Dengeli Arama Ağaçları

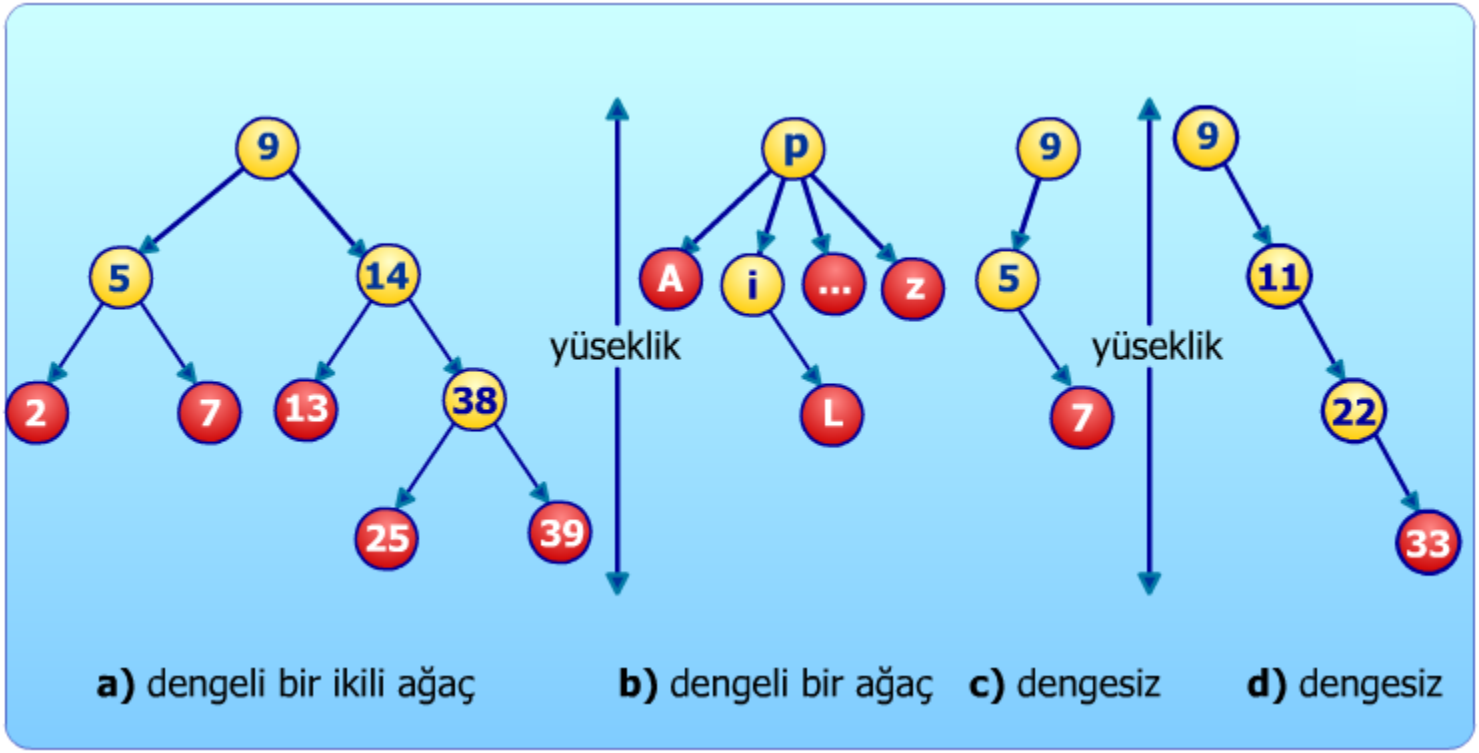
(Balanced Search Tree)

- 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

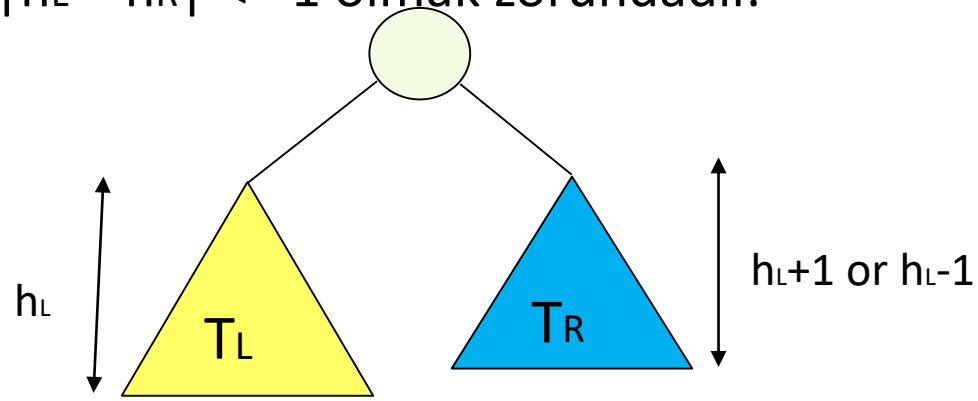
AVL TREE

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

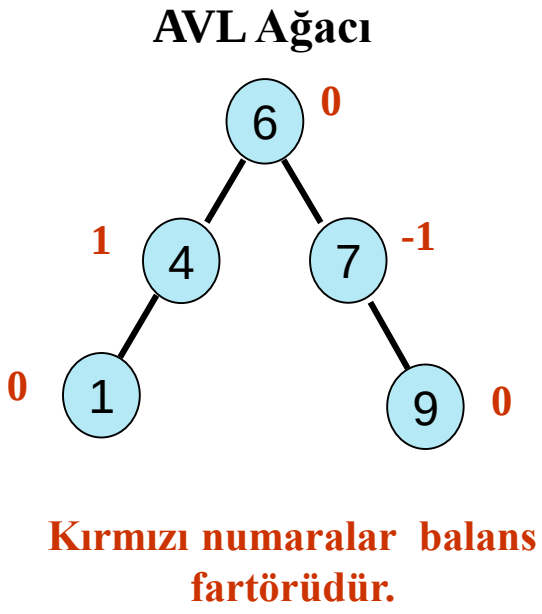
1. Tüm boş ağaç AVL ağacıdır.
2. Eğer T boş olmayan T_L ve T_R şeklinde sol ve sağ alt ağaçları olan ikili arama ağacı ise, T ancak ve ancak aşağıdaki şartları sağlarsa AVL ağacı şeklinde isimlendirilir.
    1. T_L ve T_R AVL ağaçları ise
    2. h_L ve h_R T_L ve T_R nin yükseklikleri olmak üzere $|h_L - h_R| <= 1$ olmak zorundadır.



AVL Ağaçları

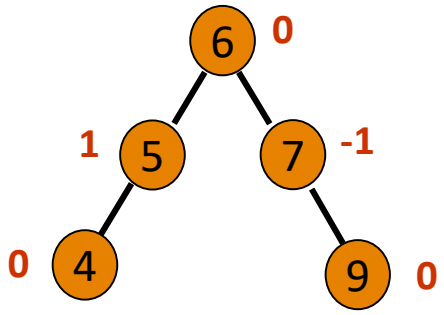
- AVL ağaçları dengeli ikili arama ağaçlarıdır.
- solh= yükseklik(sol altağaç), ve sagh=yükseklik(sağ altağaç) ise
- Bir düğümdeki denge faktörü= solh - sagh

- AVL ağaçlarında balans faktörü sadece -1, 0, 1 olabilir.
  - **Eşit ise** **0**
  - **Sol sazla ise** **1**
  - **Sağ fazla ise** **-1**

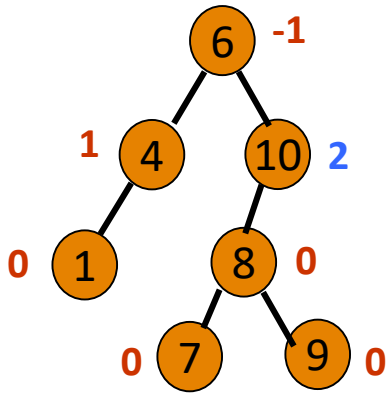


- Her bir düğümün sol ve sağ alt ağaçlarının yükseklikleri arasındaki fark en fazla 1 olabilir.

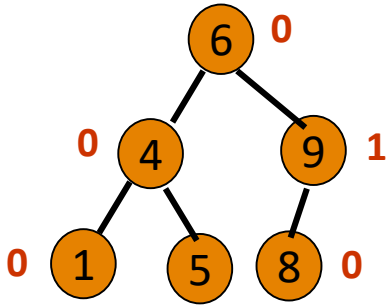
AVL Ağaçları: Örnekler



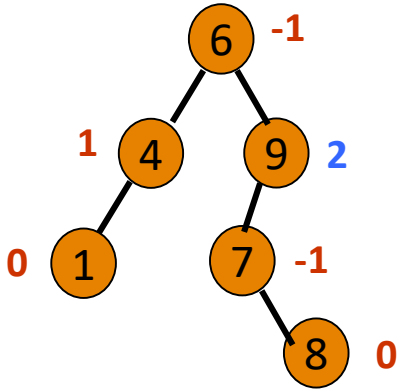
AVL Ağacı



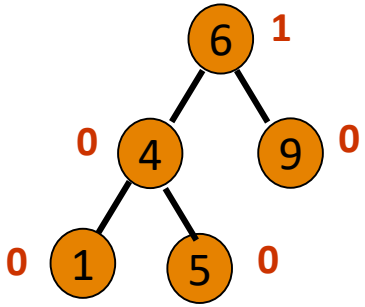
AVL Ağacı Değildir



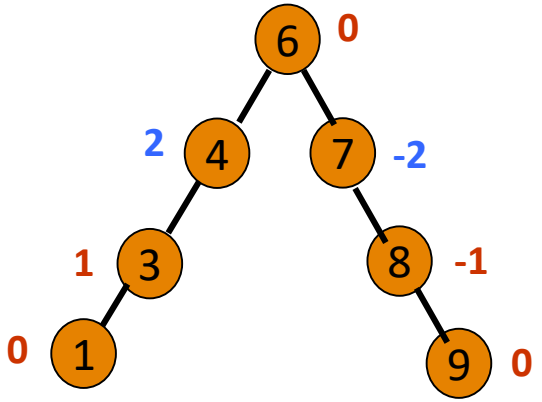
AVL Ağacı



AVL Ağacı Değildir



AVL Ağacı

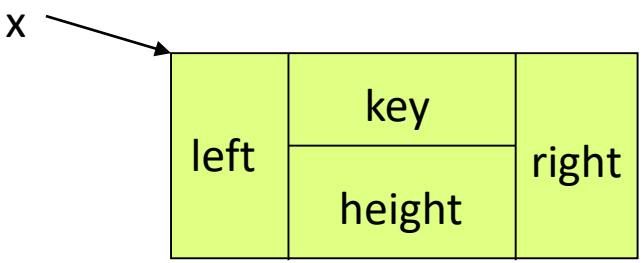


AVL Ağacı Değildir

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.



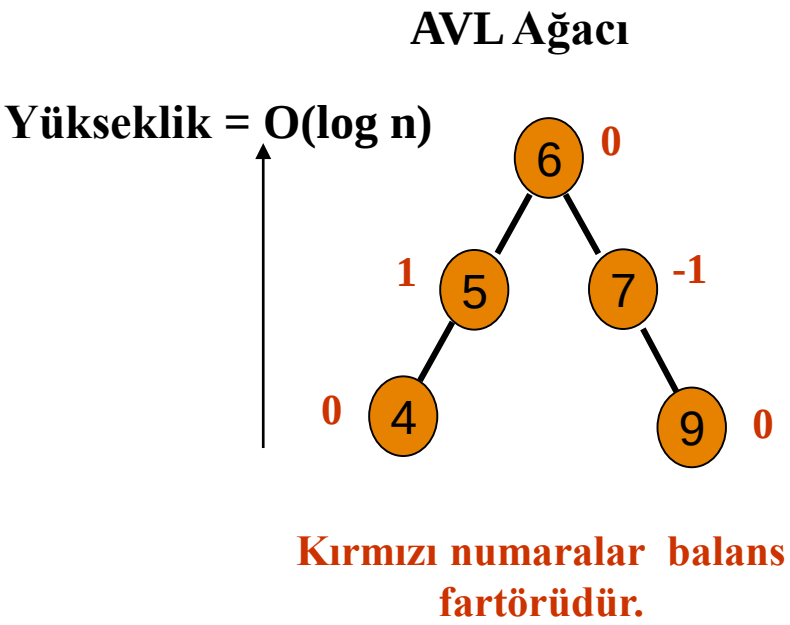

```
class AVLDugumu
{
        int deger;
        int yukseklik;
        AVLDugumu sol;
        AVLDugumu sag;
}
```

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

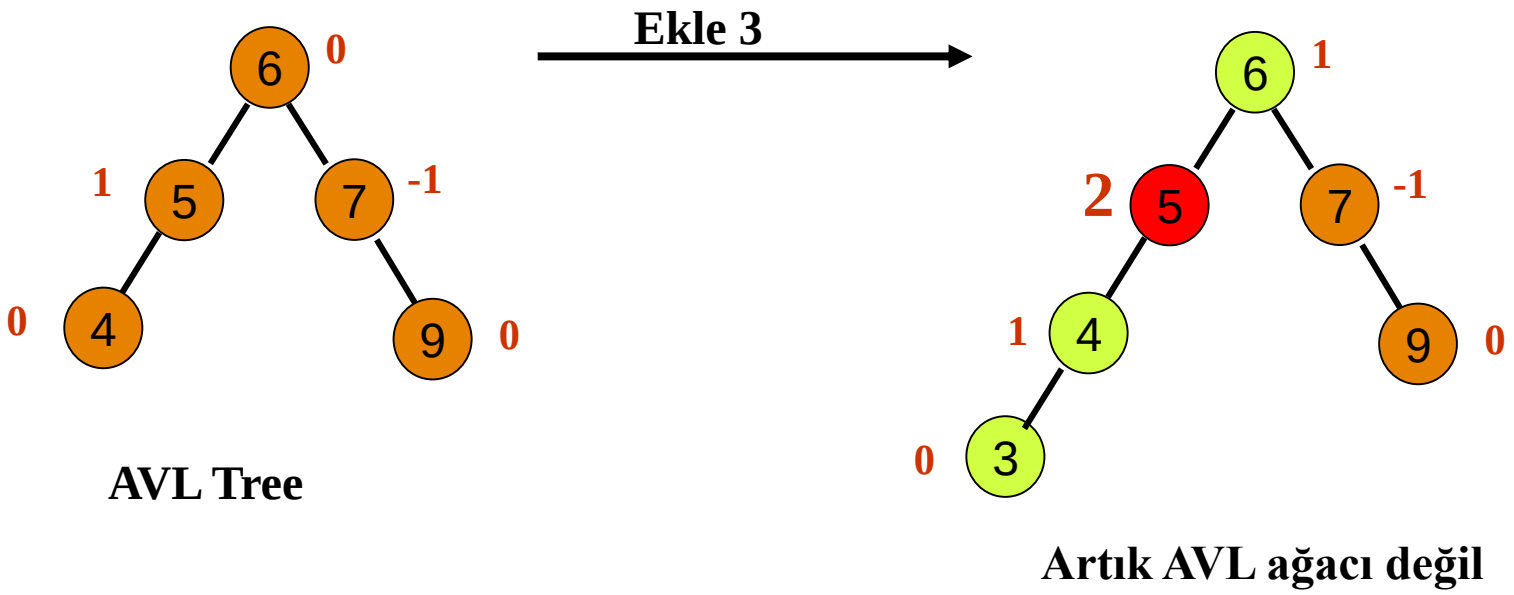
- N düğümlü bir AVL ağacının yüksekliği daima O(log n) dir.
- Peki nasıl?



Kırmızı numaralar balans fartörüdür.

AVL Ağaçları: Şanslı ve Şanssız

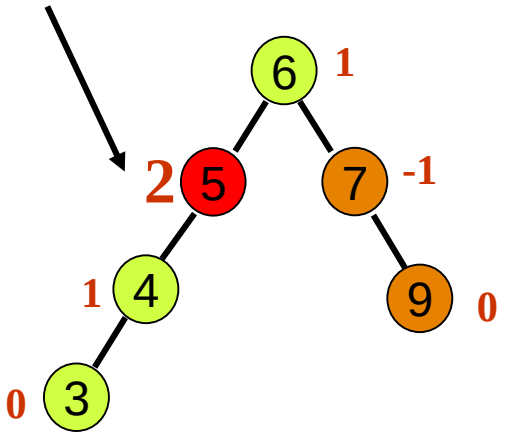
- Şanslı:
  - Arama süresi O(h) = O(log n)
- Şansız
  - Ekleme ve silme işlemleri ağacın dengesiz olmasına neden olabilir.



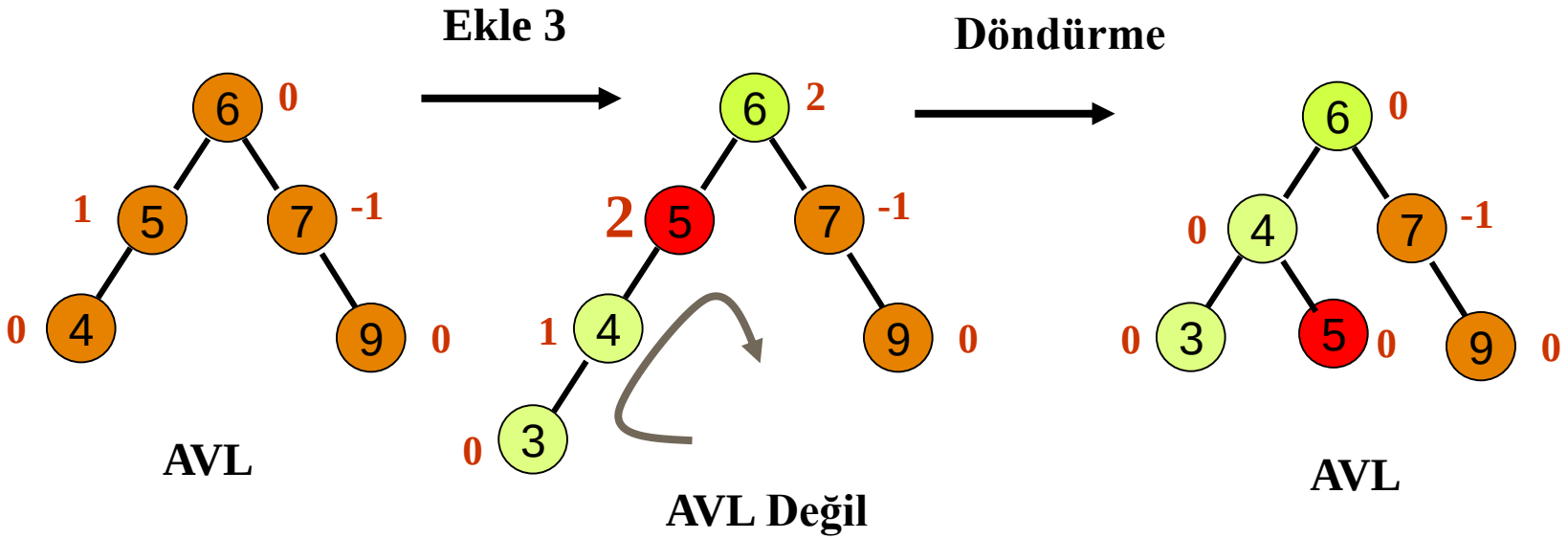
AVL Ağacında Dengenin Sağlanması

- 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.

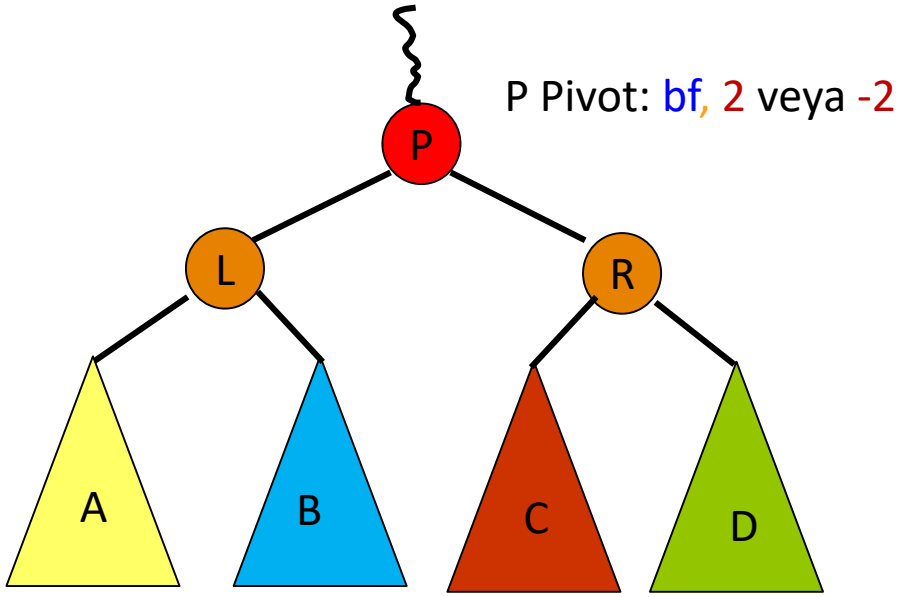


Denge Sağlama: Örnek



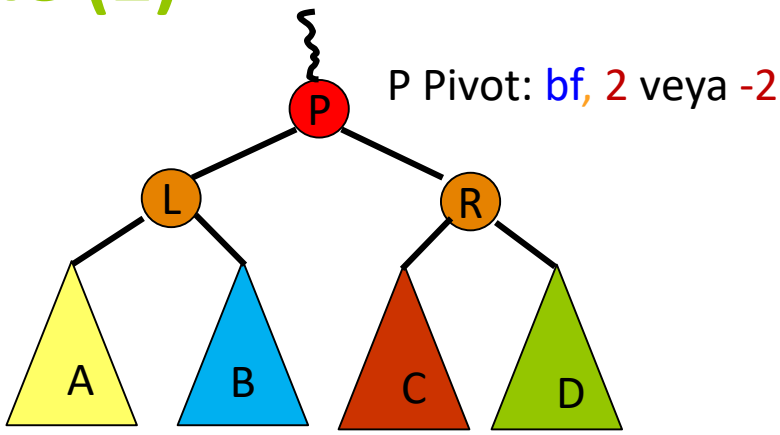
- Yeni düğümü ekledikten sonra
  1. Balans faktörü düzelterek köke doğru çık.
  2. Eğer düğümün balans faktörü 2 veya -2 ise ağaç bu düğüm üzerinde döndürülerek düzeltilir.

AVL Ağacı - Ekleme (1)



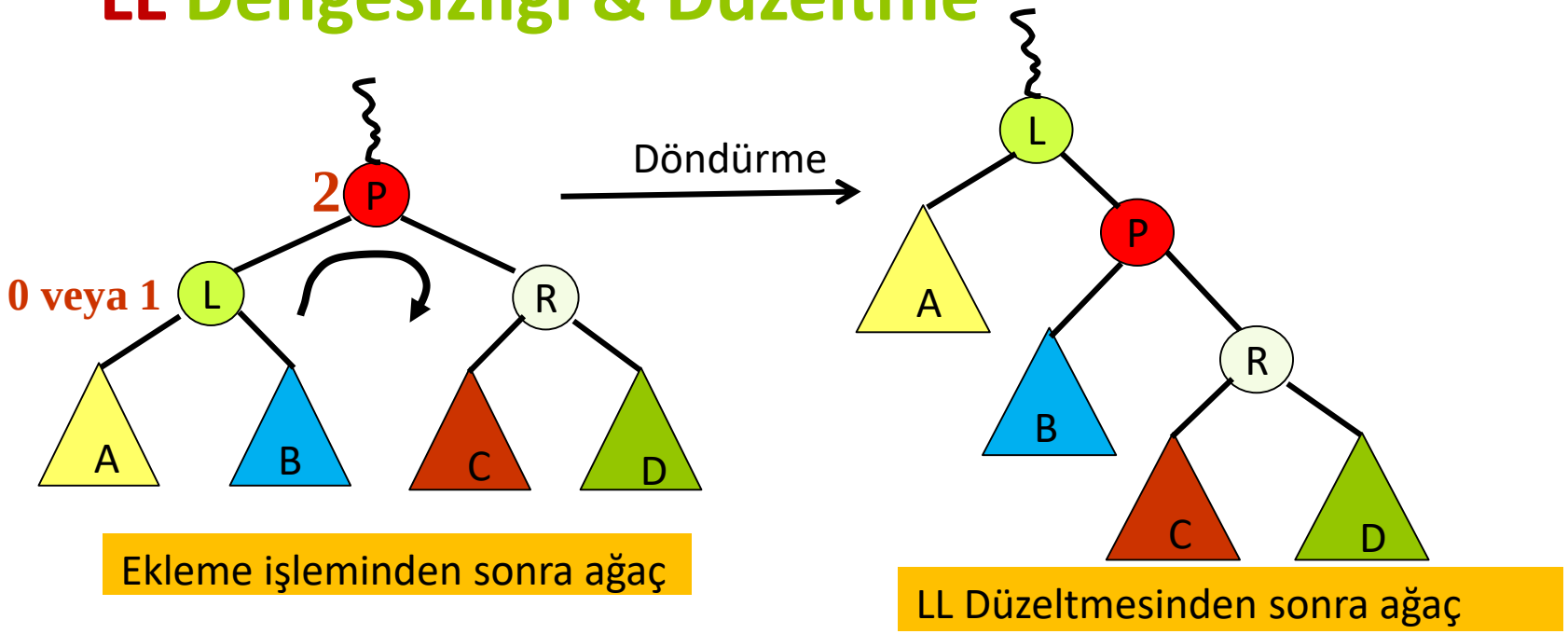
- P düğümünün dengeyi bozan düğüm olduğu düşünülürse.
  - P pivot düğüm şeklinde isimlendirilir.
  - Eklemeden sonra köke doğru çıkarken bf 'nin 2 veya -2 olduğu ilk düğümdür.

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

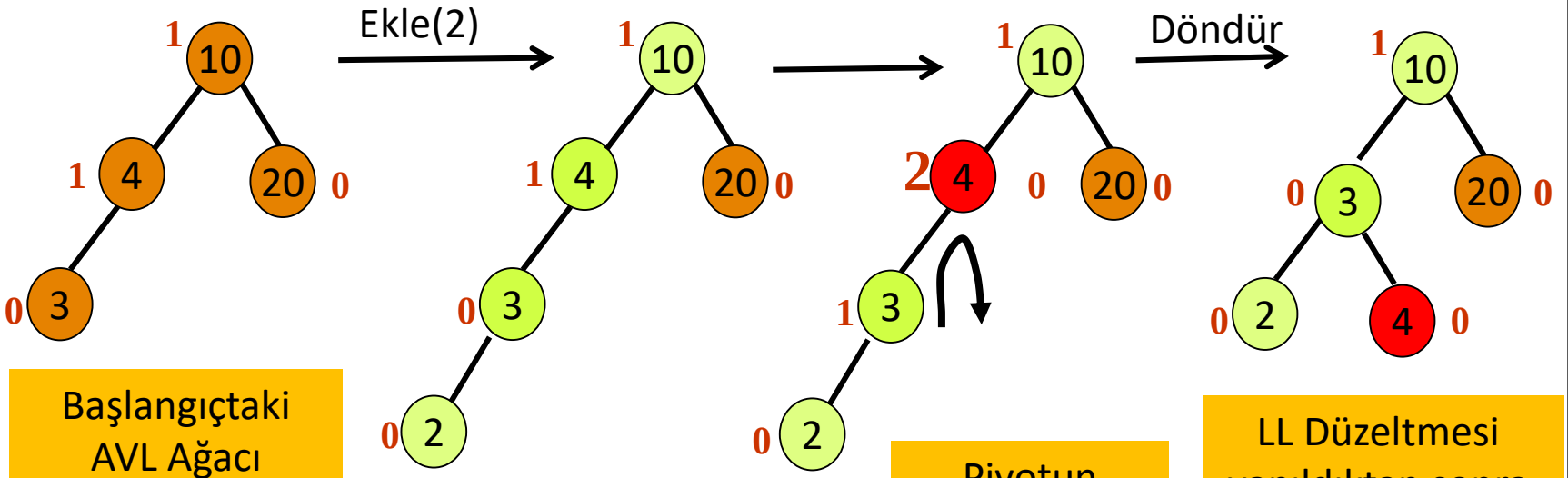


Ekleme işleminden sonra ağaç

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

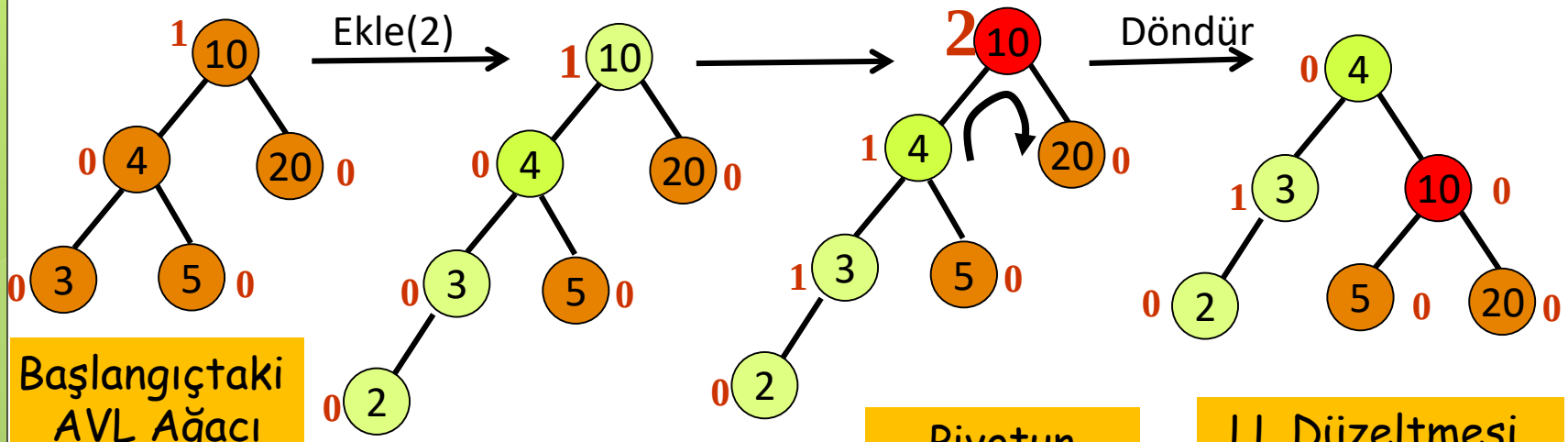
LL Düzeltmesi yapıldıktan sonra AVL ağacı

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

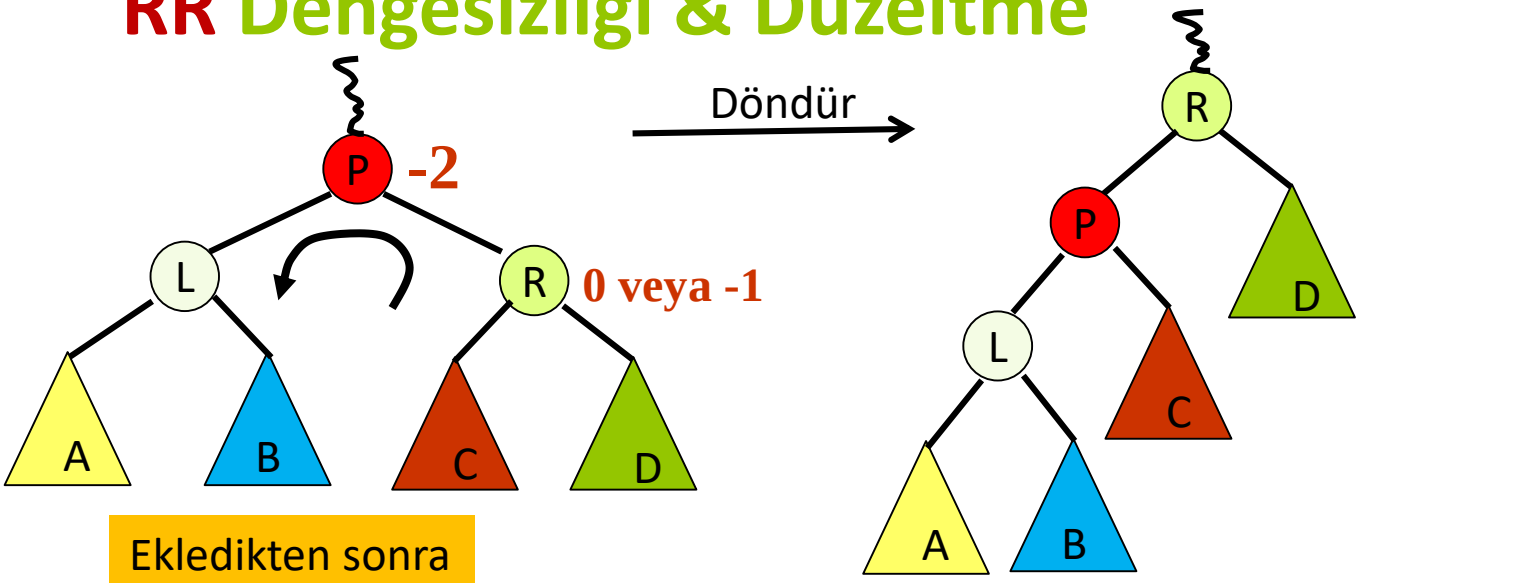
LL Düzeltmesi yapıldıktan sonra AVL ağacı

Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

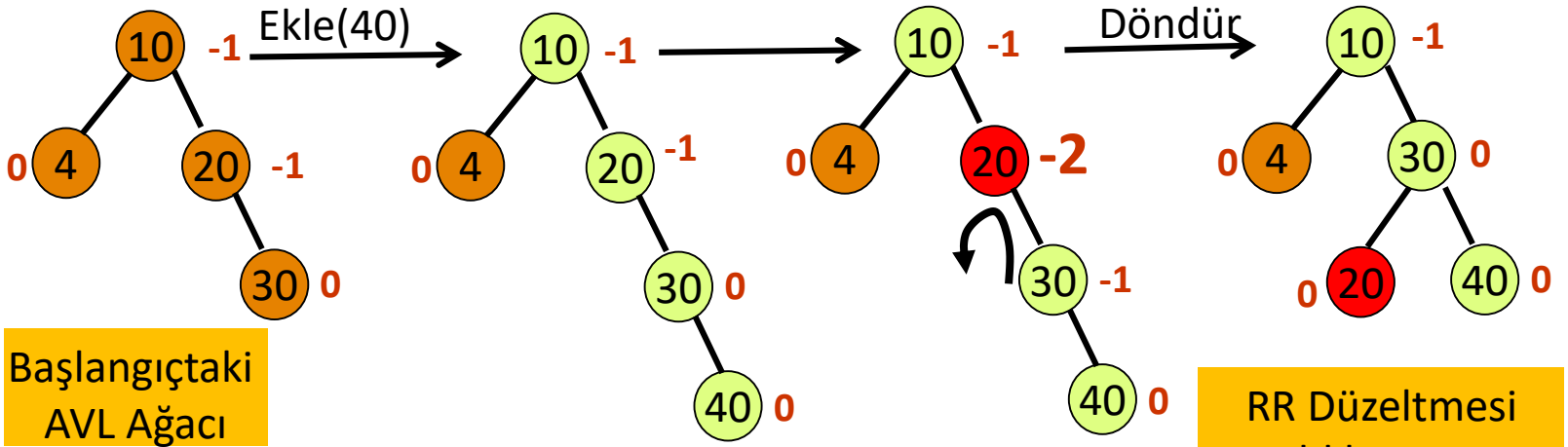
- LL Dengesizliği:
  - P(4)'ün bfsi 2
  - L(3)'ün bfsi 0 veya 1

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

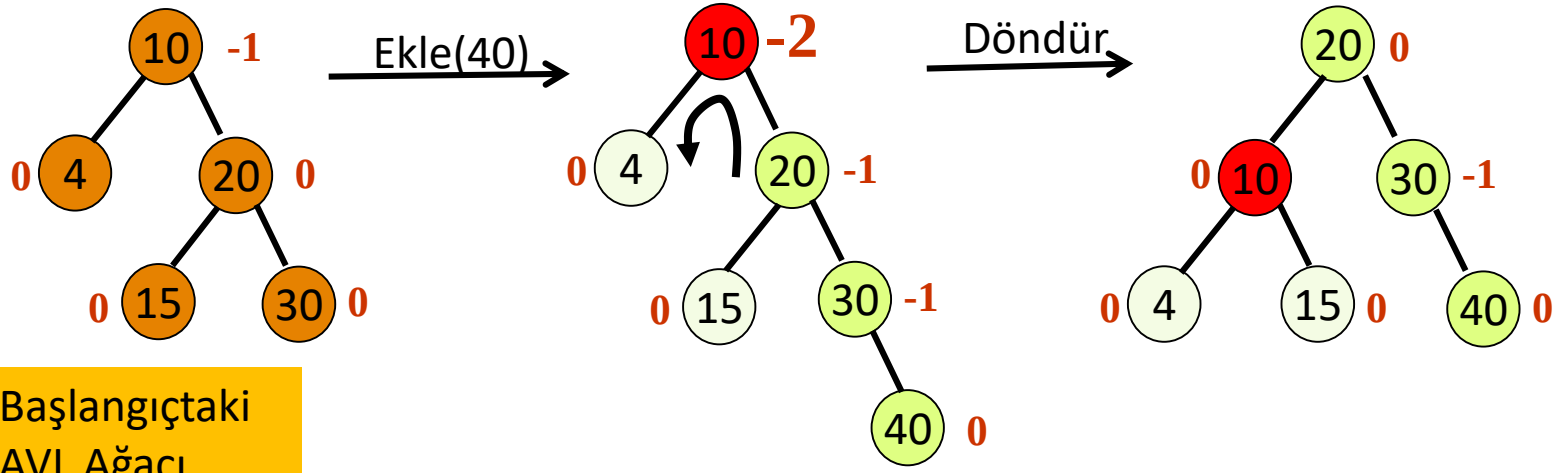
RR Düzeltmesi yapıldıktan sonra AVL ağacı

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

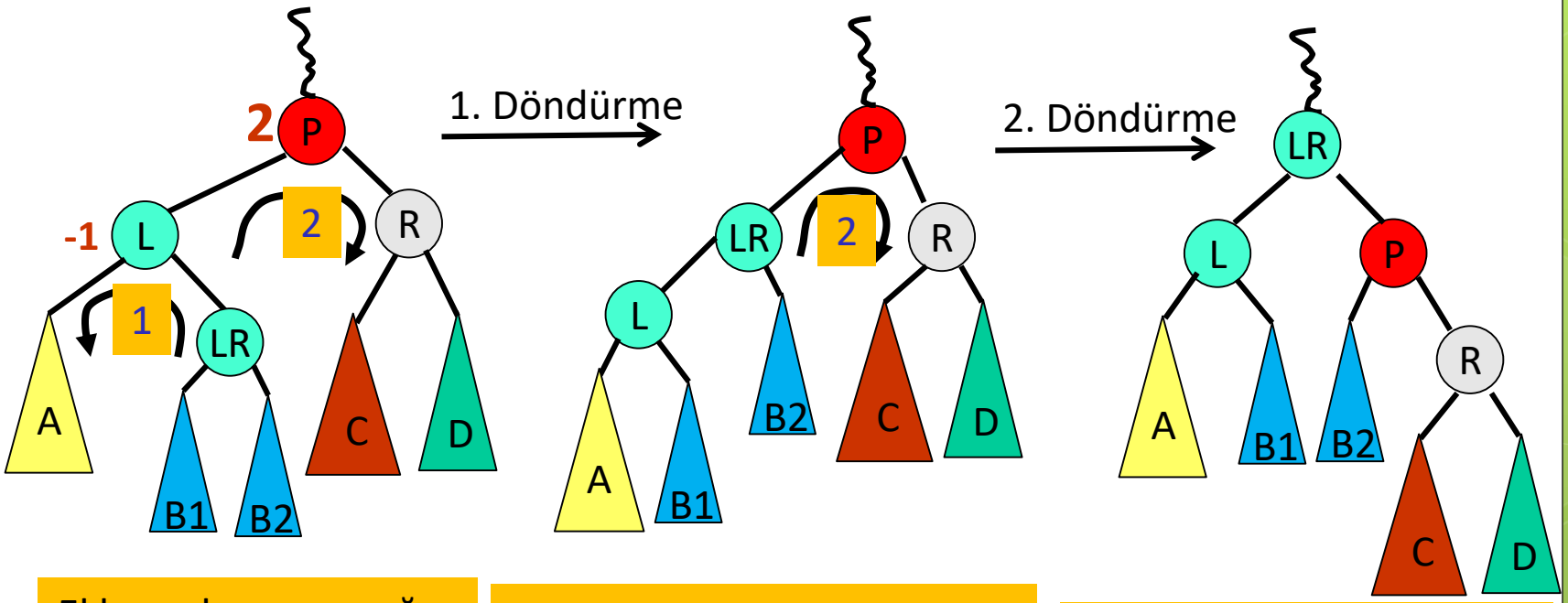
RR Düzeltmesi yapıldıktan sonra AVL ağacı

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



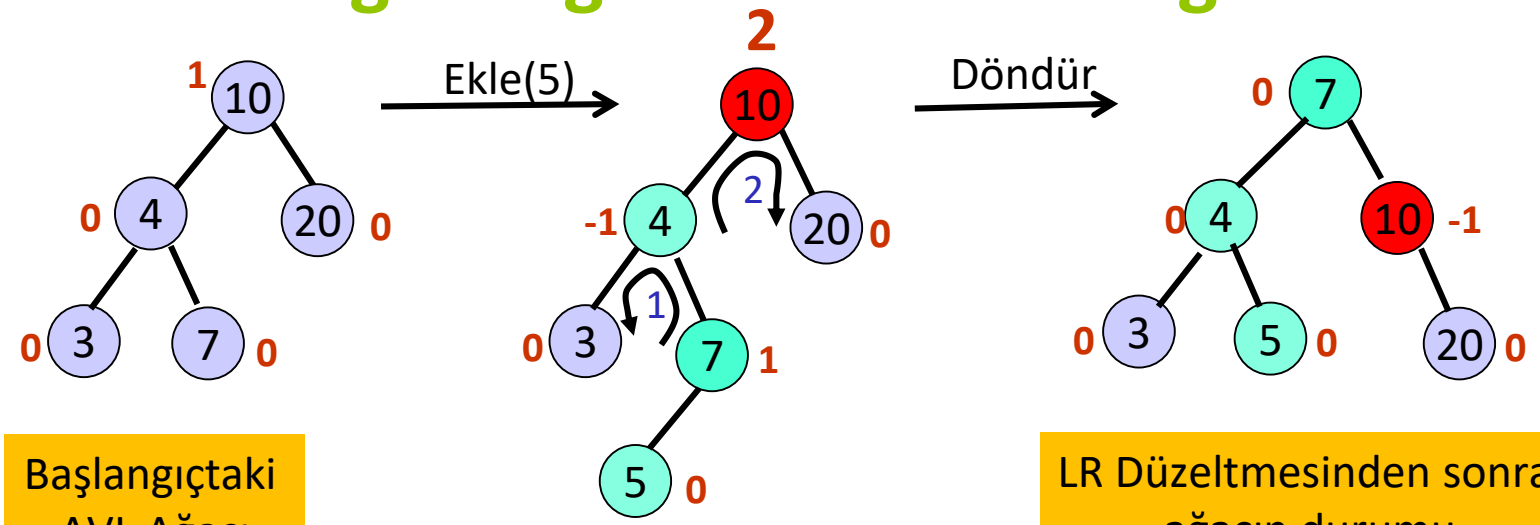
Eklemeden sonra ağaç

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

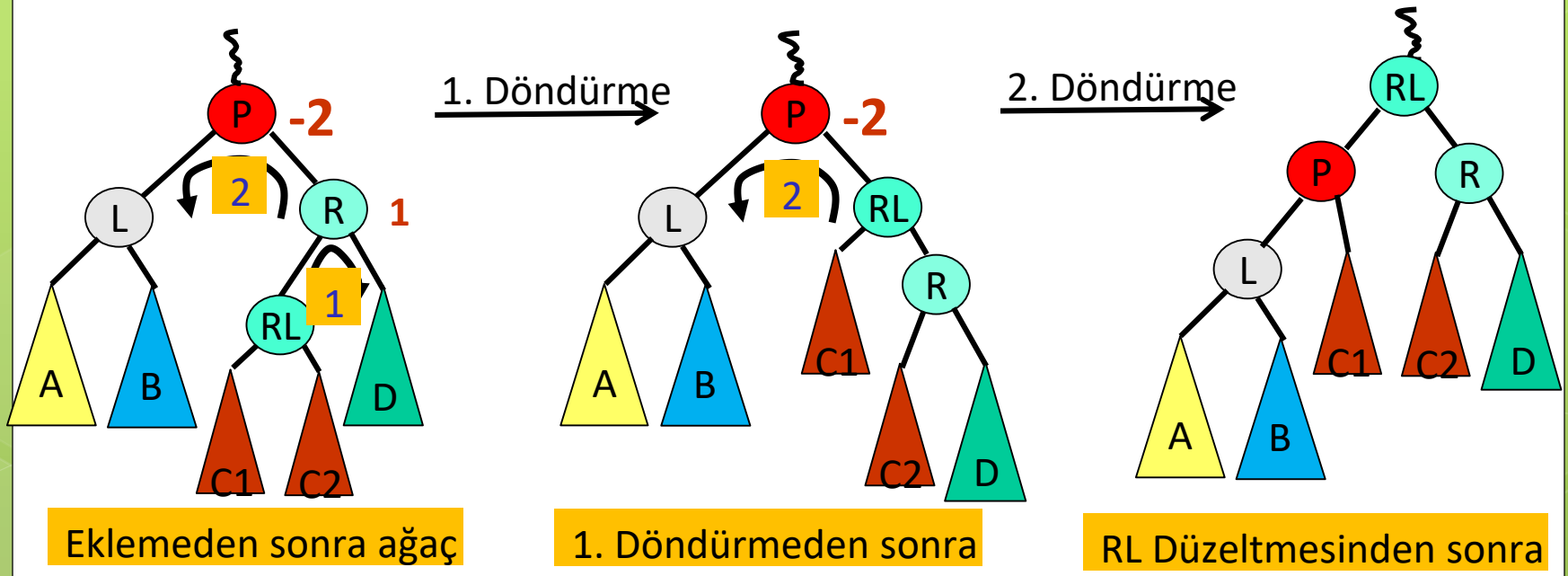
LR Düzeltmesinden sonra ağacın durumu

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



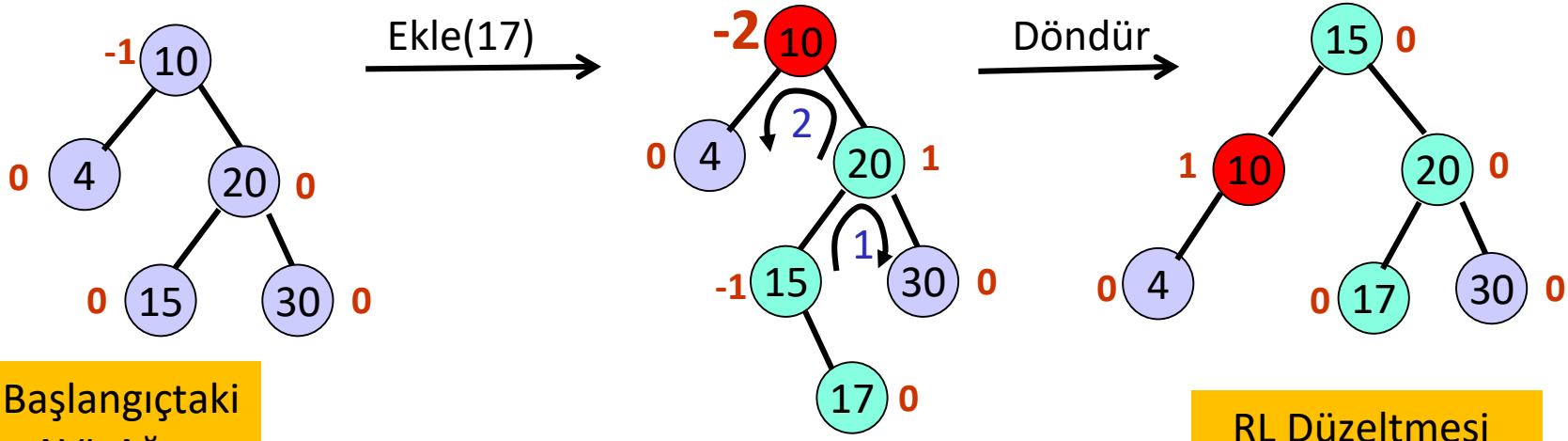
Eklemeden sonra ağaç

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;
}
```

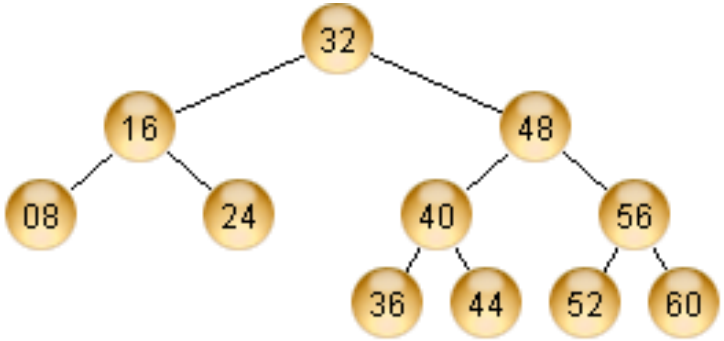
AVL AĞAÇLARI

- **Fonksiyon 7.** AVL Ağacı için RL çift döndürme

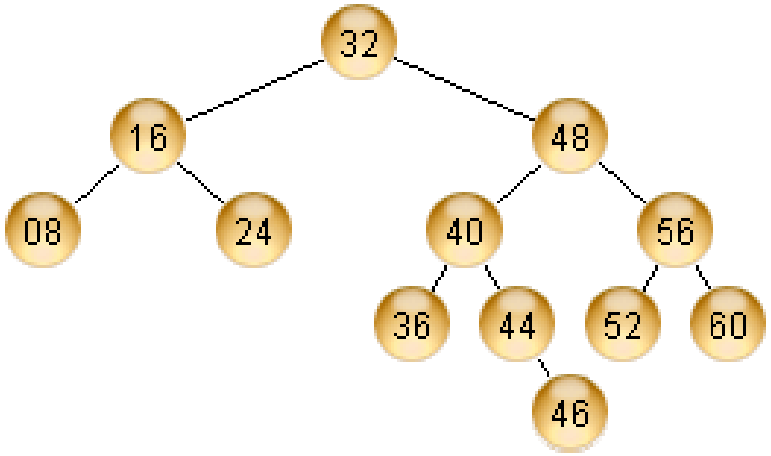
```
AvlAgacdoublerotateleft(AvlAgac p)
 {
 p.right = singlerotateright(p.right);
 return singlerotateleft(p);
 }
```

AVL AĞAÇLARI

- **Örnek:**

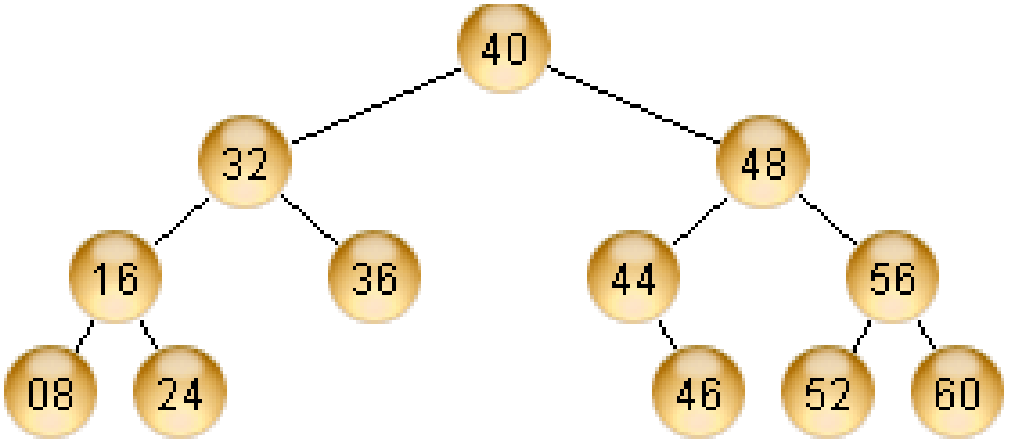


- **46, eklendi**



AVL AĞAÇLARI

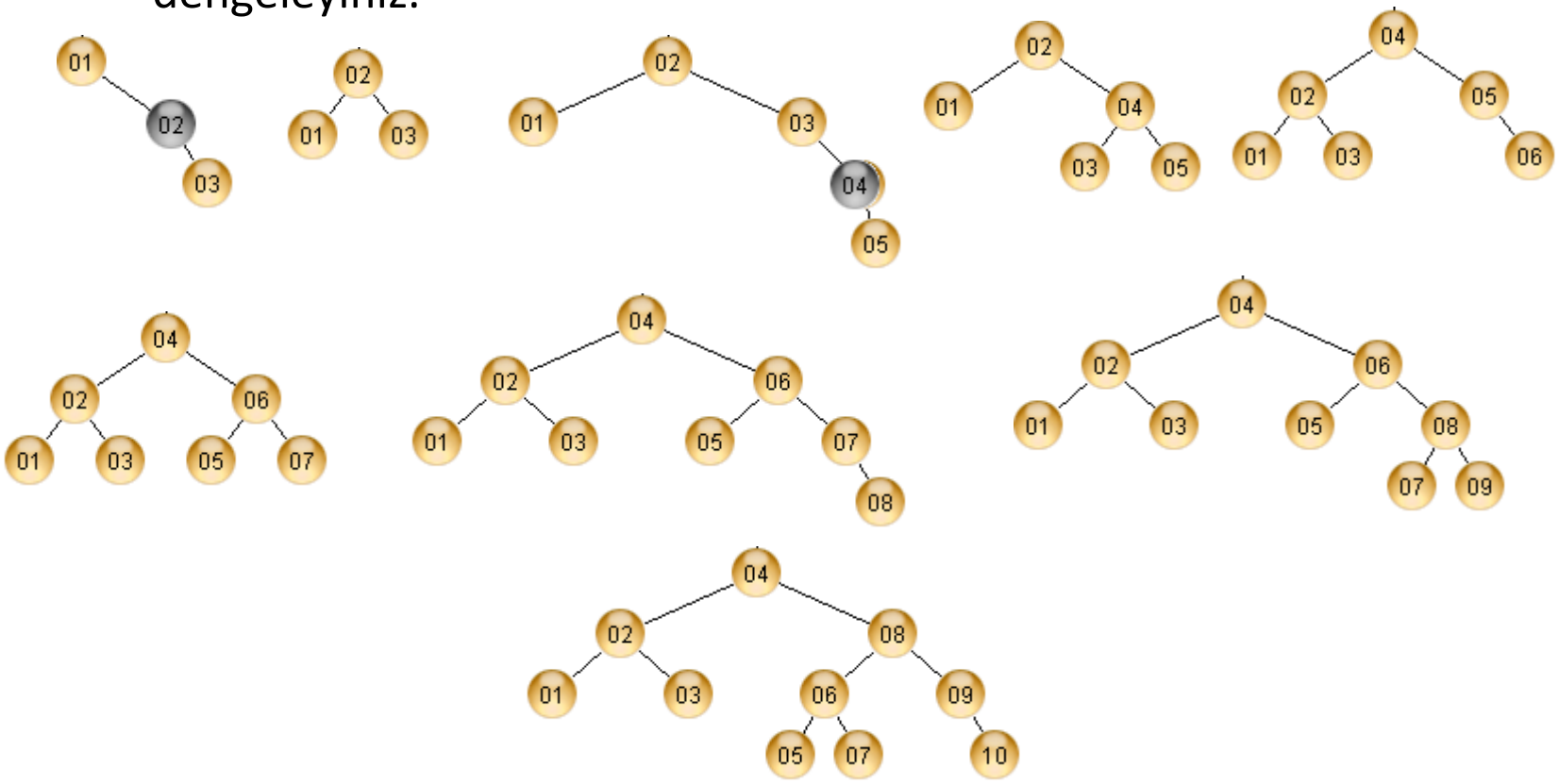
- **Örnek:**
-
-



- Dengeli AVL ağacı

AVL AĞAÇLARI

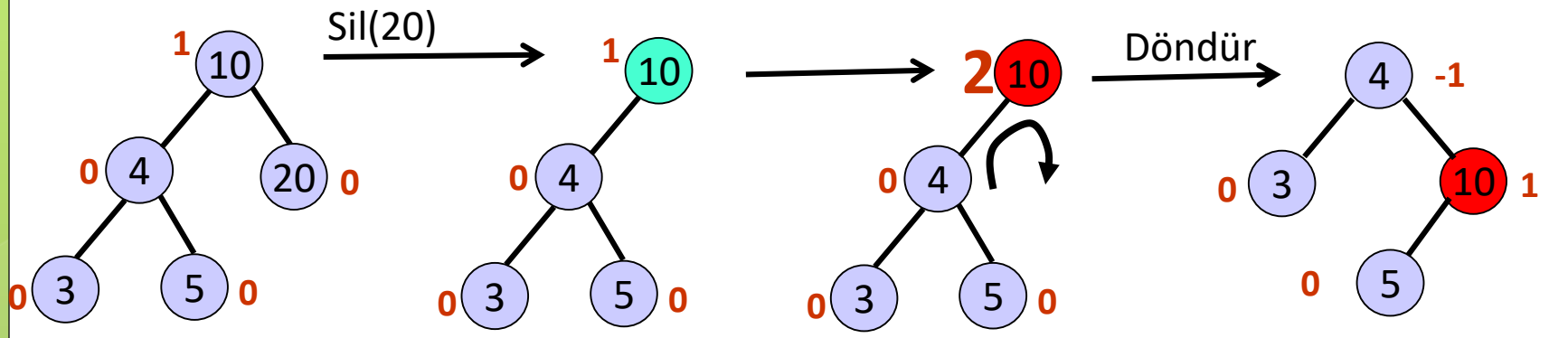
- Örnek: 1,2,3,4,5,6,7,8,9,10 ağacı oluşturup adım adım dengeleyiniz.



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

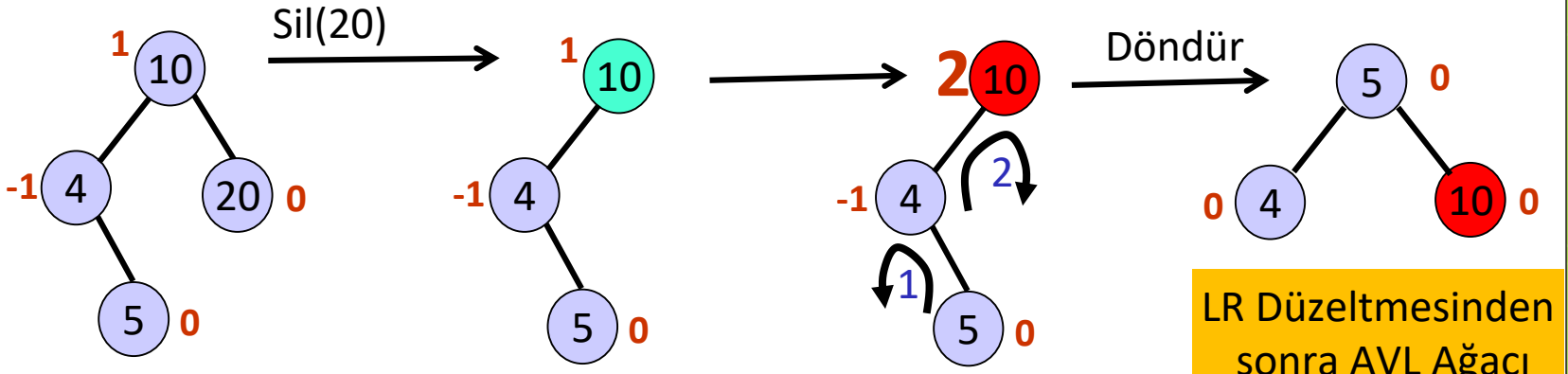
Balans faktörü=0 ise en az döndürme olacak şekilde seç.

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

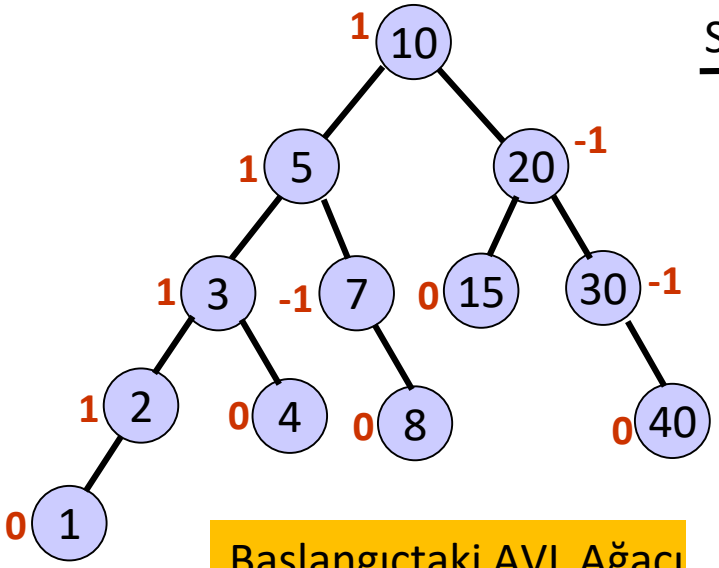
Balans faktörü düzelterek köke doğru ilerle

Dengesizliğin türünün belirlenmesi

LR Dengesizliği:

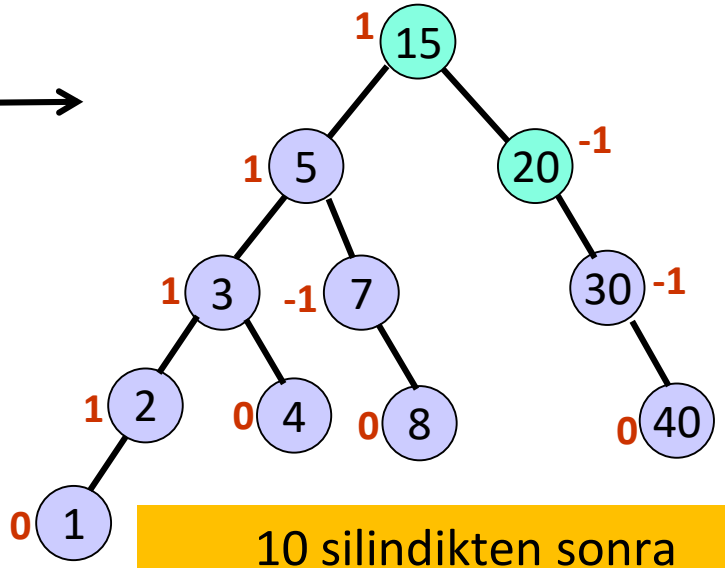
- P(10) → bf = 2
- L(4) → bf = -1

Silme Örneği (3)



Başlangıçtaki AVL Ağacı

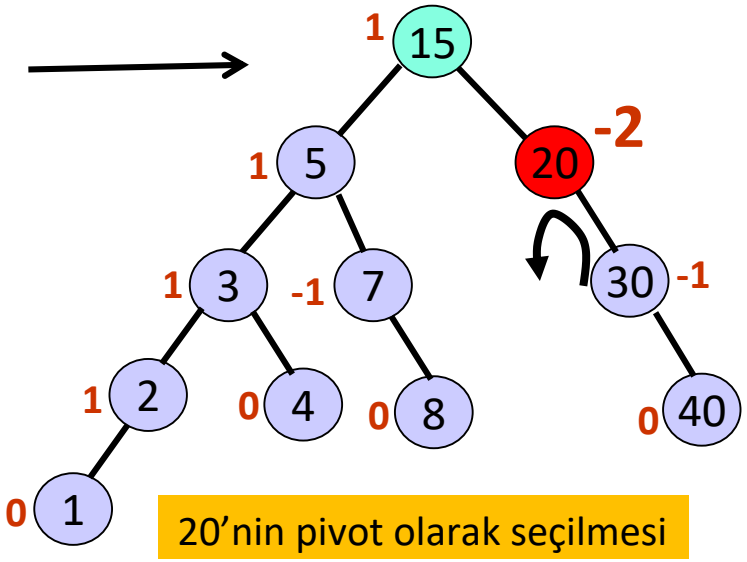
Sil(10)



10 silindikten sonra
10 ve 15(sağ alt ağaçtaki en küçük eleman) yer değiştirdi ve 15 silindi.

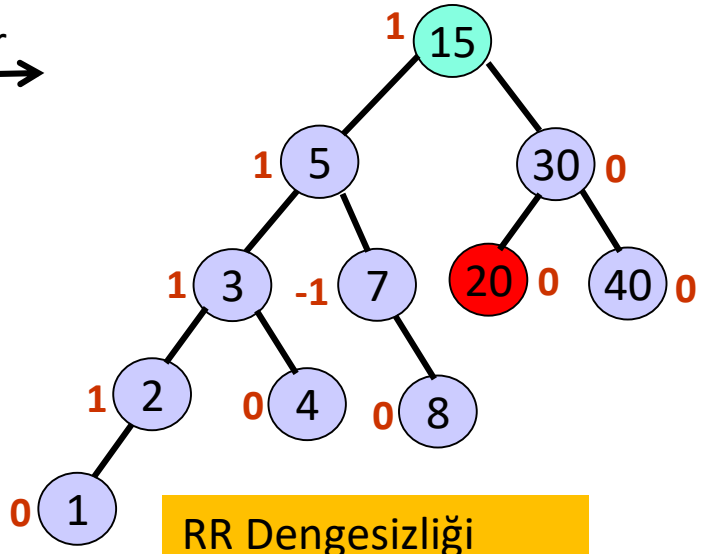
Balans faktörü düzelterek köke doğru ilerle

Silme Örneği (3)



20'nin pivot olarak seçilmesi

Döndür



RR Dengesizliği düzeltildikten sonra

Dengesizliğin türünü belirle

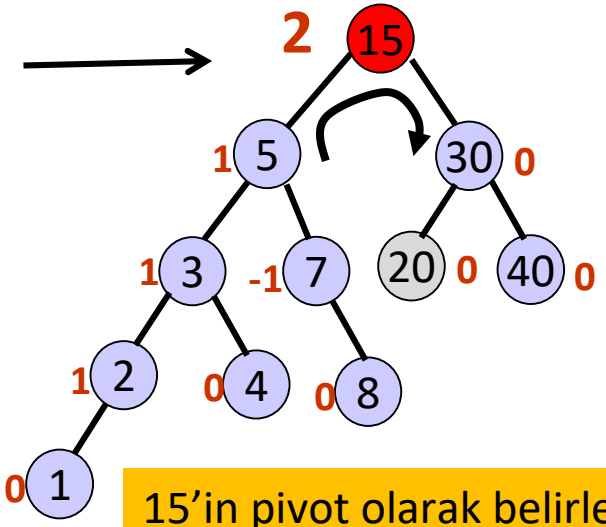
RR Dengesizliği:

- P(20) → bf = -2
- R(30) → bf = 0 veya -1

Yukardaki AVL ağacı mıdır?

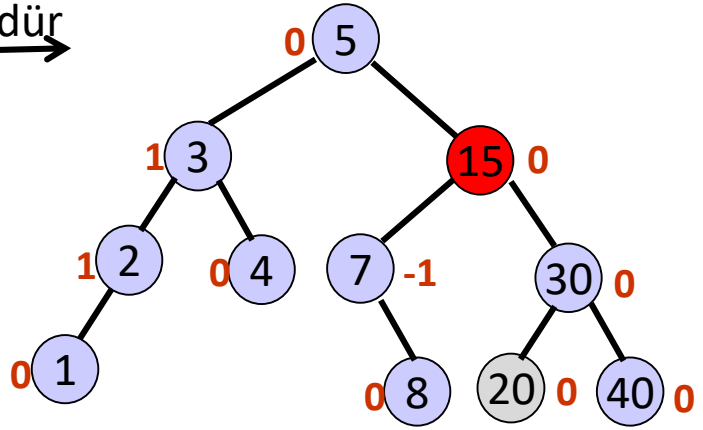
Balans faktörü düzelterek köke doğru ilerle.

Silme Örneği (3)



15'in pivot olarak belirlenmesi

Döndür



Düzeltilmiş AVL ağacı

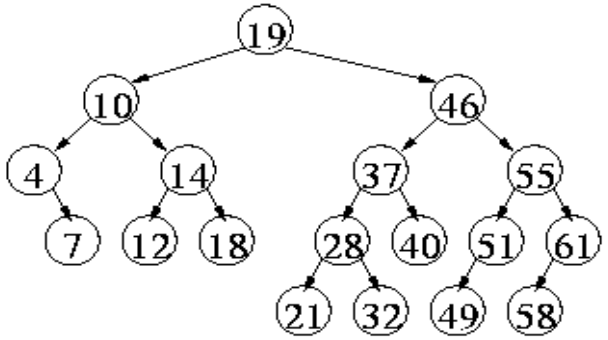
Dengesizliğin türünü belirle

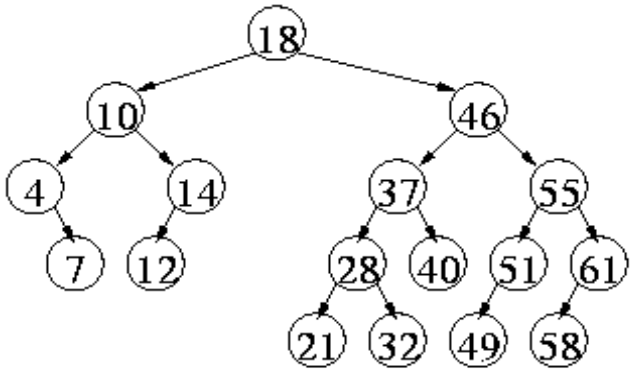
LL Dengesizliği:

- P(15) → bf = 2
- L(5) → bf = 0 veya 1

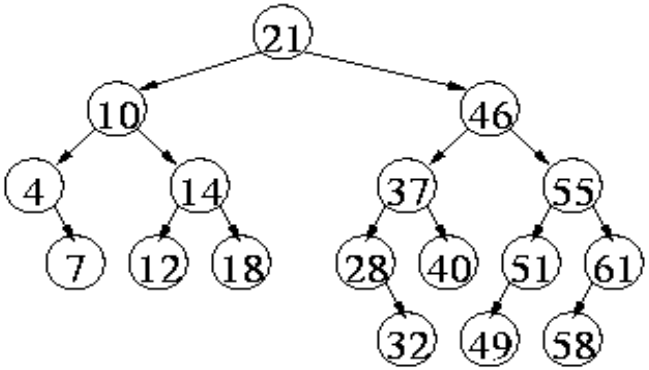
Silme Örneği– Kural 2

- **19 silindi (Soldaki en büyük düğüm veya sağdaki en küçük düğüm kök yapılır.**



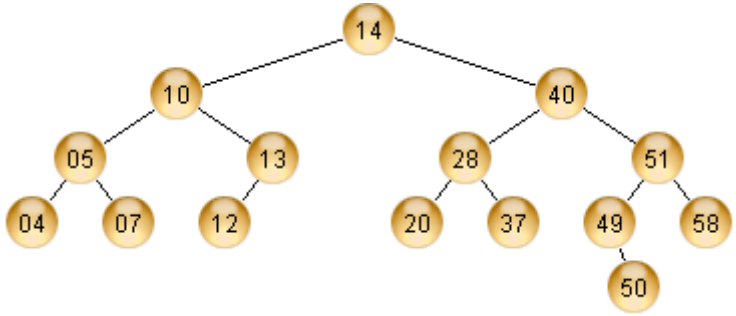


veya

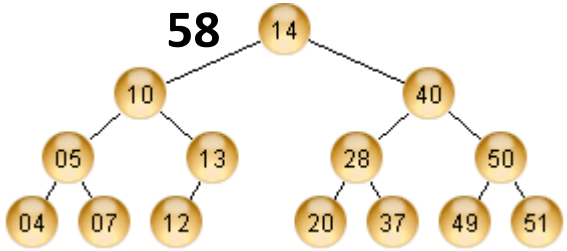


Silme Örneği– Kural 1-2

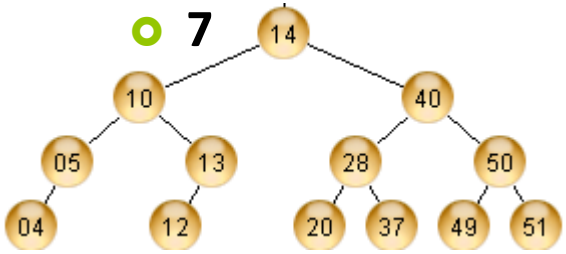
- **Örnek:** Sırasıyla siliniz; 58 ,7, 40,10,14 (Silme işleminde Soldaki en büyük düğüm alınacak)



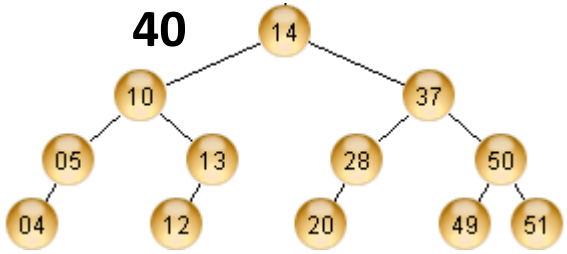
58



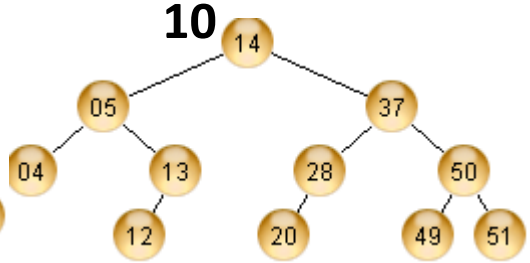
- **7**



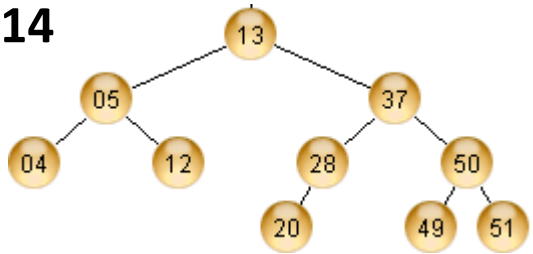
40



10

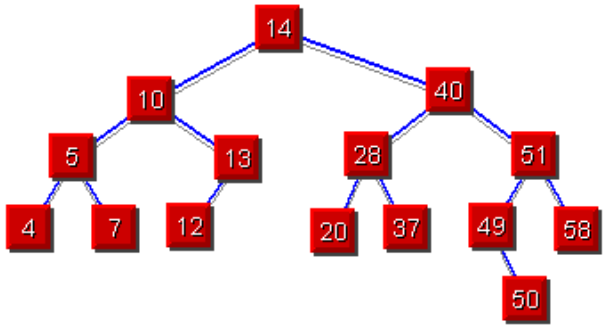


- **14**

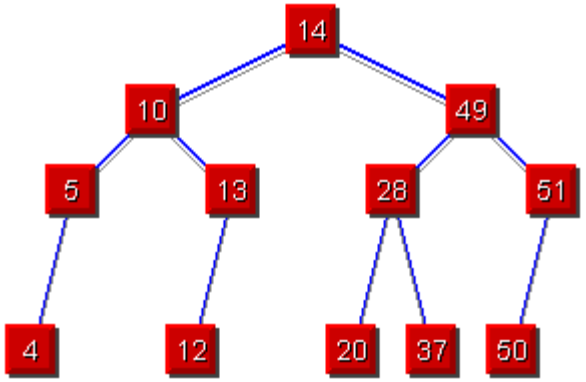


Silme Örneği– Kural 1-2

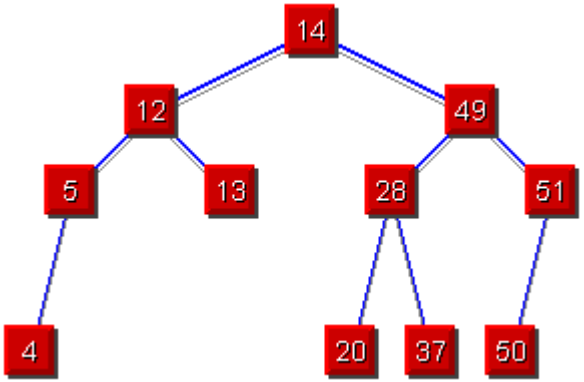
- **Örnek:** Sırasıyla siliniz; 40,58 , 7, 10,14 (Silme işleminde Sağdaki en küçük düğüm alınacak)



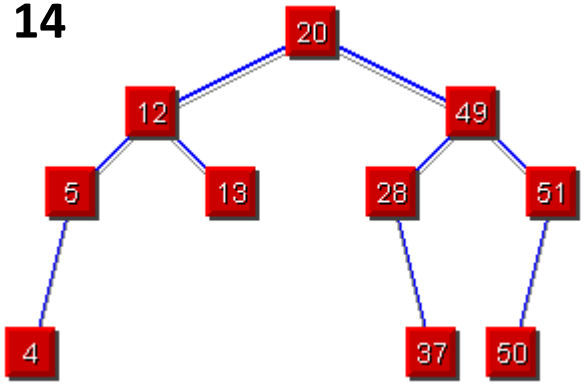
40,58,7



- **10**

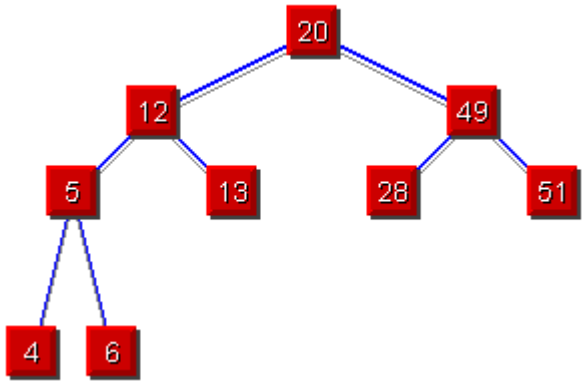


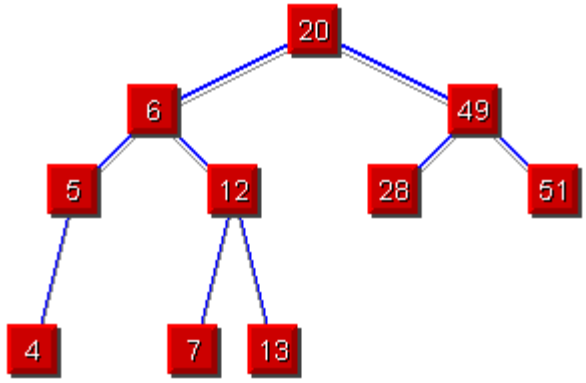
14



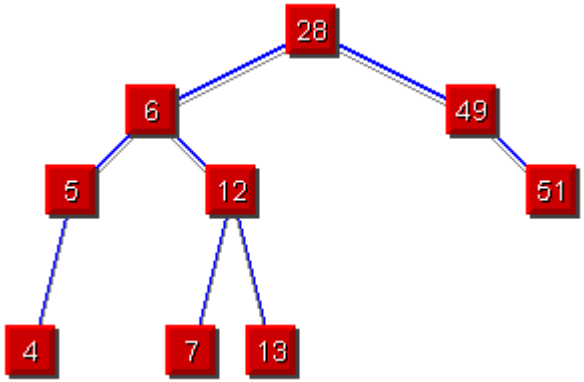
Silme Örneği

- **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

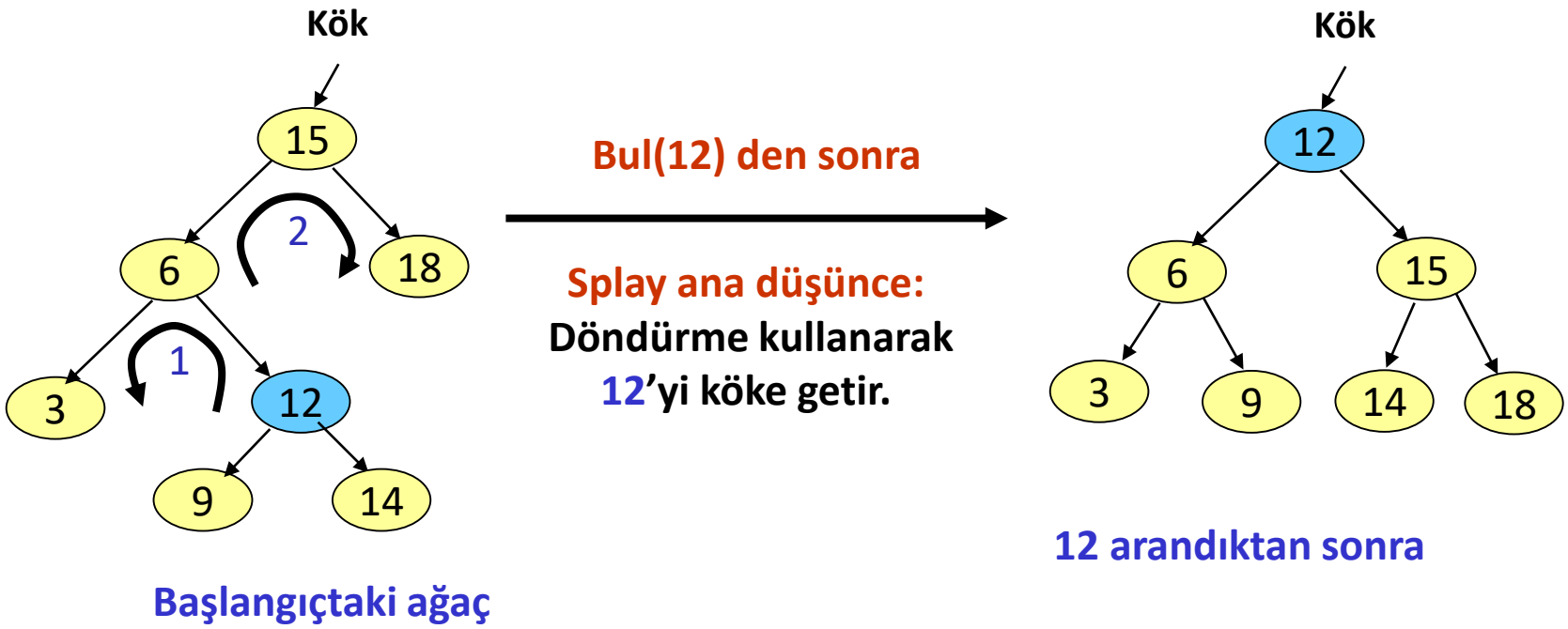
Splay Ağaçları

- 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.

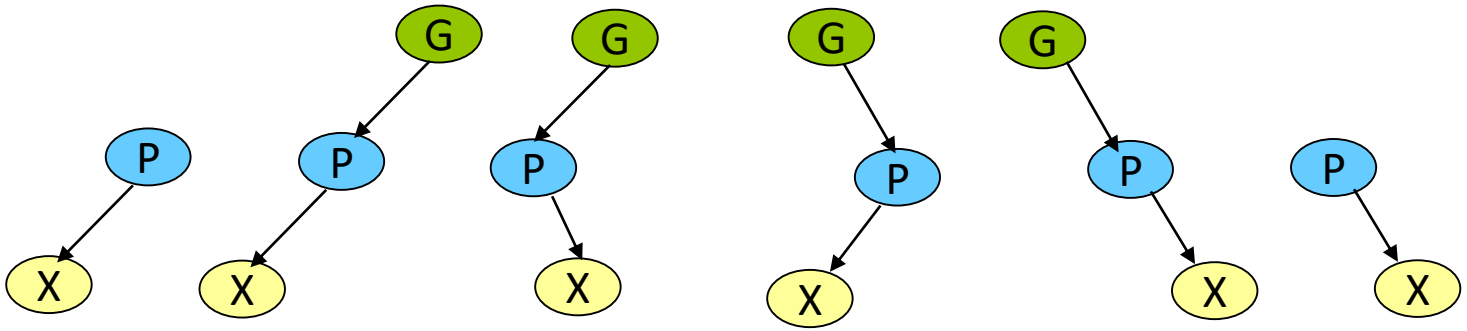
Örnek

- 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:

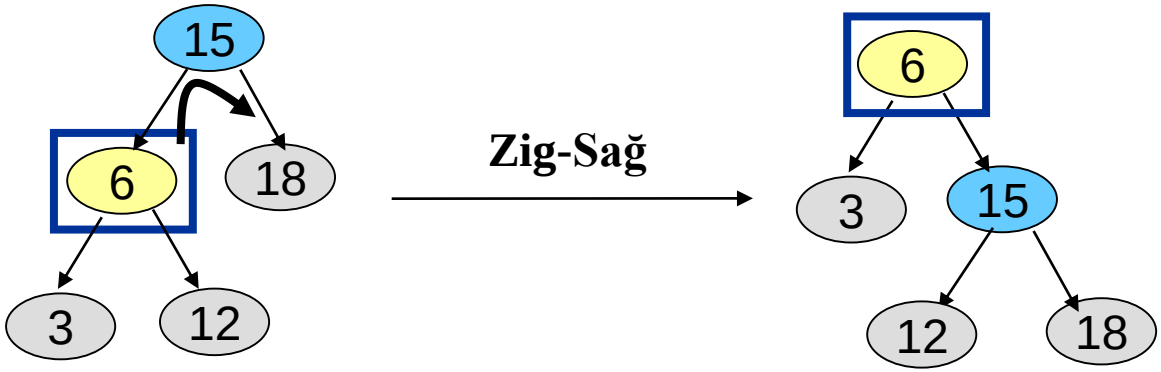


Splay Ağaç İşlemleri

- 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

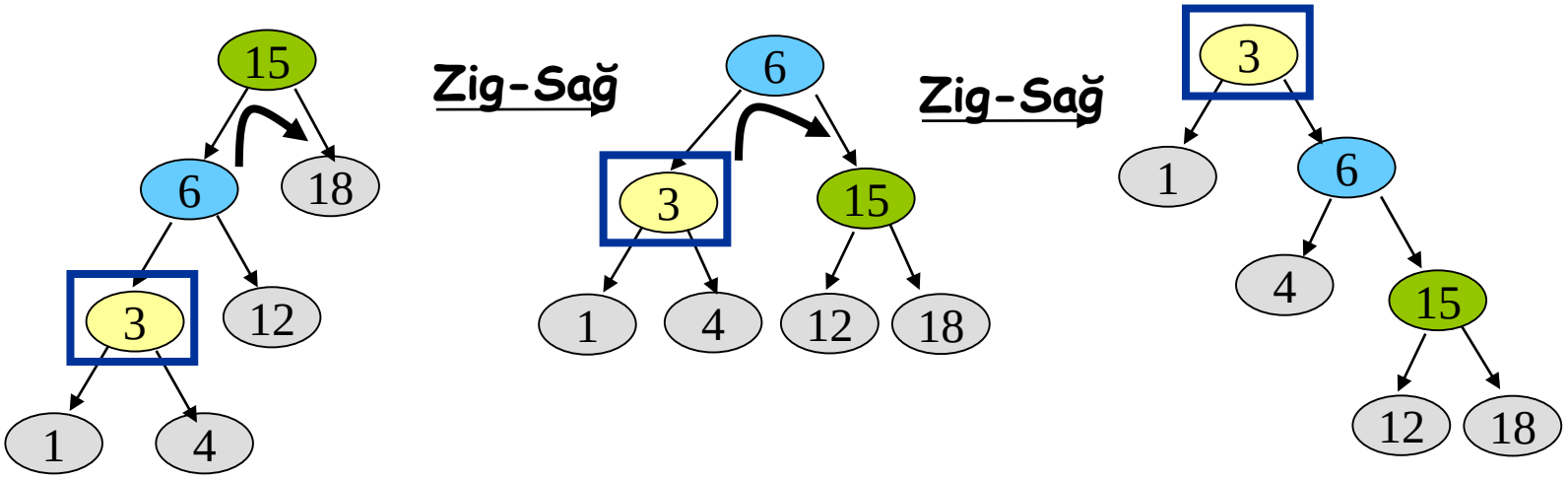
- "Zig" işlemi AVL ağacındaki gibi tek döndürme işlemidir.
- Örneğin erişilen elemanın 6 olduğu düşünülürse.



- "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

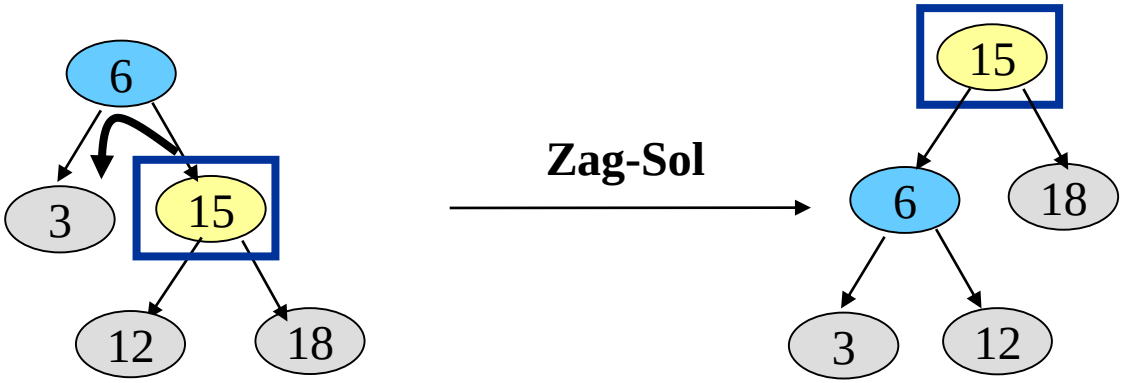
- "Zig-Zig" işlemi aynı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 3 olduğu düşünülürse.



- "zig-zig" işlemi ile birlikte 3 köke taşınmış oldu.
- Not: Aile - ata arasındaki döndürme önce yapılıyor.

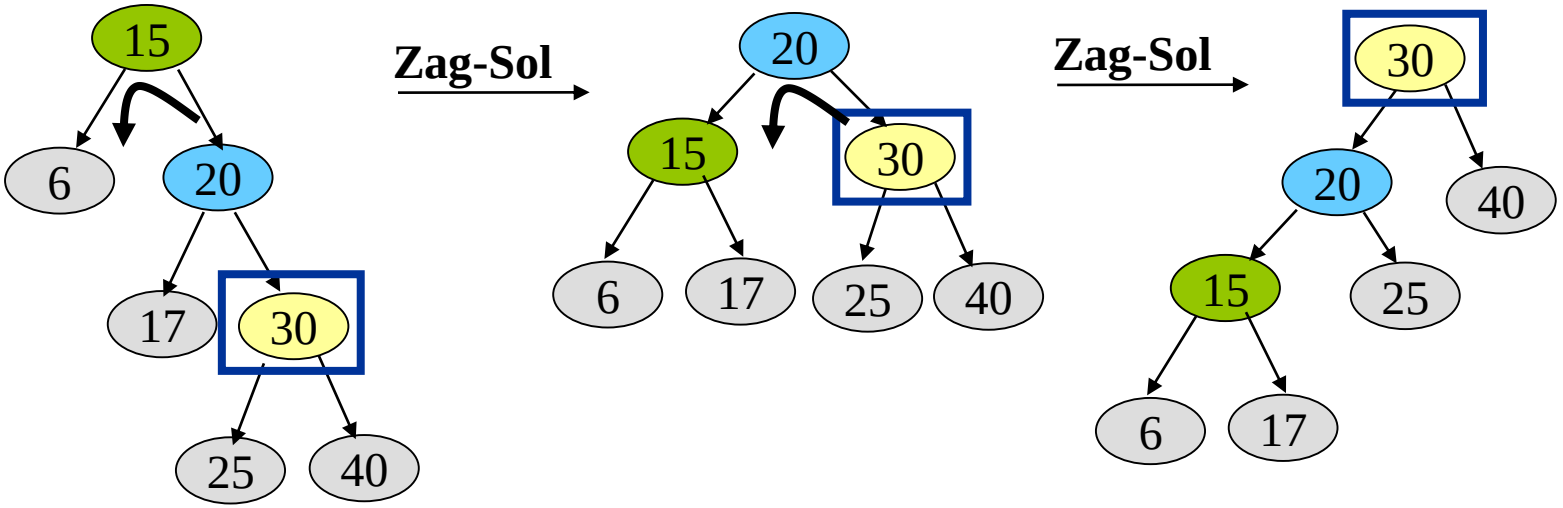
Splay Ağaçları: Zag İşlemi

- “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.



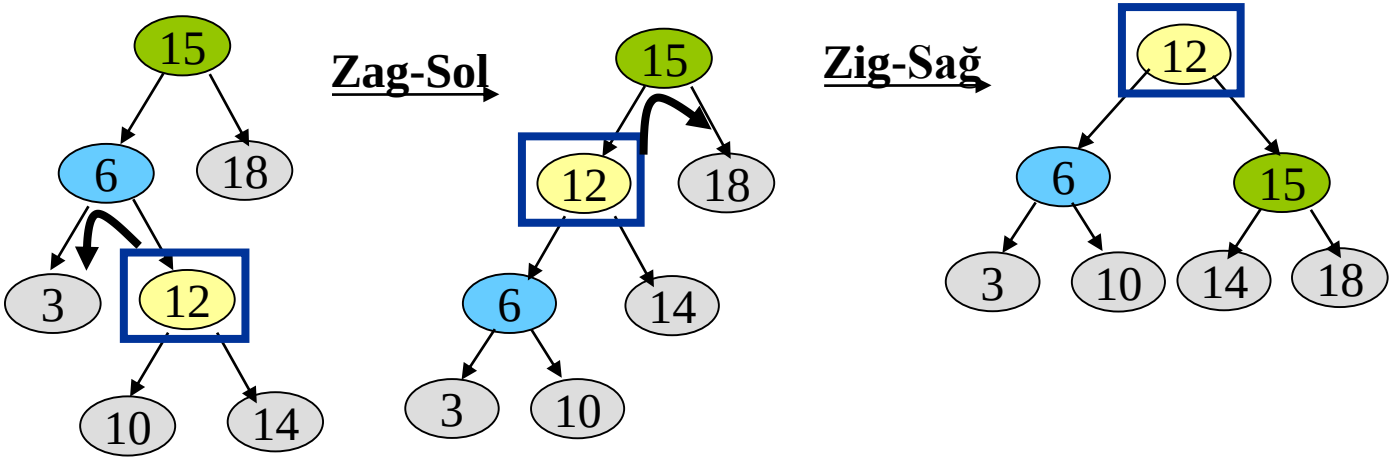
Splay Ağaçları: Zag-Zag İşlemi

- “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.



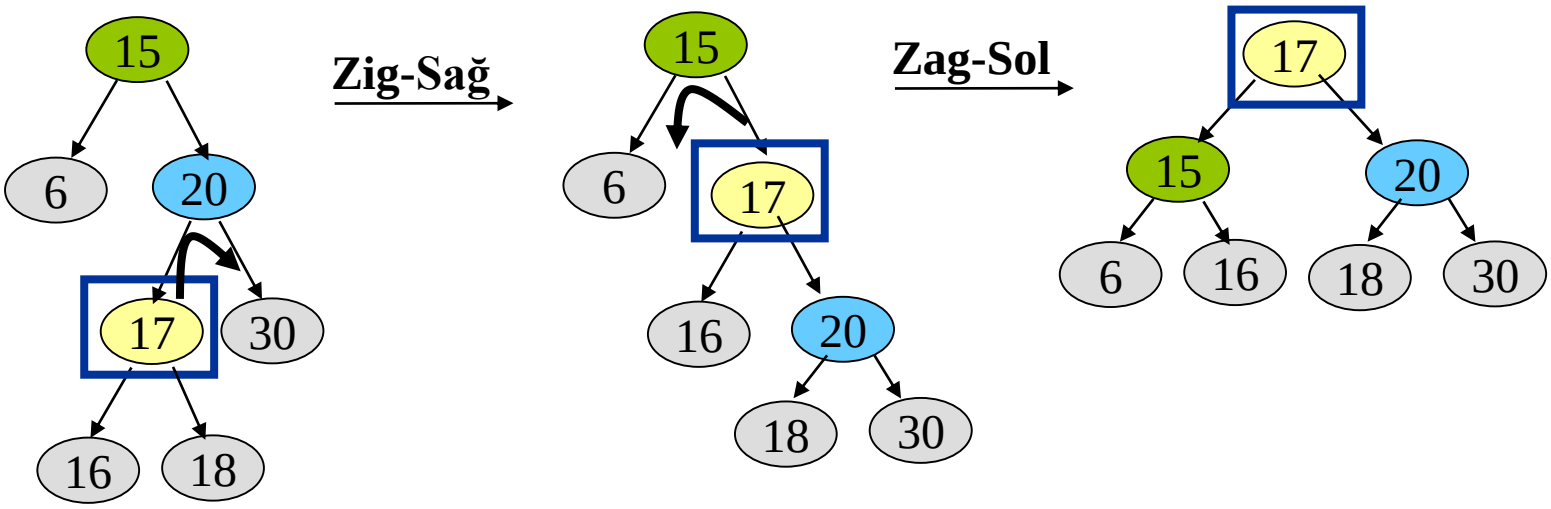
Splay Ağaçları: Zig-Zag durumu

- 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)

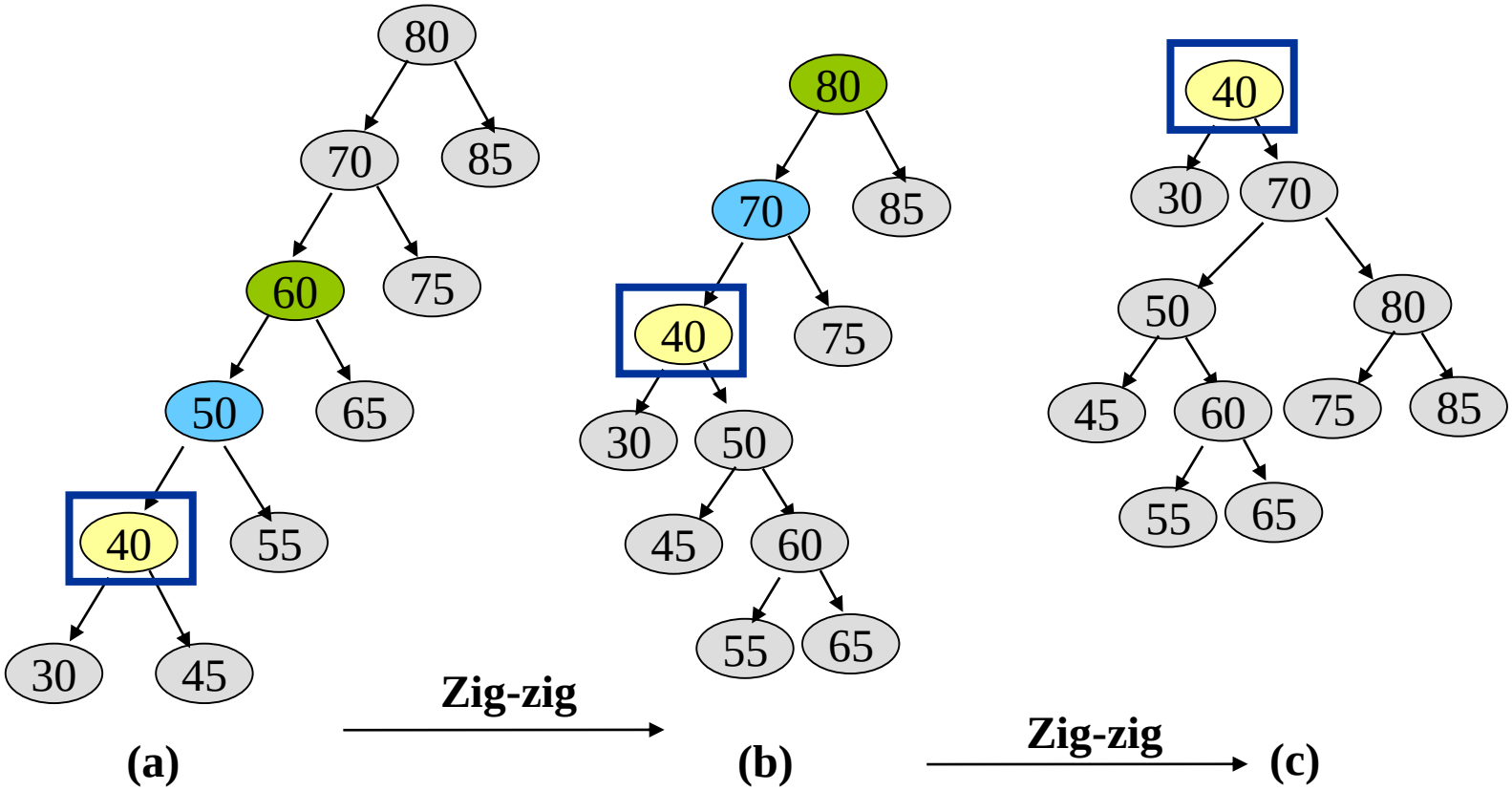


Splay Ağaçları: Zag-Zig durumu

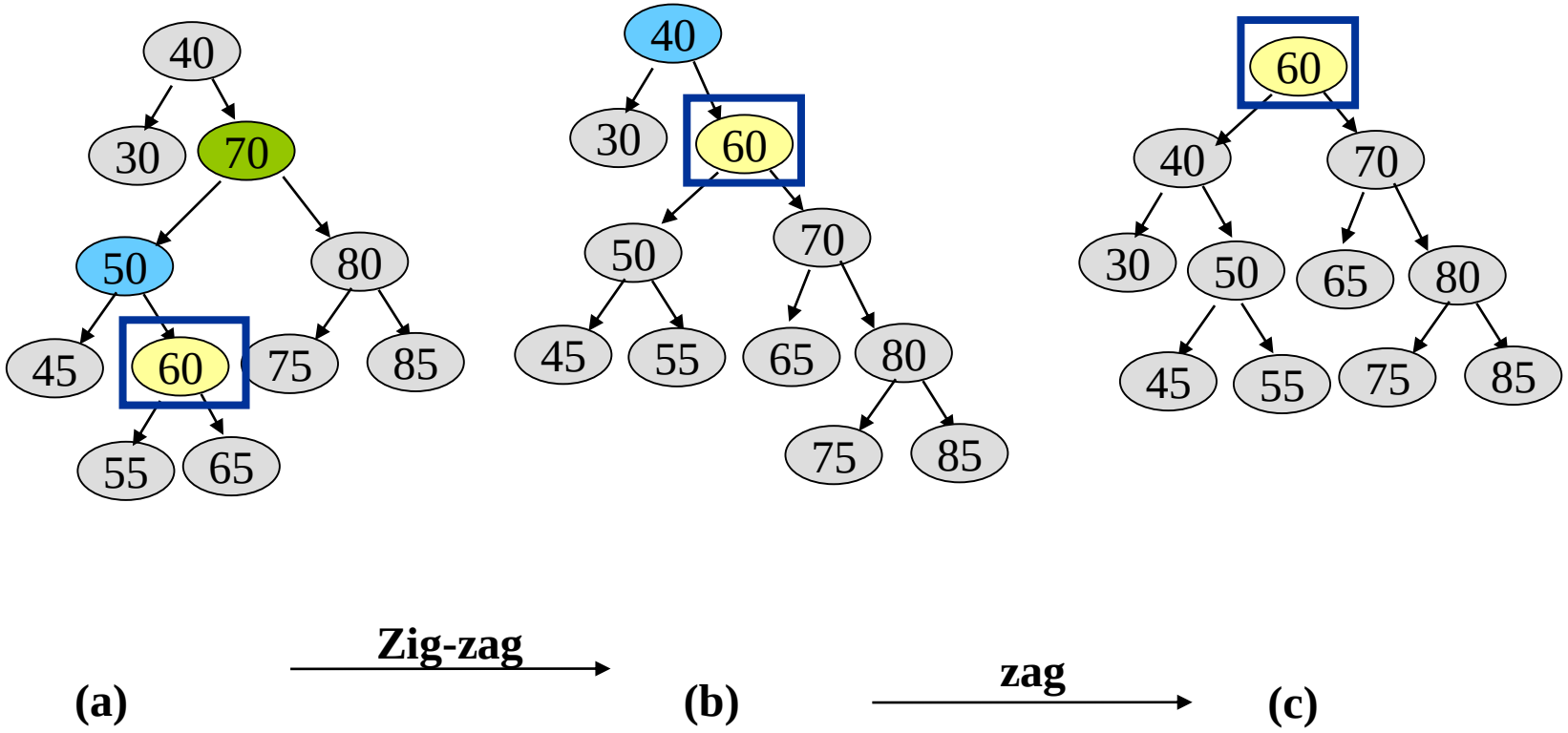
- Zag-Zig durumunda X, P nin sol çocuğu ve G'de atası olduğu durumdur.
- “Zag-Zig” durumu farklı türde 2 tane dönme işlemi içerir.
- Örneğin erişilen elemanın 17 olduğu düşünülürse.
- zag-zig” işlemi ile birlikte 17 köke taşınmış oldu.
- AVL ağacındaki RL dengesizliğini düzeltmek için kullanılan işlemler ile aynıdır.(önce sağ dönme, daha sonra sol dönme)



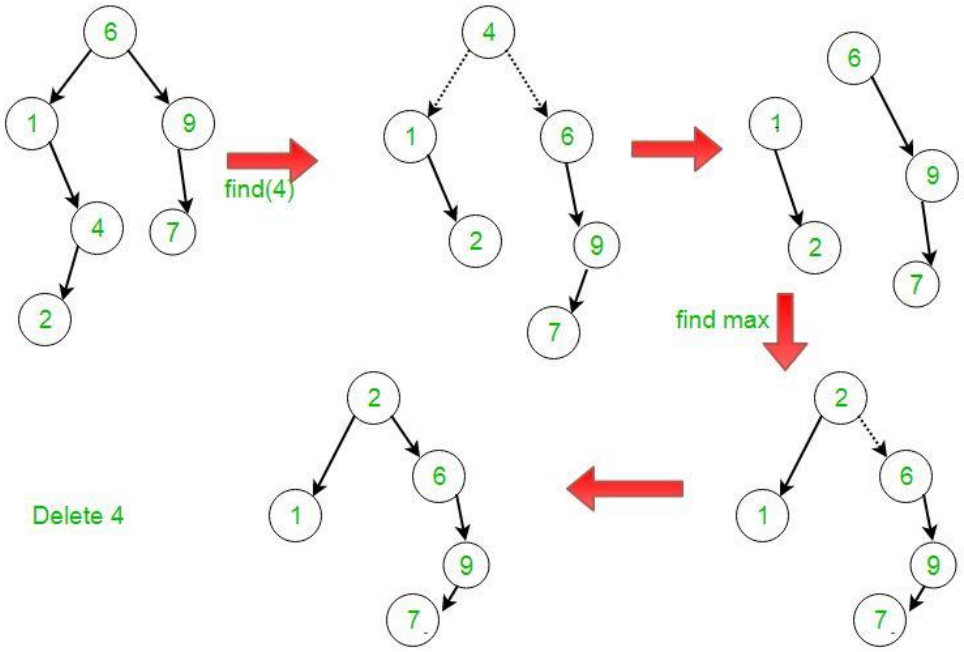
Splay Ağacı Örnek: 40'a Erişildiğinde



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;
}
```