Algoritma Analizi

Ders 11

Doç. Dr. Mehmet Dinçer Erbaş
Bolu Abant İzzet Baysal Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği Bölümü

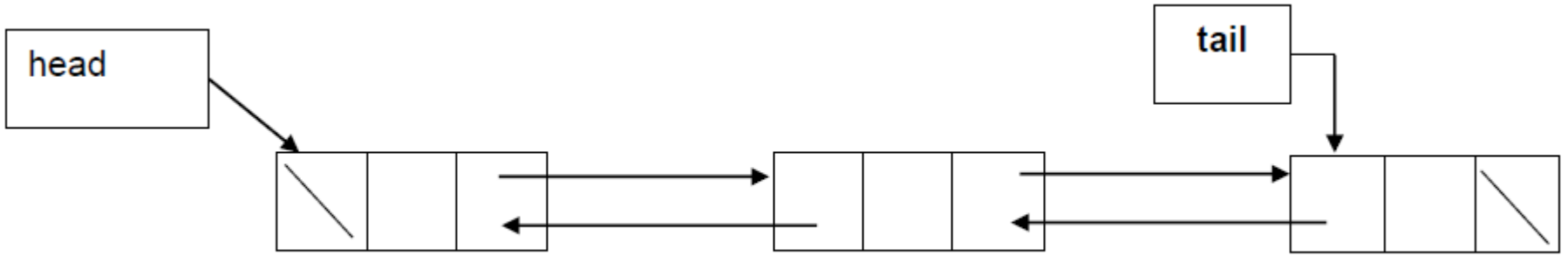
Bağlı liste

- Çift yönlü bağlı listeler
  - Çift yönlü bağlı listelerde düğümler iki işaretçiye sahiptir.
  - Bu sebeple iki yöne hareket etmek mümkündür.
  - ListSearch ve DisplayList O(n) süre alır, diğer bütün operasyonlar O(1) süre alır.

```
typedef struct node
{
        int key;
        struct node* prev;
        struct node* next;
}node;
```

Bağlı liste

- Çift yönlü bağlı listeler
  - Tek yönlü bağlı listede yaptığımız gibi, bağlı listemize düğüm eklemek veya bağlı listemizden düğüm silebilmek için listedeki ilk ve son düğümü hatırlayacağız.
    - İlk düğüm head, son düğüm ise tail olarak adlandırılacak.




```
node *head = NULL;
node *tail = NULL;
```

Bağlı liste

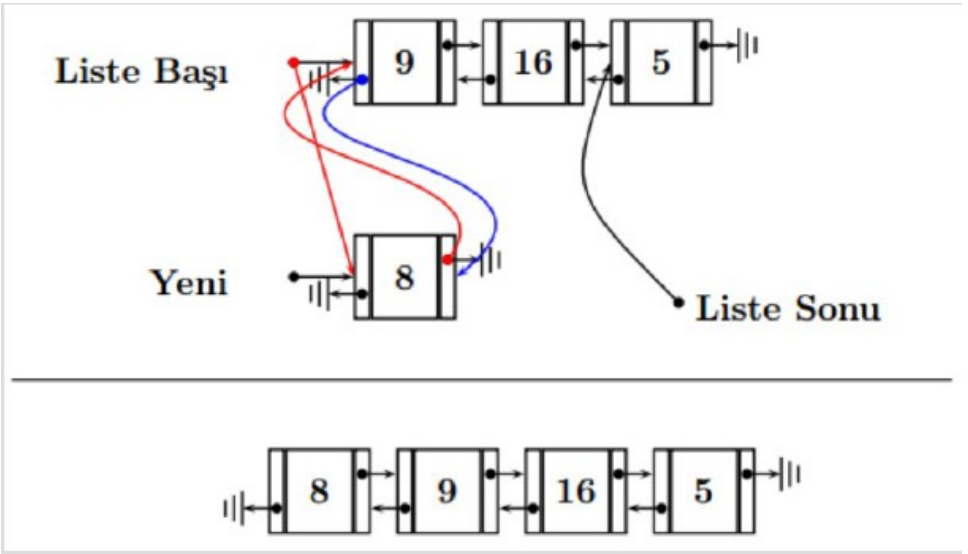
- Çift yönlü bağlı listeler

```
node* CreateNode(int value)
{
    node *newNode = malloc(sizeof(node));
    newNode->key = value;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
```

Bağlı liste

- Çift yönlü bağlı listeler

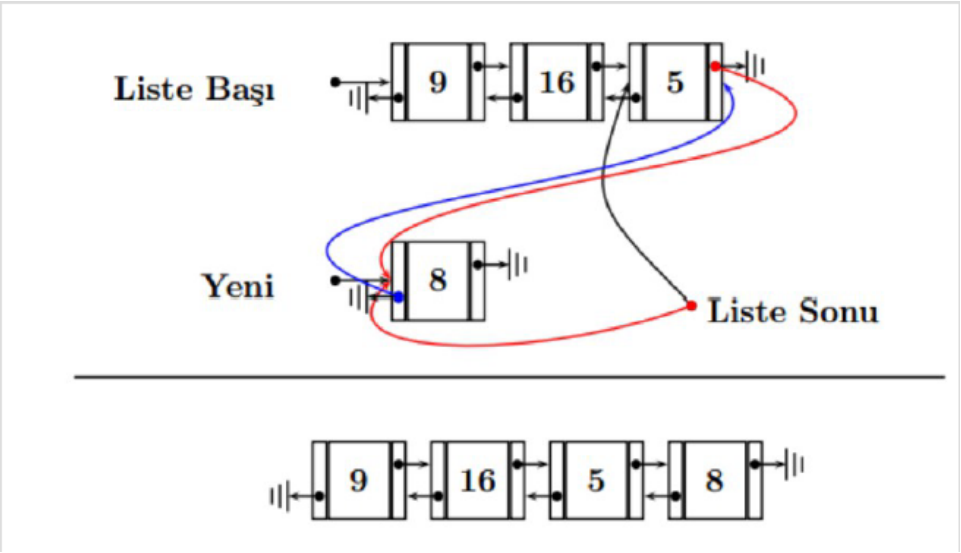
```
void ListInsertHead(node *x)
{
     if(head == NULL)
     {
          head = x;
          tail = x;
     }
     else
     {
          x->next = head;
          head->prev = x;
          head = x;
     }
}
```



Bağlı liste

- Çift yönlü bağlı listeler

```
void ListInsertTail(node *x)
{
     if(tail == NULL)
     {
          head = x;
          tail = x;
     }
     else
     {
          x->prev = tail;
          tail->next = x;
          tail = x;
     }
}
```

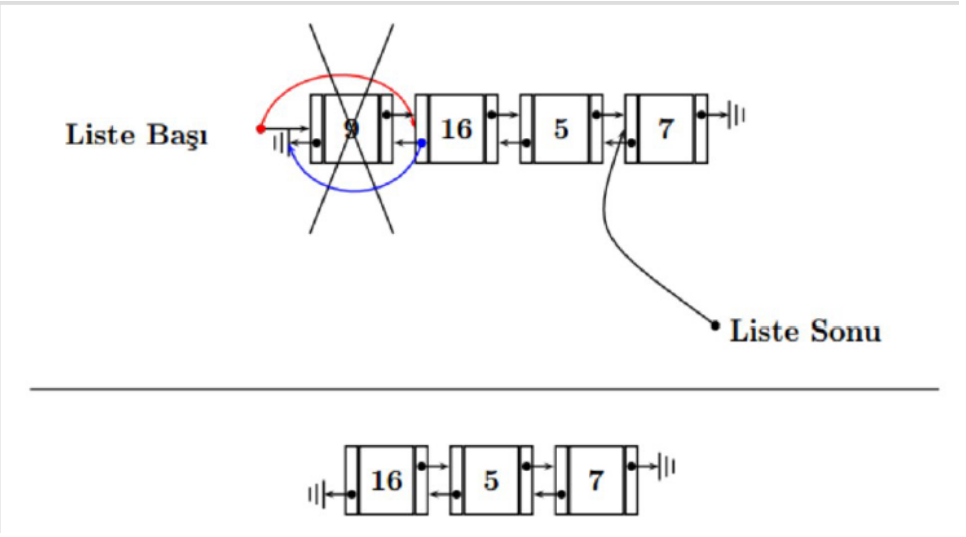


Bağlı liste

- Çift yönlü bağlı listeler

```
void ListDeleteHead()
{
     if(head == NULL)
          return;
     node* x = head;

     head = head->next;
     if(head != NULL)
          head->prev = NULL;
     else
          tail = NULL;
     free(x);
}
```

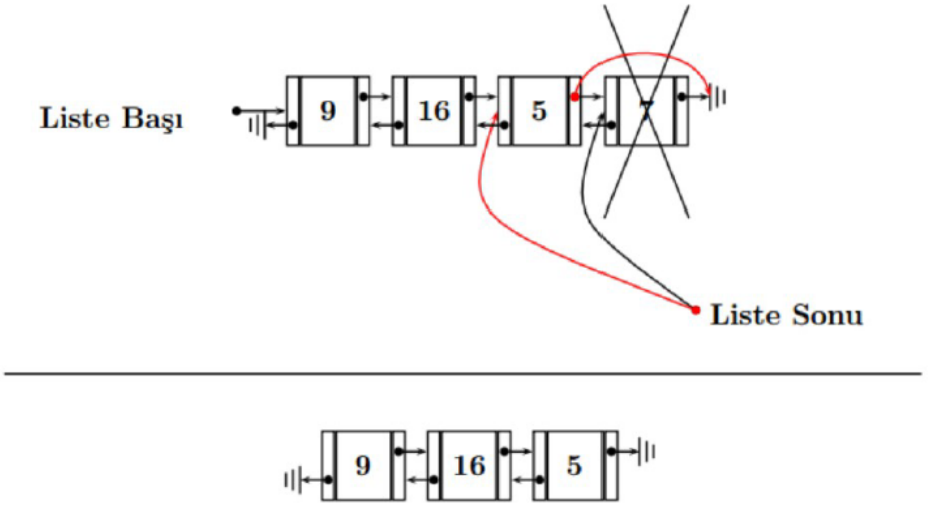


Bağlı liste

- Çift yönlü bağlı listeler

```
void ListDeleteTail()
{
    if(tail == NULL)
        return;
    node* x = tail;

    tail = tail->prev;
    if(tail != NULL)
        tail->next = NULL;
    else
        head = NULL;
    free(x);
}
```



Bağlı liste

- Çift yönlü bağlı listeler

```
void DisplayList()
{
    printf("DisplayList starts.....................\n");
       node *x = head;
       while(x != NULL)
       {
              printf("%d ", x->key);
              x = x->next;
       }
       printf("\nDisplayList ends.....................\n\n\n");
}
```

Bağlı liste

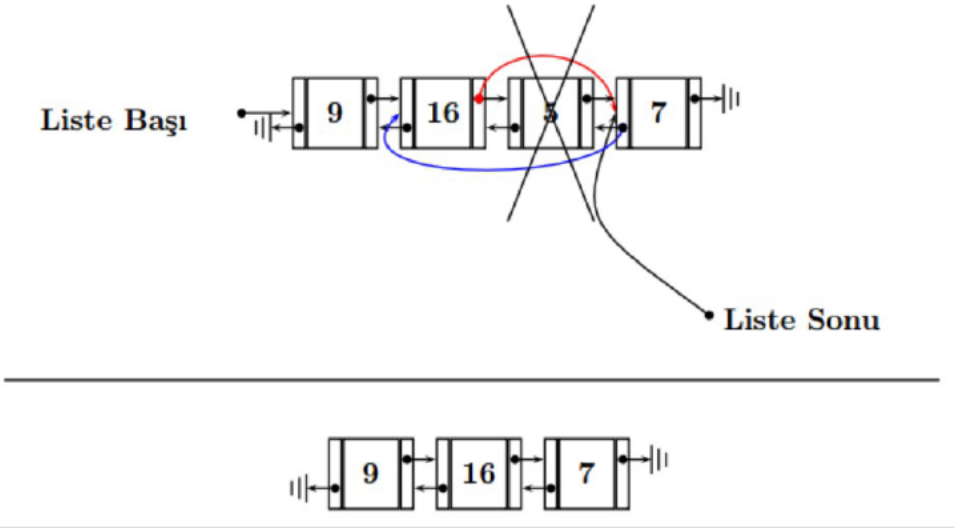
- Çift yönlü bağlı listeler

```
node* ListSearch(int k)
{
        node *x;
        x = head;
        while(x != NULL && x->key != k)
        {
                x = x->next;
        }
        return x;
}
```

Bağlı liste

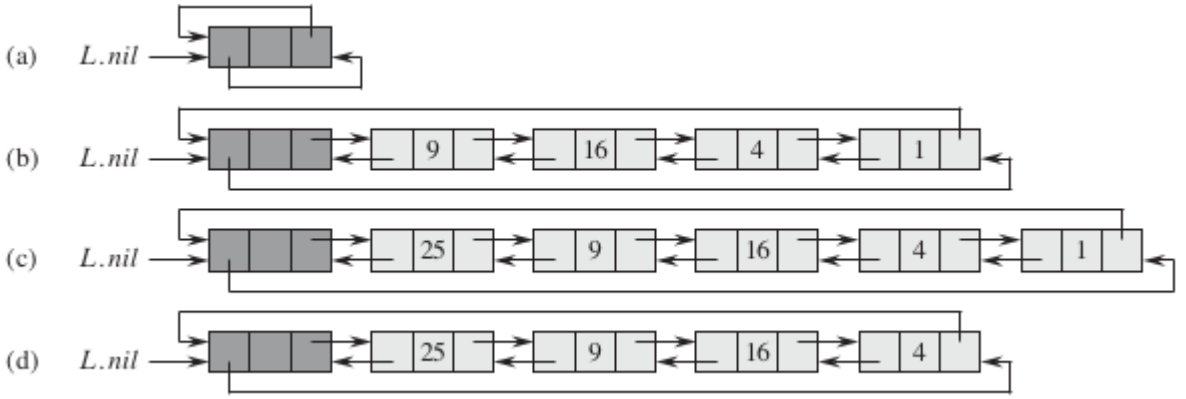
- Çift yönlü bağlı listeler

```
void ListDeleteNode(node* x)
{
    if(x == NULL)
        return;
    if(x == head)
        ListDeleteHead();
    else if(x == tail)
        ListDeleteTail();
    else
    {
        x->prev->next = x->next;
        x->next->prev = x->prev;
        free(x);
    }
}
```



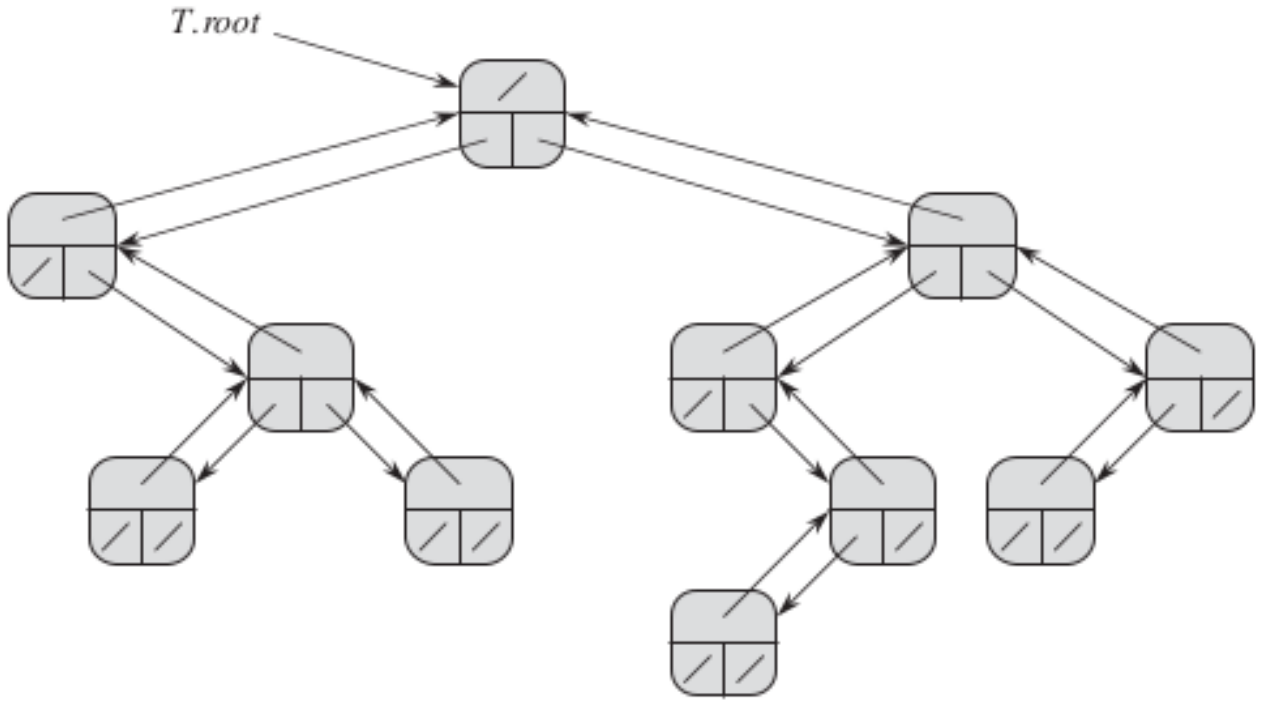
Bağlı liste

- Çift yönlü bağlı listeler
  - head ve tail düğümlerini gözcü değer olarak kullanarak fonksiyonları oldukça basitleştirebiliriz.
  - Ayrıca bağlı listeyi dairesel hale getirerek tek bir gözcü değer ile işlemleri yapabiliriz.



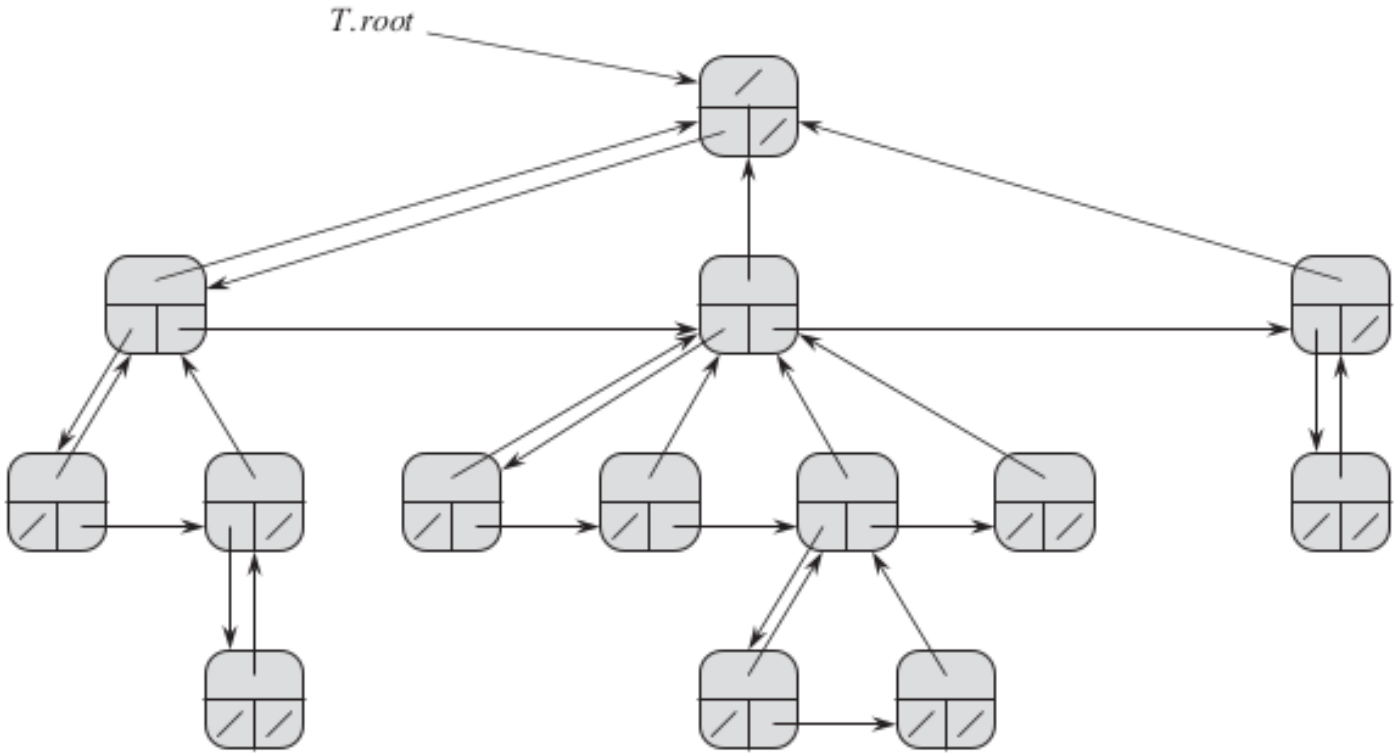
Bağlı liste

- Köklü ağaçları göstermek
  - Bağlı listeler kullanılarak ağaç yapısının oluşturulması mümkündür.
  - Örneğin: ikili ağaçlar (Binary trees)



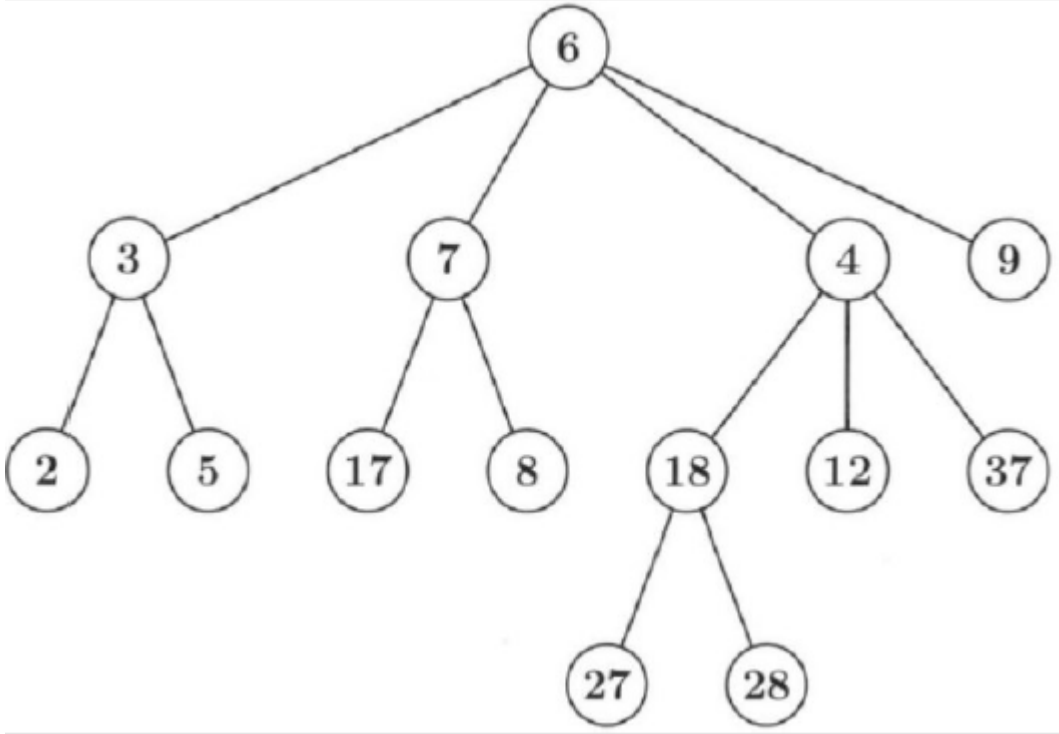
Bağlı liste

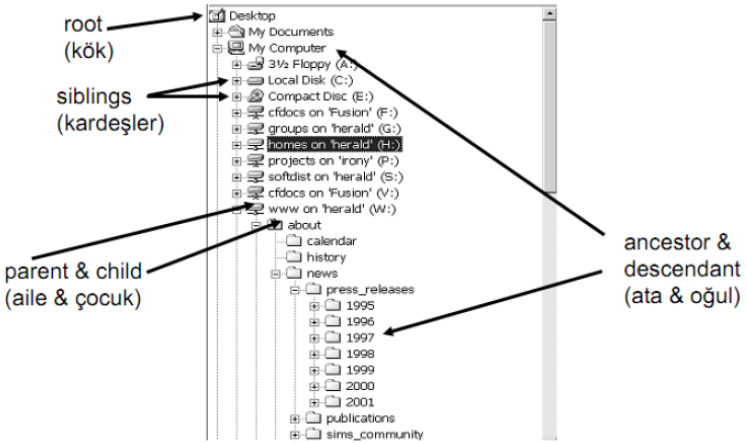
- Köklü ağaçları göstermek
  - Bağlı listeler kullanılarak ağaç yapısının oluşturulması mümkündür.
  - Örneğin: Sınırsız sayıda dalları olan ağaçlar



Ağaç veri yapısı

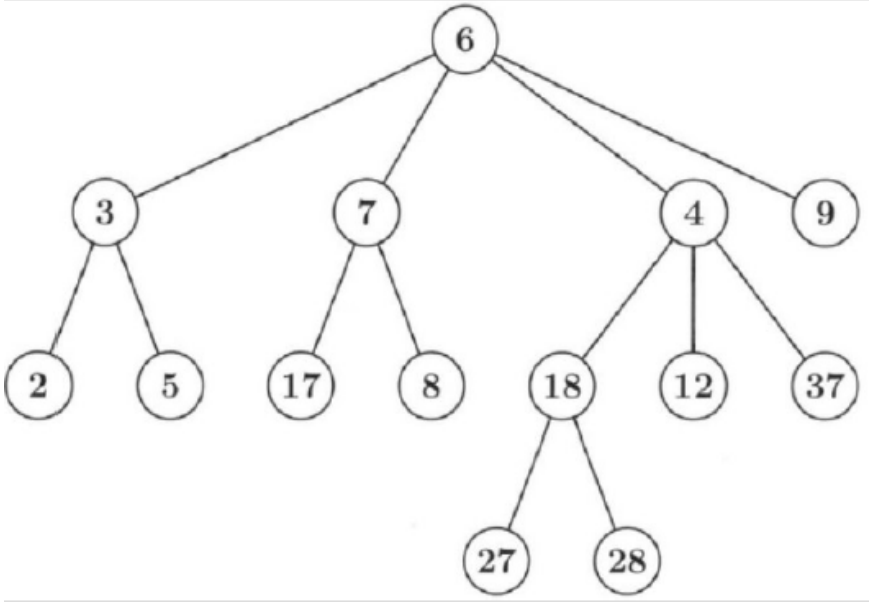
- Ağaç veri yapısı doğrusal olmayan veri yapılarına örnektir.
- Ağaç, bir kök, sonlu sayıda düğümleri ve onları birbirine bağlayan dalları olan bir veri modelidir.
- Ağaçlar özellikle hiyerarşik yapıları gösterebilmek için yaygınlıkla kullanılır.
- Birçok problemin çözümü veya modellenmesinde ağaç yapısı kullanılır.





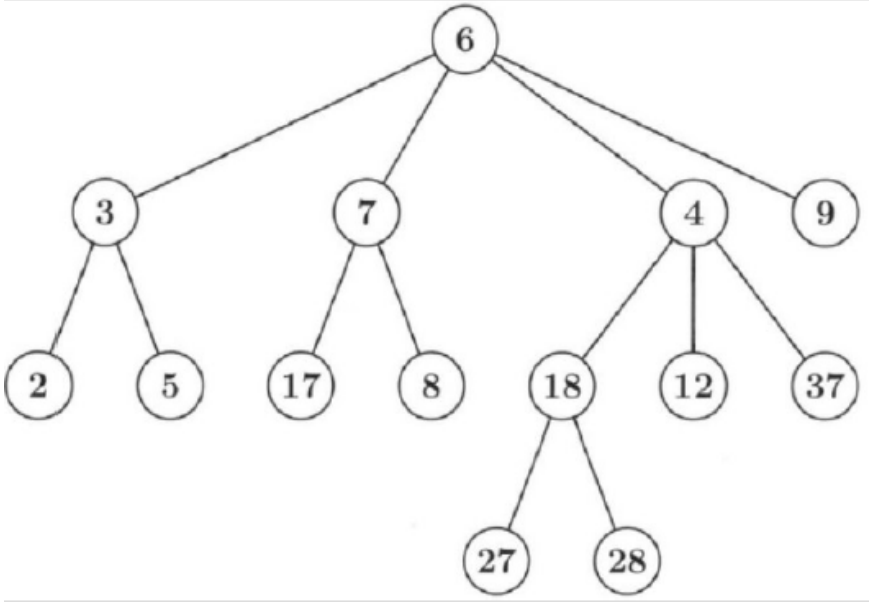
Ağaç veri yapısı

- Ağaç veri yapısına ait tanımlar.
  - Düğüm (Node): ağacın her bir elemanına düğüm adı verilir.
  - Kök (Root): Ağacın başlangıç düğümüdür.
  - Çocuk (child): Bir düğüme doğrudan bağlı alt düğümlere o düğümün çocukları ismi verilir.
  - Kardeş (Sibling): Aynı düğümün çocukları olan düğümlere kardeş düğüm denir.
  - Ebeveyn (Parent): Bir düğüme doğrudan bağlı üst düğümlere o düğümün ebeveyni denir. Düğümden köke giden yol üzerinde ziyaret edilen ilk düğümdür.
  - Ata (Ancestor): Bir düğümünün ataları, o düğümden köke olan yol üzerindeki tüm düğümlerdir.
  - Soyundan gelenler (Descendant): Bir düğüme bağlı olan alt düğümler kümesi.
  - Derece (Degree): Ağaçtaki bir düğümün sahip olabileceği maksimum çocuk sayısı.



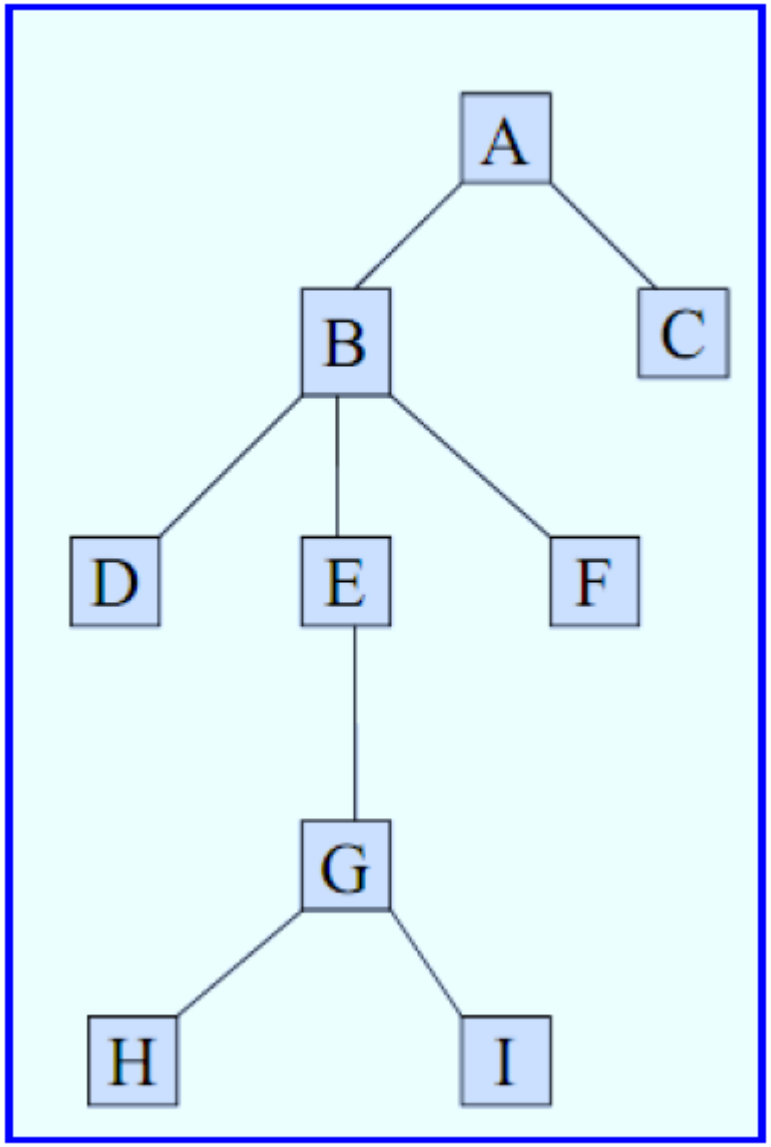
Ağaç veri yapısı

- Ağaç veri yapısına ait tanımlar.
  - Yol (Path): Bir düğümün bir başka düğüme gidebilmek için üzerinden geçilmesi gereken düğümlerin listesidir.
  - Düzey (Level) ve Derinlik (Depth): Bir düğümün kök düğümden olan uzaklığıdır. Kök düğümün düzeyi ve derinliği 1 olarak kabul edilir.
    - Yandaki ağaçta (3) değerine sahip düğümün derinliği ve düzeyi 2’dir.
  - Yükseklik: Bir düğümün en uzak alt düğümüne olan yolun uzunluğudur. Ağacın yüksekliği ise kök düğümün yüksekliğidir.
    - Yandaki ağaçta kök düğümünün yüksekliği 3’tür.
  - Yaprak (Leaf): Ağacın en altında bulunan ve çocukları olmayan düğümlerdir.
    - Yandaki ağaçta yapraklar: 2, 5, 17, 8, 27, 28, 12, 37, 9.



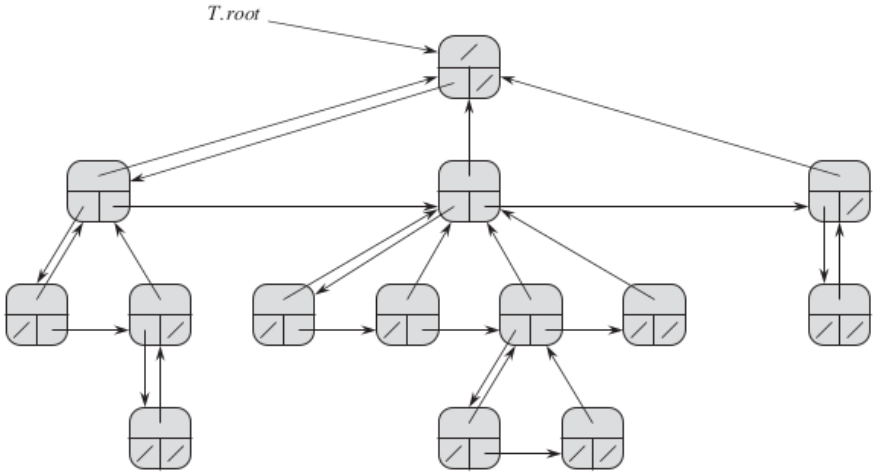
Ağaç veri yapısı

- Örnek: Yandaki ağacı inceleyelim.
  - Düğüm sayısı: 9
  - Kök düğüm: A
  - Ağacın yüksekliği: 4
  - Düzey sayısı: 5
  - Yapraklar: D, H, I, F, C
  - B‘nin çocukları: D, E, F
  - B‘nin kardeşi: C
  - B‘nin soyundan gelenler: D, E, F, G, H, I.
  - B’nin ataları: A



Ağaç veri yapısı

- Her tür ağacı oluşturmak için veri ve belli sayıda işaretçi içeren düğümler oluşturulmalıdır.




```
typedef struct node
{
        int key;
        struct node* parent;
        struct node* firstChild
        struct node* nextSibling;
}node;
```

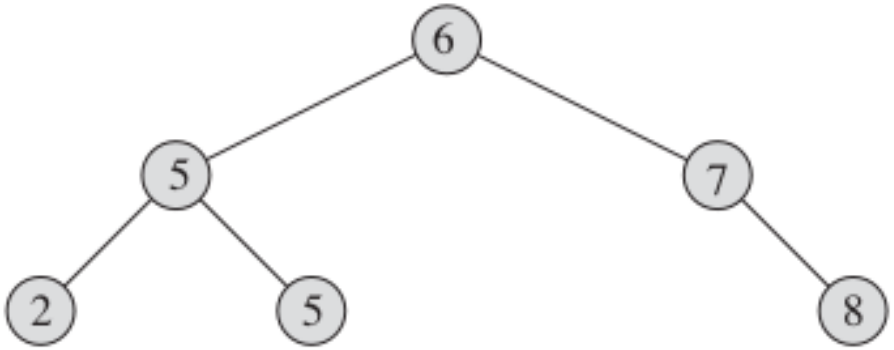
İkili arama ağaçları

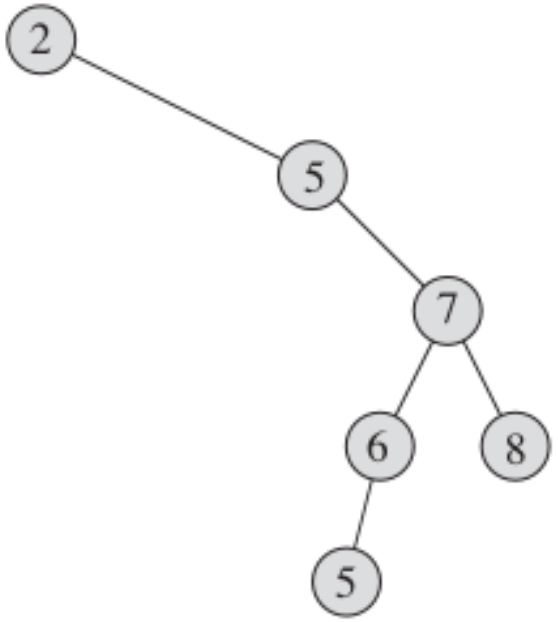
- İkili arama ağaçları dinamik kümeler üzerine yapılan birçok işlemi desteklerler.
  - Arama (Search)
  - Minimum
  - Maksimum
  - Önceki (Predecessor)
  - Sonraki (Successor)
  - Ekle (Insert)
  - Sil (Delete)
- Bu basit operasyonlar ağacın yüksekliği ile orantılı süre alır.
  - Tam ikili ağaçlarda O(lg n) süre alır.
  - Ancak ağaç bağlı listeye dönüşürse O(n) süre alır.
  - Bazı özelliklere göre oluşuturulan ağaçlarda O(lg n) ortalama süreye erişilir.

İkili arama ağaçları

- Ikili arama ağaçları her düğümün en fazla iki çocuk düğüme sahip olabileceği ağaçlardır.
- Bu tür ağaçlar her düğümün bir obje olarak temsil edildiği linklenmiş listeler yoluyla gösterilebilir.
- Her düğüm ek bilgilerle birlikte üç tane işaretleyici içerir:
  - left: sol taraftaki çocuk düğüm.
  - right: sağ taraftaki çocuk düğüm.
  - parent: ebeveyn düğüm.
- Bu düğümler mevcut değilse değerler NIL (veya NULL) olur.
  - Ağacın kök düğümü parent düğümü NIL olan tek düğümdür.
- İkili arama ağaçları özelliği şu şekilde tanımlanır:
  - x ağaçta bir düğüm olsun. Eğer y x‘in sol alt ağacında bir düğüm ise key[y] ≤ key[x]‘dir. Eğer y x‘in sağ alt ağacında bir düğüm ise key[y] ≥ key[x]‘dir.

İkili arama ağaçları





İkili arama ağaçları

- İkili ağaç yapısı oluşturabilmek için öncelikle bir düğüm yapısı oluşturmalıyız

```
typedef struct node
{
        int key;
        struct node* parent;
        struct node* left;
        struct node* right;
}node;
```

İkili arama ağaçları

- Daha sonra bir düğümü dinamik olarak oluşturan bir fonksiyon yazmalıyız.

```
node* CreateNewNode(int key)
{
        node *newNode = malloc(sizeof(node));
        newNode->key = key;
        newNode->left = NULL;
        newNode->right = NULL;
        newNode->parent = NULL;
        return newNode;
}
```

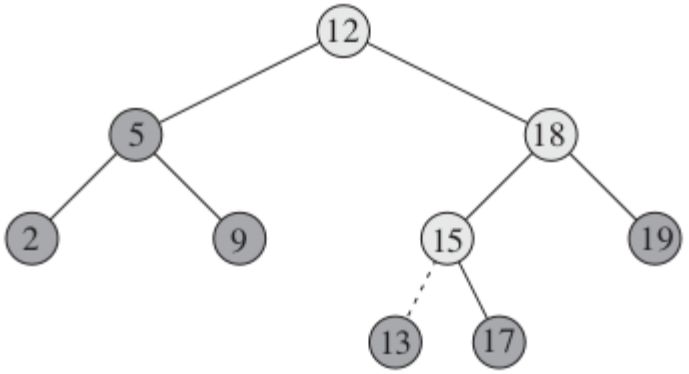
- Düğüm işlemleri için kök düğümü hatırlamalıyız.

```
node *root = NULL;
```

İkili arama ağaçları

- Ağacımıza yeni bir düğüm eklemek için bir fonksiyon yazmalıyız.

```
void TreeInsert(node *z)
{
          node *y = NULL;
          node *x = root;
          while(x != NULL)
          {
                    y = x;
                    if(z->key < x->key)
                              x = x->left;
                    else
                              x = x->right;
          }
          z->parent = y;
          if(y == NULL)
                    root = z;
          else if(z->key < y->key)
                    y->left = z;
          else
                    y->right = z;
}
```



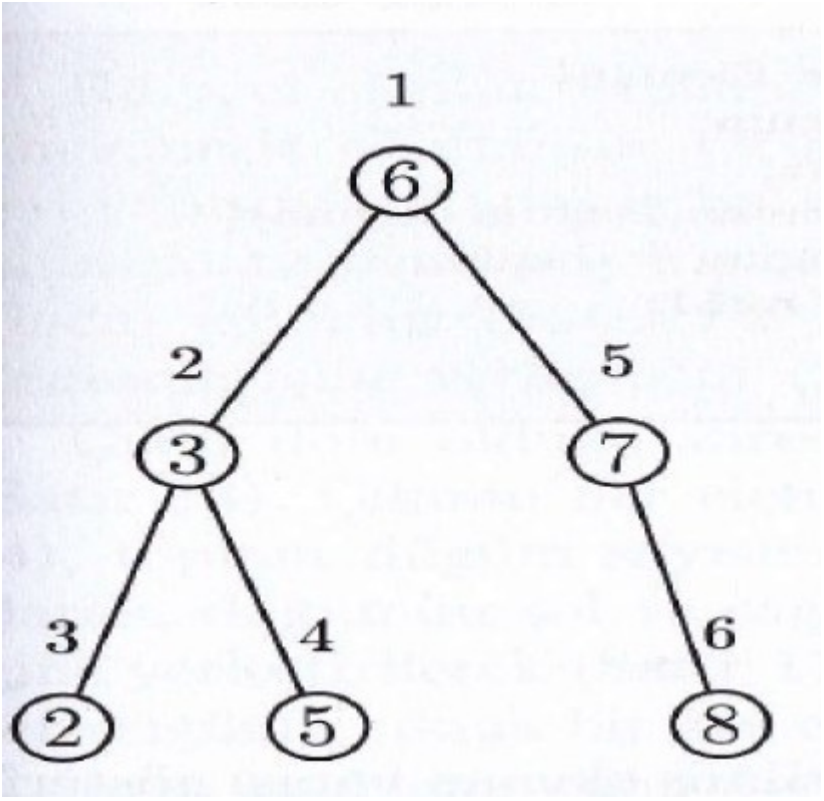
İkili arama ağaçları

- Ağaç üzerinde dolaşmak ve elemanlarını göstermek istediğimizde bunu farklı şekillde yapabiliriz:
  - Preorder (Önce Kök) Dolaşma
  - Inorder (Kök Ortada) Dolaşma
  - Postorder (Kök Sonda) Dolaşma

İkili arama ağaçları

- Ağaç üzerinde dolaşmak ve elemanlarını göstermek istediğimizde bunu farklı şekillde yapabiliriz:
  - Preorder (Önce Kök) Dolaşma

```
void PreOrderTreeWalk(node *x)
{
        if(x != NULL)
        {
                printf("%d ", x->key);
                PreOrderTreeWalk(x->left);
                PreOrderTreeWalk(x->right);
        }
}
```

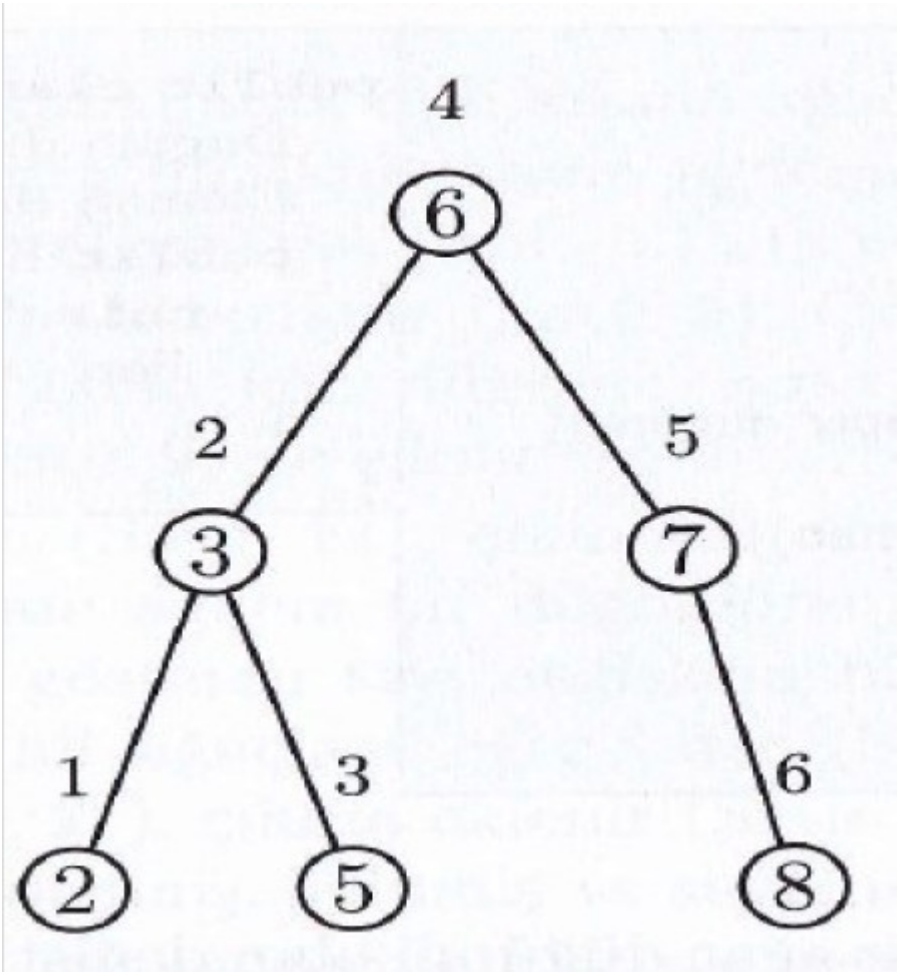


İkili arama ağaçları

- Ağaç üzerinde dolaşmak ve elemanlarını göstermek istediğimizde bunu farklı şekillde yapabiliriz:
  - Inorder (Kök ortada) Dolaşma

```
void InOrderTreeWalk(node *x)
{
        if(x != NULL)
        {
                InOrderTreeWalk(x->left);
                printf("%d ", x->key);
                InOrderTreeWalk(x->right);
        }
}
```

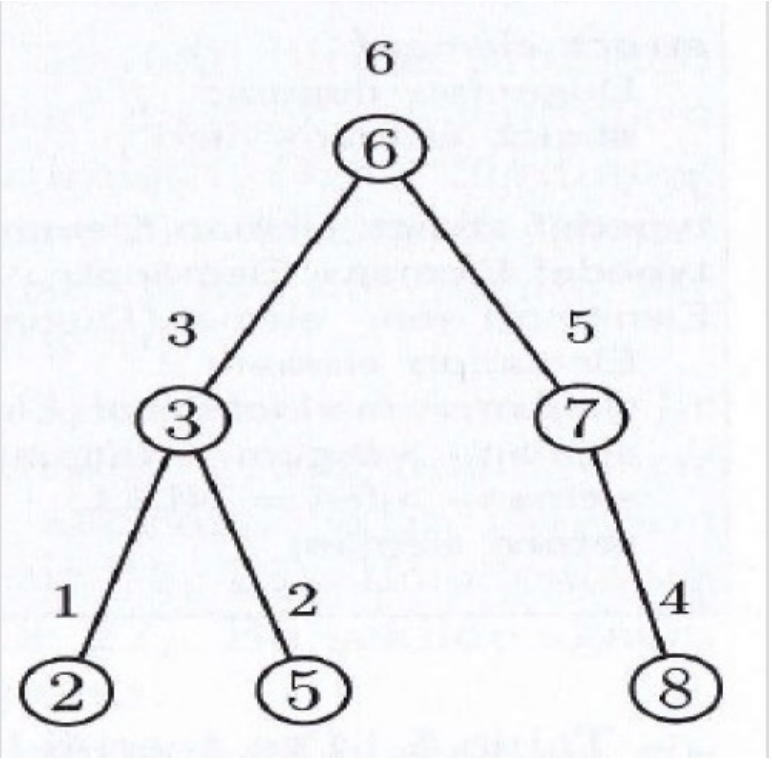
- Inorder dolaşma elemanları sıralar.



İkili arama ağaçları

- Ağaç üzerinde dolaşmak ve elemanlarını göstermek istediğimizde bunu farklı şekillde yapabiliriz:
  - Postorder (Sonra Kök) Dolaşma

```
void PostOrderTreeWalk(node *x)
{
        if(x != NULL)
        {
                PostOrderTreeWalk(x->left);
                PostOrderTreeWalk(x->right);
                printf("%d ", x->key);
        }
}
```

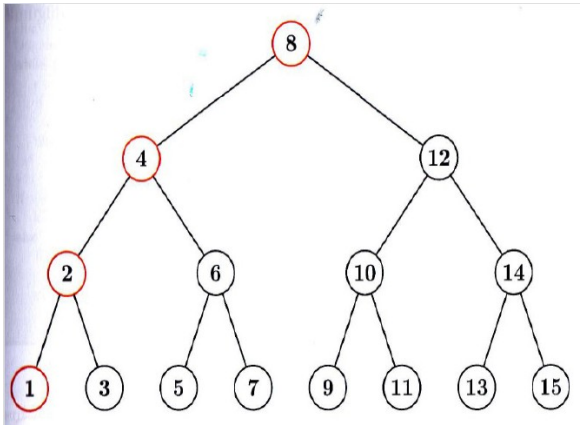


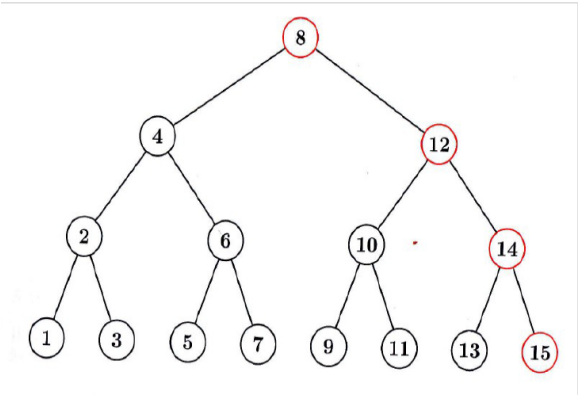
İkili arama ağaçları

- Ağaç üzerindeki minimum ve maksimum değerleri bulmak için aşağıda belirtilen fonksiyonları yazabiliriz.

```
node* TreeMinimum(node *x)
{
        while(x->left != NULL)
                x = x->left;
        return x;
}

node* TreeMaximum(node *x)
{
        while(x->right != NULL)
                x = x->right;
        return x;
}
```

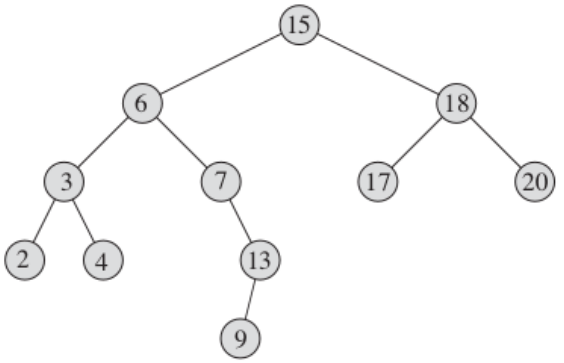




İkili arama ağaçları

- Ağaç üzerinde arama yapmak için aşağıda belirtilen fonksiyonu kullanabiliriz.

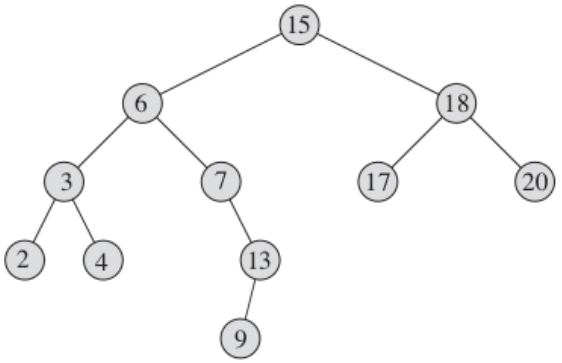
```
node* TreeSearch(node *x, int k)
{
        if(x == NULL || k == x->key)
        {
                return x;
        }
        if(k < x->key)
                return TreeSearch(x->left,k);
        else
                return TreeSearch(x->right,k);
}
```



İkili arama ağaçları

- Ağaç üzerinde arama yapmak için aşağıda belirtilen fonksiyonu kullanabiliriz.

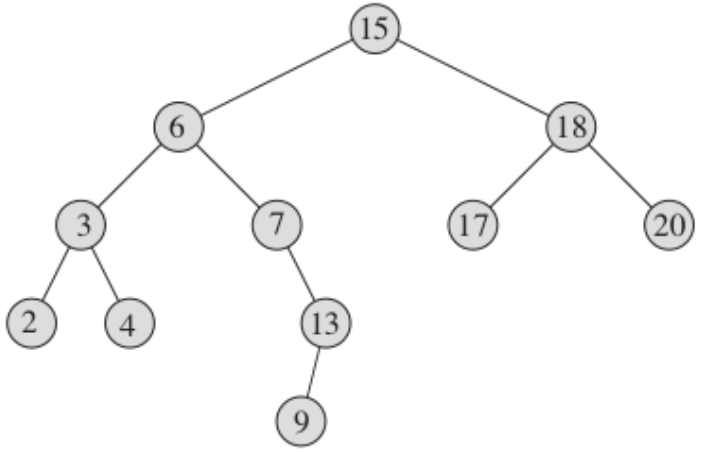
```
node* IterativeTreeSearch(node *x, int k)
{
        while(x != NULL && k != x->key)
        {
                if(k < x->key)
                        x = x->left;
                else
                        x = x->right;
        }
        return x;
}
```



İkili arama ağaçları

- Sonraki ve önceki değere sahip anahtara sahip düğümleri bulma
  - İkili arama ağacı özelliğinden yararlanarak karşılaştırma yapmadan sıralı durumda önceki ve sonraki değere sahip düğümleri bulabiliriz.

```
node* TreeSuccessor(node *x)
{
        if(x->right != NULL)
                return TreeMinimum(x->right);
        node* y = x->parent;
        while(y != NULL && x == y->right)
        {
                x = y;
                y = y->parent;
        }
        return y;
}
```

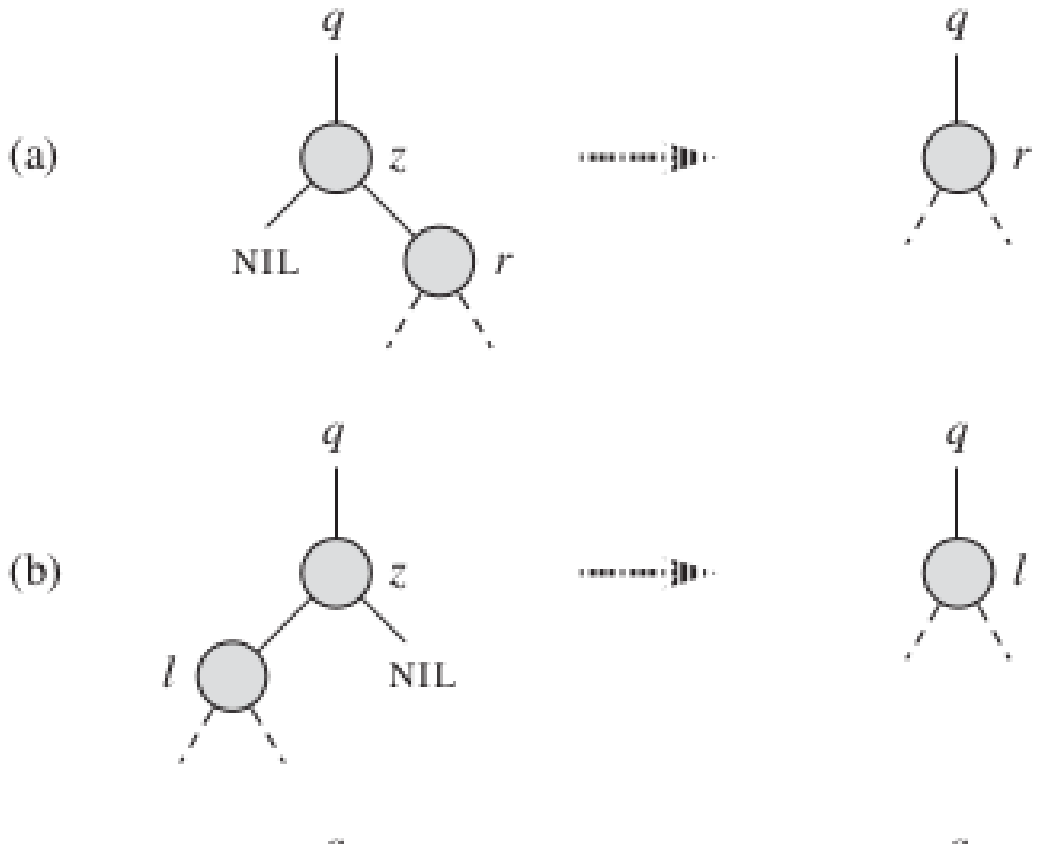


  - Eğer x düğümünün sağ alt ağacı var ise, sağ alt ağaçtaki en küçük değere sahip düğüm y sonrakidir.
  - Eğer x düğümünün sağ alt ağacı yok ise, sonraki düğüm y, sol altdüğümü x'in atası olan x'in atalarının en düşüğüdür.

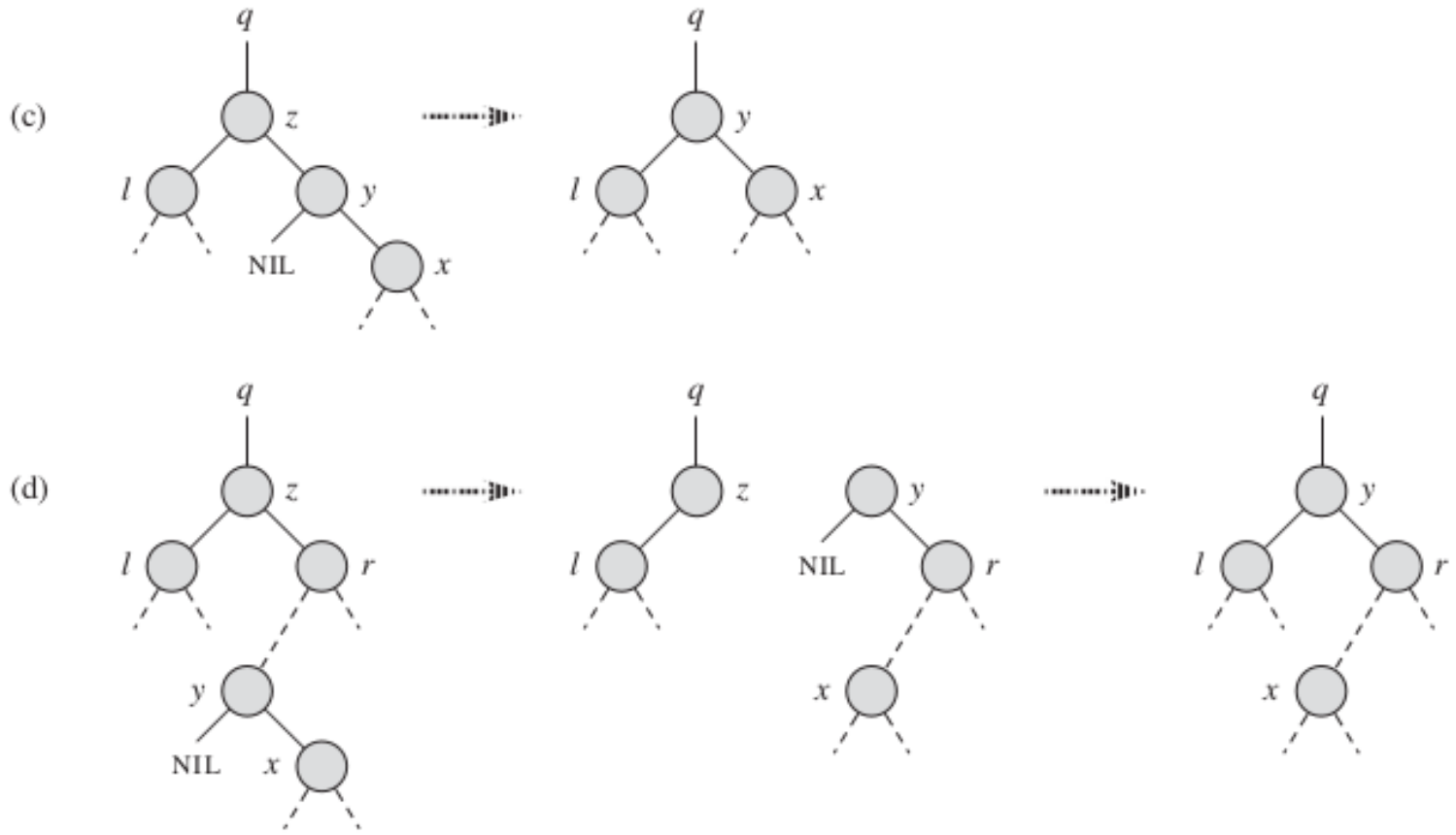
İkili arama ağaçları

- Silme
  - Silme işlemi, eklemeye göre daha karmaşıktır.
  - Silecek düğüm z olsun.
  - Silme yapılacak düğümün pozisyonuna göre üç faklı durum olabilir:
    - z düğümünün alt düğümleri yoksa, sadece bu düğüm silinir ve yerine NIL koyulur.
    - z düğümünün sadece bir alt düğümü var ise z düğümünün yerine z‘nin alt düğümü getirilir.
    - z düğümünün iki alt düğümü varsa, z‘nin sonraki düğümü y bulunur. Y düğümü z‘nin yerini alır. z‘nin sol alt düğümü y‘nin sol alt düğümü olur ve z‘nin sağ alt düğümü y‘nin sağ alt düğümü olur.

İkili arama ağaçları



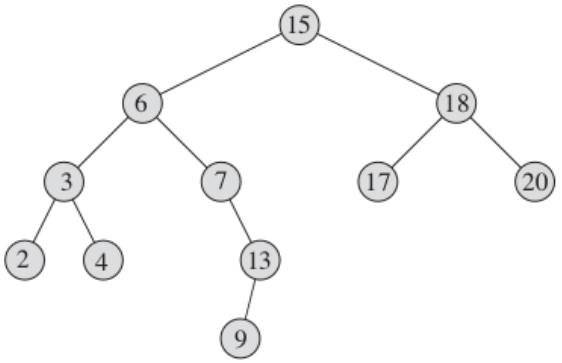
İkili arama ağaçları



İkili arama ağaçları

- Ağaç üzerinde iki düğüm arasında yer değiştirmek için aşağıdaki metot kullanılır

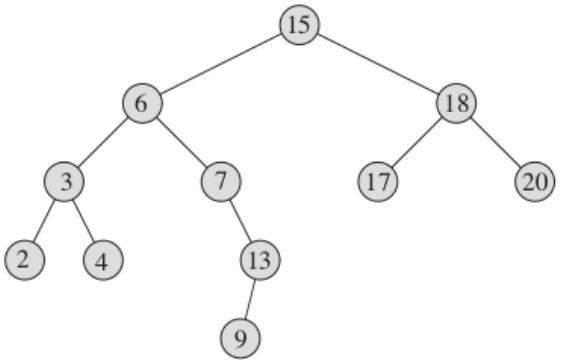
```
void Transplant(node *u, node *v)
{
        if(u->parent == NULL)
                root = v;
        else if(u == u->parent->left)
                u->parent->left = v;
        else
                u->parent->right = v;
        if(v != NULL)
                v->parent = u->parent;
}
```



İkili arama ağaçları

- Ağaç üzerinde bir düğümü silmek için aşağıdaki fonksiyon kullanılabilir

```
void TreeDelete(node *z)
{
        if(z->left == NULL)
                Transplant(z,z->right);
        else if(z->right == NULL)
                Transplant(z,z->left);
        else
        {
                node *y = TreeMinimum(z->right);
                if(y->parent != z)
                {
                        Transplant(y,y->right);
                        y->right = z->right;
                        y->right->parent = y;
                }
                Transplant(z,y);
                y->left = z->left;
                y->left->parent = y;
        }
}
```



Ağaç veri yapısı

- İkili arama ağacında gördüğümüz bütün operasyon ağacın boyu ile orantılı süre alıyor.
- n düğüme sahip dengeli bir ağaçta bu operasyonlar O(log n) süre alır.
  - Oldukça iyi bir çalışma zamanı.
- Ancak dengesiz bir ağacın bağlı listeye dönüşme ihtimali var.
  - n düğüme sahip bir sıralı listede birçok işlemin O(n) süre aldığını görmüştük.
- Öyleyse ağacın dengeli olduğunu garanti edebilirsek O(log n) çalışma süresine sahip olabiliriz.
- Bu amaçla oluşturulmuş özel ağaç türleri mevcuttur.
  - Örneğin AVL ağaçları, Kırmızı-siyah ağaçları