

BLM267

Bölüm 10: Verimli İkili Ağaçlar
**C Kullanarak Veri Yapıları,
İkinci Baskı**

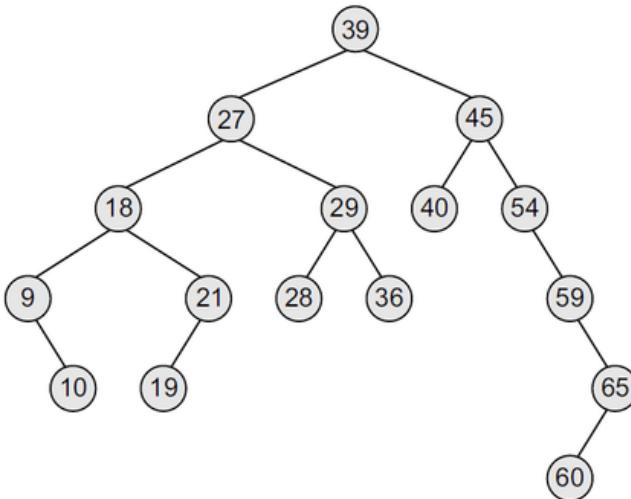
C Kullanarak Veri Yapıları, İkinci Baskı
Reema Thareja

- İkili Arama Ağaçları
- İkili Arama Ağaçları Üzerindeki İşlemler
- İş Parçacıklı İkili Ağaçlar
- AVL Ağaçları
- Kırmızı-Siyah Ağaçlar
- Yayılmış Ağaçlar

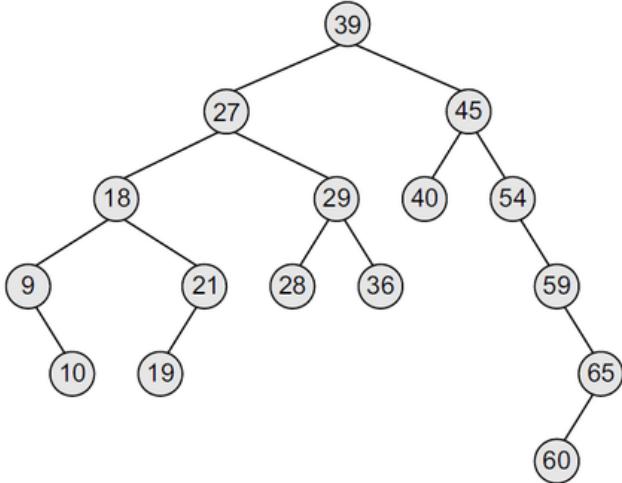
İkili Arama Ağaçları

- İkili ağaçları daha önceki bölümde ele almıştık.
- İkili arama ağacı, sıralı ikili ağaç olarak da bilinir, düğümlerin bir sıraya göre düzenlendiği ikili ağaçların bir çeşididir.
- **İkili arama ağacında, sol alt ağaçtaki tüm düğümlerin değeri kök düğümün değerinden küçüktür.**
- **Buna karşılık, sağ alt ağaçtaki tüm düğümlerin değeri kök düşüme eşit veya ondan büyütür.**
- Aynı kural ağacın her alt ağacı için geçerlidir.
- (İkili arama ağacının, uygulamasına bağlı olarak yinelenen değerler içerebileceğini veya içermeyebileceğini unutmayın.)

İkili Arama Ağaçları



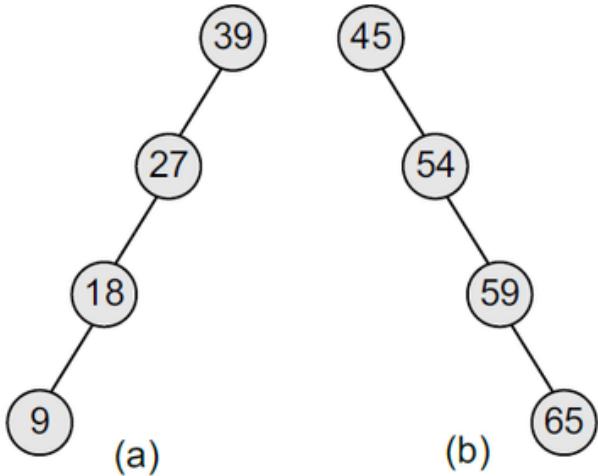
- Kök düğüm 39'dur. Kök düğümün sol alt ağıacı 9, 10, 18, 19, 21, 27, 28, 29 ve 36 düğümlerinden oluşur.
- Tüm bu düğümler kök düğümden daha küçük değerlere sahiptir.
- Kök düğümün sağ alt ağıacı 40, 45, 54, 59, 60 ve 65 düğümlerinden oluşur.
- Yinelemeli olarak, alt ağaçların her biri ikili arama ağıacı kısıtlamasına da uyar.
- Örneğin, kök düğümün sol alt ağıacında 27 köktür ve sol alt ağıacındaki tüm eleman (9, 10, 18, 19, 21) 27'den küçüktür, sağ alt ağıacındaki tüm düğümler (28, 29 ve 36) is kök düğümün değerinden büyütür.



- İkili arama ağacındaki düğümler sıralı olduğundan, ağaçtaki bir öğeyi aramak için gereken süre büyük ölçüde azalır.
- Bir öğeyi aradığımızda, tüm ağaçın dolaşmamız gereklidir. Her düğümde, hangi ağaçta arama yapacağımıza dair bir ipucu alırız.
- Örneğin, verilen ağaçta 29'u aramamız gerekiyorsa, yalnızca sol alt ağaç taramamam gerektiğini biliyoruz. Değer ağaçta mevcutsa, yalnızca sol alt ağaçta olacaktır, çünkü 29, 39'dan (kök düğümün değeri) küçüktür. Sol alt ağaçın 27 değerine sahip bir kök düğümü vardır. 29, 27'den büyük olduğundan, öğeyi bulacağımız sağ alt ağaç geçeceğiz.
- Bu nedenle, bir arama işleminin ortalama çalışma süresi $O(\log_2 n)$ olur, çünkü her adımda arama sürecinden alt ağaçın yarısını elemiş oluruz.
- Elemanları aramadaki verimliliği nedeniyle ikili arama ağaçları, kodun her zaman bir anahtar değerle indekslenen elemanları eklediği ve aradığı sözlük problemlerinde yaygın olarak kullanılır.

- İkili arama ağaçları aynı zamanda ekleme ve silme işlemlerini de hızlandırır.
- Yapıdaki veriler hızla değiştiğinde ağaç hız avantajına sahip olur.
- İkili arama ağaçları, özellikle sıralı doğrusal diziler ve bağlı listelerle karşılaştırıldığında verimli veri yapıları olarak kabul edilir.
- Sıralı bir dizide arama $O(\log_2 n)$ sürede yapılabilir, ancak ekleme ve çıkarma işlemleri oldukça maliyetlidir.
- Buna karşılık, bağlı listelerde eleman eklemek ve silmek daha kolaydır, ancak eleman arama işlemi $O(n)$ sürede yapılır.

İkili Arama Ağaçları



- Ancak en kötü durumda, ikili arama ağacının bir elemanı araması $O(n)$ zaman alacaktır.
- En kötü durum, ağacın Şekil'de gösterildiği gibi doğrusal bir düğüm zinciri olması durumunda ortaya çıkar.
- Özetlemek gerekirse, ikili arama ağaçları aşağıdaki özelliklere sahip bir ikili ağaçtır:
 - N düğümünün sol alt ağıacı, N'nin değerinden küçük değerleri içerir.
 - N düğümünün sağ alt ağıacı, N'nin değerinden büyük değerleri içerir.
 - Hem sol hem de sağ ikili ağaçlar bu özelliklerini sağlar ve dolayısıyla ikili arama ağaçlarıdır.

İkili Arama Ağaçları

Example 10.1 State whether the binary trees in Fig. 10.3 are binary search trees or not.

Solution

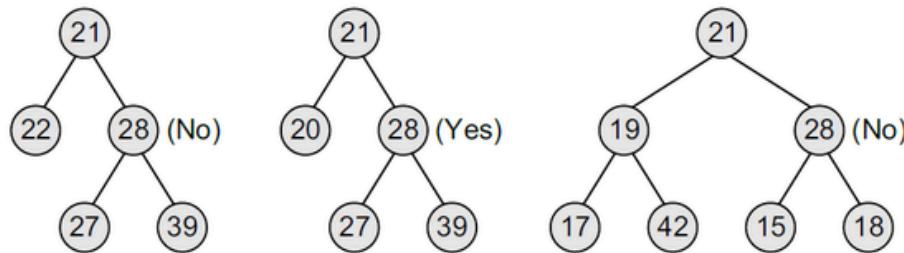


Figure 10.3 Binary trees

İkili Arama Ağaçları

Example 10.2 Create a binary search tree using the following data elements:

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81

Solution

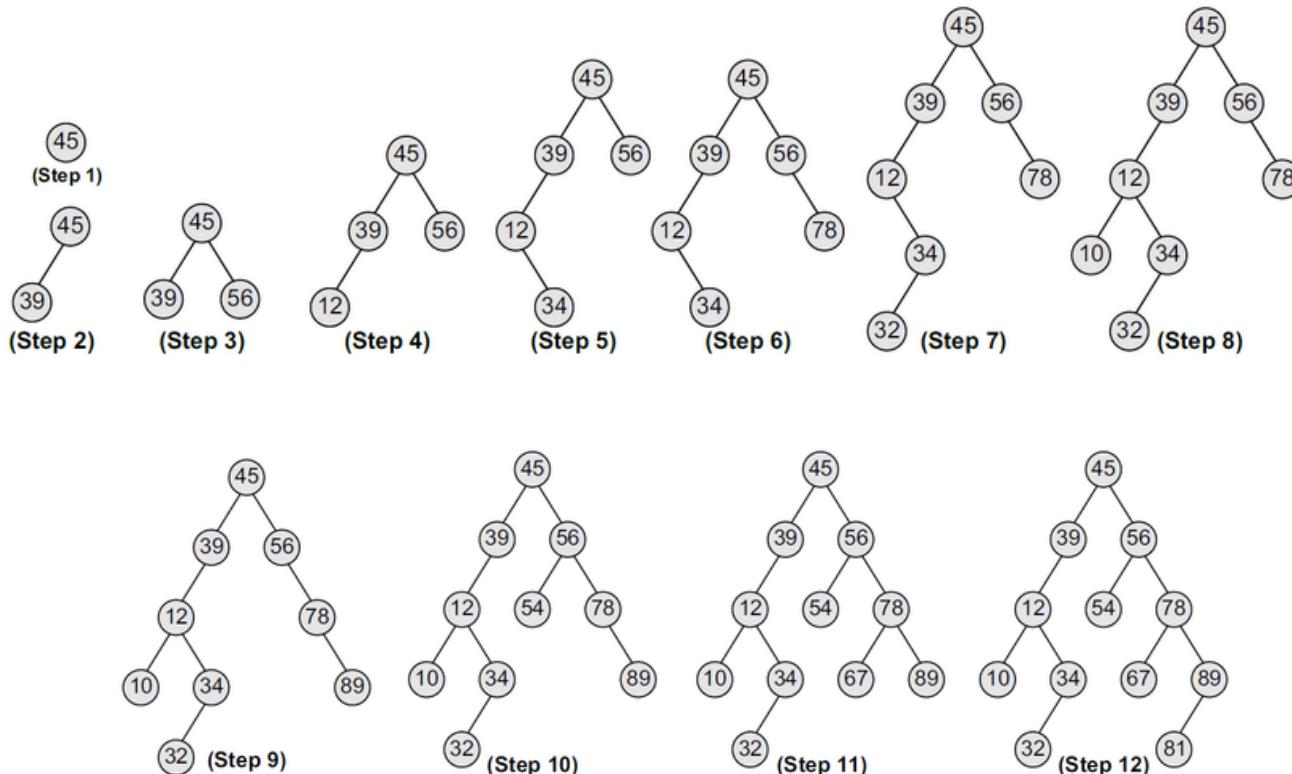


Figure 10.4 Binary search tree

Arama

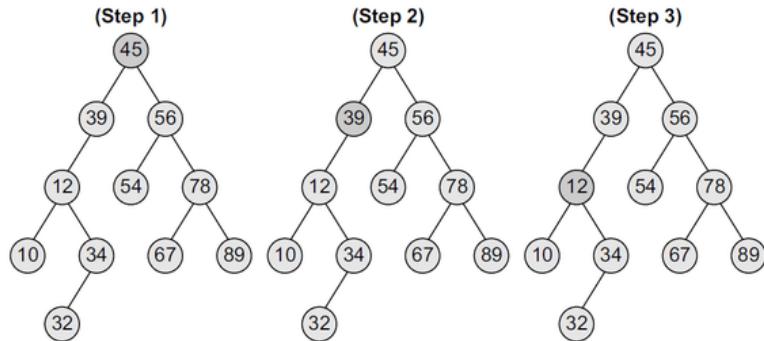


Figure 10.5 Searching a node with value 12 in the given binary search tree

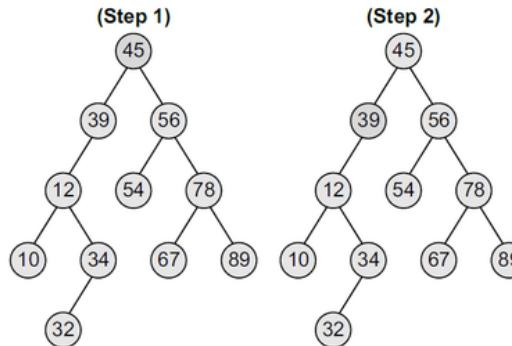


Figure 10.7 Searching a node with the value 40 in the given binary search tree

- Arama fonksiyonu, verilen değerin ağaçta bulunup bulunmadığını bulmak için kullanılır.
- Arama süreci kök düğümde başlar. Fonksiyon ilk önce ikili arama ağacının boş olup olmadığı kontrol eder. Boşsa, aradığımız değer ağaçta mevcut değildir.
- Böylece arama algoritması uygun bir mesaj görüntülenerek sonlanır.
- Ancak ağaçta düğümler varsa, arama fonksiyonu geçerli düğümün anahtar değerinin aranacak değere eşit olup olmadığını kontrol eder.
- Aksi takdirde aranacak değerin geçerli düğümün değerinden küçük olup olmadığını kontrol eder. Bu durumda sol alt düğümde yinelemeli olarak çağrılmalıdır.
- Değerin geçerli düğümün değerinden büyük olması durumunda, sağ alt düğümde yinelemeli olarak çağrılması gereklidir.

Arama

```
searchElement (TREE, VAL)
```

Step 1: IF TREE → DATA = VAL OR TREE = NULL

 Return TREE

 ELSE

 IF VAL < TREE → DATA

 Return searchElement(TREE → LEFT, VAL)

 ELSE

 Return searchElement(TREE → RIGHT, VAL)

 [END OF IF]

 [END OF IF]

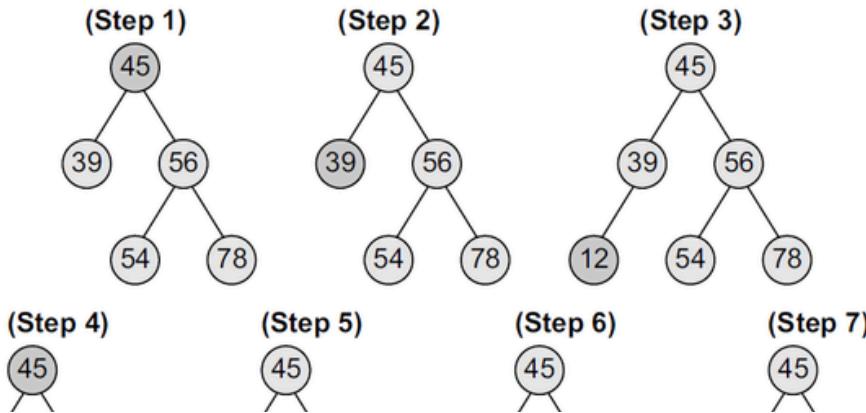
Step 2: END

Figure 10.8 Algorithm to search for a given value in a binary search tree

- Adım 1'de TREE'nin geçerli düğümünde depolanan değerin VAL'e eş olup olmadığını veya geçerli düğümün NULL olup olmadığını kontrol ediyoruz, ardından TREE'nin geçerli düğümünü döndürüyoruz.
- Aksi takdirde, geçerli düğümde depolanan değer VAL'den küçükse, algoritma sağ alt ağacında yinelemeli olarak çağrırlar, aksi takdirde algoritma sol alt ağacında çağrırlar.

İkili Arama Ağacına Yeni Bir Düğüm Ekleme

12



- Insert fonksiyonu, ikili arama ağacında doğru konuma verilen değere sahip yeni bir düğüm eklemek için kullanılır.
- Düğümün doğru konuma eklenmesi, yeni düğümün ikili arama ağacının özelliklerini ihlal etmemesi gerektiği anlamına gelir.
- Ekleme işlevi için başlangıç kodu arama işlevine benzerdir. Bunun nedeni, önce eklemenin yapılması gereken doğru konumu bulmamız ve ardından düğümü o konuma eklememizdir.
- Ekleme işlevi ağacın yapısını değiştirir. Bu nedenle, ekleme işlevi yinelemel olarak çağrıldığında, işlev yeni ağaç işaretçisini döndürmelidir.

İkili Arama Ağacına Yeni Bir Düğüm Ekleme

13

Insert (TREE, VAL)

```
Step 1: IF TREE = NULL
        Allocate memory for TREE
        SET TREE -> DATA = VAL
        SET TREE -> LEFT = TREE -> RIGHT = NULL
    ELSE
        IF VAL < TREE -> DATA
            Insert(TREE -> LEFT, VAL)
        ELSE
            Insert(TREE -> RIGHT, VAL)
    [END OF IF]
```

- Algoritmanın 1. Adımında, insert fonksiyonu TREE'nin geçerli düğümünün NULL olup olmadığını kontrol eder. Eğer NULL ise, algoritma sadece düğümü ekler, aksi takdirde geçerli düğümün değerine bakar ve sonra sol veya sağ alt ağaçta tekrar eder.
- Mevcut düğümün değeri yeni düğümün değerinden küçükse sağ alt ağaç, değilse sol alt ağaç geçilir.
- Ekleme işlevi, bir yaprak düğümüne ulaşana kadar ikili ağacın seviyelerinde aşağı doğru hareketmeye devam eder. Yeni düğüm, ikili arama ağaçlarının kurallarına uyularak eklenir.
- Yani eğer yeni düğümün değeri ana düğümün değerinden büyükse yeni düğüm sağ alt ağaca, büyük değilse sol alt ağaca eklenir.
- Ekleme fonksiyonu en kötü durumda ağacın yüksekliğine orantılı bir zaman gerektirir.
- Ortalama durumda yürütülmesi $O(\log n)$ zaman alırken, en kötü durumda $O(n)$ zaman alır.

Silme

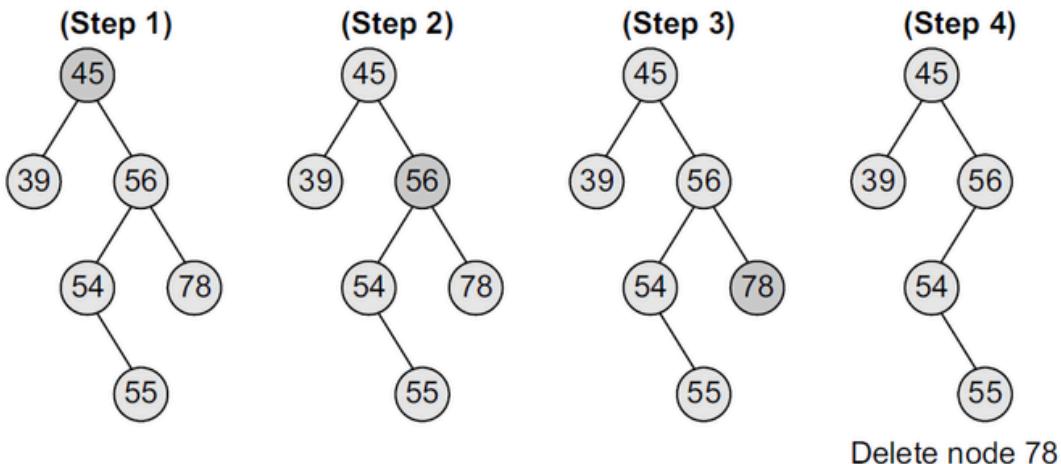


Figure 10.11 Deleting node 78 from the given binary search tree

- **Durum 1: Çocuğu Olmayan Bir Düğümü Silme**
- Şekilde verilen ikili arama ağacına bakın.
- Eğer 78 nolu nodu silmemiz gerekirse, bu nodu herhangi bir sorun yaşamadan kolayca kaldırabiliriz.
- Bu, silmenin en basit halidir.

Silme

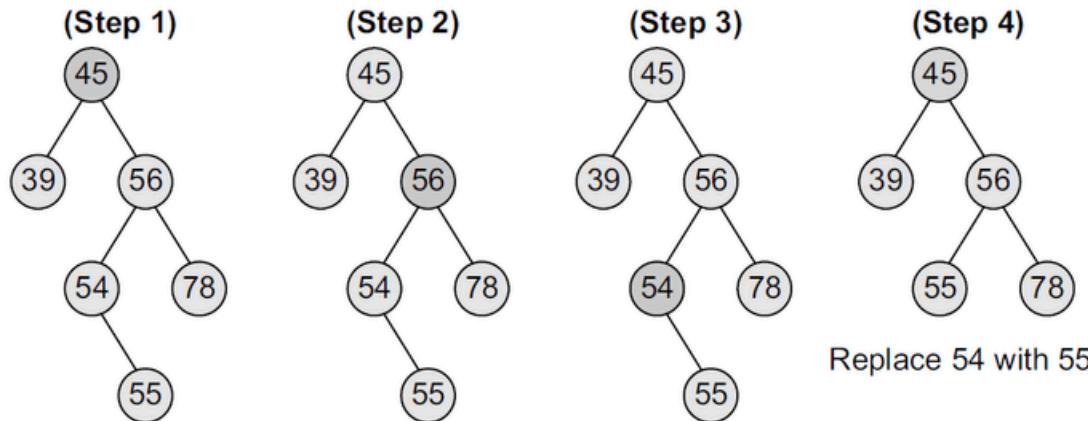


Figure 10.12 Deleting node 54 from the given binary search tree

Durum 2: Bir Çocuklu Bir Düğümü Silme

- Bu durumu ele almak için, düğümün çocuğu, düğümün ebeveyninin çocuğu olarak ayarlanır. Başka bir deyişle, düğümü çocuğuyla değiştirin.
- Şimdi, eğer düğüm ebeveyninin sol çocuğu ise, düğümün çocuğu düğümün ebeveyninin sol çocuğu olur.
- Buna karşılık, eğer düğüm ebeveyninin sağ çocuğu ise, düğümün çocuğu düğümün ebeveyninin sağ çocuğu olur.
- Şekilde gösterilen ikili arama ağacına bakın ve 54 numaralı düğümün silinmesinin nasıl işlendiğini görün.

Silme

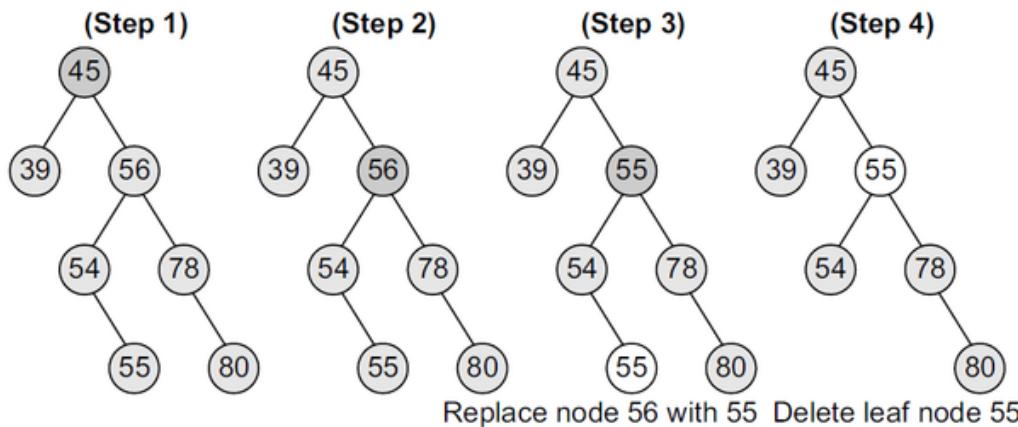


Figure 10.13 Deleting node 56 from the given binary search tree

- **Durum 3: İki Çocuğu Olan Bir Düğümü Silme**
- Bu durumu ele almak için, düğümün değerini sıralı öncülüyle (sol alt ağaçtaki en büyük değer) veya sıralı ardılııyla (sağ alt ağaçtaki en küçük değer) değiştirebilir.
- Sıralı öncül veya halef daha sonra yukarıdaki durumlardan herhangi biri kullanılarak silinebilir.
- Şekil 10.13'te verilen ikili arama ağacına bakın ve değeri 56 olan düğümün silinmesinin nasıl işlendiğini görün.

Silme

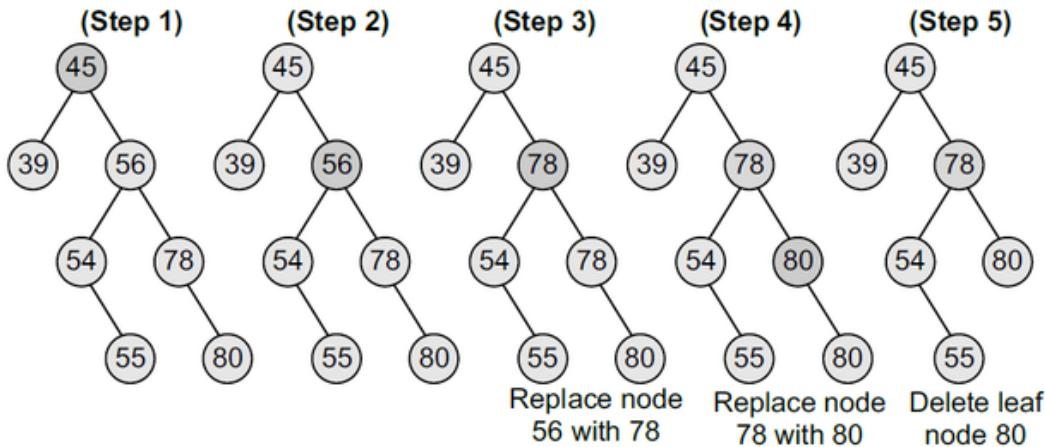


Figure 10.14 Deleting node 56 from the given binary search tree

- Durum 3: İki Çocuğu Olan Bir Düğümü Silme**
- Bu durumu ele almak için, düğümün değerini sıralı öncülüyle (sol alt ağaçta en büyük değer) veya sıralı ardılıyla (sağ alt ağaçtaki en küçük değer) değiştirin.
- Sıralı öncül veya halef daha sonra yukarıdaki durumlardan herhangi biri kullanılarak silinebilir.
- Şekil 10.13'te verilen ikili arama ağacına bakın ve değeri 56 olan düğümün silinmesinin nasıl işlendiğini görün.
- Bu silme işlemi, Şekil 10.14'te gösterildiği gibi, düğüm 56'nın kendi ardılıyla değiştirilmesiyle de gerçekleştirilebilir.**

Silme

Delete (TREE, VAL)

```
Step 1: IF TREE = NULL
        Write "VAL not found in the tree"
    ELSE IF VAL < TREE->DATA
        Delete(TREE->LEFT, VAL)
    ELSE IF VAL > TREE->DATA
        Delete(TREE->RIGHT, VAL)
    ELSE IF TREE->LEFT AND TREE->RIGHT
        SET TEMP = findLargestNode(TREE->LEFT)
        SET TREE->DATA = TEMP->DATA
        Delete(TREE->LEFT, TEMP->DATA)
    ELSE
```

- Algoritmanın 1. Adımında, öncelikle TREE=NULL olup olmadığını kontrol ediyoruz, çünkü eğer doğruysa, silinecek düğüm ağaçta mevcut değildir.
- Ancak durum böyle değilse, silinecek değerin geçerli düğümün verilerinden az olup olmadığını kontrol ederiz. Değer azsa, algoritmayı düğümün sol alt ağacında yinelemeli olarak çağırırız, aksi takdirde algoritma düğümün sağ alt ağacında yinelemeli olarak çağrılır.
- Değeri VAL'e eşit olan düğümü bulduysak, bunun hangi silme durumu olduğunu kontrol ederiz. Silinecek düğümün hem sol hem de sağ çocukları varsa, findLargestNode(TREE -> LEFT) çağrıarak düğümün sıralı öncülünü buluruz ve geçerli düğümün değerini sıralı öncülünün değeriyle değiştiririz.

Silme

Delete (TREE, VAL)

```
Step 1: IF TREE = NULL
        Write "VAL not found in the tree"
    ELSE IF VAL < TREE->DATA
        Delete(TREE->LEFT, VAL)
    ELSE IF VAL > TREE->DATA
        Delete(TREE->RIGHT, VAL)
    ELSE IF TREE->LEFT AND TREE->RIGHT
        SET TEMP = findLargestNode(TREE->LEFT)
        SET TREE->DATA = TEMP->DATA
        Delete(TREE->LEFT, TEMP->DATA)
    ELSE
```

- Sonra, sıralı öncülün başlangıç düğümünü silmek için Delete(TREE -> LEFT, TEMP -> DATA) çağrılarıız. Böylece, silmenin 3. durumunu silmenin 1. veya 2. durumuna indirgeriz.
- Silinecek düğümün hiçbir çocuğu yoksa, o zaman düğümü basitçe NULL olarak ayarlarız. Son olarak, silinecek düğümün sol veya sağ çocuğu varsa ancak ikisi birden yoksa, geçerli düğüm kendi çocuğuyla değiştirilir ve ilk çocuk düğümü ağaçtan silinir.
- Silme fonksiyonu en kötü durumda ağacın yüksekliğine orantılı bir zaman gerektirir.
- Ortalama durumda yürütülmesi $O(\log n)$ zaman alırken, en kötü durumda $W(n)$ zaman alır.

İkili Arama Ağacının Yüksekliğini Belirleme

20

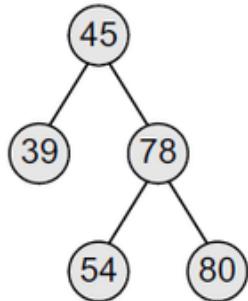


Figure 10.16 Binary search tree with height = 3

- İkili arama ağacının yüksekliğini belirlemek için sol alt ağacın ve sağ alt ağacın yüksekliğini hesaplarız.
- Hangisi daha büyüğse ona 1 eklenir. Örneğin, sol alt ağacın yüksekliği sağ alt ağacın yüksekliğinden büyük sol alt ağaca 1 eklenir, aksi takdirde sağ alt ağaca 1 eklenir.
- Şekil 10.16'ya bakın. Sağ alt ağacın yüksekliği sol alt ağacın yüksekliğinden büyük olduğundan, ağacın yüksekliği = yükseklik (sağ alt ağaç) + 1 = 2 + 1 = 3.
- Şekil 10.17, ikili arama ağacının yüksekliğini belirleyen yinelemeli bir algoritmayı gösterir. Algoritmanın 1. Adımında, önce TREE'nin geçerli düğümünün NULL olup olmadığını kontrol ederiz. Koşul doğruysa, çağrıran koda 0 döndürülür.
- Aksi takdirde, her düğüm için algoritmayı yinelemeli olarak çağrıarak sol alt ağacının ve sağ alt ağacının yüksekliğini hesaplarız.
- Ağacın o düğümdeki yüksekliği, sol alt ağacın yüksekliğine veya sağ alt ağacın yüksekliğine (hangisi büyüğü 1 eklenerek bulunur.

Height (TREE)

```
Step 1: IF TREE = NULL  
        Return 0  
    ELSE  
        SET LeftHeight = Height(TREE → LEFT)  
        SET RightHeight = Height(TREE → RIGHT)  
        IF LeftHeight > RightHeight  
            Return LeftHeight + 1  
        ELSE  
            Return RightHeight + 1  
    [END OF IF]  
[END OF IF]  
Step 2: END
```

Figure 10.17 Algorithm to determine the height of a binary search tree

Düğüm Sayısının Belirlenmesi

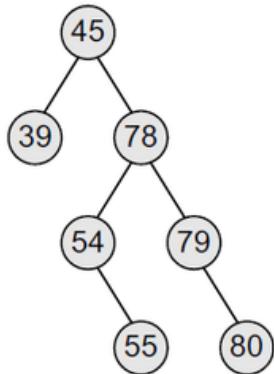


Figure 10.18 Binary search tree

totalNodes(TREE)

```

Step 1: IF TREE = NULL
        Return 0
    ELSE
        Return totalNodes(TREE -> LEFT)
            + totalNodes(TREE -> RIGHT) + 1
    [END OF IF]
Step 2: END
  
```

Figure 10.19 Algorithm to calculate the number of nodes in a binary search tree

- İkili arama ağacındaki düğüm sayısını belirlemek, ağacın yüksekliğini belirlemeye benzer.
- Ağaçtaki toplam eleman/düğüm sayısını hesaplamak için sol alt ağaçtaki ve sağ alt ağaçtaki düğüm sayıları sayıyoruz.
- Düğüm sayısı = toplamDüğümler(sol alt ağaç) + toplamDüğümler(sağ alt ağaç) + 1
- Şekil 10.18'de verilen ağacı ele alalım. Ağaçtaki toplam düğüm sayısı şu şekilde hesaplanabilir:
 - Sol alt ağaçın toplam düğüm sayısı = 1
 - Sol alt ağaçın toplam düğüm sayısı = 5
 - Ağacın toplam düğüm sayısı = $(1 + 5) + 1$
 - Ağacın toplam düğüm sayısı = 7
- Şekil 10.19, ikili arama ağacındaki düğüm sayısını hesaplamak için yinelemeli bir algoritma gösterir. Her düğüm için algoritmayı sol alt ağaçında ve sağ alt ağaçında yinelemeli olarak çağırırız. Belirli bir düğümdeki toplam düğüm sayısı daha sonra sol ve sağ alt ağaçındaki düğüm sayısına 1 eklenecek döndürülür. Ancak eğer boşsa, yani TREE = NULL ise, düğüm sayısı sıfır olacaktır.

Dahili Düğüm Sayısının Belirlenmesi

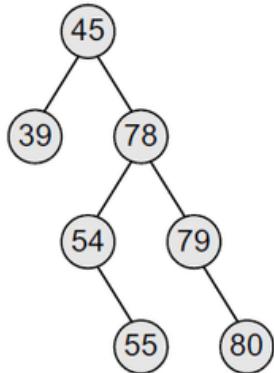


Figure 10.18 Binary search tree

totalInternalNodes(TREE)

```

Step 1: IF TREE = NULL
        Return 0
    [END OF IF]
    IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
        Return 0
    ELSE
        Return totalInternalNodes(TREE -> LEFT) +
              totalInternalNodes(TREE -> RIGHT) + 1
    [END OF IF]
Step 2: END
  
```

- Toplam iç düğüm veya yaprak olmayan düğüm sayısını hesaplamak için sol alt ağaçtaki ve sağ alt ağaçtaki iç düğüm sayılarını sayıyoruz ve buna 1 ekliyoruz (kök düğüm için 1 ekleniyor).

Dahili düğüm sayısı = toplam Dahili Düğümler (sol alt ağaç) + toplam Dahili Düğümler (sağ alt ağaç) + 1

- Şekil 10.18'de verilen ağaç ele alalım. Ağaçtaki toplam iç düğüm sayısı
- şu şekilde hesaplanabilir:
 - Sol alt ağaçın toplam iç düğümleri = 0
 - Sağ alt ağaçın toplam iç düğümleri = 3
 - Ağacın toplam iç düğümleri = $(0 + 3) + 1 = 4$
- Şekil 10.20, ikili arama ağacındaki toplam iç düğüm sayısını hesaplamak için yinelemeli bir algoritma göstermektedir. Bu algoritma, iç düğüm için, algoritmayı sol alt ağacında ve sağ alt ağacında yinelemeli olarak çağırırız. Daha sonra, belirli bir iç düğümdeki toplam iç düğüm sayısını, sol ve sağ alt ağacındaki iç düğümler eklenerek döndürür.
- Ancak, ağaç boşsa, yani TREE = NULL ise, o zaman iç düğümlerin sayısı sıfır olacaktır. Ayrıca ağaçta yalnızca bir iç düğüm varsa, o zaman iç düğümlerin sayısı sıfır olacaktır.

Harici Düğümlerin Sayısını Belirleme

23

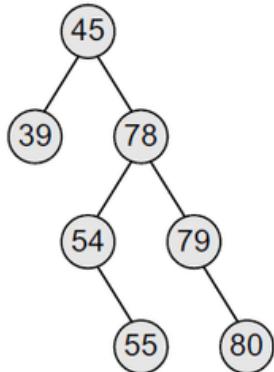


Figure 10.18 Binary search tree

- Toplam dış düğüm veya yaprak düğüm sayısını hesaplamak için sol alt ağaçtaki ve sağ alt ağaçtaki dış düğüm sayılarını toplam
- Ancak ağaç boşsa, yani TREE = NULL ise, o zaman harici düğümlerin sayısı sıfır olacaktır. Ancak ağaçta yalnızca bir düğüm varsa o zaman harici düğümlerin sayısı bir olacaktır.
- Dış düğüm sayısı = totalExternalNodes(sol alt ağaç) + totalExternalNodes (sağ alt ağaç)
- Şekil 10.18'de verilen ağacı ele alalım. Verilen ağaçtaki toplam dış düğüm sayısı şu şekilde hesaplanabilir:

Sol alt ağaçın toplam dış düğümleri = 1

Sol alt ağaçın toplam dış düğümleri = 2

Ağacın toplam dış düğümleri = 1 + 2 = 3

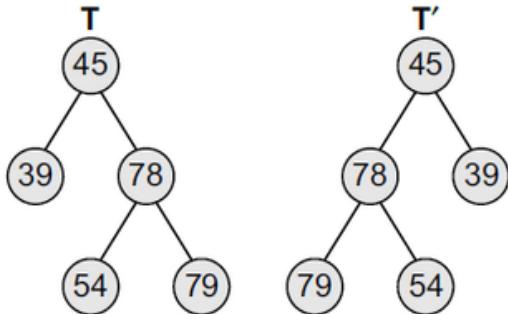
- Şekil 10.21, ikili arama ağacındaki toplam harici düğüm sayısını hesaplamak için yinelemeli bir algoritmayı göstermektedir.
- Her düğüm için algoritmayı sol alt ağacında ve sağ alt ağacında yinelemeli olarak çağırırız. Belirli bir düğümdeki toplam dış düğüm sayısı daha sonra sol ve sağ alt ağacındaki dış düğümleri ekleyerek döndürülür.
- Ancak ağaç boşsa, yani TREE = NULL ise, o zaman harici düğüm sayısı sıfır olacaktır. Ayrıca ağaçta yalnızca bir düğüm varsa, o zaman yalnızca bir harici düğüm (yani kök düğüm) olacaktır.

totalExternalNodes(TREE)

```
Step 1: IF TREE = NULL  
        Return 0  
    ELSE IF TREE->LEFT = NULL AND TREE->RIGHT = NULL  
        Return 1  
    ELSE  
        Return totalExternalNodes(TREE->LEFT) +  
              totalExternalNodes(TREE->RIGHT)  
    [END OF IF]  
Step 2: END
```

Figure 10.21 Algorithm to calculate the total number of external nodes in a binary search tree

Görüntüsünü Bulma

**MirrorImage(TREE)**

```

Step 1: IF TREE != NULL
    MirrorImage(TREE → LEFT)
    MirrorImage(TREE → RIGHT)
    SET TEMP = TREE → LEFT
    SET TREE → LEFT = TREE → RIGHT
    SET TREE → RIGHT = TEMP
    [END OF IF]
Step 2: END
  
```

Figure 10.23 Algorithm to obtain the mirror image
mirror image T' of a binary search tree

- İkili arama ağacının ayna görüntüsü, ağacın her düğümünde sol alt sağ alt ağaçla değiştirilmesiyle elde edilir.
- Örneğin, bir T ağacı verildiğinde, T 'nin ayna görüntüsü T' olarak elde edilebilir. Şekilde verilen T ağacını ele alalım.
- Şekil 10.23, ikili arama ağacının ayna görüntüsünü elde etmek için yinelemeli bir algoritmayı göstermektedir.
- Algoritmada, eğer $\text{TREE} \neq \text{NULL}$ ise, yani ağaçtaki geçerli düğüm bir veya daha fazla düğüme sahipse, algoritma ağacın her düğümünde sağ alt ağaçlarındaki düğümleri değiştirmek için yinelemeli olarak çağrırlar.

İkili Arama Ağacını Silme

```
deleteTree(TREE)
Step 1: IF TREE != NULL
    deleteTree (TREE -> LEFT)
    deleteTree (TREE -> RIGHT)
    Free (TREE)
[END OF IF]
Step 2: END
```

Figure 10.24 Alogrithm to delete a binary search tree

- Bir ikili arama ağacının tamamını bellekten silmek/kaldırmak için önce sol alt ağaçtaki elemanları/düğümleri, sonra da sağ alt ağaçtaki düğümleri sileriz.
- Şekil 10.24'te gösterilen algoritma ikili arama ağacını kaldırmak için yinelemeli bir prosedür sunar.

İkili Arama Ağacındaki En Küçük Düğümü Bulma

26

```
findSmallestElement(TREE)
Step 1: IF TREE = NULL OR TREE->LEFT = NULL
        Return TREE
    ELSE
        Return findSmallestElement(TREE->LEFT)
    [END OF IF]
Step 2: END
```

Figure 10.25 Algorithm to find the smallest node in a binary search tree

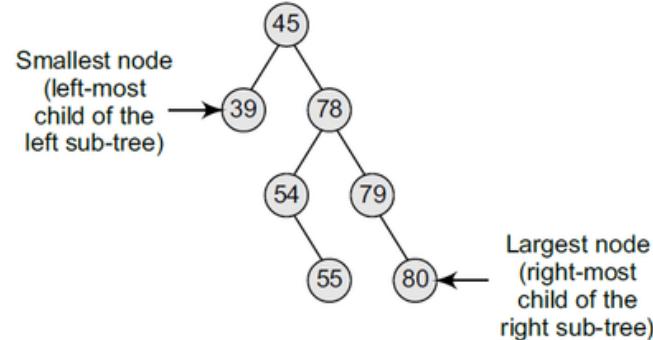


Figure 10.27 Binary search tree

- İkili arama ağacının en temel özelliği, daha küçük değerin sol alt ağaçta yer alacağıdır.
- Eğer sol alt ağaç NULL ise, kök düğümün değeri sağ alt ağaçtan düğümlere kıyasla en küçük olacaktır.
- Yani en küçük değere sahip düğümü bulmak için sol alt ağacın solundaki düğümün değerini buluyoruz.
- İkili arama ağacındaki en küçük düğümü bulmak için kullanılan yinelemeli algoritma Şekil 10.25'te gösterilmiştir.

İkili Arama Ağacındaki En Büyük Düğümü Bulma

27

```
findLargestElement(TREE)
Step 1: IF TREE = NULL OR TREE -> RIGHT = NULL
        Return TREE
    ELSE
        Return findLargestElement(TREE -> RIGHT)
    [END OF IF]
Step 2: END
```

Figure 10.26 Algorithm to find the largest node in a binary search tree

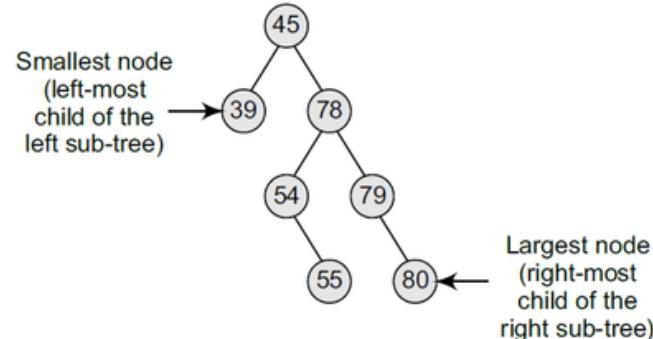


Figure 10.27 Binary search tree

- En büyük değere sahip düğümü bulmak için sağ alt ağacın sağdaki düğümünün değerini buluruz.
- Ancak sağ alt ağaç boş ise o zaman kök düğüm ağacındaki en büyük değer olacaktır.
- İkili arama ağacındaki en büyük düğümü bulmak için kullanılan yinelemeli algoritma Şekil 10.26'da gösterilmiştir.

İş Parçacıklı İkili Ağaçlar

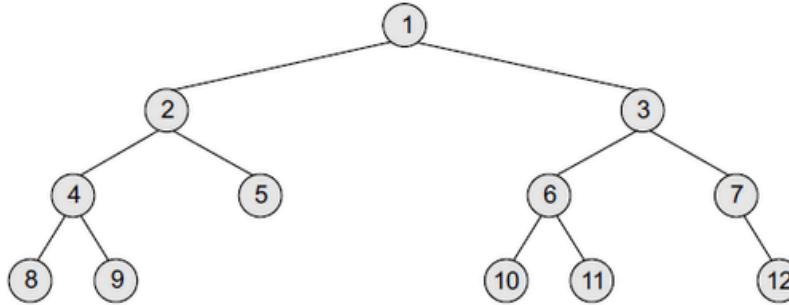


Figure 10.29 (a) Binary tree without threading

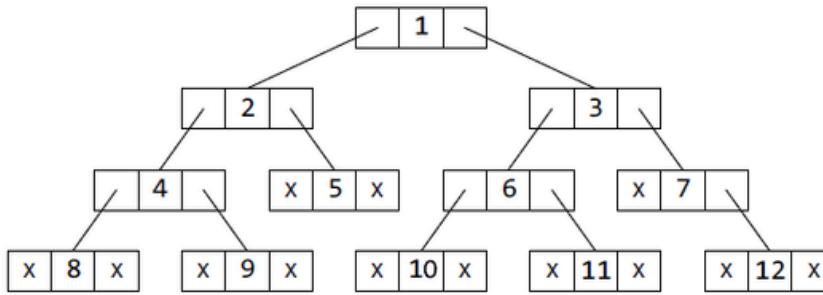


Figure 10.29 (b) Linked representation of the binary tree
(without threading)

- İş parçacıklı ikili ağaç, ikili ağaçla aynıdır ancak NULL işaretçilerini depolamada bir fark vardır. Şekil 10.29'da verilen ağacın bağlantılı gösterimini düşünün.

İş Parçacıklı İkili Ağaçlar

- Bağlantılı gösterimde, bir dizi düğüm, sol veya sağ alanlarında veya her ikisinde de NULL işaretçisi içerir.
- NULL işaretçisini depolamak için harcanan bu alan, başka yararlı bilgileri depolamak için verimli bir şekilde kullanılabilir.
- Örneğin, NULL girdileri, düğümün sıralı öncülüne veya sıralı ardılına bir işaretçi depolamak için değiştirilebilir.
- Bu özel işaretçilere thread (iş parçacığı) adı verilir ve thread'leri içeren ikili ağaçlara ise thread'li ağaçlar denir.
- *İş parçacıklı bir ikilinin bağlantılı gösteriminde*
- Ağaçta, iş parçacıkları oklar kullanılarak gösterilecektir.

İş Parçacıklı İkili Ağaçlar

30

- İkili bir ağaç işlemenin birçok yolu vardır ve her bir tür, ağacın işlenme biçimine göre farklılık gösterebilir.
- Bu kitapta, ağacın sıralı geçişini ele alacağız. Bunun dışında, iş parçacıklı bir ikili ağaç tek yönlü iş parçacığına veya iki yönlü iş parçacığına karşılık gelebilir.
- Tek yönlü iş parçacığında, bir iş parçacığı düğümün sağ veya sol alanında görünür. Tek yönlü iş parçacıklı bir ağaca tek iş parçacıklı ağaç da denir. İş parçacığı sol alanda görünürse, sol alan düğümün sıralı öncülüne işaret edecek şekilde yapılır.
- Böyle tek yönlü iş parçacıklı bir ağaca sol iş parçacıklı ikili ağaç denir. Tersine, iş parçacığı sağ alanda görünürse, o zaman düğümün sıralı halefine işaret edecktir. Böyle tek yönlü iş parçacıklı bir ağaca sağ iş parçacıklı ikili ağaç denir.
- Çift yönlü iş parçacıklı ağaç olarak da adlandırılan iki yönlü iş parçacıklı ağaçta, iş parçacıkları düğümün hem sol hem de sağ alanında görünür.
- Sol alan düğümün sıralı öncülüne işaret ederken, sağ alan onun ardılına işaret edecektir.
- İki yönlü iş parçacıklı ikili ağaca tam iş parçacıklı ikili ağaç da denir. İkili ağaçların teknik özellikleri, iki yönlü iş parçacığı ve iki yönlü iş parçacığı aşağıda açıklanmıştır.
- Şekil 10.29, iş parçacığı olmayan bir ikili ağaç ve buna karşılık gelen bağlantılı göstermektedir.
- Ağacın sıralı geçisi 8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12 olarak verilmiştir

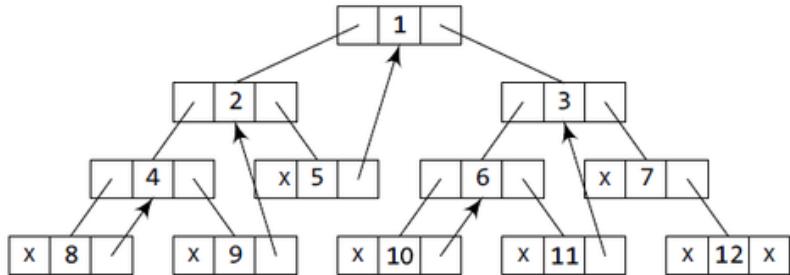


Figure 10.30 (a) Linked representation of the binary tree with one-way threading

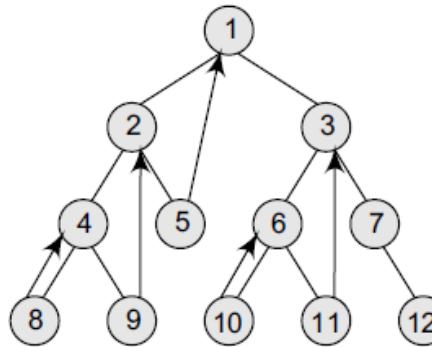


Figure 10.30 (b) Binary tree with one-way threading

- Şekil 10.30, tek yönlü iş parçacığına sahip bir ikili ağacı ve buna karşı gelen bağlantılı gösterimi göstermektedir.
- Düğüm 5, RIGHT alanında bir NULL işaretçisi içeriyor, bu nedenle sıra halefi olan node1'i gösterecek şekilde değiştirilecek.
- Benzer şekilde, düğüm 8'in SAĞ alanı düğüm 4'ü, düğüm 9'un SAĞ alanı düğüm 2'yi, düğüm 10'un SAĞ alanı düğüm 6'yı, düğüm 11'in SAĞ alanı düğüm 3'ü ve düğüm 12'nin SAĞ alanı sıralı bir ardılı olmadığı için NULL içerecektir.

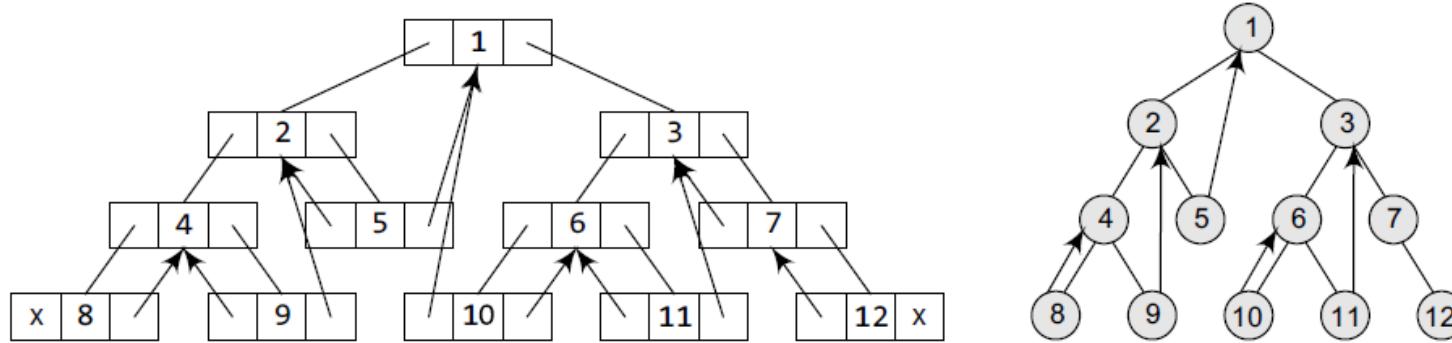


Figure 10.31 (a) Linked representation of the binary tree with threading, (b) binary tree with two-way threading

- Şekil 10.31, iki yönlü iş parçacığına sahip bir ikili ağacı ve buna karşılık gelen bağlantılı gösterimi göstermektedir.
- Düğüm 5, SOL alanında bir NULL işaretçisi içeriyor, bu nedenle sıralı öncülü olan düğüm 2'yi gösterecek şekilde değiştirilecek.
- Benzer şekilde, düğüm 8'in SOL alanı, sıralı bir öncülü olmadığı için NULL içerecektir, düğüm 7'nin SOL alanı düğüm 3'ü, düğüm 9'un SOL alanı düğüm 4'ü, düğüm 10'un SOL alanı düğüm 1'i, düğüm 11'in SOL alanı 6'yı ve düğüm 12'nin SOL alanı düğüm 7'yi işaret edecktir.

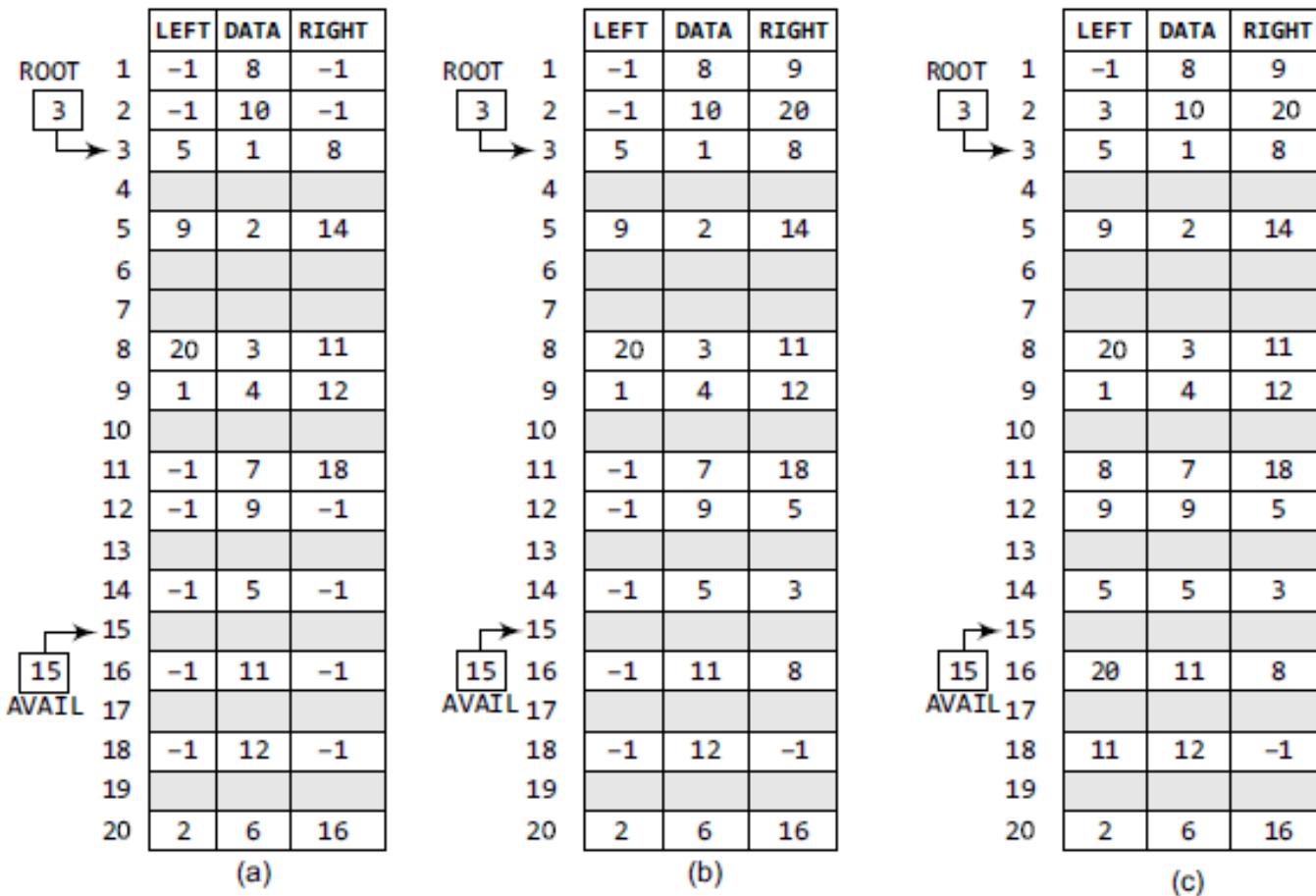


Figure 10.32 Memory representation of binary trees: (a) without threading, (b) with one-way, and (c) two-way threading

- Her düğüm için, daha önce ziyaret edilmemişse ve varsa, önce sol alt ağacı ziyaret edin.
- Daha sonra düğümün (kökün) kendisi, varsa sağ alt ağacını ziyaret edip takip edilir.
- Sağ alt ağaç yoksa, iş parçacıklı bağlantıyı kontrol edin ve iş parçacıklı düğümü dikkate alınan geçerli düğüm yapın. İş parçacıklı ikili ağacın sıralı geçışı için algoritma Şekil 10.33'te verilmiştir.

Step 1: Check if the current node has a left child that has not been visited. If a left child has not been visited, go to Step 2, else go to Step 3.

Step 2: Add the left child in the list of visited nodes. Make it as the current node and go to Step 6.

Step 3: If the current node has a right child, go to Step 4 else go to Step 5.

Step 4: Make that right child as current node and go to Step 6.

Step 5: Print the node and if there is a threaded node make it the current node.

Step 6: If all the nodes have visited then END else go to Step 1.

Figure 10.33 Algorithm for in-order traversal of a threaded binary tree

- Şekil 10.34'te verilen iş parçacıklı ikili ağacı ele alalım ve algoritmayı kullanarak üzerinde gezinelim.
- 1. Düğüm 1'in sol çocuğu yani ziyaret edilmemiş 2 var. Bu yüzden, ziyaret edilen düğümler listesi 2'yi ekleyin, onu geçerli düğüm yapın.
- 2. Düğüm 2'nin sol çocuğu yani ziyaret edilmemiş 4 vardır. Bu yüzden, ziyaret edilen düğümler listesine 4 ekleyin, onu geçerli düğüm yapın.
- 3. Düğüm 4'ün sol veya sağ çocuğu yoktur, bu yüzden 4 yazdırın ve iş parçacıklı bağlantısını koru edin. Düğüm 2'ye iş parçacıklı bir bağlantısı vardır, bu yüzden düğüm 2'yi geçerli düğüm yapın.
- 4. Düğüm 2'nin daha önce ziyaret edilmiş bir sol çocuğu var. Ancak, sağ çocuğu yok. Şimdi, 2'yi yazdırın ve düğüm 1'e giden iş parçacıklı bağlantısını takip edin. Düğüm 1'i geçerli düğüm yapın.
- 5. Düğüm 1'in daha önce ziyaret edilmiş bir sol çocuğu var. Bu yüzden 1 yazdır. Düğüm 1'in henüz ziyaret edilmemiş bir sağ çocuğu 3 var, bu yüzden onu geçerli düğüm yap.
- 6. Düğüm 3'ün henüz ziyaret edilmemiş bir sol çocuğu (düğüm 5) var, bu yüzden onu geçerli düğüm yapın.
- 7. Düğüm 5'in sol veya sağ çocuğu yoktur. Bu yüzden 5 yazdırın. Ancak, düğüm 3'e işaret eden iş parçacıklı bir bağlantısı vardır. Düğüm 3'ü geçerli düğüm yapın.
- 8. Düğüm 3'ün daha önce ziyaret edilmiş bir sol çocuğu var. Bu yüzden 3 yazdır.
- 9. Artık düğüm kalmadı, bu yüzden burada bitiriyoruz. Yazdırılan düğümlerin sırası şudur: 4 2 1 5

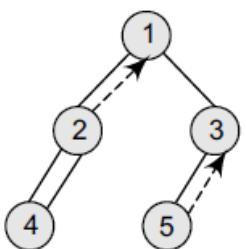


Figure 10.34 Threaded

İş Parçacıklı İkili Ağacın Avantajları

- Ağaçtaki elemanların doğrusal olarak dolaşılmasını sağlar.
- Doğrusal geçiş, yiğinların kullanımını ortadan kaldırır; yiğinlar da çok fazla bellek alanı ve bilgisayar zamanı tüketir.
- Ebeveyn işaretçilerinin açıkça kullanılmasına gerek kalmadan, belirli bir ögenin ebeveynini bulmayı sağlar.
- Düğümler, ardıl ve öncüllere ait işaretçileri sıralı olarak içerdiginden, iş parçacıklı ağaç, düğümlerin sıralı bir şekilde verildiği şekilde ileri ve geri dolaşılmasına olanak tanır.
- Böylece ikili ağaç ile iş parçacıklı ikili ağaç arasındaki temel farkı görüyoruz; ikili ağaçlarda bir düğümün çocuğu yoksa NULL işaretçisi depolaması ve böylece geriye dönmenin bir yolu olmamasıdır.

AVL Ağaçları

- AVL ağaç, G.M. Adelson-Velsky ve E.M. Landis tarafından 1962 yılında icat edilen kendi kendini dengeleyen bir ikili arama ağacıdır.
- Ağacın ismi mucitlerinin anısına AVL olarak adlandırılmıştır.
- Bir AVL ağacında, bir düğümün iki alt ağacının yükseklikleri en fazla bir farklılık gösterebilir.
- Bu özelliğinden dolayı AVL ağacına aynı zamanda yükseklik dengeli ağaç da denir.
- AVL ağacının kullanılmasının en önemli avantajı, ağacın yüksekliğinin $O(\log n)$ ile sınırlı olması nedeniyle, hem ortalama hem de en kötü durumda arama, ekleme ve silme işlemlerini gerçekleştirmenin $O(\log n)$ zaman almasıdır.

AVL Ağaçları

- AVL ağacının yapısı ikili arama ağacının yapısıyla aynıdır ancak küçük bir fark vardır.
- Yapısında, BalanceFactor adı verilen ek bir değişkeni depolar. Bu nedenle, her düğümün kendisiyle ilişkili bir denge faktörü vardır.
- Bir düğümün denge faktörü, sağ alt ağacının yüksekliğinin sol alt ağacının yüksekliğinden çıkarılmasıyla hesaplanır.
- Her düğümün denge faktörü -1, 0 veya 1 olan ikili arama ağacına yükseklik dengeli denir.
- Herhangi bir denge faktörüne sahip bir düğüm dengesiz olarak kabul edilir ve ağacın yeniden dengelenmesini gerektirir.

Denge faktörü = Yükseklik (sol alt ağaç) – Yükseklik (sağ alt ağaç)

AVL Ağaçları

- Bir düğümün denge faktörü 1 ise bu, ağacın sol alt ağacının sağ alt ağacından bir seviye yukarıda olduğu anlamına gelir.
- Bu tür ağaçlara bu nedenle sol ağırlıklı ağaç adı verilir.
- Bir düğümün denge faktörü 0 ise bu, sol alt ağacın yüksekliğinin (sol alt ağaçtaki en uzun yol) sağ alt ağacın yüksekliğine eşit olduğu anlamına gelir.
- Bir düğümün denge faktörü -1 ise, bu ağacın sol alt ağacını sağ alt ağacından bir seviye aşağıda olduğu anlamına gelir. nedenle böyle bir ağaca sağ ağırlıklı ağaç denir.

AVL Ağaçları

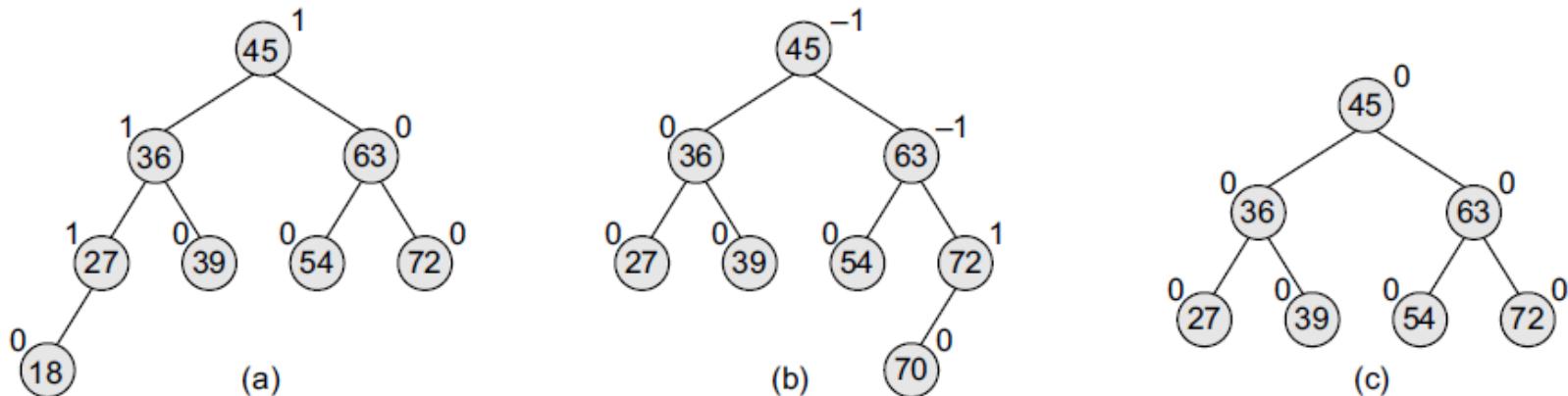


Figure 10.35 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

- Şekil 10.35'e bakın. 18, 39, 54 ve 72 düğümlerinin çocuğu olmadığını, dolayısıyla denge faktörlerinin = 0 olduğunu unutmayın.
- Düğüm 27'nin bir sol çocuğu ve sıfır sağ çocuğu vardır. Bu nedenle, sol alt ağacın yüksekliği = 1 iken, sağ alt ağacın yüksekliği = 0'dır. Bu nedenle, denge faktörü = 1'dir.
- 36. düğüme bakın, yüksekliği = 2 olan sol alt ağacı var, oysa sağ alt ağacın yüksekliği 1. Bu nedenle, denge faktörü = $2 - 1 = 1$. Benzer şekilde, 45. düğümün denge faktörü = $-2 = 1$; ve 63. düğümün denge faktörü 0'dır ($1 - 1$).
- Şimdi, sağ ağırlıklı AVL ağacını ve dengeli AVL ağacını gösteren Şekil 10.35 (b) ve (c)'ye bakın.

AVL Ağaçları

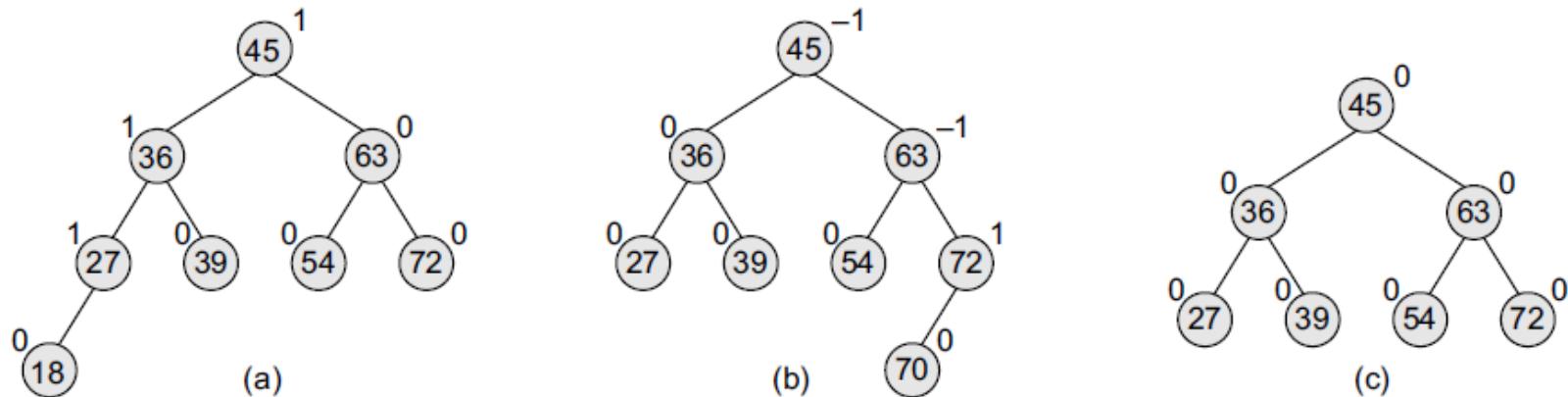


Figure 10.35 (a) Left-heavy AVL tree, (b) right-heavy tree, (c) balanced tree

- Şekil 10.35'te verilen ağaçlar, her düğümün dengeleme faktörünün 1, 0 veya -1 olması nedeniyle tipik AVL ağacı adaylarıdır.
- Ancak, bir AVL ağacından yapılan eklemeler ve silmeler düğümlerin denge faktörünü bozabilir ve bu nedenle ağacın yeniden dengelenmesi gerekebilir. Ağac, kritik düğüm rotasyon gerçekleştirilecek yeniden dengelenir.
- Dört tip rotasyon vardır: LL rotasyonu, RR rotasyonu, LR rotasyonu ve RL rotasyonu.
- Yapılması gereken döndürme türü, belirli duruma bağlı olarak değişecektir. Aşağıdak bölmende, AVL ağaçlarında ekleme, silme, arama ve döndürmeleri ele alacağız.

AVL Ağacında Bir Düğüm Arama

- AVL ağacında arama, ikili arama ağacında yapılan aramayı aynı şekilde yapılır.
- Ağacın yükseklik dengelemesi nedeniyle arama işleminin tamamlanması $O(\log n)$ zaman alır.
- İşlem ağacın yapısını değiştirmediği için özel bir düzenlemeye gerek yoktur.

AVL Ağacına Yeni Bir Düğüm Ekleme

- AVL ağacına ekleme işlemi de ikili arama ağacında yapıldığı gibi yapılır.
- AVL ağacında, yeni düğüm her zaman yaprak düğümü olarak eklenir. Ancak ekleme adımını genellikle ek bir döndürme adımı takip eder.
- Ağacın dengesini sağlamak için rotasyon yapılır.
- Ancak yeni düğümün eklenmesi denge faktörünü bozmuyorsa, yani her düğümün denge faktörü hala -1, 0 veya 1 ise, o zaman rotasyonlara gerek yoktur.

AVL Ağacına Yeni Bir Düğüm Ekleme

- Ekleme sırasında yeni düğüm yaprak düğümü olarak eklenir, böylece denge faktörü her zaman sıfıra eşit olur.
- Denge faktörleri değişecek olan tek düğümler, ağacın kökü ile eklenen düğüm arasındaki yolda bulunanlardır.
- Yol üzerindeki herhangi bir düğümde gerçekleşebilecek olası değişiklikler şu şekildedir:
 - Başlangıçta düğüm ya sol ya da sağ ağırlıklıdır ve yerleştirildikten sonra dengeli hale gelir.
 - Başlangıçta düğüm dengelidir ve yerleştirildikten sonra ya sol ya da sağ ağırlıklı hale gelir.
 - Başlangıçta düğüm ağırdı (sol veya sağ) ve yeni düğüm ağır alt ağa eklendi, böylece dengesiz bir alt ağaç yaratıldı. Böyle bir düğüme kritik düğüm denir.

AVL Ağacına Yeni Bir Düğüm Ekleme

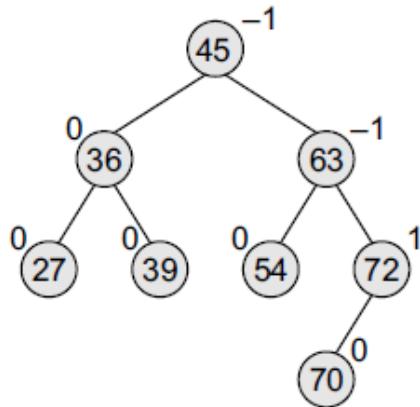


Figure 10.36 AVL tree

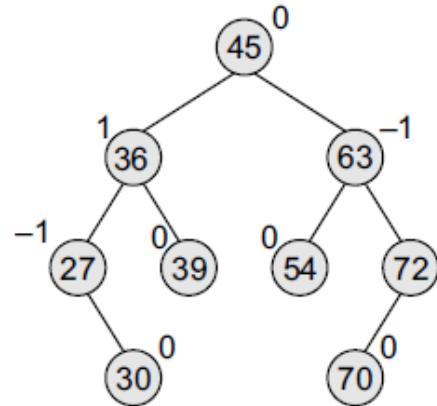


Figure 10.37 AVL tree after inserting a node with the value 30

- Eğer 30 değerine sahip yeni bir düğüm eklersek, yeni ağacı yine dengeli olacak ve bu durumda herhangi bir döndürmey gerek kalmayacaktır.
- Şekil 10.37'de verilen ağaca bakın; bu ağaca 30. düğüm eklendikten sonraki durum gösterilmektedir.

AVL Ağacına Yeni Bir Düğüm Ekleme

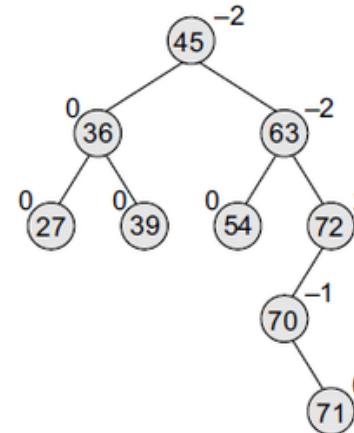
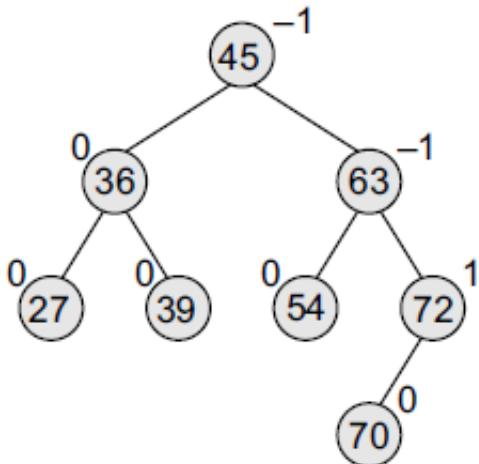


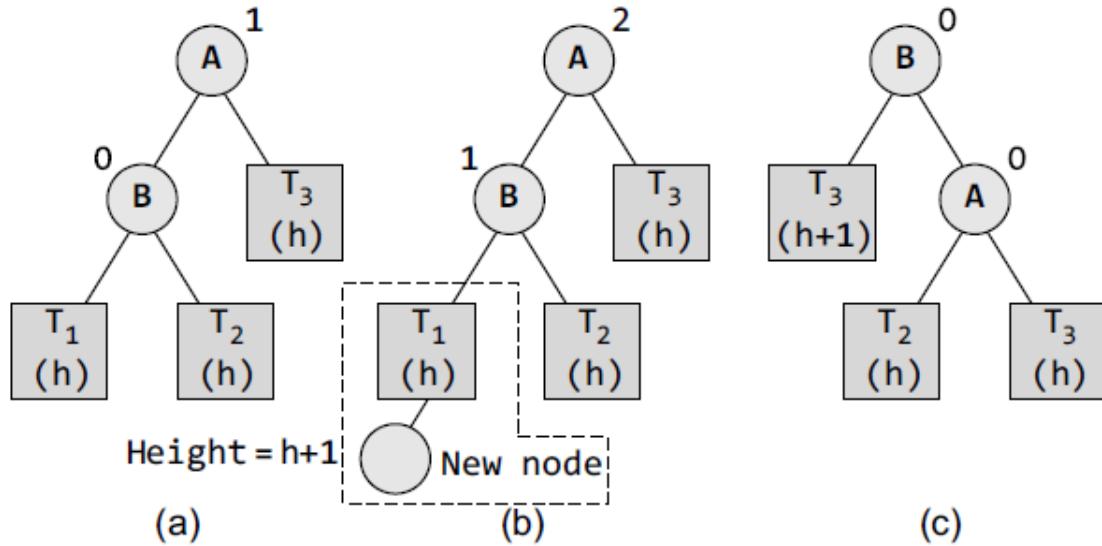
Figure 10.39 AVL tree after inserting a

- 71 değerine sahip yeni bir düğüm eklendikten sonra, yeni ağaç Şekil 10.39'da gösterildiği gibi olacaktır. Ağaçta denge faktörleri 2, -2 ve -2 olan üç düğüm olduğunu ve bu nedenle ağacın AVLness'inin bozulduğunu unutmayın.
- İşte burada *rotasyon yapma ihtiyacı* ortaya çıkıyor.
- Rotasyonu gerçekleştirmek için ilk görevimiz kritik düğümü bulmaktır. Kritik düğüm, eklenen düğümden köke giden yoldaki denge faktörü ne -1, ne 0 ne de 1 olan en yakın ata düğümdür. Yukarıda verilen ağaçta kritik düğüm 72'dir.
- Ağacı yeniden dengelemedeki ikinci görev, hangi tür rotasyonun yapılması gerektiğini belirlemektir.
- Dört tip yeniden dengeleme rotasyonu vardır ve bu rotasyonların uygulanması, eklenen düğümü kritik düğüme göre pozisyonuna bağlıdır.

AVL Ağacına Yeni Bir Düğüm Ekleme

- Rotasyonun dört kategorisi şunlardır:
 - *LL rotasyonu* Kritik düğümün sol alt ağacının sol alt ağacına yeni düğüm eklenir.
 - *RR rotasyonu* Kritik düğümün sağ alt ağacının sağ alt ağacına yeni düğüm eklenir.
 - *LR dönüşü* Yeni düğüm kritik düğümün sol alt ağacının sağ alt ağacına eklenir.
 - *RL rotasyonu* Yeni düğüm kritik düğümün sağ alt ağacının sol alt ağacına eklenir.

AVL Ağacına Yeni Bir Düğüm Ekleme



- **LL Rotasyonu**
- Bu rotasyonların her birini detaylı olarak inceleyelim.
- İlk olarak, LL rotasyonunun nerede ve nasıl uygulandığını göreceğiz. Şekil 10.40'ta verilen ve AVL ağacını gösteren ağacı ele alalım.
- Ağaç (a) bir AVL ağaçıdır. Ağaç (b)'de, kritik düğüm A'nın sol alt ağacının sol alt ağacı yeni bir düğüm eklenir (A düğümü kritik düğümdür çünkü denge faktörü -1, 0 veya 1 olmayan en yakın atadır), bu nedenle ağaç (c)'de gösterildiği gibi LL rotasyonu uygulanır.
- Dönme sırasında, B düğümü kök olur, T_1 ve A onun sol ve sağ çocuğu olur. T_2 ve T_3 is A'nın sol ve sağ alt ağaçları olur.

Figure 10.40 LL rotation in an AVL tree

AVL Ağacına Yeni Bir Düğüm Ekleme

Example 10.3 Consider the AVL tree given in Fig. 10.41 and insert 18 into it.

Solution

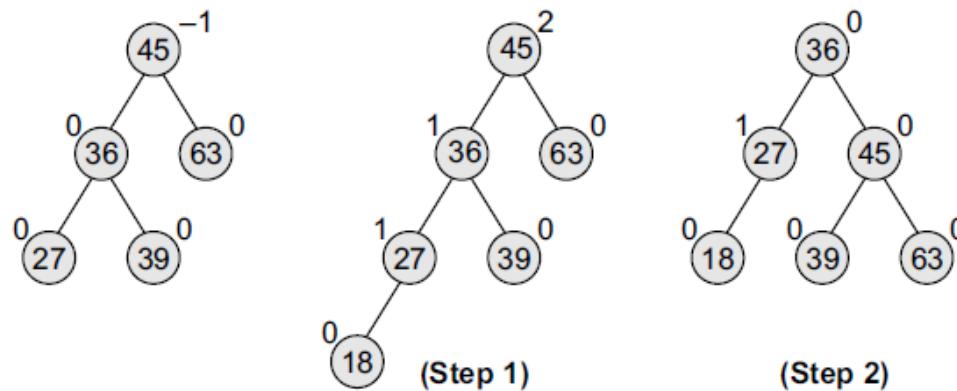
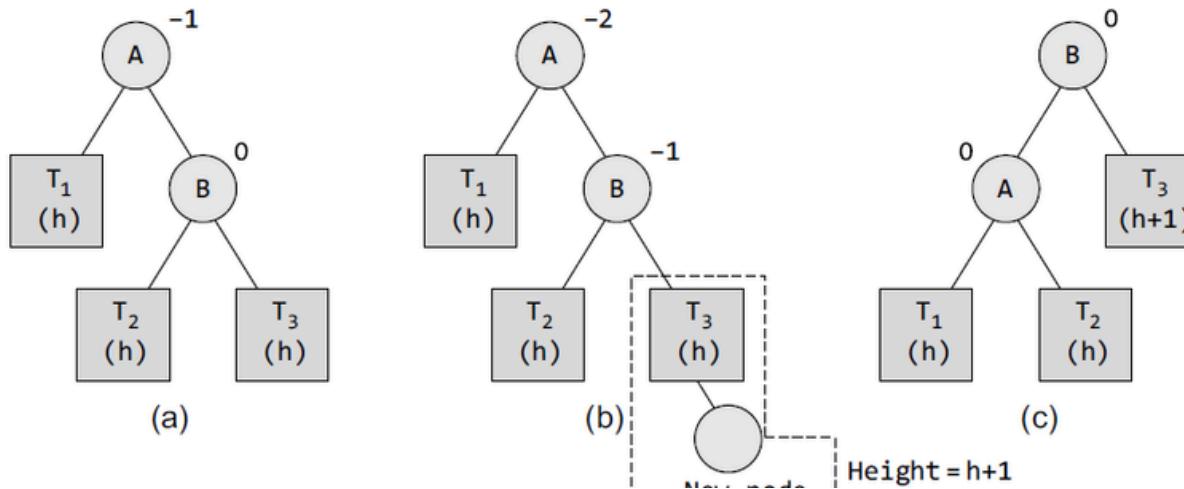


Figure 10.41 AVL tree

AVL Ağacına Yeni Bir Düğüm Ekleme



- **RR Rotasyonu**
- Ağacı (a) bir AVL ağacıdır.
- (b) ağacında, kritik düğüm A'nın sağ alt ağacının sağ alt ağacına yeni düğüm eklenir (A düğümü kritik düğümdür çünkü denge faktörü $-1, 0$ veya 1 olmayan en yakın atadır), bu nedenle (c) ağacında gösterildiği gibi RR rotasyonu uygularız.
- Yeni düğümün artık T_3 ağacının bir parçası haline geldiğine dikkat edin.
- Dönme sırasında, B düğümü kök olur, A ve T_3 ise onun sol ve sağ çocuğu olur. T_1 ve T_2 ise A'nın sol ve sağ alt ağaçları olur.

AVL Ağacına Yeni Bir Düğüm Ekleme

Example 10.4 Consider the AVL tree given in Fig. 10.43 and insert 89 into it.

Solution

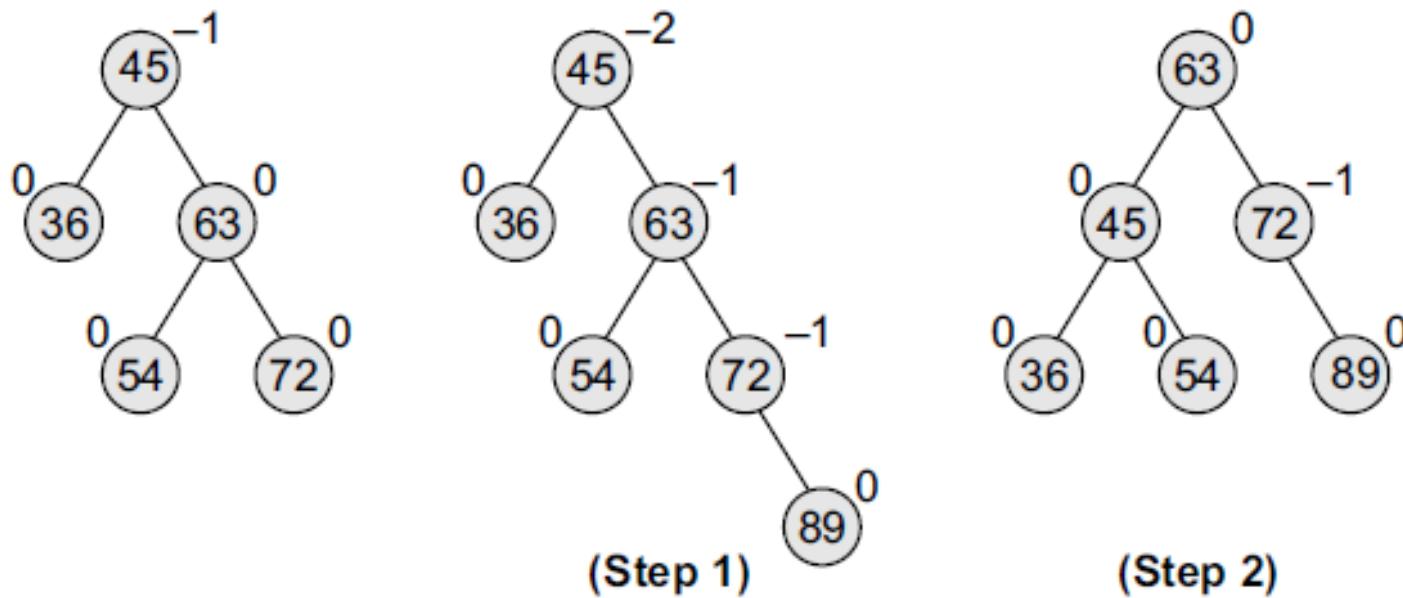


Figure 10.43 AVL tree

AVL Ağacına Yeni Bir Düğüm Ekleme

Example 10.5 Consider the AVL tree given in Fig. 10.45 and insert 37 into it.

Solution

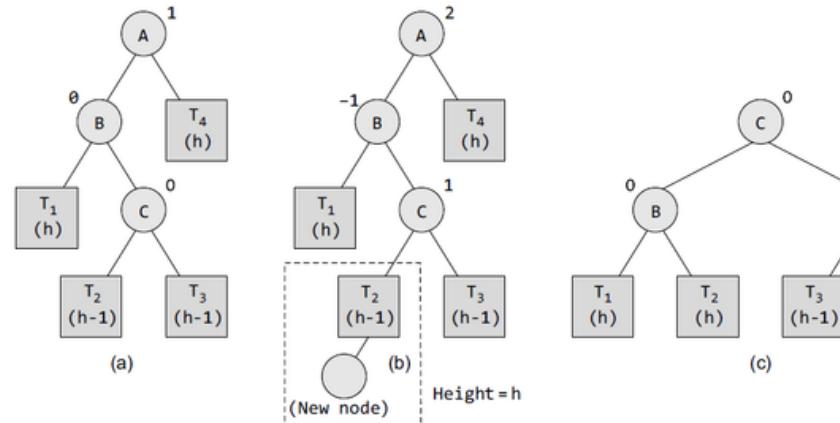
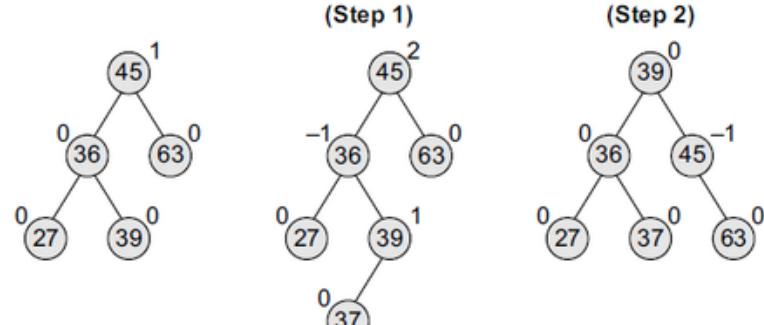


Figure 10.44 LR rotation in an AVL tree

- **LR ve RL Rotasyonları**
- Şekil 10.44'te verilen AVL ağacını ele alın ve LR r Ağacı (a)'nın nasıl bir AVL ağacı olduğunu görün. Ağaç (b)'de, kritik düğüm A'nın sol alt ağacının sağ alt ağacına yeni bir düğüm eklenir (A düğümü kritik düğümdür çünkü denge faktörü $-1, 0$ veya 1 olmayan en yakın atadır), bu nedenle ağaç (c)'de gösterildiği gibi LR rotasyonu uygularız.
- Yeni düğümün artık T2 ağacının bir parçası haline geldiğine dikkat edin. Dönüş sırasında, düğüm C kök olur, B ve A ise onun sol ve sağ çocukları olur.
- B düğümünün sol ve sağ alt ağaçları T1 ve T2'dir ve T3 ve T4, A düğümünün ve sağ alt ağaçları olur.

AVL Ağacına Yeni Bir Düğüm Ekleme

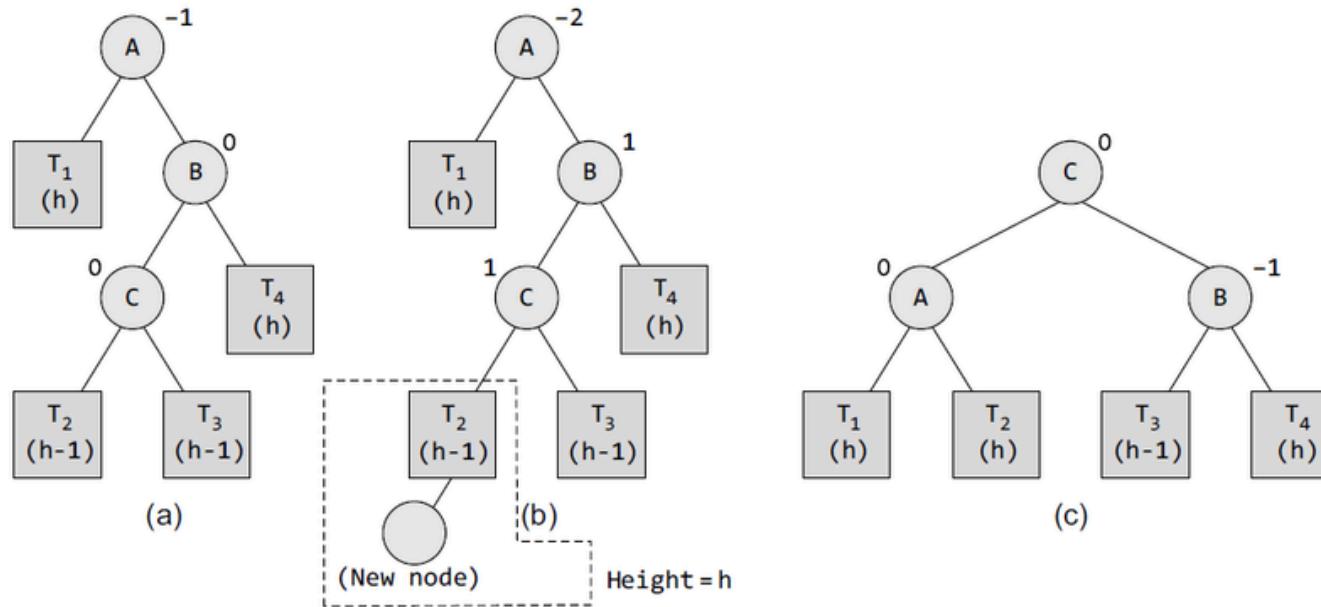


Figure 10.46 RL rotation in an AVL tree

- ***LR ve RL Rotasyonları***
- Şimdi, Şekil 10.46'da verilen AVL ağacını ele alalım ve ağacı yeniden dengelemek için RL rotasyonunun nasıl yapıldığını görelim. Ağacı yeniden dengelemek için rotasyon yapılır.
- Ağaç (a) bir AVL ağaçıdır. Ağaç (b)'de, kritik düğüm A'nın sağ alt ağacının sol alt ağacına yeni bir düğüm eklenir (A düğümü kritik düğümdür çünkü denge faktörü $-1, 0$ veya 1 olmayan en yakın atadır), bu nedenle ağaç (c)'de gösterildiği gibi RL rotasyonu uygularız.
- Yeni düğümün artık T_2 ağacının bir parçası haline geldiğine dikkat edin.
- Dönme sırasında C düğümü kök olur, A ve B ise onun sol ve sağ çocukları olur.
- A düğümünün sol ve sağ alt ağaçları T_1 ve T_2 'dir ve T_3 ve T_4 , B düğümünün sol ve sağ alt ağaçları olur.

AVL Ağacına Yeni Bir Düğüm Ekleme

Example 10.6 Construct an AVL tree by inserting the following elements in the given order.
63, 9, 19, 27, 18, 108, 99, 81.

Solution

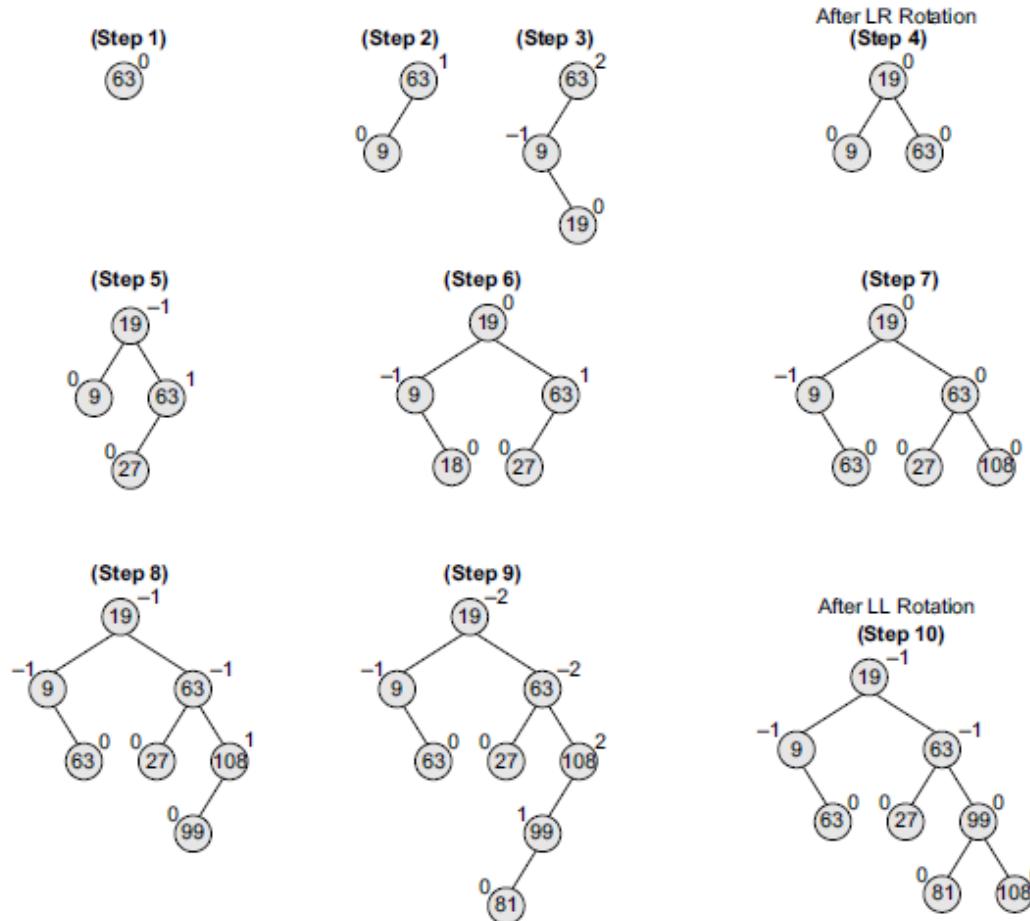


Figure 10.47 AVL tree

Bir AVL Ağacından bir düğümü silme

- Bir AVL ağacındaki bir düğümün silinmesi, ikili arama ağaçlarındakine benzerdir. Ancak bir adım öndedir.
- Silme, ağacın AVL'liğini bozabilir, bu nedenle AVL ağacını yeniden dengelemek için rotasyonlar gerçekleştirmemiz gereklidir. Belirli bir düğümü sildikten sonra bir AVL ağacında gerçekleştirilebilecek iki rotasyon sınıfı vardır.
- Bu rotasyonlar R rotasyonu ve L rotasyonudur.
- X düğümünün AVL ağacından silinmesi üzerine, eğer A düğümü kritik düğüm haline gelirse (X'ten kök düşüme giden yoldaki denge faktörü 1, 0 veya -1 olmayan en yakın ata düğüm), o zaman rotasyon türü X'in A'nın sol alt ağacı mı yoksa sağ alt ağacında mı olduğuna bağlıdır.
- Silinecek düğüm A'nın sol alt ağacında ise L rotasyonu, X'in sağ alt ağacında ise R rotasyonu uygulanır.
- Ayrıca, L ve R rotasyonlarının üç kategorisi vardır. L rotasyonunun varyasyonları L-1, L0 ve L1 rotasyonudur. Buna karşılık R rotasyonu için R0, R-1 ve R1 rotasyonları vardır.
- Bu bölümde sadece R rotasyonunu ele alacağız. L rotasyonları R rotasyonları ayna görüntüleridir.

Bir AVL Ağacından bir düğümü silme

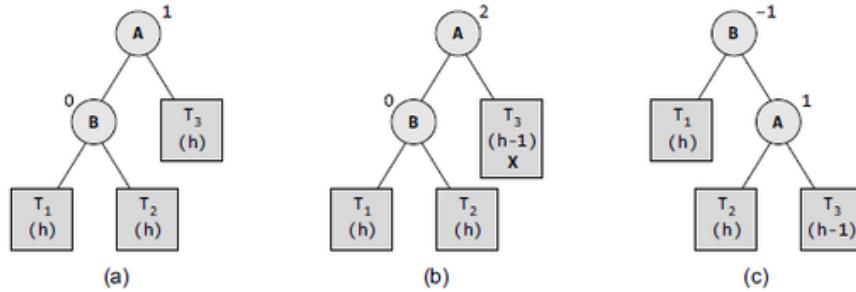


Figure 10.48 R0 rotation in an AVL tree

Example 10.7 Consider the AVL tree given in Fig. 10.49 and delete 72 from it.

Solution

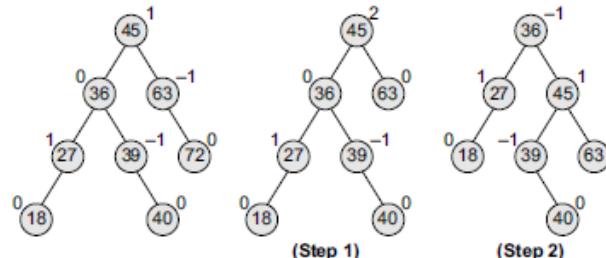


Figure 10.49 AVL tree

- **R0 Dönmesi**
- B, A'nın sol veya sağ alt ağacının kökü olsun (kritik düğüm). B'nin denge faktörü 0 ise R0 dönüşümü uygulanır. Bu, Şekil 10.48'de gösterilmiştir.
- Ağaç (a) bir AVL ağaçıdır. Ağaç (b)'de, düğüm X kritik düğüm A'nın sağ alt ağacından silinecektir (düğüm A kritik düğümdür çünkü denge faktörü -1, 0 veya 1 olmayan en yakın atadır). Düğüm B'nin denge faktörü 0 olduğundan, ağaç (c)'de gösterildiği gibi R0 rotasyonunu uygularız.
- Dönme işlemi sırasında B düğümü kök olur, T1 ve A ise onun sol ve sağ çocuğu olur.
- T2 ve T3, A'nın sol ve sağ alt ağaçları olur.

Bir AVL Ağacından bir düğümü silme

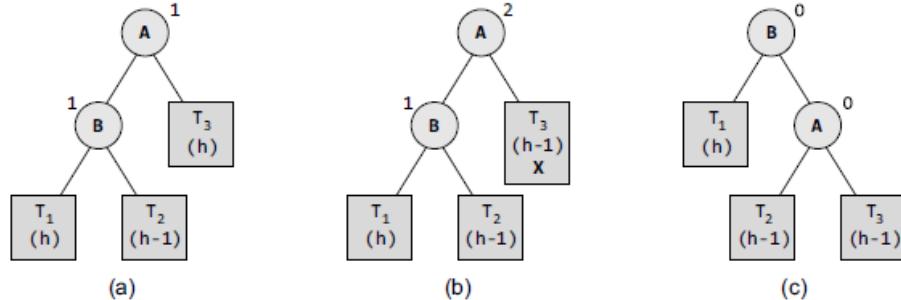


Figure 10.50 R1 rotation in an AVL tree

Example 10.8 Consider the AVL tree given in Fig. 10.51 and its solution.

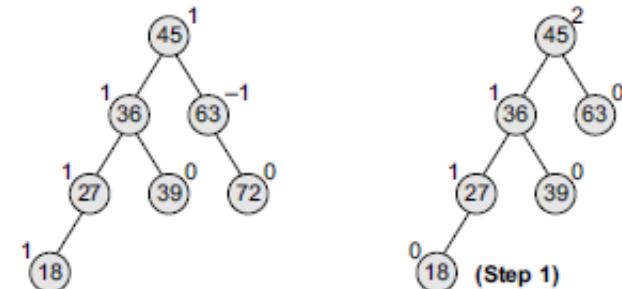


Figure 10.51 AVL tree

- **R1 Dönmesi**
- B, A'nın sol veya sağ alt ağacının kökü olsun (kritik düğüm). B'nin denge faktörü 1 ise rotasyonu uygulanır. R0 ve R1 rotasyonlarının LL rotasyonlarına benzer olduğunu; tek farkın R0 ve R1 rotasyonlarının farklı denge faktörleri üretmesi olduğunu gözlemleyin.
- Ağaç (a) bir AVL ağacıdır. Ağaç (b)'de, düğüm X kritik düğüm A'nın sağ alt ağacından silinecektir (düğüm A kritik düğümdür çünkü denge faktörü -1, 0 veya 1 olmayan en yakın atadır).
- B düğümünün denge faktörü 1 olduğunu, ağaçta (c) gösterildiği gibi R1 dönüşünü uygularız.
- Dönme işlemi sırasında, B düğümü kök olur, T1 ve A ise onun sol ve sağ çocukları olur. T2 ve T3 ise A'nın sol ve sağ alt ağaçları olur.

Bir AVL Ağacından bir düğümü silme

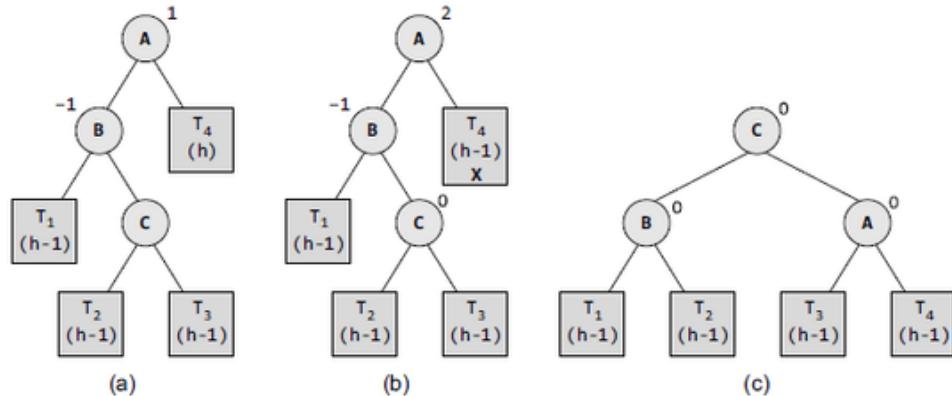


Figure 10.52 R1 Rotation in an AVL tree

Example 10.9 Consider the AVL tree given in Fig. 10.53 and **Solution**

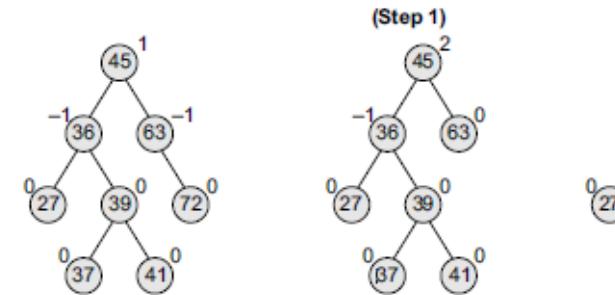


Figure 10.53 AVL tree

- **R-1 Dönmesi**
- B, A'nın sol veya sağ alt ağacının kökü olsun (kritik düğüm). B'nin denge faktörü -1 ise R-1 rotasyonu uygulanır.
- R-1 dönüşünün LR dönüşüne benzediğini gözlemleyin. Ağacı (a) bir AVL ağacıdır. Ağacı (b)'de, düğüm X kritik düğüm A'nın sağ alt ağacından silinecektir (düğüm A kritik düğümdür çünkü denge faktörü -1, 0 veya 1 olmayan en yakın atadır).
- B düğümünün denge faktörü -1 olduğundan, ağacta (c) gösterildiği gibi R-1 dönüşünü uygularız.
- Dönme sırasında, C düğümü kök olur, T1 ve A onun sol ve sağ çocuğu olur. T2 ve T3, A'nın sol ve sağ alt ağaçları olur.

Bir AVL Ağacından bir düğümü silme

Example 10.10 Delete nodes 52, 36, and 61 from the AVL tree given in Fig. 10.54.

Solution

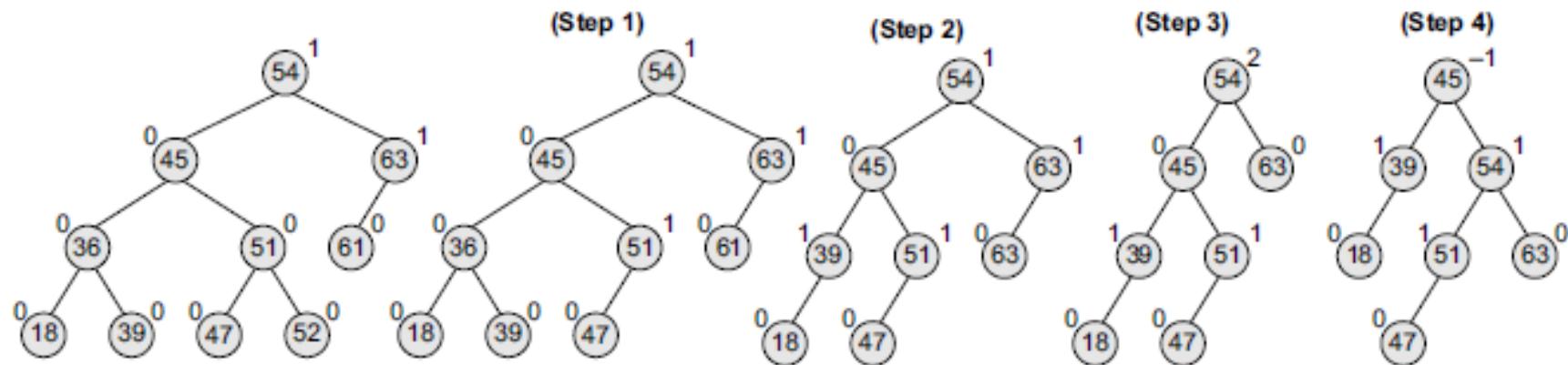


Figure 10.54 AVL tree

Kırmızı-Siyah Ağaçlar

- Kırmızı-siyah ağaç, 1972 yılında Rudolf Bayer tarafından ic edilen ve ‘simetrik ikili B-ağacı’ adını veren, kendi kendini dengeleyen bir ikili arama ağacıdır.
- Kırmızı-siyah ağaç karmaşık olmasına rağmen, işlemleri içi iyi bir en kötü durum çalışma süresine sahiptir ve arama, ekleme ve silme işlemlerinin tümü $O(\log n)$ sürede yapılabildiğinden kullanımı verimlidir; burada n , ağactaki düğüm sayısıdır.
- Pratikte, kırmızı-siyah ağaç, ağacı makul bir dengede tutm için akıllıca ekleme ve çıkarma işlemleri yapan bir ikili arama ağacıdır.
- Kırmızı-siyah ağaç hakkında dikkat edilmesi gereken özel bir nokta, bu ağaçta yaprak düğümlerinde hiçbir verinin saklanmamasıdır.

Kırmızı-Siyah Ağaçlar

- **Kırmızı-Siyah Ağaçların Özellikleri**
- Kırmızı-siyah ağaç, her düğümün kırmızı veya siyah renkte olduğu ikili arama ağacıdır.
- İkili arama ağacının diğer kısıtlamalarının yanı sıra, kırmızı-siyah ağacın aşağıdaki ek gereksinimleri vardır:
 - 1. Bir düğümün rengi kırmızı veya siyadır.
 - 2. Kök düğümün rengi her zaman siyadır.
 - 3. Tüm yaprak düğümleri siyadır.
 - 4. Her kırmızı düğümün iki çocuğu da siyah renktedir.
 - 5. Belirli bir düğümden herhangi bir yaprak düğümüne giden her basit yol, eşit sayıda siyah düğüme sahiptir.

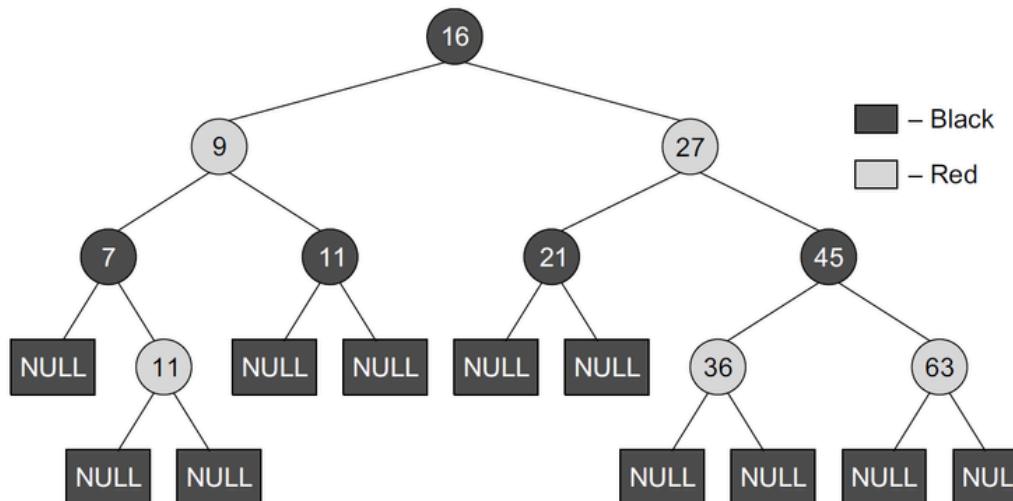


Figure 10.55 Red-black tree

Kırmızı-Siyah Ağaçlar

- Bu kısıtlamalar kırmızı-siyah ağaçların kritik bir özelliğini zorunlu kılar. Kök düğümden herhangi bir yaprak düşüme giden en uzun yol, o ağaçtaki kökten herhangi bir diğer yaprağa giden en kısa yolun iki katından daha uzun değildir.
- Bu, kabaca dengeli bir ağaçla sonuçlanır. Ekleme, silme ve arama gibi işlemler ağacın yüksekliğine orantılı en kötü durum zamanları gerektirdiğinden, yükseklikteki bu teorik üst sınır, sıradan ikili arama ağaçlarının aksine, kırmızı-siyah ağaçların en kötü durumda verimli olmasını sağlar.
- Bu özelliklerin önemini anlamak için, 4. özelliğe göre hiçbir yolun yan yana iki kırmızı düşüme sahip olamayacağını belirtmek yeterlidir.
- En kısa olası yol tüm düşümleri siyaha boyayacak, en uzun olası yol is dönüşümlü olarak bir kırmızı ve bir siyah düşüme sahip olacaktır.
- Tüm maksimal yolların aynı sayıda siyah düşümü olduğundan (özellik 5), hiçbir yol diğerinin iki katından daha uzun değildir.

Kırmızı-Siyah Ağaçlar

- Şekil 10.56 kırmızı-siyah ağaç olmayan bazı ikili arama ağaçlarını göstermektedir.

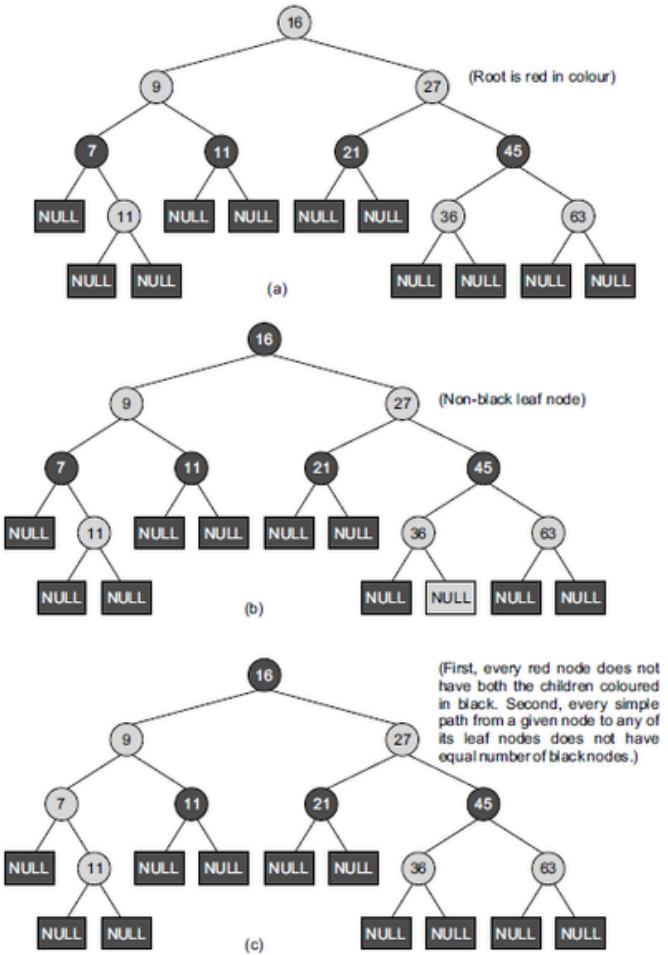


Figure 10.56 Trees

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

64

- Kırmızı-siyah bir ağaçta salt okunur bir işlem (örneğin bir ağaçtaki düğümler dolaşmak) gerçekleştirmek, ikili arama ağaçları için kullanılanlardan hiçbir değişiklik gerektirmez.
- Her kırmızı-siyah ağacın ikili arama ağacının özel bir durumu olduğunu unutmayın.
- Ancak ekleme ve silme işlemleri kırmızı-siyah ağacın özelliklerini ihlal edebilir.
- Bu nedenle, bu işlemler, az sayıda ($O(\log n)$ veya amortize edilmiş $O(1)$) renk değişikliği gerektirebilecek kırmızı-siyah özelliklerinin geri yüklenmesi ihtiyacını yaratabilir.

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- Ekleme işlemi, ikili arama ağacına yeni bir düğüm eklediğimiz şekilde başlar.
- Ancak ikili arama ağacında, yeni düğümü her zaman bir yaprak olarak ekleriz kırmızı-siyah ağaçta ise yaprak düğümleri veri içermez.
- Yani yeni düğümü yaprak düğüm olarak eklemek yerine, iki siyah yaprak düğümü olan kırmızı bir iç düğüm ekliyoruz.
- Yeni düğümün renginin kırmızı, yaprak düğümlerinin ise siyah renkte olduğunu dikkat edin.

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

65

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- Yeni bir düğüm eklendiğinde, kırmızı-siyah ağacının bazı özelliklerini ihlal edebilir.
- Yani malını iade etmek için belli durumları kontrol ediyoruz ve eklemeden sonra çıkış duruma göre malı iade ediyoruz.
- Ancak bu durumları detaylı bir şekilde öğrenmeden önce, kullanılacak bazı önemli terimleri tartışalım.
- *Bir düğümün (N) büyük ebeveyn düğümü (G), insan aile ağaçlarındaki gibi, N'nin ebeveyninin (P) ebeveynini ifade eder.*
- Bir düğümün büyük ebeveynini bulmak için C kodu aşağıdaki gibi verilebilir:
 yapı düğümü * grand_parent(yapı düğümü *n)
 {
 // Ebeveyn yoksa büyukanne ve büyükbaşa da yok demektir
 eğer ((n != NULL) && (n -> ebeveyn != NULL))
 return n -> ebeveyn -> ebeveyn;
 başka
 NULL döndür;
 }

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

66

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- Bir düğümün (N) amca düğümü (U), insan aile ağaçlarında olduğu gibi, N 'nin ebeveyninin (P) kardeşini ifade eder.
- Bir düğümün amcasını bulmak için C kodu aşağıdaki gibi verilebilir:

```
yapı düğümü *amca(yapı düğümü *n)
{
    yapı düğümü *g;
    g = büyük_ebeveyn(n);
    //Büyükbabası ve büyükannesi olmayan birinin amcası da olamaz
    eğer (g == NULL)
        NULL döndür;
    eğer (n -> ebeveyn == g -> sol)
        g -> sağa dön;
    başka
        g -> sola dön;
}
```

- Kırmızı-siyah bir ağaca yeni bir düğüm eklediğimizde aşağıdakilere dikkat edin:
 - Tüm yaprak düğümleri her zaman siyadır. Bu yüzden özellik 3 her zaman geçerlidir.
 - Özellik 4 (her kırmızı düğümün iki çocuğu da siyadır) yalnızca kırmızı bir düğüm ekleyerek, siyah bir düğümü kırmızıya boyayarak veya bir döndürme yaparak tehdit edilir.
 - Özellik 5 (belirli bir düğümden yaprak düğümlerine giden tüm yolların eşit sayıda siyah düğümü vardır) yalnızca siyah bir düğüm ekleyerek, kırmızı bir düğümü siyaya boyayarak veya bir döndürme yaparak tehdit edilir.

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

67

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- **Durum 1: Yeni Düğüm N, Ağacın Kökü Olarak Eklenir**
- Bu durumda N siyaha boyanır, çünkü ağacın kökü her zaman siyahdır.
- N her yola aynı anda bir siyah düğüm eklediğinden, Özellik 5 ihlal edilmez.
- 1. durum için C kodu aşağıdaki gibi verilebilir:

```
void case1(yapı düğümü *n)
{
    eğer (n -> ebeveyn == NULL) // Kök düğüm
        n -> renk = SİYAH;
    başka
        durum2(n);
}
```

Kırmızı-Siyah Ağaçlar Üzerindeki⁶⁸ Operasyonlar

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- Durum 2: Yeni Düğümün Ana Düğümü P Siyahdır**
- Bu durumda, her kırmızı düğümün her iki çocuğu da siyadır, dolayısıyla Özellik 4 geçersiz kılınmaz.
- Özellik 5 de tehdit altında değildir. Bunun nedeni, yeni düğüm N'nin iki siyah yaprak çocuğuna sahip olmasıdır, ancak N kırmızı olduğundan, çocukların her birinden geç yollar aynı sayıda siyah düşüme sahiptir.
- 2. durumu kontrol etmek için C kodu aşağıdaki gibi verilebilir:

```
void case2(yapı düğümü *n)
{
    eğer (n -> ebeveyn -> renk == SİYAH)
        geri dönmek;
        /* Kırmızı siyah ağaç özelliği ihlal edilmemiştir*/
        başka
        durum3(n);
}
```
- Aşağıdaki durumlarda, N'nin bir büyük ebeveyn düğümü G'ye sahip olduğu varsayıılır, çünkü ebeveyni P kırmızıdır ve eğer kök olsaydı siyah olurdu. Bu nedenle, N'nin ayrıca bir amca düğümü U'su vardır (U'nun bir yaprak düğümü veya bir iç düğüm olup olmadığına bakılmaksızın).

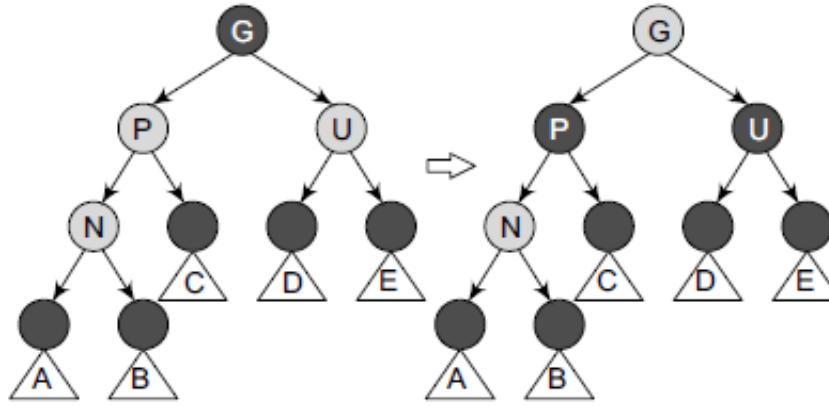
Kırmızı-Siyah Ağacı⁶⁹lar Üzerindeki Operasyonlar

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- **Durum 3: Hem Ebeveyn (P) hem de Amca (U) Kırmızıysa**
- Bu durumda, Özellik 4 ihlal edilir. Üçüncü durumdaki ekleme Şekil 10.57'de gösterilmiştir.
- Özellik 4'ü eski haline getirmek için her iki düğüm (P ve U) siyaha boyanır ve büyük ebeveyn G ise kırmızıya boyanır.
- Şimdi, yeni kırmızı düğüm N'nin siyah bir ebeveyni var. Ebeveyn veya amcadan geçen herhangi bir yol büyük ebeveyinden geçmek zorunda olduğundan, bu yollardaki siyah düğümlerin sayısı değişmedi.
- Ancak, büyük ebeveyn G artık kök düğümün her zaman siyah olduğunu belirten Özellik 2'yi veya her kırmızı düğümün her iki çocuğunun da siyah olduğunu belirten Özellik 4'ü ihlal edebilir.
- G'nin kırmızı bir ebeveyni olduğunda Özellik 4 ihlal edilecektir. Bu sorunu düzeltmek için, bu prosedürün tamamı Durum 1'den G üzerinde yinelemeli olarak gerçekleştirilir.

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

70



Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- Durum 3: Durum 3 eklemesini ele alan C kodu aşağıdaki gibidir:

```
void case3(yapı düğümü *n)
{
    yapı düğümü *u, *g;
    u = amca (n);
    g = büyük_ebeveyn(n);
    eğer ((u != NULL) && (u -> renk == KIRMIZI))
        n -> ebeveyn -> renk = SİYAH;
        u -> renk = SİYAH;
        g -> renk = KIRMIZI;
        durum1(g);
    }
    başka {
        insert_case4(n);
    }
}
```

Not Kalan durumlarda, ebeveyn düğümü P'nin ebeveyninin sol çocuğu olduğunu varsayıyoruz. Eğer sağ çocuksa, o zaman 4 ve 5. durumlarda sol ve sağ yer değiştirir.

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

71

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- Durum 4: Ebeveyn P Kırmızı ama Amca U Siyah ve N, P'nin Sağ Çocuğu ve P, G'nin Sol Çocuğu**
- Bu sorunu gidermek için yeni düğüm N ve onun üst düğümü P'nin rollerini değiştirmek için sola dönüş yapılır.
- Döndürmeden sonra, C kodunda N ve P'yi yeniden etiketlediğimizi ve ardından yeni düğümün ebeveyniyle ilgilenmek için case 5'in çağrıdığını unutmayın. Bu, her kırmızı düğümün her iki çocuğunun da siyah olması gerektiğini söyleyen Property 4'ün hala ihlal edilmesi nedeniyle yapılır.
- Şekil 10.58, Durum 4 eklemesini göstermektedir. N'nin P'nin sol çocuğu ve P'nin G'nin sağ çocuğu olması durumunda, bir sağ dönüş gerçekleştirmemiz gerektiğini unutmayın. Durum 4'ü işleyen C kodunda, P ve N'yi kontrol ediyoruz ve sonra, bir sol veya sağ dönüş gerçekleştiriyoruz.

```
void case4(yapı düğümü *n)
{
    yapı düğümü *g = grand_parent(n);
    eğer ((n == n -> ebeveyn -> sağ) && (n -> ebeveyn == g -> sol))
    {
        rotate_left(n -> ebeveyn);
        n = n -> sol;
    }
    aksi takdirde eğer ((n == n -> ebeveyn -> sol) && (n -> ebeveyn == g -> sağ))
    {
        rotate_right(n -> ebeveyn);
        n = n -> sağ;
    }
    durum5(n);
}
```

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

72

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- Vaka 4:

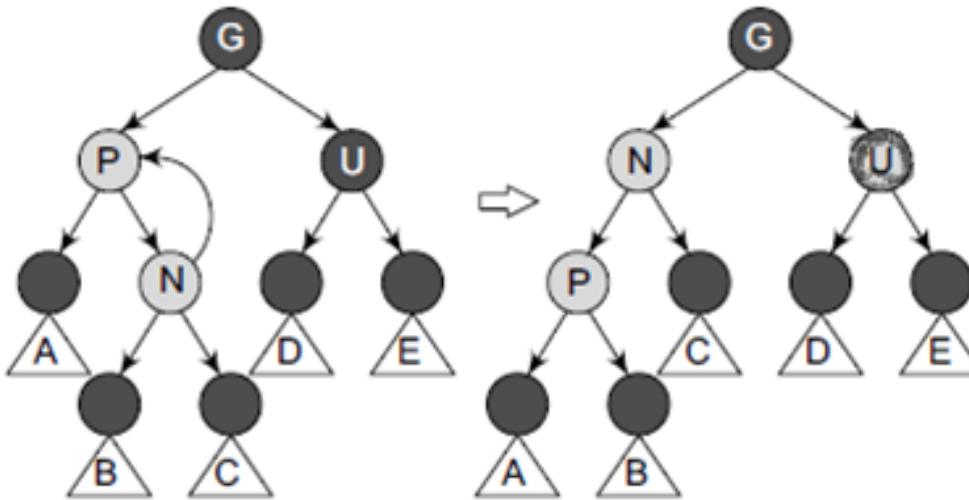


Figure 10.58 Insertion in Case 4

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

73

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- **Durum 5: Ana Düğüm P Kırmızı ama Amca U Siyah ve Yeni Düğüm N, P'nin Sol Çocuğu ve P, Ana Düğüm G'nin Sol Çocuğu.**
- Bu sorunu gidermek için G (N'nin büyük ebeveyni) üzerinde sağa doğru bir rotasyon yapılır.
- Bu rotasyondan sonra eski ebeveyn P artık hem yeni düğüm N'nin hem de eski büyük ebeveyn G'nin ebeveyni.
- G'nin renginin siyah olduğunu biliyoruz (aksi takdirde önceki çocuğu P kırmızı olamazdı), bu yüzden şimdiden P ve G'nin renklerini, ortaya çıkan ağacın her iki çocuğunun da siyah olduğunu belirten Özellik 4'ü sağlaması için değiştirelim.
- N'nin P'nin sağ çocuğu ve P'nin G'nin sağ çocuğu olması durumunda sola dönüş yapacağımızı unutmayın.
- Case 5'i işleyen C kodunda, P ve N'yi kontrol ediyoruz ve ardından sola veya sağa dönüş gerçekleştiriyoruz.

```
void case5(yapı düğümü *n)
{
    yapı düğümü *g;
    g = büyük ebeveyn(n);
    eğer ((n == n -> ebeveyn -> sol) && (n -> ebeveyn == g -> sol))
        sağa_döndür(g);
    aksi takdirde eğer ((n == n -> ebeveyn -> sağ) && (n -> ebeveyn == g -> sağ))
        sola_döndür(g);
    n -> ebeveyn -> renk = SİYAH;
    g -> renk = KIRMIZI;
}
```

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

74

Kırmızı-Siyah Ağacına Bir Düğüm Ekleme

- Vaka 5:

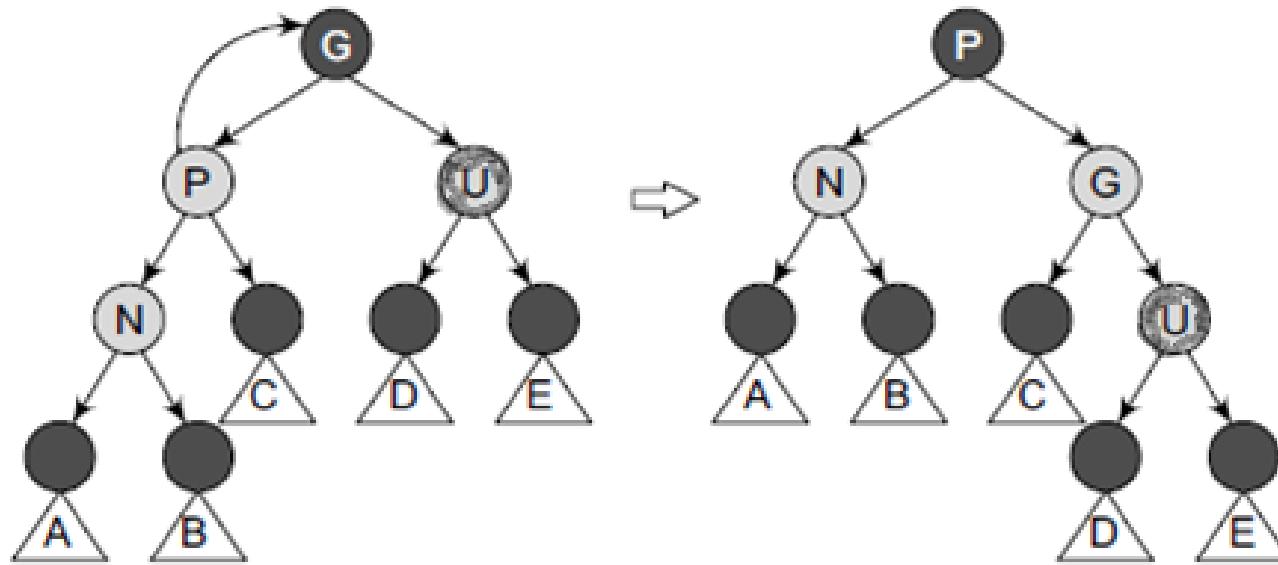


Figure 10.59 Insertion in case 5

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

75

Kırmızı-Siyah Ağaçtan Bir Düğümü Silme

- Kırmızı-siyah ağaçtan bir düğümü silmeye, ikili arama ağacında yaptığımız gibi başla. İkili arama ağacında, iki yaprak olmayan çocuğu olan bir düğümü sildiğimizde, düğüm sol alt ağacındaki maksimum elemanı veya sağ alt ağacındaki minimum elemanı bulu ve değerini silinen düğüme taşıriz.
- Bundan sonra, değeri kopyaladığımız düğümü sileriz. Bu düğümün ikiden az yaprak olmayan çocuğu sahip olması gerektiğini unutmayın. Bu nedenle, yalnızca bir değeri kopyalamak hiçbir kırmızı-siyah özelliğini ihlal etmez, ancak silme sorununu en fazla yaprak olmayan çocuğu sahip bir düğümü silme sorununa indirger.
- Bu bölümde, en fazla bir yaprak olmayan çocuğu olan bir düğümü sildiğimizi varsayıcağız, buna çocuğu diyeceğiz. Bu düğümün her iki yaprak çocuğu varsa, bunlardan birinin çocuğu olmasına izin verin.
- Bir düğümü silerken eğer rengi kırmızı ise onu çocuğu olan siyahla değiştirebiliriz.
- Silinen düğümden geçen tüm yollar, yalnızca bir kırmızı düğüm daha azından geçecektir. Silinen düğümün hem üst düğümü hem de alt düğümü siyah olmalı, böylece hiçbir özelleşme edilmeyecektir.
- Başka bir basit durum ise kırmızı bir çocuğu olan siyah bir düğümü sildiğimiz zaman. Bu durumda, özellik 4 ve özellik 5 ihlal edilebilir, bu yüzden onları geri yüklemek için silinen düğümün çocuğunu siyahla yeniden boyamanız yeterlidir.

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

76

Kırmızı-Siyah Ağaçtan Bir Düğümü Silme

- Ancak, hem silinecek düğüm hem de onun çocuğu siyah olduğunda karmaşık bir durum ortaya çıkar. Bu durumda, silinecek düğümü onun çocuğuyla değiştirerek başlarız.
- C kodunda, çocuk düğümü (yeni pozisyonunda) N olarak ve kardeşini (yeni ebeveyninin diğer çocuğu) S olarak etiketliyoruz.
- Bir düğümün kardeşini bulmak için C kodu aşağıdaki gibi verilebilir:
 `yapı düğümü *kardeş(yapı düğümü *n)`
 {
 `eğer (n == n -> ebeveyn -> sol)`
 `return n -> ebeveyn -> sağ;`
 `başka`
 `return n -> ebeveyn -> sol;`
 }
}

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

77

Kırmızı-Siyah Ağacından Bir Düğümü Silme

- Silme işlemini, replace node fonksiyonunun çocuğu ağaçtaki N'nin yerine koyduğu aşağıdaki kodu kullanarak başlatabiliriz.
- Kolaylık olması açısından, boş yaprakların NULL yerine gerçek düğüm nesneleri tarafından temsil edildiğini varsayıyoruz.

```
void delete_child(struct düğüm *n)
{
    /* N'nin en fazla bir tane boş olmayan çocuğu varsa */
    yapı düğümü *çocuk;
    eğer (yaprak_ise(n -> sağ))
        çocuk = n -> sol;
    başka
        çocuk = n -> sağ;
    replace_node(n, çocuk);
    eğer (n -> renk == SİYAH) {
        eğer (çocuk -> renk == KIRMIZI)
            çocuk -> renk = SİYAH;
        başka
            del_case1(çocuk);
    }
    ücretsiz(n);
}
```

- Hem N hem de onun üst ögesi P siyah olduğunda, P'yi silmek, N'den önce gelen yolların diğer yollardan bir siyah düşüme sahip olmasına neden olur. Bu, Özellik 5'i ihlal eder.
- Bu nedenle ağacın yeniden dengelenmesi gereklidir. Aşağıda tartışılan dikkate alınması gereken birkaç durum

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

Kırmızı-Siyah Ağaçtan Bir Düğümü Silme

- **Durum 1: N Yeni Köktür**
- Bu durumda, her yoldan bir siyah düğümü kaldırılmış olduk ve yeri siyah oldu, dolayısıyla hiçbir özellik ihlal edilmedi.

```
void del_case1(yapı düğümü *n)
{
    eğer (n -> ebeveyn != NULL)
        del_case2(n);
}
```

Not 2, 5 ve 6. durumlarda, N'nin ebeveyni P'nin sol çocuğu olduğunu varsayıyoruz. Eğer sağ çocuksa, bu üç durumda da sol ve sağ yer değiştirmelidir.

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

79

Kırmızı-Siyah Ağacından Bir Düğümü Silme

- **Vaka 2: Kardeş S Kırmızı**
- Bu durumda P ve S'nin renklerini değiştirin ve ardından P'de sola döndürün. Ortaya çıkan ağaçta N'nin büyük ebeveyni olacaktır.
- Şekil 10.60, Durum 2'nin silinmesini göstermektedir.
- 2. durum silme işlemini gerçekleştiren C kodu aşağıdaki gibi verilebilir:

```
void del_case2( yapı düğümü *n)
{
    yapı düğümü *s;
    s = kardeş(n);
    eğer (s -> renk == KIRMIZI)
    {
        eğer (n == n -> ebeveyn -> sol)
            rotate_left(n -> ebeveyn);
        başka
            rotate_right(n -> ebeveyn);
        n -> ebeveyn -> renk = KIRMIZI;
        s -> renk = SİYAH;
    }
    del_case3(n);
}
```

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

80

Kırmızı-Siyah Ağacından Bir Düğümü Silme

- Vaka 2: Kardeş S Kırmızı

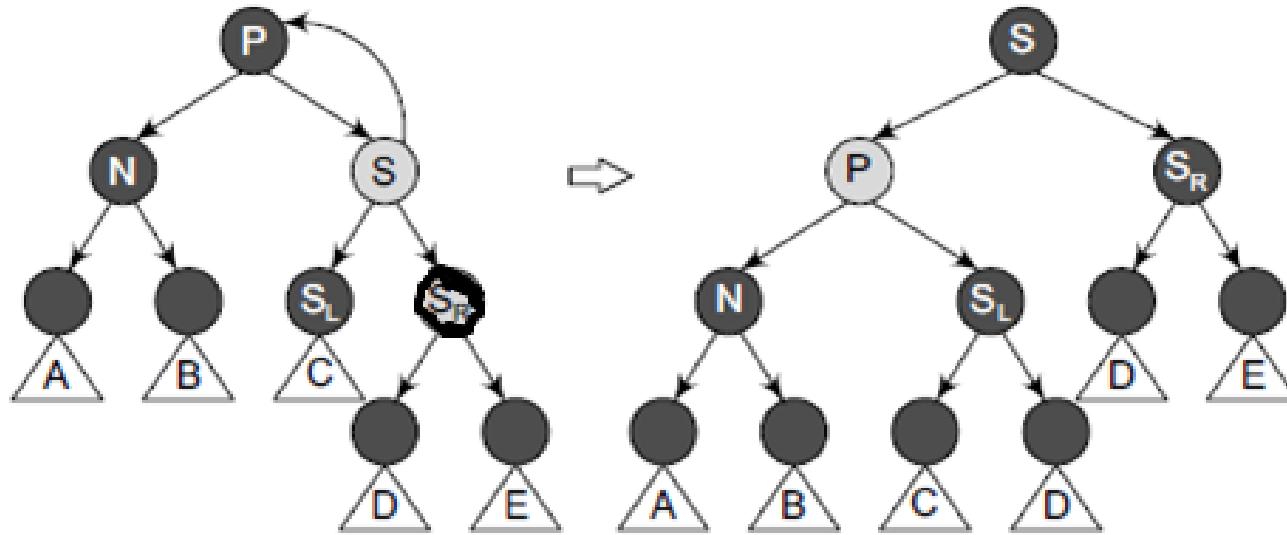


Figure 10.60 Deletion in case 2

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

81

Kırmızı-Siyah Ağacından Bir Düğümü Silme

- **Vaka 3: P, S ve S'nin Çocukları Siyah**
- Bu durumda, S'yi kırmızıyla yeniden boyayın. Ortaya çıkan ağaçta, S'den geçen tüm yolların bir siyah düğümü daha az olacaktır.
- Dolayısıyla, P'den geçen tüm yollar, P'den geçmeyen yollardan bir tane daha az siyah düşüme sağlıduğundan, Özellik 5 hala ihlal edilmektedir.
- Bu sorunu gidermek için, Durum 1'den başlayarak P üzerinde yeniden dengeleme prosedürüne gerçekleştirelim. Durum 3, Şekil 10.61'de gösterilmektedir.
- Durum 3 için C kodu aşağıdaki gibi verilebilir:

```
void del_case3(yapı düğümü *n)
{
    yapı düğümü *s;
    s = kardeş(n);
    eğer ((n -> ebeveyn -> renk == SİYAH) && (s -> renk == SİYAH) && (s -> sol -> renk ==
SİYAH) &&
        (s -> sağ -> renk == SİYAH))
    {
        s -> renk = KIRMIZI;
        del_case1(n -> ebeveyn);
    } başka
    del_case4(n);
}
```

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

82

Kırmızı-Siyah Ağacından Bir Düğümü Silme

- Vaka 3: P , S ve S' nin Çocukları Siyah

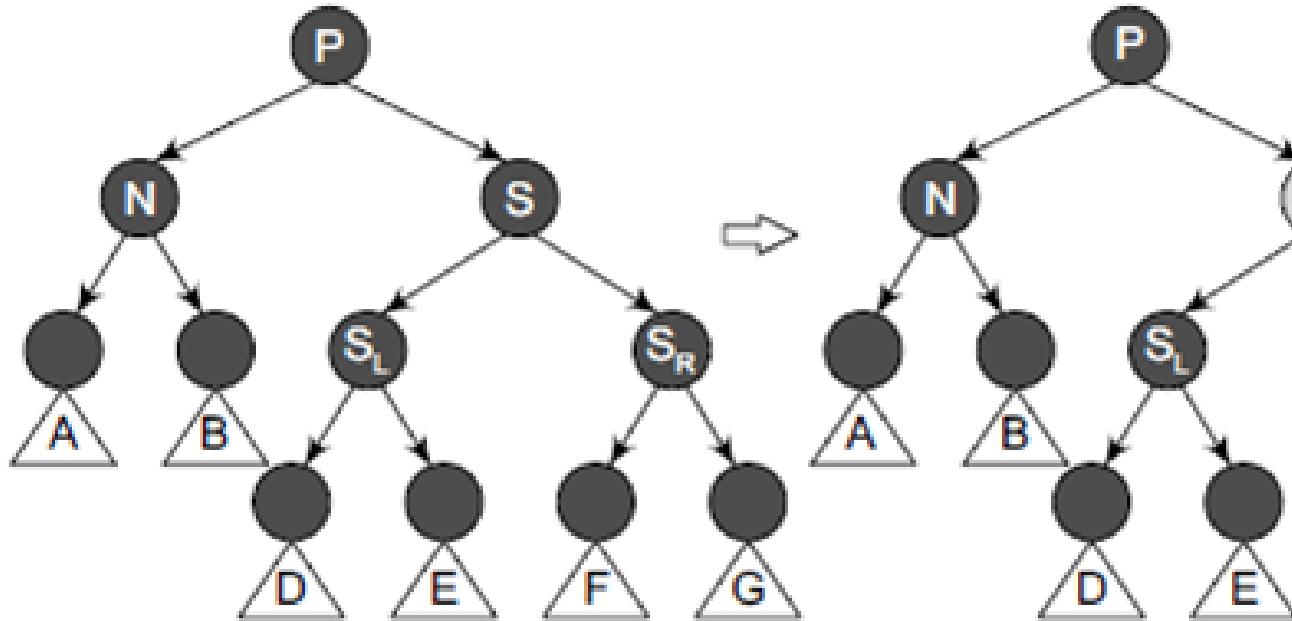


Figure 10.61 Deletion in case 3

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

Kırmızı-Siyah Ağaçtan Bir Düğümü Silme

- **Vaka 4: S ve S'nin Çocukları Siyah, Ancak P Kırmızı**
- Bu durumda S ve P renklerini yer değiştiriyoruz.
- Bu durum S'den geçen yollardaki siyah düğüm sayısını etkilemeyecek olsa da, N'den geçen yollara bir siyah düğüm ekleyecek ve bu yollardaki silinen siyah düğümü telafi edecektir.
- Case 4'ü ele alan C kodu şu şekildedir:

```
void del_case4(yapı düğümü *n)
{
    yapı düğümü *s;
    s = kardeş(n);
    eğer((n->üst -> renk == KIRMIZI) & & (s -> renk == SİYAH) & & (s -> sol ->
renk == SİYAH) && (s -> sağ -> renk == SİYAH))
    {
        s -> renk = KIRMIZI;
        n -> ebeveyn -> renk = SİYAH;
    } başka
    del_case5(n);
}
```

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

84

Kırmızı-Siyah Ağacından Bir Düğümü Silme

- Vaka 4: S ve S'nin Çocukları Siyah, Ancak P Kırmızı

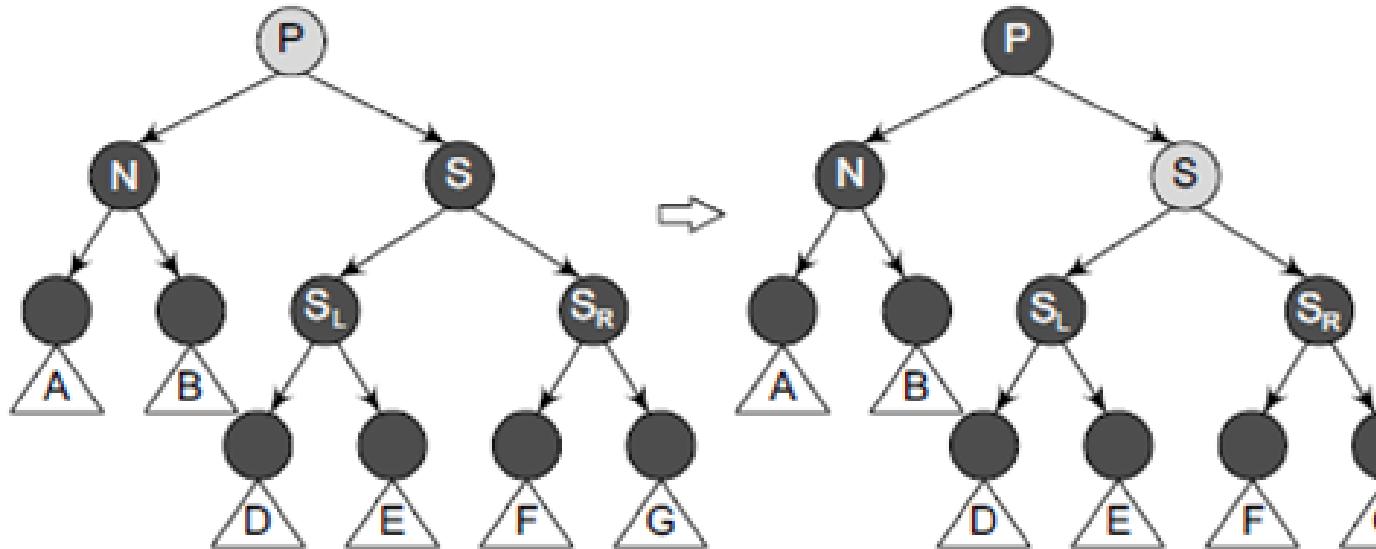


Figure 10.62 Deletion in case 4

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

85

Kırmızı-Siyah Ağaçtan Bir Düğümü Silme

- Durum 5: N, P'nin Sol Çocuğu ve S Siyahıtır, S'nin Sol Çocuğu Kırmızıdır, S'nin Sağ Çocuğu Siyahıtır.**
- Bu durumda, S'de sağ rotasyon gerçekleştirin. Rotasyonda sonra, S'nin sol çocuğu S'nin ebeveyni ve N'nin yeni kardeşi olur. Ayrıca, S'nin ve yeni ebeveyninin renklerini değiştirin.
- Şimdi tüm yolların hala eşit sayıda siyah düşüme sahip olduğunu, ancak N'nin sağ çocuğu kırmızı olan siyah bir kardeşi olduğunu, dolayısıyla Durum 6'ya girdiğimizi unutmayın.

Kırmızı-Siyah Ağaçtan Bir Düğümü Silme

- Durum 5: N, P'nin Sol Çocuğu ve S Siyahtır, S'nin Sol Çocuğu Kırmızıdır, S'nin Sağ Çocuğu Siyahtır.

- 5. durumu ele alan C kodu aşağıdaki şekilde verilmiştir:
void del_case5(yapı düğümü *n)

{

yapı düğümü *s;

s = kardeş(n);

eğer (s -> renk == SİYAH)

{

/* Aşağıdaki kod, 6. durumda doğru bir şekilde çalıştırılabilmesi için kırmızı rengin ebeveyni solunda veya sağında olmasını zorlar. */

eğer ((n == n -> ebeveyn -> sol) && (s -> sağ -> renk == SİYAH) && (s -> sol -> renk == KIRMIZI))

sağa_döndür(ler);

aksi takdirde eğer ((n == n -> ebeveyn -> sağ) && (s -> sol -> renk == SİYAH) && (s -> sağ -> renk == KIRMIZI))

sola_döndür(ler);

s -> renk = KIRMIZI;

s -> sağ -> renk = SİYAH;

}

del_case6(n);

}

Kırmızı-Siyah Ağaçlar Üzerindeki Operasyonlar

Kırmızı-Siyah Ağacından Bir Düğümü Silme

- Durum 5: N, P'nin Sol Çocuğu ve S Siyahıtır, S'nin Sol Çocuğu Kırmızıdır, S'nin Sağ Çocuğu Siyahıtır.

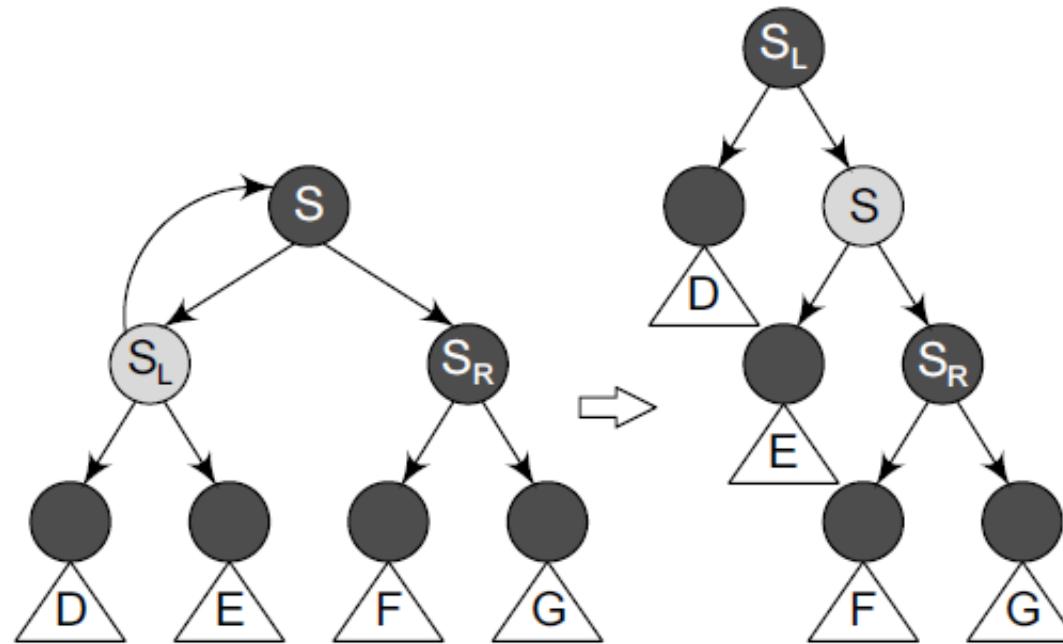


Figure 10.63 Deletion in case 5

Kırmızı-Siyah Ağaç⁸⁸lar Üzerindeki Operasyonlar

Kırmızı-Siyah Ağacından Bir Düğümü Silme

- **Durum 6: S Siyah, S'nin Sağ Çocuğu Kırmızı ve N Ebeveyni P'nin Sol Çocuğu**
- Bu durumda, P'de bir sol rotasyon yapılır ve S, P'nin ve S'nin sağ çocuğunun ebeveyni olur. Rotasyondan sonra, P ve S'nin renkleri değiştirilir ve S'nin sağ çocuğu siyaha boyanır. Bu adımla takip edildiğinde, özellik 4 ve özellik 5'in geçerli kaldığını göreceksiniz.
- Case 6'yı düzeltmek için C kodu aşağıdaki gibi verilebilir:

```
void del_case6( yapı düğümü *n)
{
    yapı düğümü *s;
    s = kardeş(n);
    s -> renk = n -> ebeveyn -> renk;
    n -> ebeveyn -> renk = SİYAH;
    eğer (n == n -> ebeveyn -> sol) {
        s -> sağ -> renk = SİYAH;
        rotate_left(n -> ebeveyn);
    } başka {
        s -> sol -> renk = SİYAH;
        rotate_right(n -> ebeveyn);
    }
}
```

Kırmızı-Siyah Ağaçlar Üzerindeki⁸⁹ Operasyonlar

Kırmızı-Siyah Ağaçtan Bir Düğümü Silme

- Durum 6: S Siyah, S'nin Sağ Çocuğu Kırmızı ve N Ebeveyni P'nin Sol Çocuğu

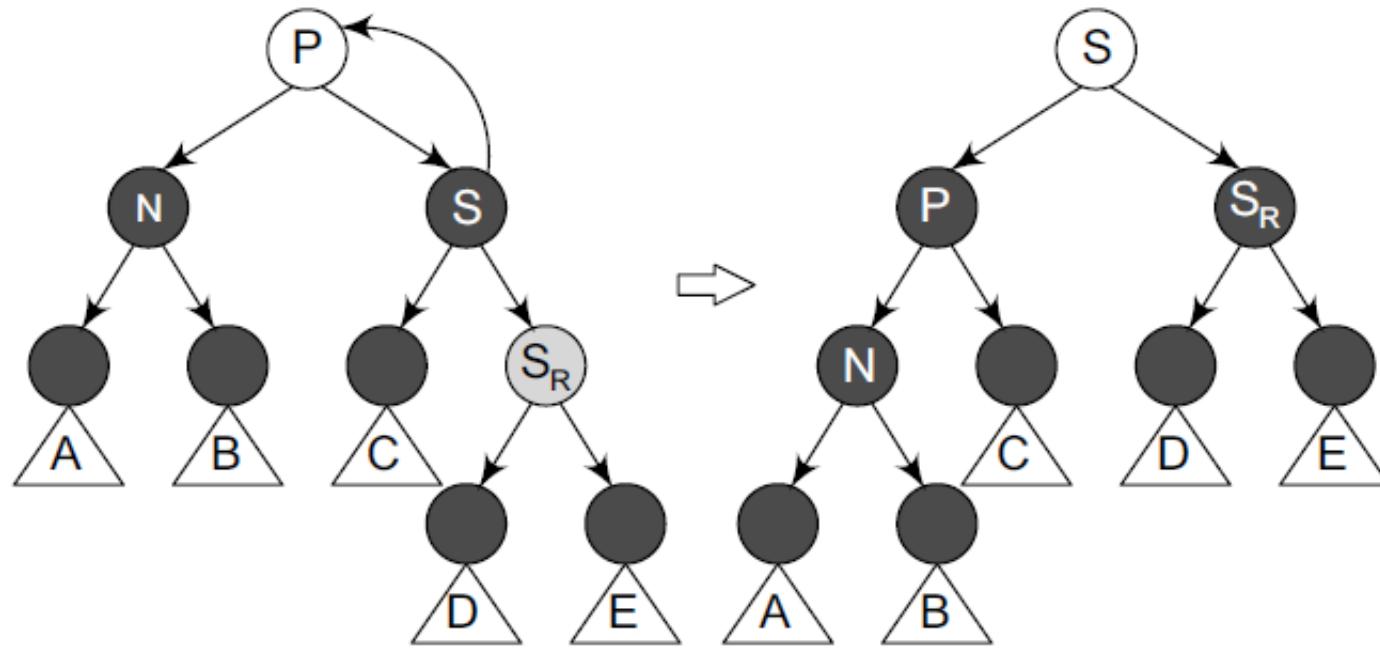


Figure 10.64 Deletion in case 6

Yayılmış Ağaçlar

90

- Splay ağaçları Daniel Sleator ve Robert Tarjan tarafından icat edilmiştir. Splay ağaç yakın zamanda erişilen öğelere hızlı bir şekilde yeniden erişilebilmesi gibi ek bir özellik sahip, kendi kendini dengeleyen bir ikili arama ağacıdır.
- Temel ekleme, arama ve silme işlemlerini $O(\log(n))$ amortize sürede gerçekleştirdiği verimli bir ikili ağaç olduğu söylenmektedir.
- Birçok tekdüze olmayan işlem dizisi için, dizinin belirli deseni bilinmese bile, yayılma ağaçları diğer arama ağaçlarından daha iyi performans gösterir.
- Bir splay ağaç, ek alanları olmayan ikili bir ağaçtan oluşur. Bir splay ağacındaki bir düğüme erişildiğinde, köke döndürülür veya 'splay' edilir, böylece ağacın yapısı değişir.
- En sık erişilen düğüm her zaman aramanın başlangıç noktasına (veya kök düğüme) yakın taşındığından, bu düğümler daha hızlı bulunur.
- Bunun ardından basit fikir, bir öğeye erişildiğinde, ona tekrar erişilme olasılığının da yüksek olmasıdır.
- Bir yayılma ağacında, ekleme, arama ve silme gibi işlemler yayılma adı verilen tek bir temel işlemle birleştirilir.
- Ağacı *belirli bir düğüm için yaymak, ağacı o düğümü köke yerlestirecek şekilde yeniden düzenler.*
- Bunu yapmanın bir tekniği, önce o düğüm için standart bir ikili ağaç araması yapmak ardından düğümü en üste getirmek için belirli bir sırayla dönüşler kullanmaktır.

Splay Trees Üzerindeki İşlemler⁹¹

- Bu bölümde, bir splay ağacında gerçekleştirilen dört ana işlemi ele alacağız. Bunlar arasında splaying, insertion, search ve deletion bulunur.
- **Yayılma**
- Bir N düğümüne eriştiğimizde, köke taşımak için N üzerinde yayılma işlemi gerçekleştirilir. Bir yayılma işlemi gerçekleştirmek için, her adımda N'yi köke yaklaştıran belirli yayılma adımları gerçekleştirilir.
- *Her erişimden sonra ilgi duyulan belirli bir düğümün yayılması, yakın zamanda erişilen düğümlerin köke daha yakın tutulmasını ve ağacın kabaca dengede kalmasını sağlar, böylece istenen amortize edilmiş zaman sınırları elde edilebilir.*
- Her yayılma adımı üç faktöre bağlıdır:
 - N, ebeveyni P'nin sol veya sağ çocuğu olsun,
 - P'nin kök olup olmadığı ve eğer değilse,
 - P, ebeveyni G'nin (N'nin büyük ebeveyni) sol çocuğu mu yoksa sağ çocuğu mu?
- Bu üç faktöre bağlı olarak her faktöre bağlı bir yayılma adımıımız var.

Splay Trees Üzerindeki İşlemler⁹²

- **Zig adım**
- Zig işlemi, P'nin (N'nin ebeveyni) splay ağacının kökü olması durumunda yapılır.
- Zig-to-step'te ağaç N ve P arasındaki kenarda döndürülür.
- Zig adımı genellikle bir yayılma işleminin son adımı olarak ve yalnızca N'nin işlemin başlangıcında tek bir derinliğe sahip olma durumunda gerçekleştirilir. Şekil 10.65'e bakın.

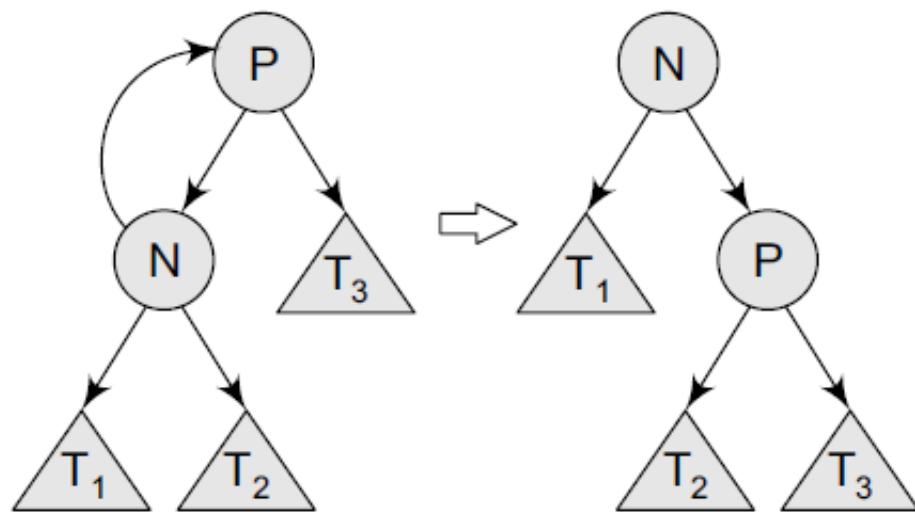
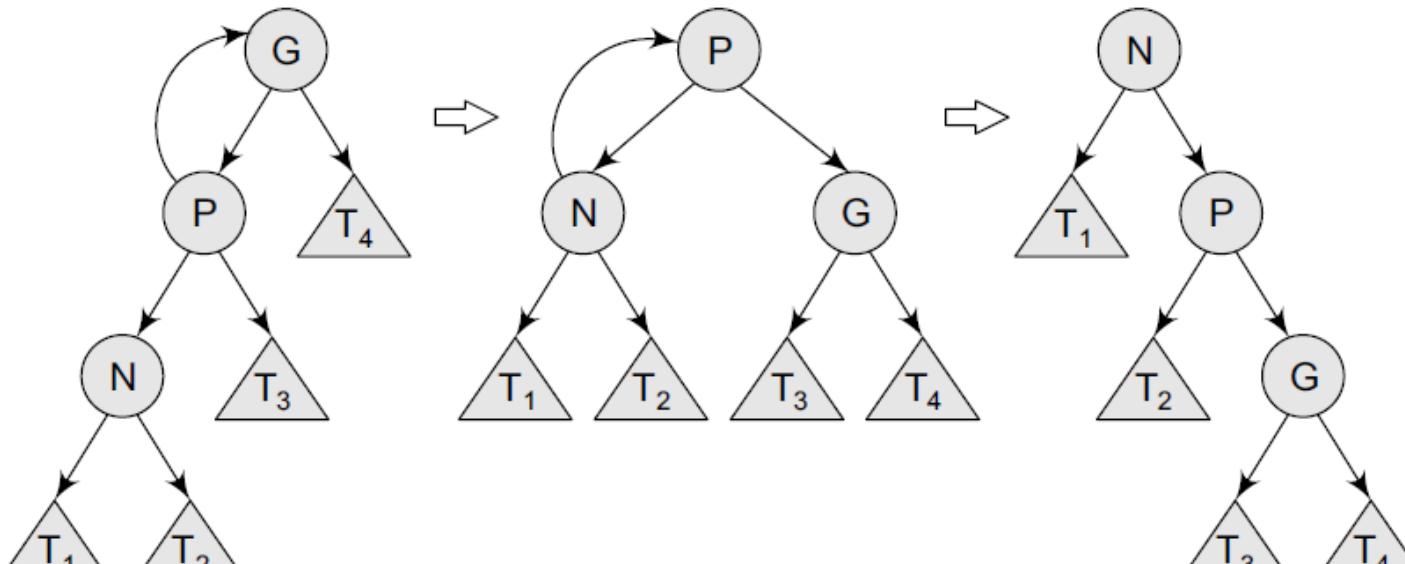


Figure 10.65 Zig step

Splay Trees Üzerindeki İşlemler

93

- **Zig-zig adım**
- Zig-zig işlemi, P kök olmadığında gerçekleştirilir.
- Ayrıca N ve P'nin her ikisi de anne ve babalarının sağ veya sol çocuğuudur.
- Şekil 10.66, N ve P'nin sol çocukları olduğu durumu göstermektedir.
- Zig-zig admiminda, ağaç önce P ve ana ağıacı G'yi birleştirken kenarda döndürülür ve ardından N ve P'yi birleştirken kenarda tekrar döndürülü



Splay Trees Üzerindeki İşlemler⁹⁴

- **Zig-zag adımı**
- Zig-zag işlemi, P kök olmadığında gerçekleştirilir. Buna ek olarak, N, P'nin sağ çocuğu ve P, G'nin sol çocuğu veya tam tersi.
- Zig-zag adımda, ağaç önce N ve P arasındaki kenarda döndürülür ve sonra N ve G arasındaki kenarda döndürülür Şekil 10.67'ye bakın.

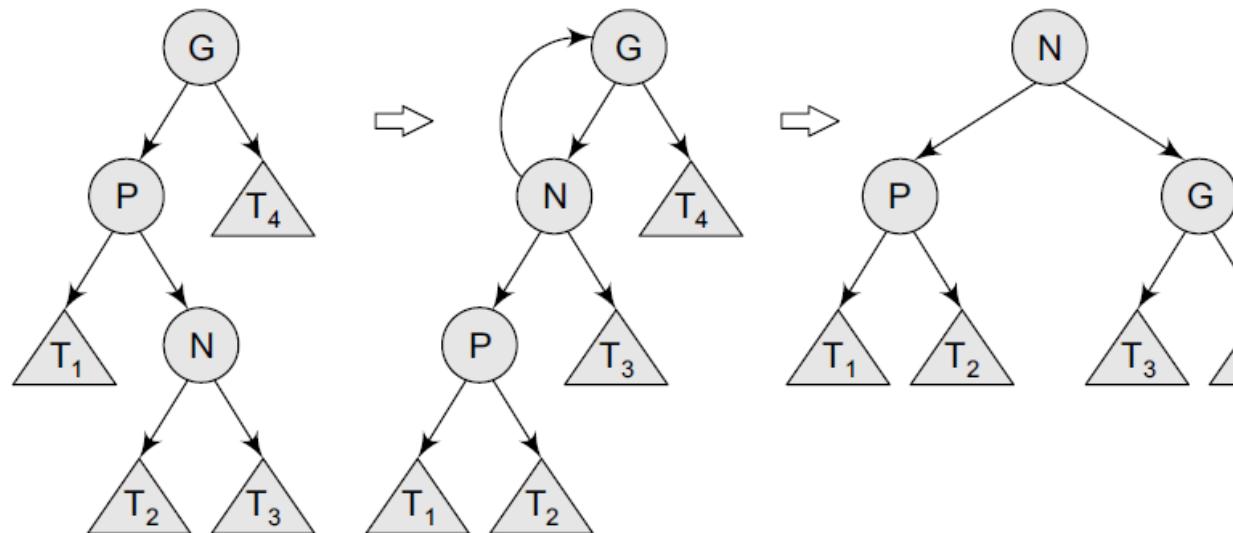


Figure 10.67 Zig-zag step

Splay Trees Üzerindeki İşlemler⁹⁵

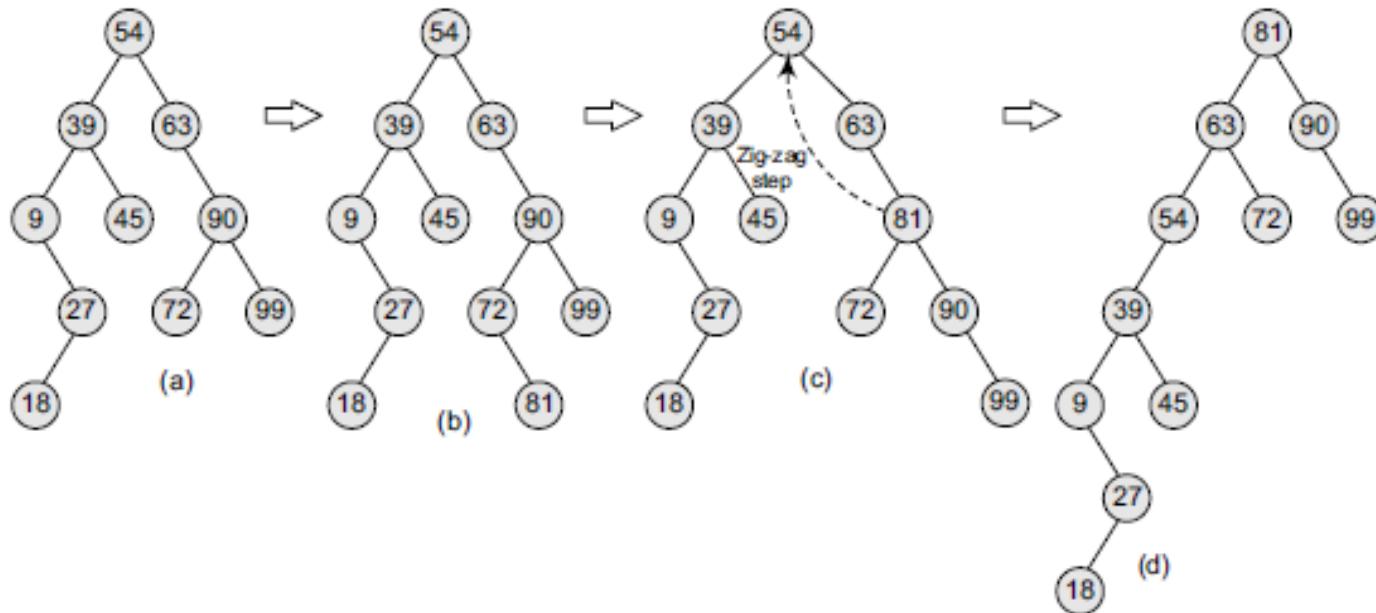
- **Bir Splay Ağacına Düğüm Ekleme**
- Yeni bir düğüm olan N'yi splay ağacına ekleme süreci, ikili arama ağacına düğüm ekleme süreciyle aynı şekilde başlar da, ekleme işleminden sonra N, splay ağacının yeni kökü haline gelir.
- Bir splay ağacına yeni bir N düğümü eklemek için gerçekleştirilen adımlar aşağıdaki gibi verilebilir:
 - Adım 1 Splay ağacında N'yi arayın. Arama başarılı olursa, N düğümünde splay yapın.
 - Adım 2 Arama başarısız olursa, arama sırasında ulaşılan NULL işaretçisini yeni bir N düğümüne işaret eden bir işaretçiyle değiştirecek şekilde yeni düğüm N'yi ekleyin. Ağacı N'de yayın.

Splay Trees Üzerindeki İşlemler⁹⁶

- *Bir Splay Ağacına Düğüm Ekleme*

Example 10.11 Consider the splay tree given on the left. Observe the change in its structure when 81 is added to it.

Solution



Note

To get the final splay tree, first apply zig-zag step on 81. Then apply zig-zag step to make 81 the root node.

Splay Trees Üzerindeki İşlemler⁹⁷

- **Splay Tree'de Bir Düğüm Arama**
- Eğer splay ağacında belirli bir düğüm N varsa, o zaman N 'yı bir işaretçi döndürür; aksi takdirde boş düğüme bir işaretçi döndürür.
- Bir splay ağacında N düğümünü aramak için gerçekleştirilecek adımlar şunlardır:
 - Yayılan ağacın köküne doğru N 'yi arayın.
 - Eğer arama başarılı olursa ve N 'ye ulaşırsak, ağacı N 'de yayar ve N 'ye bir işaretçi döndürürüz.
 - Arama başarısız olursa, yani splay ağacı N içermiyorsa, o zaman boş bir düğüme ulaşırız. Arama sırasında ulaşılan son boş olmayan düğümde ağacı splay edin ve boş bir işaretçi döndürü.

Splay Trees Üzerindeki İşlemler⁹⁸

- **Splay Tree'de Bir Düğüm Arama**

Example 10.12 Consider the splay tree given in Fig. 10.68. Observe the change in its structure when a node containing 81 is searched in the tree.

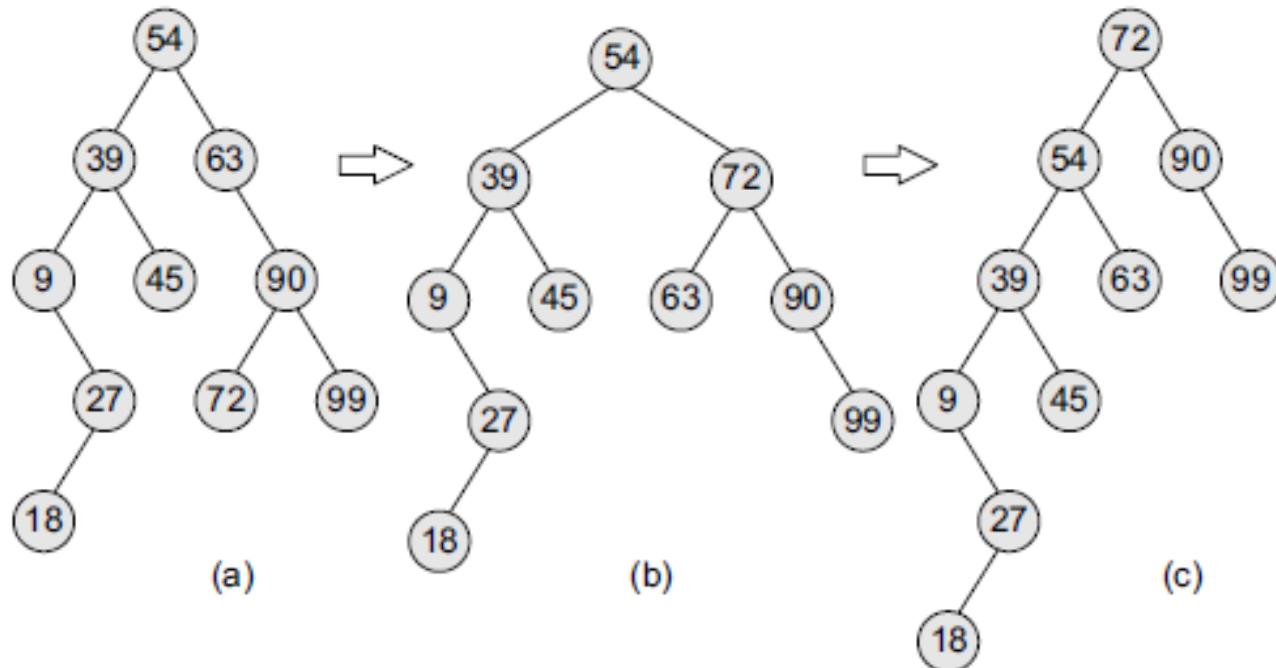


Figure 10.69 Splay tree

Splay Trees Üzerindeki İşlemler⁹⁹

- **Bir Splay Ağacından Bir Düğümü Silme**
- Bir splay ağacından N düğümünü silmek için aşağıdaki adımları gerçekleştiririz:
 - Silinmesi gereken N'yi arayın. Arama başarısız olursa, arama sırasında karşılaşılan son boş olmayan düğümde ağacı yayın.
 - Arama başarılıysa ve N kök düğüm değilse, o zaman P'nin N'ni ebeveyni olmasına izin verin. N'yi P'nin uygun bir alt ögesiyle değiştirin (ikili arama ağacında yaptığımız gibi). Son olarak ağacı P'de yayın.

Splay Trees Üzerindeki İşlemler

- *Bir Splay Ağacından Bir Düğümü Silme*

Example 10.13 Consider the splay tree at the left. When we delete node 39 from it, the new structure of the tree can be given as shown in the right side of Fig. 10.70(a).

After splaying the tree at P, the resultant tree will be as shown in Fig. 10.70(b):

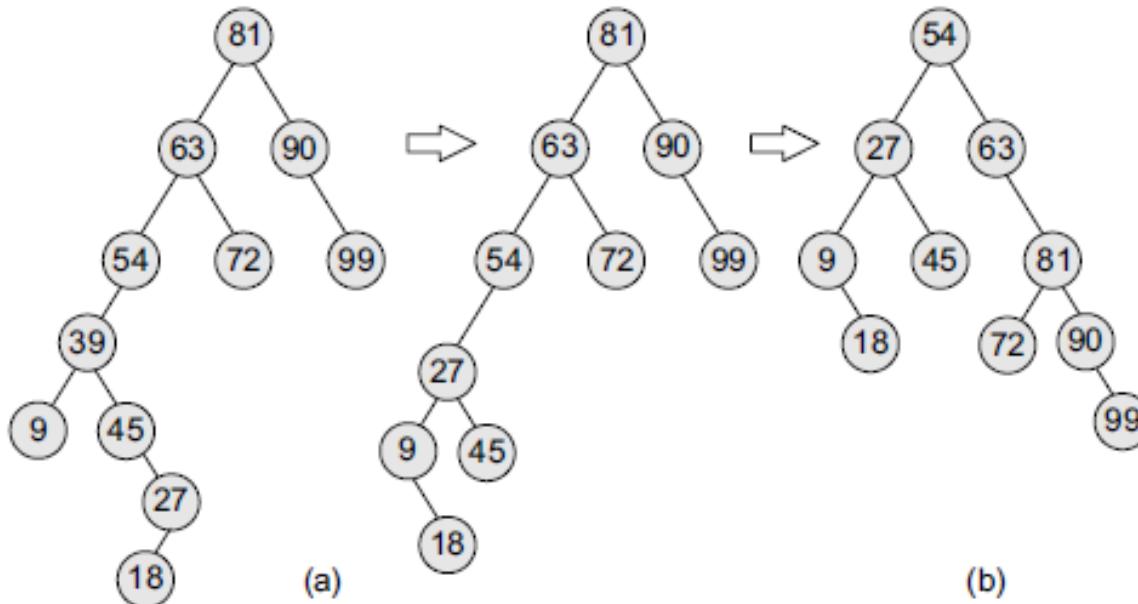


Figure 10.70 (a) Splay tree (b) Splay tree

Yayvan Ağaçlarının Avantajları ve Dezavantajları

- Yayılmış ağaç kullanmanın avantajları şunlardır:
 - Bir splay ağaçtı, arama, ekleme ve silme işlemleri için iyi performans sağlar. Bu avantaj, splay ağacının, sık erişilen düğümlerin köke daha yakın bir yere taşınarak hızlı bir şekilde erişilebildiği, kendi kendini dengeleyen ve kendi kendini iyileştiren bir veri yapısı olması gerçeğine dayanır. Bu avantaj, önbellekleri ve çöp toplama algoritmalarını uygulamak için özellikle yararlıdır.
 - Splay ağaçları, kırmızı-siyah ağaçlar veya AVL ağaçları gibi diğer kendi kendini dengeleyen ikili arama ağaçlarına kıyasla uygulanma ölçüde daha basittir ve ortalama durum performansları da derecede verimlidir.
 - Splay ağaçları herhangi bir muhasebe verisi depolamadığından bel gereksinimlerini en aza indirir.
 - Diğer kendi kendini dengeleyen ağaç türlerinin aksine, splay ağaçları aynı anahtarları içeren düğümlerle iyi bir performans (amortize edilmiş $O(\log n)$) sağlar.

Yayvan Ağaçlarının Avantajları ve Dezavantajları

- Ancak, geniş yapraklı ağaçların dezavantajları şunlardır:
 - Bir ağacın tüm düğümlerine sıralı bir düzende erişirken, ortaya çıkan ağaç tamamen dengesiz hale gelir. Bu, her erişimin $O(\log n)$ zaman aldığı ağaca n erişim gerektirir. Örneğin, ilk düğüme yeniden erişim ilk düğümü döndürmeden önce ağacı yeniden dengelemek için $O(n)$ işlem gerektiren bir işlemi tetikler. Bu, son işlem için önemli bir gecikme yaratrsa da, tüm dizi boyunca amortize edilmiş performans hala $O(\log n)$ 'dir.
 - Tekdüze erişim için, bir splay ağacının performansı, bir şekilde dengeli basit bir ikili arama ağacından önemli ölçüde daha kötü olacaktır. Tekdüze erişim için, splay ağaçlarının aksine, bu diğer veri yapıları en kötü durum zaman garantileri sağlar ve kullanımı daha verimli olabilir.