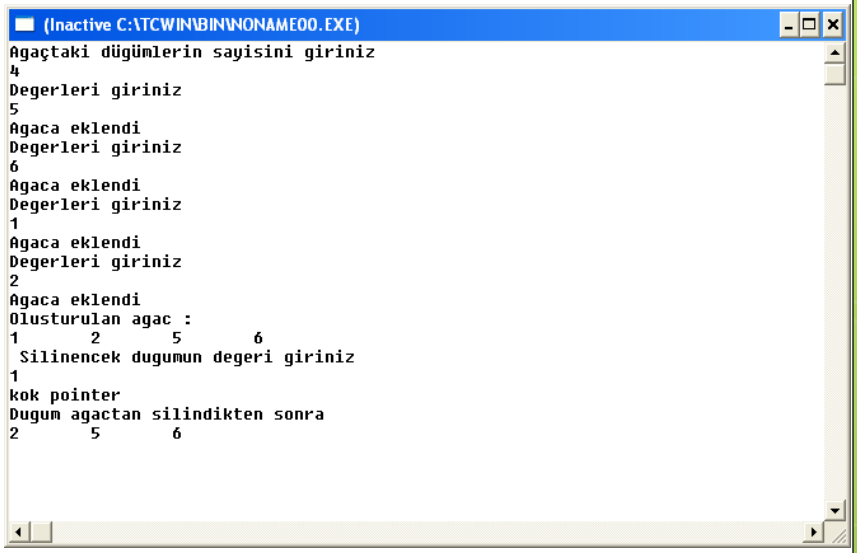
 struct tnode *insert(struct tnode *p,int val)  {
  printf("Agaca eklendi\n");
    struct tnode *temp1,*temp2;
    if(p == NULL)     {
      p = (struct tnode *) malloc(sizeof(struct tnode)); /* koke bir dugum ilave ediliyor*/
      if(p == NULL)         {
    printf("Erisim izni yok\n");
    exit(0);         }
      p->data = val;
      p->lchild=p->rchild=NULL;  }
  else
 {
    temp1 = p;
  /* child (cocuk) olacak dugumun pointer(gostergesini) almak icin agacta dolasma*/
 while(temp1 != NULL)
  {
   temp2 = temp1;
   if( temp1 ->data > val)      temp1 = temp1->lchild;
   else     temp1 = temp1->rchild;
  }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
if( temp2->data > val)   {
    temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode));/*Sol cocuk (left child) olarak yeni olusturulan dugumu ekler*/
    temp2 = temp2->lchild;
    if(temp2 == NULL)         {
    printf("Erisim izni yok\n");
    exit(0);         }
   temp2->data = val;
   temp2->lchild=temp2->rchild = NULL;   }
  else  {
     temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/*Sag cocuk (right child) olarak yeni olusturulan dugumu ekler*/
    temp2 = temp2->rchild;
    if(temp2 == NULL)         {
    printf("Erisim izni yok\n");
    exit(0);        }
   temp2->data = val;
   temp2->lchild=temp2->rchild = NULL;   }   }
  return(p);
  }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
void main()
 {
   struct tnode *root = NULL;
   int n,x;
   printf("Agaçtaki dügümlerin sayisini giriniz\n");
   scanf("%d",&n);
   while( n-->0)
     {
       printf("Degerleri giriniz\n");
       scanf("%d",&x);
       root = insert(root,x);
     }
    printf("Olusturulan agac :\n");
    inorder1(root);
    printf("\n Silinencek dugumun degeri giriniz\n");
    scanf("%d",&n);
    root=delete1(root,n);
    printf("Dugum agactan silindikten sonra \n");
    inorder1(root);
}
```



İkili arama Ağacı– C# (1. Örnek)

- **İkilik Arama Ağacı Oluşturmak**
- **class bstNodeC**
- **{**
- **public int sayi;**
- **public bstNodeC leftNode, rightNode;**
- **public bstNodeC(int sayi, bstNodeC leftNode, bstNodeC rightNode)**
- **{**
- **this.sayi = sayi;**
- **this.leftNode = leftNode;**
- **this.rightNode = rightNode;**
- **}**
- **}**

İkili arama Ağacı– C# (1. Örnek)

```
private void preOrder(bstNodeC node)
{
listBox1.Items.Add(node.sayi);
if (node.leftNode != null) preOrder(node.leftNode);
if (node.rightNode != null) preOrder(node.rightNode);
}
private void inOrder(bstNodeC node)
{
if (node.leftNode != null) inOrder(node.leftNode);
listBox1.Items.Add(node.sayi);
if (node.rightNode != null) inOrder(node.rightNode);
}
private void postOrder(bstNodeC node)
{
if (node.leftNode != null) postOrder(node.leftNode);
if (node.rightNode != null) postOrder(node.rightNode);
listBox1.Items.Add(node.sayi);
}
```

İkili arama Ağacı– C# (1. Örnek)

```
//Boşaltma (makeEmpty)
private void makeEmpty(bstNodeC node)
{
if (node.leftNode node.leftNode != null)  makeEmpty(node.leftNode );
if (node.rightNode node.rightNode != null) makeEmpty(node.rightNode);
node = null;
}
//Arama (find)
private void find(int sayi, bstNodeC node)
{
if (node.sayi == 0) textBox2.Text = "Boş ağaç";
else if ((sayi < node.sayi)&&(node.leftNode!=null))
find(sayi, node.leftNode);
else if ((sayi > node.sayi)&&(node.rightNode!=null))
find(sayi, node.rightNode);
else if (sayi == node.sayi) textBox2.Text = “Sayı bulundu";
else textBox2.Text = “Sayı yok";
}
```

İkili arama Ağacı– C# (1. Örnek)

```
//Ekleme (append)
private void append(int sayi, bstNodeC node) {
bstNodeC yeniNode = new bstNodeC(sayi, null, null);
if (node.sayi == 0) node.sayi = sayi;
else  {
bstNodeC current = node; bstNodeC parent;
while(true)
{   parent = current;
if(sayi < current.sayi)
{ current = current.leftNode;
if(current == null)  {parent.leftNode = yeniNode;break; }
}
else
{ current = current.rightNode;
if(current==null) { parent.rightNode = yeniNode; return; }
}
}
}
}
```

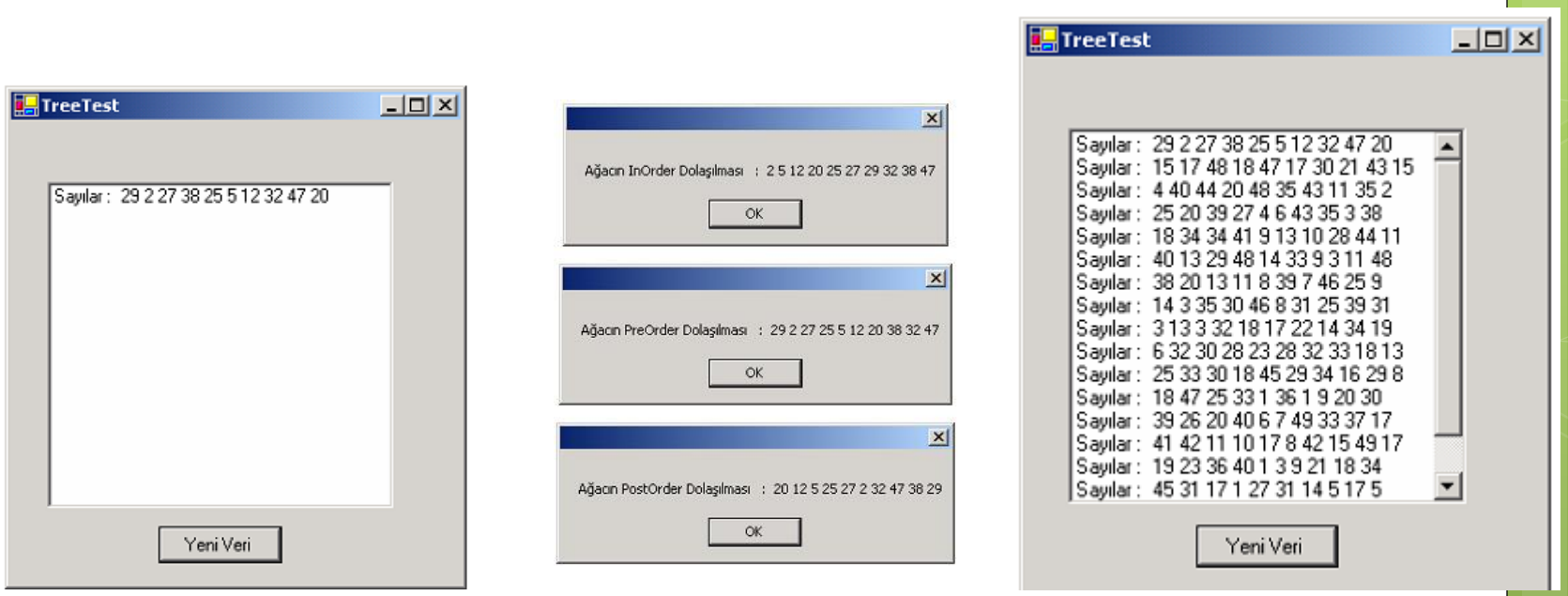
İkili arama Ağacı– C# (1. Örnek)

```
//Ekleme (append)
private void append(int sayi, bstNodeC node) {
bstNodeC yeniNode = new bstNodeC(sayi, null, null);
if (node.sayi == 0) node.sayi = sayi;
else  {
bstNodeC current = node; bstNodeC parent;
while(true)
{   parent = current;
if(sayi < current.sayi)
{ current = current.leftNode;
if(current == null)  {parent.leftNode = yeniNode;break; }
}
else
{ current = current.rightNode;
if(current==null) { parent.rightNode = yeniNode; return; }
}
}
}
}
```



İkili arama Ağacı– C# (2. Örnek)

- “TreeTest” adlı programın formu üzerinde 1 adet liste kutusu ve 1 adet düğme bulunmaktadır. “Yeni Veri” düğmesine basıldıkça, 1 ile 50 arasında 10 tane rastgele sayı üretmektedir ve arka arkaya gelen üç mesaj penceresi ile, InOrder, PreOrder ve PostOrder dolaşmalarda ikili ağaç üzerinde hangi düğüm sırasının izleneceğini göstermektedir.



İkili arama Ağacı– C# (2. Örnek)

```
public class GLOBAL   {      public static string tempStr;   }
class TreeNode
{   public int data;     public TreeNode leftChild;     public TreeNode rightChild;
  public void displayNode()     { GLOBAL.tempStr += (" "+data); }
 }
 // Ağaç Sınıfı
class Tree
{  private TreeNode root;
   public Tree()     { root = null; }
   public TreeNode getRoot()     { return root; }
// Ağacın preOrder Dolaılması
public void preOrder(TreeNode localRoot)
{
  if(localRoot!=null)
  {
    localRoot.displayNode();
    preOrder(localRoot.leftChild);
    preOrder(localRoot.rightChild);
  }
 }
```

İkili arama Ağacı– C# (2. Örnek)

```
// Ağacın inOrder Dolaşılması
  public void inOrder(TreeNode localRoot)
  {
   if(localRoot!=null)
   {
    inOrder(localRoot.leftChild);
    localRoot.displayNode();
    inOrder(localRoot.rightChild);
   }
  }
  // Ağacın postOrder Dolaşılması
  public void postOrder(TreeNode localRoot)
  {
   if(localRoot!=null)
   {
    postOrder(localRoot.leftChild);
    postOrder(localRoot.rightChild);
    localRoot.displayNode();
   }
  }
```

İkili arama Ağacı– C# (2. Örnek)

```
// Ağaca bir düğüm eklemeyi sağlayan metot
public void insert (int newdata)
{
 TreeNode newNode = new TreeNode();
 newNode.data = newdata;
 if(root==null)     root = newNode;
 else
 {
  TreeNode current = root;
  TreeNode parent;
  while(true)
  {
   parent = current;
   if(newdata<current.data)
   {
    current = current.leftChild;
    if(current==null)
```

İkili arama Ağacı– C# (2. Örnek)

```
{        parent.leftChild=newNode;
          return;
         }
        }
        else
        {
         current = current.rightChild;
         if(current==null)
         {
          parent.rightChild=newNode;
          return;
         }
        }
      }  // end while
     } // end else not root
   } // end insert()
} // class Tree
```

İkili arama Ağacı– C# (2. Örnek)

```
private void dugme1_Click(object sender, System.EventArgs e)     {
    Random r = new Random();            Tree theTree = new Tree();
    // Ağaca 10 tane sayı yerleştirilmesi
    string str = "";        str += "Sayılar : ";
    for (int i=0;i<10;++i)
    {
      int sayi = (int) (r.Next(1,50));        str += (" "+sayi);        theTree.insert(sayi);
    }
     listBox1.Items.Add(str);      GLOBAL.tempStr = "";
    theTree.inOrder(theTree.getRoot());
    MessageBox.Show("\nAğacın InOrder Dolaşılması   : "+GLOBAL.tempStr);

   GLOBAL.tempStr = "";       theTree.preOrder(theTree.getRoot());
    MessageBox.Show("\nAğacın PreOrder Dolaşılması   : "+GLOBAL.tempStr);

    GLOBAL.tempStr = "";       theTree.postOrder(theTree.getRoot());
    MessageBox.Show("\nAğacın PostOrder Dolaşılması   : "+GLOBAL.tempStr);
  }
```

AĞAÇ (TREE) Veri Modeli

Ağaç Veri Modeli Temel Kavramları

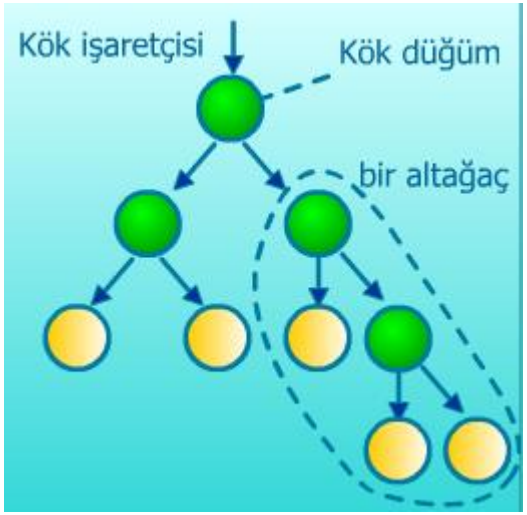
- Ağaç, bir kök işaretçisi, sonlu sayıda düğümleri ve onları birbirine bağlayan dalları olan bir veri modelidir; aynı aile soyağacında olduğu gibi hiyerarşik bir yapısı vardır ve orada geçen birçok kavram buradaki ağaç veri modelinde de tanımlıdır.

- Örneğin çocuk, kardeş düğüm, aile, ata gibi birçok kavram ağaç veri modelinde de kullanılır. Genel olarak, veri, ağacın düğümlerinde tutulur; dallarda ise geçiş koşulları vardır denilebilir.

- Her biri değişik bir uygulamaya doğal çözüm olan ikili ağaç, kodlama ağacı, sözlük ağacı, kümeleme ağacı gibi çeşitli ağaç şekilleri vardır; üstelik uygulamaya yönelik özel ağaç şekilleri de çıkarılabilir.

Ağaç Veri Modeli Temel Kavramları

- Bağlı listeler, yığıtlar ve kuyruklar doğrusal (linear) veri yapılarıdır. Ağaçlar ise doğrusal olmayan belirli niteliklere sahip iki boyutlu veri yapılarıdır.
  - Ağaçlar hiyerarşik ilişkileri göstermek için kullanılır.
  - Her ağaç node'lar ve kenarlardan (edge) oluşur.
  - Herbir node(düğüm) bir nesneyi gösterir.
  - Herbir kenar (bağlantı) iki node arasındaki ilişkiyi gösterir.
  - Arama işlemi bağlı dizilere göre çok hızlı yapılır.

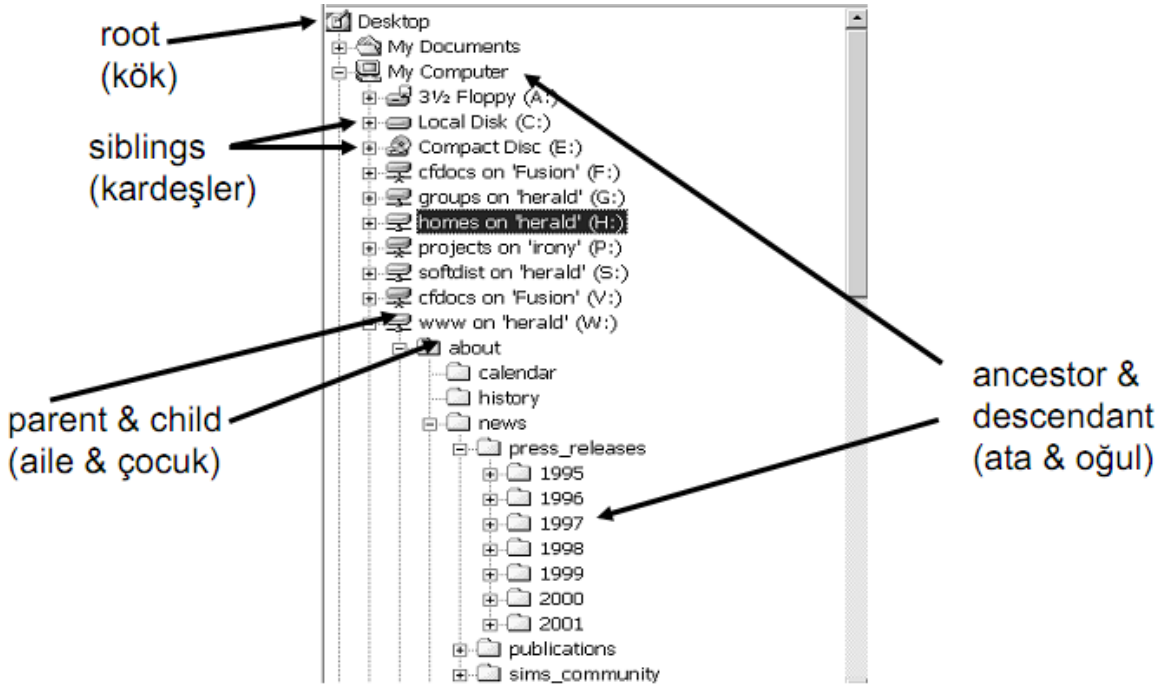


Ağaç Veri Modeli Temel Kavramları

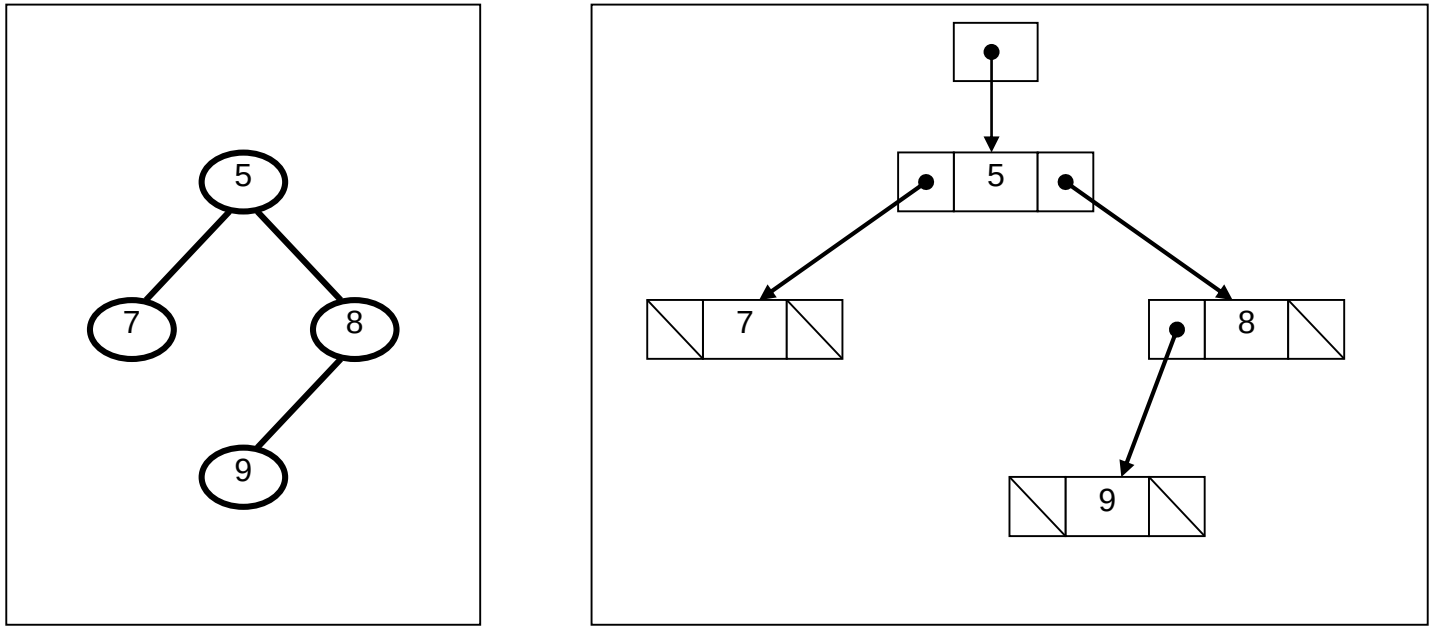
- Ağaçlardaki düğümlerden iki veya daha fazla bağ çıkabilir. İkili ağaçlar (binary trees), düğümlerinde en fazla iki bağ içeren (0,1 veya 2) ağaçlardır. Ağacın en üstteki düğümüne kök (root) adı verilir.
- **Uygulamaları:**
  - Organizasyon şeması
  - Dosya sistemleri
  - Programlama ortamları

Ağaç Veri Modeli Temel Kavramları

- Örnek ağaç yapısı



Ağaç Veri Modeli Temel Kavramları

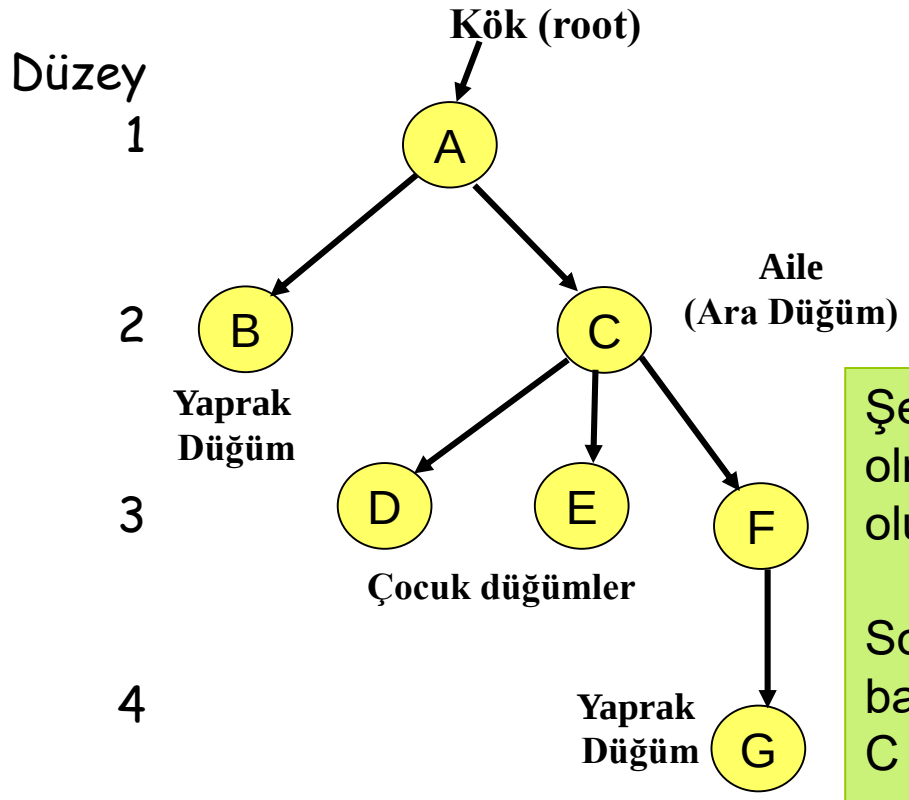


- **Şekil 1:** İkili ağacın grafiksel gösterimleri

Ağaç Veri Modeli Temel Kavramları

- Şekil 1’de görülen ağacın düğümlerindeki bilgiler sayılardan oluşmuştur. Her düğümdeki sol ve sağ bağlar yardımı ile diğer düğümlere ulaşılır. Sol ve sağ bağlar boş ("NULL" = "/" = "\") da olabilir.
- Düğüm yapıları değişik türlerde bilgiler içeren veya birden fazla bilgi içeren ağaçlar da olabilir.
- Doğadaki ağaçlar köklerinden gelişip göğe doğru yükselirken veri yapılarındaki ağaçlar kökü yukarıda yaprakları aşağıda olacak şekilde çizilirler.

Ağaç Veri Modeli Temel Kavramları



- **Şekil :** Ağaçlarda düzeyler

Şekildeki ağaç, A düğümü kök olmak üzere 7 düğümden oluşmaktadır.

Sol alt ağaç B kökü ile başlamakta ve sağ alt ağaç da C kökü ile başlamaktadır.

A'dan solda B'ye giden ve sağda C'ye giden iki dal (branch) çıkmaktadır.

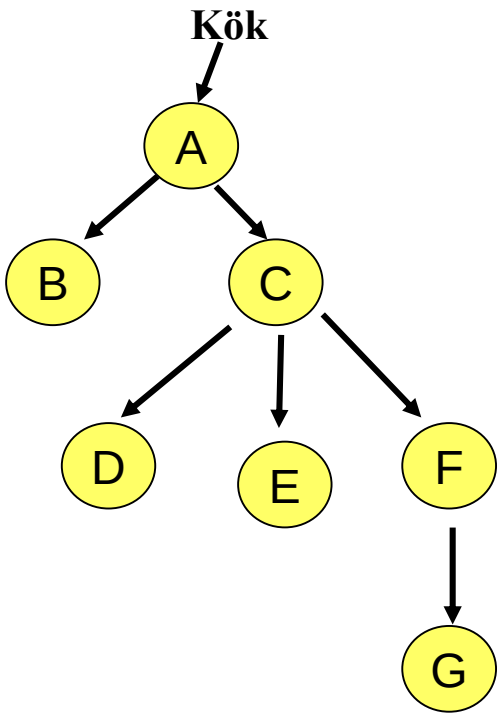
Ağaç Veri Modeline İlişkin Tanımlar

- **Düğüm (Node)**
  - Ağacın her bir elemanına düğüm adı verilir.
- **Kök Düğüm (Root)**
  - Ağacın başlangıç düğümüdür.
- **Çocuk (Child)**
  - Bir düğüme doğrudan bağlı olan düğümlere o çocukları denilir.
- **Kardeş Düğüm (Sibling)**
  - Aynı düğüme bağlı düğümlere kardeş düğüm veya kısaca kardeş denir.
- **Aile (Parent)**
  - Düğümlerin doğrudan bağlı oldukları düğüm aile olarak adlandırılır; diğer bir deyişle aile, kardeşlerin bağlı olduğu düğümdür.
- **Ata (Ancestor) ve Torun (Dedscendant)**
  - Aile düğümünün daha üstünde kalan düğümlere ata denilir; torun, bir düğümün çocuğuna bağlı olan düğümlere denir.

Ağaç Veri Modeline İlişkin Tanımlar

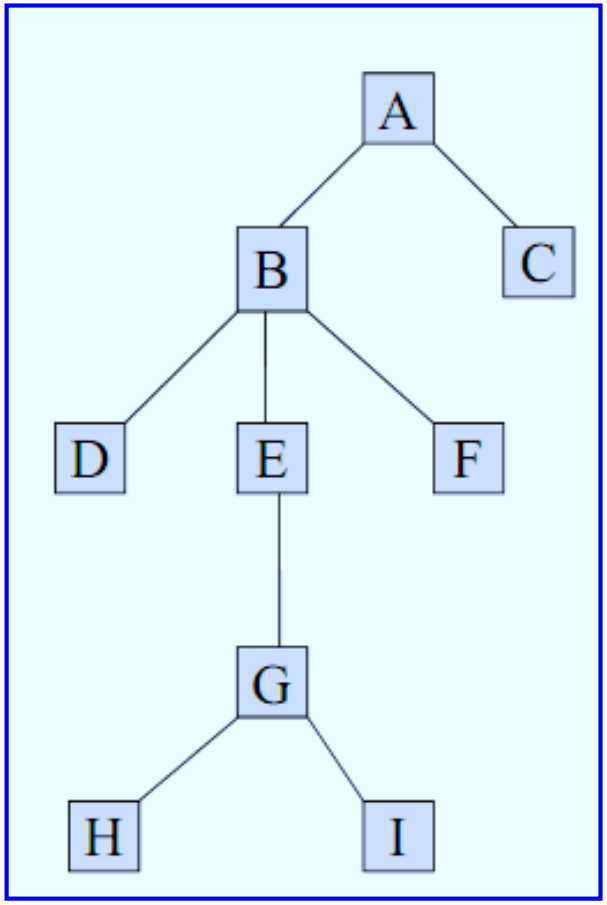
- **Derece (Degree)**
  - Bir düğümden alt hiyerarşiye yapılan bağlantıların sayısıdır; yani çocuk veya alt ağaç sayısıdır.
- **Düzey (Level) ve Derinlik (Depth)**
  - Düzey, iki düğüm arasındaki yolun üzerinde bulunan düğümlerin sayısıdır. Kök düğümün düzeyi 1, doğrudan köke bağlı düğümlerin düzeyi 2'dir. Bir düğümün köke olan uzaklığı ise derinliktir. Kök düğümün derinliği 1 dir.
- **Yaprak (Leaf)**
  - Ağacın en altında bulunan ve çocukları olmayan düğümlerdir.
- **Yükseklik (Height)**
  - Bir düğümün kendi silsilesinden en uzak yaprak düğüme olan uzaklığıdır.(İki düğüm arasındaki kenarların sayısı)
- **Yol(Path)**
  - Bir düğümün aşağıya doğru (çocukları üzerinden) bir başka düğüme gidebilmek için üzerinden geçilmesi gereken düğümlerin listesidir.

Ağaç Veri Modeline İlişkin Tanımlar



Tanım	Kök	B	D
Çocuk/Derece	2	0	0
Kardeş sayısı	1	2	3
Düzey	1	2	3
Aile	yok	kök	C
Ata	yok	yok	Kök
Yol	A	A, B	A,C,D
Derinlik	1	2	3
Yükseklik	3	2	1

Ağaç Veri Modeline İlişkin Tanımlar

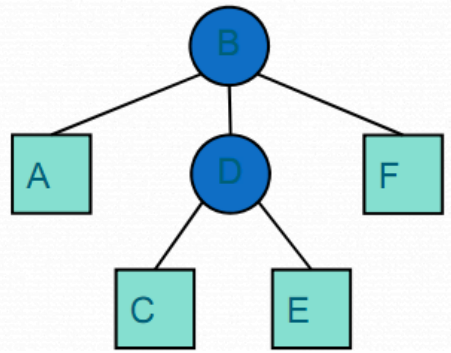


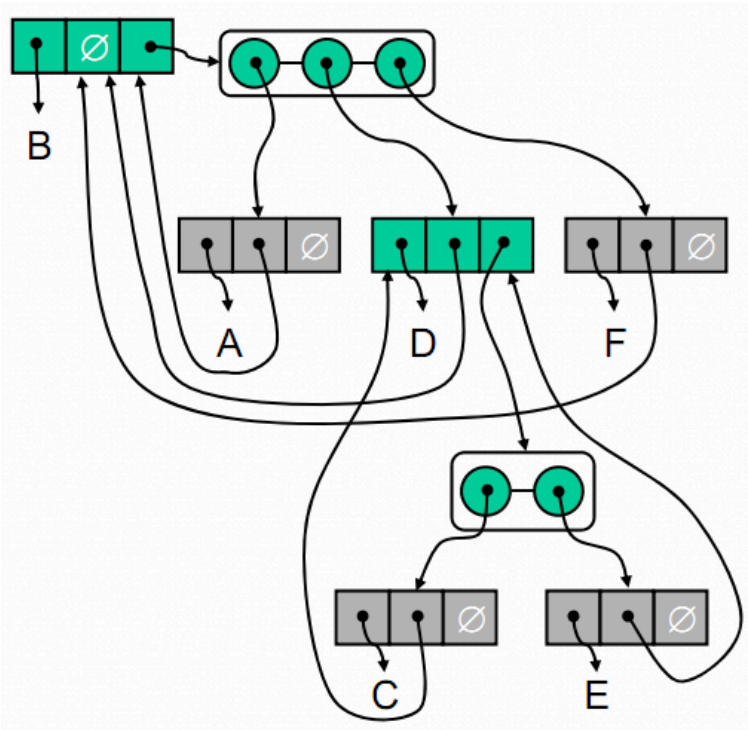
Tanım	Değer
Düğüm sayısı:	9
Yükseklik:	4
Kök düğüm:	A
Yapraklar:	C, D, F, H, I
Düzey sayısı:	5
H'nin ataları:	E, B, A
B'nin torunları:	G, H, I
E'nin kardeşleri:	D, F
Sağ alt ağaç:	Yok
Sol alt ağaç:	B

Ağaçların Bağlı Yapısı

- Bir düğüm çeşitli bilgiler ile ifade edilen bir nesnedir. Her bir bağlantı için birer bağlantı bilgisi tutulur.
  - Nesne/Değer (Element)
  - Ana düğüm (Parent node)
  - Çocuk düğümlerin listesi
- Problem: Bir sonraki elemanın çocuk sayısını bilmiyoruz.
- Daha iyi: Çocuk/Kardeş Gösterimi
  - Her düğümde iki bağlantı bilgisi tutularak hem çocuk hem de yandaki kardeş tutulabilir.
  - İstenildiği kadar çocuk/kardeş olabilir.

```
JAVA Declaration
class AgacDugumu {
    int eleman;
    AgacDugumu ilkCocuk;
    AgacDugumu kardes; }
```





Ağaç İşlemleri

- Genel Yöntemler:
  - integer size()
  - boolean isEmpty()
  - elements()
  - positions()
- Erişim yöntemleri:
  - root()
  - parent(p)
  - children(p)
- Sorgu (Query) Yöntemleri:
  - boolean isInternal(p)
  - boolean isRoot(p)
  - boolean isExternal(p)
  - Güncelleme (Update) Yöntemi:
  - object replace (p, o)
- Ek yöntemler de tanımlanabilir

Ağaç Türleri

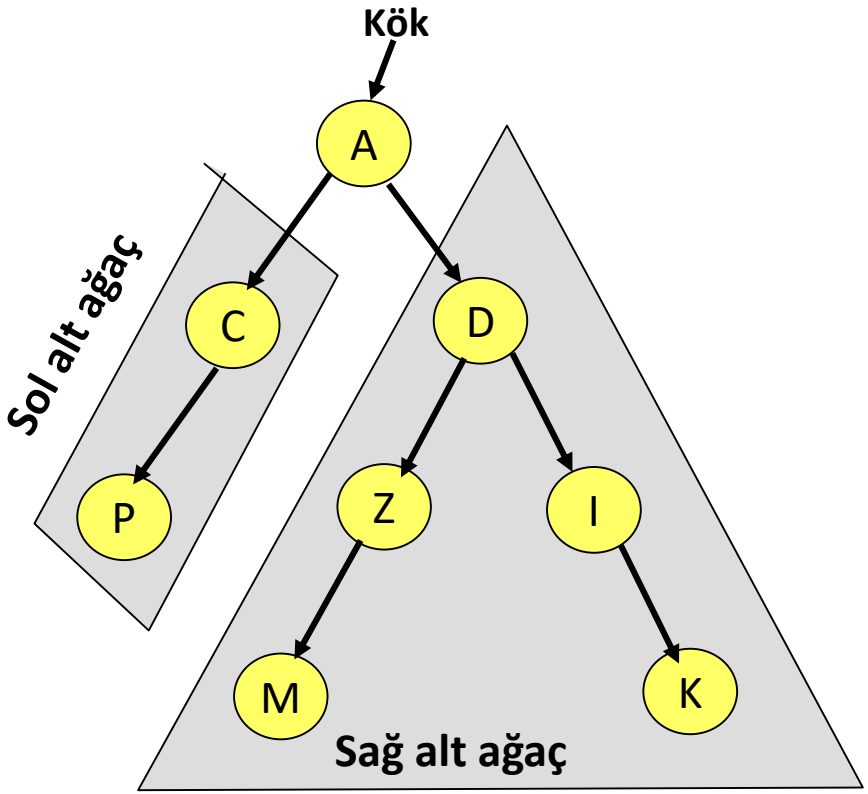
- En çok bilinen ağaç türleri ikili arama ağacı olup kodlama ağacı, sözlük ağacı, kümeleme ağacı gibi birçok ağaç uygulaması vardır.
  - **İkili Arama Ağacı (Binary Search Tree):**
    - İkili arama ağacında bir düğüm en fazla iki tane çocuğa sahip olabilir ve alt/çocuk bağlantıları belirli bir sırada yapılır.
  - **Kodlama Ağacı (Coding Tree):**
    - Bir kümedeki karakterlere kod ataması için kurulan ağaç şeklidir. Bu tür ağaçlarda kökten başlayıp yapraklara kadar olan yol üzerindeki bağlantı değerleri kodu verir.
  - **Sözlük Ağacı(Trie: Keyword Tree):**
    - Bir sözlükte bulunan sözcüklerin tutulması için kurulan bir ağaç şeklidir. Amaç arama işlemini en performanslı bir şekilde yapılması ve belleğin optimum kullanılmasıdır.
  - **Kümeleme Ağacı (Heap Tree):**
    - Bir çeşit sıralama ağacıdır. Çocuk düğümler her zaman aile düğümlerinden daha küçük değerlere sahip olur.

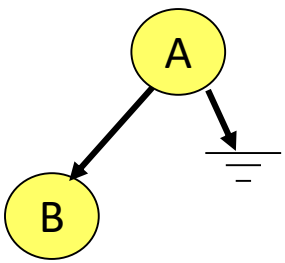
İkili Ağaç (Binary Tree) :

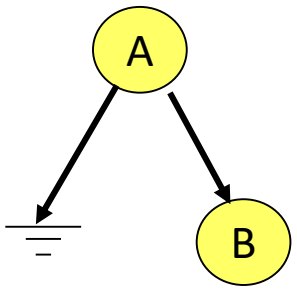
- Sonlu düğümler kümesidir. Bu küme boş bir küme olabilir (empty tree). Boş değilse şu kurallara uyar.
  - Kök olarak adlandırılan özel bir düğüm vardır.
  - Her düğüm en fazla iki düğüme bağlıdır.
    - Left child : Bir node'un sol işaretçisine bağlıdır.
    - Right child : Bir node'un sağ işaretçisine bağlıdır.
  - Kök hariç her düğüm bir daldan gelmektedir.
  - Tüm düğümlerden yukarı doğru çıkıldıkça sonuçta köke ulaşılır.

İkili Ağaç (Binary Tree) :

- Bilgisayar bilimlerinde en yaygın ağaçtır.



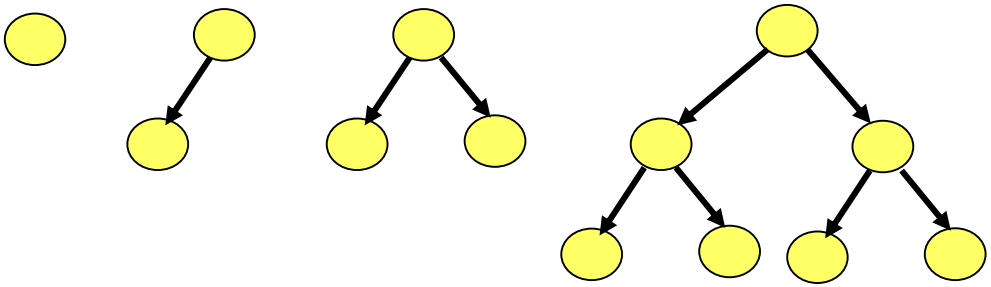




İki farklı ikili ağaç

İkili Ağaç (Binary Tree)

- **n** tane düğüm veriliyor, İkili ağacın minimum derinliği nedir.



Derinlik 1: n = 1, 1 düğüm 2^1 -1

Derinlik 2: n = 2, 3 düğüm, 2^2 -1 düğüm

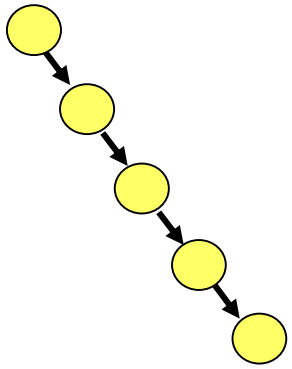
Herhangi bir d derinliğinde, n = ?

Derinlik d: n = 2^d-1 düğüm (tam bir ikili ağaç)

En küçük derinlik: **Θ(log n)**

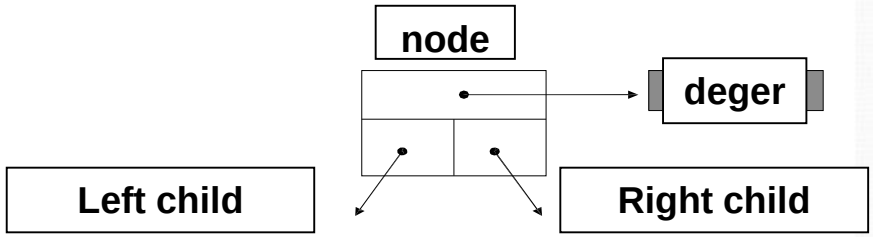
İkili Ağaç

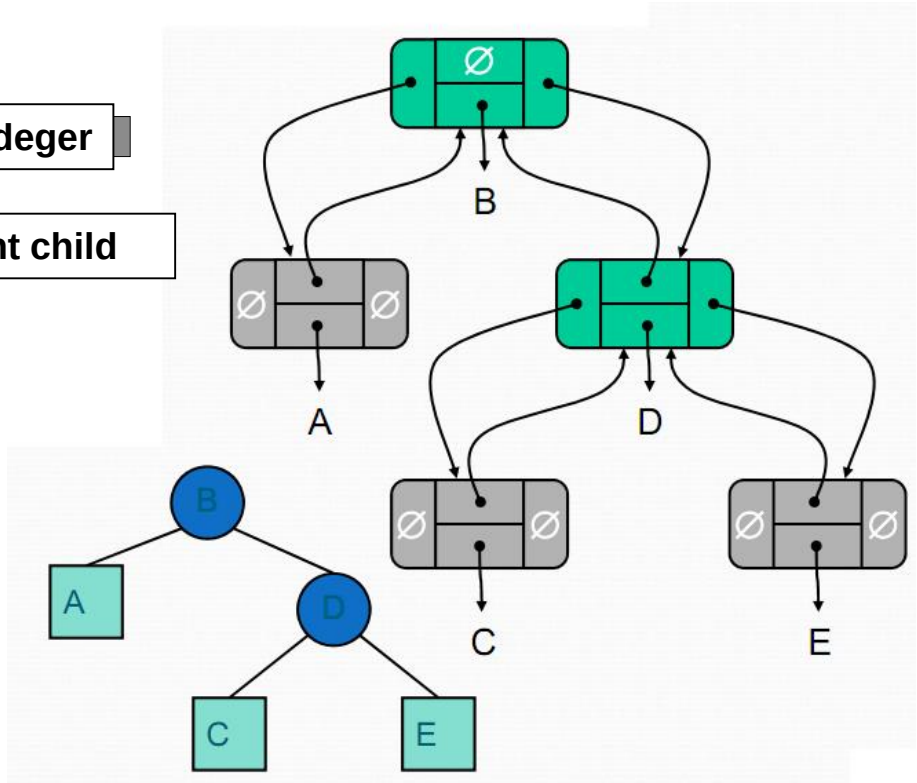
- ***n*** düğümlü ikili ağacın minimum derinliği: **Θ(log n)**
- İkili ağacın maksimum derinliği ne kadardır?
  - Dengesiz ağaç: Ağaç bir bağlantılı liste olursa!
  - Maksimum derinlik = n
- Amaç: Arama gibi operasyonlarda bağlantılı listeden daha iyi performans sağlamak için derinliğin log n de tutulması gerekmektedir.



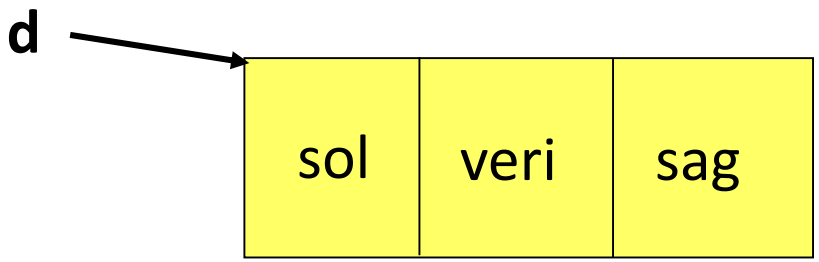
- Bağlantılı liste
- Derinlik = n

İkili Ağaç Bağlı Liste Yapısı

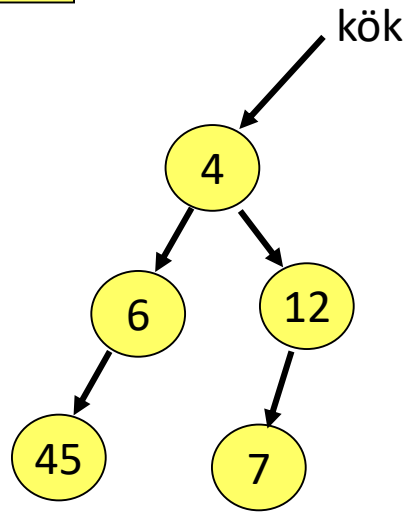




İkili Ağaç Gerçekleştirimi



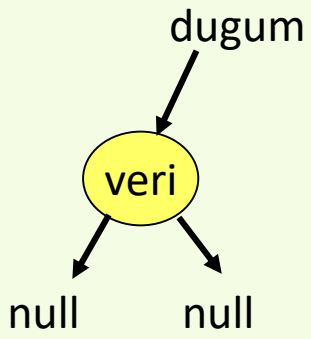

```
public class İkiliAgacDugumu {
    public İkiliAgacDugumu sol;
    public int veri;
    public İkiliAgacDugumu sag;
}
```



İkili Ağaç Gerçekleştirimi

- İkili ağaç için düğüm oluşturma. Her defasında sol ve sağı boş olan bir düğüm oluşturulur.

```
/* İkili ağaç düğümü oluşturur. */
  İkiliAgacDugumu DugumOlustur(int veri){
  İkiliAgacDugumu dugum = new İkiliAgacDugumu();
  dugum.veri = veri;
  dugum.sol = null;
  dugum.sag = null;
  return dugum;
}
```



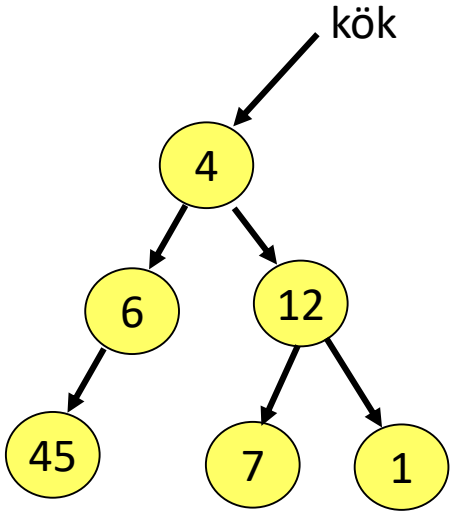
İkili Ağaç Gerçekleştirimi

- İteratif düğüm oluşturma ve ağaca ekleme.

```
İkiliAgacDugumu dugum = null;

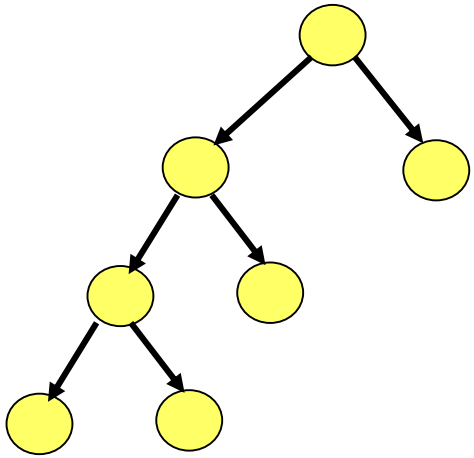
public static void main main(){
   kok = DugumOlustur(4);

   kok.sol = DugumOlustur(6);
   kok.sag = DugumOlustur(12);
   kok.sol.sol = DugumOlustur(45);
   kok.sag.sol = DugumOlustur(7);
   kok.sag.sag = DugumOlustur(1);
} /* main */
```

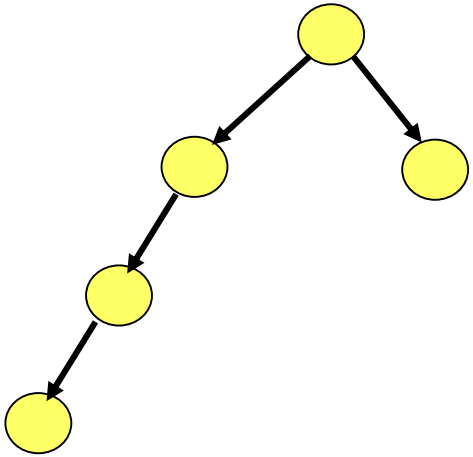


Proper (düzgün) Binary Trees

- Yaprak olmayan düğümlerin tümünün iki çocuğu olan **T** ağacı proper(düzgün) binary tree olarak adlandırılır.



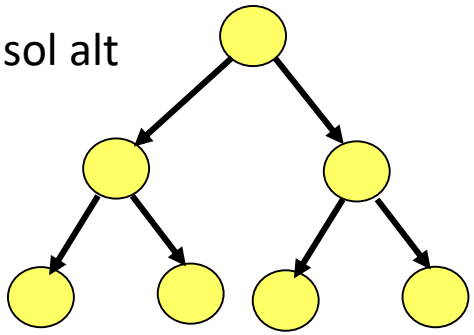
- Proper Tree



ImProper Tree

Full Binary Tree

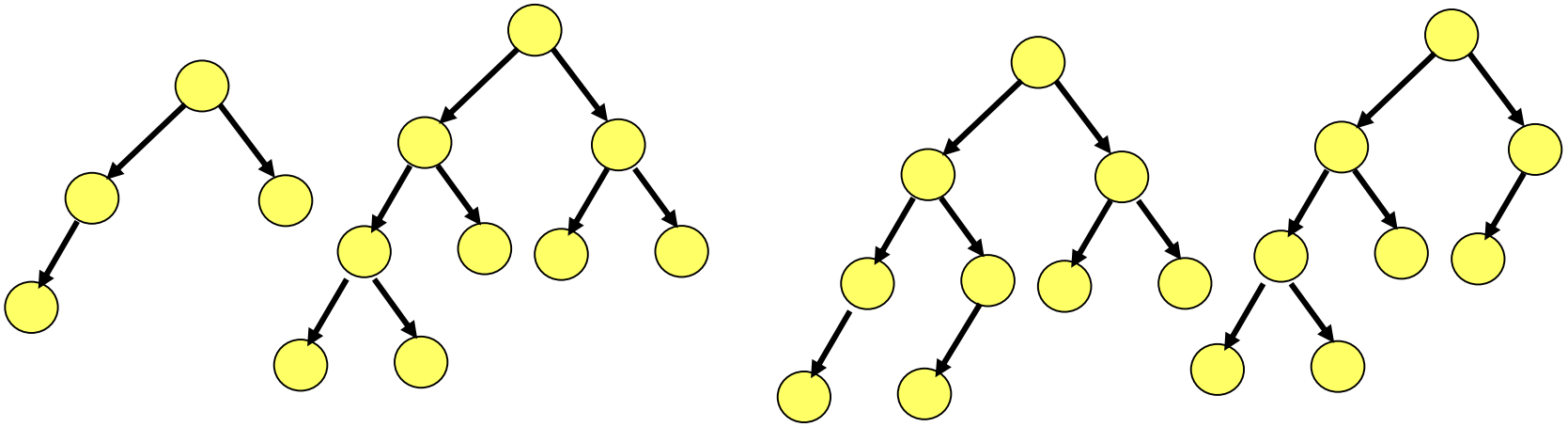
- Full binary tree:
  - **1-** Her yaprağı aynı derinlikte olan
  - **2-** Yaprak olmayan düğümlerin tümünün iki çocuğu olan ağaç **Full Binary Tree**'dir.
- Bir full binary tree'de **m** tane yaprak varsa bu ağaçta toplam **n=2m-1** düğüm vardır. Başka bir şekilde ifade edilirse,
  - Eğer **T** ağacı boş ise, **T** yüksekliği **0** olan bir full binary ağaçtır.
  - **T** ağacının yüksekliği **h** ise ve yüksekliği **h**'den küçük olan tüm node'lar iki child node'a sahipse, **T full binary** tree'dir.
  - Full binary tree'de her node aynı yüksekliğe eşit sağ ve sol alt ağaçlara sahiptir. En iyi bilinen proper tree ağacıdır.



Complete Binary Tree (Tamamlanmış İkili Ağaç)

- Full Binary Tree'de, yeni bir derinliğe soldan sağa doğru düğümler eklendiğinde oluşan ağaçlara **Complete Binary Tree** denilir.
- Böyle bir ağaçta bazı yapraklar diğerlerinden daha derindir. Bu nedenle Full Binary Tree olmayabilirler. En derin düzeyde düğümler olabildiğince soldadır. (Full Binary Tree aynı zamanda Complete binary tree dir)
-
  - **T**, **n** yükseklikte Complete Binary Tree ise, tüm yaprak node'ları n veya **n-1** derinliğindedir ve yeni bir derinliğe soldan sağa doğru ekleme başlanır.
  - Her node iki tane child node'a sahip olmayabilir.

Complete Binary Tree



- **Complete** **incomplete**
  - Her node iki tane child node’a sahip olmayabilir.

Örnek

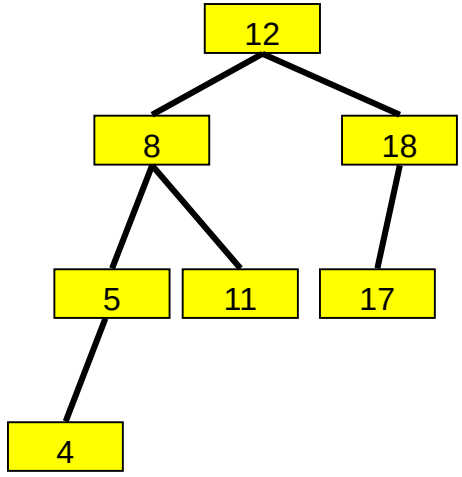
- Yüksekliği (h) 3 olan Tam (Full) ikili ağaçta minimum ve maksimum düğüm sayısı kaçtır?

Örnek

- Yüksekliği (h) 3 olarak verilen bir ağaçta sol alt ağaç ile sağ alt ağaç yükseklik farkı 1 ise Komple (Complete) ikili ağacın maksimum düğüm sayısı kaçtır?

Balanced Binary Trees

- Yüksekliği ayarlanmış ağaçlardır.
- Bütün düğümler için sol altağacın yüksekliği ile sağ altağacın yüksekliği arasında en fazla bir fark varsa balanced binary tree olarak adlandırılır.
- Complete binary tree'ler aynı zamanda balanced binary tree'dir.
- Her balanced binary tree, complete binary tree olmayabilir. (Neden?)

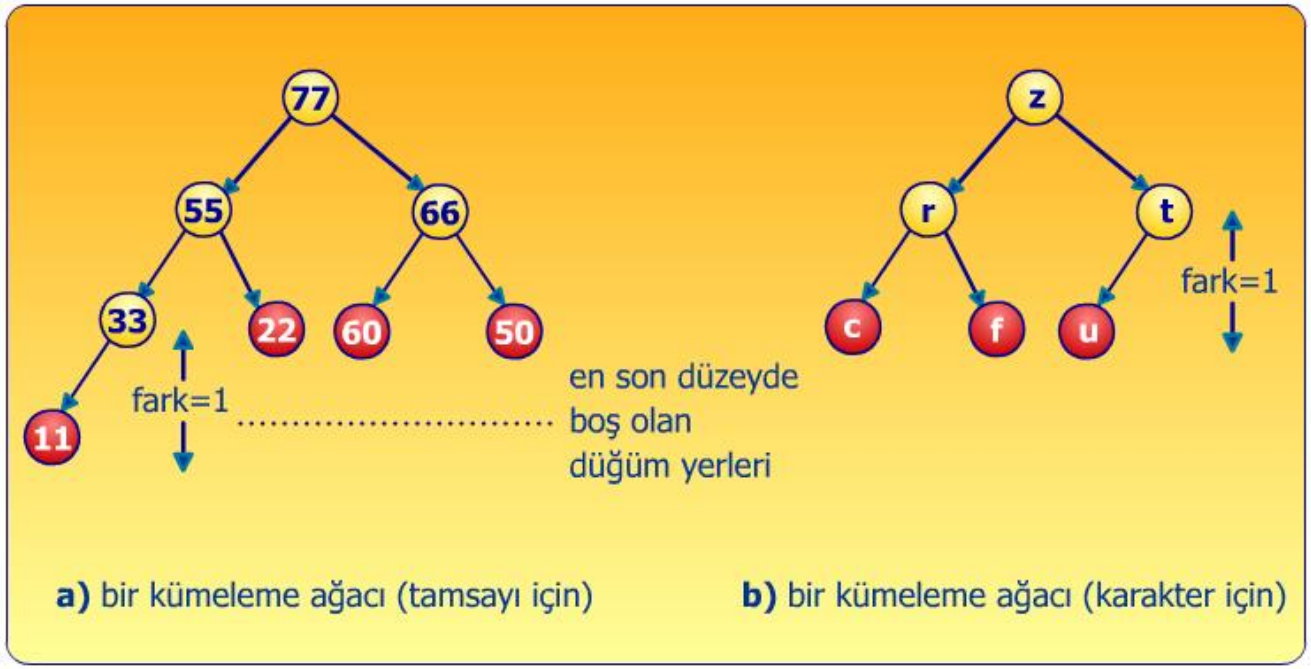


Heap Tree (Kümele Ağacı)

- Kümeleme ağacı, ağaç oluşturulurken değerleri daha büyük olan düğümlerin yukarıya, küçük olanların da aşağıya eklenmesine dayanan tamamlanmış bir ağaçtır.
- Böylece 0. düzeyde, kökte, en büyük değer bulunurken yaprakların olduğu *k*. düzeyde en küçük değerler olur. Yani büyükten küçüğe doğru bir kümeleme vardır; veya tersi olarak küçükten büyüğe kümeleme de yapılabilir.
- Aksi belirtilmediği sürece kümeleme ağacı, sayısal veriler için büyükten küçüğe, sözcükler için alfabetik sıralamaya göre yapılır.
- Kümeleme ağacı bilgisayar biliminde bazı problemlerin çözümü için çok uygundur; hatta birebir uyuşmaktadır denilebilir. Üstelik, hem bellek alanı hem de yürütme zamanı açısından getirisi olur.

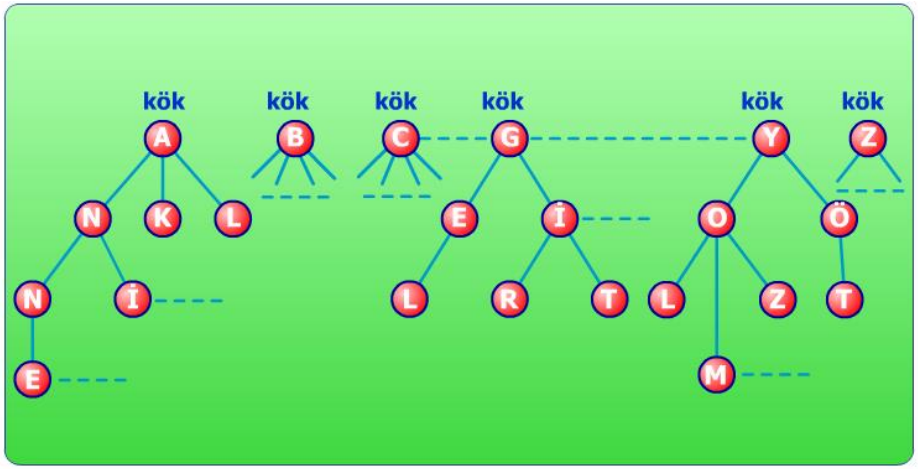
Heap Tree (Kümele Ağacı)

- Bir düğüm her zaman için çocuklarından daha büyük değere sahiptir.
- Yaprak düğümlerin herhangi ikisi arasındaki fark en fazla 1 dir.
- En son düzey hariç tüm düzeyler düğümlerle doludur.
- En son düzeyde oluşan boşluklar sadece sağ taraftadır
- Sıralama işlemlerinde kullanılır.



Trie Ağacı/Sözlük Ağacı

- Sözlük ağacı, bir sözlükte bulunan sözcükleri tutmak ve hızlı arama yapabilmek amacıyla düşünülmüştür; bellek gereksinimi arttırmadan, belki de azaltarak, onbinlerce, yüzbinlerce sözcük bulunan bir sözlükte 10-15 çevrim yapılarak aranan sözcüğün bulunması veya belirli bir karakter kadar uyuşanının bulunması için kullanılmaktadır.
- Sözlük ağacı, sıkıştırma, kodlama gibi sözlük kurulmasına dayalı algoritmalarda ve bir dilin sözlüğünü oluşturmada kullanılmaktadır.



Ödev

- Aşağıda verilen ağaç yapılarını araştırınız. Bu ağaçlara ait bilgileri Word ve Powerpoint ortamında hazırlayıp getiriniz.
  - Sözlük Ağaçları
  - Kodlama Ağaçları
  - Sıkıştırma Ağaçları
  - Bağıntı Ağaçları
  - Küme Ağacı

Geçiş İşlemleri

İkili Ağaçlar Üzerindeki Geçiş İşlemleri

- Geçiş (traverse), ağaç üzerindeki tüm düğümlere uğrayarak gerçekleştirilir. Ağaçlar üzerindeki geçiş işlemleri, ağaçtaki tüm bilgilerin listelenmesi veya başka amaçlarla yapılır.

- Doğrusal veri yapılarında baştan sona doğru dolaşmak kolaydır. Ağaçlar ise düğümleri doğrusal olmayan veri yapılarıdır. Bu nedenle farklı algoritmalar uygulanır.

İkili Ağaçlar Üzerindeki Geçiş İşlemleri

- **Preorder (depth-first order) Dolaşma (Traversal) (Önce Kök)**
  - Köke uğra (visit)
  - Sol alt ağacı preorder olarak dolaş.
  - Sağ alt ağacı preorder olarak dolaş.
- **Inorder (Symmetric order) Dolaşma(Ortada Kök)**
  - Sol alt ağacı inorder'a göre dolaş
  - Köke uğra (visit)
  - Sağ alt ağacı inorder'a göre dolaş.
- **Postorder Dolaşma (Sonra Kök)**
  - Sol alt ağacı postorder'a göre dolaş
  - Sağ alt ağacı postorder'a göre dolaş.
  - Köke uğra (visit)
- **Level order Dolaşma (Genişliğine dolaşma)**
  - Köke uğra
  - Soldan sağa ağacı dolaş

Preorder Geçişi

- Preorder geçişte, bir düğüm onun neslinden önce ziyaret edilir.
- Uygulama: yapılandırılmış bir belgenin yazdırılması

Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)

```
OnceKok(IkiliAgacDugumu kok)
{
    if (kok == null) return;
    System.out.print(kok.veri+" ");
    OnceKok(kok.sol);
    OnceKok(kok.sag);
}
```

Postorder Geçişi

- Postorder geçişte, bir düğüm onun neslinden sonra ziyaret edilir.
- Uygulama: Bir klasör ve onun alt klasörlerindeki dosyaların kullandıkları alanın hesaplanması.

```
Algorithm postOrder(v)
    for each child w of v
        postOrder (w)
    visit(v)
```

```
SonraKok(IkiliAgacDugumu kok)
{
    if (kok == null) return;
    SonraKok(kok.sol);
    SonraKok(kok.sag);
    System.out.print(kok.veri+" ");
}
```

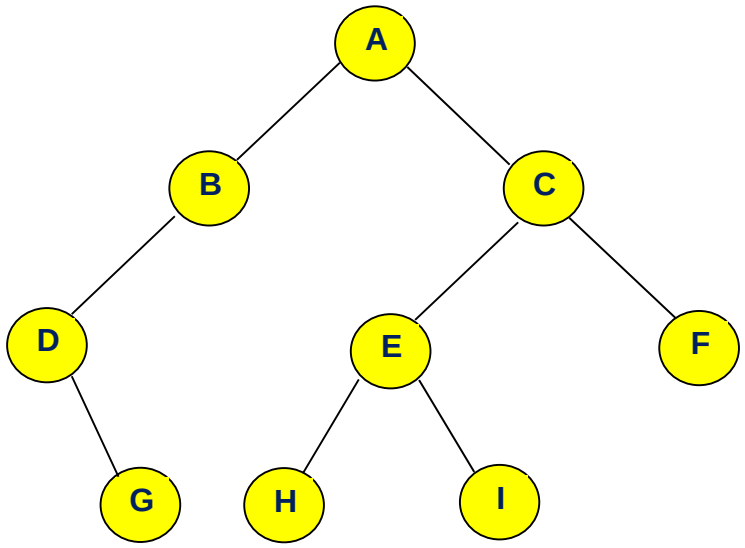
Inorder Geçişi

- Inorder geçişte, bir düğüm sol alt ağaçtan sonra ve sağ alt ağaçtan önce ziyaret edilir.
- Uygulama: ikili ağaç çizmek
- x(v) = v düğümünün inorder sıralaması (rank)
- y(v) = v'nin derinliği




```
OrtadaKok (IkiliAgacDugumu kok)
{
    if (kok == null) return;
    OrtadaKok(kok.sol);
    System.out.print(kok.veri+" ");
    OrtadaKok(kok.sag);
}
```

İkili Ağaçlar Üzerindeki Geçiş İşlemleri



PreOrder : A B D G C E H I F

InOrder : D G B A H E I C F

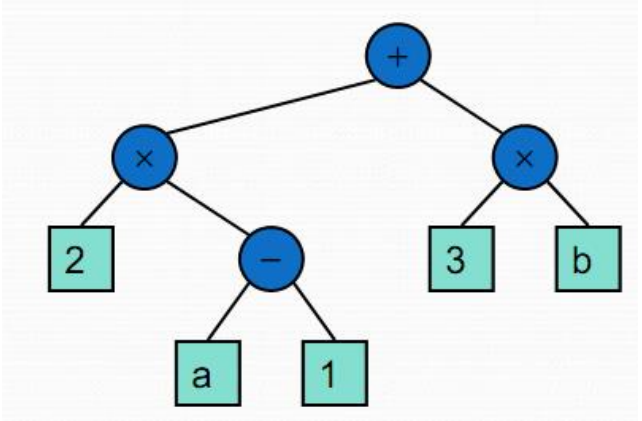
PostOrder : G D B H I E F C A

LevelOrder: A B C D E F G H I

Bağıntı (İfade) Ağaçları (Expression Tree)

- Bağıntı ağaçları bir matematiksel bağıntının ağaç şeklinde tutulması için tanımlanmıştır. Bir aritmetik ifade ile ilişkili ikili ağaçtır.
- Ağacın genel yapısı:
  - Yaprak düğüm = değişken/sabit değer
  - Kök veya ara düğümler = operatörler
  - Birçok derleyicide kullanılır. Parantez gereksinimi yoktur.

- Örnek: (2 × (a − 1) + (3 × b)) aritmetik ifadesi için ağaç yapısı

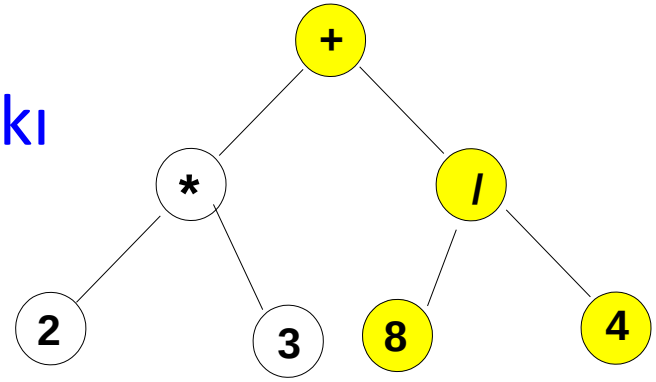


Bağıntı (İfade) Ağaçları (Expression Tree)

- Verilen denklemden bağıntı ağacı kurulması veya verilen ağaçtan denklemin çıkartılması için üç değişik yöntem vardır:
  - İç Ek(Infix): matematikte alışılagelen şekilde, operatörlerin ortada, değişken veya sabit değerlerin operatörün kenarında bulunan yöntemdir.
  - Ön Ek (Prefix): operatörler kendilerine karşı düşen parametrelerin(değişken/sabit) önündedir.
  - Son Ek(Postfix): Polonyalı notasyonu olarak da adlandırılır ve operatörler kendi parametrelerinin arkasından gelir.

Bağıntı Ağaçlarında Geçiş

- preOrder, (prefix)- Ön-takı
  - + * 2 3 / 8 4
- inOrder, (infix)- İç-takı
  - 2 * 3 + 8 / 4
- postOrder, (postfix) Son-takı
  - 2 3 * 8 4 / +
- levelOrder,
  - + * / 2 3 8 4

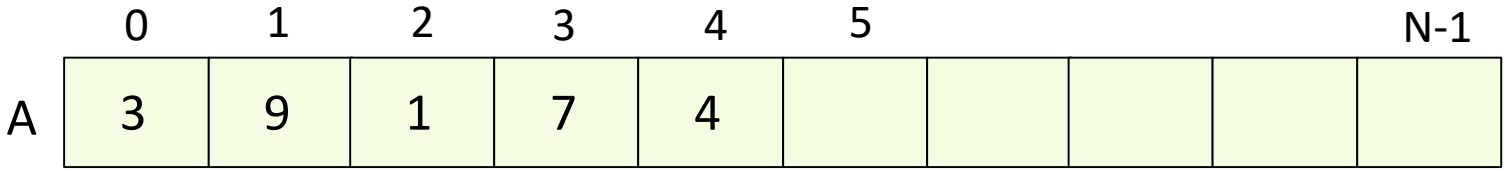


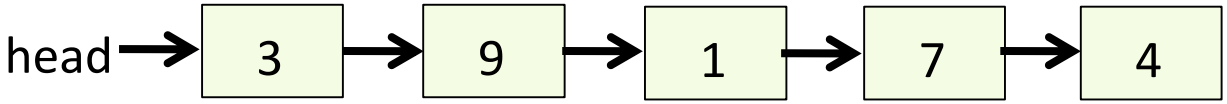
Arama Ağaçları

- Bir veri yapısı içerisine çok sayıda (anahtar, değer) çiftleri saklamak istediğimizi varsayalım.
- Aşağıdaki işlemleri etkili bir şekilde yerine getirebilecek bir veri yapısına ihtiyacımız var.
  - Ekle(anahtar, değer)
  - Sil(anahtar, değer)
  - Bul(anahtar)
  - Min()
  - Max()
- Alternatif veri yapıları?
  - Dizi kullanmak
  - Bağlantılı liste kullanmak

Arama Ağaçları

Örnek: Yandaki değerleri saklayalım: 3, 9, 1, 7, 4





Operasyon	Sırasız Dizi	Sıralı Dizi	Sırasız Liste	Sıralı List
Bul (Arama)	O(N)	O(logN)	O(N)	O(N)
Ekle	O(1)	O(N)	O(1)	O(N)
Sil	O(1)	O(N)	O(1)	O(1)

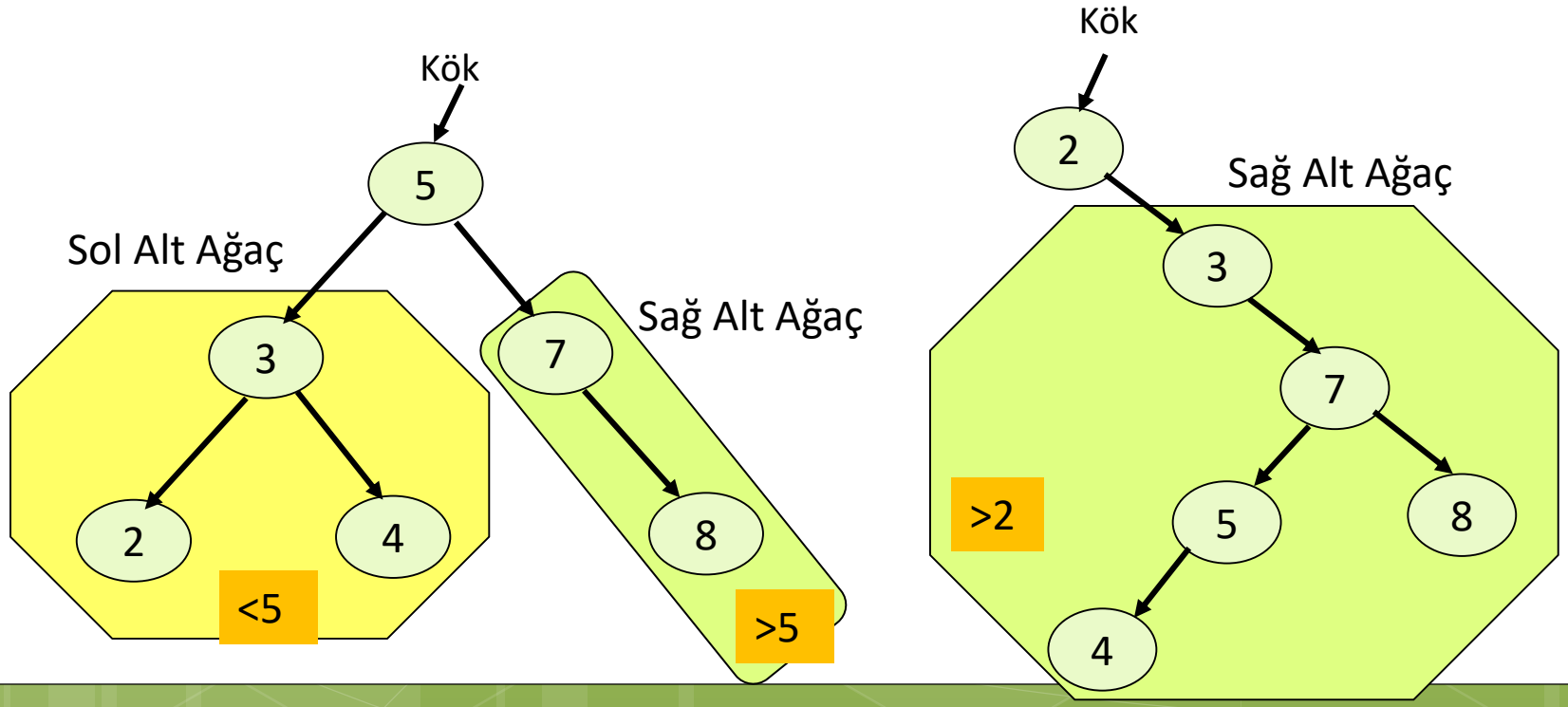
Bul/Ekle/Sil işlemlerinin hepsini O(logN) de yapabilir miyiz?

Kullanılan Verimli Arama Ağaçları

- **Fikir:** Verileri arama ağacı yapısına göre düzenlersek arama işlemi daha verimli olacaktır.
  - İkili Arama Ağacı (**B**inary **S**earch **T**ree (**BST**))
  - AVL Ağacı
  - Splay Ağacı
  - 2-3-4 Ağacı
  - Red-Black Ağacı
  - B Ağacı ve B+ Ağacı

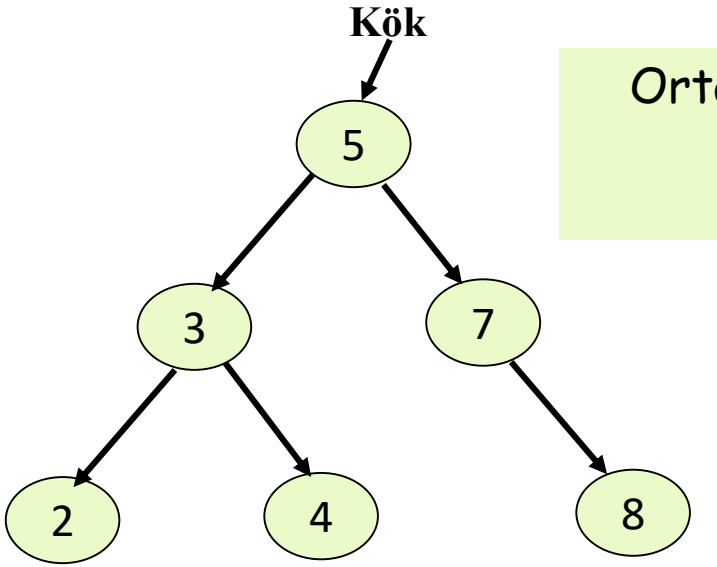
İkili Arama Ağacı (Binary Search Tree)

- İkili Arama Ağacı her bir düğümdeki değerlere göre düzenlenir:
  - Sol alt ağaçtaki tüm değerler kök düğümünden küçüktür.
  - Sağ alt ağaçtaki tüm değerler kök düğümünden büyüktür.



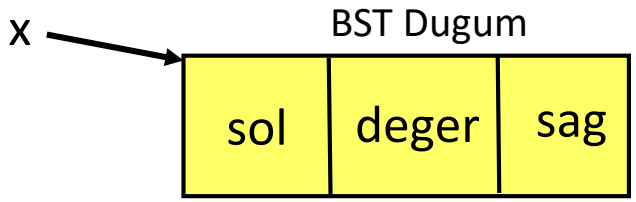
İkili Arama Ağacında Sıralama

- İkili arama ağacı önemli özelliklerinden birisi Ortada-kök (Inorder) dolaşma algoritması ile düğümlere sıralı bir şekilde ulaşılmasını sağlar.



Ortada-kök sonucu
2 3 4 5 7 8

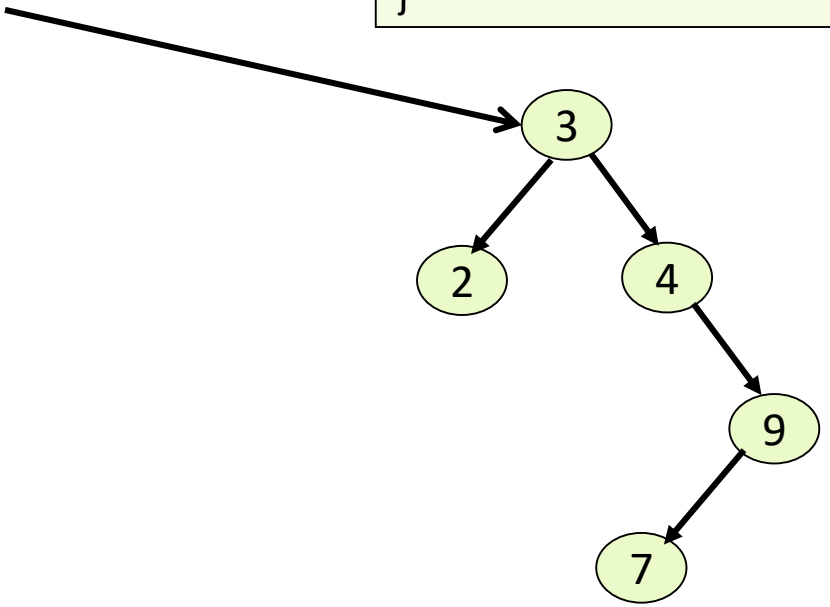
BST İşlemleri- Tanımlama




```
public class BSTDugum
{
  public BSTDugum sol;
  public int deger;
  public BSTDugum sag;
}
```

```
/* İKİLİ ARAMA AĞACI */
public class BST {
  Private BSTDugum kok;

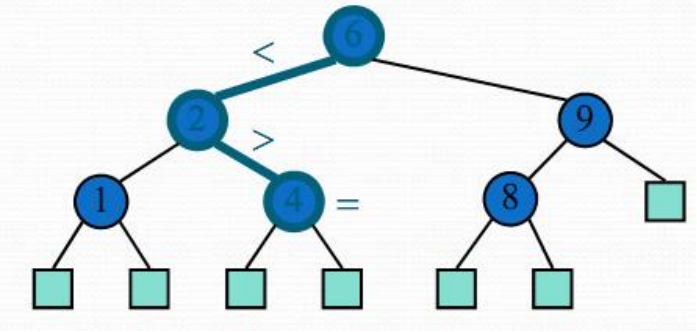
  public BST(){kok=null;}
  public void Ekle(int deger);
  public void Sil(int deger);
  public BSTNode Bul(int key);
  public BSTNode Min();
  public BSTNode Max();
};
```



BST İşlemleri-Arama

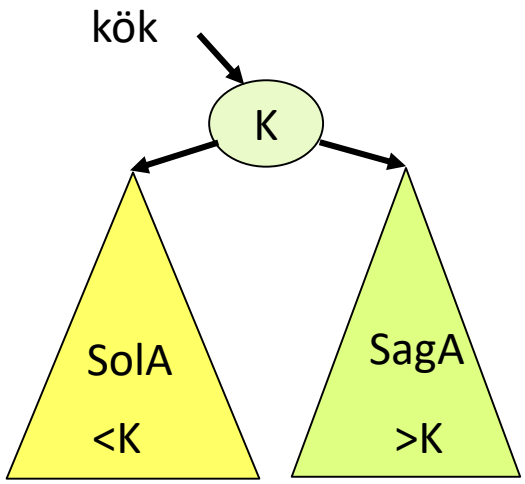
- Bir k anahtarını aramak için, kök düğümden başlayarak aşağı doğru bir yol izlenir.
- Bir sonraki ziyaret edilecek düğüm, k anahtar değerinin geçerli düğümün anahtar değeriyle karşılaştırılması sonucuna bağlıdır.
- Eğer yaprağa ulaşıldıysa anahtar bulunamamıştır ve null değer geri döndürülür.
- Örnek: Bul(4)

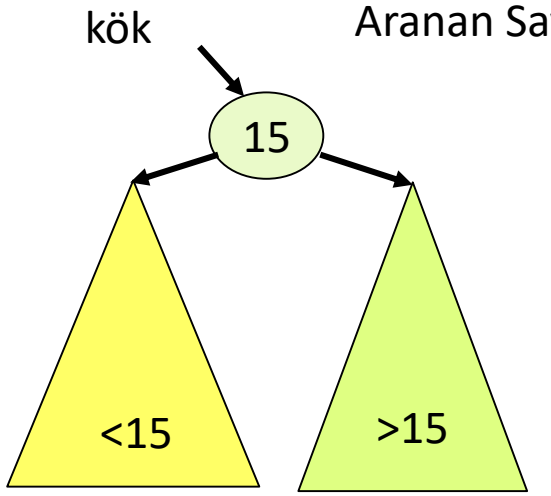
```
Algorithm TreeSearch(k, v)
    if T.isExternal (v)
        return v
    if k < key(v)
        return TreeSearch(k, T.left(v))
    else if k = key(v)
        return v
    else { k > key(v) }
        return TreeSearch(k, T.right(v))
```



BST İşlemleri- Arama

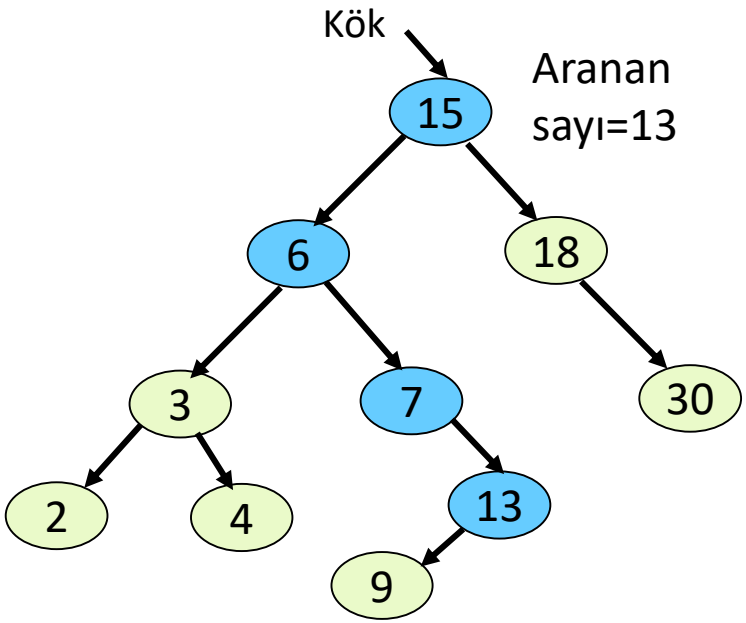
- Değeri içeren düğümü bul ve bu düğümü geri döndür.






```
1.  Arama işlemine kökten başla
2.  if (aranaDeger == kok.deger)      return kok;
3.  if (aranaDeger < kok.deger)       Ara SolAltAğaç
4.  else                              Ara SagAltAğaç
```

BST İşlemleri- Arama




```
public BSTDugum Bul(int deger)
{  return Bul2(kok, deger); }
```

```
public BSTDugum Bul2(BSTDugum kok, int deger)
{
  if (kok == null) return null;
  if (deger == kok.deger)
    return kok;
  else if (deger < kok.deger)
    return Bul2(kok.sol, deger);
  else /* deger > kok.deger */
    return Bul2(kok.sag, deger);
}
```

Mavi renkli düğümler arama sırasında ziyaret edilen düğümlerdir.

Algoritmanın çalışma karmaşıklığı O(d)'dir. (d = ağacın derinliği)

BST İşlemleri-Arama

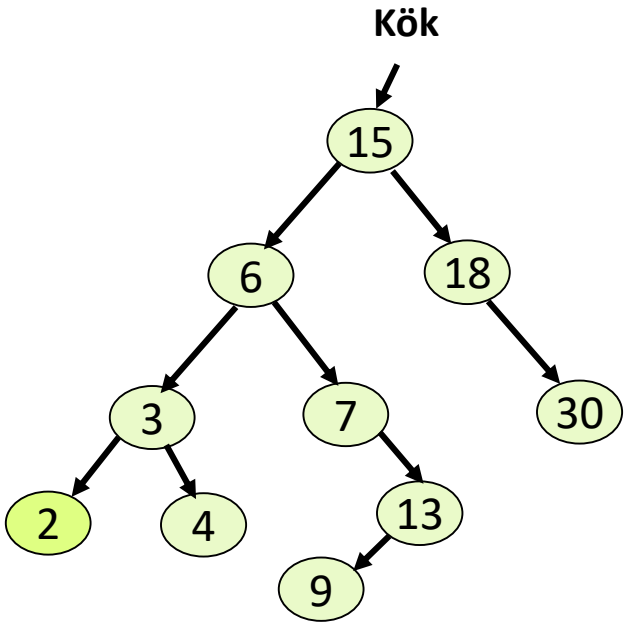
- Aynı algoritma while döngüsü yardımıyla yinelemeli şekilde yazılabilir. Yinelemeli versiyon özyinelemeli versiyona göre daha verimli çalışır.

```
public BSTDugum Bul(int deger){
  BSTDugum p = kok;
  while (p){
    if (deger == p.deger)      return p;
    else if (deger < p.deger)  p = p.sol;
    else /* deger > p.deger */  p = p.sag;
  } /* while-bitti */
  return null;
} //bul-Bitti
```

BST İşlemleri- Min

- Ağaçtaki en küçük elemanı içeren düğümü bulur ve geri döndürür.
  - En küçük elemanı içeren düğüm en soldaki düğümde bulunur.
  - Kökten başlayarak devamlı sola gidilerek bulunur.

```
public BSTDugum Min()
{
  if (kok == null)
    return null;
  BSTDugum p = kok;
  while (p.sol != null){
    p = p.sol;
  }
  return p;
}
```



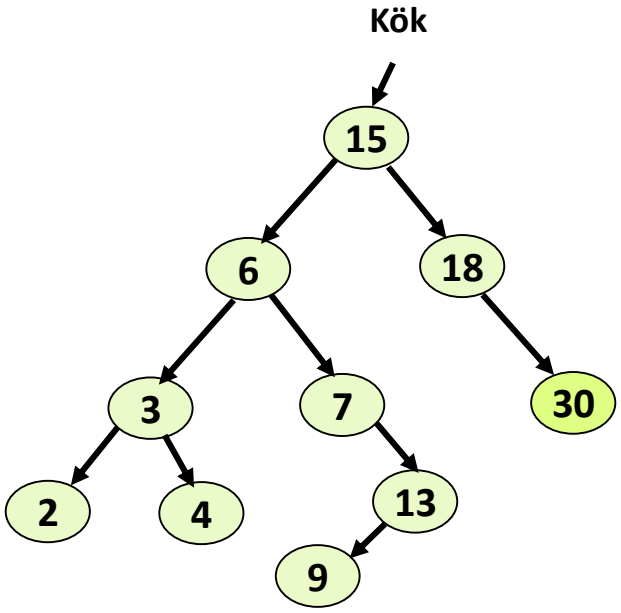
BST İşlemleri-Max

- Ağaçtaki en büyük elemanı içeren düğümü bulur ve geri döndürür.
  - En büyük elemanı içeren düğüm en sağdaki düğümde bulunur.
  - Kökten başlayarak devamlı sağa gidilerek bulunur.

```
public BSTDugum Max(){
  if (kok == null)
    return null;

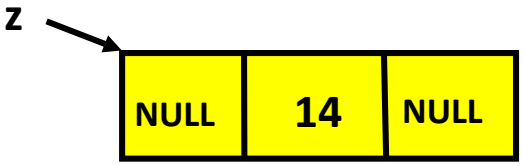
  BSTDugum p = kok;
  while (p.sag != null){
    p = p.sag;
  }

  return p;
}
```

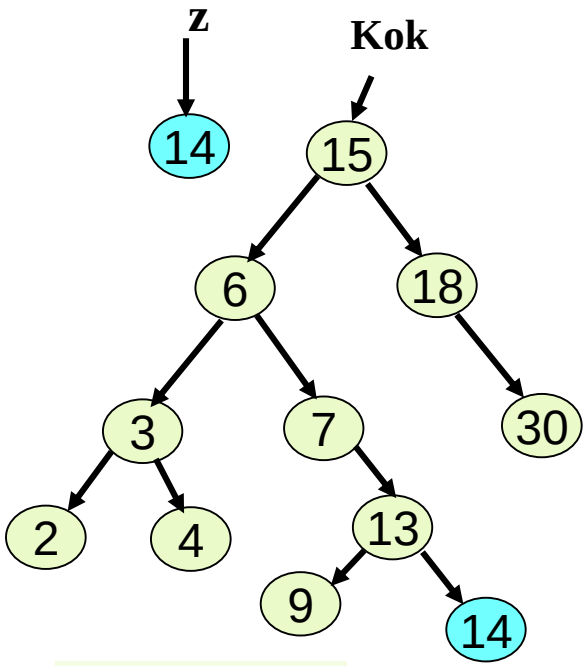


BST İşlemleri– Ekle(int deger)

- Eklenecek değeri içeren “z” isimli yeni bir düğüm oluştur.
- Ö.g.: Ekle 14
- Kökten başlayarak ağaç üzerinde eklenecek sayıyı arıyormuş gibi aşağıya doğru ilerle.
- Yeni düğüm aramanın bittiği düğümün çocuğu olmalıdır.



Eklenecek “z” düğümü.
z.deger = 14



Eklemeden sonra

BST İşlemleri- Ekle(int deger)

```
public void Ekle(int deger) {
  BSTDugum pp = null;  /* pp p'nin ailesi */
  BSTDugum p = kok;   /* Kökten başla ve aşağıya doğru ilerle*/
  while (p) {
    pp = p;
    if (deger == p.deger)      return;  /* Zaten var */
    else if (deger < p.deger)    p = p.sol;
    else /* deger > p.deger */ p = p.sag;
  }
 /* Yeni değeri kaydedeceğimiz düğüm */
  BSTDugum z = new BSTDugum();
  z.deger = deger; z.sol = z.sag = null;
  if (kok == null) kok = z; /* Boş ağaca ekleme */
  else if (deger < pp.deger) pp.sol = z;
  else pp.sag = z;
} // ekleme işlemi bitti.
```

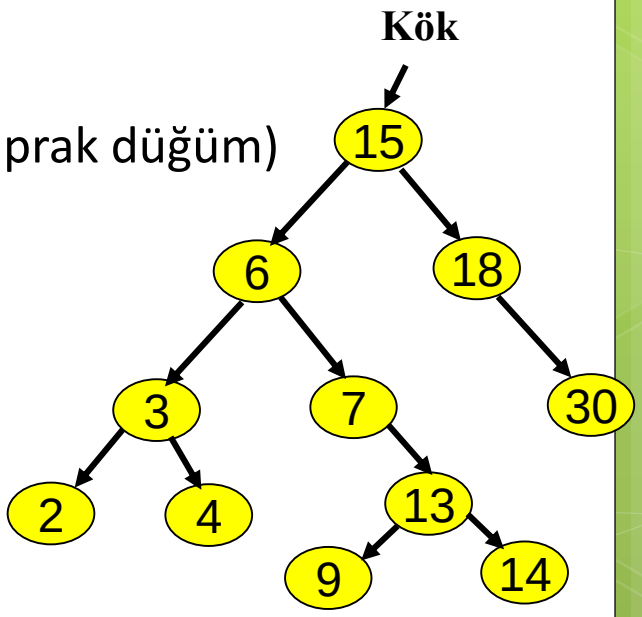
BST-Silme Kaba Kod

- while(silinecek düğümün ve ailesinin adresini bulana kadar) {
- q <- silinecek düğümün, qa<- ailesinin adresi;
- if(silinmek istenen bulunamadı ise)
  - yapacak bir şey yok dön;
- if(silinecek düğümüm iki alt çocuğu da varsa)
  - sol alt ağacın en büyük değerli düğümünü bul;
  - (veya denge bozulmuş ise sağ alt ağacın en küçük değerli düğümünü bul)
  - bu düğümdeki bilgiyi silinmek istenen düğüme aktar;
  - bu aşamada en fazla bir çocuğu olan düğümü sil;
- silinen düğümün işgal ettiği bellek alanını serbest bırak;
  }

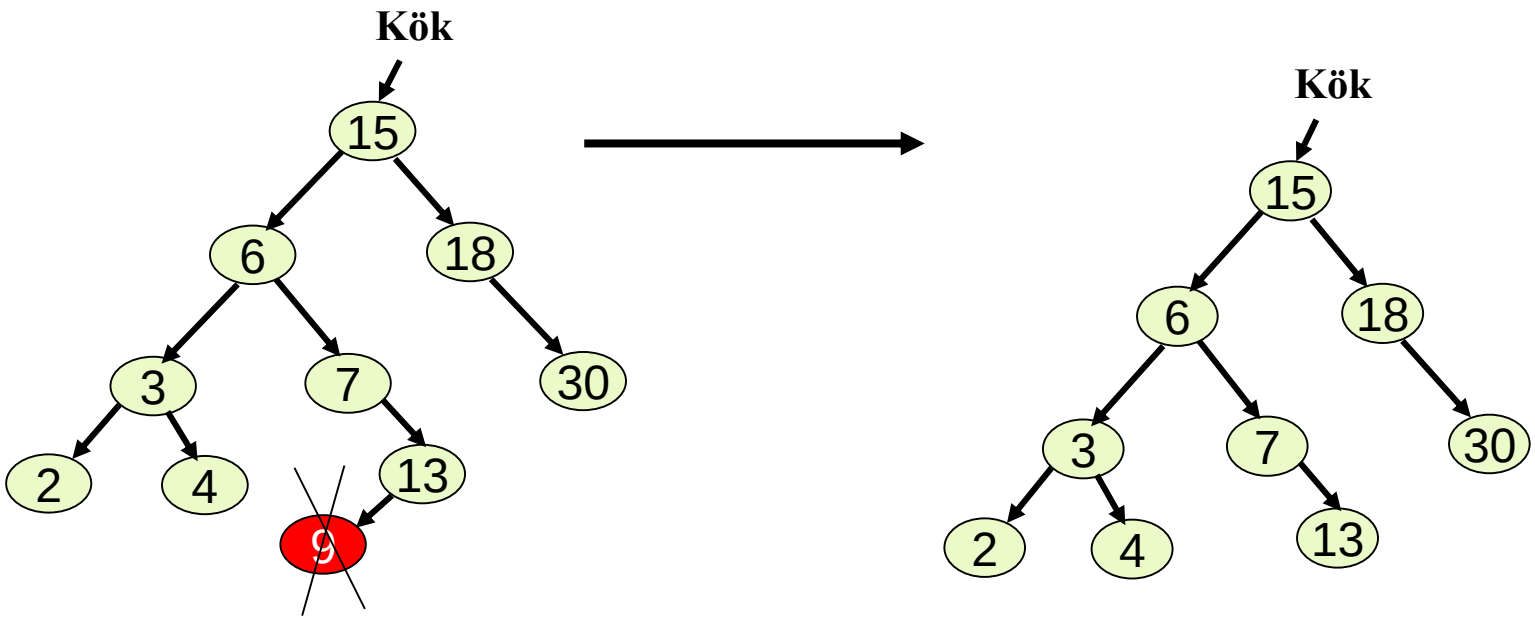
BST İşlemleri- Sil(int deger)

- Silme işlemi biraz karmaşıktır.
- 3 durum var:

1) Silinecek düğümün hiç çocuğu yoksa (yaprak düğüm)
   - Sil 9
2) Silinecek düğümün 1 çocuğu varsa
   - Sil 7
3) Silinecek düğümün 2 çocuğu varsa
   - Sil 6



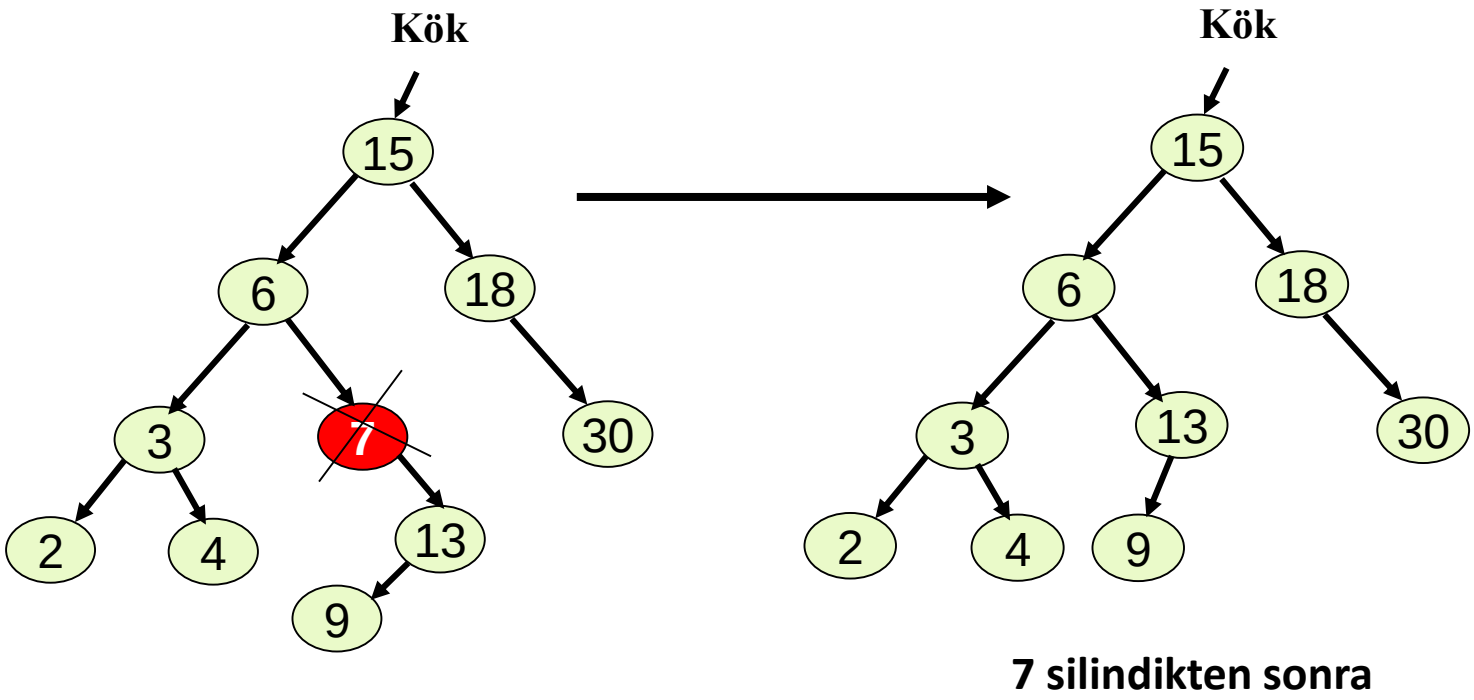
Silme: Durum 1 – Yaprak Düğümü Silme



Sil 9: Düğümü kaldırın ve bağlantı kısmını güncelleyin

9 silindikten sonra

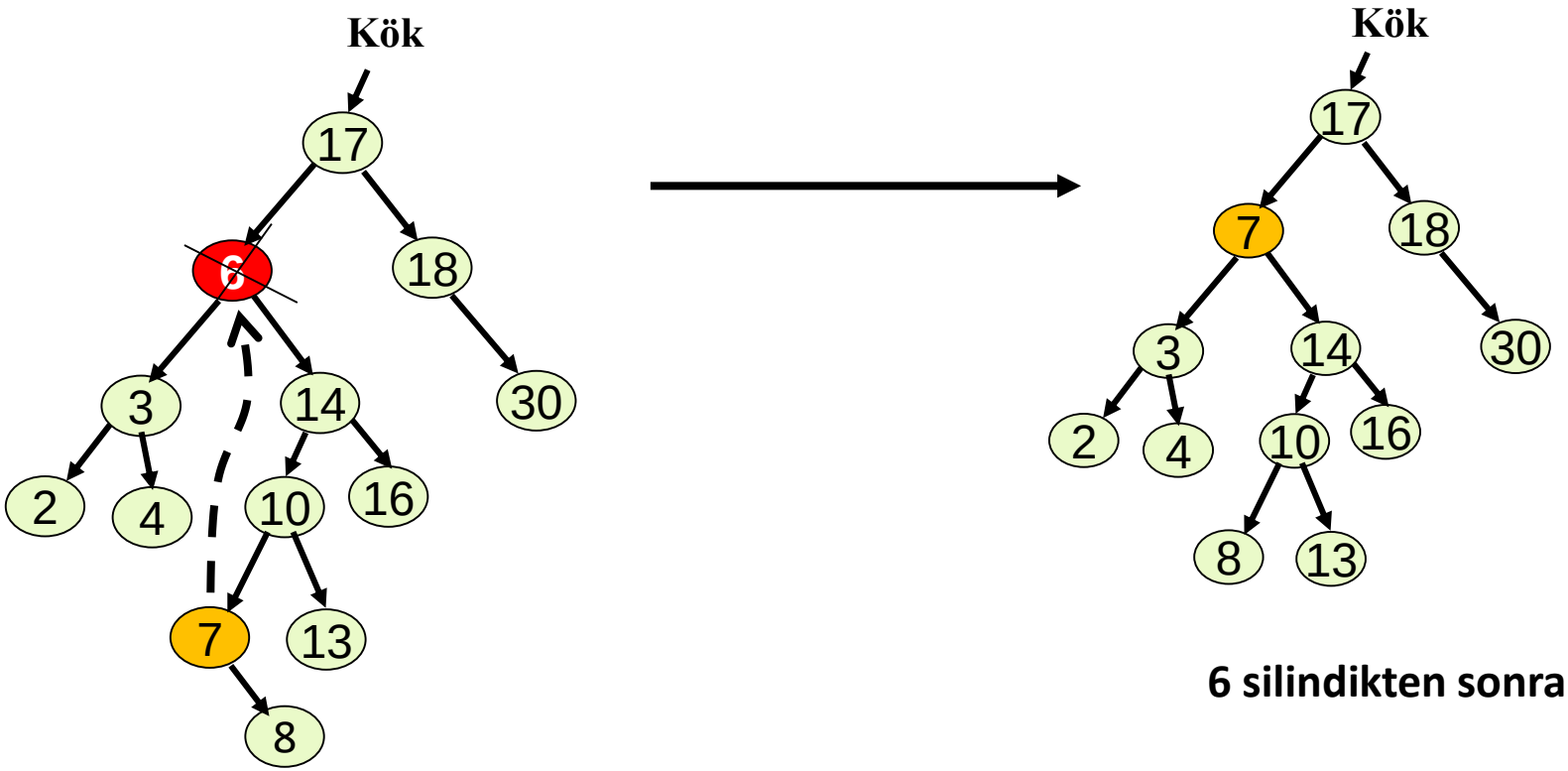
Silme: Durum 2 – 1 Çocuklu Düğüm



7 silindikten sonra

Sil 7: Silinecek düğümün ailesi ve çocuğu arasında bağ kurulur

Silme: Durum 3 – 2 Çocuklu Düğüm



6 silindikten sonra

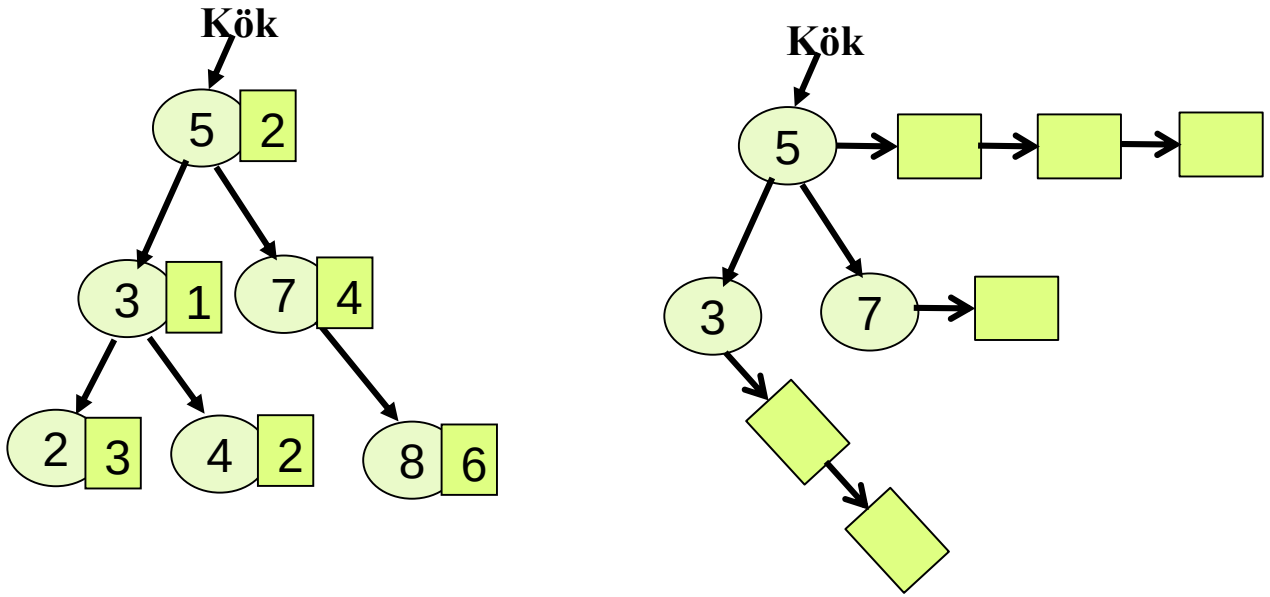
Sil 6:

1) **Sağ alt ağaçtaki en küçük eleman bulunur.(7)**
2) **Bu elemanın sol çocuğu olmayacaktır.**
3) **6 ve 7 içeren düğümlerin içeriklerini değiştirin**
4) **6 nolu eleman 1 çocuğu varmış gibi silinir.**

Not: Sağ alt ağaçtaki en küçük eleman yerine sol alt ağaçtaki en büyük eleman bulunarak aynı işlemler yapılabilir.

BST-Aynı Sayılarla Başa Çıkma

- Ağaç içerisindeki aynı sayılarla aşağıda verilen iki şeklide başa çıkılabilir:
  - Düğümde saklanan bir sayaç değişkeni ile
- veya
  - Düğümde kullanılan bağlantılı liste ile

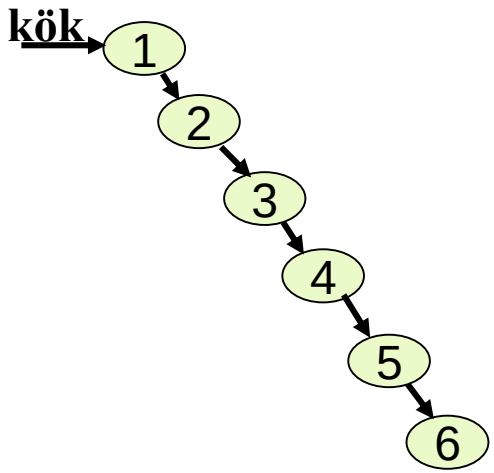


İkili Arama Ağacı Uygulamaları

- İkili arama ağacı harita, sözlük gibi birçok uygulamada kullanılır.
  - İkili arama ağacı (anahtar, değer) çifti şeklinde kullanılacak sistemler için uygundur.
    - Ö.g.: Şehir Bilgi Sistemi
      - Posta kodu veriliyor , şehir ismi döndürülüyor. (posta kodu/ Şehir ismi)
    - Ö.g.: telefon rehberi
      - İsim veriliyor telefon numarası veya adres döndürülüyor. (isim, Adres/Telefon)
    - Ö.g.: Sözlük
      - Kelime veriliyor anlamı döndürülüyor. (kelime, anlam)

İkili Arama Ağacı

- Bul, Min, Max, Ekle, Sil işlemlerinin karmaşıklığı O(d)
- d ağacın derinliğine bağlı.
- Örnek: 1,2,3,4,5,6 sayılarını sıralı bir şekilde ekleyelim.
- Ortaya çıkan ağaç bağlantılı listeye benzemektedir. Dolayısıyla karmaşıklık O(n) şeklinde olacaktır.



- Daha iyisi yapılabilir mi?
- Ağacımızı dengeli yaparsak evet
  1. AVL-ağaçları
  2. Splay ağaçları
  3. 2-3-4 Ağaçları
  4. Red-Black ağaçları
  5. B ağaçları, B+ ağaçları

Ödev

- **1-**İkili arama ağacını Oluşturma-Arama-Ekleme-Silme olaylarını bir bütün olarak gerçekleştiriniz. (Önerilen Ders kitabından yararlanabilirsiniz)
- **2-** İkili binary ağacında minimum ve maksimum değerin bulunduğu düğümün atasını ve kardeşlerini bulan metotları yazınız.
- **3-** İkili binary ağacını level-order dolaşmayı sağlayan metodu yazınız.
- **4-** İkili binary ağacında her düğümün sol ve sağ ağaçları arasındaki denge farklarını bularak bu farkın en çok olduğu düğümü bulunuz.

Örnek Programlar

İkili arama Ağacı–Java

```
class TreeNode  // Düğüm Sınıfı
 {
  public int data;
  public TreeNode leftChild;
  public TreeNode rightChild;
  public void displayNode()  {  System.out.print(" "+data+" ");  }
 }

 // Ağaç Sınıfı
 class Tree
 {
  private TreeNode root;
  public Tree()   { root = null; }

  public TreeNode getRoot()  {  return root;  }
```

İkili arama Ağacı–Java

```
// Ağacın preOrder Dolaşılması
public void preOrder(TreeNode localRoot)
{
 if(localRoot!=null)    {
  localRoot.displayNode();
  preOrder(localRoot.leftChild);
  preOrder(localRoot.rightChild);
 }
}

// Ağacın inOrder Dolaşılması
public void inOrder(TreeNode localRoot)
{
 if(localRoot!=null)
 {
  inOrder(localRoot.leftChild);
  localRoot.displayNode();
  inOrder(localRoot.rightChild);
 }
}
```

İkili arama Ağacı–Java

```
// Ağacın postOrder Dolaşılması
public void postOrder(TreeNode localRoot)
{
  if(localRoot!=null)
  {
    postOrder(localRoot.leftChild);
    postOrder(localRoot.rightChild);
    localRoot.displayNode();
  }
}

// Ağaca bir düğüm eklemeyi sağlayan metot
public void insert(int newdata)
{
  TreeNode newNode = new TreeNode();
  newNode.data = newdata;

if(root==null)
    root = newNode;
```

İkili arama Ağacı–Java

```
else    {
    TreeNode current = root;      TreeNode parent;
    while(true)     {
     parent = current;
     if(newdata<current.data)
     {  current = current.leftChild;
      if(current==null)
      {
       parent.leftChild=newNode;
       return;
      }
     } else
     {
      current = current.rightChild;
      if(current==null)
      {          parent.rightChild=newNode;          return;        }
     }
    }  // end while
   } // end else not root
  } // end insert()
} // class Tree
```

İkili arama Ağacı–Java

```
// BinTree Test sınıfı
class BinTree
{
  public static void main(String args[])
  {
    Tree theTree = new Tree();

    // Ağaca 10 tane sayı yerleştirilmesi
    System.out.println("Sayılar : ");
    for (int i=0;i<10;++i) {
      int sayi = (int) (Math.random()*100);
      System.out.print(sayi+" ");
      theTree.insert(sayi);
    };

    System.out.print("\nAğacın InOrder Dolaşılması   : ");
    theTree.inOrder(theTree.getRoot());
    System.out.print("\nAğacın PreOrder Dolaşılması  : ");
    theTree.preOrder(theTree.getRoot());
    System.out.print("\nAğacın PostOrder Dolaşılması : ");
    theTree.postOrder(theTree.getRoot());
  }
}
```

İkili arama Ağacı– C++

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
/* Ağaca ait düğüm yapısı tanımlanıyor */
struct agacdugum {
  struct agacdugum *soldal;
  int data;
  struct agacdugum *sagdal;
};
// Düğüm yapısı için değişken tanımlarının yapıldığı kısım
typedef struct agacdugum AGACDUGUM;
typedef struct agacdugum * AGACDUGUMPTR;
```

İkili arama Ağacı– C++

```
// Ağaca düğüm eklemeyi saglayan fonksiyon yapisi
AGACDUGUMPTR dugumekle(AGACDUGUMPTR agacptr, int veri) {
/* her defasında tek dallı ağaç oluşturuluyor. Daha sonra ikili arama ağacındaki kurala göre sol veya sağ dala yerleştiriliyor. */

 if(agacptr==NULL)
  {
/*eger agaç işaretçisi boş ise ağaca eklenecek yeni düğüm için hafızada yer ayrılıyor*/

    agacptr =(agacdugum *) malloc(sizeof(agacdugum));
   if (agacptr!=NULL)
   {
  // Düğümler tek hücre, sağ ve sol dalları boş olarak oluşturuluyor
  //  printf("Ağaça veri eklendi\n ");
     agacptr->data = veri;
     agacptr->soldal = NULL;
     agacptr->sagdal= NULL;
   }
   else printf("%d eklenemedi. Bellek yetersiz.\n",veri);
  }
```

İkili arama Ağacı– C++

```
/*Gelen veri değeri daha önce girilen değerler ile karşılaştırılıp
uygun düğümün sol veya sağ dalına yerleştiriliyor.*/
  else

    if(veri<agacptr->data){ printf("Ağaçın soluna veri eklendi\n ");
      agacptr->soldal = dugumekle(agacptr->soldal,veri);}
    else

    if(veri>agacptr->data){printf("Ağaçın sağına veri eklendi\n ");
      agacptr->sagdal = dugumekle(agacptr->sagdal,veri);}

// eğer girilen değer ler daha önce var ise alınmıyor.
    else printf("Eşit olduğu için alınmadı\n ");
    return agacptr;
}
```

İkili arama Ağacı– C++

```
/* Agacın inorder dolaşılması */
void inorder(AGACDUGUMPTR agacptr) {
 if (agacptr != NULL) {
   inorder(agacptr->soldal);
   printf("%3d",agacptr->data);
   inorder(agacptr->sagdal);   }            }
/* Ağacın preorder dolaşılması */
void preorder(AGACDUGUMPTR agacptr) {
 if (agacptr != NULL) {
   printf("%3d",agacptr->data);
   preorder(agacptr->soldal);
   preorder(agacptr->sagdal);  }            }
/* Ağacın postorder dolaşılması */
void postorder(AGACDUGUMPTR agacptr) {
 if (agacptr != NULL) {
   postorder(agacptr->soldal);
   postorder(agacptr->sagdal);
   printf("%3d",agacptr->data); }            }
```

İkili arama Ağacı– C++

```
void main() {
    int i, dugum;
    AGACDUGUMPTR agacptr = NULL;
    for(i=0; i<12; ++i)
     { /* Ağaca yerleştirilecek sayılar */
       scanf("%d",&dugum);  printf("\n");
    // girilen değeri düğüm ekleme fonksiyonuna gönderiyoruz.
       agacptr = dugumekle(agacptr, dugum);
    }  printf("\n");

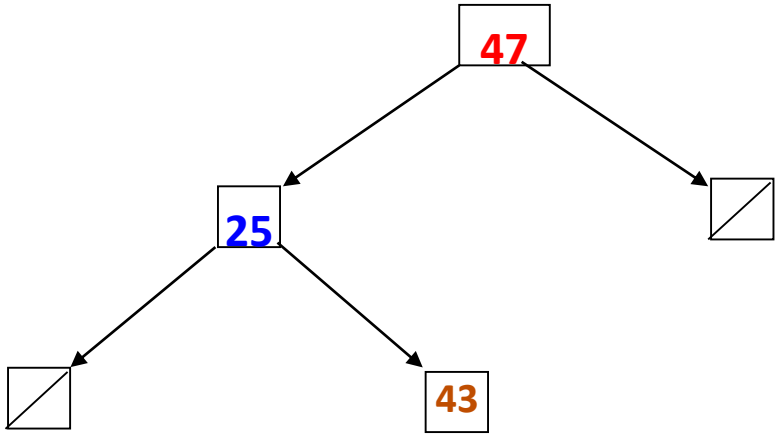
     printf("Ağacın preorder dolaşılması  :\n");
      preorder(agacptr); printf("\n");

      printf("Ağacın inorder dolaşılması   :\n");
      inorder(agacptr); printf("\n");

      printf("Ağacın postorder dolaşılması :\n");
      postorder(agacptr); printf("\n");
    }
```

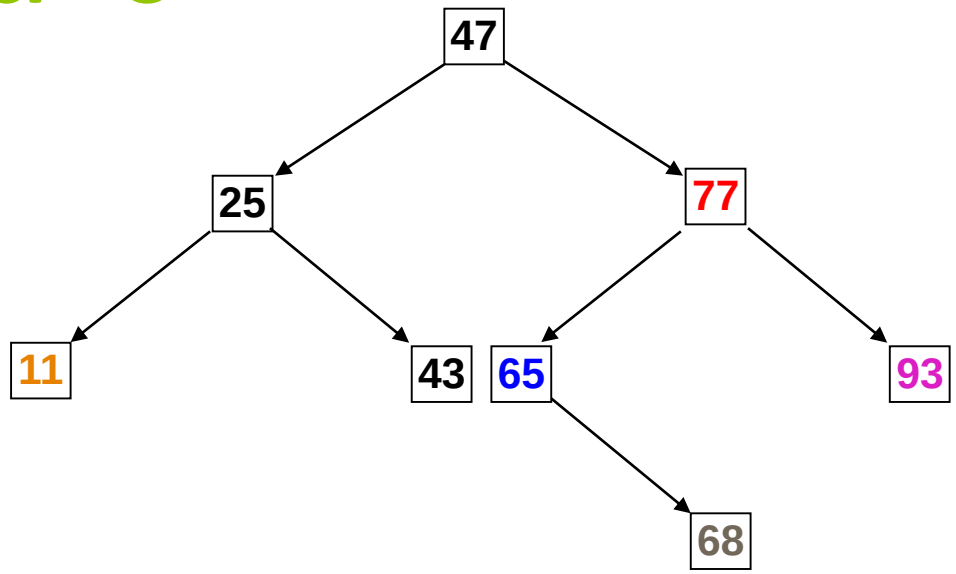
İkili arama Ağacı– C++

- 47
- 25
- Ağaçın soluna veri eklendi
- 43
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi



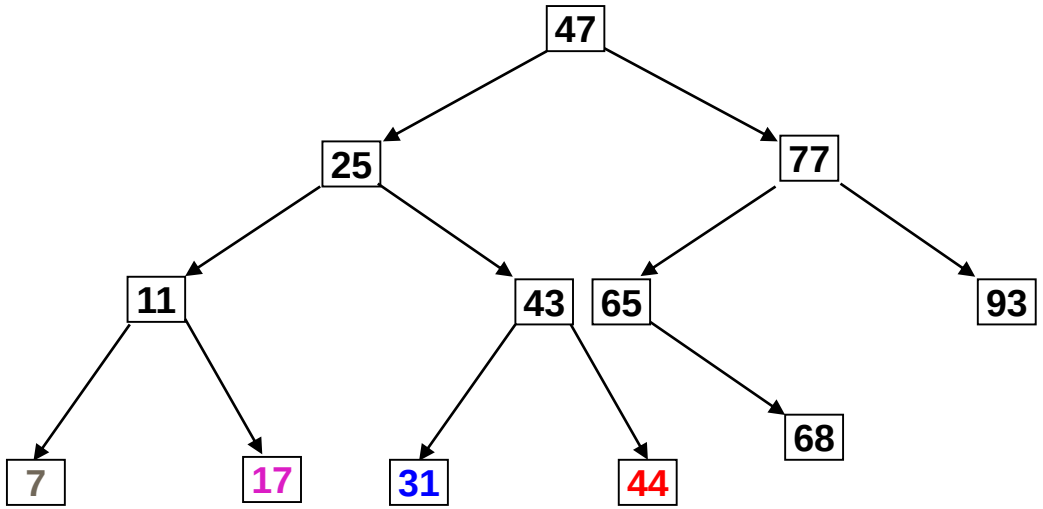
İkili arama Ağacı– C++

- 77
- Ağaçın sağına veri eklendi
- 65
- Ağaçın sağına veri eklendi
- Ağaçın soluna veri eklendi
- 68
- Ağaçın sağına veri eklendi
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi
- 93
- Ağaçın sağına veri eklendi
- Ağaçın sağına veri eklendi
- 11
- Ağaçın soluna veri eklendi
- Ağaçın soluna veri eklendi



İkili arama Ağacı– C++

- 17
- Ağaçın soluna veri eklendi
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi
- 44
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi
- Ağaçın sağına veri eklendi
- 31
- Ağaçın soluna veri eklendi
- Ağaçın sağına veri eklendi
- Ağaçın soluna veri eklendi
- 7



İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
#include <stdio.h>
#include <stdlib.h>
struct tnode {
 int data;
 struct tnode *lchild, *rchild; };
/* Dugum degeri verilen dugumun kok pointer degerini elde eden bir foksiyondur*/
struct tnode *getptr(struct tnode *p, int key, struct tnode **y) {
printf("kok pointer\n");//silme anında devreye giriyor
 struct tnode *temp;
   if( p == NULL)      return(NULL);
   temp = p;    *y = NULL;
    while( temp != NULL)     {
      if(temp->data == key)   return(temp);
      else     {
         *y = temp; /* bu pointer root (kok) olarak depolanir */
   if(temp->data > key)
     temp = temp->lchild;
   else
     temp = temp->rchild;      }     }
    return(NULL);  }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* Veri degeri verilen dugumu silmek icin bir fonksiyon */
struct tnode *delete1(struct tnode *p,int val)  {
   struct tnode *x, *y, *temp;
   x = getptr(p,val,&y);
   if( x == NULL)   {     printf("Dugum mevcut degil\n");     return(p);    }
   else
   {
   /* bu kod kok (root) dugumu silmek icindir*/
   if( x == p)     {
     printf("kok dugum siliniyor\n");
      temp = x->lchild;
      y = x->rchild;
      p = temp;
      while(temp->rchild != NULL)     temp = temp->rchild;
    temp->rchild=y;
    free(x);
    return(p);
   }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* bu kod dugumun sahip oldugu cocukları silmek icindir.*/
 if( x->lchild != NULL && x->rchild != NULL)    {
      if(y->lchild == x)      {
temp = x->lchild;
y->lchild = x->lchild;
while(temp->rchild != NULL)    temp = temp->rchild;

temp->rchild=x->rchild;
x->lchild=NULL;
x->rchild=NULL;
      }
      else       {
        temp = x->rchild;
        y->rchild = x->rchild;
         while(temp->lchild != NULL)
     temp = temp->lchild;
   temp->lchild=x->lchild;
x->lchild=NULL;
 x->rchild=NULL;
       }
free(x);
   return(p);   }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* bu kod cocukları ile birlikte bir dugum siliyor*/
  if(x->lchild == NULL && x->rchild !=NULL)
    {
      if(y->lchild == x)
   y->lchild = x->rchild;
        else
          y->rchild = x->rchild;
        x->rchild=NULL;
        free(x);
        return(p);
      }
      if( x->lchild != NULL && x->rchild == NULL)
       {
        if(y->lchild == x)
          y->lchild = x->lchild ;
        else
          y->rchild = x->lchild;
        x->lchild = NULL;
        free(x);
        return(p);
            }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* bu kod cocukları olmadan bir dugumu siliyor*/
      if(x->lchild == NULL && x->rchild == NULL)
       {
         if(y->lchild == x)
           y->lchild = NULL ;
         else
           y->rchild = NULL;
         free(x);
         return(p);
       }
   }
}
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/*inorder binary agacını tekrarlamalı olarak yazdıran bir fonksiyon*/
void inorder1(struct tnode *p) {
 struct tnode *stack[100]; //yigin
 int top;
 top = -1;
if(p != NULL)   {
   top++;
   stack[top] = p;
   p = p->lchild;
   while(top >= 0)       {
        while ( p!= NULL)/* sol cocuk yigindan (stack dizisinden) cikariliyor*/
         {     top++;
              stack[top] =p;
              p = p->lchild;   }
    p = stack[top];
    top--;
    printf("%d\t",p->data);
    p = p->rchild;
    if ( p != NULL) /* sag cocuk cikariliyor*/
      {     top++;
      stack[top] = p;
         p = p->lchild;    }   }    }   }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
/* Olusturulan agaca yeni bir dugum ilave etmek icin olusturulan bir fonksiyon*/

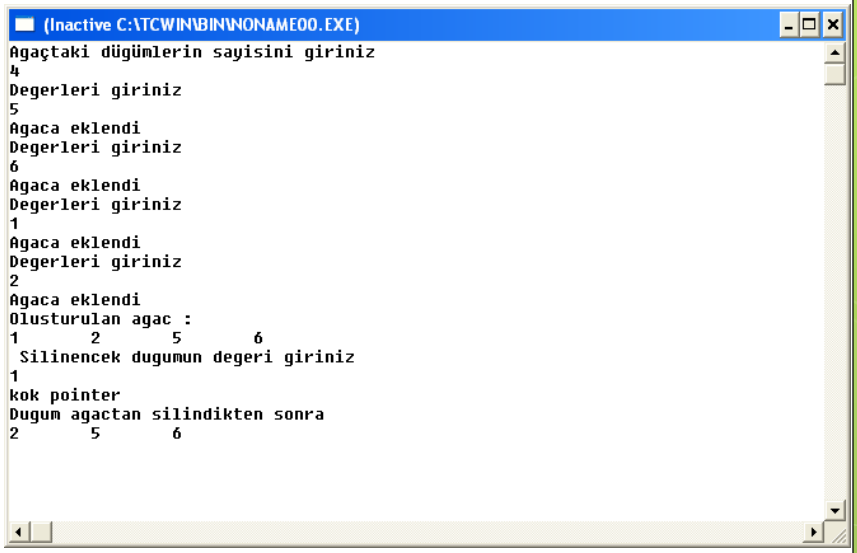
 struct tnode *insert(struct tnode *p,int val)  {
  printf("Agaca eklendi\n");
    struct tnode *temp1,*temp2;
    if(p == NULL)     {
      p = (struct tnode *) malloc(sizeof(struct tnode)); /* koke bir dugum ilave ediliyor*/
      if(p == NULL)         {
    printf("Erisim izni yok\n");
    exit(0);         }
      p->data = val;
      p->lchild=p->rchild=NULL;  }
  else
 {
    temp1 = p;
  /* child (cocuk) olacak dugumun pointer(gostergesini) almak icin agacta dolasma*/
 while(temp1 != NULL)
  {
   temp2 = temp1;
   if( temp1 ->data > val)      temp1 = temp1->lchild;
   else     temp1 = temp1->rchild;
  }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
if( temp2->data > val)   {
    temp2->lchild = (struct tnode*)malloc(sizeof(struct tnode));/*Sol cocuk (left child) olarak yeni olusturulan dugumu ekler*/
    temp2 = temp2->lchild;
    if(temp2 == NULL)         {
    printf("Erisim izni yok\n");
    exit(0);         }
   temp2->data = val;
   temp2->lchild=temp2->rchild = NULL;   }
  else  {
     temp2->rchild = (struct tnode*)malloc(sizeof(struct tnode));/*Sag cocuk (right child) olarak yeni olusturulan dugumu ekler*/
    temp2 = temp2->rchild;
    if(temp2 == NULL)         {
    printf("Erisim izni yok\n");
    exit(0);        }
   temp2->data = val;
   temp2->lchild=temp2->rchild = NULL;   }   }
  return(p);
  }
```

İkili arama Ağacı– C++, Ekleme, Silme, Dolaşma

```
void main()
 {
   struct tnode *root = NULL;
   int n,x;
   printf("Agaçtaki dügümlerin sayisini giriniz\n");
   scanf("%d",&n);
   while( n-->0)
     {
       printf("Degerleri giriniz\n");
       scanf("%d",&x);
       root = insert(root,x);
     }
    printf("Olusturulan agac :\n");
    inorder1(root);
    printf("\n Silinencek dugumun degeri giriniz\n");
    scanf("%d",&n);
    root=delete1(root,n);
    printf("Dugum agactan silindikten sonra \n");
    inorder1(root);
}
```



İkili arama Ağacı– C# (1. Örnek)

- **İkilik Arama Ağacı Oluşturmak**
- **class bstNodeC**
- **{**
- **public int sayi;**
- **public bstNodeC leftNode, rightNode;**
- **public bstNodeC(int sayi, bstNodeC leftNode, bstNodeC rightNode)**
- **{**
- **this.sayi = sayi;**
- **this.leftNode = leftNode;**
- **this.rightNode = rightNode;**
- **}**
- **}**

İkili arama Ağacı– C# (1. Örnek)

```
private void preOrder(bstNodeC node)
{
listBox1.Items.Add(node.sayi);
if (node.leftNode != null) preOrder(node.leftNode);
if (node.rightNode != null) preOrder(node.rightNode);
}
private void inOrder(bstNodeC node)
{
if (node.leftNode != null) inOrder(node.leftNode);
listBox1.Items.Add(node.sayi);
if (node.rightNode != null) inOrder(node.rightNode);
}
private void postOrder(bstNodeC node)
{
if (node.leftNode != null) postOrder(node.leftNode);
if (node.rightNode != null) postOrder(node.rightNode);
listBox1.Items.Add(node.sayi);
}
```

İkili arama Ağacı– C# (1. Örnek)

```
//Boşaltma (makeEmpty)
private void makeEmpty(bstNodeC node)
{
if (node.leftNode node.leftNode != null)  makeEmpty(node.leftNode );
if (node.rightNode node.rightNode != null) makeEmpty(node.rightNode);
node = null;
}
//Arama (find)
private void find(int sayi, bstNodeC node)
{
if (node.sayi == 0) textBox2.Text = "Boş ağaç";
else if ((sayi < node.sayi)&&(node.leftNode!=null))
find(sayi, node.leftNode);
else if ((sayi > node.sayi)&&(node.rightNode!=null))
find(sayi, node.rightNode);
else if (sayi == node.sayi) textBox2.Text = “Sayı bulundu";
else textBox2.Text = “Sayı yok";
}
```

İkili arama Ağacı– C# (1. Örnek)

```
//Ekleme (append)
private void append(int sayi, bstNodeC node) {
bstNodeC yeniNode = new bstNodeC(sayi, null, null);
if (node.sayi == 0) node.sayi = sayi;
else  {
bstNodeC current = node; bstNodeC parent;
while(true)
{   parent = current;
if(sayi < current.sayi)
{ current = current.leftNode;
if(current == null)  {parent.leftNode = yeniNode;break; }
}
else
{ current = current.rightNode;
if(current==null) { parent.rightNode = yeniNode; return; }
}
}
}
}
```

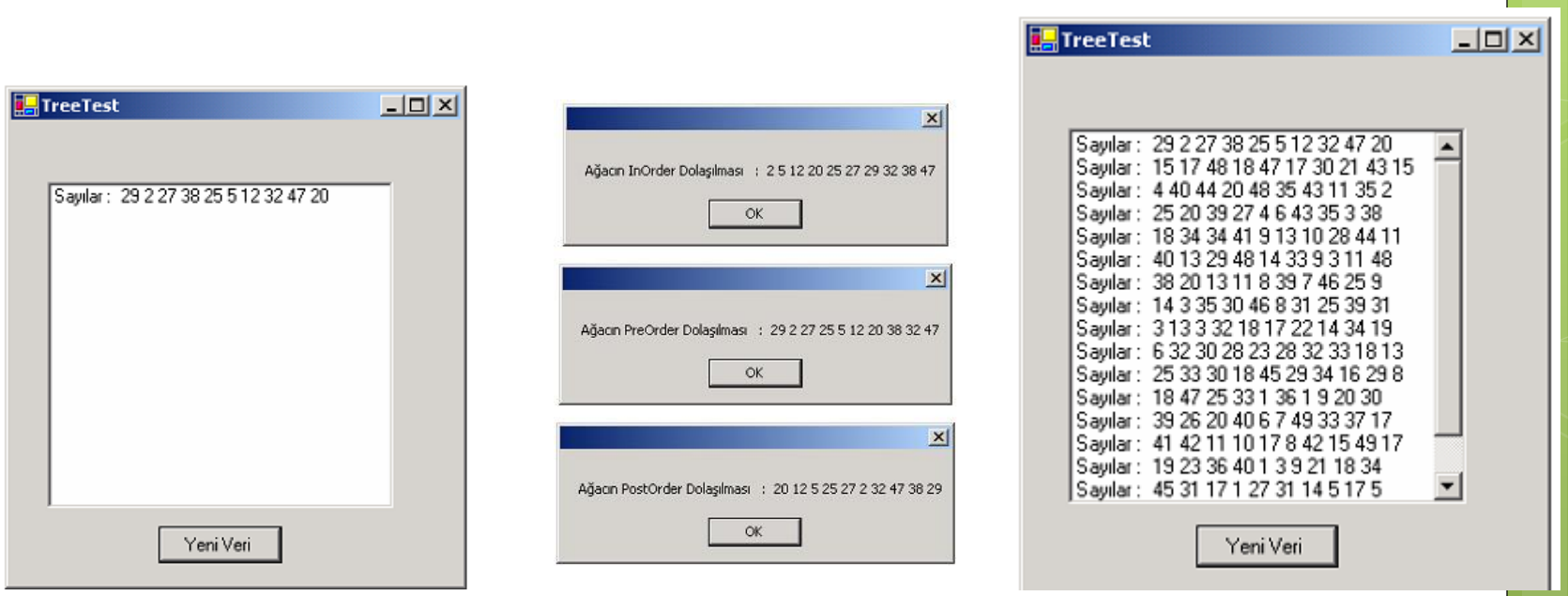
İkili arama Ağacı– C# (1. Örnek)

```
//Ekleme (append)
private void append(int sayi, bstNodeC node) {
bstNodeC yeniNode = new bstNodeC(sayi, null, null);
if (node.sayi == 0) node.sayi = sayi;
else  {
bstNodeC current = node; bstNodeC parent;
while(true)
{   parent = current;
if(sayi < current.sayi)
{ current = current.leftNode;
if(current == null)  {parent.leftNode = yeniNode;break; }
}
else
{ current = current.rightNode;
if(current==null) { parent.rightNode = yeniNode; return; }
}
}
}
}
```



İkili arama Ağacı– C# (2. Örnek)

- “TreeTest” adlı programın formu üzerinde 1 adet liste kutusu ve 1 adet düğme bulunmaktadır. “Yeni Veri” düğmesine basıldıkça, 1 ile 50 arasında 10 tane rastgele sayı üretmektedir ve arka arkaya gelen üç mesaj penceresi ile, InOrder, PreOrder ve PostOrder dolaşmalarda ikili ağaç üzerinde hangi düğüm sırasının izleneceğini göstermektedir.



İkili arama Ağacı– C# (2. Örnek)

```
public class GLOBAL   {      public static string tempStr;   }
class TreeNode
{   public int data;     public TreeNode leftChild;     public TreeNode rightChild;
  public void displayNode()     { GLOBAL.tempStr += (" "+data); }
 }
 // Ağaç Sınıfı
class Tree
{  private TreeNode root;
   public Tree()     { root = null; }
   public TreeNode getRoot()     { return root; }
// Ağacın preOrder Dolaılması
public void preOrder(TreeNode localRoot)
{
  if(localRoot!=null)
  {
    localRoot.displayNode();
    preOrder(localRoot.leftChild);
    preOrder(localRoot.rightChild);
  }
 }
```

İkili arama Ağacı– C# (2. Örnek)

```
// Ağacın inOrder Dolaşılması
  public void inOrder(TreeNode localRoot)
  {
   if(localRoot!=null)
   {
    inOrder(localRoot.leftChild);
    localRoot.displayNode();
    inOrder(localRoot.rightChild);
   }
  }
  // Ağacın postOrder Dolaşılması
  public void postOrder(TreeNode localRoot)
  {
   if(localRoot!=null)
   {
    postOrder(localRoot.leftChild);
    postOrder(localRoot.rightChild);
    localRoot.displayNode();
   }
  }
```

İkili arama Ağacı– C# (2. Örnek)

```
// Ağaca bir düğüm eklemeyi sağlayan metot
public void insert (int newdata)
{
 TreeNode newNode = new TreeNode();
 newNode.data = newdata;
 if(root==null)     root = newNode;
 else
 {
  TreeNode current = root;
  TreeNode parent;
  while(true)
  {
   parent = current;
   if(newdata<current.data)
   {
    current = current.leftChild;
    if(current==null)
```

İkili arama Ağacı– C# (2. Örnek)

```
{        parent.leftChild=newNode;
          return;
         }
        }
        else
        {
         current = current.rightChild;
         if(current==null)
         {
          parent.rightChild=newNode;
          return;
         }
        }
      }  // end while
     } // end else not root
   } // end insert()
} // class Tree
```

İkili arama Ağacı– C# (2. Örnek)

```
private void dugme1_Click(object sender, System.EventArgs e)     {
    Random r = new Random();            Tree theTree = new Tree();
    // Ağaca 10 tane sayı yerleştirilmesi
    string str = "";        str += "Sayılar : ";
    for (int i=0;i<10;++i)
    {
      int sayi = (int) (r.Next(1,50));        str += (" "+sayi);        theTree.insert(sayi);
    }
     listBox1.Items.Add(str);      GLOBAL.tempStr = "";
    theTree.inOrder(theTree.getRoot());
    MessageBox.Show("\nAğacın InOrder Dolaşılması   : "+GLOBAL.tempStr);

   GLOBAL.tempStr = "";       theTree.preOrder(theTree.getRoot());
    MessageBox.Show("\nAğacın PreOrder Dolaşılması   : "+GLOBAL.tempStr);

    GLOBAL.tempStr = "";       theTree.postOrder(theTree.getRoot());
    MessageBox.Show("\nAğacın PostOrder Dolaşılması   : "+GLOBAL.tempStr);
  }
```

Dengeli Arama Ağaçları

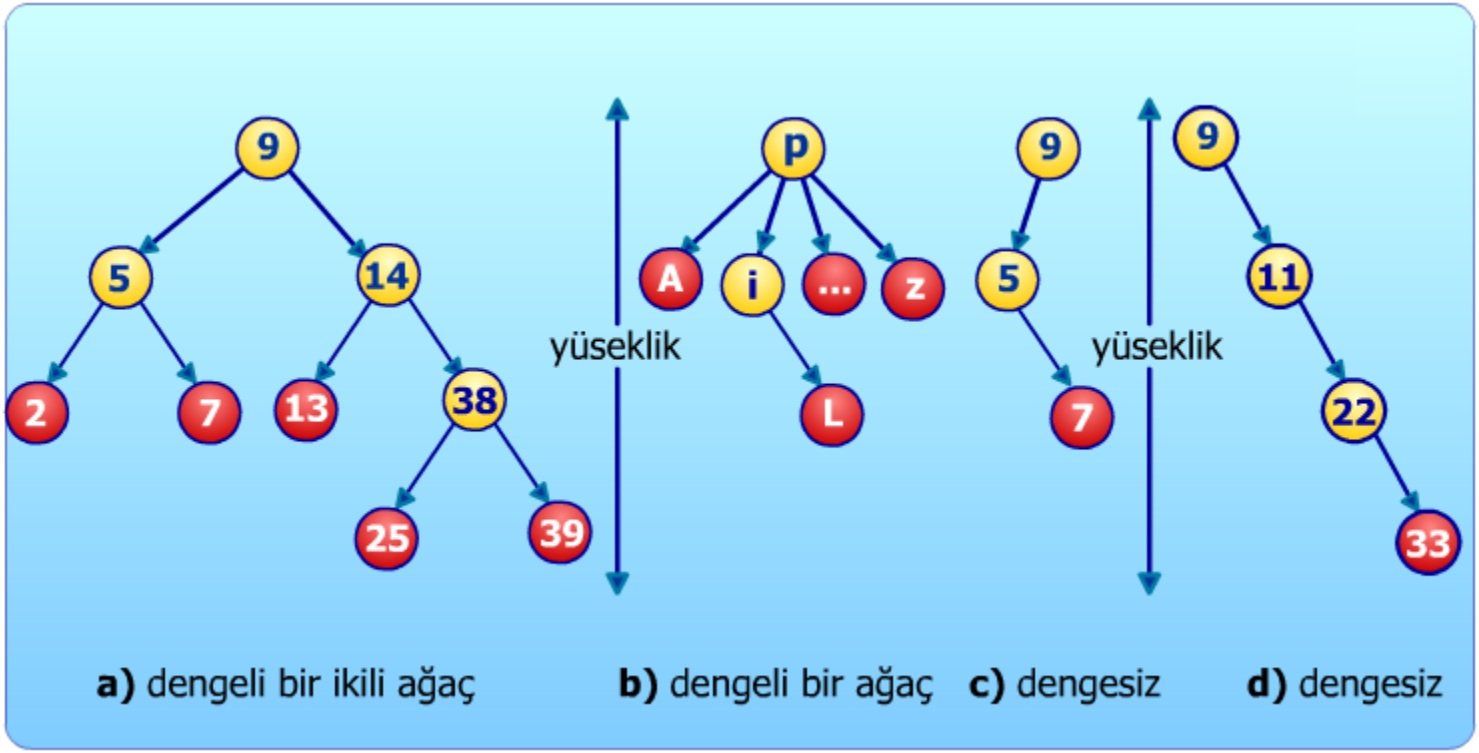
(Balanced Search Tree)

- AVL tree
- 2-3 ve 2-3-4 tree
- Splay tree
- Red-Black Tree
- B- tree

DENGELİ AĞAÇ (BALANCED TREE)

- Dengeli ağaç (balanced tree), gelişmesini tüm dallarına homojen biçimde yansıtan ağaç şeklidir; tanım olarak, herhangi bir düğümüne bağlı altağaçların yükseklikleri arasındaki fark, şekil a) ve b)'de görüldüğü gibi, **en fazla 1 (bir)** olmalıdır. Bir dalın fazla büyümesi ağacın dengesini bozar ve ağaç üzerine hesaplanmış karmaşıklık hesapları ve yürütme zamanı bağıntılarından sapılır. Dolayısıyla ağaç veri modelinin en önemli getirisi kaybolmaya başlar.
- Yapılan istatistiksel çalışmalarda, ağacın oluşturulması için gelen verilerin rastgele olması durumda ağacın dengeli olduğu veya çok az sapma gösterdiği gözlenmiştir [HIBBARD-1962].

DENGELİ AĞAÇ (BALANCED TREE)



Yükseklik Dengeli Ağaçlar

- BST operasyonlarını daha kısa sürede gerçekleştirmek için pek çok BST dengeleme algoritması vardır. Bunlardan bazıları;
  - Adelson-Velsky ve Landis (AVL) ağaçları (1962)
  - Splay ağaçları (1978)
  - B-ağacı ve diğer çok yönlü arama ağaçları (1972)
  - Red-Black ağaçları (1972)
    - Ayrıca Simetrik İkili B-Ağaçları(Symmetric Binary B-Trees) şeklinde de bilinir

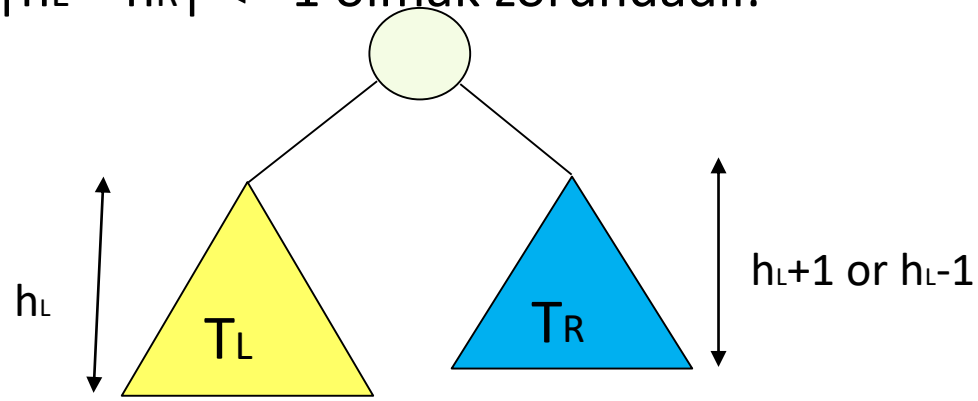
AVL TREE

AVL AĞAÇLARI

- AVL(G.M. Adelson-Velskii and E.M. Landis) yöntemine göre kurulan bir ikili arama ağacı, ikili AVL arama ağacı olarak adlandırılır.
- AVL ağacının özelliği, sağ alt ağaç ile sol alt ağaç arasındaki yükseklik farkının en fazla bir düğüm olmasıdır. Bu kural ağacın tüm düğümleri için geçerlidir.
- Normal ikili arama ağaçları için ekleme ve silme algoritmaları, ikili AVL ağaçları üzerinde ağacın yanlış şekil almasına, yani ağacın dengesinin bozulmasına neden olur.

AVL Ağaçları: Tanım

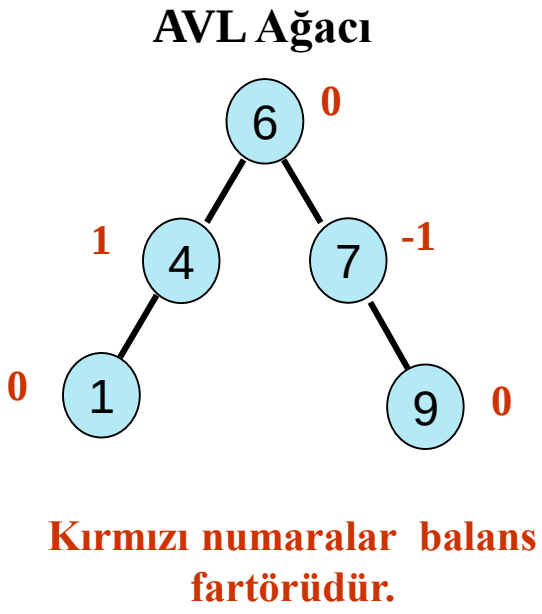
1. Tüm boş ağaç AVL ağacıdır.
2. Eğer T boş olmayan T_L ve T_R şeklinde sol ve sağ alt ağaçları olan ikili arama ağacı ise, T ancak ve ancak aşağıdaki şartları sağlarsa AVL ağacı şeklinde isimlendirilir.
    1. T_L ve T_R AVL ağaçları ise
    2. h_L ve h_R T_L ve T_R nin yükseklikleri olmak üzere $|h_L - h_R| <= 1$ olmak zorundadır.



AVL Ağaçları

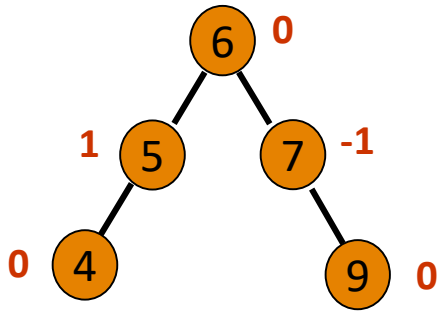
- AVL ağaçları dengeli ikili arama ağaçlarıdır.
- solh= yükseklik(sol altağaç), ve sagh=yükseklik(sağ altağaç) ise
- Bir düğümdeki denge faktörü= solh - sagh

- AVL ağaçlarında balans faktörü sadece -1, 0, 1 olabilir.
  - **Eşit ise** **0**
  - **Sol sazla ise** **1**
  - **Sağ fazla ise** **-1**

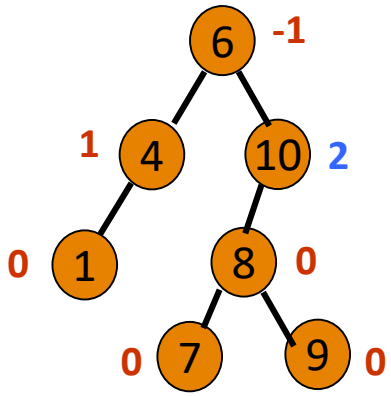


- Her bir düğümün sol ve sağ alt ağaçlarının yükseklikleri arasındaki fark en fazla 1 olabilir.

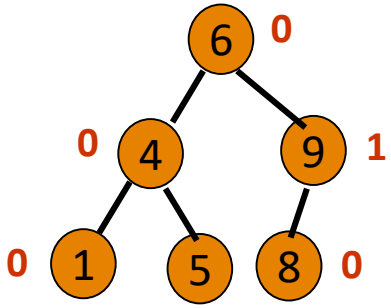
AVL Ağaçları: Örnekler



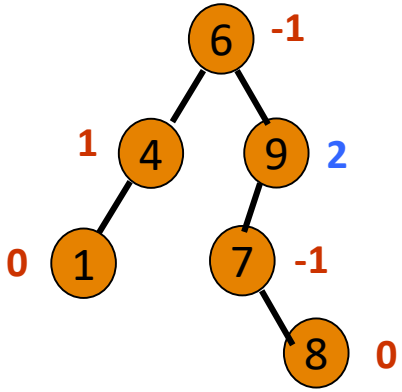
AVL Ağacı



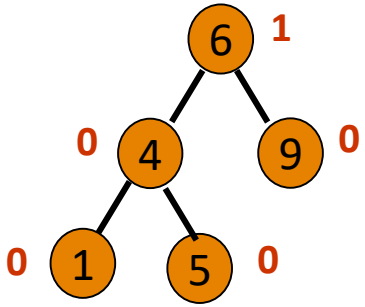
AVL Ağacı Değildir



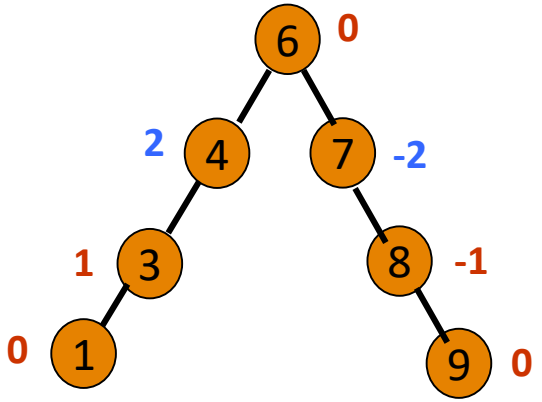
AVL Ağacı



AVL Ağacı Değildir



AVL Ağacı

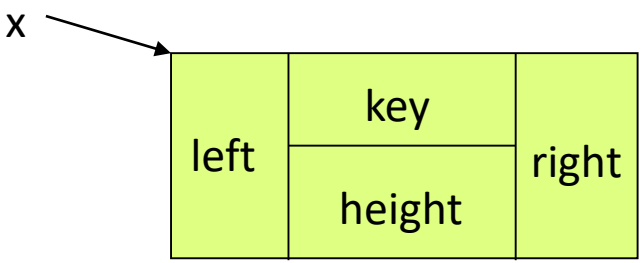


AVL Ağacı Değildir

Kırmızı numaralar balans faktörüdür.

AVL Ağacı: Gerçekleştirim

- AVL ağacının gerçekleştirimi için her x düğümünün yüksekliği kaydedilir.
- x'in balans faktörü = x'in sol alt ağacının yüksekliği – x'in sağ alt ağacının yüksekliği
- AVL ağaçlarında, "bf" sadece {-1, 0, 1} değerlerini alabilir.



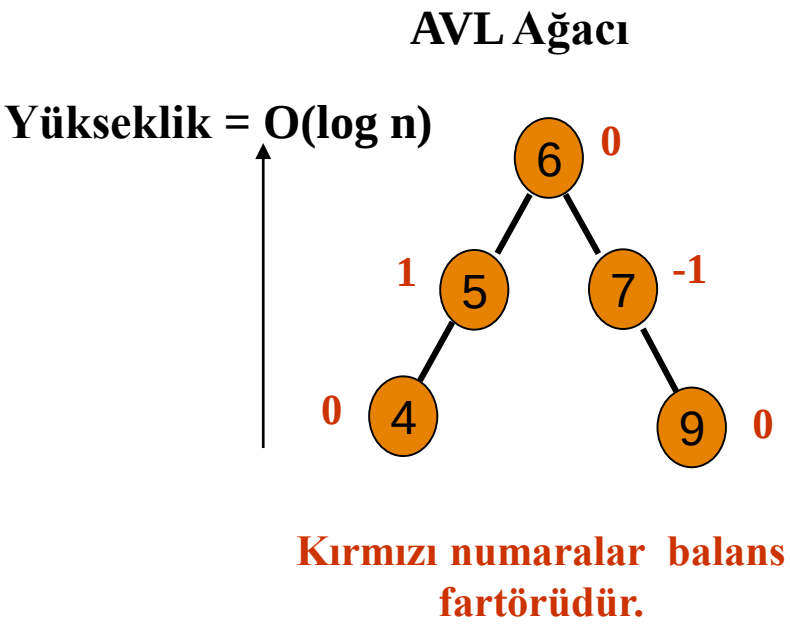

```
class AVLDugumu
{
        int deger;
        int yukseklik;
        AVLDugumu sol;
        AVLDugumu sag;
}
```

AVL AĞAÇLARI

- Normal ikili arama ağaçlarında fonksiyonların en kötü çalışma süreleri Tw = O (n) dir. Bu durum lineer zaman olduğundan çok kötü bir sonuç yaratır.
- AVL ağaç veri modeli oluşturularak bu kötü çalışma süresi ortadan kaldırılır.
- AVL ağaçları için Tw = O (log n)'dir.

AVL Ağaçları

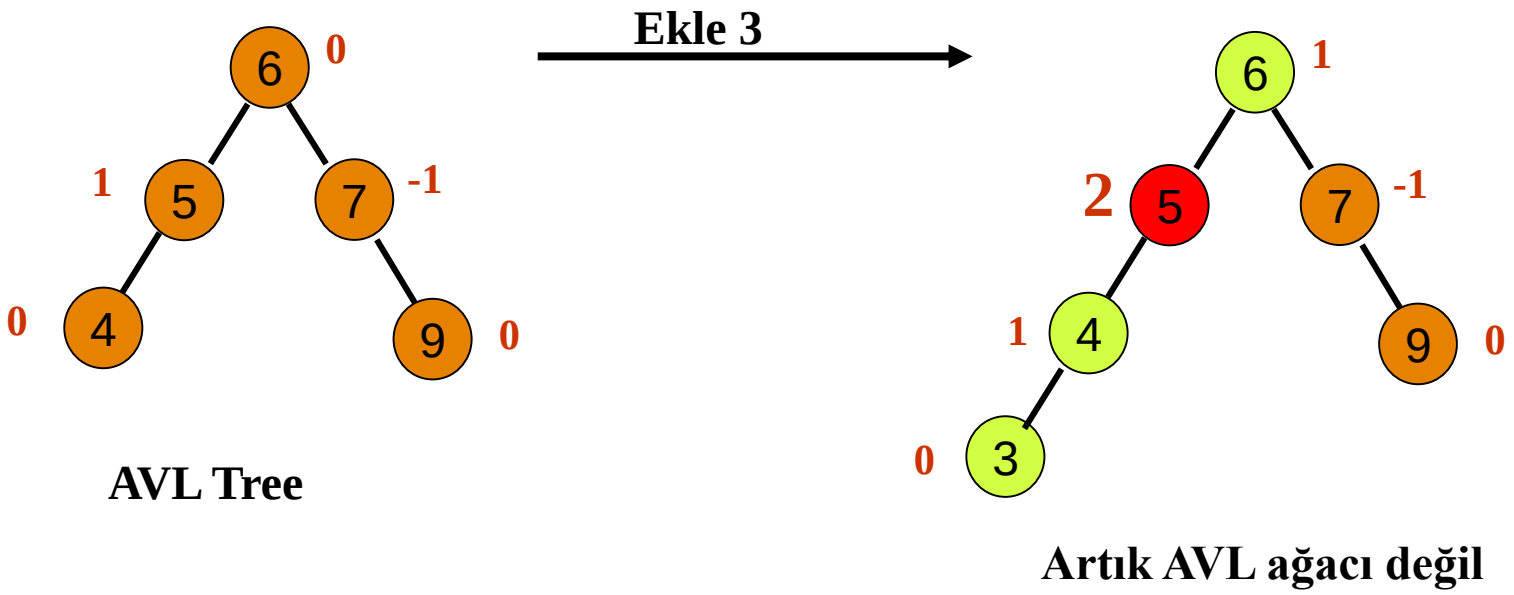
- N düğümlü bir AVL ağacının yüksekliği daima O(log n) dir.
- Peki nasıl?



Kırmızı numaralar balans fartörüdür.

AVL Ağaçları: Şanslı ve Şanssız

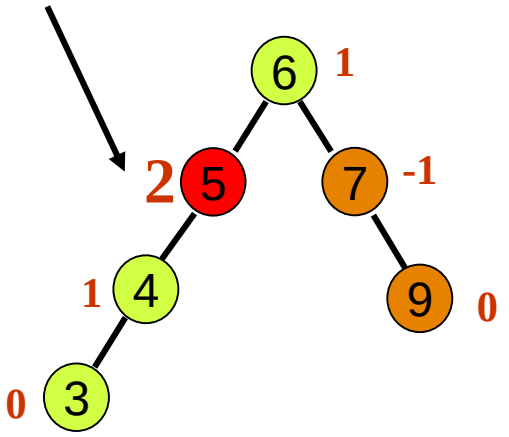
- Şanslı:
  - Arama süresi O(h) = O(log n)
- Şansız
  - Ekleme ve silme işlemleri ağacın dengesiz olmasına neden olabilir.



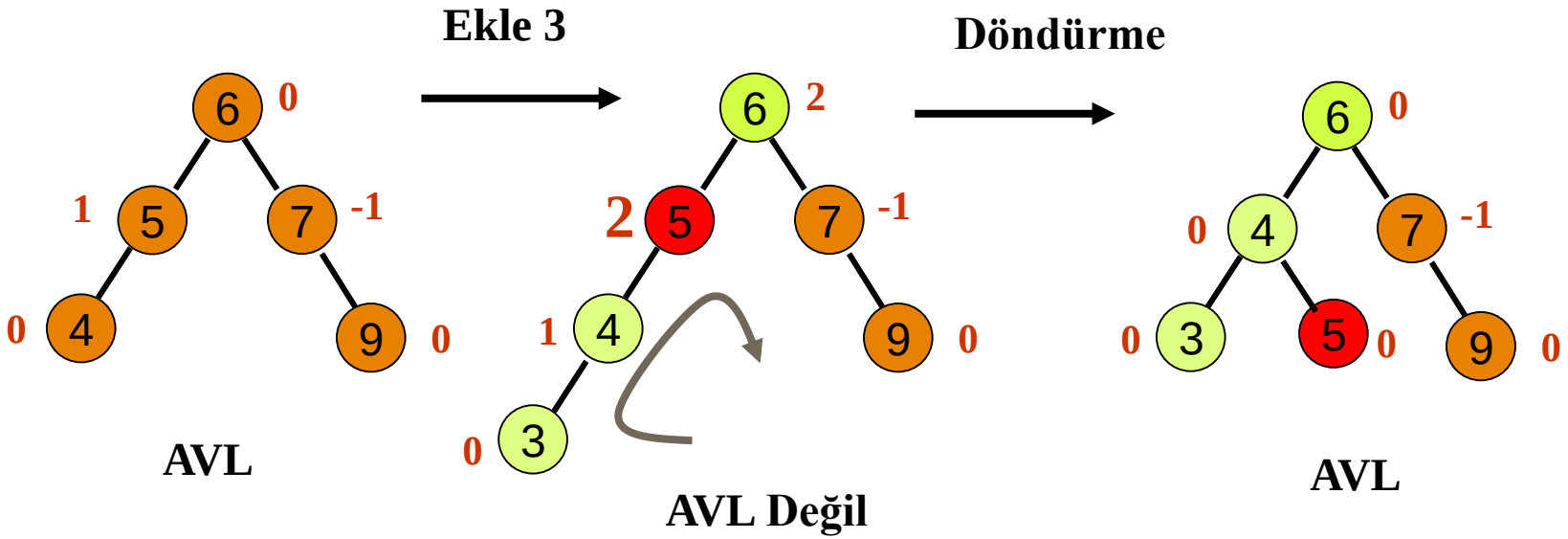
AVL Ağacında Dengenin Sağlanması

- Problem: Ekleme işlemi bazı durumlarda ekleme noktasına göre kök olan bölgelerde balans faktörün 2 veya -2 olmasına neden olabilir.

- Fikir: Yeni düğümü ekledikten sonra
  1. Balans faktörü düzelterek köke doğru çık.
  2. Eğer düğümün balans faktörü 2 veya -2 ise ağaç bu düğüm üzerinde döndürülerek düzeltilir.

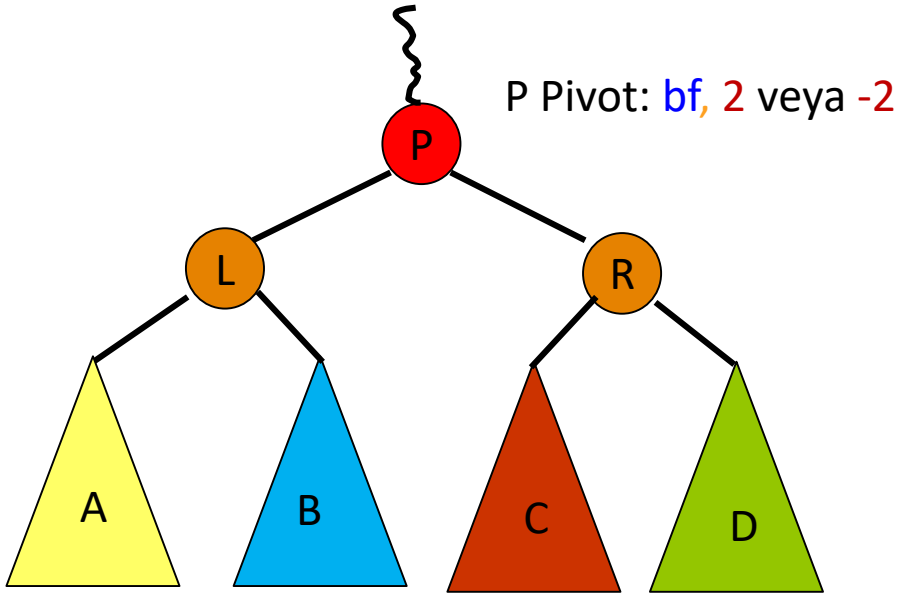


Denge Sağlama: Örnek



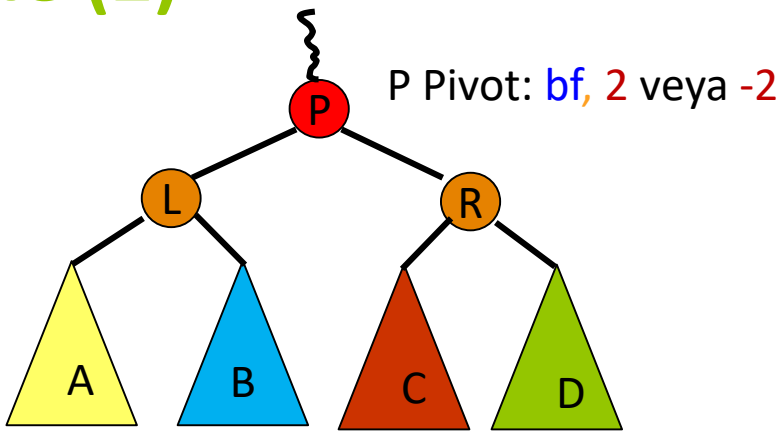
- Yeni düğümü ekledikten sonra
  1. Balans faktörü düzelterek köke doğru çık.
  2. Eğer düğümün balans faktörü 2 veya -2 ise ağaç bu düğüm üzerinde döndürülerek düzeltilir.

AVL Ağacı - Ekleme (1)



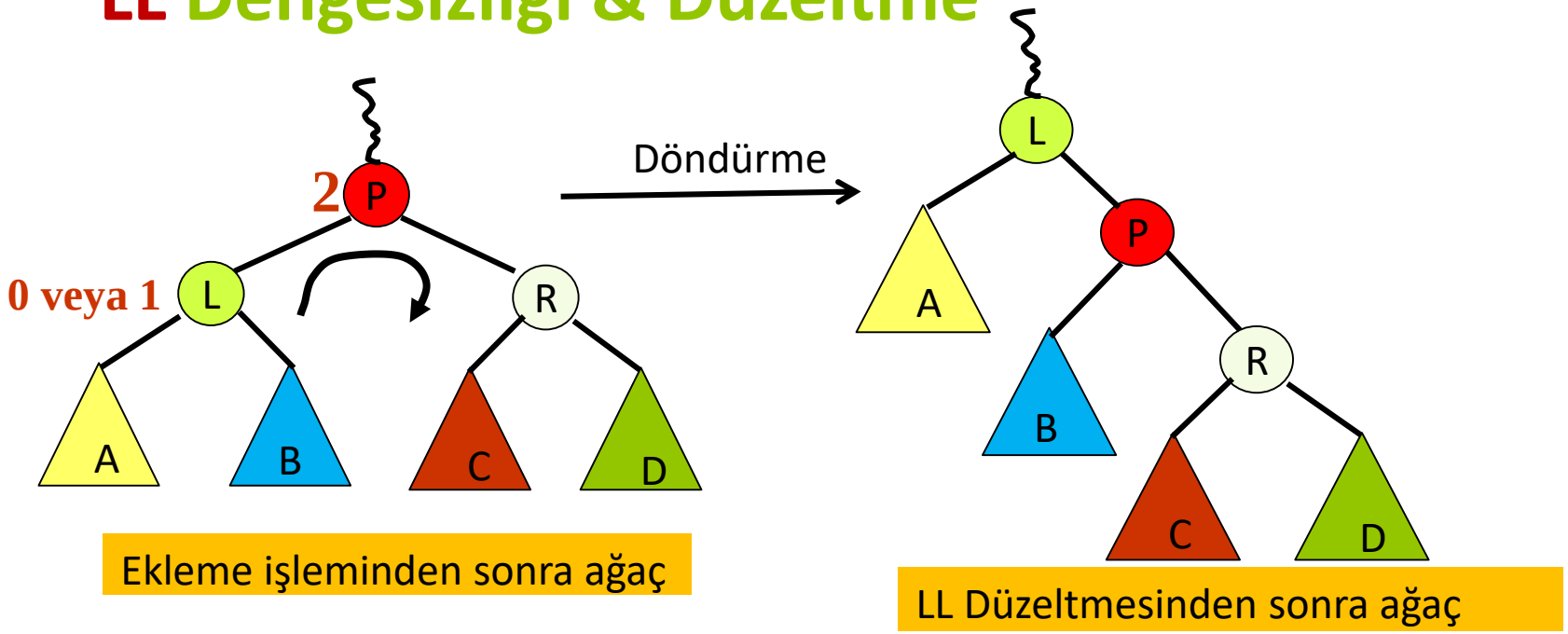
- P düğümünün dengeyi bozan düğüm olduğu düşünülürse.
  - P pivot düğüm şeklinde isimlendirilir.
  - Eklemeden sonra köke doğru çıkarken bf 'nin 2 veya -2 olduğu ilk düğümdür.

AVL Ağacı - Ekleme (2)



- 4 farklı durum vardır:
  - **Dış Durum** (tek döndürme gerektiren) :
    1. P'nin sol alt ağacının soluna eklendiğinde (LL Dengesizliği).
    2. P'nin sağ alt ağacının sağına eklendiğinde (RR Dengesizliği)
  - **İç Durum** (2 kez döndürme işlemi gerektiren) :
    3. P'nin sol alt ağacının sağına eklendiğinde (RL Dengesizliği)
    4. P'nin sağ alt ağacının soluna eklendiğinde(LR Dengesizliği)

LL Dengesizliği & Düzeltme

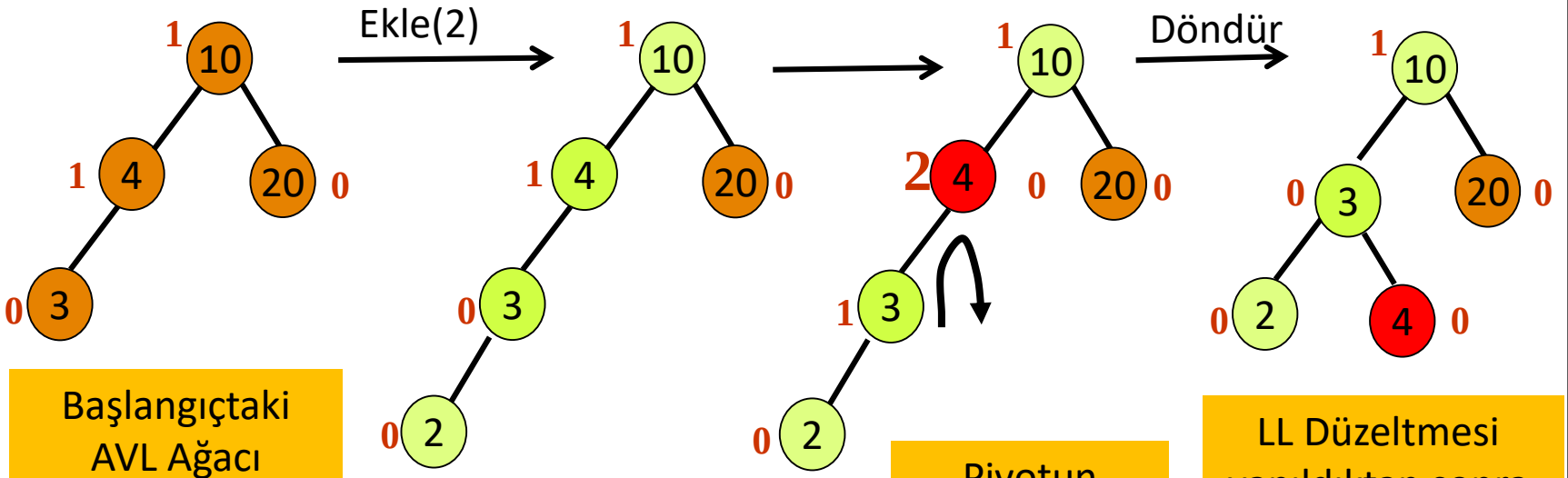


Ekleme işleminden sonra ağaç

LL Düzeltmesinden sonra ağaç

- LL Dengesizliği: P'nin sol alt ağacının soluna eklendiğimizde (A alt ağacınına)
  - P 'nin bf si 2
  - L 'nin bf si 0 veya 1
- Düzeltme: P etrafında sağa doğru tek dönderme.

LL Dengesizliği Düzeltme Örneği (1)



Başlangıçtaki AVL Ağacı

2 eklendikten sonra ağacın durumu

Pivotun belirlenmesi

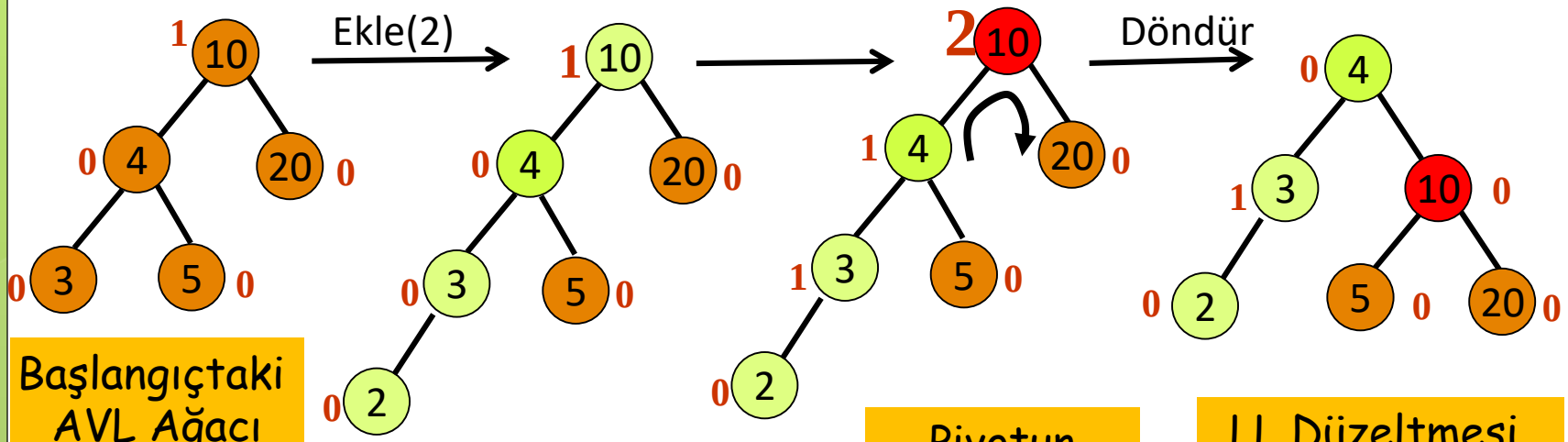
LL Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

- LL Dengesizliği:
  - **P(4)**'ün **bf** si 2
  - **L(3)**'ün **bf** si 0 veya 1

LL Dengesizliği Düzeltme Örneği (2)



Başlangıçtaki AVL Ağacı

2 eklendikten sonra ağacın durumu

Pivotun belirlenmesi

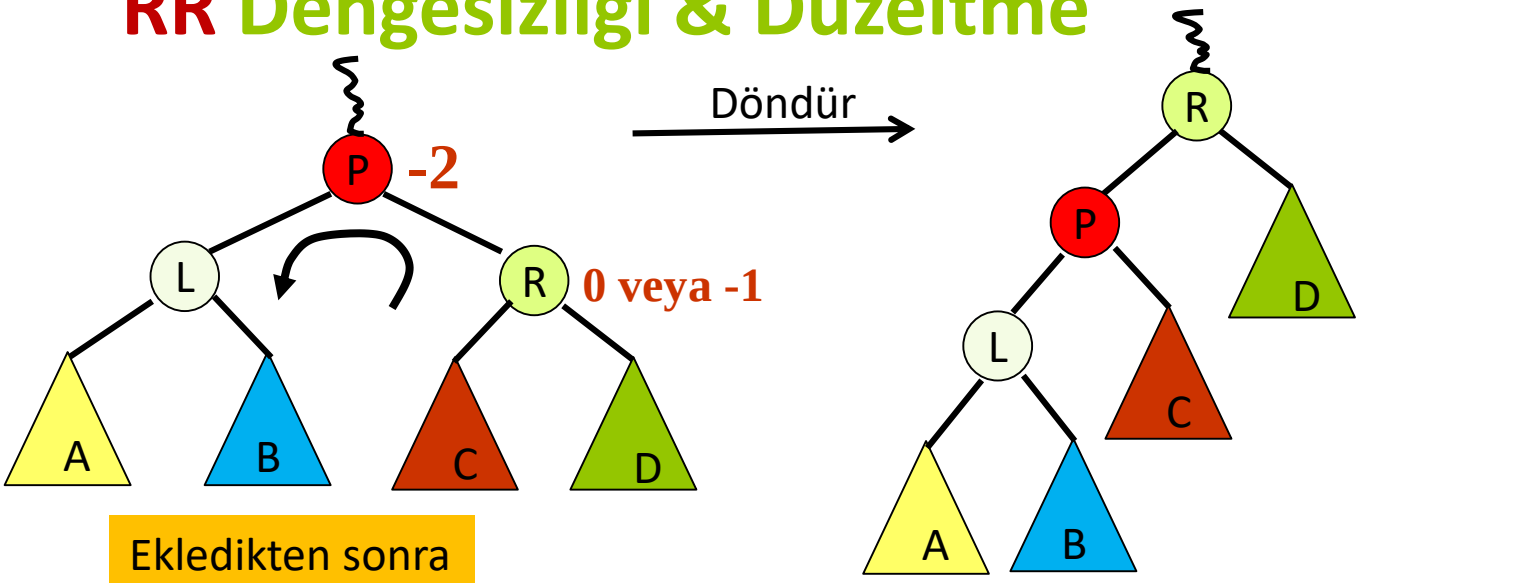
LL Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

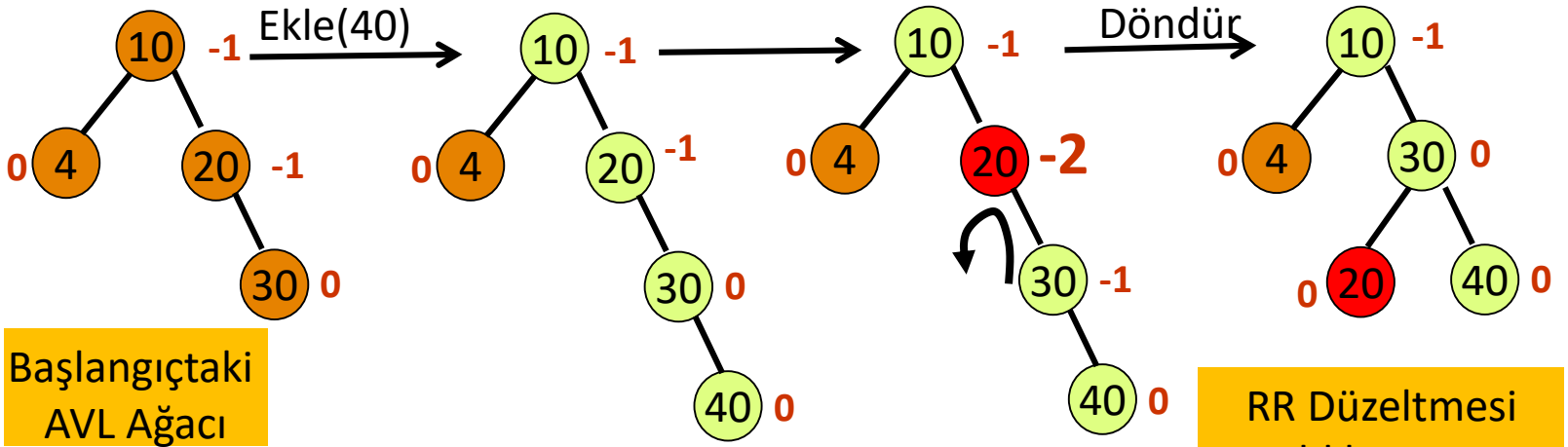
- LL Dengesizliği:
  - P(4)'ün bfsi 2
  - L(3)'ün bfsi 0 veya 1

RR Dengesizliği & Düzeltme



- RR Dengesizliği: P'nin sağ alt ağacının sağına eklendiğinde (D alt ağacına eklendiğinde)
  - P → bf = -2
  - R → bf = 0 veya -1
- Düzeltme: P etrafında sola doğru tek dönderme

RR Dengesizliği Düzeltme Örneği (1)



Başlangıçtaki AVL Ağacı

40 eklendikten sonra ağacın durumu

Pivotun belirlenmesi

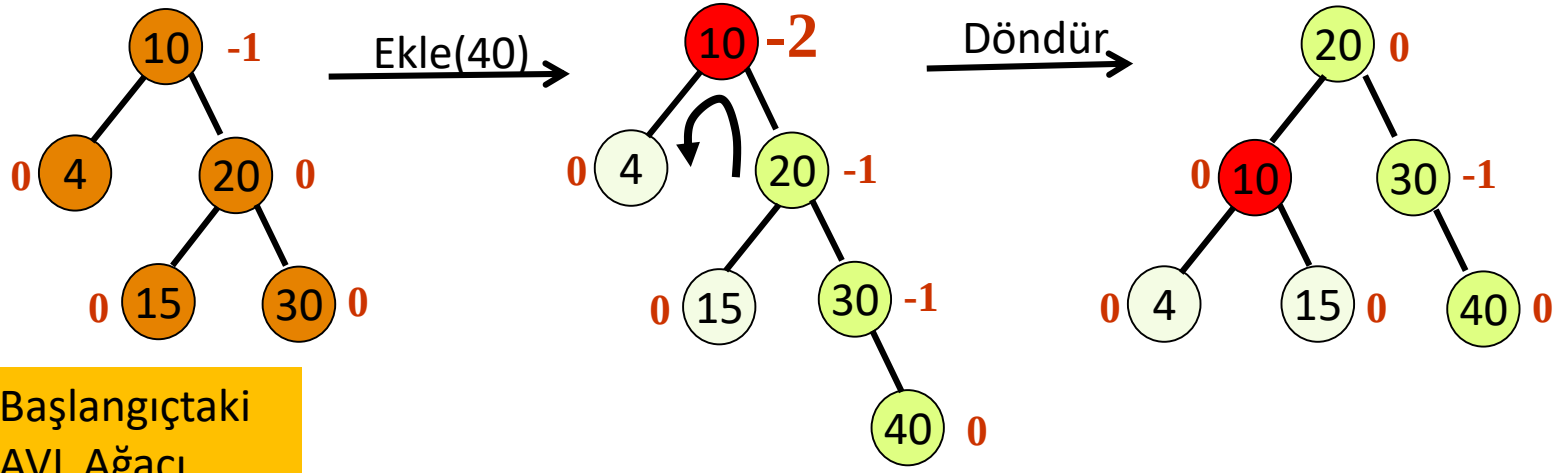
RR Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

- RR Dengesizliği:
  - P(20) → bf = -2
  - R(30) → bf = 0 veya -1

RR Dengesizliği Düzeltme Örneği (2)



Başlangıçtaki AVL Ağacı

40 eklendikten sonra ağacın durumu

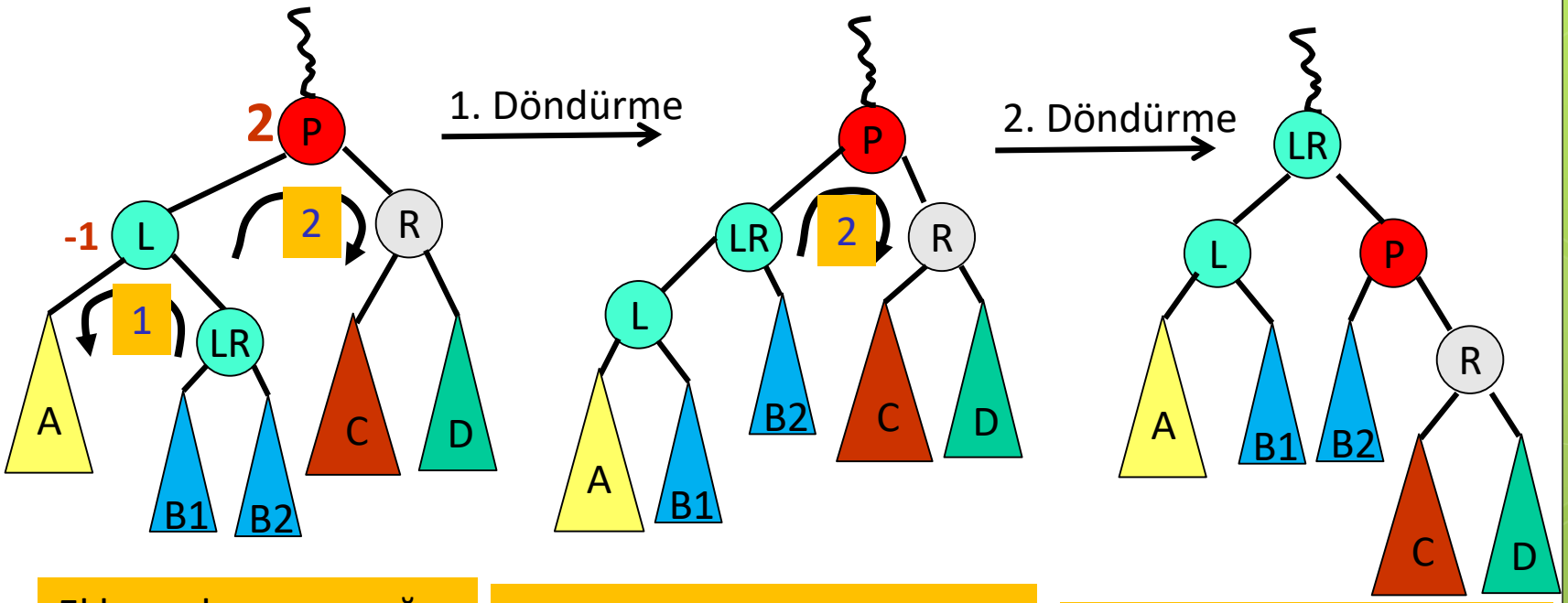
RR Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle ve 10'u pivot olarak belirle

Dengesizliğin türünün belirlenmesi

- RR Dengesizliği:
  - P(10) → bf = -2
  - R(20) → bf = 0 veya -1

LR Dengesizliği & Düzeltme



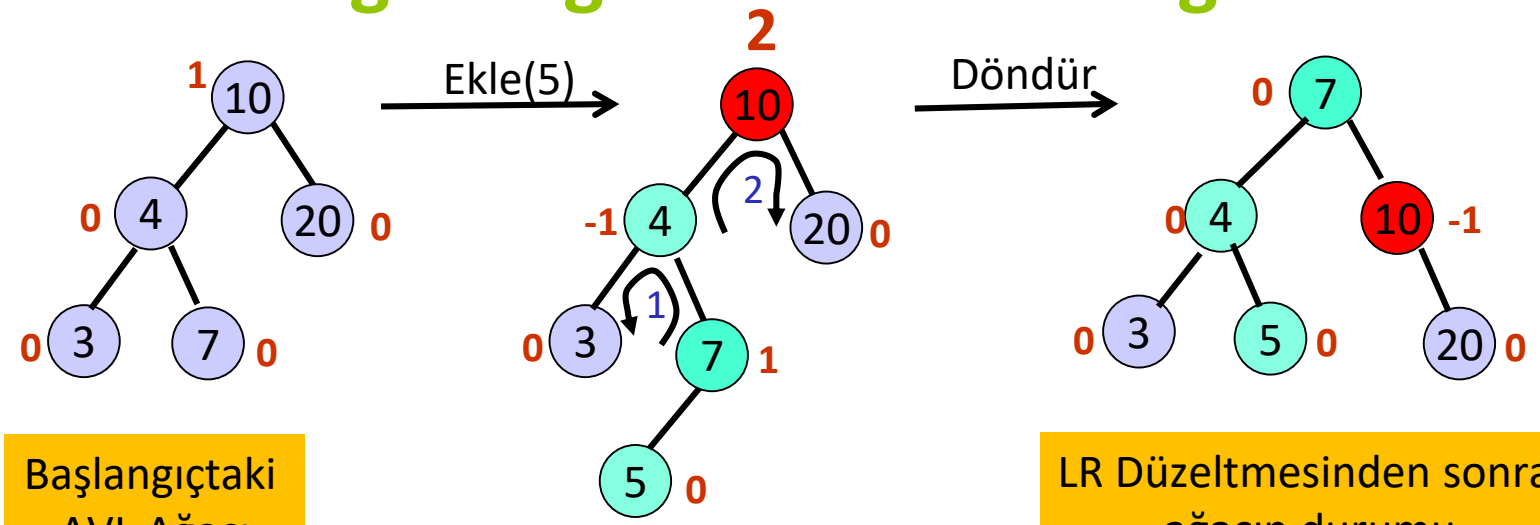
Eklemeden sonra ağaç

1. Döndürmeden sonra ağaç

LR Düzeltmesinden sonra

- LR Dengesizliği: P'nin sol alt ağacının sağına eklendiğinde (LR ağacına)
  - P → bf = 2
  - L → bf = -1
- Düzeltme: L & P etrafında 2 kez döndürme

LR Dengesizliği Düzeltme Örneği



Başlangıçtaki AVL Ağacı

5 eklendikten sonra ağacın durumu

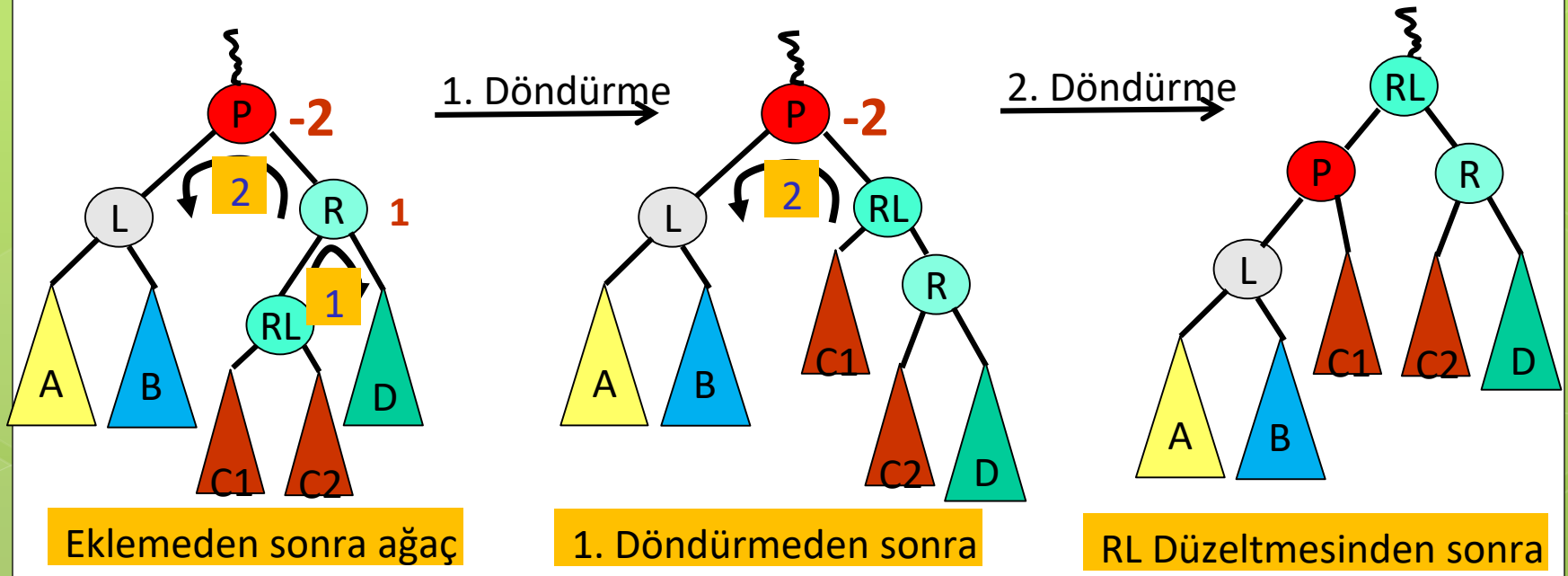
LR Düzeltmesinden sonra ağacın durumu

Balans faktörü düzelterek köke doğru ilerle ve 10'u pivot olarak belirle

Dengesizliğin türünün belirlenmesi

- LR Dengesizliği:
  - P(10) → bf = 2
  - L(4) → bf = -1

RL Dengesizliği & Düzeltme



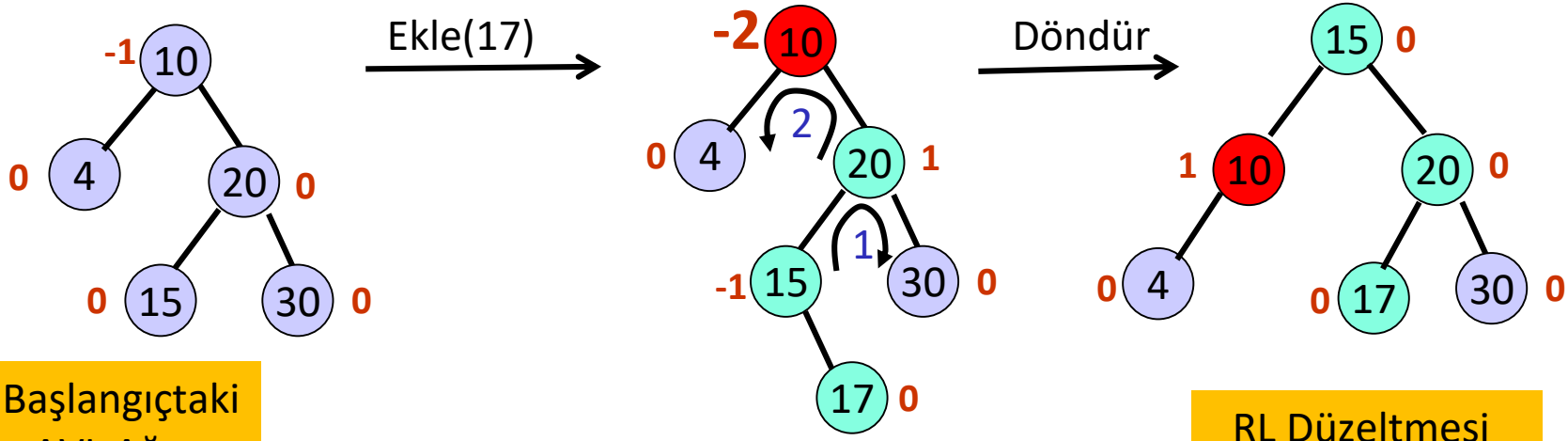
Eklemeden sonra ağaç

1. Döndürmeden sonra

RL Düzeltmesinden sonra

- RL Dengesizliği: P'nin sağ alt ağacının soluna eklendiğinde (RL alt ağacına)
  - P → bf = -2
  - R → bf = 1
- Düzeltme: R & P etrafında 2 kez döndürme.

RL Dengesizliği Düzeltme Örneği



Başlangıçtaki AVL Ağacı

17 eklendikten sonra ağacın durumu

RL Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle ve 10'u pivot olarak belirle

Dengesizliğin türünün belirlenmesi

- RL Dengesizliği:
  - P(10) → bf = -2
  - R(20) → bf = 1

AVL AĞAÇLARI

- **AVL Ağaçlarının C Kodları İle Yazılmış Bazı Fonksiyonları**

- **Fonksiyon 1.** AVL Ağacı İçim Düğüm Tanımı

```
public class AvlAgac {
public long data;
public AvlAgac left;
public AvlAgac right;
public int height=0;
}
```

AVL AĞAÇLARI

- **Fonksiyon 2.** Bir AVL Düğümünün Yüksekliğini Döndüren Fonksiyon

```
int Height (AvlAgac p)
{ if (p == null)
      return -1;
  else
  {
    return p.height;
  }
}
```

AVL AĞAÇLARI

- **Fonksiyon 3.** AVL Ağacı İçin Ekleme Fonksiyonu

```
public AvlAgac insert(AvlAgac p, int x) {
if (p == null)
{ p = new AvlAgac(); p.data = x; }
else
{ if (x <p.data)
  {
    p.left = insert(p.left, x);
    if (Height(p.left)-Height(p.right) == 2)
    if (x < p.left.data)
       p = singlerotateright(p);
    else  p = doublerotateright(p);
  }
```

AVL AĞAÇLARI

```
 else if (x > p.data)
{ p.right = insert(p.right, x);
 if (Height(p.left) - Height(p.right) == -2)
 if (x > p.right.data)
   p = singlerotateleft(p);
 else p = doublerotateleft(p);
 }
}
 p.height = max(Height(p.left), Height(p.right)) + 1;
 return p;
}
int max(int a, int b)
{ if (a > b) return a;
  else return b;
}
```

AVL AĞAÇLARI

- **Fonksiyon 4**. AVL Ağacı için sağa doğru tek döndürme (LL) Fonksiyonu

```
AvlAgac singlerotateright(AvlAgac p)
{
 AvlAgac lc = p.left;
 p.left = lc.right;
 lc.right = p;
 p.height = max(Height(p.left), Height(p.right)) + 1;
 lc.height = max(Height(lc.left), lc.height) + 1;
 return lc;
}
```

AVL AĞAÇLARI

- **Fonksiyon 5.** AVL Ağacı için LR çift döndürme

```
AvlAgac doublerotateright(AvlAgac p)
 {
 p.left = singlerotateleft(p.left);
 return singlerotateright(p);
 }
```

AVL AĞAÇLARI

- **Fonksiyon 6.** AVL Ağacı için sağa doğru tek döndürme (RR) Fonksiyonu

```
AvlAgac singlerotateleft(AvlAgac p)
{
 AvlAgac rc = p.right;
 p.right = rc.left;
 rc.left = p;
 p.height = max(Height(p.left), Height(p.right))+1;
 rc.height = max(Height(rc.right), rc.height)+ 1;
 return rc;
}
```

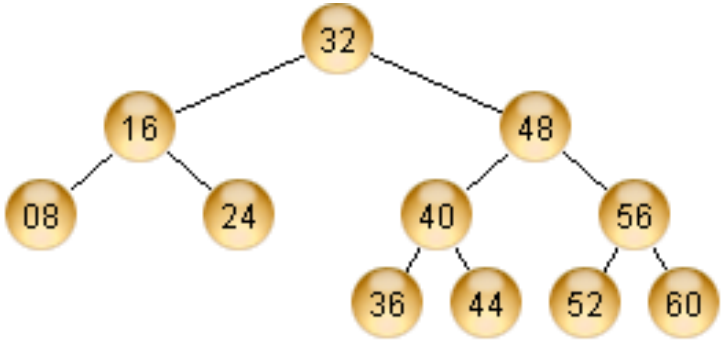
AVL AĞAÇLARI

- **Fonksiyon 7.** AVL Ağacı için RL çift döndürme

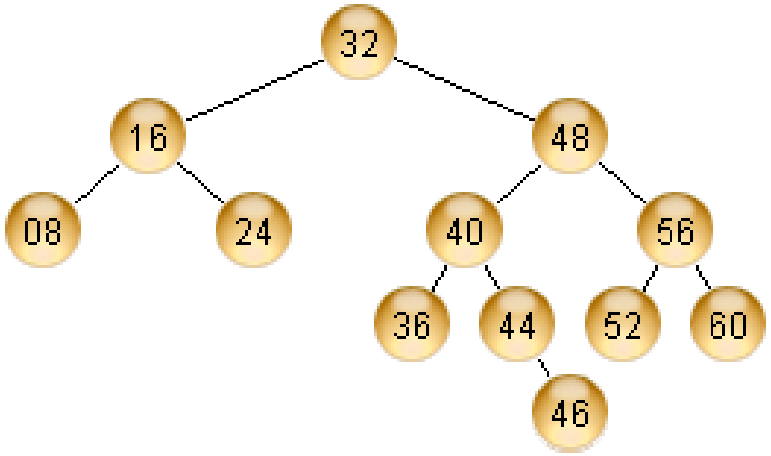
```
AvlAgacdoublerotateleft(AvlAgac p)
 {
 p.right = singlerotateright(p.right);
 return singlerotateleft(p);
 }
```

AVL AĞAÇLARI

- **Örnek:**

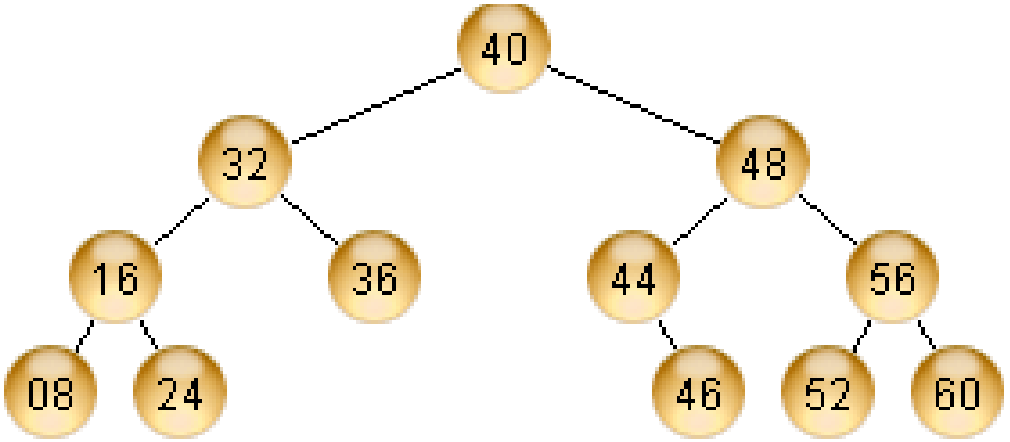


- **46, eklendi**



AVL AĞAÇLARI

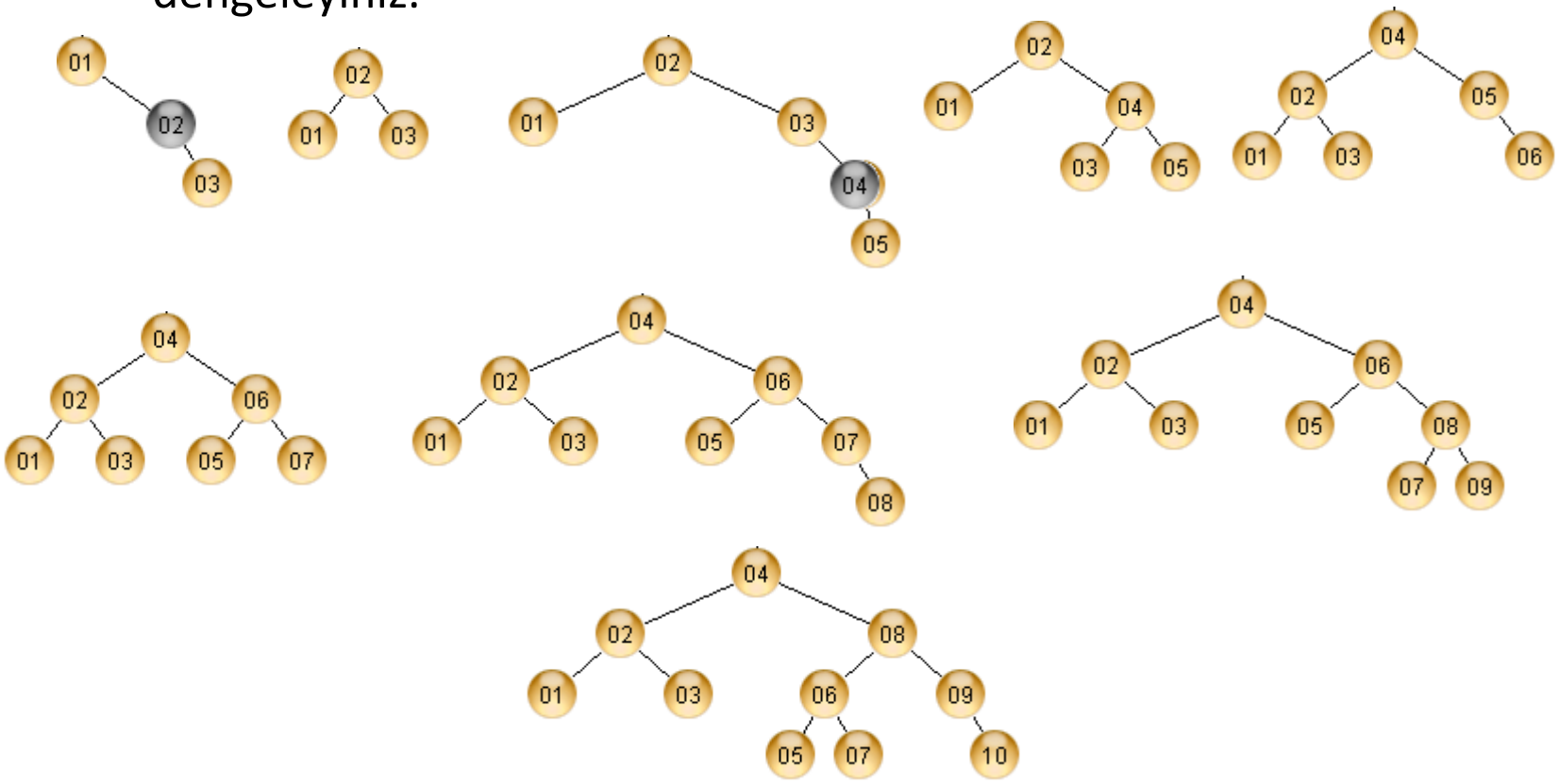
- **Örnek:**
-
-



- Dengeli AVL ağacı

AVL AĞAÇLARI

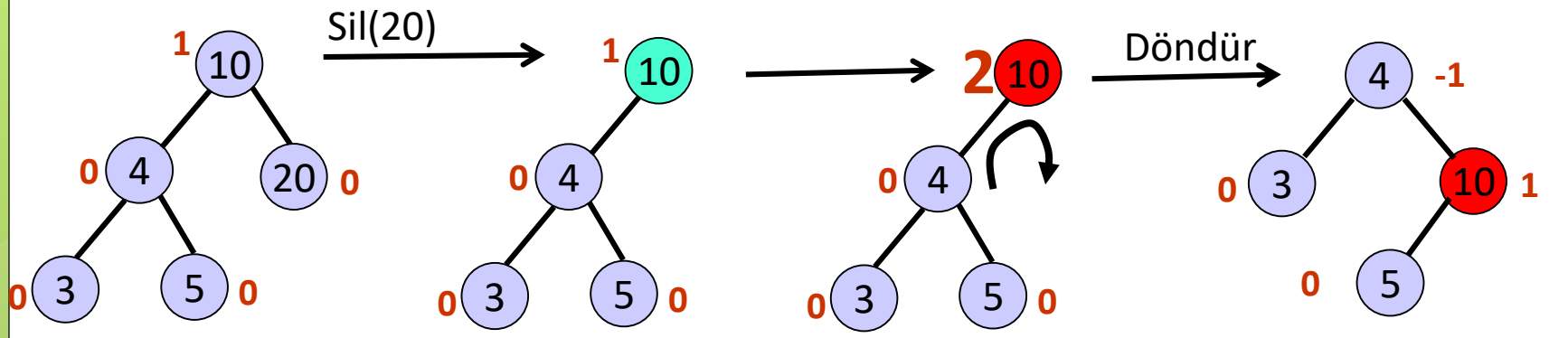
- Örnek: 1,2,3,4,5,6,7,8,9,10 ağacı oluşturup adım adım dengeleyiniz.



AVL AĞAÇLARI- Silme

- Silme işlemi ekleme işlemi ile benzerlik gösterir.
- AVL ağaçlarında silme işlemini yaparken özellikle çocukları olan düğümlerin silme işlemi farklı yapılarda gözükür. Amaç dengeyi bozmayacak düğümün öncelikle sağlanması.
- Soldaki en büyük çocuk veya Sağdaki en küçük çocuk yeni düğüm olur. Önemli olan silme olayından sonra dengeleme işleminin yeniden sağlanmasıdır. Bu işlem sırasında birden fazla döndürme işlemi yapılabilir.
- Kural 1: Silinen düğüm yaprak ise, problem yok dengeleme işlemi gerekiyorsa yap.
- Kural 2: Silinen düğüm çocukları olan bir düğüm ise ya soldaki en büyük çocuğu veya sağdaki en küçük çocuğu al. Bu işlem sırasında dengeleme için birden fazla döndürme işlemi yapılabilir.

Silme Örneği (1)



Başlangıçtaki AVL Ağacı

20 silindikten sonra Ağacın durumu

10'un pivot olarak belirlenmesi

LL Düzeltmesinden Sonra AVL Ağacı

Balans faktörü düzelterek köke doğru ilerle

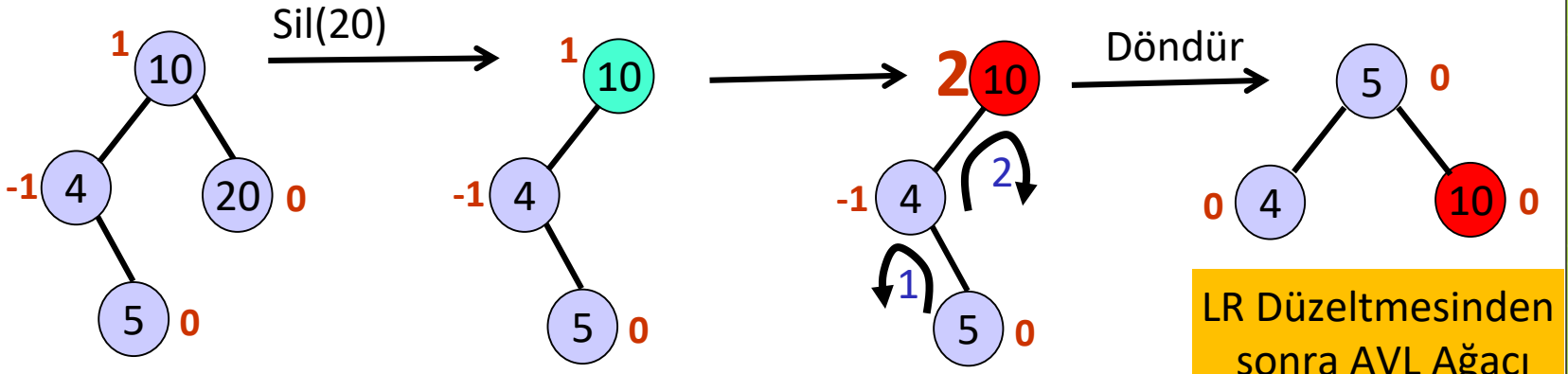
Balans faktörü=0 ise en az döndürme olacak şekilde seç.

Dengesizliğin türünün belirlenmesi

LL Dengesizliği:

- **P(10)**'ün **bf**si 2
- **L(4)**'ün **bf**si 0 veya 1

Silme Örneği (2)



Başlangıçtaki AVL Ağacı

20 silindikten sonra Ağacın durumu

10'un pivot olarak belirlenmesi

LR Düzeltmesinden sonra AVL Ağacı

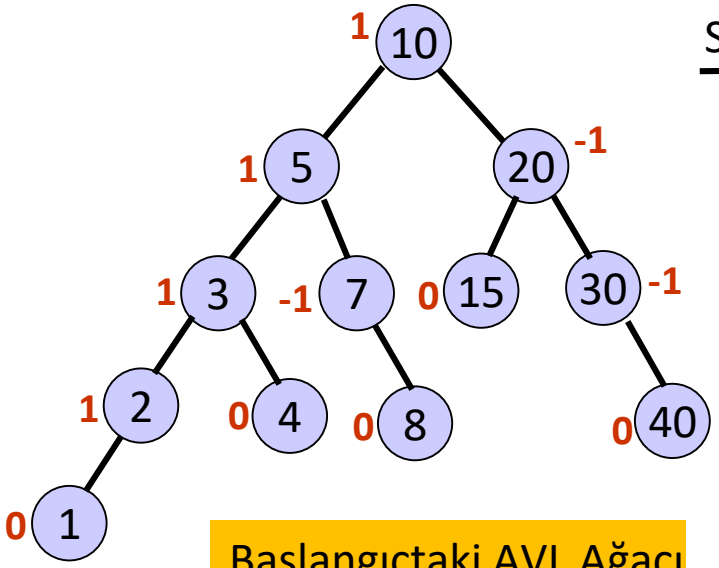
Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

LR Dengesizliği:

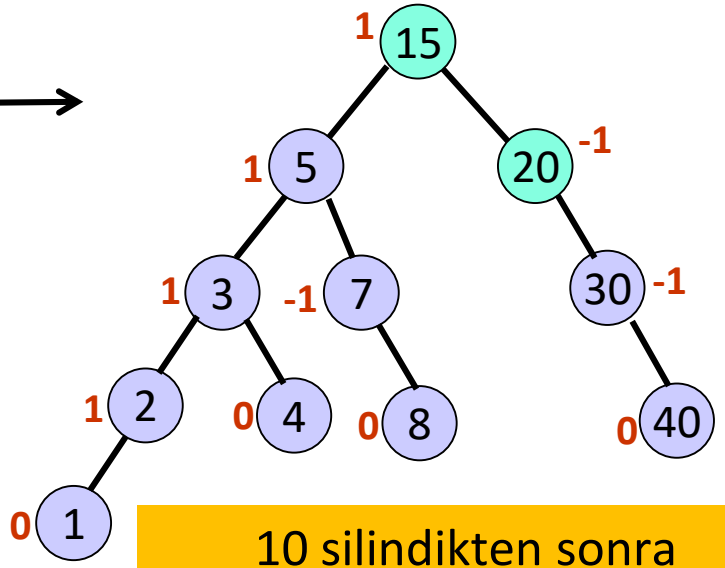
- P(10) → bf = 2
- L(4) → bf = -1

Silme Örneği (3)



Başlangıçtaki AVL Ağacı

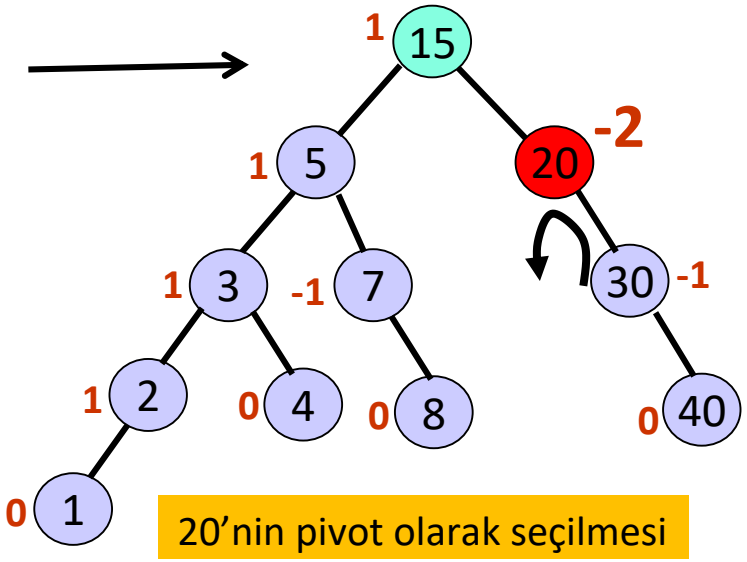
Sil(10)



10 silindikten sonra
10 ve 15(sağ alt ağaçtaki en küçük eleman) yer değiştirdi ve 15 silindi.

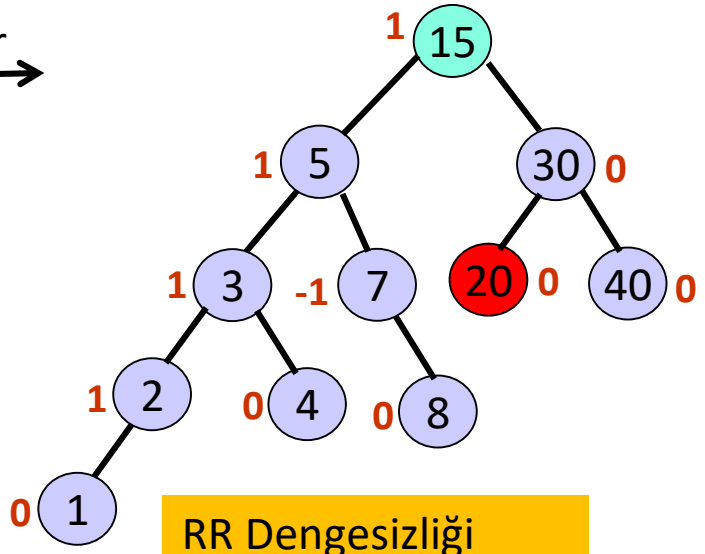
Balans faktörü düzelterek köke doğru ilerle

Silme Örneği (3)



20'nin pivot olarak seçilmesi

Döndür



RR Dengesizliği düzeltildikten sonra

Dengesizliğin türünü belirle

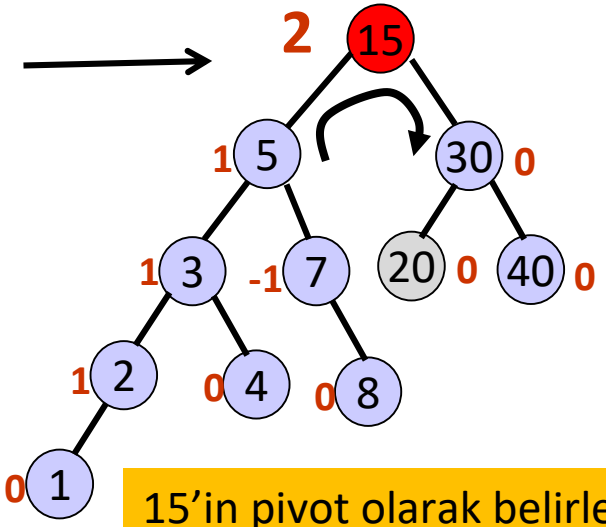
RR Dengesizliği:

- P(20) → bf = -2
- R(30) → bf = 0 veya -1

Yukardaki AVL ağacı mıdır?

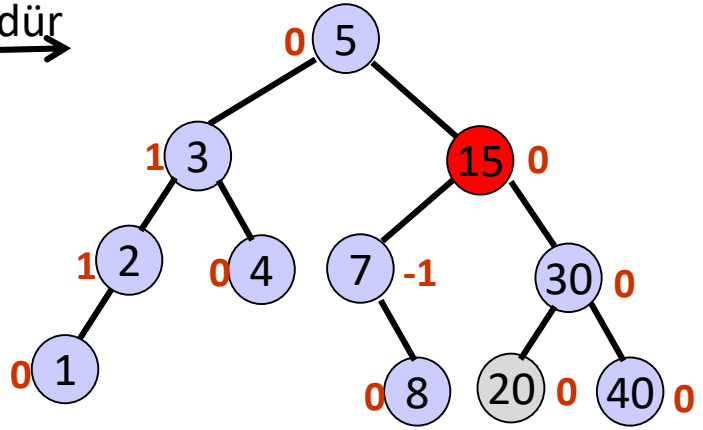
Balans faktörü düzelterek köke doğru ilerle.

Silme Örneği (3)



15'in pivot olarak belirlenmesi

Döndür



Düzeltilmiş AVL ağacı

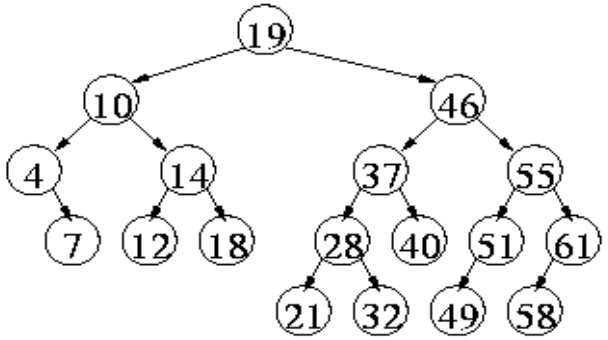
Dengesizliğin türünü belirle

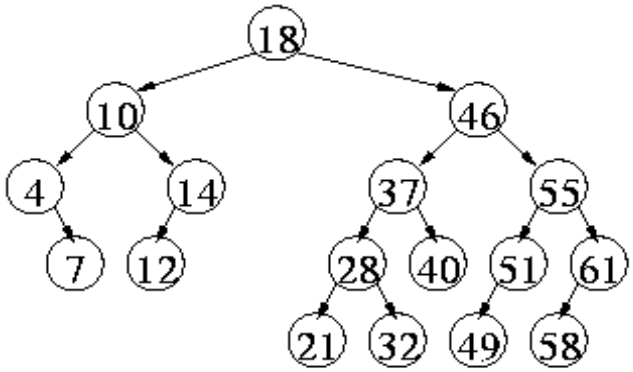
LL Dengesizliği:

- P(15) → bf = 2
- L(5) → bf = 0 veya 1

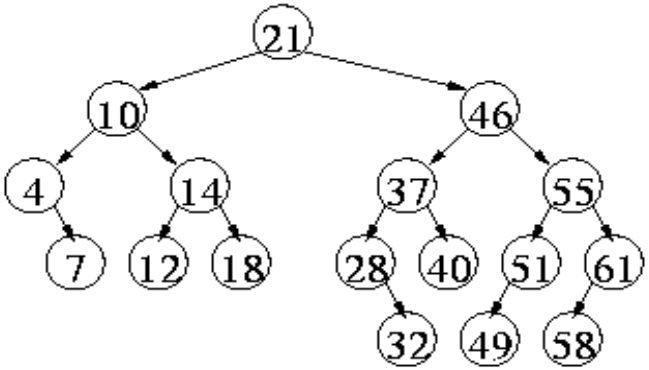
Silme Örneği– Kural 2

- **19 silindi (Soldaki en büyük düğüm veya sağdaki en küçük düğüm kök yapılır.**



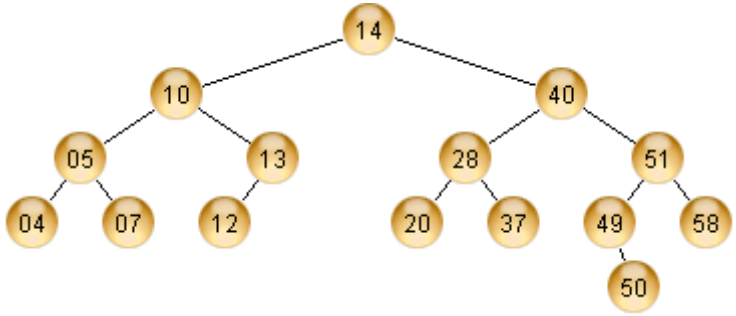


veya

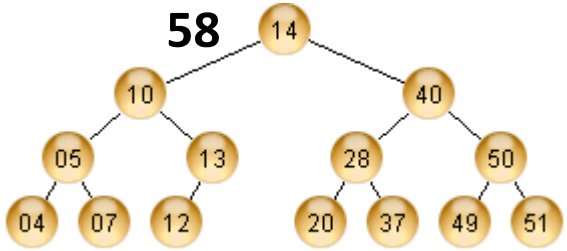


Silme Örneği– Kural 1-2

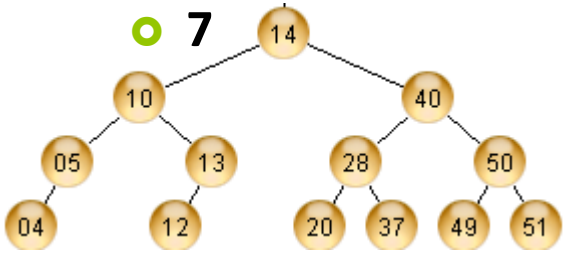
- **Örnek:** Sırasıyla siliniz; 58 ,7, 40,10,14 (Silme işleminde Soldaki en büyük düğüm alınacak)



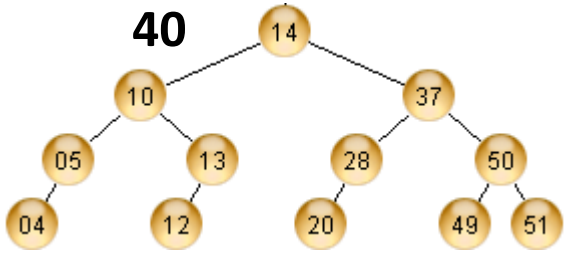
58



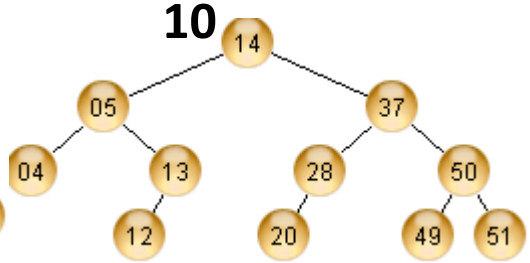
- **7**



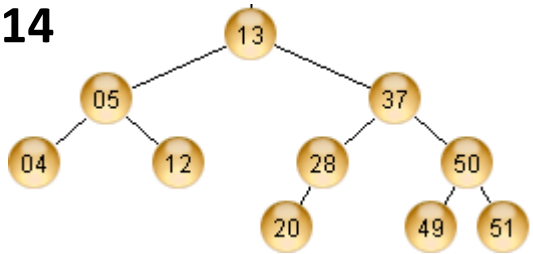
40



10

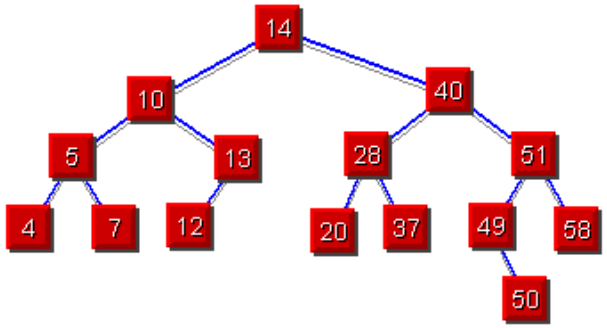


- **14**

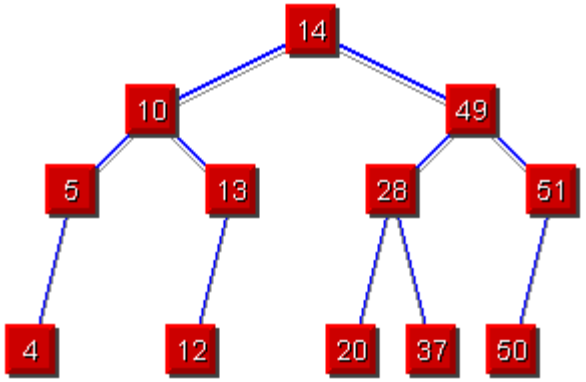


Silme Örneği– Kural 1-2

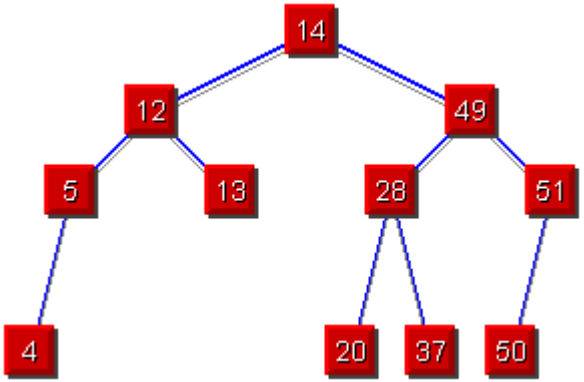
- **Örnek:** Sırasıyla siliniz; 40,58 , 7, 10,14 (Silme işleminde Sağdaki en küçük düğüm alınacak)



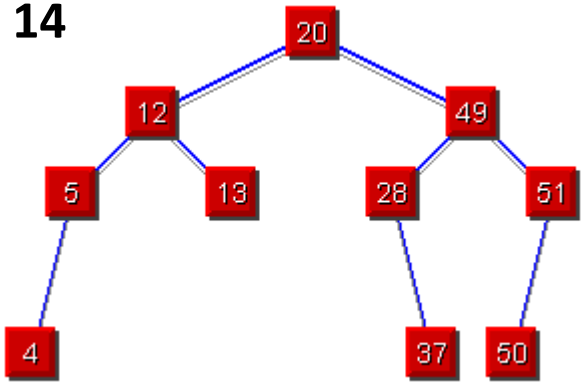
40,58,7



- **10**

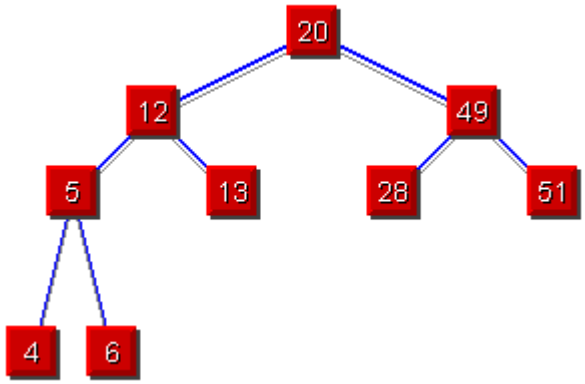


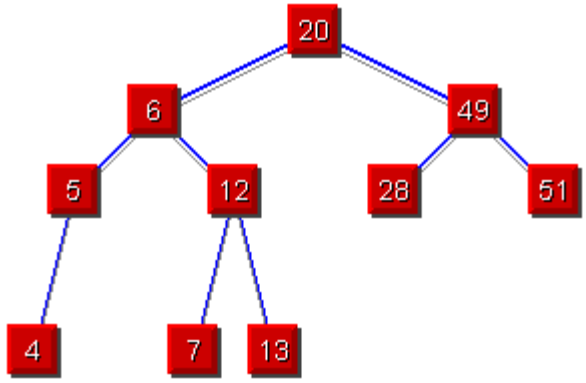
14



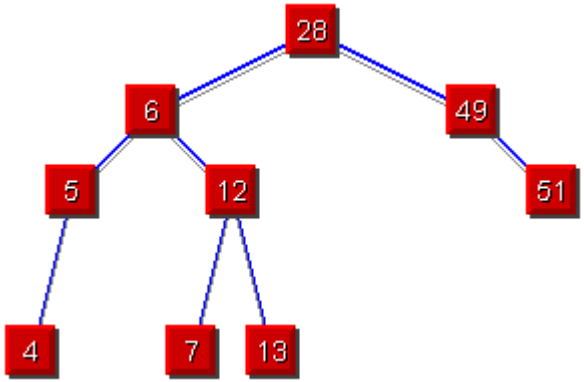
Silme Örneği

- **Örnek:** Ağaca 7 ekleyip dengeleyiniz.





- Ağaçtan 20 değerini siliniz (Sağdaki en küçük düğüm alınacak)



AVL -Arama (Bul)

- AVL ağacı bir tür İkili Arama Ağacı (BST) olduğundan arama algoritması BST ile aynıdır.
- O(logN) de çalışması garantidir.

AVL Ağaçları – Özet

- AVL Ağaçlarının Avantajları
  1. AVL ağacı devamlı dengeli olduğu için Arama/Ekleme/Silme işlemi O(logN) yapılır.
  2. Düzeltme işlemi algoritmaların karmaşıklığını etkilemez.

- AVL Ağaçlarının Dezavantajları:
  1. Balans faktörü için ekstra yer gereklidir.
  2. Yarı dengeli ağaçlarda AVL ağaçları kadar iyi performans verir ayrıca bu tür algoritmalarda balans faktör kullanılmaz
     - Splay ağaçları

Ödev

- Uygulama : S,E,L,İ,M, K,A,Ç,T,I AVL ağacını yapınız.
- Cevap: Kök düğümünün değeri L
- Yaprak düğümün değerleri: A,I,K,M,T
- **DSW metod (Recreate)**
- - Yükseklikleri ağacı yeniden oluşturarak ayarlar
- -Ağaçtan backbone elde eder ve ağacı yeniden oluşturur
- - Colin Day, Quentin F. Stout ve Bette L.Warren tarafından geliştirilmiştir
- **Self-Adjusting Trees (Priority)**
- - Kullanılma sıklığına göre ağacı sürekli düzenler
- - Her elemana erişimde root'a doğru yaklaştırır.
- Kırmızı olarak belirtilen kısımları en kısa zamanda programı ile birlikte word belgesi olarak hazırlanıp teslim edilecektir.

SPLAY TREE

- AVL tree
- 2-3 ve 2-3-4 tree
- Splay tree
- Red-Black Tree
- B- tree

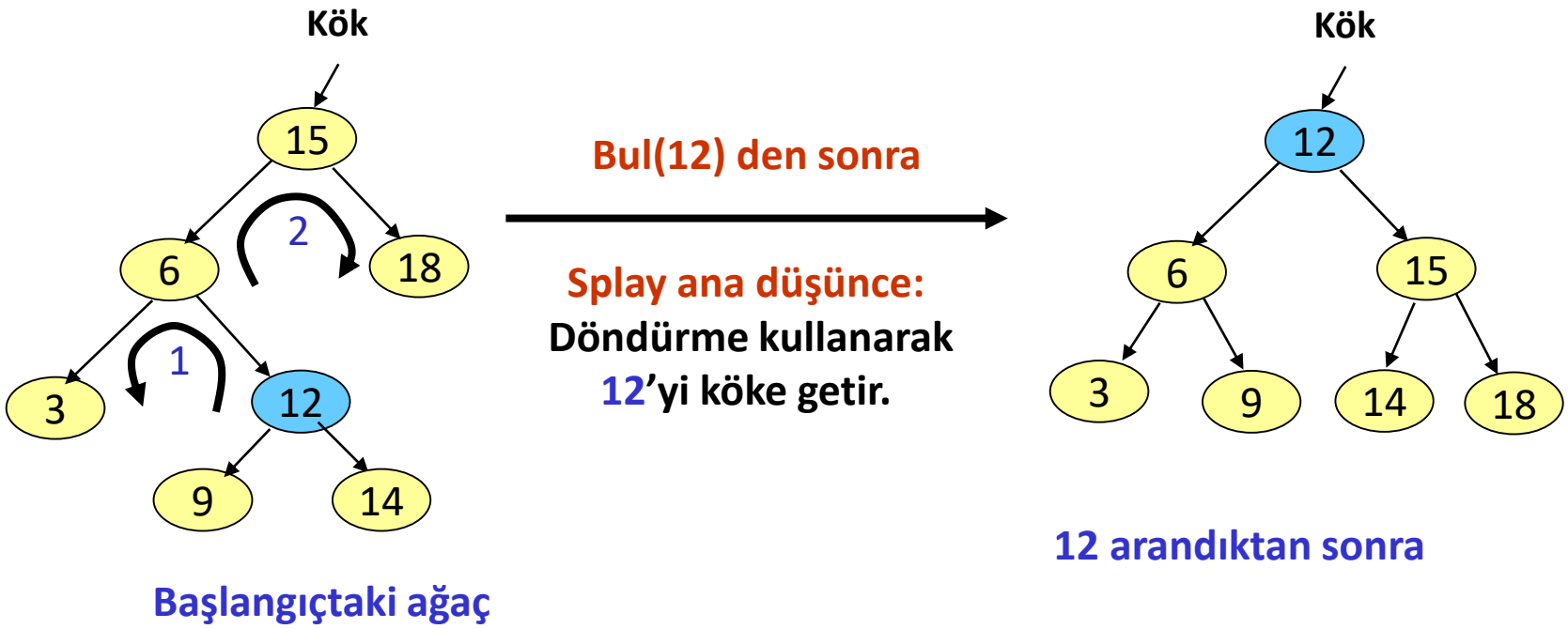
Splay Ağaçları

- Splay ağaçları ikili arama ağaç yapılarından birisidir.
  - Her zaman dengeli değildir.
  - Arama ve ekleme işlemlerinde dengelemeye çalışılır böylece gelecek operasyonlar daha hızlı çalışır.

- Sezgiseldir.
  - Eğer X bir kez erişildi ise, bir kez daha erişilecektir.
  - X erişildikten sonra "splaying" operasyonu yapılır.
    - X köke taşınır.

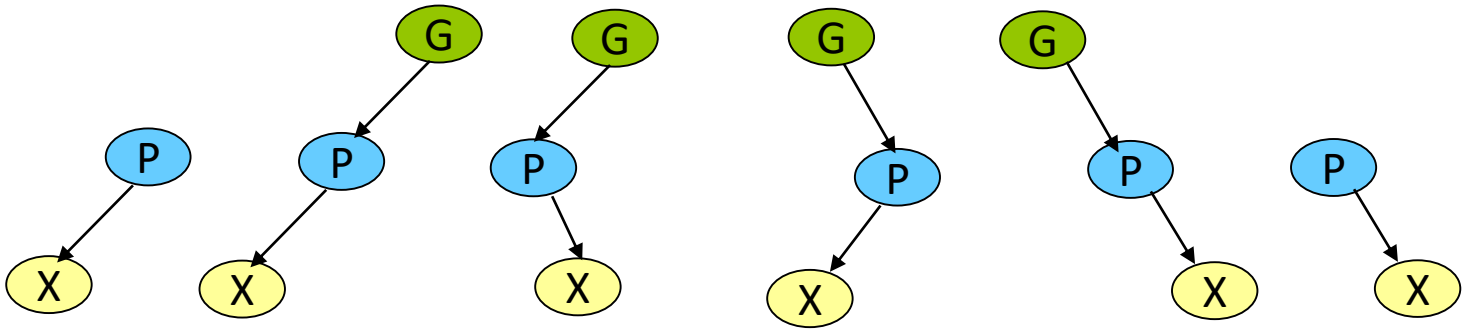
Örnek

- Burada ağacın dengeli olmasının yanı sıra 12 ye O(1) zamanında erişilebiliyor.
- Aktif (son erişilenler) düğümler köke doğru taşınırken aktif olmayan düğümler kökten uzaklaştırılır.



Splay Ağacı Terminolojisi

- X kök olmayan bir düğüm olsun. (ailesi olan)
- P düğümü X'in aile düğümü olsun.
- G düğümü X'in ata düğümü olsun.
- G ve X arasındaki yol düşünüldüğünde:
  - Her sağa gitme işlemine "zig" işlemi denilmektedir.
  - Her sola gitme işlemine "zag" işlemi denilmektedir.
- Toplam 6 farklı durum oluşabilir:

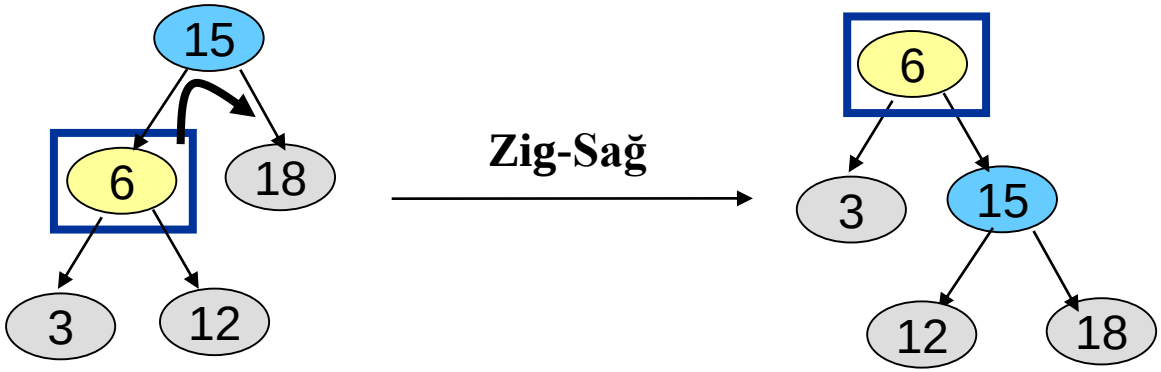


Splay Ağaç İşlemleri

- X'e erişildiğinde 6 tane rotasyon işleminden birisi uygulanır:
  - Tek Dönme (X, P'ye sahip ama G'ye sahip değil)
    - zig, zag
  - Çift Dönme (X hem P'ye hem de G'ye sahip)
    - zig-zig, zig-zag
    - zag-zig, zag-zag

Splay Ağaçları: Zig İşlemi

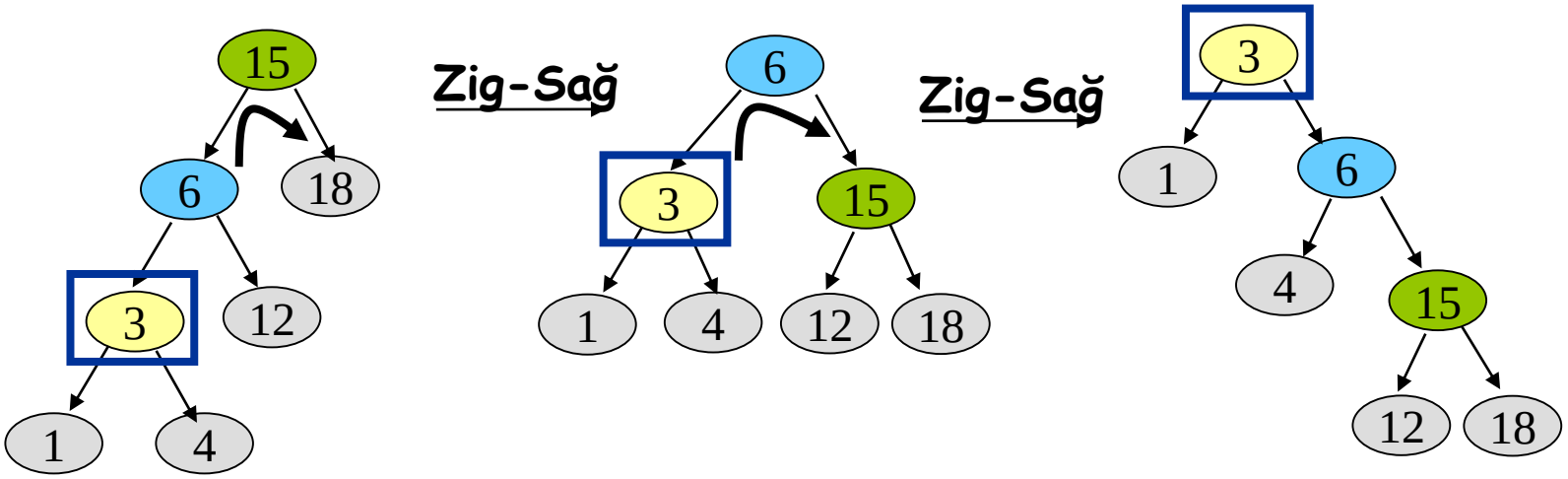
- "Zig" işlemi AVL ağacındaki gibi tek döndürme işlemidir.
- Örneğin erişilen elemanın 6 olduğu düşünülürse.



- "Zig-Sağ" işlemi 6'yı köke taşır.
- Bir sonraki işlemde 6'ya O(1) de erişilebilir.
- AVL ağacındaki sağ dönme işlemi ile benzerdir.

Splay Ağaçları: Zig-Zig İşlemi

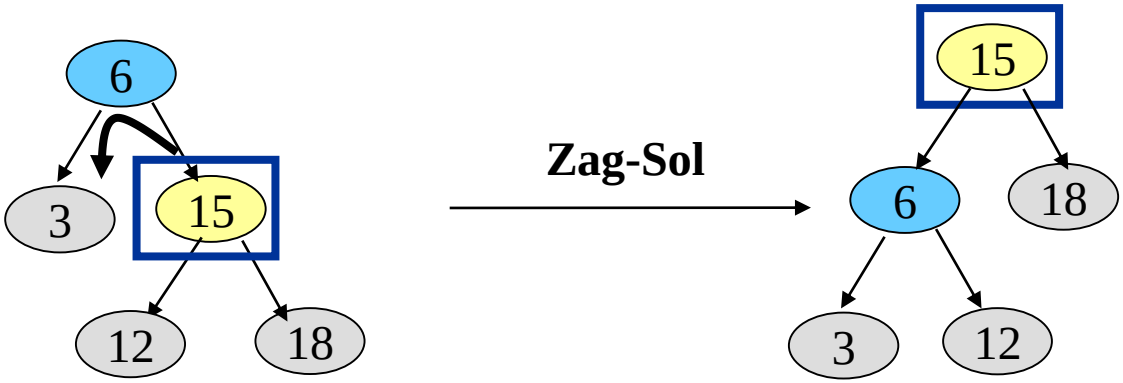
- "Zig-Zig" işlemi aynı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 3 olduğu düşünülürse.



- "zig-zig" işlemi ile birlikte 3 köke taşınmış oldu.
- Not: Aile - ata arasındaki döndürme önce yapılıyor.

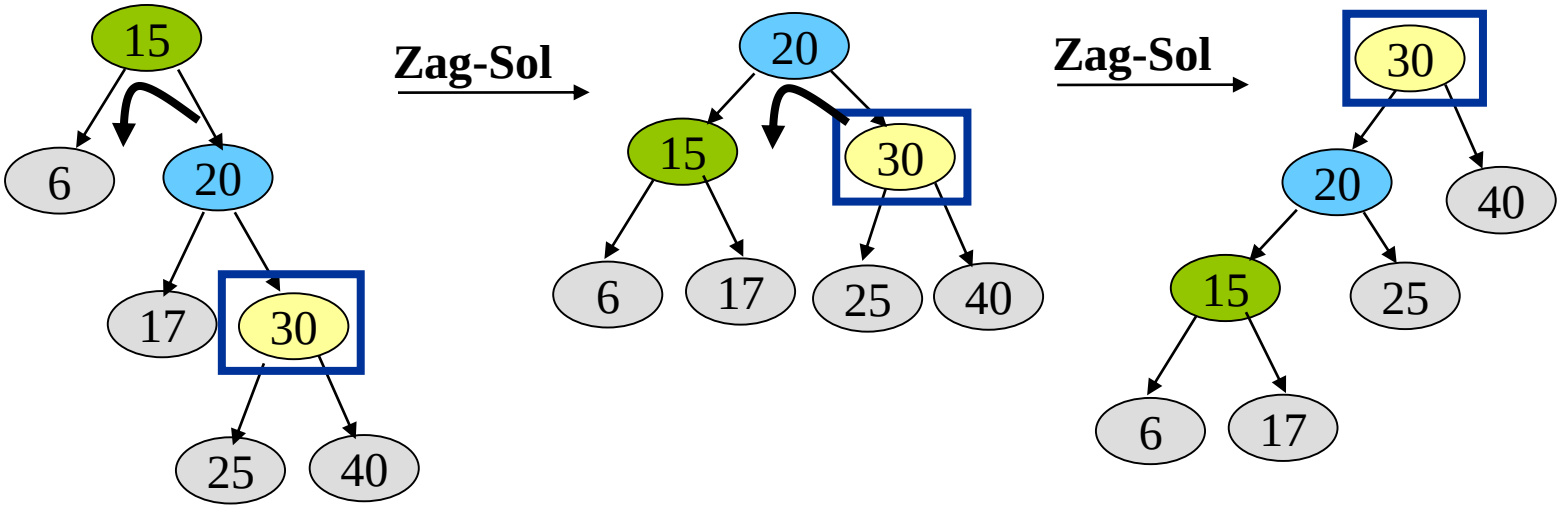
Splay Ağaçları: Zag İşlemi

- “Zag” işlemi AVL ağacındaki gibi tek döndürme işlemidir.
- Örneğin erişilen elemanın 15 olduğu düşünülürse.
- “Zag-sol işlemi 15’i köke taşır.
- Bir sonraki işlemde 15’e O(1) de erişilebilir.
- AVL ağacındaki sol dönme işlemi ile benzerdir.



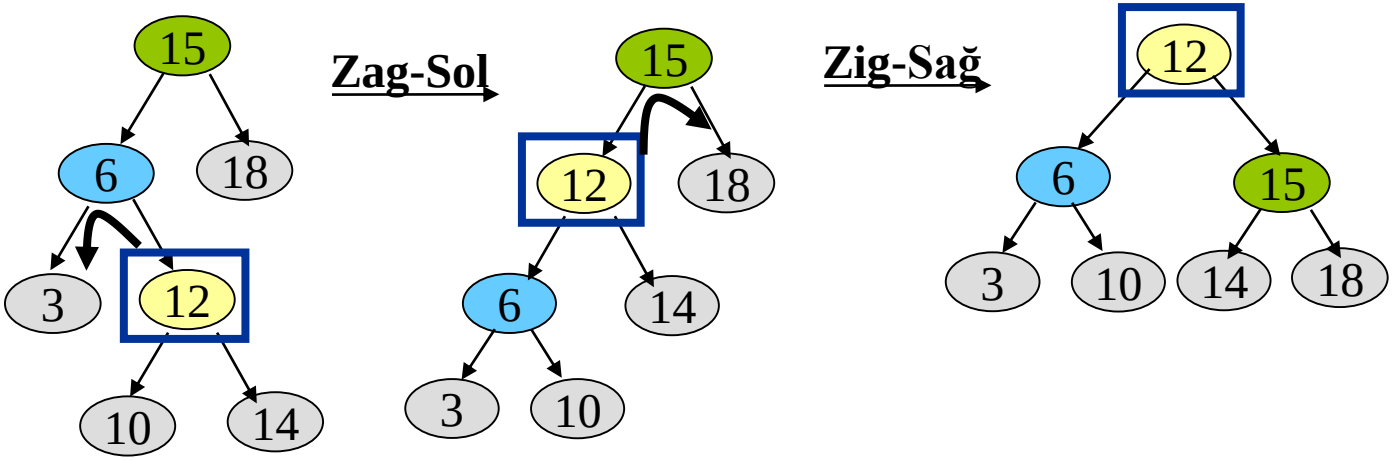
Splay Ağaçları: Zag-Zag İşlemi

- “Zag-Zag” işlemi aynı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 30 olduğu düşünülürse.
- “zag-zag” işlemi ile birlikte 30 köke taşınmış oldu.
- Not: Aile - ata arasındaki döndürme önce yapılıyor.



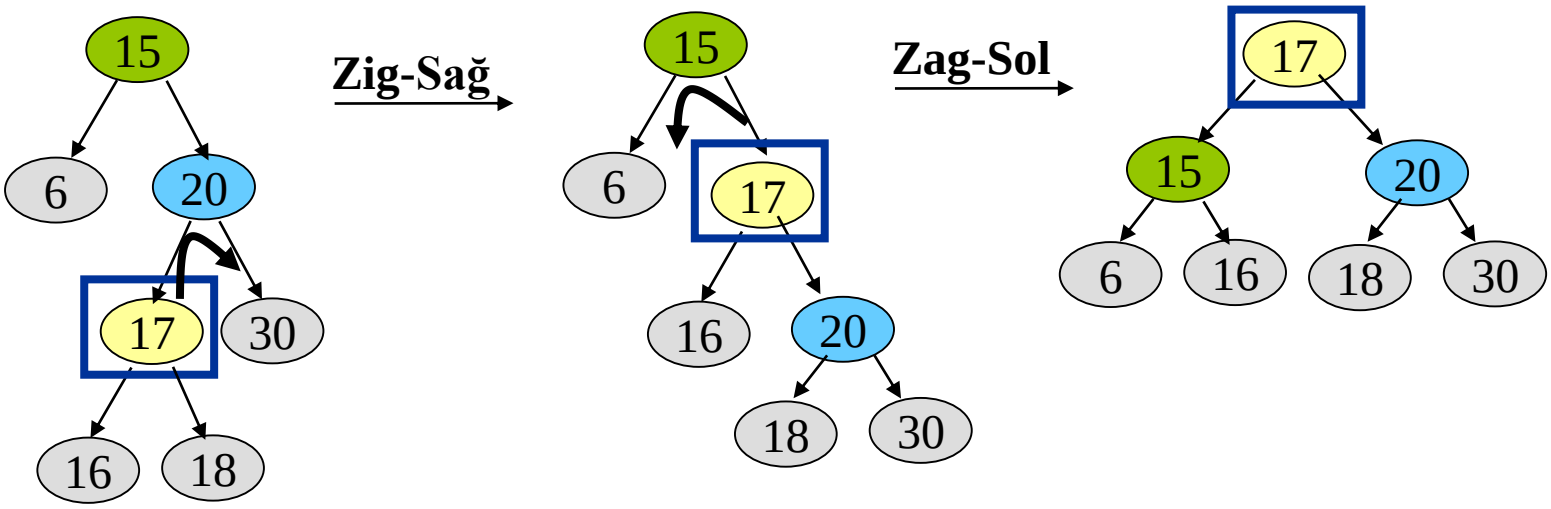
Splay Ağaçları: Zig-Zag durumu

- Zig-zag durumunda X, P nin sağ çocuğu ve G'de atası olduğu durumdur.
- “Zig-Zag” durumu farklı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 12 olduğu düşünülürse.
  - “zig-zag” işlemi ile birlikte 12 köke taşınmış oldu.
  - AVL ağacındaki LR dengesizliğini düzeltmek için kullanılan işlemler ile aynıdır.(önce sol dönme, daha sonra sağ dönme)

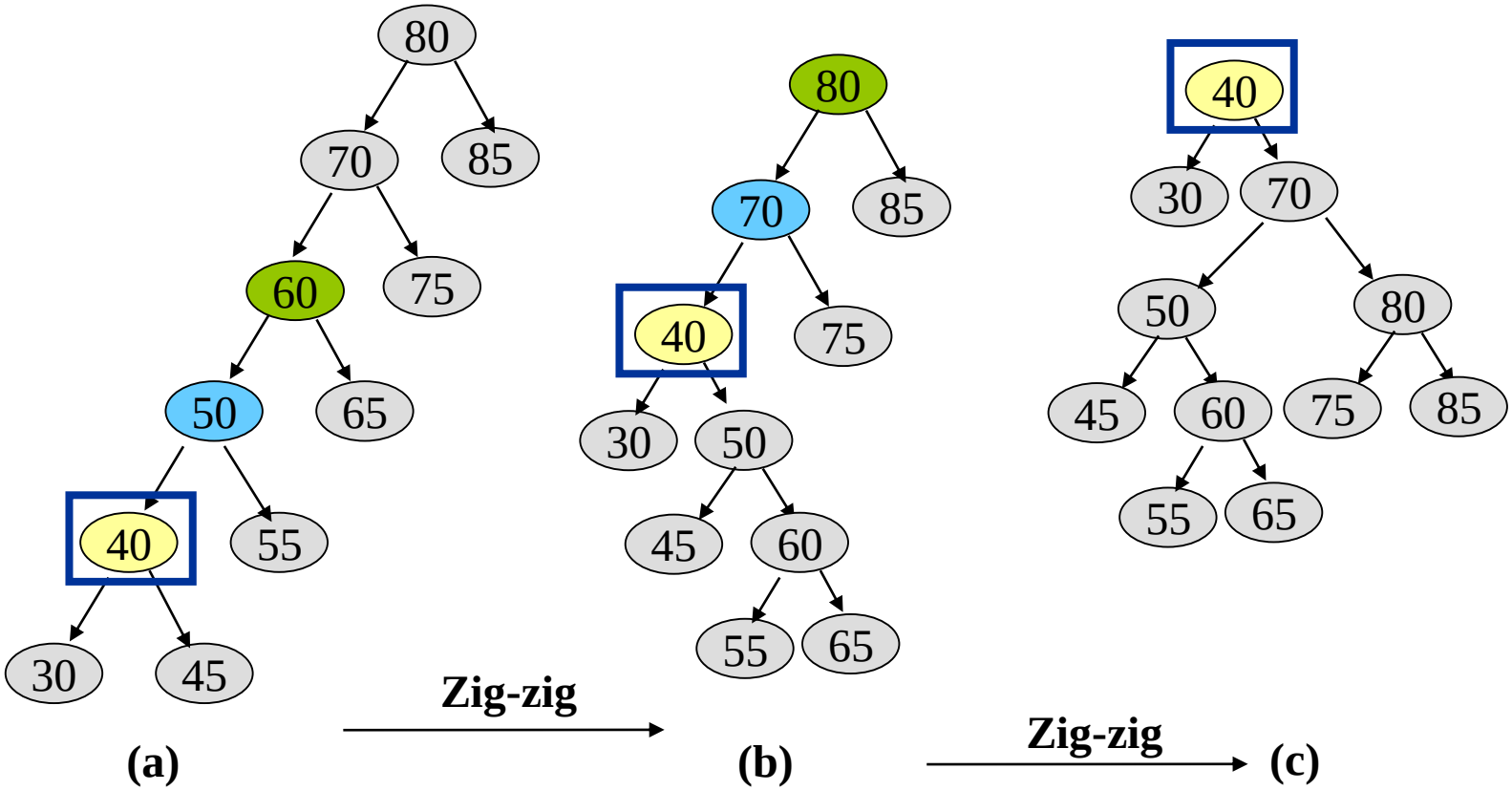


Splay Ağaçları: Zag-Zig durumu

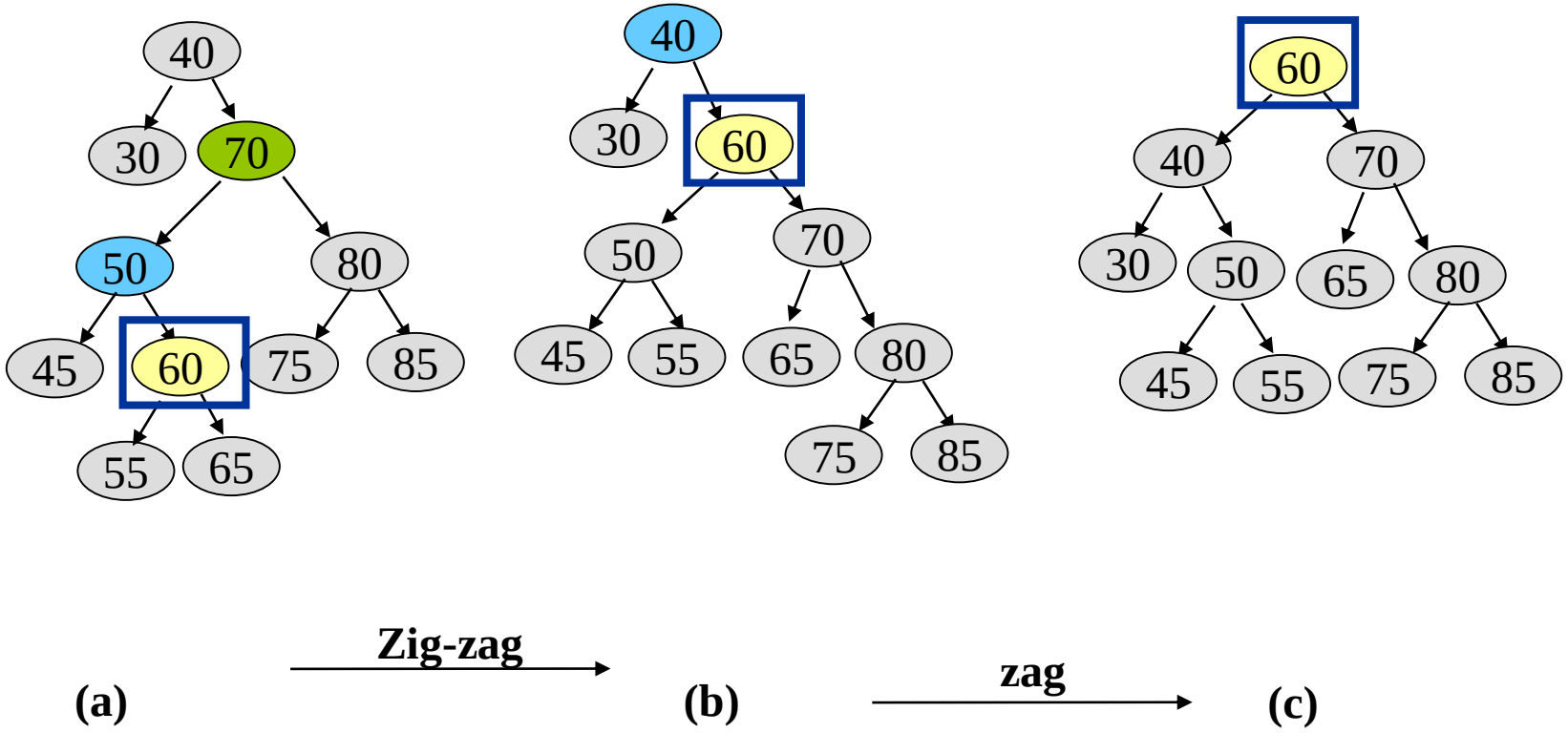
- Zag-Zig durumunda X, P nin sol çocuğu ve G'de atası olduğu durumdur.
- “Zag-Zig” durumu farklı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 17 olduğu düşünülürse.
- zag-zig” işlemi ile birlikte 17 köke taşınmış oldu.
- AVL ağacındaki RL dengesizliğini düzeltmek için kullanılan işlemler ile aynıdır.(önce sağ dönme, daha sonra sol dönme)



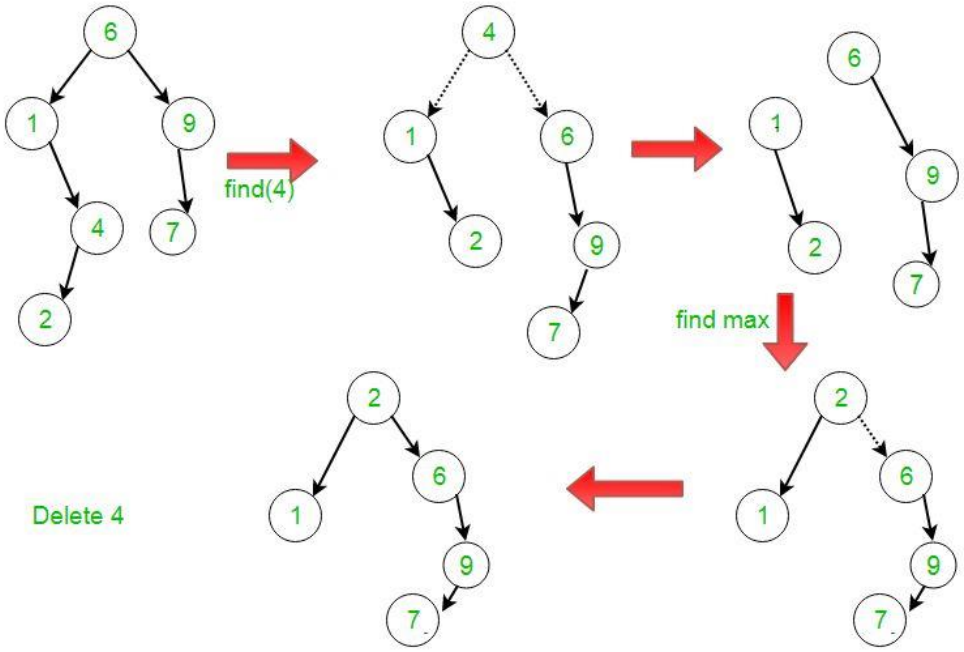
Splay Ağacı Örnek: 40'a Erişildiğinde



Splay Ağacı Örnek: 60'a Erişildiğinde



Splay Ağacı Örnek: 4’ silindiğinde



Diğer İşlemler Sırasında Splaying

- Splaying sadece Arama işleminden sonra değil Ekle/Sil gibi değer işlemlerden sonra da uygulanabilir.
- Ekle X: X yaprak düğüme eklendikten sonra (BST işlemi) X'i köke taşı.
- Sil X: X'i ara ve köke taşı. Kökte bulunan X'i sil ve sol alt ağaçtaki en büyük elemanı veya sağ kökteki en küçük elamanı (doğrudan taşıma olmayacak arama ile aynı) köke taşı.
- Bul X: Eğer X bulunamazsa aramanın sonlandığı yaprak düğümü köke taşı.

Splay Ağaçları – Özet

- Splaying işlemi ile ağaç genel olarak dengede kalıyor.
- Analiz Sonucu: N boyutlu bir Splay ağacı üzerinde k tane işlemin çalışma süresi O(k log N) dir. Dolayısıyla tek bir işlem için çalışma zamanı O(log N) dir.
- Erişilecek elemanların derinliği çok büyük olsa bile, arama işlemlerinin süresi bir süre sonra kısalacaktır. Çünkü her bir arama işlemi ağacın dengelenmesini sağlıyor.

Ödev

- A,V,L, A,Ğ,A,C,I,N,A,E,K,L,E,M,E elamanları sırası ile boş bir AVL ağacına ekleniyor.
- Her bir eleman eklendiğinde oluşan ağaçları çizerek gösteriniz.

Ödev

- S,P,L,A,Y,A,Ğ,A,C,I,N,A,E,K,L,E,M,E elemanları sırası ile boş bir Splay ağacına ekleniyor.
- Oluşan ağacı çizerek gösteriniz.
- Önce “S" ye sonra "A" ya erişimi sağlandıktan sonra ağacın yapısını çizerek gösteriniz.
- Aşağıda verilen splay ağacına ait C programını Java veya C# ile yapınız.

Örnek Program C++

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
struct node
{     int data;
      struct node *parent;
      struct node *left;
      struct node *right;
};
int data_print(struct node *x);
struct node *rightrotation(struct node *p,struct node *root);
struct node *leftrotation(struct node *p,struct node *root);
void splay (struct node *x, struct node *root);
struct node *insert(struct node *p,int value);
struct node *inorder(struct node *p);
struct node *delete(struct node *p,int value);
struct node *successor(struct node *x);
struct node *lookup(struct node *p,int value);
```

Örnek Program C++

```
void splay (struct node *x, struct node *root)
{       struct node *p,*g;
        /*check if node x is the root node*/
        if(x==root)             return;
        /*Performs Zig step*/
        else if(x->parent==root)
        {
                if(x==x->parent->left)          root=rightrotation(root,root);
                else                    root=leftrotation(root,root);
        }
        else
        {       p=x->parent; /*now points to parent of x*/
                g=p->parent; /*now points to parent of x's parent*/
                /*Performs the Zig-zig step when x is left and x's parent is left*/
                if(x==p->left&&p==g->left)
                {       root=rightrotation(g,root);
                        root=rightrotation(p,root);
                }
```

Örnek Program C++

```
                /*Performs the Zig-zig step when x is right and x's parent is right*/
                else if(x==p->right&&p==g->right)
                {       root=leftrotation(g,root);
                        root=leftrotation(p,root);
                }
                /*Performs the Zig-zag step when x's is right and x's parent is left*/
                else if(x==p->right&&p==g->left)
                {       root=leftrotation(p,root);
                        root=rightrotation(g,root);
                }
                /*Performs the Zig-zag step when x's is left and x's parent is right*/
                else if(x==p->left&&p==g->right)
                {       root=rightrotation(p,root);
                        root=leftrotation(g,root);
                }
                splay(x, root);
        }
    }
```

Örnek Program C++

```
struct node *rightrotation(struct node *p,struct node *root)
{     struct node *x;
      x = p->left;
      p->left = x->right;
      if (x->right!=NULL) x->right->parent = p;
      x->right = p;
      if (p->parent!=NULL)
            if(p==p->parent->right) p->parent->right=x;
            else
                  p->parent->left=x;
      x->parent = p->parent;
      p->parent = x;
      if (p==root)        return x;
      else            return root;
}
```

Örnek Program C++

```
struct node *leftrotation(struct node *p,struct node *root)
{       struct node *x;
        x = p->right;
        p->right = x->left;
        if (x->left!=NULL) x->left->parent = p;
        x->left = p;
        if (p->parent!=NULL)
                if (p==p->parent->left) p->parent->left=x;
                else
                        p->parent->right=x;
        x->parent = p->parent;
        p->parent = x;
        if(p==root)
                return x;
        else
                return root;
}
```

Örnek Program C++

```
struct node *insert(struct node *p,int value)
{     struct node *temp1,*temp2,*par,*x;
      if(p == NULL)
      {     p=(struct node *)malloc(sizeof(struct node));
            if(p != NULL)
            {     p->data = value;
                  p->parent = NULL;
                  p->left = NULL;
                  p->right = NULL;
            }
            else
            {     printf("No memory is allocated\n");
                  exit(0);
            }
            return(p);
      } //the case 2 says that we must splay newly inserted node to root
```

Örnek Program C++

```
else    {    temp2 = p;
             while(temp2 != NULL)
             {    temp1 = temp2;
                  if(temp2->data > value)        temp2 = temp2->left;
                  else if(temp2->data < value)   temp2 = temp2->right;
                  else
                       if(temp2->data == value)     return temp2;
             }
             if(temp1->data > value)
             {    par = temp1;//temp1 having the parent address,so that's it
                  temp1->left = (struct node *)malloc(sizeof(struct node));
                  temp1= temp1->left;
                  if(temp1 != NULL)
                  {    temp1->data = value;
                       temp1->parent = par;//store the parent address.
                       temp1->left = NULL;     temp1->right = NULL;                  }
                  else    {    printf("No memory is allocated\n");      exit(0);        }
             }
```

Örnek Program C++

```
                else
                {   par = temp1;//temp1 having the parent node address.
                    temp1->right = (struct node *)malloc(sizeof(struct node));
                    temp1 = temp1->right;
                    if(temp1 != NULL)
                    {   temp1->data = value;
                        temp1->parent = par;//store the parent address
                        temp1->left = NULL;   temp1->right = NULL;
                    }
                    else   {   printf("No memory is allocated\n");   exit(0);   }
                }
        }
        splay(temp1,p);//temp1 will be new root after splaying
        return (temp1);
    }
```

Örnek Program C++

```
struct node *inorder(struct node *p)
{
    if(p != NULL)
    {
        inorder(p->left);
        printf("CURRENT %d\t",p->data);
        printf("LEFT %d\t",data_print(p->left));
        printf("PARENT %d\t",data_print(p->parent));
        printf("RIGHT %d\t\n",data_print(p->right));
        inorder(p->right);
    }
}
```

Örnek Program C++

```
struct node *delete(struct node *p,int value)
{     struct node *x,*y,*p1;       struct node *root;       struct node *s;       root = p;
      x = lookup(p,value);
      if(x->data == value)
      {     //if the deleted element is leaf
            if((x->left == NULL) && (x->right == NULL))
            {     y = x->parent;
                  if(x ==(x->parent->right))     y->right = NULL;
                  else     y->left = NULL;          free(x);
            }
            //if deleted element having left child only
            else if((x->left != NULL) &&(x->right == NULL))
            {     if(x == (x->parent->left))
                  {  y = x->parent;    x->left->parent = y;    y->left = x->left;      free(x);         }
                  else   {  y = x->parent;  x->left->parent = y; y->right = x->left;   free(x);       }
            }
```

Örnek Program C++

```
//if deleted element having right child only
 else if((x->left == NULL) && (x->right != NULL))
 {   if(x == (x->parent->left)) { y = x->parent; x->right->parent = y;  y->left = x->right;  free(x); }
    else {  y = x->parent;  x->right->parent = y;  y->right = x->right;  free(x); }
    }
          //if the deleted element having two children
 else if((x->left != NULL) && (x->right != NULL))
          {   if(x == (x->parent->left))
             {  s = successor(x);
                if(s != x->right)
                   { y = s->parent;
                          if(s->right != NULL)
                          { s->right->parent = y; y->left = s->right; }
                          else y->left = NULL;
                          s->parent = x->parent;    x->right->parent = s;
                          x->left->parent = s;        s->right = x->right;
                          s->left = x->left;            x->parent->left = s;
                     }
```

Örnek Program C++

```
else    { y = s;   s->parent = x->parent;
          x->left->parent = s;   s->left = x->left;      x->parent->left = s;   }
          free(x);     }
               else if(x == (x->parent->right))
               {  s = successor(x);
                    if(s != x->right)
                    {    y = s->parent;
                         if(s->right != NULL) {s->right->parent = y; y->left = s->right;  }
                         else y->left = NULL;
                         s->parent = x->parent; x->right->parent = s;
                         x->left->parent = s; s->right = x->right;
                         s->left = x->left; x->parent->right = s;       }
                    else  { y = s; s->parent = x->parent; x->left->parent = s;
                      s->left = x->left; x->parent->right = s;                    }
                    free(x);
               }    }      splay(y,root);
     }        else      { splay(x,root); }
   }
```

Örnek Program C++

```
struct node *successor(struct node *x)
{   struct node *temp,*temp2;   temp=temp2=x->right;
      while(temp != NULL)  {   temp2 = temp;   temp = temp->left;  }
      return temp2; }
//p is a root element of the tree
struct node *lookup(struct node *p,int value)
{   struct node *temp1,*temp2;
      if(p != NULL)
      {  temp1 = p;
            while(temp1 != NULL)
            {  temp2 = temp1;
                  if(temp1->data > value)  temp1 = temp1->left;
                  else if(temp1->data < value) temp1 = temp1->right;
                  else return temp1;            }
            return temp2;
      }
      else { printf("NO element in the tree\n"); exit(0); }
}
```

Örnek Program C++

```
struct node *search(struct node *p,int value)
{      struct node *x,*root;        root = p;       x = lookup(p,value);
       if(x->data == value) {  printf("Inside search if\n"); splay(x,root); }
       else { printf("Inside search else\n"); splay(x,root); }
}
main()
{ struct node *root;//the root element
  struct node *x;//x is which element will come to root.
  int i; root = NULL; int choice = 0; int ele;
       while(1)
       { printf("\n\n 1.Insert"); printf("\n\n 2.Delete"); printf("\n\n 3.Search"); printf("\n\n
4.Display\n");
         printf("\n\n Enter your choice:");scanf("%d",&choice);
            if(choice==5)  exit(0);
            switch(choice)
            { case 1: printf("\n\n Enter the element to be inserted:");
                      scanf("%d",&ele); x = insert(root,ele);
                      if(root != NULL) { splay(x,root);}
                      root = x;   break;
```

Örnek Program C++

```
struct node *search(struct node *p,int value)
{      struct node *x,*root;        root = p;       x = lookup(p,value);
       if(x->data == value) {  printf("Inside search if\n"); splay(x,root); }
       else { printf("Inside search else\n"); splay(x,root); }
}
main()
{ struct node *root;//the root element
  struct node *x;//x is which element will come to root.
  int i; root = NULL; int choice = 0; int ele;
       while(1)
       { printf("\n\n 1.Insert"); printf("\n\n 2.Delete"); printf("\n\n 3.Search"); printf("\n\n
4.Display\n");
         printf("\n\n Enter your choice:");scanf("%d",&choice);
            if(choice==5)  exit(0);
            switch(choice)
            { case 1: printf("\n\n Enter the element to be inserted:");
                      scanf("%d",&ele); x = insert(root,ele);
                      if(root != NULL) { splay(x,root);}
                      root = x;   break;
```

Örnek Program C++

```
                case 2: if(root == NULL) { printf("\n Empty tree..."); continue; }
                    printf("\n\n Enter the element to be delete:");
                    scanf("%d",&ele); root = delete(root,ele); break;
                case 3: printf("Enter the element to be search\n");
                    scanf("%d",&ele); x = lookup(root,ele); splay(x,root);      root = x; break;
                case 4: printf("The elements are\n");    inorder(root); break;
                default: printf("Wrong choice\n"); break;
            }    }}
int data_print(struct node *x)
{ if ( x==NULL ) return 0;
    else return x->data;  }
/*some suggestion this code is not fully functional for example
if you have inserted some elements then try to delete root then it may not work
because we are calling right and left child of a null value(parent of root)
which is not allowed and will give segmentation fault
Also for inserting second element because of splaying twice(once in insert and one in main)
will give error So I have made those changes but mainly in my cpp( c plus plus file) file,
but I guess wiki will itself look into this and made  these changes */
```

Örnek SPLAY Tree Silme C++

```
// C implementation to delete a node from Splay Tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the given key and
    NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key   = key;
    node->left  = node->right  = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}
```

Örnek SPLAY Tree Silme C++

```
// A utility function to left rotate subtree rooted with x // See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}
// This function brings the key at root if key is present in tree. // If key is not present, then it brings the
last accessed item at  // root.  This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;
    // Key lies in left subtree
    if (root->key > key)
    {
// Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)         {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is       // done after else
            root = rightRotate(root);
        }
```

Örnek SPLAY Tree Silme C++

```
else if (root->left->key < key) // Zig-Zag (Left Right)
        {    // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);
               // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
        // Do second rotation for root
        return (root->left == NULL)? root: rightRotate(root);
    }
    else // Key lies in right subtree
    {   // Key is not in tree, we are done
        if (root->right == NULL) return root;

        // Zag-Zig (Right Left)
        if (root->right->key > key)
        {   // Bring the key as root of right-left
            root->right->left = splay(root->right->left, key);
            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)// Zag-Zag (Right Right)
        {   // Bring the key as root of right-right and do
            // first rotation
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }
        // Do second rotation for root
        return (root->right == NULL)? root: leftRotate(root);
    }
}
```

Örnek SPLAY Tree Silme C++

```
// The delete function for Splay tree. Note that this function
// returns the new root of Splay Tree after removing the key
struct node* delete_key(struct node *root, int key)
{
    struct node *temp;
    if (!root)
        return NULL;

    // Splay the given key
    root = splay(root, key);

    // If key is not present, then
    // return root
    if (key != root->key)
        return root;

    // If key is present
    // If left child of root does not exist
    // make root->right as root
    if (!root->left)
    {
        temp = root;
        root = root->right;
    }
```

Örnek SPLAY Tree Silme C++

```
// Else if left child exits
    else
    {
        temp = root;

        /*Note: Since key == root->key,
        so after Splay(key, root->lchild),
        the tree we get will have no right child tree
        and maximum node in left subtree will get splayed*/
        // New root
        root = splay(root->left, key);

        // Make right child of previous root  as
        // new root's right child
        root->right = temp->right;
    }

    // free the previous root node, that is,
    // the node containing the key
    free(temp);

    // return root of the new Splay Tree
    return root;

}
```

Örnek SPLAY Tree Silme C++

```
// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
/* Driver program to test above function*/
int main()
{    // Splay Tree Formation
    struct node *root = newNode(6);
    root->left = newNode(1);
    root->right = newNode(9);
    root->left->right = newNode(4);
    root->left->right->left = newNode(2);
    root->right->left = newNode(7);

     int key = 4;

    root = delete_key(root, key);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}
```

Dengeli Arama Ağaçları

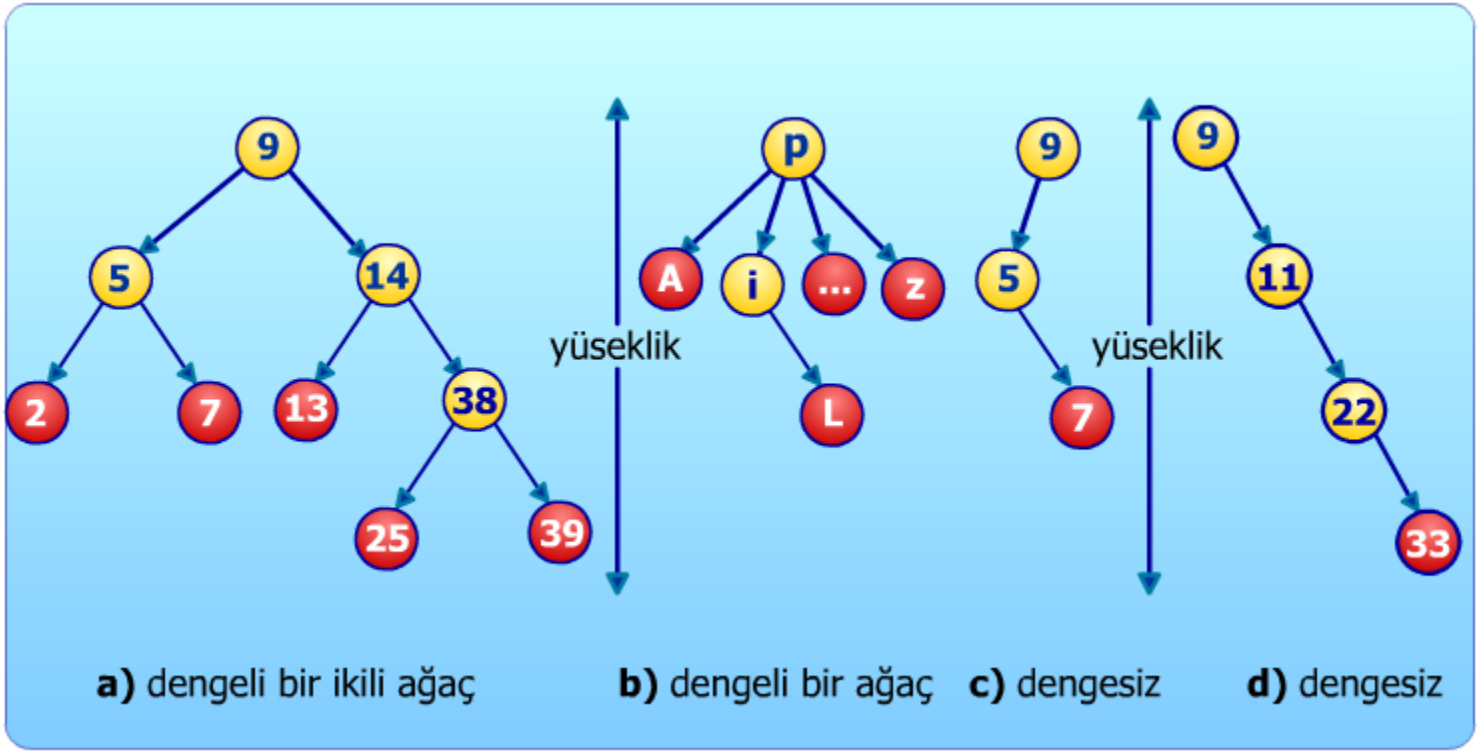
(Balanced Search Tree)

- AVL tree
- 2-3 ve 2-3-4 tree
- Splay tree
- Red-Black Tree
- B- tree

DENGELİ AĞAÇ (BALANCED TREE)

- Dengeli ağaç (balanced tree), gelişmesini tüm dallarına homojen biçimde yansıtan ağaç şeklidir; tanım olarak, herhangi bir düğümüne bağlı altağaçların yükseklikleri arasındaki fark, şekil a) ve b)'de görüldüğü gibi, **en fazla 1 (bir)** olmalıdır. Bir dalın fazla büyümesi ağacın dengesini bozar ve ağaç üzerine hesaplanmış karmaşıklık hesapları ve yürütme zamanı bağıntılarından sapılır. Dolayısıyla ağaç veri modelinin en önemli getirisi kaybolmaya başlar.
- Yapılan istatistiksel çalışmalarda, ağacın oluşturulması için gelen verilerin rastgele olması durumda ağacın dengeli olduğu veya çok az sapma gösterdiği gözlenmiştir [HIBBARD-1962].

DENGELİ AĞAÇ (BALANCED TREE)



Yükseklik Dengeli Ağaçlar

- BST operasyonlarını daha kısa sürede gerçekleştirmek için pek çok BST dengeleme algoritması vardır. Bunlardan bazıları;
  - Adelson-Velsky ve Landis (AVL) ağaçları (1962)
  - Splay ağaçları (1978)
  - B-ağacı ve diğer çok yönlü arama ağaçları (1972)
  - Red-Black ağaçları (1972)
    - Ayrıca Simetrik İkili B-Ağaçları(Symmetric Binary B-Trees) şeklinde de bilinir

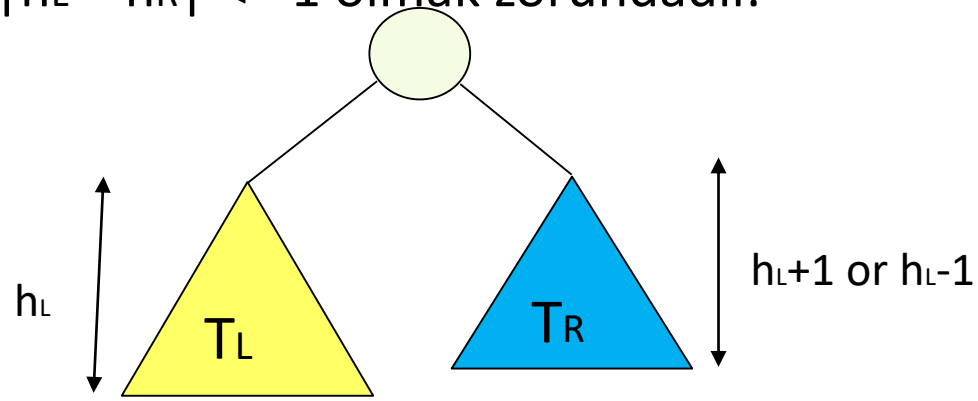
AVL TREE

AVL AĞAÇLARI

- AVL(G.M. Adelson-Velskii and E.M. Landis) yöntemine göre kurulan bir ikili arama ağacı, ikili AVL arama ağacı olarak adlandırılır.
- AVL ağacının özelliği, sağ alt ağaç ile sol alt ağaç arasındaki yükseklik farkının en fazla bir düğüm olmasıdır. Bu kural ağacın tüm düğümleri için geçerlidir.
- Normal ikili arama ağaçları için ekleme ve silme algoritmaları, ikili AVL ağaçları üzerinde ağacın yanlış şekil almasına, yani ağacın dengesinin bozulmasına neden olur.

AVL Ağaçları: Tanım

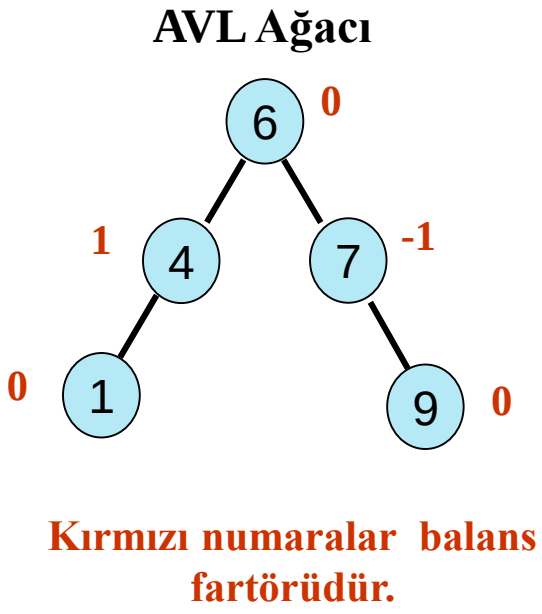
1. Tüm boş ağaç AVL ağacıdır.
2. Eğer T boş olmayan T_L ve T_R şeklinde sol ve sağ alt ağaçları olan ikili arama ağacı ise, T ancak ve ancak aşağıdaki şartları sağlarsa AVL ağacı şeklinde isimlendirilir.
    1. T_L ve T_R AVL ağaçları ise
    2. h_L ve h_R T_L ve T_R nin yükseklikleri olmak üzere $|h_L - h_R| <= 1$ olmak zorundadır.



AVL Ağaçları

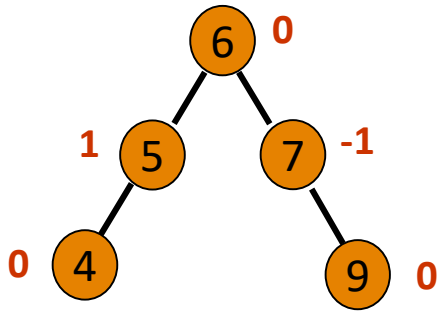
- AVL ağaçları dengeli ikili arama ağaçlarıdır.
- solh= yükseklik(sol altağaç), ve sagh=yükseklik(sağ altağaç) ise
- Bir düğümdeki denge faktörü= solh - sagh

- AVL ağaçlarında balans faktörü sadece -1, 0, 1 olabilir.
  - **Eşit ise** **0**
  - **Sol sazla ise** **1**
  - **Sağ fazla ise** **-1**

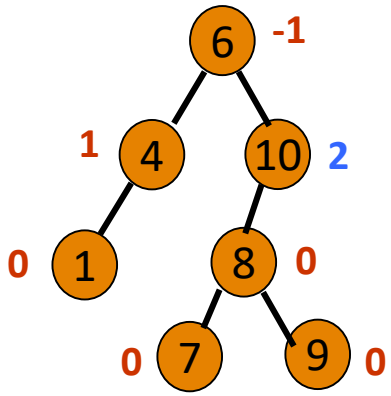


- Her bir düğümün sol ve sağ alt ağaçlarının yükseklikleri arasındaki fark en fazla 1 olabilir.

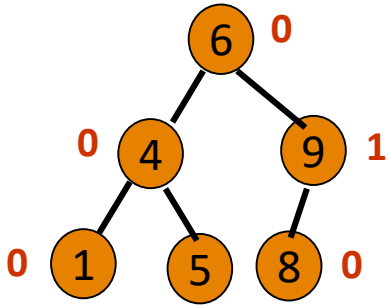
AVL Ağaçları: Örnekler



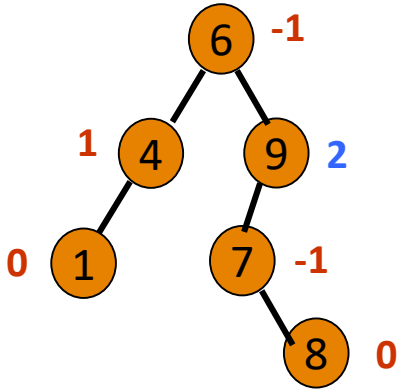
AVL Ağacı



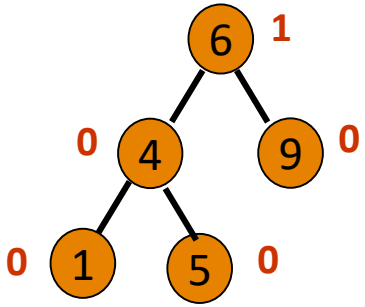
AVL Ağacı Değildir



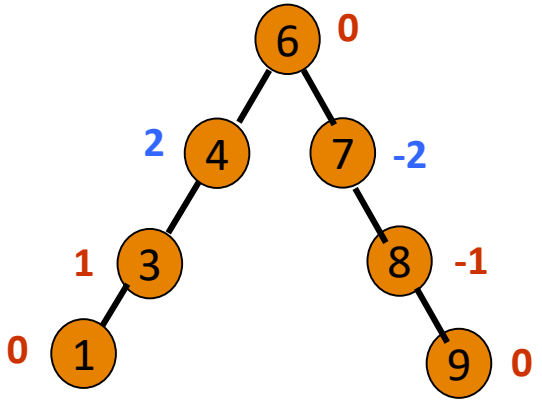
AVL Ağacı



AVL Ağacı Değildir



AVL Ağacı

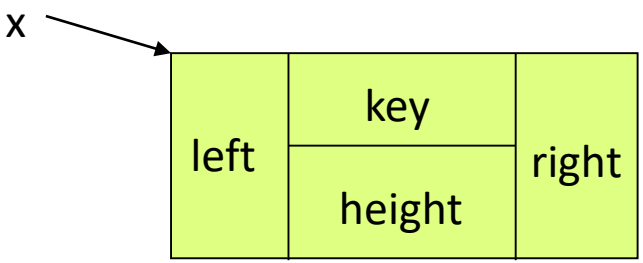


AVL Ağacı Değildir

Kırmızı numaralar balans faktörüdür.

AVL Ağacı: Gerçekleştirim

- AVL ağacının gerçekleştirimi için her x düğümünün yüksekliği kaydedilir.
- x'in balans faktörü = x'in sol alt ağacının yüksekliği – x'in sağ alt ağacının yüksekliği
- AVL ağaçlarında, "bf" sadece {-1, 0, 1} değerlerini alabilir.



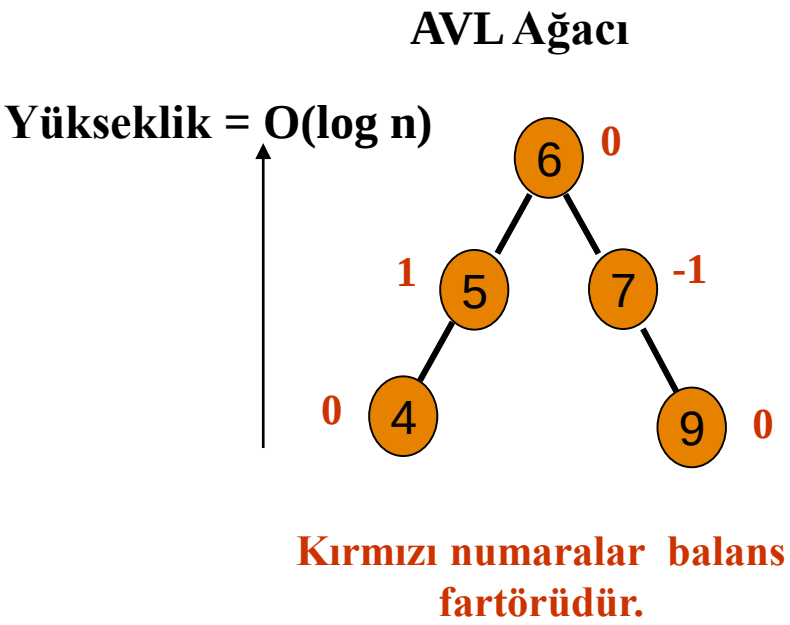

```
class AVLDugumu
{
        int deger;
        int yukseklik;
        AVLDugumu sol;
        AVLDugumu sag;
}
```

AVL AĞAÇLARI

- Normal ikili arama ağaçlarında fonksiyonların en kötü çalışma süreleri Tw = O (n) dir. Bu durum lineer zaman olduğundan çok kötü bir sonuç yaratır.
- AVL ağaç veri modeli oluşturularak bu kötü çalışma süresi ortadan kaldırılır.
- AVL ağaçları için Tw = O (log n)'dir.

AVL Ağaçları

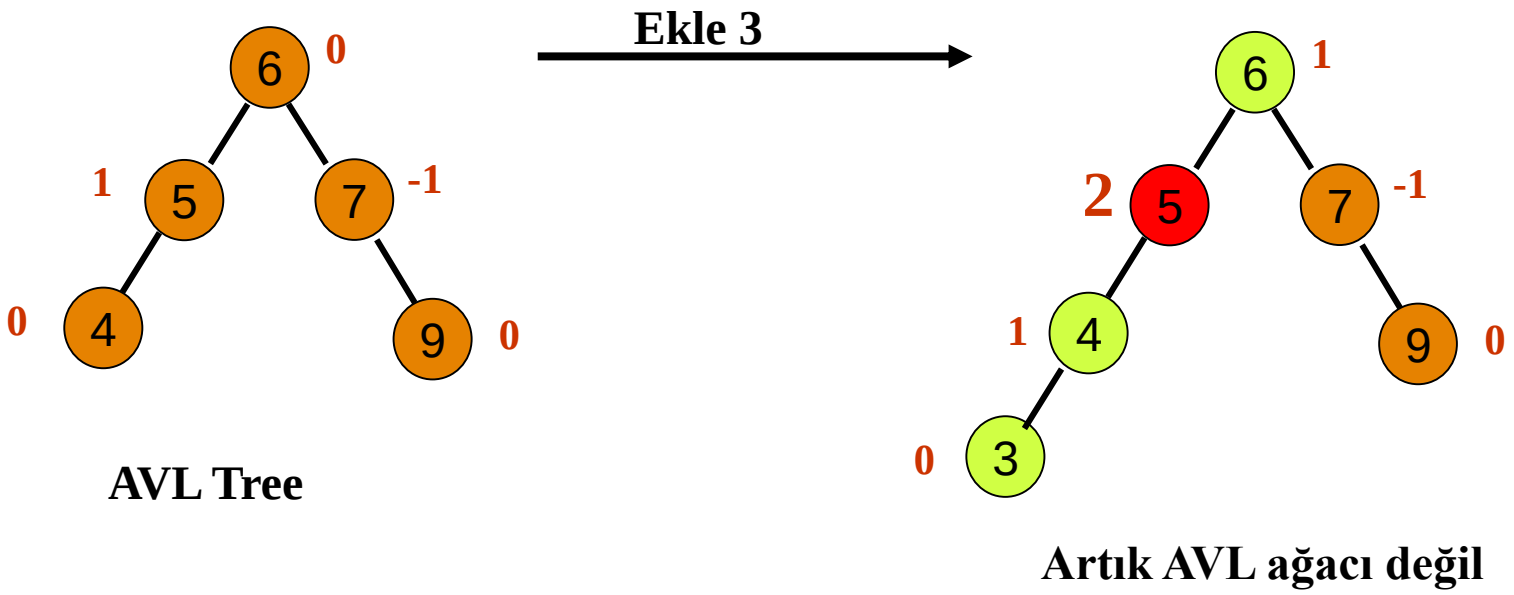
- N düğümlü bir AVL ağacının yüksekliği daima O(log n) dir.
- Peki nasıl?



Kırmızı numaralar balans fartörüdür.

AVL Ağaçları: Şanslı ve Şanssız

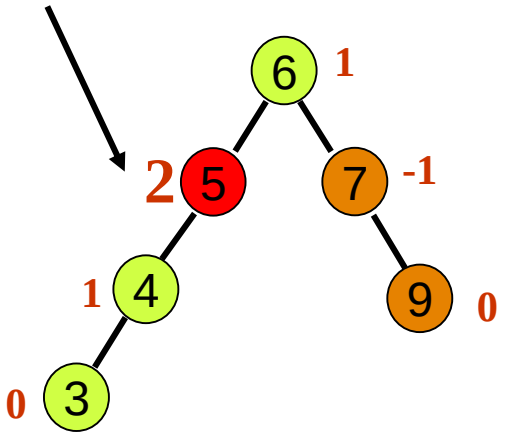
- Şanslı:
  - Arama süresi O(h) = O(log n)
- Şansız
  - Ekleme ve silme işlemleri ağacın dengesiz olmasına neden olabilir.



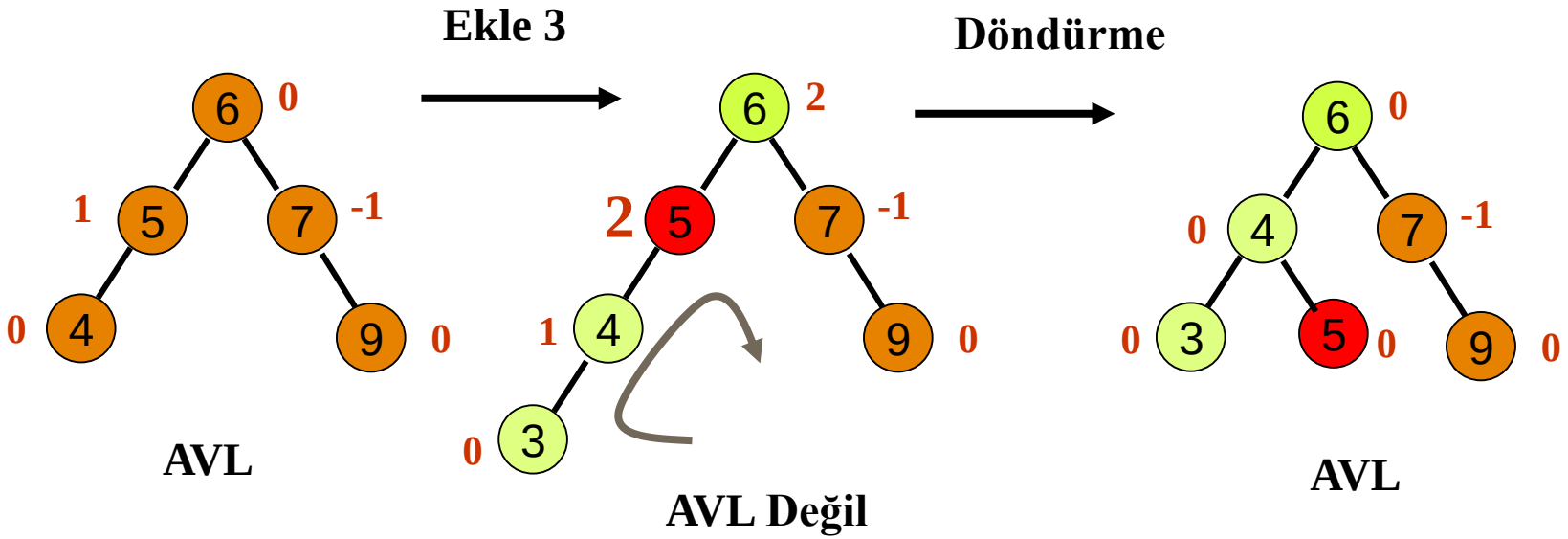
AVL Ağacında Dengenin Sağlanması

- Problem: Ekleme işlemi bazı durumlarda ekleme noktasına göre kök olan bölgelerde balans faktörün 2 veya -2 olmasına neden olabilir.

- Fikir: Yeni düğümü ekledikten sonra
  1. Balans faktörü düzelterek köke doğru çık.
  2. Eğer düğümün balans faktörü 2 veya -2 ise ağaç bu düğüm üzerinde döndürülerek düzeltilir.

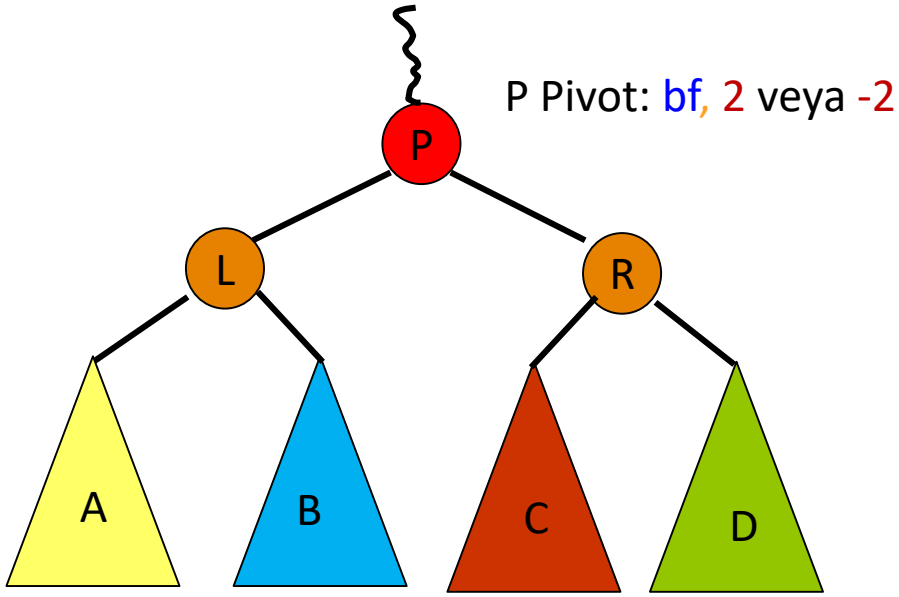


Denge Sağlama: Örnek



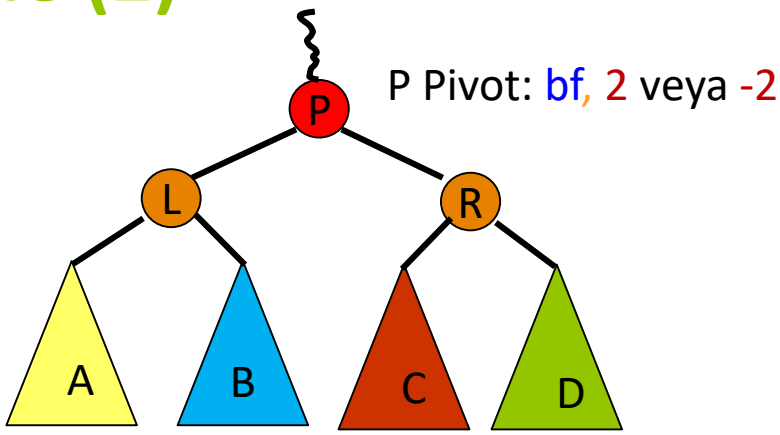
- Yeni düğümü ekledikten sonra
  1. Balans faktörü düzelterek köke doğru çık.
  2. Eğer düğümün balans faktörü 2 veya -2 ise ağaç bu düğüm üzerinde döndürülerek düzeltilir.

AVL Ağacı - Ekleme (1)



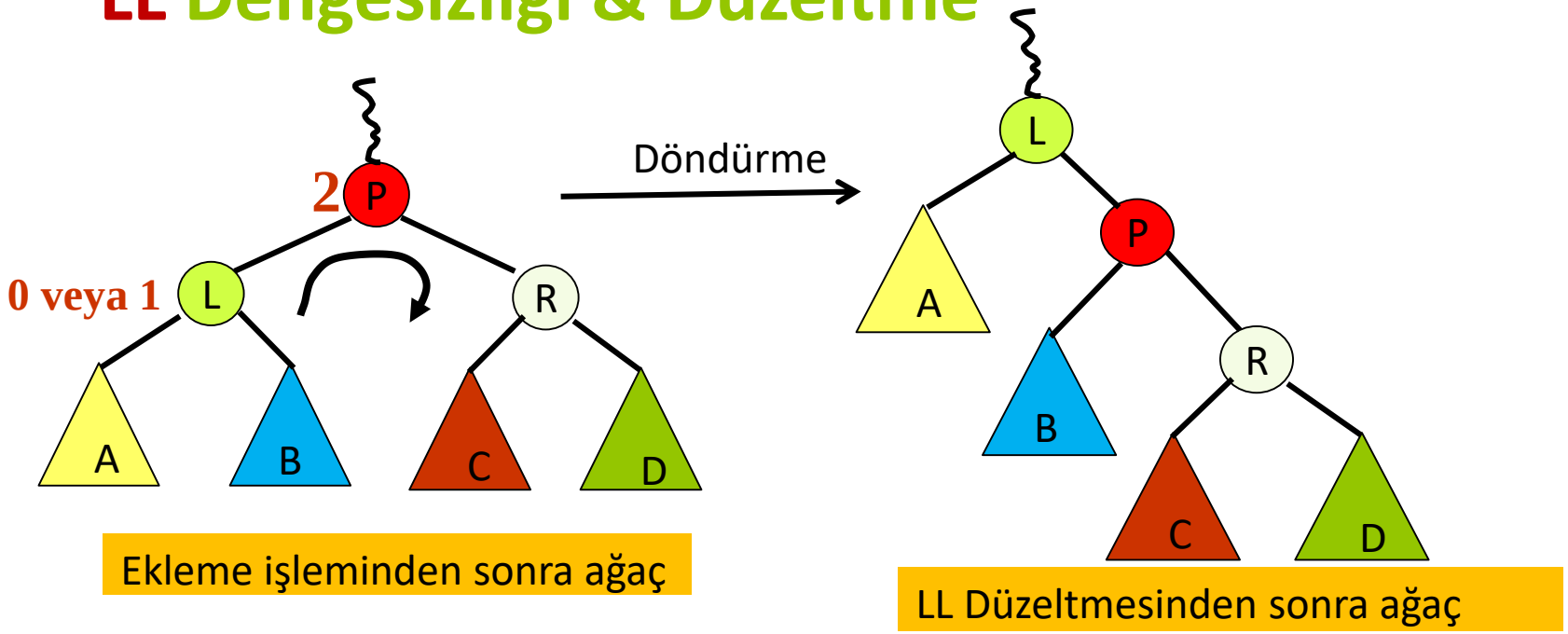
- P düğümünün dengeyi bozan düğüm olduğu düşünülürse.
  - P pivot düğüm şeklinde isimlendirilir.
  - Eklemeden sonra köke doğru çıkarken bf 'nin 2 veya -2 olduğu ilk düğümdür.

AVL Ağacı - Ekleme (2)



- 4 farklı durum vardır:
  - **Dış Durum** (tek döndürme gerektiren) :
    1. P'nin sol alt ağacının soluna eklendiğinde (LL Dengesizliği).
    2. P'nin sağ alt ağacının sağına eklendiğinde (RR Dengesizliği)
  - **İç Durum** (2 kez döndürme işlemi gerektiren) :
    3. P'nin sol alt ağacının sağına eklendiğinde (RL Dengesizliği)
    4. P'nin sağ alt ağacının soluna eklendiğinde(LR Dengesizliği)

LL Dengesizliği & Düzeltme

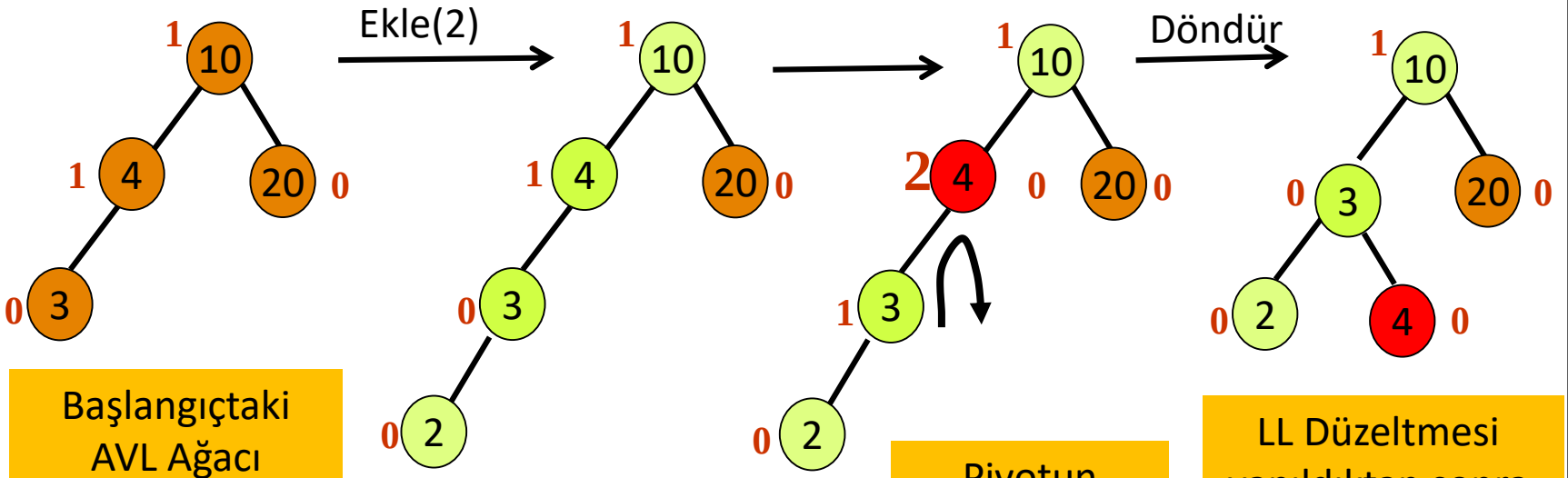


Ekleme işleminden sonra ağaç

LL Düzeltmesinden sonra ağaç

- LL Dengesizliği: P'nin sol alt ağacının soluna eklendiğimizde (A alt ağacınına)
  - P 'nin bf si 2
  - L 'nin bf si 0 veya 1
- Düzeltme: P etrafında sağa doğru tek dönderme.

LL Dengesizliği Düzeltme Örneği (1)



Başlangıçtaki AVL Ağacı

2 eklendikten sonra ağacın durumu

Pivotun belirlenmesi

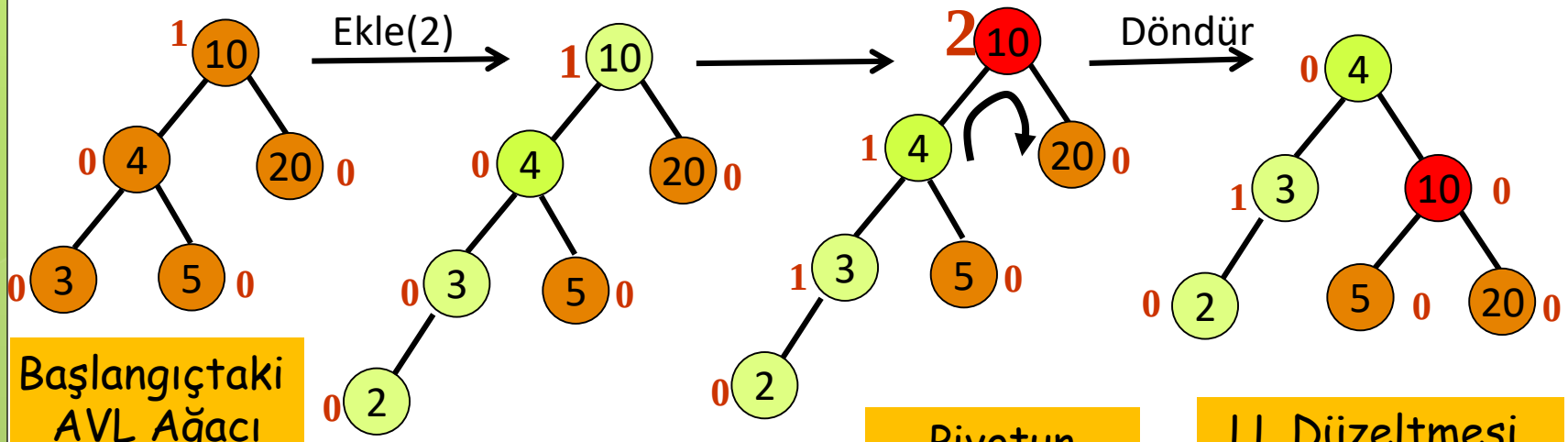
LL Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

- LL Dengesizliği:
  - **P(4)**'ün **bf** si 2
  - **L(3)**'ün **bf** si 0 veya 1

LL Dengesizliği Düzeltme Örneği (2)



Başlangıçtaki AVL Ağacı

2 eklendikten sonra ağacın durumu

Pivotun belirlenmesi

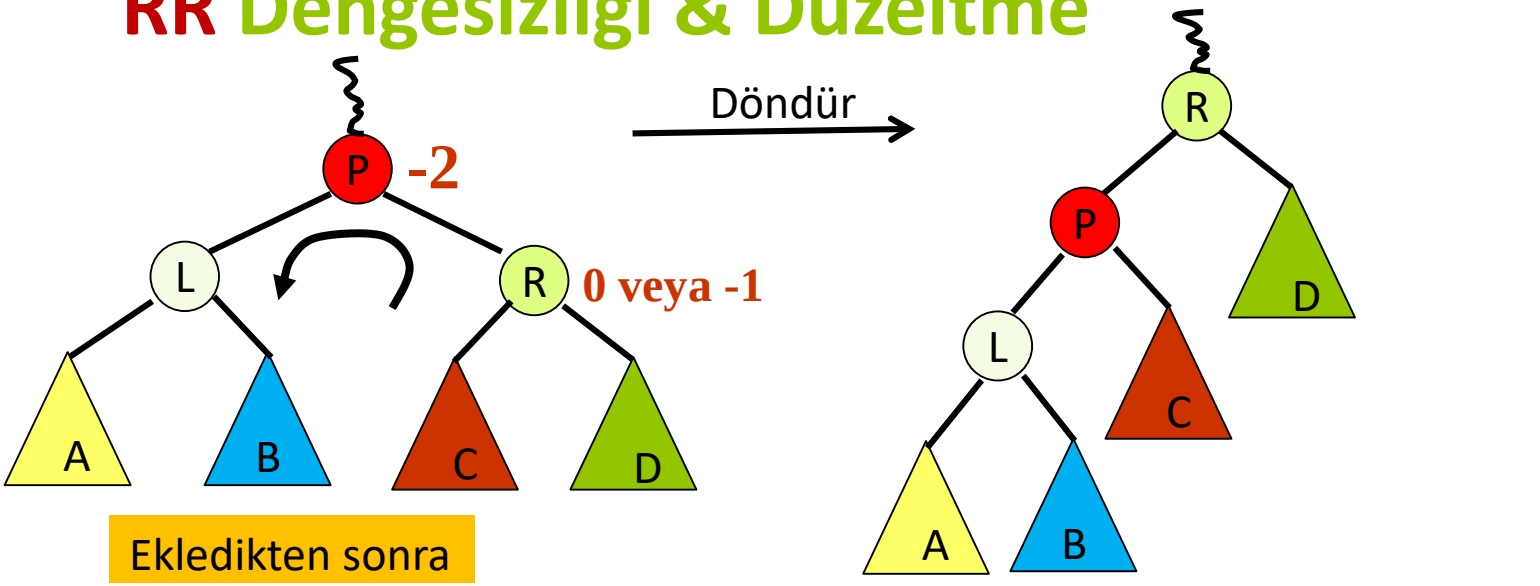
LL Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

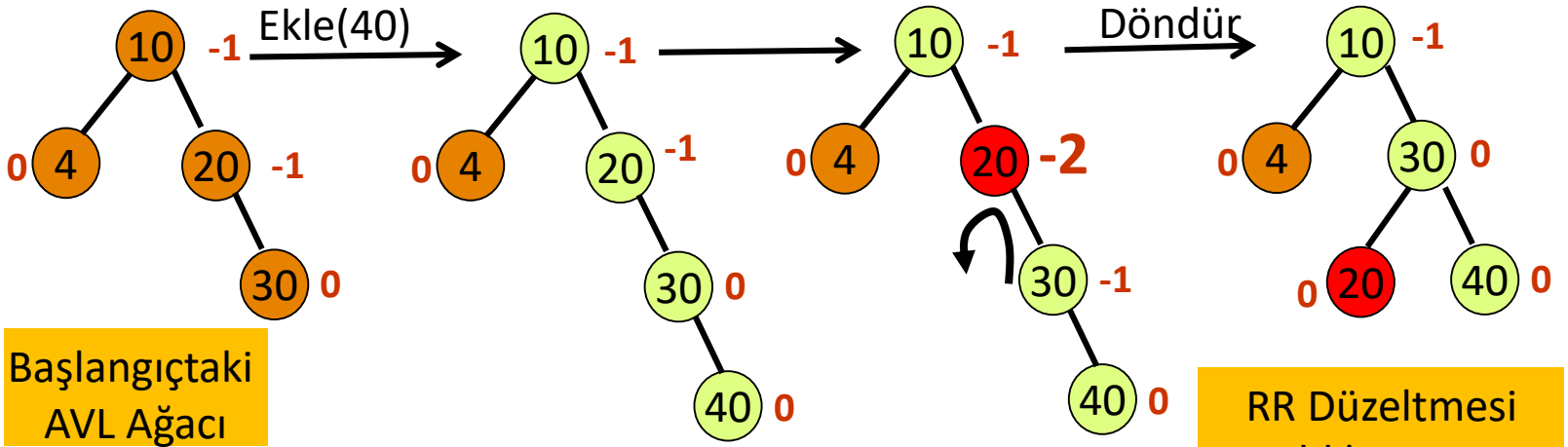
- LL Dengesizliği:
  - P(4)'ün bfsi 2
  - L(3)'ün bfsi 0 veya 1

RR Dengesizliği & Düzeltme



- RR Dengesizliği: P'nin sağ alt ağacının sağına eklendiğinde (D alt ağacına eklendiğinde)
  - P → bf = -2
  - R → bf = 0 veya -1
- Düzeltme: P etrafında sola doğru tek dönderme

RR Dengesizliği Düzeltme Örneği (1)



Başlangıçtaki AVL Ağacı

40 eklendikten sonra ağacın durumu

Pivotun belirlenmesi

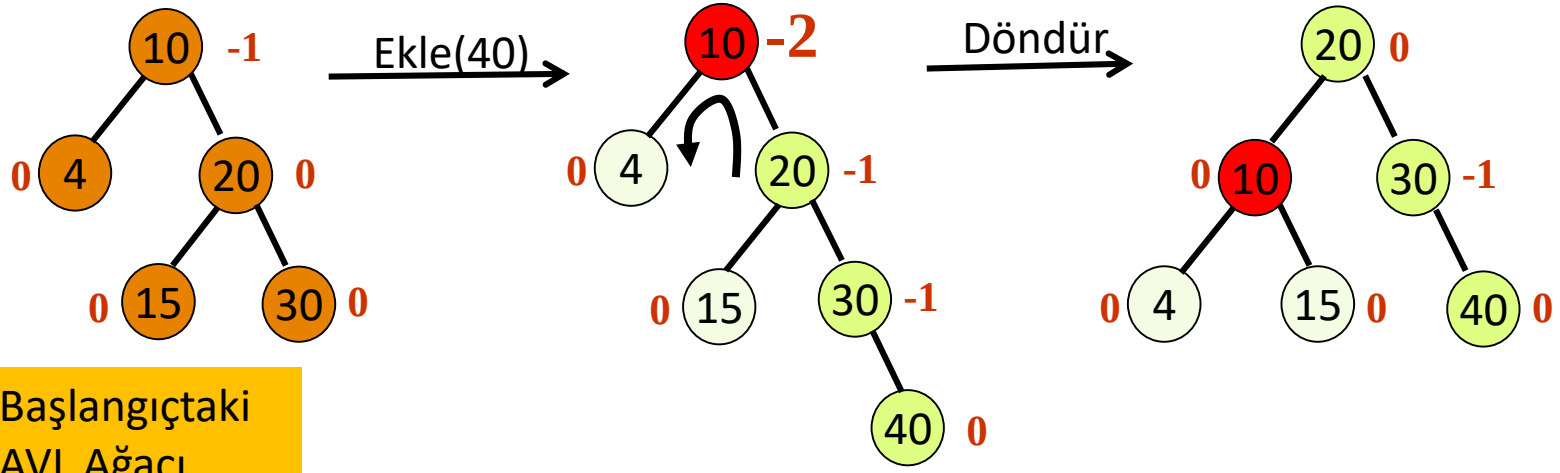
RR Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

- RR Dengesizliği:
  - P(20) → bf = -2
  - R(30) → bf = 0 veya -1

RR Dengesizliği Düzeltme Örneği (2)



Başlangıçtaki AVL Ağacı

40 eklendikten sonra ağacın durumu

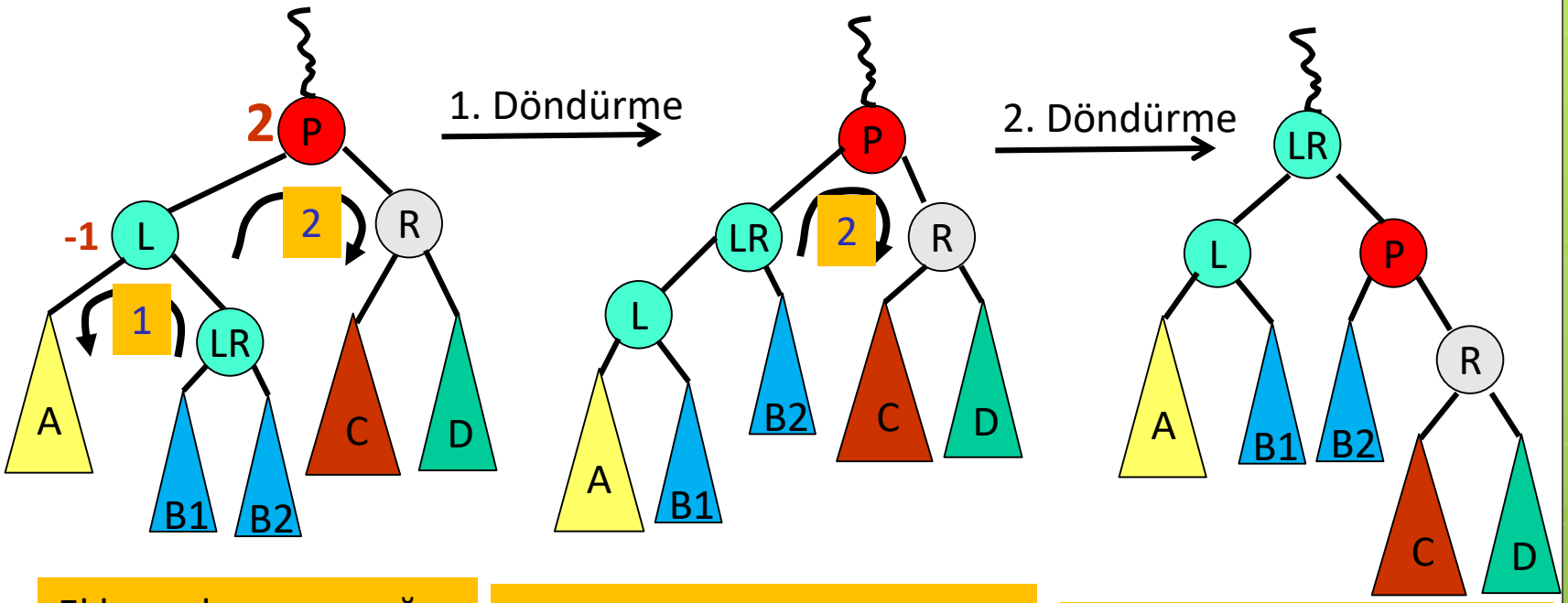
RR Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle ve 10'u pivot olarak belirle

Dengesizliğin türünün belirlenmesi

- RR Dengesizliği:
  - P(10) → bf = -2
  - R(20) → bf = 0 veya -1

LR Dengesizliği & Düzeltme



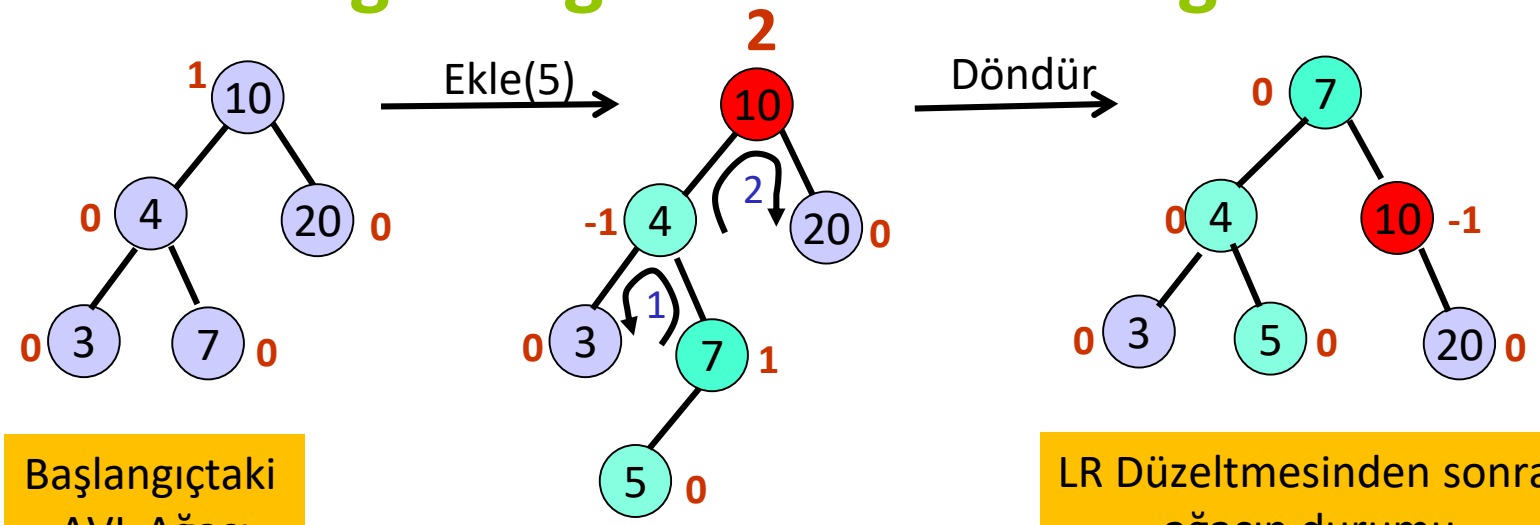
Eklemeden sonra ağaç

1. Döndürmeden sonra ağaç

LR Düzeltmesinden sonra

- LR Dengesizliği: P'nin sol alt ağacının sağına eklendiğinde (LR ağacına)
  - P → bf = 2
  - L → bf = -1
- Düzeltme: L & P etrafında 2 kez döndürme

LR Dengesizliği Düzeltme Örneği



Başlangıçtaki AVL Ağacı

5 eklendikten sonra ağacın durumu

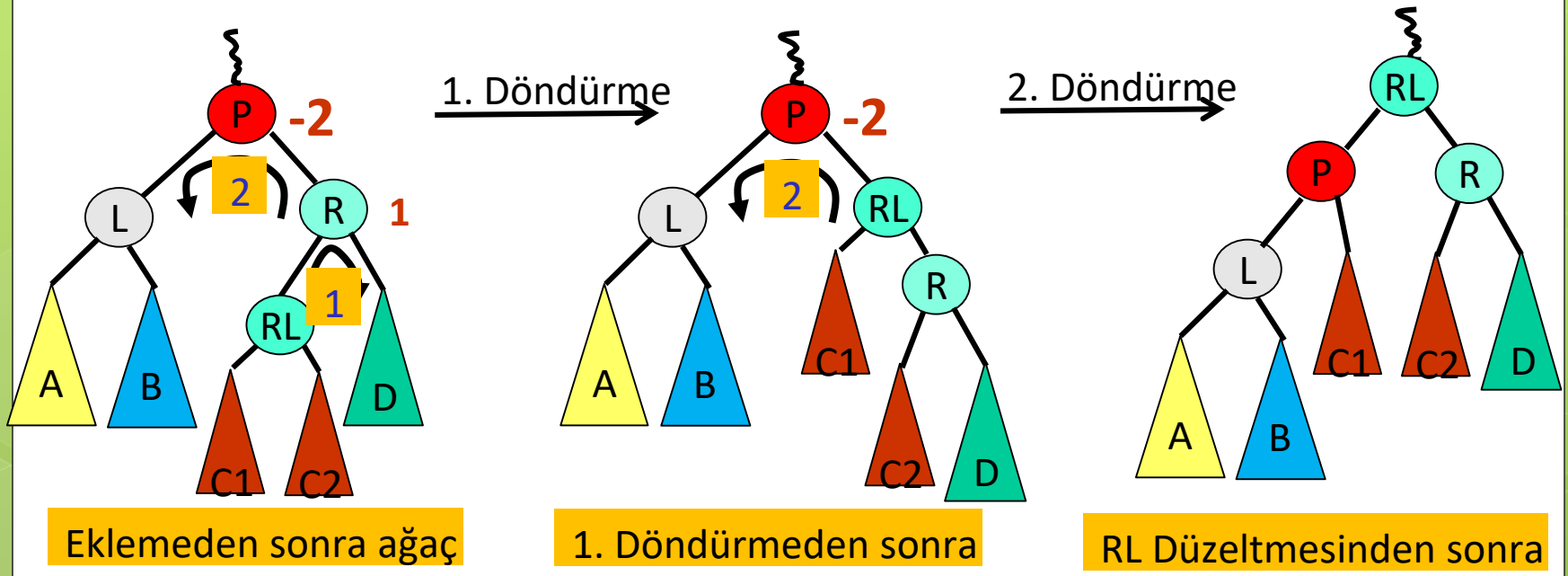
LR Düzeltmesinden sonra ağacın durumu

Balans faktörü düzelterek köke doğru ilerle ve 10'u pivot olarak belirle

Dengesizliğin türünün belirlenmesi

- LR Dengesizliği:
  - P(10) → bf = 2
  - L(4) → bf = -1

RL Dengesizliği & Düzeltme



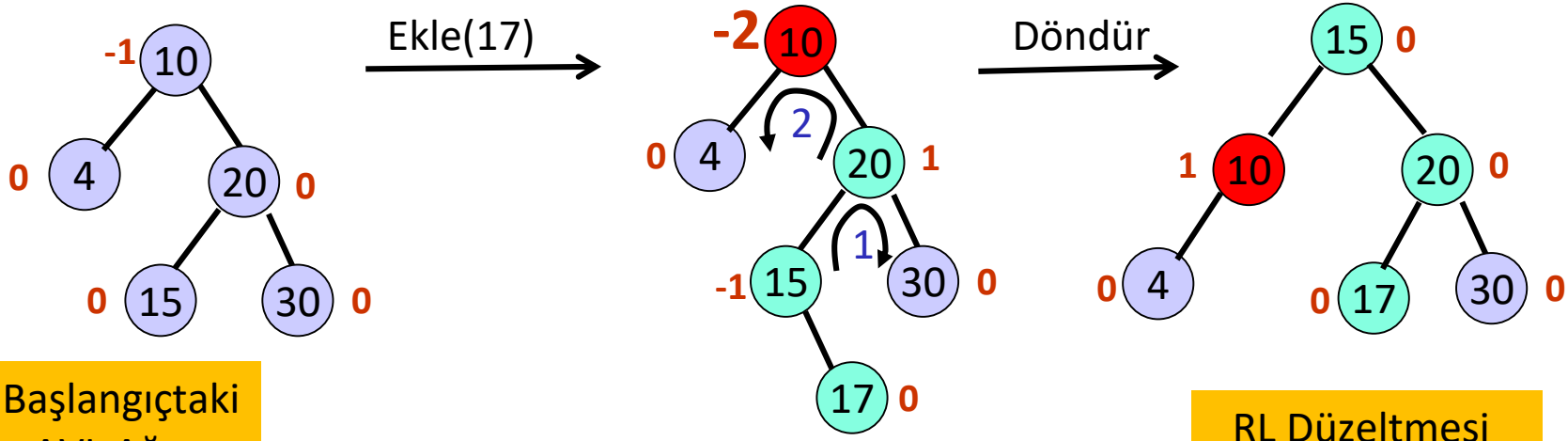
Eklemeden sonra ağaç

1. Döndürmeden sonra

RL Düzeltmesinden sonra

- RL Dengesizliği: P'nin sağ alt ağacının soluna eklendiğinde (RL alt ağacına)
  - P → bf = -2
  - R → bf = 1
- Düzeltme: R & P etrafında 2 kez döndürme.

RL Dengesizliği Düzeltme Örneği



Başlangıçtaki AVL Ağacı

17 eklendikten sonra ağacın durumu

RL Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle ve 10'u pivot olarak belirle

Dengesizliğin türünün belirlenmesi

- RL Dengesizliği:
  - P(10) → bf = -2
  - R(20) → bf = 1

AVL AĞAÇLARI

- **AVL Ağaçlarının C Kodları İle Yazılmış Bazı Fonksiyonları**

- **Fonksiyon 1.** AVL Ağacı İçim Düğüm Tanımı

```
public class AvlAgac {
public long data;
public AvlAgac left;
public AvlAgac right;
public int height=0;
}
```

AVL AĞAÇLARI

- **Fonksiyon 2.** Bir AVL Düğümünün Yüksekliğini Döndüren Fonksiyon

```
int Height (AvlAgac p)
{ if (p == null)
      return -1;
  else
  {
    return p.height;
  }
}
```

AVL AĞAÇLARI

- **Fonksiyon 3.** AVL Ağacı İçin Ekleme Fonksiyonu

```
public AvlAgac insert(AvlAgac p, int x) {
if (p == null)
{ p = new AvlAgac(); p.data = x; }
else
{ if (x <p.data)
  {
    p.left = insert(p.left, x);
    if (Height(p.left)-Height(p.right) == 2)
    if (x < p.left.data)
       p = singlerotateright(p);
    else  p = doublerotateright(p);
  }
```

AVL AĞAÇLARI

```
 else if (x > p.data)
{ p.right = insert(p.right, x);
 if (Height(p.left) - Height(p.right) == -2)
 if (x > p.right.data)
   p = singlerotateleft(p);
 else p = doublerotateleft(p);
 }
}
 p.height = max(Height(p.left), Height(p.right)) + 1;
 return p;
}
int max(int a, int b)
{ if (a > b) return a;
  else return b;
}
```

AVL AĞAÇLARI

- **Fonksiyon 4**. AVL Ağacı için sağa doğru tek döndürme (LL) Fonksiyonu

```
AvlAgac singlerotateright(AvlAgac p)
{
 AvlAgac lc = p.left;
 p.left = lc.right;
 lc.right = p;
 p.height = max(Height(p.left), Height(p.right)) + 1;
 lc.height = max(Height(lc.left), lc.height) + 1;
 return lc;
}
```

AVL AĞAÇLARI

- **Fonksiyon 5.** AVL Ağacı için LR çift döndürme

```
AvlAgac doublerotateright(AvlAgac p)
 {
 p.left = singlerotateleft(p.left);
 return singlerotateright(p);
 }
```

AVL AĞAÇLARI

- **Fonksiyon 6.** AVL Ağacı için sağa doğru tek döndürme (RR) Fonksiyonu

```
AvlAgac singlerotateleft(AvlAgac p)
{
 AvlAgac rc = p.right;
 p.right = rc.left;
 rc.left = p;
 p.height = max(Height(p.left), Height(p.right))+1;
 rc.height = max(Height(rc.right), rc.height)+ 1;
 return rc;
}
```

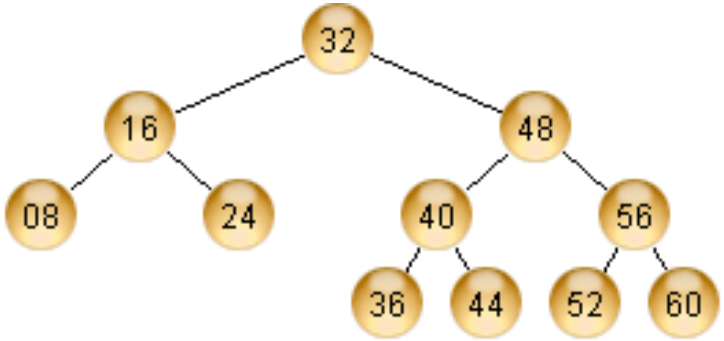
AVL AĞAÇLARI

- **Fonksiyon 7.** AVL Ağacı için RL çift döndürme

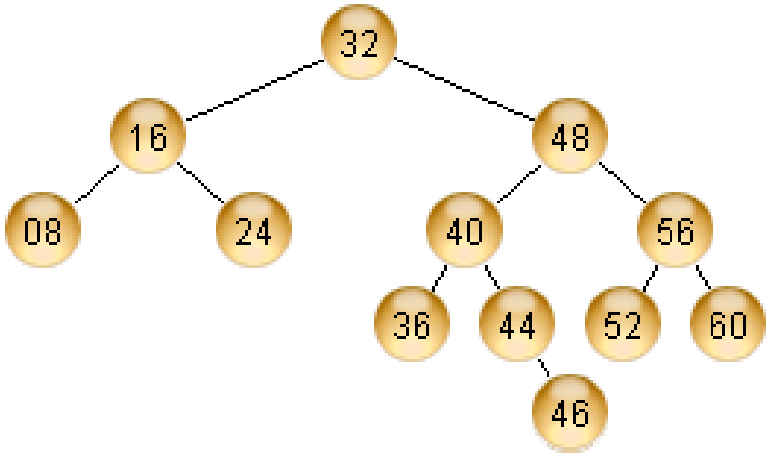
```
AvlAgacdoublerotateleft(AvlAgac p)
 {
 p.right = singlerotateright(p.right);
 return singlerotateleft(p);
 }
```

AVL AĞAÇLARI

- **Örnek:**

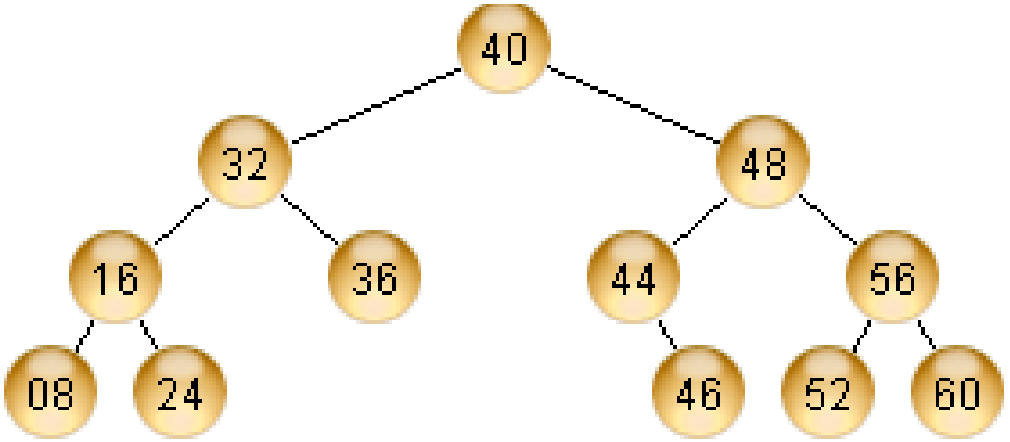


- **46, eklendi**



AVL AĞAÇLARI

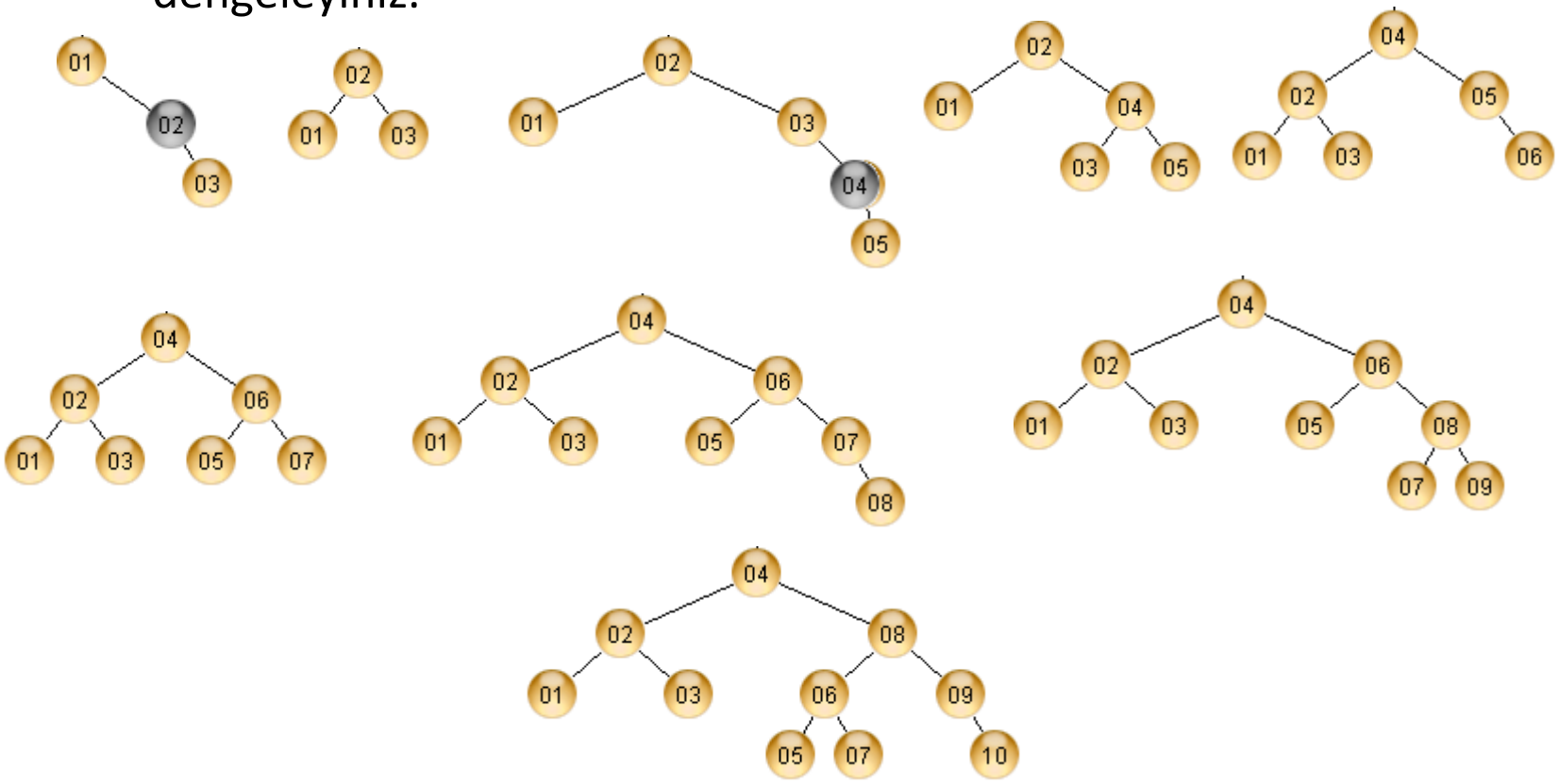
- **Örnek:**
-
-



- Dengeli AVL ağacı

AVL AĞAÇLARI

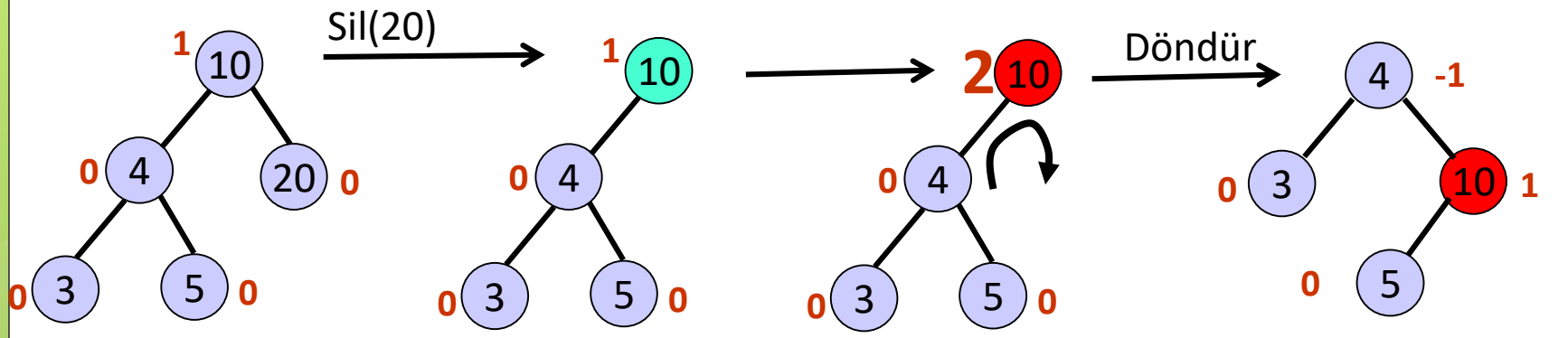
- Örnek: 1,2,3,4,5,6,7,8,9,10 ağacı oluşturup adım adım dengeleyiniz.



AVL AĞAÇLARI- Silme

- Silme işlemi ekleme işlemi ile benzerlik gösterir.
- AVL ağaçlarında silme işlemini yaparken özellikle çocukları olan düğümlerin silme işlemi farklı yapılarda gözükür. Amaç dengeyi bozmayacak düğümün öncelikle sağlanması.
- Soldaki en büyük çocuk veya Sağdaki en küçük çocuk yeni düğüm olur. Önemli olan silme olayından sonra dengeleme işleminin yeniden sağlanmasıdır. Bu işlem sırasında birden fazla döndürme işlemi yapılabilir.
- Kural 1: Silinen düğüm yaprak ise, problem yok dengeleme işlemi gerekiyorsa yap.
- Kural 2: Silinen düğüm çocukları olan bir düğüm ise ya soldaki en büyük çocuğu veya sağdaki en küçük çocuğu al. Bu işlem sırasında dengeleme için birden fazla döndürme işlemi yapılabilir.

Silme Örneği (1)



Başlangıçtaki AVL Ağacı

20 silindikten sonra Ağacın durumu

10'un pivot olarak belirlenmesi

LL Düzeltmesinden Sonra AVL Ağacı

Balans faktörü düzelterek köke doğru ilerle

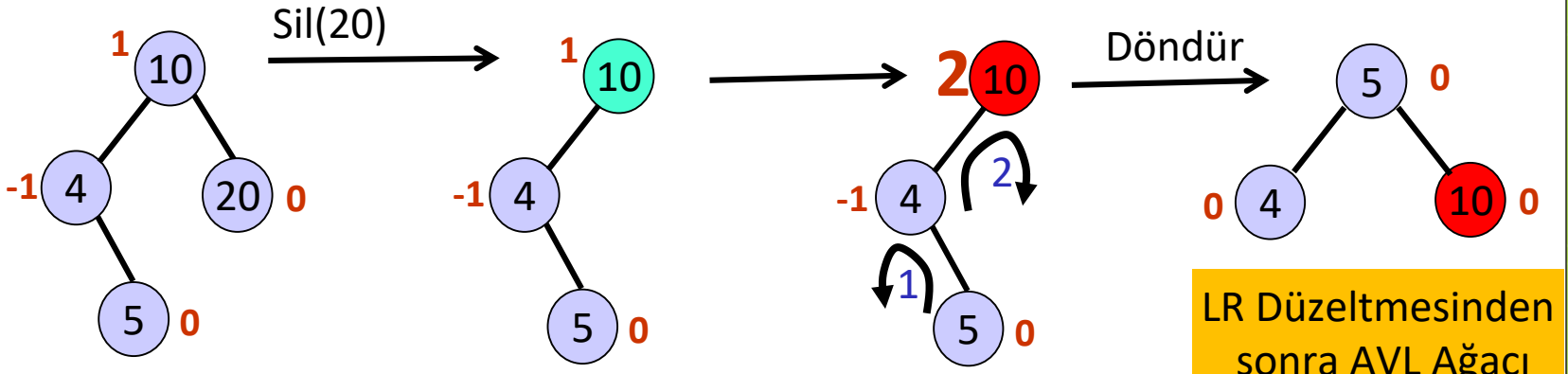
Balans faktörü=0 ise en az döndürme olacak şekilde seç.

Dengesizliğin türünün belirlenmesi

LL Dengesizliği:

- **P(10)**'ün **bf**si 2
- **L(4)**'ün **bf**si 0 veya 1

Silme Örneği (2)



Başlangıçtaki AVL Ağacı

20 silindikten sonra Ağacın durumu

10'un pivot olarak belirlenmesi

LR Düzeltmesinden sonra AVL Ağacı

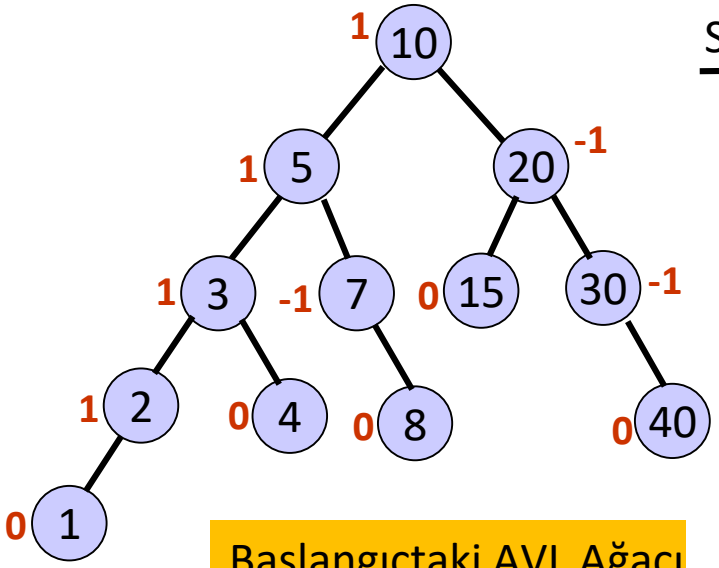
Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

LR Dengesizliği:

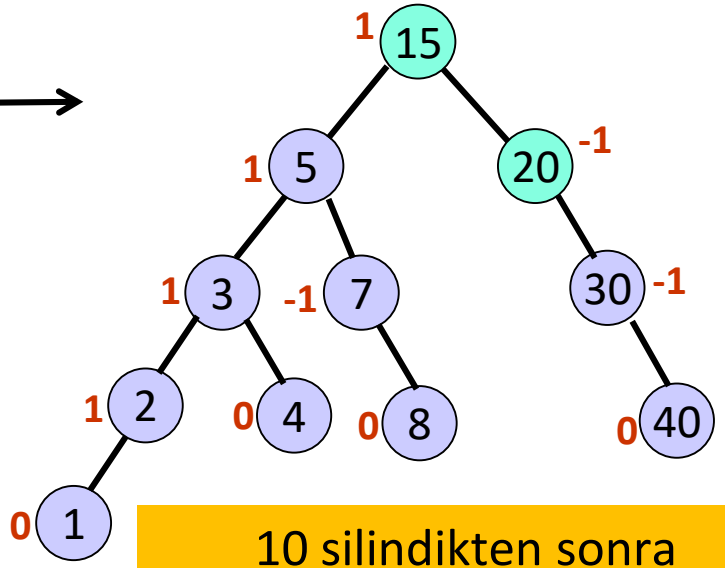
- P(10) → bf = 2
- L(4) → bf = -1

Silme Örneği (3)



Başlangıçtaki AVL Ağacı

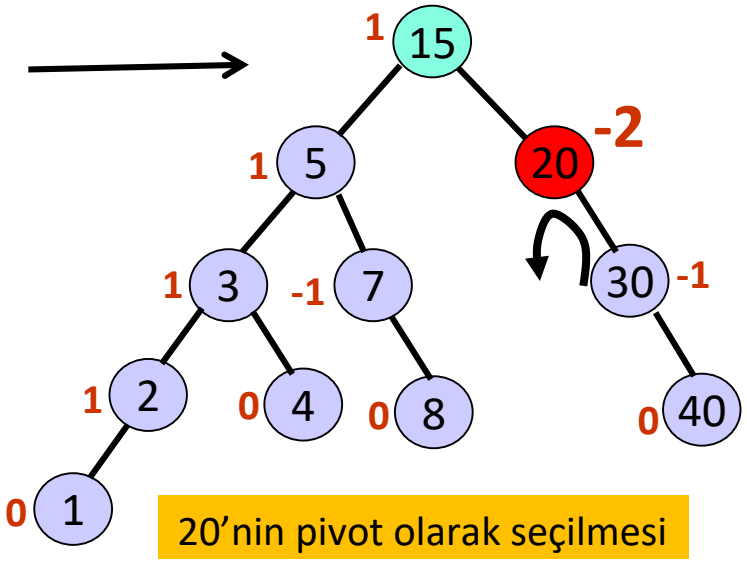
Sil(10)



10 silindikten sonra
10 ve 15(sağ alt ağaçtaki en küçük eleman) yer değiştirdi ve 15 silindi.

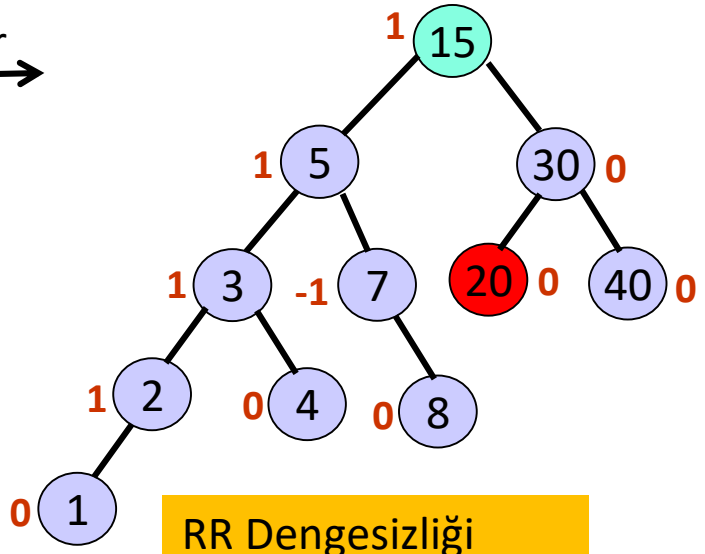
Balans faktörü düzelterek köke doğru ilerle

Silme Örneği (3)



20'nin pivot olarak seçilmesi

Döndür



RR Dengesizliği düzeltildikten sonra

Dengesizliğin türünü belirle

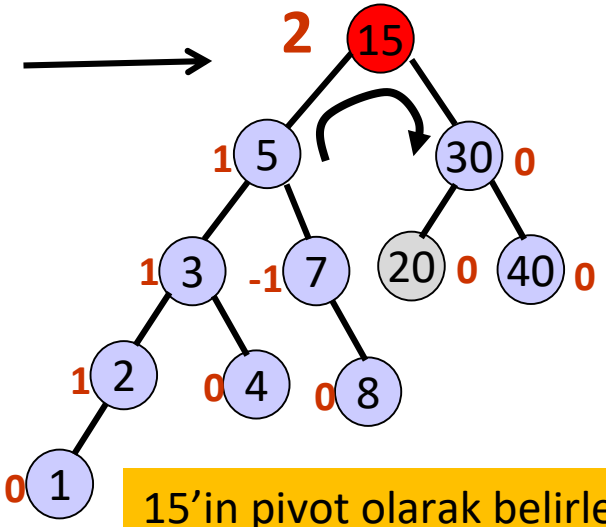
RR Dengesizliği:

- P(20) → bf = -2
- R(30) → bf = 0 veya -1

Yukardaki AVL ağacı mıdır?

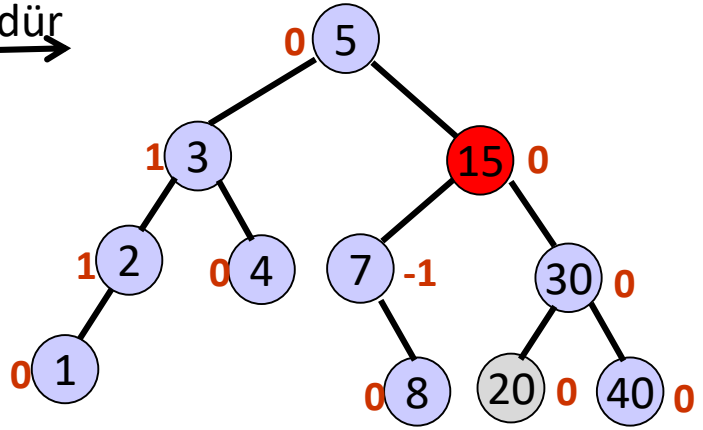
Balans faktörü düzelterek köke doğru ilerle.

Silme Örneği (3)



15'in pivot olarak belirlenmesi

Döndür



Düzeltilmiş AVL ağacı

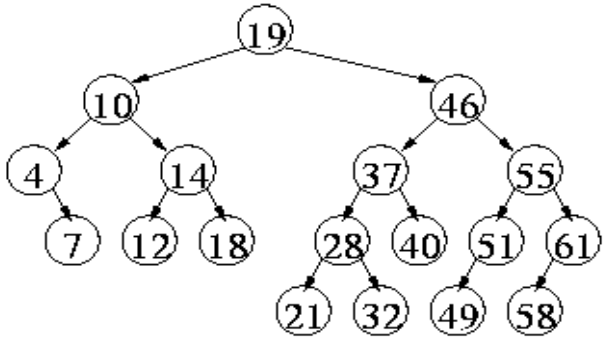
Dengesizliğin türünü belirle

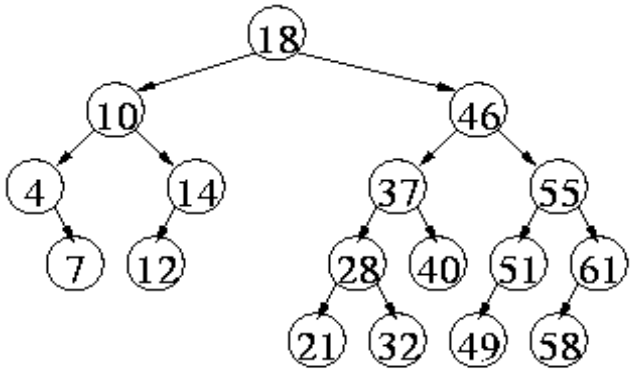
LL Dengesizliği:

- P(15) → bf = 2
- L(5) → bf = 0 veya 1

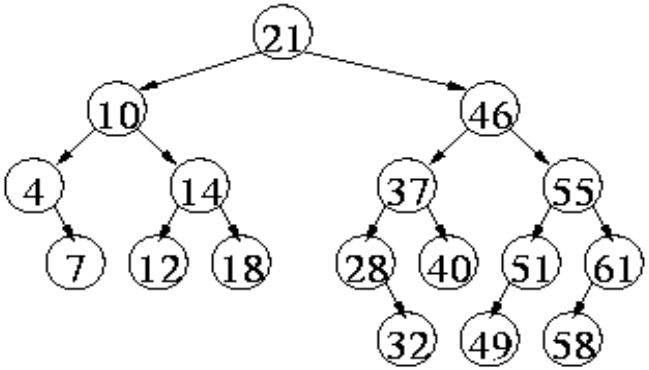
Silme Örneği– Kural 2

- **19 silindi (Soldaki en büyük düğüm veya sağdaki en küçük düğüm kök yapılır.**



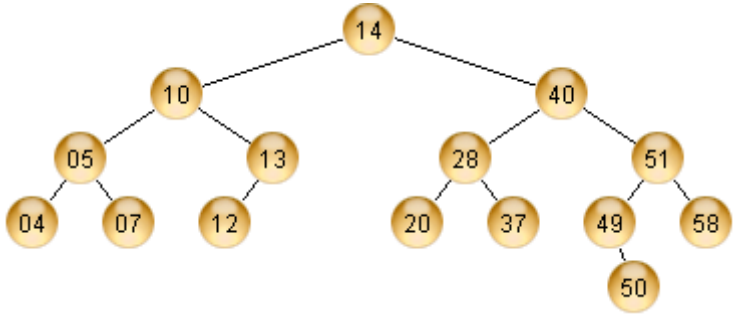


veya

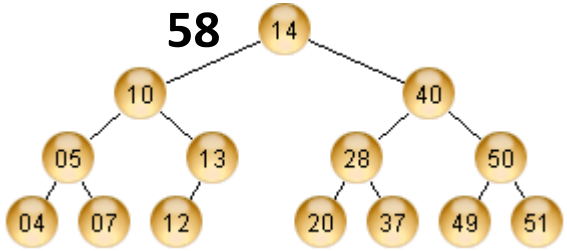


Silme Örneği– Kural 1-2

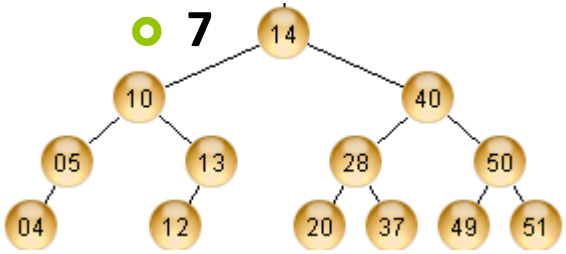
- **Örnek:** Sırasıyla siliniz; 58 ,7, 40,10,14 (Silme işleminde Soldaki en büyük düğüm alınacak)



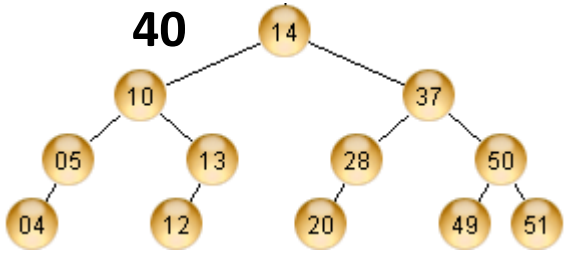
58



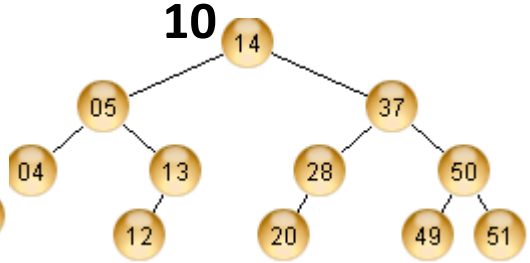
- **7**



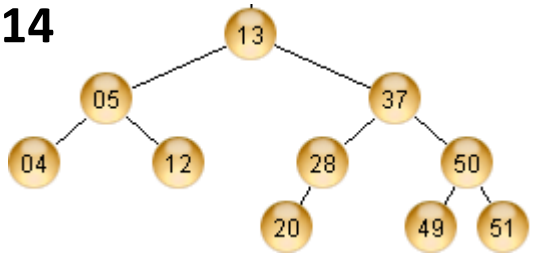
40



10

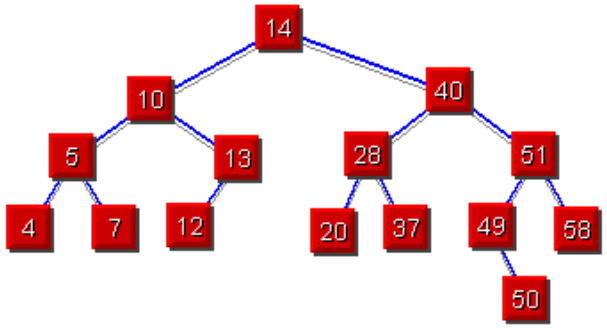


- **14**

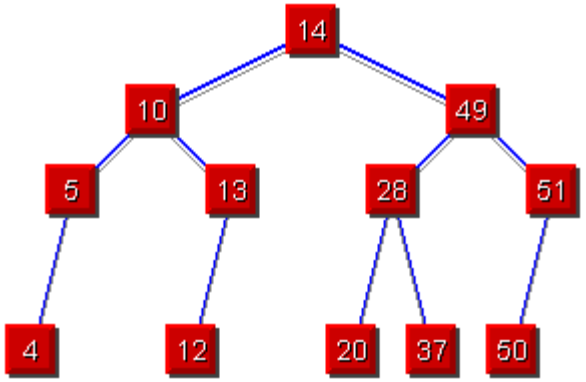


Silme Örneği– Kural 1-2

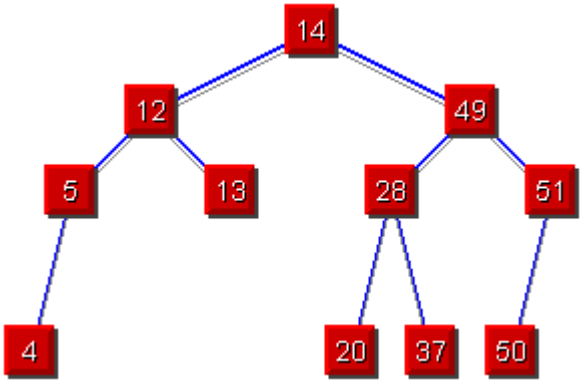
- **Örnek:** Sırasıyla siliniz; 40,58 , 7, 10,14 (Silme işleminde Sağdaki en küçük düğüm alınacak)



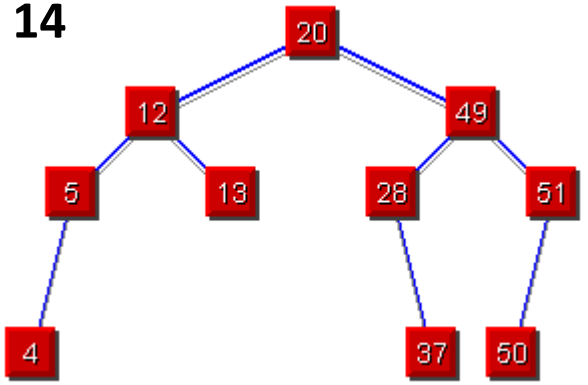
40,58,7



- **10**

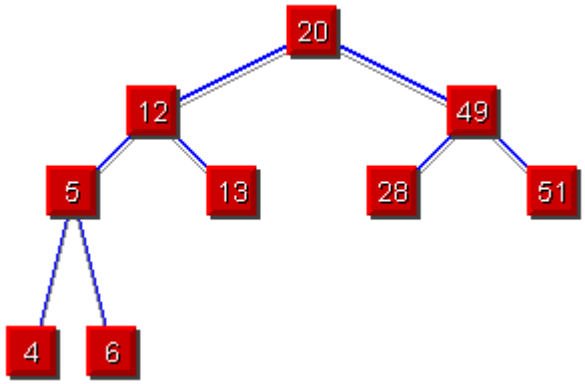


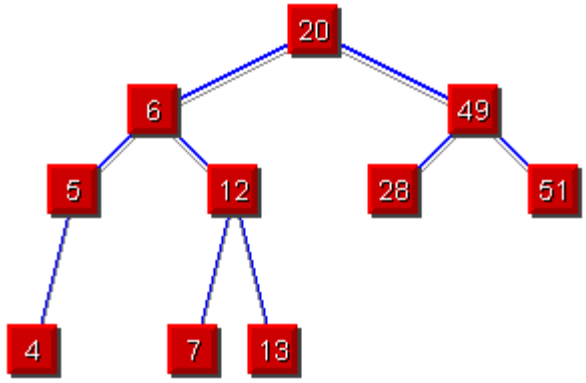
14



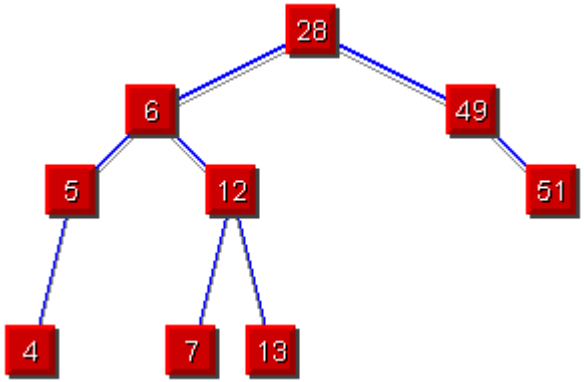
Silme Örneği

- **Örnek:** Ağaca 7 ekleyip dengeleyiniz.





- Ağaçtan 20 değerini siliniz (Sağdaki en küçük düğüm alınacak)



AVL -Arama (Bul)

- AVL ağacı bir tür İkili Arama Ağacı (BST) olduğundan arama algoritması BST ile aynıdır.
- O(logN) de çalışması garantidir.

AVL Ağaçları – Özet

- AVL Ağaçlarının Avantajları
  1. AVL ağacı devamlı dengeli olduğu için Arama/Ekleme/Silme işlemi O(logN) yapılır.
  2. Düzeltme işlemi algoritmaların karmaşıklığını etkilemez.

- AVL Ağaçlarının Dezavantajları:
  1. Balans faktörü için ekstra yer gereklidir.
  2. Yarı dengeli ağaçlarda AVL ağaçları kadar iyi performans verir ayrıca bu tür algoritmalarda balans faktör kullanılmaz
     - Splay ağaçları

Ödev

- Uygulama : S,E,L,İ,M, K,A,Ç,T,I AVL ağacını yapınız.
- Cevap: Kök düğümünün değeri L
- Yaprak düğümün değerleri: A,I,K,M,T
- **DSW metod (Recreate)**
- - Yükseklikleri ağacı yeniden oluşturarak ayarlar
- -Ağaçtan backbone elde eder ve ağacı yeniden oluşturur
- - Colin Day, Quentin F. Stout ve Bette L.Warren tarafından geliştirilmiştir
- **Self-Adjusting Trees (Priority)**
- - Kullanılma sıklığına göre ağacı sürekli düzenler
- - Her elemana erişimde root'a doğru yaklaştırır.
- Kırmızı olarak belirtilen kısımları en kısa zamanda programı ile birlikte word belgesi olarak hazırlanıp teslim edilecektir.

SPLAY TREE

- AVL tree
- 2-3 ve 2-3-4 tree
- Splay tree
- Red-Black Tree
- B- tree

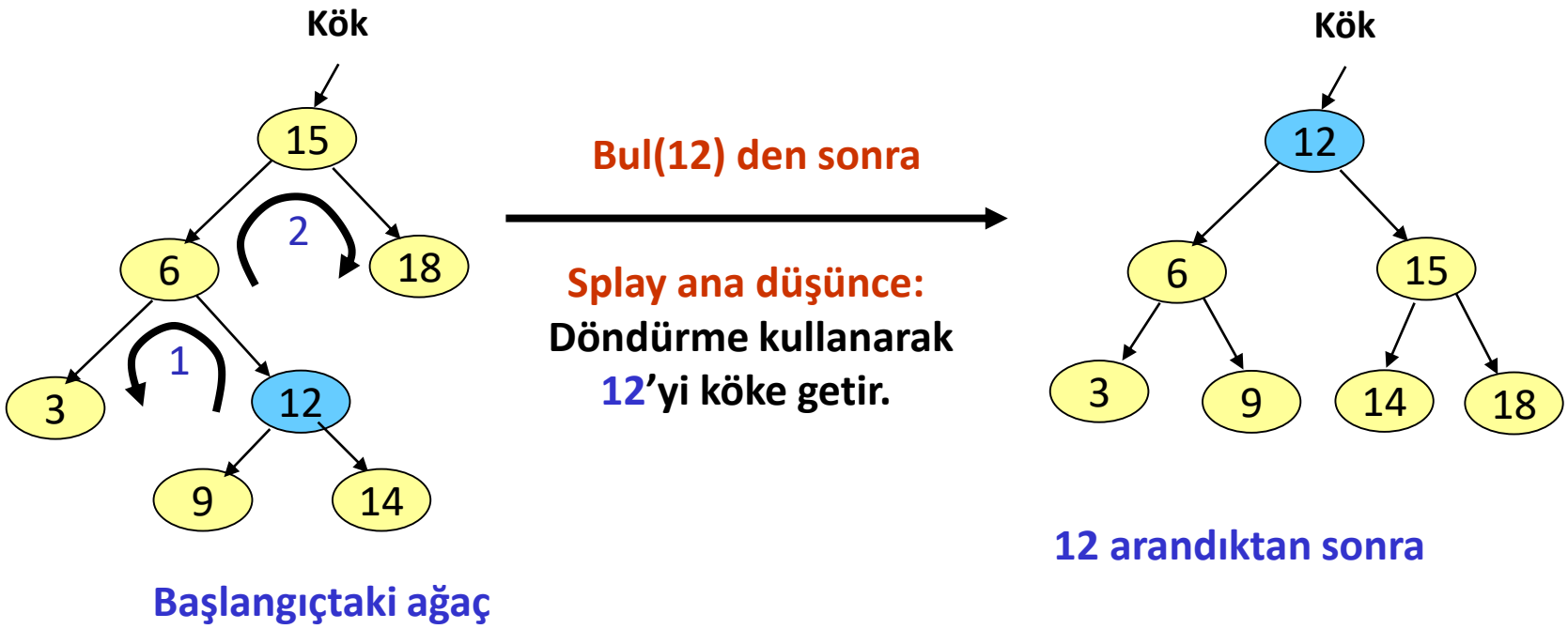
Splay Ağaçları

- Splay ağaçları ikili arama ağaç yapılarından birisidir.
  - Her zaman dengeli değildir.
  - Arama ve ekleme işlemlerinde dengelemeye çalışılır böylece gelecek operasyonlar daha hızlı çalışır.

- Sezgiseldir.
  - Eğer X bir kez erişildi ise, bir kez daha erişilecektir.
  - X erişildikten sonra "splaying" operasyonu yapılır.
    - X köke taşınır.

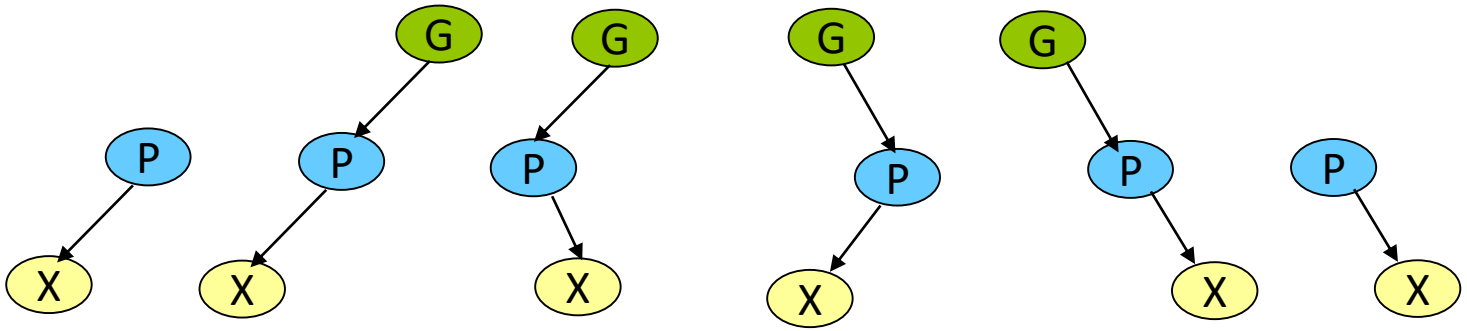
Örnek

- Burada ağacın dengeli olmasının yanı sıra 12 ye O(1) zamanında erişilebiliyor.
- Aktif (son erişilenler) düğümler köke doğru taşınırken aktif olmayan düğümler kökten uzaklaştırılır.



Splay Ağacı Terminolojisi

- X kök olmayan bir düğüm olsun. (ailesi olan)
- P düğümü X'in aile düğümü olsun.
- G düğümü X'in ata düğümü olsun.
- G ve X arasındaki yol düşünüldüğünde:
  - Her sağa gitme işlemine "zig" işlemi denilmektedir.
  - Her sola gitme işlemine "zag" işlemi denilmektedir.
- Toplam 6 farklı durum oluşabilir:

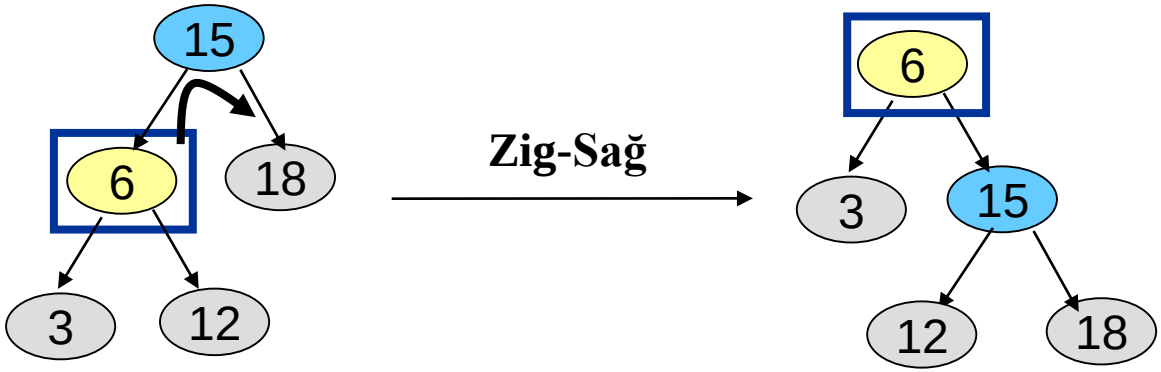


Splay Ağaç İşlemleri

- X'e erişildiğinde 6 tane rotasyon işleminden birisi uygulanır:
  - Tek Dönme (X, P'ye sahip ama G'ye sahip değil)
    - zig, zag
  - Çift Dönme (X hem P'ye hem de G'ye sahip)
    - zig-zig, zig-zag
    - zag-zig, zag-zag

Splay Ağaçları: Zig İşlemi

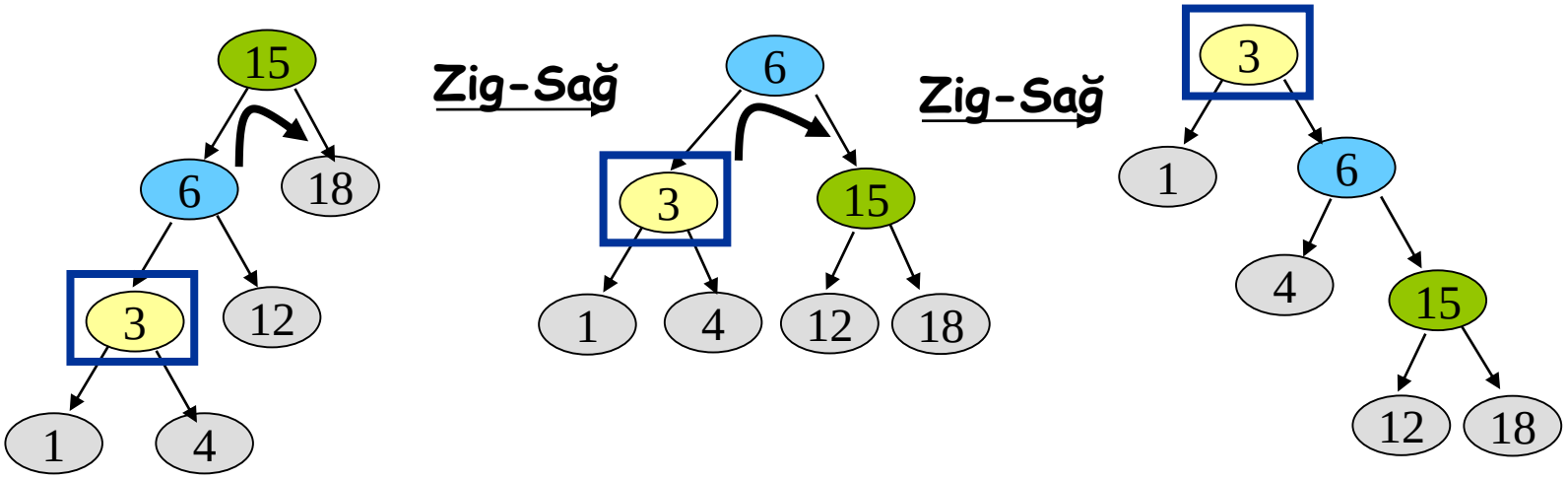
- "Zig" işlemi AVL ağacındaki gibi tek döndürme işlemidir.
- Örneğin erişilen elemanın 6 olduğu düşünülürse.



- "Zig-Sağ" işlemi 6'yı köke taşır.
- Bir sonraki işlemde 6'ya O(1) de erişilebilir.
- AVL ağacındaki sağ dönme işlemi ile benzerdir.

Splay Ağaçları: Zig-Zig İşlemi

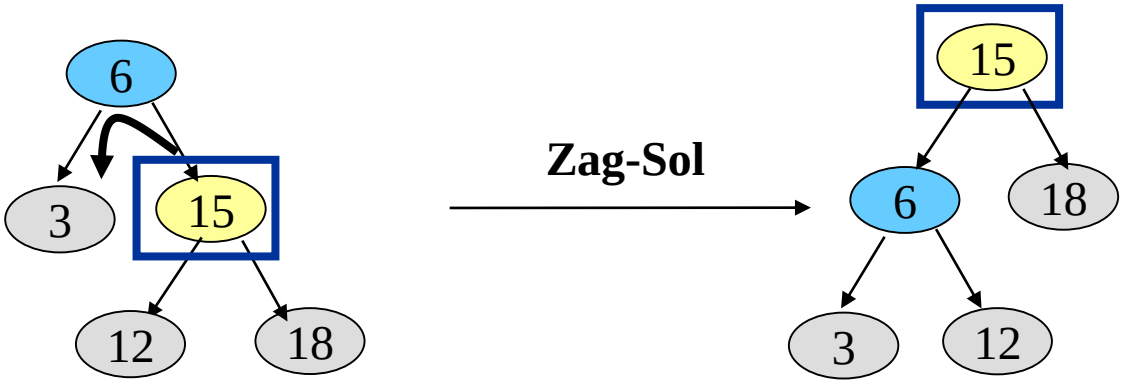
- "Zig-Zig" işlemi aynı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 3 olduğu düşünülürse.



- "zig-zig" işlemi ile birlikte 3 köke taşınmış oldu.
- Not: Aile - ata arasındaki döndürme önce yapılıyor.

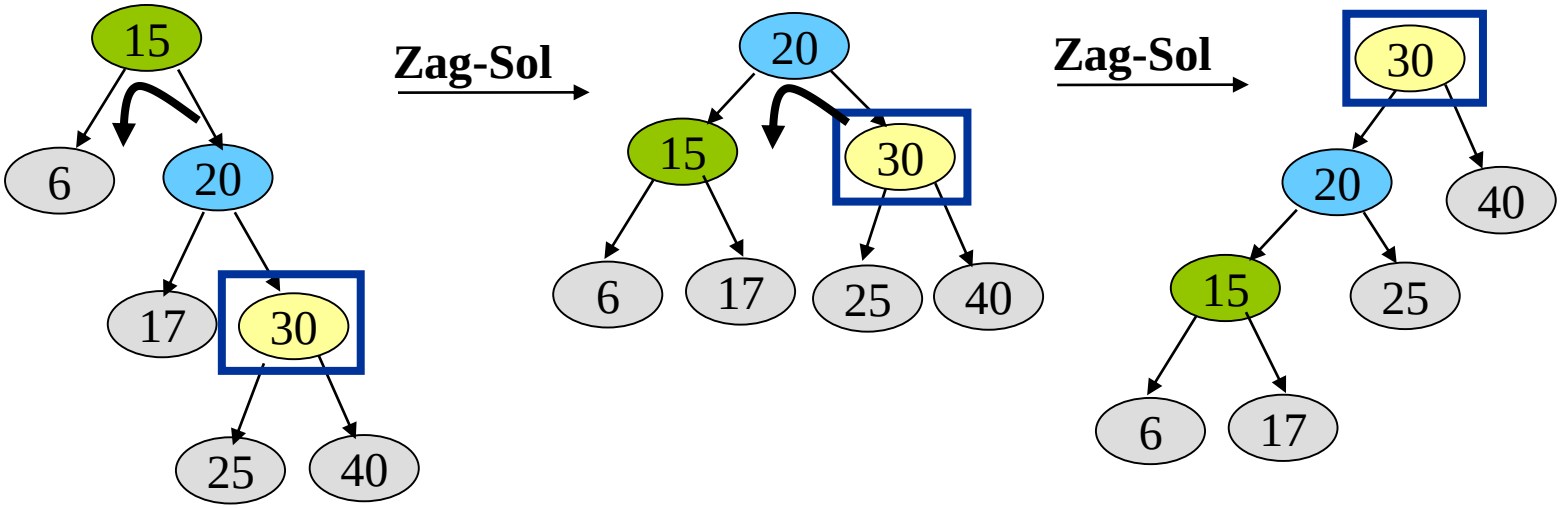
Splay Ağaçları: Zag İşlemi

- “Zag” işlemi AVL ağacındaki gibi tek döndürme işlemidir.
- Örneğin erişilen elemanın 15 olduğu düşünülürse.
- “Zag-sol işlemi 15’i köke taşır.
- Bir sonraki işlemde 15’e O(1) de erişilebilir.
- AVL ağacındaki sol dönme işlemi ile benzerdir.



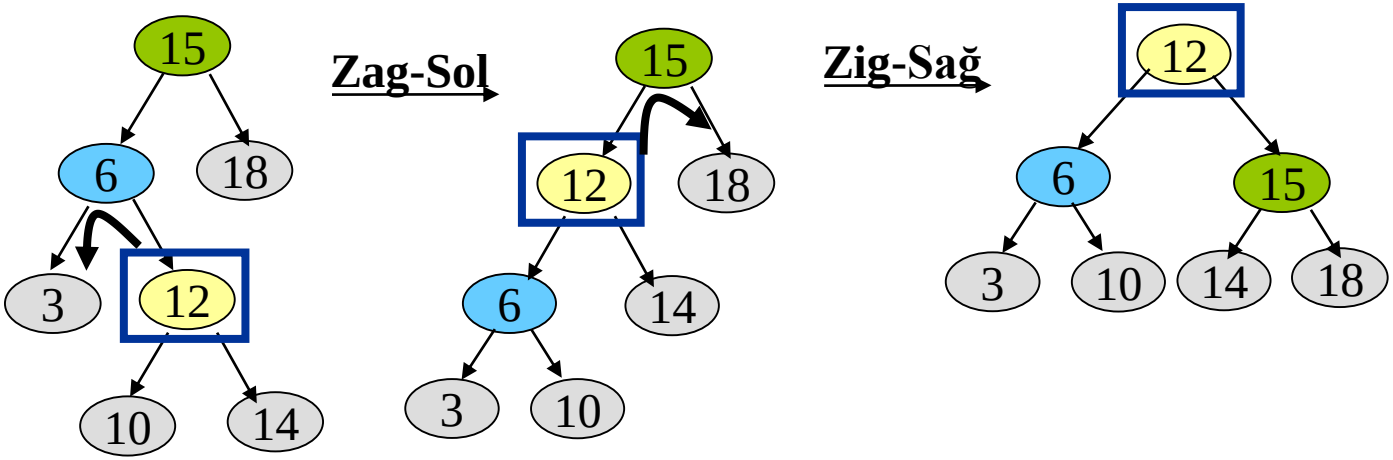
Splay Ağaçları: Zag-Zag İşlemi

- “Zag-Zag” işlemi aynı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 30 olduğu düşünülürse.
- “zag-zag” işlemi ile birlikte 30 köke taşınmış oldu.
- Not: Aile - ata arasındaki döndürme önce yapılıyor.



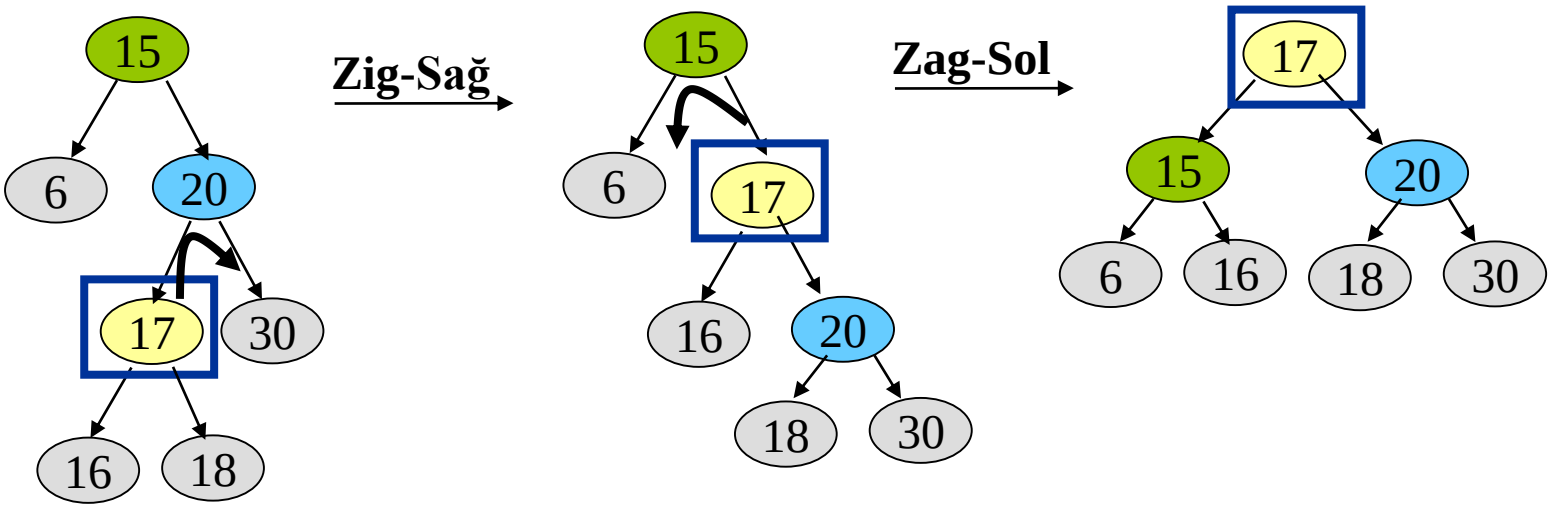
Splay Ağaçları: Zig-Zag durumu

- Zig-zag durumunda X, P nin sağ çocuğu ve G'de atası olduğu durumdur.
- “Zig-Zag” durumu farklı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 12 olduğu düşünülürse.
  - “zig-zag” işlemi ile birlikte 12 köke taşınmış oldu.
  - AVL ağacındaki LR dengesizliğini düzeltmek için kullanılan işlemler ile aynıdır.(önce sol dönme, daha sonra sağ dönme)

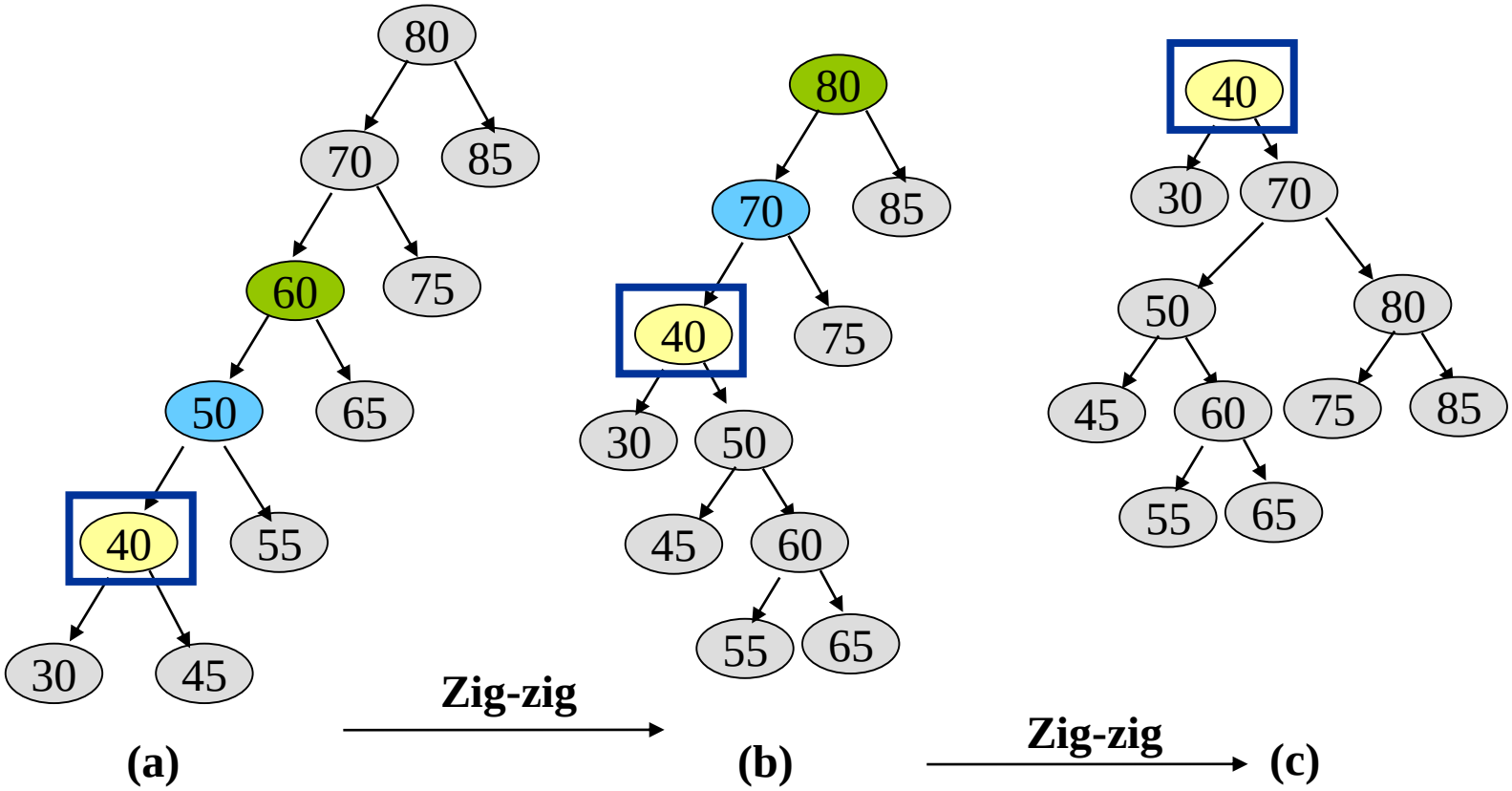


Splay Ağaçları: Zag-Zig durumu

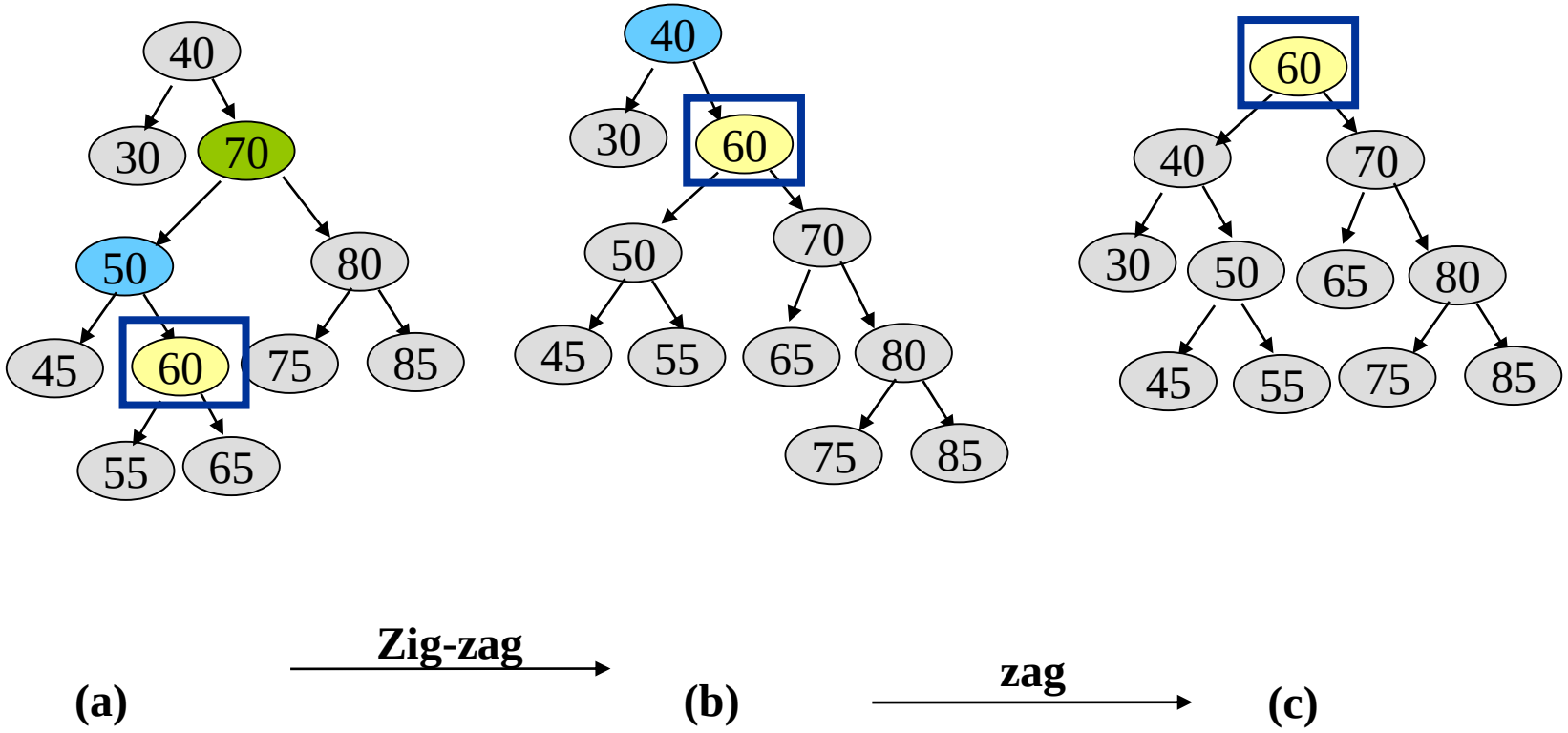
- Zag-Zig durumunda X, P nin sol çocuğu ve G'de atası olduğu durumdur.
- “Zag-Zig” durumu farklı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 17 olduğu düşünülürse.
- zag-zig” işlemi ile birlikte 17 köke taşınmış oldu.
- AVL ağacındaki RL dengesizliğini düzeltmek için kullanılan işlemler ile aynıdır.(önce sağ dönme, daha sonra sol dönme)



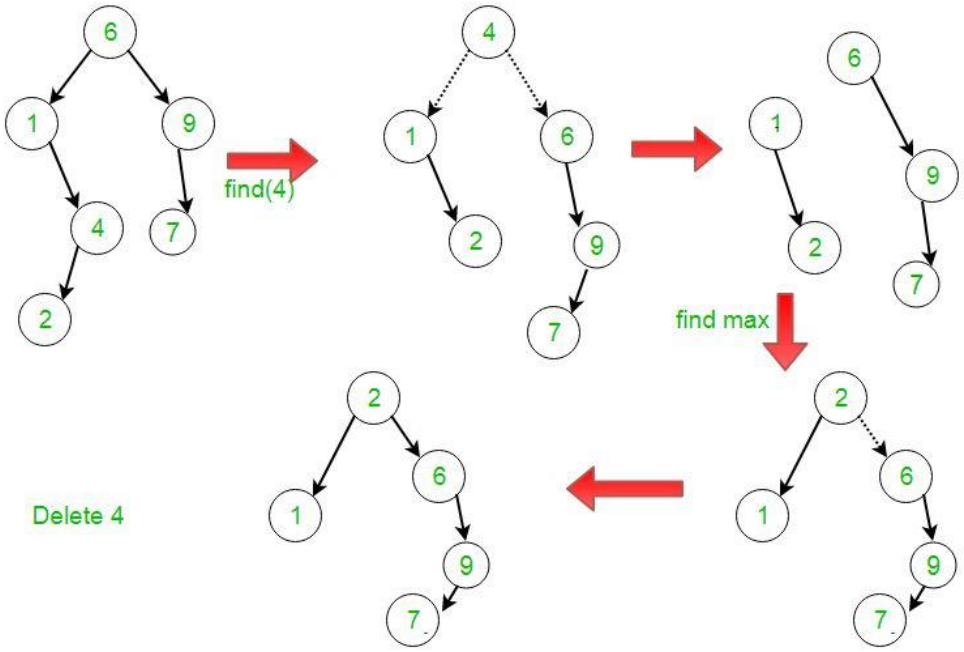
Splay Ağacı Örnek: 40'a Erişildiğinde



Splay Ağacı Örnek: 60'a Erişildiğinde



Splay Ağacı Örnek: 4’ silindiğinde



Diğer İşlemler Sırasında Splaying

- Splaying sadece Arama işleminden sonra değil Ekle/Sil gibi değer işlemlerden sonra da uygulanabilir.
- Ekle X: X yaprak düğüme eklendikten sonra (BST işlemi) X'i köke taşı.
- Sil X: X'i ara ve köke taşı. Kökte bulunan X'i sil ve sol alt ağaçtaki en büyük elemanı veya sağ kökteki en küçük elamanı (doğrudan taşıma olmayacak arama ile aynı) köke taşı.
- Bul X: Eğer X bulunamazsa aramanın sonlandığı yaprak düğümü köke taşı.

Splay Ağaçları – Özet

- Splaying işlemi ile ağaç genel olarak dengede kalıyor.
- Analiz Sonucu: N boyutlu bir Splay ağacı üzerinde k tane işlemin çalışma süresi O(k log N) dir. Dolayısıyla tek bir işlem için çalışma zamanı O(log N) dir.
- Erişilecek elemanların derinliği çok büyük olsa bile, arama işlemlerinin süresi bir süre sonra kısalacaktır. Çünkü her bir arama işlemi ağacın dengelenmesini sağlıyor.

Ödev

- A,V,L, A,Ğ,A,C,I,N,A,E,K,L,E,M,E elamanları sırası ile boş bir AVL ağacına ekleniyor.
- Her bir eleman eklendiğinde oluşan ağaçları çizerek gösteriniz.

Ödev

- S,P,L,A,Y,A,Ğ,A,C,I,N,A,E,K,L,E,M,E elemanları sırası ile boş bir Splay ağacına ekleniyor.
- Oluşan ağacı çizerek gösteriniz.
- Önce “S" ye sonra "A" ya erişimi sağlandıktan sonra ağacın yapısını çizerek gösteriniz.
- Aşağıda verilen splay ağacına ait C programını Java veya C# ile yapınız.

Örnek Program C++

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
struct node
{     int data;
      struct node *parent;
      struct node *left;
      struct node *right;
};
int data_print(struct node *x);
struct node *rightrotation(struct node *p,struct node *root);
struct node *leftrotation(struct node *p,struct node *root);
void splay (struct node *x, struct node *root);
struct node *insert(struct node *p,int value);
struct node *inorder(struct node *p);
struct node *delete(struct node *p,int value);
struct node *successor(struct node *x);
struct node *lookup(struct node *p,int value);
```

Örnek Program C++

```
void splay (struct node *x, struct node *root)
{       struct node *p,*g;
        /*check if node x is the root node*/
        if(x==root)             return;
        /*Performs Zig step*/
        else if(x->parent==root)
        {
                if(x==x->parent->left)          root=rightrotation(root,root);
                else                    root=leftrotation(root,root);
        }
        else
        {       p=x->parent; /*now points to parent of x*/
                g=p->parent; /*now points to parent of x's parent*/
                /*Performs the Zig-zig step when x is left and x's parent is left*/
                if(x==p->left&&p==g->left)
                {       root=rightrotation(g,root);
                        root=rightrotation(p,root);
                }
```

Örnek Program C++

```
                /*Performs the Zig-zig step when x is right and x's parent is right*/
                else if(x==p->right&&p==g->right)
                {       root=leftrotation(g,root);
                        root=leftrotation(p,root);
                }
                /*Performs the Zig-zag step when x's is right and x's parent is left*/
                else if(x==p->right&&p==g->left)
                {       root=leftrotation(p,root);
                        root=rightrotation(g,root);
                }
                /*Performs the Zig-zag step when x's is left and x's parent is right*/
                else if(x==p->left&&p==g->right)
                {       root=rightrotation(p,root);
                        root=leftrotation(g,root);
                }
                splay(x, root);
        }
    }
```

Örnek Program C++

```
struct node *rightrotation(struct node *p,struct node *root)
{     struct node *x;
      x = p->left;
      p->left = x->right;
      if (x->right!=NULL) x->right->parent = p;
      x->right = p;
      if (p->parent!=NULL)
            if(p==p->parent->right) p->parent->right=x;
            else
                  p->parent->left=x;
      x->parent = p->parent;
      p->parent = x;
      if (p==root)        return x;
      else            return root;
}
```

Örnek Program C++

```
struct node *leftrotation(struct node *p,struct node *root)
{       struct node *x;
        x = p->right;
        p->right = x->left;
        if (x->left!=NULL) x->left->parent = p;
        x->left = p;
        if (p->parent!=NULL)
                if (p==p->parent->left) p->parent->left=x;
                else
                        p->parent->right=x;
        x->parent = p->parent;
        p->parent = x;
        if(p==root)
                return x;
        else
                return root;
}
```

Örnek Program C++

```
struct node *insert(struct node *p,int value)
{     struct node *temp1,*temp2,*par,*x;
      if(p == NULL)
      {     p=(struct node *)malloc(sizeof(struct node));
            if(p != NULL)
            {     p->data = value;
                  p->parent = NULL;
                  p->left = NULL;
                  p->right = NULL;
            }
            else
            {     printf("No memory is allocated\n");
                  exit(0);
            }
            return(p);
      } //the case 2 says that we must splay newly inserted node to root
```

Örnek Program C++

```
else    {    temp2 = p;
             while(temp2 != NULL)
             {    temp1 = temp2;
                  if(temp2->data > value)        temp2 = temp2->left;
                  else if(temp2->data < value)   temp2 = temp2->right;
                  else
                       if(temp2->data == value)     return temp2;
             }
             if(temp1->data > value)
             {    par = temp1;//temp1 having the parent address,so that's it
                  temp1->left = (struct node *)malloc(sizeof(struct node));
                  temp1= temp1->left;
                  if(temp1 != NULL)
                  {    temp1->data = value;
                       temp1->parent = par;//store the parent address.
                       temp1->left = NULL;     temp1->right = NULL;                  }
                  else    {    printf("No memory is allocated\n");      exit(0);        }
             }
```

Örnek Program C++

```
                else
                {   par = temp1;//temp1 having the parent node address.
                    temp1->right = (struct node *)malloc(sizeof(struct node));
                    temp1 = temp1->right;
                    if(temp1 != NULL)
                    {   temp1->data = value;
                        temp1->parent = par;//store the parent address
                        temp1->left = NULL;   temp1->right = NULL;
                    }
                    else   {   printf("No memory is allocated\n");   exit(0);   }
                }
        }
        splay(temp1,p);//temp1 will be new root after splaying
        return (temp1);
    }
```

Örnek Program C++

```
struct node *inorder(struct node *p)
{
    if(p != NULL)
    {
        inorder(p->left);
        printf("CURRENT %d\t",p->data);
        printf("LEFT %d\t",data_print(p->left));
        printf("PARENT %d\t",data_print(p->parent));
        printf("RIGHT %d\t\n",data_print(p->right));
        inorder(p->right);
    }
}
```

Örnek Program C++

```
struct node *delete(struct node *p,int value)
{     struct node *x,*y,*p1;       struct node *root;       struct node *s;       root = p;
      x = lookup(p,value);
      if(x->data == value)
      {     //if the deleted element is leaf
            if((x->left == NULL) && (x->right == NULL))
            {     y = x->parent;
                  if(x ==(x->parent->right))     y->right = NULL;
                  else     y->left = NULL;          free(x);
            }
            //if deleted element having left child only
            else if((x->left != NULL) &&(x->right == NULL))
            {     if(x == (x->parent->left))
                  {  y = x->parent;    x->left->parent = y;    y->left = x->left;      free(x);         }
                  else   {  y = x->parent;  x->left->parent = y; y->right = x->left;   free(x);       }
            }
```

Örnek Program C++

```
//if deleted element having right child only
 else if((x->left == NULL) && (x->right != NULL))
 {   if(x == (x->parent->left)) { y = x->parent; x->right->parent = y;  y->left = x->right;  free(x); }
    else {  y = x->parent;  x->right->parent = y;  y->right = x->right;  free(x); }
    }
          //if the deleted element having two children
 else if((x->left != NULL) && (x->right != NULL))
          {   if(x == (x->parent->left))
             {  s = successor(x);
                if(s != x->right)
                   { y = s->parent;
                          if(s->right != NULL)
                          { s->right->parent = y; y->left = s->right; }
                          else y->left = NULL;
                          s->parent = x->parent;    x->right->parent = s;
                          x->left->parent = s;        s->right = x->right;
                          s->left = x->left;            x->parent->left = s;
                     }
```

Örnek Program C++

```
else    { y = s;   s->parent = x->parent;
          x->left->parent = s;   s->left = x->left;      x->parent->left = s;   }
          free(x);     }
               else if(x == (x->parent->right))
               {  s = successor(x);
                    if(s != x->right)
                    {    y = s->parent;
                         if(s->right != NULL) {s->right->parent = y; y->left = s->right;  }
                         else y->left = NULL;
                         s->parent = x->parent; x->right->parent = s;
                         x->left->parent = s; s->right = x->right;
                         s->left = x->left; x->parent->right = s;       }
                    else  { y = s; s->parent = x->parent; x->left->parent = s;
                      s->left = x->left; x->parent->right = s;                    }
                    free(x);
               }    }      splay(y,root);
     }        else      { splay(x,root); }
   }
```

Örnek Program C++

```
struct node *successor(struct node *x)
{   struct node *temp,*temp2;   temp=temp2=x->right;
      while(temp != NULL)  {   temp2 = temp;   temp = temp->left;  }
      return temp2; }
//p is a root element of the tree
struct node *lookup(struct node *p,int value)
{   struct node *temp1,*temp2;
      if(p != NULL)
      {  temp1 = p;
            while(temp1 != NULL)
            {  temp2 = temp1;
                  if(temp1->data > value)  temp1 = temp1->left;
                  else if(temp1->data < value) temp1 = temp1->right;
                  else return temp1;            }
            return temp2;
      }
      else { printf("NO element in the tree\n"); exit(0); }
}
```

Örnek Program C++

```
struct node *search(struct node *p,int value)
{      struct node *x,*root;        root = p;       x = lookup(p,value);
       if(x->data == value) {  printf("Inside search if\n"); splay(x,root); }
       else { printf("Inside search else\n"); splay(x,root); }
}
main()
{ struct node *root;//the root element
  struct node *x;//x is which element will come to root.
  int i; root = NULL; int choice = 0; int ele;
       while(1)
       { printf("\n\n 1.Insert"); printf("\n\n 2.Delete"); printf("\n\n 3.Search"); printf("\n\n
4.Display\n");
         printf("\n\n Enter your choice:");scanf("%d",&choice);
            if(choice==5)  exit(0);
            switch(choice)
            { case 1: printf("\n\n Enter the element to be inserted:");
                      scanf("%d",&ele); x = insert(root,ele);
                      if(root != NULL) { splay(x,root);}
                      root = x;   break;
```

Örnek Program C++

```
struct node *search(struct node *p,int value)
{      struct node *x,*root;        root = p;       x = lookup(p,value);
       if(x->data == value) {  printf("Inside search if\n"); splay(x,root); }
       else { printf("Inside search else\n"); splay(x,root); }
}
main()
{ struct node *root;//the root element
  struct node *x;//x is which element will come to root.
  int i; root = NULL; int choice = 0; int ele;
       while(1)
       { printf("\n\n 1.Insert"); printf("\n\n 2.Delete"); printf("\n\n 3.Search"); printf("\n\n
4.Display\n");
         printf("\n\n Enter your choice:");scanf("%d",&choice);
            if(choice==5)  exit(0);
            switch(choice)
            { case 1: printf("\n\n Enter the element to be inserted:");
                      scanf("%d",&ele); x = insert(root,ele);
                      if(root != NULL) { splay(x,root);}
                      root = x;   break;
```

Örnek Program C++

```
                case 2: if(root == NULL) { printf("\n Empty tree..."); continue; }
                    printf("\n\n Enter the element to be delete:");
                    scanf("%d",&ele); root = delete(root,ele); break;
                case 3: printf("Enter the element to be search\n");
                    scanf("%d",&ele); x = lookup(root,ele); splay(x,root);      root = x; break;
                case 4: printf("The elements are\n");    inorder(root); break;
                default: printf("Wrong choice\n"); break;
            }    }}
int data_print(struct node *x)
{ if ( x==NULL ) return 0;
    else return x->data;  }
/*some suggestion this code is not fully functional for example
if you have inserted some elements then try to delete root then it may not work
because we are calling right and left child of a null value(parent of root)
which is not allowed and will give segmentation fault
Also for inserting second element because of splaying twice(once in insert and one in main)
will give error So I have made those changes but mainly in my cpp( c plus plus file) file,
but I guess wiki will itself look into this and made  these changes */
```

Örnek SPLAY Tree Silme C++

```
// C implementation to delete a node from Splay Tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the given key and
    NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key   = key;
    node->left  = node->right  = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}
```

Örnek SPLAY Tree Silme C++

```
// A utility function to left rotate subtree rooted with x // See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}
// This function brings the key at root if key is present in tree. // If key is not present, then it brings the
last accessed item at  // root.  This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;
    // Key lies in left subtree
    if (root->key > key)
    {
// Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)         {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is       // done after else
            root = rightRotate(root);
        }
```

Örnek SPLAY Tree Silme C++

```
else if (root->left->key < key) // Zig-Zag (Left Right)
        {    // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);
               // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
        // Do second rotation for root
        return (root->left == NULL)? root: rightRotate(root);
    }
    else // Key lies in right subtree
    {   // Key is not in tree, we are done
        if (root->right == NULL) return root;

        // Zag-Zig (Right Left)
        if (root->right->key > key)
        {   // Bring the key as root of right-left
            root->right->left = splay(root->right->left, key);
            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)// Zag-Zag (Right Right)
        {   // Bring the key as root of right-right and do
            // first rotation
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }
        // Do second rotation for root
        return (root->right == NULL)? root: leftRotate(root);
    }
}
```

Örnek SPLAY Tree Silme C++

```
// The delete function for Splay tree. Note that this function
// returns the new root of Splay Tree after removing the key
struct node* delete_key(struct node *root, int key)
{
    struct node *temp;
    if (!root)
        return NULL;

    // Splay the given key
    root = splay(root, key);

    // If key is not present, then
    // return root
    if (key != root->key)
        return root;

    // If key is present
    // If left child of root does not exist
    // make root->right as root
    if (!root->left)
    {
        temp = root;
        root = root->right;
    }
```

Örnek SPLAY Tree Silme C++

```
// Else if left child exits
    else
    {
        temp = root;

        /*Note: Since key == root->key,
        so after Splay(key, root->lchild),
        the tree we get will have no right child tree
        and maximum node in left subtree will get splayed*/
        // New root
        root = splay(root->left, key);

        // Make right child of previous root  as
        // new root's right child
        root->right = temp->right;
    }

    // free the previous root node, that is,
    // the node containing the key
    free(temp);

    // return root of the new Splay Tree
    return root;

}
```

Örnek SPLAY Tree Silme C++

```
// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
/* Driver program to test above function*/
int main()
{    // Splay Tree Formation
    struct node *root = newNode(6);
    root->left = newNode(1);
    root->right = newNode(9);
    root->left->right = newNode(4);
    root->left->right->left = newNode(2);
    root->right->left = newNode(7);

     int key = 4;

    root = delete_key(root, key);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}
```

Max - Min Heap Tree

(Max ve Min Yığıt Ağaçları)

Max - Min Heap

- Öncelikli kuyruk konusunu hatırlayın. Kuyruğa sonradan eklenmesine rağmen öncelik seviyesine göre önce çıkabiliyordu.
- Öncelik kuyruğu oluşturmada farklı veri yapıları benimsenebilir. Yığınlar bunlardan sadece biridir.
- Tam ikili ağaç, yığıt kurmak amacıyla kullanılabilir. Yığınlar ise bir dizide gerçekleştirilebilir.
- Yığını dizide tam ikili ağaç yapısını kullanarak gerçekleştirdiğimizde yığın, dizinin 1. elemanından başlar, 0. indis kullanılmaz. Dizi sıralı değildir, yalnız 1. indisteki elemanın en öncelikli (ilk alınacak) eleman olduğu garanti edilir.

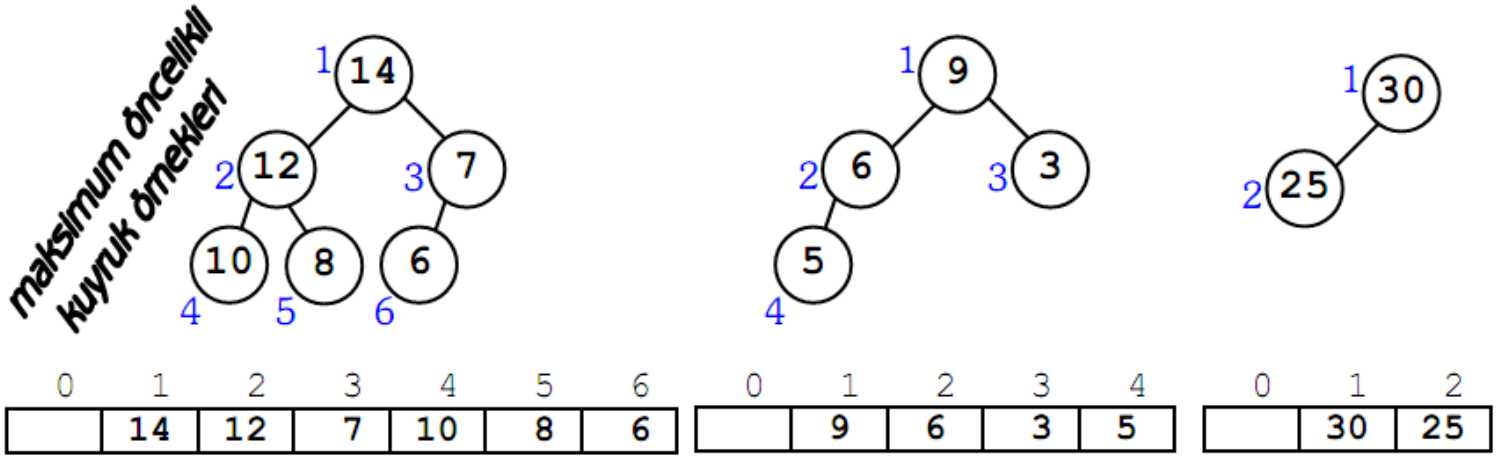
Max - Min Heap

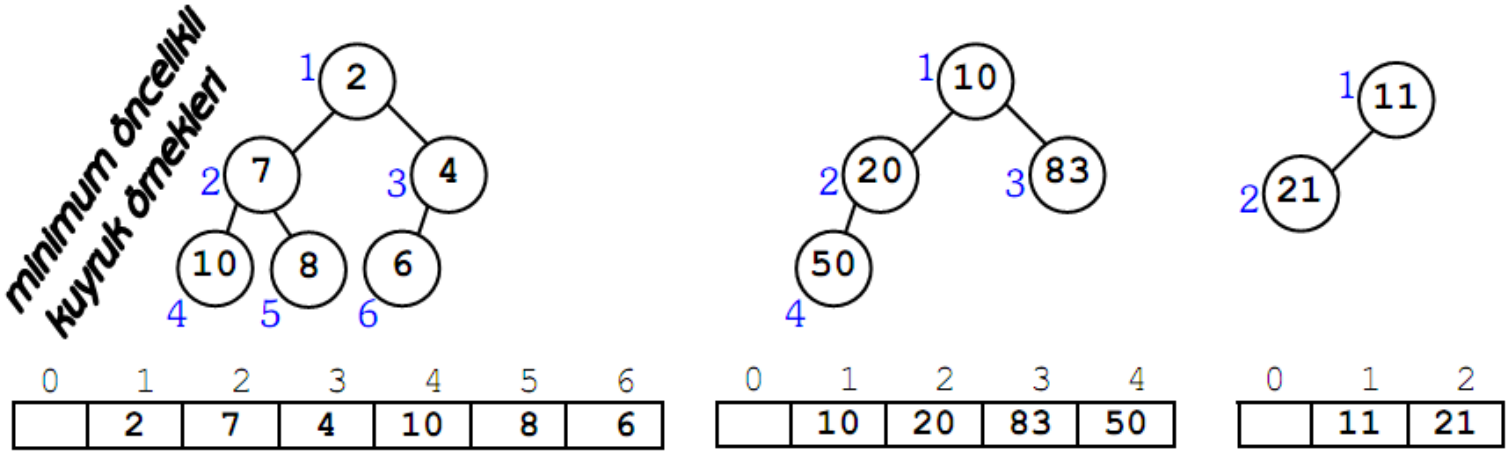
- Yığın denince aklımıza complete binary tree gelecek, search tree değil. Hatırlayalım; tam ikili ağacın tüm düzeyleri dolu, son düzeyi ise soldan sağa doğru doludur. İkili arama ağacı ile yığın arasında ise bir bağlantı yoktur.

- **Tanım**:
- Tam ikili ağaçtaki her düğümün değeri çocuklarından küçük değilse söz konusu ikili ağaç **maksimum yığın (max heap)** olarak isimlendirilir.

- Tam ikili ağaçtaki her düğümün değeri, çocuklarından büyük değilse söz konusu ikili ağaç **minimum yığın (min heap)** olarak isimlendirilir.

Max - Min Heap

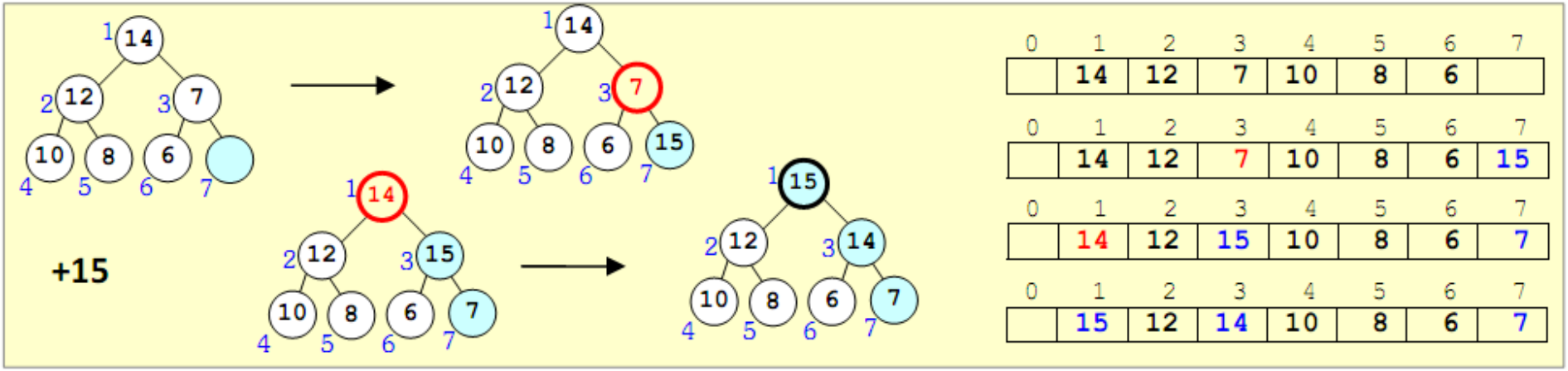

maksimum öncelikli kuyruk örnekleri
1 14
2 12
3 7
4 10
5 8
6 6
0 1 2 3 4 5 6
14 12 7 10 8 6
1 9
2 6
3 3
4 5
0 1 2 3 4
9 6 3 5
1 30
2 25
0 1 2
30 25


minimum öncelikli kuyruk örnekleri
1 2
2 7
3 4
4 10
5 8
6 6
0 1 2 3 4 5 6
2 7 4 10 8 6
1 10
2 20
3 83
4 50
0 1 2 3 4
10 20 83 50
1 11
2 21
0 1 2
11 21

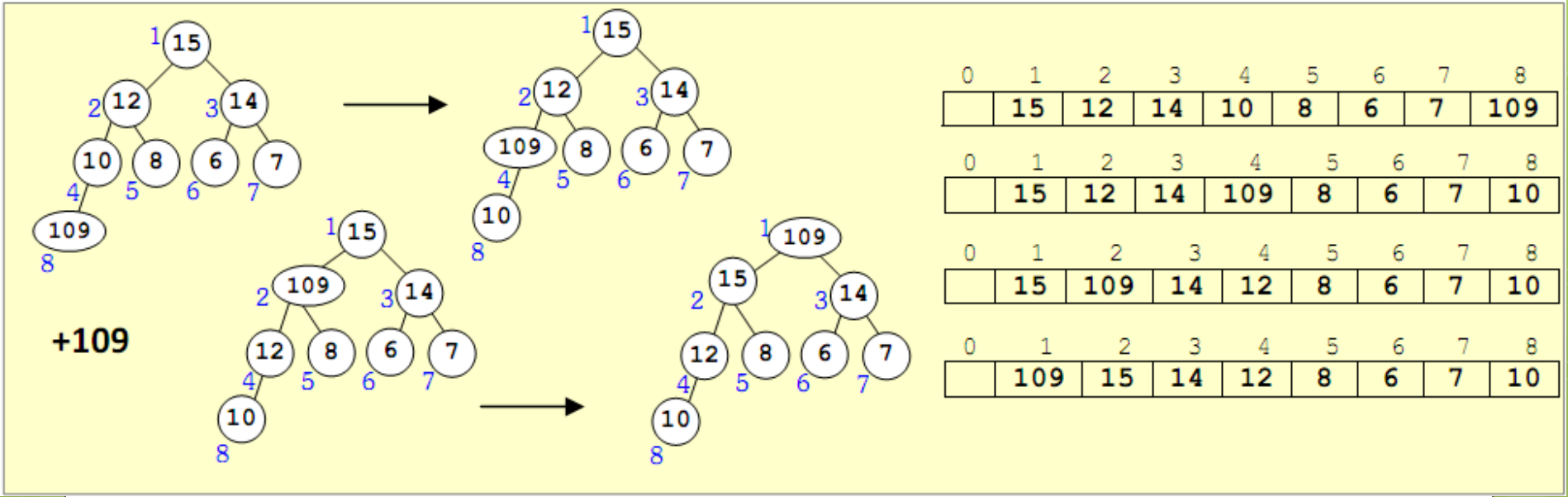
Maksimum Öncelikli Kuyruğa Öğe Ekleme

- A[1..n]
- Sırasıyla 15, 109, 107, 3, 15 değerleri eklenecektir. En iyi durumda Ω(1)'de de ekleme yapılır (3'ün eklemesi).
- Root at A[1]
- Parent of A[i] = A[⌊i/2⌋]
- Left child of A[i] = A[2i]
- Right child of A[i] = A[2i + 1]
- Mevcut tüm elemanlardan daha büyük bir öğe eklendiğinde ise yeni öğenin köke kadar kaydırılması gerekeceği için O(lgn)'de ekleme yapıldığını görüyoruz (109'un eklenmesi). Yani karmaşıklığın üst sınırı **lgn**, alt sınırı ise **1** olur. Gerçekte karmaşıklık bu ikisi arasında değişebilir. Bu durumda zaman karmaşıklığı **O(lgn)** 'dir.

Maksimum Öncelikli Kuyruğa Öğe Ekleme

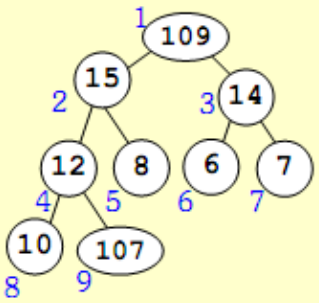


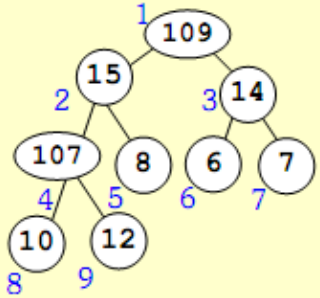
0	1	2	3	4	5	6	7
	14	12	7	10	8	6	
	14	12	7	10	8	6	15
	14	12	15	10	8	6	7
	15	12	14	10	8	6	7

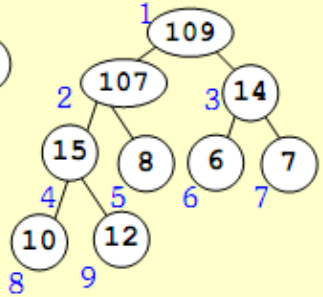


0	1	2	3	4	5	6	7	8
	15	12	14	10	8	6	7	109
	15	12	14	109	8	6	7	10
	15	109	14	12	8	6	7	10
	109	15	14	12	8	6	7	10

Maksimum Öncelikli Kuyruğa Öğe Ekleme





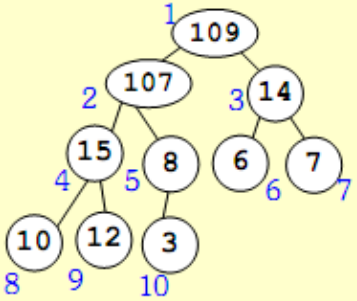


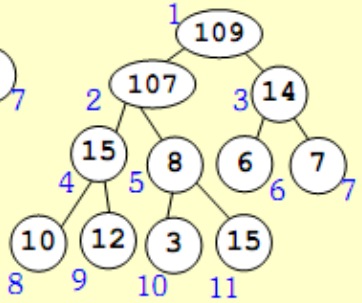
+107

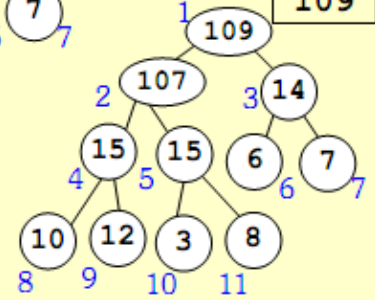
0	1	2	3	4	5	6	7	8	9
	109	15	14	12	8	6	7	10	107

0	1	2	3	4	5	6	7	8	9
	109	15	14	107	8	6	7	10	12

0	1	2	3	4	5	6	7	8	9
	109	107	14	15	8	6	7	10	12







+3 +15

0	1	2	3	4	5	6	7	8	9	10
	109	107	14	15	8	6	7	10	12	3

1	2	3	4	5	6	7	8	9	10	11
109	107	14	15	8	6	7	10	12	3	15

1	2	3	4	5	6	7	8	9	10	11
109	107	14	15	15	6	7	10	12	3	8

Maksimum Öncelikli Kuyruk fonksiyonları

```
HeapControl(dizi) //yığının boş olup olmadığı
{
  if (heapsize != dizi.Length)
     return true;
  return false;
}

Left(i)  { return 2 * (i + 1) - 1; }
Right(i) { return 2*(i+1); }
Parent(i){ return (i-1)/2;}
```

Maksimum Öncelikli Kuyruğa Öğe Ekleme

```
HeapInsert(dizi, key)
{ if (HeapControl(dizi))
  {i = heapsize;
     heapsize++;
    while (i > 0 && dizi[Parent(i)] < key)
    { dizi[i] = dizi[Parent(i)];
      i = Parent(i);
    }
    dizi[i] = key;
  }
}
```

Maksimum Öncelikli Kuyruğu İnşa Etme

```
BuildHeap(dizi)
{
   heapsize = dizi.Length - 1;

   for ( i = (dizi.Length-1)/2;i>=0; i--)

        heapify(dizi, i);
}
```

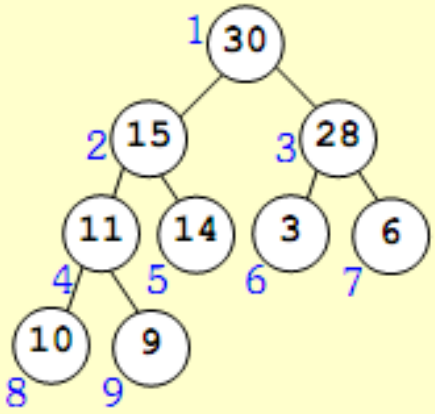
Maksimum Öncelikli Kuyruğu oluşturma

```
heapify(dizi, i)
{ L = Left(i); R = Right(i);
 if (L <= heapsize && dizi[L] > dizi[i])
    largest = L;
 else  largest = i;
 if (R <= heapsize && dizi[R] > dizi[largest])
    largest = R;
 if (largest != i)
  {temp = dizi[largest]; dizi[largest] = dizi[i];
   dizi[i] = temp;  heapify(dizi, largest);
  }
}
```

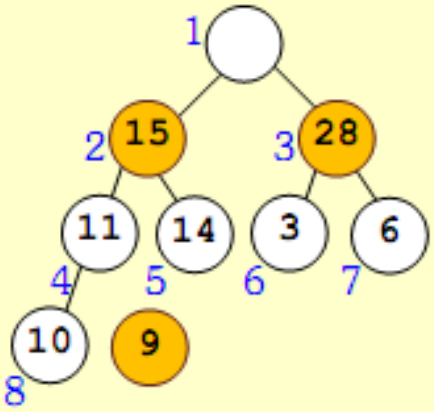
Maksimum Öncelikli Kuyruktan Öğe Alma/Silme

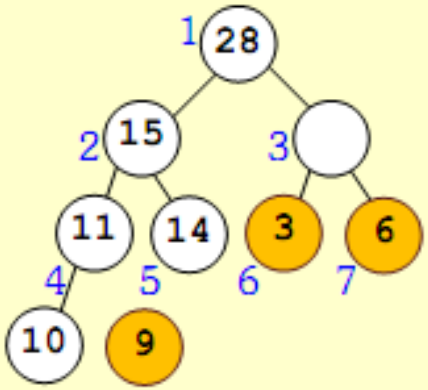
- Yığını diziyle gerçekleştirmiştik. Yığının tanımı gereği ilk alınacak eleman, dizideki ilk elemandır. Bunun yanında “tam ağaç” yapısını bozmamak için en sondaki elemanı da uygun bir konuma kaydırmalıyız.
- Bu kaydırmayı yaparken maksimum yığın yapısını korumaya dikkat etmeliyiz.
- Her adımda (iki çocuk ve sondaki eleman olmak üzere) dikkat edilecek 3 değer var. İki gösterge gibi; i ile, en sondaki elemanı kaydıracağım konumu ararken; j ile de üzerinde bulunduğum düğümün çocuklarını kontrol ediyorum. Bu üç değerden (iki çocuk ve sondaki eleman) en büyük olanı, i’nin gösterdiği yere taşıyorum. Sonra da taşınan değerin eski konumuna ilerleyip aynı kıyaslamaları yapıyorum. En sonunda, sondaki düğüm bir yere yerleşene kadar. En sondaki öğe bazı durumlarda arada bir yere girebilir; en kötü durumda ise arada bir yere yerleşemez, en sona konması gerekir. O(lgn).

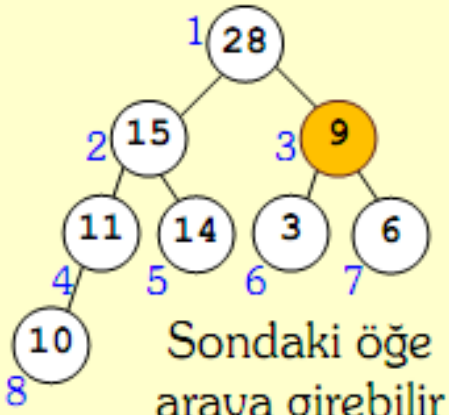
Maksimum Öncelikli Kuyruktan Öğe Alma/Silme



30, 15, 28, 11, 14, 3, 6, 10, 9
verisi üzerinde silme işlemleri



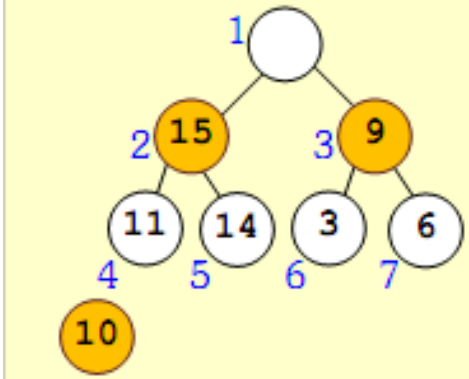


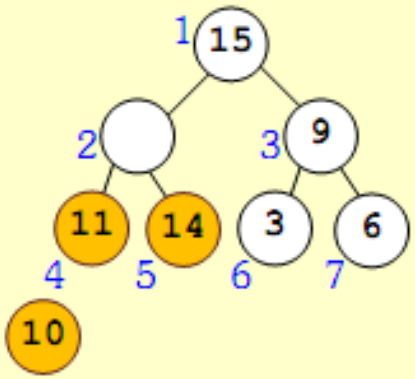


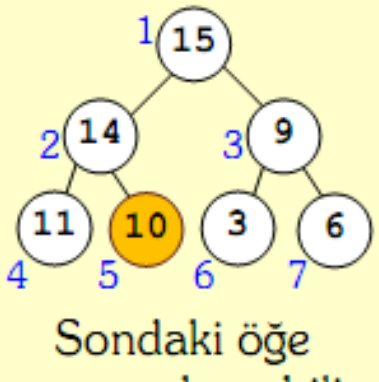
Sondaki öğe
araya girebilir

Maksimum Öncelikli Kuyruktan Öğe Alma/Silme

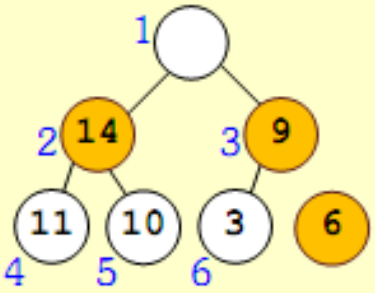
30, 15, 28, 11, 14, 3, 6, 10, 9
verisi üzerinde silme işlemleri

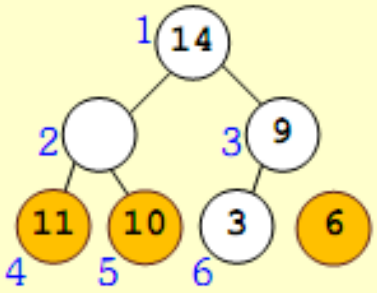


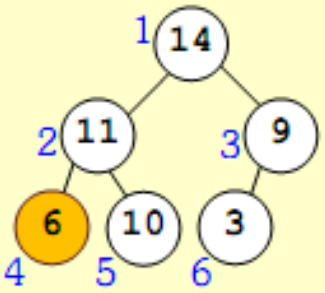




Sondaki öğe
en sona konabilir

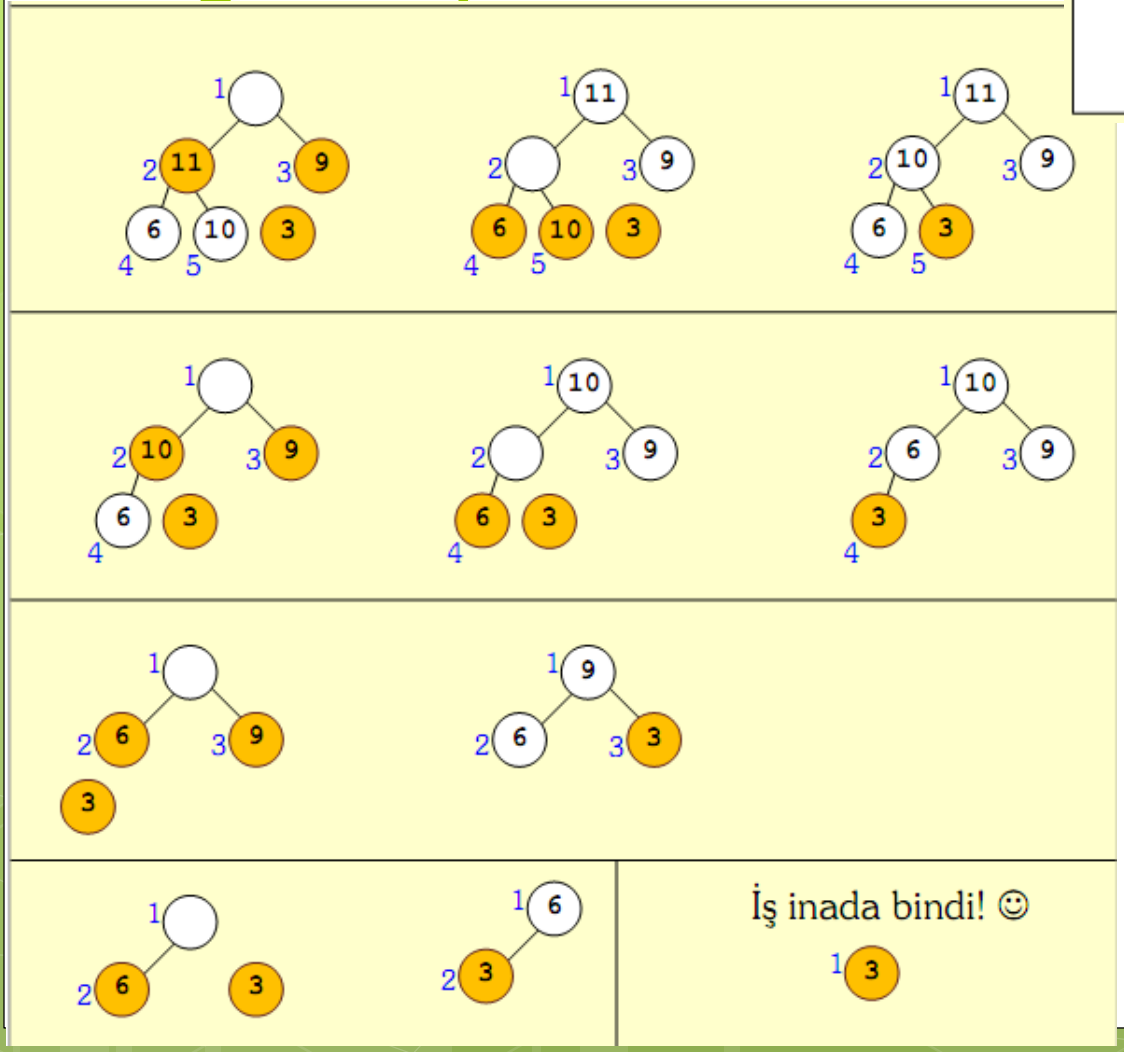






Maksimum Öncelikli Kuyruktan Öğe Alma/Silme

30, 15, 28, 11, 14, 3, 6, 10, 9
verisi üzerinde silme işlemleri



Maksimum Öncelikli Kuyruk Java

```
HeapExtract_or_Sort(dizi)
{ BuildHeap(dizi);
    for (i = heapsize; i >=0; i--)
      {       temp = dizi[0];
               dizi[0] = dizi[i];
               dizi[i] = temp;
               heapsize--;
               heapify(dizi, 0);
          }
}
```

Yazan:Şadi Evren ŞEKER

Çok Yollu Ağaçlar

(Multi-Way Trees)

Çok Yollu Ağaçlar (Multi-Way Trees)

- B -Trees
- B* -Trees
- B+ -Trees
- B# -Trees

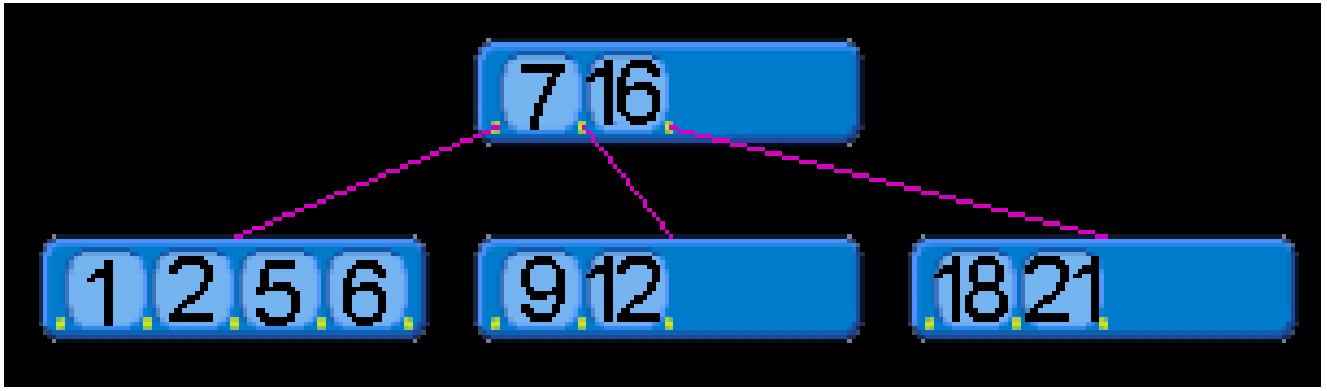
Çok Yollu Ağaçlar (Multi-Way Trees)

- Disk üzerindeki bilgilerin elde edilmesinde kullanılır.
- 3600 rpm ile dönen bir disk için bir tur 16.7ms'dir.
- Ortalama olarak 8 ms'de (gecikme zamanı) istediğimiz noktaya konumlanırız.
- Saniyede yaklaşık 125 kez diskte konumlanabiliriz.
- Bir saniyede 25 milyon komut gerçekleştirebiliriz.
- Bir disk erişimi yaklaşık 200.000 komut zamanı almaktadır.
- (**7200 Rpm** disk **bir turunu** 8.3 **ms**'de tamamlar, ortalama gecikme süresi 4 ms dir.

- **Multi-Way ağaçlar disk erişim sayısını azaltmayı amaçlamaktadır.**

Çok Yollu Ağaçlar (Multi-Way Trees)

- Bir multi-way ağaç sıralı bir ağaçtır ve aşağıdaki özelliklere sahiptir.
  - Bir m-way arama ağacındaki her node, m-1 tane anahtar (key) ve m tane çocuğa sahiptir.
  - Bir node'daki anahtar, sol alt ağaçtaki tüm anahtarlardan büyüktür ve sağ alt ağaçtaki tüm anahtarlardan küçüktür.



Çok Yollu Ağaçlar -B-Trees

- Root (kök) node en az iki tane yaprak olmayan node'a sahiptir.
- Yaprak ve kök olmayan her node k-1 tane anahtara ve k adet alt ağaç referansına sahiptir. ($m/2 \leq k \leq m$)
- Her yaprak node'u k -1 anahtara sahiptir. ($m/2 \leq k \leq m$)
- Bütün yapraklar aynı seviyededir.
- Herhangi bir döndürmeye gerek kalmadan (AVL tree deki gibi) otomatik olarak balance edilirler.

Çok Yollu Ağaçlar -B-Trees

- **Bir anahtar ekleme;**

1. Eğer boş alanı olan bir yaprağa yerleştirilecekse doğrudan yaprağın ilgili alanına yerleştirilir.

2. Eğer ilgili yaprak doluysa, yaprak ikiye bölünür ve anahtarların yarısı yeni bir yaprak oluşturur. Eski yapraktaki en son anahtar bir üst seviyedeki node' aktarılır ve yeni yaprağı referans olarak gösterir.

3. Eğer root ve tüm yapraklar doluysa, önce ilgili yaprak ikiye bölünür ve eski yapraktaki en son anahtar root'a aktarılır. Root node'da dolu olduğu için ikiye bölünür ve eski node'daki en son anahtar root yapılır.

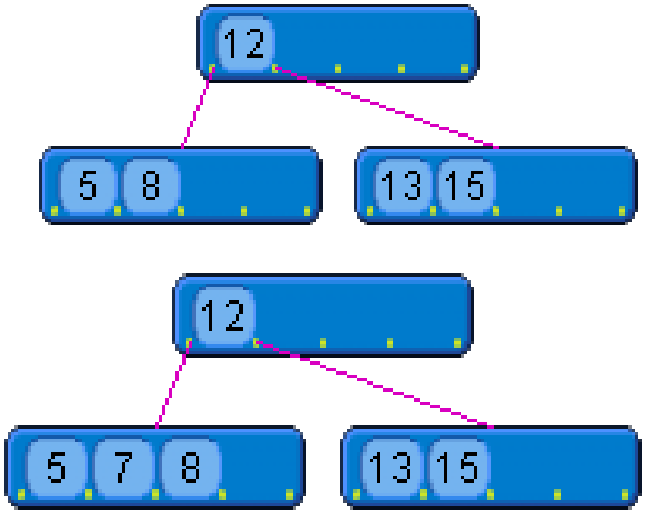
Çok Yollu Ağaçlar -B-Trees

- B -Tree Oluşturulması
- class BtreeNodeC
- {
- public int m = 4;
- public bool yaprak = true;
- public int[] keys = new int[m-1];
- BtreeNodeC[] referanslar = new BtreeNodeC[m];
- public BtreeNodeC(int key)
- {
- this.keys[0] = key;
- for (int i=0; i<m; i++)
- referanslar[i] = null;
- }
- }

GENEL ÖRNEK:
http://www.jbixbe.com/doc/tutorial/BTree.html

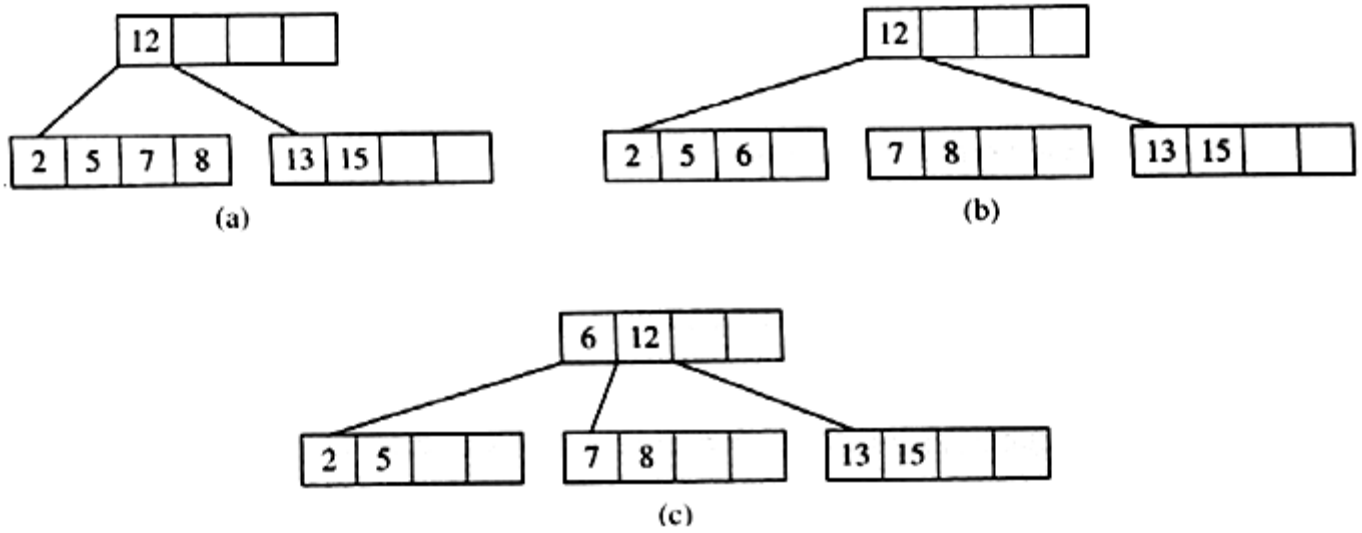
Çok Yollu Ağaçlar -B-Trees

- **1- Yerleştirilecek yaprak boş ise,**
- Örnek olarak 7’nin eklenmesi



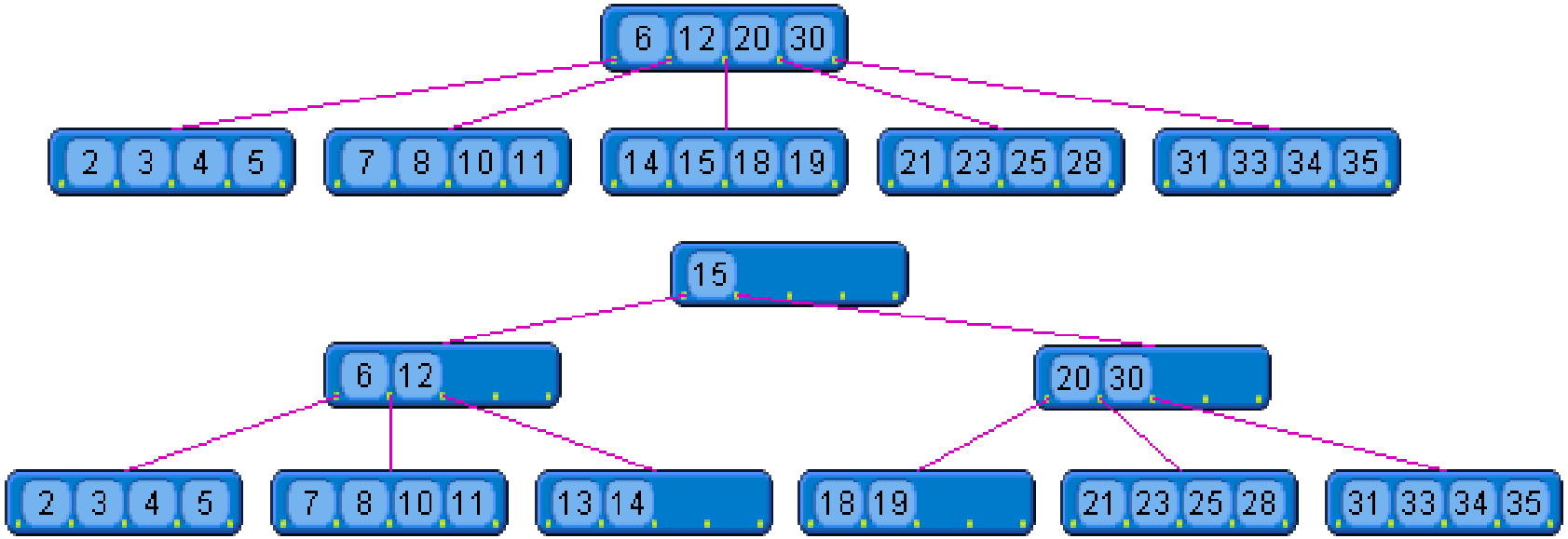
Çok Yollu Ağaçlar -B-Trees

- **2- Yerleştirilecek yaprak dolu ise,**
- Örnek: Anahtar olarak 6 eklenmesi



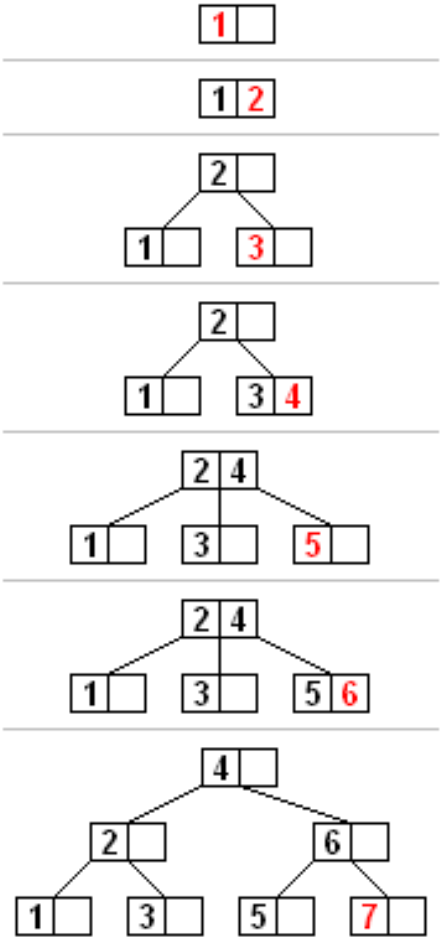
Çok Yollu Ağaçlar -B-Trees

- **3-Yerleştirilecek yaprak dolu ve root node'da dolu ise,**
- Örnek: Anahtar olarak 13 eklenmesi



Çok Yollu Ağaçlar -B-Trees

- Örnek

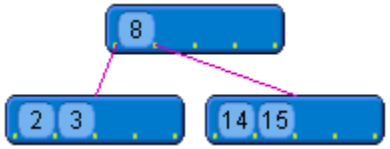


Çok Yollu Ağaçlar -B-Trees

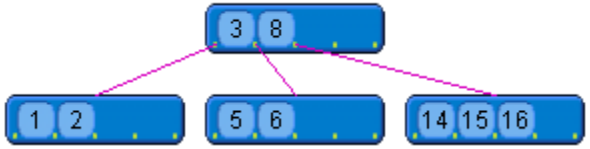
- Örnek:5.Derece bir M-Way ağaca anahtar ekleme
- Ekle: 2,8,14,15



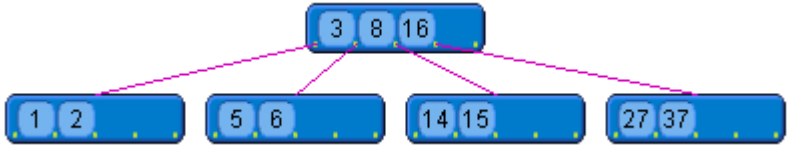
- Ekle:3



- Ekle:1,16,6,5



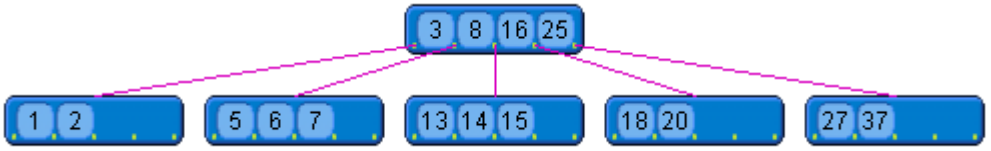
- Ekle:27,37



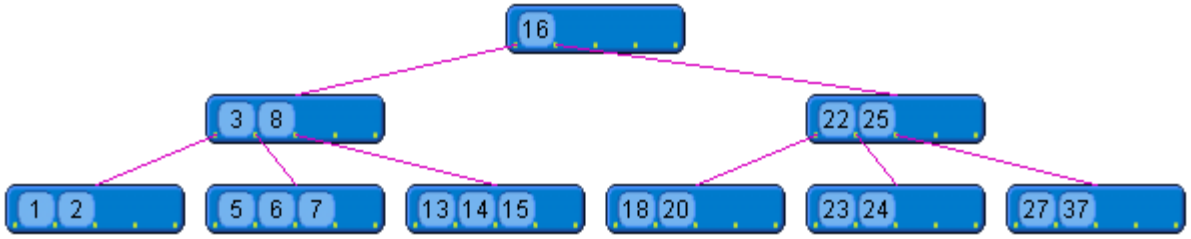
Çok Yollu Ağaçlar -B-Trees

- Örnek:5.Derece bir M-Way ağaca anahtar ekleme

- Ekle: 18,25,7,13,20

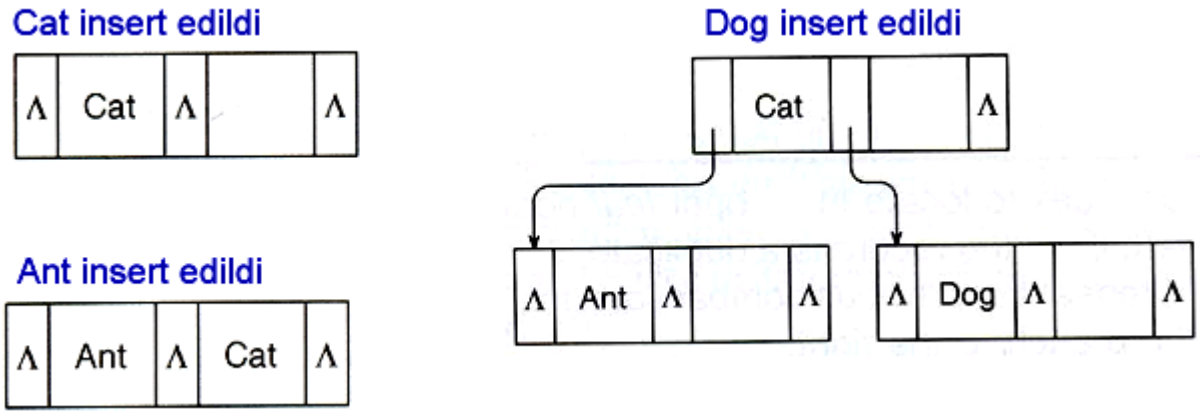


- Ekle: 22,23,24



Çok Yollu Ağaçlar -B-Trees

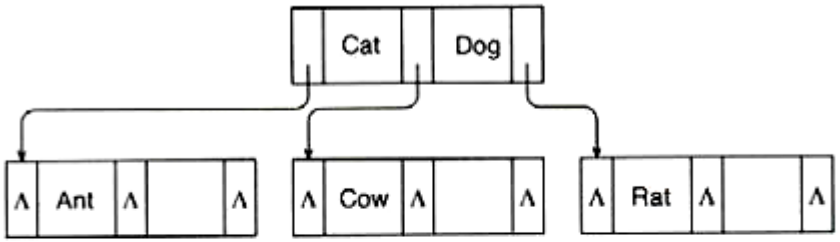
- Örnek
- *d = 1* (capacity order)
- *cat, ant, dog, cow, rat, pig, gnu*



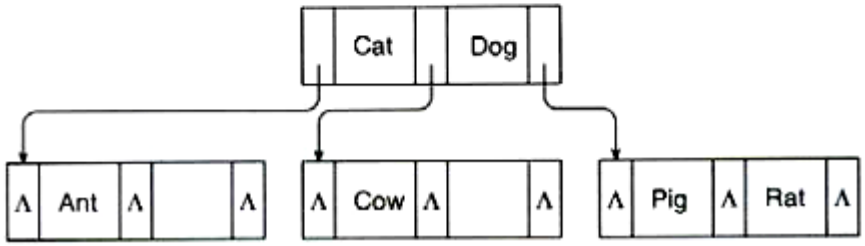
Çok Yollu Ağaçlar -B-Trees

- Örnek
- *d = 1* (capacity order)
- *cat, ant, dog, cow, rat, pig, gnu*

Cow ve Rat insert edildi

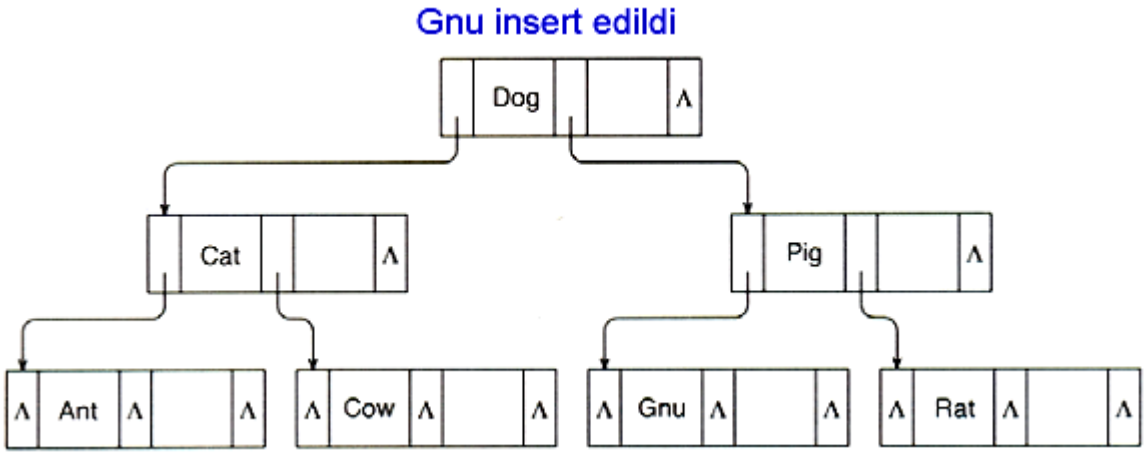


Pig insert edildi



Çok Yollu Ağaçlar -B-Trees

- Örnek
- *d = 1* (capacity order)
- *cat, ant, dog, cow, rat, pig, gnu*

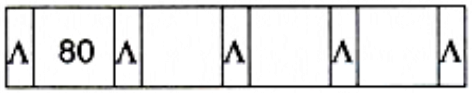


- Doldurma faktörü = depolanan kayıt sayısı / kullanılan yer sayısı
- = 7 / 14 = 50 %

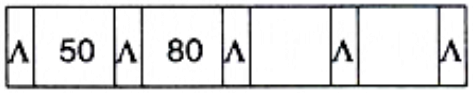
Çok Yollu Ağaçlar -B-Trees

- Örnek
- *d = 2* (capacity order)
- *80,50,100,90,60,65,70,75,55,64,51,76,77,78,200,300,150*

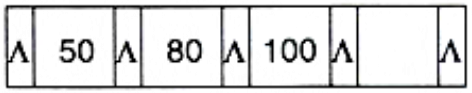
1.Insert 80.



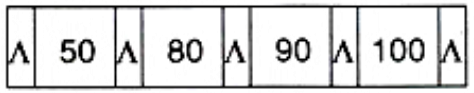
2. Insert 50.



3. Insert 100.

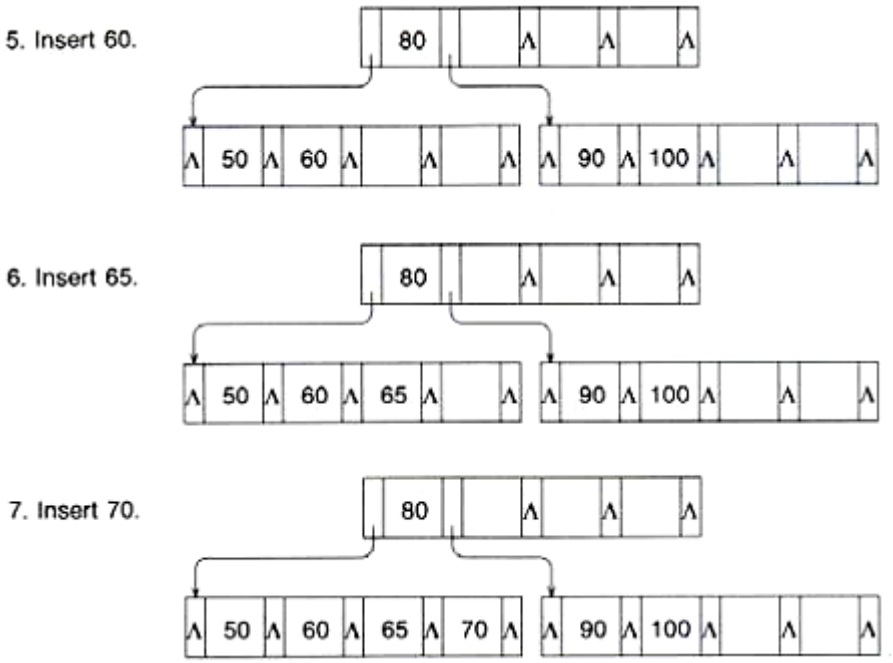


4. Insert 90



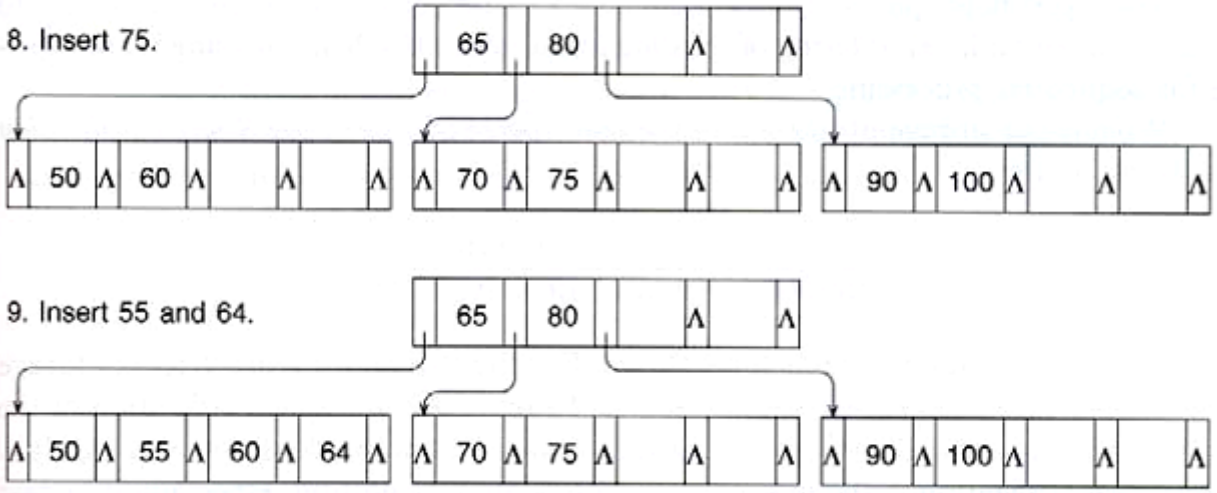
Çok Yollu Ağaçlar -B-Trees

- Örnek
- *d = 2* (capacity order)
- *80,50,100,90,60,65,70,75,55,64,51,76,77,78,200,300,150*



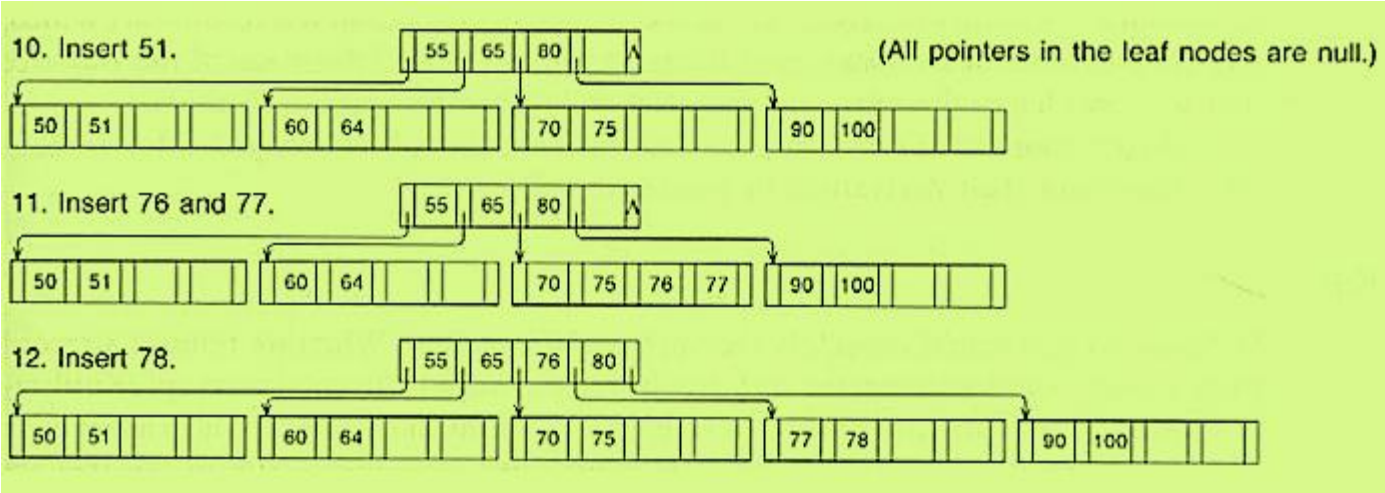
Çok Yollu Ağaçlar -B-Trees

- Örnek
- *d = 2* (capacity order)
- *80,50,100,90,60,65,70,75,55,64,51,76,77,78,200,300,150*



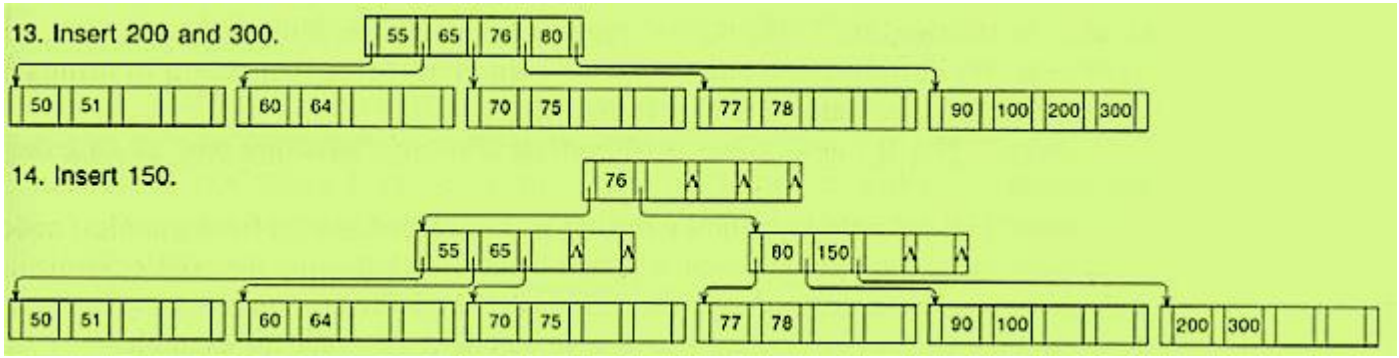
Çok Yollu Ağaçlar -B-Trees

- Örnek
- *d = 2* (capacity order)
- *80,50,100,90,60,65,70,75,55,64,51,76,77,78,200,300,150*



Çok Yollu Ağaçlar -B-Trees

- Örnek
- *d = 2* (capacity order)
- *80,50,100,90,60,65,70,75,55,64,51,76,77,78,200,300,150*

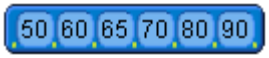


- *Packing factor = depolanan kayıt sayısı / kullanılan yer sayısı*
- *= 17 / 36 = 47 %*

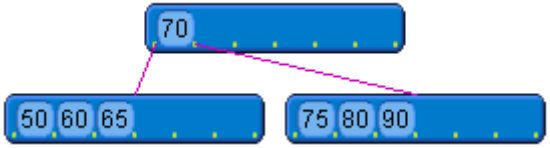
Çok Yollu Ağaçlar -B-Trees

- Örnek
- *d = 3* (capacity order)
- *80,50,90,60,65,70,75,55,64,51,76,77,78,10,13,15,1,3,5,6,20,32*

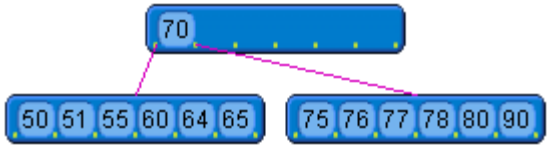
- *80,50,90,60,65,70*



- *75*



- *55,64,51,76,77,78*



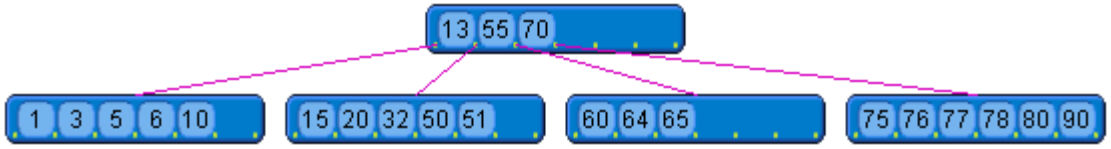
Çok Yollu Ağaçlar -B-Trees

- *10*



- *13,15,1,3,5,6,20,32*





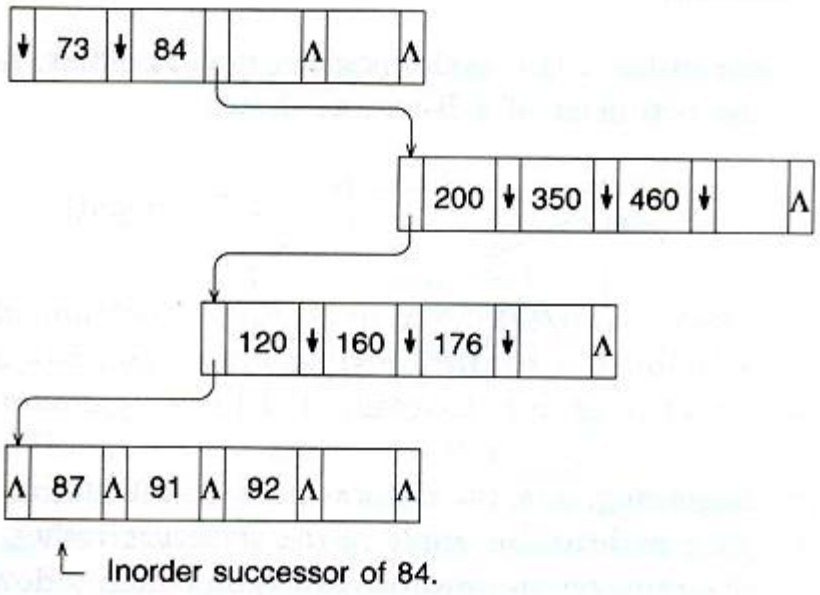
- *Packing factor = depolanan kayıt sayısı / kullanılan yer sayısı*
- *= 22 / 30 = 73 %*

B-Trees

Silme

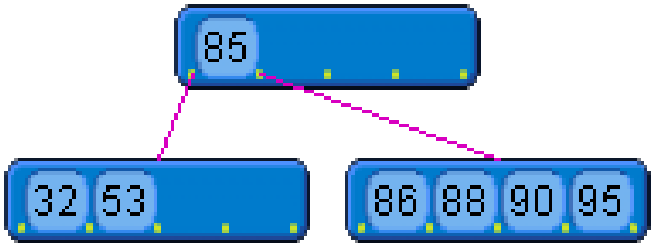
Çok Yollu Ağaçlar -B-Trees

- **Kural 1:** Minumum kapasitenin üzerindeki yapraklardan kayıt rahatlıkla silinebilir.
- **Kural 2:** Bir yaprak olmayan node üzerinden kayıt silindiğinde inorder takipçisi yerine yazılır. (Not: Eklerken soldaki en büyük düğüm, silerken sağdaki en küçük düğüm alınıyor)

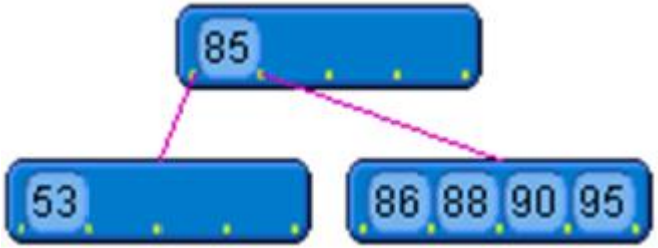


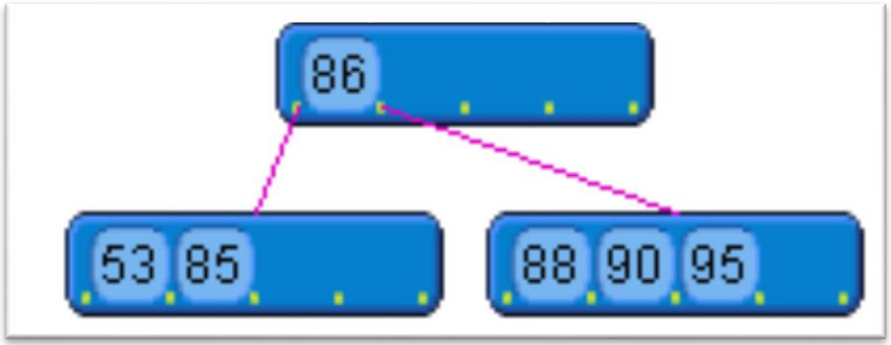
Çok Yollu Ağaçlar -B-Trees

- **Kural 3:** Bir node'daki kayıt sayısı minimum kapasite'den aşağı düşerse ve kardeş node'u fazla kayda sahipse, parent ve kardeş node ile yeniden düzenleme yapılır. Bir anlamda sola döndürme yapılır.



32 silindi.

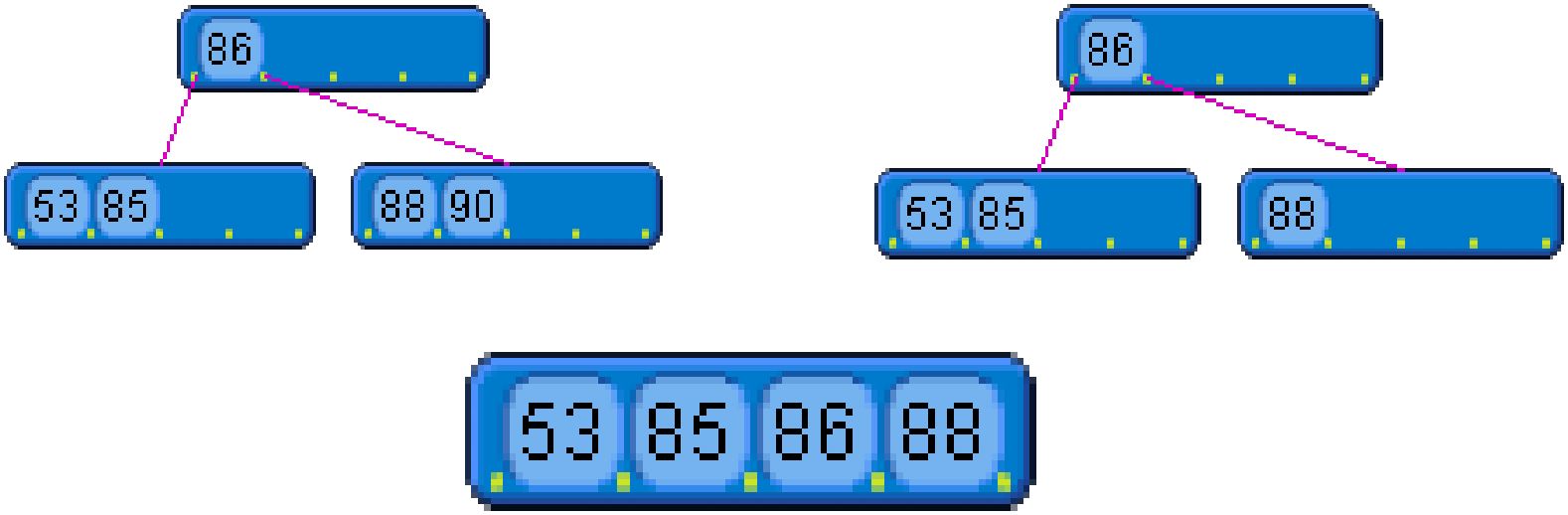




Çok Yollu Ağaçlar -B-Trees

- **Kural 4:** İki kardeş node minimum kapasitenin altına düşerse ikisi ve parent node'daki kayıt birleştirilir.

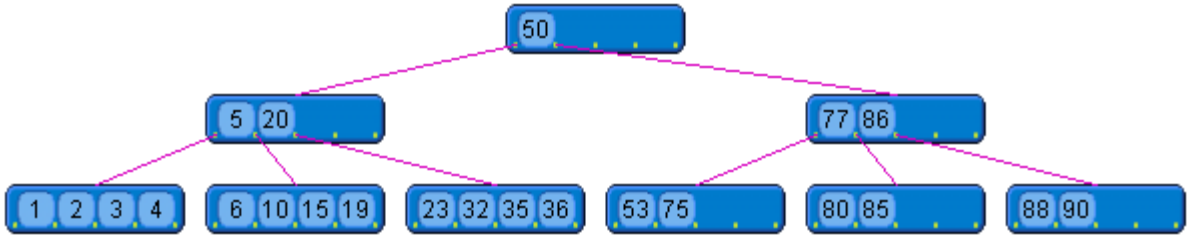
90 silindi



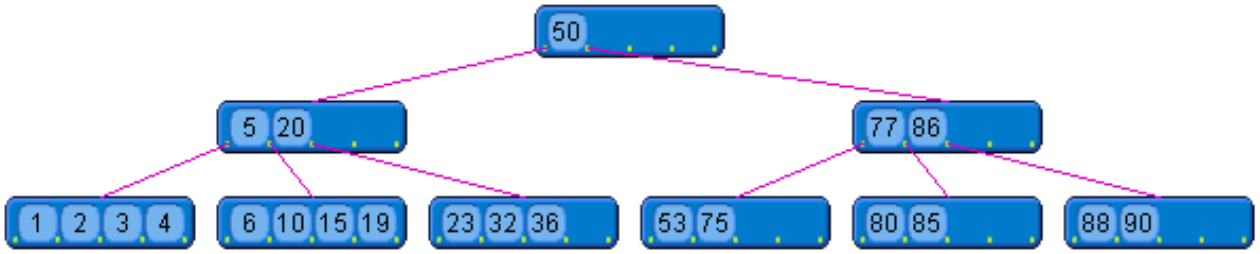
- *85 sağa döndürülürse soldaki minimum kapasitenin altına düşer*

Çok Yollu Ağaçlar -B-Trees

- **Örnek :1. Kural>** Minimum kapasitenin üstündeki yapraktan kayıt silinmesi

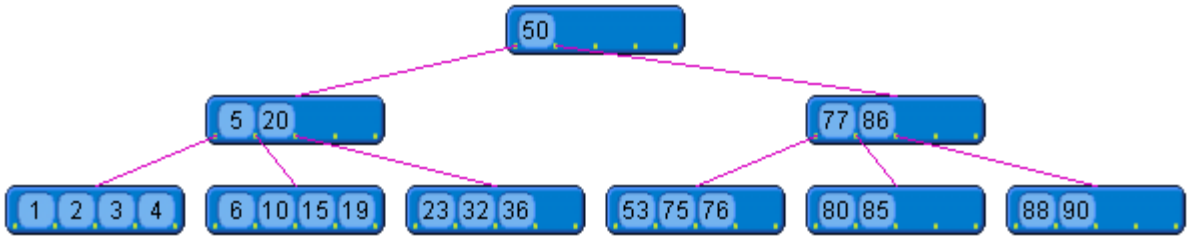


- *35 silindi.*

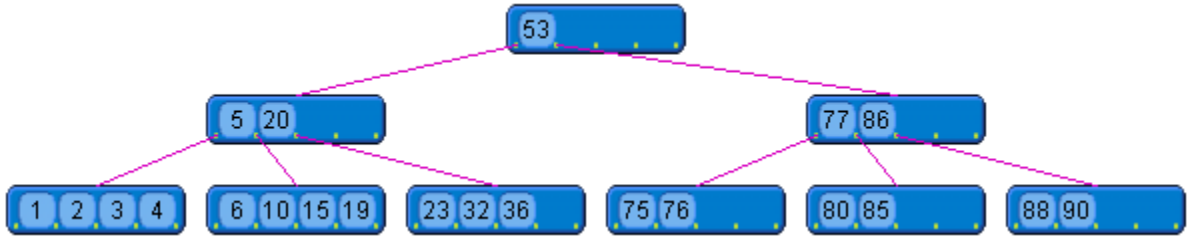


Çok Yollu Ağaçlar -B-Trees

- **Örnek : 2. Kural>** Bir nonleaf node'da kayıt silinmesi ve minimum kapasitenin üzerindeki bir node'dan kayıt aktarılması.

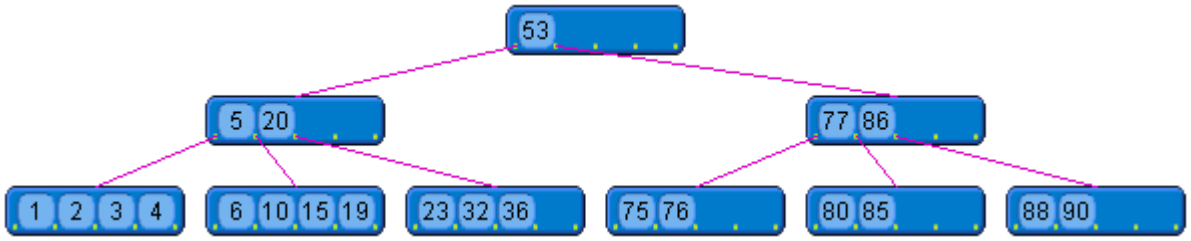


- 50 silindi

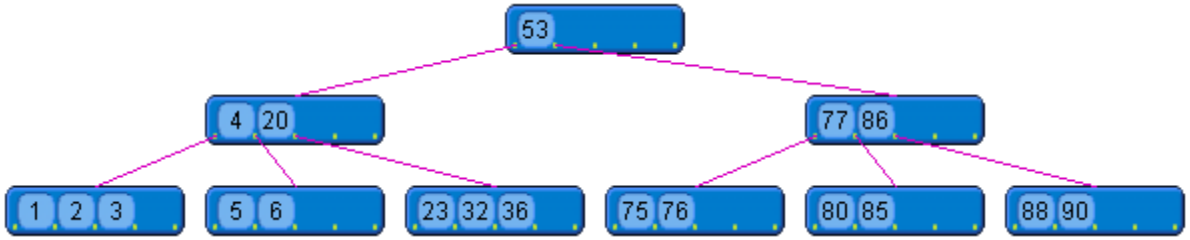


Çok Yollu Ağaçlar -B-Trees

- **Örnek :3.Kural>** Bir leaf node'da kayıt silinmesi ve minimum kapasitenin altına düşülmesi.

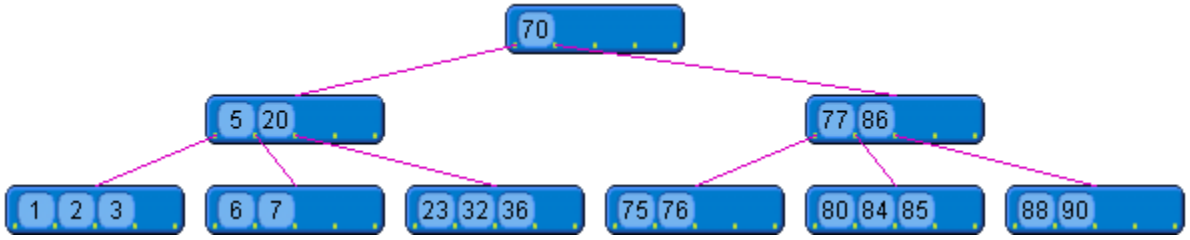


- 10,15,19 silinmesi (en fazla çocuğa sahip düğümden al)
- Sağa döndürme

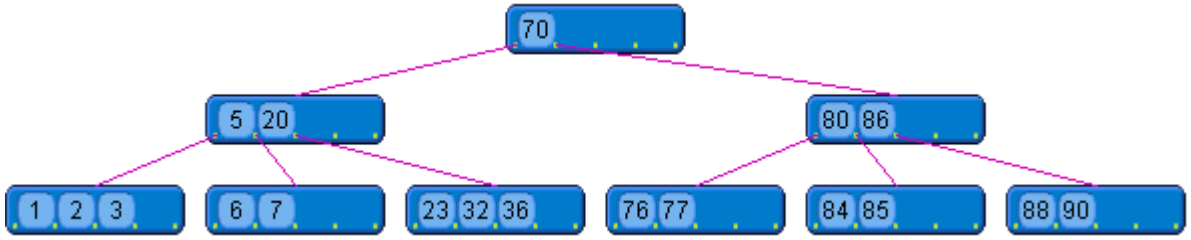


Çok Yollu Ağaçlar -B-Trees

- **Örnek :3.Kural>** Bir leaf node'da kayıt silinmesi ve minimum kapasitenin altına düşülmesi.

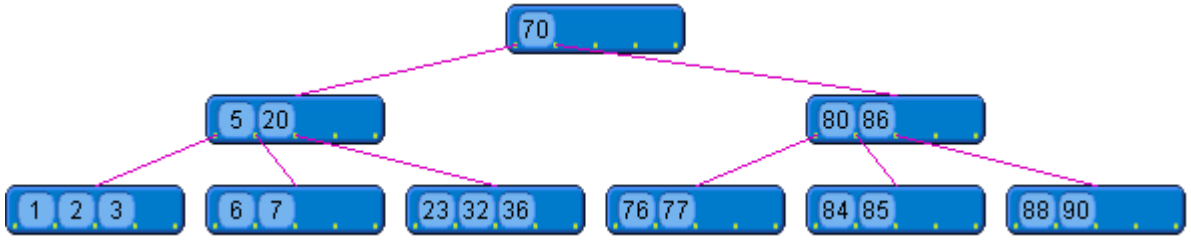


- 75 silindi. (silinen değer 77 solunda -Sola döndürme)

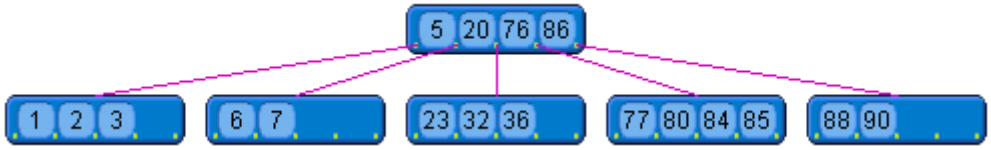


Çok Yollu Ağaçlar -B-Trees

- **Örnek:4.Kural >** Bir leaf node'da kayıt silinmesi ve minimum kapasitenin altına düşülmesi ve node'ların birleştirilmesi. (Kök dahil sağ taraftaki hangi düğüm silinirse silinsin minumum kapasitenin altına düşülecektir. Birleştirme işlemi gerekir.)

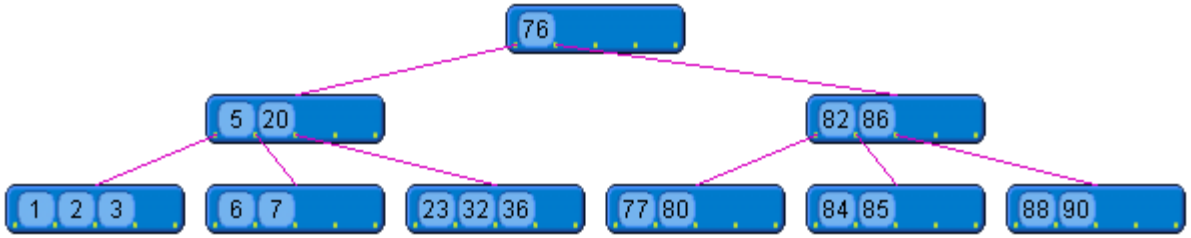


- 70 silindi

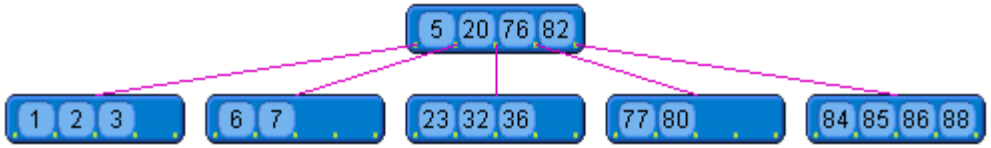


Çok Yollu Ağaçlar -B-Trees

- **Örnek:4.Kural >** Bir leaf node'da kayıt silinmesi ve minimum kapasitenin altına düşülmesi ve node'ların birleştirilmesi. (Kök dahil sağ taraftaki hangi düğüm silinirse silinsin minumum kapasitenin altına düşülecektir. Birleştirme işlemi gerekir.)

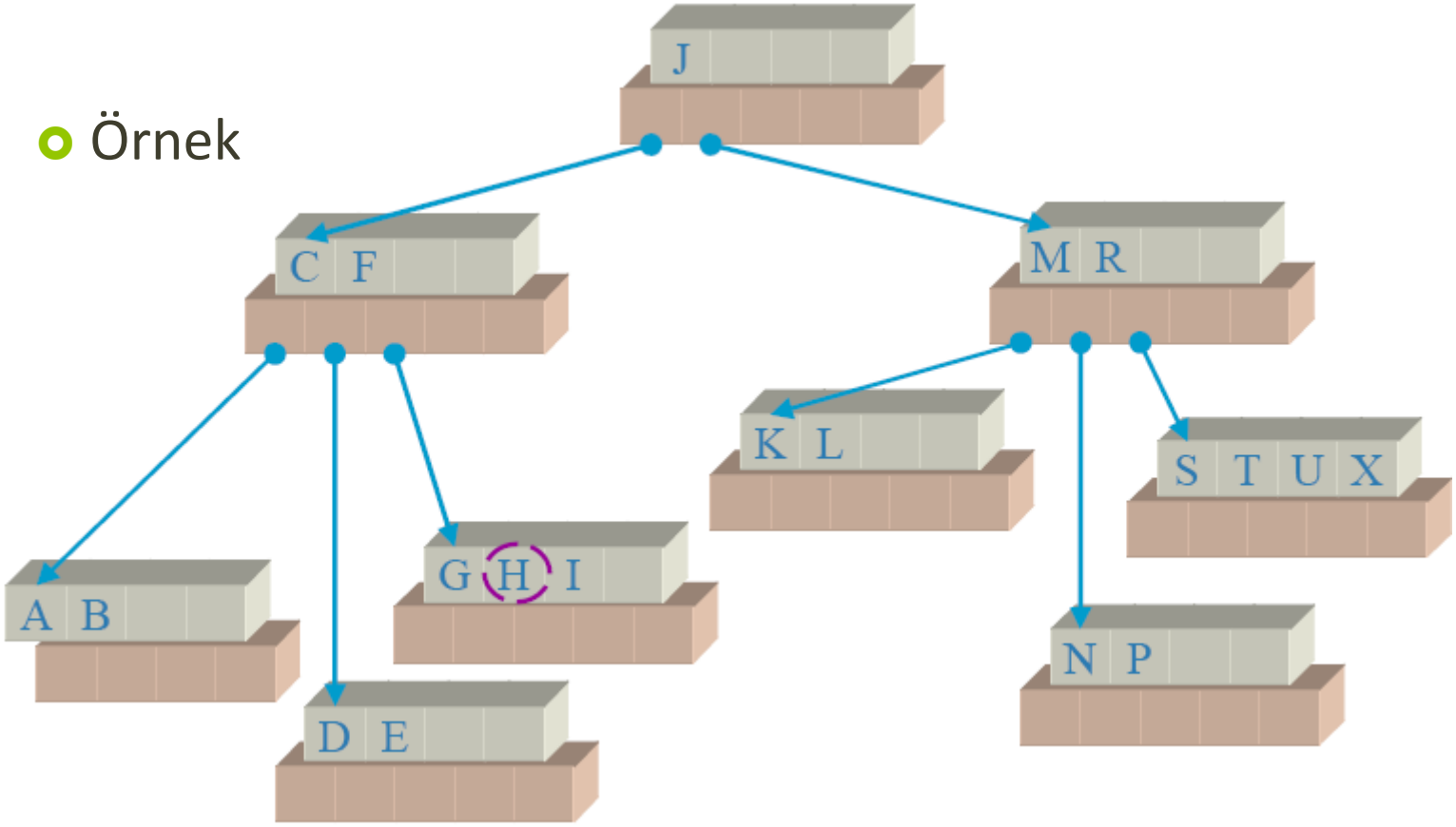


- 90 silindi

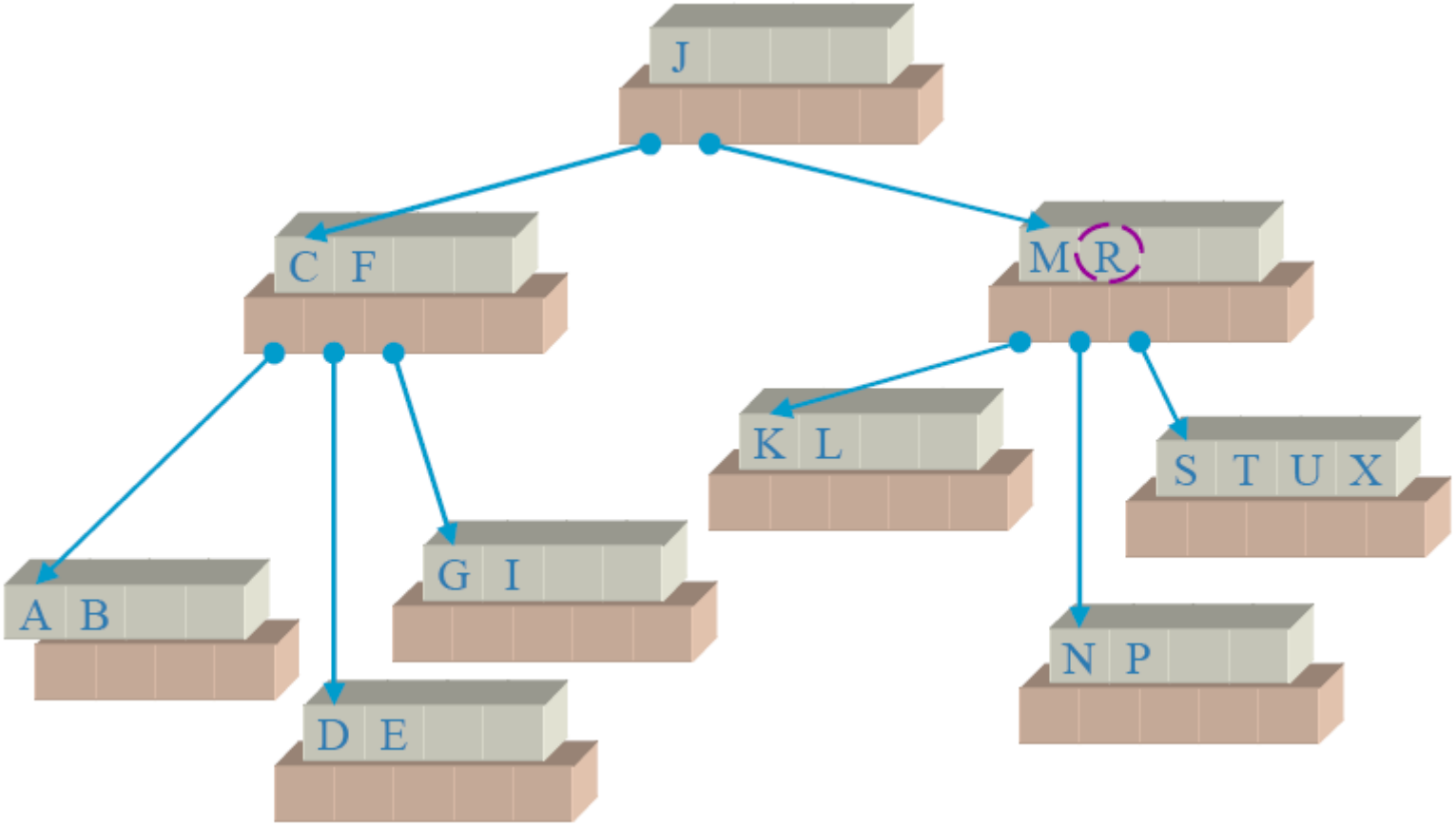


Çok Yollu Ağaçlar -B-Trees

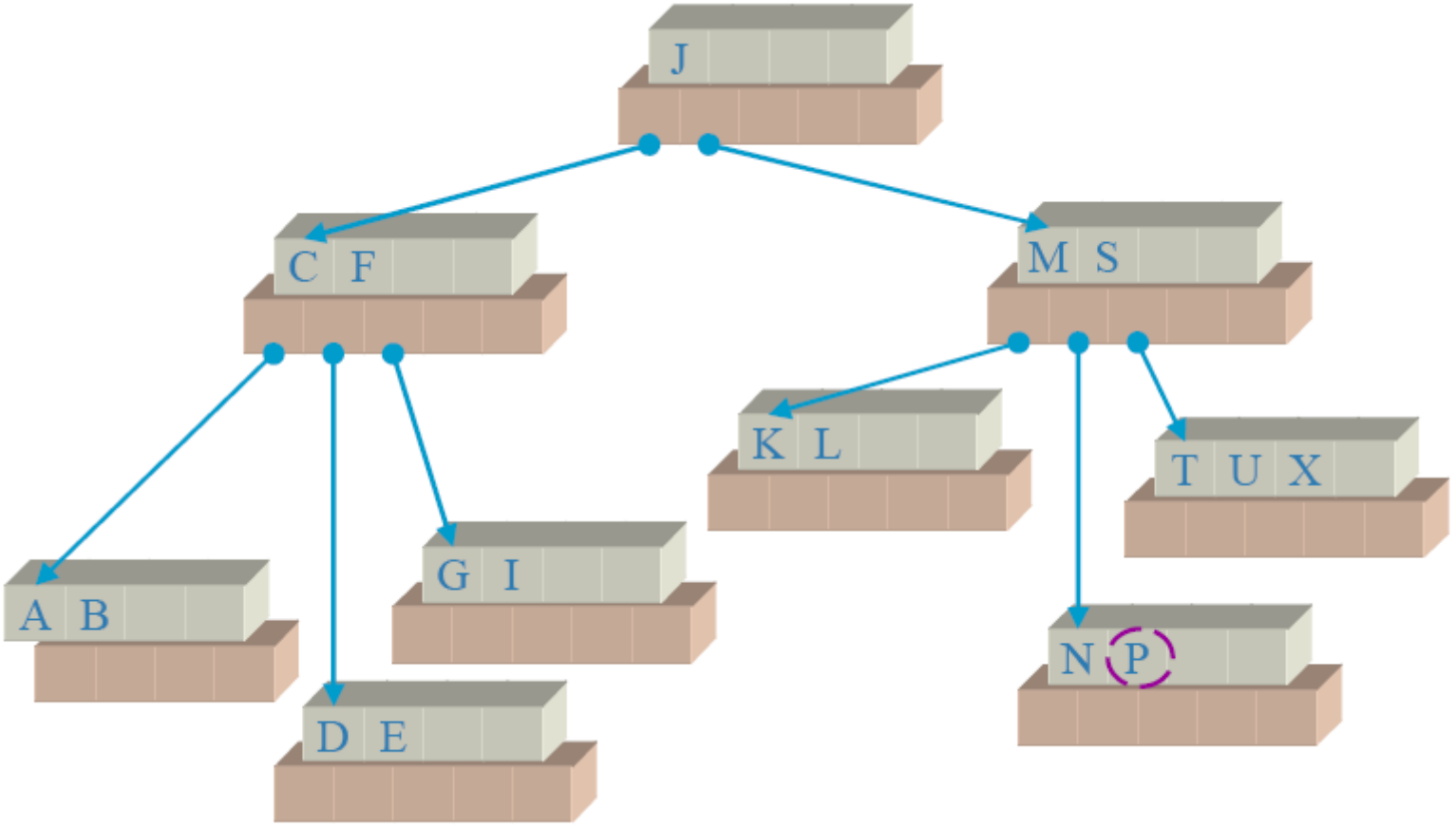
- Örnek



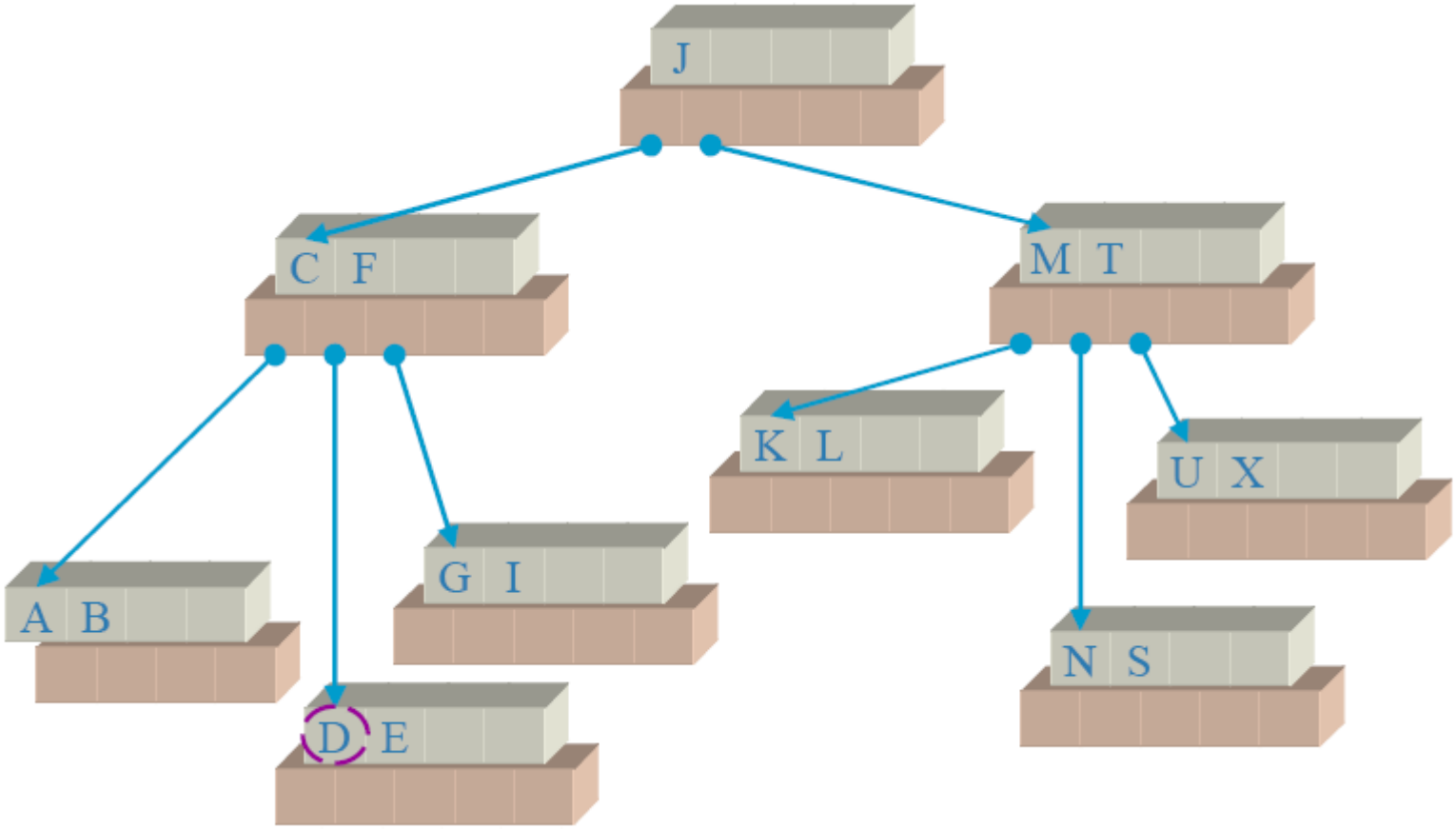
Çok Yollu Ağaçlar -B-Trees



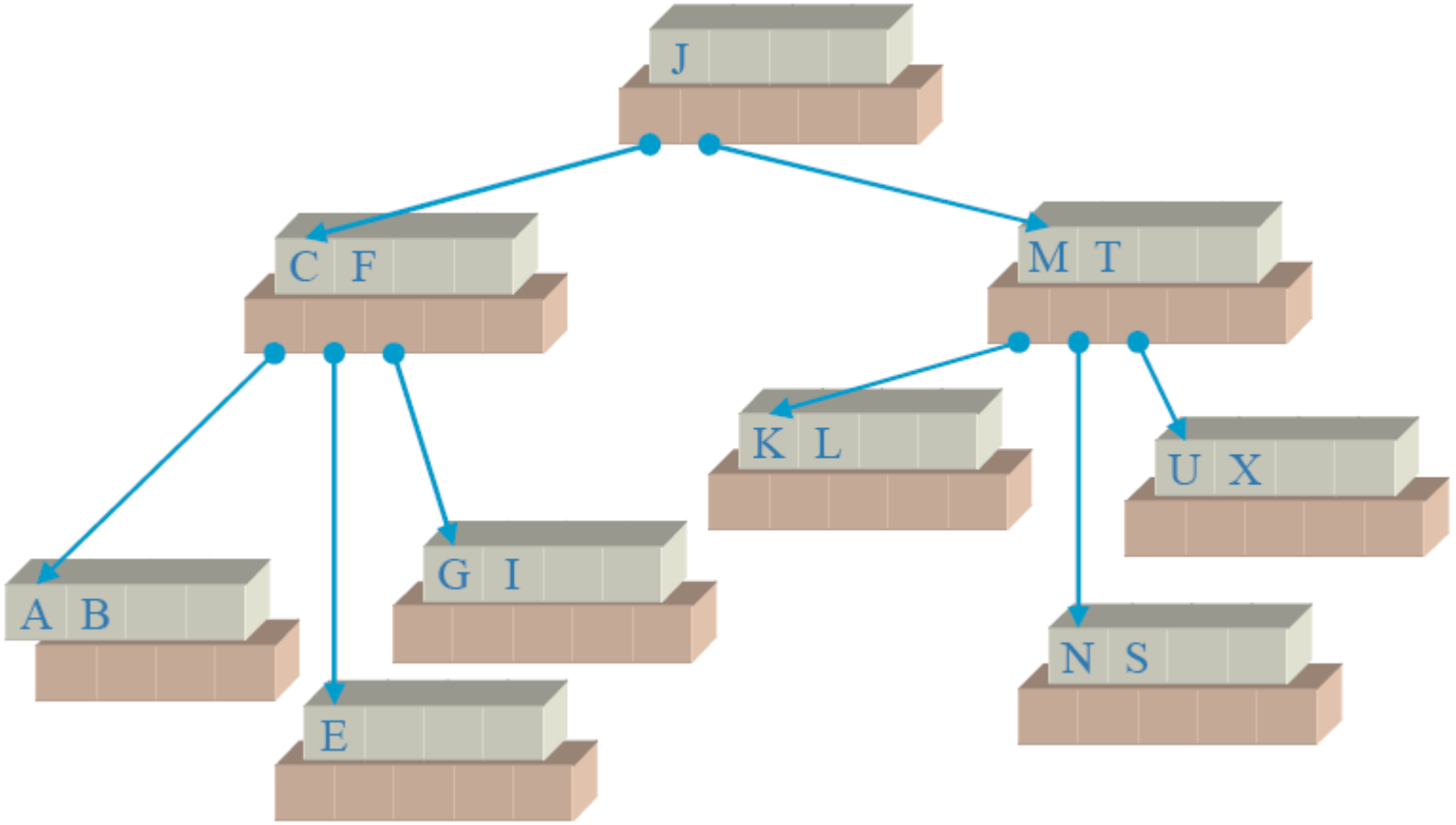
Çok Yollu Ağaçlar -B-Trees



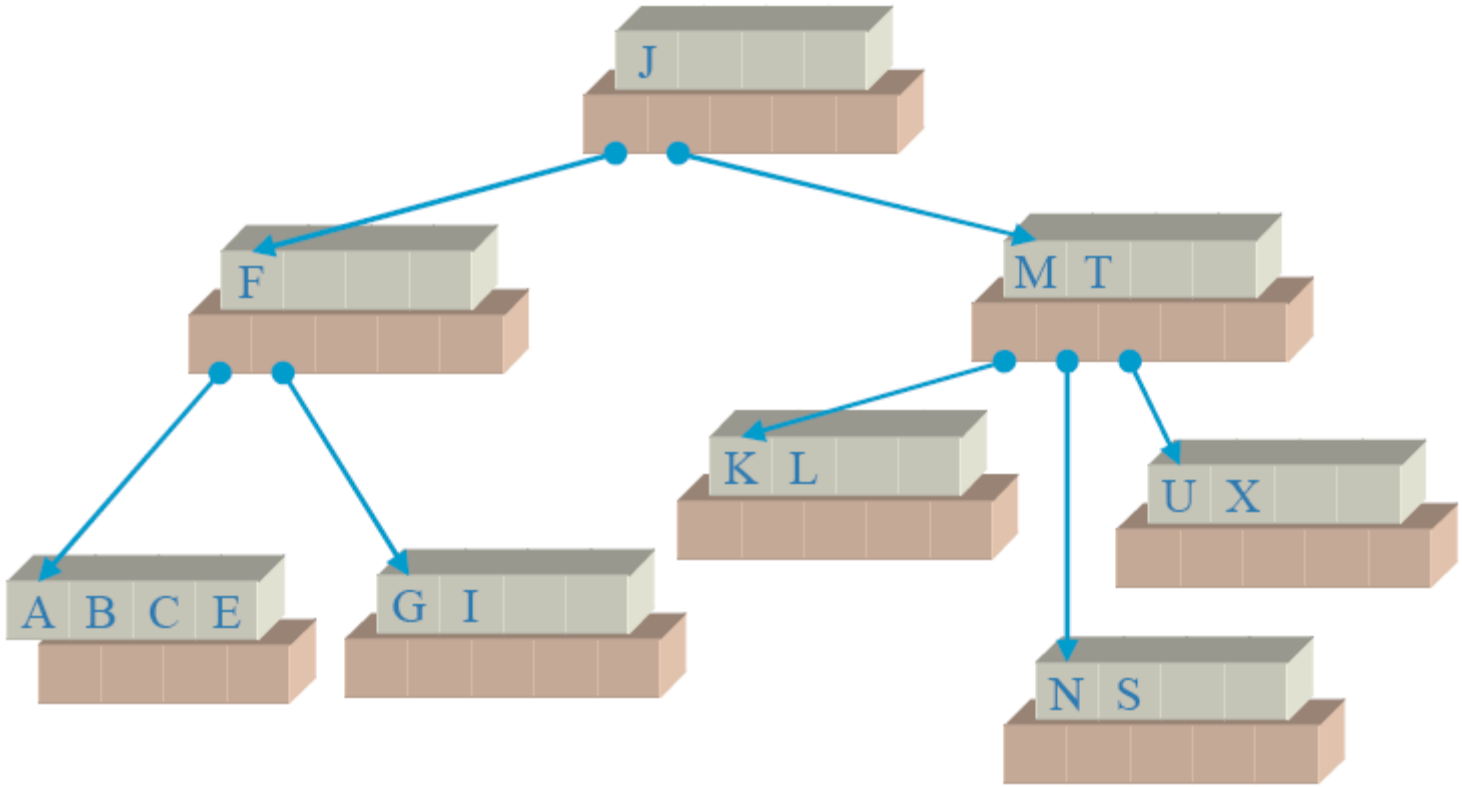
Çok Yollu Ağaçlar -B-Trees



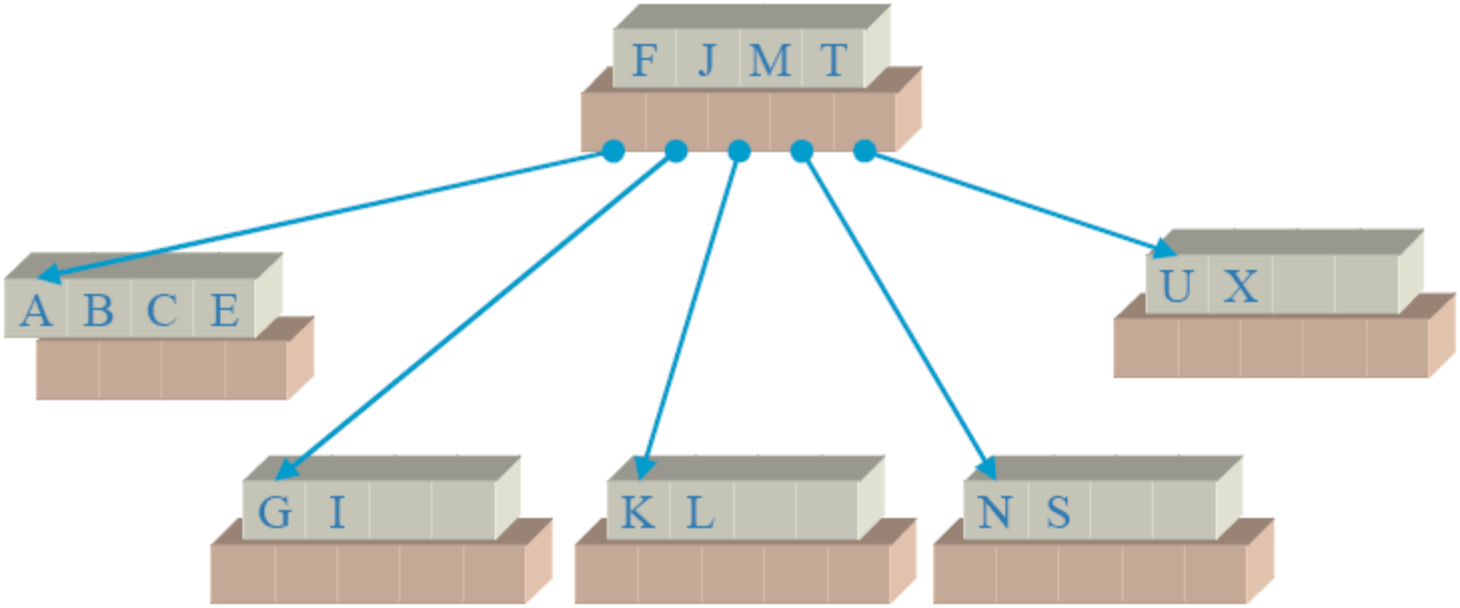
Çok Yollu Ağaçlar -B-Trees



Cok Yollu Ağaclar -B-Trees

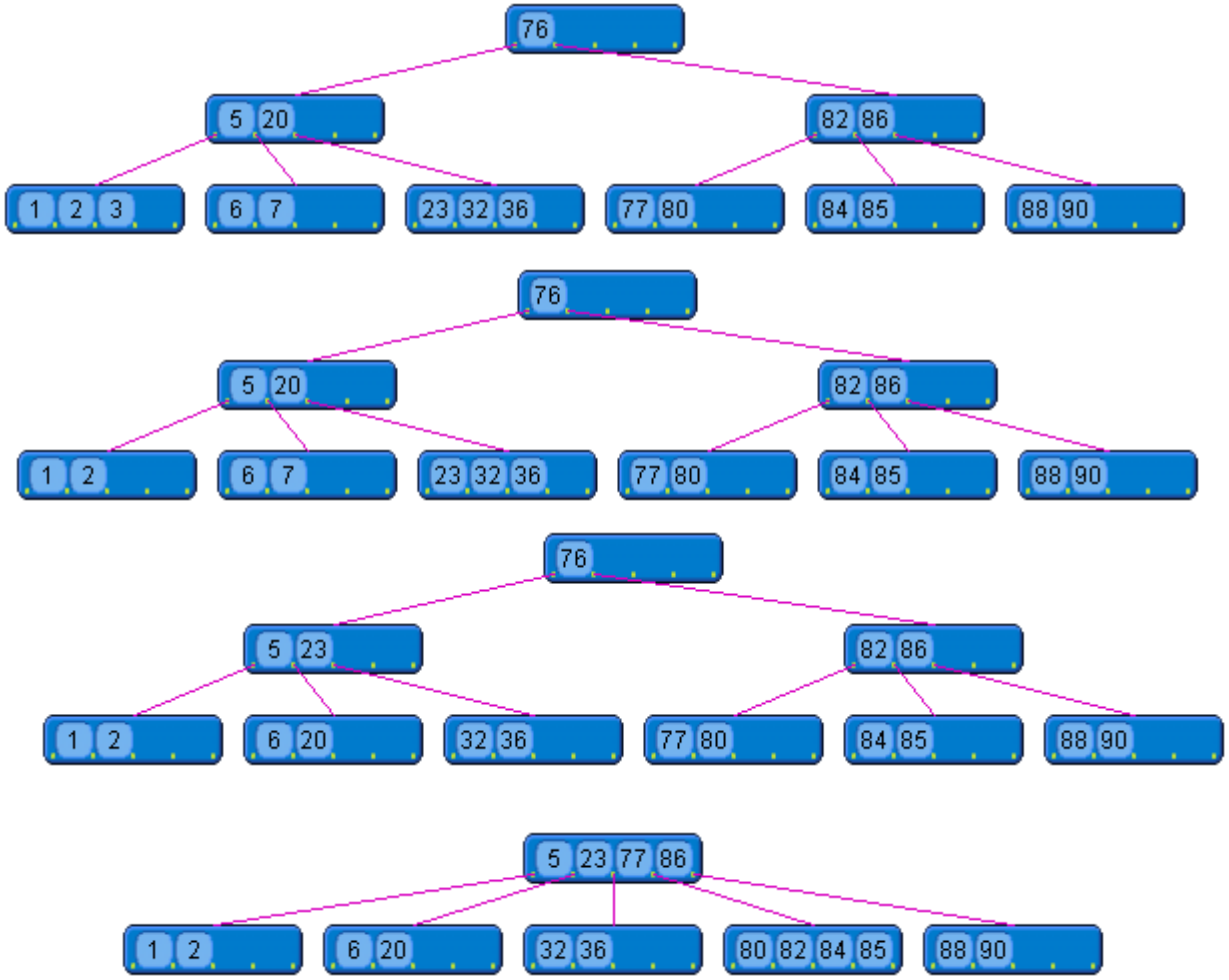


Çok Yollu Ağaçlar -B-Trees



Çok Yollu Ağaçlar -B-Trees

- **Örnek: Verilen ağaçtan sırasıyla 3,7,76 değerlerini siliniz**

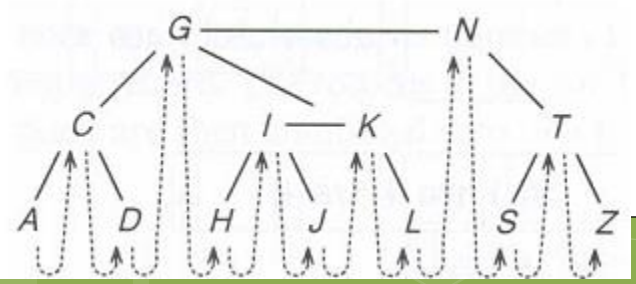


Çok Yollu Ağaçlar -B-Trees

- **Değerlendirme**
- B-Tree'lerde erişim araştırması kayda ulaşma değil node'a ulaşmadır
- Ağacın yüksekliği aşağıdaki gibi ifade edilir

$$\text{Height} \leq \log_{d+1} \frac{n+1}{2} + 1$$

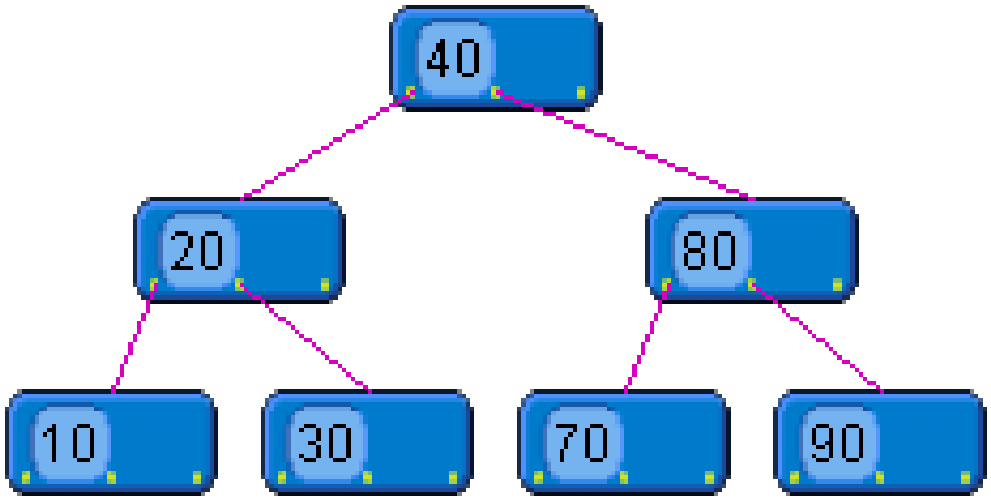
- Worst case erişim performansı $O(\log_d n)$ 'dir. n, B-tree'deki kayıt sayısıdır, d, kapasitedir.
- Kapasite sırası 50 olan ve 1 milyon kayıt bulunduran B-tree'de bir kayda ulaşmak için en fazla 4 erişim araştırması gerekir.
- B-tree'deki kayıtlara sıralı ulaşmak için inorder dolaşma yapılır.



Çok Yollu Ağaçlar -B-Trees

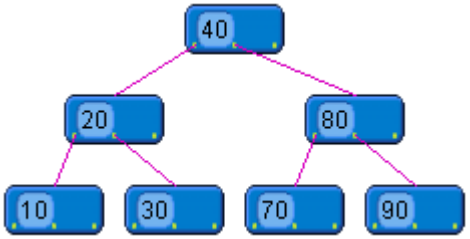
- Uygulama 1: d=1 olan ağaç için aşağıdaki değerleri girip B-tree ağacını oluşturunuz

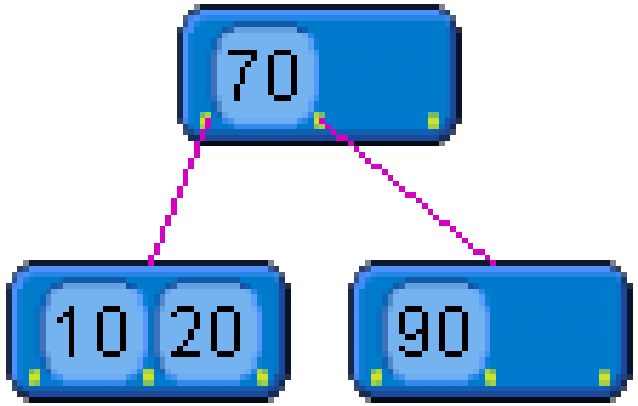
- 20,10,90,40,30,80,70



Çok Yollu Ağaçlar -B-Trees

- Uygulama 2: Ağaçtan sırasıyla 30,80,40 düğümlerini siliniz





Çok Yollu Ağaçlar: B#-Trees

- B-tree'lerde bir node dolunca bölme işlemi yapılmaktadır
  Bu bölünmeler node larda doluluk oranı %50 olmaktadır.
- B# Tree de bu bölünme işlemi geciktirilerek node lardaki doluluk oranı arttırılmıştır. Ortalama Insert süresi uzar ve ğacın yüksekliği daha azdır. Tüm ağacın packing faktor değeri B-tree'lere göre daha yüksektir
  Bölünmede orta kaydın seçilmesinde ve diğer blokların kayıt dağılımını belirlemede aşağıdaki formül kullanılır

$$M_k = \left[(r+1)/n \right]$$

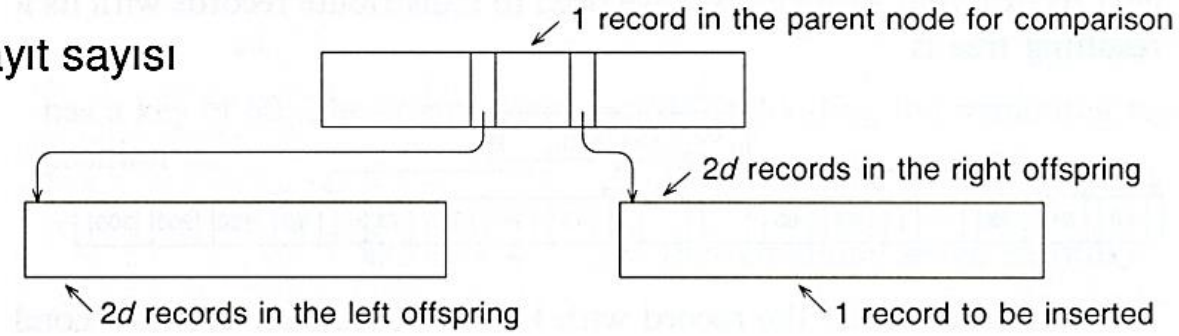
  - r: dağıtılacak toplam kayıt sayısı, n: yaprak node sayısı
  - B#- tree' lerde erişim performansı daha yüksektir. Literatürde B*-tree şeklinde variantları vardır.

Çok Yollu Ağaçlar: B#-Trees

- Tanım: Bir node üzerinde bulunabilecek toplam anahtar ve pointer aralığı;

$$\frac{4d-2}{3} \le \text{keys} \le 2d \qquad \frac{4d+1}{3} \le \text{pointers} \le 2d+1$$

Toplam kayıt sayısı
$4d + 2$



Node'lara dağıtılacak en az kayıt sayısı

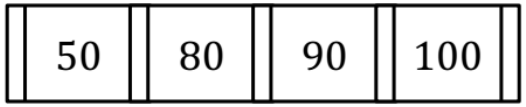
$$\left\lceil \frac{4d}{3} \right\rceil$$

Minimum doluluk oranı

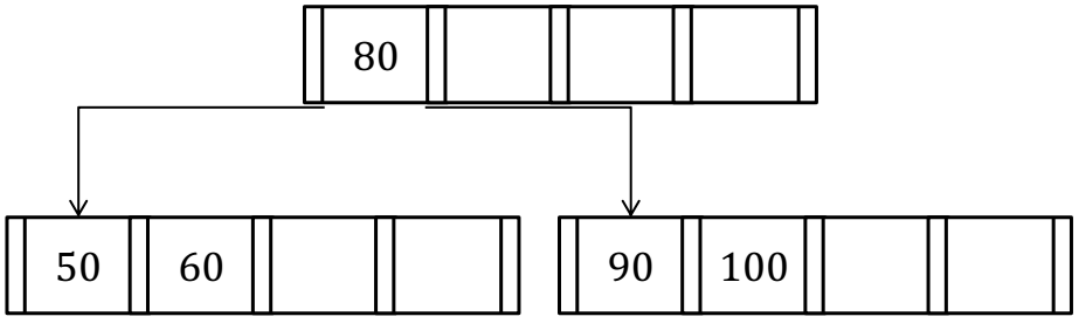
$$\frac{(4d-2)/3}{2d} \rightarrow \frac{4d-2}{6d} \rightarrow 2/3$$

Çok Yollu Ağaçlar: B#-Trees

- *d = 2* (capacity order)
  Anahtarlar= 80, 50, 100, 90, 60, 65, 70, 75, 55, 64, 78, 200, 300, 150
  - 80, 50, 100, 90 anahtarlı kayıtların eklenmesi

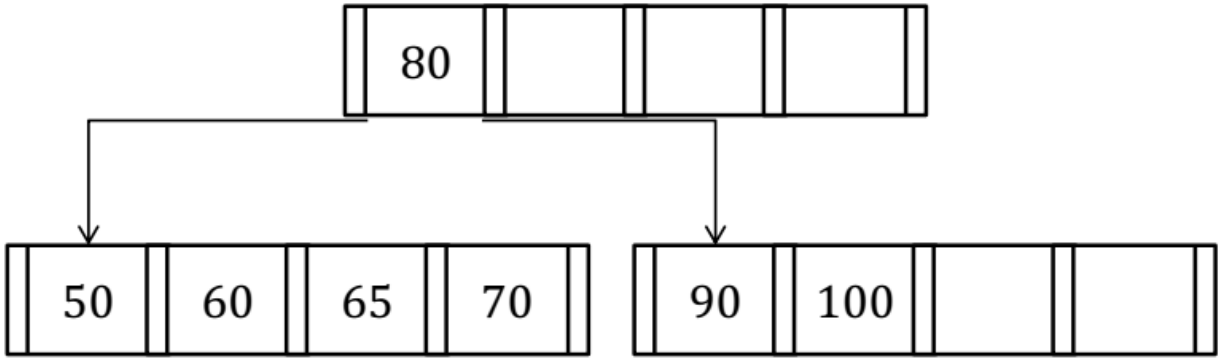


  - 60 anahtarlı kaydın eklenmesi



Çok Yollu Ağaçlar: B#-Trees

- 65 ve 70 anahtarlı kayıtların eklenmesi



- 75 anahtarlı kaydın eklenmesi

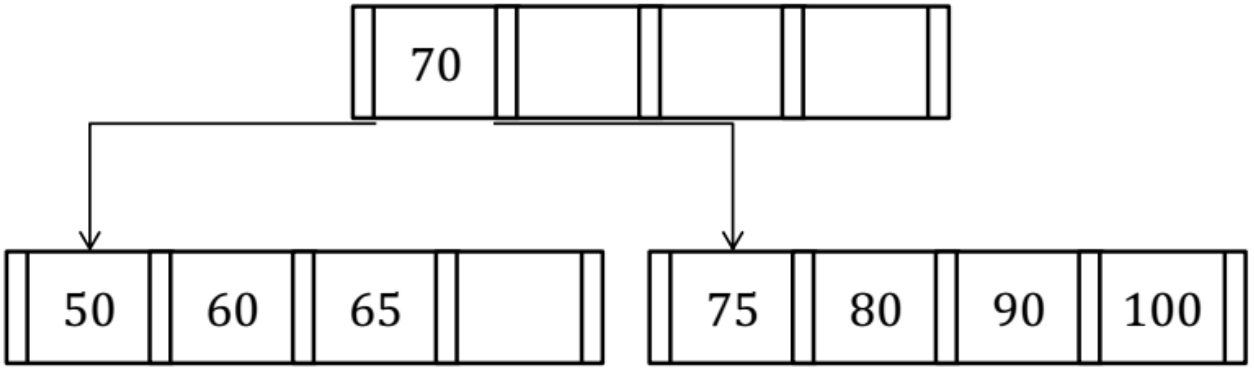
Bu kaydın eklenmesi taşmaya neden olur. Sağ kardeş node'da boşluk vardır. Bu durumda node bölünmesi yerine kayıtların bu nodelar arasında dağıtımı tekrar yapılır.

Çok Yollu Ağaçlar: B#-Trees

- Kök node da bulunacak mukayese kaydı bulunur.

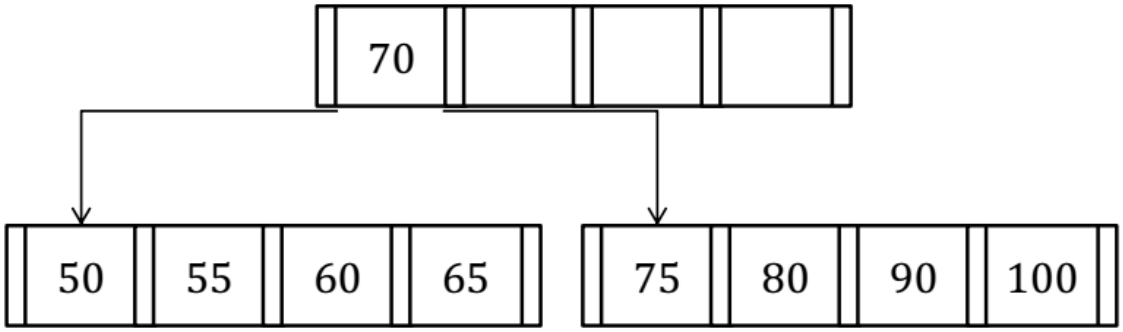
$$M_k = [(r+1)/n] = [(8+1)/2] = 4$$

- 4. pozisyondaki kayıt 70 anahtarlı kayıttır.



Çok Yollu Ağaçlar: B#-Trees

- 55 anahtarlı kaydın eklenmesi



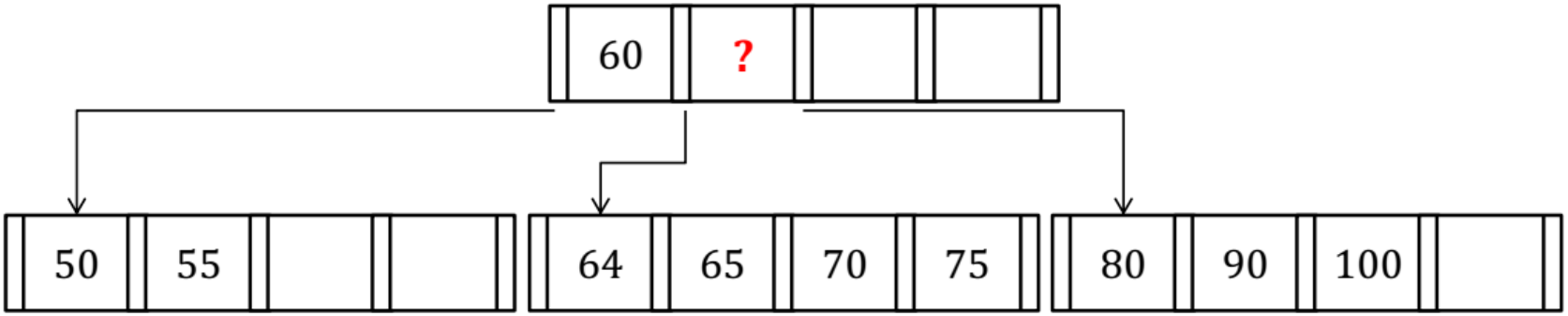
- 64 anahtarlı kaydın eklenmesi

Her iki node da dolu olduğu için bölünme gerektirir. İlk mukayese kaydı hesaplanır:

$$M_k = \left[(r+1)/n\right] = \left[(10+1)/3\right] = 3$$

3. posizyondaki kayıt 60 anahtarlı kayıttır.

Çok Yollu Ağaçlar: B#-Trees

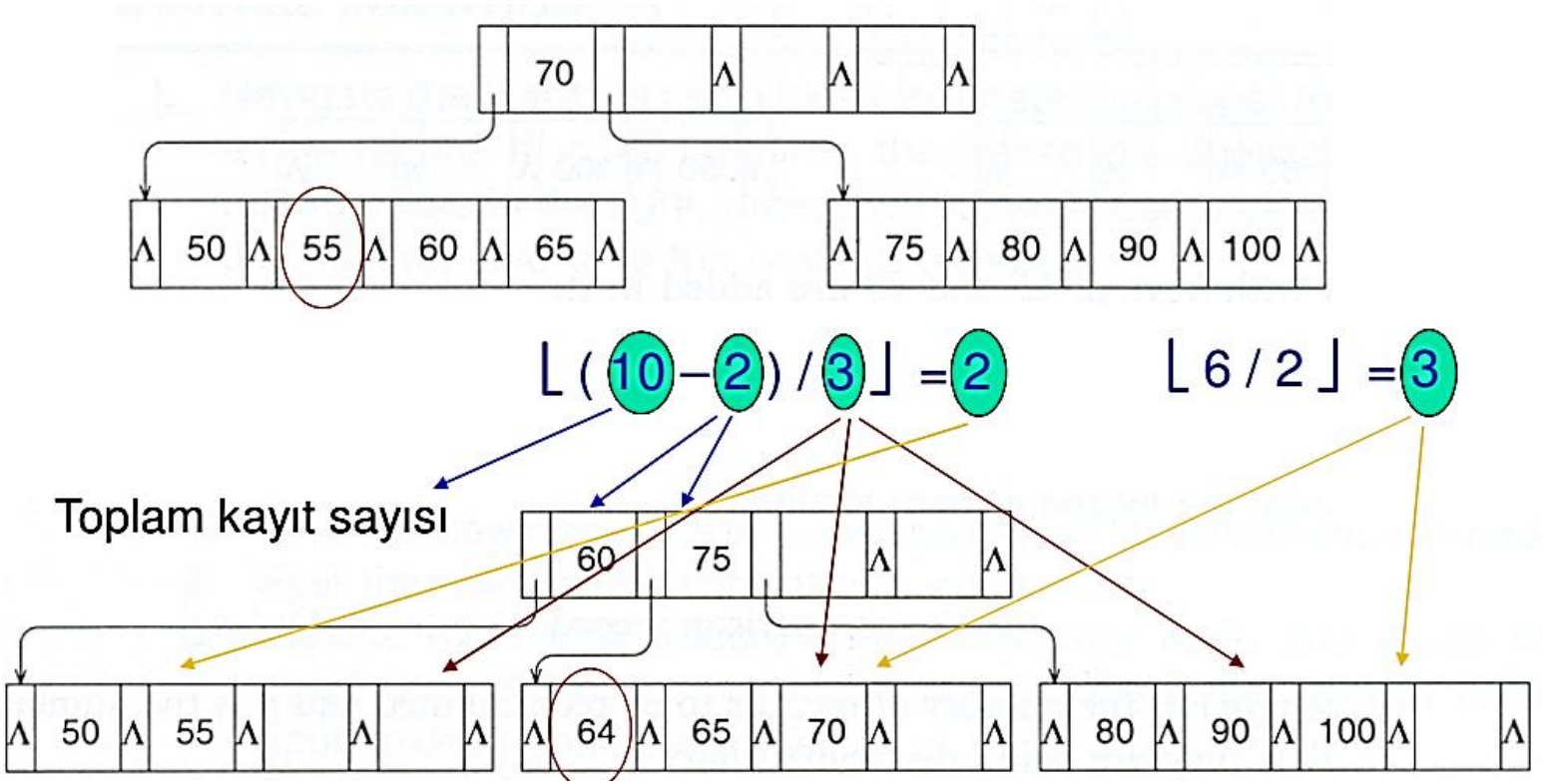


- Kalan kayıtları bölmek için ikinci mukayese kaydı hesaplanır:

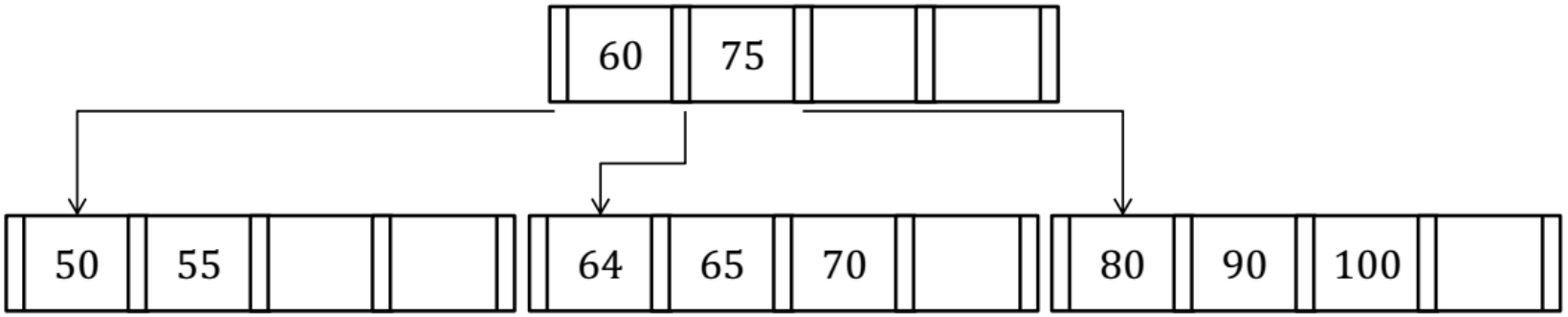
$$M_k = \left[(r+1)/n\right] = \left[(7+1)/2\right] = 4$$

- Kalan 7 kayıt içinde 4. pozisyondaki kayıt 75 anahtarlı kayıttır.

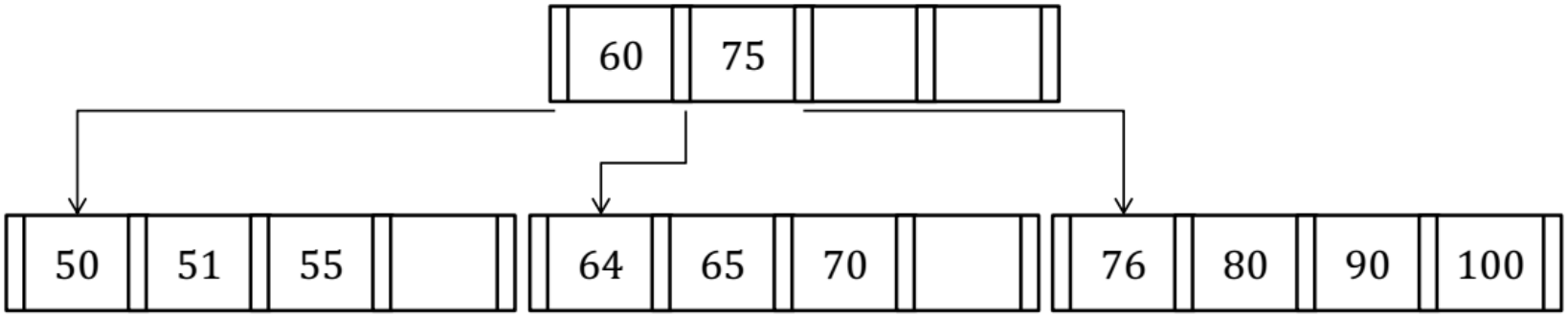
Çok Yollu Ağaçlar: B#-Trees



Çok Yollu Ağaçlar: B#-Trees



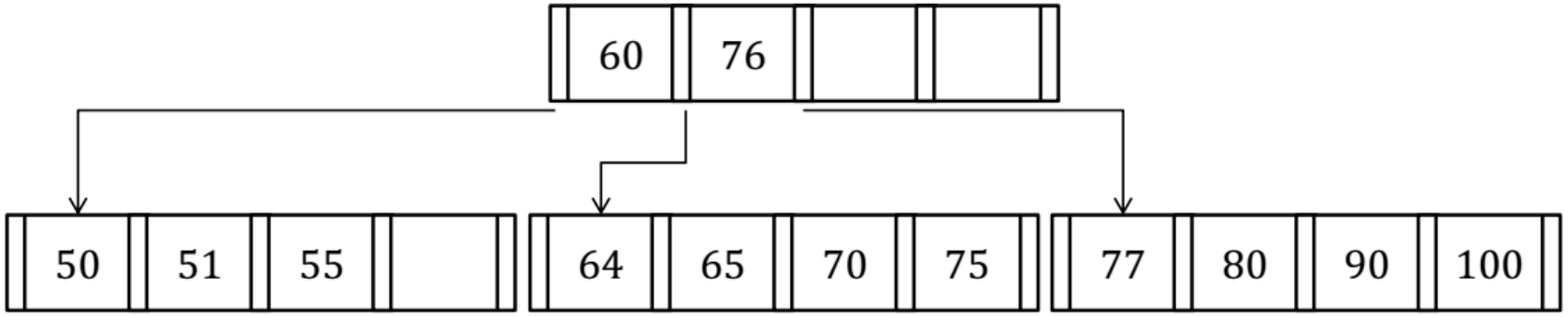
- 51 ve 76 anahtarlı kayıtların eklenmesi



Çok Yollu Ağaçlar: B#-Trees

- 77 anahtarlı kaydın eklenmesi

$$M_k = [(r+1)/n] = [(9+1)/2] = 5 \longrightarrow 76$$

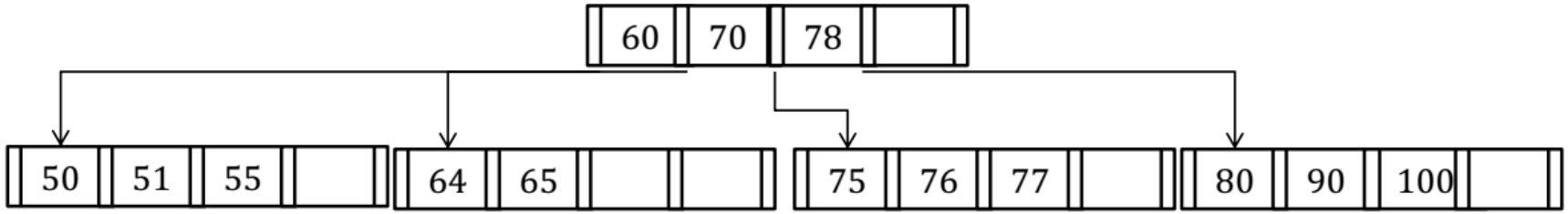


Çok Yollu Ağaçlar: B#-Trees

- 78 anahtarlı kaydın eklenmesi

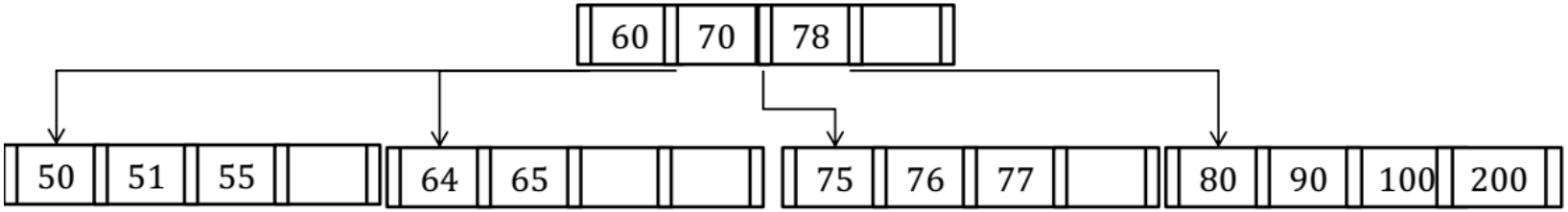
$$1.mukayese kaydı \longrightarrow M_k = [(r+1)/n] = [(10+1)/3] = 3 \longrightarrow 70$$

$$2.mukayese kaydı \longrightarrow M_k = [(r+1)/n] = [(7+1)/2] = 4 \longrightarrow 78$$



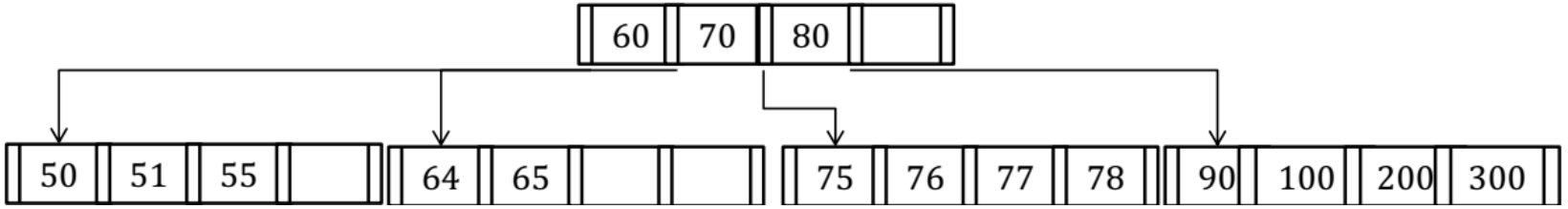
Çok Yollu Ağaçlar: B#-Trees

- 200 anahtarlı kaydın eklenmesi



- 300 anahtarlı kaydın eklenmesi

$$1.mukayase kaydı \longrightarrow M_k = [(r+1)/n] = [(9+1)/2] = 5 \longrightarrow 80$$

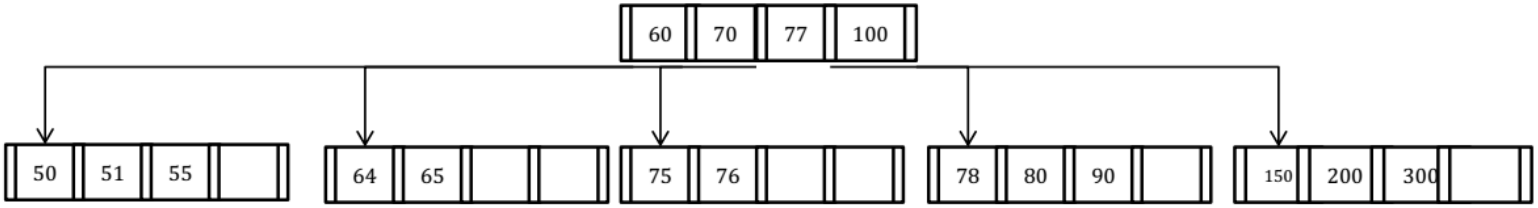


Çok Yollu Ağaçlar: B#-Trees

- 150 anahtarlı kaydın eklenmesi

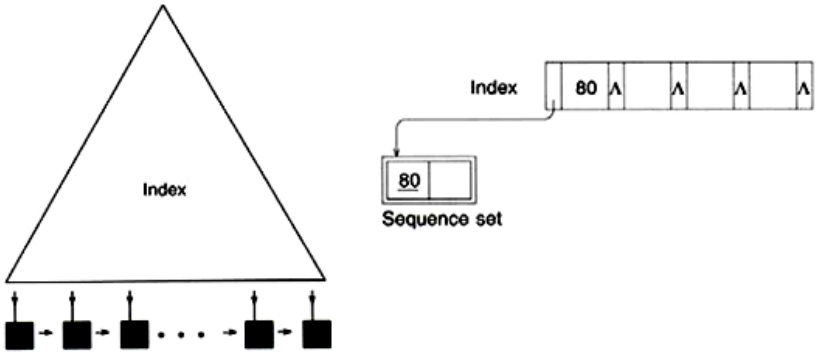
$$1.mukayasekaydı \longrightarrow M_k = [(r+1)/n] = [(10+1)/3] = 3 \longrightarrow 77$$

$$2.mukayesekaydı \longrightarrow M_k = [(r+1)/n] = [(7+1)/2] = 4 \longrightarrow 100$$



Çok Yollu Ağaçlar : B+ Trees

- B+ tree'lerde sıralı okuma için parent'a ulaşmaya gerek yoktur. Bilgiler yapraklarda bulunur.
- Yaprak olmayan düğümler (nonleaf node) sadece indeks için kullanılır. Tüm yapraklar tek bağlı listeyle bağlanır B+ tree'lerde indeks kısmı B tree ile aynıdır.
- İndeks kısmındaki tüm kısıtlamalar ve işlemler B tree ile aynıdır
- Internal node'lar index-set (index node) olarak, yapraklar ise sequence set (veri node) olarak adlandırılmaktadır.

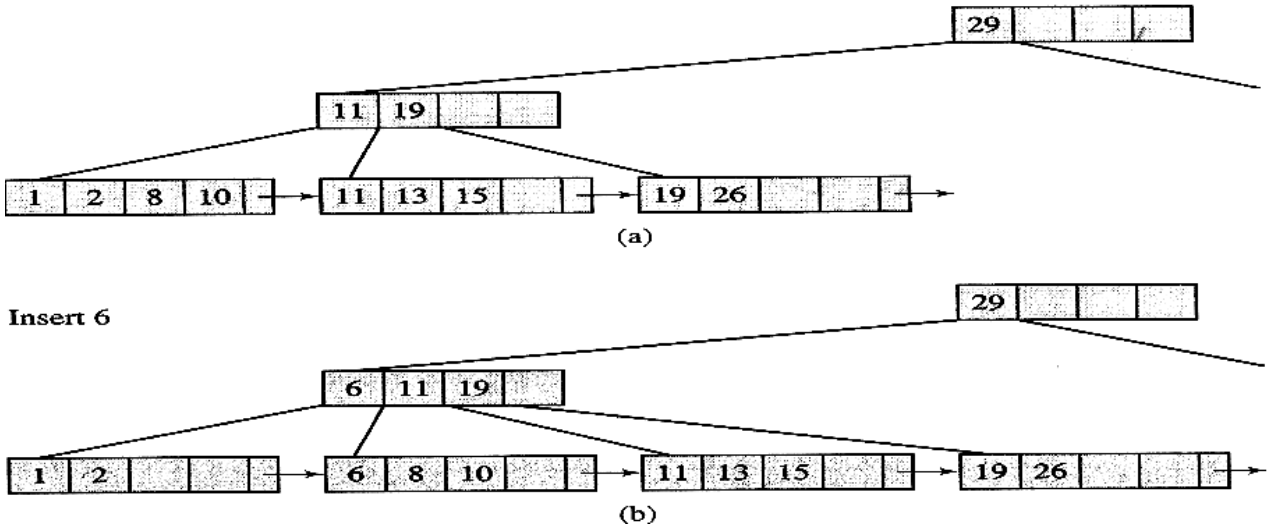


Çok Yollu Ağaçlar : B+ Trees

- İndeks node kayıt ekleme ve güncelleştirmede doğru veri node bulmak için kullanılır. Bunun için indeks node'daki anahtar değerleri ile karşılaştırma yaparak veri node'a ulaşılır.
- Eğer bir veri node dolup taşma olursa veri node bölünür ve taşan kayıt yeni oluşturulan veri node içerisinde bulunur. Bu esnada indeks node'da da gerekli düzenleme yapılır.
- Bir B+ Tree içine ilk kaydın yerleştirilmesi özel bir durumdur. Kayıt veri node içine yerleştirilir ve kaydın anahtar değeri de indeks node içine yerleştirilir.

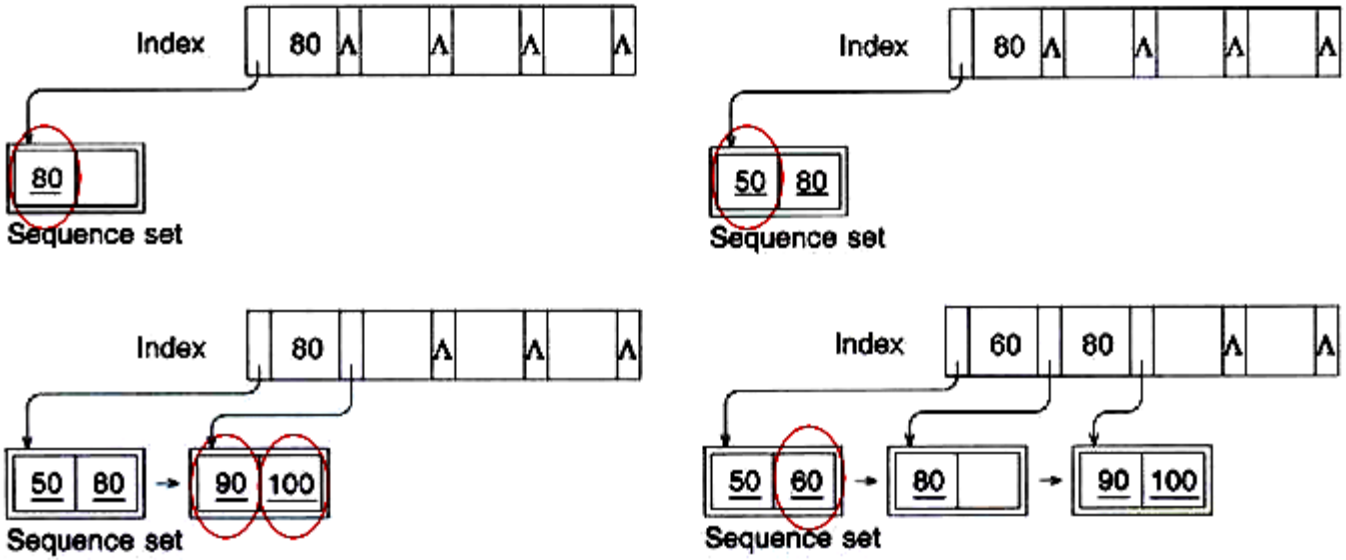
Çok Yollu Ağaçlar :B+ Trees

- Diğer bir ifadeyle;
- B Tree'de referans herhangi bir node ile yapılabilmektedir.
- B+ Tree'de ise referanas sadece yaprak node'ları ile yapılabilmektedir.



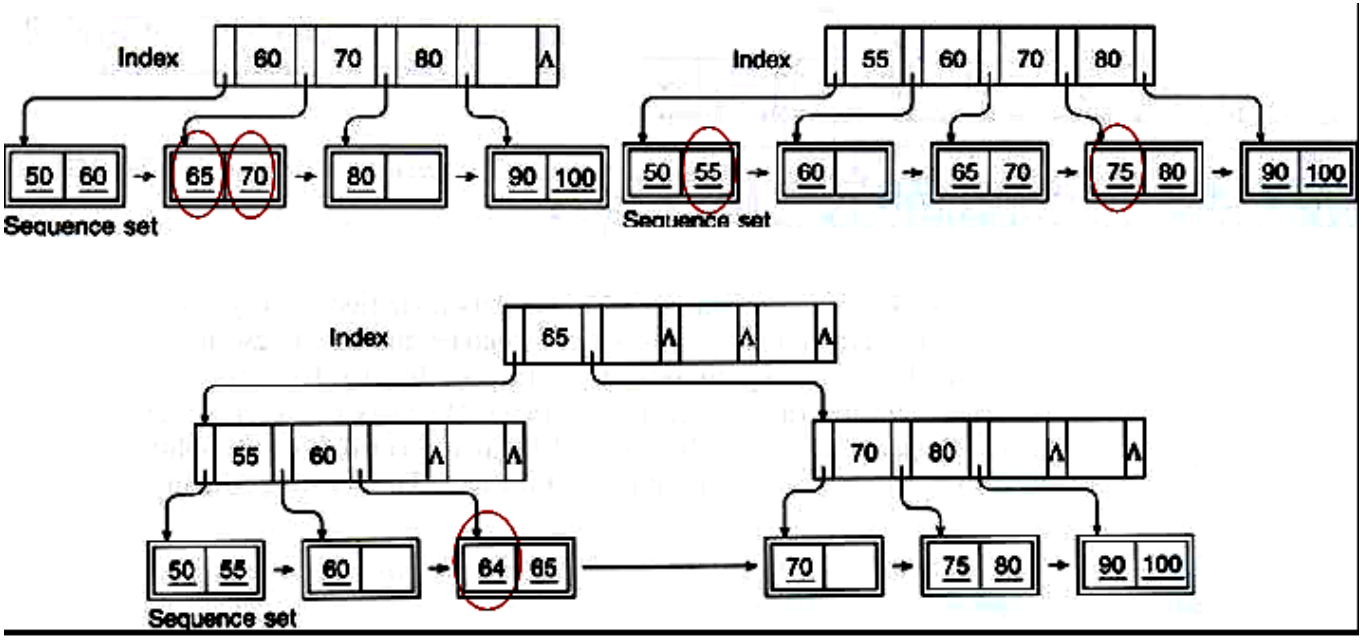
Çok Yollu Ağaçlar: B+ Trees

- **B+ Trees**
- Örnek
- *d = 2* (capacity order)
- *s = 1* (sequence set order)
- *80,50,100,90,60,65,70,75,55,64,51,76,77,78,200,300,150*



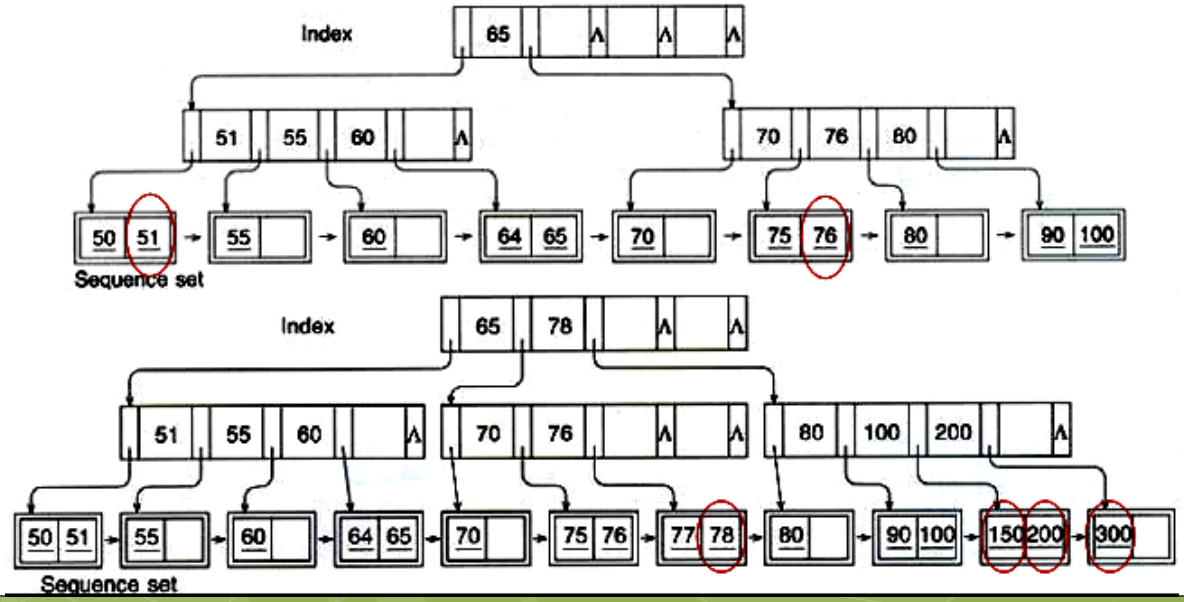
Çok Yollu Ağaçlar –B+ Trees

- **B+ Trees**
- Örnek
- *d = 2* (capacity order)
- *s = 1* (sequence set order)
- *80,50,100,90,60,65,70,75,55,64,51,76,77,78,200,300,150*



Çok Yollu Ağaçlar –B+ Trees

- **B+ Trees**
- Örnek
- *d = 2* (capacity order)
- *s = 1* (sequence set order)
- *80,50,100,90,60,65,70,75,55,64,51,76,77,78,200,300,150*



Çok Yollu Ağaçlar –B+ Trees

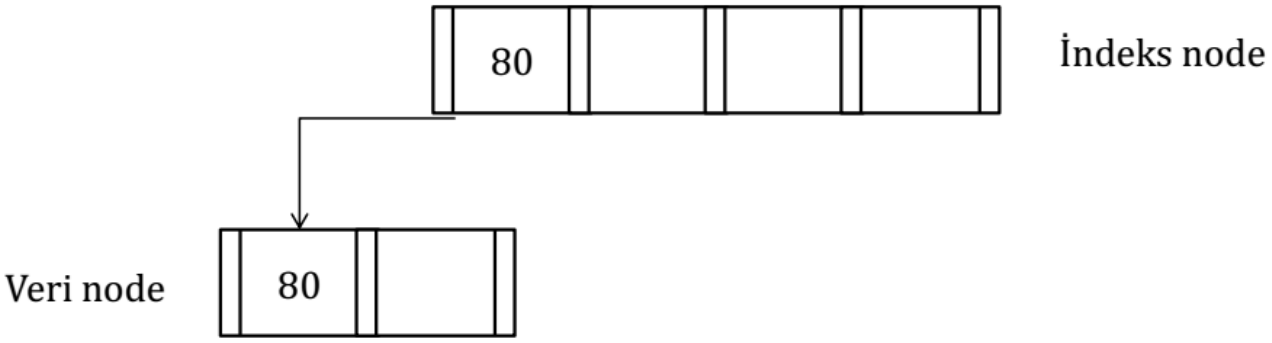
- **B+ Trees**

Örnek:

Anahtarlar= 80, 50, 100, 90, 60, 65, 70, 75, 55, 64, 51, 76, 77, 78, 200, 300, 150

İndeks node: 4 anahtar kapasiteli

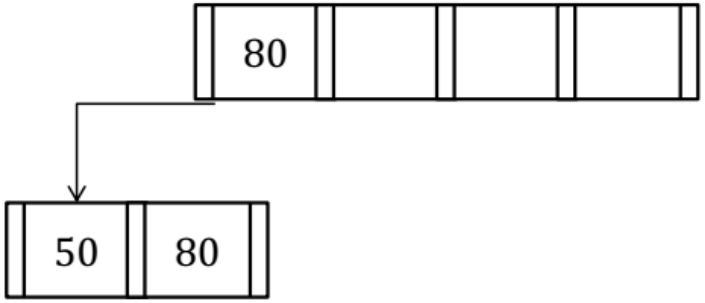
Veri node: 2 kayıt kapasiteli



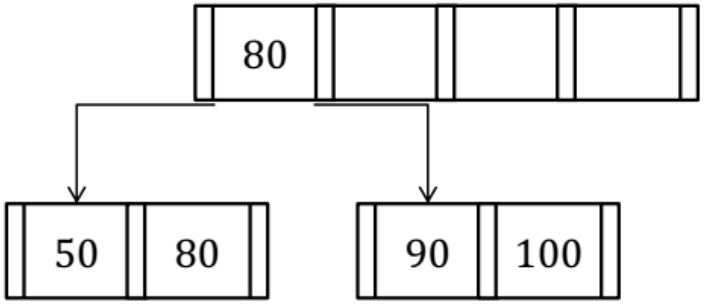
Çok Yollu Ağaçlar –B+ Trees

- **B+ Trees**

50 anahtarlı kaydın eklenmesi



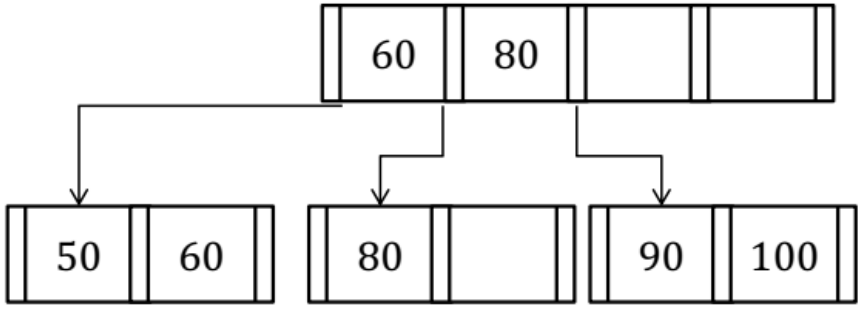
90 ve 100 anahtarlı kayıtların eklenmesi



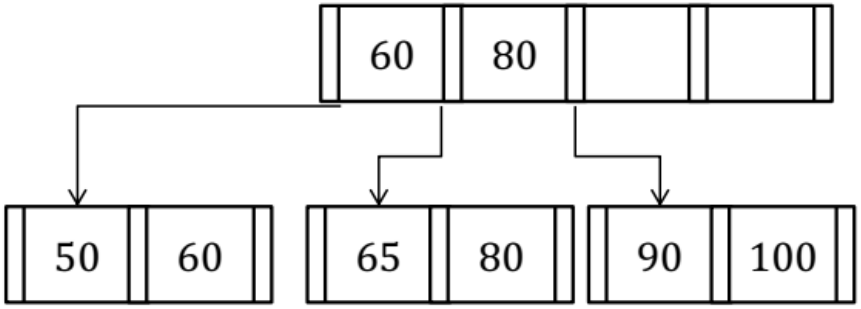
Çok Yollu Ağaçlar –B+ Trees

- **B+ Trees**

60 anahtarlı kaydın eklenmesi



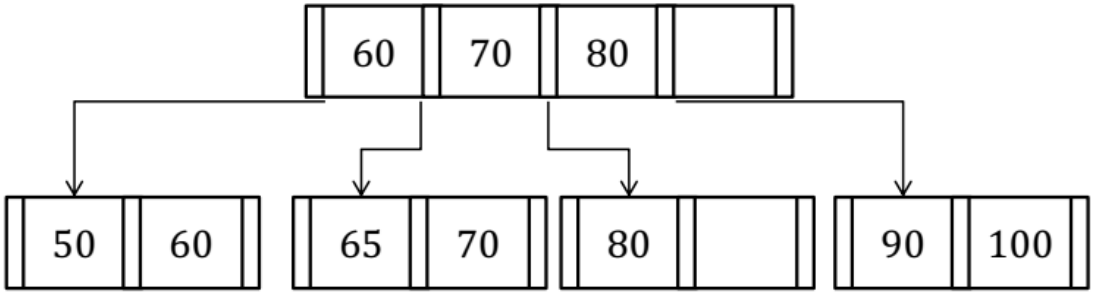
65 anahtarlı kaydın eklenmesi



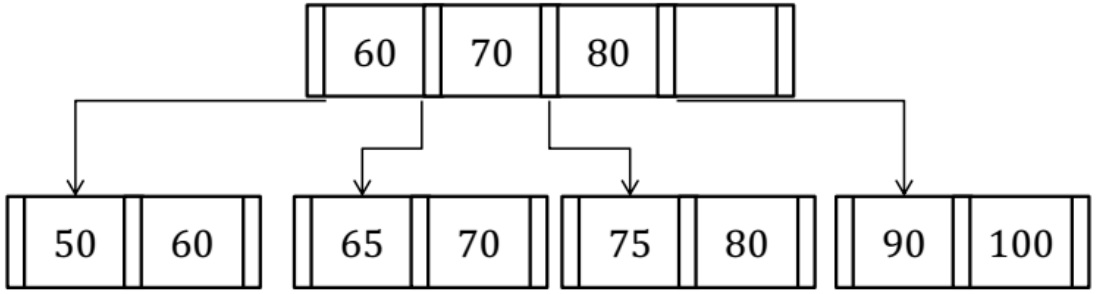
Çok Yollu Ağaçlar –B+ Trees

- **B+ Trees**

70 anahtarlı kaydın eklenmesi



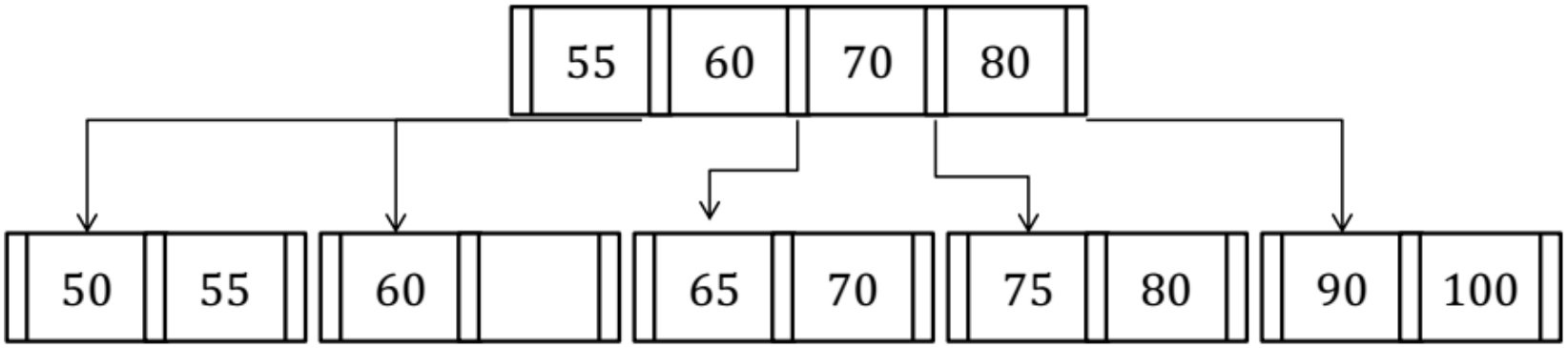
75 anahtarlı kaydın eklenmesi



Çok Yollu Ağaçlar –B+ Trees

- **B+ Trees**

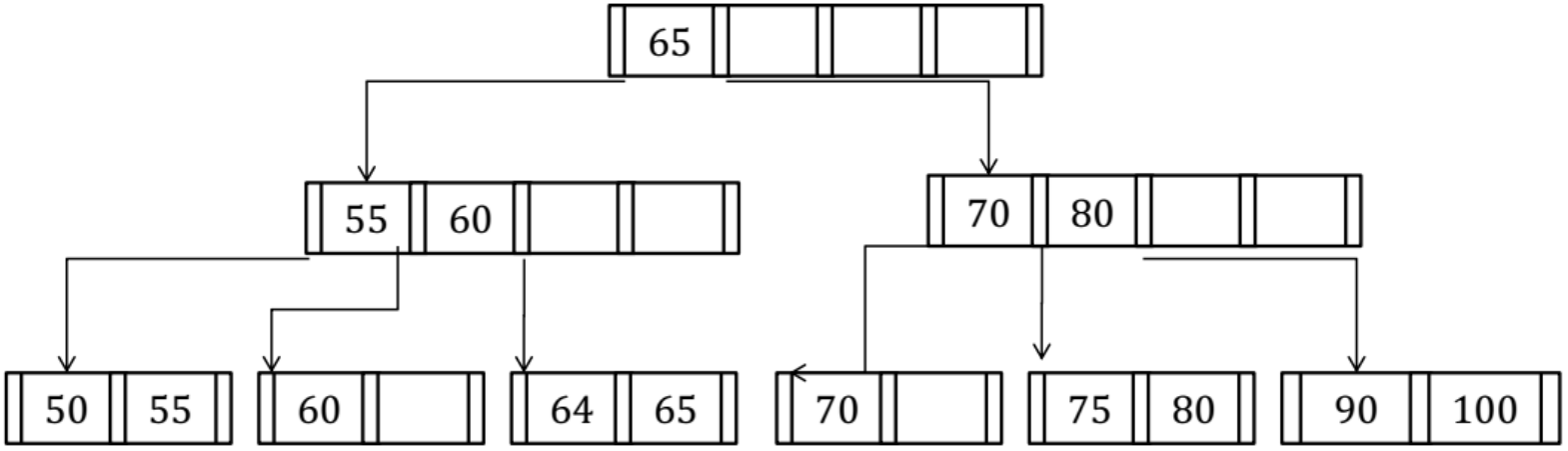
55 anahtarlı kaydın eklenmesi



Çok Yollu Ağaçlar –B+ Trees

- **B+ Trees**

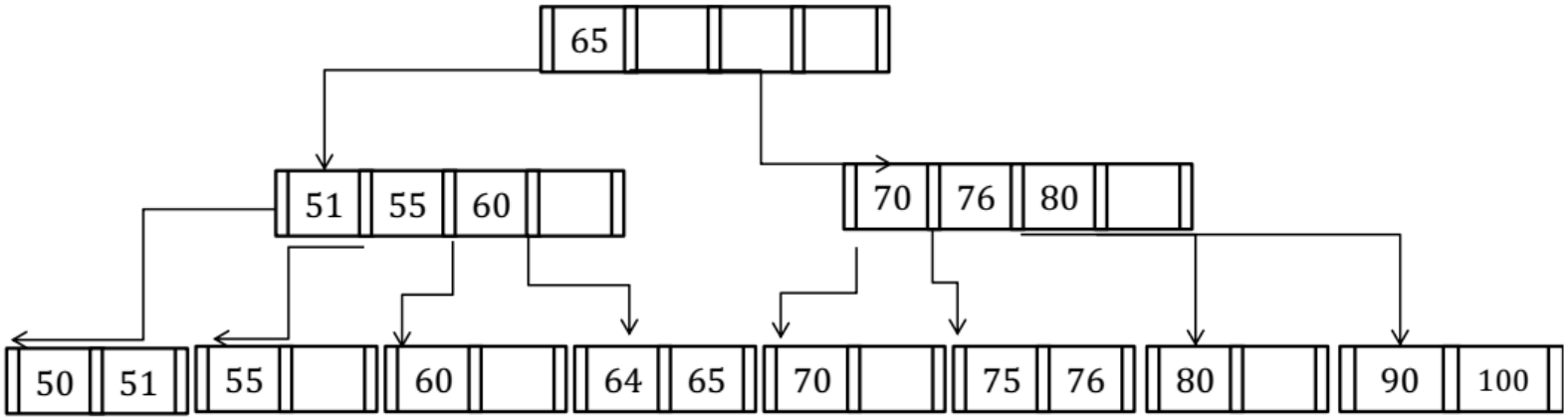
- 64 anahtarlı kaydın eklenmesi



Çok Yollu Ağaçlar –B+ Trees

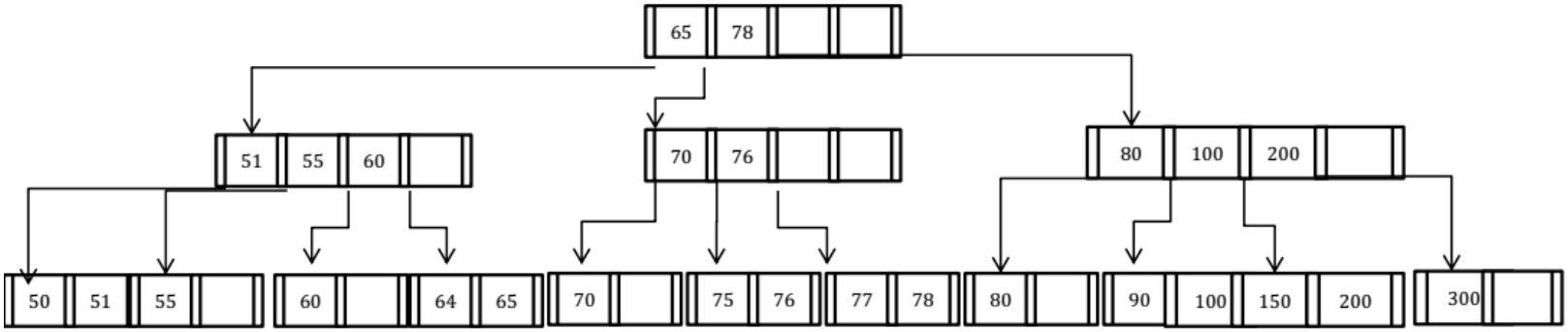
- **B+ Trees**

- 51 ve 76 anahtarlı kayıtların eklenmesi



Çok Yollu Ağaçlar –B+ Trees

- **B+ Trees**
  - 77, 78, 200, 300 ve 150 anahtarlı kayıtların eklenmesi



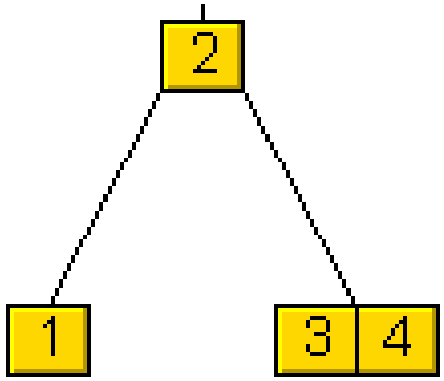
Çok Yollu Ağaçlar –B+ Trees

- Haftalık Ödev
- B,B*,B+ ağaçlarının kullanıldığı yerler hakkında araştırma yapınız. Literatür taraması yaparak elde ettiğiniz makaleleri inceleyiniz. Kullanıldığı yerlerde ne amaçla kullanıldığına yönelik bilgileri içeren bir rapor hazırlayınız .
- B*- tree hakkında bilgi toplayarak ekleme işlemi nasıl gerçekleştirilir araştırınız ve bir rapor hazırlayınız .
- B+ ve B* ve B# - tree'lerde silme işleminin nasıl yapıldığını araştırınız ve bir rapor hazırlayınız .
- Bunlara ait program örneğini ve algoritmasını gerçekleştiriniz

2-3-4 tree

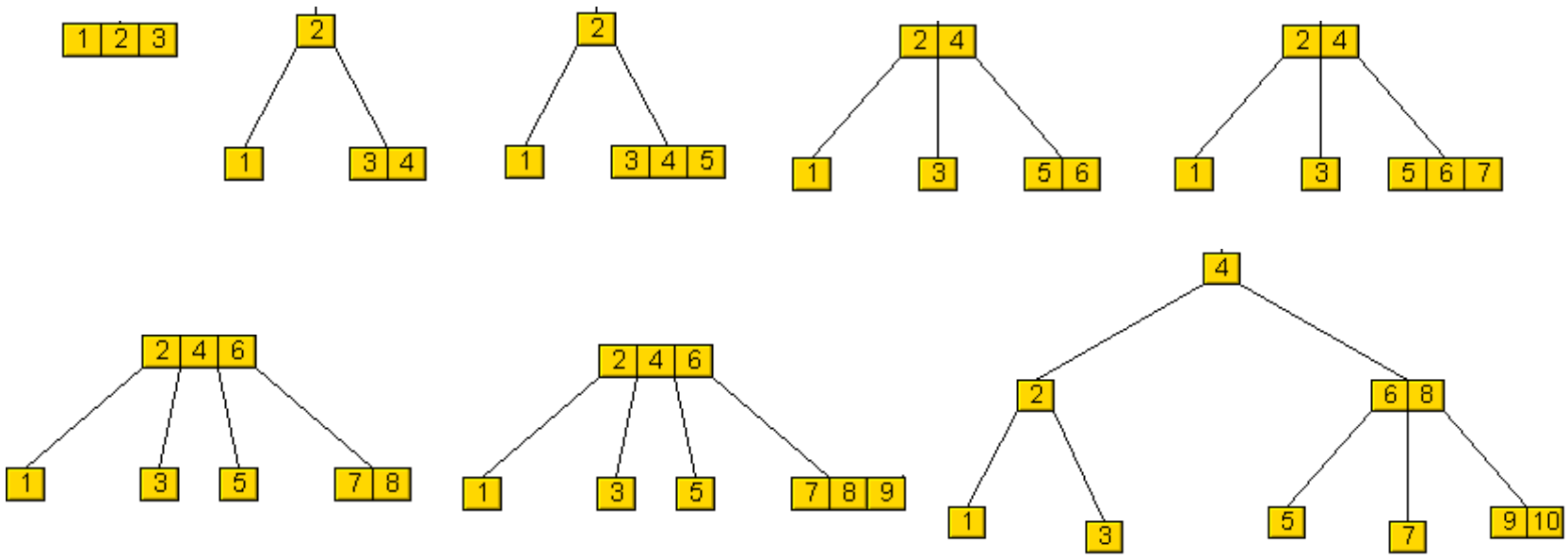
- B-tree yapısının basit bir halidir. Daha önce verilen kurallar bu ağaç içinde aynıdır.
- 2-3-4 tree anlamı 2 düğüm var ise 3 çocuğu olur. 3 düğüm var ise 4 çocuğu bulunur.
- Ekleme işleminde eğer maksimum kapasite aşılırsa eklenmeden önceki ortada bulunan düğüm kök alınır ekleme işlemi sonra yapılır.
- 1,2,3,4





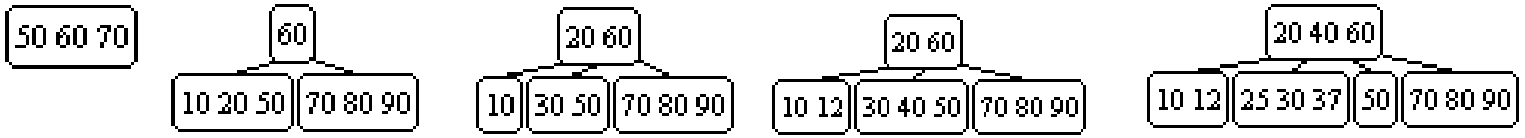
2-3-4 tree

- Örnek:1,2,3,4,5,6,7,8,9,10

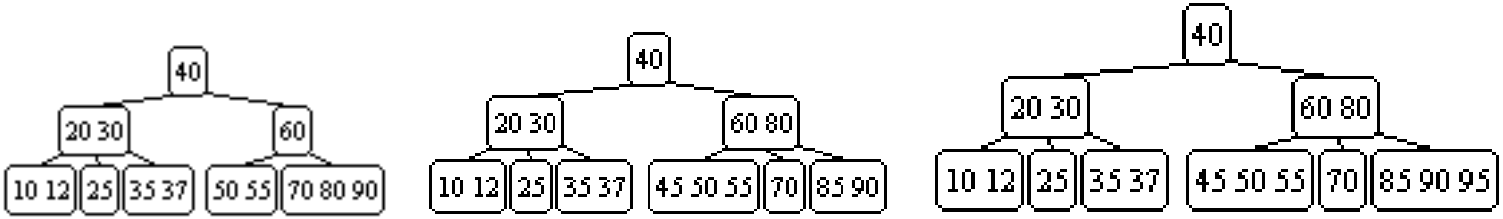


2-3-4 tree

- Örnek:,3 düğüm: Ekle: 50,70,60,90,20,10,80,30,40,12,25,37



- Ekle:55,35,45,85,95

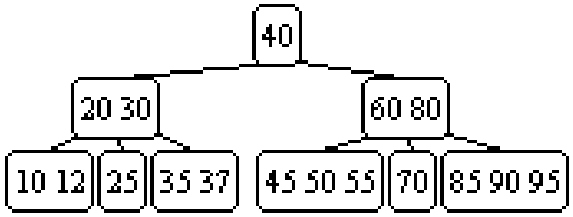


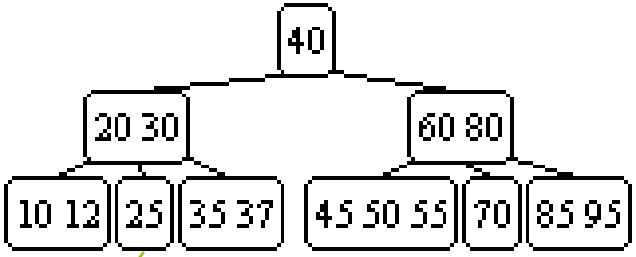
2-3-4 tree Silme

- Silme işlemlerinde düğümlere ait çocuk sayılarında denge bozulmuyorsa sadece kurala göre döner.(Soldaki en büyük veya sağdaki en küçük düğüm alınarak.)
- Sol veya sağa göre işlem yapıldığında öncelik çocuk sayısı fazla olandadır.
- Silme işleminde denge bozuluyorsa silinen düğümün kardeşi ve ebeveyni düzenlemeye girer.
- Çocukları olan silinecek düğüm tek ise ya birleştirme yada döndürme yapılarak yanına başka bir değer getirilip daha sonra silinir.
- Silinecek çocuğun atası tek ise atasının ebeveyni ve kardeşleri düzenlemeye girer.

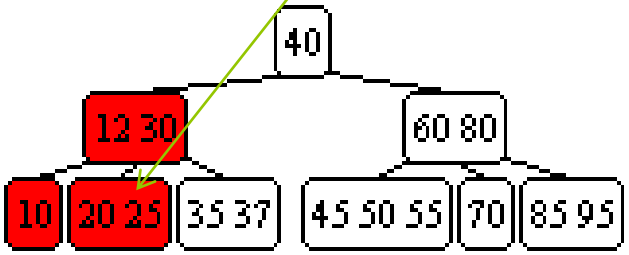
2-3-4 tree

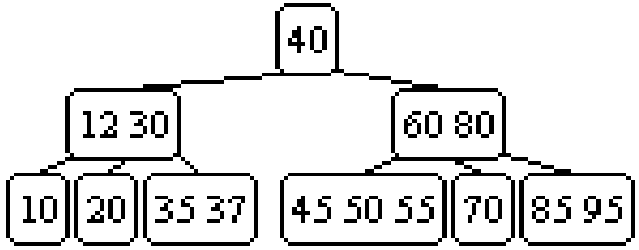
- Kural 1- Silinen düğüm yaprak ve minumum kapasitenin altına düşmüyorsa silinebilir.
- Örnek: 90 silindi





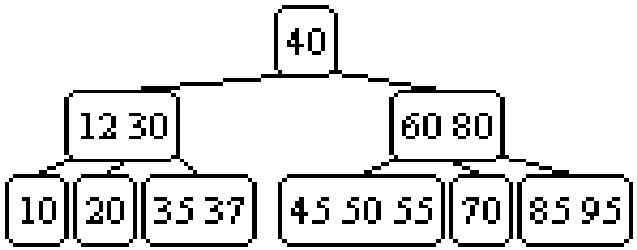
- Kural2-Silinen düğüm yaprak ve minumum kapasitenin altına düşüyor ise (öncelik sol) ebeveyn ile çocuklar yeniden düzenlenir (Soldaki en büyük veya sağdaki en küçük düğüm alınır).
- 25 silindi

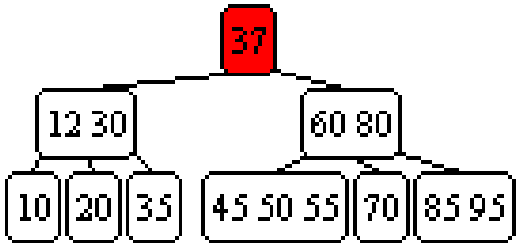




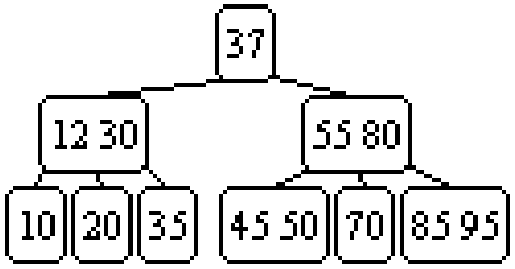
2-3-4 tree

- Kural 3- Silinen değer kök ve çocukları tek değil ise, soldaki en büyük veya sağdaki en küçük düğüm kök olur. (soldaki değer alındı, çocukları tek ise sonraki kurallar işletilir.)
- Örnek: 40 silindi



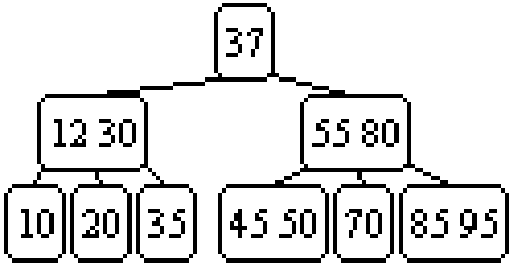


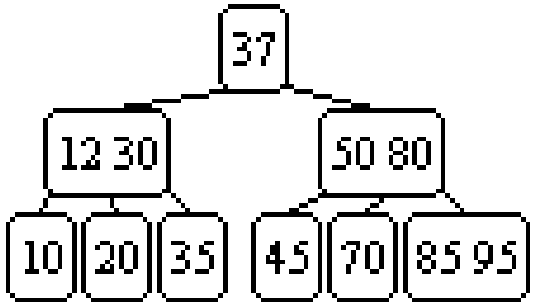
- Kural 4-Silinen düğüm çocukları olan bir düğüm ise çocukları fazla olandan değer alınır.
- 60 silindi



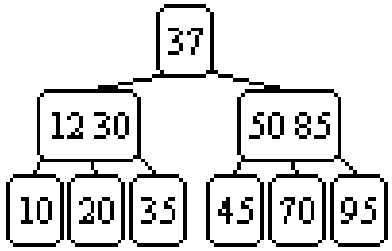
2-3-4 tree

- Örnek: 55 silindi (Kural-4)





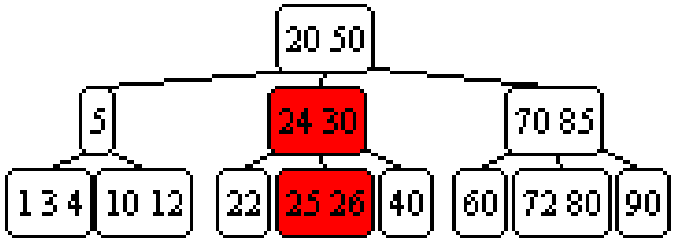
- 80 silindi (Kural-4)

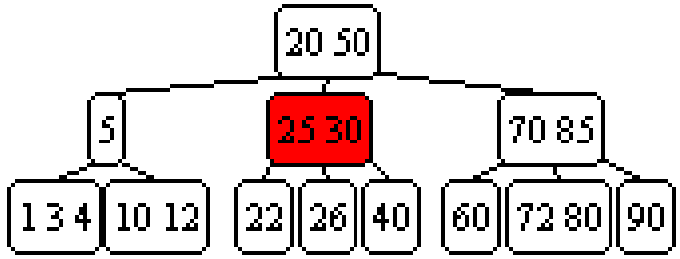


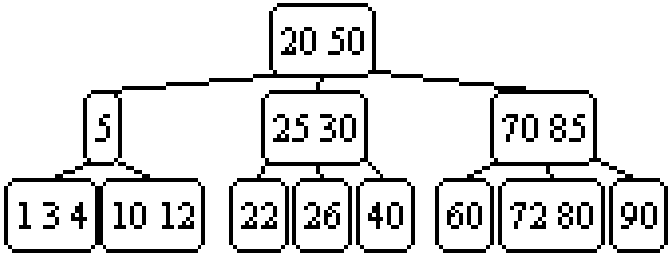
2-3-4 tree

Soldaki en büyük düğüme göre

- Örnek: 24 nolu düğüm sil (Öncelik fazla olan çocuktadır)

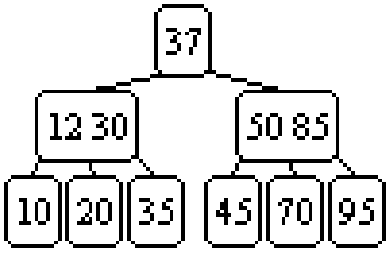


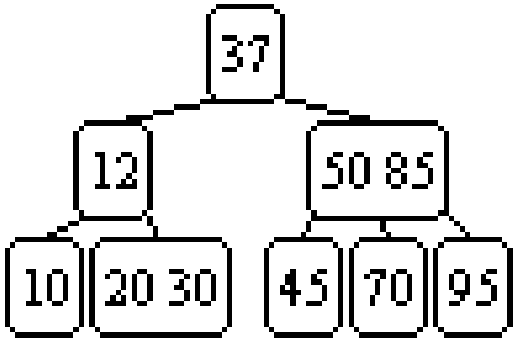




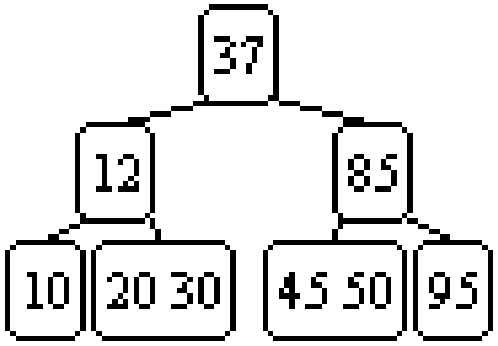
2-3-4 tree

- Kural-5: Minumum kapasitenin altına düşülürse ebeveyn ve çocuklar birleştirilir.
- Örnek: 35 silindi



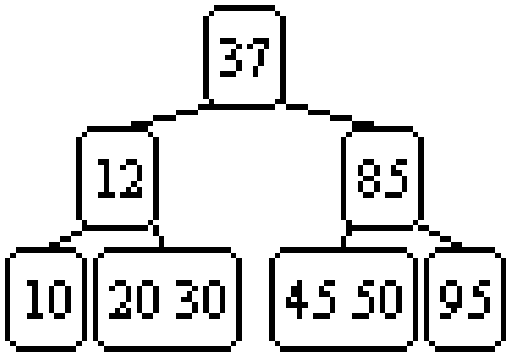


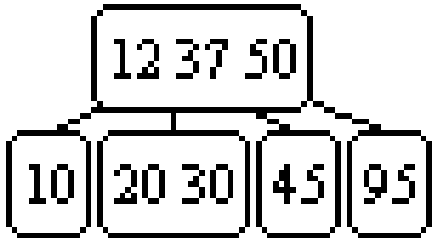
- Örnek:70 silindi



2-3-4 tree

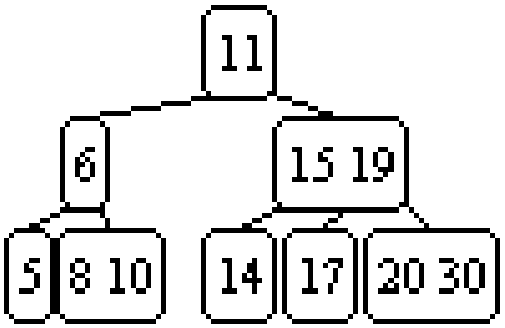
- Kural-6: (B-tree den farkı) Çocukları olan bir düğüm silinmek istenirse; eğer silinen düğüm ve kardeşi tek değere sahip ise ebeveyn, kardeş ve çocuk düğümler birleştirilir (Döndürme).
- Örnek: 85 silindi, (Kural 4,5,6) Ebeveyn ve çocuklar birleştirildi.

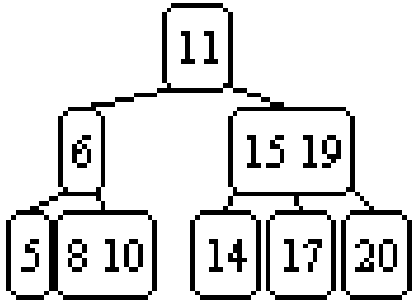


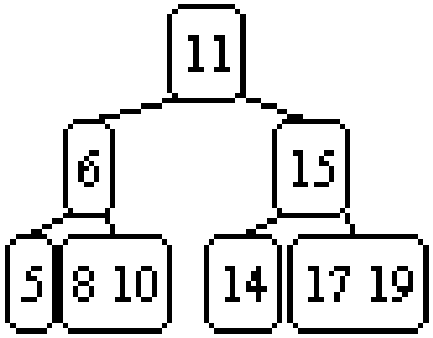


2-3-4 tree

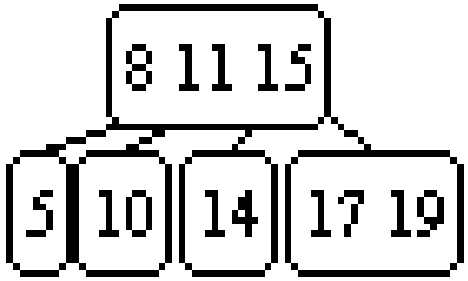
- Örnek: 30,20 silindi





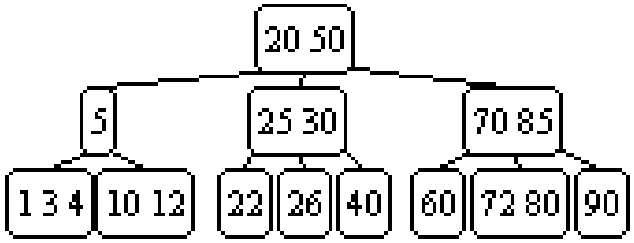


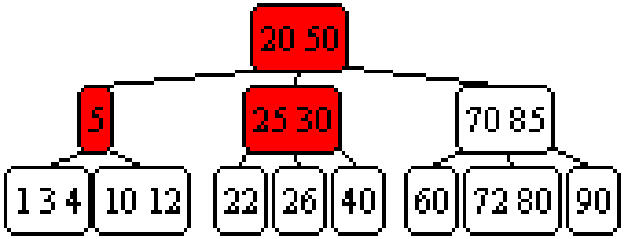
- 6 silindi



2-3- veya 2-3-4 tree

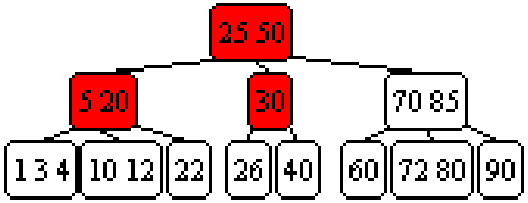
- Kural-7: (B-tree den farkı), Çocukları olan bir düğüm silinmek istenirse; Silinen düğüm tek sahip ama kardeşi tek değer değil ise ebeveyn, kardeş düğümler döndürülür ve çocuk düğüm ile birleştirilir.(Döndür ve birleştir.)
- Örnek: 5 nolu düğüm sil.
- 1-Ebeveyn ve kardeş düğümler seçildi



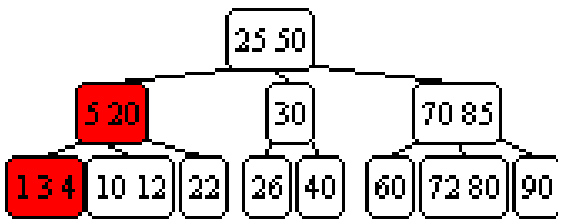


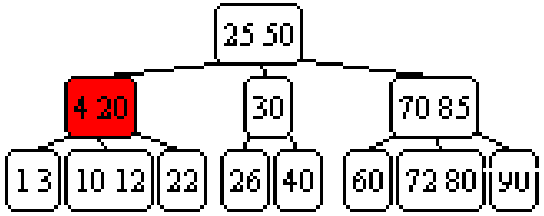
2-3- veya 2-3-4 tree

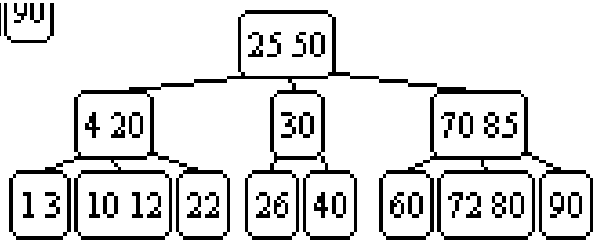
- 2-Döndürme işlemi gerçekleşerek silinecek düğüme değer getirildi.



- 3- Çocuklardan düğüm getirildi ve 5 silindi.



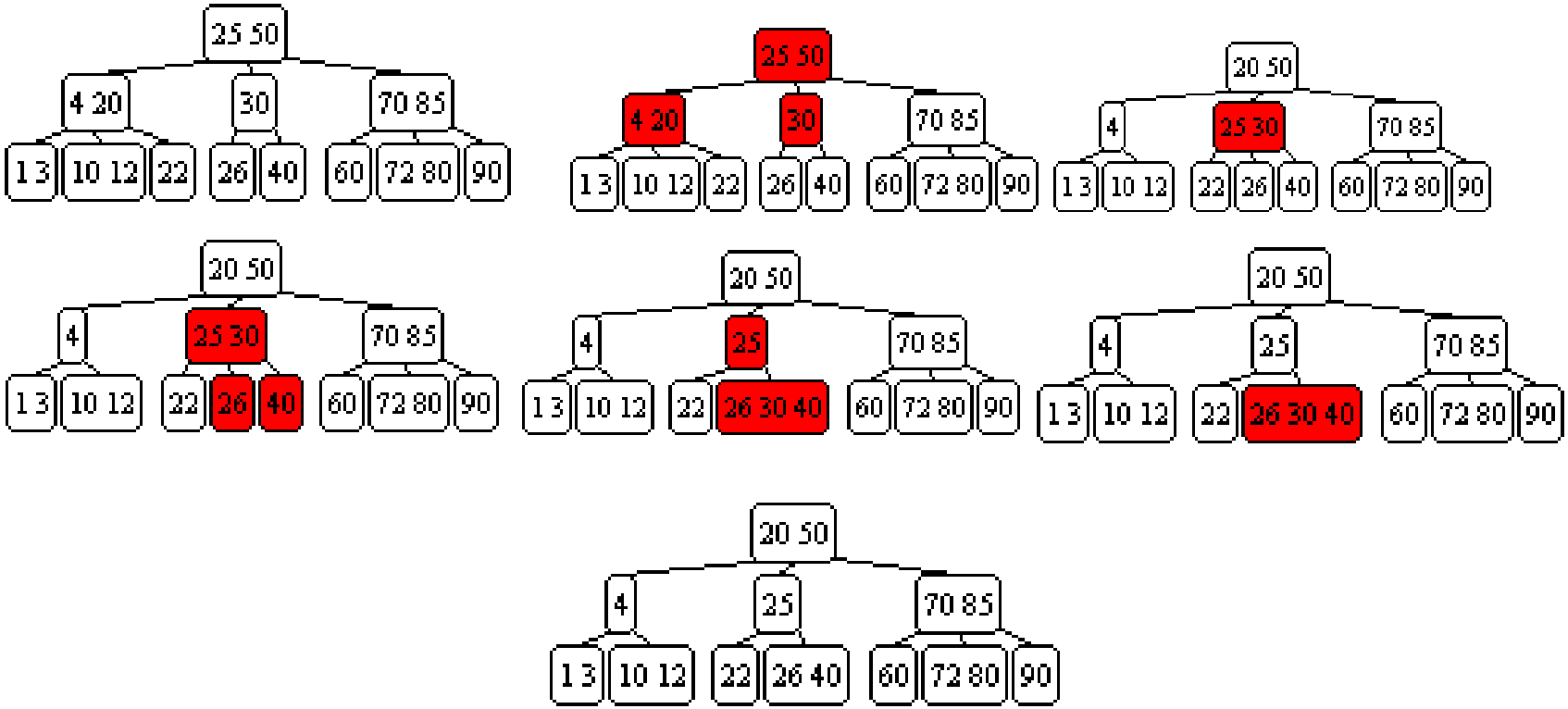




2-3- veya 2-3-4 tree

Soldaki en büyük düğüme göre

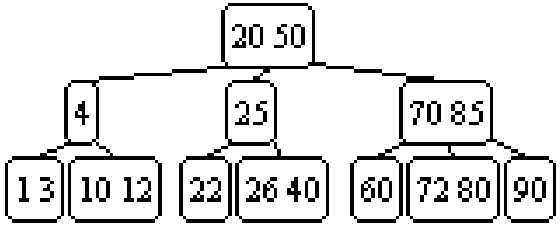
- Örnek: 30 nolu düğüm sil

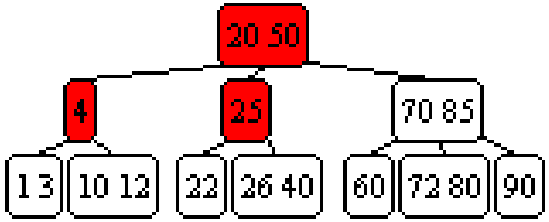


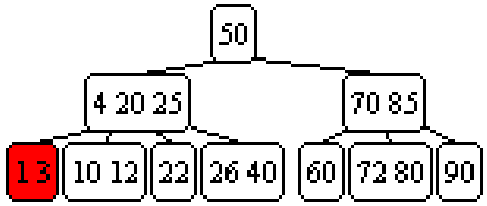
2-3- veya 2-3-4 tree

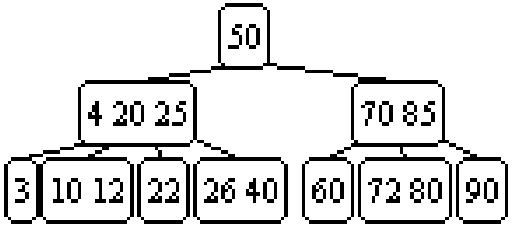
- Kural-8: B-tree den farkı, Silinen yaprak düğümün atası ve onun kardeşi tek değere sahip ise ebeveyn, kardeş ve çocuk düğümler birleştirilir.(Döndür ve birleştir.)(Kural-6 ya benzer)

- Örnek: 1 nolu düğüm sil (Kural 5,6,7)





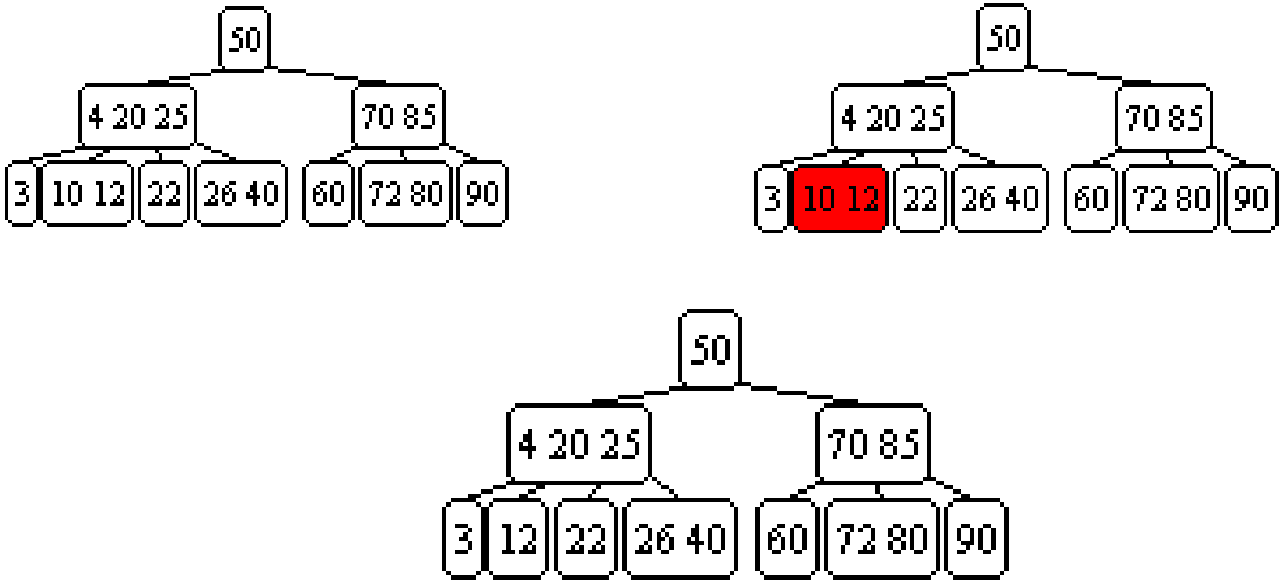




2-3- veya 2-3-4 tree

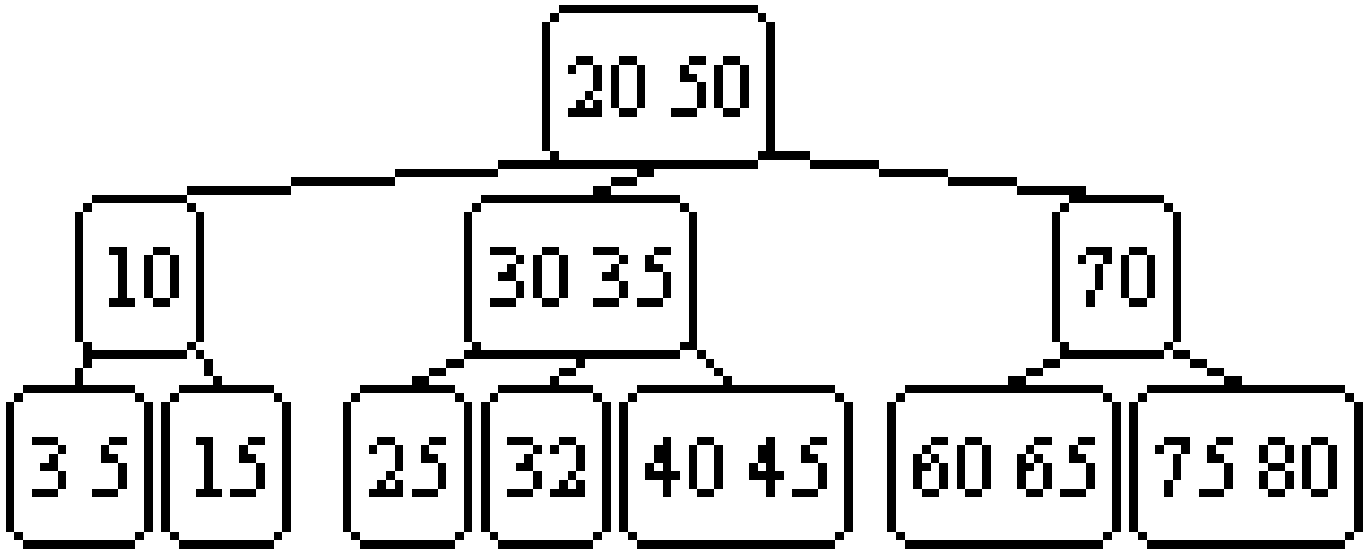
Soldaki en büyük düğüme göre

- Örnek: 10 nolu düğüm sil



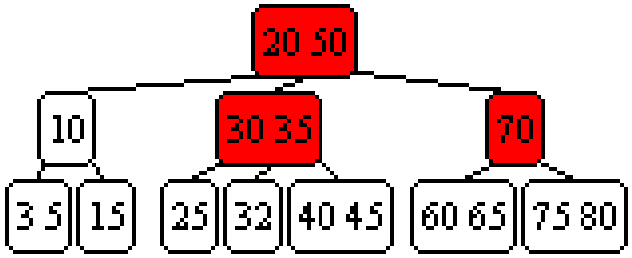
2-3- veya 2-3-4 tree

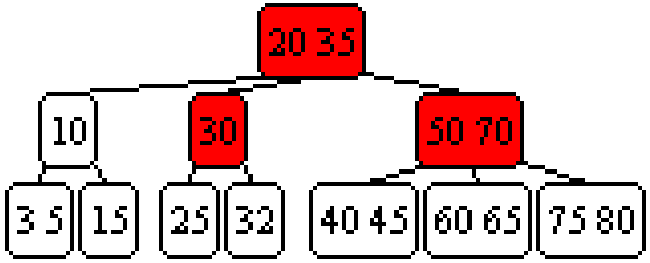
- Örnek: 70,30, 20,40,65,75,35 nolu düğümleri siliniz

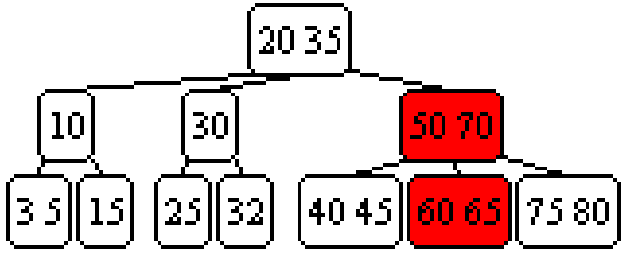


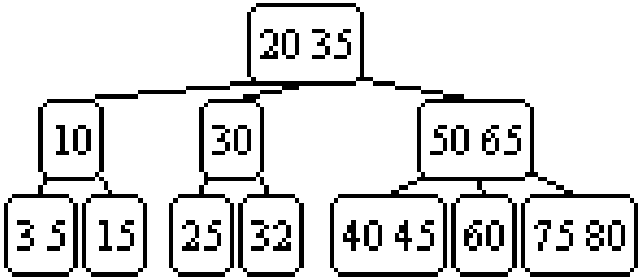
2-3- veya 2-3-4 tree

- Örnek: 70 (Kural 7)



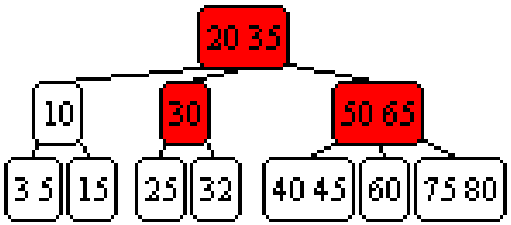


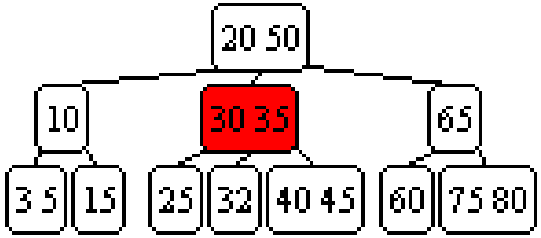


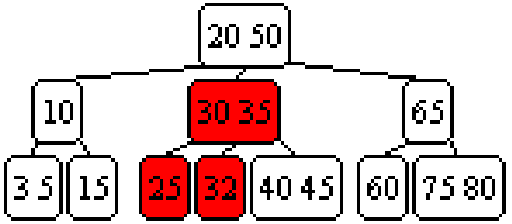


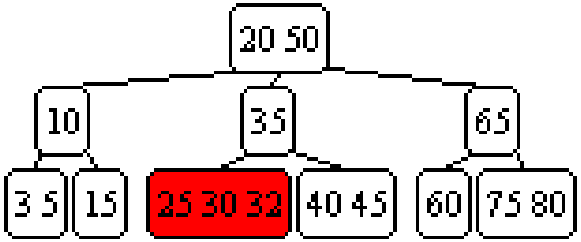
2-3- veya 2-3-4 tree

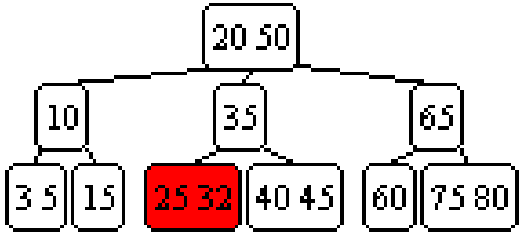
- Örnek: 30 (Kural 7)

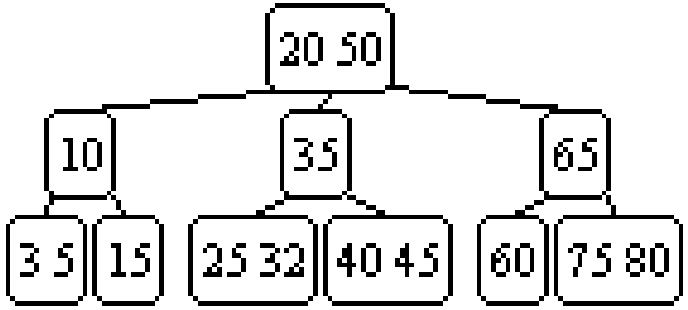






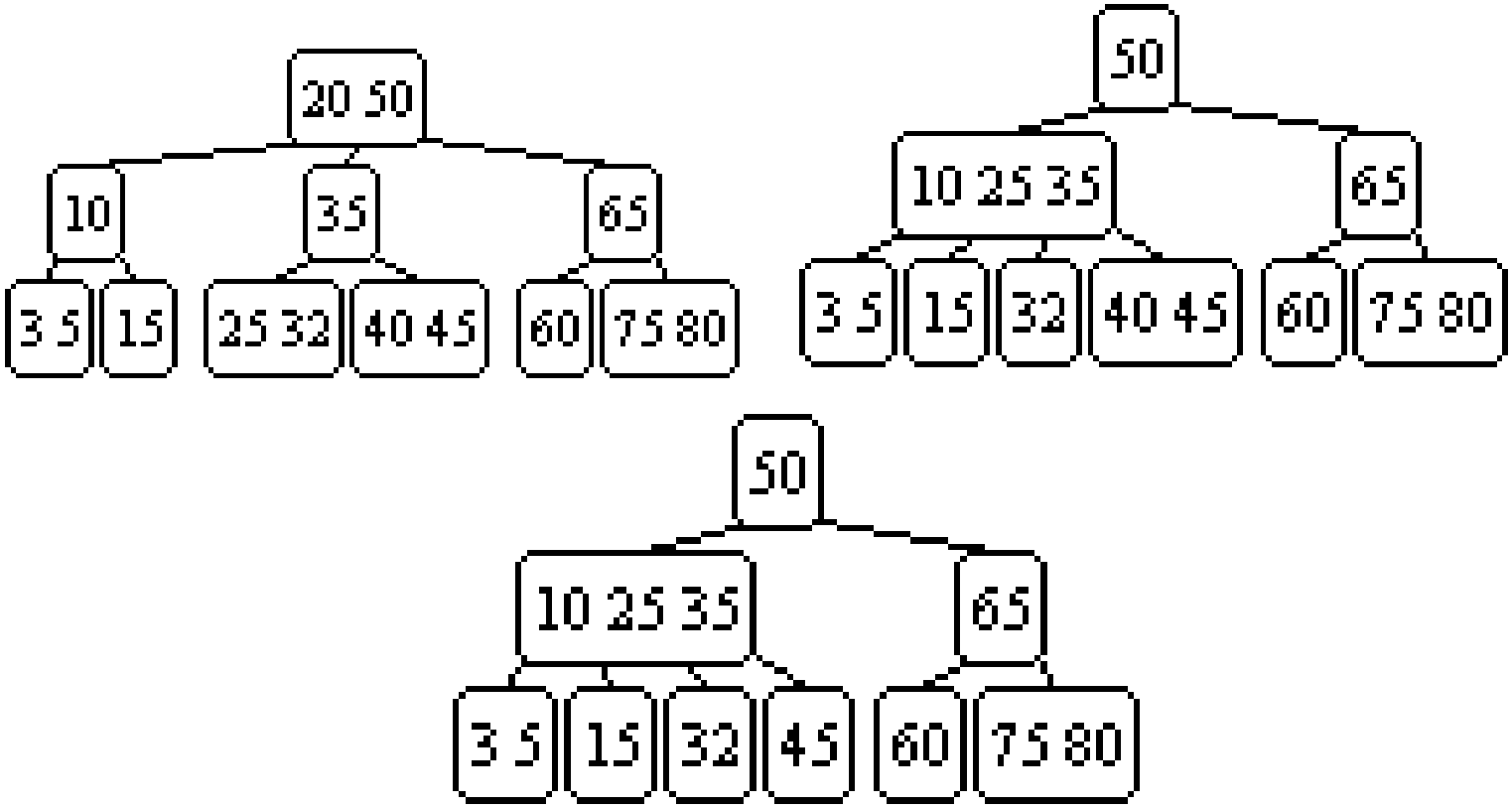






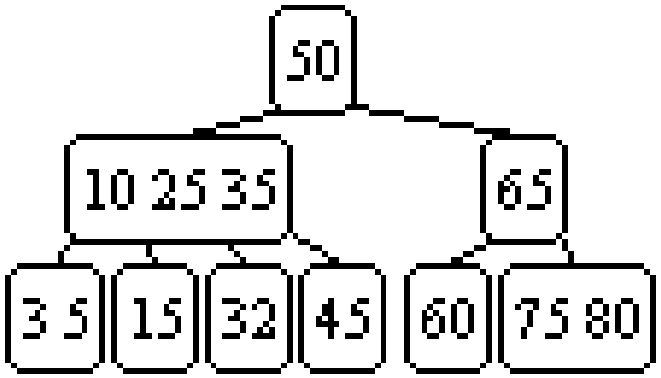
2-3- veya 2-3-4 tree

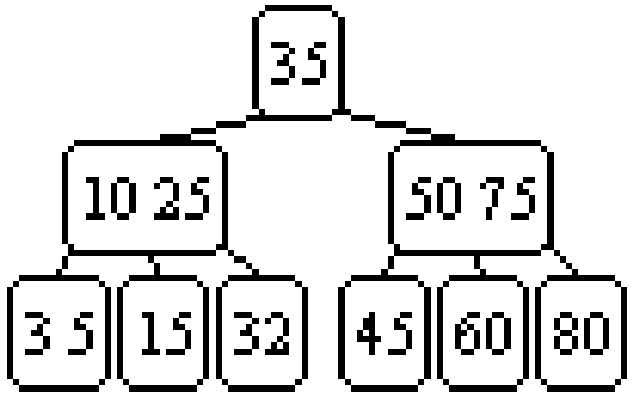
- Örnek: 20 (Kural 4-7), 40 (Kural 1)

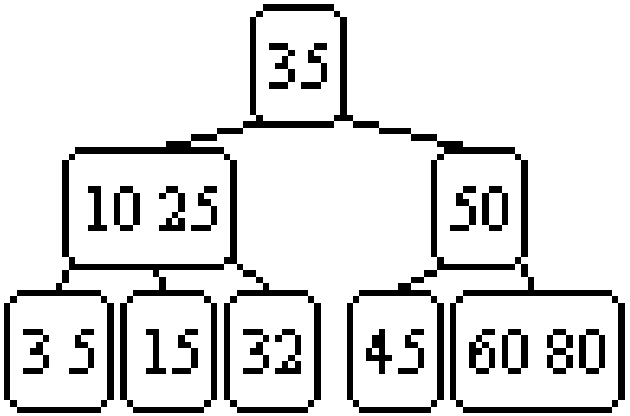


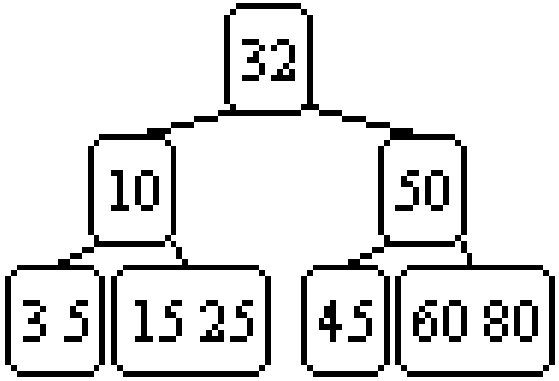
2-3- veya 2-3-4 tree

- Örnek: 65 (Kural 7),75 (Kural 5), 35 (Kural 4)



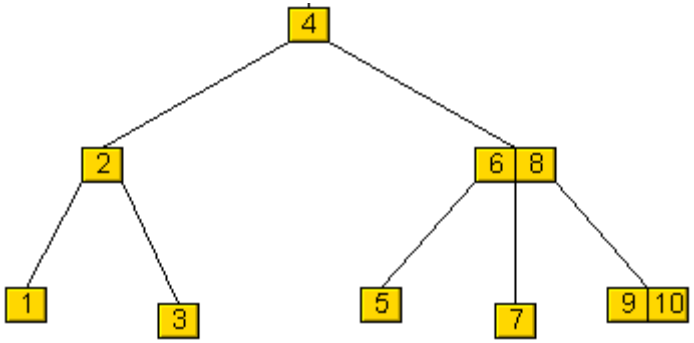


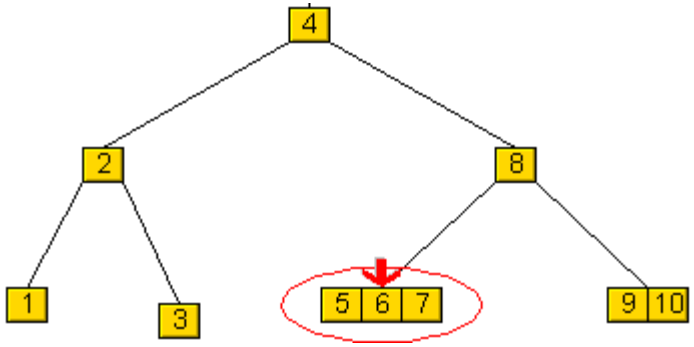




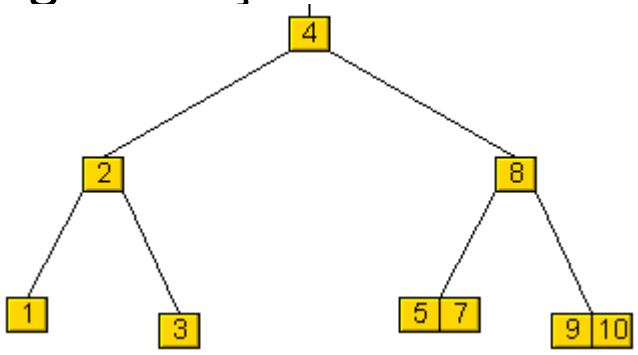
2-3- veya 2-3-4 tree

- Örnek: Silme, 6 silindi



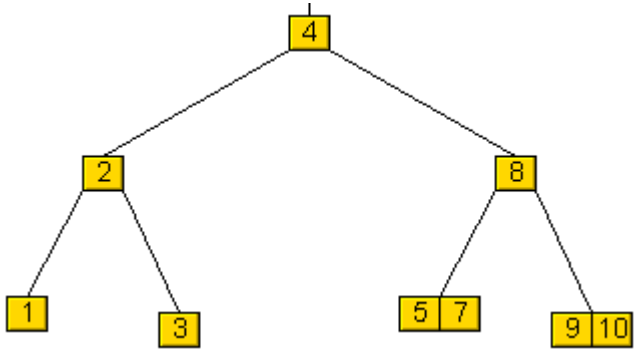


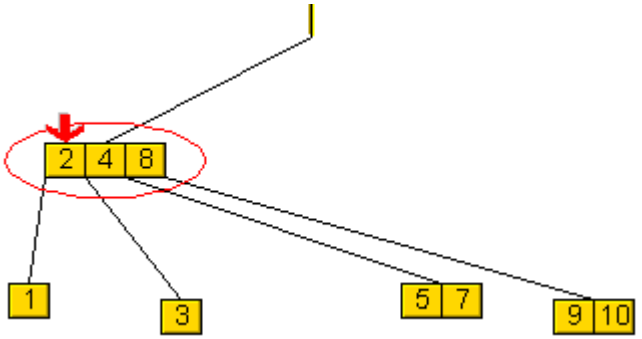
- Silme işlemi gerçekleşmeden 6 değeri sol çocuklar ile birleştiriliyor. Daha sonra silin



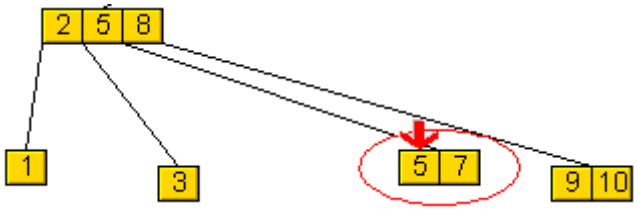
2-3- veya 2-3-4 tree

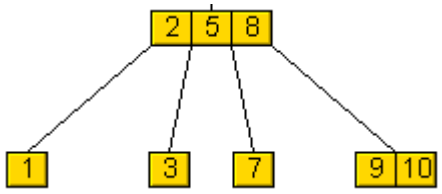
- Örnek: 4 silindi





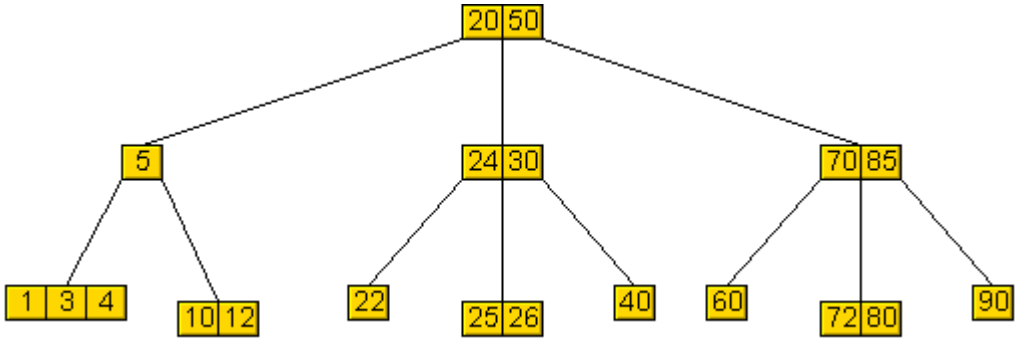
- 4 çocuk olabilmesi için çocuktan da değer alındı.





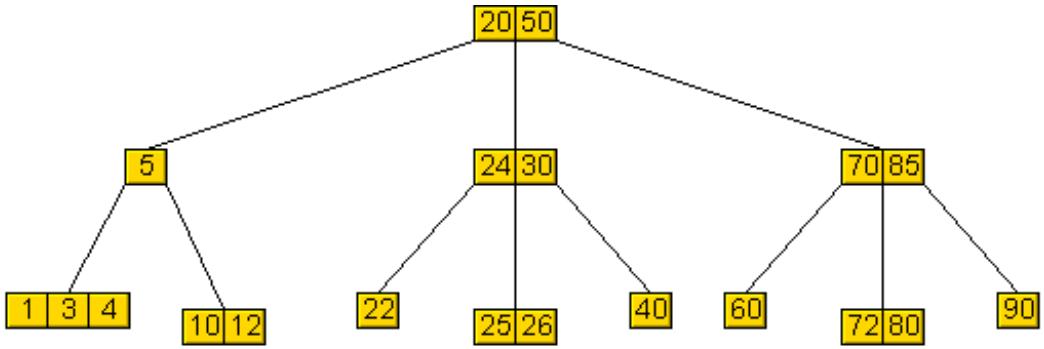
2-3- veya 2-3-4 tree

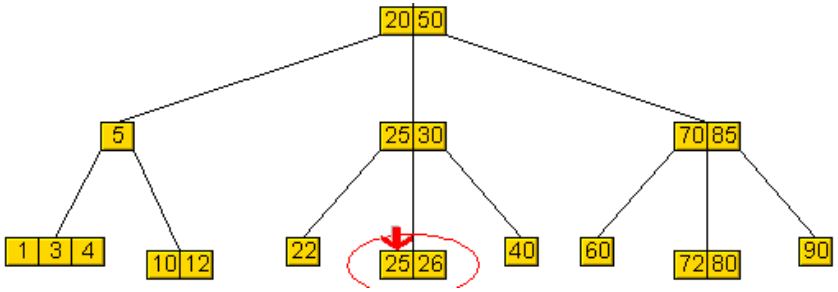
- Örnek:
  50,60,20,30,10,5,90,25,40,70,80,22,85,72,3,1,4,
  12,24,26 ağacı oluşturunuz.

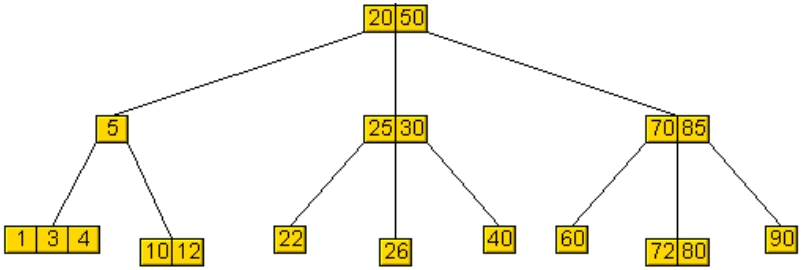


2-3- veya 2-3-4 tree

- Örnek: 24 nolu düğüm sil

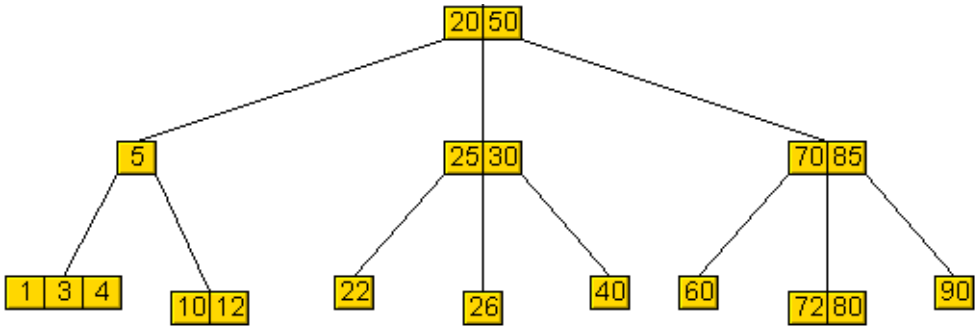


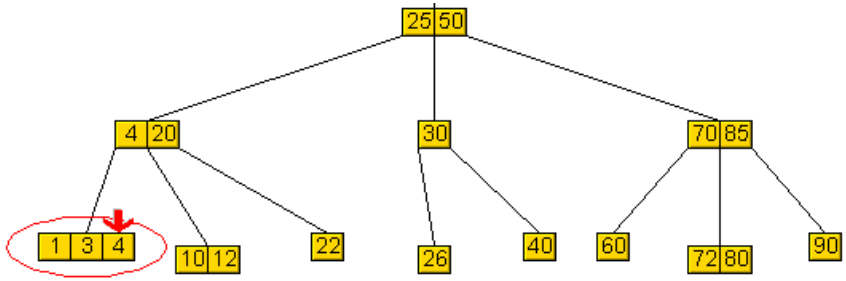


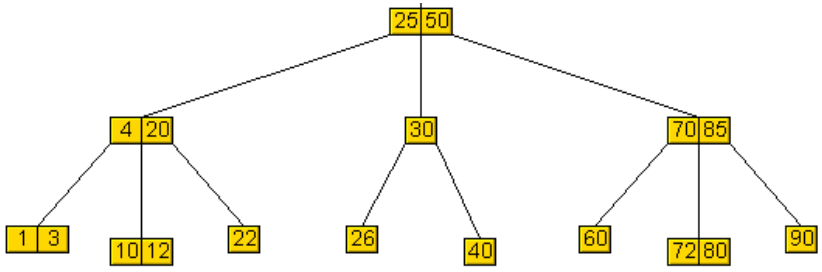


2-3- veya 2-3-4 tree

- Örnek: 5 nolu düğüm sil

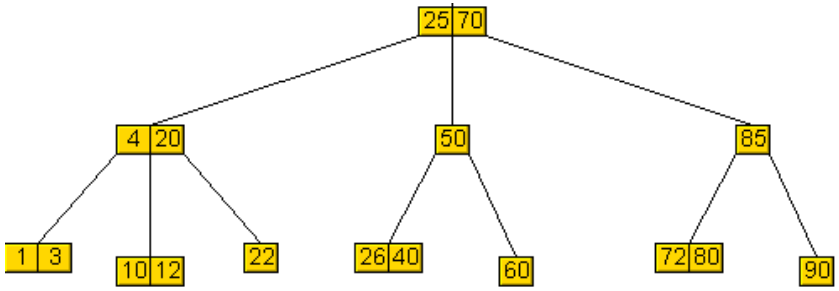


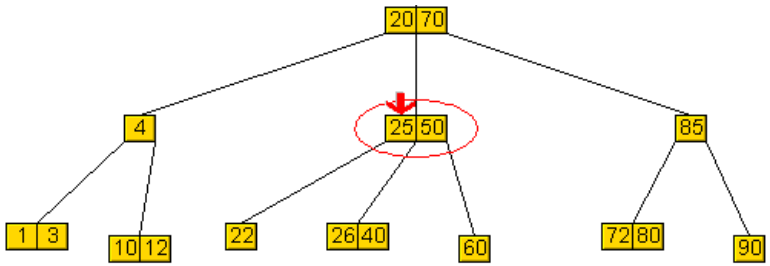


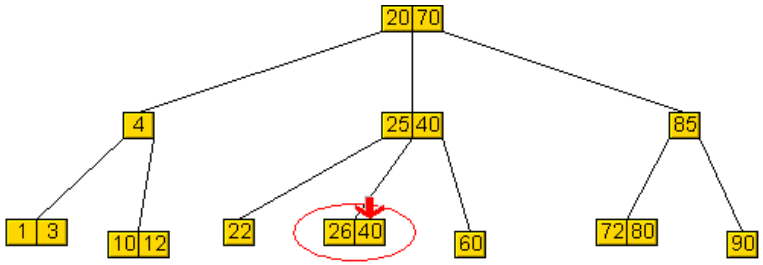


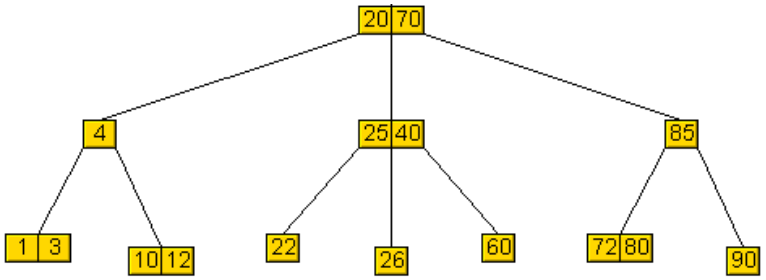
2-3- veya 2-3-4 tree

- Örnek: 50 nolu düğüm sil



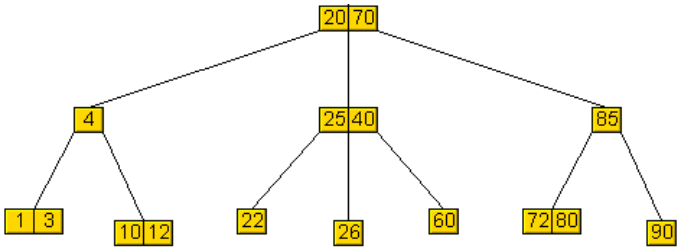


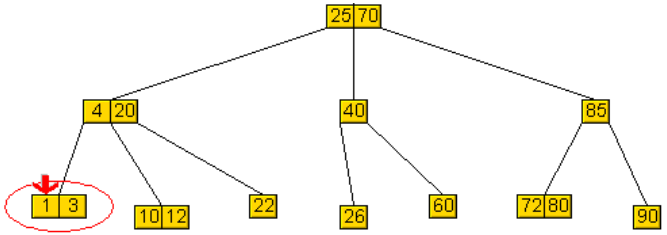


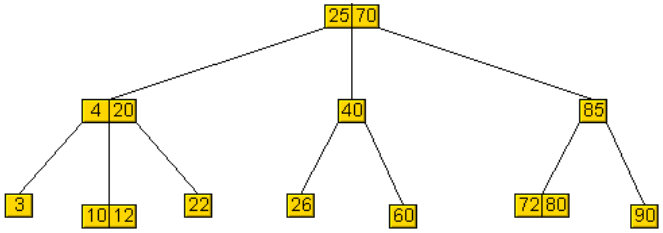


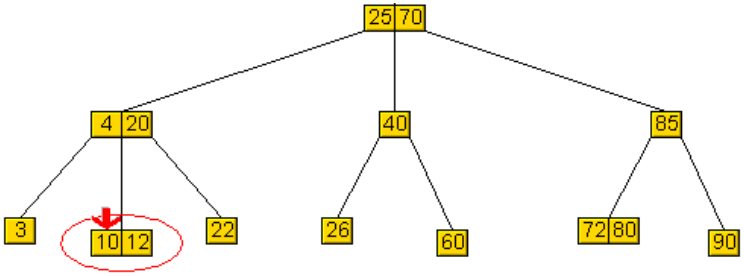
2-3- veya 2-3-4 tree

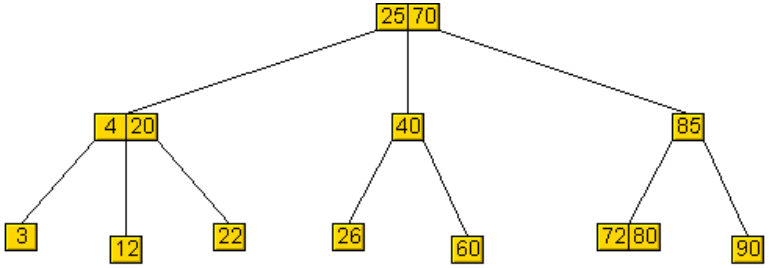
- Örnek: 1,10 nolu düğüm sil





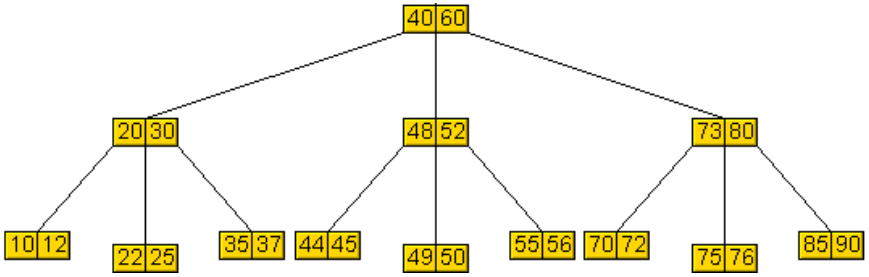


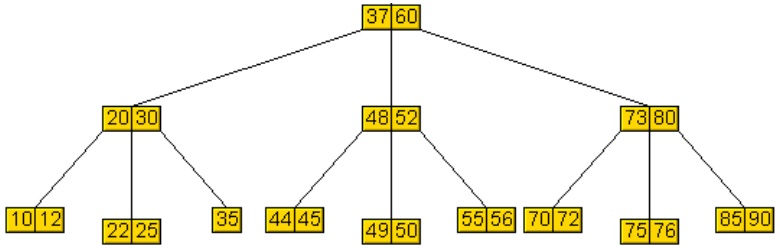


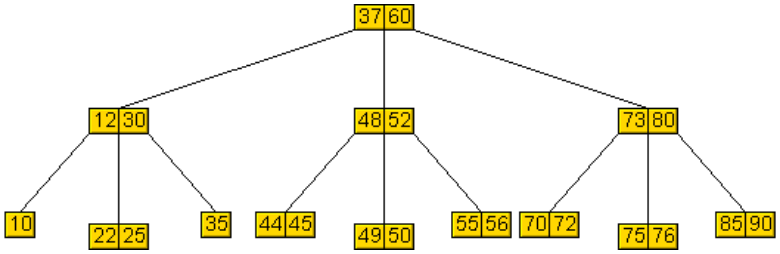


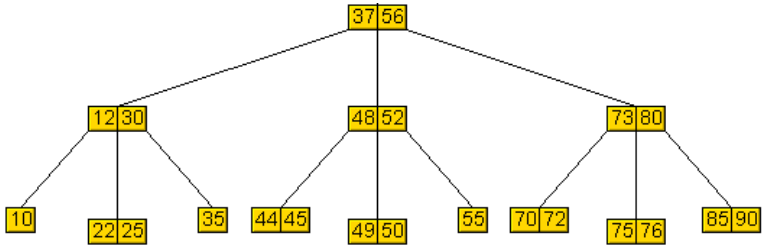
2-3- veya 2-3-4 tree

- Örnek: 40, 20,60 nolu düğüm sil



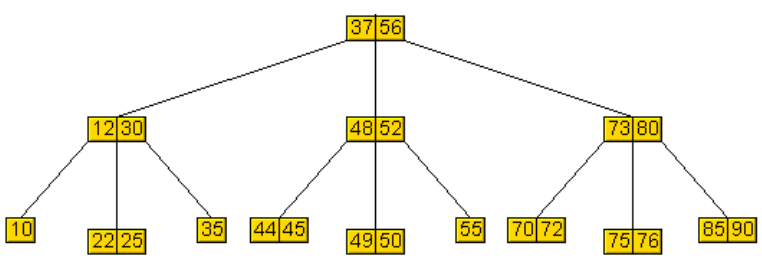


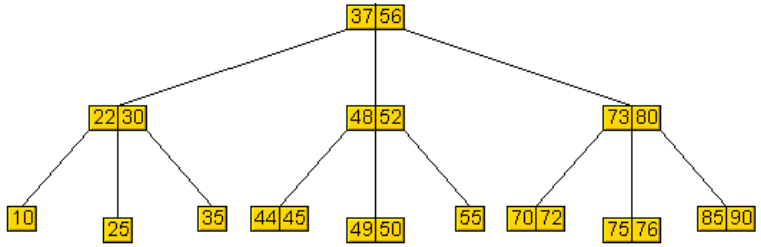


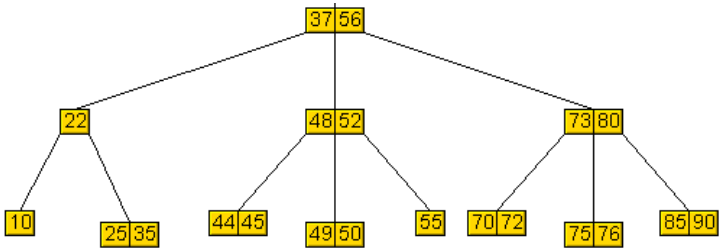


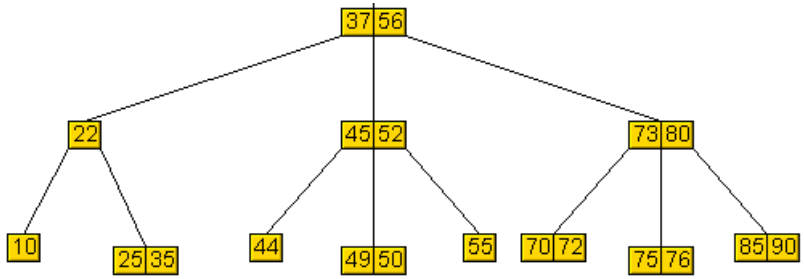
2-3- veya 2-3-4 tree

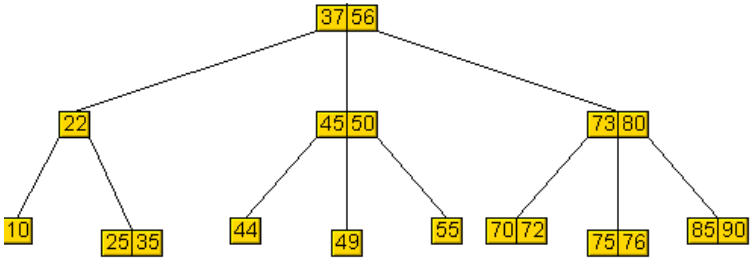
- Örnek: 12,30,48,52,70,90 nolu düğüm sil

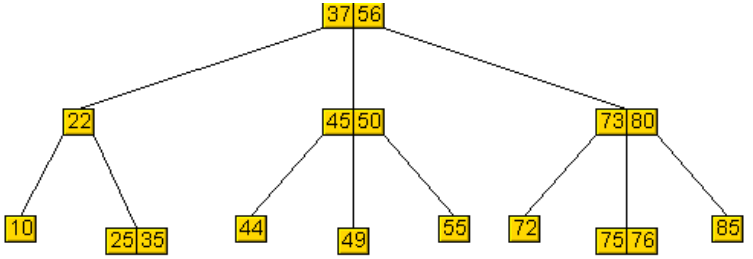






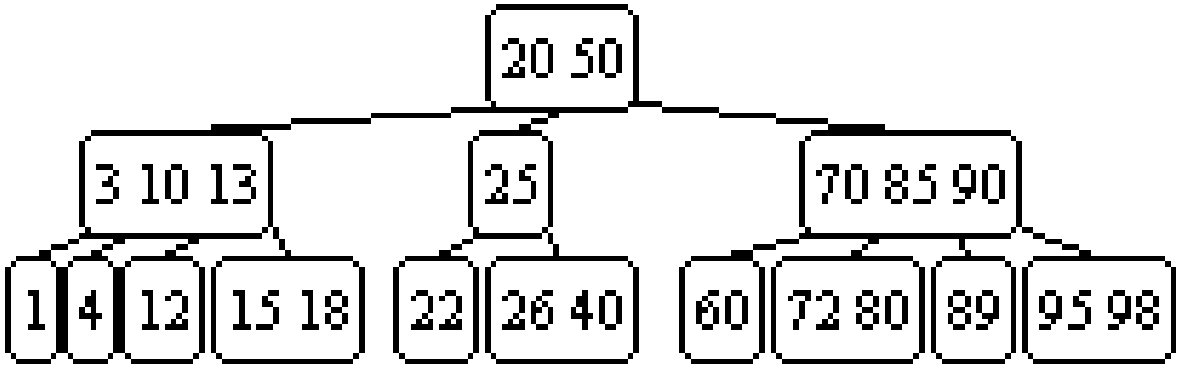






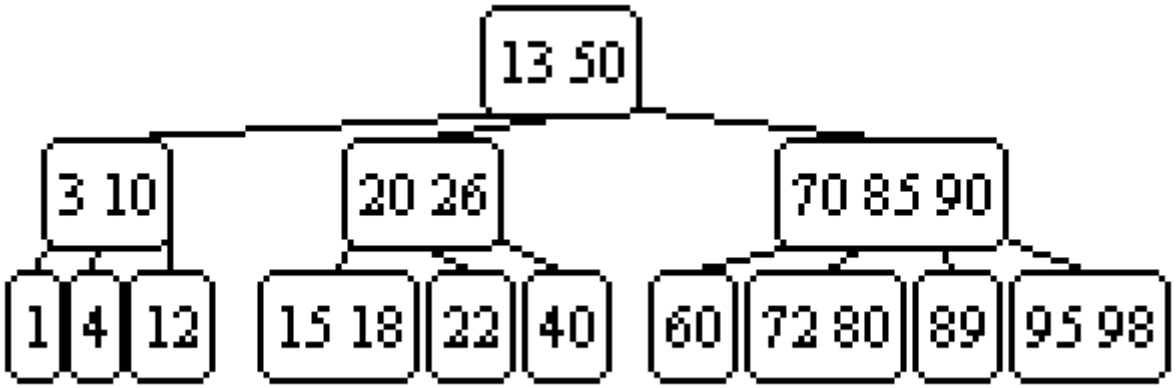
2-3- veya 2-3-4 tree

- Örnek: 25 değerini siliniz



2-3- veya 2-3-4 tree

- Örnek: 25 değerini siliniz



2-3-4 tree java code

- /* Tree234.java */- http://www.lensovet.net/~sysadmin/w/CS/61b/Homework/hw7/dict/Tree234.java
- package dict;
- /**
- * A Tree234 implements an ordered integer dictionary ADT using a 2-3-4 tree.
- * Only int keys are stored; no object is associated with each key. Duplicate
- * keys are not stored in the tree.
- * * @author Jonathan Shewchuk, lensovet
- **/

SIRALAMA ALGORİTMALARI

SIRALAMA ALGORİTMALARI

- Sıralama ve arama tekniklerinden pek çok programda yararlanılmaktadır. Günlük yaşamımızda elemanların sıralı tutulduğu listeler yaygın olarak kullanılmaktadır.
- Sıralama, sıralanacak elemanlar bellekte ise internal (içsel), kayıtların bazıları ikincil bellek ortamındaysa external (dışsal) sıralama olarak adlandırılır.

KABARCIK SIRALAMA (BUBBLE SORT) ALGORİTMASI

- Dizinin elemanları üzerinden ilk elemandan başlayarak ve her geçişte sadece yan yana bulunan iki eleman arasında sıralama yapılır.

- Dizinin başından sonuna kadar tüm elemanlar bir kez işleme tabi tutulduğunda dizinin son elemanı (küçükten büyüğe sıralandığında) en büyük eleman haline gelecektir.

BUBBLE SORT

- Bir sonraki tarama ise bu en sağdaki eleman dışarıda bırakılarak gerçekleştirilmektedir. Bu dışarıda bırakma işlemi de dış döngüdeki sayaç değişkeninin değerinin her işletimde bir azaltılmasıyla sağlanmaktadır. Sayaç değişkeninin değeri 1 değerine ulaştığında ise dizinin solunda kalan son iki eleman da sıralanmakta ve sıralama işlemi tamamlanmaktadır.

- Bubble sort, sıralama teknikleri içinde anlaşılması ve programlanması kolay olmasına rağmen etkinliği en az olan algoritmalardandır (n elemanlı x dizisi için).

BUBBLE SORT

- **Örnek:**

- 9, 5, 8, 3, 1. rakamlarının azalan şekilde sıralanmasını kabarcık algoritmasıyla gerçekleştirelim.

BUBBLE SORT

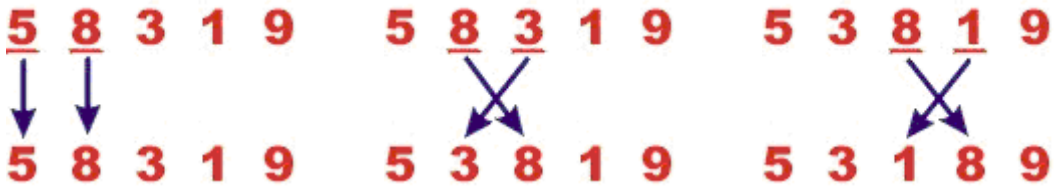
- **1.Tur:**



- 1. tur tamamlandığında en büyük eleman olan 9 en sona yerleşmiş olur ve bir daha karşılaştırmaya gerek yoktur.

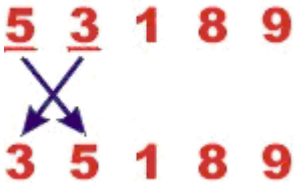
SIRALAMA ALGORİTMALARI-BUBBLE SORT

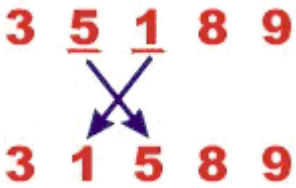
- **2.Tur:**
-

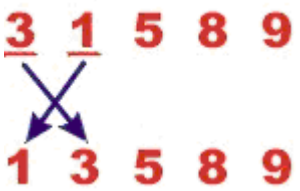


SIRALAMA ALGORİTMALARI-BUBBLE SORT

- **3.Tur:**





- **4.Tur:**
- 

BUBBLE SORT

- En büyük elemanı sona taşıma

```
public static void bublesort(int [] x)
{       int n = x.Length;          int tut, j, i;
        bool sirali= false;
        for (i=0; (i<n-1) &&!sirali; i++)
        { sirali=true;
              for(j=0; j<n-i-1; j++)
               {   if (x[j] > x[j+1])
                   {   tut = x[j];      x[j] = x[j+1];    x[j+1] = tut;
                       sirali=false;
                   }
               }
          }
      }
```

BUBBLE SORT

- **En küçük elamanı başa taşıma**
- **public static void bubleSort(Item a[], int n)**
- **{ bool sorted = false; int last = n-1;**
- **for (int i = 0; (i < last) && !sorted; i++)**
- **{ sorted = true;**
- **for (int j=last; j > i; j--)**
- **if (a[j-1] > a[j])**
- **{ swap(a[j],a[j-1]);**
- **sorted = false; // signal exchange**
- **}**
- **}**
- **}**

BUBBLE SORT

- **Veriler** : 25 57 48 37 12 92 86 33
- **Tekrar 1** : 25 48 37 12 57 86 33 92
- **Tekrar 2** : 25 37 12 48 57 33 86 92
- **Tekrar 3** : 25 12 37 48 33 57 86 92
- **Tekrar 4** : 12 25 37 33 48 57 86 92
- **Tekrar 5** : 12 25 33 37 48 57 86 92
- **Tekrar 6** : 12 25 33 37 48 57 86 92
- **Tekrar 7** : 12 25 33 37 48 57 86 92

BUBBLE SORT

- Analizi kolaydır (İyileştirme yapılmamış algoritmada) :
- (n-1) iterasyon ve her iterasyonda (n-1) karşılaştırma.
- Toplam karşılaştırma sayısı : $(n-1)*(n-1) = n^2-2n+1 = O(n^2)$
- (Yukarıdaki gibi iyileştirme yapılmış algoritmada etkinlik) :
- iterasyon i'de, (n-i) karşılaştırma yapılacaktır.
- Toplam karşılaştırma sayısı = $(n-1)+(n-2)+(n-3)+...+(n-k)$
  $= kn - k*(k+1)/2$
  $= (2kn - k^2 - k)/2$
- ortalama iterasyon sayısı, k, $O(k.n)$ olduğundan = $O(n^2)$
- **Performans:**
  Kabarcık sıralama algoritması ortalama **$N^2/2$** karşılaştırma ve **$N^2/2$** yer değiştirme işlemi gerçekleştirir.

BUBBLE SORT

- Best-case: O(n)
  - – Dizi artan şekilde sıralı ise
  - – Sıfır yer değiştirme olur: 0 → O(1)
  - – Anahtarların karşılaştırılması: (n-1)→O(n)
- • Worst-case: $O(n^2)$
  - – Dizi tersten sıralı ise:
  - – Dış döngü n-1 kez çalışır,
  - – Yer değiştirme: $3*(1+2+...+n-1) = 3 * n*(n-1)/2$ →$O(n^2)$
  - – Karşılaştırma: $(1+2+...+n-1)= n*(n-1)/2$ → $O(n^2)$
- • Average-case: $O(n^2)$

SEÇMELİ SIRALAMA (SELECTION SORT)

- Dizideki en küçük elemanı bul, bu elemanı dizinin son (yer olarak) elemanıyla yer değiştir.
- Daha sonra ikinci en küçük elemanı bul ve bu elemanı dizinin ikinci elemanıyla yer değiştir. Bu işlemi dizinin tüm elemanları sıralanıncaya kadar sonraki elemanlarla tekrar et.
- Elemanların seçilerek uygun yerlerine konulması ile gerçekleştirilen bir sıralamadır :

SEÇMELİ SIRALAMA (SELECTION SORT)

- Örnek: 9, 5, 8, 3, 1. rakamlarının azalan şekilde sıralanmasını seçmeli sıralama algoritmasıyla gerçekleştirelim. (Küçük olanı bul)
- 1. Tur:
- 9 5 8 3 1 9 5 8 3 1 9 5 8 3 1 9 5 8 3 1
- 9 5 8 3 1 9 5 8 3 1 9 5 8 3 1 9 5 8 3 1
- 9 5 8 3 1
- 1 5 8 3 9

SEÇMELİ SIRALAMA (SELECTION SORT)

- 2. Tur:
- 1 5 8 3 9 1 5 8 3 9 1 5 8 3 9
- 1 5 8 3 9 1 5 8 3 9 1 3 8 5 9
- 3. Tur:
- 1 3 8 5 9 1 3 8 5 9
- 1 3 8 5 9 1 3 5 8 9
- 4. Tur:
- Sıralama tamam

SEÇMELİ SIRALAMA (SELECTION SORT)-java

```
public static int[] selectionsort(int[] A, int n)
{
    int tmp; int min;
    for (int i = 0; i < n - 1; i++)
    {
       min = i;
       for (int j = i; j < n - 1; j++)
       {
          if (A[j] < A[min]) { min = j; }
       }
       tmp = A[i]; A[i] = A[min]; A[min] = tmp;
     }
     return A;
}
```

SEÇMELİ SIRALAMA (SELECTION SORT) -C

```
#include <stdio.h>
void selectsort(int x[], int n) {
  int i, indx, j, large;
  for(i=0; i<n; i++)   {
    large = x[i];    indx = i;
    for(j=i+1; j<n; j++)
       if (x[j] < large)
       { large = x[j];    indx = j;   printf("a=%d \n ",x[j]);   }

     x[indx] = x[i];
     x[i] = large;        }

}
void main() {
  int dizi[8] = {25, 57, 48, 37, 12, 92, 86, 33};
 selectsort( &dizi[0], 8);
   for(int i=0;i<8;i++)  printf("%d\n  ",dizi[i]);
}
```

SEÇMELİ SIRALAMA (SELECTION SORT)

- En küçüğe göre sıralama
- Veriler : 25 57 48 37 12 92 86 33
- Tekrar 1 : 12 57 48 37 25 92 86 33
- Tekrar 2 : 12 25 48 37 57 92 86 33
- Tekrar 3 : 12 25 33 37 57 92 86 48
- Tekrar 4 : 12 25 33 37 57 92 86 48
- Tekrar 5 : 12 25 33 37 48 92 86 57
- Tekrar 6 : 12 25 33 37 48 57 86 92
- Tekrar 7 : 12 25 33 37 48 57 86 92

Tekrar 8 : 12 25 33 37 48 57 86 92

- Selection Sort'un analizi doğrudandır.
- turda (n-1),
- turda (n-2),
- ...
- (n-1). Turda 1, karşılaştırma yapılmaktadır.
- Toplam karşılaştırma sayısı = (n-1)+(n-2)+...+1 = n*(n-1)/2
- = (1/2)n^2-(1/2)n = O(n^2)

SEÇMELİ SIRALAMA (SELECTION SORT)

```
//Enbüyük elemanı bulup sona atarak selection sıralama
#include <stdio.h>
void selectsort(int x[], int n) {
  int i, indx, j, large;
  for(i=0; i<n; i++)  {
    large = x[n-i-1];    indx=n-i-1;
    for(j=0; j<n-i; j++)
     if(x[j]>large)      { large = x[j];       indx = j;    }
     x[indx] = x[n-i-1];
     x[n-i-1] = large;
       }
  }
 void main() {
 int dizi[8] = {25, 57, 48, 37, 12, 92, 86, 33};
 selectsort( &dizi[0], 8);
   for(int i=0;i<8;i++)  printf("%d\n  ",dizi[i]);
}
```

SEÇMELİ SIRALAMA (SELECTION SORT)

- En büyüğüne göre sıralama (en sondakini en büyük al)
- Veriler : 25 57 48 37 12 92 86 33
- Tekrar 1 : 25 57 48 37 12 33 86 92
- Tekrar 2 : 25 57 48 37 12 33 86 92
- Tekrar 3 : 25 33 48 37 12 57 86 92
- Tekrar 4 : 25 33 12 37 48 57 86 92
- Tekrar 5 : 25 33 12 37 48 57 86 92
- Tekrar 6 : 25 12 33 37 48 57 86 92
- Tekrar 7 : 12 25 33 37 48 57 86 92

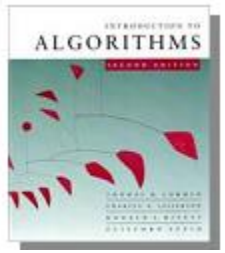
SEÇMELİ SIRALAMA (SELECTION SORT)

- **Performans :**
  n elemanlı bir dizi için, seçerek sıralama algoritması yaklaşık $n^2/2$ karşılaştırma ve n yer değiştirme işlemi gerçekleştirmektedir. Bu özelliği seçerek sıralama işlevinin gerçekleştiriminden çıkarmak mümkündür.

- Dış döngünün her işletiminde bir tek yer değiştirme işlemi gerçekleştirildiğinden, bu döngü n adet işletildiğinde (n=dizi boyutu) n tane yer değiştirme işlemi gerçekleştirilecektir.

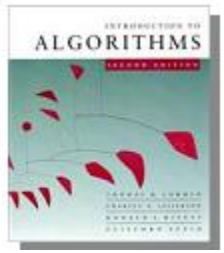
- Bu döngünün her işletiminde ayrıca N-i adet karşılaştırma gerçekleştirildiğini göz önüne alırsak toplam karşılaştırma sayısı (n-1)+(n-2)+...+2+1→ $n^2/2$ olacaktır.

Çabuk sıralama (Quick Sort)



- C.A.R. Hoare tarafından 1962'de önerildi.
- Böl ve fethet algoritması.
- "Yerinde" sıralar (araya yerleştirme sıralamasında olduğu gibi; birleştirme sıralamasından farklı).

Çabuk sıralama (Quick Sort)
Böl ve fethet



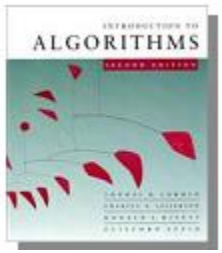
- ***n*-elemanlı bir dizilimin çabuk sıralanması:**
- ***1. Böl:*** Dizilimi **pivot (eksen sabit) *x***'in etrafında iki altdizilime bölüntüle; burada soldaki altdizilim elemanları **≤ *x* ≤** sağdaki altdizilim elemanları olsun.



- ***2. Fethet:*** İki altdizilimi özyinelemeli sırala.
- ***3. Birleştir:*** Önemsiz (yerinde sıraladığı için)

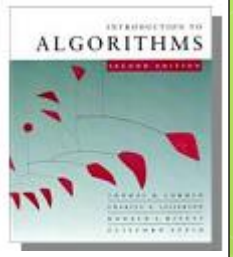
Anahtar: Doğrusal-zamanlı ($\Theta(n)$)bölüntü altyordamı.

Çabuk sıralama (Quick Sort)
Böl ve fethet



- Quicksort algoritmasında yapılan ana iş öz yinelemede bölüntülere ayırma işlemidir. Bütün iş bölüntüleme de yapılmaktadır.
- Buradaki anahtar olay bölüntü alt yordamı doğrusal zamanda yani $\Theta(n)$ olması.
- Merge sort algoritmasında ana iş ise öz yinelemeli birleştirme yapmadır.

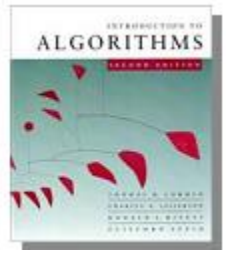
Çabuk sıralama (quicksort) için sözdekod




```
QUICKSORT(A, p, r)
    if p < r
        then q ← PARTITION(A, p, r)
             QUICKSORT(A, p, q–1)
             QUICKSORT(A, q+1, r)
```

İlk arama: QUICKSORT$(A, 1, n)$

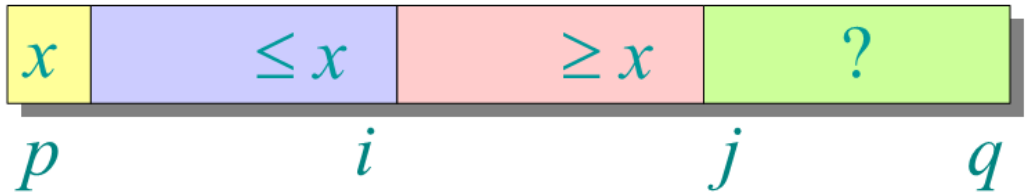
Çabuk sıralama (Quick Sort)
Bölüntüleme örneği

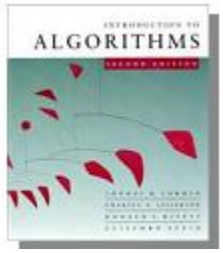



```
PARTITION (Bölüntü)(A, p, q)  ▷ A[p . . q]
    x ← A[p]              ▷ pivot = A[p]
    i ← p                 (eksen sabit)
    for j ← p + 1 to q
        do if A[j] ≤ x    (öyleyse yap)
              then  i ← i + 1
                    exchange A[i] ↔ A[j]  (değiştir)
    exchange A[p] ↔ A[i]  (değiştir)
    return i              (dön)
```

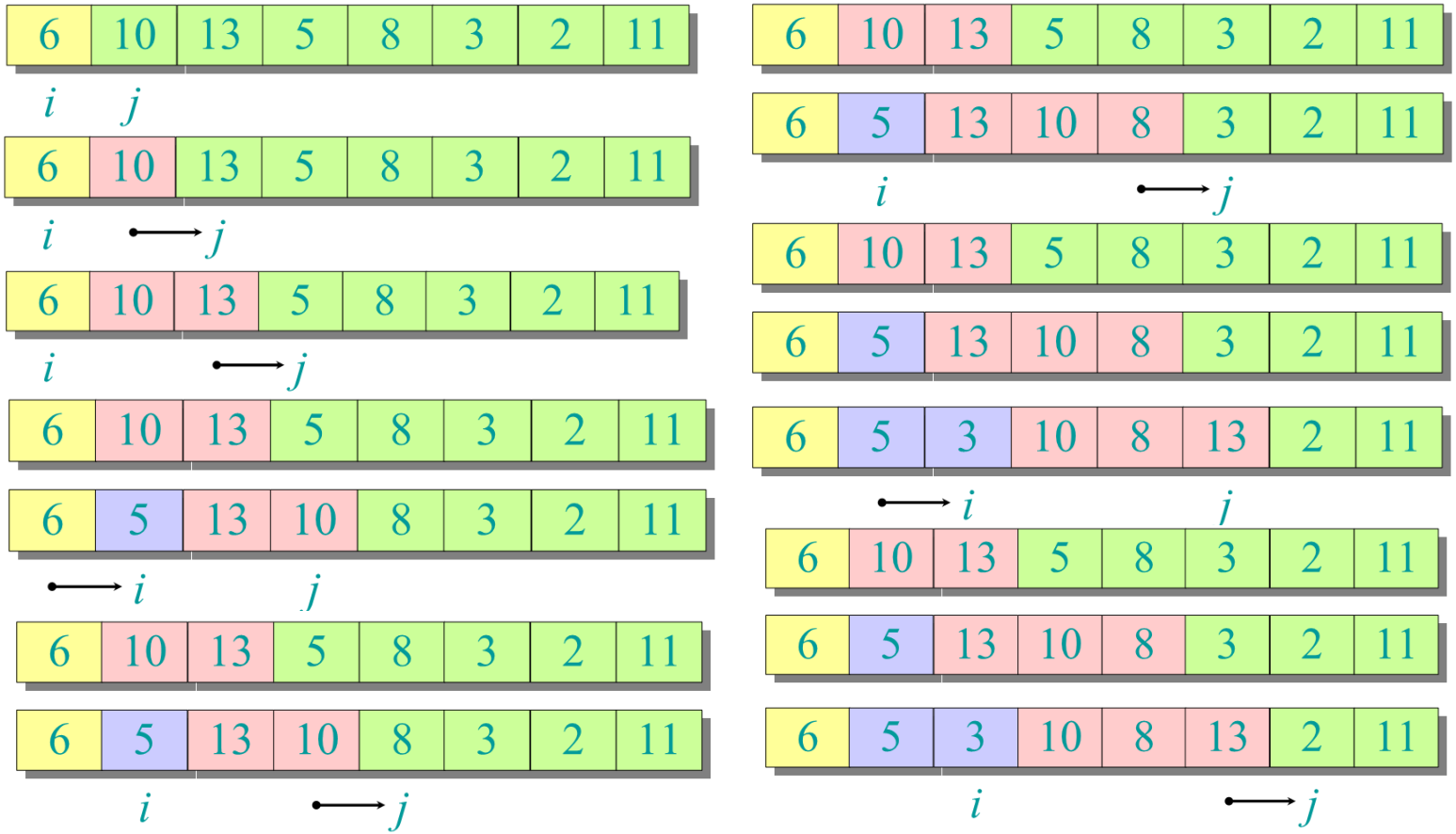
Koşma süresi
=O(n)
n eleman için

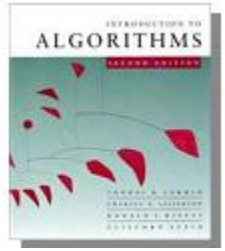
Değişmez:



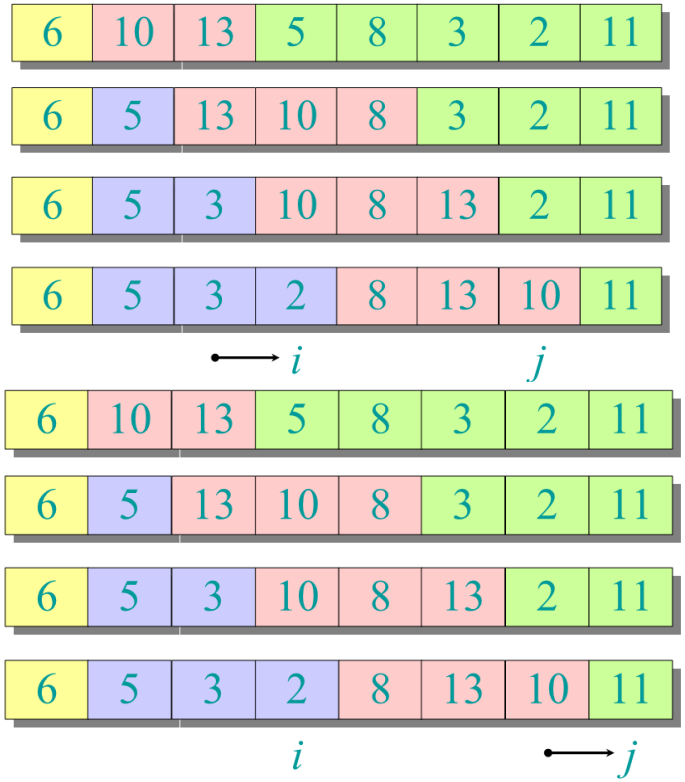


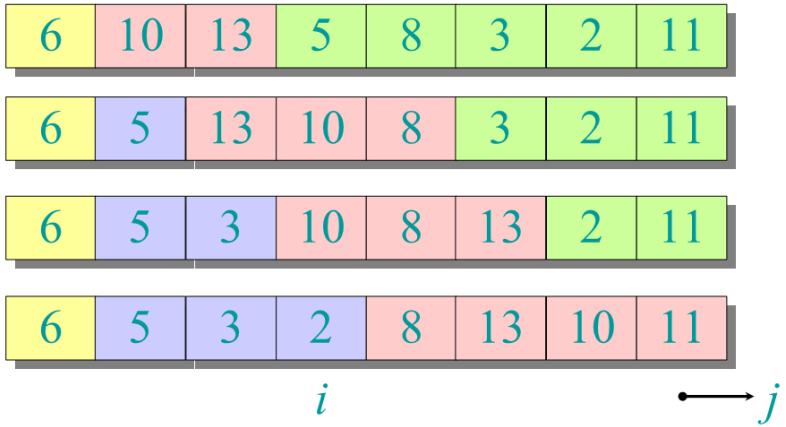
Çabuk sıralama (Quick Sort)
Bölüntüleme örneği



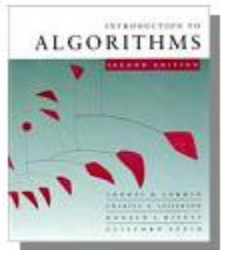


Çabuk sıralama (Quick Sort)
Bölüntüleme örneği





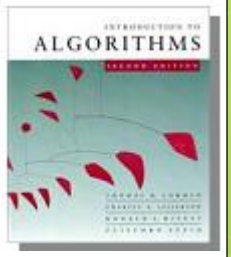
Çabuk sıralama (Quick Sort)
Bölüntüleme örneği



6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11

i

Çabuk sıralamanın en kötü ve ortalama durumu



- Girdiler sıralı ya da ters sıralı. (Ancak sıralı girişler insert sort için en iyi durum olur)
- En küçük yada en büyük elemanların etrafında bölüntüleme.
- Bölüntünün bir yanında hiç eleman yok veya parçalardan biri sadece bir elemana sahip;
- En kötü durum :$T(n)= O(n^2)$
- Ortalama Durum; $T(n)=O(nlogn)$ (Bölüntü var ise)

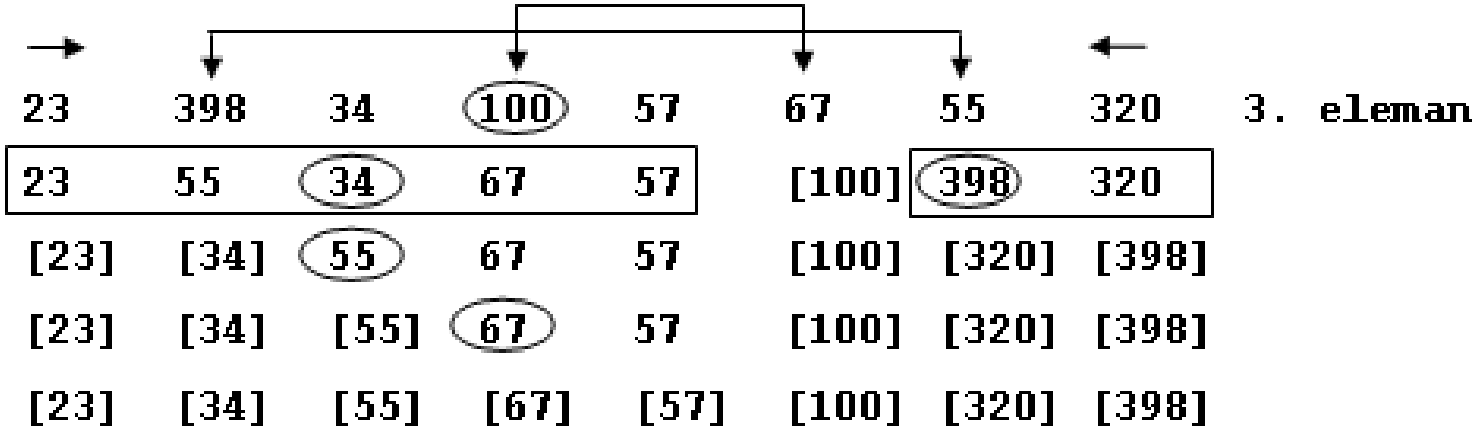
Rastgele çabuk sıralama

- **Genelde şanslı olmak için**
  - **1-Ortadaki elamanın yakınından (n/2) bölme yapılır**
  - **2-Rastgele seçilen bir elamana göre bölme yapılır (Pratik daha iyi çalışır.)**

Çabuk SIRALAMA (QUICKSORT)

- **1-Ortadaki elamanın yakınından (n/2) bölme yapılır.**
- **n** elemanlı bir dizi sıralanmak istendiğinde dizinin herhangi bir yerinden x elemanı seçilir (örnek olarak ortasındaki eleman). X elemanı j. yere yerleştiğinde 0. İle (j-1). yerler arasındaki elemanlar x'den küçük, j+1'den (n-1)'e kadar olan elemanlar x'den büyük olacaktır. Bu koşullar gerçekleştirildiğinde x, dizide en küçük j. elemandır. Aynı işlemler, x[0]-x[j-1] ve x[j+1]-x[n-1] alt dizileri (parçaları) için tekrarlanır. Sonuçta veri grubu sıralanır.

QUICKSORT


23 398 34 100 57 67 55 320 3. eleman
23 55 34 67 57 [100] 398 320
[23] [34] 55 67 57 [100] [320] [398]
[23] [34] [55] 67 57 [100] [320] [398]
[23] [34] [55] [67] [57] [100] [320] [398]

QUICKSORT

p							q
0	4	8	12	16	20	24	28

- 23 398 34 100 57 67 55 320
- 23 398 34 [100] 57 67 55 320
- 23 **55** 34 [100] 57 67 **398** 320
- 23 55 34 [100] 57 67 398 320
- 23 55 34 [100] 57 67 398 320
- 23 55 34 **67** 57 **[100]** 398 320

QUICKSORT

- 0 4 8 12 16 20 24 28
- 23 55 34 67 57 100 398 320
- 23 [34] 55 67 57 100 320 [398]
- 23 34 55 67 57 100 320 398
- 23 34 55 57 [67] 100 320 398
- 23 34 55 57 67 100 320 398

QUICKSORT

- Eğer şanslı isek, seçilen her eleman ortanca değere yakınsa $\log_2 n$ iterasyon olacaktır = **$O(n \log_2 n)$**. Ortalama durumu işletim zamanı da hesaplandığında $O(n \log_2 n)$'dir, yani genelde böyle sonuç verir. En kötü durumda ise parçalama dengesiz olacak ve n iterasyonla sonuçlanacağında **$O(n^2)$** olacaktır (en kötü durum işletim zamanı).

QUICKSORT – C code

- Pivot orta elaman seçildi
- #include <stdio.h>
- void qsort2 (double *left, double *right){
- double *p = left, *q = right, w, x=*(left+(right-left>>1));
- /* ilk aders ile son adres değerinin farkını alıp 2 ye böldükten sonra çıkan değeri tekrar ilk adresle topluyor.adres değerlerini görmek için
- printf("%f ",*(left+(right-left>>1)));
- printf("r %d\n",right); printf("l %d\n",left);*/
- do {
- while(*p<x) p++;
- while(*q>x) q--;
- if(p>q) break;
- w = *p; *p = *q; *q = w;
- } while(++p <= --q);
- if(left<q) qsort2(left,q);
- if(p<right) qsort2(p,right); }
- void main(){
- double dizi[8] = { 23, 398, 34, 100, 57, 67, 55, 320 };
- qsort2 (&dizi[0], &dizi[7]);
- for(int i=0;i<8;i++) printf("%f ",dizi[i]);}

Rastgele çabuk sıralama (Randomized Quicksort)

- **2-Rastgele seçilen bir elamana göre bölme yapılır.**

```
Randomized-Partition(A,p,r)
01  i←Random(p,r)
02  exchange A[r] ↔A[i]
03  return Partition(A,p,r)


Randomized-Quicksort(A,p,r)
01  if p<r then
02      q←Randomized-Partition(A,p,r)
03      Randomized-Quicksort(A,p,q)
04      Randomized-Quicksort(A,q+1,r)
```

QUICKSORT – C# Code

- Başlangıçta Pivot en son elaman seçildi

```
class Program {
public static void QuickSort(int[] input, int left, int right)
{ if (left < right)
  { int q = Partition(input, left, right);
    QuickSort(input, left, q - 1); QuickSort(input, q + 1, right);
  }
   }
private static int Partition(int[] input, int left, int right)
{   int pivot = input[right];    int temp; int i = left;
    for (int j = left; j < right; j++)
    { if (input[j] <= pivot)
      { temp = input[j]; input[j] = input[i]; input[i] = temp; i++; }
    }
    input[right] = input[i]; input[i] = pivot;
    return i;
 }
```

QUICKSORT – C# Code

```
    static void Main(string[] args)
    {
     int[] dizi = { 23, 398, 34, 100, 57, 67, 55, 320 };

     QuickSort(dizi, 0, dizi.Length - 1);

     for (int i = 0; i < 8; i++) Console.Write("  " + dizi[i]);
    }
}
```

QUICKSORT – Java Code

- Pivot ilk elaman seçildi
- import TerminalIO.*;
- class QuickSort{
- public static void main(String []args){
- KeyboardReader input=new KeyboardReader();
- int n=input.readInt("enter list size : ");
- int[] Arr=new int[n];
- for(int i=0;i<n;i++)
- Arr[i]=input.readInt("enter elements :");
- int pivot1;
- pivot1=partition(Arr,0, n-1);
- System.out.println("Pivot Value is "+pivot1);
-
-
- quicksort(Arr, 0, n-1);
- for(int j=0; j<Arr.length ; j++)
- System.out.println(Arr[j]);
- }
-

QUICKSORT – Java Code

```
public static void  quicksort(int array[], int left,int  right)
 {   int pivot =partition(array, left, right);
    if (left<pivot)
      quicksort(array, left, pivot-1);
     if (right>pivot)
     quicksort(array, pivot+1, right);
   }
public static int partition(int numbers[],int left,int right)
{     int I_hold,r_hold,i;      int pivot;    I_hold=left;              r_hold=right;
      pivot=numbers[left];
      while(left<right){
      while((numbers[right]>=pivot)&&(left<right))   right--;
      if(left!=right) {           numbers[left]=numbers[right];    left++;       }
      while((numbers[left]<=pivot)&&(left<right))                  left++;
      if(left!=right){ numbers[right]=numbers[left]; right--;     }
}
      numbers[left]=pivot;                pivot=left;               left=I_hold;
      right=r_hold;                       return pivot;
}     }
```

EKLEMELİ SIRALAMA (INSERTION SORT)

- Yerleştirerek sıralama işlevi belirli bir anda dizinin belirli bir kısmını sıralı tutarak ve bu kısmı her adımda biraz daha genişleterek çalışmaktadır. Sıralı kısım işlev son bulunca dizinin tamamına ulaşmaktadır.

- Elemanların sırasına uygun olarak listeye tek tek eklenmesi ile gerçekleştirilen sıralamadır :

INSERTION SORT

Veriler	**25**	57	48	37	12	92	86	33
Tekrar 1	**25**	**57**	48	37	12	92	86	33
Tekrar 2	**25**	**48**	**57**	37	12	92	86	33
Tekrar 3	**25**	**37**	**48**	**57**	12	92	86	33
Tekrar 4	**12**	**25**	**37**	**48**	**57**	92	86	33
Tekrar 5	**12**	**25**	**37**	**48**	**57**	**92**	86	33
Tekrar 6	**12**	**25**	**37**	**48**	**57**	**86**	**92**	33
Tekrar 7	**12**	**25**	**33**	**37**	**48**	**57**	**86**	**92**

INSERTION SORT

```
void insertsort(int x[], int n)
{
  int i,k,y;
  for(k=1; k<n; k++)
  {
    y=x[k];
    for(i=k-1; i>=0 && y<x[i]; i--)
      x[i+1]=x[i];
    x[i+1]=y;
  };
}
```

INSERTION SORT

- **Performans :**
- Eğer veriler sıralı ise her turda 1 karşılaştırma yapılacaktır ve **O(n)** olacaktır.
- Veriler ters sıralı ise toplam karşılaştırma sayısı:
- **(n-1)+(n-2)+...+3+2+1 = n*(n+1)/2 = $O(n^2)$** olacaktır.
- Simple Insertion Sort'un ortalama karşılaştırma sayısı ise $O(n^2)$'dir.
- Selection Sort ve Simple Insertion Sort, Bubble Sort'a göre daha etkindir. Selection Sort, Insertion Sort'tan daha az atama işlemi yaparken daha fazla karşılaştırma işlemi yapar.

INSERTION SORT

- Bu nedenle Selection Sort, az elemanlı veri grupları için (atamaların süresi çok fazla olmaz) ve karşılaştırmaların daha az yük getireceği basit anahtarlı durumlarda uygundur.

- Tam tersi için, insertion sort uygundur. Elemanlar bağlı listedelerse araya eleman eklemelerde veri kaydırma olmayacağından insertion sort mantığı uygundur.

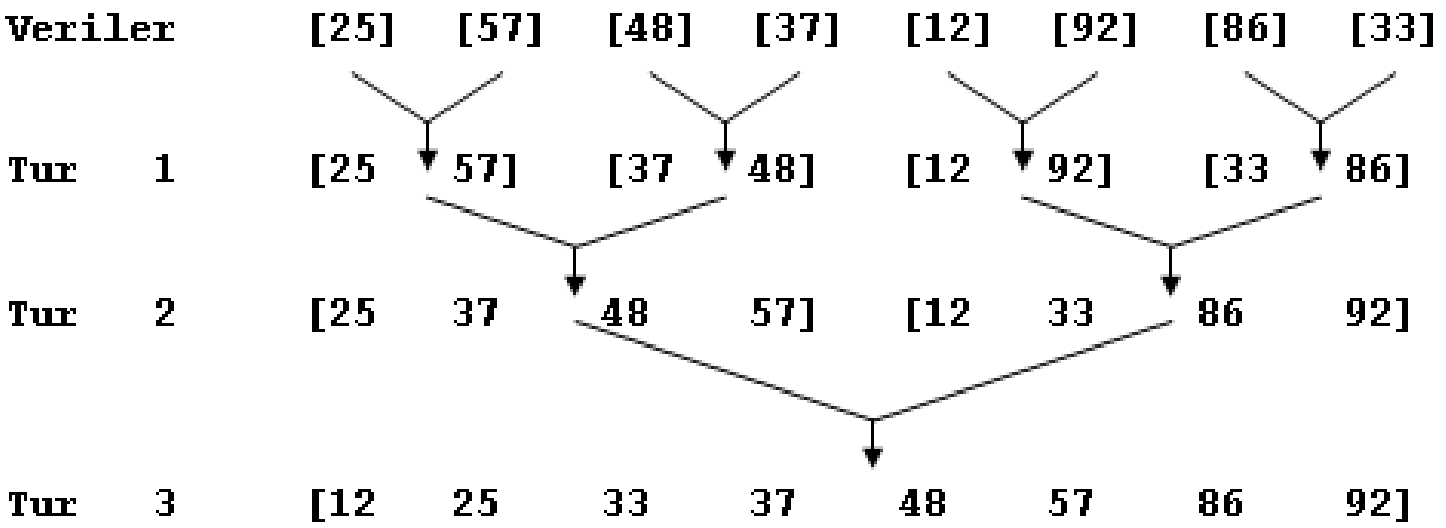
- n'in büyük değerleri için quicksort, insertion ve selection sort'tan daha etkindir. Quicksort'u kullanmaya başlama noktası yaklaşık 30 elemanlı durumlardır; daha az elemanın sıralanması gerektiğinde insertion sort kullanılabilir.

Birleştirmeli Sıralama (MERGE SORT)

- Verinin hafızada sıralı tutulması için geliştirilen sıralama algoritmalarından (sorting algorithms) bir tanesidir.
- Basitçe sıralanacak olan diziyi ikişer elemanı kalan parçalara inene kadar sürekli olarak ikiye böler. Sonra bu parçaları kendi içlerinde sıralayarak birleştirir.
- Sonuçta elde edilen dizi sıralı dizinin kendisidir. Bu açıdan bir parçala fethet (divide and conquere) yaklaşımıdır.

Birleştirmeli Sıralama (MERGE SORT)

- Sıralı iki veri grubunu birleştirerek üçüncü bir sıralı veri grubu elde etmeye dayanır.



Birleştirmeli Sıralama (MERGE SORT)

- Sıralanmak istenen verimiz:
- 5,7,2,9,6,1,3,7olsun.
- Bu verilerin bir oluşumun(composition) belirleyici alanları olduğunu düşünebiliriz. Yani örneğin vatandaşlık numarası veya öğrenci numarası gibi. Dolayısıyla örneğin öğrencilerin numaralarına göre sıralanması durumunda kullanılabilir.
- Birleştirme sıralamasının çalışması yukarıdaki bu örnek dizi üzerinde adım adım gösterilmiştir. Öncelikle parçalama adımları gelir. Bu adımlar aşağıdadır.
- 1. adım diziyi ikiye böl:
- 5,7,2,9 ve 6,1,3,7
- 2. adım çıkan bu dizileri de ikiye böl:
- 5,7 ; 2,9 ; 6,1 ; 3,7

Birleştirmeli Sıralama (MERGE SORT)

- 3. adım elde edilen parçalar 2 veya daha küçük eleman sayısına ulaştığı için dur (aksi durumda bölme işlemi devam edecekti)
- 4. adım her parçayı kendi içinde sırala
- 5,7 ; 2,9 ; 1,6 ; 3,7
- 5. Her bölünmüş parçayı birleştir ve birleştirirken sıraya dikkat ederek birleştir (1. ve 2. parçalar ile 3. ve 4. parçalar aynı gruptan bölünmüştü)
- 2,5,7,9 ve 1,3,6,7
  6. adım, tek bir bütün parça olmadığı için birleştirmeye devam et
- 1,2,3,5,6,7,7,9
- 7. adım sonuçta bir bütün birleşmiş parça olduğu için dur. İşte bu sonuç dizisi ilk dizinin sıralanmış halidir.

MERGE SORT- Java

- Öncelikle birleştirme sıralamasının ana fonksiyonu:

```
 public class MergeSort {
    private int[] list;
    public MergeSort(int[] listToSort){list = listToSort;   }
    public int[] getList() { return list;    }
   // dışarıdan çağırılan sıralama fonksiyonu
    public void sort() { list = sort(list);   }
  // Özyineli olarak çalışan ve her parça için kullanılan
sıralama fonksiyonu
   private int[] sort(int[] whole) {
if (whole.length == 1) {     return whole;}
else {
   // diziyi ikiye bölüyoruz ve solu oluşturuyoruz
    int[] left = new int[whole.length/2];
    System.arraycopy(whole, 0, left, 0, left.length);
```

MERGE SORT- Java

```
//dizinin sağını oluşturuyoruz ancak tek sayı ihtimali var
int[] right = new int[whole.length-left.length];
 System.arraycopy(whole, left.length, right, 0, right.length);
    // her iki tarafı ayrı ayrı sıralıyoruz
 left = sort(left);
 right = sort(right);
    // Sıralanmış dizileri birleştiriyoruz
 merge(left, right, whole);
 return whole;
  }
}
```

MERGE SORT- Java

```
// birleştirme fonksiyonu
 private void merge(int[] left, int[] right, int[] result)
 {
    int x = 0; int y = 0; int k = 0;
   // iki dizide de eleman varken
    while (x < left.length && y < right.length)
    {
       if (left[x] < right[y]) { result[k] = left[x]; x++; }
       else { result[k] = right[y]; y++; }
       k++;
    }
    int[] rest; int restIndex;
    if (x >= left.length) { rest = right; restIndex = y; }
    else { rest = left; restIndex = x; }
    for (int i = restIndex; i < rest.length; i++)
    {  result[k] = rest[i]; k++; }
 }
```

MERGE SORT- Java

```
public static void main(String[] args)
{
  int[] arrayToSort = {15, 19, 4, 3, 18, 6, 2, 12, 7, 9, 11, 16};
  System.out.println("Unsorted:");
  for(int i = 0;i< arrayToSort.length ; i++)
  {    System.out.println(arrayToSort[i] + " ");      }

  MergeSort sortObj = new MergeSort(arrayToSort);
  sortObj.sort();
  System.out.println("Sorted:");
  int [] sirali = sortObj.getList();
  for(int i = 0;i< sirali.length ; i++)
  {   System.out.println(sirali[i] + " "); }
 }
}
```

MERGE SORT- C

```
#include <conio.h>
#define Boyut 8
void mergesort(int x[], int n) {
  int aux[Boyut], i,j,k,L1,L2,size,u1,u2,tur=0;   size = 1;
  while(size<n)
  {  L1 = 0; tur++;     k = 0;  printf("Tur Sayısı:%d\n",tur);    getch();
     while(L1+size<n)
    { L2 = L1+size;   u1 = L2-1;   u2 = (L2+size-1<n) ? L2+size-1 : n-1;
      for(i=L1,j=L2; i<=u1 && j<=u2; k++)
      { if(x[i]<=x[j]) aux[k]=x[i++];         else aux[k]=x[j++];
        printf("aux[]=%d\n",aux[k]);
      }
      for(;i<=u1;k++) { aux[k] = x[i++]; printf("aux[]=%d\n",aux[k]);}
      for(;j<=u2;k++) {aux[k] = x[j++];  printf("aux[]=%d\n",aux[k]);}
      L1 = u2+1;
     }
    for(i=L1;k<n;i++){  aux[k++] = x[i]; printf("aux[]=%d\n",aux[k]);}
    for(i=0;i<n;i++)     x[i]=aux[i];
    size*=2;
  }
}
void main() { int dizi[8] = { 25,57, 48, 37, 12, 33, 86,92 };
mergesort( &dizi[0], 8);   for(int i=0;i<8;i++)  printf("%d\n  ",dizi[i]);  }
```

MERGE SORT

- Analiz : $\log_2 n$ tur ve her turda n veya daha az karşılaştırma = $O(n \log_2 n)$ karşılaştırma.

- Quicksort'ta en kötü durumda $O(n^2)$ karşılaştırma gerektiği düşünülürse daha avantajlı. Fakat mergesort'ta atama işlemleri fazla ve dizi için daha fazla yer gerekiyor.

Yığın Sıralaması (HEAP SORT)

- Her düğümün çocuk düğümlerinin kendisinden küçük veya eşit olma kuralını esas alır.
- Sıralama yapısı; dizinin ilk elamanı her zaman en büyük olacaktır.
- Dizi üzerinde i. elemanla, çocukları 2i. ve (2i.+1) karşılaştırılıp büyük olan elemanlar yer değiştirilecektir.

HEAP SORT

- Dizinin son elamanları dizinin ortasındaki elamanların çocuk düğümü olacağından bu işlem dizinin yarısına kadar yapılır.
- Elde edilen diziyi sıralamak için ise dizinin ilk elamanı en büyük olduğu bilindiğinden dizinin son elamanıyla ilk elemanı yer değiştirilerek büyük elaman sona atılır.
- Bozulan diziye yukarıdaki işlemler tekrar uygulanır. Dizi boyutu 1 oluncaya kadar işleme devam edilir.

HEAP SORT

- Bu algoritmanın çalışma zamanı, O(nlogn)'dir.
- En kötü durumda en iyi performansı garanti eder.
- Fakat karşılaştırma döngülerinden dolayı yavaş çalışacaktır.

HEAP SORT

```
heapify(x, i)
{ L = Left(i); R = Right(i);
 if (L <= heapsize && x[L] > x[i])
    largest = L;
 else  largest = i;
 if (R <= heapsize && x[R] > x[largest])
    largest = R;
 if (largest != i)
  {temp = x[largest]; x[largest] = x[i];
   x[i] = temp;  heapify(x, largest);
  }
}
```

HEAP SORT

```
BuildHeap(dizi)
{
   heapsize = dizi.Length - 1;
   for (i = dizi.Length /2;i >= 0; i--)
   heapify(dizi, i);
}

Left(i)  { return 2 * (i + 1) - 1; }
Right(i) { return 2*(i+1); }
Parent(i){ return (i-1)/2;}
```

HEAP SORT

```
Heap_Sort(dizi)
{
BuildHeap(dizi);
    for (i = heapsize; i >=0; i--)
      {       temp = dizi[0];
               dizi[0] = dizi[i];
               dizi[i] = temp;
               heapsize--;
               heapify(dizi, 0);
          }
}
```

SIRALAMA ALGORİTMALARI-SHELL SORT

- Shell algoritması etkin çalışması ve kısa bir program olmasından dolayı günümüzde en çok tercih edilen algoritmalardan olmuştur. **h** adım miktarını kullanarak her defasında dizi elemanlarını karşılaştırarak sıralar.
- Insertion sort sıralamanın geliştirilmesiyle elde edilmiştir.
- Azalan artış sıralaması olarak da adlandırılır. Insertion Sort'un aksine ters sıralı dizi dağılımından etkilenmemektedir. Algoritmanın analizi zor olduğundan çalışma zamanları net olarak çıkarılamamaktadır. Aşağıda verilen h dizisi için $n^{1,5}$ karşılaştırmadan fazla karşılaştırma yapmayacağından dolayı $\theta(n^{1,25})$, $O(n^{1,5})$ olarak belirlenmiştir.
- Pratikte en kötü durum: **$O(n\log^2 n)$**

Kabuk sıralaması(SHELL SORT)

```
void shellsort() {
 int dizi[5] = {9, 5, 8, 3, 1 };   int n=5;  int h=1;
 while ((h*3+1<n)) h=3*h+1;
 while (h>0)
 {
  for(int i=h;i<n;i++)
  {  int b=dizi[i];  int j;

       for(j=i;(j>h-1)&&(dizi[j-h]>b);j-=h)
         dizi[j]=dizi[j-h];
       dizi[j]=b;
  }
  h/=3;
 }
 for(int i=0;i<5;i++)  printf("%d\n  ",dizi[i]);
}
 void main() { shellsort(); }
```

SIRALAMA ALGORİTMALARI-SHELL SORT

- 1.Tur, h=4, i=4
-
- 9 5 8 3 1
- 1 5 8 3 9

- 2.Tur h=1, i=1, i=2, i=3, i=4

- 1 5 8 3 9 1 5 3 8 9 1 3 5 8 9
- 1 5 3 8 9 1 3 5 8 9 1 3 5 8 9

Doğrusal Zamanda Sıralama Algoritmları

- Şimdiye dek gördüğümüz sıralama algoritmalarının zamana göre etkinlikleri $O(n^2)$ ya da $O(nlogn)$ ile ölçülüyordu.

- **Counting Sort** ve **Radix Sort** ise çoğunlukla bu ikisi arasında bir etkinliğe sahiptir.

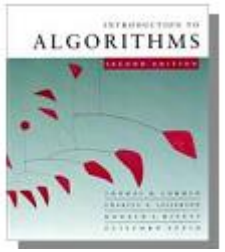
Sayma Sıralaması (Counting Sort)

- **Sayma sıralaması** algoritmasının en büyük özelliği, sıralama için elemanlar arası karşılaştırma yapılmamasıdır.
- **Giriş: A[1 . . n]**, burada **A[j]∈{1, 2, …, k} .**
- **k, küçük ise iyi bir algoritma olur, k, büyük ise çok kötü bir algoritma olur (nlogn daha kötü)**
- **Çıkış: B[1 . . n],** sıralı.
- **Yedek depolama: C[1 . . k]** .

Sayma Sıralaması (Counting Sort)

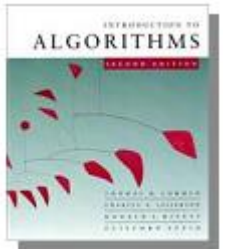
- Girişlerin **0** ile **k** arasında, ***n*** adet girişin tamsayı olduğu kabul edilir.
- Temel olarak bir x elamanı için kendisinden küçük elemanların sayısını bulmayı amaçlar.
- Girilen dizi boyutunda ve k boyutunda ek dizilere ihtiyaç vardır.

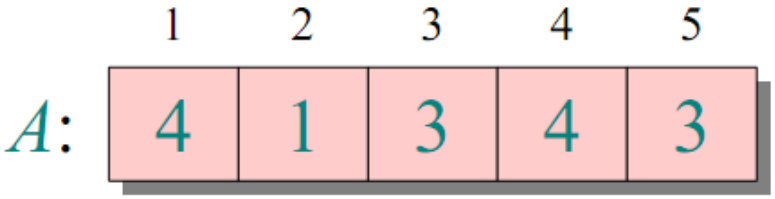
Sayma sıralaması

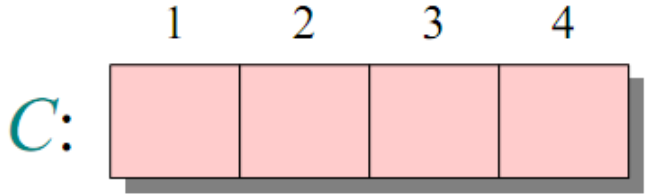


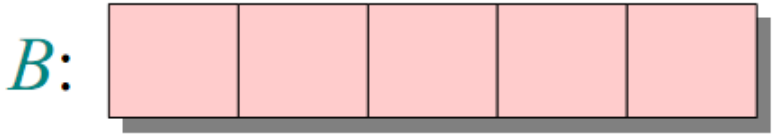

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1      ▷ C[i] = |{key = i}|
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]       ▷ C[i] = |{key ≤ i}|
for j ← n down to 1               (down to 1: 1'e inene kadar)
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] – 1
```

Sayma sıralaması



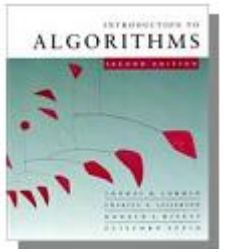






- Dizi girişi 1 ile 4 arasındadır. O zaman **k=4** olur.

Döngü 1



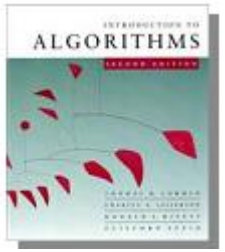
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	0

B:					

for $i \leftarrow 1$ **to** k
do $C[i] \leftarrow 0$

Döngü 2



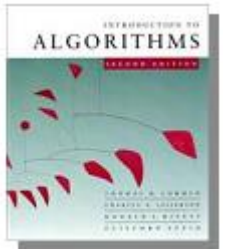
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	0	0	0	1

B:					

for $j \leftarrow 1$ **to** n
do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Döngü 2



	1	2	3	4	5
A:	4	1	3	4	3

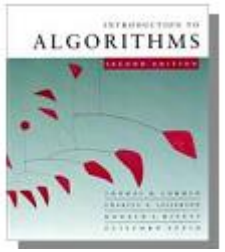
	1	2	3	4
C:	1	0	0	1

B:					

for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ $\quad \triangleright\ C[i] = |\{\text{key} = i\}|$

Döngü 2



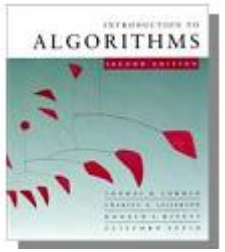
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	1	1

B:					

for $j \leftarrow 1$ **to** n
do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Döngü 2



	1	2	3	4	5
A:	4	1	3	4	3

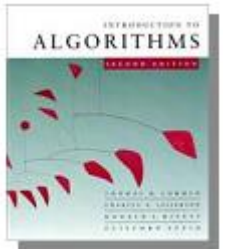
	1	2	3	4
C:	1	0	1	2

B:					

for $j \leftarrow 1$ **to** n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Döngü 2



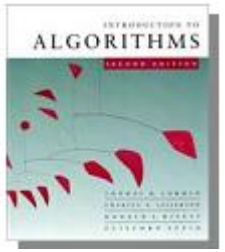
	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

B:					

for $j \leftarrow 1$ **to** n
do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

Döngü 3



	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

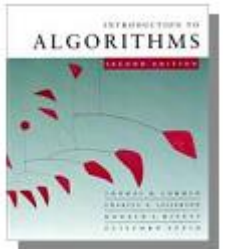
B:					

	1	2	3	4
C':	1	1	2	2

for $i \leftarrow 2$ **to** k

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

Döngü 3



	1	2	3	4	5
A:	4	1	3	4	3

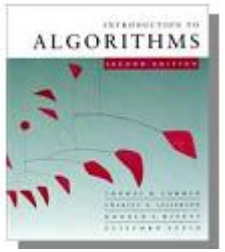
	1	2	3	4
C:	1	0	2	2

B:					

	1	2	3	4
C':	1	1	3	2

for $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

Döngü 3



	1	2	3	4	5
A:	4	1	3	4	3

	1	2	3	4
C:	1	0	2	2

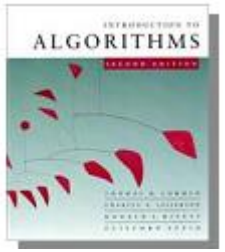
B:					

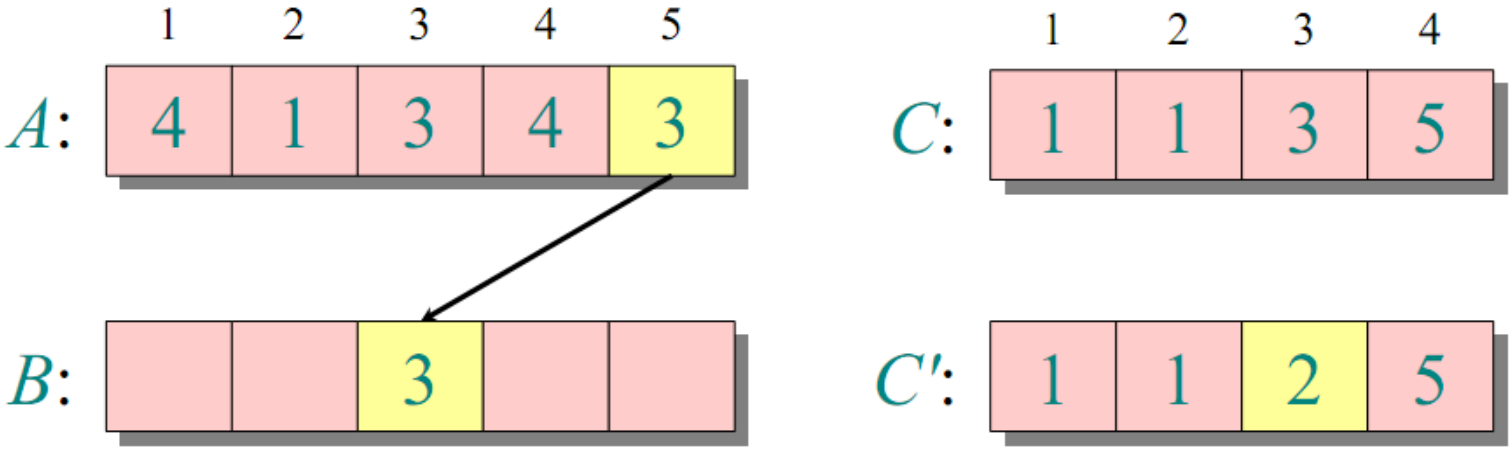
	1	2	3	4
C':	1	1	3	5

for $i \leftarrow 2$ **to** k

  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

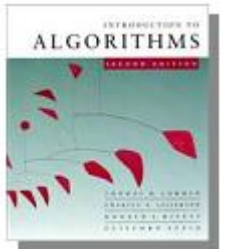
Döngü 4

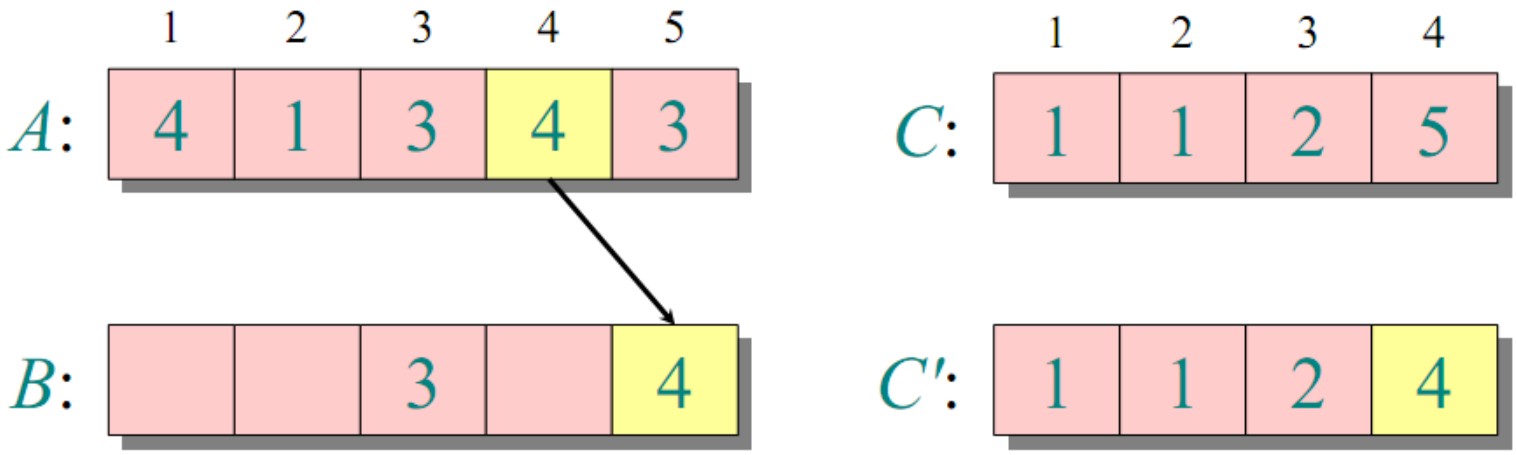




for $j \leftarrow n$ **down to** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

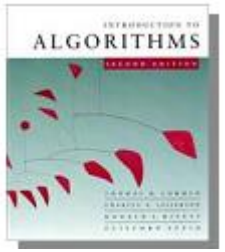
Döngü 4





for $j \leftarrow n$ **down to** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Döngü 4

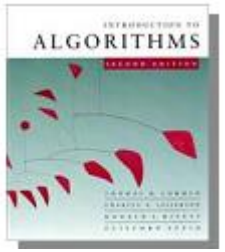


	1	2	3	4	5
A:	4	1	3	4	3
B:		3	3		4

	1	2	3	4
C:	1	1	2	4
C':	1	1	1	4

for $j \leftarrow n$ **down to** 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

Döngü 4

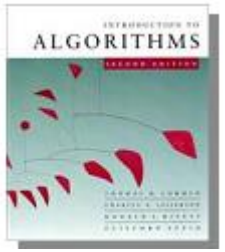


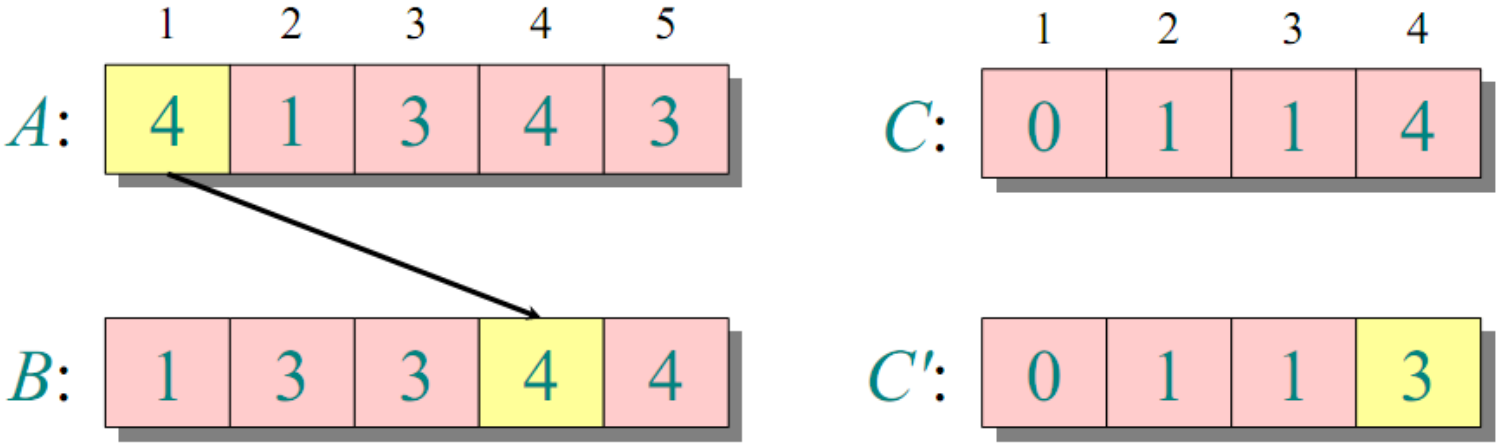
	1	2	3	4	5
A:	4	1	3	4	3
B:	1	3	3		4

	1	2	3	4
C:	1	1	1	4
C':	0	1	1	4

for $j \leftarrow n$ **down to** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

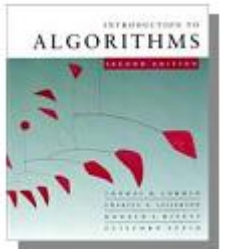
Döngü 4





for $j \leftarrow n$ **down to** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Çözümleme



$\Theta(k)$ { **for** $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow 0$

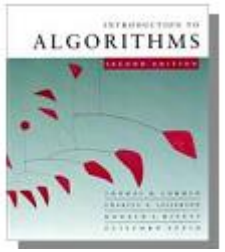
$\Theta(n)$ { **for** $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ { **for** $i \leftarrow 2$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$ { **for** $j \leftarrow n$ **down to** 1
  do $B[C[A[j]]] \leftarrow \mathrm{A}[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

Sayma sıralamanın artıları eksileri



- **k = O(n)** ise, sayma sıralaması **Θ(n)** süresi alır. Eğer **$k=n^2$** veya **$k=2^n$** çok kötü bir algoritma olur.
- **Artıları:**
  - **n** ve **k** da doğrusaldır (lineer).
  - Kolay uygulanır.
- **Eksileri:**
  - Yerinde sıralama yapmaz. Ekstra depolama alanına ihtiyaç duyar.
  - Sayıların küçük tam sayı olduğu varsayılır.
  - Sayılar int ise yani 32 bit lik sayılar ise k, maksimum 2^{32}= 4.2 milyar sayı eder oda yaklaşık 16 Gb yer tutar.

Basamağa göre sıralama (Radix Sort)

- Sayıları basamaklarının üzerinde işlem yaparak sıralayan bir sıralama algoritmasıdır.

- Her sıralama algoritmasında olduğu gibi, Radix Sort algoritmasının etkinliği de sıralanacak diziye bağlı olarak değişkenlik gösterir. Ortalama olarak, Etkinlik derecesi ***O(n)*** dir. Dolayısıyla, **$O(n^2)$** grubuna dahil olan sıralama algoritmalarından daha iyidir.

Basamağa göre sıralama (Radix Sort)

- Sayma sayıları adlar ya da tarihler gibi karakter dizilerini göstermek için kullanılabildiği için *basamağa göre sıralama* algoritması yalnızca sayma sayılarını sıralamak için kullanılan bir algoritma değildir.

- Çoğu bilgisayar veri saklamak için ikilik tabandaki sayıların elektronikteki gösterim biçimlerini kullandığı için sayma sayılarının basamaklarını ikilik tabandaki sayılardan oluşan öbekler biçiminde göstermek daha kolaydır.

Basamağa göre sıralama (Radix Sort)

- Basamağa göre sıralama algoritması *en anlamlı basamağa göre sıralama* ve *en anlamsız basamağa göre sıralama* olarak ikiye ayrılır.
- ***En anlamsız basamağa*** *(*Least significant digit) *göre sıralama* algoritması sayıları en anlamsız (en küçük, en sağdaki) basamaktan başlayıp en anlamlı basamağa doğru yürüyerek sıralarken *en anlamlı basamağa göre sıralama* bunun tam tersini uygular.
- Sıralama algoritmaları tarafından işlenen ve kendi sayı değerlerini gösterebildiği gibi başka tür verilerle de eşleştirilebilen sayma sayılarına çoğu zaman "anahtar" denir.

Basamağa göre sıralama (Radix Sort)

- En anlamsız basamağa göre sıralamada kısa anahtarlar uzunlardan önce gelirken aynı uzunluktaki anahtarlar sözlükteki sıralarına göre sıralanırlar. Bu sıralama biçimi sayma sayılarının kendi değerlerine göre sıralandıklarında oluşan sırayla aynı sırayı oluşturur.

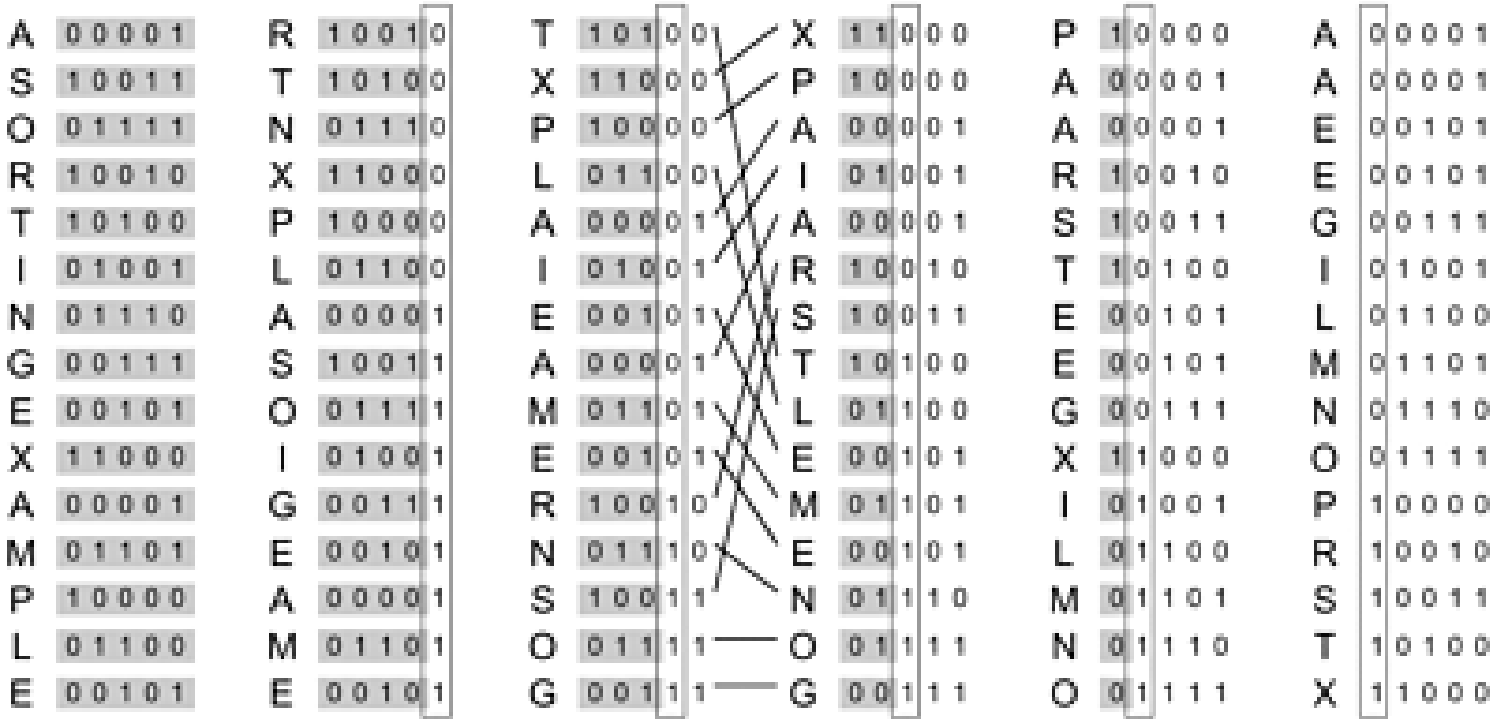
- Örneğin 1'den 10'a kadar olan sayılar sıralandığında ortaya 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 dizisi çıkacaktır.

Basamağa göre sıralama (Radix Sort)

- **En anlamlı basamağa göre sıralama** sözcükler ya da aynı uzunluktaki sayılar gibi dizgileri sıralamak için uygun olan sözlükteki sıraya göre sıralar.

- Örneğin "b, c, d, e, f, g, h, i, j, ba" dizisi sözlük sırasına göre "b, ba, c, d, e, f, g, h, i, j" olarak sıralanacaktır. Eğer sözlük sırası değişken uzunluktaki sayılarda uygulanırsa sayılar değerlerinin gerektirdiği konumlara konulmazlar.

Basamağa göre sıralama (Radix Sort)



Basamağa göre sıralama (Radix Sort)

- Yukarıdaki sayıları 2 tabanına göre yazalım:
- 57 111001
- 43 101011
- 24 110000
- 70 1000110
- 111 1101111
- 102 1100110
- 213 11010101
- 37 100101
- 44 101100
- 23 10111

Basamağa göre sıralama (Radix Sort)

- Şimdi sırayla şu işleri yapalım:
- İlk geçişte sayıları 2^0 (birler) basamağına (en sağdaki basamak) göre,
- ikinci geçişte öncekini 2^1 (ikiler) basamağına göre,
- üçüncü geçişte öncekini 2^2 (dörtler) basamağına göre,
- dördüncü geçişte öncekini 2^3 (sekizler) basamağına göre,
- beşinci geçişte öncekini 2^4 (onaltılar) basamağına göre,
- altıncı geçişte öncekini 2^5 (otuzikiler) basamağına göre,
- yedinci geçişte öncekini 2^6 (altmış dörtler) basamağına göre,
- sekizinci geçişte öncekini 2^7 (yüz yirmi sekizler) basamağına göre sıralayalım.
- Son geçişte dizi sıralanmış olur.

Basamağa göre sıralama (Radix Sort)

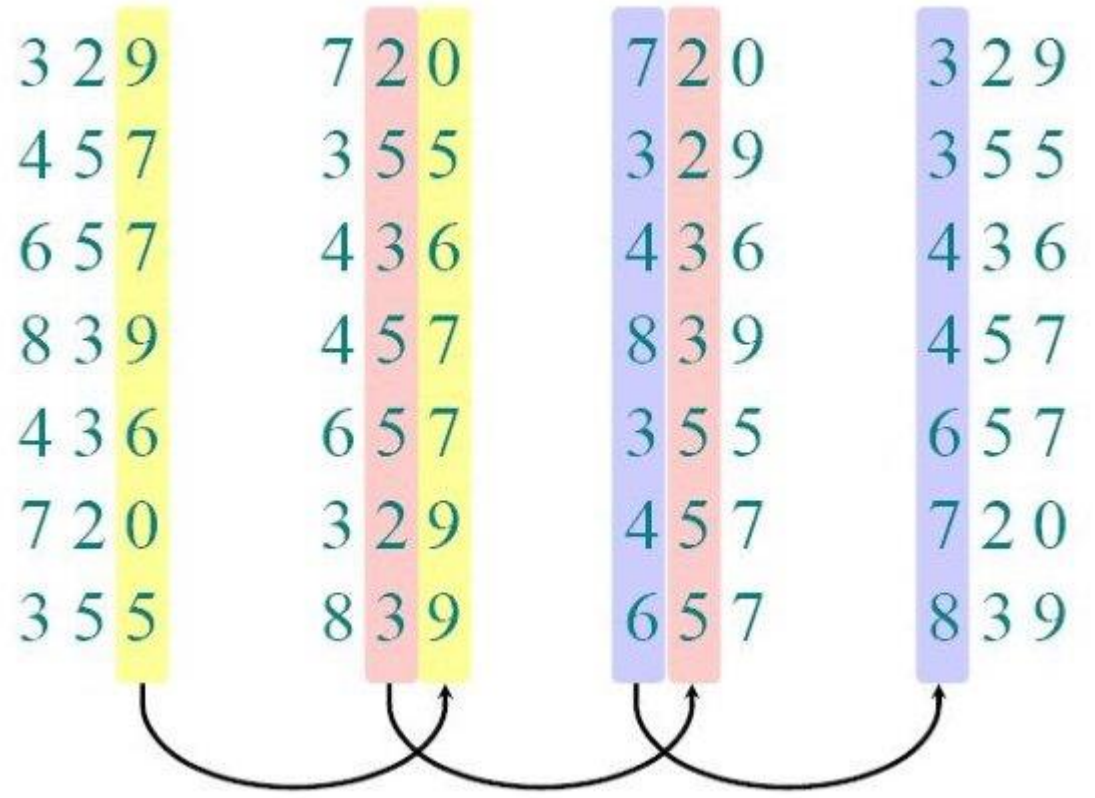
- Bu geçişler aşağıdaki tabloda gösterilmiştir.

2^0 (birler) basamağına göre sıralama		2^1 (ikiler) basamağına göre sıralama		2^2 (dörtler) basamağına göre sıralama		2^3 (sekizler) basamağına göre sıralama	
24	11000	24	11000	24	11000	213	11010101
70	1000110	44	101100	57	111001	37	100101
102	1100110	57	111001	43	101011	70	1000110
44	101100	213	11010101	44	101100	102	1100110
57	111001	37	100101	213	11010101	23	10111
43	101011	70	1000110	37	100101	24	11000
111	1101111	102	1100110	70	1000110	57	111001
213	11010101	43	101011	102	1100110	43	101011
37	100101	111	1101111	111	1101111	44	101100
23	10111	23	10111	23	10111	111	1101111

2^4 (onaltılar) basamağına göre sıralama		2^5 (otuzikiler) basamağına göre sıralama		2^6 (altmışdörtler) basamağına göre sıralama		2^7 (yüz yirmi sekizler basamağına göre sıralama	
37	100101	70	1000110	23	10111	23	10111
70	1000110	23	10111	24	11000	24	110000
102	1100110	24	11000	37	100101	37	100101
43	101011	213	11010101	43	101011	43	101011
44	101100	37	100101	44	101100	44	101100
111	1101111	102	1100110	57	111001	57	111001
23	10111	43	101011	70	1000110	70	1000110
24	11000	44	101100	213	11010101	102	1100110
213	11010101	111	1101111	102	1100110	111	1101111
57	111001	57	111001	111	1101111	213	11010101

Basamağa göre sıralama (Radix Sort)

- Örnek:



Basamağa göre sıralama (Radix Sort)

- Örneğin 10'luk sayı tabanında her sayıdan birer tane bulunması halinde her hane için 10 ihtimal bulunur. Bu her ihtimalin ayrı bir hafıza bölümünde (örneğin bir dizi veya bağlı liste) tutulması durumunda sıralama işlemi **en büyük hane sayısı * n** olmaktadır.

- Örneğin 3 haneli sayılar için **O(3n) ~ O(n)** olmaktadır. Bu değer zaman verimliliği (time efficiency) arttırırken hafıza verimliliğini (memory efficiency) azaltmaktadır.
- En kötü etkinlik zamanı **$O(n^2)$** ile **0(nlogn)** arası

Radix Sort

```
public void RadixSort(int[] Dizi)    {
        // Yardımcı dizimiz
        int[] t = new int[Dizi.Length];
        // her defasında kaç bit işleme alınacak
        int r = 4; // 2, 8 veya 16 bit ile kaç geçiş yapılacağı denenebilir
        // int 4 byte yani 32 bit
        int b = 32;
        // counting ve prefix dizileri
        // (unutmayın bu dizlerin boyutu 2^r düzeyinde olacaktır.
        // Bu değerin uygun seçilmesi gerekir r=4 için dizi boyutu 16
        // r=16 için dizi boyutu 65535)
        int[] count = new int[1 << r];
        int[] pref = new int[1 << r];
        // grupların sayısı yani geçiş  bulunuyor
        int groups = (int)Math.Ceiling((double)b / (double)r);
        // gruplara uygulanacak maske
        int mask = (1 << r) - 1; //r=4 için 15
```

Radix Sort

```
for (int c = 0, shift = 0; c < groups; c++, shift += r)
{   // count dizisini resetleme
     for (int j = 0; j < count.Length; j++)  count[j] = 0;
          // c inci grubun elamanlarının sayılması
     for (int i = 0; i < Dizi.Length; i++)
         count[(Dizi[i] >> shift) & mask]++;
     // prefix dizisinin hesaplanması
     pref[0] = 0;
     for (int i = 1; i < count.Length; i++)
         pref[i] = pref[i - 1] + count[i - 1];
   // t[]  dizisine c. inci grupta sıralanmış elamanların indisine
   //uygun değerin Dizi den atanması
     for (int i = 0; i < Dizi.Length; i++)
         t[pref[(Dizi[i] >> shift) & mask]++] = Dizi[i];
// Dizi[]=t[] c inci guruba göre sıralanmış değerlerin diziye aktarılması
         t.CopyTo(Dizi, 0);
     }
       // Dizi sıralı
}
```

```
Çıktı: 32/4= 8 geçiş
256 , 1 , 120 , 10 , 235 , 987
256 , 1 , 10 , 120 , 987 , 235
1 , 10 , 120 , 235 , 256 , 987
1 , 10 , 120 , 235 , 256 , 987
1 , 10 , 120 , 235 , 256 , 987
1 , 10 , 120 , 235 , 256 , 987
1 , 10 , 120 , 235 , 256 , 987
1 , 10 , 120 , 235 , 256 , 987
```

SIRALAMA ALGORİTMALARI

Çalışma Süresi	Sorting Algoritması-en iyi
$O(n^2)$	BubleSort SelectionSort InsertionSort ShellSort
O(nlogn)	HeapSort MergeSort QuickSort
O(n)	Counting Sort RadixSort BucketSort

Ödev

- 1-Verilen sıralama algoritmalarının C# veya Java programlarını yazınız. Algoritmaları tur olarak programda karşılaştırınız.

- 2- Kullanılan diğer sıralama algoritmaları hakkında bilgi toplayıp program kodları ile birlikte slayt hazırlayınız.

Basamağa göre sıralama (Radix Sort)

```
import java.lang.*;
import java.io.*;

public class RadixSort {

      public static void radixSort(int[] arr) {
            if (arr.length == 0)
                  return;
            int[][] np = new int[arr.length][2];
            int[] q = new int[0x100];
            int i, j, k, l, f = 0;
            for (k = 0; k < 4; k++) {
                  for (i = 0; i < (np.length - 1); i++)
                        np[i][1] = i + 1;
                  np[i][1] = -1;
                  for (i = 0; i < q.length; i++)
                        q[i] = -1;
                  for (f = i = 0; i < arr.length; i++) {
                        j = ((0xFF << (k << 3)) & arr[i]) >> (k << 3);
                        if (q[j] == -1)
                              l = q[j] = f;
                        else {
                              l = q[j];
                              while (np[l][1] != -1)
```

Char type,

0 in decimal is 0x0 in hexadecimal
7 in decimal is 0x7 in hexadecimal
8 in decimal is 0x8 in hexadecimal
9 in decimal is 0x9 in hexadecimal
10 in decimal is 0xa in hexadecimal
11 in decimal is 0xb in hexadecimal
12 in decimal is 0xc in hexadecimal
13 in decimal is 0xd in hexadecimal
14 in decimal is 0xe in hexadecimal
15 in decimal is 0xf in hexadecimal
16 in decimal is 0x10 in hexadecimal
17 in decimal is 0x11 in hexadecimal
31 in decimal is 0x1f in hexadecimal
32 in decimal is 0x20 in hexadecimal
255 in decimal is 0xff in hexadecimal
256 in decimal is 0x100 in hexadecimal

```
                                l = np[l][1];
                            np[l][1] = f;
                            l = np[l][1];
                        }
                        f = np[f][1];
                        np[l][0] = arr[i];
                        np[l][1] = -1;
                }
                for (l = q[i = j = 0]; i < 0x100; i++)
                        for (l = q[i]; l != -1; l = np[l][1])
                                arr[j++] = np[l][0];
        }
}

public static void main(String[] args) {
        int i;
        int[] arr = new int[15];
        System.out.print("Sırasız dizi: ");
        for (i = 0; i < arr.length; i++) {
                arr[i] = (int) (Math.random() * 1024);
                System.out.print(arr[i] + " ");
        }
        radixSort(arr);
        System.out.print("\nSıralı dizi : ");
        for (i = 0; i < arr.length; i++)
                System.out.print(arr[i] + " ");

}
}
```

Basamağa göre sıralama (Radix Sort) JAVA

- RadixSort.java

```
/* Copyright (c) 2012 the authors listed at the following URL, and/or
the authors of referenced articles or incorporated external code:
http://en.literateprograms.org/Radix_sort_(Java)?action=history&offset=20080201073641
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
Retrieved from: http://en.literateprograms.org/Radix_sort_(Java)?oldid=12461
*/
```

Basamağa göre sıralama (Radix Sort) JAVA

```
import java.util.Arrays;
class RadixSort
{   public static void radix_sort_uint(int[] a, int bits)
  {    int[] b = new int[a.length];       int[] b_orig = b;       int rshift = 0;
       for (int mask = ~(-1 << bits); mask != 0; mask <<= bits, rshift += bits)    {
          int[] cntarray = new int[1 << bits];
          for (int p = 0; p < a.length; ++p)
          {   int key = (a[p] & mask) >> rshift;              ++cntarray[key];          }

        for (int i = 1; i < cntarray.length; ++i)         cntarray[i] += cntarray[i-1];
          for (int p = a.length-1; p >= 0; --p)
        {  int key = (a[p] & mask) >> rshift;   --cntarray[key];   b[cntarray[key]] = a[p];    }
          int[] temp = b; b = a; a = temp;
       }
        if (a == b_orig)          System.arraycopy(a, 0, b, 0, a.length);
     }
```

Basamağa göre sıralama (Radix Sort) JAVA

```
public static void main(String[] args)
    {
        int[] a = {
            123,432,654,3123,654,2123,543,131,653,123,
            533,1141,532,213,2241,824,1124,42,134,411,
            491,341,1234,527,388,245,1992,654,243,987};

        System.out.println("Before radix sort:");
        System.out.println(Arrays.toString(a));

        radix_sort_uint(a, 4);

        System.out.println("After radix sort:");
        System.out.println(Arrays.toString(a));
    }
}
```

Basamağa göre sıralama (Radix Sort)- C++

```
#define NUMELTS 100
# include<stdio.h>
#include<conio.h>
#include<math.h>
void radixsort(int a[],int);
void main() {  int n,a[20],i;
printf("enter the number :"); scanf("%d",&n);
printf("ENTER THE DATA -");
for(i=0;i<n;i++) { printf("%d. ",i+1); scanf("%d",&a[i]); }
radixsort(a,n); getch(); }
```

Basamağa göre sıralama (Radix Sort)

- void radixsort(int a[],int n) {
- int rear[10],front[10],first,p,q,k,i,y,j,exp1; double exp;
- struct { int info; int next; } node[NUMELTS];
- for(i=0;i<n-1;i++)
- { node[i].info=a[i]; node[i].next=i+1; }
- node[n-1].info=a[n-1];
- node[n-1].next=-1;
- first=0;

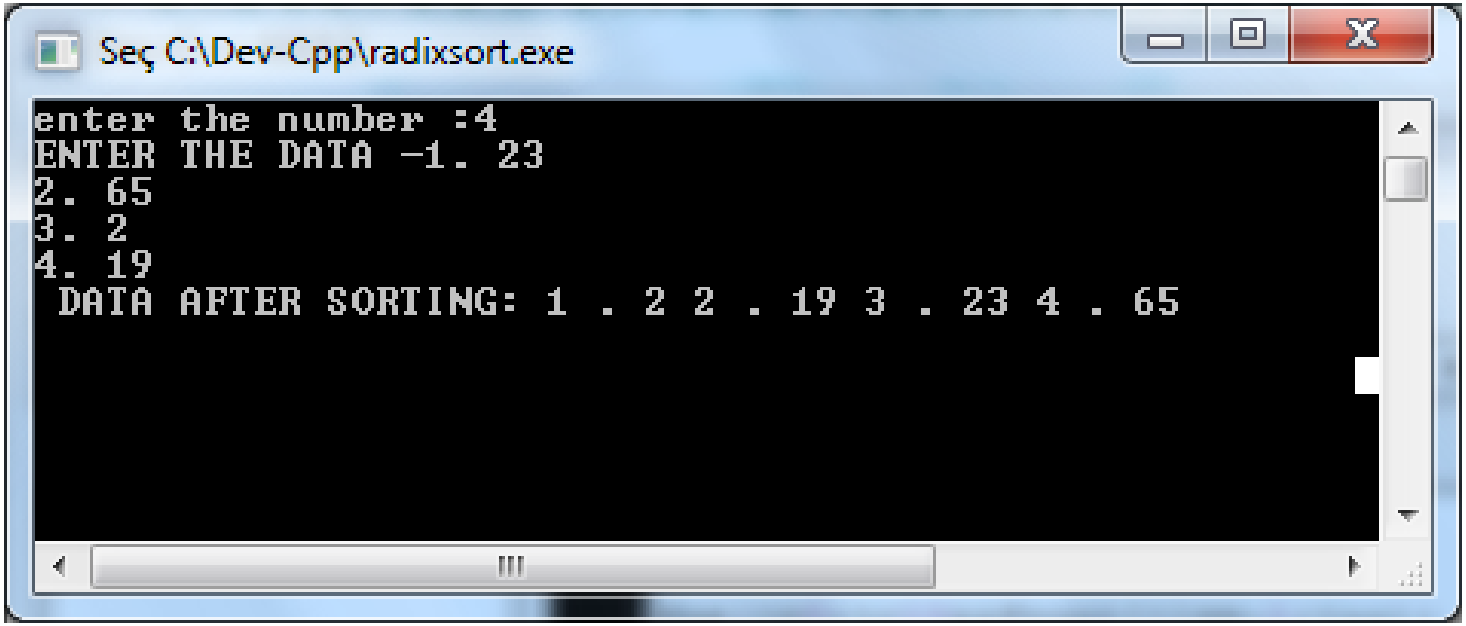
Basamağa göre sıralama (Radix Sort)

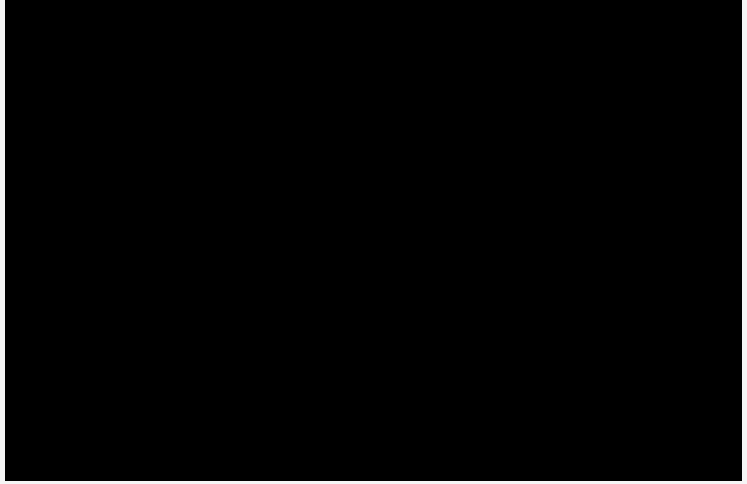
```
for(k=1;k<=2;k++) //consider only 2 digit number
{   for(i=0;i<10;i++)
    {  front[i]=-1;  rear[i]=-1;  }
while(first!=-1) {
p=first;              first=node[first].next;
y=node[p].info;  exp=pow(10,k-1);
exp1=(int)exp;   j=(y/exp1)%10;
q=rear[j];
if(q==-1)  front[j]=p;
else
node[q].next=p;
rear[j]=p;  }
```

Basamağa göre sıralama (Radix Sort)

- for(j=0;j<10&&front[j]==-1;j++)
- first=front[j];
- while(j<=9) {
- for(i=j+1;i<10&&front[i]==-1;i++)
- if(i<=9) { p=i; node[rear[j]].next=front[i]; }
- j=i; }
- node[rear[p]].next=-1; }
- //copy into original array
- for(i=0;i<n;i++) {a[i]=node[first].info; first=node[first].next;}
- printf(" DATA AFTER SORTING:");
- for(i=0;i<n;i++) printf(" %d . %d",i+1,a[i]); }

Basamağa göre sıralama (Radix Sort)



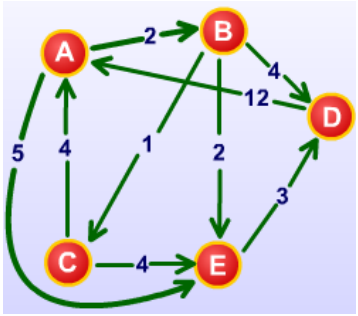


ÇİZGİ KÜMELERİ (GRAPHS)

GRAFLAR

- **Tanım**
  - Yönlendirilmiş ve yönlendirilmemiş graflar
  - Ağırlıklı graflar
- **Gösterim**
  - Komşuluk Matrisi
  - Komşuluk Listesi
- **Dolaşma Algoritmaları**
  - BFS (Breath First Search)
  - DFS (Depth-First Search)

GRAFLAR



- Graf, matematiksel anlamda, düğümlerden ve bu düğümler arasındaki ilişkiyi gösteren kenarlardan oluşan bir kümedir. Mantıksal ilişki, düğüm ile düğüm veya düğüm ile kenar arasında kurulur.
  - Bağlantılı listeler ve ağaçlar grafların özel örneklerindendir.
- Fizik, Kimya gibi temel bilimlerde ve mühendislik uygulamalarında ve tıp biliminde pek çok problemin çözümü ve modellenmesi graflara dayandırılarak yapılmaktadır.

GRAFLAR-Uygulama alanları

- **Elektronik devreler**
  - Baskı devre kartları (PCB), Entegre devreler
- **Ulaşım ağları**
  - Otoyol ağı, Havayolu ağı
- **Bilgisayar ağları**
  - Lokal alan ağları ,İnternet
- **Veritabanları**
  - Varlık-ilişki (Entity-relationship) diyagramı

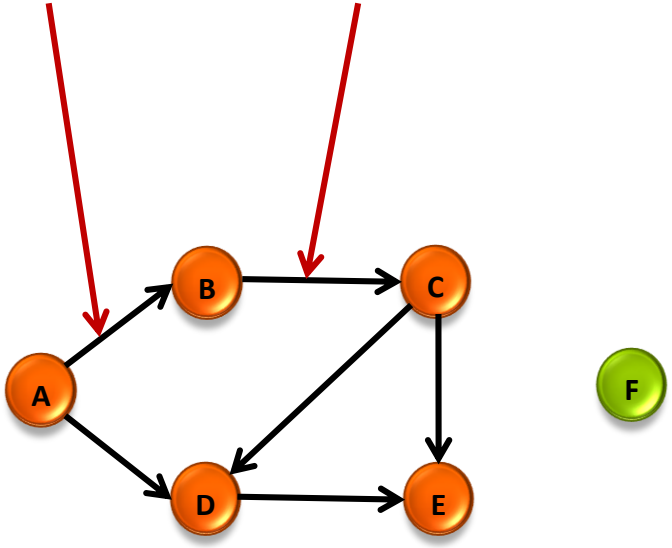
GRAFLAR

- Bir G grafı V ile gösterilen düğümlerden (node veya vertex) ve E ile gösterilen kenarlardan (Edge) oluşur.
  - Her kenar iki düğümü birleştirilerek iki bilgi (düğüm) arasındaki ilişkiyi gösterir ve ***(u,v)*** şeklinde ifade edilir. ***(u,v)*** iki düğümü göstermektedir.

- Bir graf üzerinde ***n*** tane düğüm ve ***m*** tane kenar varsa, matematiksel gösterilimi, düğümler ve kenarlar kümesinden elamanların ilişkilendirilmesiyle yapılır:
  - **$V = \{ v_0, v_1, v_2 \dots v_{n-1}, v_n \}$** Düğümler kümesi
  - **$E = \{ e_0, e_1, e_2 \dots e_{m-1}, e_m \}$** Kenarlar kümesi
  - **$G = (V, E)$** Graf

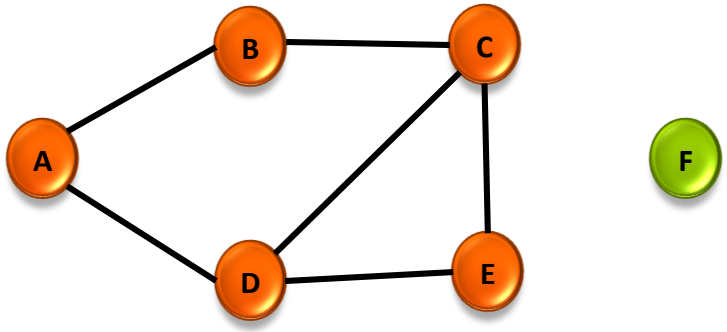
Graflar

- G = (V, E) yönlü grafına ait düğümler ve kenarlar aşağıda verilmiştir.
  - V = {A, B, C, D, E, F}
  - E = {(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)}



Graflar – Tanımlar

- **Komşu(Adjacent):** Bir ***G*** grafı üzerindeki v_i ve v_j adlı iki düğüm, kenarlar kümesinde bulunan bir kenarla ilişkilendiriliyorsa bu iki düğüm birbirine **komşu** (**adjacent**, neighbor) düğümlerdir; $e = \{ v_i, v_j \}$ şeklinde gösterilir ve ***e*** kenarı hem v_i hem de v_j düğümleriyle **bitişiktir** (**incident**) denilir.
- Diğer bir deyişle ***e*** kenarı v_i ve v_j düğümlerini birbirine bağlar veya v_i ve v_j düğümleri ***e*** kenarının uç noktalarıdır denilir.
  - (A, B) komşudur.
  - (B, D), (C, F) komşu değildir.



Graflar – Tanımlar

- **Komşuluk ve Bitişiklik:** Bir ***G*** grafı komşuluk ilişkisiyle gösteriliyorsa $G_{vv}=\{(v_i, v_j)...\}$, bitişiklik ilişkisiyle gösteriliyorsa $G_{ve}=\{(v_i, e_j)...\}$ şeklinde yazılır. (G_{vv}: $G_{düğümdüğüm}$, G_{ve}: $G_{düğümkenar}$)
  - Örneğin, şekil a) 2-düğümlü basit graf $G_{vv}=\{(v_0, v_1)\}$ veya $G_{ve}=\{(v_0, e_0), (v_1, e_0)\}$ şeklinde yazılabilir. (şekilde düğümler d, kenarlar k ile gösterilmiştir.)
  - b) verilen 4-düğümlü basit grafta $G_{vv}=\{(v_0, v_0), (v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_2), (v_2, v_3)\}$ veya $G_{ve}=\{(v_0, e_0), (v_0, e_1), (v_0, e_2), (v_0, e_3), (v_1, e_1), (v_1, e_4), (v_2, e_2), (v_2, e_4), (v_2, e_5), (v_3, e_3), (v_3, e_5)\}$ şeklinde yazılır. (şekilde düğümler d, kenarlar k ile gösterilmiştir.)



Graflar – Tanımlar

- **Yönlendirilmiş Graf (Directed Graphs):** Bir G grafı üzerindeki kenarlar bağlantının nereden başlayıp nerede sonlandığını belirten yön bilgisine sahip ise yönlü-graf veya yönlendirilmiş graf (directed graf) olarak adlandırılır. Yön graf üzerinde → ile gösterilir.

- Yönlü graflar, matematiksel olarak gösterilirken her bir ilişki oval parantezle değil de **<>** karakter çiftiyle gösterilir.

- **Yönlendirilmemiş Graf (Undirected Graphs)**
  - Hiçbir kenarı yönlendirilmemiş graftır.

Graflar – Tanımlar

- **Yönlendirilmiş Kenar (Directed Edge)**
  - Sıralı kenar çiftleri ile ifade edilir.
    - (u, v) ile (v, u) aynı değildir.
  - İlk kenar orijin ikinci kenar ise hedef olarak adlandırılır.

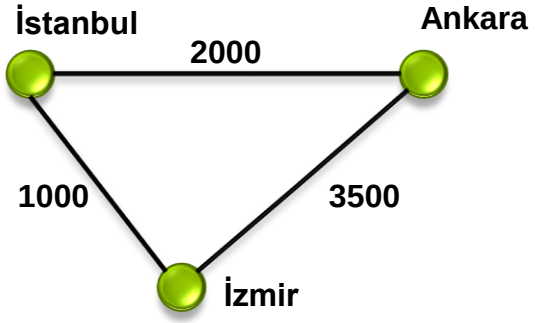
- **Yönlendirilmemiş Kenar (Undirected Edge)**
  - Sırasız kenar çiftleri ile ifade edilir.
    - (u, v) ile (v, u) aynı şeyi ifade ederler.

Graflar – Tanımlar

- **Ağırlıklı Graf(Weighted Graphs) :**

- Graf kenarları üzerinde ağırlıkları olabilir. Eğer kenarlar üzerinde ağırlıklar varsa bu tür graflara ağırlıklı/maliyetli graf denir. Eğer tüm kenarların maliyeti 1 veya birbirine eşitse maliyetli graf olarak adlandırılmaz; yön bilgisi de yoksa basit graf olarak adlandırılır.

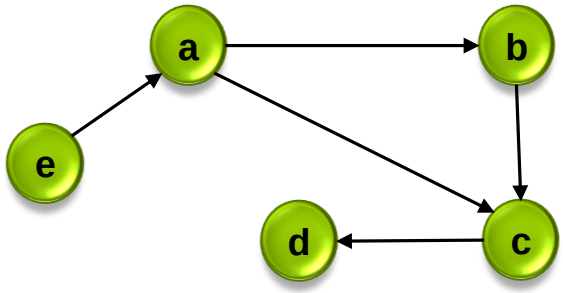
  - Ağırlık uygulamadan uygulamaya değişir.
    - Şehirler arasındaki uzaklık.
    - Routerler arası bant genişliği
      - Router: Yönlendirici



Graflar – Tanımlar

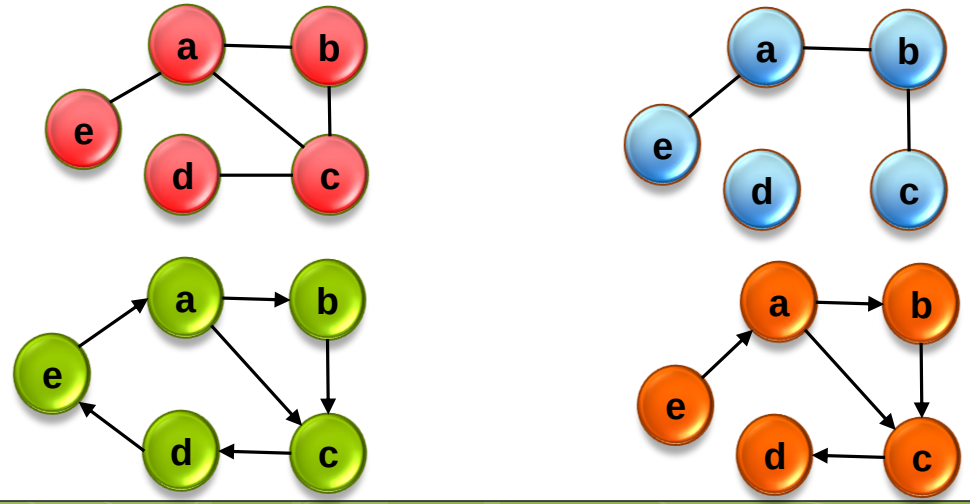
- **Yol (path) :** Bir çizgi kümesinde yol, düğümlerin sırasıdır. Kenar uzunluğu, yolun uzunluğunu verir.
  - $V_1, V_2, V_3 \ldots\ldots\ldots V_n$ ise yol $n - 1$ dir.
  - Eğer yol kenar içermiyorsa o yolun uzunluğu sıfır'dır.
  - İzmir'den Ankara'ya doğrudan veya İstanbul'dan geçerek gidilebilir (İzmir- İstanbul- Ankara).

- **Basit Yol (Simple Path) :** Bir yolda ilk ve son düğümler dışında tekrarlanan düğüm yoksa basit yol diye adlandırılır.
  - eabcd basit yol'dur fakat eabcabcd veya abcabca basit yol değildir.



Graflar – Tanımlar

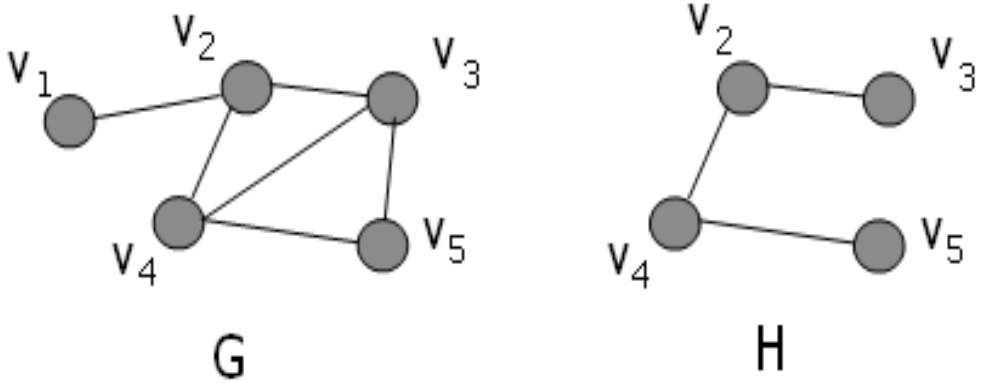
- **Uzunluk :** Bir yol üzerindeki kenarların uzunlukları toplamı o yolun uzunluğudur.
- **Bağlı veya Bağlı olmayan Çizge(Connected Graph):**
- Eğer bir graftaki tüm düğümler arasında en azından bir yol varsa bağlı graftır. Eğer bir grafta herhangi iki düğüm arasında yol bulunmuyorsa bağlı olmayan graftır.
- **a) Bağlı Graf** **b) Bağlı olmayan Graf**



Graflar – Tanımlar

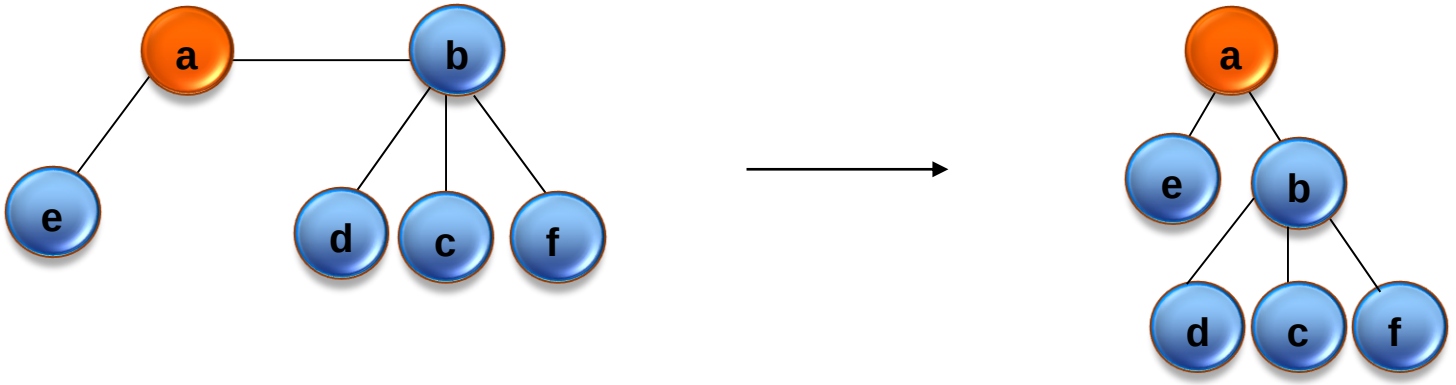
- **Alt Graf (Subgraph) :** H çizgesinin köşe ve kenarları G çizgesinin köşe ve kenarlarının alt kümesi ise; H çizgesi G çizgesinin alt çizgesidir (subgraph).

- G (V, E) şeklinde gösterilen bir grafın, alt grafı H(U, F) ise U alt küme V ve F alt küme E olur.



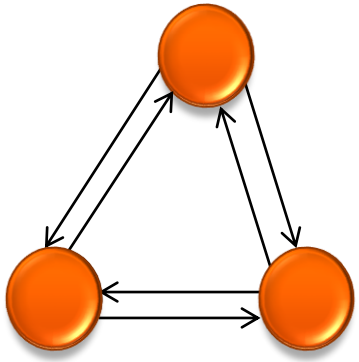
Graflar – Tanımlar

- **Ağaçlar(Trees) :** Ağaçlar özel bir çizgi kümesidir. Eğer direk olmayan bir çizgi kümesi devirli (cycle) değilse ve bağlıysa (connected) ağaç olarak adlandırılır.

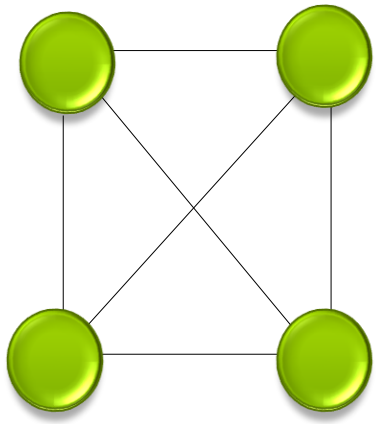


Graflar – Tanımlar

- **Tam Graf (Complete Graph) ve DiGraf :**
- Tam bir graf, farklı düğümlerin her bir çifti tek bir kenar ile bağlı olduğu basit bir yönsüz graftır. Yönlendirilmemiş grafta **n** düğüm var ise **n*(n-1)/2** kenar olmalıdır.
- Digraf ise farklı düğümlerin her bir çifti özgün kenarların her bir çifti (her iki yönde) ile bağlı olduğu yönlendirilmiş graftır. Digraf ta **n** düğüm var ise **n*(n-1)** kenar olmalıdır.



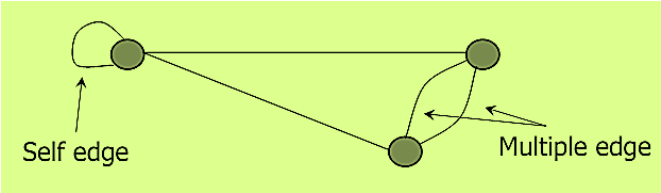
- 3 düğüm, 6 kenar (yönlü digraf)



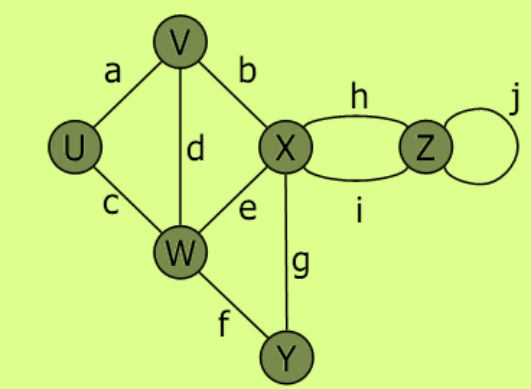
4 düğüm 6 kenar (yönsüz tam graf)

Graflar – Tanımlar

- **Multigraf:** Multigraf iki node arasında birden fazla kenara sahip olan veya bir node'un kendi kendisini gösteren kenara sahip olan graftır.



- Örnek
- a, b ve d kenarları V node'unun kenar bağlantılarıdır.
- X node'unun derecesi 5' tir.
- h ve i çoklu (multiple) kenarlardır.
- j kendi kendisine döngüdür (self loop)



Graflar – Tanımlar

- **Düğüm Derecesi (Node Degree):**

- Düğüme bağlı toplam uç sayısıdır; çevrimli kenarlar aynı düğüme hem çıkış hem de giriş yaptığı için dereceyi iki arttırır.

- Yönlü graflarda, düğüm derecesi giriş derecesi (input degree) ve çıkış derecesi (output degree) olarak ayrı ayrı belirtilir.

Graflar – Tanımlar

- **Komşuluk Matrisi (Adjacency Matrice):**
  Düğümlerden düğümlere olan bağlantıyı gösteren bir kare matrisdir; komşuluk matrisinin elemanları ki değerlerinden oluşur. Komşuluk matrisi $\mathbf{G_{vv}}$'nin matrisel şekilde gösterilmesinden oluşur. Eğer komşuluk matrisi $\mathbf{G_{vv}=[a_{ij}]}$ ise,

- yönlü-maliyetsiz graflar için; $a_{ij} = \begin{cases} 1, \text{Eğer } (v_i, v_j) \in K \text{ ise} \\ 0, \text{Diğer durumlarda} \end{cases}$
- olur.

- basit (yönsüz-maliyetsiz) graflar için ise,
- olur.

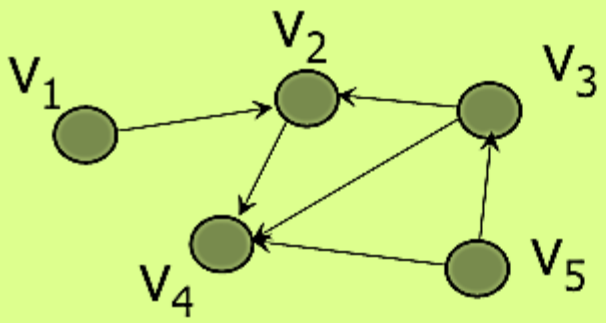
$$a_{ij} = \begin{cases} 1, \text{Eğer } (v_i, v_j) \in K \text{ ise veya } (v_j, v_i) \\ 0, \text{Diğer durumlarda} \end{cases}$$

Komşuluk Matrisi

- Yönlendirilmiş graf için komşu matrisi

Matris[i][j] = 1 if $(v_i, v_j) \in E$

0 if $(v_i, v_j) \notin E$



G

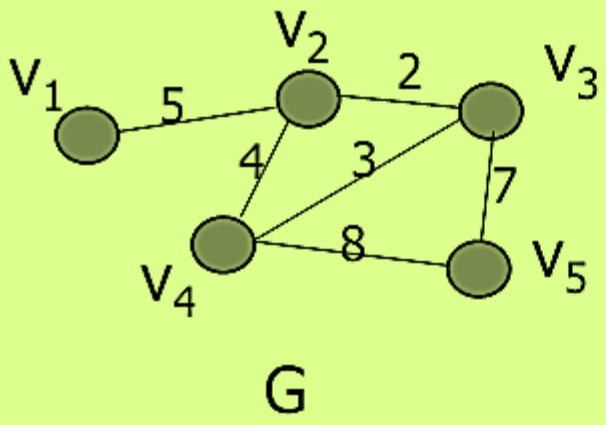
		1 (v_1)	2 (v_2)	3 (v_3)	4 (v_4)	5 (v_5)
1	v_1	0	1	0	0	0
2	v_2	0	0	0	1	0
3	v_3	0	1	0	1	0
4	v_4	0	0	0	0	0
5	v_5	0	0	1	1	0

Komşuluk Matrisi

- Ağırlıklandırılmış ancak yönlendirilmemiş graf için komşu matrisi

Matris[i][j] = $w(v_i, v_j)$ if $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$

∞ otherwise



G

		1 (v_1)	2 (v_2)	3 (v_3)	4 (v_4)	5 (v_5)
1	v_1	∞	5	∞	∞	∞
2	v_2	5	∞	2	4	∞
3	v_3	∞	2	∞	3	7
4	v_4	∞	4	3	∞	8
5	v_5	∞	∞	7	8	∞

Graflar – Tanımlar

- **Bitişiklik Matrisi (Incedence Matrice):**
- Düğümlerle kenarlar arasındaki bağlantı/bitişiklik ilişkisini gösteren bir matrisdir; matrisin satır sayısı düğüm, sütun sayısı kenar sayısına kadar olur. Bitişiklik matrisi G_{ve}'nin matrisel şekilde gösterilmesinden oluşur. Eğer bitişiklik matrisi $G_{ve}=[m_{ij}]$ ise, maliyetsiz graflar için,

$$m_{ij} = \begin{cases} 1, \text{ Eğer } (v_i, v_j) \text{ bağlantı var ise} \\ 0, \text{ diğer durumlarda.} \end{cases}$$

- olur.

Graflar – Tanımlar

- **Örnek:** Basit grafın komşuluk ve bitişiklik matrisi çıkarılması
- Aşağıda a)'da verilen grafın komşuluk ve bitişiklik matrislerini ayrı ayrı çıkarınız; hangisi simetriktir? Neden?

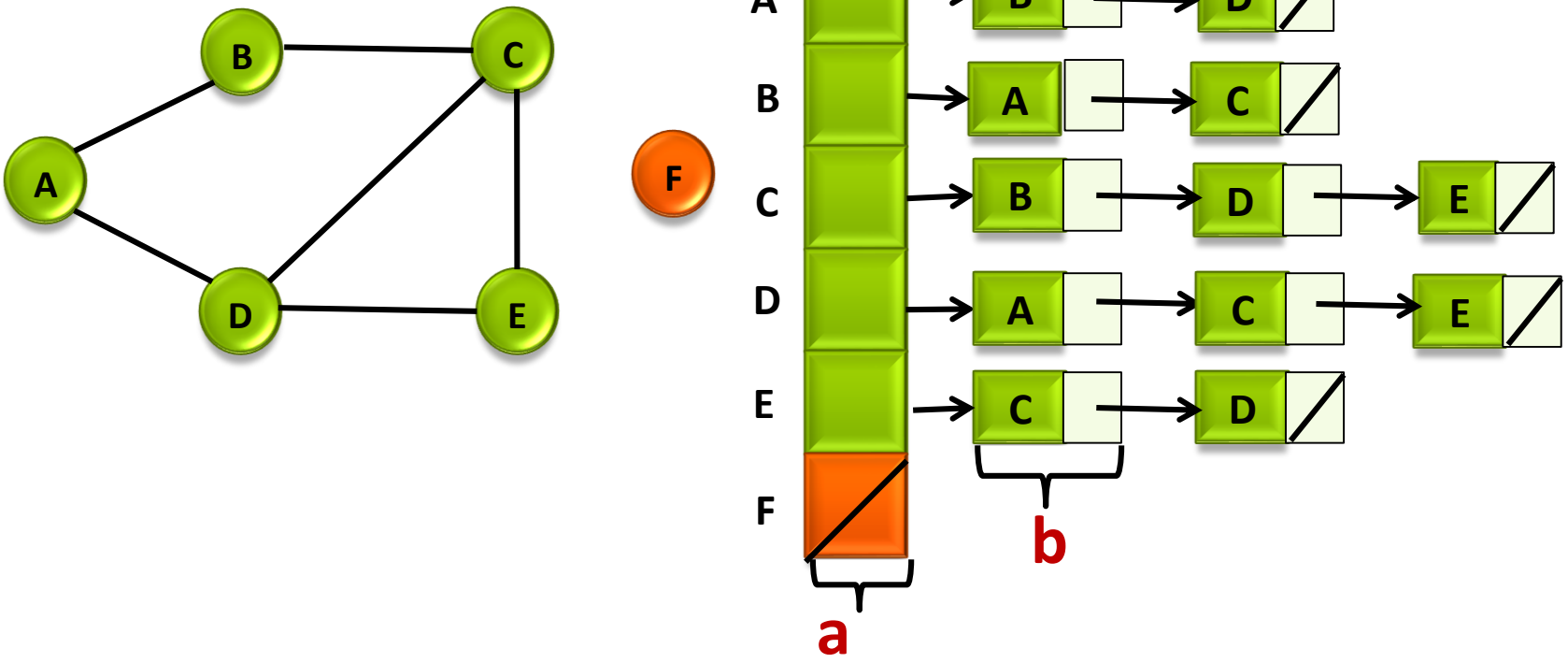


	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	1	1
D	1	1	1	0

	k_0	k_1	k_2	k_3	k_4	k_5
A	1	1	0	0	0	0
B	0	0	1	1	0	0
C	1	0	1	0	1	1
D	0	1	0	1	0	1

- Komşuluk matrisi simetriktir. Yönsüz graflarda, her iki yönde bağlantı sağlanmaktadır.

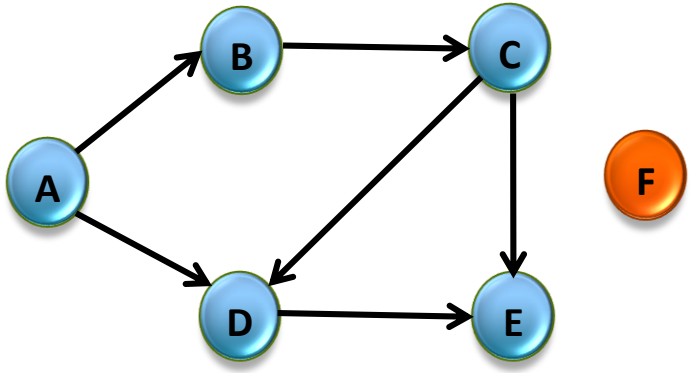
Komşuluk Listesi (Dizi Bağlantılı Liste) Gösterimi

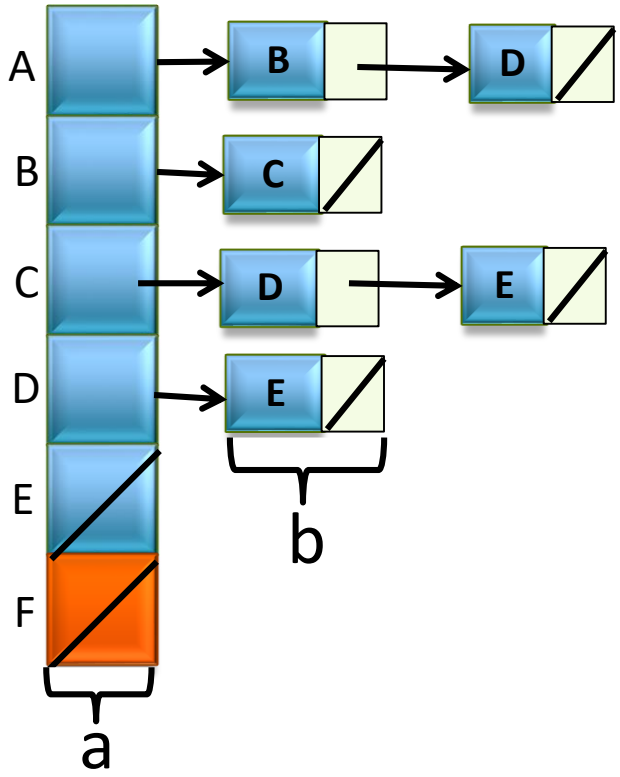


Yer karmaşıklığı= n*a + 2*k*b = O(n+2k)

Komşuluk Listesi Gösterimi

- Komşuluk Listesi (Yönlendirilmiş Graflar)

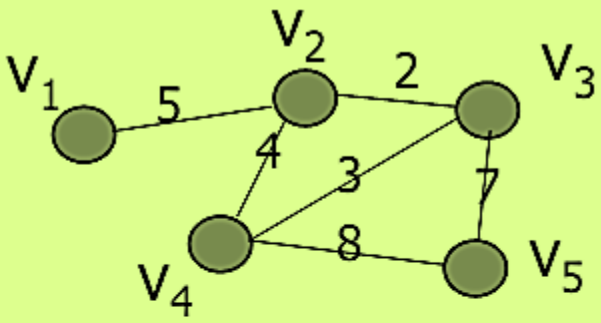




Yer karmaşıklığı= n*a + k*b = O(n+k)

Komşuluk Listesi Gösterimi

- Yönlendirilmiş ve ağırlıklandırılmış graf için komşu listesi.



G

1	v_1	$\rightarrow v_2(5)$
2	v_2	$\rightarrow v_1(5) \rightarrow v_3(2) \rightarrow v_4(4)$
3	v_3	$\rightarrow v_2(2) \rightarrow v_4(3) \rightarrow v_5(7)$
4	v_4	$\rightarrow v_2(4) \rightarrow v_3(3) \rightarrow v_5(8)$
5	v_5	$\rightarrow v_3(7) \rightarrow v_4(8)$

Komşu Matrisi-Komşu Listesi

- **Avantajları-dezavantajları;**

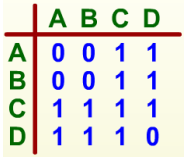
- **Komşu matrisi**
  - Çok fazla ihtiyaç duyulmasalana ihtiyaç duyar. Daha az hafızaya ı için sparse (seyrek matris) matris tekniklerinin kullanılması gerekir.
  - Herhangi iki node'un komşu olup olmadığına çok kısa sürede karar verilebilir.

- **Komşu listesi**
  - Bir node'un tüm komşularına hızlı bir şekilde ulaşılır.
  - Daha az alana ihtiyaç duyar.
  - Oluşturulması matrise göre daha zor olabilir.

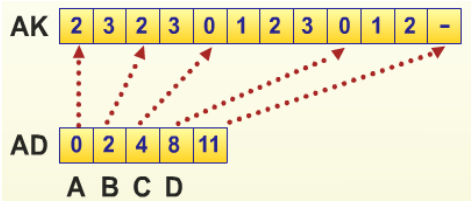
Not: Sparse matris; m*n boyutlu matriste değer bulunan hücreleri x boyutlu matriste saklayarak hafızadan yer kazanmak için kullanılan yöntemin adı.

Grafların Bellek Üzerinde Tutulması

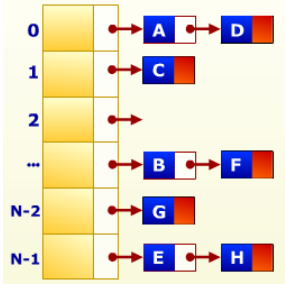
- Matris üzerinde



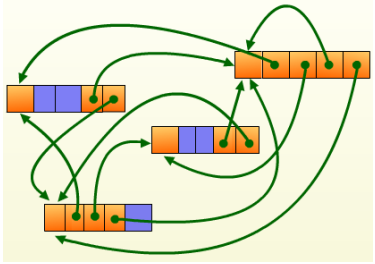
- İki-Dizi Üzerinde



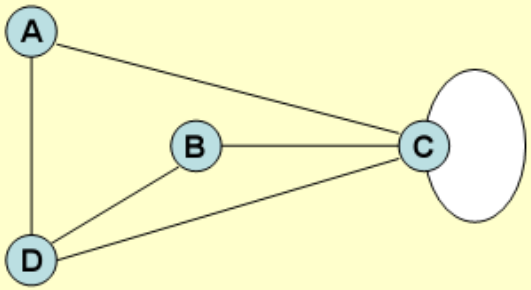
- Dizili Bağlantılı liste ile



- Bağlantılı Liste ile



Matris Üzerinde



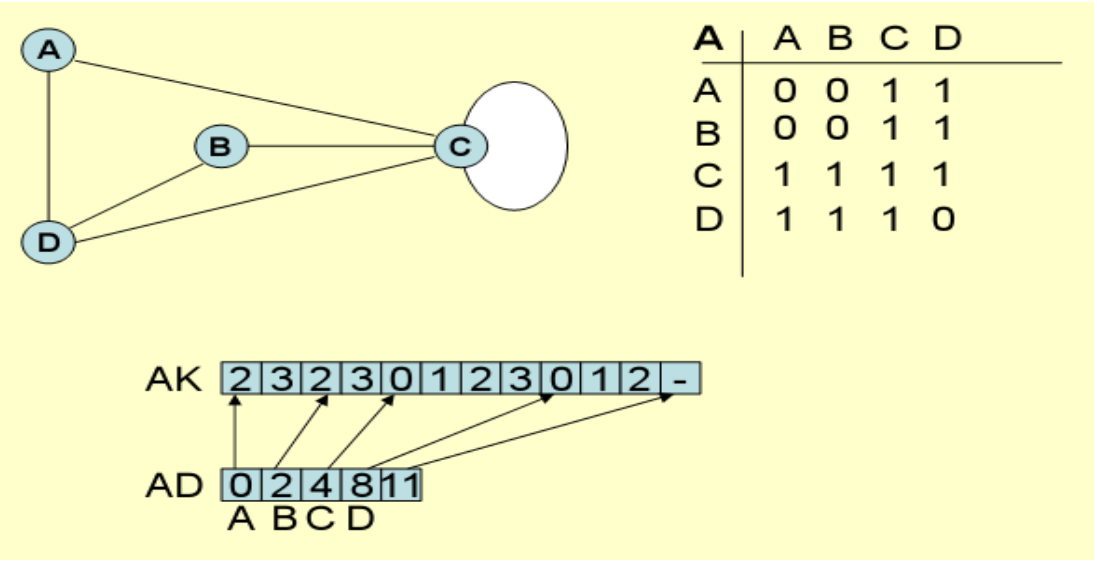
A	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	1	1
D	1	1	1	0

$$A=\begin{vmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{vmatrix}$$

- int A[4][4]={{0,0,1,1}, {0,0,1,1}, {1,1,1,1}, {1,1,1,0}}
- int baglantivarmi (int durum1, int durum2)
- {
- if(A[durum1][durum2]!=0)
- return 0;
- else
- return 1;
- }

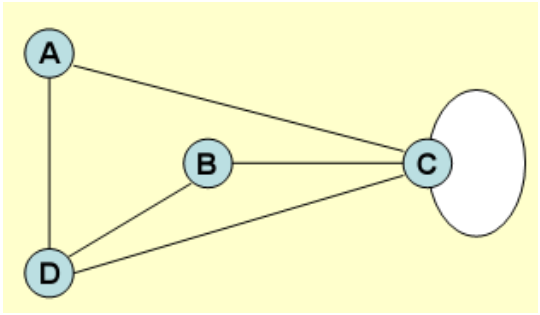
İki-Dizi Üzerinde

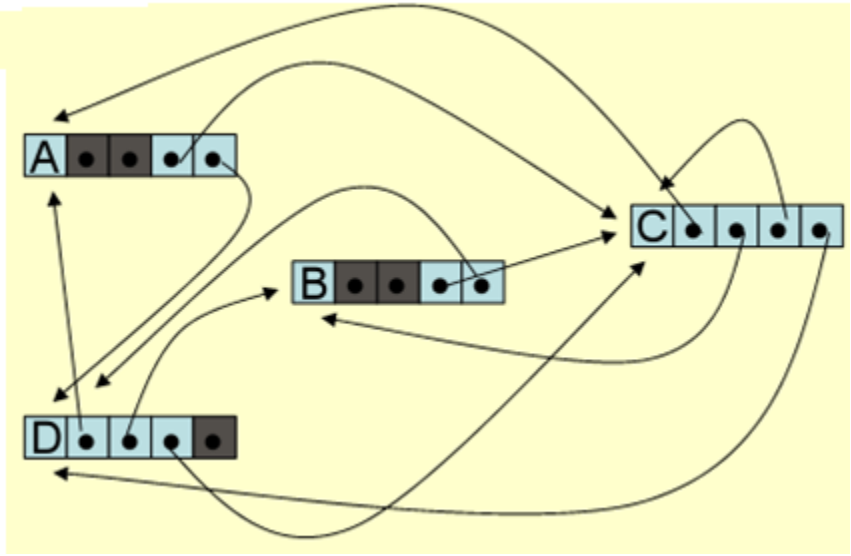
- Komşular AK, İndisler AD yi oluşturuyor



- A=0, B=1, C=2, D=3 numaralanmış
- A, 2 ve 3 komşu başlangıç indisi 0 da 2, indis 1 de 3
- B, 2 ve 3 komşu başlangıç indisi 2 de 2 indis 3 te 3
- C, 0, 1, 2 ve 3 komşu başlangıç indisi 4 de 0, 5 te 1, 6 da 2, 7 de 3
- D, 0, 1 ve 2 ye komşu başlangıç indisi 8 de 0 , 9 da 1, 10 da 2

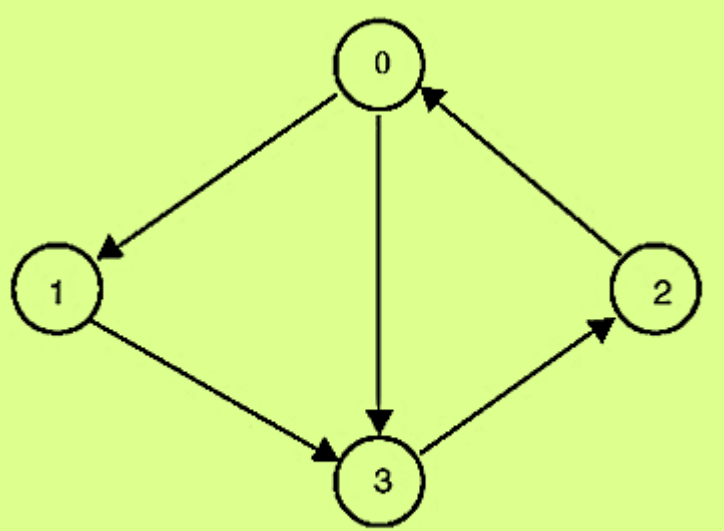
Bağlantılı Liste Üzerinde





- struct grafDrum{
- verituru dugumadi;
- struct graftdrum *bag[4];
- };

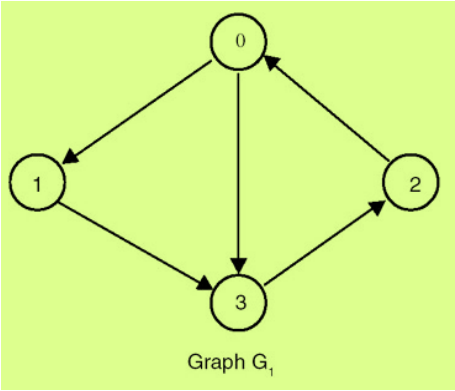
- Indegree (girenler) ve outdegree (çıkanlar) matris üzerinde gösterimi



Graph G_1

	0	1	2	3
0	0	1	0	1
1	0	0	0	1
2	1	0	0	0
3	0	0	1	0

Örnek



Graph G_1

	0	1	2	3
0	0	1	0	1
1	0	0	0	1
2	1	0	0	0
3	0	0	1	0

- Indegree ve outdegree hesaplanması

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
/* Bitişiklik matrisine ait fonksiyonun tanımlanması*/
void buildadjm(int adj[][MAX], int n)   {
 int i,j;
for(i=0;i<n;i++)
for(j=0;j<n;j++)        {
printf("%d ile %d arasında kenar var ise 1 yoksa 0 giriniz\n",i,j);
scanf("%d",&adj[i][j]);        }   }
```

Örnek

- /* Outdegree ye ait düğümlerin hesaplandığı fonksiyon*/

- int outdegree(int adj[][MAX],int x,int n) {
- int i, count =0;
- for(i=0;i<n;i++)
- if(adj[x][i] ==1) count++; return(count);
- }

Örnek

- /* Indegree ye ait düğümlerin hesaplandığı fonksiyon */

- int indegree(int adj[][MAX],int x,int n)
- {
- int i, count =0;
- for(i=0;i<n;i++)
- if(adj[i][x] ==1) count++; return(count);
- }

Örnek

```
 void main()    {
int adj[MAX][MAX],node,n,i;
printf(“Graph için maksimum düğüm sayısını giriniz (Max= %d):",MAX);

scanf("%d",&n);

buildadjm(adj,n);
for(i=0;i<n;i++)
{
printf(“\n%d Indegree düğüm sayısı %d dır ",i,indegree(adj,i,n));
printf(“\n%d Outdegree düğüm sayısı %d dır ",i,outdegree(adj,i,n));   }
}
```

Graf Üzerinde Dolaşma

- Graf üzerinde dolaşma grafın düğümleri ve kenarları üzerinde istenen bir işi yapacak veya bir problemi çözecek biçimde hareket etmektir.

- Graf üzerinde dolaşma yapan birçok yaklaşım yöntemi vardır; en önemli iki tanesi, kısaca, **DFS (Depth First Search)** ve **BFS (Breadth First Search)** olarak adlandırılmıştır ve graf üzerine geliştirilen algoritmaların birçoğu bu yaklaşım yöntemlerine dayanmaktadır denilebilir.

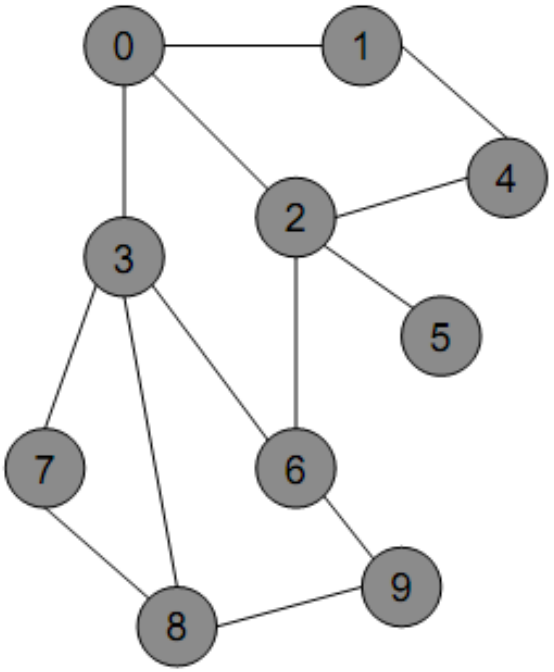
Graf Üzerinde Dolaşma

- **DFS (Depth First Search) Yöntemi**
- DFS, graf üzerinde dolaşma yöntemlerinden birisidir; **önce derinlik araması** olarak adlandırılabilir; başlangıç düğümünün bir kenarından başlayıp o kenar üzerinden gidilebilecek en uzak (derin) düğüme kadar sürdürülür.

Graf Üzerinde Dolaşma

- Depth First Arama İşlem Adımları
  - Önce bir başlangıç node'u seçilir ve ziyaret edilir.
  - Seçilen node'un bir komşusu seçilir ve ziyaret edilir.
  - 2.adım ziyaret edecek komşu kalmayıncaya kadar tekrar edilir.
  - Komşu kalmadığında tekrar geri dönülür ve önceki ziyaret edilmiş node'lar için adım 2 ve 3 tekrar edilir.

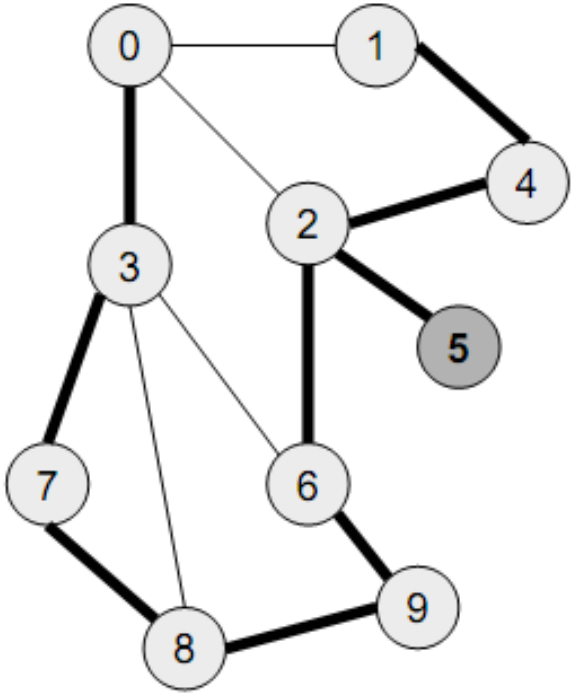
Graf Üzerinde Dolaşma

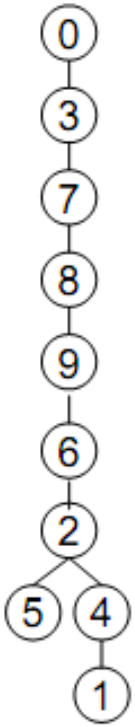



```
1.  s node'unu seç
2.  visit s
    // örn. ekrana yaz
3.  for each edge <s, U>
    // U komşu node
4.      if U is not visit
5.          DFS(G, U)
```

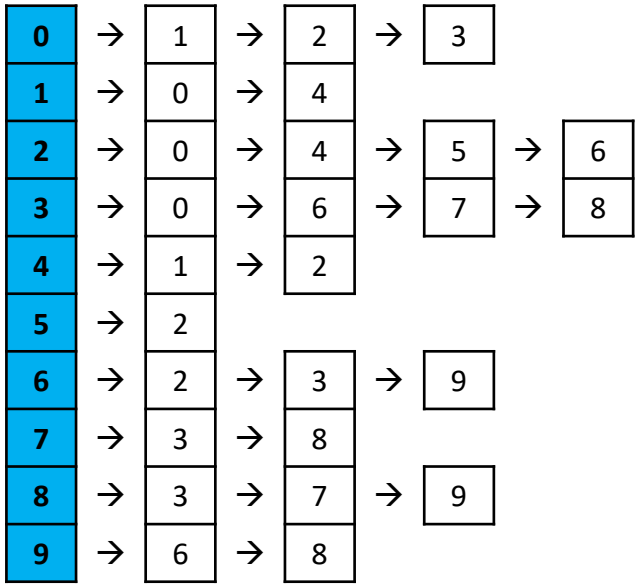
Graf Üzerinde Dolaşma

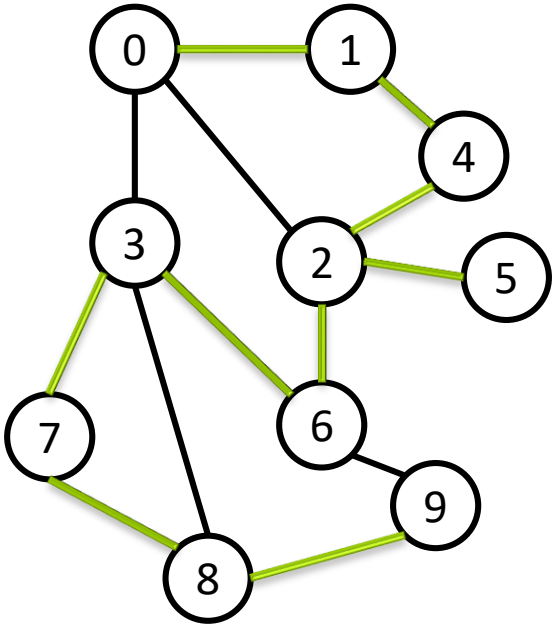
- Depth first arama

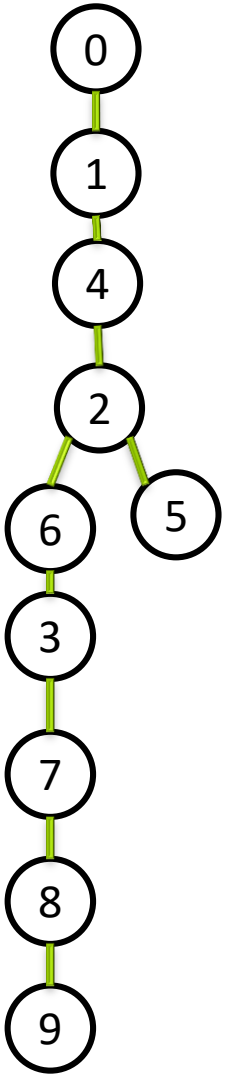




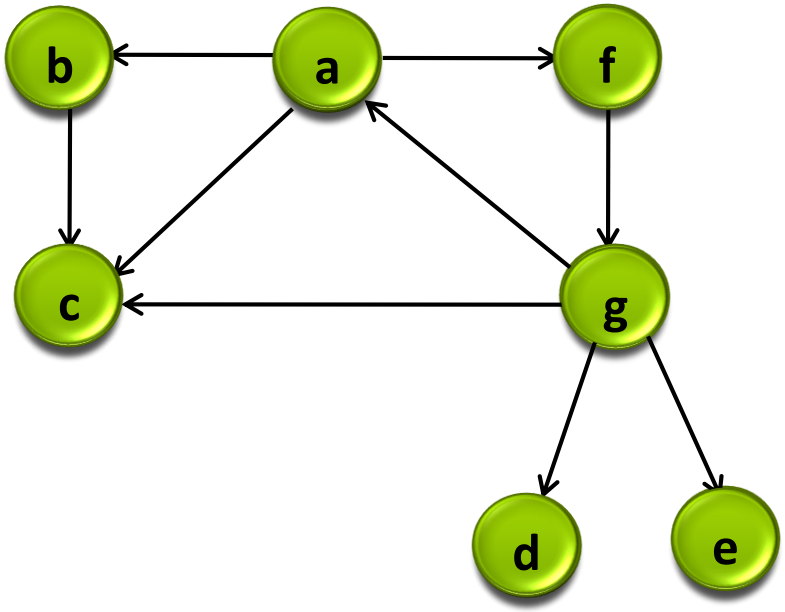
Graf Üzerinde Dolaşma



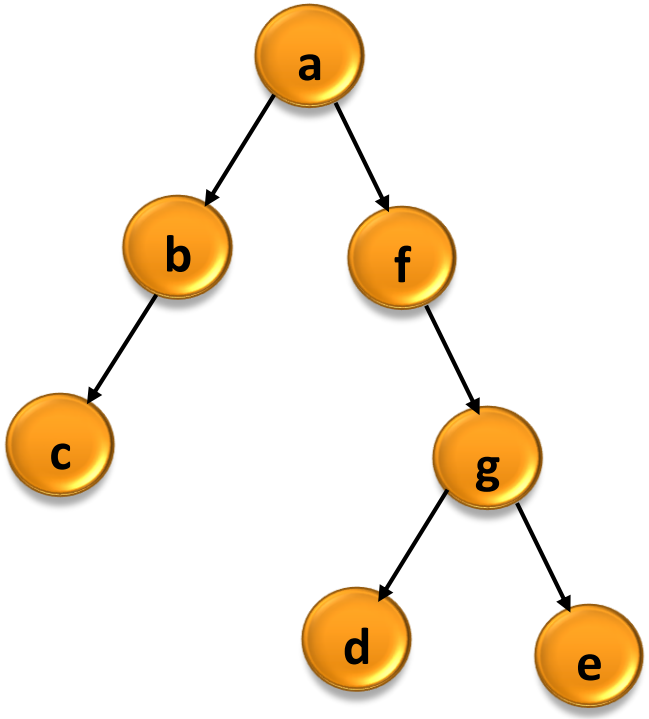




DFS – Örnek



DFS(a) ağacı



Graf Üzerinde Dolaşma

- **BFS Yöntemi :**
- **BFS (Breadth First Search)**, önce genişlik araması olarak adlandırılabilir. Bu yöntemin DFS'den farkı, dolaşmaya, başlangıç düğümünün bir kenarı ayrıtı üzerinden en uzağa gidilmesiyle değil de, başlangıç düğümünden gidilebilecek tüm komşu düğümlere gidilmesiyle başlanır.

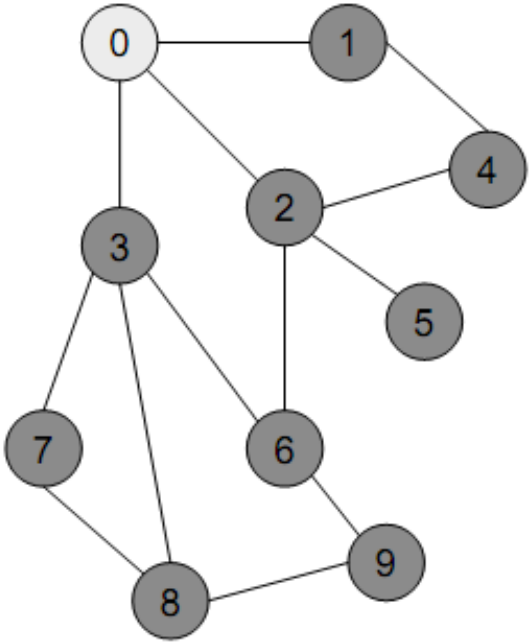
- BFS yöntemine göre graf üzerinde dolaşma, graf üzerinde dolaşarak işlem yapan diğer birçok algoritmaya esin kaynağı olmuştur denilebilir. Örneğin, kenar maliyetler yoksa veya eşitse, BFS yöntemi en kısa yol algoritması gibidir; bir düğümden herbir düğüme olan en kısa yolları bulur denilebilir.

Graf Üzerinde Dolaşma

- Breadth First Arama İşlem Adımları
  - Breadth first arama ağaçlardaki level order aramaya benzer.
  - Seçilen node'un tüm komşuları sırayla seçilir ve ziyaret edilir.
  - Her komşu queue içerisine atılır.
  - Komşu kalmadığında Queue içerisindeki ilk node alınır ve 2.adıma gidilir.

Graf Üzerinde Dolaşma

- **Breadth first arama**

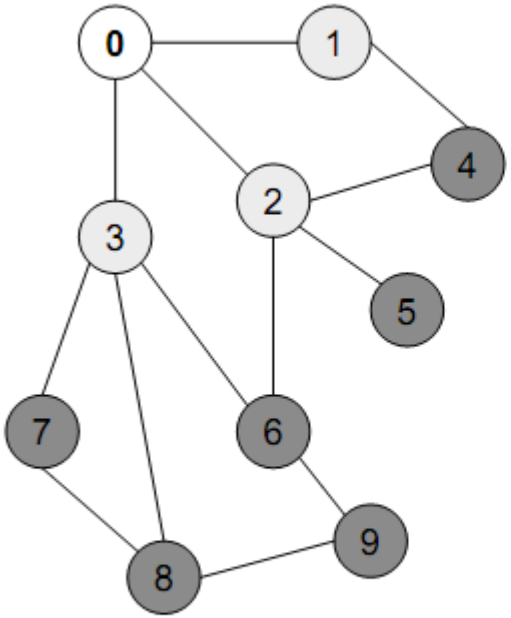


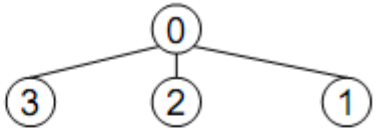




Graf Üzerinde Dolaşma

- **Breadth first arama**

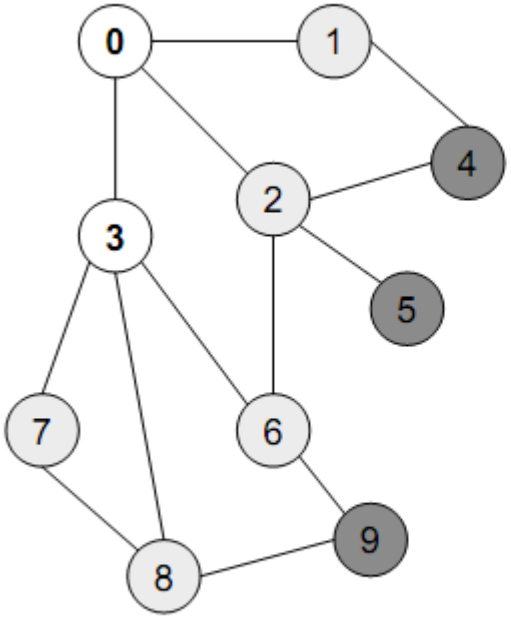


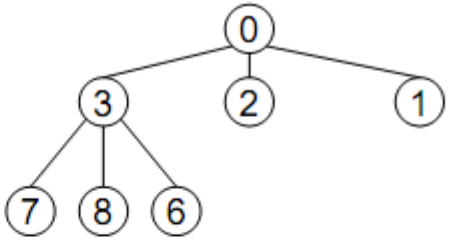




Graf Üzerinde Dolaşma

- **Breadth first arama**

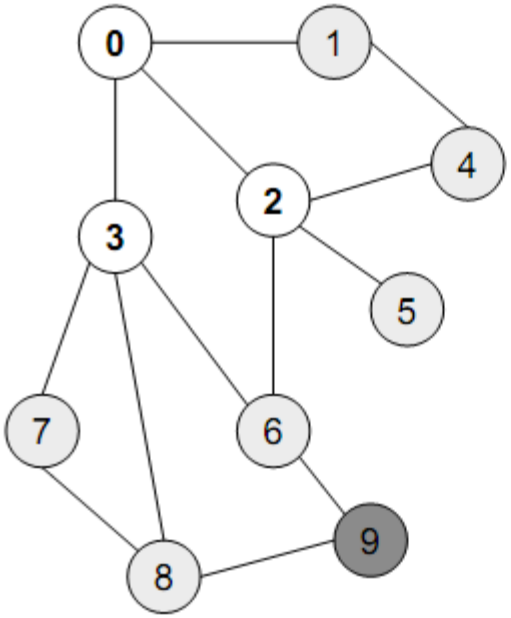


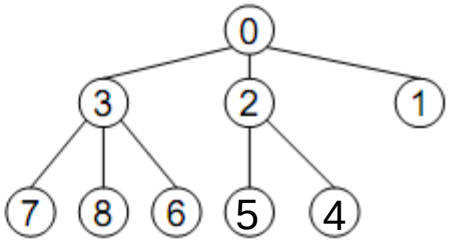




Graf Üzerinde Dolaşma

- **Breadth first arama**

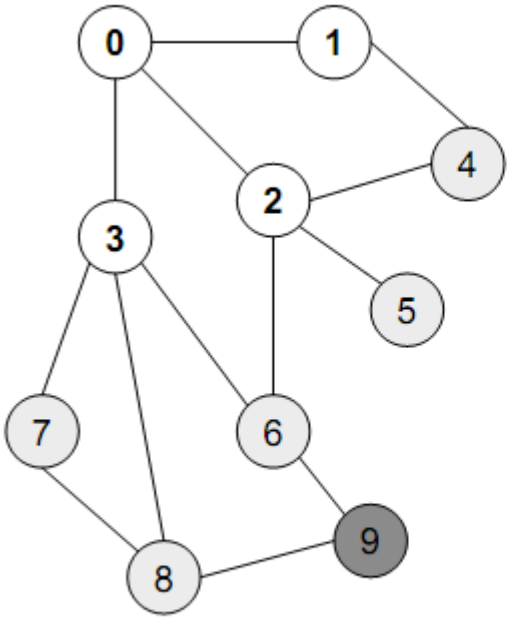


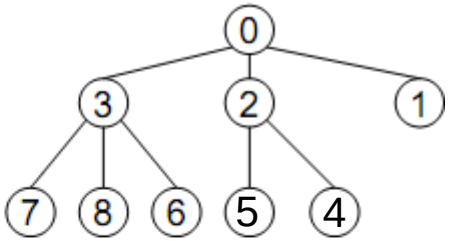


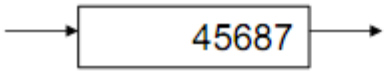


Graf Üzerinde Dolaşma

- **Breadth first arama**

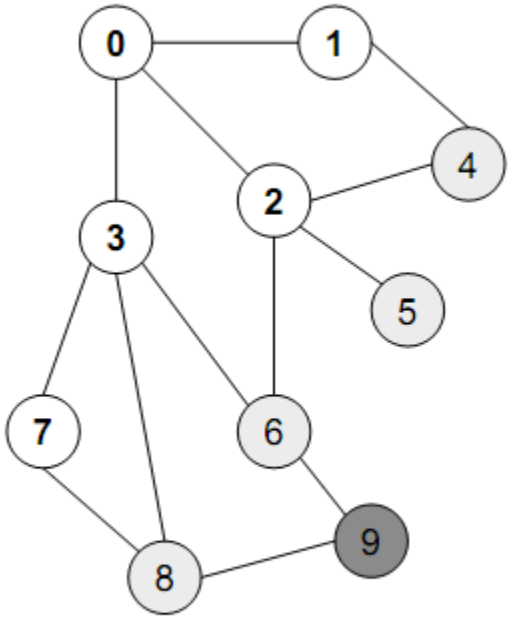


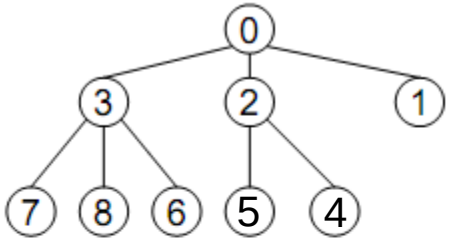


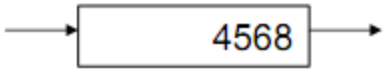


Graf Üzerinde Dolaşma

- **Breadth first arama**

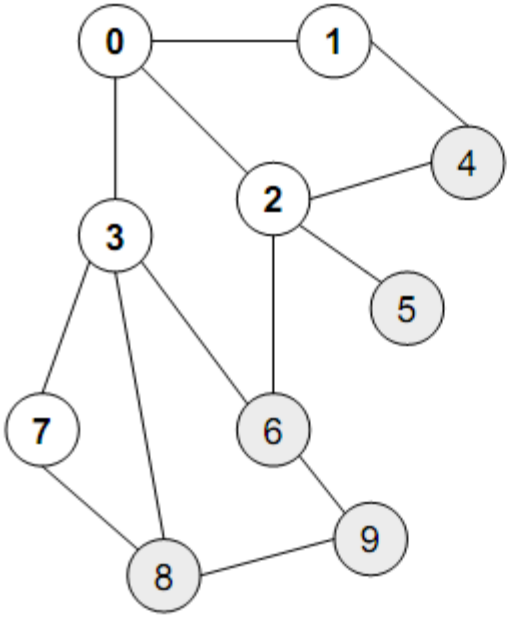


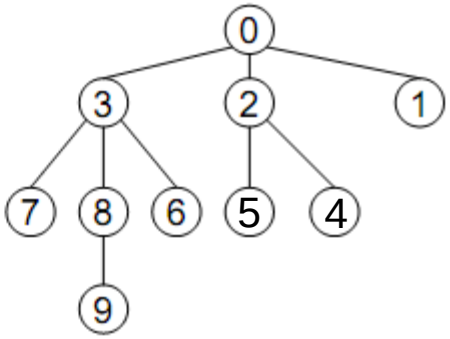


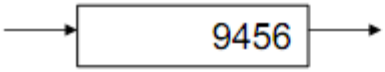


Graf Üzerinde Dolaşma

- **Breadth first arama**

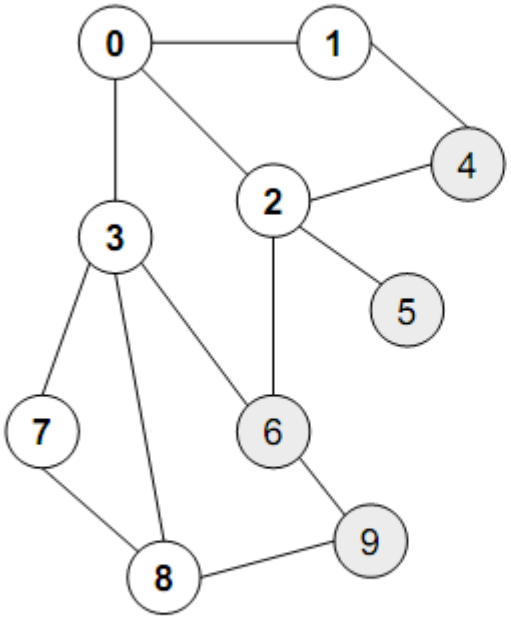


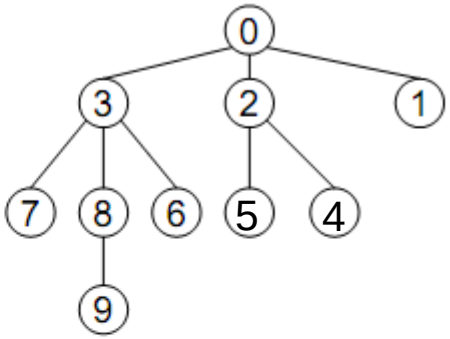


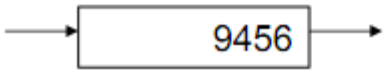


Graf Üzerinde Dolaşma

- **Breadth first arama**

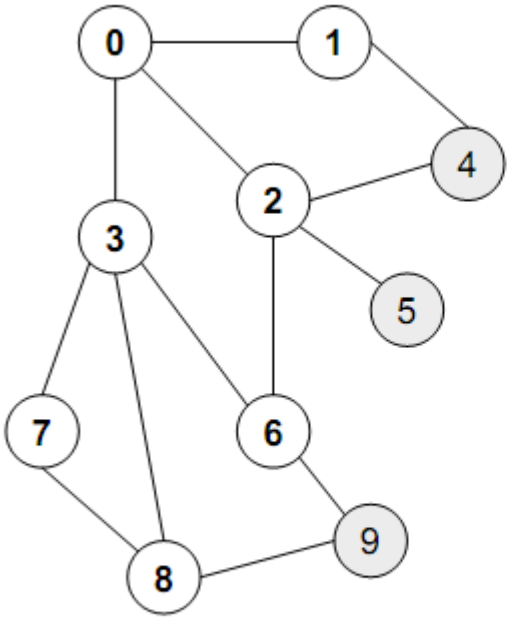


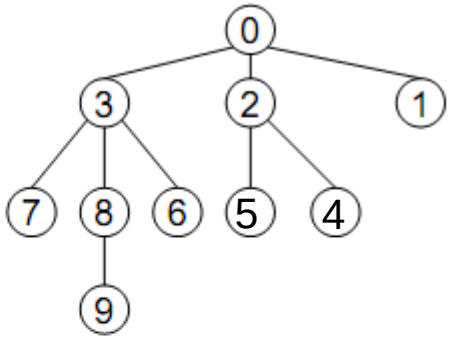




Graf Üzerinde Dolaşma

- **Breadth first arama**

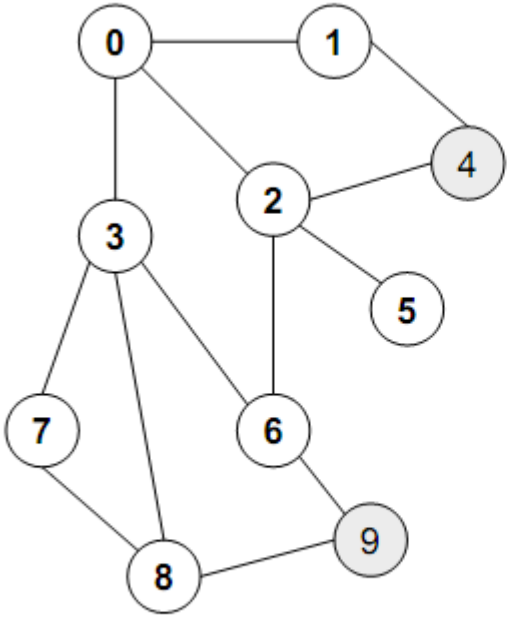


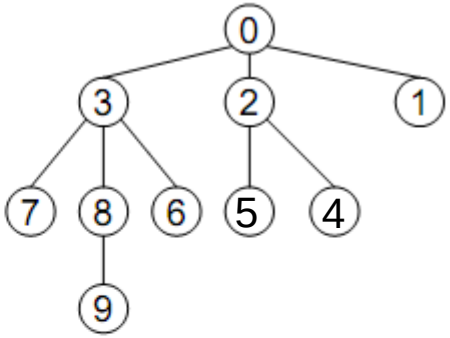




Graf Üzerinde Dolaşma

- **Breadth first arama**

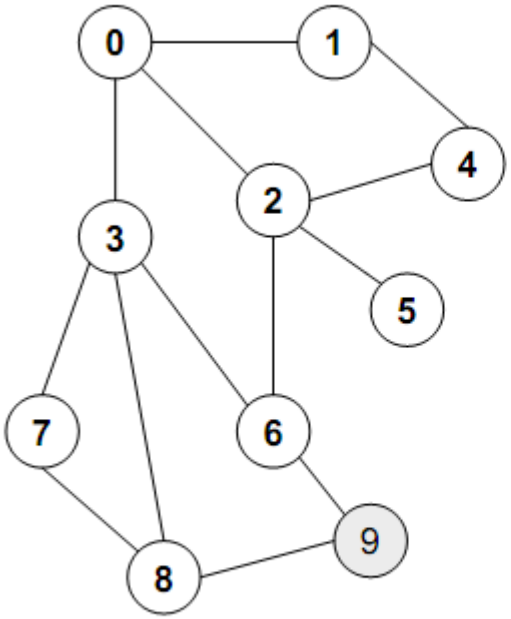


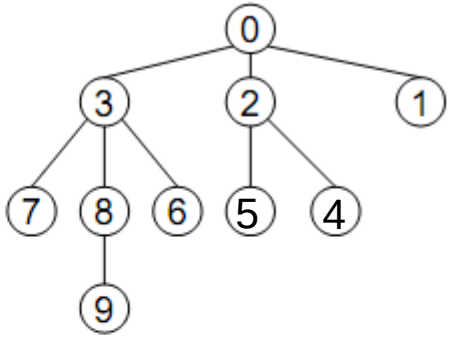




Graf Üzerinde Dolaşma

- **Breadth first arama**

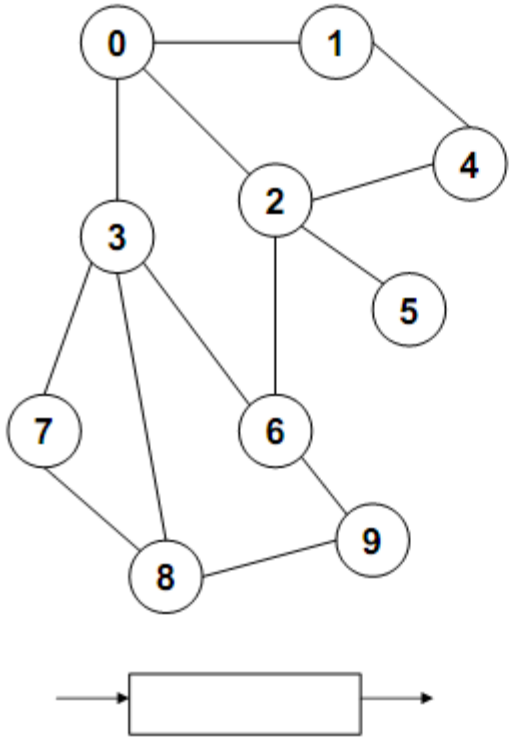


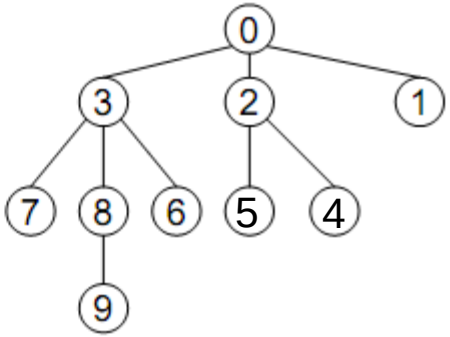




Graf Üzerinde Dolaşma

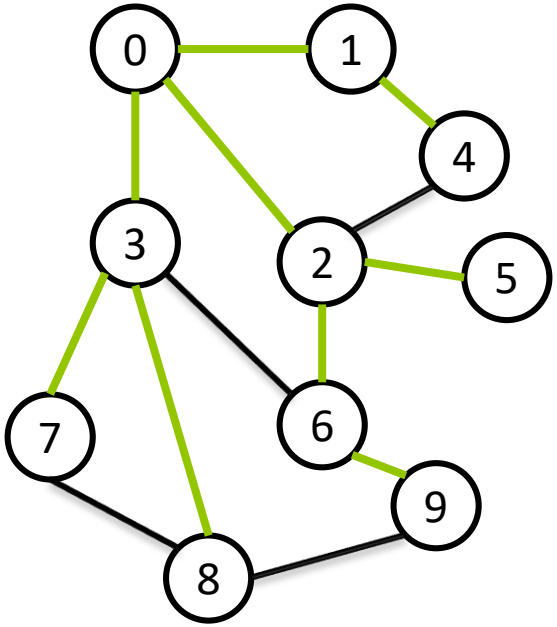
- **Breadth first arama**

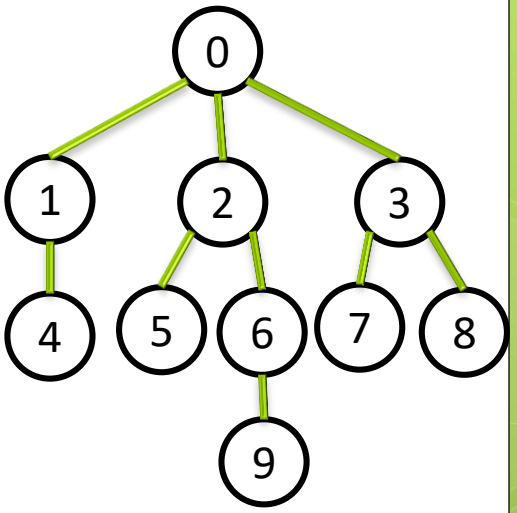




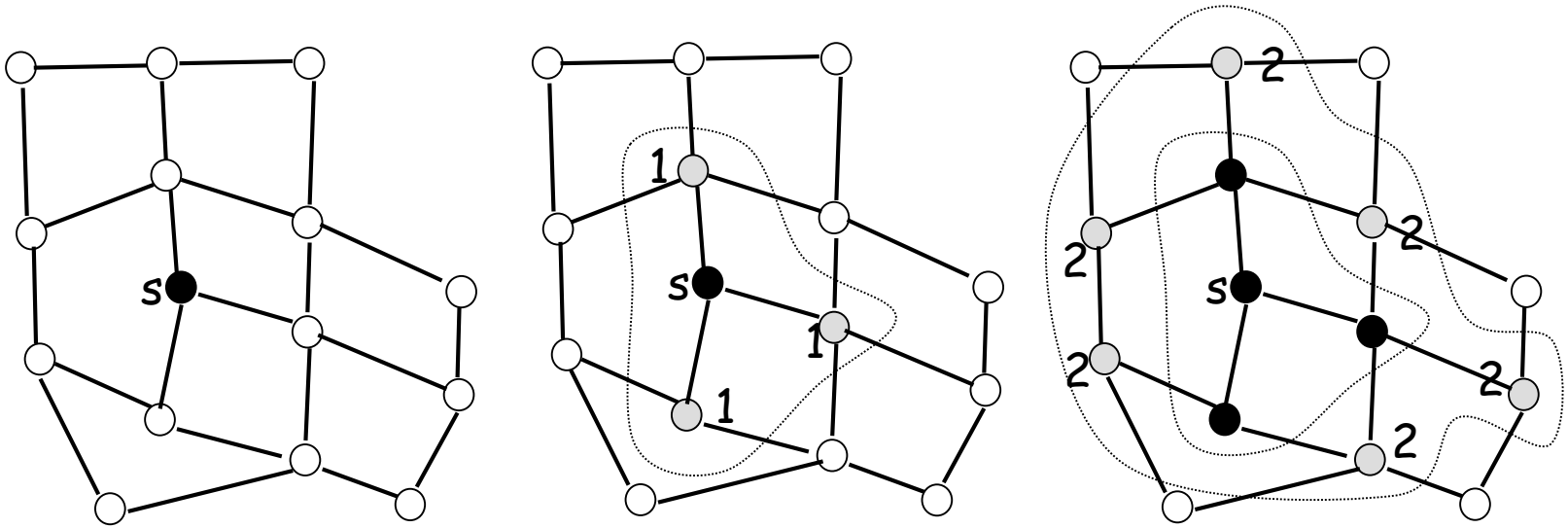
Graf Üzerinde Dolaşma

0	→	1	→	2	→	3		
1	→	0	→	4				
2	→	0	→	4	→	5	→	6
3	→	0	→	6	→	7	→	8
4	→	1	→	2				
5	→	2						
6	→	2	→	3	→	9		
7	→	3	→	8				
8	→	3	→	7	→	9		
9	→	6	→	8				

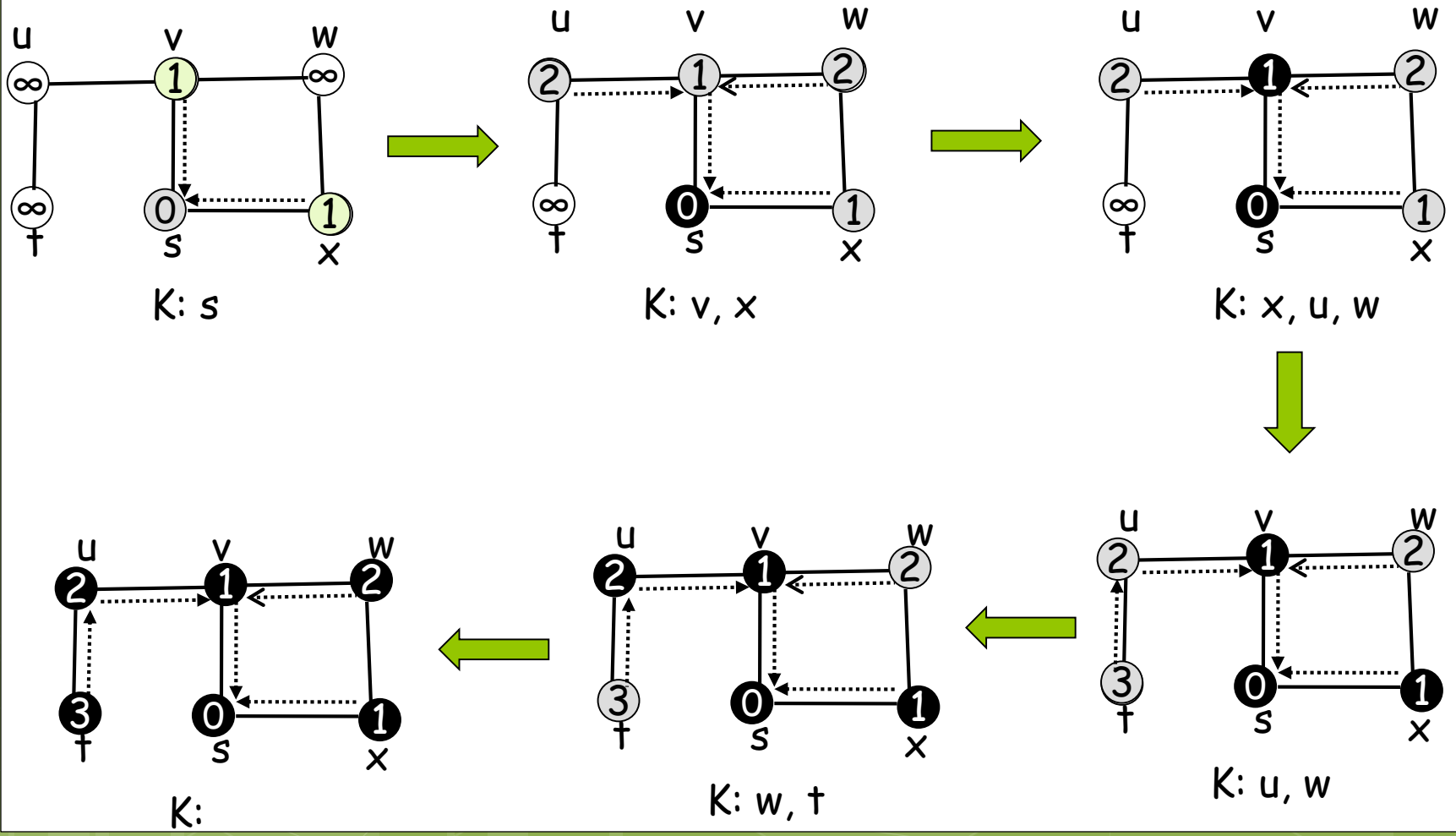




BFS-Örnek



BFS – Örnek


u
v
w
∞
1
∞
∞
0
1
t
s
x
K: s
2
1
2
∞
0
1
K: v, x
2
1
2
∞
0
1
K: x, u, w
2
1
2
3
0
1
K: u, w
2
1
2
3
0
1
K: w, t
2
1
2
3
0
1
K:

Örnekler

- **Depth-First Search**

```
class Node
 {
      int label; /* vertex label*/
      Node next; /* next node in list*/
      Node( int b ) { label = b; }
}

class Graph
{
    int size;  Node adjList[];  int mark[];
    Graph(int n)
    {  size = n;
       adjList = new Node[size];
       mark = new int[size]; //elements of mark are initialized to 0
  }
```

Örnekler

- **Depth-First Search**

```
public void createAdjList(int a[][]) // create adjacent lists
{
  Node p; int i, k;
  for( i = 0; i < size; i++ )
  {
    p = adjList[i] = new Node(i); //create first node of ith adj. list
    for( k = 0; k < size; k++ )
    {
      if( a[i][k] == 1 )
      { p.next = new Node(k); /* create next node of ith adj. List*/
        p = p.next;
      }
    }
  }
}
```

Örnekler

```
public void dfs(int head) // recursive depth-first search
{
   Node w; int v;
   mark[head] = 1;
   System.out.print( head + " ");
   w = adjList[head];
  while( w != null)
  {
     v = w.label;
     if( mark[v] == 0 ) dfs(v);
     w = w.next;
   }
 }
}
```

Örnekler

- **Dfs.java**

```
class Dfs
{
  public static void main(String[] args)
  { Graph g = new Graph(5); // graph is created with 5 nodes
    int a[][] = { {0,1,0,1,1}, {1,0,1,1,0},
                  {0,1,0,1,1}, {1,1,1,0,0},
                  {1,0,1,0,0}};
    g.createAdjList(a);
    g.dfs(0); // starting node to dfs is 0 (i.e., A)
   }
}
```

- **Programın çıktısı: 0 1 2 3 4**
- **Burada A için 0, B için 1, C için 2, D için 3 ve E için 4**

Örnekler

- Breadth-First Search

```
class Node
 {
      int label; /* vertex label*/
      Node next; /* next node in list*/
      Node( int b ) { label = b; }
}

class Graph
{
    int size;  Node adjList[];  int mark[];
    Graph(int n)
    {  size = n;
       adjList = new Node[size];
       mark = new int[size]; //elements of mark are initialized to 0
  }
```

Örnekler

```
public void createAdjList(int a[][]) // create adjacent lists
{
  Node p; int i, k;
  for( i = 0; i < size; i++ )
  {
    p = adjList[i] = new Node(i); //create first node of ith adj. list
    for( k = 0; k < size; k++ )
    {
      if( a[i][k] == 1 )
      { p.next = new Node(k); /* create next node of ith adj. List*/
        p = p.next;
      }
    }
  }
}
```

Örnekler

```
public void bfs(int head)
{ int v;  Node adj;
  Queue q = new Queue(size);
  v = head;
  mark[v] = 1;  System.out.print(v + "");
  q.qinsert(v);
  while( !q.IsEmpty() )
  {  v = q.qdelete();       adj = adjList[v];
     while( adj != null )
     { v = adj.label;
     if( mark[v] == 0 )
      { q.qinsert(v);
        mark[v] = 1; System.out.print(v + "");
      }
     adj = adj.next;
     }
  }
 }
} // end of Graph class
```

Örnekler

```
class Queue
{ private int maxSize; // max queue size
  private int[] que; // que is an array of integers
  private int front;
  private int rear;
  private int count; // count of items in queue
  public Queue(int s) // constructor
  { maxSize = s;  que = new int[maxSize];
    front = rear = -1;
  }
public void qinsert(int item)
{
if( rear == maxSize-1 )  System.out.println("Queue is Full");
else { rear = rear + 1;
que[rear] = item;   if( front == -1 ) front = 0;  }
}
```

Örnekler

```
public int qdelete()
{  int item;
if(IsEmpty())
{ System.out.println("\n Queue is Empty"); return(-1); }
item = que[front];
if( front == rear ) front = rear = -1;
else front = front+1;        return(item); }
public boolean IsEmpty()   { return( front == -1 ); }  }

class BfsDemo
{ public static void main(String[] args)
  {   Graph g = new Graph(5);
       int a[][] = { {0,1,0,1,1}, {1,0,1,1,0}, {0,1,0,1,1},
{1,1,1,0,0}, {1,0,1,0,0}};
       g.createAdjList(a); g.bfs(0);     }
}
Output of this program is: 0 1 3 4 2
```

Graf Renklendirme

- Graf renklendirme, graf üzerinde birbirine komşu olan düğümlere farklı renk atama işlemidir; amaç, en az sayıda renk kullanılarak tüm düğümlere komşularından farklı birer renk vermektir. Renklendirmede kullanılan toplam renk sayısı kromatik (chromatik) sayı olarak adlandırılır.

- Uygulamada, graf renklendirmenin kullanılacağı alanların başında, ilk akla gelen, harita üzerindeki bölgelerin renklendirilmesi olmasına karşın, graf renklendirme bilgisayar biliminde ve günlük yaşamdaki birçok problemin çözümüne ciddi bir yaklaşımdır.

Graf Renklendirme

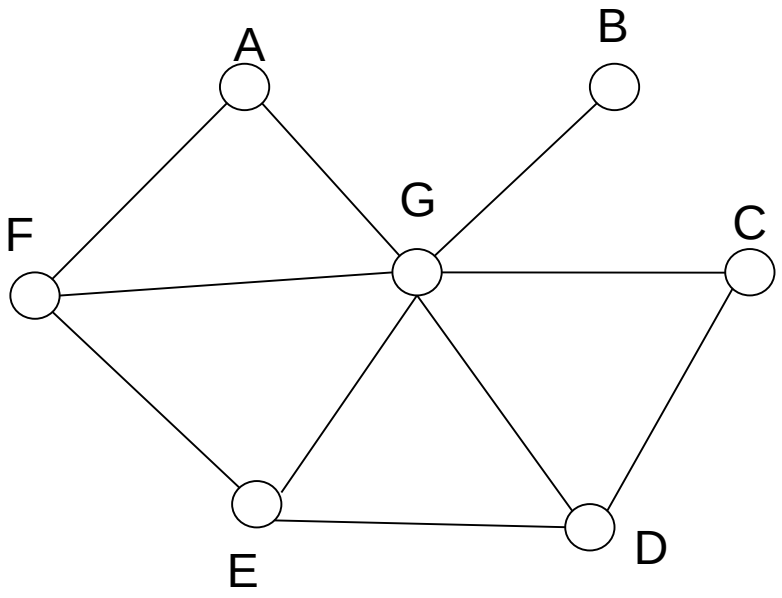
- Uygulama Alanları;
  - Harita renklendirme,
  - İşlemcilerin işlem sırasını belirleme,
  - Ders ve sınav programı ayarlama
  - Hava alanlarında iniş ve kalkış sırasını belirleme vs.

Graf Renklendirme

- Graf renklendirmede kullanılan algoritmaların başında Welch ve Powel algoritmasıdır.
- Welch ve Powel Algoritması:
  - Düğümler derecelerine göre büyükten küçüğe doğru sıralanır.
  - İlk renk birinci sıradaki düğüme atanır ve daha sonra aynı renk bitişik olamayacak şekilde diğer düğümlere verilir.
  - Bir sonraki renge geçilir ve aynı işlem düğümlerin tamamı renklendirilinceye kadar devam ettirilir.

Graf Renklendirme

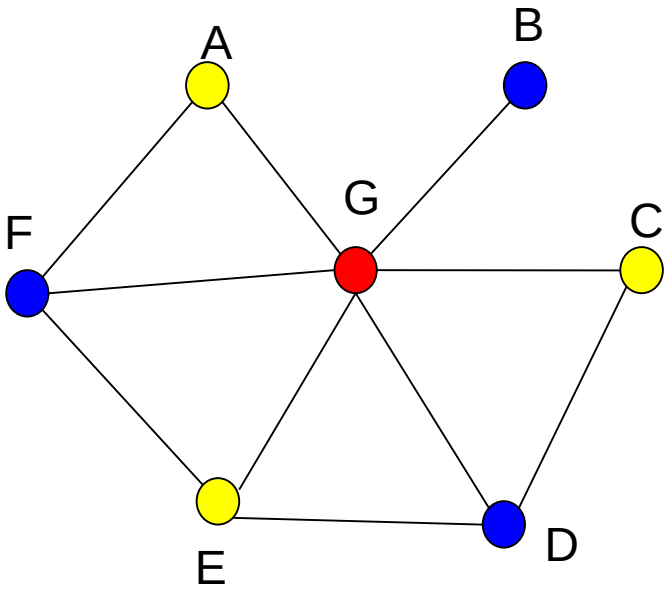
- Örnek: a) En az renk kullanılarak düğümleri renklendiriniz. Kromatik sayı kaç olur?



Düğüm	Derece
G	6
D	3
E	3
F	3
A	2
C	2
B	1

Graf Renklendirme

- Örnek: a) En az renk kullanılarak düğümleri renklendiriniz. Kromatik sayı kaç olur?

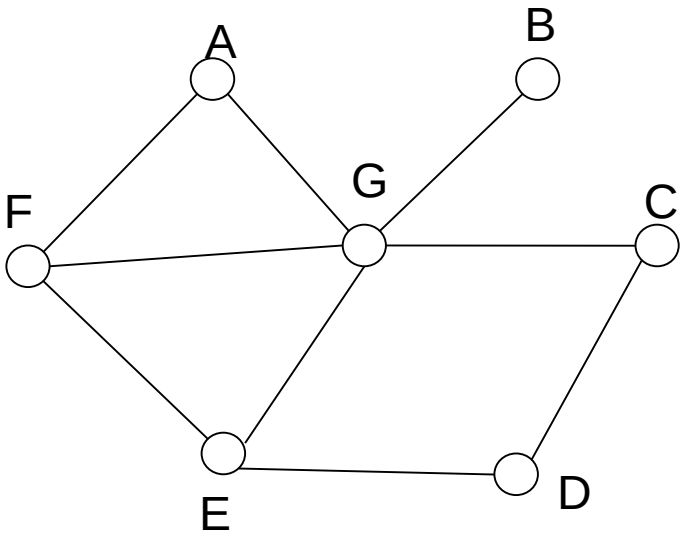


Düğüm	Derece
G	6
D	3
E	3
F	3
A	2
C	2
B	1

Kırmızı: G
Mavi: D,F,B
Sarı: E,A, C
Kromatik Sayı:3

Graf Renklendirme

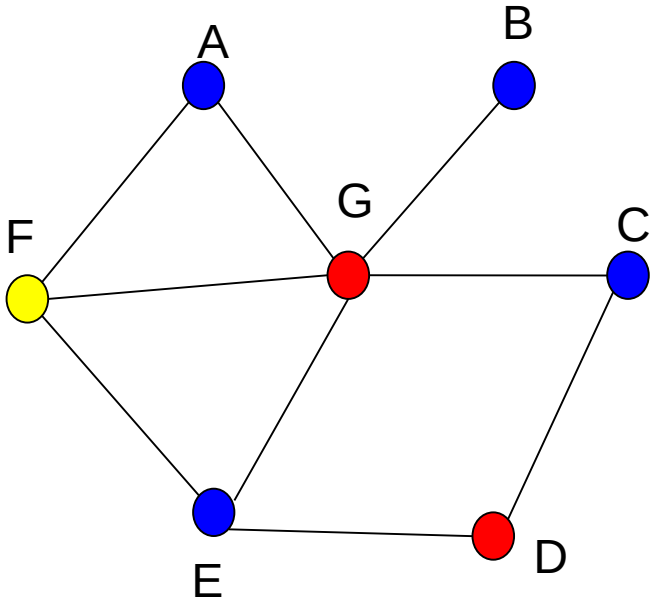
- b) G ile D arasındaki bağlantı kalkarsa yeni durum ne olur.



Düğüm	Derece
G	5
E	3
F	3
A	2
C	2
D	2
B	1

Graf Renklendirme

- b) G ile D arasındaki bağlantı kalkarsa yeni durum ne olur.



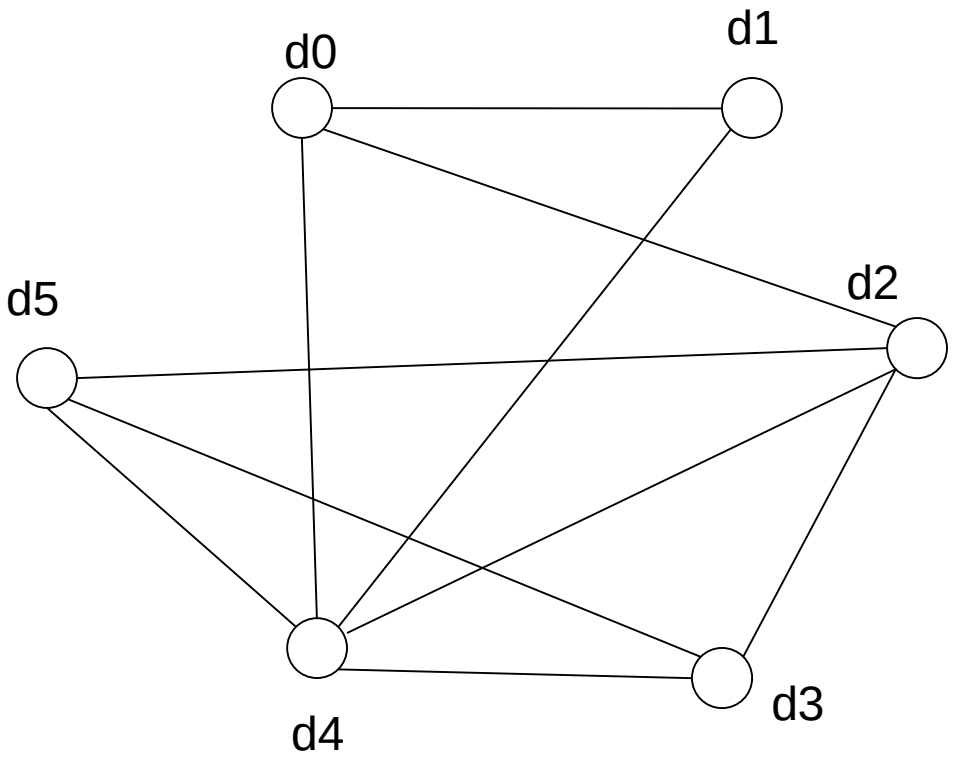
Düğüm	Derece
G	5
E	3
F	3
A	2
C	2
D	2
B	1

Kırmızı: G,D
Mavi:E,C,B,A
Sarı: F

Graf Renklendirme

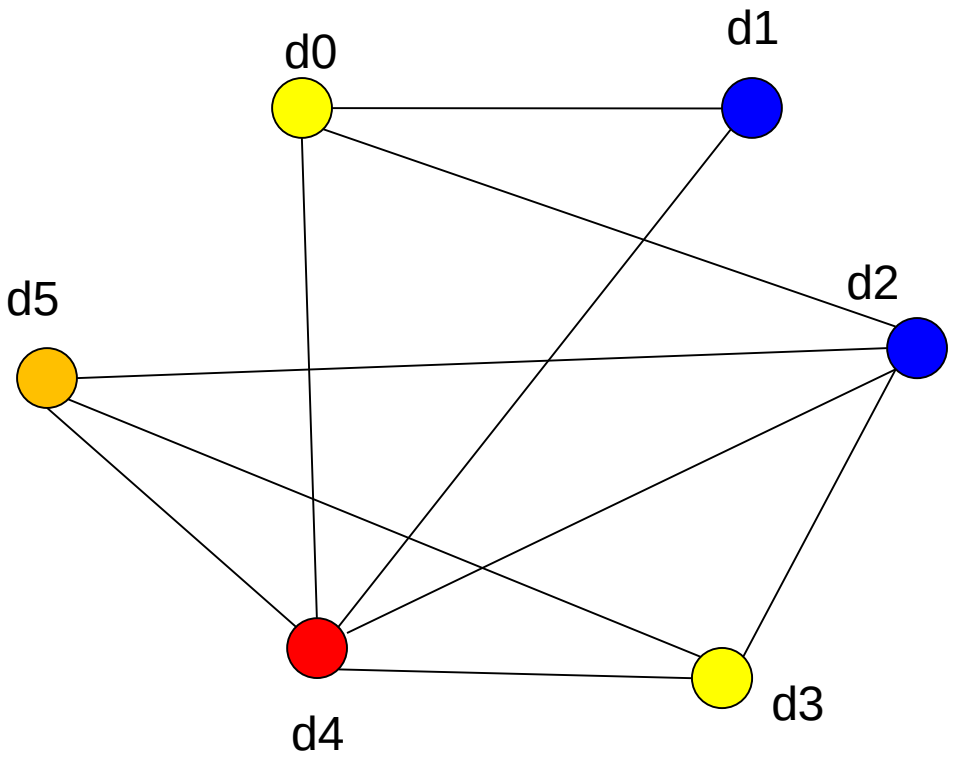
- Örnek: Farklı sınıflardaki öğrencilerin sınavlarını hazırlarken öğrencilerin gireceği sınavların çakışmasını engellemek için en az kaç farklı oturum yapılmalıdır
  - Öğrenci1: d0,d1,d4
  - Öğrenci2: d0,d2,d4
  - Öğrenci3: d2,d3,d5
  - Öğrenci4: d3,d4,d5

Graf Renklendirme



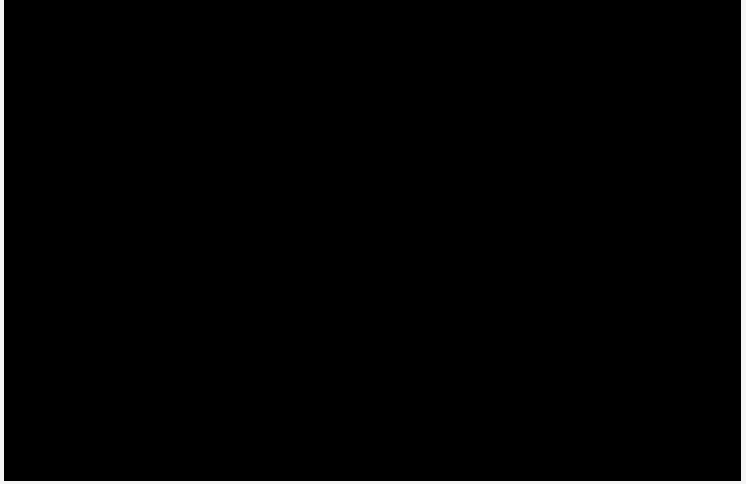
Düğüm	Derece
d4	5
d2	4
d0	3
d3	3
d5	3
d1	2

Graf Renklendirme



Düğüm	Derece
d4	5
d2	4
d0	3
d3	3
d5	3
d1	2

Kırmızı: d4
Mavi: d1,d2
Sarı: d0,d3
Turuncu: d5



En Küçük Yol Ağacı (Minimum Spanning Tree)

En Kısa Yol Problemi

- ***G = (V, E)*** grafı verilsin ve ***s*** başlangıç düğümünden ***v*** düğümüne giden en az maliyetli yol bulma.

- Farklı varyasyonlar mevcut
  - Ağırlıklı ve ağırlıksız graflar
  - Sadece pozitif ağırlık veya negatif ağırlığın da olması.

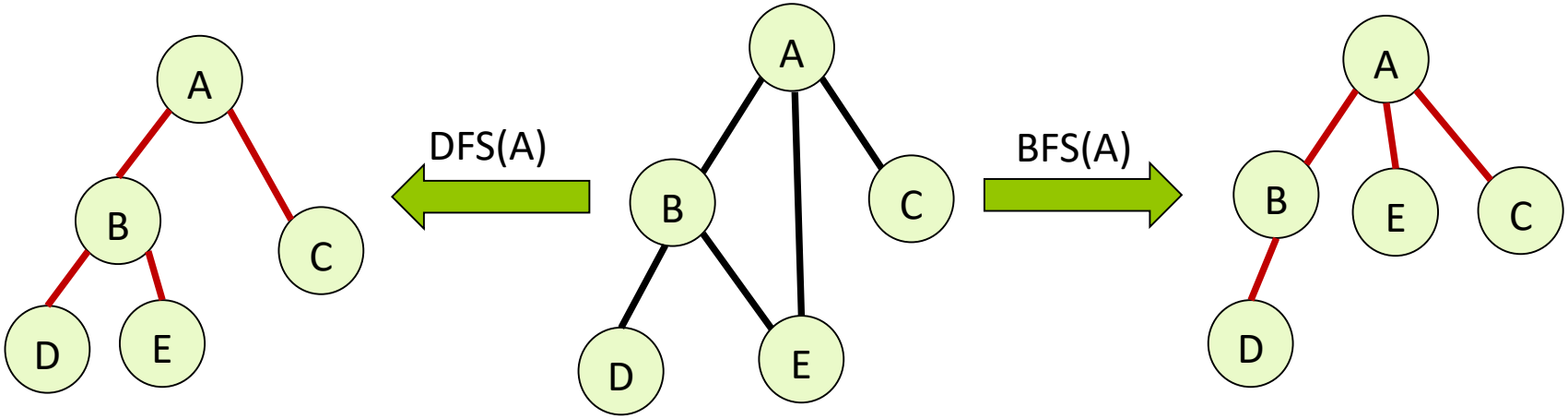
En Küçük Yol Ağacı (Minimum Spanning Tree)

- Yol ağacı, bir graf üzerinde tüm düğümleri kapsayan ağaç şeklinde bir yoldur.
  - Ağaç özelliği olduğu için kapalı çevrim(çember) içermez.

- Bir graf üzerinde birden çok yol ağacı olabilir. En az maliyetli olan en küçük yol ağacı (minimum spanning tree) olarak adlandırılır.

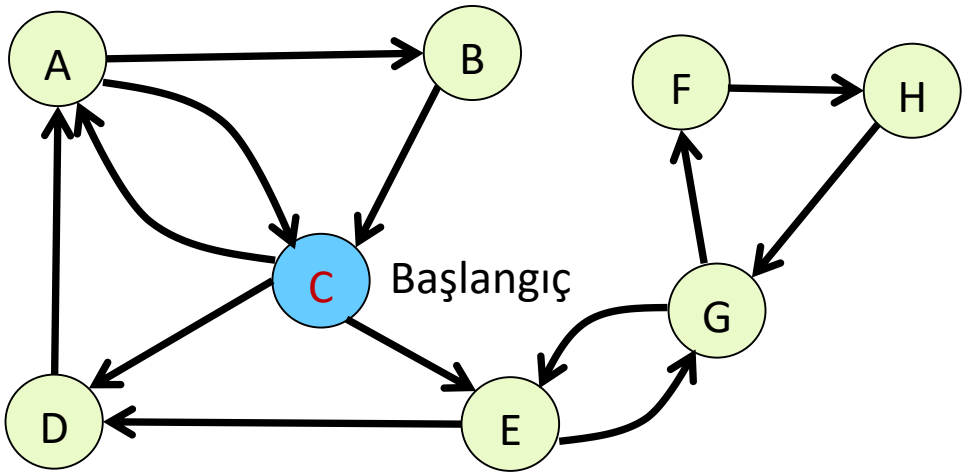
MST Hesaplama – Ağırlıksız Graf

- Graf ağırlıksızsa veya tüm kenarların ağırlıkları eşit ise MST nasıl bulunur?
  - BSF veya DSF çalıştırın oluşan ağaç MST'dir



Ağırlıksız En Kısa Yol Problemi

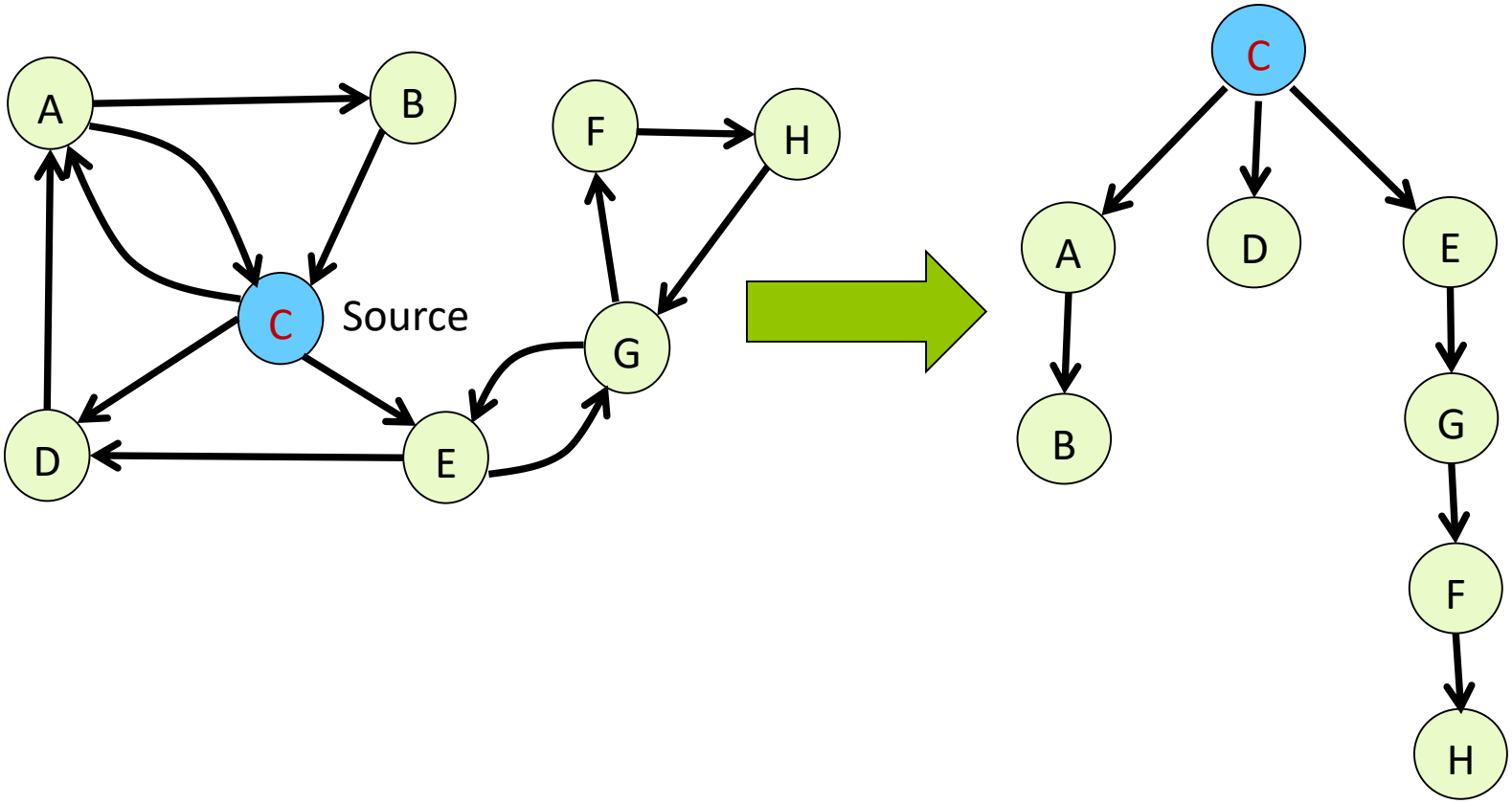
- Problem: G = (V, E) ağırlıksız grafında ***s*** başlangıç düğümü veriliyor ve ***s***'den diğer düğümlere giden en kısa yol nasıl bulunur.



- C'den diğer düğümlere giden en kısa yolu bulun?

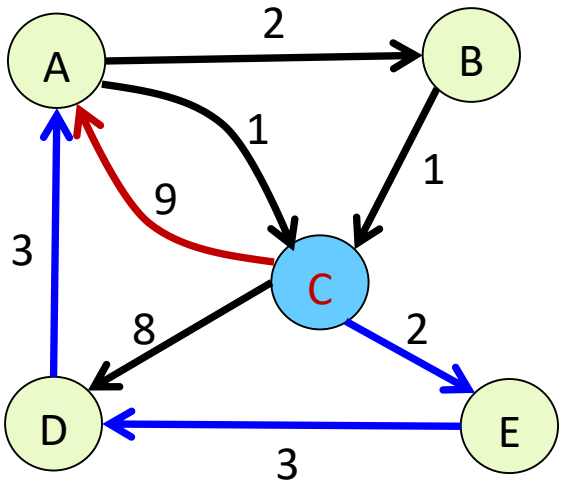
BFS Tabanlı Çözüm

- ***s*** ’den başla BFS algoritmasını uygula



Ağırlıklı Graflarda En Kısa Yol Problemi

- BFS algoritması bu graf içinde çalışır mı?
  - Hayır!



- C den A'ya en kısa yol:
  - C->A (uzunluk: 1, maliyet: 9)
  - BFS ile hesaplandı
- C den A ' ya en az maliyetli yol:
  - C->E->D->A (uzunluk: 3, maliyet: 8)
  - Peki nasıl hesaplayacağız?

Algoritmalar

- En küçük yol ağacını belirlemek için birçok algoritma geliştirilmiştir.
  - **Kruskal'ın Algoritması:** Daha az maliyetli kenarları tek tek değerlendirerek yol ağacını bulmaya çalışır. Ara işlemler birden çok ağaç oluşturabilir.
  - **Prim'in Algoritması:** En az maliyetli kenardan başlayıp onun uçlarından en az maliyetle genişleyecek kenarın seçilmesine dayanır. Bir tane ağaç oluşur.
  - **Sollin'in Algoritması:** Doğrudan paralel programlamaya yatkındır. Aynı anda birden çok ağaçla başlanır ve ilerleyen adımlarda ağaçlar birleşerek tek bir yol ağacına dönüşür.
  - **Dijkstra Algoritması:**
    - Ağırlıklı ve yönlü graflar için geliştirilmiştir.
    - Graf üzerindeki kenarların ağırlıkları 0 veya sıfırdan büyük sayılar olmalıdır.
    - Negatif ağırlıklar için çalışmaz.
  - **Bellman ve Ford Algoritması:**
    - Negatif ağırlıklı graflar için geliştirilmiştir.
  - **Floyd (veya Floyd-Warshall) Algoritması**
    - Bir düğümün komşularına en az maliyetli kenardan başlayıp kenar değerlerini güncellemeye dayanır. Dijkstra algortimasına benzer.
  - **Johnson algoritması**
    - Bellman Ford, Reweighting ve Dijkstra Algoritması tabanlı, tüm ikili yollar problemini çözmek için kullanılan bir algoritmadır.

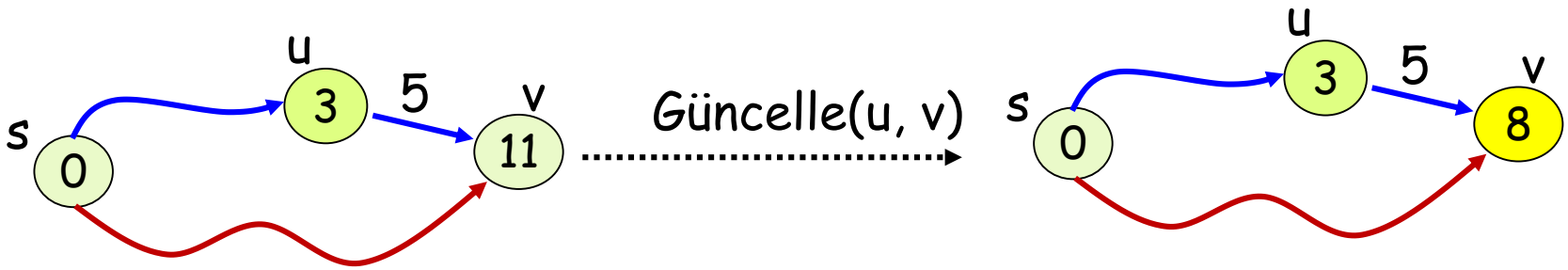
Dijkstra Algoritması

- Başlangıç olarak sadece başlangıç düğümünün en kısa yolu bilinir. (s=0'dır.)

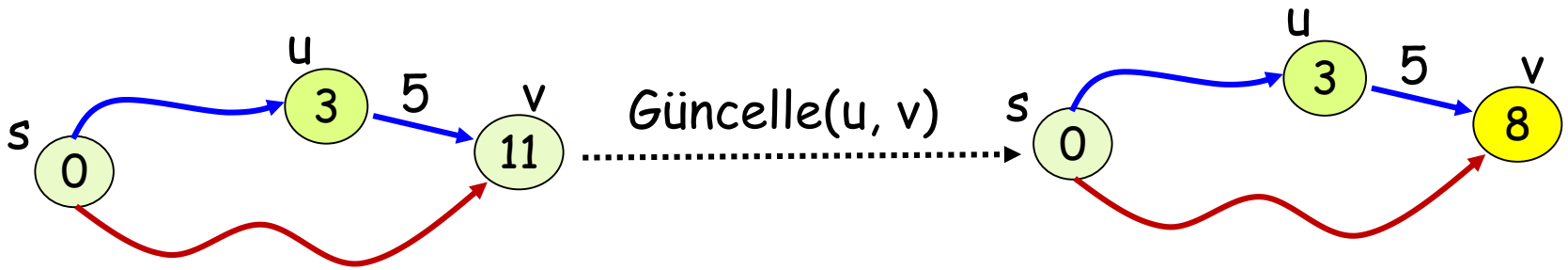
- Tüm düğümlerin maliyeti bilinene kadar devam et.
  1. O anki bilinen düğümler içerisinden en iyi düğümü şeç. (en az maliyetli düğümü seç, daha sonra bu düğümü bilinen düğümler kümesine ekle)
  2. Seçilen düğümün komşularının maliyetlerini güncelle.

Güncelleme

- Adım-1 de seçilen düğüm u olsun.
- u düğümünün komşularının maliyetini güncelleme işlemi aşağıdaki şekilde yapılır.
  - s'den v'ye gitmek için iki yol vardır.
  - Kırmızı yol izlenebilir. Maliyet 11. veya mavi yol izlenebilir. Önce s'den u'ya 3 maliyeti ile gidilir. Daha sonra (u, v) kenarı üzerinden 8 maliyetle v'ye ulaşılır.

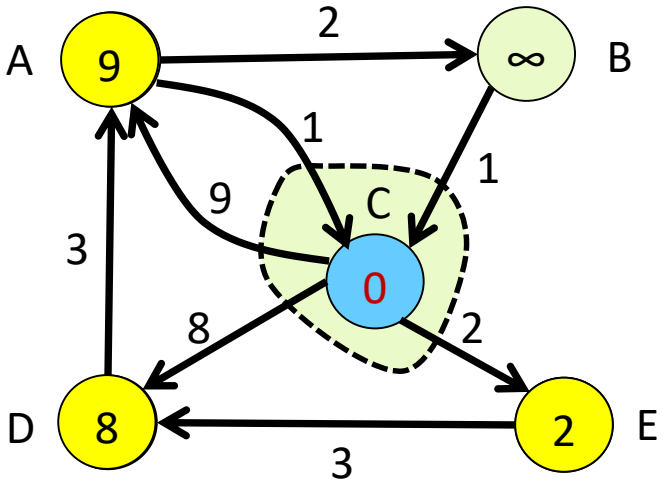


Güncelleme - Kaba Kod




```
Guncelle(u, v){
        if (maliyet[u] + w(u, v) < maliyet[v]) {      // U üzerinden yol daha kısa ise
              maliyet[v] = maliyet[u] + w(u, v);      // Evet! Güncelle
              pred[v] = u;                            // u'dan geldiğimizi kaydet.
        }
}
```

Dijkstra'nın Algoritması



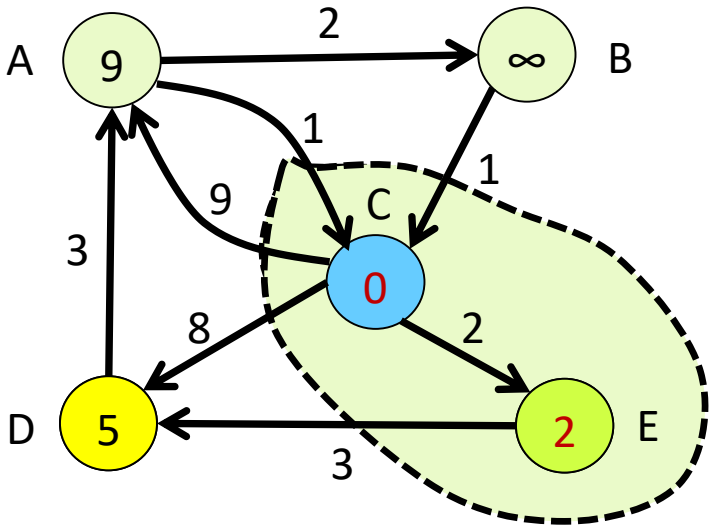
1. O anki en iyi düğümü seç – C
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

Komşu A: 0 + 9 < ∞ → maliyet(A) = 9

Komşu D: 0 + 8 < ∞ → maliyet(D) = 8

Komşu E: 0 + 2 < ∞ → maliyet(E) = 2

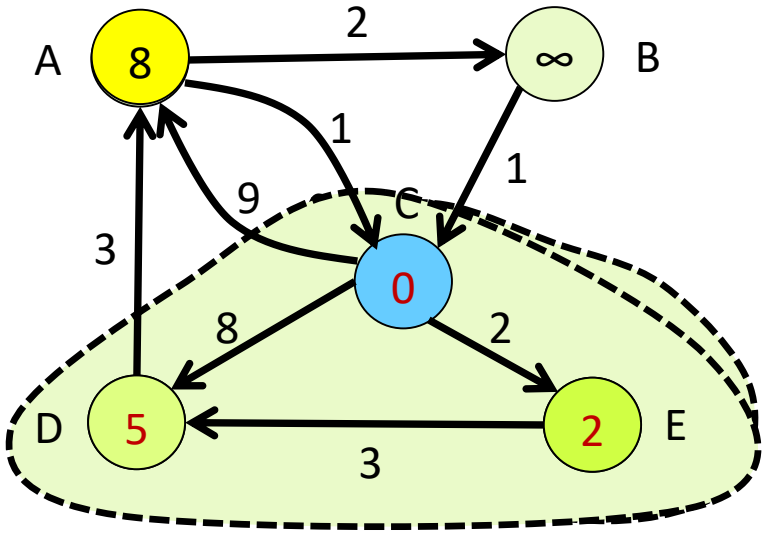
Dijkstra'nın Algoritması



1. O anki en iyi düğümü seç – E
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

Komşu D: 2 + 3 = 5 < 8 → maliyet(D) = 5

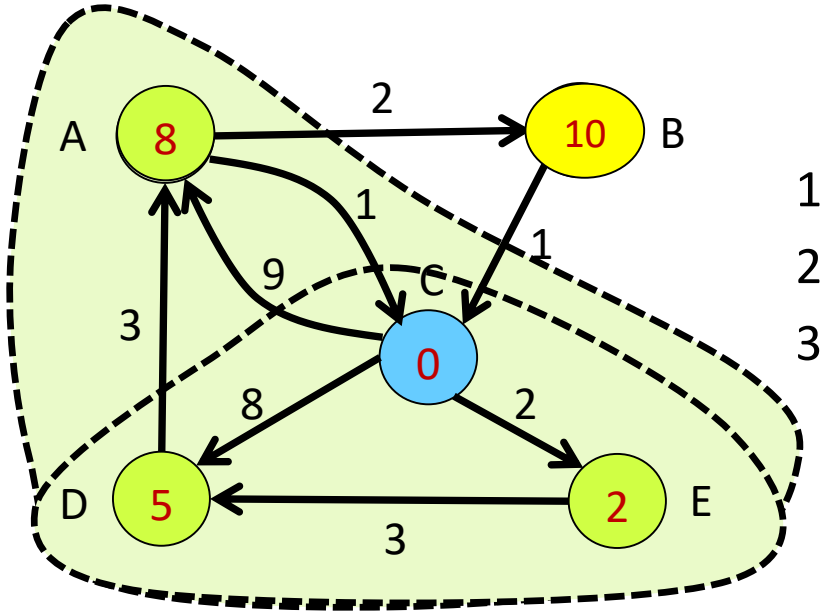
Dijkstra'nın Algoritması



1. O anki en iyi düğümü seç – D
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

Komşu A: 5 + 3 = 8 < 9 → maliyet(A) = 8

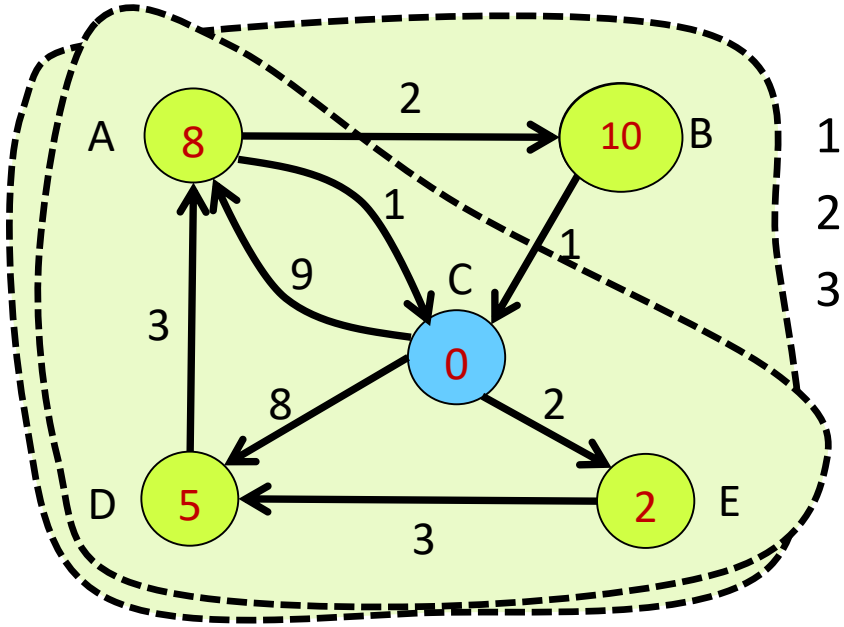
Dijkstra'nın Algoritması



1. O anki en iyi düğümü seç – A
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

Komşu B: $8 + 2 = 10 < \infty \rightarrow$ maliyet(B) = 10

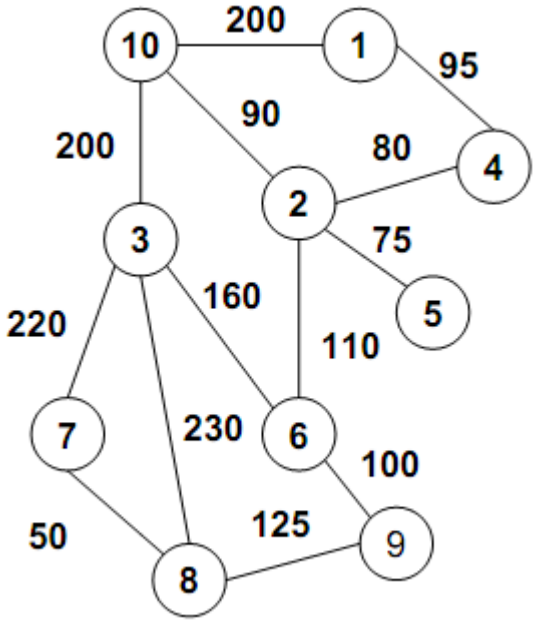
Dijkstra’nın Algoritması

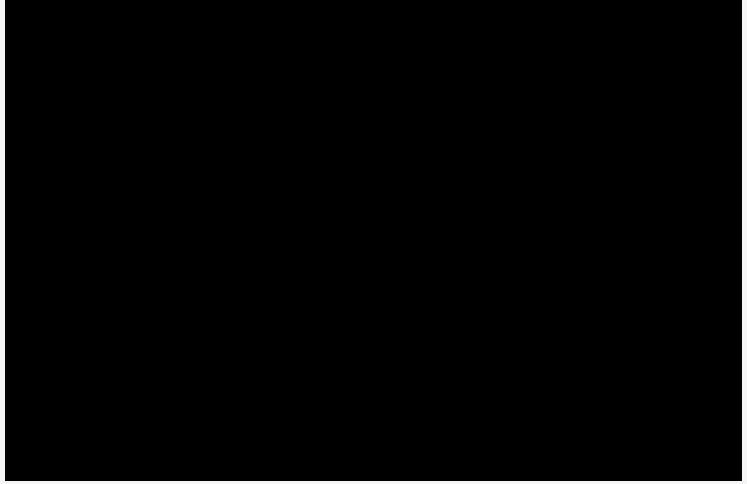


1. O anki en iyi düğümü seç – B
2. Bilinen düğümler kümesine ekle
3. Seçilen düğümün tüm komşularının maliyetini güncelle.

ÖDEV

- Programları Java/C # programları ile gerçekleştiriniz.
- 1- 10 tane şehir için bir graf yapısı oluşturunuz. Her şehirden komşu şehirlere olan uzaklık kenar ağırlıkları olarak kullanılacaktır. Herhangi bir şehirden başlayarak tüm şehirleri dolaşmak için gerekli olan algoritmayı ve grafiksel çözümü BFS ve DFS ile yapınız.
- 2-Dijkstra algoritmasını kullanarak en kısa yolu bulmak için gerekli olan algoritmayı ve grafiksel çözümü yapınız.
- 3- Bölümünüze ait haftalık ders ve sınav programının çakışmadan yerleşebilmesi için gerekli programı yazınız.
- 4-Herhangi bir labirent için gerekli komşulukları belirleyip, istenilen konumdan çıkışa gitmesi için gerekli yolu bulan programı yazınız.
- 5- Bölümünüze ait sınav programını çakışmadan yerleştirmek için gerekli olan çizimi gerçekleştirip kromatik sayıyı bulunuz. Ayrıca gerekli program kodlarını yazınız.





Örnekler

Örnekler

- **Depth First Arama**
- #include <stdio.h>
- #define max 10
- void buildadjm(int adj[][max], int n)
- {
- int i,j;
- for(i=0;i<n;i++)
- for(j=0;j<n;j++) {
- printf("%d ile %d komşu ise 1 değilse 0 gir \n", i,j);
  scanf("%d",&adj[i][j]); }
- }

Örnekler

- void dfs(int x,int visited[],int adj[][max],int n)
- {
- int j;
- visited[x] = 1;
- printf("Ziyaret edilen düğüm numarası %d\n",x);
- for(j=0;j<n;j++)
- if(adj[x][j] ==1 && visited[j] ==0)
  dfs(j,visited,adj,n);
- }

Örnekler

- void main() {
- int adj[max][max],node,n,i, visited[max];
- printf(“Düğüm sayısını girin (max = %d)\n",max);
- scanf("%d",&n);
- buildadjm(adj,n);
- for(i=0; i<n; i++)
- visited[i] =0;
- for(i=0; i<n; i++) if(visited[i] ==0) dfs(i,visited,adj,n); }

Örnekler

- Breadth-first
- #include <stdio.h>
- #include <stdlib.h>
- #define MAX 10
- struct node
- {
- int data;
- struct node *link;
- };

Örnekler

- **Breadth-first**
- **void buildadjm(int adj[][MAX], int n)**
- **{**
- **int i,j;**
- **printf("enter adjacency matrix \n",i,j);**
  for(i=0;i<n;i++)
- **for(j=0;j<n;j++) scanf("%d",&adj[i][j]);**
- **}**

Örnekler

```
struct node *addqueue(struct node *p,int val)
{
struct node *temp;
if(p == NULL)   {
p = (struct node *) malloc(sizeof(struct node));
if(p == NULL)        {
                printf("Cannot allocate\n");            exit(0);
}
 p->data = val;
 p->link=NULL;  }
```

Örnekler

- else { temp= p;
- while(temp->link != NULL){
- temp = temp->link;}
- temp->link = (struct node*)malloc(sizeof(struct node));
- temp = temp->link;
- if(temp == NULL) {
- printf("Cannot allocate\n"); exit(0); }
- temp->data = val; temp->link = NULL;}
- return(p);
- }

Örnekler

- struct node *deleteq(struct node *p,int *val){
- struct node *temp;
- if(p == NULL) {
- printf("queue is empty\n"); return(NULL); }
- *val = p->data;
- temp = p;
- p = p->link;
- free(temp);
- return(p); }

Örnekler

```
void bfs(int adj[][MAX], int x,int visited[], int n, struct node **p)
{

int y,j,k;    *p = addqueue(*p,x);
do {
*p = deleteq(*p,&y);

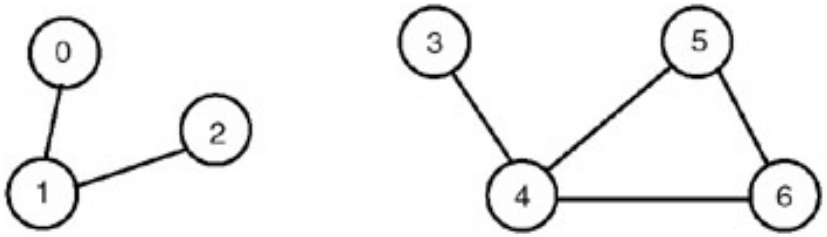
if(visited[y] == 0)  {
printf("\nnode visited = %d\t",y);
visited[y] = 1;
for(j=0;j<n;j++)
if((adj[y][j] ==1) && (visited[j] == 0))
*p = addqueue(*p,j);
      }

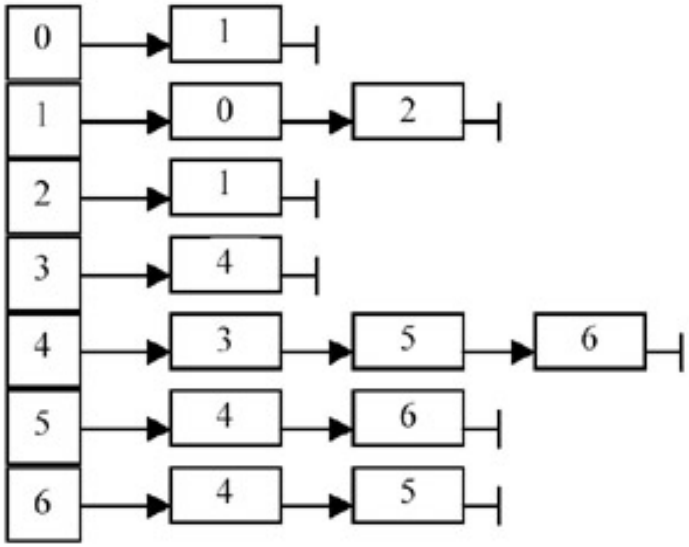
}while((*p) != NULL);}
```

Örnekler

- void main() {
- int adj[MAX][MAX];
- int n;
- struct node *start=NULL;
- int i, visited[MAX];
- printf("enter the number of nodes in graph maximum = %d\n",MAX);
- scanf("%d",&n);
- buildadjm(adj,n);
- for(i=0; i<n; i++) visited[i] =0;
- for(i=0; i<n; i++) if(visited[i] ==0) bfs(adj,i,visited,n,&start);
- }

Örnekler





Örnekler

- #include <stdio.h>
- #include <stdlib.h>
- #define MAXVERTICES 20
- #define MAXEDGES 20
- typedef enum {FALSE, TRUE, TRISTATE} bool;
- typedef struct node node;
- struct node {
- int dst;
- node *next;};

Örnekler

```
void printGraph( node *graph[], int nvert ) {
  /*    * graph çiz. */
  int i, j;
  for( i=0; i<nvert; ++i ) {
     node *ptr;
     for( ptr=graph[i]; ptr; ptr=ptr->next )
        printf( "[%d] ", ptr->dst );
     printf( "\n" );
  }
}
```

Örnekler

```
void insertEdge( node **ptr, int dst ) {
    /*
     * Başlamak için yeni düğüm ekle.
     */
    node *newnode = (node *)malloc(sizeof(node));
    newnode->dst = dst;
    newnode->next = *ptr;
    *ptr = newnode;
}
```

Örnekler

- void buildGraph (node *graph[], int edges[2][MAXEDGES], int nedges) {

- /* Graph kenar dizilerinde bitişik olanların listesi ile doldur. */
- int i;
- for(i=0; i<nedges; ++i) {
- insertEdge(graph+edges[0][i], edges[1][i]);
- insertEdge(graph+edges[1][i], edges[0][i]); // yönsüz graph.
- }
- }

Örnekler

```
void dfs( int v, int *visited, node *graph[] ) {
    /*
     * Tekrarlamalı olarak v kullanılarak ziyaret edilen graph
     * TRISTATE olarak işaretlenmekte
     */
    node *ptr;
    visited[v] = TRISTATE;
    //printf( "%d \n", v );
    for( ptr=graph[v]; ptr; ptr=ptr->next )
      if( visited[ ptr->dst ] == FALSE )
          dfs( ptr->dst, visited, graph );
}
```

Örnekler

```
void printSetTristate( int *visited, int nvert ) {
   /*
    * Tüm düğümler ziyaret edilmiş ise (TRISTATE) TRUE değerini ata.
    */
   int i;
   for( i=0; i<nvert; ++i )
     if( visited[i] == TRISTATE ) {
        printf( "%d ", i );
        visited[i] = TRUE;
     }
   printf( "\n\n" );
}
```

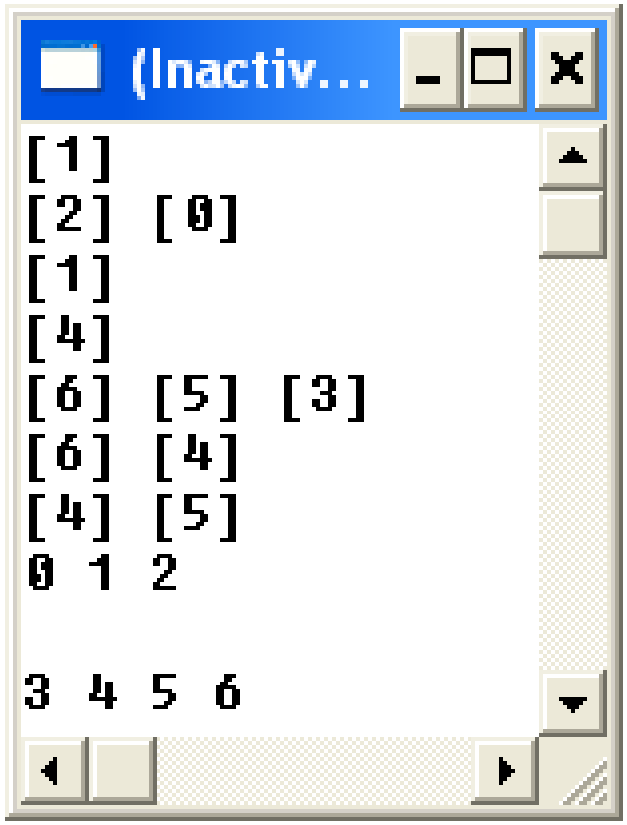
Örnekler

```
void compINC(node *graph[], int nvert) {
   /* Birbirine bağlı tüm bileşenlierin sunulduğu graph listesi.   */
   int *visited;
   int i;
   visited = (int *)malloc( nvert*sizeof(int) );
   for( i=0; i<nvert; ++i )
      visited[i] = FALSE;
   for( i=0; i<nvert; ++i )
      if( visited[i] == FALSE ) {
         dfs( i, visited, graph );
         // print all vertices which are TRISTATE. and mark them to TRUE.
         printSetTristate( visited, nvert );
      }
  free( visited );
}
```

Örnekler

```
int main() {
    int edges[][MAXEDGES] = { {0,2,4,5,5,4},
                              {1,1,3,4,6,6}
                            };
    int nvert = 7; // hiç bir düğüm.
    int nedges = 6; // ve hiçbir kenarın olmadığı graph.
    node **graph = (node **)calloc( nvert, sizeof(node *) );
    buildGraph( graph, edges, nedges );
    printGraph( graph, nvert );
    compINC( graph, nvert );
    return 0;
}
```

Örnekler



Örnekler

- http://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/
- // A C++ program to implement greedy algorithm for graph coloring
- #include <iostream>
- #include <list>
- using namespace std;
- // A class that represents an undirected graph
- class Graph
- { int V; // No. of vertices
- list<int> *adj; // A dynamic array of adjacency lists
- public:
- // Constructor and destructor
- Graph(int V) { this->V = V; adj = new list<int>[V]; }
- ~Graph() { delete [] adj; }

Örnekler

