9.3 İkili Arama Ağaçları (BSTs - Binary Search Trees)

İkili ağaç yapısının önemli bir özelliği de düğümlerin yerleştirilmesi sırasında uygulanan işlemlerdir. İkili bir ağaçta çocuklar ve ebeveyn arasında büyüklük ya da küçüklük gibi bir ilişki yoktur. Her çocuk ebeveyninden küçük veya büyük ya da ebeveynine eşit olabilir. İkili arama ağaçlarındaki (**BST - Binary Search Tree**) durum ise farklıdır. Bir ikili arama ağacındaki her düğüm, sol alt ağacındaki tüm değerlerden büyük, sağ alt ağacındaki tüm değerlerden küçük ya da eşittir. İkili arama ağaçlarında her bir düğümün alt ağacı yine bir ikili arama ağacıdır. inorder sıralama yapıldığında küçükten büyüğe veriler elde edilir.

İkili arama ağaçlarında (**BST**) aynı değerlere sahip düğümlerin eklenip eklenmeyeceği sıkça sorulan bir sorudur. Bazı kaynaklarda eklenmesinin veri tekrarı açısından uygun olmayacağı belirtilmektedir. Ayrıca boş bir *BST* ağacına 7 defa 10 değerine sahip düğümün eklendiğini varsayarsak ağacın yüksekliğinin 6 olmasını gerektirir. Bu da ağacın dengesiz olmasına neden olur. Bununla birlikte *BST* ağaçları zaten dengeyi korumamaktadır. Bunun için *AVL* ve *Kırmızı-Siyah ağaçlar* gibi veri yapıları önerilmektedir.

Verinin bütünlüğünün korunması açısından ise aynı değerler eklenmelidir. Burada ise şöyle bir sorun ortaya çıkmaktadır. Tekrarlı değerlere sahip bir *BST* ağacından tekrarı olan bir değeri silmek istediğimizde hangisini sileceğiz? Ya da arama yapmak istediğimizde hangisini bulacağız?

Introduction to Algorithms, 3. baskı kitabında bir *BST* ağacındaki eşit değerlere sahip düğümleri bir bağlı liste ile gösterilmesi önerilmektedir. Ancak bu da veri yapısında fazladan bir alan demektir.

Biz bu derste aşağıdaki algoritmada gösterildiği üzere eşit değerleri ağacın sağ tarafına ekleyeceğiz.

9.3.1 İkili Arama Ağacına Veri Eklemek

İkili bir arama ağacında sol çocuğun verisi ebeveyninin datasından küçük olmalı ve sağ çocuğun verisi de ebeveyninin datasından büyük ya da eşit olmalıdır. Altta `insertBST` isimli ekleme fonksiyonunun kodu görülüyor. Fonksiyon, veri eklendikten sonra ağacın kök adresiyle geri dönmektedir.

```
BTREE* insertBST(BTREE *root, int data) {
```

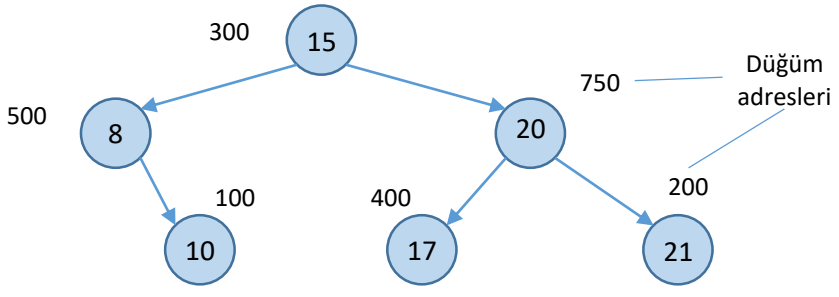
```
    if(root != NULL){
        if(data < root -> data) // data, root'un datasından küçükse
            root -> left = insert(root -> left, data);
        else // data, root'un datasından büyük ya da eşitse
            root -> right = insert(root -> right, data);
    }
    else
        root = new_node(data);
    return root;
}
```

9.3.2 Bir Ağacın Düğümlerinin Sayısını Bulmak

`Inorder` sıralama biçimine biraz benzemektedir. Sıralamada önce sol taraf yazdırılıyor, ardından kök ve sonra da sağ taraf yazdırılarak işlem tamamlanıyordu. Bu benzetimle ağaç üzerinde dolaşılarak tüm düğümlerin sayısı bulunabilir. Fonksiyon rekürsif olarak modellendiğinde algoritma daha kolay bir şekil almaktadır. Geri dönüş değeri tamsayı olacağı için `int` türden olmalı, input parametresi olarak ağacın kökünü almalıdır. Rekürsif fonksiyonlarda mutlaka bir çıkış yolu vardır ve çıkış yolu olarak `if-else` bloğunun kullanıldığından bahsetmiştik. Şimdi bir ağacın düğümlerinin sayısını veren `size` isimli fonksiyonu yazalım.

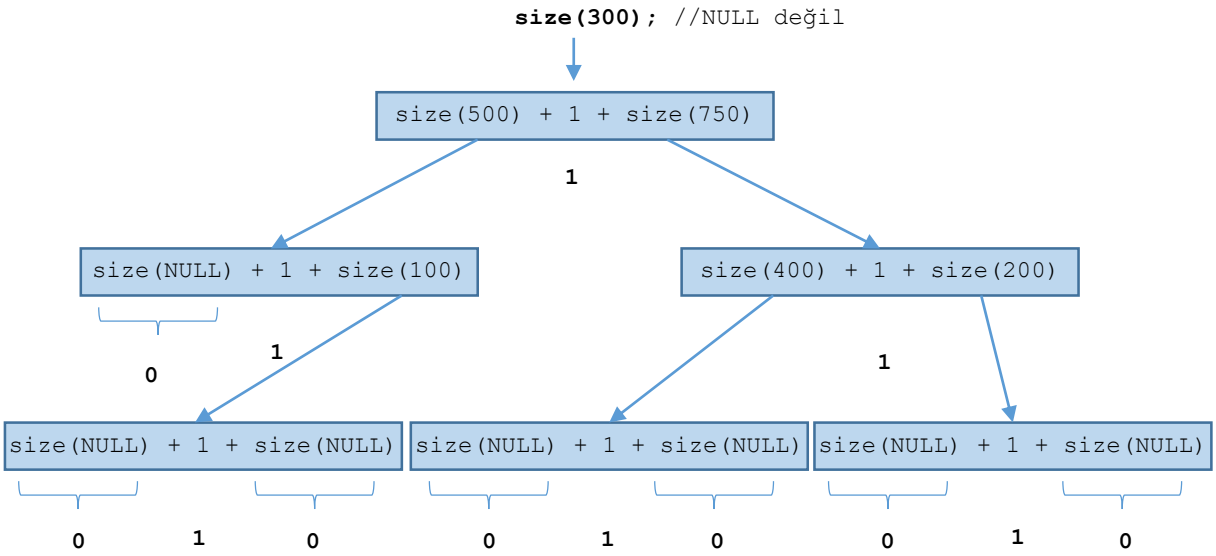
```
/* İkili bir ağacın düğüm sayılarını veren fonksiyon */
int size(BTREE *root) {
    if(root == NULL)
        return 0;
    else
        return size(root -> left) + 1 + size(root -> right);
}
```

Fonksiyonun nasıl çalıştığını inceleyelim.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-1. 6 düğümü olan ikili bir arama ağacı

Aşağıda Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-2'de de açıkça görüldüğü gibi `size(300)` çağrısıyla `root` `NULL` olmadığı için fonksiyonun `else` kısmı çalışıyor ve ilk olarak `size(root->left)` ile sol `descendant` dolaşılıyor. Fonksiyondan geri dönen `0` ve `1` rakamlarının `stack`'te tutulduğu daha önceki konularda anlatılmıştı. Sonra da `size(root->right)` ile fonksiyonlar tekrar çağrılıyor ve sağ `descendant` dolaşılıyor. **LIFO** (*Last in First Out*) son giren ilk çıkar prensibine göre `stack`'te tutulan rakamlar çıkarılacak ve toplanarak ağacın düğüm sayısı bulunacaktır.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-2. size() fonksiyonunun çalışması

9.3.3 Bir Ağacın Yüksekliğini Bulmak

Ağacın yüksekliği bir tamsayıdır, dolayısıyla fonksiyonun geri dönüş türü `int` olmalıdır. Parametre olarak yüksekliği bulunacak olan ağacın adresini almakta ve fonksiyon rekürsif olarak kendini çağırmaktadır.

```
int height(BTREE *root) {
     if(root == NULL)
          return -1;
     else {
          int left_height, right_height;
          left_height = height(root -> left);
          right_height = height(root -> right);
          if(right_height > left_height)
              return right_height + 1;
          else
              return left_height + 1;
```

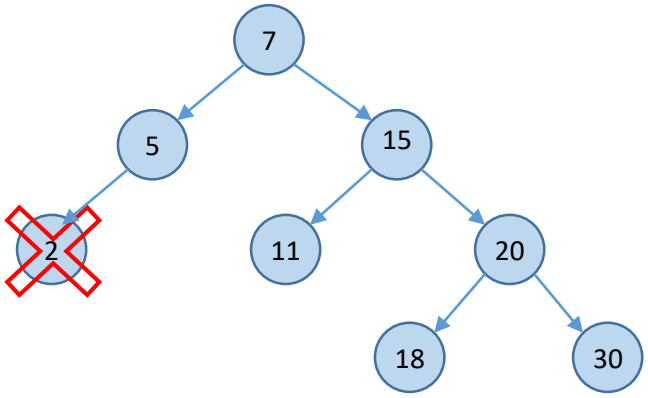
```
        }
    }
```

9.3.4 İkili Arama Ağacından Bir Düğüm Silmek

Bir düğümü silmek istediğimizde ilk olarak fonksiyona parametre olarak gönderdiğimiz ağaçta eleman ya vardır veya yoktur. Eğer ağaçta hiç eleman yoksa fonksiyon `NULL` ile geri dönmelidir. Ağaçta eleman var ise, silme işlemi ancak aranılan değere bir eşitlik varsa gerçekleşecektir. Yani bir ağaçtan 8'i silmek istiyorsak, ancak o ağaçta 8 içeren bir düğüm varsa silme işlemi gerçekleşir. Fakat silinecek düğüm ağacın çocuklarından herhangi biri olabilir. Bu durumda silinecek düğüm, üzerinde bulunduğumuz düğümden küçükse, sol tarafa doğru bir düğüm ilerleyip yeni bir kontrol yapmamız gerekir. Büyükse bu defa sağ tarafta bir düğüm ilerleyip kontrolü tekrarlamamız gerekmektedir. Aranılan düğüm bulunduğunda da silme işleminden önce üç durumdan bahsetmeliyiz. İkili bir arama ağacında bir düğümün en fazla iki çocuğu olabileceğini düşündüğümüzde bunlar;

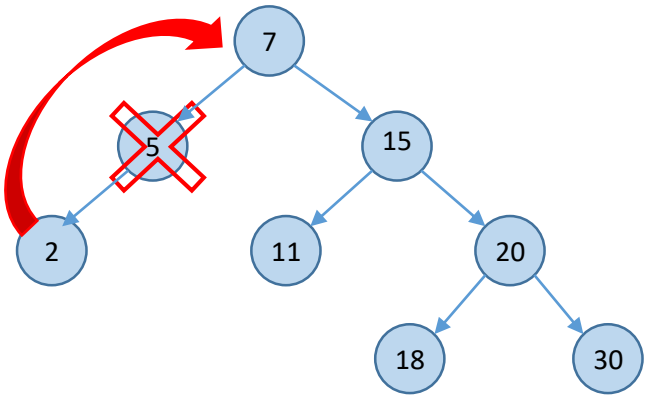
1- Silinecek olan düğümün çocuğu yok ise,
2- Silinecek olan düğümün sadece bir çocuğu var ise,
3- Silinecek olan düğümün iki çocuğu var ise,

Nasıl bir algoritma kurmamız gerektiğidir. Bir örnek ile gösterelim;



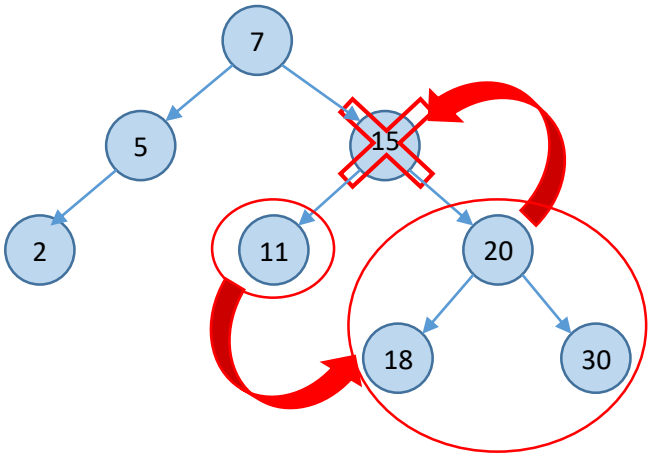
Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-3. İkili arama ağacından bir yaprağın silinmesi.

Diyelim ki silinecek olan veriler yapraklardan (`leaf`) biri olsun. Yani `2, 11, 18` ya da `30`'dan biri olsun. Bu durumda silinecek düğümü direk `free()` fonksiyonuyla yok edebiliriz. Yani `1` no'lu seçeneğin işi oldukça kolay. Bir çocuğu olan düğümü nasıl silebiliriz? Örneğin Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-4'de görüldüğü gibi `5`'i silmek isteyelim. Bu durumda düğümün çocuğunu ebeveynine bağlar ve istenen düğümü silebiliriz. 2.seçenek de kolay bir işlem gibi görünüyor.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-4. İkili arama ağacından bir çocuğu olan düğümün silinmesi.

Peki, Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-5'teki gibi 15'i silmek istersek ne olacak? Bir düğümün kendisinin varsa sağ tarafındaki tüm çocuklardan küçük ya da eşit olduğu bilindiğine göre, sağ çocuklarından kendisine en yakın değerdeki düğüm, sağ tarafındaki çocuklarının en küçük düğümüdür. Öyleyse silinecek düğümün sol çocukları, sağ tarafın en küçük çocuğunun sol çocukları olacak şekilde bağlanır ve bu alt ağaç silinecek düğümün yerine kaydırılır.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-5. İkili arama ağacından iki çocuğu olan düğümün silinmesi

Bu algoritmaya göre silme işlemi yapacak fonksiyonu tasarlayabiliriz. Fonksiyon verilen düğümü sildikten sonra ağacı tekrar düzenleyip kökle geri döneceğinden, geri dönüş değeri `root` yani kök olmalı ve türü de `BTREE` olmalıdır. Parametre olarak hangi ağaçtan silme işlemi yapılacaksa o ağacın

kökünü ve silinecek veriyi (x verisine sahip olan düğüm) almalıdır. Şimdi `delete_node` isimli fonksiyonu yazalım.

```
BTREE *delete_node(BTREE *root, int x) {
     BTREE *p, *q;
     if(root == NULL)
          return NULL;
     if(root -> data == x) {
          if(root -> left == root -> right){
              free(root);
              return NULL;
          }
          else {
              if(root -> left == NULL) {
                    p = root -> right;
                    free(root);
                    return p;
              }
              else if(root -> right == NULL){
                    p = root -> left;
                    free(root);
                    return p;
              }
              else {
                    p = q = root -> right;
                    while(p -> left != NULL)
                          p = p -> left;
                    p -> left = root -> left;
                    free(root);
                    return q;
              }
          }
     }
     else if(root -> data < x)
          root -> right = delete_node(root -> right, x);
     else
          root -> left = delete_node(root -> left, x);
     return root;
}
```

Ağaç yoksa çalışacak olan kısım

1. durum

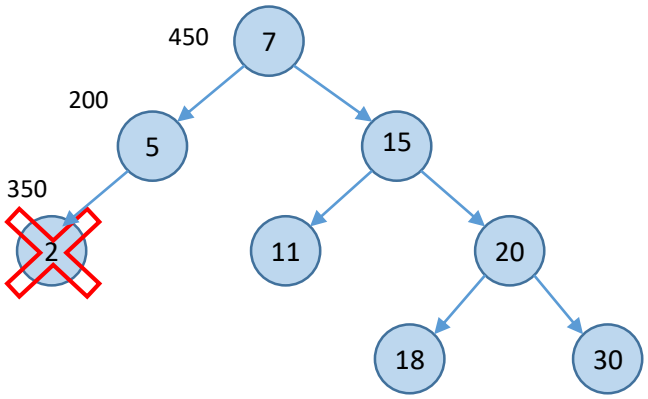
2. durum

3. durum

Aranan düğüm bulunmuşsa çalışacak olan kısım

Aranan düğüm bulunmamış ise çalışacak olan kısım

Birinci durum için fonksiyonun rekürsif olarak çalışmasını anlatalım. Diyelim ki 2 verisine sahip düğüm silinecek olsun.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-6. İkili arama ağacında birinci durum görünümü.

Şekilde düğümün adresinin `350` olduğu, kökün adresinin de `450` olduğu görülüyor. Kodu çalıştırdığımızda ilk olarak,

```
delete_node(450, 2);
```

çağrısı yapılacak ve aranan düğüm henüz bulunamadığı için en altta bulunan `else if` blokları devreye girecektir. 7, 2'den küçük olmadığı için bir sonraki `else` kısmı yürütülecek ve orada da fonksiyon kendisini çağıracaktır.

```
else
     root->left = delete_node(root->left, x); // Yürütülecek olan satır
```

`450` adresinin `left`'i `200` adresini gösterdiğine göre,

```
delete_node(200, 2);
```

çağrısı yapılıyor. Dikkat edilirse fonksiyonun geri dönüş değeri `root->left`'e atanacak. Aranan düğüm yine bulunamadığı için en altta bulunan `else if` blokları tekrar devreye girecektir. 5, 2'den küçük olmadığı için bir sonraki `else` kısmı yürütülecek ve rekürsif olarak fonksiyon tekrar çalışacaktır.

```
else
     root->left = delete_node(root->left, x); // Yürütülecek olan satır
```

Şimdi `200` adresinin `left`'i `350` adresini gösteriyor;

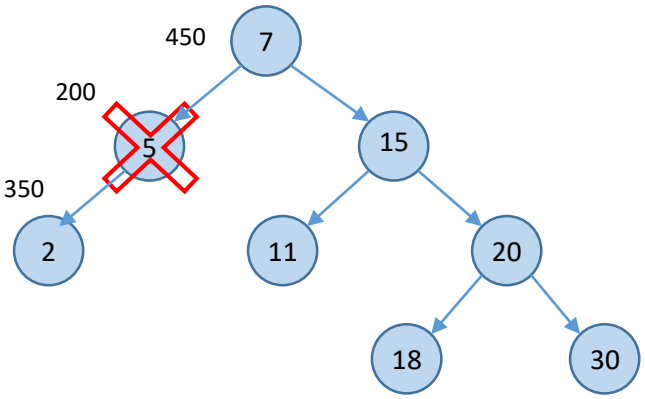
```
delete_node(350, 2);
```

Aranan düğüm şimdi bulunuyor. Düğüm bellekten siliniyor ve `root->left`'e `NULL` değeri atanıyor.

```
if(root -> data == x){ // root'un data'sı yani 2, 2'ye eşittir
     if(root -> left == root -> right) { // koşul doğru, ikisi de NULL
          free(root); // 350 adresine sahip düğüm bellekten silindi
```

```
            return NULL; // Fonksiyon NULL döndürerek sonlandı.
        }
        ...
        return root;
    }
```

Fonksiyon bir önceki çağrıldığı noktaya (2 numaralı çağrı) geri dönüyor ve o noktadan sonra `return root` kodu çalışıyor. O sırada root'un adresi `200` olduğu için geriye `200` adresini döndürerek `root->left`'e atama yapıyor. Tekrar çağrıldığı bir önceki noktaya yani 1 numaralı bölüme dönerek sonraki kod olan `return root` icra edilerek, o esnadaki kök adresi olan `450` geri döndürüyor.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-7. İkili arama ağacında ikinci durum görünümü.

Şekil 5.17'de görülen ikinci durum için fonksiyonu tekrar çalıştıralım. 5 silinecek ve bu düğümün adresi 200, kökün adresi de `450`'dir. İlk olarak kökün adresiyle fonksiyon çağrılıyor;

```
delete_node(450, 5);
```

Kökte 7 var, aranan düğüm henüz bulunamadığı için yine en altta bulunan else if blokları devreye giriyor. 7, 5'ten küçük olmadığı için bir sonraki `else` kısmı yürütülecek ve tekrar fonksiyon kendisini çağıracaktır.

```
else
    root->left = delete_node(root->left, x); // Yürütülecek olan satır
```

`450` adresinin `left`'i `200` adresini gösterdiğine göre,

```
delete_node(200, 5);
```

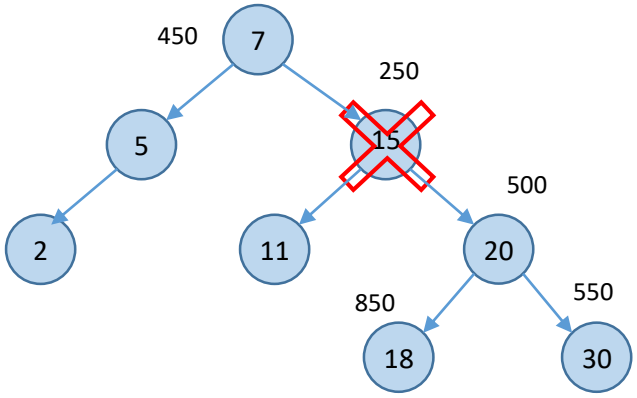
Aranan düğüm şimdi bulunuyor. Bu düğümün sol çocuğu var fakat sağ çocuğu yoktur. Bu yüzden aşağıda gösterilen `else` bloğu çalışıyor.

```
if(root -> left == NULL) {
    p = root -> right;
```

```
        free(root);
        return p;
    }
    else if(root -> right == NULL){
        p = root -> left; // Çalışacak blok
        free(root);
        return p;
    }
```

Silinecek düğümün sağ çocuğu `NULL` olduğu için üstteki kodda görüldüğü gibi `else if` bloğu devreye giriyor. Sol çocuk `p` işaretçisine atanarak `5` siliniyor ve p işaretçisi geri döndürülüyor. Fonksiyon en son `delete_node(200, 5)` çağrısıyla `2` numara ile gösterilen bölümde çalışmıştı ve `return` komutu icra edilerek `p` geri döndürülmüştü. Öyleyse ilk çağrıldığı nokta olan 1.bölüme giderek `p` kökün sol çocuğu olarak atanıp henüz icra edilmemiş olan alttaki kodlar çalışacak ve kökün adresini geri döndürerek silme işlemi tamamlanmış olacaktır.

Son olarak Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-8'de görüldüğü gibi üçüncü durum için fonksiyonu adım adım çalıştırıyoruz.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-8. İkili arama ağacında üçüncü durum görünümü.

Silinecek olan veri `15` ve adresi `250`, kökün adresi de `450`'dir. İlk olarak kökün adresiyle fonksiyon çağrılıyor;

```
delete_node(450, 15);
```

Kökte `7` var, aranan düğüm henüz bulunamadığı için yine en altta bulunan `else if` blokları devreye giriyor. `7`, `15`'ten küçük olduğu için fonksiyon bu defa aşağıdaki sağ çocuğu ile kendisini çağıracaktır.

```
else if
        root->right = delete_node(250, 15); // Yürütülecek olan satır
```

Aranan düğüm bulundu. Yani `if(root->data == x)` koşulundaki `root`'un `data`'sı da `15`'tir, `x` de `15`'tir. Sol ve sağ çocuğu `NULL` olmadığından dolayı fonksiyonda üçüncü durum olarak belirtilen else bloğu çalışıyor.

```
else {
     p = q = root -> right;
     while(p -> left != NULL)
          p = p -> left;
     p -> left = root -> left;
     free(root);
     return q;
}
```

Sol ve sağ çocuğun adresi `p` ile `q` işaretçisine atanıyor. Sonra silinecek düğümün sağ soyundaki en küçük `data`'yı barındıran düğümün adresi `p` işaretçisine atanıyor. Bu işlem while döngüsü içerisinde `15`'in sağ çocuğu olan `20`'nin sol tarafındaki yaprağa kadar inilerek sağlanıyor. Şimdi `p` işaretçisinde bu en küçük veriyi barındıran düğümün adresi, `q` işaretçisinde ise silinecek düğümün sağ çocuğunun adresi tutuluyor (`p=850, q=500`). Daha sonra `15`'in sol çocuğu olan `11`'in adresi `p->left = root->left` atamasıyla, bulunan en küçük veriyi barındıran düğümün sol çocuğu olarak bağlanıyor. Ardından `15` siliniyor ve `q` işaretçisi geri döndürülüyor. Fonksiyon en son `delete_node(250, 15)` çağrısıyla `2` numara ile gösterilen bölümde çalışmıştı ve return komutu icra edilerek `q` geri döndürülmüştü. Şimdi ilk çağrıldığı nokta olan 1.bölüme giderek `q` kökün sağ çocuğu olarak atanıp henüz icra edilmemiş olan alttaki kodlar çalışacak ve o bölümdeki kökün adresi olan `450`'yi geri döndürerek silme işlemi tamamlanmış olacaktır. Yeni oluşan ağaç ise hala bir **BST** (*Binary Searh Tree*) ikili arama ağacıdır.

9.3.5 İkili Arama Ağacında Bir Düğümü Bulmak

```
BTREE* searchtree(BTREE* tree, int data) {
     if(tree != NULL)
          if(tree -> data == data)
              return tree;
          else if(tree -> data > data)
              searchtree(tree -> left, data);
          else
              searchtree(tree -> right, data);
     else
          return NULL;
}
```

9.3.6 İkili Arama Ağacı Kontrolü

```
boolean isBST(BTREE* root) { // boolean türü stack kısmında anlatılmıştı
     if(root == NULL)
          return true;
     if(root -> left != NULL && maxValue(root -> left) > root -> data)
          return false;
     if(root -> right != NULL && minValue(root -> right) <= root -> data)
          return false;
     if(!isBST(root -> left) || !isBST(root -> right))
          return false;
     return true;
}
```

9.3.7 İkili Arama Ağacında Minimum Elemanı Bulmak

```
int minValue(BTREE* root) {
     if(root == NULL)
          return 0;
     while(root -> left != NULL)
          root = root -> left;
     return(root -> data);
}
```

9.3.8 İkili Arama Ağacında Maximum Elemanı Bulmak

```
int maxValue(BTREE* root) {
     if(root == NULL)
          return 0;
     while(root -> right != NULL)
          root = root -> right;
     return(root -> data);
}
```

9.3.9 Verilen İki Ağacı Karşılaştırmak

```
int isSame(BTREE* a, BTREE* b) {
     if(a == NULL && b == NULL)
          return 1; // İki ağaç da boş ise true döndürür
     else if(a != NULL && b != NULL)
          return (
              a -> data == b -> data && isSame(a -> left, b -> left) &&
     isSame(a -> right, b -> right) // Koşul doğru ise true döndürür
```

```
            );
        else
            return 0;
    }
```

9.3.10 Alıştırmalar

Örnek 23: Girilen bir x değeri, kökten itibaren yaprak dahil olmak üzere o yol üzerindeki verilerin toplamına eşitse true, eşit değilse false döndüren path isimli programı kodlayınız.

```
int path(BTREE* root, int sum) {
    int pathSum;
    if(root == NULL) // Ağaç NULL ise
        return (sum == 0); // sum 0'a eşitse true dönüyor
    else {
        pathSum = sum - root -> data;
        return (
            path(root -> left, pathSum) || path(root -> right, pathSum)
        );
    }
}
```

Örnek 24: Bir ikili arama ağacındaki verilerden tek olanları diğer bir BST ağacına kopyalayan copyOdd isimli programı yazınız.

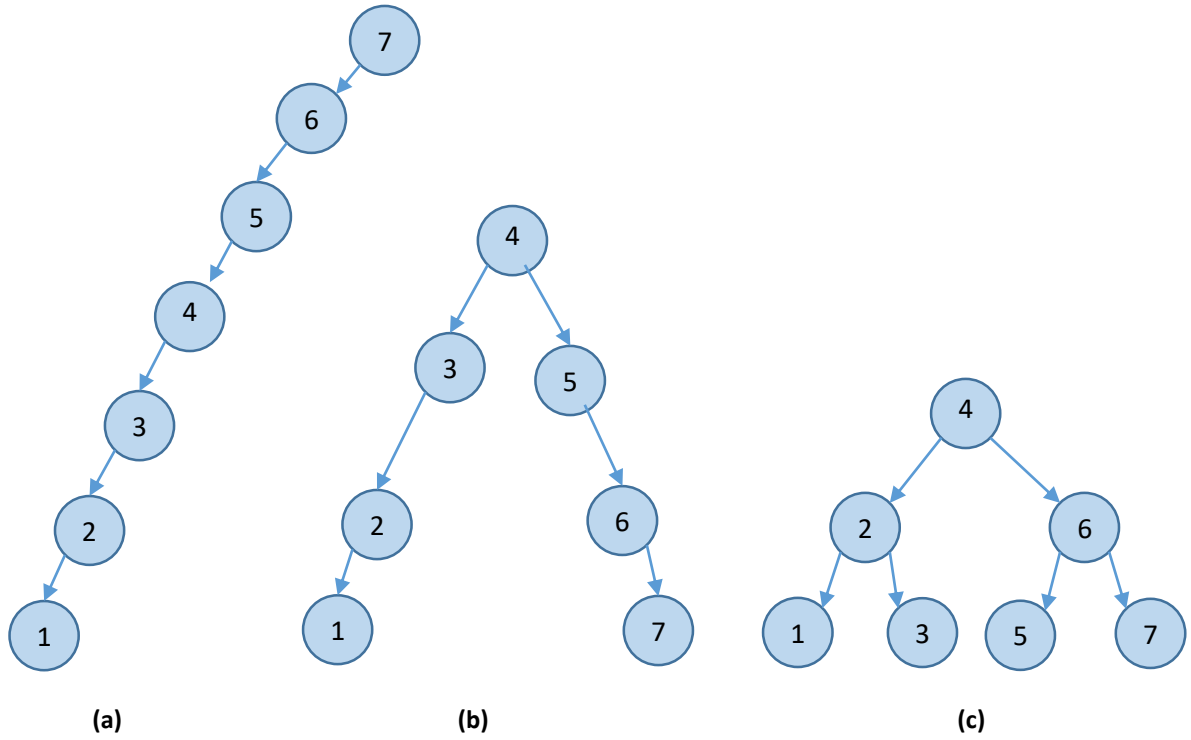
```
BTREE* copyOdd(BTREE* root, BTREE* root2) {
    if(root != NULL) {
        if(root -> data % 2 == 1)
            root2 = insertBST(root2, root -> data);
        root2 = copyOdd(root -> left, root2);
        root2 = copyOdd(root -> right, root2);
    }
    return root2;
}
```

9.4 AVL Ağaçları

Yahudi bir matematikçi ve bilgisayar bilimcisi olan Sovyet Georgy Maximovich Adelson-Velsky (8 Ocak 1922 – 26 Nisan 2014) ile yine Sovyet matematikçi Yevgeny (Evgenii) Mikhaylovich Landis (6 Kasım 1921 – 12 Aralık 1997) adlı kişilerin 1962 yılında buldukları bir veri yapısıdır. Bu veri yapısı, isimlerinin baş harflerinden alıntı yapılarak AVL ağaçları olarak adlandırılmıştır. AVL ağaçları hem dengelidir hem de BST (binary search tree) ikili arama ağacıdır. Normal bir ikili ağacın yüksekliği maksimum `n` adet düğüm için `h = n - 1`'dir. AVL yöntemine göre kurulan bir ikili arama ağacında sağ alt ağaç ile sol

alt ağaç arasındaki yükseklik farkı **en fazla bir** olabilir. Bu kural ağacın tüm düğümleri için geçerlidir. Herhangi bir düğümün sağ ve sol alt ağaçlarının yükseklik farkı 1'den büyükse o ikili arama ağacı, AVL ağacı değildir.

Normal ikili arama ağaçları için ekleme ve silme işlemleri ağacın orantısız büyümesine, yani ağacın dengesinin bozulmasına neden olabilir. 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. Aşağıda `1-7` arası sayıların farklı sırada ikili arama ağacına eklenmesiyle oluşan üç değişik ağaç gösterilmiştir.



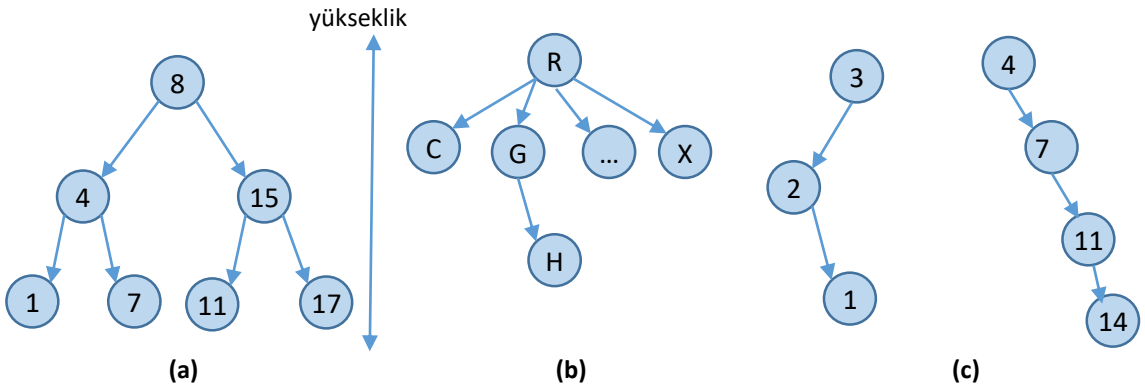
Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-9. a, b, c) İkili bir arama ağacı modelleri.

Sayıların hangi sırada geleceği bilinemediğinden uygulamada bu üç ağacın biri ile karşılaşılabilir. Bu durumda dengeli ağaçlar tercih edilir. Dolayısıyla ağacın dengesi bozulduğunda ağacı yeniden dengelemek gerekir. İkili arama ağacının yüksekliğinin olabildiğince düşük olması arzu edilir. Bu yüzden aynı zamanda height balanced tree olarak da bilinirler.

AVL ağacında bilinmesi gereken bir kavram **denge faktörüdür** (**balance factor**). Denge koşulu oldukça basittir ve ağacın derinliğinin θ(*logn*) kalmasını sağlar.

$$balanceFactor = height_{(sağAltAğaç)} - height_{(solAltAğaç)}$$

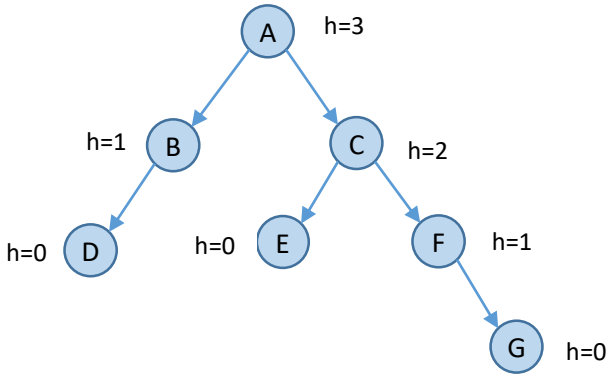
Denge faktörü `−1`, `0` ve `1` değerini alabilir. Dikkat edilmesi gereken önemli bir nokta herhangi bir düğümün denge faktörü hesaplanırken sol ve sağ ağaçların yüksekliklerinin belirlenmesidir. Yoksa herhangi bir kayıt için düğümlerin sayılması değildir. Bir başka deyişle AVL ağacında sol alt ağaç ile sağ alt ağaç farkının mutlak değeri `1`'den küçük ya da `1`'e eşit olmalıdır.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-10. (a) Dengeli ikili ağaç (b) Dengeli ağaç (c) Dengesiz ikili ağaçlar

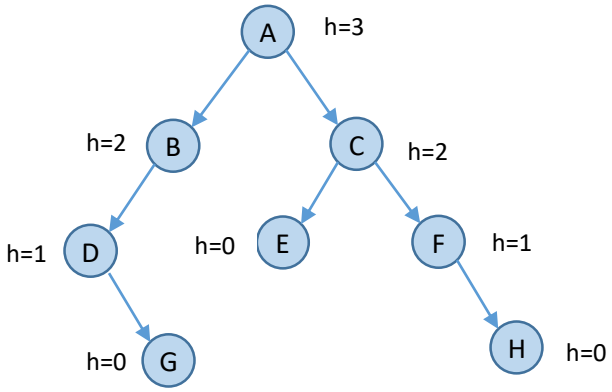
Sağ veya sol çocuk yok ise (yani olmayan düğümlerin ya da diğer bir söyleyişle NULL değerlerin) yüksekliği -1'dir. Bir AVL ağacında sol alt ağacın yüksekliği 3 ise, sağ alt ağacın yüksekliği ancak 2, 3 veya 4 olabilir.

Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-11'de ise AVL ağacının sol ve sağ alt ağaçlarının her bir düğümünün yüksekliği karşılaştırılıyor. Her düğüm **kardeşiyle** (**sibling**) kıyaslanıyor. Kardeşi yoksa `NULL` olanın yükseklik değerinin `−1` olduğunu belirtmiştik. Örneğin `D` düğümünün çocuğu yoktur. Dolayısıyla D düğümünün yükseklik değeri `0`'dır. Sağında düğüm yoktur ve yükseklik değeri de `−1` kabul edilmektedir. `G` yaprağının yüksekliği `0`'dır. Solunda düğüm yoktur ve yükseklik değeri `−1`'dir. Diğer düğümlerin arasındaki yükseklik farkı aşağıdaki şekilde açıkça görülmektedir. Yüksekliği `1` olan düğümlere dikkat ediniz. Solundaki ya da sağındaki düğümle arasındaki yükseklik farkları `1`'dir.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-11. AVL ağacında sol ve sağ alt ağaçların her bir düğümünün yüksekliği.

Şimdi de AVL olmayan bir ağacı inceleyelim. Şekil 5.23'te yer alan ikili ağaç dengeli bir ağaç değildir. Çünkü D düğümünün yüksekliği 1 iken, kardeşi olmadığından sağ tarafının yüksekliği -1'dir ve aradaki fark 2'dir. Bu nedenle bir AVL ağacı değildir. Şekildeki B düğümü gibi tek çocuğu olan ve aynı zamanda torunu da olan ağaçlar bir AVL ağacı değildir de denilebilir.



Şekil **Hata! Belgede belirtilen stilde metne rastlanmadı.**-12. Dengeli olmayan ikili bir ağaçta düğümlerin yükseklikleri

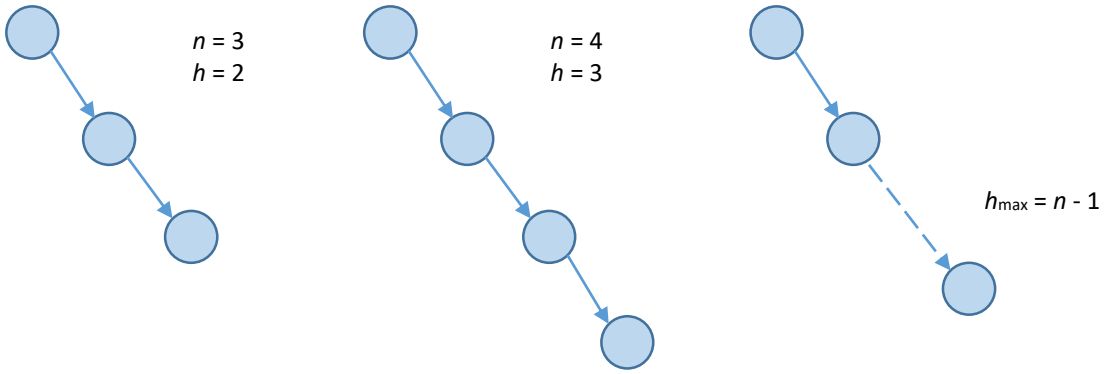
Bir ikili ağacın yüksekliği (*h*):

$\log n < h < n$

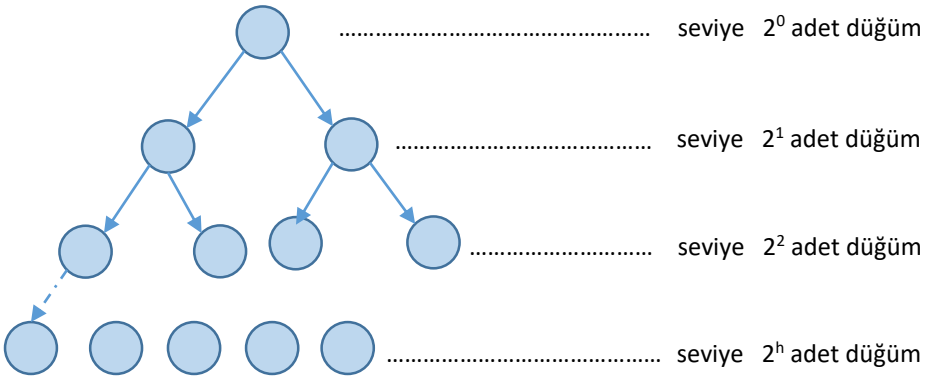
iken, *n* düğümlü bir AVL ağacının yüksekliği $\theta(\log n)$'dir. Burada, θ gayri resmi olarak mertebe ya da civarı anlamındadır (gösteriminin formal tanımı, Algoritmalar dersinde verilecektir). Öyleyse ifadeyi, bir AVL ağacının yüksekliği 2 tabanında logaritma *n* civarındadır veya mertebesindedir şeklinde de söyleyebiliriz. log*n*, *n*'den çok çok küçük bir sayıdır. Örneğin, $n = 10^6$ ise $\log 10^6 \approx 20$ civarındadır. Bu önerme AVL ağaçları için bir kolaylık sağlamaktadır. Yüksekliğin az olması dengeyi sağlamakta ve bu denge de arama, ekleme ve silme işlemlerinde yüksek bir performans kazandırmaktadır.

Örnek 25: *n* düğümlü bir ikili ağacın maksimum ve minimum yüksekliğini bulunuz.

Çözüm: İkili ağacın maksimum yüksekliğini bulmak oldukça kolaydır. Bir ikili ağacın maksimum yüksekliğe sahip olabilmesi için aşağıdaki şekilde görüldüğü gibi düğümlerin ardı ardına sıralanması gerekir.



Şekilde 3 düğümlü bir ikili ağacın yüksekliği 2, 4 düğümlü bir ikili ağacın yüksekliği ise 3'tür. O halde *n* düğümlü bir ağacın maksimum yüksekliği $h = n - 1$'dir. Bir ikili ağaçta minimum yüksekliği sağlamak için bir seviye tamamlanmadan diğer seviyeye düğüm bağlanmamalıdır. Mümkün olduğu kadar ağacı sıkışık yapmak gerekmektedir. Böyle bir ikili ağaçta bir düğümün kök düğümden olan uzaklığına düzey (level) dendiği bilindiğine göre kök düğümün düzeyi 0 (sıfır), yaprakların düzeyi ise h'a eşit olacaktır.



Son düzeydeki düğüm sayısının en fazla 2^h adet olabileceğine dikkat ediniz. Öyleyse bu şekilde bir h yüksekliğindeki ikili ağaçta olabilecek maksimum düğüm sayısı, $1 + 2 + 2^2 + \cdots + 2^h$ geometrik serisi olacaktır. Bu da $2^{h+1} - 1$'e eşittir. İkili bir arama ağacındaki *n* adet düğümle h yükseklik ilişkisini $2^{h+1} - 1 \geq n$ şeklinde ifade edebiliriz. Düzenlendiğinde ise, $2^{h+1} \geq n + 1$ şeklini alan ifadenin her iki tarafının 2 tabanında logaritmasını aldığımızda;

$h + 1 \geq \log(n+1)$ ise $h \geq \log(n+1) - 1$

olur. Buradan ikili bir ağacın minimum yüksekliği $h_{\min} = \log(n+1) - 1$ bulunur. Bulunan bu sonuç, ikili bir ağacın maksimum yüksekliğinin *n* mertebesinde, minimum yüksekliğinin ise log*n* mertebesinde olduğunu göstermektedir. Yükseklik *n* ile log*n* arasında değişmektedir.