廣州工商學院

Guangzhou College of Technology and Business

实验指导书

课程名称：数据结构与算法

课程代码：106010644

适用专业：软件工程

适用年级：2021 级

工学院

二0 二二 年 八 月 二十 日

目　录

实验 1:　顺序表的操作实验

一、实验名称和性质

所属课程	数据结构与算法
实验名称	顺序表的操作
实验学时	2
实验性质	√验证 □综合 □设计
必做/选做	√必做 □选做

二、实验目的

1．掌握线性表的顺序存储结构的表示和实现方法。

2．掌握顺序表基本操作的算法实现。

3．了解顺序表的应用。

三、实验内容

1．建立顺序表。

2．在顺序表上实现插入、删除和查找操作（验证性内容）。

3．删除有序顺序表中的重复元素（设计性内容）。

4．完成一个简单学生成绩管理系统的设计（应用性设计内容）。

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件：PC 软件：　Win2000 系统及以上;　VC 编译器

五、知识准备

前期要求熟练掌握了 C 语言的编程规则、方法和顺序表的基本操作算法。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）根据输入顺序表的长度 n 和各个数据元素值建立一个顺序表，并输出顺序表中各元素值，观察输入的内容与输出的内容是否一致。

（2）在顺序表的第 i(0≤i≤n)个元素之前插入一个值为 x 的元素，并输出插入后的顺序表中各元素值。

（3）删除顺序表中第 i(0≤i≤n-1)个元素，并输出删除后的顺序表中各元素值。

（4）在顺序表中查找值为 x 的数据元素初次出现的位置。如果查找成功，则返回该数据元素在顺序表中的位序号；如果查找失败，则返回-1。

2. 实验相关原理

线性表的顺序存储结构称为顺序表，线性表的顺序存储结构在线性表 C 接口的实现类中描述如下：

```
public class SqList implements IList{
    private Object[] listElem; // 线性表存储空间
```

```
    private int curLen; //线性表的当前长度
    ……
  }
```

【核心算法提示】

⑴顺序表插入操作的基本步骤：要在当前的顺序表中的第 i(0≤i≤n, n 为线性表的当前长度)个数据元素之前插入一个数据元素 x，首先要判断插入位置 i 是否合法， i 的合法值范围：1≤i≤n+1，若是合法位置，就再判断顺序表是否满，如果不满，则将第 i 个数据元素及其之后的所有数据元素都后移一个位置，此时第 i 个位置已经腾空，再将待插入的数据元素 x 插入到该位置上，最后将线性表的当前长度值增加 1，否则抛出异常。

⑵顺序表删除操作的基本步骤：要删除当前顺序表中的第 i(0≤i≤n-1)个数据元素，首先仍然要判断 i 的合法性，i 的合法范围是 0≤i≤n-1，若是合法位置，则将第 i 个数据元素之后的所有数据元素都前移一个位置，最后将线性表的当前长度减 1，否则抛出异常。

⑶顺序表查找操作的基本步骤：要在当前顺序表中查找一个给定值的数据元素，则可以采用顺序查找的方法，从顺序表中第 0 个数据元素开始依次将数据元素值与给定值进行比较，若相等则返回该数据元素在顺序表中的位置，如果所有数据元素都与 x 比较但都不相等，表明值为 x 的数据元素在顺序表中不存在，则返回-1 值。

【核心算法描述】

⑴在当前顺序表上的插入操作算法

```
void insert(int i, Object x) throws Exception {
      if (curLen == listElem.length)  // 判断顺序表是否已满
           throw new Exception("顺序表已满"); // 抛出异常
       if (i < 0 || i > curLen)   // i 不合法
           throw new Exception("插入位置不合法");// 抛出异常
       for (int j = curLen; j > i; j--)
           listElem[j] = listElem[j - 1];// 插入位置及其之后的所有数据元素后移一位
       listElem[i] = x;  // 插入 x
       curLen++;   // 表长加 1
  }
```

⑵ 在当前顺序表上的删除操作算法

```
void remove(int i) throws Exception {
  if (i < 0 || i > curLen - 1) // i 不合法
       throw new Exception("删除位置不合法");// 抛出异常
  for (int j = i; j < curLen - 1; j++)
         listElem[j] = listElem[j + 1];// 被删除元素及其之后的数据元素左移一个存储位置
  curLen--; // 表长减 1
 }
```

⑶ 在当前顺序表是的查找操作算法

```
int indexOf(Object x) {
   int j = 0;  // j 指示顺序表中待比较的数据元素，其初始值指示顺序表中第 0 个数据元素
    while (j < curLen && !listElem[j].equals(x))  //依次比较
         j++;
```

```
        if (j < curLen)   // 判断 j 的位置是否位于顺序表中
            return j;  // 返回值为 x 的数据元素在顺序表中的位置
        else
            return -1;    // 值为 x 的数据元素在顺序表中不存在
}
```

3．源程序代码参考

```
package sy;
import java.util.Scanner;
class SqList {
    private Object[] listElem; // 线性表存储空间
    private int curLen; // 当前长度
    public int getCurLen() {
        return curLen;
    }
    public void setCurLen(int curLen) {
        this.curLen = curLen;
    }
    public Object[] getListElem() {
        return listElem;
    }
    public void setListElem(Object[] listElem) {
        this.listElem = listElem;
    }
    // 顺序表的构造函数，构造一个存储空间容量为 maxSize 的空线性表
    public SqList(int maxSize) {
        curLen = 0; // 置顺序表的当前长度为 0
        listElem = new Object[maxSize];// 为顺序表分配 maxSize 个存储单元
    }
    // 在线性表的第 i 个数据元素之前插入一个值为 x 的数据元素。其中 i 取值范围为：0≤i≤curLen。
    public void insert(int i, Object x) throws Exception {
        if (curLen == listElem.length) // 判断顺序表是否已满
            throw new Exception("顺序表已满");// 输出异常
        if (i < 0 || i > curLen) // i 小于 0 或者大于表长
            throw new Exception("插入位置不合理");// 输出异常
        for (int j = curLen; j > i; j--)
            listElem[j] = listElem[j - 1];// 插入位置及之后的元素后移
        listElem[i] = x; // 插入 x
        curLen++;// 表长度增 1
    }
```

```
    // 将线性表中第 i 个数据元素删除。其中 i 取值范围为：0≤i≤curLen- 1,如果 i 值不在此范围则抛出异常
    public void remove(int i) throws Exception {
        if (i < 0 || i > curlew - 1) // i 小于 1 或者大于表长减 1
            throw new Exception("删除位置不合理");// 输出异常

        for (int j = i; j < curLen - 1; j++)
            listElem[j] = listElem[j + 1];// 被删除元素之后的元素左移
        curLen--; // 表长度减 1
    }
    //查找顺序表中值的 x 元素,若查找成功则返回元素在表中的位序(0～curLen-1),否则返回-1
    public int indexOf(Object x) {
        int j = 0;   // j 指示顺序表中待比较的数据元素，其初始值指示顺序表中第 0 个数据元素
        while (j < curLen && !listElem[j].equals(x))   //依次比较
            j++;
        if (j < curLen)     // 判断 j 的位置是否位于顺序表中
          return j;   // 返回值为 x 的数据元素在顺序表中的位置
        else
            return -1;      // 值为 x 的数据元素在顺序表中不存在
}
    // 输出顺序表中的数据元素
    public void display() {
        for (int j = 0; j < curLen; j++)
            System.out.print(listElem[j] + " ");
        System.out.println();// 换行
    }
}
//测试类
public class SY1_SqList{
    public static void main(String[] args) throws Exception{
        SqList L=new SqList(20);   //构造一个存储容量为 0 的空顺序表
        Scanner sc=new Scanner(System.in);
        System.out.println("请输入顺序表的长度:");
        int n=sc.nextInt();
        System.out.println("请输入顺序表中的各个数据元素:");
        for(int i=0;i<n;i++)
            L.insert(i,sc.nextInt());
        System.out.println("请输入待插入的位置 i(0～curLen):");
```

```
            int i=sc.nextInt();
            System.out.println("请输入待插入的数据值 x:");
            int x=sc.nextInt();
            L.insert(i, x);
            System.out.println("插入后的顺序表为:");
            L.display();
            System.out.println("请输入待删除元素的位置(0～curLen-1):");
            i=sc.nextInt();
            L.remove(i);
            System.out.println("删除后的顺序表为:");
            L.display();
            System.out.println("请输入待查找的数据元素:");
            x=sc.nextInt();
            int order=L.indexOf(x);
            if (order==-1)
                System.out.println("此顺序表中不包含值为"+x+"的数据元素!");
            else
                 System.out.println("值为"+x+"元素在顺序表中的第"+order+"个位置上");
        }
}
```

4．运行结果参考如图 1-1 所示：

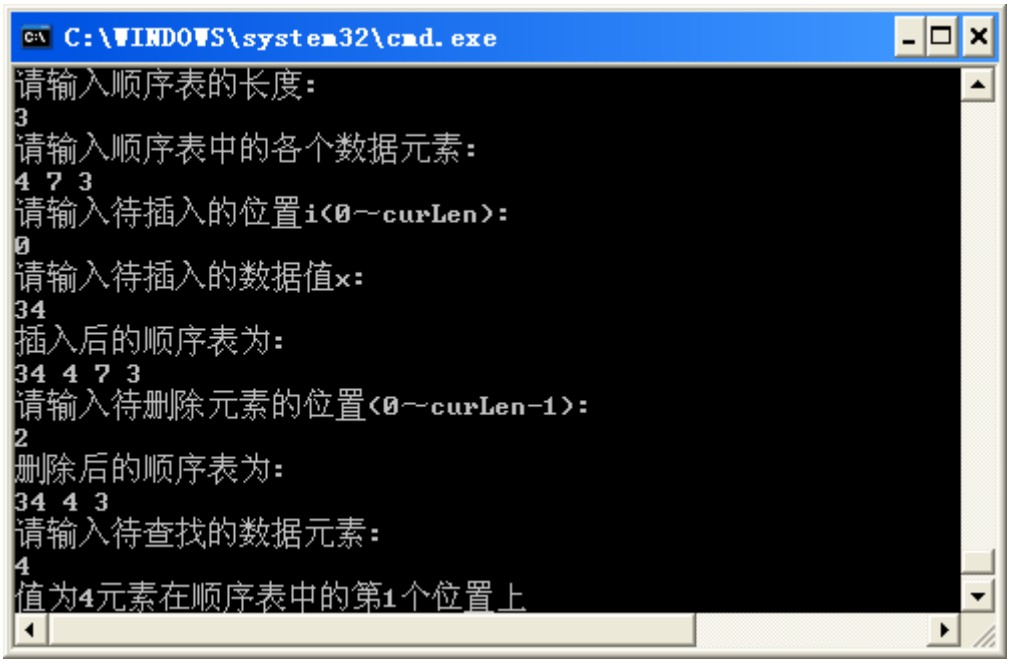


图 1-1：验证性实验运行结果

备注：以下设计性和应用性实验内容学生可根据自己的掌握程度或兴趣自行选择其一或其二完成。

七、设计性实验

编程实现删除有序顺序表中的所有重复元素，即使有序顺序表中相同的元素只保留一个。

1. 实验要求

⑴ 根据输入的 n 个非递减的有序数据建立一个有序顺序表，并输出有序顺序表中各元素值。

⑵ 删除有序顺序表中所有的重复元素，并显示删除后的有序顺序表中各元素值。

2. 核心算法提示

要在有序顺序表中删除重复的元素，首先就要抓住有序顺序表的特性：**重复的元素总是在相邻的位置上**，如：12，15，15，15，35，56，56，78。则删除重复元素后所得的有序表为：12，15，35，56，78。下面给出大致的操作步骤：从第 0 个元素开始，依次将它与后面相邻的元素进行比较，如果相等则将前面那个相等的元素从顺序表中删除；如果不相等，则继续往下比较，如此重复，直到最后一个元素为止。

3. 核心算法描述

```
// 删除有序顺序表 L 中的所有重复元素，即使得有序顺序表中相同的元素只保留一个
public static void remove_repeat(SqList L)  {
        int i=0;
        while(i<L.getCurLen()-1)
            if (L.getListElem()[i].equals(L.getListElem()[i+1])){
                                        //如果第i个及第i+1个相邻元素值相等
                for (int j=i+1;j<L.getCurLen();j++)
                                        //将第i+1个元素及其之后的所有元素前移一个位地置
                        L.getListElem()[j-1]=L.getListElem()[j];
                 L.setCurLen(L.getCurLen()-1);          //有序顺序表的表长减1
            }
           else
              i++;
    }
```

八、应用性设计实验

编程实现一个简单学生成绩管理系统的设计。

实验要求

此系统的功能包括：

① 查询：按特定的条件查找学生

② 修改：按学号对某个学生的某门课程成绩进行修改

③ 插入：增加新学生的信息

④ 删除：按学号删除已退学的学生的信息。

学生成绩表的数据如下：

学号	姓名	性别	大学英语	高等数学
2008001	Alan	F	93	88

2008002	Danie	M	75	69
2008003	Helen	M	56	77
2008004	Bill	F	87	90
2008006	Peter	M	79	86
2008006	Amy	F	68	75

要求采用顺序存储结构来实现对上述成绩表的相关操作。

实验 2： 链表的操作实验

一、实验名称和性质

所属课程	数据结构与算法
实验名称	链表的操作
实验学时	2
实验性质	√验证 □综合 √设计
必做/选做	√必做 □选做

二、实验目的

1．掌握线性表的链式存储结构的表示和实现方法。

2．掌握链表基本操作的算法实现。

三、实验内容

1．建立单链表，并在单链表上实现插入、删除和查找操作（验证性内容）。

2．建立双向链表，并在双向链表上实现插入、删除和查找操作（设计性内容）。

3．计算已知一个单链表中数据域值为一个指定值 x 的结点个数（应用性设计内容）。

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件： PC 软件： Win2000 系统及以上; VC 编译器

五、知识准备

前期要求熟练掌握了 Java 语言的编程规则、方法和单链表和双向链表的基本操作算法。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）根据输入的一系列整数，以 0 标志结束，用头插法建立单链表，并输出单链表中各元素值，观察输入的内容与输出的内容是否一致。

（2）在单链表的第 i 个元素之前插入一个值为 x 的元素，并输出插入后的单链表中各元素值。

（3）删除单链表中第 i 个元素，并输出删除后的单链表中各元素值。

（4）在单链表中查找第 i 个元素，如果查找成功，则显示该元素的值，否则显示该元素不存在。

2. 实验相关原理：

线性表的链式储结构是用一组任意的存储单元依次存放线性表中的元素，这组存储单元可以是连续的，也可以是不连续的。为反映出各元素在线性表中的前后逻辑关系，对每个数据元素来说，除了存储其本身数据值之外，还需增加一个或多个指针域，每个指针域的值称为指针，又称作链，它用来指示它在线性表中的前驱或后继的存储地址。这两个部分的的一起组成一个数据元素的存储映像，称为结点，若干个结点链接成链表。如果一个结点中只含一个指针

的链表，则称单链表。单链表中结点类描述如下：

```
class Node {
    private Object data; // 存放结点值
    private Node next; // 后继结点的引用
        // 无参数时的构造函数
    public Node() {
        this(null, null);
    }
    //带一个参数时的构造函数
public Node(Object data) {
        this(data, null);
    }
   //带两个参数时的构造函数
    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
     ……
}
```

【核心算法提示】

⑴ 链表建立操作的基本步骤：链表是一个动态的结构，它不需要予分配空间，因此建立链表的过程是一个结点“逐个插入” 的过程。先建立一个只含头结点的空单链表，然后依次生成新结点，再不断地将其插入到链表的头部或尾部，分别称其为“头插法”和“尾插法”。

⑵ 链表查找操作的基本步骤：因链表是一种"顺序存取"的结构，则要在带头结点的链表中查找到第 i 个 元素，必须从头结点开始沿着后继指针依次"点数"，直到点到第 i 个结点为止,如果查找成功，则用 e 返回第 i 个元素值。头结点可看成是第 0 个结点。

⑶ 链表插入操作的基本步骤：先确定要插入的位置，如果插入位置合法，则再生成新的结点，最后通过修改链将新结点插入到指定的位置上。

⑷ 链表删除操作的基本步骤：先确定要删除的结点位置，如果位置合法，则再通过修改链使被删结点从链表中“卸下”，最后释放被删结点的空间。

【核心算法描述】

⑴ 用头插法创建带头结点的单链表操作算法

```
void creat() throws Exception {///*输入一系列整数，以 0 标志结束，将这些整数作为//data 域并
                                用头插法建立一个带头结点的单链表
        Scanner sc = new Scanner(System.in);// 构造用于输入的对象
        for (int x=sc.nextInt(); x!=0; x=sc.nextInt())// 输入 n 个元素的值
                insert(0, x);// 生成新结点,插入到表头
  }
```

⑵ 在当前带头结点的单链表上的查找操作算法

```
Object get(int i) throws Exception {
    Node p = head.getNext();// 初始化，p 指向首结点，j 为计数器
    int j = 0;
```

```
    while (p != null && j < i) {// 从首结点开始向后查找，直到 p 指向第 i 个结点或 p 为空
        p = p.getNext();// 指向后继结点
        ++j;// 计数器的值增 1
    }
    if (j > i || p == null) { // i 小于 0 或者大于表长减 1
        throw new Exception("第" + i + "个元素不存在");// 抛出异常
    }
    return p.getData();  // 返回结点 p 的数据域的值
}
```

(3) 在当前带头结点的单链表上的插入操作算法

```
 void insert(int i, Object x) throws Exception {
        Node p = head;// 初始化 p 为头结点,j 为计数器
        int j = -1;
        while (p != null && j < i - 1) {// 寻找 i 个结点的前驱
            p = p.getNext();
            ++j;// 计数器的值增 1
        }
        if (j > i - 1 || p == null) // i 不合法
            throw new Exception("插入位置不合理");// 输出异常

        Node s = new Node(x); // 生成新结点
        s.setNext(p.getNext());// 插入单链表中
        p.setNext(s);
    }
```

(4) 在当前带头结点的单链表上的删除操作算法

```
 void remove(int i) throws Exception {
        Node p = head;// 初始化 p 为头结点,j 为计数器
        int j = -1;
        while (p.getNext() != null && j < i - 1) {// 寻找 i 个结点的前驱
            p = p.getNext();
            ++j;// 计数器的值增 1
        }
        if (j > i - 1 || p.getNext() == null) { // i 小于 0 或者大于表长减 1
            throw new Exception("删除位置不合理");// 输出异常
        }
        p.setNext(p.getNext().getNext());// 删除结点
  }
```

3．源程序代码参考

```
package sy;
import java.util.Scanner;
//单链表的结点类
class Node {
    private Object data; // 存放结点值
```

```
    private Node next; // 后继结点的引用
    public Node() { // 无参数时的构造函数
        this(null, null);
    }
    public Node(Object data) { // 构造值为 data 的结点
        this(data, null);
    }
    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Object getData() {
        return data;
    }
    public void setData(Object data) {
        this.data = data;
    }
    public Node getNext() {
        return next;
    }
    public void setNext(Node next) {
        this.next = next;
    }
}
//实现链表的基本操作类
class LinkList {
    Node head=new Node();//生成一个带头结点的空链表
    // 根据输入的一系列整数，以 0 标志结束，用头插法建立单链表
    public void creat() throws Exception {
        Scanner sc = new Scanner(System.in); // 构造用于输入的对象
        for (int x=sc.nextInt(); x!=0; x=sc.nextInt()) // 输入若干个数据元素的值(以 0 结束)
            insert(0, x);// 生成新结点,插入到表头
    }
    //返回带头结点的单链表中第 i 个结点的数据域的值。其中：0≤i≤curLen-1
    public Object get(int i) throws Exception {
        Node p = head.getNext();// 初始化，p 指向首结点，j 为计数器
        int j = 0;
        while (p != null && j < i) {// 从首结点开始向后查找，直到 p 指向第 i 个结点或 p 为空
```

```
            p = p.getNext();// 指向后继结点
            ++j;// 计数器的值增 1
        }
        if (j > i || p == null) { // i 小于 0 或者大于表长减 1
            throw new Exception("第" + i + "个元素不存在");// 抛出异常
        }
        return p.getData();   // 返回结点 p 的数据域的值
    }
   //在带头结点的单链表中的第 i 个数据元素之前插入一个值为 x 的元素
    public void insert(int i, Object x) throws Exception {
        Node p = head;// 初始化 p 为头结点,j 为计数器
        int j = -1;
        while (p != null && j < i - 1) {// 寻找 i 个结点的前驱
            p = p.getNext();
            ++j;// 计数器的值增 1
        }
        if (j > i - 1 || p == null) // i 不合法
            throw new Exception("插入位置不合理");// 输出异常

        Node s = new Node(x); // 生成新结点
        s.setNext(p.getNext());// 插入单链表中
        p.setNext(s);
    }
    // 删除带头结点的第 i 个数据元素。其中 i 取值范围为：0≤i≤length()- 1,如果 i 值不在此范围则抛出异常
    public void remove(int i) throws Exception {
        Node p = head;// 初始化 p 为头结点,j 为计数器
        int j = -1;
        while (p.getNext() != null && j < i - 1) {// 寻找 i 个结点的前驱
            p = p.getNext();
            ++j;// 计数器的值增 1
        }
        if (j > i - 1 || p.getNext() == null) { // i 小于 0 或者大于表长减 1
            throw new Exception("删除位置不合理");// 输出异常
        }
        p.setNext(p.getNext().getNext());// 删除结点
    }
    // 输出线性表中的数据元素
```

```
    public void display() {
        Node p = head.getNext();// 取出带头结点的单链表中的首结点元素
        while (p != null) {
            System.out.print(p.getData() + " ");// 输出数据元素的值
            p = p.getNext();// 取下一个结点
        }
        System.out.println();// 换行
    }
}
//测试类
public class SY2_LinkList{
    public static void main(String[] args) throws Exception{
        Scanner sc=new Scanner(System.in);
        LinkList L=new LinkList();
        System.out.println("请输入链表中的各个数据元素(0 为结束):");
        L.creat();
        System.out.println("建立的单链表为:");
        L.display();
        System.out.println("请输入待插入的位置 i(0～curLen):");
        int i=sc.nextInt();
        System.out.println("请输入待插入的数据值 x:");
        int x= sc.nextInt();
        L.insert(i, x);
        System.out.println("插入后的链表为:");
        L.display();
        System.out.println("请输入待删除元素的位置(0～curLen-1):");
        i=sc.nextInt();
        L.remove(i);
        System.out.println("删除后的链表为:");
        L.display();
        System.out.println("请输入待查找的数据元素的位序号(0～curLen-1):");
        i=sc.nextInt();
        Object o=L.get(i);
        System.out.println("此单链表中第"+i+"个结点的数据元素值为"+o);

    }
}
```

4．运行结果参考如图 2-1 所示：

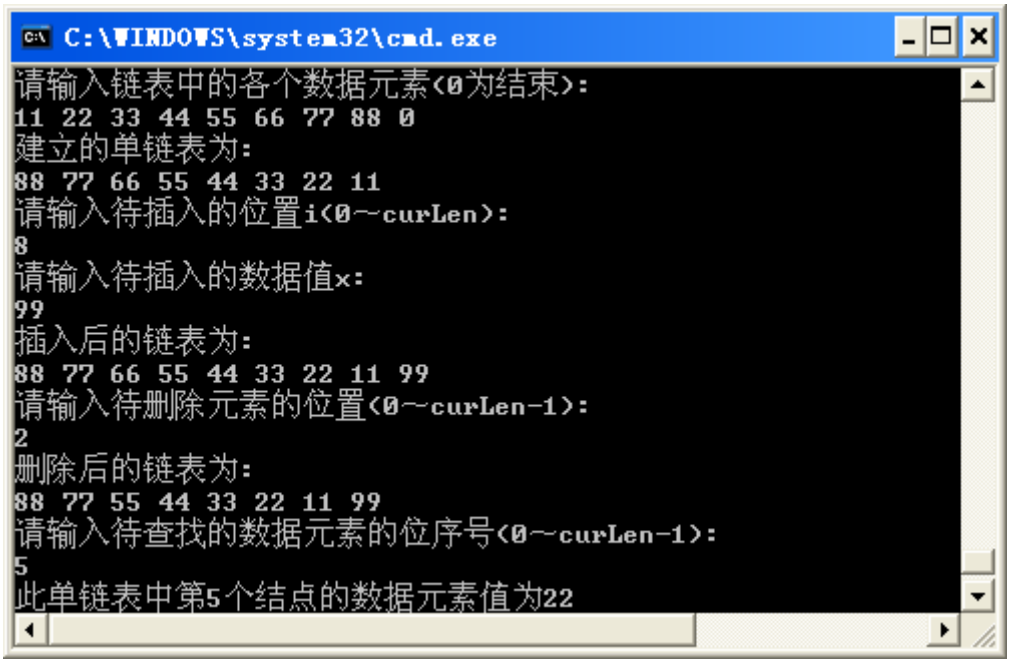


图 2-1：验证性实验运行结果

备注：以下设计性和应用性实验内容学生可根据自己的掌握程度或兴趣自行选择其一或其二完成。

七、设计性实验

编程实现在双向循环链表上的插入和删除操作

1． 实验要求

（1）输入链表的长度和各元素的值，用尾插法建立双向循环链表，并输出链表中各元素值，观察输入的内容与输出的内容是否一致。

（2）在双向循环链表的第 i 个元素之前插入一个值为 x 的元素，并输出插入后的链表中各元素值。

（3）删除双向循环链表中第 i 个元素，并输出删除后的链表中各元素值。

（4）在双向循环链表中查找值为 x 元素，如果查找成功，则显示该元素在链表中的位序号，否则显示该元素不存在。

2．核心算法提示

双向循环链表中的结点类描述如下：

```
class DuLNode {
    private Object data;// 存放结点值
    private DuLNode prior; // 前驱结点的引用
    private DuLNode next; // 后继结点的引用
    ……
}
```

双向循环链表类描述如下：

```
class DuCircleLinkList {
     private DuLNode head;// 双向循环链表的头结点
```

```
    // 构造函数:构造一个只含头结点的空双向循环链表
    public DuCircleLinkList() {
        head = new DuLNode(); // 初始化头结点
        head.setPrior(head);// 初始化头结点的前驱和后继
        head.setNext(head);
    }
    ……
}
```

不论是建立双向循环链表还是在双向循环链表中进行插入、删除和查找操作，其操作方法和步骤都跟单链表类似。只不过要注意两点：

（1）凡是在操作中遇到有修改链的地方，都要进行前驱和后继两个指针的修改。

（2）单链表操作算法中的判断条件：p= =null 或 p!=null ,在循环链表的操作算法中则需改为：p= =head 或 p!= head,其中 head 为链表的头指针。

3．核心算法描述

⑴ 在当前带头结点的双向循环链表上的插入操作的算法

```
void insert(int i, Object x) throws Exception {
    DuLNode p = head.getNext();// 初始化,p 指向首结点,j 为计数器
    int j = 0;
    while (!p.equals(head) && j < i) {// 确定插入位置 i
        p = p.getNext();// 指向后继结点
        ++j;// 计数器的值增 1
    }
    if (j !=i ) // i 小于 0 或者大于表长
        throw new Exception("插入位置不合理");// 输出异常
    DuLNode s = new DuLNode(x);// 生成新结点 s
    p.getPrior().setNext(s);  //将结点 s 插入到 p 结点的前面
    s.setPrior(p.getPrior());
    s.setNext(p);
    p.setPrior(s);
}
```

⑵ 在当前带头结点的双向循环链表上的删除操作算法

```
void remove(int i) throws Exception {
    DuLNode p = head.getNext();// 初始化,p 指向首节点结点,j 为计数器
    int j = 0;
    while (!p.equals(head) && j < i) {// 确定删除位置 i
        p = p.getNext();// 指向后继结点
        ++j;// 计数器的值增 1
    }
```

```
    if (j != i||p.equals(head)) // i 小于 0 或者大于表长减 1
        throw new Exception("删除位置不合理");// 输出异常
    p.getPrior().setNext(p.getNext());
    p.getNext().setPrior(p.getPrior());
}
```

(3) 在带头结点的双向循环链表上的查找操作算法

```
int indexOf(Object x) {
    DuLNode p = head.getNext();// 初始化,p 指向首结点,j 为计数器
    int j = 0;
    while (!p.equals(head) && !p.getData().equals(x)) {// 从链表中的首结点元素开始查找,
直到 p.getData()指向元素 x 或到达链表的表尾
        p = p.getNext();// 指向下一个元素
        ++j;// 计数器的值增 1
    }
    if (!p.equals(head))// 如果 p 指向表中的某一元素
        return j;// 返回 x 元素在双向循环链表中的位置
    else
        return -1;// x 元素不在双向循环链表中
}
```

提醒:请将上述算法与单链表上相应操作的算法对照学习，特别注意它们不相同的地方。

八、应用性设计实验

编写一个程序，计算出一个单链表中数据域值为一个指定值 x 的结点个数。

实验要求：

(1) 从键盘输入若干个整数，以此序列为顺序建立一个不带头结点的单链表；

(2) 输出此单链表中的各个数据元素值；

(3) 给定一个 x 的具体整数值，计算并返回此单链表中数据域值为 x 的结点个数值。

实验 3： 栈的操作实验

一、实验名称和性质

所属课程	数据结构与算法
实验名称	栈的操作
实验学时	2
实验性质	√验证 □综合 √设计
必做/选做	√必做 □选做

二、实验目的

1．掌握栈的存储结构的表示和实现方法。

2．掌握栈的入栈和出栈等基本操作算法实现。

3．了解栈在解决实际问题中的简单应用。

三、实验内容

1．建立顺序栈，并在顺序栈上实现入栈和出栈操作（验证性内容）。

2．建立链栈，并在链栈上实现入栈和出栈操作（设计性内容）。

3．实现汉诺（Hanoi）塔求解问题（应用性设计内容）。

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件： PC 软件： Win2000 系统及以上; VC 编译器

五、知识准备

前期要求熟练掌握了 Java 语言的编程规则、方法和顺序栈、链栈的基本操作算法。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）根据输入的栈中元素个数 n 和各元素值建立一个顺序栈，并输出栈中各元素值。

（2）将数据元素 e 入栈，并输出入栈后的顺序栈中各元素值。

（3）将顺序栈中的栈顶元素出栈，并输出出栈元素的值和出栈后顺序栈中各元素值。

2. 实验相关原理：

栈是一种插入和删除操作都限制在表的一端进行的特殊线性表,它的操作具有“先进后出”的特性。采用顺序存储结构的栈称为顺序栈。与顺序表一样，顺序栈也是用数组来实现的，假设数组名为 stackElem。由于入栈和出栈操作只能在栈顶进行，所以需再加上一个变量 top 来指示栈顶元素的位置。顺序栈的类描述如下：

```
class SqStack {
    private Object[] stack; //对象数组
    private int top; //当栈非空时，top 始终指向栈顶元素的存储位置；当栈为空时，top 值为-1
     ……
}
```

【核心算法提示】

⑴ 顺序栈入栈操作的基本步骤：首先判断顺序栈是否为满，如果满，则抛出异常，否则将待入栈的数据元素存放在 top 所指示的存储单元中，再使 top 后移一个存储单元位置，即将 top 值增 1。

⑵ 顺序栈出栈操作的基本步骤：首先判断顺序栈是否为空，如果空，则函数返回 null，否则将栈顶指针前移一个存储单元位置，即将 top 值减 1,再返回 top 所指示的栈顶元素值。

【核心算法描述】

⑴ 在当前顺序栈上的入栈操作算法

```
void push(Object e) throws Exception { //将数据元素 e 压入栈顶
    if (top == stack.length-1)// 栈满
        throw new Exception("栈已满");// 输出异常
    else                    // 栈未满
        stack [++top] = e;// top 增 1， e 赋给 top 所指位置，为新的栈顶元素
}
```

⑵ 在当前顺序栈上的出栈操作算法

```
Object pop() {// 移除栈顶对象并作为此函数的值返回该对象
    if (top == -1)           // 栈为空
        System.out.println(“当前栈为空，不能进行出栈操作”);
    else {                  // 栈非空
        return stack [top--];//返回栈顶元素，并修改栈顶指针
    }
}
```

3．源程序代码参考

```
package zsTest;
import java.util.Scanner;
public class SqStack {
    private String[] stack; // 栈存储空间

    private int top = -1; //

    public SqStack(int maxSize) {
        stack = new String[maxSize];// 为栈分配 maxSize 个存储单元
    }

    // 判断顺序栈是否为空
    public Boolean empty() {
        return top == -1;// 如果栈是空的话，返回 true
    }
```

```
// 移除栈顶对象并作为此函数的值返回该对象
public String pop() {
    if (empty()) {// 栈为空
        System.out.println("当前栈为空，不能进行出栈操作");
        return null;
    } else // 栈非空
        return stack[top--];// 修改栈顶指针，并返回栈顶元素
}

// 将数据元素 e 压入栈顶
public void push(String e) {
    if (top == stack.length - 1) // 栈满
        System.out.println("当前栈已满，不能进行入栈操作");
    else
        // 栈未满
        stack[++top] = e; // top 加 1， e 赋给 top 所指位置，为新的栈顶元素
}

// 输出函数，输出栈中所有的元素(从栈顶到栈底)
public void display() {
    for (int i = top; i >= 0; i--)
        System.out.print(stack[i].toString() + " ");// 打印
    System.out.println();
}

public static void main(String[] args) throws Exception {
    SqStack S = new SqStack(100); // 初始化一个新的容量为 100 的顺序栈
    Scanner sc = new Scanner(System.in);
    System.out.print("请输入顺序栈的长度:");
    int n = sc.nextInt();
    System.out.println("请输入顺序栈中的各个数据元素值:");
    for (int i = 0; i < n; i++)
        S.push(Integer.toString(sc.nextInt()));
    System.out.println("建立的顺序栈中各元素为(从栈顶到栈底)： ");
    S.display();
    System.out.println("请输入待入栈的数据值 e:");
    int e = sc.nextInt();
```

```
            S.push(Integer.toString(e));
            System.out.println("入栈后的顺序栈中各元素为(从栈顶到栈底):");
            S.display();
            System.out.println("去除栈顶元素后，顺序栈中各元素为(从栈顶到栈底)：  ");
            S.pop();
            S.display();
            sc.close();
        }
```

}**4．运行结果参考如图 3-1 所示：**

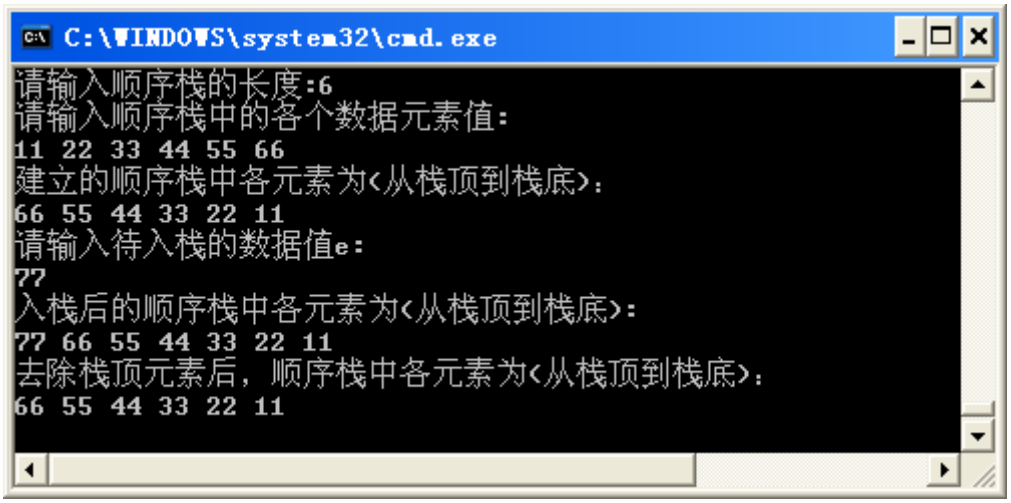


图 3-1：验证性实验运行结果

备注：以下设计性和应用性实验内容学生可根据自己的掌握程度或兴趣自行选择其一或其二完成。

七、设计性实验

编程实现链栈的入栈和出栈操作。

1．实验要求

（1）根据输入的栈中元素个数和各元素值建立一个链栈，并输出链栈中各元素值， 观察输入的内容与输出的内容是否一致,特别注意栈顶元素的位置。

（2）将数据元素 e 入栈，并输出入栈后的链栈中各元素值。

（3）将链栈中的栈顶元素出栈，并输入出栈元素的值和出栈后链栈中各元素值。

2．核心算法提示

采用链式存储结构的栈称为链栈，链栈的结点类描述采用单链表中结点类 Node 的描述。

如果栈中元素序列为$\{a_0, a_1, \cdots, a_{n-1}\}$，则链栈的存储结构如下图 3-1 所示：

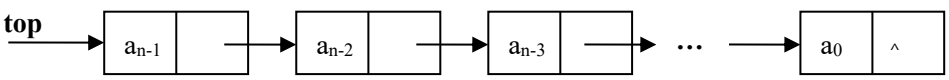


图 3-2： 链栈的存储结构示意图

从图 3-2 可看出，栈的链式存储结构与单链表的存储结构相同，所有在链栈上进行入栈和出栈操作与单链表上的插入和删除操作的主要步骤相同。只不过要特别注意以下几点：

（1）链栈中无需加头结点。

（2）链栈中的指针是指向栈中元素序列的前驱结点。

（3）链栈中的入栈和出栈操作都是在链表的表头进行。

3．核心算法描述

⑴ 在当前链栈上的入栈操作算法

```
StackNode push(StackNode S)
{
        S.next = top;
        top = S;//S 成为新的栈顶
        return top;
   }
```

⑵ 在当前链栈上的出栈操作算法

```
StackNode pop(){
        if(empty()){
          System.out.println ("当前链栈为空！") ;
        }
        StackNode sn = top;
        top = top.next;
        return sn;
}
```

（3）判断当前链栈是否为空操作算法

```
Boolean empty(){
  Return top==null;
}
```

3．源程序代码参考

```
class StackNode{
    String data;
    StackNode next;
}
public class LinkStack {
     StackNode top;
     boolean empty(){
          return top == null;
      }
     StackNode push(StackNode S) {
            S.next = top;
            top = S;
            return top;
      }
```

```
    StackNode pop(){
        if(empty()){
          System.out.println ("当前链栈为空！") ;
        }
        StackNode sn = top;
        top = top.next;
        return sn;
    }
    public static void main(String[] args) {
        try {
            String[] bus = new String[] { "001", "002", "003", "004" };
            LinkStack os = new LinkStack();
            StackNode sn = null;
            for(int i = 0; i < bus.length; i ++){
                sn = new StackNode();
                sn.data = bus[i];
                os.push(sn);
            }
            while(os.top != null)    {
                sn = os.pop();
                System.out.println(sn.data);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

八、应用性设计实验

编程实现汉诺（Hanoi）塔求解问题。

1．实验要求

假设有三个命名为 X、Y 和 Z 的塔座，在塔座 X 上插有 n 个直径大小各不相同且从小到大编号为 1,2,……,n 的圆盘。现要求将塔座 X 上的 n 个圆盘借助于塔座 Y 移至塔座 Z 上，并仍按同样顺序叠排。圆盘移动时必须遵循下列规则：

① 每次只能移动一个圆盘；

② 圆盘可以插在 X、Y 和 Z 中的任何一个塔座上；

③ 任何时刻都不能将一个较大的圆盘压在较小的圆盘上。

2．核心算法提示

当 n=1 时，问题比较简单，只要将编号为 1 圆盘从塔座 X 直接移动到塔座 Z 上即可；当

n>1 时，若能将压在编号为 n 的圆盘上的 n-1 个圆盘从塔座 X 借助于塔座 Z 移至塔座 Y 上，则可先将编号为 n 的圆盘从塔座 X 移至塔座 Z 上，再将塔座 Y 上的 n-1 个圆盘借助于塔座 X 移至塔座 Z 上。而将 n-1 个圆盘从一个塔座借助于另一个塔座而移至到第三个塔座上是一个与原问题具有相同特征属性的问题，只是问题规模小 1,因此可以用解决原问题的方法来解决。

实验 4: 队列的操作实验

一、实验名称和性质

所属课程	数据结构与算法
实验名称	队列的操作
实验学时	2
实验性质	√验证 □综合 √设计
必做/选做	√必做 □选做

二、实验目的

1．掌握队列存储结构的表示和实现方法。

2．掌握队列的入队和出队等基本操作的算法实现。

3．了解队列在解决实际问题中的简单应用。

三、实验内容

1．建立循环顺序队列，并在循环顺序队列上实现入队、出队基本操作（验证性内容）。

2．建立循环链队列，并在循环链队列上实现入队、出队基本操作（设计性内容）。

3．设计一个程序模仿操作系统的进程管理问题（应用性设计内容）。

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件： PC 软件： Win2000 系统及以上; VC 编译器

五、知识准备

前期要求熟练掌握了 Java 语言的编程规则、方法和顺序循环队列、循环链队列的基本操作算法。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）根据输入的队列长度 n 和各元素值建立一个循环顺序表表示的队列（循环队列），并输出队列中各元素值。

（2）将数据元素 x 入队，并输出入队后的队列中各元素值。

（3）将循环队列的队首元素出队，并输出出队元素的值和出队后队列中各元素值。

2. 实验相关原理：

队列是一种插入操作限制在表尾，而删除操作限制在表头进行的特殊线性表，它的操作具有“先进先出”的特性。采用顺序存储结构的队列称为顺序队列，顺序队列的类描述如下：

```
class CircleSqQueue implements IQueue {
    private Object[] queueElem; // 队列存储空间
    private int front;// 队首的引用，若队列不空，指向队首元素
    private int rear;// 队尾的引用，若队列不空，指向队尾元素的下一个位置
```

……

}

【核心算法提示】

⑴ 循环顺序队列入队操作的基本步骤：首先判断队列是否为满，如果队列满，则函数返回 ERROR，否则将待入队的数据元素 x 存放在尾指针 rear 所指示的存储单元中，再使尾指针沿着顺序循环存储空间后移一个位置。

⑵ 循环顺序队列出队操作的基本步骤：首先判断队列是否为空，如果队列空，则函数返回 null，否则将头指针所指示的队首元素用 t 返回其值，再使头指针沿着顺序循环存储空间后移一个位置。

【核心算法描述】

假设以**少用一个存储单元的方法**来解决区分队列判空和判满的条件，则入队和出队操作算法可描述如下：

⑴ 在当前循环顺序队列上的入队操作算法

```
void offer(Object x) throws Exception { // 将指定的元素 x 插入队列
    if ((rear + 1) % queueElem.length == front)// 队列满
        throw new Exception("队列已满");// 输出异常
    else {// 队列未满
        queueElem[rear] = x;// x 赋给队尾元素
        rear = (rear + 1) % queueElem.length;// 修改队尾指针
    }
}
```

⑵ 在当前循环顺序队列上的出队操作算法

```
Object poll() {// 移除队首元素并返回其值，如果此队列为空，则返回 null
    if (front == rear)// 队列为空
        return null;
    else {
        Object t = queueElem[front];// 取出队首元素
        front = (front + 1) % queueElem.length;// 更改队首的位置
        return t;// 返回队首元素
    }
}
```

3．源程序代码参考

```
class CircleSqQueue  {
    private Object[] queueElem; // 队列存储空间
    private int front;// 队首的引用，若队列不空，指向队首元素
    private int rear;// 队尾的引用，若队列不空，指向队尾元素的下一个位置
    // 循环队列类的构造函数
    public CircleSqQueue(int maxSize) {
        front = rear = 0;// 队头、队尾初始化为 0
        queueElem = new Object[maxSize];// 为队列分配 maxSize 个存储单元
    }
```

```
    // 将指定的元素 x 插入队列
    public void offer(Object x) throws Exception {
        if ((rear + 1) % queueElem.length == front)// 队列满
            throw new Exception("队列已满");// 输出异常
        else {// 队列未满
            queueElem[rear] = x;// x 赋给队尾元素
            rear = (rear + 1) % queueElem.length;// 修改队尾指针
        }
    }
    // 移除队首元素并返回其值，如果此队列为空，则返回 null
    public Object poll() {
        if (front == rear)// 队列为空
            return null;
        else {
            Object t = queueElem[front];// 取出队首元素
            front = (front + 1) % queueElem.length;// 更改队首的位置
            return t;// 返回队首元素
        }
    }
    // 输出函数，输出队列中的所有元素(从队首到队尾)
    public void display() {
        if (front==rear) {
            for (int i = front; i != rear; i = (i + 1) % queueElem.length)
                System.out.print(queueElem[i].toString() + " ");
            System.out.println();
       }
       else
            System.out.println("此队列为空");
        }
}
//测试类
public class SY4_CircleSqQueue {
 public static void main(String[] args) throws Exception {
        CircleSqQueue Q = new CircleSqQueue(100);  //创建一个容量为 100 的空队列
        Scanner sc=new Scanner(System.in);
          System.out.print("请输入循环顺序队列的长度:");
          int n=sc.nextInt();
          System.out.println("请输入循环顺序队列中的各个数据元素值:");
```

```
            for(int i=0;i<n;i++)
                Q.offer(sc.nextInt());
            System.out.println("建立的循环顺序队列中各元素为(从队首到队尾)：  ");
                Q.display();
            System.out.println("请输入待入队的数据值 x:");
            int x=sc.nextInt();
            Q.offer(x);
            System.out.println("入队后的循环顺序队列中各元素为(从队首到队尾):");
            Q.display();
            System.out.println("去除队首元素后，队列中各元素为(队首到队尾)::   ");
                Q.poll();
                Q.display();
        }
    }
```

4．运行结果参考如图 4-1 所示：

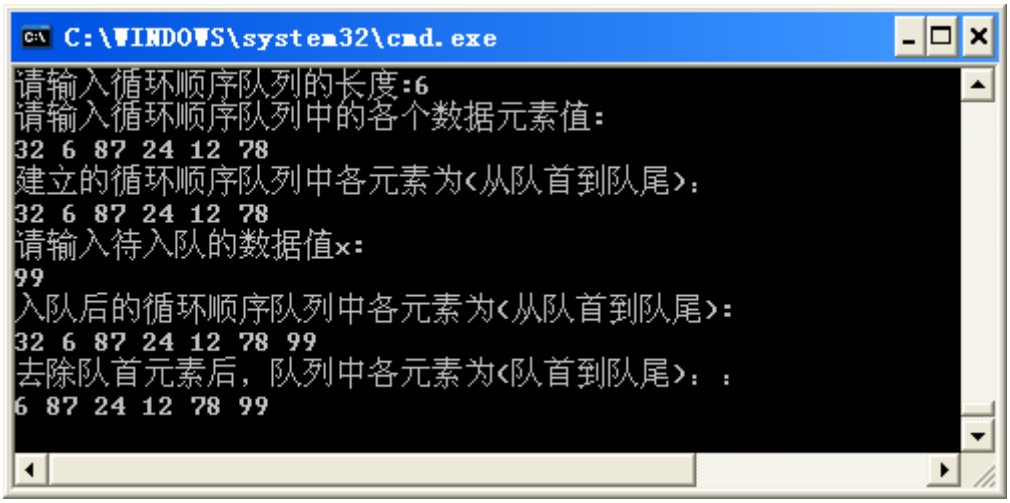


图 4-1：验证性实验运行结果

备注：以下设计性和应用性实验内容学生可根据自己的掌握程度或兴趣自行选择其一或其二完成。

七、设计性实验

编程实现对循环链队列的入队和出队操作。

1．实验要求

⑴根据输入的队列长度 n 和各元素值建立一个带头结点的循环链表表示的队列（循环链队列），并且只设一个尾指针来指向尾结点，然后输出队列中各元素值。

⑵将数据元素 e 入队，并输出入队后的队列中各元素值。

⑶将循环链队列的队首元素出队，并输出出队元素的值和出队后队列中各元素值。

2．核心算法分析

采用链式存储结构的队列称为链队列，链队列中的结点类描述采用单链表中结点类 Node 的描述。

如果队列中元素序列为{a_0, a_1，…,a_{n-1}}，则只用尾指针指示带头结点的循环链队列的存储结构如下图 4-2 所示：

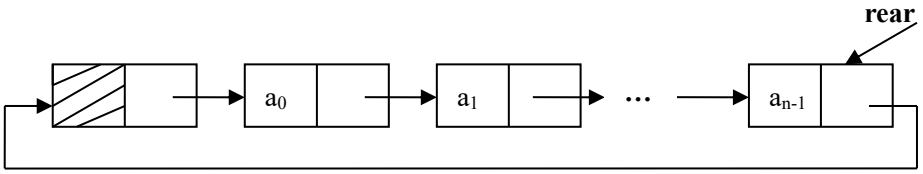


图 4-2： 循环链队列的存储结构示意

此循环链队列类可描述如下：

```
class CircleLinkQueue {
    private Node rear;// 循环链队列的尾指针
    // 构造方法
    public CircleLinkQueue() {//构造一个带头结点的空循环链队列
        rear = new Node(); // 初始化头结点并用尾指针指向它
        rear.setNext(rear);
     }
   ……
}
```

从图 4-2 可看出，队列的循环链式存储结构与循环单链表的存储结构相同，所以在循环链队列上进行入队和出队操作与单链表上的插入和删除操作的主要步骤相同。只不过要特别注意以下几点：

⑴ 此循环链队列是通过一个指向队尾结点的指针 rear 来标识的,则 rear 指向队尾元素结点，rear 的后继指针 rear.next 指向队列的头结点，而 rear 的后继的后继指针 rear.next.next 则指向队首元素结点。

⑵ 队列的入队操作是在链表的表尾（队尾）进行，而出队操作则在链表的表头（队首）进行。

⑶ 在出队操作时要注意：如果当链队列中只有一个结点，即被删结点既是队首结点，又是队尾结点时，则要进行尾指针的修改，并使删除后的队列形成空的循环链队列。

3．核心算法描述

⑴ 在当前循环链队列上的入队操作算法

```
void offer  ( Object x) { //将指定的元素 x 插入到带头结点的循环链队列中
        Node p= new Node(x); // 生成新结点
        p.setNext(rear.getNext());// 插入链队列的尾部
        rear.setNext(p);
        rear=p;
}
```

⑵ 在当前循环链队列上的出队操作算法

```
Object poll(){
```

```
// 移除带头结点的循环链队列中的队首元素并返回其值，如果此队列为空，则返回 null
  if (rear.getNext()==rear)    //队列为空
     return  null;
 else{
      Node p = rear.getNext().getNext();// p 指向待删除的队首结点
       if (p==rear) {//被删结点是即是队首结点，又是队尾结点
          rear=rear.getNext();
          rear.setNext(rear);  //删除队首结点后，链队列变成了空循环链队列
       }
       else
             rear.getNext().setNext(p.getNext());// 删除队首结点
       return p.getData();
   }
}
```

八、应用性设计实验

编程实现模仿操作系统的进程管理问题。

1．问题描述

操作系统中采用一个优先队列来管理进程。当优先级队列中有多个进程排队等待系统响应时，只要 CPU 空闲，进程管理系统就会从优先队列中找出优先级最高的进程首先出队并占有 CPU 资源，即按进程服务的优先级，优先级高的先服务；优先级相同的按先到先服务的原则进行管理。

2．问题分析

假设操作系统中每个进程的模仿数据由进程号和进程优先级两部分组成，进程号是每个不同进程的惟一标识，优先级通常是一个 0 到 40 的数值，规定 0 为优先级最高，40 为优先级最低。如下为一组模拟数据：

进程号	进程优先级
1	5
2	10
3	20
4	10
5	40
6	10

则按操作系统的进程管理，进程的服务顺序应该是：

进程号	进程优先级
1	5
2	10
4	10
6	10

3 20

5 40

此问题只要通过优先级队列的入队和出队操作即可得到解决，所以需要编写优先级队列类 PriorityQueue，在此类中包含实现入队和出队的相应方法。

3．算法提示

为操作方便，优先级队列可以采用链式存储结构，其结点的类可采用单链表的结点 Node 类，但 Node 类中的 data 成员又必须定义为由进程号和优先级两个成员所构成的一个类。优先级队列入队也不仅仅限制在队尾进行，而是顺序插入到队列的合适位置，以确保队列是按优先级从高到低度的顺序排放。

实验 5:　二叉树的操作实验

一、实验名称和性质

所属课程	数据结构与算法
实验名称	二叉树的操作
实验学时	2
实验性质	√验证 □综合 √设计
必做/选做	√必做 □选做

二、实验目的

1．理解二叉树的类型定义与性质。

2．掌握二叉树的二叉链表存储结构的表示和实现方法。

3．掌握二叉树遍历操作的算法实现。

4．熟悉二叉树遍历操作的应用。

三、实验内容

1．建立二叉树的二叉链表存储结构。

2．实现二叉树的先根、中根和后根三种遍历操作（验证性内容）。

3．应用二叉树的遍历操作来实现判断两棵二叉树是否相等的操作（设计性内容）。

4．求从二叉树根结点到指定结点 p 之间的路径（应用性设计内容）。

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件：　PC　软件：　Win2000 系统及以上;　VC 编译器

五、知识准备

前期要求掌握二叉树的二叉链表的存储结构表示和三种遍历操作算法。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）假设二叉树的结点值是字符，根据输入的一棵二叉树的标明了空子树的完整先根遍历序列建立一棵以二叉链表表示的二叉树。

（2）对二叉树进行先根、中根和后根遍历操作，并输出遍历序列，观察输出的序列是否与逻辑上的序列一致。

（3）主程序中要求设计一个菜单，允许用户通过菜单来多次选择执行哪一种遍历操作。

2. 实验相关原理：

二叉树的形式定义：二叉树或为空树，或是由一个根结点加上两棵分别称为左子树和右子树的、互不交的二叉树组成。

二叉树的二叉链式存储结构的结点类描述如下：

```
class BinTreeNode {
```

```
        private Object data;// 结点的数据域
        private BinTreeNode lchild, rchild; // 左、右孩子域
        // 构造一个空结点
        public BinTreeNode() {
            this(null);
        }
        // 构造一棵左、右孩子域为空的二叉树
        public BinTreeNode(Object data) {
            this(data, null, null);
        }
        // 构造一棵数据域和左、右孩子域都不为空的二叉树
  public BinTreeNode(Object data, BiTreeNode lchild, BiTreeNode rchild) {
            this.data = data;
            this.lchild = lchild;
            this.rchild = rchild;
        }
       ……
    }
```

【核心算法提示】

二叉树的遍历是指按某条搜索路径周游二叉树，对树中每个结点访问一次且仅访问一次。其中先根、中根和后根遍历操作步骤分别为：

（1）先根遍历：若二叉树为空树，则空操作；否则先访问根结点，再先要遍历左子树，最后先根遍历右子树。

（2）中根遍历：若二叉树为空树，则空操作；否则先中根遍历左子树，再访问根结点，最后中根遍历右子树。

（3）后根遍历：若二叉树为空树，则空操作；否则先后根遍历左子树，再后根遍历右子树，最后访问根结点。

注意：这里的“访问”的含义可以很广，几乎可以含盖对结点的任何一种操作。如：输出结点的信息、判断结点是否为叶子结点等等。

由于二叉树是一种递归定义，所以，要根据二叉树的某种遍历序列来实现建立一棵二叉树的二叉链表存储结构，则可以模仿对二叉树遍历的方法来加以实现。如：如果输入的是一棵二叉树的标明了空子树的完整先根遍历序列，则可利用先根遍历方法先生成根结点，再用递归函数调用来实现左子树和右子树的建立。所谓标明了空子树的完整先根遍历序列就是在先序遍历序列中加入空子树信息。

【核心算法描述】

(1) 由标明空子树的完整先根遍历序列建立一棵二叉树的算法

```
private static int index = 0;// 用于记录 preStr 的索引值
public BinTree(String preStr) {// 由标明空子树的先根遍历序列建立一棵二叉树
    char c = preStr.charAt(index++);// 取出字符串索引为 index 的字符，且 index 增 1
    if (c !=' ') {// 字符不为空格
        root = new BinTreeNode(c);// 建立树的根结点
```

```
            root.setLchild(new BinTree(preStr).root);// 建立树的左子树
            root.setRchild(new BinTree(preStr).root);// 建立树的右子树
        } else
            root = null;
    }
```

(2)先根遍历二叉树的递归算法

```
    public void preRootTraverse(BinTreeNode T) {
        if (T != null) {
            System.out.print(T.getData()); // 访问根结点
            preRootTraverse(T.getLchild());// 先根遍历左子树
            preRootTraverse(T.getRchild());// 先根遍历右子树
        }
    }
```

(3)中根遍历二叉树的递归算法

```
    public void inRootTraverse(BiTreeNode T) {
        if (T != null) {
            inRootTraverse(T.getLchild());// 中根遍历左子树
            System.out.print(T.getData()); // 访问根结点
            inRootTraverse(T.getRchild());// 中根遍历右子树
        }
    }
```

(4)后根遍历二叉树的递归算法

```
    public void postRootTraverse(BiTreeNode T) {
        if (T != null) {
            postRootTraverse(T.getLchild());// 后根遍历左子树
            postRootTraverse(T.getRchild());// 后根遍历右子树
            System.out.print(T.getData()); // 访问根结点
        }
    }
```

3．源程序代码参考

```
package zsTest;

import java.io.BufferedReader;
import java.io.CharArrayReader;
import java.io.InputStreamReader;
import java.util.Scanner;
// 二叉树的二叉链式存储结构中的结点类

class BinTreeNode {
```

```
private Object data;

private BinTreeNode lchild, rchild;

public BinTreeNode() {
    this(null);
}

public BinTreeNode(Object data) {
    this(data, null, null);
}

public BinTreeNode(Object data, BinTreeNode lchild, BinTreeNode rchild) {
    this.data = data;
    this.lchild = lchild;
    this.rchild = rchild;
}

public Object getData() {
    return data;
}

public void setData(Object data) {
    this.data = data;
}

public BinTreeNode getLchild() {
    return lchild;
}

public void setLchild(BinTreeNode lchild) {
    this.lchild = lchild;
}

public BinTreeNode getRchild() {
    return rchild;
}
```

```
    public void setRchild(BinTreeNode rchild) {
        this.rchild = rchild;
    }
}

// 二叉链式存储结构下的二叉树类
class BinTree {
    private BinTreeNode root;// 树的根结点

    public BinTree() {// 构造一棵空树
        this.root = null;
    }

    public BinTree(BinTreeNode root) {// 构造一棵树
        this.root = root;
    }

    // 根据输入的前序序列，创建二叉树
    public BinTree createBinTree() {
        BinTree r;
        char ch = ' ';
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            CharArrayReader car = new CharArrayReader(in.readLine().toCharArray());
            ch = (char) car.read(); // 输入字符
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        if (ch == ' ')
            // 空字符时停止创建
            r = null;
        else {
            r = new BinTree();
            // 创建二叉树对象
            r.root = new BinTreeNode(ch);
            // 生成二叉树对象的根结点
            BinTreeNode lChildNode = null;
            BinTree lChild = createBinTree();
```

```
            if (lChild != null) lChildNode = lChild.root;
            r.root.setLchild(lChildNode);
            // 创建其左子树
            BinTreeNode rChildNode = null;
            BinTree rChild = createBinTree();
            if (rChild != null) rChildNode = rChild.root;
            r.root.setRchild(rChildNode);
            // 创建其右子树
        }
        return r;
    }
    // 查找元素算法：在以 R 为根结点的二叉树中查找数据元素 x，如果查找到则返回该结点，否则返回空。

    public BinTreeNode search(BinTreeNode r, Object x) {
        BinTreeNode temp;
        if (r == null) // 空树，查找失败
            return null;

        if (r.getData().toString().equals(x.toString())) // 比较根结点，相同则成功返回该结点
            return r;
        if (r.getLchild() != null) { // 在左子树查找

            temp = search(r.getLchild(), x); // 左子树查找结果
            if (temp != null) // 查找结果不空，则成功返回该结点
                return temp;
        }
        if (r.getRchild() != null) { // 在右子树查找
            temp = search(r.getRchild(), x); // 右子树查找结果
            if (temp != null) // 查找结果不空，则成功返回该结点
                return temp;
        }
        return null; // 查找不到，返回空
    }

    // 求二叉树的高度：二叉树的高度为二叉树中结点层次的最大值。所以，需要遍历左子树求其高度，再遍历右子树求其高度，而二叉树的高度是左、右子树层次高者再加 1。
    public int depthofBinTree(BinTreeNode r) {
        int high, lh, rh;
```

```
    if (r == null) // 二叉树空，高度为0
        return 0;
    else {
        lh = depthofBinTree(r.getLchild()); // 左子树高度
        rh = depthofBinTree(r.getRchild()); // 右子树高度
        if (lh > rh) // 左子树高度大
            high = lh + 1; // 二叉树的高度为左子树高度加1
        else
            high = rh + 1; // 否则高度为右子树高度加1
    }
    return high;
}

// 先根遍历二叉树基本操作的递归算法
public void preRootTraverse(BinTreeNode T) {
    if (T != null) {
        System.out.print(T.getData()); // 访问根结点
        preRootTraverse(T.getLchild());// 访问左子树
        preRootTraverse(T.getRchild());// 访问右子树
    }
}

// 中根遍历二叉树基本操作的递归算法
public void inRootTraverse(BinTreeNode T) {
    if (T != null) {
        inRootTraverse(T.getLchild());// 访问左子树
        System.out.print(T.getData()); // 访问根结点
        inRootTraverse(T.getRchild());// 访问右子树
    }
}

// 后根遍历二叉树基本操作的递归算法
public void postRootTraverse(BinTreeNode T) {
    if (T != null) {
        postRootTraverse(T.getLchild());// 访问左子树
        postRootTraverse(T.getRchild());// 访问右子树
        System.out.print(T.getData()); // 访问根结点
    }
```

```
    }

    public BinTreeNode getRoot() {
        return root;
    }

    public void setRoot(BinTreeNode root) {
        this.root = root;
    }
}

/**
 * 测试类。
 */
public class SY5_TraverBiTree {
    public static void main(String[] args) {
        BinTree T = new BinTree();
        BinTreeNode r;
        BinTreeNode temp = null;
        Object x = null;
        int height = 0;
        System.out.println("请输入前序序列以便创建二叉树，以空格字符代表空节点！");//每输入一个字符按下回车
        T = T.createBinTree();//
        r = T.getRoot();//
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入要查找的节点！");
        x = sc.next();
        temp = T.search(r, x);//
        if (temp == null)
            System.out.println("查找数据元素" + x + "失败！");
        else
            System.out.println("查找数据元素" + x + "成功：");
        height = T.depthofBinTree(r);//
        System.out.println("该二叉树的高度为：" + height);

        while (true) {
            System.out.println(" 1--先根遍历    2--中根遍历    3--后根遍历    4--退出 ");
```

```
            System.out.print("请输入选择(1-4):");
            int i = sc.nextInt();
            switch (i) {
            case 1:
                System.out.print("先根遍历为:  ");
                T.preRootTraverse(T.getRoot());//
                System.out.println();
                break;
            case 2:
                System.out.print("中根遍历为:  ");
                T.inRootTraverse(T.getRoot());//
                System.out.println();
                break;
            case 3:
                System.out.print("后根遍历为:  ");
                T.postRootTraverse(T.getRoot());//
                System.out.println();
                break;
            case 4:
                sc.close();
                return;
            }
        }
    }
}
```

4．运行结果参考如图 5-1 所示：

图 5-1： 验证性实验运行结果

说明：上述对应的二叉树如图 5-2 所示。

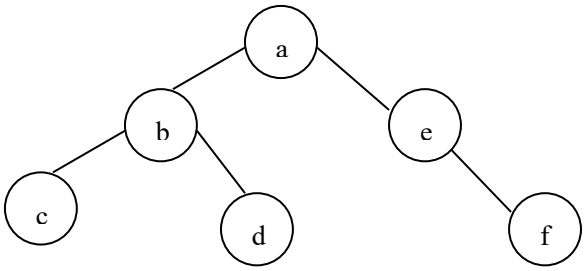


图 5-2：建立的二叉树

备注：以下设计性和应用性实验内容学生可根据自己的掌握程度或兴趣自行选择其一或其二完成。

七、设计性实验

编程实现根据二叉树的先序遍历序列和中序遍历序列来建立两棵二叉树，并判断这两棵二叉树是否相等。

1. 实验要求

⑴ 假设二叉树的结点值是字符，请分别根据输入的两棵二叉树的先根遍历序列和中根遍历序列来建立二叉链表表示的两棵二叉树。

⑵ 分别利用先根、中根和后根遍历方法来实现判断两棵二叉树是否相等的操作。

⑶ 主程序中要求设计一个菜单，允许用户通过菜单来多次选择执行利用哪一种遍历方法来判断两棵二叉树是否相等。

2. 核心算法提示

⑴ 二叉树的建立

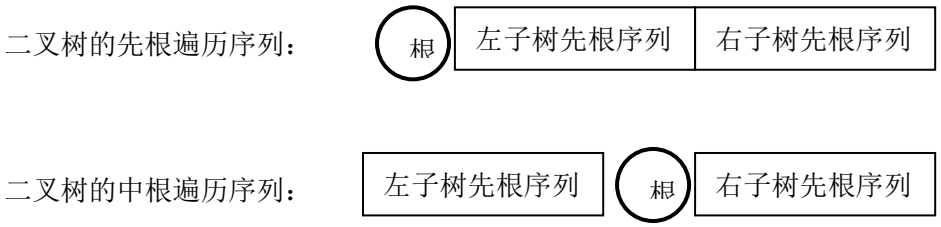


图 5-3： 二叉树的先根、中根遍历序列特征图

假设二叉树的先根遍历序列和中根遍历序列已经存储在数组 preOrder 和 inOrder 中，其序列分列具有如图 5-3 所示的特征。

根据此特征，建立二叉树的基本步骤可归纳为：

① 取先根遍历序列中的第一个字符作为根结点的数据域值建立根结点；

② 在中根遍历序列中查找确定这个根结点在中根遍历序列中的位置，由此得到在根结点左边的序列即为此根结点左子树的中根遍历序列，而右边的序列即为此根结点右子树的中根遍历序列。

③ 根据左、右子树的中根遍历序列中的字符个数再在先序遍历序列中确定左、右子树的

先序遍历序列。

④ 根据（2）、（3）确定的左、右子树的先根和中根遍历序列采用递归调用的方法建立根结点的左、右子树。

要实现上述建立二叉树的步骤，需要引入 5 个参数：preOrder、inOrder、preIndex、inIndex 和 count。其中：参数 preOrder 是整棵树的先根遍历序列；inOrder 是整棵树的中根遍历序列；preIndex 是先根遍历序列在 preOrder 中的开始位置；inIndex 是中根遍历序列在 inOrder 中的开始位置；count 表示树中结点的个数。

⑵ 判断两棵二叉树是否相等

假设 T1 和 T2 是两棵二叉树，如果两棵二叉树都是空树；或者两棵树的根结点的值相等，并且根结点的左、右子树分别也相等，则称二叉树 T1 与 T2 是相等的。所以，也可以利用先序、中序和后序遍历方法，采用递归函数来判断两棵树是否相等。假设两棵树相等，函数返回 1，否则返回 0。下面以用先序遍历方法为例，说明其基本操作步骤为：

① 如果两棵二叉树都为空，则函数返回 1；

② 如果两棵二叉树都不为空，则先判断两棵树的根结点值是否相等，如果相等，则再采用递归调用的方法判断它的左、右子树是否相等，如果都相等则函数返回 1；

③ 其它情况都返回 0。

3. 核心算法描述

⑴ 根据先根遍历和中序遍历序列建立一棵二叉树的算法

```
BiTreeNode creatBiTree(String preOrder,String inOrder,int preIndex,int inIndex,int count){
        if (count > 0) {// 先根和中根非空
         char r = preOrder.charAt(preIndex);// 取先根字符串中的第一个元素作为根结点
         int i = 0;
         for (; i < count; i++)
             // 寻找根结点在中根遍历字符串中的索引
             if (r == inOrder.charAt(i + inIndex))
                 break;
         root = new BiTreeNode(r);// 建立树的根结点
         root.setLchild(creatBiTree(preOrder, inOrder, preIndex + 1, inIndex,i));
                                                              // 建立树的左子树
         root.setRchild(creatBiTree(preOrder, inOrder, preIndex + i + 1,
                 inIndex + i + 1, count - i - 1));            // 建立树的右子树
    }
      return root;
   }
```

⑵用先根遍历方法判断两棵二叉树是否相等的算法

```
boolean isEqual_pre(BiTreeNode T1,BiTreeNode T2){
       if (T1==null&&T2==null)
           return true;
       if (T1!=null&&T2!=null)
           if (T1.getData().equals(T2.getData()))
               if (isEqual_pre(T1.getLchild(),T2.getLchild()))
```

```
                if(isEqual_pre(T1.getRchild(),T2.getRchild()))
                    return true;
        return false;
    }
```

(3) 用中根遍历方法判断两棵二叉树是否相等的算法

```
    boolean isEqual_in(BiTreeNode T1,BiTreeNode T2){
        if (T1==null&&T2==null)
            return true;
        if (T1!=null&&T2!=null)
            if (isEqual_in(T1.getLchild(),T2.getLchild()))
                if (T1.getData().equals(T2.getData()))
                   if(isEqual_in(T1.getRchild(),T2.getRchild()))
                       return true;
        return false;
    }
```

(4) 用后根遍历方法判断两棵二叉树是否相等的算法

```
    boolean isEqual_post(BiTreeNode T1,BiTreeNode T2){
        if (T1==null&&T2==null)
            return true;
        if (T1!=null&&T2!=null)
            if (isEqual_post(T1.getLchild(),T2.getLchild()))
                if(isEqual_post(T1.getRchild(),T2.getRchild()))
                   if (T1.getData().equals(T2.getData()))
                       return true;
        return false;
    }
```

八、设计性实验

编程求从二叉树根结点到到指定结点 p 之间的路径。

1. 实验要求

⑴ 假设二叉树的结点值是字符，请根据输入的二叉树后序遍历序列和中序遍历序列来建立二叉链表表示的二叉树,并对其进行某种遍历,输出遍历序列以验证建立的二叉树是否正确。

⑵ 任意给定一结点 p,输出从二叉树的根结点到该结点 p 的路径。

2. 核心算法提示

要求出从根结点到指定结点 p 之间的路径，可利用后序遍历的非递归算法来实现。由于后序遍历当访问到 p 所指结点时，栈中所有结点均为 p 所指结点的祖先，由这些祖先便构成了一条从根结点到 p 结点之间的路径。

实验 6:　二叉排序树的操作实验

一、实验名称和性质

所属课程	数据结构与算法
实验名称	二叉排序树的操作
实验学时	2
实验性质	√验证 □综合 √设计
必做/选做	□必做 √选做

二、实验目的

1．掌握二叉排序树的含义及其在计算机中的存储实现。

2．掌握在二叉排序树上查找操作的算法实现。

3．掌握二叉排序树的插入、删除操作的算法实现。

三、实验内容

1．建立二叉排序树。

2．在二叉排序树上实现对给定值进行查找操作（验证性内容）。

3．判断一棵二叉树是否为二叉排序树（设计性内容）。

4．在采用二叉排序树为数据结构的学生信息管理系统中实现查找操作（应用性设计内容）。

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件：　PC　软件：　Win2000 系统及以上;　VC 编译器

五、知识准备

前期要求掌握二叉排序树的含义、二叉排序树上的查找算法和二叉排序上的插入、删除操作的算法。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）按照输入的 n 个关键字序列顺序建立二叉排序树，二叉排序树采用二叉链表的存储结构。

（2）先输入待查找记录的关键字值 key,然后在二叉排序树上查找该记录，如果在二叉排序树中存在该记录，则显示“找到”的信息，否则显示“找不到”的信息。

2. 实验相关原理：

二叉排序树的性质如下：

（1）若左子树不空，则左子树上所有结点的值均小于根结点的值。

（2）若右子结不空，则右子树上所有结点的值均大于根结点的值。

（3）左右子树又分别是二叉排序树。

有关类描述如下：

(1) 二叉排序树的二叉链表的结点类描述为：

```
class BiTreeNode {
     private int key;// 结点的关键字
     private BiTreeNode lchild, rchild; // 左右孩子
     ……
}
```

(2) 二叉排序树类描述为：

```
class BSTree { //二叉排序树类
    protected BiTreeNode root;   //根结点
    public BSTree() {  //构造空二叉排序树
        root = null;
    }
    ……
}
```

【核心算法提示】

（1）二叉排序树查找操作的基本步骤：对于给定的待查找记录的关键字值 key，在二叉排序树非空的情况下，先将给定的值 key 与二叉排序树的根结点的关键字值进行比较，如果相等，则查找成功，函数返回找到的结点，否则，如果给定的值 key 小于根结点的关键字值，则在二叉排序树的左子树上继续查找；如果给定的值 key 大于根结点的关键字值，则在二叉排序树的右子树上继续查找，直到查找成功，或子树为空即查找失败函数返回 null 为止。

（2）二叉排序树的插入操作的基本步骤（用递归的方法）：如果已知二叉排序树是空树，则插入的结点成为二叉排序树的根结点；如果待插入结点的关键字值小于根结点的关键字值，则插入到左子树中；如果待插入结点的关键字值大于根结点的关键字值，则插入到右子树中。

【核心算法描述】

(1) 二叉排序树上的查找算法

```
//查找关键字值为 key 的结点,若查找成功返回该结点，否则返回 null
  public BiTreeNode searchBST(int key) {
         return searchBST(root, key);
  }
   //二叉排序树上的递归查找算法
  private BiTreeNode searchBST(BiTreeNode p, int key) {
      if (p != null) {
          if (key==p.getKey()){ //查找成功
              return p;
          }
          //     System.out.print(((RecordNode) p.getData()).getKey() + "? ");
          if (key< p.getKey()) {
              return searchBST(p.getLchild(), key);     //在左子树中查找
          } else {
              return searchBST(p.getRchild(), key);    //在右子树中查找
          }
      }
```

```
            return null;
        }
```

(2) 二叉排序树上的插入算法

```
//在二叉排序树中插入关键字为Keyt 的结点,若插入成功返回 true,否则返回 false
public boolean insertBST(int key) {
    if (root == null) {
        root = new BiTreeNode(key); //建立根结点
        return true;
    }
    return insertBST(root, key);
}
//将关键字为keyt 的结点插入到以 p 为根的二叉排序树中的递归算法
private boolean insertBST(BiTreeNode p, int key) {
    if (key==p.getKey()) {
        return false;             //不插入关键字重复的结点
    }
    if (key<p.getKey()) {
        if (p.getLchild() == null) {        //若 p 的左子树为空
            p.setLchild(new BiTreeNode(key));  //建立叶子结点作为 p 的左孩子
            return true;
        } else {                        //若 p 的左子树非空
            return insertBST(p.getLchild(), key);   //插入到 p 的左子树中
        }
    } else if (p.getRchild() == null) {    //若 p 的右子树为空
        p.setRchild(new BiTreeNode(key));    //建立叶子结点作为 p 的右孩子
        return true;
    } else {                            //若 p 的右子树非空
        return insertBST(p.getRchild(), key);   //插入到 p 的右子树中
    }
}
```

3．源程序代码参考

```
import java.util.Scanner;
//二叉树的二叉链表的结点类
class BiTreeNode {
 private int key;// 结点的关键字
 private BiTreeNode lchild, rchild; // 左右孩子
    public BiTreeNode(int key) {// 构造一棵左右孩子为空的结点
     this(key, null, null);
 }
 public BiTreeNode(int key, BiTreeNode lchild, BiTreeNode rchild) {// 构造一棵数据元素和左右孩子都不为空的结点
     this.key = key;
```

```
        this.lchild = lchild;
        this.rchild = rchild;
    }
    public int getKey() {
        return key;
    }
    public BiTreeNode getLchild() {
        return lchild;
    }
    public BiTreeNode getRchild() {
        return rchild;
    }
    public void setKey(int key) {
        this.key = key;
    }
    public void setLchild(BiTreeNode lchild) {
        this.lchild = lchild;
    }
    public void setRchild(BiTreeNode rchild) {
        this.rchild = rchild;
    }
}
class BSTree { //二叉排序树类
    protected BiTreeNode root;   //根结点
    public BSTree() {  //构造空二叉排序树
        root = null;
    }
    public BSTree(BiTreeNode root) {  //构造根结点为 root 的二叉排序树
        this.root = root;
    }
    public BiTreeNode getRoot() {
        return root;
    }
    public void setRoot(BiTreeNode root) {
        this.root = root;
    }
    public void inOrderTraverse(BiTreeNode T) { //中根次序遍历以 T 结点为根的二叉树
        if (T!= null) {
```

```
            inOrderTraverse(T.getLchild());
            System.out.print(T.getKey() + "  ");
            inOrderTraverse(T.getRchild());
        }
    }
    //查找关键字值为key的结点,若查找成功返回该结点，否则返回null
    public BiTreeNode searchBST(int key) {
           return searchBST(root, key);
    }
    //二叉排序树查找的递归算法
    private BiTreeNode searchBST(BiTreeNode p, int key) {
        if (p != null) {
            if (key==p.getKey()) //查找成功
            {
                return p;
            }
            //     System.out.print(((RecordNode) p.getData()).getKey() + "? ");
            if (key< p.getKey()) {
                return searchBST(p.getLchild(), key);     //在左子树中查找
            } else {
                return searchBST(p.getRchild(), key);    //在右子树中查找
            }
        }
        return null;
    }
   //在二叉排序树中插入关键字为Keyt的结点,若插入成功返回true,否则返回false
    public boolean insertBST(int key) {
        if (root == null) {
            root = new BiTreeNode(key); //建立根结点
            return true;
        }
        return insertBST(root, key);
    }
    //将关键字为keyt的结点插入到以p为根的二叉排序树中的递归算法
    private boolean insertBST(BiTreeNode p, int key) {
        if (key==p.getKey()) {
            return false;             //不插入关键字重复的结点
        }
```

```
        if (key<p.getKey()) {
            if (p.getLchild() == null) {        //若 p 的左子树为空
                p.setLchild(new BiTreeNode(key));  //建立叶子结点作为 p 的左孩子
                return true;
            } else {                        //若 p 的左子树非空
                return insertBST(p.getLchild(), key);   //插入到 p 的左子树中
            }
        } else if (p.getRchild() == null) {    //若 p 的右子树为空
            p.setRchild(new BiTreeNode(key));    //建立叶子结点作为 p 的右孩子
            return true;
        } else {                            //若 p 的右子树非空
            return insertBST(p.getRchild(), key);   //插入到 p 的右子树中
        }
    }
}
//测试类
public class SY7_BSTree {
   public static void main(String args[]) {
        BSTree bstree = new BSTree();
        Scanner sc=new Scanner(System.in);
        System.out.print("请输入二叉排序树的结点个数:");
        int n=sc.nextInt();
        System.out.print("请输入结点的关键字序列:");
        for (int i = 0; i < n; i++) {  //输入关键字序列
            bstree.insertBST(sc.nextInt()) ;
         }
        System.out.println("\n 创建的二叉排序树的中序遍历序列为:   ");
        bstree.inOrderTraverse(bstree.getRoot());
         System.out.println();
         System.out.println("请输入待查找的关键字:   ");
        int key=sc.nextInt();
         BiTreeNode found = bstree.searchBST(key);
         if (found != null) {
            System.out.println("查找成功!");
         } else {
            System.out.println("查找失败!");
         }
   }
```

}

4. 运行结果参考如图 7-1 所示：

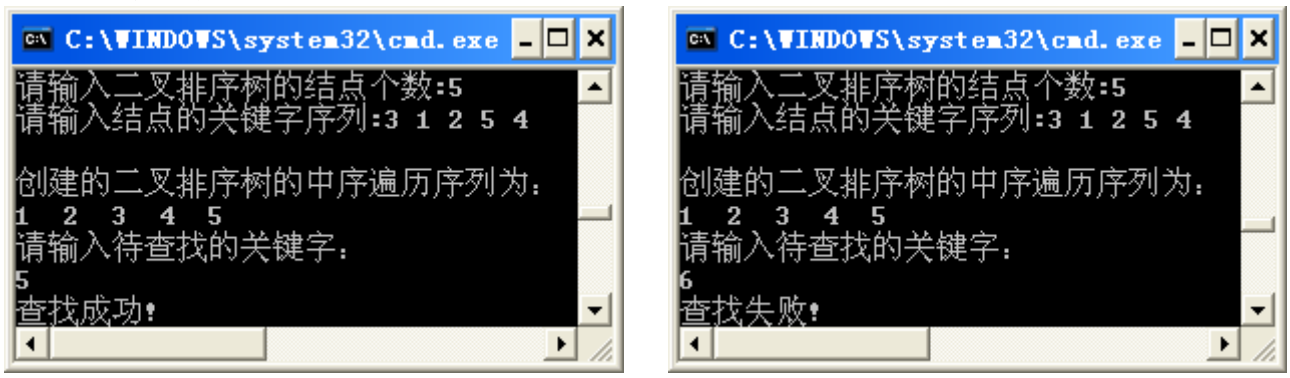


图 7-1 验证性实验运行结果

备注：以下设计性和应用性实验内容学生可根据自己的掌握程度或兴趣自行选择其一或其二完成。

七、设计性实验

编程判断一棵二叉树是否为二叉排序树。

1. 实验要求

⑴ 二叉树采用二叉链表作为存储结构，且树中结点的关键字均不相同。

⑵ 要输出最后的判断结果。

2. 核心算法分析

二叉排序树的二叉链表的结点类与验证性实验中的内容相同。

二叉排序树有一个重要的特点就是对它进行中根遍历得到的遍历序列一定是一个有序序列。根据这个特点，可以利用中根遍历的方法对二叉树进行判断，其中访问根结点的操作变成判断根结点的关键字值是否比它在中根遍历序列的前驱结点的关键字值更小，如果是，则此二叉树不是二叉排序树，如果不是，则继续对其右子树进行判断，与此同时记下此根结点的关键字值作为下一个要访问的根结点的前驱结点的关键字值。

为实现此操作，算法需引进两个变量 flag 和 lastkey。前者其初值为 true, 用于标志当前访问到的结点的关键字值是否比其中根遍历序列中的前驱结点的关键值更小，如果是，则其值为 false；后者用于记载下一个要访问的根结点的前驱结点的关键字值。

3. 核心算法描述

```
boolean flag = true;
int lastkey = 0;
boolean Is_BSTree(BiTreeNode T) {
    if (T.getLchild() != null && flag)
        Is_BSTree(T.getLchild());
    if (lastkey> T.getKey() )
        flag = false;     //与其中根序列的前驱相比较
   lastkey= T.getKey();
   if (T.getRchild() != null && flag) {
```

```
            Is_BSTree(T.getRchild());
        }
        return flag;
    }
```

八、应用性设计实验

编程设计一个简单的学生信息管理系统。每个学生的信息包括学号、姓名、性别、班级和电话等。

实验要求：

⑴ 采用二叉排序树的结构创建学生的信息表。

⑵ 能按照学号或姓名查找学生的信息。如果查找成功，则输出学生的所有信息，否则输入查找失败的提示信息。

实验 7： 图的操作实验

一、实验名称和性质

所属课程	数据结构
实验名称	图的操作
实验学时	2
实验性质	√验证 □综合 √设计
必做/选做	□必做 √选做

二、实验目的

1．掌握图的相关概念。

2．掌握用邻接矩阵和邻接表的方法描述图的存储结构。

3．掌握图的深度优先搜索和广度优先搜索遍历的方法及其计算机的实现。

4．理解最小生成树的有关算法

三、实验内容

1．用邻接表作为图的存储结构建立一个图，并对此图分别进行深度优先搜索和广度优先搜索遍历（验证性内容）。

2．用邻接矩阵作为图的存储结构建立一个网，并构造该网的最小生成树（设计性内容）。

3．校园导游程序的实现（应用性设计内容）。

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件： PC 软件： Win2000 系统及以上; VC 编译器

五、知识准备

前期要求掌握图与网的含义、图的邻接矩阵和邻接表的存储表示、图的深度优先搜索遍历和广度优先搜索遍历方法、最小生成树的概念及其构造算法。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）建立用邻接表表示的如图 10-1 所示的无向图 G,并输出其邻接表。

（2）对 G 图进行深度优先搜索和广度优先搜索遍历，并分别输出其遍历序列。

2. 实验相关原理：

图的两种常用的存储结构是邻接矩阵和邻接表，选用哪一种种存储结构，取决于具体的应用。要实现图的遍历操作则用邻接表作为存储结构比较方便。如果图 G 是一个具有 n 个顶点的无向图（或有向图），则图 G 的邻接表是由 n 个单链表所组成，且第 i(1≤i≤n)个链表中的结点是由与顶点 i 相关联的边所构成。每个结点由二个域组成，其中邻接点域（adjvex），指示与顶点 i 邻接的顶点在图中的位置；链域(nextarc)，指示与顶点 i 相邻接的另一个顶点。

n 个单链表的头指针通过按顺序存储方式进行存储，构成一个顺序表。表头结点含二个域，其中访问标志域（tag），存储该顶点访问标志值 0 或 1，0 代表未被访问，1 代表已被访问；链域（firstarc），指向后面单链表中第一个结点，也是指向 i 相邻接的第一个顶点。如图 10-1 的图所对应的邻接表存储结构如图 10-2 所示。

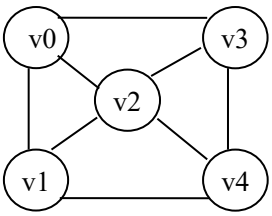


图 10-1 无向图

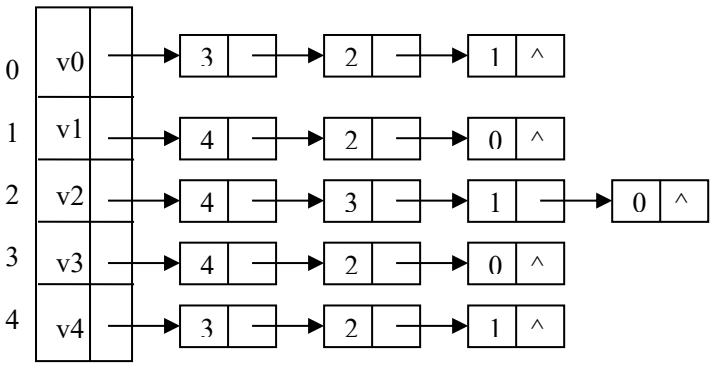


图 10-2 无向图 G 的邻接表

邻接表的存储结构可用以下类来进行描述：

(1) 图的邻接表存储表示中的弧结点类

```
class ArcNode {
      private int adjVex;// 该弧所指向的顶点位置
      private ArcNode nextArc;// 指向下一条弧
      public ArcNode() {
           this(-1,null);
      }
      public ArcNode(int adjVex) {
           this(adjVex, null);
      }
      public ArcNode(int adjVex, ArcNode nextArc) {
           this.adjVex = adjVex;
           this.nextArc = nextArc;
      }
     ……
}
```

(2) 图的邻接表存储表示中的顶点结点类

```
class VNode {
      private Object data;// 顶点信息
      private ArcNode firstArc;// 指向第一条依附于该顶点的弧
      public VNode() {
           this(null, null);
      }
```

```
        public VNode(Object data) {
            this(data, null);
        }
        public VNode(Object data, ArcNode firstArc) {
            this.data = data;
            this.firstArc = firstArc;
        }
        ……
    }
```

(3) 无向图的邻接表类

```
    class ALGraph  {
        private int vexNum, arcNum;// 图的当前顶点数和边数
        private VNode[] vexs;// 顶点
        public ALGraph() {
            this( 0, 0, null);
        }
        public ALGraph( int vexNum, int arcNum, VNode[] vexs) {
            this.vexNum = vexNum;
            this.arcNum = arcNum;
            this.vexs = vexs;
        }
        ……
    }
```

【核心算法提示】

（1）无向图的深度优先搜索遍历过程：首先访问出发顶点 V，然后选择一个与 V 相邻接且未被访问过的顶点 w，再从 w 开始进行深度优先搜索。此遍历的特点是尽可能先对纵深方向进行搜索。

（2）无向图的广度优先搜索遍历过程：首先访问出发点 V，接着访问 V 的所有邻接点 W1，W2，…，Wt，然后再依次访问与 W1，W2，…，Wt 邻接的所有未访问过的顶点，直到图中所有与初始出发点 Vi 有路径相通的顶点都已访问到为止。此遍历的特点是尽可能先对横向进行搜索。

【核心算法描述】

(1) 图的深度优先搜索遍历算法

```
public static void DFSTraverse(ALGraph G) throws Exception {// 对图 G 做深度优先遍历
        visited = new boolean[G.getVexNum()];
        for (int v = 0; v < G.getVexNum(); v++)
            // 访问标志数组初始化
            visited[v] = false;
```

```
        for (int v = 0; v < G.getVexNum(); v++)
            if (!visited[v])// 对尚未访问的顶点调用 DFS
                DFS(G, v);
}
// 从第 v 个顶点出发递归地深度优先遍历图 G
public static void DFS(ALGraph G, int v) throws Exception {
        visited[v] = true;
        System.out.print(G.getVex(v).toString() + " ");// 访问第 v 个顶点
        for (int w = G.firstAdjVex(v); w >= 0; w = G.nextAdjVex(v, w))
            if (!visited[w])// 对 v 的尚未访问的邻接顶点 w 递归调用 DFS
                DFS(G, w);
}
```

(2) 图的广度优先搜索遍历算法

```
public static void BFSTraverse(ALGraph G) throws Exception { // 对图 G 做广度优先遍历
        visited = new boolean[G.getVexNum()];// 访问标志数组
        for (int v = 0; v < G.getVexNum(); v++)
            // 访问标志数组初始化
            visited[v] = false;
        for (int v = 0; v < G.getVexNum(); v++)
            if (!visited[v]) // v 尚未访问
                BFS(G, v);
}
// 从第 v 个顶点出发递归地广度优先遍历图 G
private static void BFS(ALGraph G, int v) throws Exception {
        visited[v] = true;
        System.out.print(G.getVex(v).toString() + " ");
        LinkQueue Q = new LinkQueue();// 辅助队列 Q
        Q.offer(v);// v 入队列
        while (!Q.isEmpty()) {
            int u = (Integer) Q.poll();// 队头元素出队列并赋值给 u
            for (int w = G.firstAdjVex(u); w >= 0; w = G.nextAdjVex(u, w))
                if (!visited[w]) {// w 为 u 的尚未访问的邻接顶点
                    visited[w] = true;
                    System.out.print(G.getVex(w).toString() + " ");
                    Q.offer(w);
                }
        }
```

```
}
```

3．源程序代码参考

```
import ch03.LinkQueue;
//图的邻接表存储表示中的弧结点类
class ArcNode {
    private int adjVex;// 该弧所指向的顶点位置
    private ArcNode nextArc;// 指向下一条弧
    public ArcNode() {
        this(-1,null);
    }
    public ArcNode(int adjVex) {
        this(adjVex, null);
    }
    public ArcNode(int adjVex, ArcNode nextArc) {
        this.adjVex = adjVex;
        this.nextArc = nextArc;
    }
    public ArcNode getNextArc() {
        return nextArc;
    }
    public int getAdjVex() {
        return adjVex;
    }
    public void setAdjVex(int adjVex) {
        this.adjVex = adjVex;
    }
    public void setNextArc(ArcNode nextArc) {
        this.nextArc = nextArc;
    }
}

//图的邻接表存储表示中的顶点结点类
 class VNode {
    private Object data;// 顶点信息
    private ArcNode firstArc;// 指向第一条依附于该顶点的弧
    public VNode() {
        this(null, null);
    }
```

```
    public VNode(Object data) {
        this(data, null);
    }
    public VNode(Object data, ArcNode firstArc) {
        this.data = data;
        this.firstArc = firstArc;
    }
    public Object getData() {
        return data;
    }
    public ArcNode getFirstArc() {
        return firstArc;
    }
    public void setData(Object data) {
        this.data = data;
    }
    public void setFirstArc(ArcNode firstArc) {
        this.firstArc = firstArc;
    }
}
//无向图的邻接表类
 class ALGraph   {
    private int vexNum, arcNum;// 图的当前顶点数和边数
    private VNode[] vexs;// 顶点
     public ALGraph() {
        this( 0, 0, null);
    }
    public ALGraph( int vexNum, int arcNum, VNode[] vexs) {
        this.vexNum = vexNum;
        this.arcNum = arcNum;
        this.vexs = vexs;
    }
   public int getArcNum() {
        return arcNum;
    }
   public void setArcNum(int arcNum) {
        this.arcNum = arcNum;
    }
```

```
public int getVexNum() {
	return vexNum;
}
public void setVexNum(int vexNum) {
	this.vexNum = vexNum;
}
public VNode[] getVexs() {
	return vexs;
}
public void setVexs(VNode[] vexs) {
	this.vexs = vexs;
}
 // 返回 v 表示结点的值， 0 <= v < vexNum
 public Object getVex(int v) throws Exception {
	if (v < 0 && v >= vexNum)
		throw new Exception("第" + v + "个顶点不存在!");
	return vexs[v].getData();
 }
// 返回 v 的第一个邻接点，若 v 没有邻接点则返回-1， 0 <= v < vexnum
 public int firstAdjVex(int v) throws Exception {
	if (v < 0 && v >= vexNum)
		throw new Exception("第" + v + "个顶点不存在!");
	VNode vex = vexs[v];
	if (vex.getFirstArc() != null)
		return vex.getFirstArc().getAdjVex();
	else
		return -1;
 }
// 返回 v 相对于 w 的下一个邻接点，若 w 是 v 的最后一个邻接点，则返回-1，其中 0≤v, w<vexNum
 public int nextAdjVex(int v, int w) throws Exception {
	if (v < 0 && v >= vexNum)
		throw new Exception("第" + v + "个顶点不存在!");
	VNode vex = vexs[v];
	ArcNode arcvw = null;
	for (ArcNode arc = vex.getFirstArc(); arc != null; arc = arc
			.getNextArc())
		if (arc.getAdjVex() == w) {
			arcvw = arc;
```

```
                    break;
                }
            if (arcvw != null && arcvw.getNextArc() != null)
                return arcvw.getNextArc().getAdjVex();
            else
                return -1;
        }
//邻接表的输出
public void displayALGraph(){
            for (int i=0; i<vexs.length;i++){
                System.out.print(vexs[i].getData().toString()+":");
                ArcNode p=vexs[i].getFirstArc();
                while(p!=null){
                    System.out.print(p.getAdjVex()+"    ");
                    p=p.getNextArc();
                }
               System.out.println();
            }
       }
  }
//图的遍历类
  class TraverserALGraph {
        private static boolean[] visited;// 访问标志数组

        // 对图 G 做深度优先遍历
        public static void DFSTraverse(ALGraph G) throws Exception {
            visited = new boolean[G.getVexNum()];
            for (int v = 0; v < G.getVexNum(); v++)
                // 访问标志数组初始化
                visited[v] = false;
            for (int v = 0; v < G.getVexNum(); v++)
                if (!visited[v])// 对尚未访问的顶点调用 DFS
                    DFS(G, v);
        }

        // 从第 v 个顶点出发递归地深度优先遍历图 G
        public static void DFS(ALGraph G, int v) throws Exception {
            visited[v] = true;
```

```
        System.out.print(G.getVex(v).toString() + " ");// 访问第 v 个顶点
        for (int w = G.firstAdjVex(v); w >= 0; w = G.nextAdjVex(v, w))
            if (!visited[w])// 对 v 的尚未访问的邻接顶点 w 递归调用 DFS
                DFS(G, w);
    }

    // 对图 G 做广度优先遍历
    public static void BFSTraverse(ALGraph G) throws Exception {
        visited = new boolean[G.getVexNum()];// 访问标志数组
        for (int v = 0; v < G.getVexNum(); v++)
            // 访问标志数组初始化
            visited[v] = false;
        for (int v = 0; v < G.getVexNum(); v++)
            if (!visited[v]) // v 尚未访问
                BFS(G, v);
    }

    private static void BFS(ALGraph G, int v) throws Exception {
        visited[v] = true;
        System.out.print(G.getVex(v).toString() + " ");
        LinkQueue Q = new LinkQueue();// 辅助队列 Q
        Q.offer(v);// v 入队列
        while (!Q.isEmpty()) {
            int u = (Integer) Q.poll();// 队头元素出队列并赋值给 u
            for (int w = G.firstAdjVex(u); w >= 0; w = G.nextAdjVex(u, w))
                if (!visited[w]) {// w 为 u 的尚未访问的邻接顶点
                    visited[w] = true;
                    System.out.print(G.getVex(w).toString() + " ");
                    Q.offer(w);
                }
        }
    }
}
//测试类
public class SY10_Graph {
    public static ALGraph generateALGraph() {   //创建如图 10-1 所示的无向图的邻接表
        ArcNode v01 = new ArcNode(1);
        ArcNode v02 = new ArcNode(2, v01);
```

```
        ArcNode v03 = new ArcNode(3, v02);
        VNode v0 = new VNode("v0", v03);

        ArcNode v10 = new ArcNode(0);
        ArcNode v12 = new ArcNode(2,v10);
        ArcNode v14 = new ArcNode(4,v12);
        VNode v1 = new VNode("v1", v14);

        ArcNode v20 = new ArcNode(0);
        ArcNode v21 = new ArcNode(1,v20);
        ArcNode v23 = new ArcNode(3,v21);
        ArcNode v24 = new ArcNode(4,v23);
        VNode v2 = new VNode("v2", v24);

        ArcNode v30 = new ArcNode(0);
        ArcNode v32 = new ArcNode(2,v30);
        ArcNode v34 = new ArcNode(4,v32);
       VNode v3 = new VNode("v3", v34);

        ArcNode v41 = new ArcNode(1);
        ArcNode v42 = new ArcNode(2,v41);
        ArcNode v43 = new ArcNode(3,v42);
        VNode v4 = new VNode("v4", v43);

        VNode[] vexs = { v0, v1, v2, v3, v4 };
        ALGraph G = new ALGraph( 5, 16, vexs);
        return G;
}

public static void main(String[] args) throws Exception {
        ALGraph G = generateALGraph();

        System.out.print("无向图的广度优先遍历序列为：");
      TraverserALGraph.BFSTraverse(G);
      System.out.println();
      System.out.print("无向图的深度优先遍历序列为：");
      TraverserALGraph.DFSTraverse(G);
      System.out.println();
```

```
        }
    }
```

4．运行结果参考如图 10-3 所示：

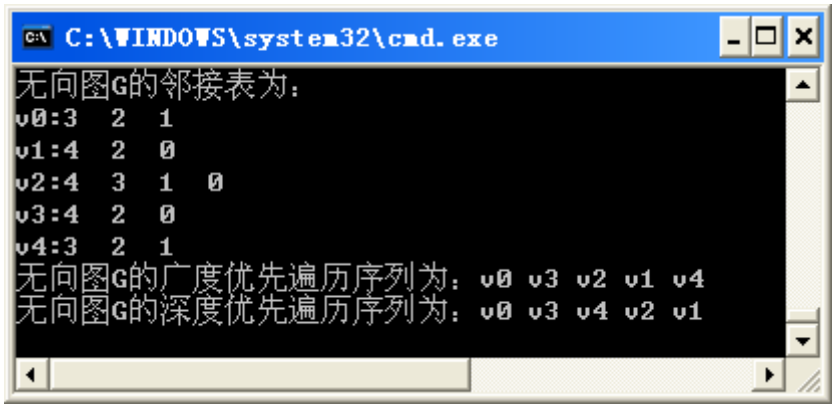


图 10-3　验证性实验运行结果

备注：以下设计性和应用性实验内容学生可根据自己的掌握程度或兴趣自行选择其一或其二完成。

七、设计性实验

用邻接矩阵作为图的存储结构建立一个网，并构造该网的最小生成树。

1．实验要求

(1) 输入无向网的顶点数、边数及各条边的顶点序号对和边上的权值，建立用邻接矩阵表示的无向网。

(2) 构造该无向网的最小生成树。

2．核心算法分析

如果连通图是一个网络，生成树的各边权值之和称为这棵生成树的代价，称该网络中所有生成树中权值最小的生成树为最小代价生成树，简称为最小生成树。常见的构造最小生成树的算法有普里姆（Prim）算法和克鲁斯卡尔(Kruskal)算法。下面只说明普里姆算法思想及实现方法。

普里姆算法的基本思想：假设网 G=（V，E）是连通的，T=（U，TE）为要构造的一棵最小生成树，其中 U 是 G 上最小生成树顶点的集合，TE 是 G 上最小生成树中边的集合。开始时 U={u0}，TE={}。重复进行如下操作：在所有 u∈U，v∈V-U 的边（u,v）∈E 中，选择一条权最小的边（u,v）并入 TE 中，同时将 v 并入 U，直到 U=V 为止。这时产生的 TE 中必有 n-1 条边，T=（U，TE）是 G 的一棵最小生成树。如下图 10-4 所示：

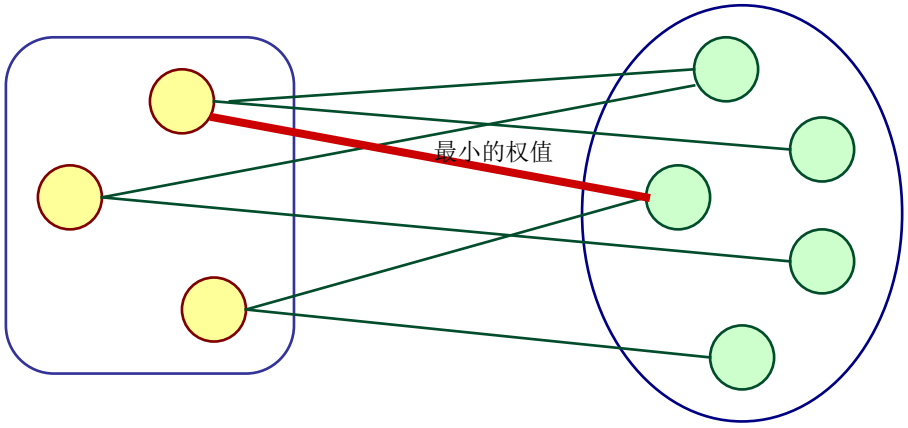


图 10-4

在实现普里姆算法时，采用邻接矩阵 cost 表示给定的无向网，矩阵元素定义为：

$$cost=\begin{cases} w_{ij} & (i,j)\in TE,\ i\neq j,\ w_{ij}\text{ 为 }(i,j)\text{ 上的权} \\ 0 & i=j \\ \infty & \text{否则} \end{cases}$$

为实现普里姆算法，需要此入一个辅助数组 closedge，用于记录从 U 到 V-U 具有最小代价的边（u*,v*）。针对每一个顶点 vi∈V-U，在辅助数组中存在一个相应分量 closedge[i], 它包括两个成员 adjvex 和 lowcost，分别用于存放顶点和权值。所有的顶点 closedge[i].adjvex 都已在 U 中。若 closedge[k].lowcost=0,则表明顶点 k 在 U 中；若 0＜closedge[k].lowcost＜∞，则 j∈V-U,且(closedge[j].adjvex,j)是与顶点 j 邻接的且两邻接顶点分别在 U 和 V-U 的所有边中权值最小的边，其最小权值就是 closedge[j].lowcost。若 closedge[k].lowcost=∞，则表示 closedge[j].closest 与顶点 j 之间没有边，算法中用 INFINITY = Integer.MAX_VALUE 表示∞。

3. 核心算法描述

无向网的邻接表存储表示类描述如下：

```
public class MGraph implements IGraph {
    public final static int INFINITY = Integer.MAX_VALUE;
    private int vexNum, arcNum;// 图的当前顶点数和边数
    private Object[] vexs;// 顶点
    private int[][] arcs;// 邻接矩阵
    ……
}
```

（1）创建无向网的算法

```
private void createUDN() {
    Scanner sc = new Scanner(System.in);
    System.out.println("请分别输入图的顶点数、图的边数:");
    vexNum = sc.nextInt();
    arcNum = sc.nextInt();
    vexs = new Object[vexNum];
```

```
        System.out.println("请分别输入图的各个顶点:");
        for (int v = 0; v < vexNum; v++)
            // 构造顶点向量
            vexs[v] = sc.next();
        arcs = new int[vexNum][vexNum];
        for (int v = 0; v < vexNum; v++)
            // 初始化邻接矩阵
            for (int u = 0; u < vexNum; u++)
                arcs[v][u] = INFINITY;
        System.out.println("请输入各个边的顶点及其权值:");
        for (int k = 0; k < arcNum; k++) {
            int v = locateVex(sc.next());
            int u = locateVex(sc.next());
            arcs[v][u] = arcs[u][v] = sc.nextInt();
        }
    }
```

(2) 用普里姆算法构造最小生成树的类

```
public class MiniSpanTree_PRIM {
 // 内部类辅助记录从顶点 U 到 V-U 的代价最小的边
 private class CloseEdge {
        Object adjVex;
        int lowCost;
        public CloseEdge(Object adjVex, int lowCost) {
            this.adjVex = adjVex;
            this.lowCost = lowCost;
        }
 }
 // 用普里姆算法从第 u 个顶点出发构造网 G 的最小生成树 T，返回由生成树边组成的二维数组
 public Object[][] PRIM(MGraph G, Object u) throws Exception {
        Object[][] tree = new Object[G.getVexNum() - 1][2];
        int count = 0;
        CloseEdge[] closeEdge = new CloseEdge[G.getVexNum()];
        int k = G.locateVex(u);
        for (int j = 0; j < G.getVexNum(); j++)// 辅助数组初始化
            if (j != k)
                closeEdge[j] = new CloseEdge(u, G.getArcs()[k][j]);
        closeEdge[k] = new CloseEdge(u, 0);// 初始，U={u}
```

```
        for (int i = 1; i < G.getVexNum(); i++) {// 选择其余 G.vexnum - 1 个顶点
            k = getMinMum(closeEdge);// 求出 T 的下一个结点：第 k 个顶点
            tree[count][0] = closeEdge[k].adjVex;// 生成树的边放入数组
            tree[count][1] = G.getVexs()[k];
            count++;
            closeEdge[k].lowCost = 0;// 第 k 个顶点并入 U 集
            for (int j = 0; j < G.getVexNum(); j++)//新顶点并入 U 后重新选择最小边
                if (G.getArcs()[k][j] < closeEdge[j].lowCost)
                    closeEdge[j] = new CloseEdge(G.getVex(k), G.getArcs()[k][j]);
        }
        return tree;
    }
    //在 closeEdge 中选出 lowCost 最小且不为 0 的顶点
    private int getMinMum(CloseEdge[] closeEdge) {
        int min = Integer.MAX_VALUE;
        int v = -1;
        for (int i = 0; i < closeEdge.length; i++)
            if (closeEdge[i].lowCost != 0 && closeEdge[i].lowCost < min){
                min = closeEdge[i].lowCost;
                v = i;
            }
        return v;
    }
}
```

八、应用性设计实验

校园导游程序的实现。

实验要求：

用无向图表示你所在学校的校园景点平面图，图中顶点表示主要景点，存放景点的编号、名称、简介等信息，图中的边表示景点间的道路，存放路径长度等信息。要求实现以下功能：

⑴ 查询各景点的相关信息。

⑵ 查询图中任意两个景点间的最短路径。

⑶ 查询图中任意两个景点间的所有路径。

实验 8:　静态表的查找操作实验

一、实验名称和性质

所属课程	数据结构与算法
实验名称	静态表的查找操作
实验学时	2
实验性质	√验证 □综合 √设计
必做/选做	√必做 □选做

二、实验目的

1．掌握顺序查找操作的算法实现。

2．掌握二分查找操作的算法实现及实现该查找的前提。

3．掌握索引查找操作的算法实现。

三、实验内容

1．建立顺序查找表，并在此查找表上实现顺序查找操作（验证性内容）。

2．建立有序顺序查找表，并在此查找表上实现二分查找操作（验证性内容）。

3．建立索引查找表，并在此查找表上实现索引查找操作（设计性内容）。

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件：　PC　软件：　Win2000 系统及以上;　VC 编译器

五、知识准备

前期要求掌握查找的含义和顺序查找、二分查找及索引查找操作的方法。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）根据输入的查找表的表长 n 和 n 个关键字值，建立顺序查找表，并在此查找表中用顺序查找方法查找给定关键值的记录，最后输出查找结果。

（2）根据输入的查找表的表长 n 和 n 个按升排列的关键字值，建立有序顺序查找表，并在此查找表中用二分查找方法查找给定关键值的记录，最后输出查找结果。

（3）主程序中要求设计一个菜单，允许用户通过菜单来多次选择执行哪一种查找操作。

2. 实验相关原理：

查找表分别静态查找表和动态查找表两种，其中只能做引用操作的查找表称为静态查找表。

静态查找表采用顺序存储结构，待查找的记录类可描述如下：

```
public class RecordNode {
    private Comparable key;       //关键字
    private Object element;       //数据元素
    ……
```

```
}
```

待排序的顺序表类描述如下：

```
public class SeqList {
    private RecordNode[] r;    //顺序表记录结点数组
    private int curlen;        //顺序表长度，即记录个数
    // 顺序表的构造方法，构造一个存储空间容量为 maxSize 的顺序表
    public SeqList(int maxSize) {
        this.r = new RecordNode[maxSize];  // 为顺序表分配 maxSize 个存储单元
        this.curlen = 0;                   // 置顺序表的当前长度为 0
    }
    ……
}
```

【核心算法提示】

查找操作是根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素或记录的过程。若查找表中存在这样一个记录，则称“查找成功”。查找结果给出整个记录的信息，或指示该记录在查找表中的位置；若在查找表中不存在这样的记录，则称“查找不成功”。查找结果给出“空记录”或“空指针”。

（1）顺序查找操作的基本步骤：从表中第一条记录开始，顺序扫描查找表，依次将扫描到的结点关键字值与给定值 key 进行比较，若当前扫描到的结点关键字值与 key 相等，则查找成功，返回记录下标；若扫描到最后一条记录，仍未找到关键字值等于 key 的记录，则查找失败，返回-1。

（2）二分查找也叫折半查找，这种查找要求查找表必须是有序顺序表。其查找操作的基本步骤：首先取出表中的中间元素，若其关键字值等于给定值 key，则查找成功，返回记录下标；否则以中间元素为分界点，将查找表分成两个子表，并判断所查的 key 值所在的子表是前部分，还是后部分，再重复上述步骤直到找到关键字值为 key 的元素或子表长度为 0，如果子表长度为 0，表示查找失败，返回-1。

【核心算法描述】

假设查找表中从 r[1]到 r[n]存放 n 条元素，第 0 号存储单元不存储数据元素。

⑴在顺序查找表上的顺序查找算法

```
public int seqSearch(Comparable key) {
        int i = 1, n = length();
        while (i < n+1 && r[i].getKey().compareTo(key) != 0) {
            i++;
        }
        if (i < n+1) {    //查找成功则返回该元素的下标 i，否则返回-1
            return i;
        } else {
            return -1;
        }
    }
```

(2) 在有序顺序表上的二分查找算法

```
public int binarySearch(Comparable key) {
        if (length() > 0) {
            int low = 1, high = length() ; //查找范围的下界和上界
            while (low <= high) {
                int mid = (low + high) / 2;   //中间位置，当前比较元素位置
                //    System.out.print(r[mid].getKey() + "? ");
                if (r[mid].getKey().compareTo(key) == 0) {
                    return mid;                                  //查找成功
                } else if (r[mid].getKey().compareTo(key) > 0) { //给定值更小
                    high = mid - 1;     //查找范围缩小到前半段
                } else {
                    low = mid + 1;      //查找范围缩小到后半段
                }
            }

        }
       return -1;  //查找不成功
    }
```

3．源程序代码参考

```
import java.util.Scanner;
class RecordNode {//记录结点类
    private Comparable key;         //关键字
    private Object element;         //数据元素
    public Object getElement() {
        return element;
    }
    public void setElement(Object element) {
        this.element = element;
    }
    public Comparable getKey() {
        return key;
    }
    public void setKey(Comparable key) {
        this.key = key;
    }
    public RecordNode(Comparable key) {   //构造方法 1
        this.key = key;
    }
    public RecordNode(Comparable key, Object element) {   //构造方法 2
        this.key = key;
```

```
        this.element = element;
    }
    public String toString() { //覆盖 toString()方法
        return "[" + key + "," + element + "]";
    }
}
 class KeyType implements Comparable<KeyType> {   //记录关键字类
    private int key;     //关键字
    public KeyType() {
    }
    public KeyType(int key) {
        this.key = key;
    }
    public int getKey() {
        return key;
    }
    public void setKey(int key) {
        this.key = key;
    }
     public int compareTo(KeyType another) {   //覆盖 Comparable 接口中比较关键字大小的方法
        int thisVal = this.key;
        int anotherVal = another.key;
        return (thisVal < anotherVal ? -1 : (thisVal == anotherVal ? 0 : 1));
    }
}//
class SeqList {//顺序查找表类
    private RecordNode[] r;       //顺序表记录结点数组
    private int curlen;              //顺序表长度,即记录个数
    // 顺序表的构造方法：构造一个存储空间容量为 maxSize 的顺序表
    public SeqList(int maxSize) {
        this.r = new RecordNode[maxSize];   // 为顺序表分配 maxSize 个存储单元
        this.curlen = 0;                                  // 置顺序表的当前长度为 0
    }
    // 求顺序表中的数据元素个数并由函数返回其值
    public int length() {
        return curlen; // 返回顺序表的当前长度
    }
    public int getCurlen() {
```

```
        return curlen;
    }
    public void setCurlen(int curlen) {
        this.curlen = curlen;
    }
    /*  在当前顺序表的第 i 个结点之前插入一个 RecordNode 类型的结点 x，其中 i 取值范围为：0≤i≤curlen。如果 i 值不在此范围则抛出异常,当 i=0 时表示在表头插入一个数据元素 x, 当 i=curlen 时表示在表尾插入一个数据元素 x */
     public void insert(int i, RecordNode x) throws Exception {
        if (curlen == r.length-1) {       // 判断顺序表是否已满,0 号存储单元不存放元素
            throw new Exception("顺序表已满");
        }
        if (i < 0 || i > curlen+1) {    // i 小于 0 或者大于表长
            throw new Exception("插入位置不合理");
        }
        for (int j = curlen; j > i; j--) {
            r[j+1] = r[j ];     // 插入位置及之后的元素后移
        }
        r[i] = x;     // 插入 x
        this.curlen++;   // 表长度增 1
     }
    /*顺序查找：从顺序表 r[1]到 r[n]的 n 个元素中顺序查找出关键字为 key 的记录，若查找成功返回其下标，否则返回-1*/
    public int seqSearch(Comparable key) {
        int i = 1, n = length();
        while (i < n+1 && r[i].getKey().compareTo(key) != 0) {
            i++;
        }
        if (i < n+1) {     //查找成功则返回该元素的下标 i，否则返回-1
            return i;
        } else {
            return -1;
        }
    }
   /*二分查找：数组元素已按升序排列，从顺序表 r[1]到 r[n]的 n 个元素中查找出关键字为 key 的元素，若查找成功返回元素下标，否则返回-1 */
    public int binarySearch(Comparable key) {
        if (length() > 0) {
```

```
            int low = 1, high = length() ; //查找范围的下界和上界
            while (low <= high) {
                int mid = (low + high) / 2;     //中间位置，当前比较元素位置
                //       System.out.print(r[mid].getKey() + "? ");
                if (r[mid].getKey().compareTo(key) == 0) {
                    return mid;                                          //查找成功
                } else if (r[mid].getKey().compareTo(key) > 0) { //给定值更小
                    high = mid - 1;          //查找范围缩小到前半段
                } else {
                    low = mid + 1;           //查找范围缩小到后半段
                }
            }
        }
        return -1;   //查找不成功
    }
}
//测试类
public class SY6_search {
    static SeqList ST = null;
    //建立待查找的顺序表
     public static void createSearchList() throws Exception {
        ST=new SeqList(20);
        Scanner sc=new Scanner(System.in);
        System.out.print("请输入查找表的表长:");
        int n=sc.nextInt();
        KeyType[] k= new KeyType[n];
        System.out.print("请输入查找表中的关键字序列:");
        for (int i = 0; i < n; i++) {    //输入关键字序列
            k[i] = new KeyType(sc.nextInt());
        }
        for(int i=1;i<n+1;i++){    //创建记录顺序表,0 号存储单元不存放元素
            RecordNode r = new RecordNode(k[i-1]);
            ST.insert(i, r);
        }
    }
     public static void main(String[] args) throws Exception{
        Scanner sc=new Scanner(System.in);
        KeyType key1,key2;
```

```
        while(true){
            System.out.println(" 1--顺序查找      2--二分查找      3--退出");
            System.out.print("请输入选择(1-3):");
            int i=sc.nextInt();
            switch(i){
                case 1: System.out.println("创建顺序查找表");
                        createSearchList();
                        System.out.print("请输入两个待查找的关键字:");
                        key1=new KeyType(sc.nextInt());
                        key2=new KeyType(sc.nextInt());
                        System.out.println("seqSearch(" + key1.getKey() + ")=" + ST.seqSearch(key1));
                        System.out.println("seqSearch(" + key2.getKey() + ")=" + ST.seqSearch(key2));
                        break;
                case 2: System.out.println("创建有序的顺序查找表");
                        createSearchList();
                        System.out.print("请输入两个待查找的关键字:");
                        key1=new KeyType(sc.nextInt());
                        key2=new KeyType(sc.nextInt());
                        System.out.println("binarySearch("+key1.getKey()+")="+ST.binarySearch(key1));
                        System.out.println("binarySearch("+key2.getKey()+")="+ST.binarySearch(key2));
                        break;
                case 3:return;
            }
        }
    }
}
```

4．运行结果参考如图 6-1 所示。

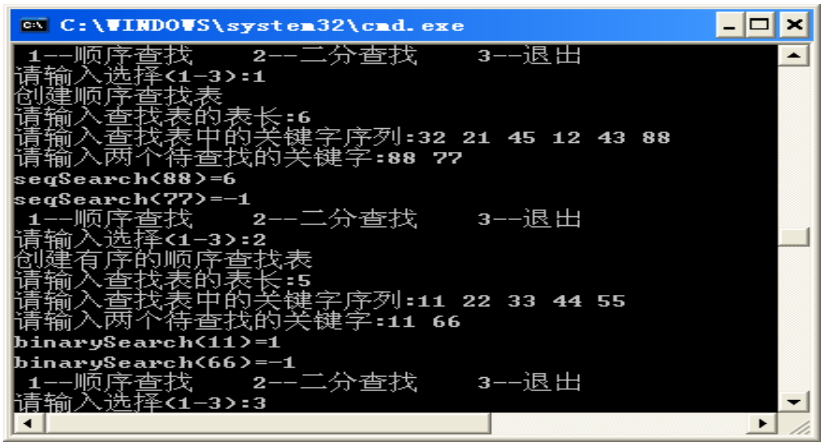


图 6-1 验证性实验运行结果

七、设计性实验

1．实验要求

编程实现如下功能：

（1）建立索引查找表

（2）利用索引查找确定给定记录在索引查找表中的块号和在块中的位置。

2．核心算法分析

索引查找表有索引表和块表两部分所构成，其中索引表存储的是各块记录中的最大关键字值和各块的起始存储地址，用顺序存储结构，各块的起始存储地址的初始值置为空指针；而块表中存储的是查找表中的所有记录并且按块有序，用链式存储或顺序存储结构，在此用链式存储结构。则索引查找表的存储结构图如 6-2 所示：

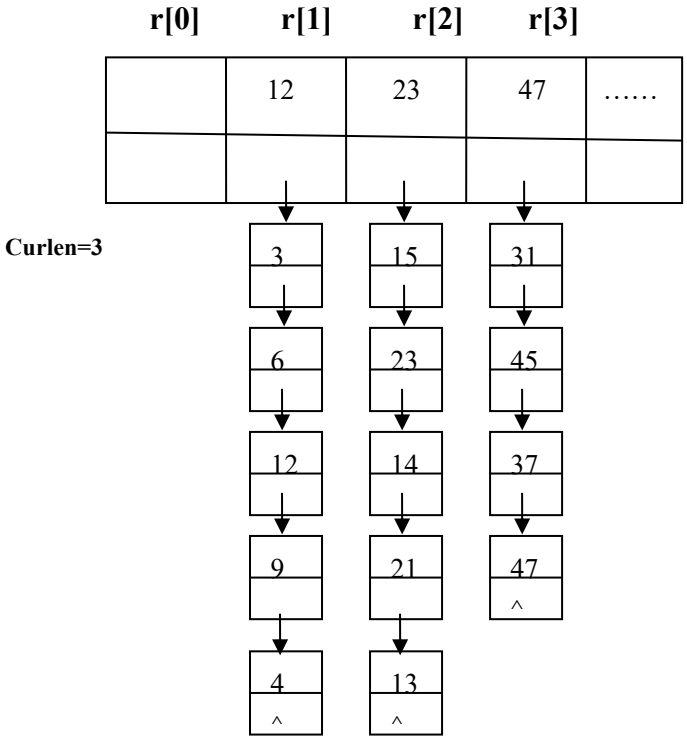


图 6-2　索引查找表的存储结构示意图

如图 6-2 所示的存储结构可用如下类进行描述：

(1) 块链中的结点类

```
class Node {
    private int key; //  存放记录关键字值
    private Node next; //  后继结点的引用
    ……
}
```

(2) 索引表中的结点类

```
class IndexNode {
    private int    maxKey;          //块中的最大关键字
```

```
    private Node    head;          //指向块中的第一个结点指针
    ……
}
```

(3) 整个索引查找表的类

```
class IndexList {

    private IndexNode[] r;       //顺序表记录结点数组
    private int curlen;          //顺序表长度,即记录个数
    ……
}
```

由于在索引查找表中，索引表是按关键字从小到大排列的有序顺序表，块链是一个无序链表，所有索引表的查找过程可归纳为：

（1）在索引表中用二分查找确定待查找的记录所在的块号；

（2）沿着块链指针用顺序查找的方法确定待查找的记录在块链中的存储位置。

如果查找成功，则输出待查找记录在索引表中的块号和在块链中的存储地址；如果查找失败，则输出“没有找到”的信息。

3．核心算法描述

(1) 在索引表中的二分查找算法

```
public int findblock(IndexList ST, int key){
/*用二分查找法在索引查找表 ST 的索引表中确定关键字值为 key 的待查找记录所在的块号,函数并返回其
  块号值*/
    int low,high,mid;
    low=1;high=ST.getCurlen();
    while(low<=high){
        mid=(low+high)/2;
        if (((ST.getR())[mid].getMaxKey())>key)
            high=mid-1;
        else
            low=mid+1;
    }
    return high+1;
}
```

(2) 在索引表中的二分查找算法

```
public Node    findrecord (IndexList ST,int key){
/*用顺序查找法在索引查找表的块链中确定关键字值为 key 的待查找记录的存储位置，如查找成功则函数
  返回其地址值，否则返回空指针*/
    int Bno=findblock(ST,key);   //调用函数，确定待查记录所在块号
    Node p=(ST.getR())[Bno].getHead();   //取到待查记录所在块链的头指针
```

```
    while (p!=null&&p.getKey()!=key)   //顺着链指针依次查找
        p=p.getNext();
    return p;     //返回查找结果
}
```

实验 9:　哈希表的查找操作实验

一、实验名称和性质

所属课程	数据结构与算法
实验名称	哈希表的查找操作
实验学时	2
实验性质	√验证 □综合 √设计
必做/选做	□必做 √选做

二、实验目的

1．掌握哈希表、哈希函数与哈希冲突的概念。

2．掌握哈希表的构造方法及其计算机的表示与实现。

3．掌握哈希表查找算法的实现。

三、实验内容

1．以开放地址法中的线性探测再散列法处理冲突，实现哈希表的建立、查找和插入操作（验证性内容）。

2．以链地址法，也叫拉链法处理冲突，实现哈希表的建立，查找和插入操作（设计性内容）。

3．用哈希查找法实现班级信息的查找（应用性设计内容）。

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件：　PC　软件：　Win2000 系统及以上;　VC 编译器

五、知识准备

前期要求掌握：

1．哈希函数、哈希地址、哈希表和冲突的概念。

2．哈希函数的构造方法，特别是除留余数法。

3．用开放地址法中的线性探测再散列法和链地址法解决冲突的原理。

4．在哈希函数和解决冲突的方法确定的情况下，哈希表的构造方法。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）设哈希表长为 20，用除留余数法构造一个哈希函数。

（2）输入哈希表中记录的个数 n(n<=20）和各记录的关键字值，然后以开放地址法中的线性探测再散列法作为解决冲突的方法，建立一个开放地址哈希表，并输出已经建立的哈希表。

（3）输入一个待查找记录的关键字 key，完成开放地址哈希表的查找操作，如果查找成功，则函数返回查找到的记录在哈希表中的位置值，否则给出查找失败的提示信息。

2. 实验相关原理：

哈希表查找是一各基于尽可能不通过比较操作而直接得到记录的存储位置的相法而提出的一种特殊查找技术。哈希表的构造方法是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，如此构造所得的查找表称之为“哈希表”。如果用开放地址法处理冲突而构造的查找表称为“开放地址哈希表”；如果用链地址法处理冲突而构造的查找表称为“链地址哈希表”。

开放定地址法解决冲突，形成下一个地址的形式是：Hi=(h(key) +di) % M ,i=1,2,…,k (k<=m-1,m 为表长)。其中 h(key) 为哈希函数，M 为某个正整数，di 为增量序列。线性探测再散列法规定 di=1,2,…,m (m 为表长)。

开放地址哈希查找表的记录类描述如下：

```
class RecordNode {
    private Comparable key;      //关键字
    private Object element;      //数据元素
     ……
}
```

待查找的哈希表类描述如下：

```
class HashTable {
    private RecordNode[] table;    //顺序表记录结点数组
    // 哈希表的构造方法，构造一个存储空间容量为 maxSize 的哈希表
    public HashTable (int maxSize) {
        this.r = new RecordNode[maxSize];  // 为哈希表分配 maxSize 个存储单元
     }
    ……
}
```

【核心算法提示】

（1） 用除留余数法构造哈希函数的基本思想：除留余数法是用关键字 key 除以某个正整数 M，所得的余数作为哈希地址的方法。对应的哈希函数为：h(key)=key % M，一般情况下 M 的取值为不大于表长的质数，所以根据实验要求，M 可取值为 19。

（2） 开放地址哈希表查找的基本思想：将待查找记录中的关键字值 key 为自变量，通过哈希函数 h,计算出 h(key)哈希地址，再将哈希表中对应位置上记录关键字值与 key 进行比较，如果相等则查找成功，否则，则按线性探测再散列法去比较下一存储位置的记录，直到查找到该记录或查找到下一存储位置为空或探测次数为表长次为止，要求返回查找到的记录在表中的位置或查找不成功时空存储空间的位置。

（3） 开放地址哈希表插入操作的基本思想：先调用哈希表查找函数查找以 key 为关键字值的记录是否存在，如果不存在，则将待插入的记录关键字值 key 为自变量，通过哈希函数 h,计算出 h(key)哈希地址，再去考察哈希表中对应存储位置是否为空，如果为空则将该待插入记录插入到此位置上；如果不为空，则按线性探测再散列法去寻求下一个探测地址，直到寻求到一个空的存储空间为止，再将待插入的记录插入到寻求到的空的存储空间中。

（4） 建立开放地址哈希表的基本思想：调用开放地址哈希表的插入操作函数依次将 n 个

记录插入到哈希表中即可。

【核心算法描述】

(1) 开放地址哈希表的查找操作算法

```
// 在开放地址哈希表中查找关键字值为 key 的记录,若哈希表满则返回 null,否则返回关键字为 key 或为 0 的记录结点
public RecordNode hashSearch(int key ){
    int i=hash(key);       //求哈希地址
    int j=0;
    while ((table[i].getKey().compareTo(0)!=0) && (table[i].getKey().compareTo(key)!=0) && (j< table.length)){     //该位置中不为空并且关键字与 key 不相等
        j++;
        i=(i+j)%20;   //用线性探测再散列法求得下一探测地址
    }     //i 指示查找到的记录在表中的存储位置或指示插入位置
    if (j>=table.length) {   //如果表已经为满
        System.out.println("哈希表已满");
        return null;
    }
    else     return table[i];
}
```

(2) 开放地址哈希表的插入操作算法

```
//查找不成功且表不满时，插入关键字值为 key 的记录到开放地址哈希表中
public void hashInsert( int key){
    RecordNode p=hashSearch(key);
    if (p.getKey().compareTo(0)==0)
        p.setKey(key);       //插入
    else
        System.out.println(" 此关键字记录已存在或哈希表已满");
}
```

3．源程序代码参考

```
//记录结点类
class RecordNode {
    private Comparable key;       //关键字
    private Object element;       //数据元素
    public Object getElement() {
        return element;
    }
    public void setElement(Object element) {
```

```
        this.element = element;
    }
    public Comparable getKey() {
        return key;
    }
    public void setKey(Comparable key) {
        this.key = key;
    }
    public RecordNode() {  //构造方法2
        this.key = null;
    }
    public RecordNode(Comparable key) {  //构造方法2
        this.key = key;
    }
}
//关键字类型类
class KeyType implements Comparable<KeyType> {
    private int key;   //关键字
    public KeyType() {
    }
    public KeyType(int key) {
        this.key = key;
    }
    public int getKey() {
        return key;
    }
    public void setKey(int key) {
        this.key = key;
    }
    public String toString() { //覆盖 toString()方法
        return key +"";
    }
    public int compareTo(KeyType another) {  //覆盖 Comparable 接口中比较关键字大小的方法
        int thisVal = this.key;
        int anotherVal = another.key;
        return (thisVal < anotherVal ? -1 : (thisVal == anotherVal ? 0 : 1));
    }
}
```

```
//采用开放地址法的哈希表类
class HashTable{
     private RecordNode[] table;                          //哈希表的对象数组
     public HashTable(int size) {                        //构造指定大小的哈希表
          this.table=new RecordNode[size];
             for(int i=0;i<table.length;i++){
             table[i]=new RecordNode(0);
           }
     }
     public int hash(int key) {                          //除留余法数哈希函数
         return key % 19;
     }
    //开放地址哈希表的查找
     public RecordNode hashSearch(int key ){
          int i=hash(key);    //求哈希地址
          int j=0;
          while ((table[i].getKey().compareTo(0)!=0) &&
(table[i].getKey().compareTo(key)!=0) && (j< table.length)){
                      //该位置中不为空并且关键字与 key 不相等
               j++;
               i=(i+j)%20;  //用线性探测再散列法求得下一探测地址
          }   //i 指示查找到的记录在表中的存储位置或指示插入位置
           if (j>=table.length) {  //如果表已经为满
                    System.out.println("哈希表已满");
                    return null;
                  }
           else   return table[i];
          }
    //开放地址哈希表的插入
    public void hashInsert( int key){
        RecordNode p=hashSearch(key);
        if (p.getKey().compareTo(0)==0)
           p.setKey(key);    //插入
        else
           System.out.println(" 此关键字记录已存在或哈希表已满");

   }
```

```
    //哈希表的输出
     void Hashdisplay() {
        for(int i=0;i<table.length; i++)
           System.out.print(table[i].getKey().toString()+"  ");
        System.out.println();
     }
}
//测试类
public class SY8_HashSearch_1 {
    static HashTable T=null;
    //建立待查找的哈希表
    public static void createHashtable() throws Exception {
        T=new HashTable(20);
        Scanner sc=new Scanner(System.in);
        System.out.print("请输入待查找的关键字的个数:");
        int n=sc.nextInt();
        System.out.print("请输入查找表中的关键字序列:");
        for (int i = 0; i < n; i++) {  //输入关键字序列,并插入哈希表中
             T.hashInsert(sc.nextInt());
         }
    }
  public static void main(String[]args)throws Exception{
      System.out.println("---创建哈希表---");
      createHashtable();
      System.out.println("创建的哈希表为:");
      T.Hashdisplay();
      System.out.print("请输入待查找的关键字:");
      Scanner sc=new Scanner(System.in);
      int key=sc.nextInt();
      RecordNode p=T.hashSearch(key);
      if ((p.getKey()).compareTo(key)==0)
          System.out.println(" 查找成功!");
      else
          System.out.println(" 查找失败!");
 }
}
```

4．运行结果参考如图 8-1 所示：

图 8-1 验证性实验运行结果

备注：以下设计性和应用性实验内容学生可根据自己的掌握程度或兴趣自行选择其一或其二完成。

七、设计性实验

以链地址法，也叫拉链法处理冲突，实现哈希表的建立，查找和插入操作。

1. 实验要求

编程实现如下功能：

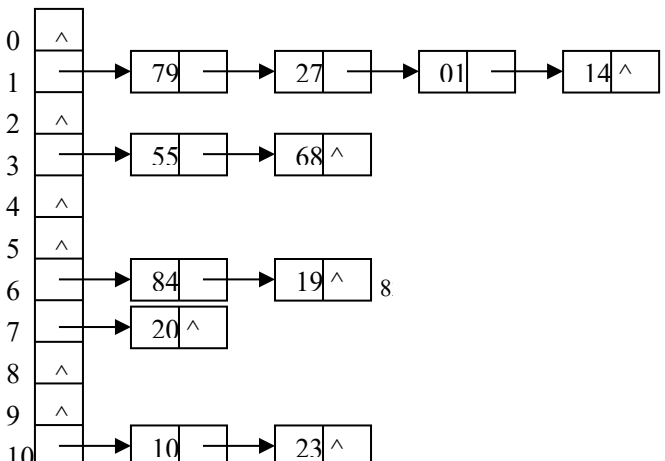
(1) 设哈希表长为 13，用除留余数法构造一个哈希函数。

(2) 输入哈希表中记录的个数 12 和各记录的关键字序列 (19，14,23,01,68,20,84,27,55,11,10,79)，然后以链地址法或叫拉链法作为解决冲突的方法，建立一个链地址哈希表，并输出已经建立的哈希表。

(3) 输入一个待查找记录的关键字 key，完成链地址哈希表的查找操作，如果查找成功，则函数返回查找到的记录在哈希表中的位置值，否则给出查找失败的提示信息。

2. 核心算法分析

链地址法解决冲突的方法是将所有哈希地址相同而关键字值不相同的记录存储在同一线性链表中。对于已知的关键字序列(19，14,23,01,68,20,84,27,55,11,10,79)，如果按哈希函数 h(key)=key % 13 和链地址法处理冲突所构造的哈希表如下图 8-2 所示：



链地址哈希表的存储结构描述为：

```
class HashTable {                //采用链地址法的哈希表类
    private LinkList[] table;
    ……
}
```

其中，LinkList 是线性表中的单链表类。

所以在如图 9-2 所示的哈希表中查找某个指定关键字 key 的记录，只要先由哈希函数 h(key)=key % M 计算出其哈希地址，然后引用单链表类中的 indexOf(key)方法到对应的线性单链表中依次进行查找即可。

3． 核心算法描述

(1) 链地址哈希表的查找操作算法

```
//在哈希表中查找指定对象，若查找成功，返回结点；否则返回 null
public Object hashSearch(int key)throws Exception{
    int i=hash(key);       //计算哈希地址
    int index=table[i].indexOf(key);                    //返回数据元素在单链表中的位置
    if(index>=0)
        return ((Object)table[i].get(index));           //返回单链表中找到的结点
    else
        return null;
}
```

(2) 链地址哈希表的插入操作算法

```
public void hashInsert(int key)throws Exception{     //在哈希表中插入指定的数据元素
        int i=hash(key);                                          //计算哈希地址
        table[i].insert(0,new KeyType(key));   //将指定数据元素插入到相应的链表中
    }
```

其中，indexOf()和 insert() 分别是 LinkList 类中的一个查找和插入方法。

八、应用性设计实验

编程用哈希查找实现班级学生信息的查找操作。

实验要求：

⑴ 为班级 20 个人的姓名设计一个哈希表，假设姓名用汉语拼音表示。要求用除留余数法构造哈希函数，用线性探测再散法处理冲突，并且要求平均查找长度上限为 2。

⑵ 任意给定一个学生的姓名，查找它是否在哈希表中存在，如果查找成功则输出其在哈希表中的哈希地址，否则输入查找失败的提示信息。

实验 10:　排序操作实验

一、实验名称和性质

所属课程	数据结构与算法
实验名称	排序操作
实验学时	2
实验性质	√验证 □综合 √设计
必做/选做	√必做 □选做

二、实验目的

1．熟悉并掌握各种排序方法的设计思路。

2．掌握各种具体排序算法在计算机上的实现。

3．掌握各种排序方法的性能比较。

三、实验内容

1．比较用直接插入排序、冒泡排序和简单选择排序方法进行排序时对关键字的比较次数和移动次数（验证性内容）。

2．希尔排序、归并排序和快速排序算法的实现（设计性内容）。

3．对学生成绩表中相关信息的排序（应用性设计内容）

四、实验的软硬件环境要求

硬件环境要求：

PC 机（单机）

软件环境要求：

硬件：　PC　软件：　Win2000 系统及以上;　VC 编译器

五、知识准备

前期要求掌握排序的含义、各种排序的方法及其性能指标分析。

六、验证性实验

1．实验要求

编程实现如下功能：

（1）输入同样一组整型数据，作为待排序记录的关键字序列。

（2）在进行直接插入排序的同时统计在排序过程中对关键字的比较次数和移动次数，并输出统计结果。

（3）在进行冒泡排序的同时统计在排序过程中对关键字的比较次数和移动次数，并输出统计结果。

（4）在进行简单选择排序的同时统计在排序过程中对关键字的比较次数和移动次数，并输出统计结果。

2. 实验相关原理：

要统计出各种排序方法在排序过程中对关键字的比较次数和移动次数，只要对各种排序算法做适当的修改。修改的方法是首先增加两个计数变量分别用来记录算法中对关键字的比较次数和移动次数，然后对算法中凡出现关键字的比较操作和移动操作的地方增加一个相应

计数变量加 1 的操作即可。

记录比较和移动次数的类描述如下：

```
class CopareMoveNum{
    private int   cpn;
    private int   mvn;
    ……
}
```

待排序的顺序表类描述如下：

```
class SeqList {
    CopareMoveNum[]   cm;      //记录排序过程中比较和移动次数的数组
    private RecordNode[] r;     //待查找的顺序表记录结点数组
    private int curlen;          //顺序表长度,即记录个数
```

下面仅给出三种排序算法的提示和描述。

【核心算法提示】

（1）直接插入排序的基本思想：先将第 0 个记录组成一个有序的子表，然后依次将后面的记录插入到这子表中，且一直保持它的有序性。

（2）冒泡排序的基本思想：将待排序的记录看成从上到下的存放，首先从第一个记录开始，依次对无序区中相邻记录进行关键字比较，如果大在上，小在下，则交换，第一趟扫描下来表中最大的沉在最下面。然后再对前 n-1 个记录进行冒泡排序，直到排序成功为止。

（3）简单选择排序的基本思想：首先在所有记录中选出关键字最小的记录，把它与第一个记录进行位置交换，然后在其余的记录中再选出关键字次小的记录与第二个记录进行位置交换，依此类推，直到所有记录排好序为止。

【核心算法描述】

(1) 不带监视哨的直接插入排序算法

```
public void insertSort() {
   RecordNode temp;
   int i, j;
   for (i = 1; i < this.curlen; i++) {//n-1 趟扫描
       temp = r[i];  //将待插入的第 i 条记录暂存在 temp 中
       for (j = i - 1; j >= 0 && temp.getKey().compareTo(r[j].getKey()) < 0; j--) {
           r[j + 1] = r[j];  //将前面比 r[i]大的记录向后移动
       }
       r[j + 1] = temp;          //r[i]插入到第 j+1 个位置
    }
}
```

(2) 冒泡排序算法

```
public void bubbleSort() {
```

```
    RecordNode temp;        //辅助结点
    boolean flag = true;    //是否交换的标记
    for (int i = 1; i < this.curlen && flag; i++) { //有交换时再进行下一趟，最多 n-1 趟
        flag = false;                               //假定元素未交换
        for (int j = 0; j < this.curlen - i; j++) { //一次比较、交换
            if (r[j].getKey().compareTo(r[j + 1].getKey()) > 0) {     //逆序时，交换
                temp = r[j];
                r[j] = r[j + 1];
                r[j + 1] = temp;
                flag = true;
            }
        }
    }
}
```

(3) 简单选择排序

```
public void selectSort() {
    RecordNode temp; //辅助结点
    for (int i = 0; i < this.curlen - 1; i++) {//n-1 趟排序
        //每趟在从 r[i]开始的子序列中寻找最小元素
        int min = i;                //设第 i 条记录的关键字最小
        for (int j = i + 1; j < this.curlen; j++) {//在子序列中选择关键字最小的记录
            if (r[j].getKey().compareTo(r[min].getKey()) < 0) {
                min = j;            //记住关键字最小记录的下标
            }
        }
        if (min != i) {    //将本趟关键字最小的记录与第 i 条记录交换
            temp = r[i];
            r[i] = r[min];
            r[min] = temp;
        }
    }
}
```

3．源程序代码参考

```
import java.util.Scanner;
//记录结点类
class RecordNode {
    private Comparable key;        //关键字
    private Object element;        //数据元素
```

```
    public Object getElement() {
        return element;
    }
    public void setElement(Object element) {
        this.element = element;
    }
    public Comparable getKey() {
        return key;
    }
    public void setKey(Comparable key) {
        this.key = key;
    }
    public RecordNode(Comparable key) {    //构造方法 1
        this.key = key;
    }
    public RecordNode(Comparable key, Object element) {    //构造方法 2
        this.key = key;
        this.element = element;
    }
}
//关键字类型类
class KeyType implements Comparable<KeyType> {

    private int key;     //关键字
    public KeyType() {
    }
    public KeyType(int key) {
        this.key = key;
    }
    public int getKey() {
        return key;
    }
    public void setKey(int key) {
        this.key = key;
    }
    public String toString() { //覆盖 toString()方法
        return key +"";
    }
```

```
    public int compareTo(KeyType another) {    //覆盖 Comparable 接口中比较关键字大小的方法
        int thisVal = this.key;
        int anotherVal = another.key;
        return (thisVal < anotherVal ? -1 : (thisVal == anotherVal ? 0 : 1));
    }
}
//比较与移动次数类
class CopareMoveNum{
    private int    cpn;
    private int    mvn;
    public int getCpn() {
        return cpn;
    }
    public void setCpn(int cpn) {
        this.cpn = cpn;
    }
    public int getMvn() {
        return mvn;
    }
    public void setMvn(int mvn) {
        this.mvn = mvn;
    }
}
//顺序排序表类
class SeqList {
    CopareMoveNum[]    cm;       //比较和移动次数
    private RecordNode[] r;       //顺序表记录结点数组
    private int curlen;              //顺序表长度,即记录个数
    public RecordNode[] getRecord() {
        return r;
    }
    public void setRecord(RecordNode[] r) {
        this.r = r;
    }
    //  顺序表的构造方法：构造一个存储空间容量为 maxSize 的顺序表，同时建立记录比较与移动次数的数组对象并赋初值
    public SeqList(int maxSize) {
        this.r = new RecordNode[maxSize];    //  为顺序表分配 maxSize 个存储单元
```

```
        this.curlen = 0;                            // 置顺序表的当前长度为 0
        this.cm=new CopareMoveNum[3];     //记录比较和移动次数的数组
        for(int i=0;i<3;i++){       //数组初始化为 0
            this.cm[i] = new CopareMoveNum();
            this.cm[i].setCpn(0);
            this.cm[i].setMvn(0);
        }
    }
    // 求顺序表中的数据元素个数并由函数返回其值
    public int length() {
        return curlen; // 返回顺序表的当前长度
    }
    // 在当前顺序表的第 i 个结点之前插入一个 RecordNode 类型的结点 x
    //其中 i 取值范围为：0≤i≤length()。
    //如果 i 值不在此范围则抛出异常,当 i=0 时表示在表头插入一个数据元素 x,
    //当 i=length()时表示在表尾插入一个数据元素 x
    public void insert(int i, RecordNode x) throws Exception {
        if (curlen == r.length) {       // 判断顺序表是否已满
            throw new Exception("顺序表已满");
        }
        if (i < 0 || i > curlen) {    // i 小于 0 或者大于表长
            throw new Exception("插入位置不合理");
        }
        for (int j = curlen; j > i; j--) {
            r[j] = r[j - 1];     // 插入位置及之后的元素后移
        }
        r[i] = x;     // 插入 x
        this.curlen++;    // 表长度增 1
    }
  //输出数组元素
    public void display() {
        for (int i = 0; i < this.curlen; i++) {
            System.out.print(" " + r[i].getKey().toString());
        }
        System.out.println();
    }
    // 不带监视哨的直接插入排序算法
    public void insertSort() {
```

```
        RecordNode temp;
        int i, j;
        for (i = 1; i < this.curlen; i++) {//n-1 趟扫描
            temp = r[i];   //将待插入的第 i 条记录暂存在 temp 中
            cm[0].setMvn(cm[0].getMvn()+1);     //移动次数加 1;
            for (j = i - 1; j >= 0 && temp.getKey().compareTo(r[j].getKey()) < 0; j--) { //将前面比 r[i]大的记
录向后移动
                cm[0].setCpn(cm[0].getCpn()+1);     //比较次数加 1
                r[j + 1] = r[j];
                cm[0].setMvn(cm[0].getMvn()+1);     //移动次数加 1;
            }
            r[j + 1] = temp;                //r[i]插入到第 j+1 个位置
            cm[0].setMvn(cm[0].getMvn()+1);     //移动次数加 1;
        }
    }

    // 冒泡排序算法
    public void bubbleSort() {
        RecordNode temp;            //辅助结点
        boolean flag = true;     //是否交换的标记
        for (int i = 1; i < this.curlen && flag; i++) { //有交换时再进行下一趟，最多 n-1 趟
            flag = false;                                       //假定元素未交换
            for (int j = 0; j < this.curlen - i; j++) { //一次比较、交换
                cm[1].setCpn(cm[1].getCpn()+1);     //比较次数加 1
                if (r[j].getKey().compareTo(r[j + 1].getKey()) > 0) {        //逆序时，交换
                    temp = r[j];
                    r[j] = r[j + 1];
                    r[j + 1] = temp;
                    cm[1].setMvn(cm[1].getMvn()+3);     //移动次数加 3;
                    flag = true;
                }
            }
        }
    }
     //简单选择排序
    public void selectSort() {
        RecordNode temp; //辅助结点
        for (int i = 0; i < this.curlen - 1; i++) {//n-1 趟排序
```

```
                //每趟在从 r[i]开始的子序列中寻找最小元素
                int min = i;                        //设第 i 条记录的关键字最小
                for (int j = i + 1; j < this.curlen; j++) {//在子序列中选择关键字最小的记录
                    cm[2].setCpn(cm[2].getCpn()+1);     //比较次数加 1
                    if (r[j].getKey().compareTo(r[min].getKey()) < 0) {
                        min = j;                    //记住关键字最小记录的下标
                    }
                }
                if (min != i) {       //将本趟关键字最小的记录与第 i 条记录交换
                    temp = r[i];
                    r[i] = r[min];
                    r[min] = temp;
                    cm[2].setMvn(cm[2].getMvn()+3);     //移动次数加 3;
                }
            }
        }

}
//测试类
public class SY9_Sort_1 {
    static SeqList ST = null;
     //建立待排序的顺序表
     public static void createSearchList() throws Exception {
         ST=new SeqList(20);
         Scanner sc=new Scanner(System.in);
         System.out.print("请输入排序表的表长:");
         int n=sc.nextInt();
         KeyType[] k= new KeyType[n];
         System.out.print("请输入排序表中的关键字序列:");
         for (int i = 0; i < n; i++) {    //输入关键字序列
                k[i] = new KeyType(sc.nextInt());
          }
         for(int i=0;i<n;i++){    //创建顺序排序表
                RecordNode r = new RecordNode(k[i]);
                ST.insert(i, r);
          }
     }
     public static void main(String[] args) throws Exception{
```

```
Scanner sc=new Scanner(System.in);
//System.out.println("创建顺序查找表");
//createSearchList();
  while(true){
        System.out.println(" 1--直接插入排序       2--冒泡排序       3--简单选择排序       4--退出");
        System.out.print("请输入选择(1-4):");
        int i=sc.nextInt();
        switch(i){
             case 1: System.out.println("---直接插入排序---");
                        System.out.println("创建顺序排序表");
                        createSearchList();
                        ST.insertSort();
                        System.out.print("排序结果:");
                        ST.display();
                        System.out.println("比较次数为:"+ST.cm[0].getCpn());
                        System.out.println("移动次数为:"+ST.cm[0].getMvn());
                        break;
             case 2: System.out.println("---冒泡排序---");
                        System.out.println("创建顺序排序表");
                        createSearchList();
                        ST.bubbleSort();
                        System.out.print("排序结果:");
                        ST.display();
                        System.out.println("比较次数为:"+ST.cm[1].getCpn());
                        System.out.println("移动次数为:"+ST.cm[1].getMvn());
                        break;
             case 3: System.out.print("---简单选择排序---");
                        System.out.println("创建顺序排序表");
                        createSearchList();
                        ST.selectSort();
                        System.out.print("排序结果:");
                        ST.display();
                        System.out.println("比较次数为:"+ST.cm[2].getCpn());
                        System.out.println("移动次数为:"+ST.cm[2].getMvn());
                        break;
             case 4:return;
    }
}
```

```
        }
    }
```

4．运行结果参考如图 9-1 所示：

9-1 验证性实验运行结果

备注：以下设计性和应用性实验内容学生可根据自己的掌握程度或兴趣自行选择其一或其二完成。

七、设计性实验

编程实现希尔排序、归并排序和快速排序

1 实验要求

① 对输入的同一组待排序的数据进行希尔排序、快速排序和归并排序，并分别输出排序前的数据序列和排序后的数据序列。

② 在主程序中设计一个菜单，使用户可选择执行其中的任何一种或几种排序，并查看排序结果。

2 核心算法分析

(1) 希尔排序的基本思想：对待排记录序列先作“**宏观**”调整，再作“**微观**”调整。所谓“宏观”调整，指的是 “跳跃式”的插入排序。具体做法为：将记录序列分成若干子序列，分别对每个子序列进行插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。具体做法是：先将 n 个记录分成 d（d<n） 个子序列：

{ r[0]，r[0+d]，r[0+2d]，…，r[0+kd] }；{ r[1]，r[1+d]，r[1+2d]，…，r[1+kd] }；

……； { r[d-1]，r[2d-1]，r[3d-1]，…，r[kd-1]，r[(k+1)d-1] }

其中，**d** 称为增量，它的值在排序过程中从大到小逐渐缩小，一般情况，第一个增量 d_1 取值为 n/2，然后依次取 $d_i=d_{i-1}/2$，直至取值为 1，由此，希尔排序也叫“缩小增量排序”。然后对每组进行直接插入排序，再对每个增量重复上述过程，直到增量为 1 时，最后对全体记录再做一次直接插入排序。

(2) 快速排序的基本思想：通过一趟排序，将待排序记录分割成独立的两部分，其中一部分记录的关键字均小于另一部分的关键字，然后对两部分继续快速排序，以达到整个序列不序。具体做法是：首先在待排序的记录序列（$r_0, r_1, \cdots, r_{n-1}$）中选第一个记录 r_0，以其作为“支点”记录，经过比较和移动，将所有关键字小于支点关键字的记录均移动至该记录之前，反之，将所有关键字大于支点关键字的记录均移动至该记录之后。致使一趟排序之后，
支点记录就放到最后排序结果应该在的位置上，并且将待排序的记录分割成了两个子序列，对这两个子序列又递归进行希尔排序，如此继续下去，使每个子序列长度为 1 为止。

(3) 2 路归并排序的基本思想：将两个有序子序列 “归并” 为一个有序序列。具体做法是：先将 n 个待排序记录看成是 n 个长度为 1 的有序子表，把相邻的有序子表进行两两归并，便得到[n/2]个有序表，再将[n/2]个有序的子表两两归并，如此重复，直到最后得到一个长度为 n 的有序表为止。

3．核心算法描述

(1) 希尔排序算法

```
public void shellSort(int[] d) { //d[]为增量数组
    RecordNode temp;
    int i, j;
    System.out.println("希尔排序");
    //控制增量，增量减半，若干趟扫描
    for (int k = 0; k < d.length; k++) {
        //一趟中若干子表，每个记录在自己所属子表内进行直接插入排序
        int dk = d[k];
        for (i = dk; i < this.curlen; i++) {
            temp = r[i];
            for (j = i - dk; j >= 0 && temp.getKey().compareTo(r[j].getKey()) < 0; j -= dk) {
                r[j + dk] = r[j];
            }
            r[j + dk] = temp;
        }
        System.out.print("增量 dk=" + dk + "  ");
        display();
    }
}
```

(2) 快速排序算法

```
//一趟快速排序:交换排序表 r[i..j]的记录，使支点记录到位，并返回其所在位置
//此时，在支点之前(后)的记录关键字均不大于(小于)它
public int Partition(int i, int j) {
    RecordNode pivot = r[i];          //第一个记录作为支点记录
//    System.out.print(i + ".." + j + ",  pivot=" + pivot.getKey() + "  ");
    while (i < j) {    //从表的两端交替地向中间扫描
        while (i < j && pivot.getKey().compareTo(r[j].getKey()) <= 0) {
            j--;
        }
        if (i < j) {
            r[i] = r[j];   //将比支点记录关键字小的记录向前移动
            i++;
        }
        while (i < j && pivot.getKey().compareTo(r[i].getKey()) > 0) {
            i++;
        }
        if (i < j) {
            r[j] = r[i];   //将比支点记录关键字大的记录向后移动
            j--;
        }
    }
    r[i] = pivot;         //支点记录到位
//    display();
    return i;             //返回支点位置
}

// 递归形式的快速排序算法:对子表 r[low..high]快速排序
public void qSort(int low, int high) {
    if (low < high) {
        int pivotloc = Partition(low, high);  //一趟排序，将排序表分为两部分
        qSort(low, pivotloc - 1);   //低子表递归排序
        qSort(pivotloc + 1, high);  //高子表递归排序
    }
}

//顺序表快速排序算法
public void quickSort() {
    qSort(0, this.curlen - 1);
```

```
}
```

(3) 2路归并排序算法

```
//一趟归并算法：把数组r[n]中每个长度为s的有序表两两归并到数组order[n]中
//s 为子序列的长度，n为排序序列的长度
 public void mergepass(RecordNode[] r, RecordNode[] order, int s, int n) {
     System.out.print("子序列长度s=" + s + "  ");
     int p = 0;  //p为每一对待合并表的第一个元素的下标，初值为0
     while (p + 2 * s - 1 <= n - 1) {  //两两归并长度均为s的有序表
         merge(r, order, p, p + s - 1, p + 2 * s - 1);
         p += 2 * s;
     }
     if (p + s - 1 < n - 1) {  //归并最后两个长度不等的有序表
         merge(r, order, p, p + s - 1, n - 1);
     } else {
         for (int i = p; i <= n - 1; i++) {//将剩余的有序表复制到order中
              order[i] = r[i];
         }
     }
 }
 //2-路归并排序算法
 public void mergeSort() {
     System.out.println("归并排序");
     int s = 1;                        //s为已排序的子序列长度，初值为1
     int n = this.curlen;
     RecordNode[] temp = new RecordNode[n];  //定义长度为n的辅助数组temp
     while (s < n) {
         mergepass(r, temp, s, n);  //一趟归并，将r数组中各子序列归并到temp中
         display();
         s *= 2;                       //子序列长度加倍
         mergepass(temp, r, s, n);  //将temp数组中各子序列再归并到r中
         display();
         s *= 2;
     }
 }
}
```

八、设计性实验

编程实现对学生成绩表的相关信息排序。

实验要求：

⑴ 建立一个由 n 个学生的考试成绩表，每条信息由学号、姓名和分数组成。

⑵ 按学号排序，并输出排序结果。

⑶ 按分数排序，分数相同的则按学号有序，并输出排序结果。

⑷ 排序方法及学生成绩表的存储结构不作限制，学生选择性能较好的即可。