

T.C.
EGE ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

**VERİ YAPILARI
DERS NOTLARI**

(THIRD EDITION)

Y. Doç. Dr. Aybars UĞUR

Copyright 2005, 2003, 2002, 1999

**Eylül, 2005
İZMİR**

İÇİNDEKİLER

| | |
|---------------------------------------------------------------|-----------|
| 1. VERİ YAPILARI – GİRİŞ | 1 |
| 2. JAVA PROGRAMLAMA | 2 |
| 3. ÖZYİNELEME (RECURSION) | 22 |
| 4. AĞAÇLAR (TREES) | 34 |
| 5. YIĞIT (STACK)..... | 43 |
| 6. KUYRUKLAR (QUEUES)..... | 51 |
| 7. LİSTELER ve BAĞLI LİSTELER (LINKED LISTS) | 54 |
| 8. ALGORİTMALARIN KARŞILAŞTIRILMASI..... | 61 |
| 9. SIRALAMA..... | 66 |
| 10. ÇİZGELER (GRAPHS) ve UYGULAMALARI | 72 |
| EK 1 : JAVA'DA HAZIR VERİ YAPILARI ve KOLEKSİYONLAR... | 75 |
| EK 2 : C# (C SHARP) PROGRAMLAMA | 79 |

BÖLÜM 1

VERİ YAPILARI – GİRİŞ

Algoritma : Bir problemin çözümünde kullanılan komutlar dizisi. Bir problemi çözmek için geliştirilmiş kesin bir yöntemdir. Bir algoritma, bir programlama dilinde (Java, C, Pascal gibi) ifade edildiğinde **program** adını alır.

Veri : Algoritmalar tarafından işlenen en temel elemanlar (sayısal bilgiler, metinsel bilgiler, resimler, sesler ve girdi, çıktı olarak veya ara hesaplamalarda kullanılan diğer bilgiler ...). Bir algoritmanın etkin, anlaşılır ve doğru olabilmesi için, algoritmanın işleyeceği verilerin düzenlenmesi gerekir.

Veri Yapıları : Verilerin düzenlenme biçimini belirleyen yapıtaşlarıdır. Bir program değişkeni bile basit bir veri yapısı olarak kabul edilebilir. Değişik algoritmalarda verilerin diziler, yığınlar, kuyruklar, ağaçlar ve çizgeler gibi veri yapıları şeklinde düzenlenmesi gerekebilir.

Veri, Yapı ve Algoritma

Bir programda veri, yapı ve algoritma birbirinden ayrılmaz bileşenlerdir ve her biri önemlidir. Verilerin düzenlenme biçimleri önemlidir. Çünkü, yapı iyi tasarlandığında, etkin, doğru, anlaşılır ve hızlı çalışıp az kaynak kullanan algoritma geliştirmek kolaylaşır.

Veri Tiplerinden Veri Yapılarına

- **Veri Tipleri** : Tamsayı, Gerçek Sayı, Karakter ...
- **Bileşik Veri Tipleri** : Dizi, Yapı (kayıt), ...
- **Veri Yapıları** : Liste, Yığın, Kuyruk, Ağaç, Çizge, ...

Veri Yapıları (Tekrar)

Liste : Sonlu sayıda elemandan oluşan ve elemanları doğrusal sırada yerleştirilmiş veri yapısı. Herhangi bir elemanına erişimde sınırlama yoktur.

Yığın : Elemanlarına erişim sınırlaması olan, liste uyarlı veri yapısı. LIFO listesi.

Kuyruk : Elemanlarına erişim sınırlaması olan, liste uyarlı veri yapısı. FIFO listesi.

Veri Tabanı (Database) : Kısaca, işlenecek ve işlenmiş verilerden oluşan bilgi bankası olarak tanımlanabilir. Veri yapıları yani verilerin düzenlenme biçimleri veri tabanlarını (database) için de önemlidir. Örnek olarak, bir bankadaki müşterilerin bilgileri, bir üniversite öğretim elemanlarına, çalışanlarına, derslere ilişkin bilgiler, veri tabanlarında tutulup işlenir.

Veri Yapıları : Statik Veri Yapıları ve Dinamik Veri Yapıları

BÖLÜM 2

JAVA PROGRAMLAMA

- Basit bir Java programı :

```
// Ekran, "Merhaba" yazdıran Java Programı
// Ornek1.java
public class Ornek1
{
    public static void main(String args[])
    {
        System.out.println("Merhaba");
    }
}
```

Programın Yazılması ve Derlenmesi :

- Herhangi bir ASCII metin editörü ile Ornek1.java programının yazılması
- Ornek1.java programının Java compiler ile derlenerek Java yorumlayıcısının anlayacağı byte code'lara çevrilmesi yani "Ornek1.class" dosyasının oluşturulması

"javac Ornek1.java" komutu ile

Ornek1.java $\xrightarrow{\text{javac}}$ Ornek1.class

Programın Çalıştırılması :

- "Java Ornek1" komutu ile uygulama çalıştırılır.

Ekran Çıktısı :

Merhaba

BÖLÜM 2.1

JAVA PROGRAMLAMA I

Değişken Tanımlama, Aritmetik İşlemler, String'ler, I/O İşlemleri, Metotlar, Diziler (Array), Denetim Yapıları (if, for, while, ...), GUI ...

İki tamsayıyı toplayan metot :

```
class Topla
{
    public static void main(String args[])
    {
        System.out.println(topla(5,6));
    }

    public static int topla(int sayi1,int sayi2)
    {
        return sayi1+sayi2;
    }
}
```

Örnek 2.1 : Tamsayı, Döngü, Dizi, Metot ve Ekrana Yazdırma

int dizi[] = { 5,6,7,8 }; veya benzer şekilde verilen bir tamsayı dizisinin elemanlarının toplamını bulan metodu içeren java programını yazınız.

```
class DiziTopla
{
    public static void main(String args[])
    {
        int dizi[] = { 5,6,7,8 };
        System.out.println(topla(dizi));
    }

    public static int topla(int dizi[])
    {
        int toplam = 0;
        for(int i=0; i<dizi.length; ++i)
            toplam+=dizi[i];
        return toplam;
    }
}
```

Örnek 2.2 : String'ler

Verilen bir String dizisini, ters sırada (sondan başa doğru) listeleyen Java programını yazınız.

```
class DiziListele
{
    public static void main(String args[])
    {
        String strDizi[] = { "Ali", "Zekiye", "Cemil", "Kemal" };

        int son = strDizi.length-1;
        for(int i=son; i>=0; --i)
        {
            System.out.println(strDizi[i]);
        }
    }
}
```

Ekran Çıktısı :

```
Kemal
Cemil
Zekiye
Ali
```

Örnek 2.3 : if, if else

Verilen bir kişi adını bir dizide arayan ve bulunup bulunamadığını belirten Java metodunu yazınız. Aranan kişinin String aranan = "Ali" şeklinde verildiğini varsayabilirsiniz.

```
class DiziArama
{
    public static void main(String args[])
    {
        String strDizi[] ={"Ali", "Zekiye", "Cemil", "Kemal"};

        String kelime = "Cemil";
        if (ara(strDizi, kelime))
            System.out.println(kelime+" Dizide Bulundu");
        else
            System.out.println(kelime+" Dizide Bulunamadı");

        kelime = "Yılmaz";
        if (ara(strDizi, kelime))
            System.out.println(kelime+" Dizide Bulundu");
        else
            System.out.println(kelime+" Dizide Bulunamadı");
    }

    public static boolean ara(String dizi[], String aranan)
    {
        for(int i=0; i<dizi.length; ++i)
            if (aranan.equals(dizi[i])) return true;

        return false;
    }
}
```

Ekran Çıktısı :

Cemil Dizide Bulundu
Yılmaz Dizide Bulunamadı

Örnek 2.4 : Boş bir diziye arka arkaya eleman ekleyen metodu içeren Applet'i yazınız.

```
import java.applet.Applet;
import java.awt.*;
```

```
public class DiziElemanEkle extends Applet
{
    String strDizi[];
    int elemanSayac = 0;

    public void init()
    {
        strDizi = new String[10];

        elemanEkle("Ali");
        elemanEkle("Cemil");
        listele();
    }

    public void elemanEkle(String yeniEleman)
    {
        strDizi[elemanSayac]=yeniEleman;
        elemanSayac++;
    }

    public void listele()
    { for(int i=0; i<strDizi.length; ++i)
        System.out.println(strDizi[i]); }
}
```

Örnek 2.5 : 2 x 4'lük bir matris oluşturan ve elemanlarını listeleyen Java programını yazınız.

```
class MatrisListele
{
    public static void main(String args[])
    { int matris[][] = { { 5,6,7,8 }, { 9, 10, 11, 12} };
        listele(matris); }

    public static void listele(int matris[][])
    {
        for(int i=0; i<matris.length; ++i)
        {
            for(int j=0; j<matris[i].length; ++j)
                System.out.print(matris[i][j]+" ");
            System.out.println();
        }
    }
}
```


Örnek 2.6 : String'ler ve Karakter Dizileri Farkı

```
public class Ornek06
{
    public static void main(String args[])
    {
        char charArray[] = { 'M','e','r','h','a','b','a' };
        String s = new String("Merhaba");
        String s1,s2;

        s1 = new String(s);
        s2 = new String(s);

        System.out.println("s1="+s1+" "+"s2="+s2+"\n");
        if(s1.equals(s2))
            System.out.println("Her iki string esit");

        System.out.println("Uzunluklar :");
        System.out.println("Karakter dizisi"+
            charArray.length+" karakter");
        System.out.println("s1 "+s1.length()+" karakter");
        System.out.println("s2 "+s2.length()+" karakter");
    }
}
```

Ekran çıktısı :

```
s1=Merhaba s2=Merhaba

Her iki string esit
Uzunluklar :
Karakter dizisi 7 karakter
s1 7 karakter
s2 7 karakter
```

Örnek 2.7 : Bazı String İşlemleri

```
public class Ornek07
{
    public static void main(String args[])
    {
        String s=new String("abcdefghijklmnopqrstuvwxyabcde");

        // e harfinin alfabedeki konumu
        System.out.println(s.indexOf('e'));
        // e harfinin 20. karakterden sonra konumu
        System.out.println(s.indexOf('e',20));
        // 5. karakterden 10. karaktere kadar olan string
        // parçası
        System.out.println(s.substring(5,10));
        // String birleştirme
        System.out.println(s.concat("ABCDEFGH"));
        // String atama
        s = "Merhaba"; System.out.println(s);
    }
}
```

Ekran çıktısı :

```
4
30
fghij
abcdefghijklmnopqrstuvwxyabcdeABCDEFGH
Merhaba
```

BASİT ALIŞTIRMALAR

- 1) Verilen bir ismin, bir String dizisindeki kaçınca eleman olduğunu bulan programı yazınız.
- 2) Verilen bir ismin, bir String dizisinde kaç kere tekrarlandığını bulan programı yazınız.
- 3) Bir tamsayı dizisinde, belirtilen bir sayıdan küçük kaç tane sayı olduğunu bulan programı yazınız.
- 4) Sıralı bir tamsayı dizisinden, verilen bir sayıyı silen metodu yazınız.
- 5) Sıralı bir diziye, verilen bir sayıyı ekleyen metodu yazınız.
- 6) Parametre olarak gönderilen iki tane matrisi toplayarak üçüncü matrisi elde eden metodu yazınız.
- 7) Bir matrisin satırları toplamını bir diziye aktaran metodu yazınız.
- 8) "Random" sayılardan oluşturduğunuz 10 elemanlı bir dizinin çift numaralı elemanlarını bir matrisin ilk satırına, tek numaralı elemanlarını ikinci satırına yerleştiren Java metodunu yazınız.

Mesaj ve Girdi Kutuları Kullanımı**Örnek 1 :**

Kullanıcıdan iki tamsayı isteyerek bunların toplamını, çarpımını, farkını, bölümünü ve bölümünden kalanını bulup sonuçları yazdıran Java programı.

```
import javax.swing.JOptionPane;

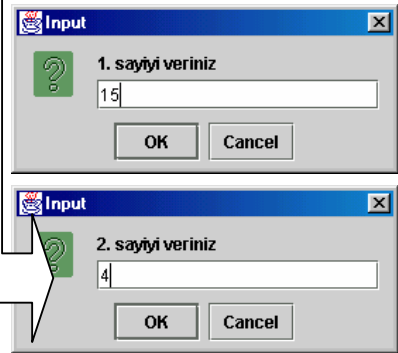
public class Ornek1
{
    public static void main(String args[])
    {
        String sayi1, sayi2;
        int tamsayi1, tamsayi2, toplam, carpim, fark, kalan;
        float bolum;

        sayi1=JOptionPane.showInputDialog("1.sayiyi veriniz");
        sayi2=JOptionPane.showInputDialog("2.sayiyi veriniz");

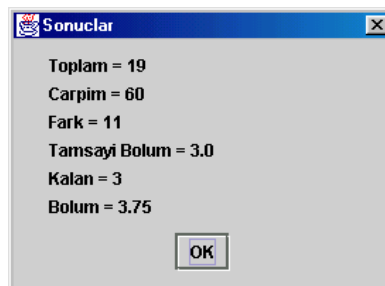
        tamsayi1 = Integer.parseInt(sayi1);
        tamsayi2 = Integer.parseInt(sayi2);

        toplam = tamsayi1+tamsayi2;
        carpim = tamsayi1*tamsayi2;
        fark = tamsayi1-tamsayi2;
        bolum = tamsayi1/tamsayi2;
        kalan = tamsayi1%tamsayi2;

        JOptionPane.showMessageDialog(null,
            "Toplam = "+toplam+"\nCarpim = "+carpim+"\nFark = "+fark+
            "\nTamsayi Bolum = "+bolum+"\nKalan = "+kalan+
            "\nBolum = "+(float)tamsayi1/tamsayi2,
            "Sonuclar", JOptionPane.PLAIN_MESSAGE);
        System.exit(0);
    }
}
```



Ekran Çıktısı : (Metin kutularına 1. sayı için 15, 2. sayı için 4 değerleri girildiğinde oluşacak sonuçlar)



Örnek 2 : Not ortalamasını bulan Java programı (-1 değeri girilene kadar notları okur).

```
import javax.swing.JOptionPane;

public class Ornek2
{
    public static void main(String args[])
    {
        float ortalama;
        int sayac=0, notu, toplam=0;

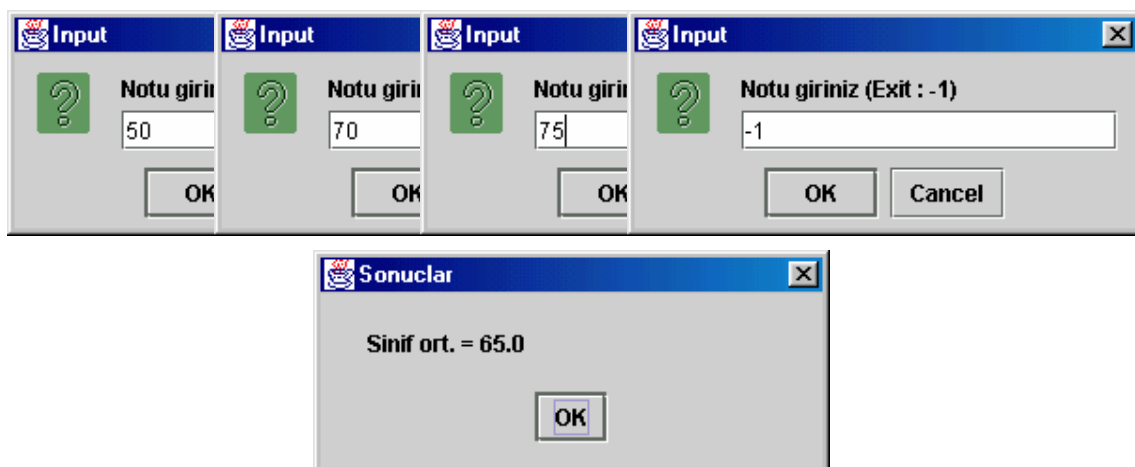
        String notStr =
            JOptionPane.showInputDialog("Notu giriniz (Exit : -1)");
        notu = Integer.parseInt(notStr);

        while(notu!=-1) {
            toplam += notu; ++sayac;
            notStr =
                JOptionPane.showInputDialog("Notu giriniz (Exit : -1)");
            notu = Integer.parseInt(notStr);
        };

        String s;
        if (sayac==0) s = "Not girilmedi!";
        else s = "Sınıf ort. = "+(float)toplam/sayac;

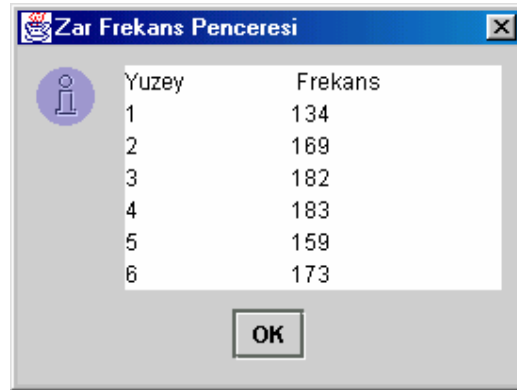
        JOptionPane.showMessageDialog(null,s,
            "Sonuclar",JOptionPane.PLAIN_MESSAGE);

        System.exit(0);
    }
}
```



Diğer Bir GUI Bileşeni (JTextArea)**Örnek 3 : "Random" sayılar (random.java)**

Altı yüzlü bir zarın 1000 kere atılması sonucu her bir yüzün kaçar kere geldiğini bularak listeleyen Java Programı.



| Yuzey | Frekans |
|-------|---------|
| 1 | 134 |
| 2 | 169 |
| 3 | 182 |
| 4 | 183 |
| 5 | 159 |
| 6 | 173 |

```
import javax.swing.*;
```

```
public class random
```

```
{  
    public static void main(String args[])  
    {  
        int[] frekans; frekans = new int[6];  
        for (int tekrar=0; tekrar<1000; ++tekrar)  
            frekans[(int)(Math.random()*6)]++;  

```

```
        JTextArea liste = new JTextArea(7,10);  
        liste.setEditable(false);  
        liste.setText("Yuzey \t Frekans");  
        for(int i=0; i<6; ++i) liste.append("\n" + (i+1) + "\t" + frekans[i]);  

```

```
        JOptionPane.showMessageDialog(null,liste,"Zar Frekans Penceresi",  
                                       JOptionPane.INFORMATION_MESSAGE);  

```

```
        System.exit(0);  

```

```
}
```

Applet Kullanımı

Örnek 4 : APPLET ve JTextArea

“kare” metodu yardımı ile, 1'den 10'a kadar olan sayıların karesini bulup ekrana yazdıran Java programı.

```
import java.awt.*;
import javax.swing.*;

public class Ornek10 extends JApplet
{
    JTextArea listelemeAlani;

    public void init()
    {
        listelemeAlani = new JTextArea();

        Container c = getContentPane();
        c.add(listelemeAlani);

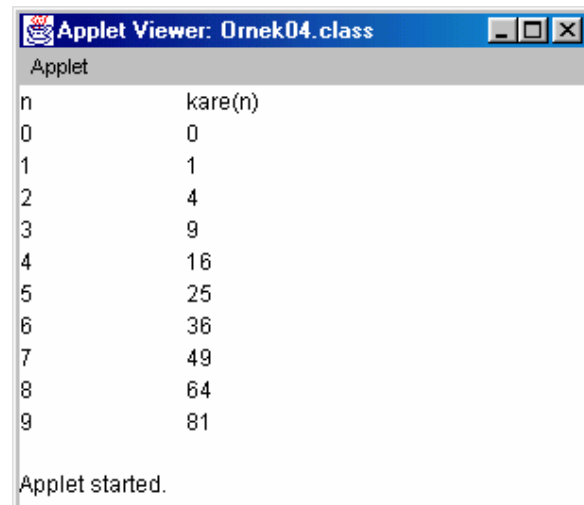
        listelemeAlani.append("\n\t" + "kare(n) \n");

        for(int i=0; i<10; ++i)
            listelemeAlani.append(i + "\t" + kare(i) + "\n");
    }

    public int kare(int sayi)
    {
        return sayi*sayi;
    }
}
```

html kodu : Ornek04.html

```
<html>
<applet code="Ornek04.class" width=300 height=200>
</applet>
</html>
```



Java programı derlendikten sonra applet,

“appletviewer Ornek04.html”

komutu verilerek görüntülenir.

Örnek 5 : Sıralama

Sayıları küçükten büyüğe doğru sıralayan Java programı (Bubble Sort)

```
import java.awt.*;
import javax.swing.*;

public class Ornek05 extends JApplet
{
    JTextArea listelemeAlani;

    public void init()
    {
        listelemeAlani = new JTextArea();

        Container c = getContentPane();
        c.add(listelemeAlani);

        int a[] = { 2,6,4,8,10,12,89,68,45,37 };

        // Sıralama işleminden önce sayıların yazdırılması
        String metin = "Before sorting :\n";
        for(int i=0; i<a.length; i++)
            metin+=" " + a[i];

        // Dizinin sıralanması
        bubbleSort(a);

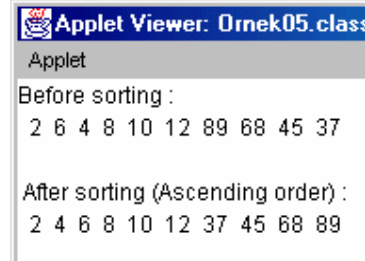
        // Sıralama işleminden sonra sayıların yazdırılması
        metin+="\n\n After sorting (Ascending order) :\n";
        for(int i=0; i<a.length; i++)
            metin+=" " + a[i];

        listelemeAlani.setText(metin);
    }

    public void bubbleSort(int b[])
    {
        for(int pass=1; pass<b.length-1; pass++)
            for(int i=0; i<b.length-1; i++)
                if(b[i]>b[i+1]) swap(b,i,i+1);
    }

    public void swap(int c[], int ilk, int ikinci)
    { int gecici = c[ilk]; c[ilk] = c[ikinci]; c[ikinci] =
      gecici; }

}
```



BÖLÜM 2.2

JAVA PROGRAMLAMA II

NESNEYE YÖNELİK PROGRAMLAMA (OBJECT ORIENTED PROGRAMMING)

JAVA'da SINIFLAR

Örnek 1 : Bir Yolcu sınıfı, yolcu nesnesi oluşturulması ve kullanılması.

```
class Yolcu {  
  
    String ad;  
    String soyad;  
    int yasi;  
  
    Yolcu() { };  
  
    Yolcu(String ad, String soyad)  
    {  
        this.ad = ad; this.soyad = soyad;  
    }  
  
    public void yazdir()  
    {  
        System.out.println("Ad      : "+ad);  
        System.out.println("Soyad   : "+soyad);  
    }  
}  
  
class Ornek_Class {  
    public static void main(String args[])  
    {  
        Yolcu yolcu1 = new Yolcu("Ali", "Yilmaz");  
        yolcu1.yazdir();  
    }  
}
```


TEMEL BİLGİ ve TERMİNOLOJİ

Sınıf (Class) : Soyut bir veri tipinin hem verilen tiplerdeki veriler kümesini, hem de bu değerler üzerinde yapılabilecek işlemler kümesini bir araya getirir.
Örnek : "Yolcu" sınıfı.

Nesne (Object) : Sınıf tipindeki değişkenlere nesne adı verilir.
Örnek : "yolcu1" nesnesi.

Metot (Method) : Bir eylemi veya işlemi gerçekleştiren sınıf üyesidir.
"Yolcu()" yapıcı metotları ve "yazdir()" metodu "Yolcu" sınıfının metotlarıdır.

Sınıf Üyeleri (Class Members) : Sınıfın elemanlarına üye adı verilir.
Değişkenler, metotlar ...
Örnekler : "ad", "soyad", "yasi" değişkenleri; "Yolcu()" yapıcı metotları ve "yazdir()" metodu "Yolcu" sınıfının üyeleridir.

Yapıcı metot (Constructor) : Sınıftan yeni bir nesne yaratıldığı anda çağrılan metoda yapıcı metot adı verilir. Yapıcı metot ismi, sınıf ismi ile aynıdır.

```
Yolcu yolcu1 = new Yolcu("Ali", "Yilmaz");
```

"yolcu1" nesnesi "new" deyiimi ile oluşturulurken "Yolcu" sınıfının iki tane "String" parametre alan yapıcı metodu devreye girer.

```
// Yapıcı metot
Yolcu(String ad, String soyad)
{
    this.ad = ad; this.soyad = soyad;
}
```

JAVA'DA VEKTÖRLER

JAVA ÖRNEK 1 : Bir vektör oluşturarak sırayla "Ali", "Cemil", "Kemal" isimlerini ekleyiniz. Vektörü iki şekilde dolaşarak (for, enum) isimleri ekrana listeleyiniz.

```
import java.util.*;

class Vektor1
{
    public static void main(String args[])
    {
        final Vector v = new Vector(1);

        v.addElement("Ali");
        v.addElement("Cemil");
        v.addElement("Kemal");

        for(int i=0; i<v.size(); ++i)
            System.out.println(v.elementAt(i));

        Enumeration enum = v.elements();
        while(enum.hasMoreElements())
            System.out.println(enum.nextElement());
    }
}
```

JAVA ÖRNEK 2 : Bir Yolcu (ad, yas) sınıfı oluşturunuz. Yolcu sınıfından yararlanarak oluşturduğunuz iki yolcuyu bir vektöre yerleştiriniz. "for" döngüsü ile tersten dolaşınız.

```
import java.util.*;

class Yolcu
{
    String ad;
    int yas;

    public Yolcu(String ad, int yas)
    {
        this.ad = ad;
        this.yas = yas;
    }
}
```

```
class Vektor2
{
    public static void main(String args[])
    {
        final Vector v = new Vector(1);

        Yolcu y1 = new Yolcu("Ali",25);
        v.addElement(y1);
        Yolcu y2 = new Yolcu("Zekiye",15);
        v.addElement(y2);

        for(int i=0; i<v.size(); ++i)
        {
            Yolcu y = (Yolcu)v.elementAt(i);
            System.out.println(y.ad+" "+y.yas);
        }

    }
}
```

ALİŞTIRMALAR

- 1) Örnek 2'yi, vektöre isme göre sıralı olarak eleman ekleyecek şekilde değiştiriniz.
- 2) Örnek 2'yi, vektöre yasa göre sıralı olarak eleman ekleyecek şekilde değiştiriniz.
- 3) Örnek 2'de, ismi verilen bir yolcuyu silen metodu ekleyiniz.
- 4) Örnek 2'de, yaşı verilen bir yolcuyu silen metodu ekleyiniz.
- 5) Örnek 2'de bir yolcunun bilgisini günleyen metodu ekleyiniz.

JAVA ÖRNEK 3 (vektor.java) : Vektörler

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class vektor extends JFrame
{
    int yer = -1;

    public vektor()
    {
        super("Vektor Ornek");
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        final Vector v = new Vector(1);

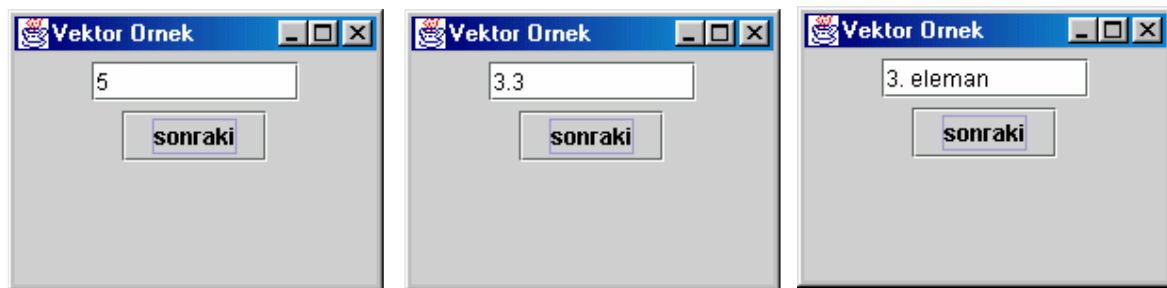
        final JTextField tf = new JTextField(10);
        c.add(tf);

        final JButton sonraki = new JButton("sonraki");
        sonraki.addActionListener
        (
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    //tf.setText((v.firstElement()).toString());
                    if (yer<(v.size()-1)) ++yer; else yer = 0;
                    tf.setText((v.elementAt(yer)).toString());
                }
            }
        );
        c.add(sonraki);

        Double d = new Double(3.3);
        Integer i = new Integer(5);
        v.addElement(i);
        v.addElement(d);
        v.addElement("3. eleman");
        setSize(200,150);
        show();
    }
}
```

```
public static void main ( String args[] )
{
    vektor app = new vektor();

    app.addWindowListener
    (
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        }
    );
}
```



Şekil 1 : Vektör programının penceresi

JAVA ÖRNEK 4 (telefon.java) : GUI Components

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class telefon extends JFrame implements
    ActionListener
{
    public Container c;
    private String names[] = { "1", "2", "3", "4", "5",
                               "6", "7", "8", "9" };

    private JButton b[];
    private GridLayout grid1;
    private JTextField tf1;

    public telefon()
    {
        super("telefon");

        grid1 = new GridLayout(3,4);

        c = getContentPane();
        c.setLayout(new BorderLayout());

        b = new JButton[names.length];

        JPanel p1 = new JPanel();
        for(int i=0; i<names.length; ++i) {
            b[i] = new JButton(names[i]);
            b[i].addActionListener(this);
            p1.add(b[i]); }
        tf1 = new JTextField();

        c.add(p1,BorderLayout.CENTER);
        c.add(tf1,BorderLayout.NORTH);
        setSize(150,150);
        show();
    }
}
```

```
public void actionPerformed(ActionEvent e)
{
    if(e.getSource()==b[0])
        tf1.setText(tf1.getText()+"1");
    else
        if(e.getSource()==b[1])
            tf1.setText(tf1.getText()+"2");
        else
            if(e.getSource()==b[2])
                tf1.setText(tf1.getText()+"3");
}

public static void main(String args[])
{
    telefon app = new telefon();
    app.addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        }
    );
}
```

BÖLÜM 3

ÖZYİNELEME (RECURSION)

3.1 Giriş

Kendini doğrudan veya dolaylı olarak çağıran fonksiyonlara özyineli (recursive) fonksiyonlar adı verilir. Özyineleme (recursion), iterasyonun (döngüler, tekrar) yerine geçebilecek çok güçlü bir programlama tekniğidir. Orijinal problemin küçük parçalarını çözmek için, bir alt programın kendi kendini çağırmasını sağlayarak, tekrarlı işlemlerin çözümüne farklı bir bakış açısı getirir. Yeni başlayan programcılara, bir fonksiyon içinde atama deyiminin sağında fonksiyon isminin kullanılmaması gerektiği söylenmekle birlikte, özyineli programlamada fonksiyon ismi doğrudan veya dolaylı olarak fonksiyon içinde kullanılır.

3.2 Örnekler

3.2.1 Faktöryel Fonksiyonu

Faktöryel fonksiyonunun matematik ve istatistik alanında önemi büyüktür. Verilen pozitif bir n tamsayısı için n faktöryel, $n!$ şeklinde gösterilir ve n ile 1 arasındaki tüm tamsayıların çarpımına eşittir.

Örnekler :

$$0! = 1$$

$$1! = 1$$

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Faktöryel fonksiyonunun tanımı (definition of factorial function) :

$$n! = 1 \quad \text{if } n == 0$$

$$n! = n * (n-1) * (n-2) * \dots * 1 \quad \text{if } n > 0$$

n tamsayısını alıp n faktöryelin değerini döndüren bir algoritmayı şu şekilde oluşturabiliriz :

```
prod = 1;
for(x=n; x>0; x--)
    prod *= x;
return prod;
```


Böyle bir algoritma tekrarlı (iterative) bir algoritmadır. Çünkü, belirli bir şart gerçekleşinceye kadar süren belirgin bir döngü veya tekrar vardır.

Faktöryel fonksiyonunun diğer bir tanımı :

$$n! = 1 \quad \text{if } n==0$$

$$n! = n * (n-1)! \quad \text{if } n>0$$

Bu, faktöryel fonksiyonunun kendi terimleri cinsinden tanımlanmasıdır. Böyle bir tanımlama, bir nesneyi kendi cinsinden daha basit olarak ifade etmesinden dolayı özyineli tanımlama olarak adlandırılır.

Örnek :

$$\begin{aligned} 3! &= 3 * 2! &= 3 * 2 * 1! &= 3 * 2 * 1 * 0! \\ & & &= 3 * 2 * 1 * 1 \\ & &= 3 * 2 * 1 \\ &= 3 * 2 \\ &= 6 \end{aligned}$$

/* n! değerini hesaplayan C fonksiyonu */

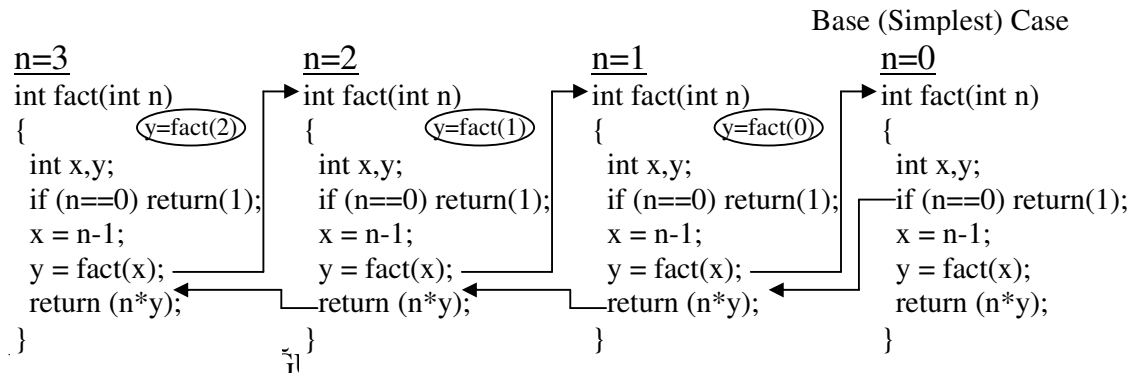
Recursive

```
int fact(int n)
{
    int x,y;
    if (n==0) return(1);
    x = n-1;
    y = fact(x); /*Kendini çağırıyor*/
    return (n*y);
}
```

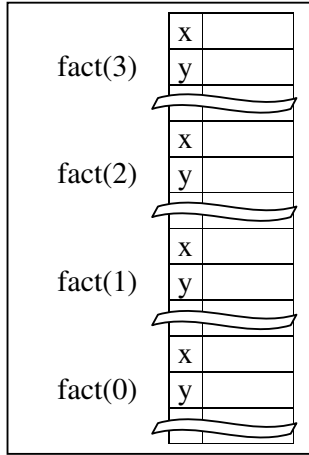
Iterative

```
int fact(int n)
{
    int x, prod;
    prod = 1;
    for(x=n; x>0; x--)
        prod *= x;
    return (prod);
}
```

Bu fonksiyonlar diğer bir fonksiyondan “printf(“%d”, fact(3));” şeklinde çağrılabilir.



Bellek Görünümü :



Fonksiyon özyineli olarak her çağrılışında yerel değişkenler ve parametreler için bellekten yer ayrılır. Her fonksiyondan çıkışta ise ilgili fonksiyonun (en son çağrılan) değişkenleri için bellekten ayrılan yerler serbest bırakılır ve bir önceki kopya yeniden etkinleştirilir. C bu işlemi yığıt (stack) kullanarak gerçekleştirir. Her fonksiyon çağrılışında fonksiyonun değişkenleri yığıtın en üstüne konulur. Fonksiyondan çıkıldığında ise en son eklenen eleman çıkarılır. Herhangi bir anda fonksiyonların bellekte bulunan kopyalarını parametre değerleri ile birlikte görmek için Debug - Call Stack seçeneği kullanılır (BorlandC 3.0).

Özyineli fonksiyonun her çağrılışında bellekte x ve y değişkenleri için yer ayrılması istenmiyorsa fonksiyon kısaltılabilir :

```
int fact(int n)
{
    return ( (n==0) ? 1 : n * fact(n-1) );
}
```

Hata durumları da kontrol edilmelidir. “fact(-1)” gibi bir fonksiyon çağrımında n azaldıkça azalacaktır ve programın sonlandırma koşulu olan “n == 0” durumu oluşmayacaktır. Bir fonksiyon kendisini sonsuza kadar çağırmamalıdır. Bunu engellemek için fonksiyon biraz değiştirilebilir :

```
int fact(int n)
{
    if(n<0)
        { printf(“%s”, “Faktöryel fonksiyonunda negatif parametre!”); exit(1); };
    return ( (n==0) ? 1 : n * fact(n-1) );
}
```

3.2.2 Fibonacci Dizisi (Fibonacci Sequence)

Fibonacci dizisi, her elemanı kendinden önceki iki elemanın toplamı şeklinde ifade edilen dizidir :

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- 0. eleman : 0
- 1. eleman : 1
- 2. eleman : $0+1 = 1$
- 3. eleman : $1+1 = 2$
- 4. eleman : $1+2 = 3$
- 5. eleman : $2+3 = 5$
-

Fibonacci dizisinin tanımı :

$\text{fib}(n) = n$ if $n==0$ or $n==1$
 $\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$ if $(n \geq 2)$

Örnek hesaplama :

$\text{fib}(4) = \text{fib}(2) + \text{fib}(3)$
 $= \text{fib}(0) + \text{fib}(1) + \text{fib}(1) + \text{fib}(2)$
 $= 0 + 1 + 1 + \text{fib}(0) + \text{fib}(1)$
 $= 2 + 0 + 1$
 $= 3$

Fibonacci dizisinin n. elemanının değerini bulan C fonksiyonu :

```
int fib(int n)
{
    int x, y;
    if(n<=1) return(n);
    x=fib(n-1);
    y=fib(n-2);
    return(x+y);
}
```

| n | x | y | return |
|---|--------|--------|--------------|
| 3 | fib(2) | | |
| 2 | fib(1) | | |
| 1 | | | return(1)->2 |
| 2 | 1 | fib(0) | |
| 1 | | | return(0)->2 |
| 2 | 1 | 0 | return(1)->3 |
| 3 | 1 | fib(1) | |
| 1 | | | return(1)->3 |
| 3 | 2 | 1 | return(3)-> |

3.2.3 İkili Arama (Binary Search)

Özyineleme sadece matematiksel fonksiyonların tanımlamasında kullanılan başarılı bir araç değildir. Arama gibi hesaplama etkinliklerinde de oldukça kullanışlıdır. Belirli bir sayıdaki eleman içerisinde belli bir elemanı bulmaya çalışma işlemine arama adı verilir. Elemanların sıralanması arama işlemini kolaylaştırır. İkili arama da sıralanmış elemanlar üzerinde gerçekleştirilir. Bir dizinin iki parçaya bölünmesi ve uygun parçada aynı işlemin sürdürülmesi ile yapılan arama işlemidir. Telefon rehberinin ortasını açıp, soyadına göre önce ise ilk yarıda, sonra ise ikinci yarıda aynı işlemi tekrarlama yolu ile arama, telefon rehberindeki baştan itibaren sona doğru sıra ile herkese bakmaktan çok daha etkindir.

```
int binsrch(int a[], int x, int low, int high)
{
    int mid;
    if(low>high)
        return(-1);
    mid=(low+high)/2;
    return(x==a[mid] ? mid : x<a[mid] ? binsrch(a,x,low,mid-1) : binsrch(a,x,mid+1,high) );
}
```

```
int i; int a[8] = { 1,3,4,5,17,18,31,33 }; // a dizisi : 1 3 4 5 17 18 31 33
i = binsrch(a,17,0,7); // 0 ve 7. elemanlar arasında 17 sayısının aratılmasını sağlar.
```

```
binsrch(a,17,0,7);
mid = (0+7)/2 = 3. eleman
a[mid] = 5 => 17>5
```

```
binsrch(a,17,4,7);
mid = (4+7)/2 = 5. eleman
a[mid] = 18 => 17<18
```

```
binsrch(a,17,4,4);
mid = (4+4)/2 = 4. eleman
a[mid] = 17 => 17==17
return(4); // Bulundu
```

a ve x global değişken olarak tanımlanırsa fonksiyon ve çağırımı şu şekilde olacaktır :

```

int binsrch(int low, int high)
{
    int mid;
    if(low>high)
        return(-1);
    mid=(low+high)/2;
    return(x==a[mid] ? mid : x<a[mid]?binsrch(low,mid-1):binsrch(mid+1,high) );
}
... i = binsrch(0,n-1);

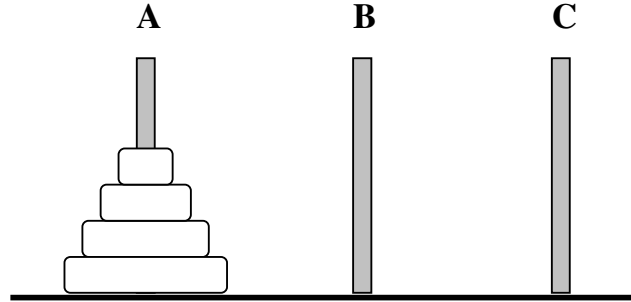
```

3.2.4 Hanoi Kuleleri Problemi (Towers of Hanoi Problem)

Üç kule (A,B,C) olan bir sistemde yarıçapı birbirinden farklı 4 tane diskin A kulesine yerleştirildiğini düşünün (Şekil 3.1).

Kurallar :

- Bir diskin altında yarıçapı daha küçük bir disk bulunamaz. Bir diskin üzerine yarıçapı daha büyük bir disk yerleştirilemez.
- Bir anda bir kulenin sadece en üstündeki disk diğer bir kuleye yerleştirilebilir. Bir anda bir disk hareket ettirilebilir.



Şekil 3.1 Hanoi Kulesinin İlk Yerleştirimi

Problemde istenen : B direğinden de yararlanarak tüm disklerin C'ye yerleştirilmesi.

Hanoi Towers Probleminin Çözümü ve C Programı Haline Getirilmesi:

Önce n disk için düşünelim.

- 1 disk olursa, doğrudan A'dan C'ye konulabilir (Doğrudan görülüyor).
- (n-1) disk cinsinden çözüm üretebilirsek özyinelemeden yararlanarak n disk için de çözümü bulabiliriz.
- 4 diskli bir sistemde, kurallara göre en üstteki 3 disk B'ye yerleştirebilirsek çözüm kolaylaşır.

Genelleştirirsek :

1. $n==1 \Rightarrow$ A'dan C'ye disk koy ve bitir.
2. En üstteki n-1 disk A'dan C'den yararlanarak B'ye aktar.
3. Kalan disk A'dan C'ye koy.
4. Kalan n-1 disk A'dan yararlanarak B'den C'ye koy.

Problem deyimi : “Hanoi Probleminin Çözümü”

Problem şu an tam olarak tanımlı değil, problem deyimi yeterli değil.

Bir diskin bir kuleden diğerine taşınmasını bilgisayarda nasıl temsil edeceğiz?

Programın Girdileri nelerdir? Çıktıları nelerdir? belli değil.

Girdi/Çıktı tasarımı herhangi bir problemin çözümünün programın algoritmasının oluşturulmasında önemli yer tutar. Oluşturulacak algoritmanın yapısı da büyük ölçüde girdi ve çıktılara bağlı olacaktır. Uygun girdi ve çıktı tasarımı, algoritmaların geliştirilmesini kolaylaştırabilir ve etkinlik sağlar.

Girdi :

disk sayısı : n
kuleler : A, B, C (uygun)

Çıktı :

disk nnn'i, yyy kulesinden alıp zzz kulesine yerleştir.
nnn : disk numarası. En küçük disk 1 numara (en küçük sayı olması doğrudan)
yyy ve zzz de kule adı.

Ana programdan towers fonksiyonunun çağırılması :

```
void main()
{
    int n;
    scanf("%d",&n);
    towers(parameters);
}
```

Şimdi parametreleri belirleme aşamasına gelindi :

```
#include <stdio.h>
void towers(int, char, char, char);

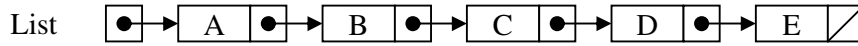
void main()
{
    int n;
    scanf("%d",&n);
    towers(n,'A','B','C');
}

void towers(int n, char frompeg, char topeg, char auxpeg)
{
    if(n==1) {
        printf("\n%d%s%c%s%c%s",n,". diski ",frompeg," kulesinden alıp ",
            topeg, " kulesine yerleştir");
        return;
    };
    towers(n-1, frompeg,auxpeg,topeg); // n-1 diskin yardımcı kuleye konulması
    printf("\n%d%s%c%s%c%s",n,". diski ",frompeg," kulesinden alıp ",
        topeg, " kulesine yerleştir");
    towers(n-1, auxpeg,topeg,frompeg); // n-1 diskin hedef kuleye konulması
}
```

Programın n=3 disk için çalıştırılması sonucu oluşan ekran çıktısı :

1. diski A kulesinden alıp C kulesine yerleştir
2. diski A kulesinden alıp B kulesine yerleştir
1. diski C kulesinden alıp B kulesine yerleştir
3. diski A kulesinden alıp C kulesine yerleştir
1. diski B kulesinden alıp A kulesine yerleştir
2. diski B kulesinden alıp C kulesine yerleştir
1. diski A kulesinden alıp C kulesine yerleştir

3.2.5 Tek Bağlı Listede Elemanların Ters Sırada Listelenmesinde Özyineleme



Procedure RevPrint (list:ListType)

```

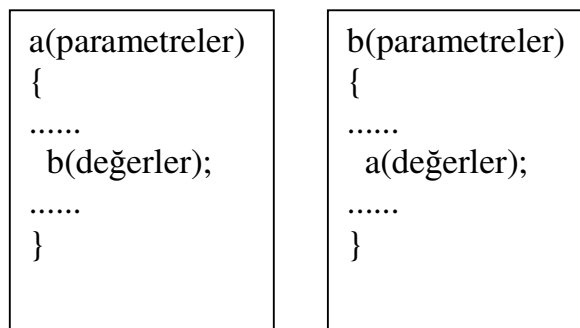
begin
  if list<>NIL
  then
    begin
      RevPrint(List^.Next);
      write(list^.info)
    end;
  end;
end;
  
```

Ekran Çıktısı : E D C B A

Elemanları bir yığta koyup ters sırada listelemek yerine doğrudan özyineleme kullanmak daha basit ve doğal.

3.3 Özyineleme Zinciri

Özyinelili bir fonksiyonun kendisini doğrudan çağırması gerekmez. Dolaylı olarak da çağırabilir. Örnek bir yapı şu şekildedir :



3.4 Özyineleme (Recursion) ve İterasyon (Iteration)

Herhangi bir fonksiyonun iteratif (iterative) yani tekrarlı versiyonu, özyineli (recursive) versiyonundan zaman (time) ve yer (space) bakımından genelde daha etkindir. Bunun nedeni, özyinelemede fonksiyonun her çağrılışında fonksiyona giriş ve çıkışta oluşan yüklerdir.

Bununla birlikte genelde yapısı uygun olan problemlerin çözümünde özyinelemenin kullanılması daha doğal ve mantıklıdır. Tanımlamalardan çözüme doğrudan ulaşılabilir. Yığıt kullanımı gerektiren durumlarda özyineli olmayan çözümlerin geliştirilmesi zordur ve hataları gidermek için daha fazla çaba harcamak gerekir. Örnek olarak, tek bağlı listedeki elemanların ters sırada yazdırılması verilebilir. Özyineli çözümde yığıt otomatik olarak oluşmaktadır ve ayrıca yığıt kullanmaya gerek kalmaz.

İterasyonda “Control Structure” olarak döngüler yolu ile tekrar kullanılırken, özyinelemede seçim yapısı kullanılır. İterasyonda tekrar, doğrudan sağlanırken, özyinelemede sürekli fonksiyon çağrıları ile sağlanır. İterasyon, döngü durum şartı geçersizliğinde sonlanırken, özyineleme en basit duruma (simplest case = base case) ulaşıldığında sonlanır. İterasyonda kullanılan sayaç değeri değiştirilerek problem çözülürken, özyinelemede orijinal problemin daha basit sürümleri oluşturularak çözüme ulaşılır.

JAVA'DA ÖZYİNELEME ÖRNEĞİ : Faktöryel

```
// Verilen bir sayının faktöryelini bulan metodu içeren örnek
import java.io.*;

class FaktoryelOrnek
{
    static int sayi;

    public static void main(String args[]) throws IOException
    {
        System.out.print("Sayi veriniz :");
        System.out.flush();
        sayi=getInt();
        int sonuc = factorial(sayi);
        System.out.println(sayi+"! =" +sonuc);
    }

    public static int factorial(int n)
    {
        if(n==0)
            return 1;
        else
            return(n*factorial(n-1));
    }
}
```

```
public static String getString() throws IOException
{
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
}

public static int getInt() throws IOException
{
    String s = getString();
    return Integer.parseInt(s);
}

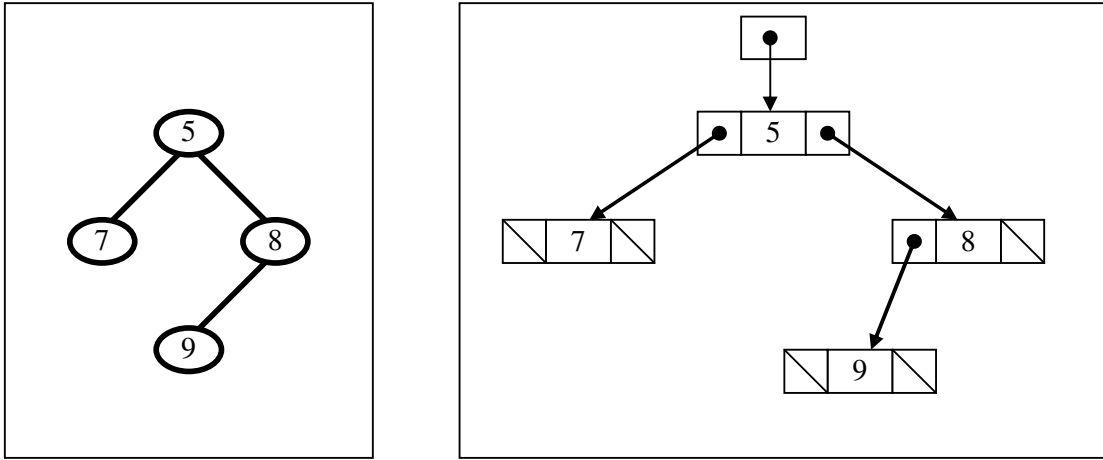
}
```

BÖLÜM 4

AĞAÇLAR (TREES)

4.1 Giriş

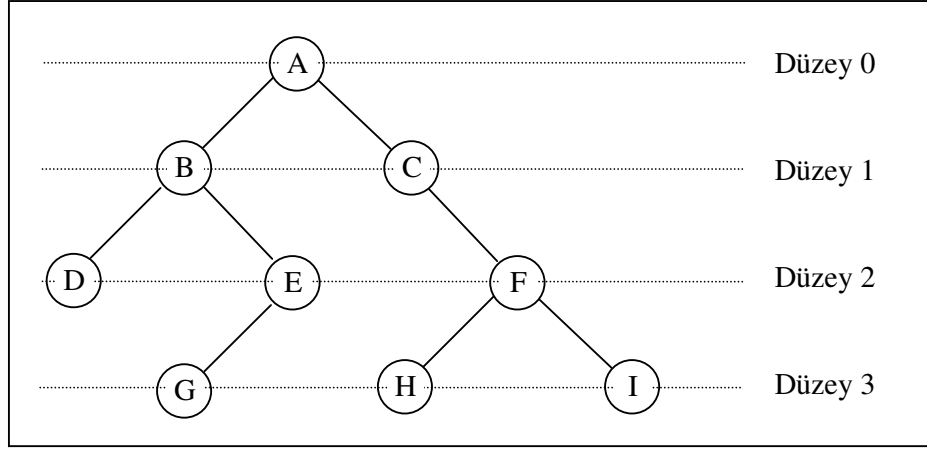
Bağlı listeler, yığıtlar ve kuyruklar doğrusal (linear) veri yapılarıdır. Ağaçlar ise doğrusal olmayan belirli niteliklere sahip iki boyutlu veri yapılarıdır (Şekil 4.1). Ağaçlardaki düğümlerden iki veya daha fazla bağ çıkabilir. İkili ağaçlar (binary trees), düğümlerinde en fazla iki bağ içeren (0,1 veya 2) ağaçlardır. Ağacın en üstteki düğüme kök (root) adı verilir.



Şekil 4.1 : Bir ikili ağacın grafiksel gösterimleri

Şekil 4.1'de görülen ağacın düğümlerindeki bilgiler sayılardan oluşmuştur. Her düğümdeki sol ve sağ bağlar yardımı ile diğer düğümlere ulaşılır. Sol (leftptr) ve sağ (rightptr) bağlar boş ("NULL" = "/" = "\") da olabilir. Düğüm yapıları değişik türlerde bilgiler içeren veya birden fazla bilgi içeren ağaçlar da olabilir. Doğadaki ağaçlar köklerinden gelişip göğe doğru yükselirken veri yapılarındaki ağaçlar kökü yukarıda yaprakları aşağıda olacak şekilde çizilirler.

Şekil 4.2'deki ağaç, A düğümü kök olmak üzere 9 düğümden oluşmaktadır. Sol alt ağaç B kökü ile başlamakta ve sağ alt ağaç da C kökü ile başlamaktadır. A'dan solda B'ye giden ve sağda C'ye giden iki dal (branch) çıkmaktadır.



Şekil 4.2 : Ağaçlarda düzeyler

4.2 Tanımlar (Örnekler Şekil 4.2'ye göre verilmiştir) :

- 1) **İkili Ağaç (Binary Tree)** : Sonlu düğümler kümesidir. Bu küme boş bir küme olabilir (empty tree). Boş değilse şu kurallara uyar.
 - i) Kök olarak adlandırılan özel bir düğüm vardır.
 - ii) Her düğüm en fazla iki düğüme bağlıdır.
 - iii) Kök hariç her düğüm bir daldan gelmektedir.
 - iv) Tüm düğümlerden yukarı doğru çıkıldıkça sonuçta köke ulaşılır.
- 2) **Düğüm (node)** : Ağacın her bir elemanına düğüm adı verilir.
Örnekler : A, B, C.
- 3) **Kök (root)** : Düzey 0'daki (şemanın en üstündeki) tek düğüm.
Örnek : Şekil 4.2'de A bilgisini içeren düğüm.
- 4) **Çocuk (child)** : Bir düğümün sol ve sağ bağı aracılığı ile bağlandığı düğümler o düğümün çocuklarıdır. Örnek : B ve C, A'nın çocuklarıdır.
- 5) **Parent** : Bir düğüm, sağ ve sol bağları ile bağlandığı düğümlerin parent'ıdır. A düğümü, B ve C düğümlerinin parent'ıdır.
- 6) **Bir düğümün düzey (level) veya derinliği (depth)** : Bir düğümün kök düğümden olan uzaklığıdır. Örnek : D düğümünün düzeyi veya derinliği 2'dir.
- 7) **Ağacın derinliği (depth of tree)** : En derindeki yaprağın derinliği veya yüksekliği (height). Örnek : Şekil 4.2'deki ağacın derinliği 3'tür.

- 8) **Yaprak (leaf)** : Sol ve sağ bağı boş olan düğümlere yaprak adı verilir. Örnekler : D,G,H,I.
- 9) **Kardeş (sibling, brother)** : Aynı parent'a sahip iki düğüme kardeş düğümler adı verilir. Örnekler : B ile C kardeştir. D ile E kardeştir. H ile I kardeştir.
- 10) **Ancestor (üst düğüm)** : Bir düğümün parent'ı birinci ancestor'ıdır. Parent'ın parent'ı (recursion) ikinci ancestor'ıdır. Kök, kendi hariç tüm düğümlerin ancestor'ıdır.
- 11) **Descendant (alt düğüm)** : Bir düğümün iki çocuğu birinci descendant'larıdır. Onların çocukları da ikinci descendant'larıdır.
- 12) **Full binary tree** : i) Her yaprağı aynı derinlikte olan ii) Yaprak olmayan düğümlerin tümünün iki çocuğu olan ağaç Full (Strictly) Binary Tree'dir (İkinci şart yeterli). Bir full binary tree'de n tane yaprak varsa bu ağaçta toplam $2n-1$ düğüm vardır.
- 13) **Complete binary tree** : Full binary tree'de yeni bir derinliğe soldan sağa doğru düğümler eklendiğinde oluşan ağaçlara Complete Binary Tree denilir. Böyle bir ağaçta bazı yapraklar diğerlerinden daha derindir. Bu nedenle full binary tree olmayabilirler. En derin düzeyde düğümler olabildiğince soldadır.
- 14) **General Tree (Ağaç)** : Her düğümün en fazla iki çocuğu olabilme sınırı olmayan ağaçlardır.
- 15) **İkili Arama Ağacı (Binary Search Tree)** : Boş olan veya her düğümü aşağıdaki şartlara uyan anahtara sahip bir ikili ağaçtır :
 - i) Kökün solundaki alt ağaçlardaki (eğer varsa) tüm anahtarlar kökteki anahtardan küçüktür.
 - ii) Kökün sağındaki alt ağaçlardaki (eğer varsa) tüm anahtarlar kökteki anahtardan büyüktür.
 - iii) Sol ve sağ alt ağaçlar da ikili arama ağaçlarıdır.

4.3 İkili Ağaçlar ve İkili Ağaçlar Üzerindeki Dolaşma İşlemleri

Dolaşma (traverse), ağaç üzerindeki tüm düğümlere uğrayarak gerçekleştirilir. Ağaçlar üzerindeki dolaşma işlemleri, ağaçtaki tüm bilgilerin listelenmesi veya başka amaçlarla yapılır. Doğrusal veri yapılarında baştan sona doğru dolaşmak kolaydır. Ağaçlar ise düğümleri doğrusal olmayan veri yapılarıdır. Bu nedenle farklı algoritmalar uygulanır. Çok bilinen yöntemler üç tane olup özyinelemeden yararlanırlar :

1) Preorder (depth-first order) Dolaşma (Traversal)

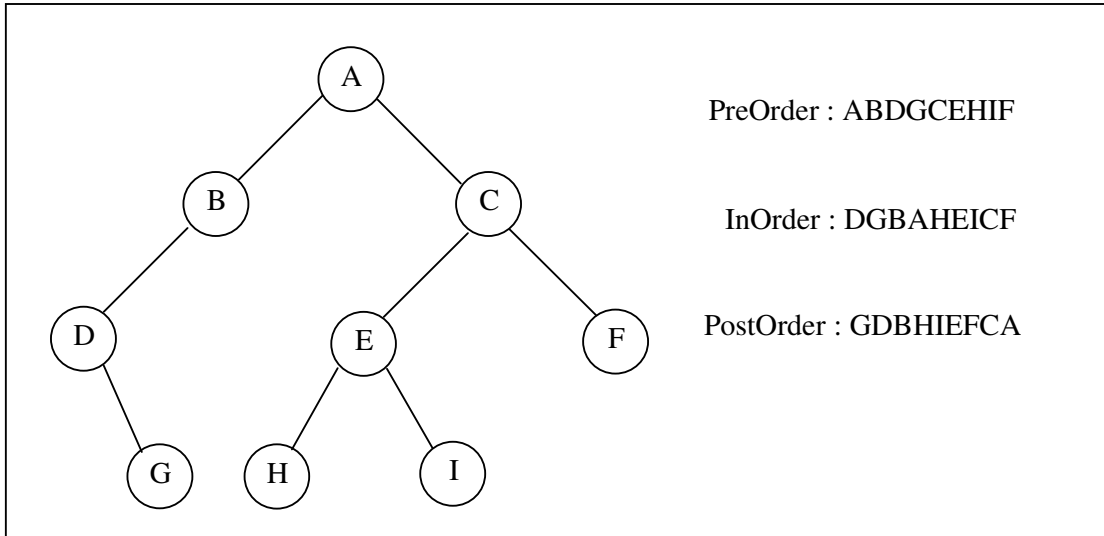
- i) Köke uğra (visit)
- ii) Sol alt ağacı preorder olarak dolaş.
- iii) Sağ alt ağacı preorder olarak dolaş.

2) Inorder (Symmetric order) Dolaşma

- i) Sol alt ağacı inorder'a göre dolaş
- ii) Köke uğra (visit)
- iii) Sağ alt ağacı inorder'a göre dolaş.

3) Postorder Dolaşma

- i) Sol alt ağacı postorder'a göre dolaş
- ii) Sağ alt ağacı postorder'a göre dolaş.
- iii) Köke uğra (visit)



Şekil 4.3 : İkili Ağaç ve değişik şekillerde dolaşılması

4.4 İkili Arama Ağaçları

İkili arama ağaçları, her bir düğümün solundaki (sol alt ağacındaki) tüm düğümler kendisinden küçük, sağındakiler (sağ alt ağacındakiler) de kendisinden büyük olacak şekilde oluşturulurlar (Şekil 4.4). İkili arama ağaçlarındaki en önemli işlemlerden birisi aramadır.

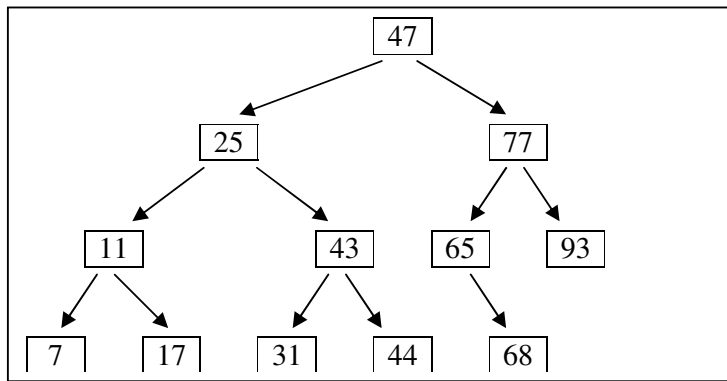
Örnek olarak aşağıdaki ağaçta, 44 sayısını aratmak için şu işlem sırası izlenir :

Karşılaştırma 1 : 44, 47 ile karşılaştırılır.
 $44 < 47$ olduğundan sol bağdan ilerlenir.
 Karşılaştırma 2 : 44, 25 ile karşılaştırılır.
 $44 > 25$ olduğundan sağ bağdan ilerlenir.
 Karşılaştırma 3 : 44, 43 ile karşılaştırılır.
 $44 > 43$ olduğundan sağ bağdan ilerlenir.
 Karşılaştırma 4 : $44 == 44$.
 Aranan anahtar değeri ağaçta bulundu!

Örnek olarak aşağıdaki ağaçta, 90 sayısını aratmak için şu işlem sırası izlenir :

Karşılaştırma 1 : 90, 47 ile karşılaştırılır.
 $90 > 47$ olduğundan sağ bağdan ilerlenir.
 Karşılaştırma 2 : 90, 77 ile karşılaştırılır.
 $90 > 77$ olduğundan sağ bağdan ilerlenir.
 Karşılaştırma 3 : 90, 93 ile karşılaştırılır.
 $90 < 93$ olduğundan sol bağdan ilerlenir.
 : NULL.

Aranan anahtar değeri ağaçta bulunamadı.



Şekil 4.4 : İkili Arama Ağacı

Görüldüğü gibi arama işleminin etkinliği ağacın yüksekliğine bağlıdır. İkili arama ağaçları dengeli tutulabilirse, bir anahtar değerini aramada oldukça hızlıdır. Böyle olduğunda n elemanlı bir ağaç en fazla $\log_2 n$ düzeyden oluşur. Bir değer bulunması veya ağaçta olmadığı belirlenmesi için en fazla $\log_2 n$ karşılaştırma yapılır. Örnek olarak 1000 elemanlı bir ikili arama ağacında bir elemanın bulunabilmesi için en fazla 10 karşılaştırma yapmak gerekecektir ($2^{10}=1024 > 1000$). Bağlı listelerde ise bulunacak elemanın değerine göre (eleman sonda ise) 1000 karşılaştırma yapmak gerekebilir.

Düğüm sayısı n olan Complete Binary Tree'de derinliği hesaplayabiliriz.

$$n = 2^0 + 2^1 + 2^2 + \dots + 2^d = 2^{d+1} - 1 \Rightarrow n+1 = 2^{d+1} \Rightarrow d = \log_2(n+1) - 1 \text{ dir.}$$

15 düğümlü bir ağaçta $d = \log_2(15+1) - 1 = 4-1 = 3$ 'tür.

İkili ağaçlardaki dolaşma işlemlerinin tümü ikili arama ağaçlarında da kullanılır. İkili arama ağaçları üzerinde inorder dolaşıldığında tüm elemanlar küçükten büyüğe sıralı bir şekilde karşımıza gelecektir.

İkili arama ağaçlarının oluşturulması ise şu şekildedir : Herhangi sıralı olmayan bir sayı dizisi gelirken her bir eleman ağaca bir yaprak düğüm olarak eklenir. Örnek olarak sıra ile, 47, 25, 43, 77, 65, 68, 93, 11, 17, 44, 31, 7 sayıları ağaca eklenmek istenirse:

47 : 47 köke eklenir.

25 : $25 < 47$, 47'nin soluna eklenir.

43 : $43 < 47$, $43 > 25$, 25'in sağına eklenir.

77 : $77 > 47$, 47'nin sağına eklenir.

65 : $65 > 47$, $65 < 77$, 77'nin sol bağına eklenir. (Ağacı tamamlayınız...).

.....

.....

İkili arama ağaçları, tekrarları önlemek açısından da oldukça yararlıdır. Ağaç oluşturulurken tekrar eden bir eleman eklenmek istendiğinde, kökten itibaren sola git, sağa git kararları ile o değer olduğu yere hızlıca ulaşılacak ve ağaçta (veriler içinde) o değer zaten olduğu anlaşılabilecektir. Bu şekilde tekrar eden verilerin olması engellenir.

Bunlar dışında ikili arama ağacından düğüm silme, ikili arama ağacını iki boyutlu ağaç biçiminde ekrana çizdirme, ağacı düzey sırasında dolaşma (düzey düzey soldan sağa) gibi algoritmalar da yazılabilir. Ayrıca ağaç düzeyini bulma, ağaca string de dahil birçok bilgi ekleme gibi işlemler de yapılabilir.

4.5 İkili Arama Ağacı Oluşturmayı ve Dolaşmayı Sağlayan Java Programı

```
// Düğüm Sınıfı
class TreeNode
{
    public int data;
    public TreeNode leftChild;
    public TreeNode rightChild;

    public void displayNode()
    { System.out.print(" "+data+" "); }
}

// Ağaç Sınıfı
class Tree
{
    private TreeNode root;

    public Tree()
    { root = null; }

    public TreeNode getRoot()
    { return root; }

    // Ağacın preOrder Dolaşılması
    public void preOrder(TreeNode localRoot)
    {
        if(localRoot!=null)
        {
            localRoot.displayNode();
            preOrder(localRoot.leftChild);
            preOrder(localRoot.rightChild);
        }
    }

    // Ağacın inOrder Dolaşılması
    public void inOrder(TreeNode localRoot)
    {
        if(localRoot!=null)
        {
            inOrder(localRoot.leftChild);
            localRoot.displayNode();
            inOrder(localRoot.rightChild);
        }
    }
}
```

```
// Ağacın postOrder Dolaşılması
public void postOrder(TreeNode localRoot)
{
    if(localRoot!=null)
    {
        postOrder(localRoot.leftChild);
        postOrder(localRoot.rightChild);
        localRoot.displayNode();
    }
}

// Ağaca bir düğüm eklemeyi sağlayan metot
public void insert(int newdata)
{
    TreeNode newNode = new TreeNode();
    newNode.data = newdata;
    if(root==null)
        root = newNode;
    else
    {
        TreeNode current = root;
        TreeNode parent;
        while(true)
        {
            parent = current;
            if(newdata<current.data)
            {
                current = current.leftChild;
                if(current==null)
                {
                    parent.leftChild=newNode;
                    return;
                }
            } else
            {
                current = current.rightChild;
                if(current==null)
                {
                    parent.rightChild=newNode;
                    return;
                }
            }
        } // end while
    } // end else not root
} // end insert()

} // class Tree
```

```
// BinTree Test sınıfı
class BinTree
{
    public static void main(String args[])
    {
        Tree theTree = new Tree();

        // Ağaca 10 tane sayı yerleştirilmesi
        System.out.println("Sayılar : ");
        for (int i=0;i<10;++i) {
            int sayi = (int) (Math.random()*100);
            System.out.print(sayi+" ");
            theTree.insert(sayi);
        };

        System.out.print("\nAğacın InOrder Dolaşılması   : ");
        theTree.inOrder(theTree.getRoot());
        System.out.print("\nAğacın PreOrder Dolaşılması   : ");
        theTree.preOrder(theTree.getRoot());
        System.out.print("\nAğacın PostOrder Dolaşılması : ");
        theTree.postOrder(theTree.getRoot());
    }
}
```

BÖLÜM 5 YIĞIT (STACK)

5.1 Giriş

Eleman ekleme çıkarmaların en üstten (top) yapıldığı veri yapısına yığıt (stack) adı verilir. Bir eleman ekleneceğinde yığıtın en üstüne konulur. Bir eleman çıkarılacağı zaman yığıtın en üstündeki eleman çıkarılır. Bu eleman da yığıtta elemanlar içindeki en son eklenen elemandır. Bu nedenle yığıtlara LIFO (Last In First Out : Son giren ilk çıkar) listesi de denilir.

5.2 Yığıt İşlemleri ve Tanımları

(Tanım) **Boş yığıt (empty stack)** : Elemanı olmayan yığıt.

push (yığita eleman ekleme) : “push(s,i)”, s yığıtının en üstüne i değerini eleman olarak ekler.

pop (yığıttan eleman çıkarma) : “i = pop(s)”, s yığıtının en üstündeki elemanı çıkartır ve değerini i değişkenine atar.

empty (yığıtın boş olup olmadığını belirleyen işlem) : empty(s), yığıt boş ise TRUE değerini, değilse FALSE değerini döndürür.

stacktop (yığıttan çıkarılmaksızın en üstteki elemanın değerini döndüren işlem)

Denk işlem : (peek)

i = pop(s);

push(s,i);

(Tanım) **Underflow** : Boş yığıt üzerinden eleman çıkarılmaya veya yığıtın en üstündeki elemanın değeri belirlenmeye çalışıldığında oluşan geçersiz durum.

(Çözümü : pop(s) veya stacktop(s) yapılmadan, empty(s) kontrolü yapılarak, boş bir yığıt olup olmadığı belirlenir. Boş ise bu işlemlerin yapılması engellenir.)

5.3 Yığıt Kullanımı (Örnek)

Bir yığıt ve üzerindeki işlemlerin tanımlandığını düşünelim. İç içe parantezler içeren bir ifadede parantezlerin geçerli olması için :

1. Açılan ve kapanan toplam parantez sayısı eşit olmalıdır. Aç “(“ ve kapa “)” parantezlerin eşitliğine bakılır.

2. Kapanan her parantezden önce bir parantez açılmış olmalıdır. Her “)” için bir “(“ olup olmadığına bakılır.

“((A+B)” ve “A+B(“ 1. şarta uymaz.

“(A+B(-C” ve “(A+B))-(C+D” 2. şarta uymaz.

Problemin çözümü için her açılan parantezde bir geçerlilik alanı açılır ve kapanan parantezde de kapanır. İfadenin herhangi bir noktasındaki Nesting Depth (parantez derinliği) o ana kadar açılmış fakat kapanmamış parantezlerin sayısıdır. İfadenin herhangi bir noktasındaki parantez sayısı = “(“ sayısı – “)” sayısı olarak belirtilebilir. Parantezleri geçerli bir ifadede şu şartlar olmalıdır.

1. İfadenin sonunda parantez sayısı 0 olmalıdır. İfadede ya hiç parantez yoktur veya açılan parantezlerin sayısı ile kapanan parantezlerin sayısı eşittir.
2. İfadenin hiçbir noktasında parantez sayısı negatif olmamalıdır. Bu, parantez açılmadan bir parantezin kapanmadığını garantiler.

Aşağıdaki ifade iki şarta da uyar :

7-((x*(x+y)/(j-3))+y)/(4-2.5))
00122234444334444322211222 2 10

Aşağıdaki ifade 1. şarta uymaz :

((A+B)
122221 (İfade sonunda 1 parantez arttı. 0 değil)

Aşağıdaki ifade 2. şarta uymaz :

(A + B)) - (C + D
1 1 1 1 0-1-1 0 0 0 0 (Arada negatif oldu)

Problem biraz değiştirildiğinde :

parantezler (parentheses) : “(“, “)”

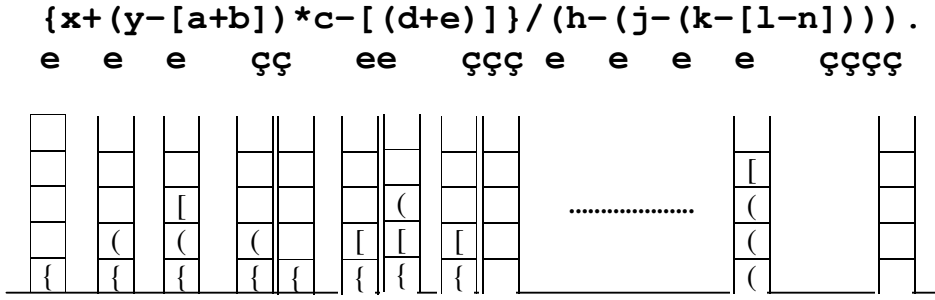
köşeli parantezler (brackets) : “[“, “]”

küme parantezleri (braces) : “{“, “}”

içerebilen ifadelerin parantez doğruluğu kontrolünün yapılması istendiğinde parantez sayılarının yanında tiplerinin de tutulması gerekecektir. Bu nedenle yukarıdaki yöntemin kullanılması uygun olmaz.

void main() { printf("Merhaba"); };
yanlış bir ifadedir.

Karşılaşılan parantezleri tutmak üzere yığıt kullanılabilir (Şekil 5.1). Bir parantezle karşılaşıldığında yığıtı eklenir. İlgili parantezlerin karşılığı ile karşılaşıldığında ise yığıtı bakılır. Yığıt boş değilse yığıttan bir eleman çıkarılarak doğru karşılık olup olmadığı kontrol edilir. Doğruysa işlem sürdürülür. Değilse ifade geçersizdir. Yığıt sonuna ulaşıldığında yığıt boş olmalıdır. Aksi halde açılmış ama kapanmamış parantez olabilir.



Şekil 5.1 : İfadelerin parantez geçerliliğinin belirlenmesinde kullanılan yığıt

5.4 Yığıt Soyut Veri Tipi (Stack ADT) (Bu gösterim şekli sınava dahil değil= optional)

Yığıt ADT'nin gösterimi (eltype yığıtın elemanlarının veri tipi olmak üzere) :

```
abstract typedef <<eltype>> STACK (eltype);
```

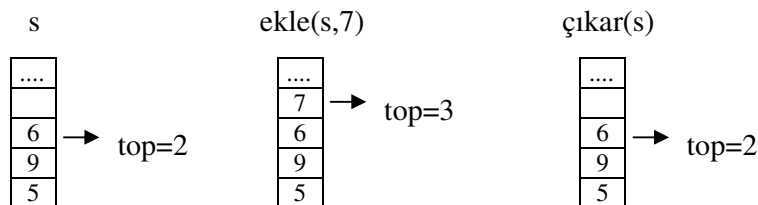
```
abstract empty(s)
STACK(eltype) s;
postcondition empty==(len(s)==0);
```

```
abstract eltype pop(s)
STACK(eltype) s;
precondition empty(s)==FALSE;
postcondition pop=first(s');
s == sub(s', 1, len(s')-1);
```

```
abstract push(s,elt)
STACK(eltype) s;
eltype elt;
postcondition s == <elt>+s';
```

5.5 Java'da Yığıtlar

Yığıtın dizi kullanılarak gerçekleştirimi (Şekil 5.2):



Şekil 5.2 : s yığıtı üzerinde eleman ekleme ve çıkarma işlemleri.

top = 2'nin anlamı yığıtta 3 eleman vardır ve yığıtın en üstündeki eleman [2]'dir. Boş yığıtta top = -1'dir.

Java'da karakter yığıtı sınıfı oluşturma ve kullanımına ilişkin bir örnek şu şekildedir :

```
import java.io.*;

class StackChar
{
    private int maxSize;
    private char[] stackArray;
    private int top;

    public StackChar(int max)
    {
        maxSize = max;
        stackArray = new char[maxSize];
        top = -1;
    }

    public void push(char j)
    { stackArray[++top] = j; }

    public char pop()
    { return stackArray[top--]; }

    public boolean isEmpty()
    { return top== -1; }
}

class Reverse
{
    public static void main(String args[])
    {
        StackChar y = new StackChar(100);
        String str = "Merhaba";
        for(int i=0; i<str.length(); ++i)
            y.push(str.charAt(i));
        while(!y.isEmpty()) System.out.println(y.pop());
    }
}
```


Java'da hazır Stack (yığıt) sınıfı da bulunmaktadır. Aşağıdaki örnekte String'ler, oluşturulan s yığıtına yerleştirilerek ters sırada listelenmektedir.

```
import java.util.*;

public class StackTest
{
    public static void main(String args[])
    {
        String str[] = { "Bilgisayar", "Dolap", "Masa",
                        "Sandalye", "Sıra" };

        Stack s = new Stack();

        for(int i=0; i<str.length;++i)
            s.push(str[i]);

        while(!s.empty()) System.out.println(s.pop());
    }
}
```

5.6 INFIX, POSTFIX, PREFIX

Bu kısımda bilgisayar alanındaki önemli konulardan biri olan infix, postfix ve prefix kavramları üzerinde durulacak ve bu kavramlarda yığıt kullanımı gösterilecektir.

$A+B$

operator (işlemci) : +

operands (işlenenler) : A, B

infix gösterim : $A+B$

prefix gösterim : $+AB$ (benzeri bir gösterim $\text{add}(A,B)$ fonksiyonu)

postfix gösterim : $AB+$

in, pre ve post, operator'ün operand'lara göre yerine karşılık gelir. Infix gösterimde işlemci (+), işlenenlerin (A,B) arasında yer alır. Prefix gösterimde işaretçi, işlenenlerden önce gelir, postfix gösterimde de sonra gelir.

$A+B*C$ infix ifadesini postfix'e çevirelim.

Bunun için işlem önceliğine bakmak gerekir. Çarpmanın toplamaya önceliği olduğu için, $A+(B*C)$ şeklinde düşünülebilir. Önce çarpma kısmı postfix'e çevrilecek sonra da sonucu.

$A+(B*C)$ anlaşılabilirliği artırmak için parantez kullandık.

$A+(BC*)$ çarpım çevrildi.

$A(BC*)+$ toplam çevrildi.

$ABC*+postfix$ form.

İşlem önceliği (büyükten küçüğe)

Üs alma

Çarpma/Bölme

Toplama/Çıkarma

Parantezsiz ve aynı önceliğe sahip işlemcilerde işlemler soldan sağa doğru yapılır (üs alma hariç). Üs almada sağdan sola doğrudur. $A-B+C$ 'de öncelik $(A-B)+C$ şeklindedir. A^B^C 'de ise $A^(B^C)$ şeklindedir. Parantezler default öncelikleri belirtmek için konulmuştur.

| Infix | Postfix | Prefix |
|-------------------------|-------------------|------------------------|
| $A+B-C$ | $AB+C-$ | $-+ABC$ |
| $(A+B)*(C-D)$ | $AB+CD-*$ | $*+AB-CD$ |
| $A^B*C-D+E/F/(G+H)$ | $AB^C*D-EF/GH+/+$ | $+-*\hat{A}BCD//EF+GH$ |
| $((A+B)*C-(D-E))^(F+G)$ | $AB+C*DE-FG+^$ | $^-*+ABC-DE+FG$ |
| $A-B/(C*D^E)$ | $ABCDE^*/-$ | $-A/B*C^DE$ |

Dikkat edilecek olunursa, postfix ile prefix ifadeler birbirinin ayna görüntüsü değildir.

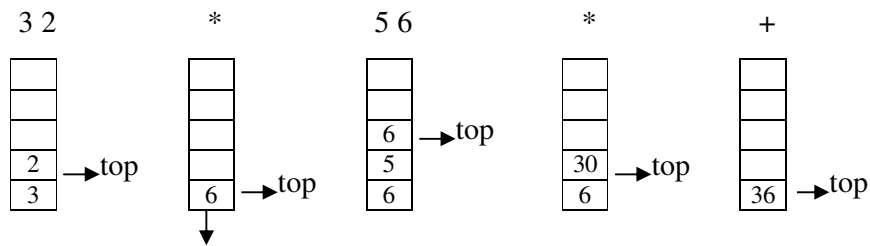
Postfix formda parantez kullanımına gerek yoktur. İki infix ifadeyi düşünün : $A+(B*C)$ ve $(A+B)*C$. İlk ifadede parantez gereksizdir. İkincide ilk ifade ile karıştırılmaması için gereklidir. Postfix forma çevirmek bu karışıklığı önler. $(A+B)*(C+D)$ 'yi infix formda parantezsiz ifade etmek için çarpım işlemi yapılırsa işlem sayısı çoğalır.

Infix formdan postfix forma çevrilen bir ifadede operand'ların (sayı veya sembol) bağlı olduğu operator'leri (+,-,*,/) görmek zorlaşır (3 4 5 * + ifadesinin sonucunun 23'e, 3 4 + 5 * ifadesinin sonucunun 35'e karşılık geldiğini bulmak zor gibi görünür). Fakat parantez kullanmadan tek anlama gelen bir hale dönüşür. İşlemleri, hesaplamaları yapmak kolaylaşır.

postfix ifadenin sonucunun hesaplanması (ve bunu gerçekleştiren algoritma) :

Bir postfix string'inde her operator kendinden önce gelen iki operand üzerinde işlem yapacaktır. Bu operandlar daha önceki operator'lerin sonuçları da olabilir. Bir anda ifadeden bir operand okunarak yığta yerleştirilir. Bir operator'e ulaşıldığında, yığtın en üstündeki iki eleman bu operator'ün operand'ları olacaktır. İki eleman yığıttan çıkarılır işlem yapılır ve sonuç tekrar yığta konulur. Artık bir sonraki operator'ün operand'ı olmaya hazırdır.

Birçok derleyici $3*2+5*6$ gibi bir infix ifadenin değerini hesaplayacağı zaman postfix forma dönüştürdükten (belirsizliği ortadan kaldırdıktan sonra) sonucu hesaplar : "3 2 * 5 6 * +" (Şekil 5.3).



Son iki eleman yığıttan çıkarılır, işlem yapılır, sonuç yığta yerleştirilir.

Şekil 5.3 : Bir postfix ifadenin "3 2 * 5 6 * +" sonucunun hesaplanması

Algoritma : (Bir postfix ifadenin sonucunu hesaplar)

```

opndstk = the empty stack;
// scan the input string reading one element at a time into symb
while (not end of input) {
    symb = next input character;
    if (symb is an operand)
        push(opndstk,symb);
    else {
        // symb is an operator
        opnd2 = pop(opndstk);
        opnd1 = pop(opndstk);
        value = result of applying symb (case *,./,+,-) to opnd1 and opnd2
        push(opndstk,value);
    } // end else
} // end while
return(pop(opndstk));

```

(optional : read **program to evaluate a postfix expression, converting an expression from infix to postfix** from textbook).

5.7 Fonksiyon ve metod çağrımları

Yığıtlar, listelerin ters sırada yazdırılması, palindrom (okunduğunda ve tersten okunduğunda aynı sonucu veren karakter dizisi) benzeri yapıların bulunması, bir ifadedeki parantez gibi sembollerin geçerliliğinin test edilmesi, ifadelerin sonuçlarının hesaplanıp değerlerinin elde edilmesi, infix ifadelerin postfix ifadeye dönüştürülmesi gibi amaçlarla kullanılabildiği gibi, programlama dili derleyicileri (compiler) fonksiyon çağrımlarında da yığıtlardan yararlanırlar.

Bir fonksiyon çağrıldığında, çağıran fonksiyonun yerel değişkenleri sistem tarafından kaydedilmelidir; aksi halde çağrılan fonksiyon çağıran fonksiyonun değişkenlerini ve değerlerini ortadan kaldıracaktır. Ayrıca çağıran fonksiyonda kalınan nokta (geri dönüş adresi), çağrılan fonksiyonun bitmesinden sonra geri dönmek üzere tutulmalıdır.

Soyut olarak bakıldığında, yeni bir fonksiyonun çağrılması ile çağıran fonksiyonun değişkenleri ve geri dönecek adres bir kağıda kaydedilir ve daha önce çağrılan fonksiyonlara ilişkin bilgilerin tutulduğu kağıtların üzerine konulur. Bundan sonra denetim çağrılan fonksiyona geçer. Kendi değişkenlerine yer açarak onlar üzerinde işlemler yapılmasını sağlar. Fonksiyondan geri döneceği zaman en üstteki kağıda bakılıp değişkenler üzerinde ilgili değişiklikler yapılarak geri dönüş adresine atlanır. Derleyici bu işlemleri kağıt yerine yığın kullanarak gerçekleştirir.

BÖLÜM 6

KUYRUKLAR (QUEUES)

6.1 Giriş

Bu bölümde gerçek yaşamdaki kuyrukların bilgisayardaki gösterimleri üzerinde durulacaktır. Kuyruklar, eleman eklemelerin sondan (rear) ve eleman çıkarmaların baştan (front) yapıldığı veri yapılarıdır. Bir eleman ekleneceği zaman kuyruğun sonuna eklenir. Bir eleman çıkarılacağı zaman kuyruktaki bulunan ilk eleman çıkarılır. Bu eleman da kuyruktaki elemanlar içinde ilk eklenen elemandır. Bu nedenle kuyruklara FIFO (First-In First-Out = ilk giren ilk çıkar) listeleri de denilmektedir. Gerçek yaşamda da bankalarda, duraklarda, gişelerde, süpermarketlerde, otoyollarda kuyruklar oluşmaktadır. Kuyruğa ilk olarak girenler işlemlerini ilk olarak tamamlayıp kuyruktan çıkarlar. Veri yapılarındaki kuyruklar bu tür veri yapılarının simülasyonunda kullanılmaktadır. Ayrıca işlemci, yazıcı, disk gibi kaynaklar üzerindeki işlemlerin yürütülmesinde ve bilgisayar ağlarında paketlerin yönlendirilmesinde de kuyruklardan yararlanılmaktadır.

6.2 Kuyruk İşlemleri ve Tanımları

insert(q,x) : q kuyruğunun sonuna x elemanını ekler. (enqueue)
x=remove(q) : q kuyruğunun başındaki elemanı silerek x'e atar. (dequeue)

(Seçimlik : **Stacks and Queues** konusunu okuyun)

6.3 Öncelik Kuyruğu (Priority Queue)

Yığıtlarda ve kuyruklarda elemanlar eklenme sırasında dayalı olarak sıralanırlar. Yığıtlarda ilk olarak son eklenen, kuyruklarda ise ilk eklenen eleman çıkarılır. Elemanlar arasındaki gerçek sıralama (sayısal sıra veya alfabetik sıra gibi) dikkate alınmaz.

Öncelik kuyrukları, temel kuyruk işlemlerinin sonuçlarını elemanların gerçek sırasının belirlediği veri yapılarıdır. Azalan ve artan sırada olmak üzere iki tür öncelik kuyruğu vardır. Artan öncelik kuyruklarında elemanlar herhangi bir yere eklenebilir ama sadece en küçük eleman çıkarılabilir. apq, artan öncelik kuyruğu olmak üzere pqinsert(apq,x) x elemanını kuyruğa ekler ve pqmindelete(apq) en küçük elemanı kuyruktan çıkararak değerini döndürür. Azalan öncelik kuyruğu ise artan öncelik kuyruğunun tam tersidir.

Artan öncelik kuyruğunda önce en küçük eleman, sonra ikinci küçük eleman sırayla çıkarılacağından dolayı elemanlar kuyruktan artan sırayla çıkmaktadırlar. Birkaç eleman çıkarıldıktan sonra ise daha küçük bir eleman eklenirse doğal olarak kuyruktan çıkarıldığında önceki elemanlardan daha küçük bir eleman çıkmış olacaktır.

Öncelik kuyruklarında sadece sayılar veya karakterler değil karmaşık yapılar da olabilir. Örnek olarak telefon rehberi listesi, soyad, ad, adres ve telefon numarası gibi elemanlardan oluşmaktadır ve soyada göre sıralıdır.

Öncelik kuyruklarındaki elemanların sırasının elemanların alanlarından birisine göre olması gerekmez. Elemanın kuyruğa eklenme zamanı gibi elemanların alanları ile ilgili olmayan dışsal bir değere göre de sıralı olabilirler.

Öncelik kuyruklarının gerçekleştiriminde dizi kullanımı etkin bir yöntem değildir.

6.4 Kuyruk Tasarımı ve Kullanımı

```
// Kuyruk sınıfı
class Kuyruk
{
    private int boyut;
    private int[] kuyrukDizi;
    private int bas;
    private int son;
    private int elemanSayisi;

    // Yapıcı Metot (Constructor)
    public Kuyruk(int s)
    {
        boyut = s;
        kuyrukDizi = new int[boyut];
        bas = 0;
        son = -1;
        elemanSayisi = 0;
    }

    public void ekle(int j) // Kuyruğun sonuna eleman ekler
    {
        if (son==boyut-1) son = -1;
        kuyrukDizi[++son] = j;
        elemanSayisi++;
    }
}
```

```
public int cikar()
{
    int temp = kuyrukDizi[bas++];
    if(bas==boyut) bas=0;
    elemanSayisi--;
    return temp;
}

public boolean bosMu()
{
    return(elemanSayisi==0);
}

}

// 1'den 10'a kadar olan sayilari kuyruğa yerlestirip
// sırayla çıkaran program
public class KuyrukTest
{
    public static void main(String args[])
    {
        Kuyruk k = new Kuyruk(25);

        k.ekle(1);
        k.ekle(2);
        System.out.println(k.cikar()); // 1
        k.ekle(3);
        for(int i=4; i<10; ++i)
            k.ekle(i);
        while(!k.bosMu())
            System.out.println(k.cikar());
    }
}
```

BÖLÜM 7

LİSTELER ve BAĞLI LİSTELER (LINKED LISTS)

7.1 Listeler

Günlük yaşamda listeler pek çok yerde kullanılmaktadır. Alışveriş listeleri, adres listeleri, davetli listeleri gibi. Bilgisayar programlarında da listeler yararlı ve yaygın olarak kullanılan veri yapılarındandır. Programlama açısından liste, aralarında doğrusal ilişki olan veriler topluluğu olarak görülebilir. Yığıt ve kuyrukların genişletilmesi yani üzerlerindeki sınırlamaların kaldırılması ile liste yapısına ulaşılır. Veri yapılarında değişik biçimlerde listeler kullanılmakta ve üzerlerinde değişik işlemler yapılmaktadır.

7.2 Listeler Üzerindeki Bazı İşlemler ve Tanımları

1. EmptyList(List) : returns Boolean

Listenin boş olup olmadığını belirleyen fonksiyon.

2. FullList(List) : returns Boolean

Listenin dolu olup olmadığını belirleyen fonksiyon.

3. LengthList(List) : returns integer

Listedeki eleman sayısını bulan fonksiyon.

4. InsertElement(List, NewElement)

Listeye yeni bir eleman ekleyen fonksiyon.

5. DeleteElement(List, Element)

Listeden bir elemanı arayarak çıkartan fonksiyon.

6. DestroyList(List)

Listedeki tüm elemanları silerek boş liste bırakan fonksiyon.

7. GetNextItem(List, Element)

Etkin elemandan bir sonrakini döndüren fonksiyon

8. RetrieveElement(List, Element, Found)

Elemanın listede olup olmadığını bulan ve döndüren fonksiyon.

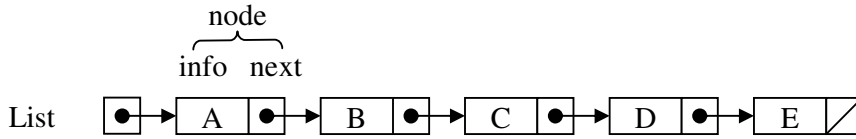
7.3 Bağlı (Bağlaçlı) Listeler

Kendi tipindeki bir yapıyı gösteren bir işaretçi üyesine sahip yapılara self-referential structures adı verilir. Örnek olarak :

```
struct node {  
    char info;  
    struct node *next; };
```


yapısı, info adlı karakter tipli bilgi elemanının yanında, bir düğüm yapısında bir bellek bölgesine işaret eden next işaretçisine sahiptir. Bu tür yapıların arka arkaya birbirine bağlanması mantığı listelerde, yığıtlarda, kuyruklarda ve ağaçlarda oldukça yararlıdır.

Listedeki her düğümde bir sonraki düğümün adresinin tutulduğu veri yapısı (doğrusal) bağlı liste olarak adlandırılır (Şekil 7.1). Listenin her bir elemanına düğüm (node) adı verilir. Düğümler, bilgi ve bağ (adres) sahalarından oluşmaktadırlar. Bağ sahalarında işaretçiler kullanılmaktadır. Listenin ilk elemanına dışarıdan bir işaretçi (list) ile erişilmektedir. Diğer düğümlere de bağlar yardımı ile ulaşılabilir. Son düğümün sonraki adres (next) sahası NULL değerini içerir. NULL bağı, liste sonunu belirtir. Elemanı olmayan liste boş liste olarak adlandırılır. Herhangi bir boyuta dinamik olarak genişletilip daraltılabilen yığıt ve kuyrukların gerçekleştirimi bağlı listeler üzerinde yapılmaktadır.



Şekil 7.1 : Doğrusal Bağlı Liste

Yığıtlarda ve kuyrukların gerçekleştiriminde sıralı bellek kullanımının (dizi) en büyük dezavantajı, hiç kullanılmasa veya az kullanılsa bile sabit miktardaki belleğin bu yapılara ayrılmış olarak tutulmasıdır. Ayrıca sabit bellek miktarı aşıldığında da taşma oluşması ve eleman ekleme işleminin yapılamamasıdır. Bağlı listeler üzerinde gerçekleştirildiklerinde ise bu problemler ortadan kalkmaktadır. Bellekten sadece gerektiği kadar yer ayrılmakta ve bellek boyutu bitene kadar bu yapılara ekleme işlemi yapılabilir.

Bağlı listeler, başka veri yapılarının gerçekleştiriminde kullanılabildikleri gibi kendileri de veri yapılarıdır. Bağlı listelerde elemanların eklenme ve çıkarılmasında bir sınırlama yoktur. Başa ve sona olduğu gibi araya da eleman eklenebilir; baştan ve sondan olduğu gibi ortadan da eleman çıkarılabilir. Bağlı liste dolaşarak herhangi bir elemanına erişilebilir. Bir bağlı listenin n. elemanına erişmek için n tane işlem yapmak yani kendinden önceki (n-1) eleman üzerinden geçmek gerekmektedir. Elemanların bellekteki yerleri dizilerdeki gibi sıralı olmadığından elemanlar ve sıraları ile yerleştikleri bellek bölgeleri arasında bir ilişki yoktur.

Bağlı listelerin diziler üzerine avantajı, bir grup eleman arasına eleman eklemeye ve bir grup eleman arasından eleman çıkarmada ortaya çıkar.

Dizilerde bir eleman silerken arada boşluk kalmasını engellemek için ilerisindeki (sağındaki) tüm elemanları bir geriye (sola) kaydırmak gerekir. Eleman eklemeye de yer açmak için konulacağı yerdeki ve ilerisindeki elemanları bir ileriye (sağa) kaydırmak gerekecektir. Kaç tane elemanın yer değiştireceği (biri kaydırılacağı) dizi boyutuna bağlı olarak ve eklenecek elemanın yerine bağlı olarak değişecektir. Bağlı listelerde ise eleman ekleme ve çıkarma için yapılan iş liste boyutundan bağımsızdır.

7.4 Öncelik Kuyruklarının Bağlı Liste Gerçekleştirimi

Yöntem 1 : (Sıralı liste tutularak) (Artan sıralı öncelik kuyruğunda)

Eleman ekleme, eklenecek elemanın listeyi sıralı tutacak şekilde liste üzerinde dolaşarak araya eklenmesi şeklinde gerçekleştirilir. Eleman çıkarma da, listenin ilk elemanının (en küçük değer) çıkarılması ile gerçekleştirilir.

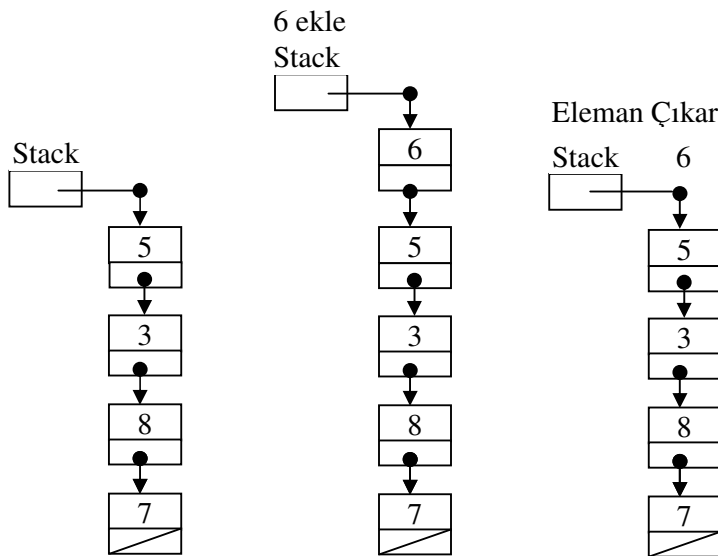
Yöntem 2 : (Sıralı olmayan liste) (Artan sıralı öncelik kuyruğunda)

Eleman ekleme kuyruğun herhangi bir yerine yapılabilir. Eleman çıkarma ise eleman bulunana kadar tüm kuyruk boyunca dolaşılması ve elemanın listeden çıkarılması ile gerçekleştirilir.

Öncelik kuyruklarında listelerin sıralanarak kullanımı sıralanmadan kullanımına göre daha etkindir.

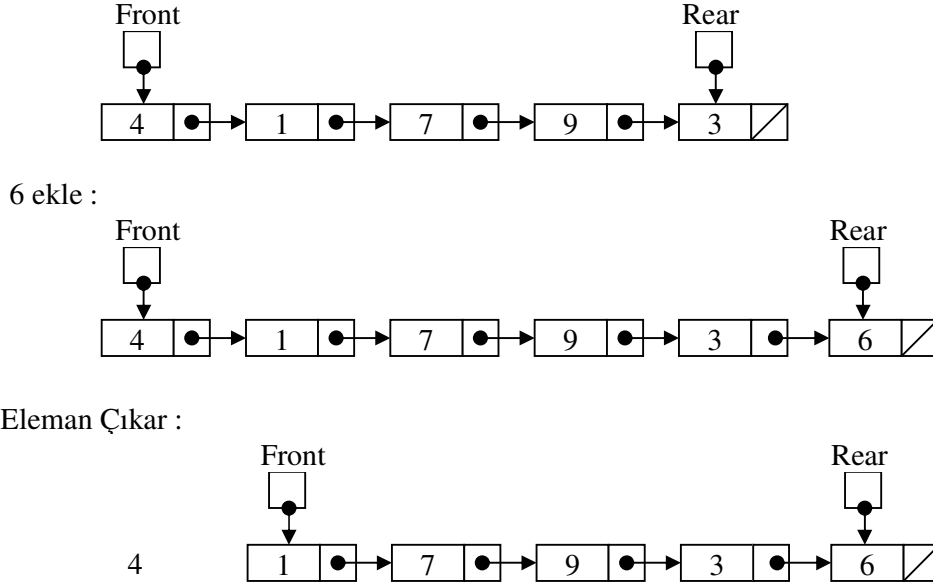
7.5 Yığıt ve Kuyrukların Bağlı Liste Gerçekleştirimleri

Yığıtların bağlı liste gerçekleştirimi Şekil 7.2’de görülmektedir :



Şekil 7.2 : Yığıtların bağlı liste gerçekleştirimi. Eleman ekleme ve çıkarma.

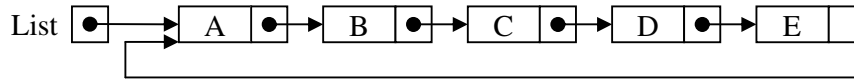
Kuyrukların bağlı liste gerçekleştirimi Şekil 7.3’de görülmektedir :



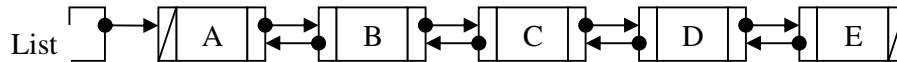
Şekil 7.3 : Kuyrukların bağlı liste gerçekleştirimi. Eleman ekleme ve çıkarma.

7.6 Diğer Bazı Liste Yapıları

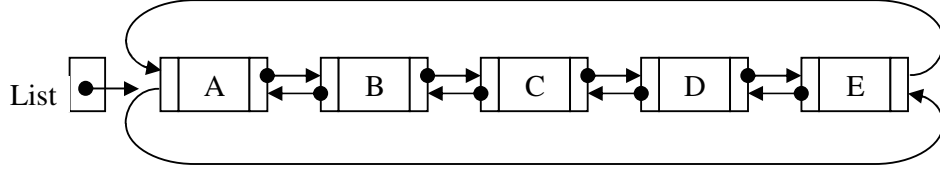
Dairesel Bağlı Listeler (Circular Linked Lists) : Tüm düğümlerin bir sonraki düğümü gösterdiği bağlı listelerdir. Son elemanın bağı NULL değildir; ilk elemanı gösterir. Böylece dairesel bir yapı oluşur.



Çift Bağlı Listeler (Doubly Linked Lists) : Her düğümü iki bağ içerdiği bağlı listelerdir. İlk bağ kendinden önceki düğümü gösterirken ikincisi de kendinden sonraki düğümü gösterir. Çift bağlı listelerde, tek bağlı listelerdeki geriye doğru listeleme ve dolaşmadaki zorluklar ortadan kalkar.



Dairesel Çift Bağlı Listeler (Circular Doubly Linked Lists) : Hem dairesellik hem de çift bağlantı özelliklerine sahip listelerdir. İlk düğümden önceki düğüm son, son düğümden sonraki düğüm de ilk düğümdür.



7.7 Java Programlama Dilinde Bağlı Liste Örneği ve Kullanımı

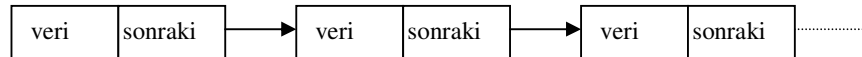
```
// Bağlı Listenin Düğüm Yapısı
//
//           Düğüm
//
//           veri | sonraki
```

```
class Dugum
{
    public int veri; // değişik tiplerde çoğaltılabilir
    public Dugum sonraki; // sonraki düğümün adresi

    public Dugum(int gelenVeri)    // Yapıcı metot
    { veri = gelenVeri; } // Düğüm yaratılırken değerini
                          // aktarır

    public void yazdir() // Düğümün verisini yazdırır
    { System.out.print(" "+veri); }
}
```

```
// Bağlı Liste Yapısı
//
//           bas
```



```
class BListe
{
    // Listenin ilk düğümünün adresini tutar
    private Dugum bas;
```

```
public BListe() // Bir BListe nesnesi yaratıldığında
{ bas = null; } // boş liste olarak açılır.

//Listede anahtar değerini bulur
public Dugum bul(int anahtar)
{
    Dugum etkin = bas;
    while(etkin.veri != anahtar)
    {
        if(etkin.sonraki==null)
            return null;
        else
            etkin = etkin.sonraki;
    };
    return etkin;
}

public void basaEkle(int yeniEleman)
{ // Liste başına eleman ekler
    Dugum yeniDugum = new Dugum(yeniEleman);
    yeniDugum.sonraki = bas;
    bas = yeniDugum;
}

public Dugum sil(int anahtar)
{ // Verilen anahtar değerindeki düğümü siler
    Dugum etkin = bas;
    Dugum onceki = bas;

    while(etkin.veri!=anahtar)
    {
        if(etkin.sonraki==null)
            return null;
        else
            { onceki = etkin; etkin = etkin.sonraki; }
    }

    if(etkin==bas)
        bas = bas.sonraki;
    else
        onceki.sonraki = etkin.sonraki;

    return etkin;
}
```

```

    public void listele()
    { // Bağlı Listeyi Baştan Sona Listeler
      System.out.println();
      System.out.print("Bastan Sona Liste : ");
      Dugum etkin = bas;
      while(etkin!=null)
      { etkin.yazdir(); etkin=etkin.sonraki; }
    }
  }

// Bir bağlı liste oluşturarak, Bliste ve Dugum
// sınıflarını metotlarıyla birlikte test eden sınıf

class BListeTest
{
  public static void main(String args[])
  {
    // liste adlı bir bağlı liste nesnesi yaratır
    BListe liste = new BListe();

    liste.basaEkle(9);
    for(int i=8; i>=1; --i) liste.basaEkle(i);

    liste.listele();

    int deger = 5;
    Dugum d = liste.bul(deger);

    if(d==null)
      System.out.println("\n"+deger+" Listede Yok");
    else
      System.out.println("\n"+deger+" Bulundu");

    Dugum s = liste.sil(5);

    liste.listele();
  }
}

Ekran Çıktısı :
// Bastan Sona Liste :  1 2 3 4 5 6 7 8 9
// 5 Bulundu
// Bastan Sona Liste :  1 2 3 4 6 7 8 9

```

BÖLÜM 8

ALGORİTMALARIN KARŞILAŞTIRILMASI

8.1 Giriş

Bir programın performansı genel olarak programın işletimi için gerekli olan bilgisayar zamanı ve belleğidir. Bir programın zaman karmaşıklığı (time complexity) programın işletim süresidir. Bir programın yer karmaşıklığı (space complexity) programın işletildiği sürece gerekli olan yer miktarıdır. Bir problemin çözümünde, kullanılabilecek olan algoritmalarından en etkin olanı seçilmelidir. En kısa sürede çözüme ulaşan veya en az işlem yapan algoritma tercih edilmelidir. Burada bilgisayarın yaptığı iş önemlidir. Bazı durumlarda da en az bellek harcayan algoritmanın tercih edilmesi gerekebilir. Ayrıca, programcının yaptığı iş açısından veya algoritmaların anlaşılabilirlikleri bakımından da algoritmalar karşılaştırılabilir. Daha kısa sürede biten bir algoritma yazmak için daha çok kod yazmak veya daha çok bellek kullanmak gerekebilir (trade-off).

Rakip algoritmaları yaptıkları iş açısından karşılaştırmak için her algoritmaya uygulanabilecek somut ölçüler tanımlanmalıdır. Aynı işi yapan algoritmalarından daha az işlemde sonuca ulaşan (hızlı olanın) belirlenmesi yani daha genel olarak algoritma analizi teorik bilgisayar bilimlerinin önemli bir alanıdır.

Yazılımcılar, iki farklı algoritmanın yaptıkları işi nasıl ölçüp karşılaştırırlar? İlk çözüm algoritmaları bir programlama dilinde kodlayıp her iki programı da çalıştırarak işletim sürelerini karşılaştırmaktır. İşletim süresi kısa olan daha iyi bir algoritma denilebilir mi? Bu yöntemde işletim süreleri belirli bir bilgisayara özeldir. Dolayısı ile işletim süresi de bu bilgisayara bağlıdır. Daha genel bir ölçüm yapabilmek için olası tüm bilgisayarlar üzerinde algoritmanın çalıştırılması gerekir.

İkinci çözüm, işletilen komut ve deyimlerin sayısını bulmaktır. Fakat bu ölçüm kullanılan programlama diline göre ve programcılarının stiline göre değişim gösterir. Bunun yerine algoritmadaki kritik geçişlerin sayısı hesaplanabilir. Her tekrar için sabit bir iş yapılıyor ve sabit bir süre geçiyorsa, bu ölçü anlamlı hale gelir.

Buradan, algoritmanın temelinde yatan bir işlemi ayırarak, bu işlemin kaç kere tekrarlandığını bulma düşüncesi doğmuştur. Örnek olarak bir tamsayı dizisindeki tüm elemanların toplamını hesaplama işleminde gerekli olan iş miktarını ölçmek için tamsayı toplama işlemlerinin sayısı bulunabilir. 100 elemanlı bir dizideki elemanların toplamını bulmak için 99 toplama işlemi yapmak gerekir. n elemanlı bir listedeki elemanların toplamını bulmak için $n-1$ toplama işlemi yapmak gerekir diye genelleştirme yapabiliriz. Böylece algoritmaları karşılaştırırken belirli bir dizi boyutu ile sınırlı kalınmaz.

İki gerçel matrisin çarpımında kullanılan algoritmaların karşılaştırılması istendiğinde, matris çarpımı için gereken gerçel sayı çarpma ve toplama işlemlerinin karışımı bir ölçü olacaktır. Bu örnekten ilginç bir sonuca ulaşılır: Bazı işlemlerin ağırlığı diğerlerine göre fazladır. Birçok bilgisayarda bilgisayar zamanı cinsinden gerçel sayı çarpımı gerçel sayı toplamından çok daha uzun sürer. Dolayısı ile tüm matris çarpımı düşünüldüğünde toplama işlemlerinin etkinlik üzerindeki etkisi az olacağından ihmal edilebilirler. Sadece çarpma işlemlerinin sayısı dikkate alınabilir. Algoritma analizinde genelde algoritmada egemen olan bir işlem bulunur ve bu diğerlerini gürültü (noise) düzeyine indirger.

8.2 Algoritmalarla Karmaşıklık (Complexity) ve Zaman Karmaşıklığı Analizi

8.2.1 İşletim Zamanı (Running Time)

İşletim zamanını girdi boyutunun bir fonksiyonu olarak ele almak tüm geçerli girdileri tek değere indirir. Bu da değişik algoritmaları karşılaştırmayı kolaylaştırır. En yaygın karmaşıklık ölçüleri “Worst –Case Running Time” (en kötü durum işletim süresi) ve “Average-Case Running Time” (ortalama durum işletim süresi)’dir.

Worst-Case Running Time :

Bu işletim süresi, her girdi boyutundaki herhangi bir girdi için en uzun işletim süresini tanımlar. Örnek olarak bir programın en kötü ihtimalle ne kadar süreceğinin tahmin edilmesi istenen bir durumdur. n elemanlı bir listede sıralı arama en kötü ihtimalle (aranan bulunamazsa) n karşılaştırma gerektirecektir. Yani worst-case running time (işletim zamanı) $T(n) = n$ ’dir. Tüm problemlerde sadece en kötü girdi dikkate alındığı için worst-case running time değerini hesaplamak göreceli olarak kolaydır.

Average-Case Running Time :

Bu işletim süresi, her girdi boyutundaki tüm girdilerin ortalamasıdır. n elemanın her birinin aranma olasılığının eşit olduğu varsayıldığında ve liste dışından bir eleman aranmayacağı varsayıldığında ortalama işletim süresi $(n+1)/2$ ’dir. İkinci varsayım kaldırıldığında ortalama işletim süresi $[(n+1)/2, n]$ aralığındadır (aranan elemanların listede olma eğilimine bağlı olarak). Ortalama durum analizi basit varsayımlar yapıldığında bile zordur ve varsayımlar da gerçek performansın iyi tahminlenememesine neden olabilir.

8.2.2 Asimptotik Analiz

Algoritmaların karşılaştırılmasında asimptotik etkinlikleri de dikkate alınabilir. Girdi boyutu sonsuza yaklaşırken işletim süresinin artışı. Asimptotik gösterimin elemanı olan 4 önemli gösterim vardır : O-notation, o-notation, Ω -notation, θ -notation. Burada sadece O gösterimi üzerinde durulacaktır. O gösterimi, fonksiyonların artış oranının üst sınırını belirler. $O(f(n))$, $f(n)$ fonksiyonundan daha hızlı artmayan fonksiyonlar kümesini gösterir.

8.2.2.1 Big-O Gösterimi (notasyonu)

n elemanlı bir listedeki elemanların toplamını bulmak için $n-1$ toplama işlemi yapmak gerekir diye genelleştirme yapmıştık. Yapılan işi, girdi boyutunun bir fonksiyonu olarak ele almış olduk. Bu fonksiyon yaklaşımını matematiksel gösterim kullanarak ifade edebiliriz : Big-O (O harfi, 0 sayısı değil) gösterimi veya büyüklük derecesi (order of magnitude). Büyüklük derecesini problemin boyutuna bağlı olarak fonksiyonda en hızlı artış gösteren terim belirler. Örnek olarak :

$$f(n) = n^4 + 100n^2 + 10n + 50 = O(n^4)$$

fonksiyonunda n ’in derecesi n^4 ’tür yani n ’in büyük değerleri için fonksiyonu en fazla n^4 etkiler. Peki daha düşük dereceli deyimlere ne olmaktadır? n ’in çok büyük değerleri için n^4 , $100n^2$ ’den, $10n$ ’den ve 50 ’den çok büyük olacağından daha düşük dereceli terimler dikkate

alınmayabilir. Bu diğer terimlerin, işlem süresini etkilemedikleri anlamına gelmez; bu yaklaşım yapıldığında n'in çok büyük değerlerinde önem taşımadıkları anlamına gelir.

n, problemin boyutudur. Yığın, liste, kuyruk, ağaç gibi veri yapılarında eleman sayılarıdır. n elemanlı bir dizi gibi ...

Bir listedeki tüm elemanların dosyaya yazılması için ne kadar iş yapılır : Cevap, listedeki eleman sayısına bağlıdır.

Algoritma

OPEN (Rewrite) the file

WHILE more elements in list DO

 Print the next element

İşlemi yapmak için geçen süre :

$(n * (\text{Bir elemanın dosyaya yazılması için geçen süre})) + \text{dosyanın açılması sırasında geçen süre}$

Algoritma $O(n)$ 'dir (Algoritmanın zaman karmaşıklığı $O(n)$ 'dir) . Çünkü, n tane işlem + sadece dosya açılması işlemi vardır. Yüzlerce elemanın dosyaya kaydedildiği düşünülürse, dosya açılması sırasında geçen süre miktarı rahatlıkla ihmal edilebilir. Ama az sayıda eleman varsa dosya açılması sırasında geçen süre miktarı önem taşıyabilir ve toplam süreye katılımı daha fazla olur.

Bir algoritmanın büyüklük derecesi, bilgisayarda işletildiğinde sonucun ne kadar sürede alınacağını belirtmez. Bazen de bu tür bir bilgiye gereksinim duyulur. Örnek olarak bir kelime işlemcinin 50 sayfalık bir yazı üzerinde yazım denetimi yapma süresinin birkaç saniye düzeyinden fazla olmaması istenir. Böyle bir bilgi istendiğinde, Big-O analizi yerine diğer ölçümler kullanılmalıdır. Program değişik yöntemlere göre kodlanır ve karşılaştırma yapılır. Programın çalıştırılmasından önce ve sonra bilgisayarın saati kaydedilir. İki saat arasındaki fark alınarak geçen süre bulunur. Bu tür bir "Benchmark" testi, işlemlerin belirli bir bilgisayarda belirli bir işlemci ve belirli kaynaklar kullanılarak ne kadar sürdüğünü gösterir.

Bilgisayarın yaptığı işin programın boyutu ile, örnek olarak satır sayısı ile ilgili olması gerekmez. N elemanlı bir diziyi 0'layan iki program da $O(n)$ olduğu halde kaynak kodlarının satır sayıları oldukça farklıdır :

Program 1 :

```
dizi[0] = 0;
dizi[1] = 0;
dizi[2] = 0;
dizi[3] = 0;
.....
dizi[n-1] = 0;
```

Program 2 :

```
for(int i=0; i<n; ++i)
    dizi[i] = 0;
```

1'den n'e kadar olan sayıların toplamını hesaplayan iki kısa programı düşünelim :

Program 1 :

```
toplam = 0;
for(int i=0; i<n; ++i)
    toplam = toplam + i;
```

Program 2 :

```
toplam = n * (n+1) / 2;
```

Program 1, $O(n)$ 'dir. $n=50$ olursa programın çalışması sırasında $n=5$ için harcanan sürenin yaklaşık 10 katı süre harcanacaktır. Program 2 ise $O(1)$ 'dir. $n=1$ de olsa $n=50$ 'de olsa program aynı sürede biter.

8.2.2.2 Artış Oranı Fonksiyonları

Yaygın olarak kullanılan bazı artış oranı fonksiyonlar Şekil 8.1'de gösterilmektedir.

| Fonksiyon | İsim |
|------------|-------------|
| 1 | constant |
| $\log n$ | logarithmic |
| n | linear |
| $n \log n$ | $n \log n$ |
| n^2 | quadratic |
| n^3 | cubic |
| 2^n | exponential |
| $n!$ | factorial |

Şekil 8.1 : Yaygın artış oranları

$O(1)$: Sabit zaman

Örnek : n elemanlı bir dizinin i . elemanına bir değer atanması $O(1)$ 'dir. Çünkü bir elemana indisinden doğrudan erişilmektedir.

$O(n)$: Doğrusal zaman

Örnek : n elemanlı bir dizinin tüm elemanlarının ekrana yazdırılması $O(n)$ 'dir.

Örnek : sıralı olmayan bir dizideki (listedeki) elemanlardan birinin aranması $O(n)$ 'dir (en kötü durumda da, ortalama durumda da).

$O(\log_2 n)$: $O(1)$ 'den fazla $O(n)$ 'den azdır.

Örnek : Sıralı bir listenin elemanları içinde ikili arama (binary search) uygulanarak belirli bir değer aranması $O(\log_2 n)$ 'dir.

$O(n^2)$: İkinci dereceli zaman

Örnek : Basit sıralama algoritmalarının birçoğu (selection sort gibi) $O(n^2)$ 'dir.

$O(n \log_2 n)$: Bazı hızlı sıralama algoritmaları $O(n \log_2 n)$ 'dir.

$O(n^3)$: Kübik zaman

Örnek : Üç boyutlu bir tamsayı tablosundaki her elemanın değerini artıran algoritma.

$O(2^n)$: Üstel zaman, çok büyük değerlere ulaşır.

8.2.2.3 Pratikte Karmaşıklık

Değişik artış fonksiyonlarının aldıkları değerlere göre bir tablo, Şekil 8.2’de gösterilmiştir.

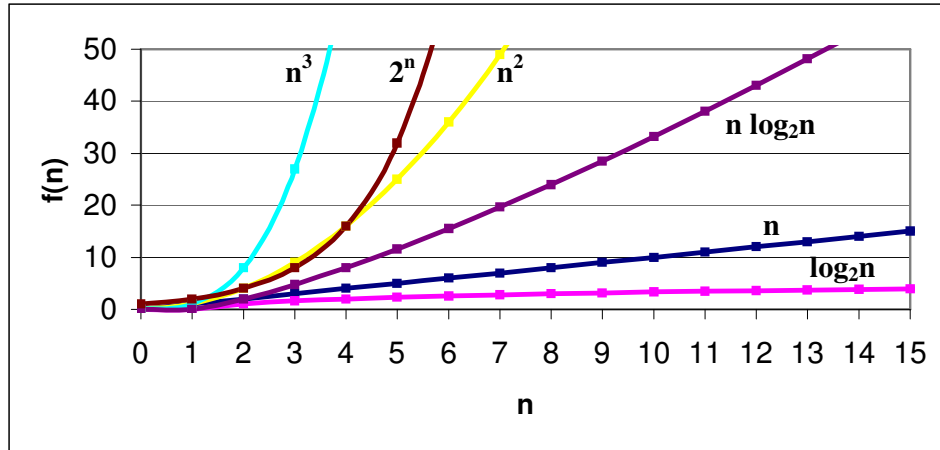
| $\log n$ | n | $n \log n$ | n^2 | n^3 | 2^n |
|----------|-----|------------|-------|-------|--------------------------|
| 0 | 1 | 0 | 1 | 1 | 2 |
| 1 | 2 | 2 | 4 | 8 | 4 |
| 2 | 4 | 8 | 16 | 64 | 16 |
| 3 | 8 | 24 | 64 | 512 | 256 |
| 4 | 16 | 64 | 256 | 4096 | 65536 |
| 5 | 32 | 160 | 1024 | 32768 | $\frac{4294967296}{296}$ |

Şekil 8.2 : Değişik fonksiyonların $f(n)$ değişik girdi boyutlarına (n) göre değerleri

Bir programın işletimi n^3 adım sürüyorsa, ve $n=1000$ ise, program 1000^3 adım sürecek demektir. Yani 1 000 000 000 (bir milyar) adım.

Kullanılan bilgisayar saniyede 1 000 000 000 adımı gerçekleştirebilecek kadar hızlı ise bu program tam 1 saniye sürecektir.

Şekil 8.2’deki fonksiyonlardan elde edilmiş bir grafik Şekil 8.3’te görülmektedir.



Şekil 8.3 : Değişik fonksiyonların grafikleri

BÖLÜM 9

SIRALAMA (SORTING)

9.1 Giriş

Sıralama ve arama tekniklerinden pek çok programda yararlanılmaktadır. Günlük yaşamımızda elemanların sıralı tutulduğu listeler yaygın olarak kullanılmaktadır. Telefon rehberindeki bir kişinin telefon numarasının bulunması bir arama (search) işlemidir. Telefon rehberlerindeki kayıtlar kişilerin soyadlarına göre sıralıdır. Bir telefon numarasının kime ait olduğunu bulmaya çalışmak da bir başka arama işlemidir. Eğer telefon rehberi kişilerin soyadlarına göre alfabetik olarak değil de telefon numaralarına göre kronolojik olarak sıralı olursa bu, arama işlemini basitleştirir. Kütüphanelerdeki kitapların özel yöntemlere göre sıralanarak raflara dizilmesi, bilgisayarın belleğindeki sayıların sıralanması da yapılacak arama işlemlerini hızlandırır ve kolaylaştırır. Genel olarak eleman toplulukları, bilgisayarda da, telefon rehberi gibi örneklerde de daha etkin erişmek (aramak ve bilgi getirmek) üzere sıralanır ve sıralı tutulurlar.

Eleman (kayıt, yapı ...) toplulukları genelde (her zaman değil) bir anahtara göre sıralı tutulurlar. Bu anahtar genelde elemanların bir alt alanı yani üyesidir (Elemanlar soyada göre sıralı olursa soyadı anahtardır, numaraya göre sıralı olursa numara anahtardır, nota göre olursa not alanı anahtardır). Elemanlar topluluğu içindeki her elemanın anahtar değeri kendinden önce gelen elemanın anahtar değerinden büyükse artan sırada, küçükse azalan sırada sıralıdır denilir (ilgili anahtara göre).

Sıralama, sıralanacak elemanlar bellekte ise internal (içsel), kayıtların bazıları ikincil bellek ortamındaysa external (dışsal) sıralama olarak adlandırılır.

Sıralama ve arama arasında bir ilişki olduğundan bir uygulamada ilk soru sıralama gerekli midir? olmalıdır. Arama işlemleri yoğun ise, sıralamanın veya o şekilde tutmanın getireceği yük, sıralı olmayan kayıtlar üzerindeki arama işlemlerinin getireceği toplam yük yanında çok hafif kalacaktır. Bu karar verilirse, arkasından sıralamanın nasıl yapılacağı ve hangi sıralama yöntemlerinin kullanılacağı kararlaştırılmalıdır. Bunun nedeni, tüm diğer yöntemlerden üstün evrensel bir sıralama tekniğinin olmamasındandır.

9.2 SIRALAMA TEKNİKLERİNİN ETKİNLİKLERİ ve ANALİZİ

9.2.1 Bubble Sort (Exchange Sorts kapsamında)

Bubble sort, sıralama teknikleri içinde anlaşılması ve programlanması kolay olmasına rağmen etkinliği en az olan algoritmalardandır (n elemanlı x dizisi için) :

```
void bubble(int x[], int n)
{
    int hold, j, pass; int switched = TRUE;

    for (pass=0; pass<n-1 && switched == TRUE; pass++)
    {
        switched = FALSE;
        for(j=0; j<n-pass-1; j++)
        {
            if (x[j] > x[j+1])
            {
                switched = TRUE;
                hold = x[j];
                x[j] = x[j+1];
                x[j+1] = hold;
            };
        };
    };
}
```

en fazla (n-1) iterasyon gerektirir.

```
Veriler      : 25 57 48 37 12 92 86 33
Tekrar 1     : 25 48 37 12 57 86 33 92
Tekrar 2     : 25 37 12 48 57 33 86 92
Tekrar 3     : 25 12 37 48 33 57 86 92
Tekrar 4     : 12 25 37 33 48 57 86 92
Tekrar 5     : 12 25 33 37 48 57 86 92
Tekrar 6     : 12 25 33 37 48 57 86 92
Tekrar 7     : Önceki turda değişen eleman olmadığından
yapılmaz.
```

Analizi kolaydır (İyileştirme yapılmamış algoritmada) :
 (n-1) iterasyon ve her iterasyonda (n-1) karşılaştırma.
 Toplam karşılaştırma sayısı : $(n-1)*(n-1) = n^2 - 2n + 1 = O(n^2)$

(Yukarıdaki gibi iyileştirme yapılmış algoritmada etkinlik) :
 iterasyon i'de, (n-i) karşılaştırma yapılacaktır.
 Toplam karşılaştırma sayısı = $(n-1)+(n-2)+(n-3)+\dots+(n-k)$
 = $kn - k*(k+1)/2$
 = $(2kn - k^2 - k)/2$
 ortalama iterasyon sayısı, k, O(n) olduğundan = $O(n^2)$

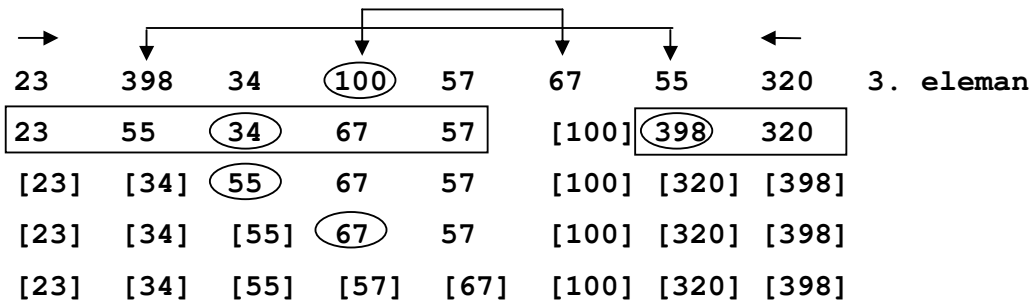
9.2.2 Quicksort (Exchange Sorts kapsamında)

```
#include <stdio.h>
```

```
void qsort2(double *left, double *right)
{
    double *p = left, *q = right, w, x=*(left+(right-left>>1));
    do
    {
        while(*p<x) p++;
        while(*q>x) q--;
        if(p>q) break;
        w = *p; *p = *q; *q = w;
    } while(++p <= --q);
    if(left<q) qsort2(left,q);
    if(p<right) qsort2(p,right);
}

void main()
{
    double dizi[8] = { 23, 398, 34, 100, 57, 67, 55, 320 };
    qsort2( &dizi[0], &dizi[7] );
}
```

n elemanlı bir dizi sıralanmak istendiğinde dizinin herhangi bir yerinden x elemanı seçilir (örnek olarak ortasındaki eleman). x elemanı j . yere yerleştildiğinde 0 . ile $(j-1)$. yerler arasındaki elemanlar x 'den küçük, $j+1$ 'den $(n-1)$ 'e kadar olan elemanlar x 'den büyük olacaktır. Bu koşullar gerçekleştirildiğinde x , dizide en küçük j . elemandır. Aynı işlemler, $x[0]-x[j-1]$ ve $x[j+1]-x[n-1]$ alt dizileri (parçaları) için tekrarlanır. Sonuçta veri grubu sıralanır.



Eğer şanslı isek, seçilen her eleman ortalama değere yakınsa $\log_2 n$ iterasyon olacaktır $= O(n \log_2 n)$. Ortalama durumu işletim zamanı da hesaplandığında $O(n \log_2 n)$ 'dir, yani genelde böyle sonuç verir. En kötü durumda ise parçalama dengesiz olacak ve n iterasyonla sonuçlanacağına $O(n^2)$ olacaktır (en kötü durum işletim zamanı).

9.2.3 Straight Selection Sort (Selection Sorts kapsamında)

Elemanların seçilerek uygun yerlerine konulması ile gerçekleştirilen bir sıralamadır :

```
void selectsort(int x[], int n)
{
    int i, indx, j, large;
    for(i=n-1; i>0; i--)
    {
        large = x[0];
        indx = 0;
        for(j=1; j<=i; j++)
            if(x[j]>large)
            {
                large = x[j];
                indx = j;
            };
        x[indx] = x[i];
        x[i] = large;
    };
}
```

| | | | | | | | | | |
|----------|---|----|----|----|----|----|----|----|----|
| Veriler | : | 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
| Tekrar 1 | : | 25 | 57 | 48 | 37 | 12 | 33 | 86 | 92 |
| Tekrar 2 | : | 25 | 57 | 48 | 37 | 12 | 33 | 86 | 92 |
| Tekrar 3 | : | 25 | 33 | 48 | 37 | 12 | 57 | 86 | 92 |
| Tekrar 4 | : | 25 | 33 | 12 | 37 | 48 | 57 | 86 | 92 |
| Tekrar 5 | : | 25 | 33 | 12 | 37 | 48 | 57 | 86 | 92 |
| Tekrar 6 | : | 25 | 12 | 33 | 37 | 48 | 57 | 86 | 92 |
| Tekrar 7 | : | 12 | 25 | 33 | 37 | 48 | 57 | 86 | 92 |

Selection Sort'un analizi doğrudandır.

1. turda (n-1),

2. turda (n-2),

3. ...

(n-1). Turda 1, karşılaştırma yapılmaktadır.

Toplam karşılaştırma sayısı = $(n-1)+(n-2)+\dots+1 = n*(n-1)/2$
 $= (1/2)n^2 - (1/2)n = O(n^2)$

9.2.4 Simple Insertion Sort (Insertion Sorts kapsamında)

Elemanların sırasına uygun olarak listeye tek tek eklenmesi ile gerçekleştirilen sıralamadır :

```
void insertsort(int x[], int n)
{
    int i,k,y;
    for(k=1; k<n; k++)
    {
        y=x[k];
        for(i=k-1; i>=0 && y<x[i]; i--)
            x[i+1]=x[i];
        x[i+1]=y;
    };
}
```

| | | | | | | | | |
|----------|----|----|----|----|----|----|----|----|
| Veriler | 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
| Tekrar 1 | 25 | 57 | 48 | 37 | 12 | 92 | 86 | 33 |
| Tekrar 2 | 25 | 48 | 57 | 37 | 12 | 92 | 86 | 33 |
| Tekrar 3 | 25 | 37 | 48 | 57 | 12 | 92 | 86 | 33 |
| Tekrar 4 | 12 | 25 | 37 | 48 | 57 | 92 | 86 | 33 |
| Tekrar 5 | 12 | 25 | 37 | 48 | 57 | 92 | 86 | 33 |
| Tekrar 6 | 12 | 25 | 37 | 48 | 57 | 86 | 92 | 33 |
| Tekrar 7 | 12 | 25 | 33 | 37 | 48 | 57 | 86 | 92 |

Eğer veriler sıralı ise her turda 1 karşılaştırma yapılacaktır ve $O(n)$ olacaktır. Veriler ters sıralı ise toplam karşılaştırma sayısı: $(n-1)+(n-2)+\dots+3+2+1 = n*(n+1)/2 = O(n^2)$ olacaktır.

Simple Insertion Sort'un ortalama karşılaştırma sayısı ise $O(n^2)$ 'dir.

Selection Sort ve Simple Insertion Sort, Bubble Sort'a göre daha etkindir. Selection Sort, Insertion Sort'tan daha az atama işlemi yaparken daha fazla karşılaştırma işlemi yapar. Bu nedenle Selection Sort büyük kayıtlardan oluşan az elemanlı veri grupları için (atamaların süresi çok fazla olmaz) ve karşılaştırmaların daha az yük getireceği basit anahtarlı durumlarda uygundur. Tam tersi için, insertion sort uygundur. Elemanlar bağlı listedelerse araya eleman eklemelerde veri kaydırma olmayacağından insertion sort mantığı uygundur.

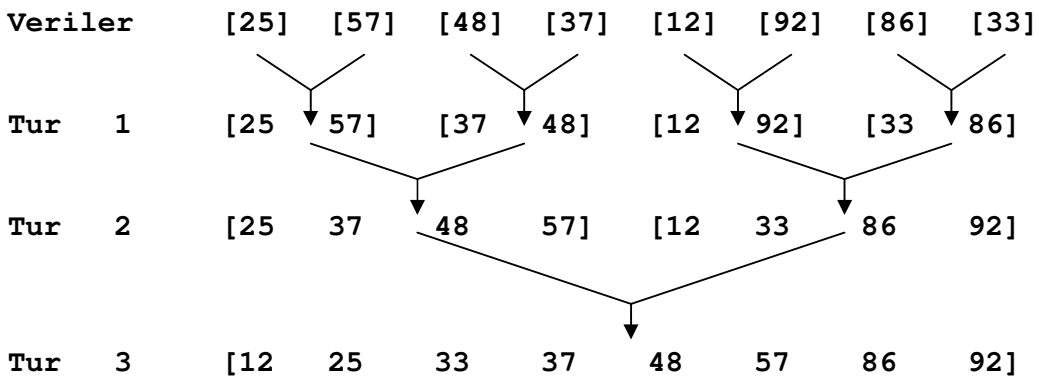
n'in büyük değerleri için quicksort insertion ve selection sort'tan daha etkindir. Quicksort'u kullanmaya başlama noktası yaklaşık 30 elemanlı durumlardır; daha az elemanın sıralanması gerektiğinde insertion sort kullanılabilir.

9.2.5 Merge Sort (Merge Sorts kapsamında)

Sıralı iki veri grubunu birleştirerek üçüncü bir sıralı veri grubu elde etmeye dayanır.

```
#include <stdio.h>
#define numelts 8

void mergesort(int x[], int n)
{
    int aux[numelts], i, j, k, l1, l2, size, u1, u2;
    size = 1;
    while(size < n) {
        l1 = 0;
        k = 0;
        while(l1 + size < n) {
            l2 = l1 + size;
            u1 = l2 - 1;
            u2 = (l2 + size - 1 < n) ? l2 + size - 1 : n - 1;
            for(i = l1, j = l2; i <= u1 && j <= u2; k++)
                if(x[i] <= x[j]) aux[k] = x[i++]; else aux[k] = x[j++];
            for(; i <= u1; k++) aux[k] = x[i++];
            for(; j <= u2; k++) aux[k] = x[j++];
            l1 = u2 + 1;
        };
        for(i = l1; k < n; i++)
            aux[k++] = x[i];
        for(i = 0; i < n; i++)
            x[i] = aux[i];
        size *= 2;
    };
}
```



Analiz : $\log_2 n$ tur ve her turda n veya daha az karşılaştırma.
 $= O(n \log_2 n)$ karşılaştırma. Quicksort'ta en kötü durumda $O(n^2)$ karşılaştırma gerektiği düşünülürse daha avantajlı. Fakat mergesort'ta atama işlemleri fazla ve aux dizi için daha fazla yer gerekiyor.

BÖLÜM 10

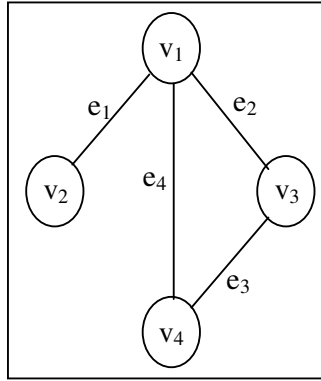
ÇİZGELER (GRAPHS) ve UYGULAMALARI

10.1 Terminoloji

Çizge (Graph) : Köşe (vertex) adı verilen düğümlerden ve kenar (edge) adı verilip köşeleri birbirine bağlayan bağlantılardan oluşan veri yapısıdır. Aynen ağaçlar gibi çizgeler de doğrusal olmayan veri yapıları grubuna girerler.

graph $G=(V,E)$, sonlu V ve E elemanları kümesidir.

V 'nin elemanları köşeler (vertices) olarak adlandırılır. E 'nin elemanları da kenarlar (edges) olarak adlandırılır. E 'nin içindeki her kenar V içindeki iki farklı köşeyi birleştirir. Bir çizgede köşeler dairelerle, kenarlar da çizgilerle gösterilir (Şekil 10.1).

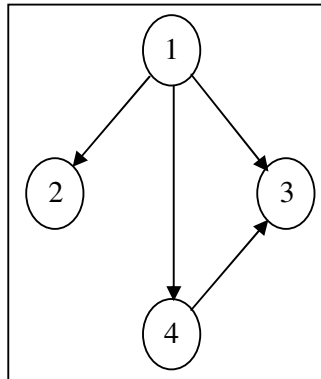


$G=(V,E)$
 $V=\{v_1, v_2, v_3, v_4\}$
 $E=\{(v_1, v_2), (v_1, v_3), (v_3, v_4), (v_1, v_4)\}$
 $e_1=(v_1, v_2)$
 $e_2=(v_1, v_3)$
 $e_3=(v_3, v_4)$
 $e_4=(v_1, v_4)$
 $E=\{e_1, e_2, e_3, e_4\}$

Şekil 10.1 : Çizge

Yönsüz Kenar (undirected edge) : Çizgi şeklinde yönü belirtilmeyen kenarlar yönsüz kenarlardır. Yönsüz kenarlarda (v_1, v_2) olması ile (v_2, v_1) olması arasında fark yoktur. Örnekler: Şekil 10.1'deki çizgedeki kenarlar.

Yönlü Kenar (directed edge) : Ok şeklinde gösterilen kenarlar yönlü kenarlardır (Şekil 10.2). (i,j) 'de okun başı ikinci köşeyi (j), okun kuyruğu birinci köşeyi (i) gösterir. Bazı kitaplarda $\langle i,j \rangle$ şeklinde gösterilir.



$G_1=(V,E)$
 $V_1=\{1, 2, 3, 4\}$
 $E_1=\{(1,2), (1,3), (1,4), (4,3)\}$

Şekil 10.2 : Yönlü ve Bağlı Çizge

Komşu Köşeler (Adjacent) : Aralarında doğrudan bağlantı (kenar) bulunan i ve j köşeleri komşudur. Diğer köşe çiftleri komşu değildir. Örnek : (Şekil 10.1’de) v_1 ve v_2 ; v_1 ve v_3 ; v_1 ve v_4 ; v_3 ve v_4 köşe çiftleri komşudur.

Bağlantı (incident) : Komşu i ve j köşeleri arasındaki kenar (i,j) bağlantıdır.

Bir Köşenin Derecesi (degree) : Bir köşeye bağlı olan kenarların sayısıdır. Şekil 10.1’de v_1 ’in derecesi 3, v_2 ’nin derecesi 1, v_3 ’ün derecesi 2, v_4 ’ün derecesi 2’dir.

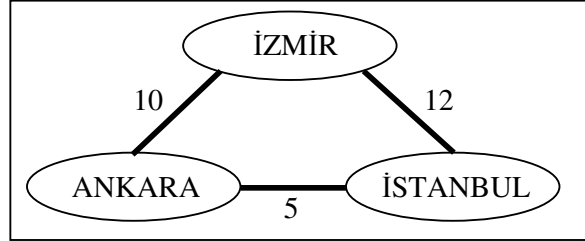
Indegree, Outdegree : Yönlü çizgede, yönlü kenar (i,j) , j köşesine gelendir (incident to), i köşesinden çıkandır (incident from). Bir köşeye gelenlerin sayısına indegree, bir köşeden çıkanların sayısına outdegree denilir. Şekil 10.2’deki köşelerin indegree ve outdegree’lerini bulunuz.

Yönsüz Çizge (undirected graph) : Tüm kenarları yönsüz olan çizgeye yönsüz çizge denilir. Yönsüz çizgede bir köşe çifti arasında en fazla bir kenar olabilir.

Yönlü Çizge (directed graph, digraph) : Tüm kenarları yönlü olan çizgeye yönlü çizge adı verilir. Yönlü çizgede bir köşe çifti arasında ters yönlerde olmak üzere en fazla iki kenar olabilir.

Döngü (Loop) : (i,i) şeklinde gösterilen ve bir köşeyi kendine bağlayan kenar.

Ağırlıklı Çizge (weighted graph) : Her kenara bir ağırlık (weight) veya maliyet (cost) değerinin atandığı çizge (Şekil 10.3).



Şekil 10.3 : Ağırlıklı Çizge

Yol (path) : $G(V,E)$ çizgesinde i_1 ve i_k köşeleri arasında $P=i_1,i_2,...,i_k$ şeklinde belirtilen köşeler dizisi (E ’de, $1 \leq j < k$ olmak üzere, her j için (i_j, i_{j+1}) şeklinde gösterilen bir kenar varsa). Şekil 10.3’te İZMİR’den ANKARA’ya doğrudan veya İSTANBUL’dan geçerek gidilebilir (İZMİR İSTANBUL ANKARA).

Basit Yol (Simple Path) : Tüm düğümlerin farklı olduğu yoldur.

Uzunluk : Bir yol üzerindeki kenarların uzunlukları toplamı o yolun uzunluğudur.

Bağlı Çizge (Connected Graph) : Her köşe çifti arasında en az bir yol olan ağaç.

Alt Çizge (Subgraph) : H çizgesinin köşe ve kenarları G çizgesinin köşe ve kenarlarının alt kümesi ise; H çizgesi G çizgesinin alt çizgesidir (subgraph).

Daire veya devir (Cycle) : Başlangıç ve bitiş köşeleri aynı olan basit yol. Şekil 10.3'te İZMİR İSTANBUL ANKARA İZMİR.

Ağaç (tree) : Daire içermeyen yönsüz bağlı çizge.

Spanning Tree : G'nin tüm köşelerini içeren bir ağaç şeklindeki alt çizgelerden her biri.

Forest : Bağlı olma zorunluluğu olmayan ağaç.

Complete Graph : n köşe sayısı olmak üzere $n*(n-1)/2$ kenarı olan çizge (kendilerine bağlantı yok). Şeklini düşününüz, 1,2,3,4 köşe sayıları için.

Complete Digraph : n köşe sayısı olmak üzere $n*(n-1)$ kenarı olan çizge.

10.2 Çizgelerin Kullanım Alanları

Bilgisayar Ağlarında, elektriksel ve diğer ağların analizinde, kimyasal bileşiklerin moleküler yapılarının araştırılmasında, ulaşım ağlarında (kara, deniz ve havayolları), planlama projelerinde, sosyal alanlarda ve diğer pek çok alanda kullanılmaktadır.

EK 1

JAVA'DA HAZIR VERİ YAPILARI ve KOLEKSİYONLAR

Java'da veri yapıları diğer dillerde olduğu gibi programlanarak oluşturulabildiği gibi, dilde olan hazır veri yapıları da kullanılabilir.

Collection (koleksiyon), diğer verileri tutabilen veri yapısıdır. Koleksiyon arayüzleri (collection interfaces), her tür koleksiyonda yapılabilecek işlemleri tanımlar. Koleksiyon gerçekleştirimleri (collection implementations), bu işlemleri çeşitli yollarla gerçekleştirir. Bazı arayüzler : Set, List, Map olup Java.util paketi içindedirler.

Arayüzler

Liste (List) : Sıralı bir tür koleksiyondur. Tekrarlı elemanları içerebilir. Listeler 0. elemandan başlar. Koleksiyondan devraldığı özellikler dışında, indislerine göre elemanları işleme (sort,...), eleman arama (binarySearch) ve elemanlar üzerinde dolaşma gibi metotları (ListIterator) da vardır. Liste arayüzü, "ArrayList", "LinkedList" ve "Vector" sınıfları ile gerçekleştirilir. "ArrayList" sınıfı, boyutu değiştirilebilen dizidir ve "Vector" sınıfından hızlı çalışır. "LinkedList" sınıfı ise bağlı liste gerçekleştirimidir. Çok sayıda metot devralmaktadırlar. Çift bağlı liste, kuyruk, yığıt (yığıt için Java'da ayrıca sınıf da vardır) vs. de gerçekleştirilebilmektedir.

Küme (Set) : Küme, elemanları tek (tekrar olmadan) tutan koleksiyon tipinde veri yapısıdır. İki önemli Küme gerçekleştirmesi : HashSet ve TreeSet'tir. HashSet, elemanlarını "Hash" tablosunda tutar, TreeSet ise ağaçta tutar.

JAVA ÖRNEK (bsearch.java) : Sıralama, İkili Arama

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class bsearch extends JFrame
{
    public bsearch()
    {
        super("BSearch Ornek");

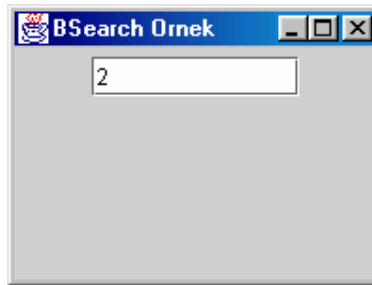
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        final JTextField tfoutput = new JTextField(10);
        c.add(tfoutput);

        //    int a[] = new int[10];
        //    float, char, ... için de yapılabilir.
        int a[] = { 4,2,1,8,6,7,9,15,11 };
        Arrays.sort(a);
        int ind = Arrays.binarySearch(a,4);

        tfoutput.setText(""+ind);

        setSize(200,150); show();
    }

    public static void main ( String args[] )
    {
        bsearch app = new bsearch();
        app.addWindowListener
        (
            new WindowAdapter()
            {
                public void windowClosing(WindowEvent e)
                {
                    System.exit(0);
                }
            }
        );
    }
}
```



Şekil 1 :

Dizi sıralandıktan (sort metodu ile) sonra
1,2,4,6,7,8,9,11,15
içinde (binarySearch metodu ile) 4 değeri
aranmaktadır.

Dizi içindeki konumu (2) metin kutusuna yazdırılır.

JAVA ÖRNEK (SortedSetTest.java) : Ağaç

```
import java.util.*;

public class SortedSetTest {
    private static String names[] = { "yellow", "green",
        "black", "tan", "grey", "white", "orange", "red", "green"
    };

    public SortedSetTest()
    {
        TreeSet m = new TreeSet(Arrays.asList(names));
        System.out.println("Set: ");
        printSet(m);

        System.out.print("orange'dan öncekiler :");
        printSet(m.headSet("orange"));
        System.out.print("orange'dan sonrakiler:");
        printSet(m.tailSet("orange"));

        System.out.println("İlk eleman :"+m.first());
        System.out.println("Son eleman :"+m.last());
    }

    public void printSet(SortedSet setRef)
    {
        Iterator i = setRef.iterator();
        while(i.hasNext())
            System.out.print(i.next()+" ");
        System.out.println();
    }

    public static void main(String args[])
    { new SortedSetTest(); }
}
```

Ekran Çıktısı :

Set:

black green grey orange red tan white yellow

orange'dan öncekiler :black green grey

orange'dan sonrakiler:orange red tan white yellow

İlk eleman :black

Son eleman :yellow

EK 2

C# PROGRAMLAMA ve ÖRNEKLER

C# Programının İşletimi (CMD Konsol)

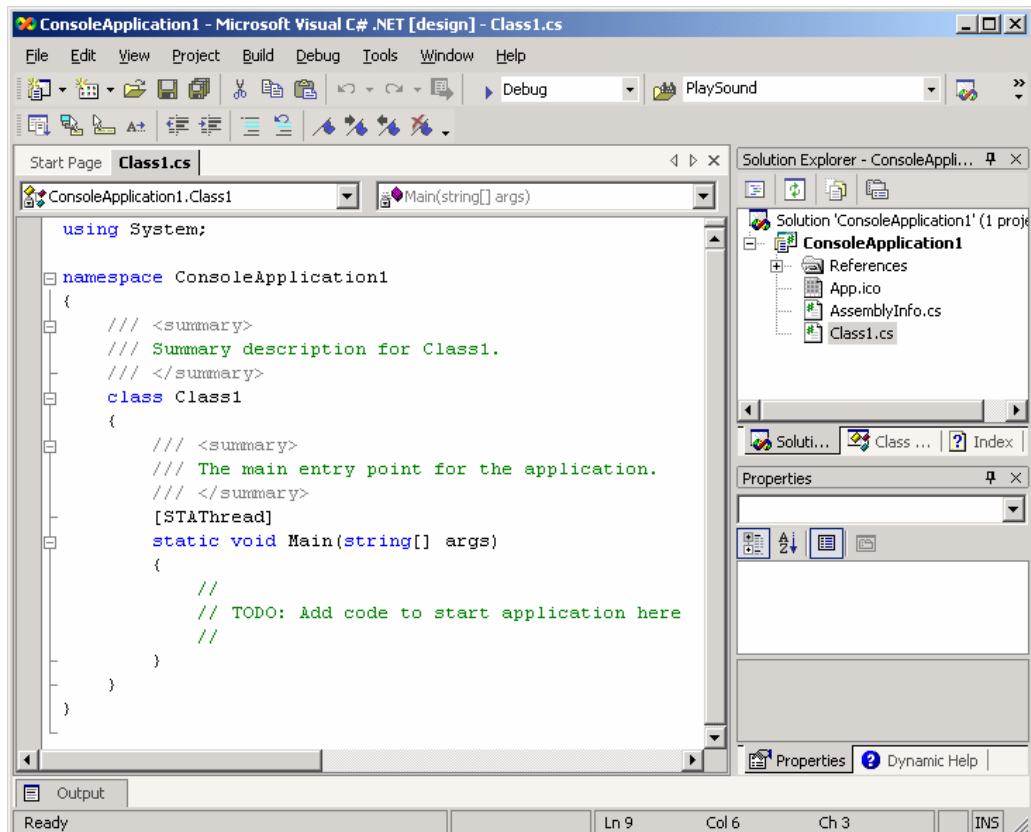
- Aşağıdaki program herhangi bir isim ile kaydedilir (Ornek1.cs gibi).
- “csc Ornek1.cs” komutu verilerek derlenir. (path ayarı!).
- “Ornek1” programı çalıştırılır.

```
using System;

class Merhaba
{
    static void Main (string[] args)
    {
        Console.WriteLine("Merhaba");
    }
}
```

C# Programının İşletimi (IDE Konsol)

New Project – Visual C# Projects – Console Application



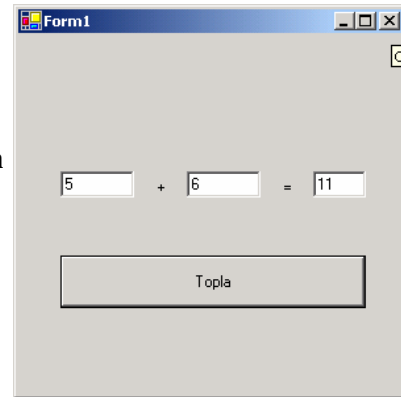
C# Programının İşletimi (Windows)

New Project – Visual C# Projects – Windows Application

- İki Etiket
- Üç Metin Kutusu
- Bir Düğme eklenir.

..... Düğme için aşağıdaki kod yazılır:

```
private void button1_Click(object sender, System.EventArgs e)
{
    textBox3.Text = "" + (Double.Parse (textBox2.Text) +
                          Double.Parse (textBox1.Text));
}
```



Sınıf Örneği : Rasyonel Sayı Sınıfı ve Kullanımı

```
using System;

class rasyonel_sayi
{
    long pay;
    long payda;

    public rasyonel_sayi()
    { pay = 2; payda = 3; }

    public rasyonel_sayi(long pay, long payda)
    { this.pay = pay; this.payda = payda; }

    public void yazdir()
    { Console.WriteLine("{0}/{1}", pay, payda); }
}

class main
{
    public static void Main()
    {
        rasyonel_sayi r1 = new rasyonel_sayi();
        r1.yazdir();
    }
}
```

Sonuç : 2/3

C# : Diziler**ÖRNEK 1)**

```
using System;
class Test
{
    static void Main() {
        int[] arr = new int[5];
        for (int i = 0; i < arr.Length; i++)
            arr[i] = i * i;
        for (int i = 0; i < arr.Length; i++)
            Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
    }
}
```

Ekran Çıktısı :

| | | |
|--------|---|----|
| arr[0] | = | 0 |
| arr[1] | = | 1 |
| arr[2] | = | 4 |
| arr[3] | = | 9 |
| arr[4] | = | 16 |

ÖRNEK 2)

```
class Test
{
    static void Main() {
        int[] a1;    // single-dimensional array of int
        int[,] a2;   // 2-dimensional array of int
        int[,,] a3;  // 3-dimensional array of int
        int[][] j2;  // "jagged" array: array of (array of int)
        int[][][] j3; // array of (array of (array of
int))
    }
}
```

ÖRNEK 3)

```
int[] tamsayi = { 5, 10, 15 };
```

tamsayi

| | | |
|---|----|----|
| 5 | 10 | 15 |
|---|----|----|

string[] str = new string[3];

| |
|--------|
| Ali |
| Cemile |
| Veli |

double[,] cift_duyarlik = new double[2, 2];

| | |
|-------|------|
| 15.12 | 7.64 |
| 56.01 | -3.9 |

bool[][] isEmpty = new bool[2][];
isEmpty[0] = new bool[2];
isEmpty[1] = new bool[1];

| | |
|-------|-------|
| true | false |
| false | |

C# : ÖZYİNELEME ÖRNEĞİ (Faktöryel)

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WindowsApplication3
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
        private System.Windows.Forms.TextBox textBox1;
        private System.Windows.Forms.TextBox textBox2;
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.Button button1;
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components = null;

        public Form1()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();

            //
            // TODO: Add any constructor code after InitializeComponent call
            //
        }

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        protected override void Dispose( bool disposing )
        {
            if( disposing )
            {
                if (components != null)
                {
                    components.Dispose();
                }
            }
            base.Dispose( disposing );
        }

        #region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.textBox1 = new System.Windows.Forms.TextBox();
```

```

this.textBox2 = new System.Windows.Forms.TextBox();
this.label1 = new System.Windows.Forms.Label();
this.label2 = new System.Windows.Forms.Label();
this.button1 = new System.Windows.Forms.Button();
this.SuspendLayout();
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(16, 32);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(48, 20);
this.textBox1.TabIndex = 0;
this.textBox1.Text = "";
//
// textBox2
//
this.textBox2.Location = new System.Drawing.Point(112, 32);
this.textBox2.Name = "textBox2";
this.textBox2.Size = new System.Drawing.Size(56, 20);
this.textBox2.TabIndex = 1;
this.textBox2.Text = "";
//
// label1
//
this.label1.Location = new System.Drawing.Point(16, 16);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(56, 16);
this.label1.TabIndex = 2;
this.label1.Text = "Sayı";
//
// label2
//
this.label2.Location = new System.Drawing.Point(112, 16);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(56, 16);
this.label2.TabIndex = 3;
this.label2.Text = "Faktöryel";
//
// button1
//
this.button1.Location = new System.Drawing.Point(16, 72);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(152, 24);
this.button1.TabIndex = 4;
this.button1.Text = "Hesapla";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// Form1
//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(184, 109);
this.Controls.AddRange(new System.Windows.Forms.Control[] {

    this.button1,

    this.label2,

    this.label1,

    this.textBox2,

```

```

        this.textBox1});
    this.Name = "Form1";
    this.Text = "Form1";
    this.Load += new System.EventHandler(this.Form1_Load);
    this.ResumeLayout(false);
}
#endregion

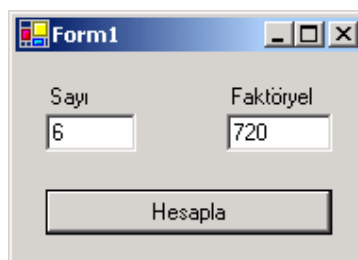
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
    Application.Run(new Form1());
}

private void button1_Click(object sender,
System.EventArgs e)
{
    uint n = UInt32.Parse(textBox1.Text);
    textBox2.Text = "" + factorial(n);
}

uint factorial(uint n)
{
    if(n==0)
        return 1;
    else
        return(n*factorial(n-1));
}

private void Form1_Load(object sender, System.EventArgs e)
{
}
}
}

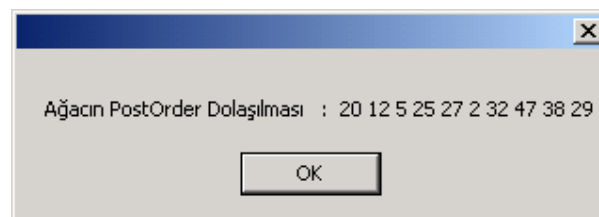
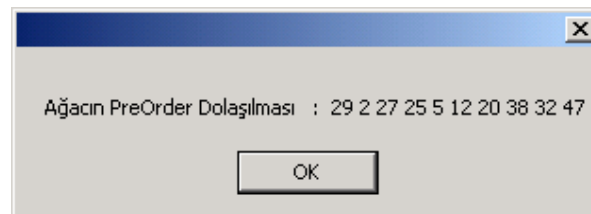
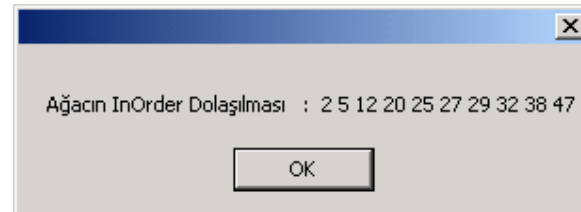
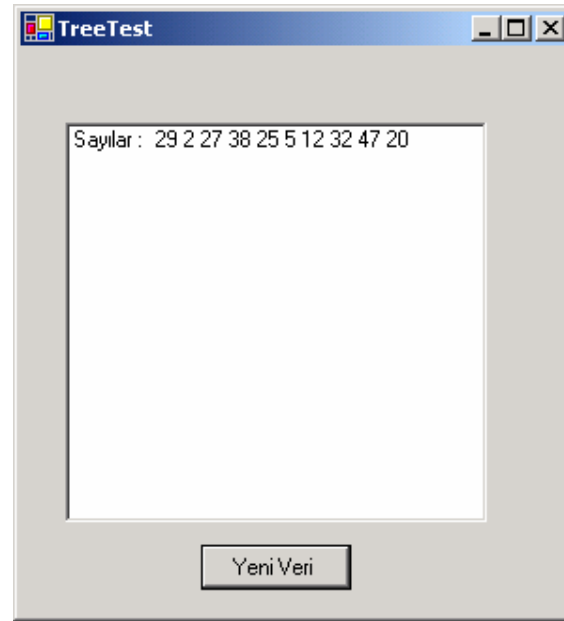
```



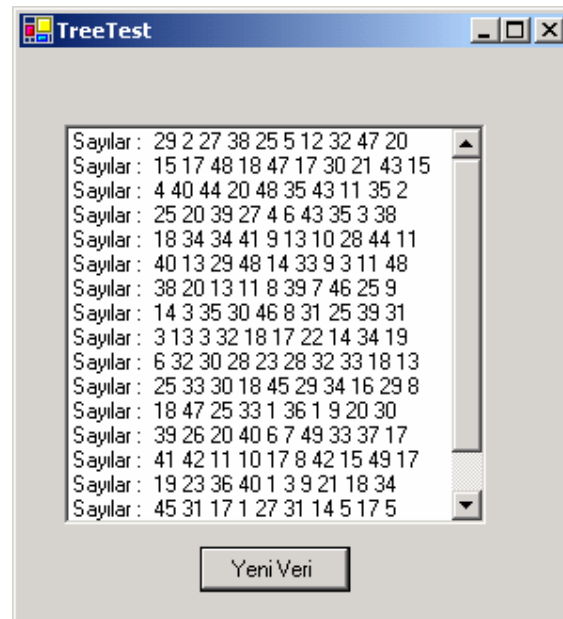
C# : İKİLİ AĞAÇ ÖRNEĞİ

C# programlama dili ile İkili Ağaçlara ilişkin bir örnek aşağıdaki gibidir.

“TreeTest” adlı programın formu üzerinde 1 adet liste kutusu ve 1 adet düğme bulunmaktadır. “Yeni Veri” düğmesine basıldıkça, 1 ile 50 arasında 10 tane rastgele sayı üretmektedir ve arka arkaya gelen üç mesaj penceresi ile, InOrder, PreOrder ve PostOrder dolaşmalarda ikili ağaç üzerinde hangi düğüm sırasının izleneceğini göstermektedir.



Program, istenildiği kadar deneme yapma fırsatı verecek şekilde yazılmıştır.



C# İKİLİ AĞAÇ PROGRAMININ KAYNAK KODU

```
public class GLOBAL
{
    public static string tempStr;
}

class TreeNode
{
    public int data;
    public TreeNode leftChild;
    public TreeNode rightChild;

    public void displayNode()
    { GLOBAL.tempStr += (" "+data); }
}

// Ağaç Sınıfı
class Tree
{
    private TreeNode root;

    public Tree()
    { root = null; }

    public TreeNode getRoot()
    { return root; }
```

```
// Ağacın preOrder Dolaşılması
public void preOrder(TreeNode localRoot)
{
    if(localRoot!=null)
    {
        localRoot.displayNode();
        preOrder(localRoot.leftChild);
        preOrder(localRoot.rightChild);
    }
}

// Ağacın inOrder Dolaşılması
public void inOrder(TreeNode localRoot)
{
    if(localRoot!=null)
    {
        inOrder(localRoot.leftChild);
        localRoot.displayNode();
        inOrder(localRoot.rightChild);
    }
}

// Ağacın postOrder Dolaşılması
public void postOrder(TreeNode localRoot)
{
    if(localRoot!=null)
    {
        postOrder(localRoot.leftChild);
        postOrder(localRoot.rightChild);
        localRoot.displayNode();
    }
}

// Ağaca bir düğüm eklemeyi sağlayan metot
public void insert(int newdata)
{
    TreeNode newNode = new TreeNode();
    newNode.data = newdata;
    if(root==null)
        root = newNode;
    else
    {
        TreeNode current = root;
        TreeNode parent;
        while(true)
        {
            parent = current;
            if(newdata<current.data)
            {
                current = current.leftChild;
                if(current==null)

```

```

        {
            parent.leftChild=newNode;
            return;
        }
    }
    else
    {
        current = current.rightChild;
        if(current==null)
        {
            parent.rightChild=newNode;
            return;
        }
    }
} // end while
} // end else not root
} // end insert()

} // class Tree

private void dugme1_Click(object sender, System.EventArgs
e)
{
    Random r = new Random();

    Tree theTree = new Tree();

    // Ağaca 10 tane sayı yerleştirilmesi

    string str = "";

    str += "Sayılar : ";

    for (int i=0;i<10;++i)
    {
        int sayi = (int) (r.Next(1,50));
        str += (" "+sayi);
        theTree.insert(sayi);
    };

    listBox1.Items.Add(str);

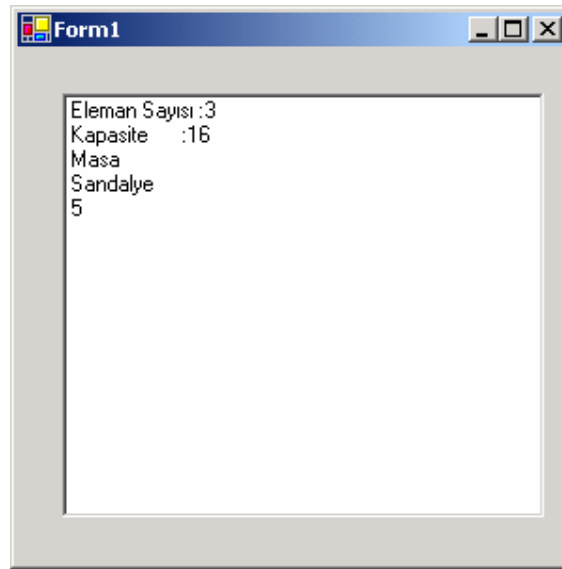
    GLOBAL.tempStr = "";
    theTree.inOrder(theTree.getRoot());
    MessageBox.Show("\nAğacın InOrder Dolaşılması    :
"+GLOBAL.tempStr);

    GLOBAL.tempStr = "";
    theTree.preOrder(theTree.getRoot());
    MessageBox.Show("\nAğacın PreOrder Dolaşılması    :
"+GLOBAL.tempStr);
}

```

```
GLOBAL.tempStr = "";
theTree.postOrder(theTree.getRoot());
MessageBox.Show("\nAğacın PostOrder Dolaşılması    :
"+GLOBAL.tempStr);
}
```

C# programlama dilinin diğer özellikleri kullanılarak daha esnek ve daha etkin ağaç yapıları oluşturmak mümkündür. Buradaki koda dikkat edilecek olursa, Java ve C# programlama dilleri arasında Ağaç kodlarında bir farklılık olmadığı göze çarpar.

C# : HAZIR VERİ YAPILARI ÖRNEĞİ (ArrayList)

```
public class Sayi
{
    public int sayi;
    public Sayi(int sayi)
    {
        this.sayi = sayi;
    }
}
....

private void Form1_Load(object sender, System.EventArgs e)
{
    ArrayList liste = new ArrayList();

    liste.Add("Masa");
    liste.Add("Sandalye");
    liste.Add(new Sayi(5));

    listBox1.Items.Add("Eleman Sayısı :"+liste.Count);
    listBox1.Items.Add("Kapasite :"+liste.Capacity);

    for(int i=0; i<liste.Count; ++i)
    {
        if(liste[i].GetType().ToString().CompareTo("System.String")==0)
            listBox1.Items.Add(liste[i]);
        else
            listBox1.Items.Add(((Sayi)liste[i]).sayi);
    }
}
```