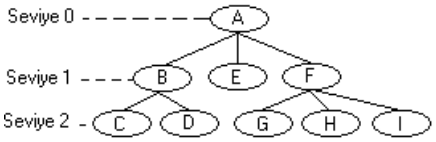
AĞAÇ VERİ YAPISI

Ağaç veri yapısında elemanlar birbirlerine liste veri yapısında olduğu gibi pointerlar aracılığı ile bağlandığı bir veri yapısıdır. Ağaç yapılarında liste yapılarının aksine her elemanın birden fazla elemanla bağlantısı olabilir. Ağaç veri yapısında bir elemanın en fazla kaç elemanla bağlanabileceği o ağaç veri yapısının derecesini belirtmektedir. Aşağıda örnek bir ağaç veri yapısı görülmektedir.



Şekilde de görüldüğü gibi her ağaç veri yapısında kendisinden önce hiçbir eleman bulunmayan elemana ağacın **kökü** (root) adı verilir. Kendisinden sonra hiçbir eleman bulunmayan elemanlara ise ağacın **yaprakları** (leaves) ismi verilir. Örnekte A değerine sahip eleman ağacın kökü, C, D, E, G, H, I elemanları ise ağacın yapraklarıdır.

Ağaç veri yapısında herhangi bir elemandan sonra gelen elemanlar o elamanın **çocukları** (children), herhangi bir elemandan önce yer alan eleman ise o elemanın **velisi** (parent) olarak isimlendirilir. Ağaç veri yapısının oluşması için her elemanın bir tek velisi olmalıdır.

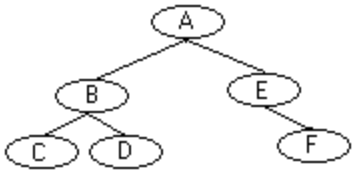
Bir node'un **seviyesi** onun kökten olan mesafesi ile ölçülür. Kök'ün kendine olan uzaklığı 0 olduğu için kök seviyesi 0 dır. Kök'ün çocuklarının köke mesafesi 1 (seviye1), onun çocuklarının köke uzaklığı 2 şeklinde devam eder.

Bir ağaç **alt ağaçlara** bölünebilir. Bu bölümlere alt ağaç ismi verilir. Alt ağaçlarda kendi içlerinde alt ağaçlara bölünebilir. Şekilde BCD, E, ve FGHI birer alt ağaçtır. Bu alt ağaçlardan B alt ağacı C ve D alt ağaçlarına, F alt ağacı G, H, ve I alt ağaçlarına ayrılmıştır.

İkili Ağaçlar

Her bir node u 2 alt ağaçtan daha fazla node'a sahip olmayan bir ağaç yapısıdır. Başka bir ifadeyle her bir node 0, 1 veya 2 alt ağaçtan fazlasına sahip olamaz. Bu alt ağaçlar sol alt ağaç ve sağ alt ağaç şeklinde dizayn edilir. Şekilde ikili ağaç örneği görülmektedir. Şekilde de görüldüğü gibi ağaç yapısının simetrik olması önemli değildir.

İkili ağacın taranması: ikili arama ağacında tutulan verileri, kullanılan erişim sırasına göre 3 farklı şekilde taramak mümkündür.



a) Preorder Tarama: Bu tarama şeklinde önce kök, daha sonra sol alt ağaç ve en son sağ alt ağaç taranır. Bu tarama şekline göre önce kök (A) dan başlanır, sonra sol alt ağacın kökü olan B tarandıktan sonra sol alt ağaç C tamamlanır. Sol alt ağacın taranmasından sonra sağ tarafa (D) geçilir. Sol alt ağaç tamemen tarandıktan sonra kökün sağ tarafı taranmaya başlanır. Sağ tarafında E alt ağaç kökü, E kökünün sol alt ağacı olmadığı için sağ alt ağaç olan F taranarak işlem tamamlanır. A, B, C, D, E, F.

b) inorder tarama: Bu tarama şeklinde önce sol alt ağaç, kök ve en son olarak sağ alt ağaç dolaşılır. C,B,D,A,E,F.

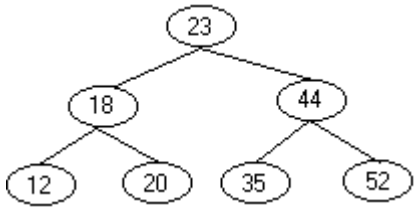
c) postorder tarama: sol alt ağaç, sağ alt ağaç ve son olarak kök şeklinde tarama yapılır. C,D,B,F,E,A

Breadth First Traversal: Bu yöntemde yeni bir seviyeye geçmeden önce o seviyedeki tüm çocukların dolaşılması gerekmektedir. Buna göre A,B,E,C,D,F şeklinde bir tarama yapılır.

İKİLİ ARAMA AĞACI (Binary Search Tree)

Etkili bir arama algoritmasıdır. İkili arama ağacı aşağıdaki özellikleri sağlamalıdır.

- Sol alt ağaçtaki tüm elemanlar kök değerinden küçük olmalıdır.
- Sağ alt ağaçtaki bütün elemanlar kök değerine büyük eşit olmalıdır.
- Her alt ağacın kendi bir ikili arama ağacı olmalıdır.



ikili arama ağacı algoritmasında ağacın taranması ikili ağaçtaki gibidir.

Preorder tarama:	23	18	12	20	44	35	52
Postorder tarama:	12	20	18	35	52	44	23
Inorder tarama:	12	18	20	23	35	44	52

İkili ağacın inorder yöntemiyle dolaşılması sonucunda ikili ağaç elemanları **sıralanmış** olur.

İkili Arama Ağacı (ekleme, Listeleme,Arama,En büyük eleman, en küçük eleman)

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Agac
{
      int eleman;
      struct Agac *sag, *sol;
}Agaclar;

Agaclar *kok=NULL;

Agaclar *DegerAl()
{     Agaclar *yeni;
      yeni=(Agaclar *)malloc(sizeof(Agaclar));
      if(yeni==NULL)
      {     puts("\nBellek dolu...");
            return NULL;
       }
       puts("\n Elemani Giriniz...: ");
       scanf("%d",&(yeni->eleman));
       yeni->sag=NULL;
       yeni->sol=NULL;
       return yeni;
}

void ekle(Agaclar *yeni) // döngüsel olarak ekleme
{     Agaclar *geçici, *parent;
      if(kok==NULL) kok=yeni; // ilk eklenen eleman kök düğümdür
      else
      {     gecici= kok;
            while(gecici!=NULL)
            {     parent= gecici;
                  if (yeni->eleman < gecici->eleman)  // küçükse
                        gecici= gecici->sol;
                  else                                // büyükse
                        gecici= gecici->sag;
            }
            if (yeni->eleman < parent->eleman)
                  parent->sol= yeni;
            else
                  parent->sag= yeni;
      }
}

void ekle_rec(Agaclar *agackok, Agaclar *yeni) // recursive olarak ekleme
{     if(kok==NULL) // ilk eklenen eleman kök düğümdür
            kok= yeni;
      else
      {     if(yeni->eleman<agackok->eleman)    // küçükse
            {     if(agackok->sol==NULL) // sol link boş ise ekle
                        agackok->sol= yeni;
                  else // sol link dolu ise aynı fonksiyonu sol link ile çağır
                        ekle_rec(agackok->sol, yeni);
             }
             else                                 // büyükse
             {    if(agackok->sag==NULL) // sağ link boş ise ekle
                        agackok->sag= yeni;
                  else // sağ link dolu ise aynı fonksiyonu sağ link ile çağır
                        ekle_rec(agackok->sag, yeni);
             }
      }
}
```