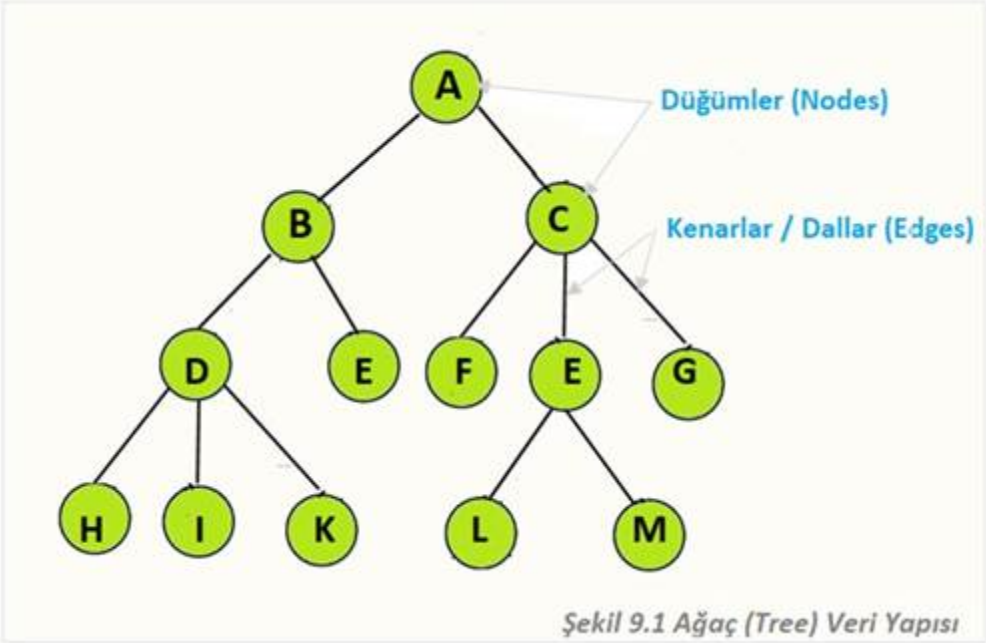
9. AĞAÇ VERİ YAPISI

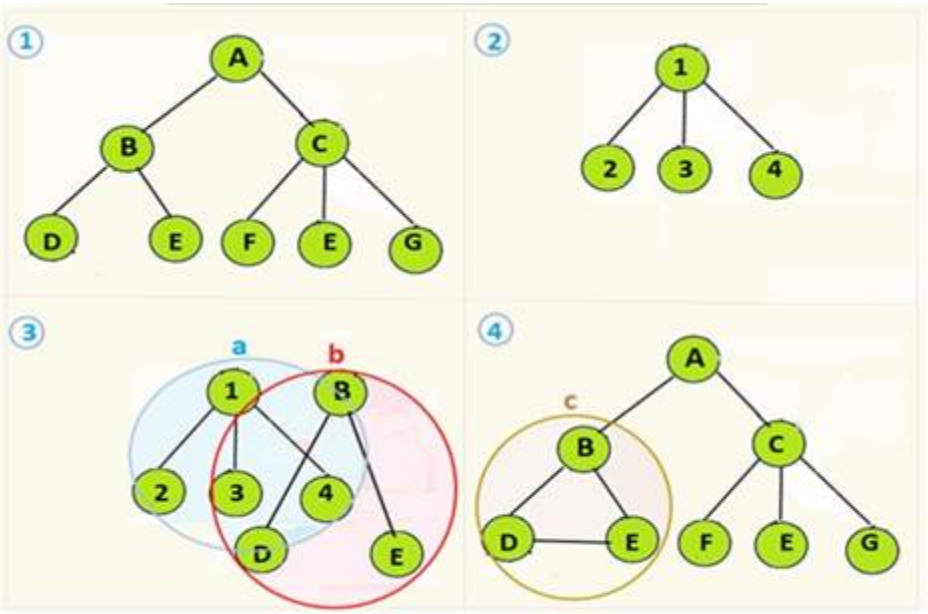
Giriş

Ağaç (Tree) veri yapısı çok yaygın olarak kullanılan çok güçlü bir veri yapısıdır. Ağaçlar (Trees) doğrusal (lineer) veri yapıları olan diziler, bağlantılı listeler, yığınlar ve kuyruklardan farklı olarak, doğrusal olmayan hiyerarşik bir veri yapısıdır. Gündelik hayattan bildiğimiz soy ağacı ağaç veri yapısı hakkında bize fikir verebilir. Ağaç, verilerin birbirine sanki bir ağaç yapısı oluşturuyormuş gibi sanal olarak bağlanmasıyla elde edilir. Bu yapıda veri, düğümlerde (node) tutulur. Düğümlere ağcın elemanı denir. Ağaç veri yapısında düğümler arası ilişki kenarlar /dallar (edges) kullanılarak oluşturulur. Başka bir ifade ile ağaç veri yapısında düğümler birbirine kenarlar/dallar kullanılarak bağlanır. Şekil 9.1' de Ağaç yapısı gösterilmiştir.



Şekil 9.1 Ağaç (Tree) Veri Yapısı

Şekil 9.1'den de anlaşılabileceği gibi ağaç sonlu sayıda düğümleri ve düğümleri birbirine bağlayan kenarları /dalları olan ve de döngü içermeyen bir veri yapısıdır.



Şekil 9.2 Ağaç ve Ağaç Olmayan Örnekler

Şekil 9.2'de 1 numaralı ve 2 numaralı şekiller hiyerarşik bir yapıya sahiptir. Kendi içlerinde birbirileri ile kenarları / dalları aracılığı ile bağlıdırlar ve yapılarında döngü oluşmamıştır. Dolayısı ile 1 (bir) ve 2 (iki numaralı şekiller Ağaç örneğidir.

3 (üç) numaralı şekle gelince a ve b bölümleri arasında kenarlar / dallar arasında bir bağlantı kurulmamıştır. Bundan dolayı şekil bir ağaç örneği değildir.

4 (dört) numaralı şekilde ise (c) bölümünde döngü oluşmuştur. Bu nedenle bu şekil de ağaç veri yapısına örnek teşkil etmez.

9.1. Ağaç Veri Yapısının Temel Kavramları

Ağaç veri yapısının elemanlarına **düğüm (node)** adı verilir. Şekil 9.3'te verilen ağaç yapısının A, B, C, D, E...M ile gösterilen dairelerin her biri birer düğümdür.

Bu hiyerarşik yapımın en tepesindeki düğüme ***kök (root)*** düğüm adı verilir. Kök düğümünden önce ağaçta herhangi bir düğüm bulunmaz.

Ağaç veri yapısında iki düğümü birleştiren bağlantılara ***kenar /dal*/edge** denir.

Çocuğu olan düğümlere ***baba (parent)*** *denir*. Şekil 9.3 'te H, I, K düğümlerinin babası D düğümüdür.

Düğüme bağlı olan alt düğümlere ***çocuk (child)*** *denir.* C düğümün çocukları F, E ve G dir.

Çocuğu olmayan düğümlere ***yaprak (leaf)*** denir. Şekil 9.3' te D1, G, H, I, K, L, M yaprak düğümlerdir.

Bir düğümün alt ağaçlarına ***subtree denir.***

Aynı babaya sahip düğümlere ***kardeş düğüm (sibling, brother)*** denir.

Bir düğüme bağlı tüm alt düğümlere o düğümün ***varisleri (descendant)*** denir.

Bir düğümden köke kadar izlenen yoldaki diğer tüm düğümler, o düğümün ***atalarıdır (ancestor)***.

Yol/iz/path, bir düğümün aşağıya doğru (çocukları üzerinden) bir başka düğüme gidebilmek için üzerinden geçilmesi gereken düğümlerin listesidir.

Kök düğümün en uçtaki yaprak düğüme olan uzaklığına ***Ağacın derinliği (depth of tree)*** denir. Şekil 9.3 'te gösterilen ağacın derinliği 3 (üç) 'tür. Kök düğümün derinliği 1 dir.

Bir düğümün kök düğümüne olan uzaklığına o ***düğümün derinliği*** denir. D düğümünün derinliği 2, H düğümünün derinliği 3'tür.

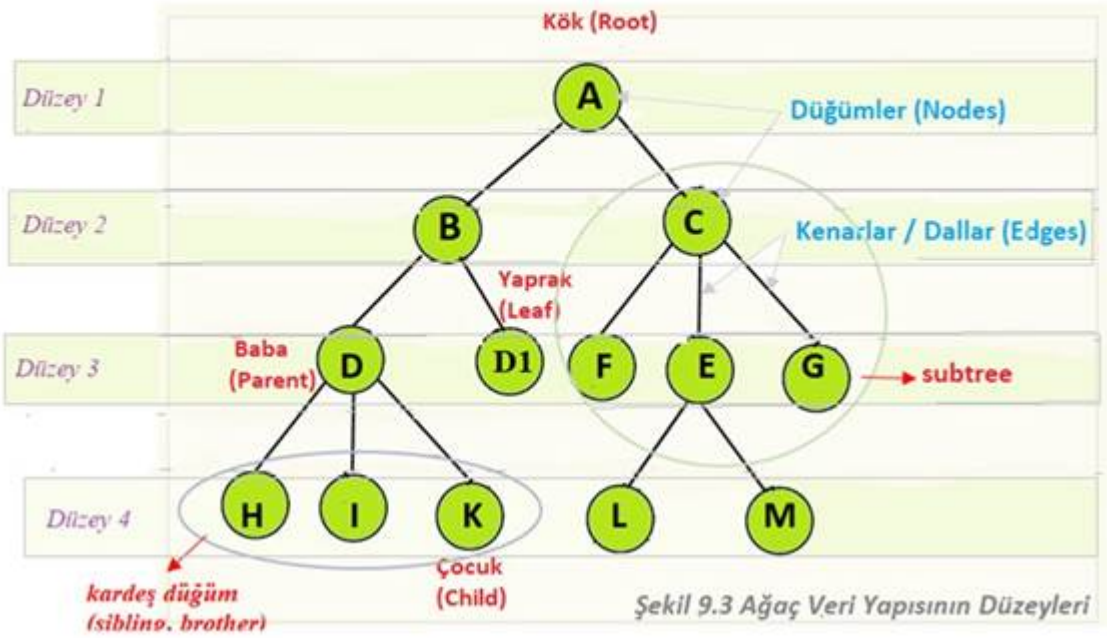
Düğümün **height**/yüksekliği, o düğümden kendisiyle ilişkili en uzak yaprak düğüme kadar giden yolun uzunluğudur.

Düzey, iki düğüm arasındaki yolun üzerinde bulunan düğümlerin sayısıdır. Kök düğümün düzeyi 1, doğrudan köke bağlı düğümlerin düzeyi 2'dir.

Bir düğümün **degree**/derecesi, çocuk sayısına eşittir.

Bir ağacın hiç düğümü yoksa boş/**empty tree** olarak adlandırılır ve şekil 9.4 'teki gibi gösterilir.

Şekil 9.4 Boş Ağaç



Şekil 9.3 Ağaç Veri Yapısının Düzeyleri

Tablo 9.1 *Şekil 9.3'e göre tabloya değerler işlenmiştir.*

	Root (A)	D	L
Derece/Çocuk Sayısı	2	3	0
Düzey	1	3	4
Derinlik	1	3	4
Ebeveyn/Parent	–	B	E
Kardeş	–	D1	M
Ata	–	A	C,A
Çocuk	B, C	H, I, K	–
Path	A	A, B, D	A,C, E, L

Ağaç veri yapısı birçok alanda çok kullanılan bir veri yapısıdır. Örneğin dosya /dizin işlerimde, oyun programlamada, veri tabanlarında, yapay zekâ uygulamalarında ve vb. ağaç veri yapısı kullanılmaktadır. Bu bölümde yukarıda sayılan ve sayılmayan alanlarda daha yaygın olarak kullanılan ikili ağaçlar (Binary Tree) anlatılacaktır.

9.2. Ağaç Veri Yapısı Türleri

En çok bilinen ağaç veri yapısı türleri İkili Arama Ağacı (Binary Search Tree), Kodlama Ağacı (Coding Tree), Sözlük Ağacı (Dictionary Tree), Kümeleme Ağacı (Heep Tree) ve Bağıntı Ağacı (Expressin Tree) dir. Her biri farklı veri yapısında olan bu ağaç türleri üzerinde koşacak arama, ekleme ve silme gibi algoritmalar da farklıdır. Bu başlık altında yukarıda sözü edilen ve en çok bilinen bu veri ağaç yapıları hakkında özet bilgi aktarılacaktır.

9.2.1. İkili Arama Ağacı (Binary Search Tree)

İkili Arama ağacında düğümler 0 (sıfır), 1 (bir), 2 (iki) çocuğa sahip olabilir, daha fazla çocuğa sahip olamaz. Ayrıca bu ağaç veri yapısında alt düğümlerin (çocukların) bağlantıları belirli bir sırada yapılır. Örneğin önce küçük veya alfabetik olarak küçük olanlar sola, eşit ve büyük olanlar sağa bağlanır.

9.2.2. Kodlama Ağacı (Coding Tree)

Kodlama ağaç veri yapısı genel olarak bir kümedeki karakterlere, örneğin alfabedeki karakterlere kod atanması için kurulan bir ağaç şeklidir. Kodlama ağaçlarının en bileneni *Huffman Kodlaması* ‘dır. Ancak başka benzer birçok kodlama ağacı da vardır. Kodlama ağaçlarında kökten başlanıp yapraklara kadar olan yol üzerindeki bağlantı değerleri kodu vermektedir.

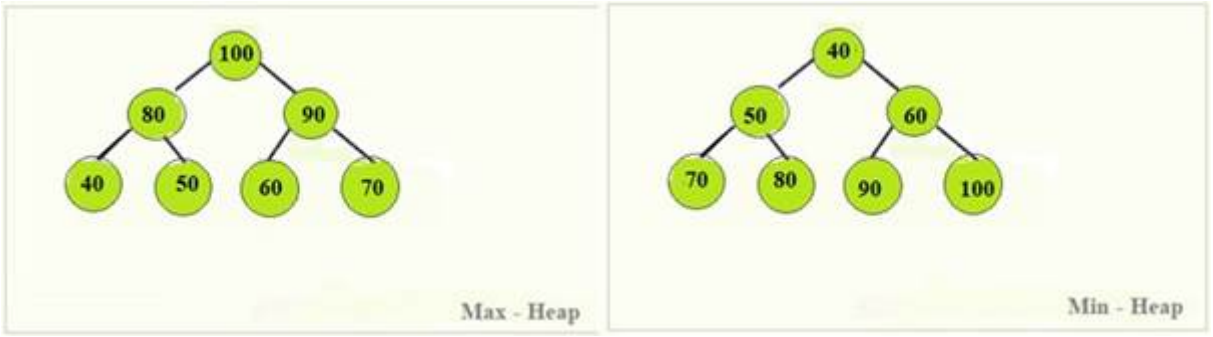
9.2.3. Sözlük Ağacı (Dictionary Tree)

Sözlük ağacı, adından da anlaşılacağı gibi bir sözlükte bulunan sözcüklerin tutulması için kurulan bir ağaç türüdür. Bu ağacın kurulmasındaki amaç veriler üzerinde arama işlemlerinin hızlı yapılması ve bilgisayar belleğinin verimli kullanılmasıdır. Sözlük ağaçları sözlüğün oluşturulabilmesi için bir araya gelip sözlük ormanını oluşturur. Bu ormanda alfabedeki karakter sayısı kadar ağaç vardır.

9.2.4. Kümeleme Ağacı (Heap Tree)

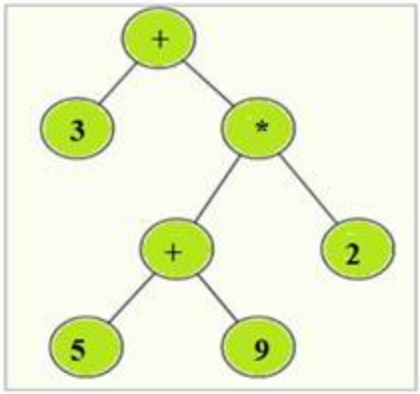
Kümeleme ağacının iki farklı türü vardır. Birincisi, çocuk düğümlerin her durumda aile düğümünden daha küçük değerlere sahip olduğu kümeleme ağaç türüdür. Bu ağaç türünde kök düğüm en büyük değere sahipken yaprak düğümler en küçük değere sahip olurlar. Bu ağaç türüne Max Heap denir.

İkincisi ise çocuk düğümlerin her durumda aile düğümünden daha büyük değerlere sahip olduğu kümeleme ağaç türüdür. Bu ağaç türünde kök düğüm en küçük değere sahipken yaprak düğümler en büyük değere sahip olurlar. Bu ağaç türüne Min Heap denir.



9.2.5. Bağıntı Ağacı (Expressin Tree)

Bağıntı ağaçları bir ikili ağaç uygulamasıdır ve matematiksel bir bağıntının ağaç şeklinde tutulması için tanımlanmıştır. Bağıntı ağacının yapraklarında değişken veya sabit değerler tutulurken kök düğüm ve iç düğümlerde operatörler tutulur. Aşağıda *3 + ((5 + 9) * 2)* ifadesinin bağıntı ağacı gösterilmiştir.

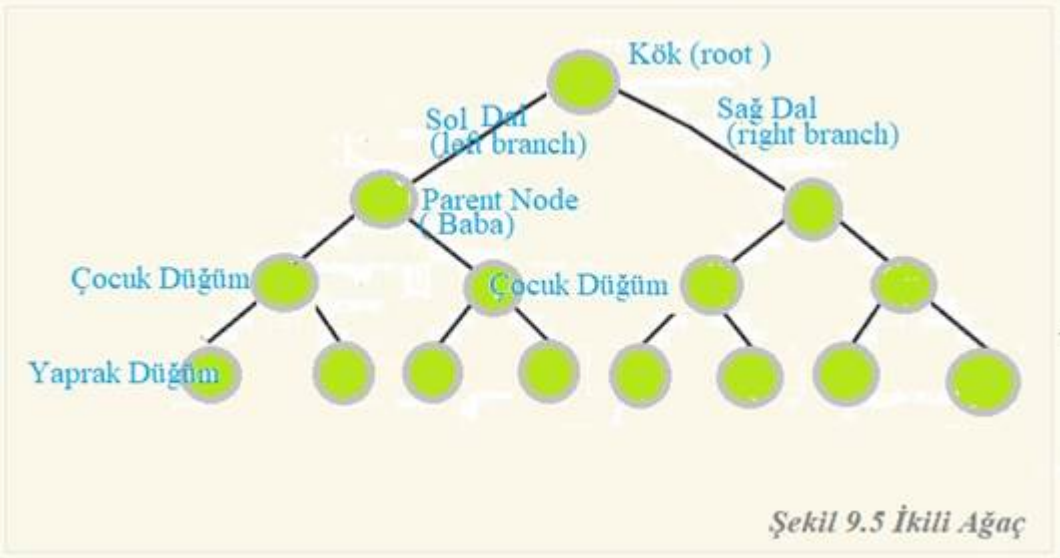


9.3. İkili Ağaçlar (Binary Trees)

İkili ağaçlar (Binary Trees) her düğümünün en fazla iki çocuk alabildiği ağaç veri yapısıdır. Buradan hareketle ikili ağaçlarda her düğümün 0

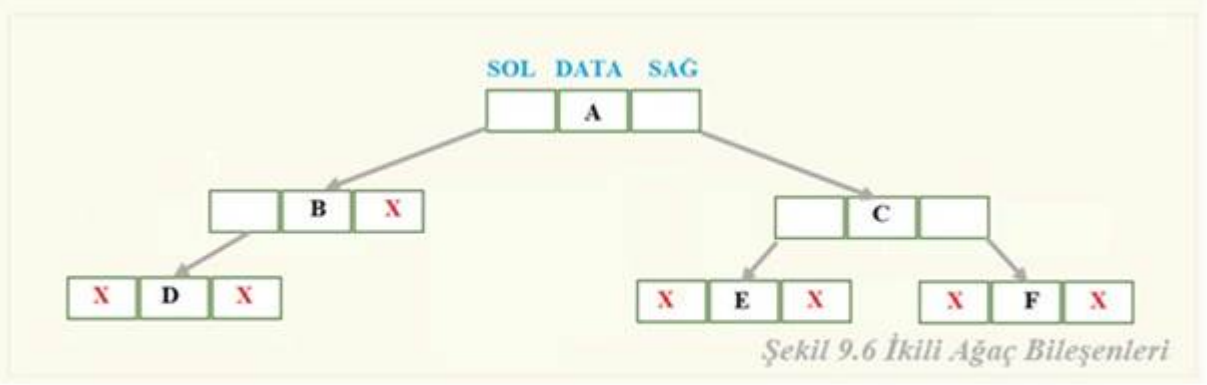
veya 1 veya 2 çocuğu olabileceğini söyleyebiliriz. İkili ağaçtaki her düğüm, veri öğesi ile birlikte bir sol ve sağ referansa sahiptir. Bu ağaçlar özellikle arama işlemlerinde daha iyi sonuç alabilmek için üretilmiştir. İkili ağaçların arama performansları dizilerden daha kötü, bağlı listelerden daha başarılıdır. Buna karşılık ikili ağaçlar ekleme (Insert) ve Silme (Delete) işlemlerinde dizilerden daha iyi sonuç vermektedirler. İkili ağaçlarda eleman eklenmesi ve silinmesi için bir sınır yoktur. Bu yönüyle ikili ağaçlar bağlı listelere benzer. Şekil 9.5'te ikili ağaç gösterilmiştir.

İkili ağaçlar veriler organize edilirken, hiyerarşik veriler üzerinde işlem yapılırken, arama performansına ihtiyaç duyulduğu zaman, yön bulma algoritmalarında, sıralı elemanları düzenleme gibi işlemlerde kullanılır.



Şekil 9.5 İkili Ağaç

İkili ağaçların düğümlerinin 3 (üç) bileşeni vardır. İkili ağacın bütün düğümleri bu bileşenlere sahiptir. Bileşenlerin ilki veri öğesi (Data element), ikincisi sol alt ağacı işaret eden işaretçi ve üçüncüsü sağ alt ağacı işaret eden işaretçidir. Şekil 9.6 'da ikili ağaç (Binary Tree) bileşenleri gösterilmiştir.



Şekil 9.6 İkili Ağaç Bileşenleri

İkili Ağaçlarda (Binary Tree) her k seviyesinde yer alabilecek maksimum düğüm sayısı: ***maksimum düğüm sayısı*** 2^k 'dir.

Kök düğümde k=0 olduğu için düğüm sayısı 1 (bir) dir.

k=2 için maksimum düğüm sayısı 4 'tür.

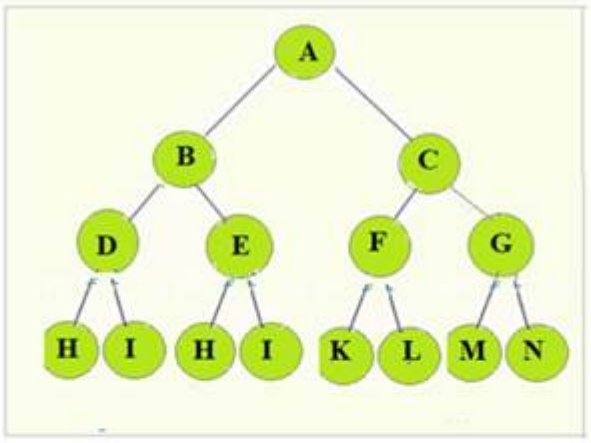
h yüksekliğindeki ikili ağaçta yer alabilecek maksimum düğüm sayısı $2^{h+1} - 1$ ile hesaplanır.

Aşağıda verilen ikili ağaçta maksimum sayıda düğüm yer almaktadır. Verilen ağacın yüksekliği h=3 'tür. Yukarıda verilen formül yardımıyla ağaçta yer alabilecek maksimum düğüm sayısını hesaplayalım:

Yüksekliği h=3 olarak verilen ikli ağaçta yer alabilecek maksimum düğüm sayısını hesaplayınız?

Maksimum Düğüm Sayısı = $2^{3+1} - 1$
$= 16 - 1$
$= 15$

Şekildeki düğümleri sayarak elde ettiğiniz sonucu formül yardımıyla hesapladığınız sonuçla karşılaştırınız?



En az iki düğüme sahip, n elemanı olan ikili ağaçta, **kenar sayısı = n -1** formülü ile hesaplanır. Buna göre yukarıda verilen ağacın kenar sayısını hesaplayınız.

Ağaç taki düğüm sayısı

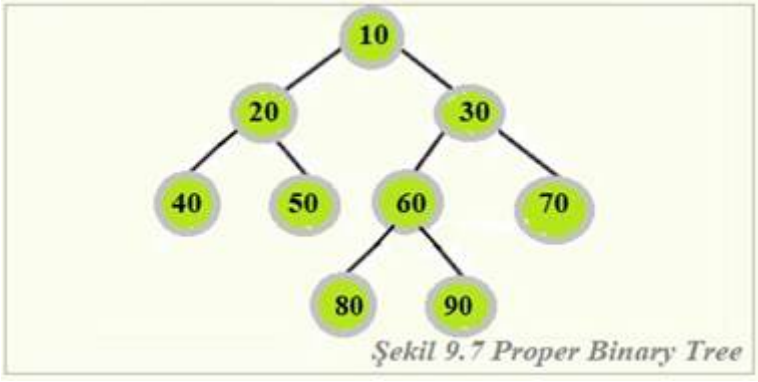
kenar sayısı = 15 - 1
= 14

Kenar Sayısı

9.3.1. İkili Ağaç Türleri

9.3.1.1. Düzgün İkili Ağaç (Proper Binary Tree)

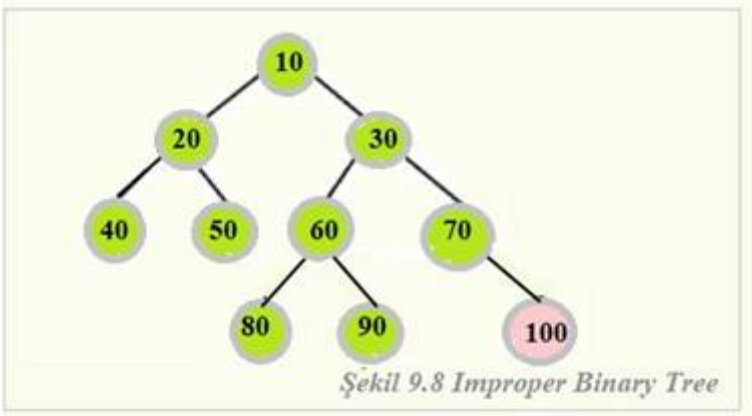
Yaprak olmayan düğümlerinin tümünde iki çocuk olan ağaçtır. Yaprak düğümler aynı derinliğe sahip olmak zorunda değildir. Düzgün ikili ağaç şekil 9.8'de gösterilmiştir.



Şekil 9.7 Proper Binary Tree

9.3.1.2. Düzgün Olmayan İkili Ağaç (Improper Binary Tree)

Yaprak olmayan düğümlerin tümünün iki çocuğu olmaması durumunda oluşan ağaçlardır. Düzgün olmayan ikili ağaç şekil 9.8'de gösterilmiştir.

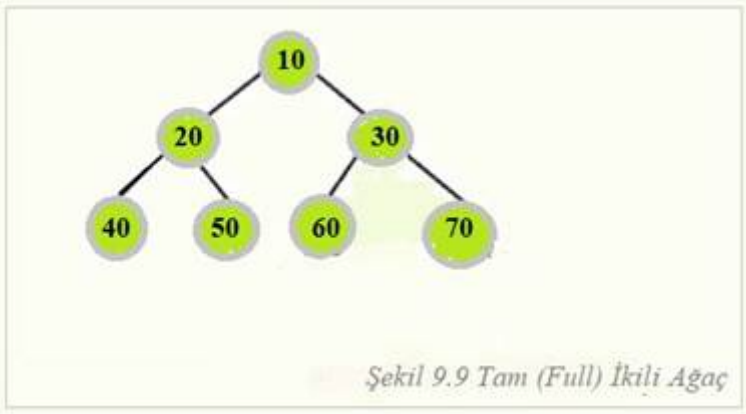


Şekil 9.8 Improper Binary Tree

9.3.1.3. Tam İkili Ağaç (Full Binary Tree)

Her yaprağı aynı derinlikte olan, yaprak olmayan düğümlerin tümünün iki çocuğu olan ağaç Tam İkili Ağaç (Full Binary Tree) 'dir. Tam ikili ağaçların her yaprağı aynı derinlikte olmalıdır. Her bir tam İkili ağaç Proper ‘dir. Bunun tersi doğru değildir. Yani Her Proper ikili ağaç tam ikili ağaç değildir.

Şekil 9.9 ‘den de görülebileceği tam ikili ağaçlarda her düğüm eşit şekilde sol ve sağ alt ağaçlara sahiptir.



Şekil 9.9 Tam (Full) İkili Ağaç

Tam (Full) ikili ağaçlarda yaprak sayısı biliniyorsa ağaç üzerindeki düğüm sayısı hesaplanabilir. Bu hesaplama n yaprak sayısı olmak üzere Düğüm Sayısı = 2n – 1 formülü ile yapılır. Örneğin şekil 9.9 ‘da verilen ağaçta dört yaprak vardır yani n =4 ‘tür. Buna göre ağaçtaki düğüm sayısı:

Düğüm sayısı = 2*4 – 1

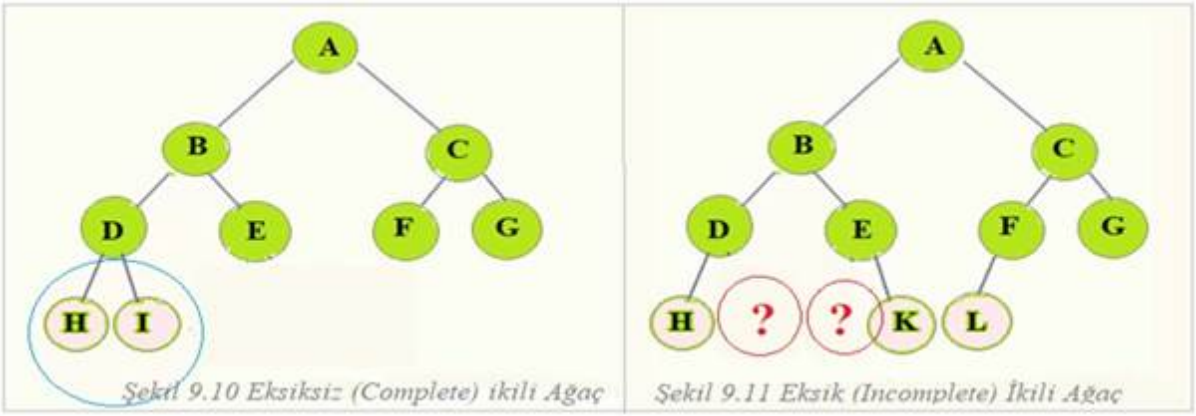
=8 – 1

=7 olarak hesaplanır.

Şekle baktığımızda da ağaçta 7 (yedi düğüm olduğu görülmektedir.

9.3.1.4. Eksiksiz İkili Ağaç (Complete Binary Tree)

Eksiksiz ikili ağaçlar (Complete Binary Tree) son seviye hariç tüm seviyelerin tam dolu olduğu ikili ağaç türüdür. Eksiksiz ikili ağaç şekil 9.10'da gösterilmiştir. Eksiksiz ikili ağaçların düğümleri sol taraftan başlanarak doldurulur. Düğümler yeni bir derinliğe soldan sağa doğru eklenir.



Şekil 9.10 Eksiksiz (Complete) ikili Ağaç

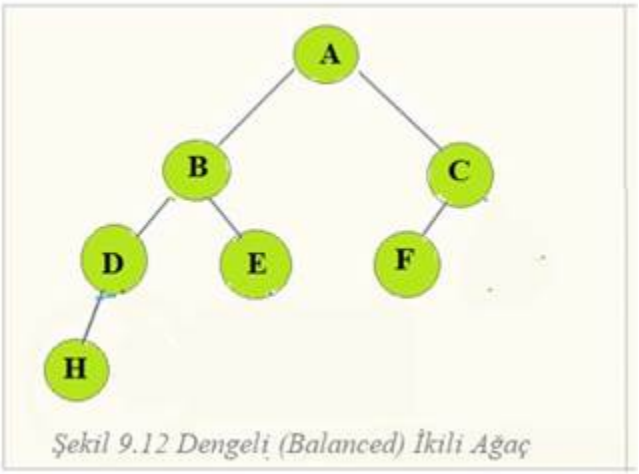
Şekil 9.11 Eksik (Incomplete) İkili Ağaç

Şekil 9.11 'de Soru işareti ile gösterilen konumlara düğüm eklenmeden sağ tarafa yeni düğümler eklenmiş ve ağacın eksiksiz ikili ağaç olma özelliği bozulmuştur.

9.3.1.5. Dengeli İkili Ağaç (Balanced Binary Tree)

Dengeli ikili ağaçlarda sol alt ağacın yüksekliği ile sağ alt ağacın yüksekliği arasında fark 1 (bir) 'dir. Bu özellik ağacın bütün düğümleri için geçerli ise bu tür ağaçlara dengeli ikili ağaç (Binary Tree) denir.

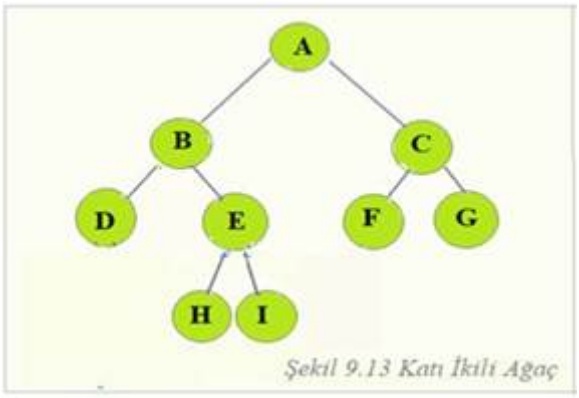
Eksiksiz ikili ağaçlar (Complete Binary Tree) aynı zamanda dengeli ikili ağaçtır (Balanced Binary Tree). Ancak her Dengeli İkili ağaç (Balanced Binary Tree) eksiksiz ikili ağaç (Complete Binary Tree) değildir.



Şekil 9.12 Dengeli (Balanced) İkili Ağaç

9.3.1.6. Katı İkili Ağaç (Strict Binary Tree)

Bir ikili ağaçta yaprak düğümler dışındaki bütün düğümler 2 veya sıfır çocuğa sahipse bu ağaçlara Katı İkili Ağaç (Strict Binary Tree) denir. Şekil 9.13 'de katı ikili ağaç gösterilmiştir.



Şekil 9.13 Katı İkili Ağaç

9.3.2. İkili Ağaç (Binary Trees) Üzerinde Dolaşma (Traverse)

Bir ağacın düğümleri arasında, ağaçtaki her düğüme sadece bir defa uğrayacak şekilde dolaşma işlemine *gezinme / geçiş* denir. İkili ağaçların düğümleri arasında çok sayıda farklı yöntemle dolaşılabilir. Biz bu başlık altında, ikili ağaçların düğümlerinin dolaşılması sırasında çok kullanılan üç yöntem üzerinde duracağız. Bu yöntemlerden ilki *önce-kök (preorder),* ikincisi *ortada-kök (inorder)* ve üçüncüsü *sonra-kök (postorder)* yöntemleridir.

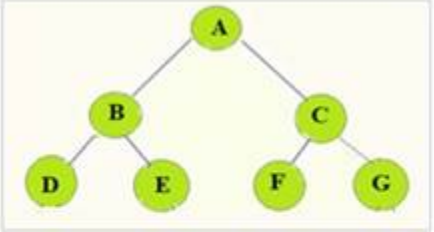
1. **Önce-kök** (Preorder) : Kök, Sol, Sağ

2. **Ortada-kök** (Inorder) : Sol, Kök, Sağ

3. **Sonra-kök** (Postorder) : Sol, Sağ, Kök

4. **Levelorder:** Önce kök, sonra soldan başlayarak ikinci düzey, üçüncü düzey …..

Önce-kök (Preorder) yönteminde önce kök, sonra sol alt ağaç, daha sonra sağ alt ağaç dolaşılır. Ortada-kök (Inorder) yönteminde ise önce sol alt ağaç, sonra kök ve daha sonra da sağ alt ağaç dolaşılır. Sonra-kök yönteminde ise önce sol alt ağaç, sonra sağ alt ağaç ve daha sonra da kök dolaşılır. Levelorder yönteminde önce kök (ilk düzey), sonra soldan başlayarak ikinci düzeydeki düğümler, daha sonra yine soldan başlayarak üçüncü düzeydeki düğümler yazdırılır.




```
Once-kok (Preorder)
A ->B ->D ->E ->C ->F ->G
Ortada-kok (Inorder)
D ->B ->E ->A ->F ->C ->G
Sonra-k÷k (Postorder)
D ->E ->B ->F ->G ->C ->A
LevelOrder
A ->B ->C ->D ->E ->F ->G
```

C programlama dilinde yazılan program aracılığı ile yukarıdaki şelilde verilen ağacın düğümleri sırasıyla Preorder, Inorder, Postorder ve Levelorder yöntemleri ile dolşılmış ve yandaki çıktılar elde edilmiştir.

```c
// Önce-kök (Preorder)
void preOrder(struct dugum* temp){
  if (temp == NULL) return;
  printf("%c ->", temp->veri);
  preOrder(temp->sol);
  preOrder(temp->sag);
}
```

*/*Preorder yönteminde önce kök sonra sol alt ağaç, daha sonra sağ alt ağaç dolaşılmıştır.*
Elde Edilen Çıktı:
A ->B ->D ->E ->C ->F ->G */

```c
// ortada kök (Inorder)
void InOrder(struct dugum* temp){
    if(temp == NULL) return;
    InOrder(temp->sol);
    printf("%c ->", temp->veri);
    InOrder(temp->sag);
}
```

Inorder yönteminde önce sol alt ağaç, sonra kök ve sonra da sağ alt ağaç dolaşılmıştır.
D ->B ->E ->A ->F ->C ->G */

```c
// Sonra-kök (Postorder)
void postOrder(struct dugum* temp) {
  if (temp == NULL) return;
  postOrder(temp->sol);
  postOrder(temp->sag);
  printf("%c ->", temp->veri);
}
```

Postorder yönteminde önce sol alt ağaç, sonra sağ alt ağaç, sonra da kök dolaşılmıştır.
D ->E ->B ->F ->G ->C ->A

```c
//levelorder
void levelOrder(struct dugum* Kok){
    front=-1;rear=-1;
    elemanEkle(Kok);
    while(front != -1 && front <= rear){
        struct dugum* temp=kuyruk[front];
        elemanSil();
        printf("%c ->",temp->veri);
        if(temp->sol!=NULL)
        elemanEkle(temp->sol);
        if(temp->sag!=NULL)
        elemanEkle(temp->sag);
    }
}
```

*/*Levelorder önce kök (ilk düzey), sonra soldan başlayarak ikinci düzeydeki düğümler, sonrada üçüncü dizeydeki düğümler dolaşılır.*/*
A ->B ->C ->D ->E ->F ->G

program 9.1 ikili ağaçta traverse işlemi (Fonksiyonlar)

9.3.3. İkili Ağaç (Binary Tree) Düğümünün Veri Yapısı Ve Düğümün Oluşturulması

```c
struct dugum
{
   int veri;
   struct dugum *sol;
   struct dugum *sag;
};
```

Yukarıda verilen veri yapısı ağacın bir düğümünün yapısıdır. Bu tanımlamaya göre ağacın her düğümünde tamsayı tipinde bir veri tutulacaktır. Bundan başka her düğümde, düğümün kendisinden sonra gelen alt ağaçlarının düğümlerini işaret etmek için kullanılacak iki işaretçi tanımlanmıştır. Aşağıda, ağacın başlangıç noktasındaki kök düğümü için işaretçi bir değişken tanımlanmıştır.

```
struct dugum *kok=NULL;
```

Bu sırada ikili ağaç oluşmuş fakat henüz *kok=NULL* ataması yapıldığı için ağaçta herhangi bir düğüm yoktur. Bunun için önce düğüm oluşturacak bir fonksiyon yazılmıştır. Fonksiyonda önce oluşturulan düğümün ***yeniDugum->veri*** fonksiyona parametre olarak gelen sayı atanmıştır ve daha sonra yeniDugum 'ün sol ve sağ işaretçilerine NULL atanmıştır (Fonksiyondaki 14 ve 15. Satırlara bakınız). Çünkü henüz işaret edilecek başka bir düğüm yoktur.

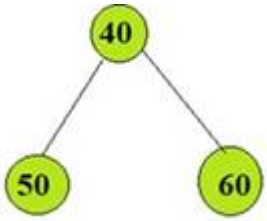
```
10  struct dugum *dOlustur(int sayi)
11  {
12      struct dugum* yeniDugum=(struct dugum*)malloc(sizeof(struct dugum));
13      yeniDugum->veri=sayi;
14      yeniDugum->sol=NULL;
15      yeniDugum->sag=NULL;
16      return yeniDugum;
17  }
```

Buraya kadar yapılan işlem, yukarıda tanımlanan veri yapısına uygun bir düğüm oluşturmak içindir. dOlustur() fonksiyonu çalıştığında bellekte yeniDugum isimli bir düğüm oluşacak ve oluşacak Bu düğümün işaretçilerinin her ikisi de NULL değerini alacaktır, düğümün veri alanına da fonksiyona parametre olarak çağıran fonksiyondan gelen değer aktarılacaktır.

Bundan sonra main fonksiyonu içerisinde düğümler oluşturulacak ve işaretçiler kullanılarak düğümler ilişkilendirilecektir.

```
20      struct  dugum *kokDugum=dOlustur(40);
21      struct dugum *dugumA=dOlustur(50);
22      struct dugum *dugumB=dOlustur(60);
23      kokDugum->sol=dugumA;          //KökDüğümün sol işaretçisi düğümA'yı
24      kokDugum->sag=dugumB;          //sağ işaretçisi dugumB'yi işaret etmektedir.
25      return 0;
26  }
```

Oluşan ağaç aşağıda verilmiştir.



Program 9.2 ikili ağaçlar için düğüm oluşturulmasını ve düğümler arasında dolaşma işleminin göstermek için yazılmıştır. Programda preorder, inorder ve postorder traverse yöntemleri kullanılmış, Levelorder yöntemi düğüme eleman ekleme işleminde kullanılmak üzere daha sonraya bırakılmıştır. Programı yazıp çalıştırınız ve ekranda elde edilen çıktıyı kullanarak ikili ağacı çiziniz?

```c
#include <stdio.h>
#include <stdlib.h>
struct dugum{
    int veri;
    struct dugum *sol;
    struct dugum *sag;
};
struct dugum *kok=NULL;
struct dugum *dOlustur(int sayi){
    struct dugum* yeniDugum=
    (struct dugum*)malloc(sizeof(struct dugum))
    yeniDugum->veri=sayi;
    yeniDugum->sol=NULL;
    yeniDugum->sag=NULL;
return yeniDugum; }
void preOrder(struct dugum* temp){
  if (temp == NULL) return;
  printf("%d ->", temp->veri);
  preOrder(temp->sol);
  preOrder(temp->sag);
}
// ortada kök (Inorder)
void InOrder(struct dugum* temp){
    if(temp == NULL) return;
    InOrder(temp->sol);
    printf("%d ->", temp->veri);
    InOrder(temp->sag);
}
// Sonra-kök (Postorder)
void postOrder(struct dugum* temp) {
  if (temp == NULL) return;
  postOrder(temp->sol);
  postOrder(temp->sag);
  printf("%d ->", temp->veri);
}
int main(){
    struct  dugum *kokDugum=dOlustur(40);
    struct dugum *dugumB=dOlustur(50);
    struct dugum *dugumC=dOlustur(60);
    struct dugum *dugumD=dOlustur(70);
    struct dugum *dugumE=dOlustur(75);
    struct dugum *dugumF=dOlustur(80);
    struct dugum *dugumG=dOlustur(85);
    kokDugum->sol=dugumB;
    kokDugum->sag=dugumC;
    dugumB->sol=dugumD;
    dugumB->sag=dugumE;
    dugumC->sol=dugumF;
    dugumC->sag=dugumG;
    printf(" \n ");
    preOrder(kokDugum);
     printf(" \n ");
    InOrder(kokDugum);
    printf(" \n ");
    postOrder(kokDugum);
    printf(" \n ");
    return 0;
}
```

Program 9.2 Düğüm oluşturma, Dolaşma

9.3.4. İkili Ağaçlara (Binary Tree) Eleman Ekleme

Aşağıda ikili ağaca (Binary Tree) eleman eklemek için C programlama dilinde yazılan kod içerisinde yer alan dugumElEkle() fonksiyonu

verilmiştir. Ayrıca fonksiyon üzerinde gerekli açıklamalar da yapılmıştır. Genel olarak verilen bu fonksiyon ikili ağaca soldan boşlayarak bulduğu boş yere eleman eklemek için kullanılacaktır.

```
void dugumeElEkle(struct dugum* ilk,int Eklenecek){
    if(kok == NULL) {
        kok = dOlustur(Eklenecek);
        return;
    }
    elemanEkle(ilk)      ;
     while(front != -1 && front <= rear){
        struct dugum* temp = kuyruk[front];
        elemanSil();
        if(temp->sol == NULL){
            temp->sol = dOlustur(Eklenecek);
            break;
        }
        else{
            elemanEkle(temp->sol);
        }
        if(temp->sag== NULL){
            temp->sag = dOlustur(Eklenecek);
            break;
        }
        else{
            elemanEkle(temp->sag);
        }
    }
}
```

/ önce kok düğümün NULL olup olmadığı kontrol ediliyor ve NULL ise fonksiyona gelen parametre değeri ile kok düğüm oluşturuluyor. */*

//elemanekle() fonksiyonu ile gelen düğüm kuyruğa // atılıyor.

/ kuruk boş değilse geçici bir düğüm oluştu oluşturu. kuyruk yapısının en önündeki eleman. alınarak kuruktan çıkar ılıyor*

Önce temp'in solu kontrol ediliyor eğer boş ise kullanıcının gönderdiği parmetre ile düğüm oluşturulur ve temp'in soluna eklenir.

Değilse kuyruğa tempin sol düğümü ekleniyor.

Aynı işlemler tempİn sağı için de tekrarlanıyor.

*Sonuç olarak boş olan yere düğüm eklenmiş oluyor */*

Program 9.3'te ikili ağaca (Binary Tree) eleman ekleyen programın tamamı verilmiştir. Programda yedinci bölümden "Kuyrukların Dizi Uygulaması" başlığı altında anlatılan kısımdan iki önemli fonksiyon alınmıştır. Bu fonksiyonlardan birincisi kuyruğa eleman ekleyen (elemanEkle()) fonksiyonu, ikincisi ise kuyruktan eleman çıkaran elemanSil() fonksiyonudur. Bu fonksiyonlar kuyrukların dizi uygulamasına ait örneklerdir.

Program 9.3'te eleman oluşturmak için dOlustur() isimli bir fonksiyon yer almıştır.

```
struct dugum *dOlustur(int sayi){
    struct dugum* yeniDugum=
    (struct dugum*)malloc(sizeof(struct dugum))
    yeniDugum->veri=sayi;
    yeniDugum->sol=NULL;
    yeniDugum->sag=NULL;
    return yeniDugum;
}
```

dOlustur fonksiyonu kendisine parametre olarak gelen sayı değeri ile her iki işaretçisi de NULL olan bir düğüm oluşturur. Oluşturulan düğüm return ile geri döndürülür.

Program 9.3'e bu fonksiyonların dışında *levelOrder()* fonksiyonu eklenmiştir. *Bkz. Program 9.1.* levelOrder() fonksiyonu ikili ağaç üzerinde önce ilk düzey, sonra ilk düzeyi takip eden ikinci, üçüncü düzeyleri dolaşmak (Traverse) işlemi için kullanılmıştır.

Program 9.3'te önce ağaca dugumElEkle() fonksiyonu ile elemanlar eklenmiş ve daha sonra da levelOrder() fonksiyonu ile düğümler dolaşılarak yazdırılmıştır.

Aşağıda program 9.3'ün tamamı verilmiştir.

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 20
struct dugum{
    int veri;
    struct dugum *sol;
    struct dugum *sag;
};
struct dugum *kok=NULL;
struct dugum* kuyruk[MAX];
int front=-1,rear=-1;
void elemanEkle(struct dugum* Dugum){
    if(rear==MAX-1){
      printf("\n Kuyruk Dolu\n");
        return;
    }
    if(front == -1)
        front = 0;
    rear++;
    kuyruk[rear] = Dugum;
}
struct dugum* elemanSil(){
    if(front == -1 || front > rear){
        printf("\n Kuyruk Bos \n");
        return;
    }
    if(front == rear){
        front = -1;
        rear = -1;
    }
    struct dugum* Cikan = kuyruk[front];
    front++;
    return Cikan;
}
```

1

```c
struct dugum *dOlustur(int sayi){
    struct dugum* yeniDugum=
    (struct dugum*)malloc(sizeof(struct dugum))
    yeniDugum->veri=sayi;
    yeniDugum->sol=NULL;
    yeniDugum->sag=NULL;
    return yeniDugum;
}
void levelOrder(struct dugum* Kok)
{

    front=-1;rear=-1;
    elemanEkle(Kok);
    while(front != -1 && front <= rear){
        struct dugum* temp=kuyruk[front];
        elemanSil();
        printf(" %d ",temp->veri);
        if(temp->sol!=NULL)
        elemanEkle(temp->sol);
        if(temp->sag!=NULL)
        elemanEkle(temp->sag);
    }
}
```

2

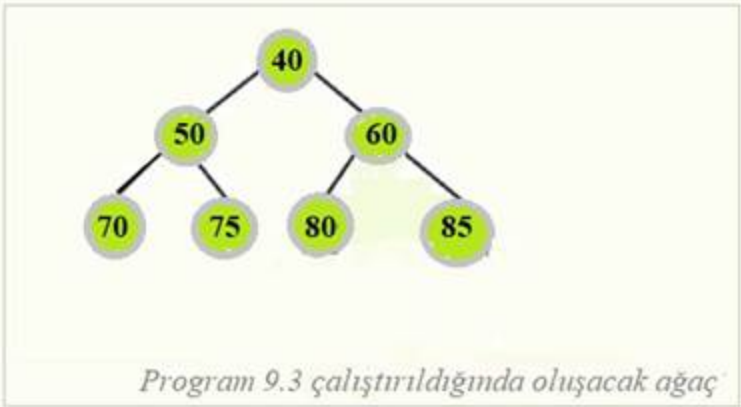
```c
void dugumeElEkle(struct dugum* ilk,int Eklenecek){
    if(kok == NULL) {
        kok = dOlustur(Eklenecek);
        return;
    }
     elemanEkle(ilk);
     while(front != -1 && front <= rear){
        struct dugum* temp = kuyruk[front];
        elemanSil();
        if(temp->sol == NULL){
            temp->sol = dOlustur(Eklenecek);
            break;
        }
        else{
            elemanEkle(temp->sol);
        }
        if(temp->sag== NULL){
            temp->sag = dOlustur(Eklenecek);
            break;
        }
        else{
            elemanEkle(temp->sag);
        }}}
```

3

```
int main(){
    dugumeElEkle(kok, 40);
    dugumeElEkle(kok, 50);
    dugumeElEkle(kok, 60);
    printf("\n");
    dugumeElEkle(kok, 70);
    dugumeElEkle(kok, 75);
    dugumeElEkle(kok, 80);
    dugumeElEkle(kok, 85);
    levelOrder(kok);
    return 0;
}
```

4

Program 9.3 İkili ağaçlara eleman ekleme



Program 9.3 çalıştırıldığında oluşacak ağaç

Program 9.3'ün ekran çıktısı incelendiğinde, levelOrder Traverse yönteminde düğümler arasında dolaşma işlemine ikili ağacın kökünden (ilk düzeyden) başlandığı, sonra ikinci düzeye geçildiği bu düzeyde de soldan başlayarak düğümlerin dolaşıldığı daha sonrada üçüncü düzeye geçilip yine soldan başlayarak dolaşma işleminin son düğüme kadar sürdürüldüğü görülmektedir.

9.4. İkili Arama Ağaçları (Binary Search Tree - Bst)

İkili arama ağaçları, daha çok yoğun olarak arama işlemi yapılan uygulamalarda kullanılan bir veri yapısıdır. İkili arama ağaçlarında;

1. Sol-alt ağacın anahtarının değeri, üst (kök) düğümünün anahtarının değerinden daha küçüktür.

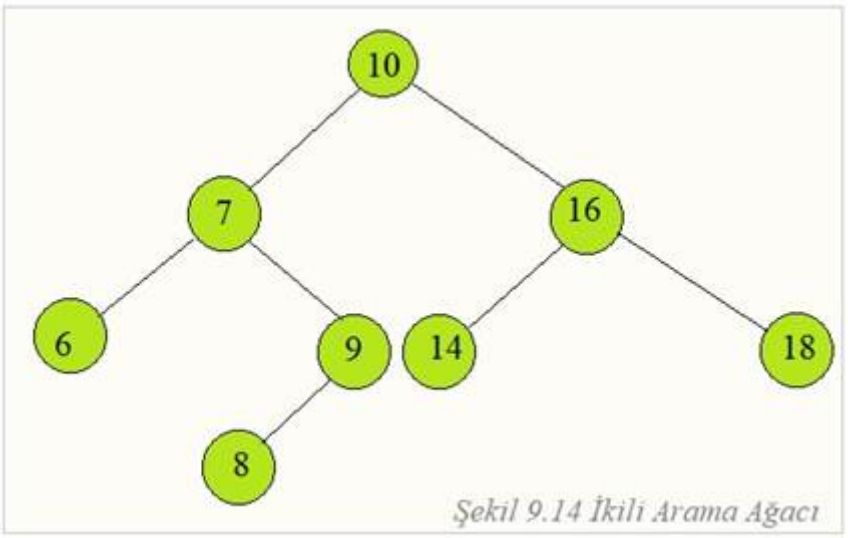
2. Sağ-alt ağacın anahtarının değeri, üst (kök) düğümünün anahtarının değerinden büyük veya ona eşittir.

3. Sol ve sağ alt ağaçların her biri ayrıca birer ikili arama ağacıdır.

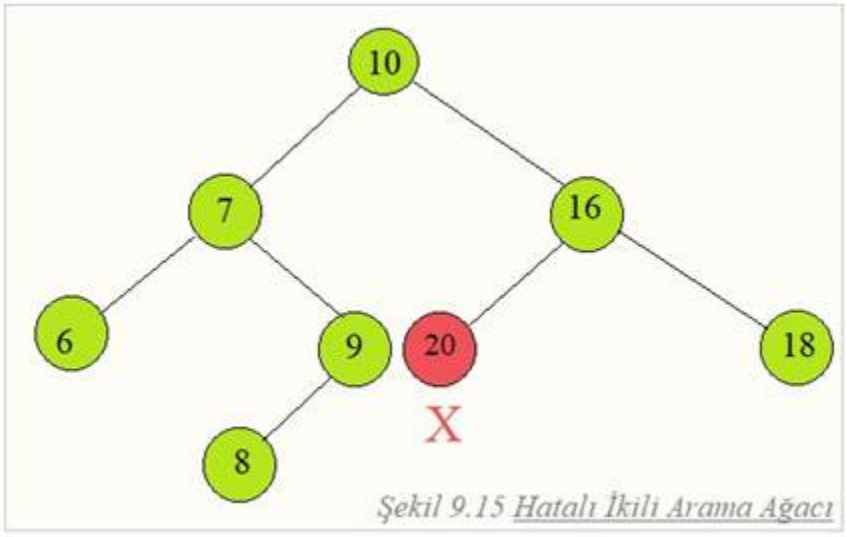
İkili arama ağaçlarının (BST) alt ağaçları sol-alt ağaç ve sağ-alt ağaç olmak üzere iki bölüme ayrılır ve aşağıdaki gibi tanımlanır.

*- ***sol-alt-ağaç(anahtar) < kök(anahtar) <= Sağ-alt-ağaç(anahtar)***

Şekil 9.14 'te doğru bir ikili arama ağacı (BST) verilmiş, Şekil 9.15 'te hatalı bir ikili arama ağacı verilmiştir.



Şekil 9.14 İkili Arama Ağacı



Şekil 9.15 Hatalı İkili Arama Ağacı

Dizi Elemanlarının İkili arama Ağacına (BST) Yerleştirilmesi

8	3	1	6	7	10	14	4

Yukarıda verilen dizi elemanlarından ikili arama ağacının oluşturması adımları;

1. Birinci adımda dizinin ilk elemanı alınır ve kök düğüm olarak belirlenir.

2. İkinci adımda dizinin 2. Elemanı alınır ve 3 (üç), 8 (sekiz) ile karşılaştırılır. (3<8) olduğu için 3 (Üç), kök düğümün soluna yerleştirilir.

3. Üçüncü adımda dizinin üçüncü elemanı olan 1 (bir) alınır. 1 (Bir), 8 (Sekiz) ile karşılaştırılır. 1<8 olduğu için ağacın sol kenarından ilerlenir. 1 (Bir), 3 (üç) ile karşılaştırılır ve 1<3 olduğu için 1 (Bir) üçün soluna yerleştirilir.

4. Dördüncü adımda, dizinin dördüncü elemanı olan 6 (altı) alınır. 6 (Altı), 8 (Sekiz) ile karşılaştırılır. 6<8 olduğu için sol kenardan ilerlenir. 6 (altı), 3 (Üç) ile karşılaştırılır. 6>3 olduğu için 6 (altı) üçün sağına yerleştirilir.

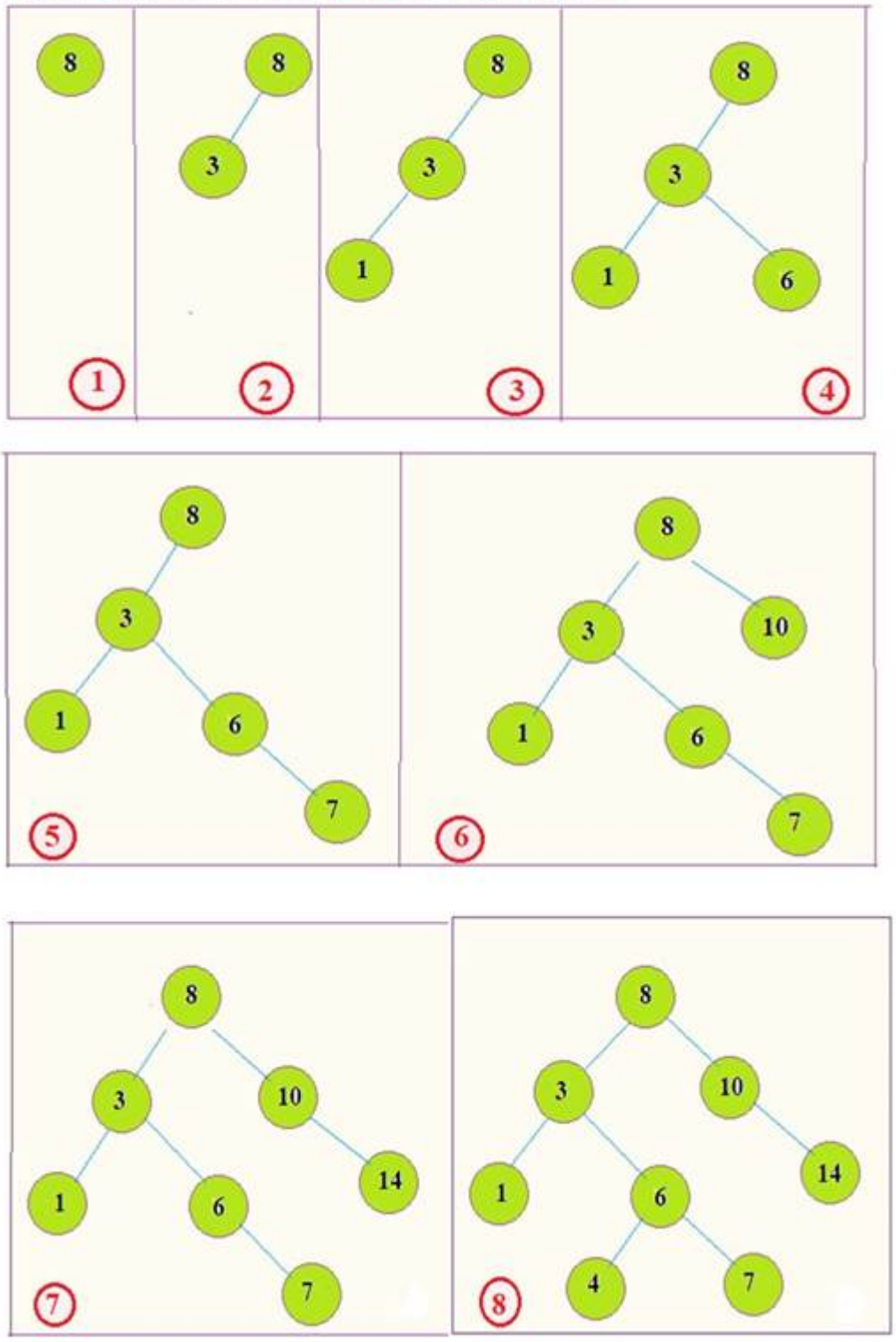
5. Beşinci adımda, dizinin 5. Elemanı olan 7 (yedi) alınır. 7 (Yedi), 8 (Sekiz) ile karşılaştırılır. 7<8 olduğu için sol kenardan ilerlenir. 7 (Yedi), 3 (Üç) ile karşılaştırılır. 7>3 olduğu için 3 (üç) ‘ün sağından ilerlenir. 7 (Yedi), 6 (Altı) ile karşılaştırılır.7>6 olduğu için 7 (Yedi), 6 (Altı) ‘nın sağına yerleştirilir.

6. Altıncı adımda dizinin 6. Elemanı olan 10 (On) alınır. 10 (On) sekizle karşılaştırılır. 10>8 olduğu için 10 (on) sekizin soluna yerleştirilir.

7. Yedinci adımda dizinin 7. Elemanı olan14 (On Dört) alınır. 14 (On Dört), sekizle karşılaştırılır. 14>8 olduğu için sekizin sağından ilerlenir. 14 (On Dört), 10 (on) ile karşılaştırılır. 14>10 olduğu için 14 (On Dört), 10 (on) ’un sağına yerleştirilir.

8. Sekizinci adımda dizinin sekizinci elemanı olan 4 (Dört) alınır. 4 (Dört), 8 (sekiz) ile karşılaştırılır. 4<8 olduğu için 8 ‘in solundan ilerlenir. 4 (Dört), 3 (üç) ile karşılaştırılır. 4>3 olduğu için 3 (ün) sağından ilerlenir. 4 (Dört), 6 (Altı) ile karşılaştırılır. 4<6 olduğu için 4 (Dört) 6 (Altı) ‘nın soluna yerleştirilir.

Dikkat ! Dizi elemanlarının ikli arama ağacına yerleştirilmesi sırasında karşılaştırma işlemlerine,her adımda kök (root) elemandan başlanır.



9.4.1. İkili Arama Ağaçlarına (Binary Search Tree - BST) Eleman Eklenmesi

İkili arama ağaçlarına eleman ekleyen fonksiyon parametre olarak kok ve eklenecek elemanı almıştır.

```
struct dugum *elemanEkle(struct dugum *dugum, int veri)
```

İkili arama ağcına (BST) eleman eklemek için gerekli işlem adımları:

1. Eğer ağaçta hiç eleman yoksa eleman oluşturmak için yazılmış olan fonksiyon gönderilen parametre ile çağırılır. Eleman oluşturulur ve return edilir.

```
if (dugum == NULL) return yeniDugum(veri);
```

2. Eklenecek olan sayı, üzerinde bulunulan düğümden küçük olduğu durumda fonksiyon, düğümün solu ve eklenecek sayı (veri) parametreleri ile özyinelemeli (Recursive) olarak çağırılır ve ağaçta boş olan yere eleman eklenir.

```
if (veri < dugum->veri)
  dugum->sol = elemanEkle(dugum->sol, veri);
```

3. Eklenecek sayı üzerinde bulunulan sayıdan büyük olması durumunda sağa gidilir. Fonksiyon dugum->sag ve veri parametreleri ile özyinelemeli olarak çağırılır.

```
dugum->sag = elemanEkle(dugum->sag, veri);
```

4. Son olarak dugum return edilir.

```
return dugum;
```

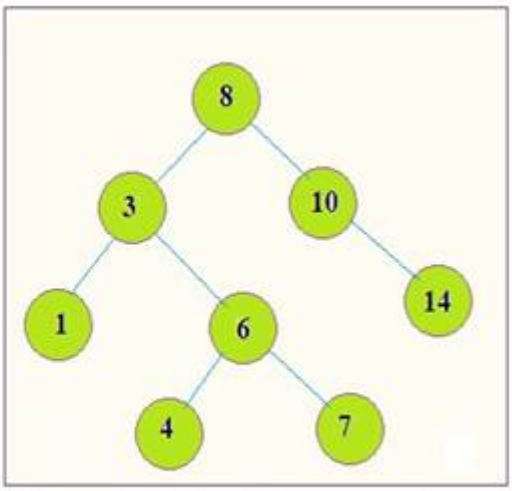
```
28  // elemanEkle() Fonksiyonu
29  struct dugum *elemanEkle(struct dugum *dugum, int veri) {
30    // Herhangi bir eleman yoksa yeniDugum 'ü döndür.
31    if (dugum == NULL) return yeniDugum(veri);
32    // Koşullara uygun Traverse ve eleman ekleme işlemi
33    if (veri < dugum->veri)
34      dugum->sol = elemanEkle(dugum->sol, veri);
35    else
36      dugum->sag = elemanEkle(dugum->sag, veri);
37
38    return dugum;
39  }
40  //Arama
```

Program 9.4 İkli Arama Ağacına eleman ekleme (Fonksiyon)

elemanEkle() fonksiyonu main() fonksiyonundan aşağıda gösterilen parametreler ile çağırılmış ve ikili arama ağacına eklenen elemanlar Inorder Traversal yöntemiyle ekrana yazdırılmıştır.

```c
int main() {
  struct dugum *kok = NULL;
  kok = elemanEkle(kok, 8);
  kok = elemanEkle(kok, 3);
  kok = elemanEkle(kok, 1);
  kok = elemanEkle(kok, 6);
  kok = elemanEkle(kok, 7);
  kok = elemanEkle(kok, 10);
  kok = elemanEkle(kok, 14);
  kok = elemanEkle(kok, 4);

 printf("Inorder traversal: ");
  inorder(kok);
}
```



Elemankar eklendikten sonra oluşan ikili arama ağacı

Ekran Çıktısı:

```
Inorder traversal: 1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 10 -> 14
```

9.4.2. İkili Arama Ağaçlarında (Binary Search Tree - BST) Arama

```c
struct dugum *dugumBul(struct dugum *kok, int bulunacakSayi){
    if(kok != NULL){
        printf(" Kontrol edilen Sayi : %d\n",kok->veri);
    }
    if(kok->veri==bulunacakSayi || kok==NULL) return kok;
    if(bulunacakSayi<kok->veri)
         dugumBul(kok->sol,bulunacakSayi);
    else
    dugumBul(kok->sag,bulunacakSayi);

}
```

Arama Fonksiyonu

Arama fonksiyonu parametre olarak başlangıç düğümü ile aranan sayı olmak üzere iki parametre alır.

İkili arama ağcına (BST) eleman aramak için gerekli işlem adımları:

1. Ağaç üzerinde arama yapılırken izlenen yolun ekranda görünmesi için başlangıç düğümü NULL olmadığı sürece *printf()* fonksiyonu ile ekrana kontrol edilen elemanlar ekrana yazdırılmıştır.

```c
if(kok != NULL){
    printf(" Kontrol edilen Sayi : %d\n",kok->veri);
```

2. Aranan sayı fonksiyona parametre olarak gönderilen düğümün içerisinde ise veya kok==NULL ise kok return edilmiştir.

```
if(kok->veri==bulunacakSayi || kok==NULL) return kok;
```

3. Aranan sayı üzerinde bulunulan düğümden küçükse arama işlemi Recursive olarak sola gidilerek yapılmıştır.

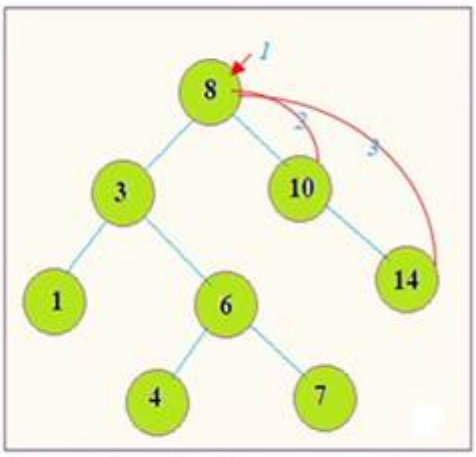
```
if(bulunacakSayi<kok->veri)
    dugumBul(kok->sol,bulunacakSayi);
```

4. Aranan sayı üzerinde bulunulan düğümden büyükse arama işlemi Recursive olarak sağa gidilerek yapılmıştır.

```
else
dugumBul(kok->sag,bulunacakSayi);
```

```
int main() {
  struct dugum *kok = NULL;
  kok = elemanEkle(kok, 8);
  kok = elemanEkle(kok, 3);
  kok = elemanEkle(kok, 1);
  kok = elemanEkle(kok, 6);
  kok = elemanEkle(kok, 7);
  kok = elemanEkle(kok, 10);
  kok = elemanEkle(kok, 14);
  kok = elemanEkle(kok, 4);
  dugumBul(kok,7);

}
```



Elemankar eklendikten sonra oluşan ikili arama ağacı

main() fonksiyonu içerisinden arama fonksiyonu 14 elemanı bulmak için çağırılmıştır. Elde edilen ekran çıktısı aşağıda verilmiştir.

```
Kontrol edilen Sayi : 8
Kontrol edilen Sayi : 10
Kontrol edilen Sayi : 14
```

Arama fonksiyonunun ekran çıktısı

Ekran çıktısından görüleceği gibi aranan sayı 8 'den büyüktür ve bu sebeple arama ağacın sağ tarafında yapılmıştır. Aranan sayı (14) ile 8 'in karşılaştırılmasından sonra 14 ile 10 karşılaştırılmış ve tekrar arama sağ tarafta yapılmıştır. Bu işlem adımından sonra aranan eleman bulunmuştur. 14 (On Dört) sayısı ikili arama ağacı üzerinde 3 (üç) adımda bulunmuştur***. İkili arama ağaçlarında en derindeki yaprakların (****verilen örnekte en derindeki yapraklar 4 ve 7 dir****) bulunması için atılması gereken adım sayısı ağacın yüksekliğine (****verilen örnekteki ağacın*

yüksekliği 4 'tür) ***eşittir.*** Konunu başında da belirtildiği gibi ikili arama ağaçları arama işlemleri için önemli performans kazançları sağlamaktadır.

İpucu ! İkli arama ağaçları üzerinde en derindeki elemanı aramak için atılması gereken adım sayısı ağacın yüksekliğine eşittir.

```
// İkli Arama Ağacı Ekle / Arama
#include <stdio.h>
#include <stdlib.h>
struct dugum {
  int veri;
  struct dugum *sol;
  struct dugum *sag;
};
// Düğüm Oluştur
struct dugum *yeniDugum(int sayi) {
  struct dugum *temp =
  (struct dugum *)malloc(sizeof(struct dugum));
  temp->veri = sayi;
  temp->sol = NULL;
  temp->sag = NULL;
  return temp;
}
// Inorder Traversal
void inorder(struct dugum *kok) {
  if (kok != NULL) {
    // Traverse sol
    inorder(kok->sol);
    // Traverse kok
    printf("%d -> ", kok->veri);
    // Traverse sag
    inorder(kok->sag);
  }}
// elemanEkle() Fonksiyonu
struct dugum *elemanEkle(struct dugum *dugum, int veri) {
  // Herhangi bir eleman yoksa yeniDugum 'ü döndür.
  if (dugum == NULL) return yeniDugum(veri);
  // Koşullara uygun Traverse ve eleman ekleme işlemi
  if (veri < dugum->veri)
    dugum->sol = elemanEkle(dugum->sol, veri);
  else
    dugum->sag = elemanEkle(dugum->sag, veri);
  return dugum;
}
```

```c
//Arama
struct dugum *dugumBul(struct dugum *kok, int bulunacakSayi){
    if(kok != NULL){
        printf(" Kontrol edilen Sayi : %d\n",kok->veri);
    }
    if(kok->veri==bulunacakSayi || kok==NULL) return kok;
    if(bulunacakSayi<kok->veri)
         dugumBul(kok->sol,bulunacakSayi);
    else
    dugumBul(kok->sag,bulunacakSayi);

}
int main() {
  struct dugum *kok = NULL;
  kok = elemanEkle(kok, 8);
  kok = elemanEkle(kok, 3);
  kok = elemanEkle(kok, 1);
  kok = elemanEkle(kok, 6);
  kok = elemanEkle(kok, 7);
  kok = elemanEkle(kok, 10);
  kok = elemanEkle(kok, 14);
  kok = elemanEkle(kok, 4);

  printf("Inorder traversal: ");
  inorder(kok);
  printf("\n");
 dugumBul(kok,14);
}
```

Program 9.5 İkili arama ağaçlarında ağaca eleman Ekleme / Arama

Bölüm Özeti

Ağaç veri yapısının temel kavramlarını ve ağaç veri yapısı türlerini açıklayabilecek,

Ağaçlar (Trees) doğrusal (lineer) veri yapıları olan diziler, bağlantılı listeler, yığınlar ve kuyruklardan farklı olarak, doğrusal olmayan hiyerarşik bir veri yapısıdır. Ağaç, verilerin birbirine sanki bir ağaç yapısı oluşturuyormuş gibi sanal olarak bağlanmasıyla elde edilir. Bu yapıda veri, düğümlerde (node) tutulur. Ağaç veri yapısında düğümler arası ilişki kenarlar /dallar (edges) kullanılarak oluşturulur. Başka bir ifade ile ağaç veri yapısında düğümler birbirine kenarlar/dallar kullanılarak bağlanır.

Bu hiyerarşik yapımın en tepesindeki düğüme **kök (root)** düğüm adı verilir. Kök düğümünden önce ağaçta herhangi bir düğüm bulunmaz. Çocuğu olan düğümlere baba (parent), Bir düğüme bağlı olan alt düğümlere çocuk (child), Çocuğu olmayan düğümlere yaprak (leaf) denir. Eğer bir ağacın düğümü yoksa ağaç boş olarak adlandırılır.

En çok bilinen ağaç veri yapısı türleri İkili Arama Ağacı (Binary Search Tree), Kodlama Ağacı (Coding Tree), Sözlük Ağacı (Dictionary Tree), Kümeleme Ağacı (Heep Tree) ve Bağıntı Ağacı (Expressin Tree) dir. Her biri farklı veri yapısında olan bu ağaç türleri üzerinde koşacak arama, ekleme ve silme gibi algoritmalar da farklıdır. Biz bu bölümde İkili Ağaçlar (Binary Tree) ve ikili ağaçların özel bir uygulaması olan İkili Arama Ağaçlarını (Binary Search Tree) konularını ele aldık.

İkili Ağaçların (Binary Tree) özelliklerini açıklayabilecek,

İkili ağaçlar (Binary Trees) her düğümünün en fazla iki çocuk alabildiği ağaç veri yapısıdır. Buradan hareketle ikili ağaçlarda her düğümün 0 veya 1 veya 2 çocuğu olabileceğini söyleyebiliriz. İkili ağaçtaki her düğüm, veri öğesi ile birlikte bir sol ve sağ referansa sahiptir. Bu ağaçlar özellikle arama işlemlerinde daha iyi sonuç alabilmek için üretilmiştir. İkili ağaçların arama performansları dizilerden daha kötü, bağlı listelerden daha başarılıdır. Buna karşılık ikili ağaçlar ekleme (Insert) ve Silme (Delete) işlemlerinde dizilerden daha iyi sonuç vermektedirler. İkili ağaçlarda eleman eklenmesi ve silinmesi için bir sınır yoktur. Bu yönüyle ikili ağaçlar bağlı listelere benzer.

İkili ağaçlarda dolaşma tekniklerini kullanabilecek,

Bir ağacın düğümleri arasında, ağaçtaki her düğüme sadece bir defa uğrayacak şekilde dolaşma işlemine *gezinme / geçiş* denir. İkili ağaçların düğümleri arasında çok sayıda farklı yöntemle dolaşılabilir. Biz bu başlık altında, ikili ağaçların düğümlerinin dolaşılması sırasında çok kullanılan üç yöntem üzerinde durduk. Bu yöntemlerden ilki *önce-kök (preorder),* ikincisi *ortada-kök (inorder)* ve üçüncüsü *sonra-kök (postorder)* yöntemleridir.

1. **Önce-kök** (Preorder) : Kök, Sol, Sağ

2. **Ortada-kök** (Inorder) : Sol, Kök, Sağ

3. **Sonra-kök** (Postorder) : Sol, Sağ, Kök

4. **Levelorder:** Önce kök, sonra soldan başlayarak ikinci düzey, üçüncü düzey dolaşarak gerçekleştirilir.

İkili Ağaçlara ve ikili arama ağaçlarına eleman ekleyebilecek,

İkili ağaçlar ve ikili arama ağaçları bölümlerinin son kısımlarında bu ağaçlara yeni eleman eklenmesini sağlayan C kodları verilmiştir. Bu kodların yazılıp çalıştırılması öğrenciler açısından konunun daha iyi anlaşılabilmesi için yaralı olacaktır.

Kaynakça

1. Robert Sedgewick, Kevin Wayne, Algoritmalar, Nobel Yayınevi, 2018

2. Muhammed Mastar, Süha Eriş, C++, KODLAB Yayın Dağıtım Yazılım ve Eğitim Hizmetleri San. ve Tic. Ltd. Şti, 2012.

3. Nejat Yumuşak, M. Fatih Adak, C, C++ ile Veri Yapıları, Seçkin Yayıncılık, 2014.

4. Rıfat Çölkesen, Veri Yapıları ve Algoritmalar, Papatya Yayıncılık, 2002.

5. G. Murat Taşbaşı, İleri C programlama, Altaş Yayıncılık ve Elektronik Tic. Ltd. Şti, 2005.