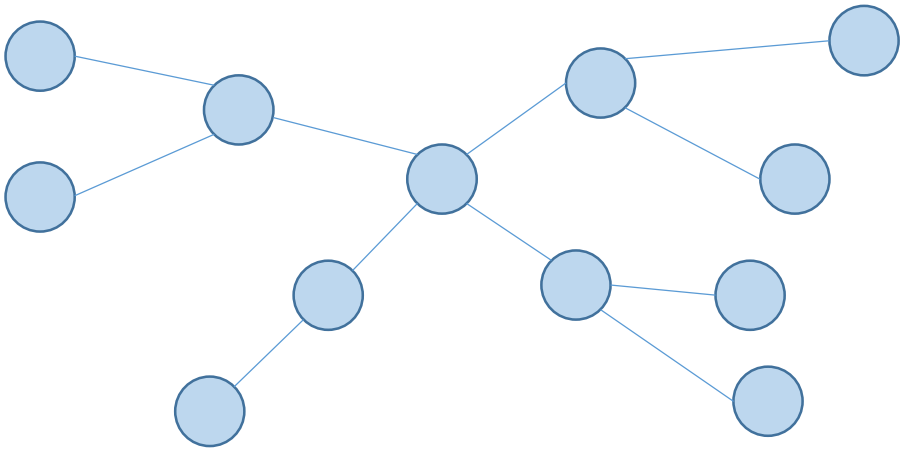
9 Ağaçlar (Trees)

9.1 Giriş

Ağaç, verilerin birbirine sanki bir ağaç yapısı oluşturuyormuş gibi sanal olarak bağlanmasıyla elde edilen hiyerarşik yapıya sahip bir veri modelidir; bilgisayar yazılım dünyasında, birçok yerde / uygulamada programcının karşısına çıkar. Ağaç veri yapılarının işletim sistemlerinin dosya sisteminde, oyunların olası hamlelerinde ve şirketlerdeki organizasyon şeması vb. gibi birçok uygulama alanları vardır. Örneğin, NTFS dosya sistemi hiyerarşik dosyalama sistemini kullanır. Yani bu sistemde bir kök dizin ve bu kök dizine ait alt dizinler bulunur. Bu yapı **parent-child** ilişkisi olarak da adlandırılır. Windows gezgininde dosyaların gösterilmesi esnasında ağaç veri yapısı kullanılır. Bunun sebebi astlık üstlük ilişkileridir. Arama algoritmalarında (**ikili arama – binary search**) kullanılabilmektedir. Daha değişik bir örnek verecek olursak, bir satranç tahtası üzerinde atın 64 hamlede tüm tahtayı dolaşması problemi ağaç veri yapısıyla çözülebilen bir problemdir. Birçok problemin çözümü veya modellenmesi, doğası gereği ağaç veri modeline çok uygun düşmektedir. Ağaç veri modeli daha fazla bellek alanına gereksinim duyar. Çünkü ağaç veri modelini kurmak için birden çok işaretçi değişken kullanılır. Buna karşın, yürütme zamanında sağladığı getiri ve ağaç üzerinde işlem yapacak fonksiyonların rekürsif yapıda kolayca tasarlanması ve kodlanması ağaç veri modelini uygulamada ciddi bir seçim yapmaktadır.

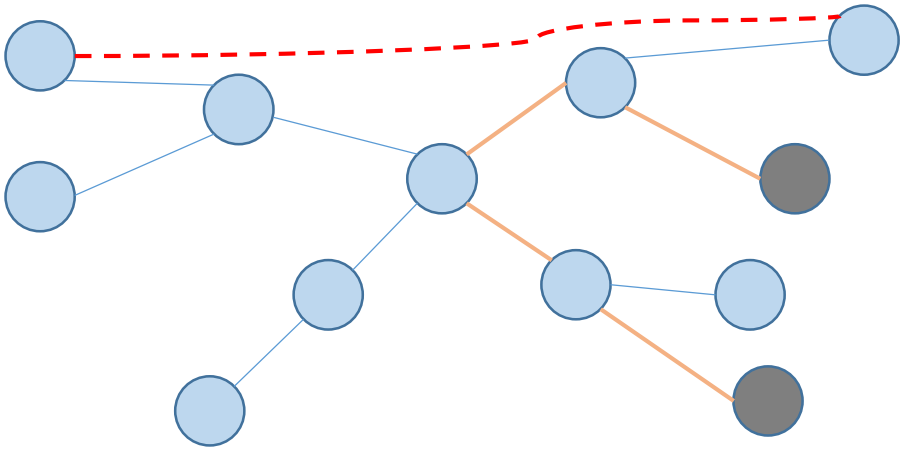
Ağaç veri yapısı **çizge** (**graph**) veri yapısının bir alt kümesidir. Bir ağaç (**tree**), **düğümler** (**node**) ve bu düğümleri birbirine bağlayan ayrıtlar (**edge**–**dal**-**kenar**) kümesidir. Her düğümü ve ayrıtları olan küme ağaç değildir. Bir çizgenin ağaç olabilmesi için her iki düğüm arasında sadece bir **yol** olmalı, **devre** (**cycle**, **çevrim**) olmamalıdır. **Yol** (**path**) birbirleri ile bağlantılı ayrıtlar (**edge**) dizisidir.



Şekil 9-1. Bir ağaç yapısının görünüşü

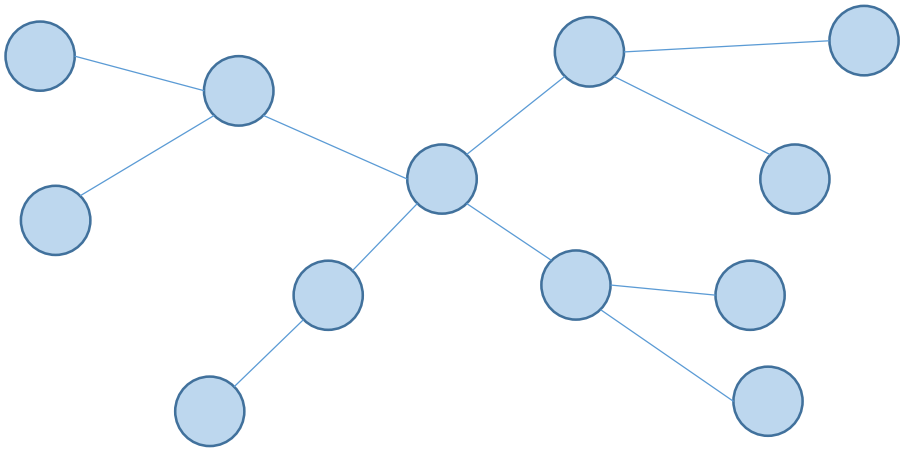
Şekil 9-1'de görülen yapı bir ağaçtır ve mutlaka bir ağaçta kök bulunmak zorunda değildir. Herhangi iki düğüm arasında yalnızca bir yol vardır ve bir çevrim içermemektedir.

Şekil 9-2'de koyu gri renkte gösterilmiş olan düğümler arasında yalnızca bir yol bulunmaktadır. Eğer kesikli kırmızı çizgi ile belirtilen bir çevrim (cycle) daha olsaydı, bu bir ağaç değil, sadece graf olacaktı. Çünkü düğümlere ikinci bir yol ile de erişebilme imkânı ortaya çıkmaktadır.



Şekil 9-2. Bir ağaçta herhangi iki düğüm arasında yalnızca bir yol bulunur

Kökü Olan Ağaç (Rooted Tree): Kökü olan ağaçta özel olan ya da farklı olan düğüm **kök** (**root**) olarak adlandırılır. Kök hariç her **c** (**child**) düğümünün bir **p** ebeveyni (**parent**) vardır. Alışılagelmiş şekliyle ağaçlarda, kök en yukarıda yer alıp çocuklar alt tarafta olacak biçimde çizilir. Fakat böyle gösterme zorunluluğu yoktur.



Şekil 9-3. Köklü ağaç (rooted tree) modeli

Ebeveyn (Parent): Bir c düğümünden köke olan yol üzerindeki ilk düğümdür. Bu c düğümü p'nin çocuğudur (child).

Yaprak (Leaf): Çocuğu olmayan düğümdür.

Kardeş (Sibling): Ebeveyni aynı olan düğümlerdir.

Ata (Ancestor): Bir d düğümünün ataları, d'den köke olan yol üzerindeki tüm düğümlerdir.

Torun (Descendant): Bir düğümün çocukları, torunları vs. gibi sonraki neslidir.

Yol Uzunluğu: Yol üzerindeki ayrıt sayısıdır.

n Düğümünün Derinliği: Bir n düğümünden köke olan yolun uzunluğudur. Kökün derinliği sıfırdır.

n Düğümünün Yüksekliği: Bir n düğümünden en alttaki descendant düğümüne olan yolun uzunluğudur. Başka bir deyişle neslinin en sonu olan düğümdür.

Bir Ağacın Yüksekliği: Kökün yüksekliğine ağacın yüksekliği de denir.

n Düğümünde Köklenen Alt Ağaç (Subtree Rooted at n): Bir n düğümü ve n düğümünün soyu (descendant) tarafından oluşan ağaçtır.

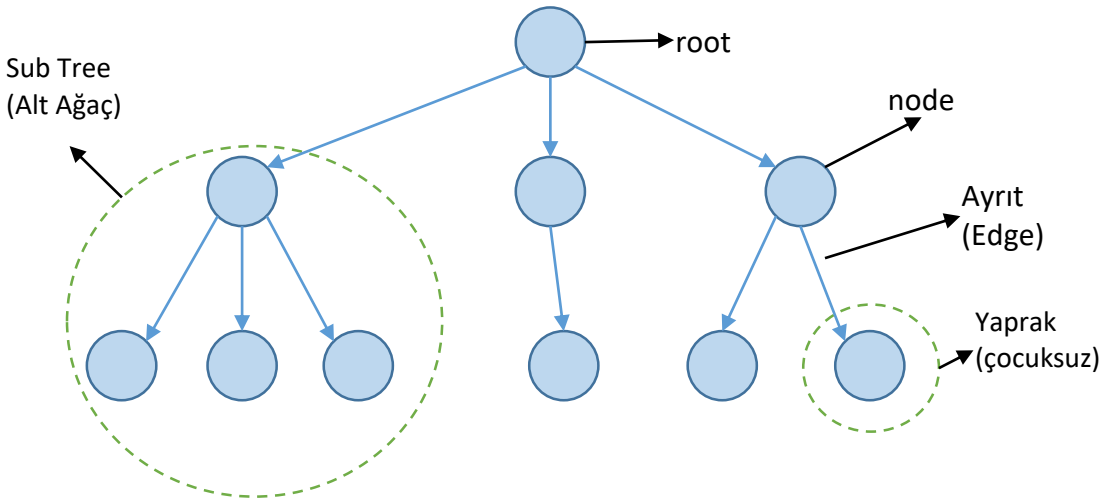
Binary Tree (İkili Ağaç): İkili ağaçta bir düğümün sol çocuk ve sağ çocuk olmak üzere en fazla iki çocuğu olabilir.

Düzey (level) / Derinlik (depth): Kök ile düğüm arasındaki yolun üzerinde bulunan ayrıtların sayısıdır. Bir düğümün kök düğümden olan uzaklığıdır. Kökün düzeyi sıfırdır (bazı yayınlarda 1 olarak da belirtilmektedir).

9.2 Ağaçların Temsili

Genel ağaç yapısında düğümlerdeki çocuk sayısında ve ağaç yapısında bir kısıtlama yoktur. Ağaç yapısına belli kısıtlamalar getirilmesiyle ağaç türleri meydana gelmiştir. Her yaprağın derinliği arasındaki fark belli bir sayıdan fazla (örneğin 2) olmayan ağaçlara **dengeli ağaçlar** (**balanced trees**) denir. İkili ağaçlar (binary trees) ise düğümlerinde en fazla iki bağ içeren (0, 1 veya 2) ağaçlardır. Ağaç yapısı kısıtlamaların az olduğu ve problemin kolaylıkla uyarlanabileceği bir yapı olduğundan birçok alanda kullanılmaktadır. Örnek olarak işletim sistemlerinde kullandığımız dosya-dizin yapısı tipik bir ağaç modellemesidir.

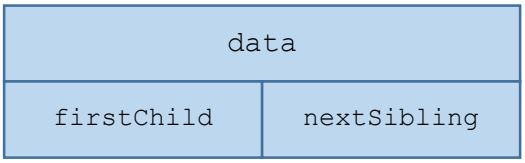
Ağaç veri modelinde, bir kök işaretçisi, sonlu sayıda düğümleri ve onları birbirine bağlayan dalları vardır. Veri ağacın düğümlerinde tutulur. Dallarda ise geçiş koşulları vardır. Her ağacın bir kök işaretçisi bulunmaktadır. Ağaca henüz bir düğüm eklenmemiş ise ağaç boştur ve kök işaretçisi `NULL` değerini gösterir. Ağaç bu kök etrafında dallanır ve genişler.



Şekil 9-4. Ağacın genel yapısı.

9.2.1 Tüm Ağaçlar İçin

Binary ağaçlarda bir sol çocuk, bir de sağ çocuk bulunur. Bu çocuklara **left** ve **right** değerleri ile ulaşılır. 2'den fazla çocuğu olan düğümler için aşağıdaki gibi bir veri yapısı tanımlamalıyız:

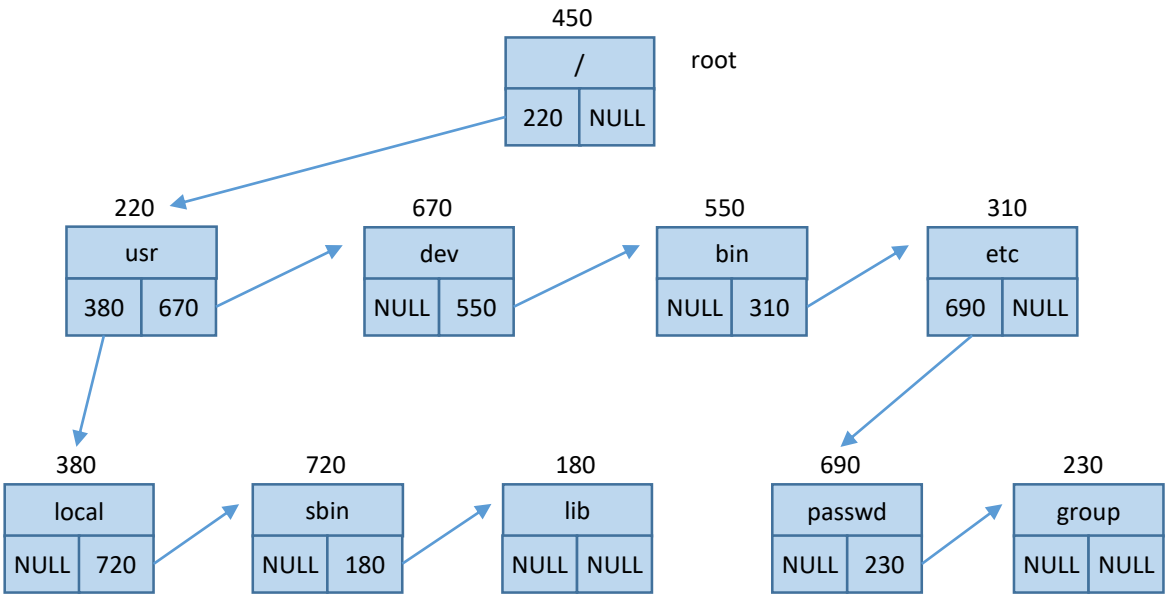


Şekil 9-5. İkili olmayan ağaçlarda bir node'un yapısı.

Kök'ün kardeşi yoktur, sadece çocukları ve torunları olabilir. İkili olmayan ağaçların gösteriminde her ebeveyn yalnızca ilk çocuğunu göstermeli ve eğer varsa kendi kardeşini de göstermelidir. Kardeşler birden fazla olabilir. Öyleyse ilk çocuktan sonra kardeşler birbirlerine bağlanmalıdır. Bu açıklamalara uygun veri yapısı aşağıdaki gibi yazılabilir. Burada da yine ihtiyaç halinde bir struct node *parent göstericisi tanımlanabilir.

```
struct node {
     int data;
     struct node *firstChild; // ilk çocuk
     struct node *nextSibling; // kardeş
};
```

Kök düğümün kardeşi olamayacağı için yalnızca `firstChild` bağlantısı var gibi görülse de diğer kardeşlerin düğümüne `firstChild` işaretçisinin gösterdiği ilk çocuk düğümünün `nextSibling` işaretçisinden erişilebilir. Bu yöntem bellek alanından kazanç sağlar; ancak, programın yürütme zamanını arttırır. Çünkü bir düğümden onun çocuklarına doğrudan erişim ortadan kalkmış, bağlı listelerde olduğu gibi ardışıl erişim ortaya çıkmıştır. Aşağıda, UNIX dosya yapısının mantıksal modellemesi gösterilmiştir.



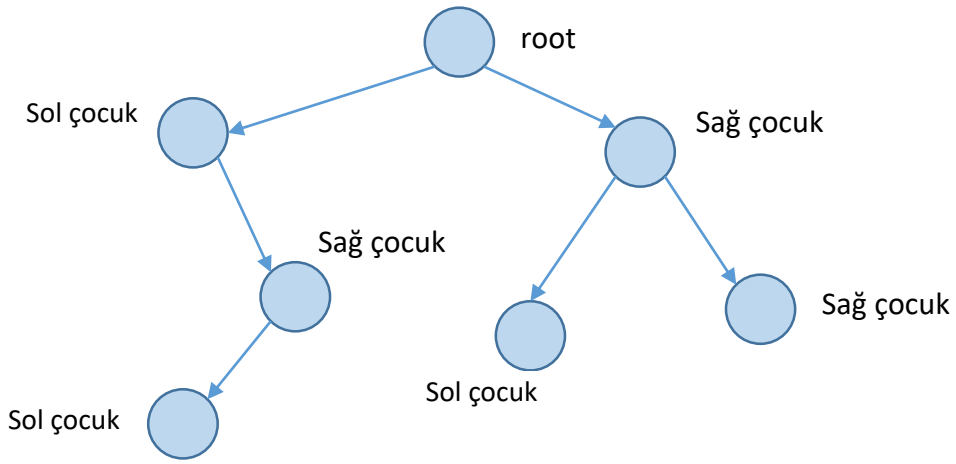
Şekil 9-6. UNIX işletim sistemindeki dosya yapısının veri modeli.

9.2.2 İkili Ağaçlar (Binary Trees) İçin

Eğer bir ağaçtaki her düğümün en fazla 2 çocuğu varsa bu ağaca ikili ağaç denir. Diğer bir deyişle bir ağaçtaki her düğümün derecesi en fazla 3 ise o ağaca ikili ağaç denir. İkili ağaçların önemli özelliği eğer dengeli ise çok hızlı bir şekilde ekleme, silme ve arama işlemlerini yapabilmesidir. Ağaç veri yapısı üzerinde işlem yapan fonksiyonların birçoğu kendi kendini çağıran (**recursive**) fonksiyonlardır. Çünkü bir ağacı alt ağaçlara böldüğümüzde elde edilen her parça yine bir ağaç olacaktır ve fonksiyonumuz yine bu ağaç üzerinde işlem yapacağı için kendi kendini çağıracaktır.

İkili ağaç veri yapısının tanımında ağaç veri yapısının tüm tanımları geçerlidir, farklı olan iki nokta ise şunlardır:

i. Ağacın derecesi (**degree of tree**) 2'dir.
ii. Sol ve sağ alt ağacın yer değiştirmesiyle çizilen ağaç aynı ağaç değildir.



Şekil 9-7. İkili ağaçların gösterimi.

İkili ağaç (binary tree) veri yapısının yoğun biçimde kullanılır oluşu iki önemli uygulama alanı ile ilişkilidir. Sıralama algoritmaları ile derleyicilerde sözdizim çözümleme (syntax analysis), kod geliştirme (code optimization) ve amaç kod üretme (object code generation) algoritmaları bu veri yapısını kullanırlar. Ayrıca kodlama kuramı (coding theory), turnuva düzenleme, aile ağacı gösterimi ve benzerleri tekil kullanımlar da söz konusudur.

Ağaçlarla ilgili özellikler:

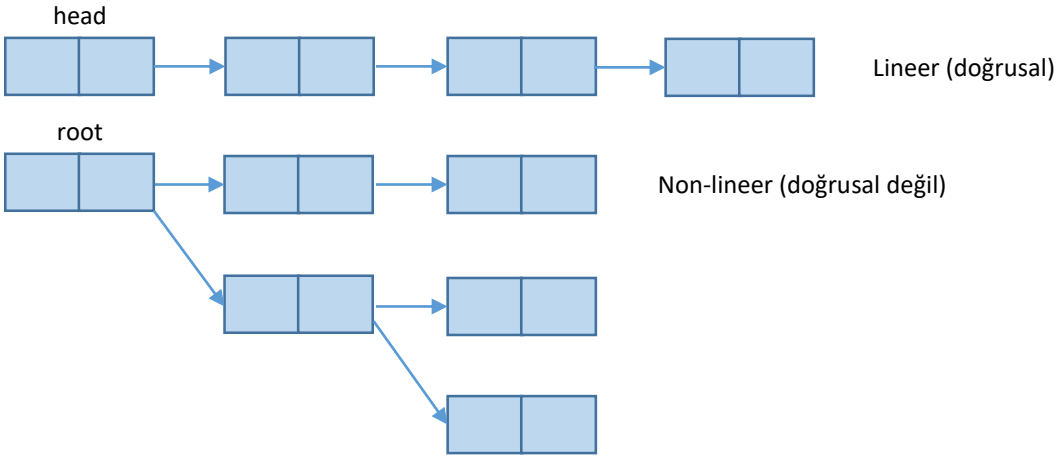
- Bir binary ağaçta, bir *i* seviyesindeki maksimum node sayısı 2^i 'dir.
- *K* derinliğine sahip bir binary ağaçta maksimum node sayısı $2^{K+1} - 1$ 'dir.

- Seviye ve derinlik kökle başlayacak şekilde aynı sayılmaktadır.
- Ağaçta `node`'ların eksik kısımları `NULL` olarak adlandırılır.

İkili bir ağaç için veri yapısı aşağıdaki gibi tanımlanabilir;

```
struct node {
     int data;
     struct node *left;
     struct node *right;
};
```

İhtiyaç olduğunda yapı içerisinde bir de **parent** tanımlanabilir. Ağaçların yapısının da bağlı liste olduğu tanımdan anlaşılabilir. Fakat normal bağlı listelerden farkı **nonlinear** olmasıdır. Yani doğrusal değildir.



Şekil 9-8. Ağaçlarla bağlı listelerin karşılaştırılması.

9.2.3 İkili Ağaçlar Üzerinde Dolaşma

İkili ağaç üzerinde dolaşma birçok şekilde yapılabilir Ancak, rastgele dolaşmak yerine, önceden belirlenmiş bir yönteme, bir kurala uyulması algoritmik ifadeyi kolaylaştırır. Üstelik rekürsif fonksiyon yapısı kullanılırsa ağaç üzerinde işlem yapan algoritmaların tasarımı kolaylaşır. **Önce-kök** (**preorder**), **kök-ortada** (**inorder**), **kök-sonda** (**postorder**) olarak adlandırılan üç değişik dolaşma şekli çeşitli uygulamalara çözüm olmaktadır.

9.2.3.1 Preorder (Önce Kök) Dolaşma

Önce kök yaklaşımında ilk olarak **root**(**kök**), sonra **left** (**sol alt ağaç**) ve ardından **right** (**sağ alt ağaç**) dolaşılır. Fonksiyonu tanımlamadan önce kolayca kullanabilmek için `typedef` bildirimiyle bir yapı nesnesi oluşturulabilir. Bildirim `typedef struct node *BTREE;` şeklinde asterisk (*) konularak da yapılabilir. Böylece her `BTREE` türünde geriye adres döndüren fonksiyonların ve adres değişkenlerinin bildiriminde asterisk işaretinden kurtulunmuş olunur fakat bazı derleyicilerin kararsız çalışmasına neden olabilmektedir.

```
typedef struct node BTREE;
```

`BTREE` yapısı `struct node` veri yapısıyla tanımlanmış olan bir çeşit node'dur. İçerisinde veri tutan `data` değişkeni, adres bilgisi tutan `left` ve `right` isimli iki işaretçisi bulunmaktadır. İkili ağaçta dolaşırken fonksiyon geriye herhangi bir şey döndürmeyeceği ve sadece ağaç üzerinde dolaşacağı için türü `void` olmalıdır.

```
void preorder(BTREE *root) {
     if(root != NULL) {
          printf("%d", root -> data);
          preorder(root -> left);
          preorder(root -> right);
     }
}
```

9.2.3.2 Inorder (Kök Ortada) Dolaşma

Inorder dolaşmada önce **left** (**sol alt ağaç**), sonra **root** (**kök**) ve **right** (**sağ alt ağaç**) dolaşılır. Fonksiyonun türü yine `void` olmalıdır.

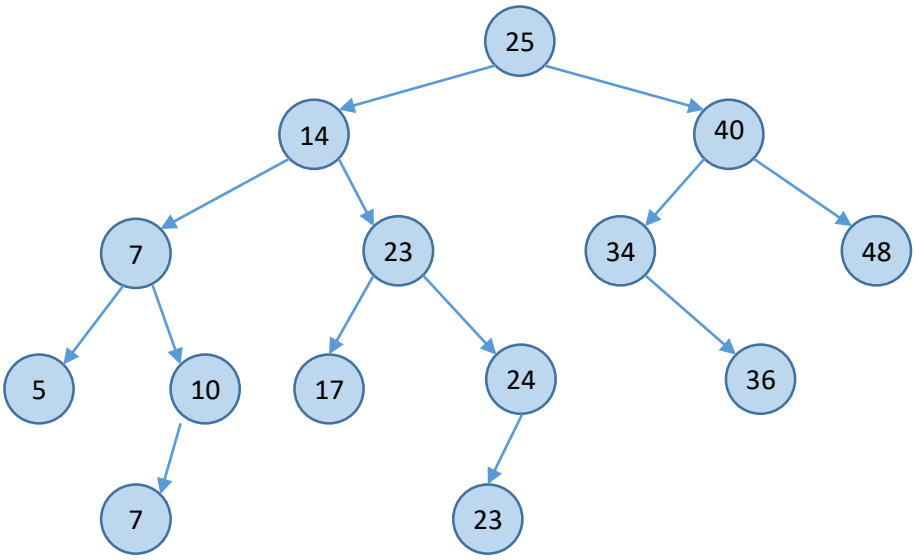
```
void inorder(BTREE *root) { if(root != NULL) { inorder(root -> left);
     printf("%d", root -> data); inorder(root -> right); } }
```

9.2.3.3 Postorder (Kök Sonda) Dolaşma

Postorder yaklaşımında ise, önce **left** (**sol alt ağaç**), sonra **right** (**sağ alt ağaç**) ve **root** (**kök**) dolaşılır. Fonksiyon `void` türdendir.

```
void postorder(BTREE *root) { if(root != NULL) { postorder(root -> left);
     postorder(root -> right); printf("%d", root -> data); } }
```

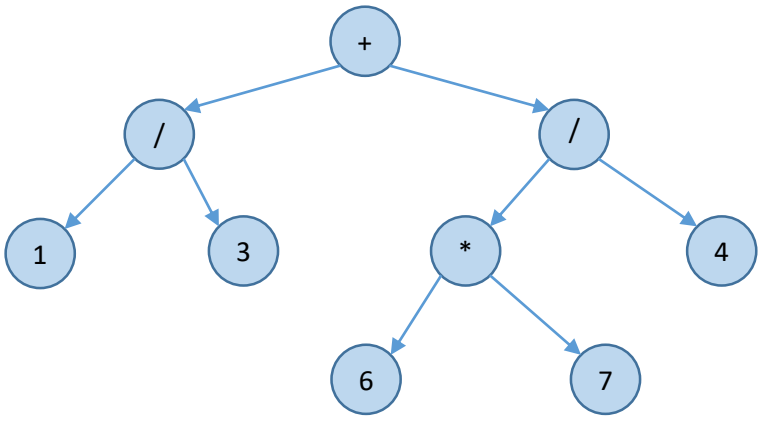
Örnek 19: "25, 14, 23, 40, 24, 23, 48, 7, 5, 34, 10, 7, 17, 36" değerlerine sahip düğümler için ikili ağaç gösterimini oluşturunuz ve üç farklı `preorder`, `inorder` ve `postorder` sıralama yöntemine göre yazınız.



Çözüm:

```
Preorder  : 25-14-7-5-10-7-23-17-24-23-40-34-36-48
Inorder   : 5-7-7-10-14-17-23-23-24-25-34-36-40-48
Postorder : 5-7-10-7-17-23-24-23-14-36-34-48-40-25
```

Örnek 20: Aşağıda görülen bağıntı ağacındaki verileri preorder, inorder ve postorder sıralama yöntemine göre yazınız.



Çözüm:

```
Preorder  : + / 1 3 / x 6 7 4
Inorder   : 1 / 3 + 6 x 7 / 4
Postorder : 1 3 / 6 7 x 4 / +
```

9.2.4 İkili Ağaç Oluşturmak

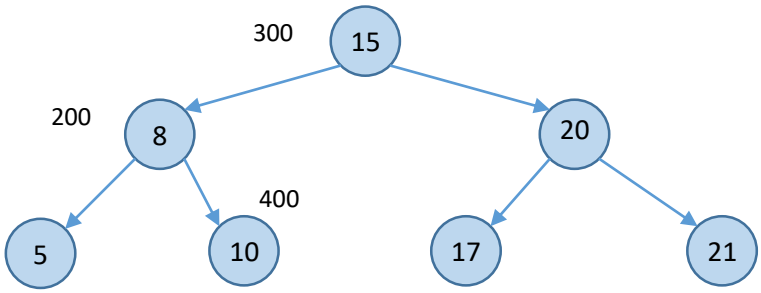
```
typedef struct node BTREE;
BTREE *new_node(int data) {
     BTREE *p = (BTREE*) malloc(sizeof(BTREE));
     // BTREE *p = new node(); // C++'ta bu şekilde
     p -> data = data;
     p -> left = NULL;
     p -> right = NULL;
     return p;
}
```

9.2.5 İkili Ağaca Veri Eklemek

İkili bir ağaca veri ekleme işleminde sol çocuğun verisi (data) ebeveyninin verisinden küçük olmalı, sağ çocuğun verisi ise ebeveyninin verisinden büyük ya da eşit olmalıdır. Altta insert isimli ekleme fonksiyonunun nasıl yazıldığını görüyorsunuz. Fonksiyon, veri eklendikten sonra ağacın kök adresiyle geri dönmektedir.

```
/* İkili ağaca veri ekle yen fonksiyon */
BTREE *insert(BTREE *root, int data) {
// Fonksiyona gönderilen adresteki ağaca ekleme yapılacak
     if(root != NULL) { // ağaç boş değilse
          if(data < root -> data) // eklenecek veri root'un data'sından
     küçükse
               root -> left = insert(root -> left, data);
               // eklenecek veri root'un data'sından büyük ya da eşitse
          else
               root -> right = insert(root -> right, data);
     }
     else // eğer ağaç boş ise
          root = new_node(data);
     return root;
}
```

Fonksiyonun nasıl çalıştığını bir şekille açıklayalım. Şekil 9-9'da görülen ağaçta bazı düğümlerin adresleri üzerlerinde belirtilmiştir.



Şekil 9-9. Veri eklenecek örnek bir ağaç

Ağaca 13 sayısının eklenecek olduğunu kabul edelim. Fonksiyon `insert(300, 13);` kodu ile çağrılacaktır.

```
if(root != NULL) { // 300 adresinde bir ağaç var, yani NULL değil ve
     koşul doğru
     if(data < root -> data) // 13 root'un data'sı olan 15'ten küçük ve
     koşul doğru
          root -> left = insert(root -> left, data); // yürütülecek satır
     else root -> right = insert(root -> right, data);
}
else
     root = new_node(data);
return root;
```

13, kökün değeri olan 15'ten küçük olduğu için sol çocuk olan 8 verisinin bulunduğu 200 adresiyle tekrar çağrılıyor.

```
insert(200, 13);
if(root != NULL) { // 200 adresi NULL değil ve koşul doğru
     if(data < root -> data) // 13, 8'den küçük mü, değil ve koşul yanlış
          root -> left = insert(root -> left, data);
     else // 13, 8'den büyük mü, evet ve koşul doğru
          root -> right = insert(root -> right, data); // yürütülecek
     satır
}
else
     root = new_node(data);
return root;
```

Bu defa 13, 8'den büyük olduğundan fonksiyon 8'in sağ çocuğu olan 10 verisinin bulunduğu 400 adresiyle çağrılıyor.

```
insert(400, 13);
if(root != NULL) { // 400 adresi NULL değil ve koşul doğru
     if(data < root -> data) // 13, 10'dan küçük mü, değil ve koşul
     yanlış
          root -> left = insert(root -> left, data);
```

```
     else // 13, 10'dan büyük mü, evet ve koşul doğru
          root -> right = insert(root -> right, data); // yürütülecek
     satır
}
else
     root = new_node(data);
return root;
```

Şimdi de fonksiyon `10`'un sağ çocuğu ile tekrar çağrılıyor. Fakat `root->right` (sağ çocuk) düğümü henüz olmadığı için adres ile değil `NULL` ile fonksiyon çağrılacaktır.

```
insert(NULL, 13);
else // eğer ağaç boş ise yani düğüm yok ise
     root = new_node(data); // düğüm oluşturuluyor ve 13 ekleniyor
return root; // oluşturulan ve veri eklenen düğümün adresi geri
     döndürülüyor
```

`400` adresindeki `10` datasını bulunduran düğümün sağ çocuğu olan `right`, `NULL` değere sahiptir. Yani herhangi bir düğümü göstermemektedir, çocukları yoktur. Fonksiyonun en sonunda geri döndürülen yeni düğümün adresi, `10`'un sağ çocuğunu gösterecek olan ve `NULL` değere sahip right işaretçisine atanıyor. Sonra fonksiyon kendisinden önceki çağrıldığı noktaya geri dönüyor. Daha önce `stack` konusu ve **rekürsif** fonksiyonların çalışma prensibi anlatılmıştı. Fonksiyon, her geri dönüşte işleyiş anındaki `root`'un adresini geri döndürmektedir.

Örnek 21: Klavyeden – 1 girilinceye kadar ağaca ekleme yapan programın kodunu yazınız.

```
#include <stdio.h>
#include <conio.h>
struct node {
     int data;
     struct node *left;
     struct node *right;
};
typedef struct node BTREE;
BTREE *new_node(int data) {
     BTREE *p = (BTREE*) malloc(sizeof(BTREE));
     p -> data = data;
     p -> left = NULL;
     p -> right = NULL;
     return p;
}
BTREE *insert(BTREE *root, int data) { // root'u verilmiş ağaca ekleme
     yapılacak
     if(root != NULL) {
          if(data < root -> data)
```

```
            root -> left = insert(root -> left, data);
        else
            root -> right = insert(root -> right, data);
    }
    else
        root = new_node(data);
    return root;
}
void preorder(BTREE *root) {
    if(root != NULL) {
        printf("%3d ", root -> data);
        preorder(root -> left);
        preorder(root -> right);
    }
}
void inorder(BTREE *root) {
    if(root != NULL) {
        inorder(root -> left);
        printf("%3d ", root -> data);
        inorder(root -> right);
    }
}
void postorder(BTREE *root) {
    if(root != NULL) {
        postorder(root -> left);
        postorder(root -> right);
        printf("%3d ", root -> data);
    }
}
int main() {
    BTREE *myroot = NULL;
    int i = 0;
    do {
        printf("\n\nAgaca veri eklemek icin sayi giriniz...\nSayi =
");
        scanf("%d", &i);
        if(i != -1)
            myroot = insert(myroot, i);
        } while(i != -1);
    preorder(myroot);
    printf("\n");
    inorder(myroot);
    printf("\n");
    postorder(myroot);
    getch();
    return 0;
}
```