Java 数据结构和算法

一、数组于简单排序

数组

数组（array）是相同类型变量的集合，可以使用共同的名字引用它。数组可被定义为任何类型，可以是一维或多维。数组中的一个特别要素是通过下标来访问它。数组提供了一种将有联系的信息分组的便利方法。

一维数组

一维数组（one-dimensional array ）实质上是相同类型变量列表。要创建一个数组，你必须首先定义数组变量所需的类型。通用的一维数组的声明格式是：

type var-name[ ];

获得一个数组需要 2 步。第一步，你必须定义变量所需的类型。第二步，你必须使用运算符 new 来为数组所要存储的数据分配内存，并把它们分配给数组变量。这样 Java 中的数组被动态地分配。如果动态分配的概念对你陌生，别担心，它将在本书的后面详细讨论。

数组的初始化（array initializer ）就是包括在花括号之内用逗号分开的表达式的列表。逗号分开了数组元素的值。Java 会自动地分配一个足够大的空间来保存你指定的初始化元素的个数，而不必使用运算符 new。

Java 严格地检查以保证你不会意外地去存储或引用在数组范围以外的值。Java 的运行系统会检查以确保所有的数组下标都在正确的范围以内（在这方面，

Java 与 C/C++ 从根本上不同，C/C++ 不提供运行边界检查）。

多维数组

在 Java 中，多维数组（multidimensional arrays ）实际上是数组的数组。你可能期望，这些数组形式上和行动上和一般的多维数组一样。然而，你将看到，有一些微妙的差别。定义多维数组变量要将每个维数放在它们各自的方括号中。例如，下面语句定义了一个名为 twoD 的二维数组变量。

```
int twoD[][] = new int[4][5];
```

简单排序

简单排序中包括了：冒泡排序、选择排序、插入排序；

1.冒泡排序的思想：

假设有 N 个数据需要排序，则从第 0 个数开始，依次比较第 0 和第 1 个数据，如果第 0 个大于第 1 个则两者交换，否则什么动作都不做，继续比较第 1 个第 2 个…，这样依次类推，直至所有数据都“冒泡”到数据顶上。

冒泡排序的的 java 代码：

```
Public void bubbleSort()
{
int in,out;
for(out=nElems-1;out>0;out--)
   for(in=0;in<out;in++)
   {
      If(a[in]>a[in+1])
         Swap(in,in+1);
}
}
```

算法的不变性：许多算法中，有些条件在算法执行过程中始终是不变的。这些条件被称 为算法的不变性，如果不变性不为真了，则标记出错了；

冒泡排序的效率 O（N*N），比较 N*N/2，交换 N*N/4；

2. 选择排序的思想：

假设有 N 条数据，则暂且标记第 0 个数据为 MIN（最小），使用 OUT 标记最左边未排序的数据，然后使用 IN 标记第 1 个数据，依次与 MIN 进行比较，如果比 MIN 小，则将该数据标记为 MIN，当第一轮比较完后，最终的 MIN 与 OUT 标记数据交换，依次类推；

选择排序的 java 代码：

```
Public void selectSort()
{
Int in,out,min;
For(out=0;out<nElems-1;out++)
{
   Min=out;
For(in=out+1;in<nElems;in++)
   If(a[in]<a[min])
   Min=in;
Swap(out,min);
}
}
```

选择排序的效率：O（N*N），比较 N*N/2，交换<N； 选择排序与冒泡排序比较，比较次数没有明显改变，但交换次数明显减少了很多；

3. 插入排序的思想：

插入排序是在部分数据有序的情况下，使用 OUT 标记第一个无序的数据，将其提取保存到一个中间变量 temp 中去，使用 IN 标记空位置，依次比较 temp 中的值与 IN-1 的值，如果 IN-值大于 temp 的值，则后移，直到遇到第一个比 temp 小的值，在其下一个位置插入；

插入排序的 java 代码：

```
Public void InsertionSort()
   {
      Int in,out;
      For(out=1;out<nElems;out++)
```

```
    {
        Long temp=a[out]
        In=out;
        While(in>0&& a[in-1]>temp)
        {
            A[in]=a[in-1];
            - -in;
}
A[in]=temp;
}
}
```

插入排序的效率：O（N*N），比较 N*N/4，复制 N*N/4；插入排序在随机数的情况下，比冒泡快一倍，比选择稍快；在基本有序的数组中，插入排序几乎只需要 O（N）；在逆序情况下，并不比冒泡快；

二、栈与队列

1、栈的定义

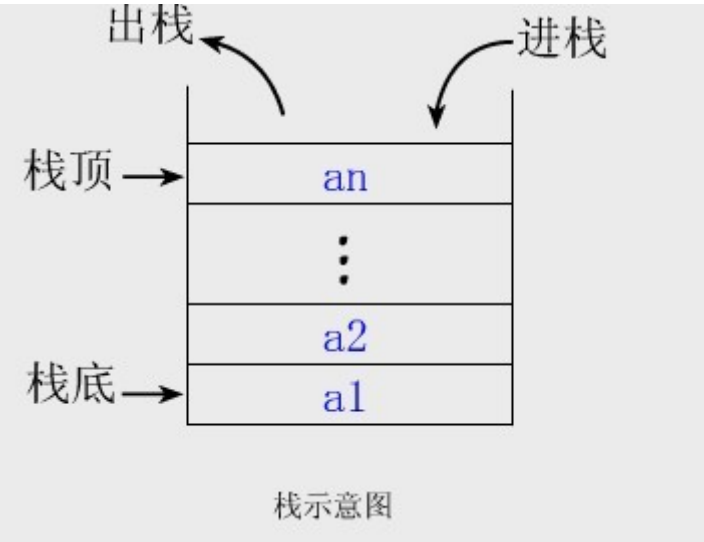
栈（Stack）是限制仅在表的一端进行插入和删除运算的线性表。

(1)通常称插入、删除的这一端为**栈顶**（Top），另一端称为**栈底**（Bottom）。

(2)当表中没有元素时称为**空栈**。

(3)栈为后进先出（Last In First Out）的线性表，简称为 LIFO 表。

栈的修改是按后进先出的原则进行。每次删除（**退栈**）的总是当前栈中"最新"的元素，即最后插入（**进栈**）的元素，而最先插入的是被放在栈的底部，要到最后才能删除。



栈示意图

【示例】元素是以 a1，a2，…，an 的顺序进栈，退栈的次序却是 an，an-1，…，a1。

2、栈的基本运算

（1）InitStack（S）

构造一个空栈 S。

（2）StackEmpty（S）

判栈空。若 S 为空栈，则返回 TRUE，否则返回 FALSE。

（3）StackFull（S）

判栈满。若 S 为满栈，则返回 TRUE，否则返回 FALSE。

注意： 该运算只适用于栈的顺序存储结构。

（4）Push（S，x）

进栈。若栈 S 不满，则将元素 x 插入 S 的栈顶。

（5）Pop（S）

退栈。若栈 S 非空，则将 S 的栈顶元素删去，并返回该元素。

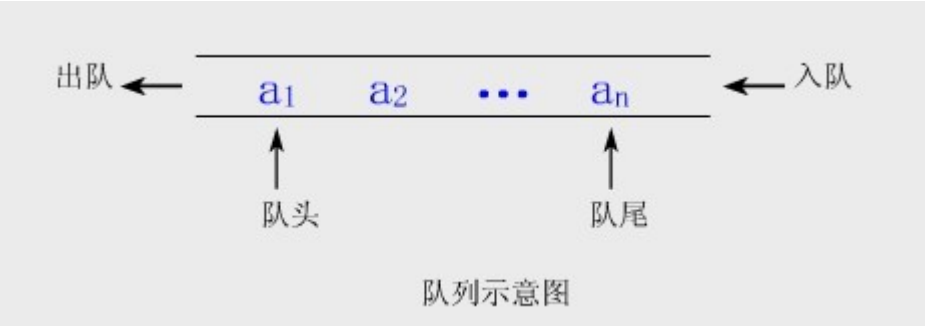
（6）StackTop（S）

取栈顶元素。若栈 S 非空，则返回栈顶元素，但不改变栈的状态。

队列的定义及基本运算

1、定义

队列（Queue）是只允许在一端进行插入，而在另一端进行删除的运算受限的线性表



队列示意图

（1）允许删除的一端称为**队头（Front）**。

（2）允许插入的一端称为**队尾（Rear）**。

（3）当队列中没有元素时称为**空队列**。

（4）队列亦称作先进先出（First In First Out）的线性表，简称为**FIFO表**。

队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（即不允许"加塞"），每次离开的成员总是队列头上的（不允许中途离队），即当前"最老的"成员离队。

【例】在队列中依次加入元素a_1，a_2，…，a_n之后，a_1是队头元素，a_n是队尾元素。退出队列的次序只能是a_1，a_2，…，a_n。

2、队列的基本逻辑运算

（1）InitQueue（Q）

置空队。构造一个空队列Q。

（2）QueueEmpty（Q）

判队空。若队列Q为空，则返回真值，否则返回假值。

（3） QueueFull（Q）

判队满。若队列Q为满，则返回真值，否则返回假值。

注意：此操作只适用于队列的顺序存储结构。

（4） EnQueue（Q，x）

若队列Q非满，则将元素x插入Q的队尾。此操作简称**入队**。

（5） DeQueue（Q）

若队列Q非空，则删去Q的队头元素，并返回该元素。此操作简称**出队**。

（6） QueueFront（Q）

若队列Q非空，则返回队头元素，但不改变队列Q的状态。

三、链表

1. 链结点

在链表中，每个数据项都被包含在‘点“中，一个点是某个类的对象，这个类可认叫做 LINK。因为一个链表中有许多类似的链结点，所以有必要用一个不同于链表的类来表达链结点。每个 LINK 对象中都包含一个对下一个点引用的字段（通常叫做 next）但是本身的对象中有一个字段指向对第一个链结点的引用

单链表

用一组地址任意的存储单元存放线性表中的数据元素。

以元素(数据元素的映象)

+ 指针(指示后继元素存储位置)

= 结点

(表示数据元素 或 数据元素的映象)

以“结点的序列”表示线性表

称作线性链表（单链表）——

单链表是一种顺序存取的结构，为找第 i 个数据元素，必须先找到第 i-1 个数据元素。

因此，查找第 i 个数据元素的基本操作为：移动指针，比较 j 和 i

1、链接存储方法

链接方式存储的线性表简称为链表（Linked List）。

链表的具体存储表示为：

① 用一组任意的存储单元来存放线性表的结点（这组存储单元既可以是连续的，也可以是不连续的）

② 链表中结点的逻辑次序和物理次序不一定相同。为了能正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其后继结点的地址（或位置）信息（称为指针（pointer）或链(link)）

注意：

链式存储是最常用的存储方式之一，它不仅可用来表示线性表，而且可用来表示各种非线性的数据结构。

2、链表的结点结构

data 域--存放结点值的数据域

next 域--存放结点的直接后继的地址（位置）的指针域（链域）

注意：

①链表通过每个结点的链域将线性表的 n 个结点按其逻辑顺序链接在一起的。

②每个结点只有一个链域的链表称为单链表（Single Linked List）。

【例】线性表（bat，cat，eat，fat，hat，jat，lat，mat）的单链表示如示意图

3、头指针 head 和终端结点指针域的表示

单链表中每个结点的存储地址是存放在其前趋结点 next 域中，而开始结点无前趋，故应设头指针 head 指向开始结点。

注意：

链表由头指针唯一确定，单链表可以用头指针的名字来命名。

【例】头指针名是 head 的链表可称为表 head。

终端结点无后继，故终端结点的指针域为空，即 NULL。

4、单链表的一般图示法

由于我们常常只注重结点间的逻辑顺序，不关心每个结点的实际位置，可以用箭头来表示链域中的指针，线性表（bat，cat，fat，hat，jat，lat，mat）的单链表就可以表示为下图形式。

5、单链表类型描述

```
typedef char DataType; //假设结点的数据域类型为字符
typedef struct node{ //结点类型定义
DataType data; //结点的数据域
struct node *next;//结点的指针域
}ListNode
typedef ListNode *LinkList;
ListNode *p;
LinkList head;
```

注意：

①*LinkList 和 ListNode 是不同名字的同一个指针类型（命名的不同是为了概念上更明确）

②*LinkList 类型的指针变量 head 表示它是单链表的头指针

③ListNode 类型的指针变量 p 表示它是指向某一结点的指针

6、指针变量和结点变量

	指针变量	结点变量
定义	在变量说明部分显式定义	在程序执行时，通过标准函数 malloc 生成
取值	非空时，存放某类型结点的地址	实际存放结点各域内容
操作方式	通过指针变量名访问	通过指针生成、访问和释放

①生成结点变量的标准函数

p=(ListNode *)malloc(sizeof(ListNode))；

//函数 malloc 分配一个类型为 ListNode 的结点变量的空间,并将其首地址放入指针变量 p 中

②释放结点变量空间的标准函数

free(p)；//释放 p 所指的结点变量空间

③结点分量的访问

利用结点变量的名字*p 访问结点分量

方法一：(*p).data 和(*p).next

方法二：p- > data 和 p- > next

④指针变量 p 和结点变量*p 的关系

指针变量 p 的值——结点地址

结点变量*p 的值——结点内容

(*p).data 的值——p 指针所指结点的 data 域的值

(*p).next 的值——*p 后继结点的地址

*((*p).next)——*p 后继结点

注意：

① 若指针变量 p 的值为空（NULL），则它不指向任何结点。此时，若通过*p 来访问结点就意味着访问一个不存在的变量，从而引起程序的错误。

② 有关指针类型的意义和说明方式的详细解释

可见，在链表中插入结点只需要修改指针。但同时，若要在第 i 个结点之前插入元素，修改的是第 i-1 个结点的指针。

因此，在单链表中第 i 个结点之前进行插入的基本操作为:
找到线性表中第 i-1 个结点，然后修改其指向后继的指针。

双端链表

双端链表与传统的链表非常相似，但是它有一个新增的特性：即对最后一个链结点的引用，就像对第一个链结点的引用一样。

对最后一个链结点的引用允许像在表头一样，在表尾直接插入一个链结点。当然，仍然可以在普通的单链表的表尾插入一个链结点，方法是遍历整个链表直到到达表尾，但是这种方法效率很低。

对最后一个链结点的引用允许像在表头一样，在表尾直接插入一个链结点。当然，仍然可以在普通的单链表的表尾插入一个链结点，方法是遍历整个链表直到到达表尾，但是这种方法效率很低。

像访问表头一样访问表尾的特性，使双端链表更适合于一些普通链表不方便操作的场合，队列的实现就是这样一个情况。

下面是一个双端链表的例子。

```
class Link3{
    public long dData;
    public Link3 next;
    //.....................
    public Link3(long d){
        dData=d;
    }
    //.....................
    public void displayLink(){
        System.out.print(dData+" ");
    }
}
////////////////////////////////////
class FirstLastList{
    private Link3 first;
    private Link3 last;
    //.....................
    public FirstLastList(){
        first=null;
        last=null;
    }
    //.....................
    public boolean isEmpty(){
        return first==null;
```

```
    }
    //.....................
    public void insertFirst(long dd){
        Link3 newLink=new Link3(dd);
        if(isEmpty())
            last=newLink;
        newLink.next=first;
        first=newLink;
    }
    //.....................
    public void insertLast(long dd){
        Link3 newLink=new Link3(dd);
        if(isEmpty())
            first=newLink;
        else
            last.next=newLink;
        last=newLink;
    }
    //.....................
    public long deleteFirst(){
        long temp=first.dData;
        if(first.next==null)
            last=null;
        first=first.next;
        return temp;
    }
    //.....................
    public void displayList(){
        System.out.print("List (first-->last): ");
        Link3 current=first;
        while(current!=null){
            current.displayLink();
            current=current.next;
        }
        System.out.println("");
    }
}
//////////////////////////////////
```

```
public class FirstLastApp {
    public static void main(String[] args) {
        FirstLastList theList=new FirstLastList();
        theList.insertFirst(22);
    theList.insertFirst(44);
        theList.insertFirst(66);
        theList.insertLast(11);
        theList.insertLast(33);
        theList.insertLast(55);
        theList.displayList();
        theList.deleteFirst();
        theList.deleteFirst();
        theList.displayList();
    }
}
```

为了简单起见，在这个程序中，把每个链结点中的数据字段个数从两个压缩到一个。这更容易显示链结点的内容。（记住，在一个正式的程序中，可能会有非常多的数据字段，或者对另外一个对象的引用，那个对象也包含很多数据字段。）

这个程序在表头和表尾各插入三个链点，显示插入后的链表。然后删除头两个链结点，再次显示。

注意在表头重复插入操作会颠倒链结点进入的顺序，而在表尾的重复插入则保持链结点进入的顺序。

双端链表类叫做 FirstLastList。它有两个项，first 和 last，一个指向链表中的第一个链结点，另一个指向最后一个链结点。如果链表中只有一个链结点，first 和 last 就都指向它，如果没有链结点，两者都为 Null 值。

这个类有一个新的方法 insertLast()，这个方法在表尾插入一个新的链结点。这个过程首先改变 last.next，使其指向新生成的链结点，然后改变 last，使其指向新的链结点。

插入和删除方法和普通链表的相应部分类似。然而，两个插入方法都要考虑一种特殊情况，即插入前链表是空的。如果 isEmpty()是真，那么 insertFirst()必须把 last 指向新的链结点，insertLast()也必须把 first 指向新的链结点。

如果用 insertFirst()方法实现在表头插入，first 就指向新的链结点，用 insertLast()方法实现在表尾插入，last 就指向新的链结点。如果链表只有一个链结点，那么多表头删除也是一种特殊情况：last 必须被赋值为 null 值。

不幸的是，用双端链表也不能有助于删除最后一个链结点，因为没有一个引用指向倒数第二个链结点。如果最后一个链结点被删除，倒数第二个链结点的 Next 字段应该变成 Null 值。为了方便的删除最后一个链结点，需要一个双向链表。（当然，也可以遍历整个链表找到最后一个链结点，但是那样做效率不是很高。）

有序链表

在有序链表中，数据是按照关键值有序排列的。有序链表的删除常常是只限于删除在链表头部的最小链结点。不过，有时也用 Find()方法和 Delete()方法在整个链表中搜索某一特定点。

一般，在大多数需要使用有序数组的场合也可以使用有序链表。有序链表优于有序数组的地方是插入的速度，另外链表可以扩展到全部有效的使用内存，而数组只能局限于一个固定的大小中。但是，有序链表实现起来比有序数组更困难一些。

后而将看到一个有序链表的应用：为数据排序。有序链表也可以用于实现优先级队列，尽管堆是更常用的实现方法。

在有序链表中插入一个数据项的 Java 代码

为了在一个有序链表中插入数据项，算法必须首先搜索链表，直到找到合适的位置：它恰好在第一个比它大的数据项的前面。

当算法找到了要插入的位置，用通常的方式插入数据项：把新链结点的 Next 字段指向下一个链结点，然后把前一个链结点的 Next 字段改为指向新的链结点。然而，需要考虑一些特殊情况：链结点有可以插在表头，或者插在表尾。看一下这段代码：

```
Public  void  insert(long  key){
    Link  newLink=new  Link(key);
    Link  previous=null;
    Link  current=first;
    While(current!=null  &&  key>current.dData){
        Previous=current;
        Current=current.next;
    }
    If(previous==null)
        First=newLink;
    Else
        Previous.next=newLink;
    newLink.next=current;
}
```

在链表上移动时，需要用一个 previous 引用，这样才能把前一个链结点的 Next 字段指向新的链结点。创建新链结点后，把 current 变量设为 first，准备搜索正确的插入点。这时也把 previous 设为 Null 值，这步操作很重要，因为后面要用这个 Null 值判断是否仍在表头。

While 循环和以前用来搜索插入点的代码类似，但是有一个附加的条件。如果当前检查的链结点的关键值不再小于待插入的链结点的关键值，则循环结束；这是最常见的情况，即新关键值插在链表中部的某个地方。

然而，如果 current 为 Null 值，while 循环也会停止。这种情况发生在表尾，或者链表为空时。

如果 current 在表头或者链表为空，previous 将为 Null 值；所以让 first 指向新的链结点。否则 current 处在链表中部或结尾，就使 previous 的 next 字段指向新的链结点。不论哪种情况、都让新链结点的 Next 字段指向 current。如果在表尾，current 为 Null 值，则新链结点的 Next 字段也本应该设为这个值（Null）。

下面是有序链表的程序

SortedList.java 程序实现了一个 SortedList 类，它拥有 insert()、remove()和 displayList()方法。只有 insert()方法与无序链表中的 insert()方法不同。

```
package 有序链表;
class Link{
    public long dData;
    public Link next;
    public Link(long dd){
        dData=dd;
    }
    //.....................
    public void displayLink(){
        System.out.print(dData+" ");
    }
}
////////////////////////////////////////
class SortedList{
    private Link first;
    //.....................
    public SortedList(){
        first=null;
    }
    //.....................
    public boolean isEmpty(){
        return (first==null);
    }
    //.....................
    public void insert(long key){
        Link newLink=new Link(key);
        Link previous=null;
        Link current=first;
        while(current!=null && key>current.dData){
            previous=current;
            current=current.next;
```

```
            }
            if(previous==null)
                first=newLink;
            else
                previous.next=newLink;
            newLink.next=current;
        }
        //.....................................
        public Link remove(){
            Link temp=first;
            first=first.next;
            return temp;
        }
        //.....................................
        public void displayList(){
            System.out.print("List (first-->last): ");
            Link current=first;
            while(current!=null){
                current.displayLink();
                current=current.next;
            }
            System.out.println("");
        }
}
public class SortedLinkApp {
        public static void main(String[] args) {
            SortedList theSortedList=new SortedList();
            theSortedList.insert(20);
            theSortedList.insert(40);
            theSortedList.displayList();
            theSortedList.insert(10);
            theSortedList.insert(30);
            theSortedList.insert(50);
            theSortedList.displayList();
            theSortedList.remove();
            theSortedList.displayList();

            System.exit(0);
```

```
    }
}
```

在 Main()方法中，插入值为 20 和 40 的两个链结点。然后再插入三个链结点，分别是 10、30 和 50。这三个值分别插在表头、表中和表尾。这说明 insert()方法正确地处理了特殊情况。最后删除了一个链结点，表现出删除操作总是从表头进行。每一步变化后，都显示整个链表。

双向链表

双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。一般我们都构造双向循环链表。

```
/* 线性表的双向链表存储结构 */
typedef struct DuLNode
{
ElemType data;
struct DuLNode *prior,*next;
}DuLNode,*DuLinkList;
/*带头结点的双向循环链表的基本操作(14 个) */
void InitList(DuLinkList *L)
{ /* 产生空的双向循环链表 L */
*L=(DuLinkList)malloc(sizeof(DuLNode));
if(*L)
(*L)->next=(*L)->prior=*L;
else
exit(OVERFLOW);
}
void DestroyList(DuLinkList *L)
{ /* 操作结果：销毁双向循环链表 L */
DuLinkList q,p=(*L)->next; /* p 指向第一个结点 */
while(p!=*L) /* p 没到表头 */
{
q=p->next;
free(p);
p=q;
}
free(*L);
*L=NULL;
}
void ClearList(DuLinkList L) /* 不改变 L */
```

```
{ /* 初始条件：L 已存在。操作结果：将 L 重置为空表 */
DuLinkList q,p=L->next; /* p 指向第一个结点 */
while(p!=L) /* p 没到表头 */
{
q=p->next;
free(p);
p=q;
}
L->next=L->prior=L; /* 头结点的两个指针域均指向自身 */
}
Status ListEmpty(DuLinkList L)
{ /* 初始条件：线性表 L 已存在。操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE */
if(L->next==L&&L->prior==L)
return TRUE;
else
return FALSE;
}
int ListLength(DuLinkList L)
{ /* 初始条件：L 已存在。操作结果：返回 L 中数据元素个数 */
int i=0;
DuLinkList p=L->next; /* p 指向第一个结点 */
while(p!=L) /* p 没到表头 */
{
i++;
p=p->next;
}
return i;
}
Status GetElem(DuLinkList L,int i,ElemType *e)
{ /* 当第 i 个元素存在时，其值赋给 e 并返回 OK，否则返回 ERROR */
int j=1; /* j 为计数器 */
DuLinkList p=L->next; /* p 指向第一个结点 */
while(p!=L&&jnext;
j++;
}
if(p==L||j>i) /* 第 i 个元素不存在 */
return ERROR;
```

```
    *e=p->data; /* 取第 i 个元素 */
    return OK;
    }
    int LocateElem(DuLinkList L,ElemType e,Status(*compare)(ElemType,Elem
Type))
    { /* 初始条件：L 已存在，compare()是数据元素判定函数 */
    /* 操作结果：返回 L 中第 1 个与 e 满足关系 compare()的数据元素的位序。 */
    /* 若这样的数据元素不存在，则返回值为 0 */
    int i=0;
    DuLinkList p=L->next; /* p 指向第 1 个元素 */
    while(p!=L)
    {
    i++;
    if(compare(p->data,e)) /* 找到这样的数据元素 */
    return i;
    p=p->next;
    }
    return 0;
    }
    Status PriorElem(DuLinkList L,ElemType cur_e,ElemType *pre_e)
    { /* 操作结果：若 cur_e 是 L 的数据元素，且不是第一个，则用 pre_e 返回它的
前驱， */
    /* 否则操作失败，pre_e 无定义 */
    DuLinkList p=L->next->next; /* p 指向第 2 个元素 */
    while(p!=L) /* p 没到表头 */
    {
    if(p->data==cur_e)
    {
    *pre_e=p->prior->data;
    return TRUE;
    }
    p=p->next;
    }
    return FALSE;
    }
    Status NextElem(DuLinkList L,ElemType cur_e,ElemType *next_e)
    { /* 操作结果：若 cur_e 是 L 的数据元素，且不是最后一个，则用 next_e 返回
它的后继， */
```

```
    /* 否则操作失败，next_e 无定义 */
    DuLinkList p=L->next->next; /* p 指向第 2 个元素 */
    while(p!=L) /* p 没到表头 */
    {
    if(p->prior->data==cur_e)
    {
    *next_e=p->data;
    return TRUE;
    }
    p=p->next;
    }
    return FALSE;
    }
    DuLinkList GetElemP(DuLinkList L,int i) /* 另加 */
    { /* 在双向链表 L 中返回第 i 个元素的地址。i 为 0，返回头结点的地址。若第 i 个元素不存在，*/
    /* 返回 NULL */
    int j;
    DuLinkList p=L; /* p 指向头结点 */
    if(i<0||i>ListLength(L)) /* i 值不合法 */
    return NULL;
    for(j=1;j<=i;j++)
    p=p->next;
    return p;
    }
    Status ListInsert(DuLinkList L,int i,ElemType e)
    { /* 在带头结点的双链循环线性表 L 中第 i 个位置之前插入元素 e，i 的合法值为 1≤i≤表长+1 */
    /* 改进算法 2.18，否则无法在第表长+1 个结点之前插入元素 */
    DuLinkList p,s;
    if(i<1||i>ListLength(L)+1) /* i 值不合法 */
    return ERROR;
    p=GetElemP(L,i-1); /* 在 L 中确定第 i 个元素前驱的位置指针 p */
    if(!p) /* p=NULL,即第 i 个元素的前驱不存在(设头结点为第 1 个元素的前驱) */
    return ERROR;
    s=(DuLinkList)malloc(sizeof(DuLNode));
    if(!s)
    return OVERFLOW;
```

```
s->data=e;
s->prior=p; /* 在第 i-1 个元素之后插入 */
s->next=p->next;
p->next->prior=s;
p->next=s;
return OK;
}
Status ListDelete(DuLinkList L,int i,ElemType *e)
{ /* 删除带头结点的双链循环线性表 L 的第 i 个元素，i 的合法值为 1≤i≤表长 */
DuLinkList p;
if(i<1) /* i 值不合法 */
return ERROR;
p=GetElemP(L,i); /* 在 L 中确定第 i 个元素的位置指针 p */
if(!p) /* p=NULL,即第 i 个元素不存在 */
return ERROR;
*e=p->data;
p->prior->next=p->next;
p->next->prior=p->prior;
free(p);
return OK;
}
void ListTraverse(DuLinkList L,void(*visit)(ElemType))
{ /* 由双链循环线性表 L 的头结点出发，正序对每个数据元素调用函数 visit() *
/
DuLinkList p=L->next; /* p 指向头结点 */
while(p!=L)
{
visit(p->data);
p=p->next;
}
printf("\n");
}
void ListTraverseBack(DuLinkList L,void(*visit)(ElemType))
{ /* 由双链循环线性表 L 的头结点出发，逆序对每个数据元素调用函数 visit()。
另加 */
DuLinkList p=L->prior; /* p 指向尾结点 */
while(p!=L)
{
```

```
visit(p->data);
p=p->prior;
}
printf("\n");
    }
```

迭代器

迭代器是一种对象，它能够用来遍历 STL 容器中的部分或全部元素，每个迭代器对象代表容器中的确定的地址。迭代器修改了常规指针的接口，所谓迭代器是一种概念上的抽象：那些行为上象迭代器的东西都可以叫做迭代器。然而迭代器有很多不同的能力，它可以把抽象容器和通用算法有机的统一起来。

迭代器提供一些基本操作符：*、++、==、！=、=。这些操作和 C/C++“操作 array 元素”时的指针接口一致。不同之处在于，迭代器是个所谓的 smart pointers，具有遍历复杂数据结构的能力。其下层运行机制取决于其所遍历的数据结构。因此，每一种容器型别都必须提供自己的迭代器。事实上每一种容器都将其迭代器以嵌套的方式定义于内部。因此各种迭代器的接口相同，型别却不同。这直接导出了泛型程序设计的概念：所有操作行为都使用相同接口，虽然它们的型别不同。

功能

迭代器使开发人员能够在类或结构中支持 foreach 迭代，而不必整个实现 IEnumerable 或者 IEnumerator 接口。只需提供一个迭代器，即可遍历类中的数据结构。当编译器检测到迭代器时，将自动生成 IEnumerable 接口或者 IEnumerator 接口的 Current，MoveNext 和 Dispose 方法。

特点

1.迭代器是可以返回相同类型值的有序序列的一段代码；

2.迭代器可用作方法、运算符或 get 访问器的代码体；

3.迭代器代码使用 yield return 语句依次返回每个元素，yield break 将终止迭代；

4.可以在类中实现多个迭代器，每个迭代器都必须像任何类成员一样有惟一的名称，并且可以在 foreach 语句中被客户端代码调用；

5.迭代器的返回类型必须为 IEnumerable 和 IEnumerator 中的任意一种；

6.迭代器是产生值的有序序列的一个语句块，不同于有一个 或多个 yield 语句存在的常规语句块；

7.迭代器不是一种成员，它只是实现函数成员的方式，理解这一点是很重要的，一个通过迭代器实现的成员，可以被其他可能或不可能通过迭代器实现的成员覆盖和重载；

8.迭代器块在 C#语法中不是独特的元素，它们在几个方面受到限制，并且主要作用在函数成员声明的语义上，它们在语法上只是语句块而已；

四、递归

递归是函数调用自身的一种特殊的编程技术，其应用主要在以下几个方面：

阶乘

在 java 当中的基本形式是：Public　void　mothed（int n）{//当满足某条件时：

Mothed（n-1）；

}

递归二分查找

Java 二分查找实现,欢迎大家提出交流意见.

```
/**
*名称:BinarySearch
*功能:实现了折半查找(二分查找)的递归和非递归算法.
*说明:
*     1、要求所查找的数组已有序,并且其中元素已实现 Comparable<T>接口,如
Integer、String 等.
*    2、非递归查找使用 search();,递归查找使用 searchRecursively();
*
*本程序仅供编程学习参考
*
*@author:    Winty
*@date:      2008-8-11
*@email:     [email]wintys@gmail.com[/email]
*/

class BinarySearch<T extends Comparable<T>> {
   private T[] data;//要排序的数据

   public BinarySearch(T[] data){
      this.data = data;
   }

   public int search(T key){
      int low;
      int high;
      int mid;
```

```
    if(data == null)
        return -1;

    low = 0;
    high = data.length - 1;

    while(low <= high){
        mid = (low + high) / 2;
        System.out.println("mid " + mid + " mid value:" + data[mid]);///

        if(key.compareTo(data[mid]) < 0){
            high = mid - 1;
        }else if(key.compareTo(data[mid]) > 0){
            low = mid + 1;
        }else if(key.compareTo(data[mid]) == 0){
            return mid;
        }
    }

    return -1;
}

private int doSearchRecursively(int low , int high , T key){
    int mid;
    int result;

    if(low <= high){
        mid = (low + high) / 2;
        result = key.compareTo(data[mid]);
        System.out.println("mid " + mid + " mid value:" + data[mid]);///

        if(result < 0){
            return doSearchRecursively(low , mid - 1 , key);
        }else if(result > 0){
            return doSearchRecursively(mid + 1 , high , key);
        }else if(result == 0){
            return mid;
        }
    }

    return -1;
}
```

```
    public int searchRecursively(T key){
        if(data ==null)return -1;

        return doSearchRecursively(0 , data.length - 1 , key);
    }

    public static void main(String[] args){
        Integer[] data = {1 ,4 ,5 ,8 ,15 ,33 ,48 ,77 ,96};
        BinarySearch<Integer> binSearch = new BinarySearch<Integer>(data);
        //System.out.println("Key index:" + binSearch.search(33) );

        System.out.println("Key index:" + binSearch.searchRecursively(3) );

        //String [] dataStr = {"A" ,"C" ,"F" ,"J" ,"L" ,"N" ,"T"};
        //BinarySearch<String> binSearch = new BinarySearch<String>(dataSt
r);
        //System.out.println("Key index:" + binSearch.search("A") );
    }
}
```

递归排序

其实在数组的全排序中完全可以使用更加易懂简便的写法——for 循环，但是通过 for 循环编写数组全排序需要有一个先决条件——知道数组全排序的个数，因为有 n 个数据全排序就需要写 n 个嵌套 for 循环。因此在写全排序时一般使用递归方法。这就是我的第一个关于递归排序的见解——递归排序可以无需已知排序数组的长度，即排序个数！

其二，不管是使用递归进行数组排序还是使用 for 循环进行数组的排序，它们都是本质都是使用枚举，因此可以得出这样一个结论:枚举可以确保找出每一种可能的排序规则！

其三，枚举是列出所有的方法并找出符合要求的算法，因此其算法效率一定比较的低，需要对其进行优化，才能达到较好的效果（递归的时候排除所有不可能的方案）

消除递归

消除递归的基本思路是用栈来模拟系统的函数调用从而消除递归。

基本上要做一下三件事：传递参数（包括返回地址）并转到函数入口；获得参数并处理参数；根据传入的返回地址返回

五、哈希表

一般的线性表、树中，记录在结构中的相对位置是随机的即和记录的关键字之间不存在确定的关系，在结构中查找记录时需进行一系列和关键字的比较。这一类查找方法建立在“比较”的基础上，查找的效率与比较次数密切相关。理想的情况是能直接找到需要的记录，因此必须在记录的存储位置和它的关键字之间建立一确定的对应关系 f，使每个关键字和结构中一个唯一的存储位置相对应。因而查找时，只需根据这个对应关系 f 找到给定值 K 的像 f(K)。若结构中存在关键字和 K 相等的记录，则必定在 f(K)的存储位置上，由此不需要进行比较便可直接取得所查记录。在此，称这个对应关系 f 为哈希函数，按这个思想建立的表为哈希表（又称为杂凑法或散列表）。

哈希表不可避免冲突(collision)现象：对不同的关键字可能得到同一哈希地址 即 key1≠key2，而 f(key1)=f(key2)。具有相同函数值的关键字对该哈希函数来说称为同义词(synonym)。 因此，在建造哈希表时不仅要设定一个好的哈希函数，而且要设定一种处理冲突的方法。可如下描述哈希表：根据设定的哈希函数 H(key)和所选中的处理冲突的方法，将一组关键字映象到一个有限的、地址连续的地址集(区间)上并以关键字在地址集中的“象”作为相应记录在表中的存储位置，这种表被称为哈希表。

六、高级排序

下周提供。

七、二叉树

1. 二叉树？

答：二叉树是一种树型结构，它的特点是每个结点至多只有二棵子树（即二叉树中不存在度大于 2 的结点),并且,二叉树的子树有左右之分,其次序不能任意颠倒。.二叉树的基本形态：(1)空二叉树；(2)只有一个根结点的二叉树；(3)右子树为空的二叉树；4)左子树为空的二叉树；(5)完全二叉树。

二叉树的存储结构 包括：

1.顺序存储结构

连续的存储单元存储二叉树的数据元素。例如图 6.4(b)的完全二叉树 ，可以向量 (一维数组) bt(1:6)作它的存储结构，将二叉树中编号为 i 的结点的数据元素存放在分量 bt[i]中 ,如图 6.6(a) 所示。但这种顺序存储结构仅适合于完全二叉树 ,而一般二叉树也按这种形式来存储 ,这将造成存 贮浪费。如和

图 6.4(c)的二叉树相应的存储结构图 6.6(b)所示，图中以 “0”表示不存在此结点 .

2. 链式存储结构

由二叉树的定义得知二叉树的结点由一个数据元素和分别指向左右子树的两个分支构成 ,则表 示二叉树的链表中的结点至少包含三个域 :数据域和左右指针域 ,如图 (b)所示。有时 ,为了便于找 到结点的双亲 ,则还可在结点结构中增加一个指向其双亲受的指针域，如图 6.7(c)所示。

遍历二叉树：

遍历二叉树 (traversing binary tree)的问题， 即如何按某条搜索路径巡访树中每个结点，使得每个结点均被访问一次，而且仅被访问一次。 其中常见的有三种情况：分别称之为先 (根)序遍历，中 (根)序遍历和后 (根)序遍历。

（1）前序遍历

前序遍历运算：即先访问根结点，再前序遍历左子树，最后再前序遍历右子树。前序遍历运算访问二叉树各结点是以根、左、右的顺序进行访问的

（2）中序遍历

中序遍历运算：即先中前序遍历左子树，然后再访问根结点，最后再中序遍历右子树。中序遍历运算访问二叉树各结点是以左、根、右的顺序进行访问的

（3）后序遍历

后序遍历运算：即先后序遍历左子树，然后再后序遍历右子树，最后访问根结点。后序遍历运算访问二叉树各结点是以左、右、根的顺序进行访问的

八、红—黑树

概念

红黑树是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它是在 1972 年由Rudolf Bayer发明的，他称之为″对称二叉B树″，它现代的名字是在 Leo J. Guibas 和 Robert Sedgewick 于 1978 年写的一篇论文中获得的。它是复杂的，但它的操作有着良好的最坏情况运行时间，并且在实践中是高效的: 它可以在O(log n)时间内做查找，插入和删除，这里的n 是树中元素的数目。

红黑树是一种很有意思的平衡检索树。它的统计性能要好于平衡二叉树(有些书籍根据作者姓名，Adelson-Velskii和Landis，将其称为AVL-树)，因此，红黑树在很多地方都有应用。在C++ STL中，很多部分(目前包括set, multiset, map, multimap)应用了红黑树的变体(SGI STL中的红黑树有一些变化，这些修改提供了更好的性能，以及对set操作的支持)。

性质

红黑树是每个节点都带有颜色属性的二叉查找树，颜色或红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求:

性质 1. 节点是红色或黑色。

性质 2. 根是黑色。

性质 3. 所有叶子都是黑色（叶子是NIL节点）。

性质 4. 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

性质 5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

这些约束强制了红黑树的关键性质: 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

要知道为什么这些特性确保了这个结果，注意到属性 4 导致了路径不能有两个毗连的红色节点就足够了。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据属性 5 所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。

一种红黑树的实现方法：

```
/**
* 红黑树的实现
**/
public class RBTreeTest<E extends Comparable<E>> {
    static enum Color {
        BLACK, RED
    }

    static class RBPrinter {
        public static void visitNode(RBNode node) {
            RBNode n = node;
            if (n != null)
                System.out.print(n.key + "("
                        + (n.color == Color.RED ? "RED" : "BLACK")
                        + "),");
        }
    }

    static class RBNode<E extends Comparable<E>> {
        E key;

        RBNode<E> left, right;

        RBNode<E> parent;

        Color color;
```

```
    RBNode(RBNode<E> p, E key, Color color) {
        this.key = key;
        this.color = color;
        this.parent = p;
        this.left = null;
        this.right = null;
    }
}

private RBNode<E> root;

public RBTreeTest() {
    root = null;
}

public boolean isEmpty() {
    return root == null;
}

public E findMax() {
    if (isEmpty())
        return null;
    RBNode<E> node = root;
    while ((node.right) != null) {
        node = node.right;
    }
    return node.key;
}

public E findMin() {
    if (isEmpty())
        return null;
    RBNode<E> node = root;
    while ((node.left) != null) {
        node = node.left;
    }
    return node.key;
}

public final boolean contains(E ele) {
    RBNode<E> tmp = root;
    int cmp = -1;
    while (tmp != null) {
        cmp = ele.compareTo(tmp.key);
        if (cmp < 0) {
```

```
            tmp = tmp.left;
        } else if (cmp > 0) {
            tmp = tmp.right;
        } else {
            return true;
        }
    }
    return false;
}

public final boolean delete(E ele) {
    RBNode<E> cur;
    int cmp;
    if (root == null)
        return false;
    cur = root;
    while (cur != null && (cmp = ele.compareTo(cur.key)) != 0) {
        if (cmp < 0)
            cur = cur.left;
        else
            cur = cur.right;
    }
    if (cur == null) {
        return false;
    }
    if ((cur.left) != null && (cur.right) != null) {
        RBNode<E> prev = cur.left;
        while ((prev.right) != null) {
            prev = prev.right;
        }
        cur.key = prev.key;
        cur = prev;
    }
    if ((cur.left) != null) {
        if (cur == root) {
            root = cur.left;
            root.color = Color.BLACK;
            return true;
        }
        if (cur.parent.left == cur) {
            cur.parent.left = cur.left;
            cur.left.parent = cur.parent;
        } else {
            cur.parent.right = cur.left;
```

```
                cur.left.parent = cur.parent;
            }
            if (cur.color == Color.BLACK) {
                cur.left.color = Color.BLACK;
            }
        } else if ((cur.right) != null) {
            if (cur == root) {
                root = cur.right;
                root.color = Color.BLACK;
                return true;
            }
            if (cur.parent.left == cur) {
                cur.parent.left = cur.right;
                cur.right.parent = cur.parent;
            } else {
                cur.parent.right = cur.right;
                cur.right.parent = cur.parent;
            }
            if (cur.color == Color.BLACK) {
                cur.right.color = Color.BLACK;
            }
        } else {
            if (cur == root) {
                root = null;
                return true;
            }
            RBNode<E> todo;
            if (cur.parent.left == cur) {
                todo = null;
                cur.parent.left = todo;
            } else {
                todo = null;
                cur.parent.right = todo;
            }
            if (cur.color == Color.BLACK) {
                fixupDoubleBlack(todo);
            }
        }
        return true;
    }

    private final void fixupDoubleBlack(RBNode<E> cur) {
        RBNode<E> sibling;
        RBNode<E> p;
```

```
while (cur != root) {
    p = cur.parent;
    if (p.left == cur) {
        sibling = p.right;
        if (sibling.color == Color.RED) {
            rotateLeft(p);
            p.color = Color.RED;
            sibling.color = Color.BLACK;
        } else {
            if (sibling.right.color == Color.RED) {
                rotateLeft(p);
                p.color = Color.BLACK;
                sibling.right.color = Color.BLACK;
                return;
            } else if (sibling.left.color == Color.RED) {
                rotateRight(sibling);
                sibling.color = Color.RED;
                sibling.parent.color = Color.BLACK;
            } else {
                sibling.color = Color.RED;
                if (p.color == Color.BLACK) {
                    cur = p;
                } else {
                    p.color = Color.BLACK;
                    return;
                }
            }
        }
    } else {
        sibling = p.left;
        if (sibling.color == Color.RED) {
            rotateRight(p);
            p.color = Color.RED;
            sibling.color = Color.BLACK;
        } else {
            if (sibling.left.color == Color.RED) {
                rotateRight(p);
                sibling.color = p.color;
                p.color = Color.BLACK;
                sibling.left.color = Color.BLACK;
                return;
            } else if (sibling.right.color == Color.RED) {
                rotateLeft(sibling);
                sibling.color = Color.RED;
```

```
                    sibling.parent.color = Color.BLACK;
                } else {
                    sibling.color = Color.RED;
                    if (p.color == Color.BLACK) {
                        cur = p;
                    } else {
                        p.color = Color.BLACK;
                        return;
                    }
                }
            }
        }
    }

    public final void insert(E ele) {
        if (root == null) { // 添加根节点
            root = new RBNode<E>(null, ele, Color.BLACK);
            return;
        } else { // 将该节点添加到合适的叶子节点的位置
            RBNode<E> parent = null;
            RBNode<E> cur = root;
            int cmp = -1;
            while (cur != null && (cmp = ele.compareTo(cur.key)) != 0)
{
                parent = cur;
                if (cmp < 0)
                    cur = cur.left;
                else
                    cur = cur.right;
            }
            if (cmp == 0) { // 不能添加相同的元素
                throw new IllegalArgumentException(
                        "can't insert duplicate key!");
            }
            cur = new RBNode<E>(parent, ele, Color.RED);
            if (cmp < 0) { // 添加为左孩子
                parent.left = cur;
            } else { // 添加为右孩子
                parent.right = cur;
            }
```

```
            insertFixup(cur); // 调整各个节点
        }
    }

    private final void insertFixup(RBNode<E> cur) {
        RBNode<E> p, g;
        while (cur != root && cur.color == Color.RED) {
            p = cur.parent;
            if (p.color == Color.BLACK) {
                return;
            }
            g = p.parent;
            if (p == g.left) { // p 是 g 的左子树
                RBNode<E> uncle = g.right;
                if (uncle != null && uncle.color == Color.RED) {
                    g.color = Color.RED;
                    uncle.color = p.color = Color.BLACK;
                    cur = g;
                } else {
                    if (cur == p.right) {
                        cur = rotateLeft(p);
                        cur = cur.left;
                    }
                    cur = rotateRight(g);
                    cur.color = Color.BLACK;
                    cur.left.color = cur.right.color = Color.RED;
                }
            } else { // p 是 g 的右子树
                RBNode<E> uncle = g.left;
                if (uncle != null && uncle.color == Color.RED) {
                    g.color = Color.RED;
                    uncle.color = p.color = Color.BLACK;
                    cur = g;
                } else {
                    if (cur == p.left) {
                        cur = rotateRight(p);
                        cur = cur.right;
                    }
                    cur = rotateLeft(g);
                    cur.color = Color.BLACK;
                    cur.left.color = cur.right.color = Color.RED;
                }
            }
        }
```

```
        root.color = Color.BLACK;
    }

    /*
     *      gr            gr            gr
     *     /             /              /
     *    p                             ch
     *   / \   =>    p ch    =>    / \          ch 绕着父节点 p 左旋（即将
父节点 p 和它的右孩子转换角色）
     * 1 ch       /       \          p     3
     *     / \     1   2    3       / \
     *   2    3                    1    2
     */
    private final RBNode<E> rotateLeft(RBNode<E> p) {
        RBNode<E> gr = p.parent;//grandfather
        RBNode<E> ch = p.right;    //children
        p.right = ch.left;
        if (ch.left != null) {
            ch.left.parent = p;
        }
        ch.left = p;
        p.parent = ch;

        if (gr != null) {    //p 不是 root 节点
            if (gr.left == p)    //如果 p 是 gr 的左孩子
                gr.left = ch;
            else                 //p 是 gr 的右孩子
                gr.right = ch;
        } else {             //p 是 root 节点
            root = ch;
        }
        ch.parent = gr;
        return ch;
    }

   /*
     *       gr            gr             gr
     *      /             /                /
     *     p                              ch
     *    / \   =>    ch p    =>    / \          ch 绕着父节点 p 右旋（即将
父节点 p 和它的左孩子转换角色）
     *   ch 3      /       \        1   p
     *   / \         1   2    3           / \
     * 1    2                         2   3
```

```
 */
private final RBNode<E> rotateRight(RBNode<E> p) {
    RBNode<E> gr = p.parent;
    RBNode<E> ch = p.left;
    p.left = ch.right;
    if (ch.right != null) {
        ch.right.parent = p;
    }
    ch.right = p;
    p.parent = ch;
    if (gr != null) {
        if (gr.left == p)
            gr.left = ch;
        else
            gr.right = ch;
    } else {
        root = ch;
    }
    ch.parent = gr;
    return ch;
}

public void inOrder(){
    inOrder(root);
}

private void inOrder(RBNode<E> cur) {
    if(cur != null){
        inOrder(cur.left);
        RBPrinter.visitNode(cur);
        inOrder(cur.right);
    }
}

public static void main(String[] args) {
    java.util.Random ran = new java.util.Random();
    RBTreeTest<Integer> rbt = new RBTreeTest<Integer>();
    int tmp = 0;
    for(int i=0;i<15;i++){
        tmp = ran.nextInt(1000);
        try{
            rbt.insert(tmp);
        } catch (IllegalArgumentException e){
            do{
```

```
                tmp = ran.nextInt(1000);
            }while(rbt.contains(tmp));
            rbt.insert(tmp);
        }
        System.out.println(tmp);
    }

    System.out.print("\nInorder begin:\n");
    rbt.inOrder();
    System.out.print("\nInorder end\n");

    rbt.delete(tmp);

    System.out.print("\nInorder begin:\n");
    rbt.inOrder();
    System.out.print("\nInorder end\n");
  }
}
```

九、堆

1.堆排序定义

n个关键字序列K_1，K_2，…，K_n称为堆，当且仅当该序列满足如下性质(简称为堆性质)：

(1) $k_i \le K_{2i}$且$k_i \le K_{2i+1}$ 或(2)$K_i \ge K_{2i}$且$k_i \ge K_{2i+1}$($1 \le i \le \lfloor n/2 \rfloor$)

若将此序列所存储的向量R[1..n]看做是一棵完全二叉树的存储结构，则堆实质上是满足如下性质的完全二叉树：树中任一非叶结点的关键字均不大于(或不小于)其左右孩子(若存在)结点的关键字。

2、大根堆和小根堆

根结点(亦称为堆顶)的关键字是堆里所有结点关键字中最小者的堆称为小根堆。

根结点(亦称为堆顶)的关键字是堆里所有结点关键字中最大者，称为大根堆。

注意：

①堆中任一子树亦是堆。

②以上讨论的堆实际上是二叉堆(Binary Heap)，类似地可定义k叉堆。

3、堆排序特点

堆排序(HeapSort)是一树形选择排序。

堆排序的特点是：在排序过程中，将R[l..n]看成是一棵完全二叉树的顺序存储结构，利用完全二叉树中双亲结点和孩子结点之间的内在关系【参见二叉树的顺序存储结构】，在当前无序区中选择关键字最大(或最小)的记录。

4、堆排序与直接插入排序的区别

直接选择排序中，为了从R[1..n]中选出关键字最小的记录，必须进行n-1 次比较，然后在R[2..n]中选出关键字最小的记录，又需要做n-2 次比较。事实上，后面的n-2 次比较中，有许多比较可能在前面的n-1 次比较中已经做过，但由于前一趟排序时未保留这些比较结果，所以后一趟排序时又重复执行了这些比较操作。

堆排序可通过树形结构保存部分比较结果，可减少比较次数。

5、堆排序

堆排序利用了大根堆(或小根堆)堆顶记录的关键字最大(或最小)这一特征，使得在当前无序区中选取最大(或最小)关键字的记录变得简单。

（1）用大根堆排序的基本思想

① 先将初始文件R[1..n]建成一个大根堆，此堆为初始的无序区

② 再将关键字最大的记录R[1](即堆顶)和无序区的最后一个记录R[n]交换，由此得到新的无序区R[1..n-1]和有序区R[n]，且满足R[1..n-1].keys≤R[n].key

③ 由于交换后新的根R[1]可能违反堆性质，故应将当前无序区R[1..n-1]调整为堆。然后再次将R[1..n-1]中关键字最大的记录R[1]和该区间的最后一个记录R[n-1]交换，由此得到新的无序区R[1..n-2]和有序区R[n-1..n]，且仍满足关系R[1..n-2].keys≤R[n-1..n].keys，同样要将R[1..n-2]调整为堆。

……

直到无序区只有一个元素为止。

（2）大根堆排序算法的基本操作：

① 初始化操作：将R[1..n]构造为初始堆；

② 每一趟排序的基本操作：将当前无序区的堆顶记录R[1]和该区间的最后一个记录交换，然后将新的无序区调整为堆(亦称重建堆)。

注意：

①只需做n-1 趟排序，选出较大的n-1 个关键字即可以使得文件递增有序。

②用小根堆排序与利用大根堆类似，只不过其排序结果是递减有序的。堆排序和直接选择排序相反：在任何时刻，堆排序中无序区总是在有序区之前，且有序区是在原向量的尾部由后往前逐步扩大至整个向量为止。

（3）堆排序的算法：

```
void HeapSort(SeqIAst R)
 { //对R[1..n]进行堆排序，不妨用R[0]做暂存单元
  int i;
  BuildHeap(R);  //将R[1-n]建成初始堆
  for(i=n;i>1; i--){ //对当前无序区R[1..i]进行堆排序，共做n-1 趟。
    R[0]=R[1]; R[1]=R[i];R[i]=R[0];  //将堆顶和堆中最后一个记录交换
     Heapify(R, 1, i-1);  //将R[1..i-1]重新调整为堆，仅有R[1]可能违反堆性质
   } //endfor
 } //HeapSort
```

（4） BuildHeap和Heapify函数的实现

因为构造初始堆必须使用到调整堆的操作，先讨论Heapify的实现。

① Heapify函数思想方法

每趟排序开始前R[l..i]是以R[1]为根的堆，在R[1]与R[i]交换后，新的无序区R[1..i-1]中只有R[1]的值发生了变化，故除R[1]可能违反堆性质外，其余任何结点为根的子树均是堆。因此，当被调整区间是R[low..high]时，只须调整以R[low]为根的树即可。

"筛选法"调整堆

R[low]的左、右子树(若存在)均已是堆，这两棵子树的根R[2low]和R[2low+1]分别是各自子树中关键字最大的结点。若R[low].key不小于这两个孩子结点的关键字，则R[low]未违反堆性质，以R[low]为根的树已是堆，无须调整；否则必须将R[low]和它的两个孩子结点中关键字较大者进行交换，即R[low]与R[large](R[large].key=max(R[2low].key，R[2low+1].key))交换。交换后又可能使结点R[large]违反堆性质，同样由于该结点的两棵子树(若存在)仍然是堆，故可重复上述的调整过程，对以R[large]为根的树进行调整。此过程直至当前被调整的结点已满足堆性质，或者该结点已是叶子为止。上述过程就象过筛子一样，把较小的关键字逐层筛下去，而将较大的关键字逐层选上来。因此，有人将此方法称为"筛选法"。

要将初始文件R[l..n]调整为一个大根堆，就必须将它所对应的完全二叉树中以每一结点为根的子树都调整为堆。

显然只有一个结点的树是堆，而在完全二叉树中，所有序号 $> \lfloor n/2 \rfloor$ 的结点都是叶子，因此以这些结点为根的子树均已是堆。这样，我们只需依次将以序号为 $\lfloor n/2 \rfloor$ ， $\lfloor n/2 \rfloor - 1$，…，1 的结点作为根的子树都调整为堆即可。

具体算法【参见教材】。

5、 算法分析

堆排序的时间，主要由建立初始堆和反复重建堆这两部分的时间开销构成，它们均是通过调用Heapify实现的。

堆排序的最坏时间复杂度为O(nlgn)。堆排序的平均性能较接近于最坏性能。

由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。

堆排序是就地排序，辅助空间为O(1)，

它是不稳定的排序方法。

十、带权图

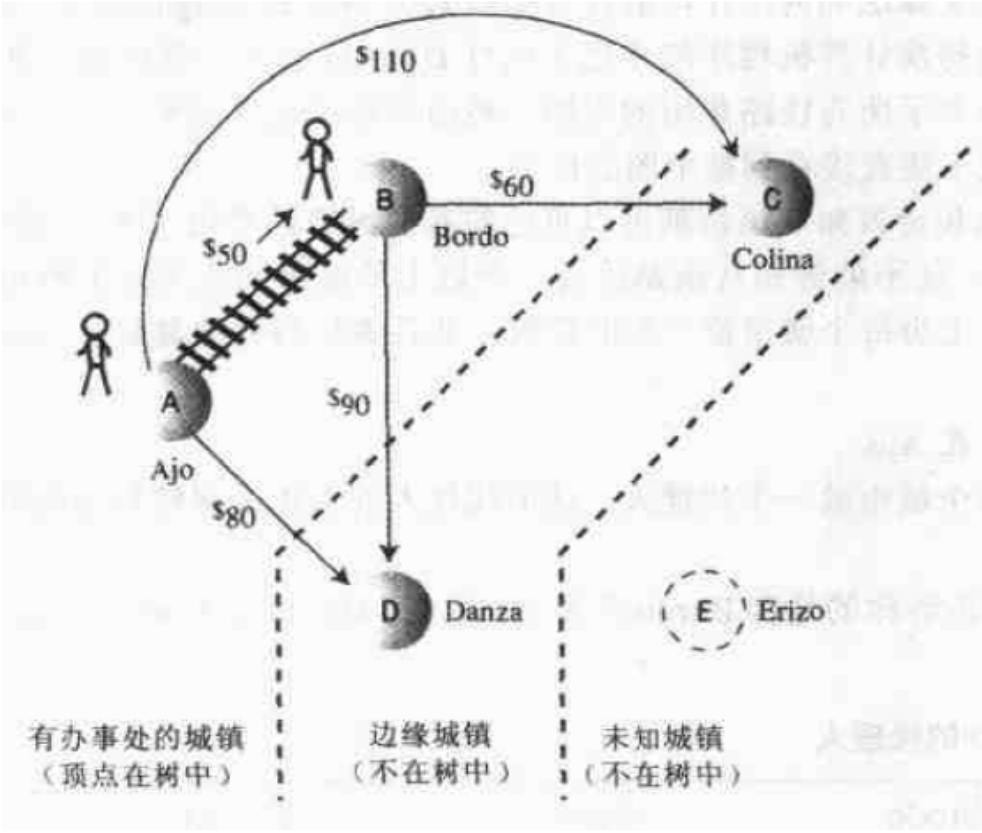
1. 带权图中，边带有一个数字，叫做权，它可能代表距离、耗费、时间或其他意义。
2. 带权图用来最常解决的问题是最短路径问题（pps）。
3. 带权图的最小生成树中有所有的顶点和连接它们的必要的边，且这些边的权值最小。

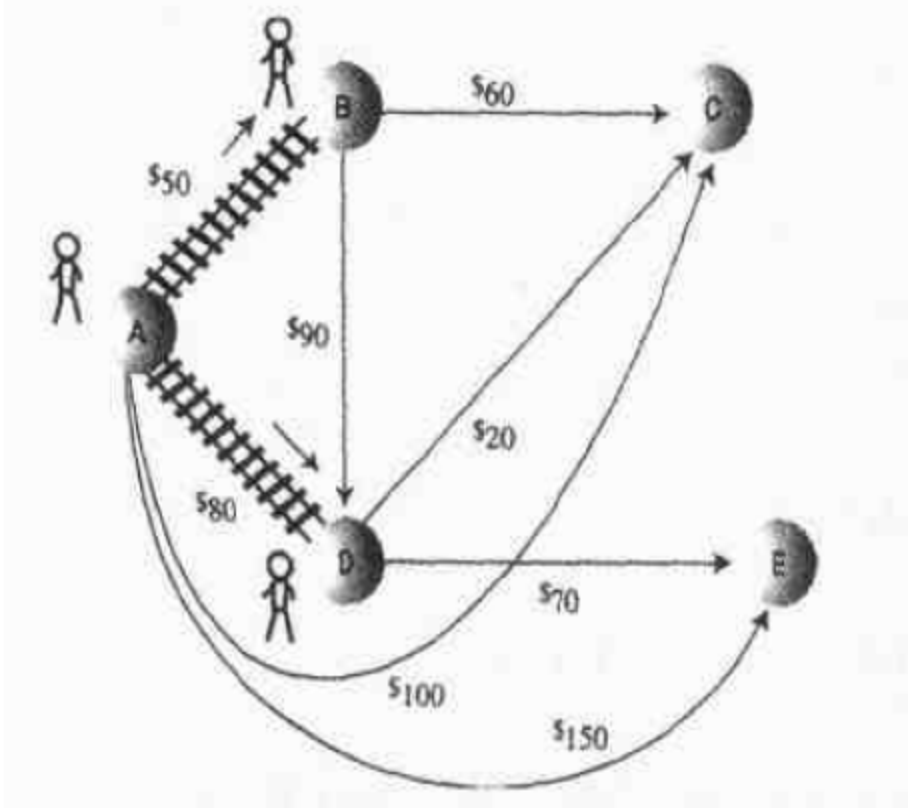
4. 优先级队列的算法可用于寻找带权图的最小生成树。
5. 解决无向带权图的最小生成树的方法
   1) 找到从最新的顶点到其他顶点的所有边，这些顶点不能在树的集合中，把这些边放如优先级队列。
   2) 找出权值最小的边，把它和它所到达的顶点放入树的集合中。
   3) 重复以上步骤，直到所有顶点都在树的集合中。
6. 带权图的最短路径问题可以用 Dijkstra 算法解决。这个算法基于图的邻接矩阵表示法，它不仅能找到任意两点间的最短路径，还可以找到某个指定点到其他所有顶点的最短路径。

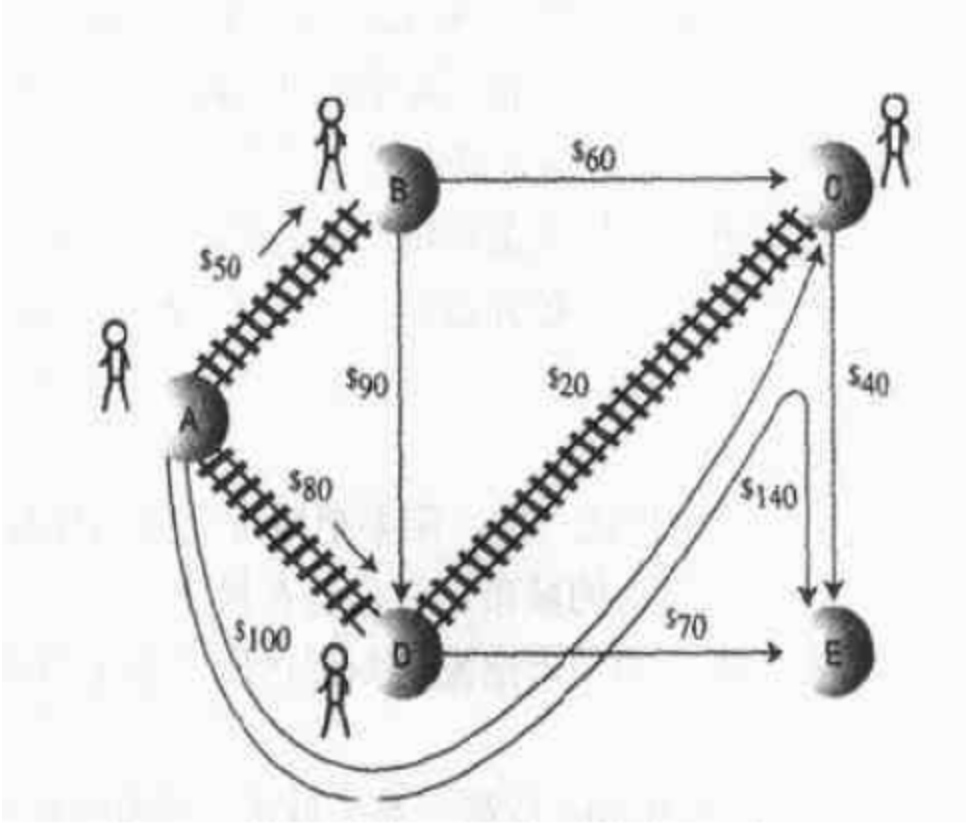
   Dijkstra 算法的思想：

   以解决寻找一条从一个城市到另一个城市的费用最低的路线为例（假设不能直接指导所有路线的费用，只能直接知道到邻接城市的费用）
   1) 每次派一个代理人到新的城市，用这个代理人提供的新信息修正和完善费用清单，在表中之保留从源点到某个城市现在一直的最低费用
   2) 不断向某个新城市派代理人，条件是从源点到那个城市的路线费用最低。

   如 图







7. 有时为了看出某条路线是否可能，需要创建一个连通表。在带权图中，用一张表给出任意两个顶点间的最低耗费，这对顶点可能通过几条边相连。这种问题叫做每一对顶点之间的最短路径问题。Warshall 算法（此算法在图章节中详述）能很快创建这样的连通表。

在带权图中类似的方法叫做 Floyd 算法。

Floyd 算法与 Warshall 算法的区别。在 Warshall 算法中当找到一个两段路径时，只是简单的在表中插入 1，在 Floyd 算法中，需要把两端的权值相加，并插入它们的和。

8. 关于各种图的算法的效率：

图的表示法有两种：邻接矩阵和邻接表

算法	时间级
邻接矩阵（所有）	O（V^2）
无权邻接表	O(V + E)
带权邻接表	O((E+V)logV)
Warshall 和 Floyd 算法	O（V^3）

其中 V 是顶点数量，E 是边的数量