前言

《数据结构》是软件工程等专业的核心专业基础课，培养学生能够分析数据对象特征，根据问题的需要，确定逻辑结构并选择合适的存储结构，实现典型算法设计及性能分析的能力。通过实验教学，使学生加深对所学知识的理解，掌握典型算法的设计。它的目的和任务是：培养和提高学生算法分析与设计能力，建立数据结构的概念，为后续课程的学习及软件开发打好基础。

本实验指导书是刘小晶主编，《数据结构（Java 语言描述）》的配套教材。编者根据计算机课程实践性强等特点，编写了本实验教程，帮助学生有计划地系统地上机实践。

根据教学内容和教学目标，实验指导书设计了十个实验，实验学时 20 学时，每个实验 2 学时。学生应按照实验指导书的要求，完成指定的实验任务，并及时提交实验报告。要求学生在每次实验之前做好预习，实验后按要求写出实验报告。在每次实验过程中教师要考核学生每次实验的完成情况。

实验说明

一、实验课课堂要求

1、实验课是对学生理论课学习的检验，也是培养动手能力的重要方法。因此实验课必须提前预习，将课堂讲授的算法写成程序。最基本的要求是将算法完整、正确、清晰、认真地抄写在笔记本上。实验课前检查，如果没有，本次实验成绩不及格。再高一级的要求是将代码敲进计算机，实验课上进行代码调试。学有余力的同学可以给自己提出更高的要求，在完成实验内容的基础上，再做一些课后的算法设计题目。

2、实验课上不许做任何与实验无关的事情，比如看电影、玩游戏、听音乐、看照片等。也不允许做和数据结构这门课无关的内容，比如完成别的科目的作业、论文等。如果发现以上行为，没有警告机会，本次实验成绩不及格。

3、注意实验纪律，上课期间不允许随意走动，说笑打闹。保持实验室整洁，自己的垃圾自己带走。

二、关于实验报告的说明

1、填写实验报告，首先按照实际情况填写时间、班级、姓名、学号这四项。

2、将完成实验的情况，按照实验报告的具体格式将对应的内容填写清楚。主要是“实验结果及分析”和“在实验中遇到的问题、解决问题采用的方法”这两项。

①“实验结果及分析”

主要写你这次实验使用的数据，输入的是什么数据，程序运行过后输出的是什么数据。分析你的结果，如果输出的数据和预想的不一样，则要分析出是哪一个方法的哪一个步骤出现的问题。

②“在实验中遇到的问题、解决问题采用的方法”

写出你在实验中遇到的各种问题，可以是算法的问题，也可以是程序运行的问题。写出你解决问题的方法，以及你在解决问题的过程中有哪些收获，对哪些知识点有了新的理解。

3、每个人的输入数据不可能完全相同，遇到的问题也不可能完全相同。所以实验报告不可能存在雷同的可能性。有问题可以互相讨论，但是绝对不允许几个人共用一段代码。如果发现雷同的实验报告或代码，不论你是抄袭者还是被抄袭者，本次实验成绩一律记零。

4、实验报告和代码放在一个文件夹里提交。

实验报告命名方式为“实验 X 班级姓名”，比如“实验一升本 1 班张三”，保存成 word2003 版本。

程序代码放在一个文件夹里，命名方式与实验报告相同。

将实验报告的 word 文档和程序代码的文件夹一起放在一个文件夹里，命名同实验报告相同。提交这个文件夹作为实验提交物。

做完的同学当堂可以提交，做不完的同学可以延后提交。最晚延后一周。延后提交者自己到办公室提交，地点在 5 号楼 B225。过期不提交者，本次实验成绩为零。

三、评分标准

优秀：独立完成代码，程序结构清晰正确，要求的功能全部实现

良好：独立完成代码，程序结构清晰正确，功能部分实现

中等：独立完成代码，程序结构有小毛病，功能部分实现

及格：独立完成代码

不及格：抄袭代码或者在实验课上做其他无关的事情

以上用红色标识的部分是底线，类似于我们讲的边界情况。希望同学们认真实验，远离这些边界情况。希望大家能充分利用时间，上好实验课。衷心地希望每位同学都能学好这门课。

实验一 顺序表

一、实验目的：

1、掌握线性表的顺序存储结构

2、验证顺序表及其操作的基本实现

3、掌握数据结构及算法的程序实现的基本方法

二、实验内容：

1、 建立含有若干元素的顺序表

2、 对已经建立的顺序表实现插入，删除，查找（按位查找、按值查找）等基本操作

三、实现提示：

首先定义顺序表的数据类型——顺序表类 SqList，包括题目要求的插入、删除、查找等基本操作。

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

    //  顺序表的构造函数，构造一个存储空间容量为 maxSize 的线性表
    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
    }
```

```
    // 读取到线性表中的第 i 个数据元素并由函数返回其值。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 Object get(int i) throws Exception {
        if (i < 0 || i > curLen - 1) // i 小于 0 或者大于表长减 1
            throw new Exception("第" + i + "个元素不存在"); // 输出异常
        return listElem[i]; // 返回顺序表中第 i 个数据元素
    }

    // 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≤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, 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≤length()- 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 > curLen - 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
    }

    // 返回线性表中首次出现指定元素的索引，如果列表不包含此元素，则返回 -1
    public int indexOf(Object x) {
        int j = 0;// j 为计数器
        while (j < curLen && !listElem[j].equals(x))
            // 从顺序表中的首结点开始查找，直到 listElem[j]指向元素 x 或
```

```
到达顺序表的表尾
                    j++;
                if (j < curLen)// 判断 j 的位置是否位于表中
                    return j; // 返回 x 元素在顺序表中的位置
                else
                    return -1;// x 元素不在顺序表中
            }

            // 输出线性表中的数据元素
            public void display() {
                for (int j = 0; j < curLen; j++)
                    System.out.print(listElem[j] + " ");
                System.out.println();// 换行

            }
        // --------调用构造函数--------
        public static void main(String[] args) throws Exception {
                SqList L = new SqList(10); // 构造一个 10 个存储空间的顺序表

                // --------调用 insert(int i, Object x)插入数据元素--------
                for (int i = 0; i <= 8; i++) // 对该顺序表的前 9 个元素进行赋值，分别为 0、1、2...8
                    L.insert(i, i);
                // --------调用 remove(int i)删除数据元素表--------
                L.remove(5);// 删除数据元素 5
                System.out.println("顺序表中删除数据元素 5 后，表的长度：" + L.length());// 输出
                System.out.println("顺序表中删除数据元素 5 后，剩余的数据元素：");// 输出
                L.display();
                // --------调用 insert(int i, Object x)把数据元素 x 插入到 i 的位置--------
                L.insert(5, 5);
                System.out.println("顺序表中在 5 的位置前插入数据元素 5 后，表的长度：" + L.length());
                System.out.println("顺序表中在 5 的位置前插入数据元素 5 后，表中的数据元素：");
                L.display();
            }
        }
```

四、特别说明：

用 Object 类实现起来有困难的同学可以用整形数据作为操作对象来实现。具体算法参见课堂笔记及教材。

实验二 链表

一、实验目的：

1、掌握线性表的链式存储结构

2、验证单链表及其操作的基本实现

3、验证单链表的其他算法，比如：合并，拆分，逆置等

二、实验内容：

1、建立含有若干元素的单链表

2、对已经建立的单链表实现插入（按位插入，有序插入），删除，查找（按位查找、按值查找）等基本操作

3、至少实现一个单链表的其他算法。比如合并，拆分，逆置等。也可以做循环链表的验证实验，比如约瑟夫问题等

4、有趣的一元多项式

1) 利用线性表实现一元多项式

$$f(x)=a_0+a_1x+a_2x^2+\cdots a_nx^n$$

2) 求解问题：某船队有载重量为 1,2,4,8 和 16 百吨的船各一艘，为了保证使用效益，用船时必须满载，问该船队有多少种载货服务功能

三、实现提示：

首先定义单链表的数据类型——单链表类 LinkList，包括题目要求的插入、删除、查找等基本操作。为了便于查看操作结果，设计一个输出方法依次输出单链表的元素。

```
public class LinkList implements IList {
    private Node head;// 单链表的头指针

    // 单链表的构造函数
    public LinkList() {
        head = new Node(); // 初始化头结点
    }

    public LinkList(int n, boolean Order) throws Exception {
        this();// 初始化头结点
        if (Order) // 用尾插法顺序建立单链表
            create1(n);
        else
            // 用头插法逆位序建立单链表
            create2(n);
    }

    // 用尾插法顺序建立单链表。其中 n 为该单链表的元素个数
```

```
public void create1(int n) throws Exception {
    Scanner sc = new Scanner(System.in);// 构造用于输入的对象
    for (int j = 0; j < n; j++)
        // 输入 n 个元素的值
        insert(length(), sc.next());// 生成新结点,插入到表尾
}

// 用头插法逆位序建立单链表。其中 n 为该单链表的元素个数
public void create2(int n) throws Exception {
    Scanner sc = new Scanner(System.in);// 构造用于输入的对象
    for (int j = 0; j < n; j++)
        // 输入 n 个元素的值
        insert(0, sc.next());// 生成新结点,插入到表头
}

// 将一个已经存在的带头结点单链表置成空表
public void clear() {
    head.setData(null);
    head.setNext(null);
}

// 判断当前带头结点的单链表是否为空
public boolean isEmpty() {
    return head.getNext() == null;// 判断首结点是否为空
}

// 求带头结点单链表中的数据元素个数并由函数返回其值
public int length() {
    Node p = head.getNext();// 初始化,p 指向首结点,length 为计数器
    int length = 0;
    while (p != null) {// 从首结点向后查找，直到 p 为空
        p = p.getNext();// 指向后继结点
        ++length;// 长度增 1
    }
    return length;
}

// 读取带头结点单链表中的第 i 个数据元素
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) {// 从首结点向后查找，直到 p 指向第 i 个元
 素或 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");// 输出异常
        }
        return p.getData();// 返回元素 p
    }

    // 在带头结点单链表中第 i 个数据元素之前插入一个值为 x 的数据元素
    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; // 第 i 个结点前驱的位置
        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);
    }

    // 将线性表中第 i 个数据元素删除。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;// p 指向要删除结点的前驱结点
        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
    }

    // 在带头结点的单链表中查找值为 x 的元素，如果找到，则函数返回该
元素在线性表中的位置，否则返回-1
    public int indexOf(Object x) {
        Node p = head.getNext();// 初始化,p 指向首结点,j 为计数器
        int j = 0;
```

```
        while (p != null && !p.getData().equals(x)) {// 从单链表中的首结点元素开始查找，直到 p.getData()指向元素 x 或到达单链表的表尾
            p = p.getNext();// 指向下一个元素
            ++j;// 计数器的值增 1
        }
        if (p != null)// 如果 p 指向表中的某一元素
            return j;// 返回 x 元素在顺序表中的位置
        else
            return -1;// x 元素不在顺序表中
    }

    // 输出线性表中的数据元素
    public void display() {
        Node node = head.getNext();// 取出带头结点的单链表中的首结点元素
        while (node != null) {
            System.out.print(node.getData() + " ");// 输出数据元素的值
            node = node.getNext();// 取下一个结点
        }
        System.out.println();// 换行
    }

    public Node getHead() {
        return head;
    }

    public void setHead(Node head) {
        this.head = head;
    }

    // 在非递减的有序单链表中插入一个值为 x 的数据元素，并使单链表仍保持有序的操作
     public void insert(int    x) {
        Node p = head.getNext();
        Node q = head;// q 用来记录 p 的前驱结点
        int temp;
        while (p != null) {
            temp = ((Integer) p.getData()).intValue();
            if (temp < x) {
                q = p;
                p = p.getNext();
            } else
                break;
        }
```

```
        Node s = new Node(x); // 生成新结点
        s.setNext(p);// 将 s 结点插入到单链表的 q 结点与 p 结点之间
        q.setNext(s);
    }

     // 实现对单链表就地逆置(采用的是头插法)
    public void reverse() {
        Node p = head.getNext();
        head.setNext(null);
        Node q;
        while (p != null) {
            q = p.getNext();
            p.setNext(head.getNext());
            head.setNext(p);
            p = q;
        }
    }

    // 实现删除单链表中数据域值等于 x 的所有结点的操作，并返回被删除结点的个数
    public int removeAll(Object x) {
        Node p = head.getNext();// 初始化,p 指向首结点,j 为计数器
        Node q = head; // 用来记录 p 的前驱结点
        int j = 0;// 用来记录被删除结点的个数
        while (p != null) { // 从单链表中的首结点开始对整个链表遍历一次
            if (p.getData().equals(x)) {
                q.setNext(p.getNext());
                ++j;// 计数器的值增 1
            } else
                q = p;
            p = p.getNext();// 指向下一个元素
        }
        return j;// 返回被删除结点的个数
    }

  public static void main(String[] args) throws Exception {
        // --------调用 create(int n)从表尾到表头逆向建立单链表--------
        System.out.println("请输入 3 个单链表中的数据元素：");
        LinkList L = new LinkList(3, true);// 从表头到表尾顺序建立一个表长为 3 的单链表
        System.out.println("单链表中各个数据元素：");
        L.display(); // 输出单链表中所有的数据元素
```

```
            // --------调用 length()求顺序表的长度--------
            System.out.println("单链表的长度:" + L.length());// 输出顺序表的长
度

            // --------调用 get(int i)取出第 i 个元素--------
            if (L.get(2) != null)// 取第二个元素
                System.out.println("单链表中第 2 个元素:" + L.get(2));

            // --------调用 indexOf(Object x)查找 x 元素所在的位置--------
            int order = L.indexOf("c");// 求出数据元素字符串 c 在顺序表中的位
置
            if (order != -1)
                System.out.println("单链表中值为字符串 c 的数据元素的位置为:
" + order);// 输出数据元素 c 的位置
            else
                System.out.println("字符'c'不在此单链表中");

            // --------调用 remove(int i)删除数据元素--------
            L.remove(2); // 删除第二个数据元素
            System.out.println("删除第二个数据元素后单链表中各个数据元素:
");
            L.display();

            // --------调用 insert(int i, Object x)插入数据元素--------
            L.insert(2, 'd');// 在单链表的第三个位置插入数据元素 d
            System.out.println("在 2 的位置插入数据元素 d 后单链表中各个数据
元素: ");
            L.display(); // 输出单链表中所有的数据元素

            // --------调用 L.clear()将顺序表置空--------
            L.clear();
            System.out.println("将单链表置空后，再次打印表中的元素:");

            // --------调用 isEmpty()判断顺序表是否为空--------
            if (L.isEmpty())
                System.out.println("单链表为空");
            else {
                System.out.println("单链表不为空,单链表中各个数据元素: ");
                L.display();
            }
        }
    }
```

有趣的多项式

案例分析：

假设现有质量分别为 1 克、2 克和 3 克的砝码各一枚，问只用这些砝码各一次你能称出哪几种质量的物体来？而对各种质量确定的物体又有多少种不同的称量方案？

首先，分析可能得到的称重情况如下表所示：

表 1 称重问题的解

拥有砝码情况	称量种数		称量方案	
	可称物体的质量种数 n	被称量物体质量克数 m	称量的方案数 P	实际称量方案
3 枚 1 克 2 克 3 克	7	$m_1=0$	$P_1=1$	无砝码
		$m_2=1$	$P_2=1$	1 克
		$m_3=2$	$P_3=1$	2 克
		$m_4=3$	$P_4=2$	(1+2)克或 3 克
		$m_5=4$	$P_5=1$	4 克=(1+3)克
		$m_6=5$	$P_6=1$	5 克=(2+3)克
		$m_7=6$	$P_7=1$	6 克=(1+2+3)克

接着，进一步分析细节

(1) 如果有 1 枚重为 K_1 的砝码，称重的质量种数为 n=2,两种物体的实际质量为 $m=\{m_1=0,m_2=K_1\}$,每种质量的称重方案数为 $P=\{P_1=1,P_2=1\}$，发现 m 和 P 之间的关系？

(2) 假设有两枚砝码，其质量分别为 K_1 和 K_2，假设 $K_1 \neq K_2$,综合(1)得到可以称重的质量情况为：

(a)0+0=0；

(b) $K_1+0=K_1$

(c)$0+K_2=K_2$

(d)K_1+K_2

(3) 分析总结

根据(1)可用一个二项式表示 $1+x^{K1}$

根据(2)可得到对应的多项式

$(1+x^{K1})(1+x^{K2})= 1+x^{K1}+x^{K2}+x^{K1+K2}$

下面得到如果是 n 个这样的砝码得到通式

$(1+x^{K1})(1+x^{K2})\cdots(1+x^{Kn})$

最后，考虑如何存储到计算机中实现实际问题的求解，利用所学的线性表结构，用顺序表还是链表

四、特别说明：

用 Object 类实现起来有困难的同学可以用整形数据作为操作对象来实现。具体算法参见课堂笔记及教材。

实验三 栈

一、实验目的：

1、掌握栈的存储结构（顺序或链栈）

2、掌握栈的操作特性

3、掌握基于栈的基本操作的实现方法

二、实验内容：

1、建立一个空栈

2、对已经建立的栈实现进栈、出栈、取栈顶元素等基本操作

三、实现提示：

首先，定义顺序栈类或链栈——SqStack 或 LinkStack

其次，设计构造函数和析构函数。

最后，设计算法完成实验内容要求的基本操作。

```
public class SqStack implements IStack {

    private Object[] stackElem; // 栈存储空间
    private int top; // 非空栈中始终表示栈顶元素的下一个位置，当栈为空时其值为 0

    // 栈的构造函数，构造一个存储空间容量为 maxSize 的栈
    public SqStack(int maxSize) {
        top = 0; // 初始化 top 为 0
        stackElem = new Object[maxSize];// 为栈分配 maxSize 个存储单元
    }
    // 将一个已经存在的栈置成空
    public void clear() {
        top = 0;
    }
    // 测试栈是否为空
    public boolean isEmpty() {
        return top == 0;
    }
    // 求栈中的数据元素个数并由函数返回其值
    public int length() {
        return top;
    }
    // 查看栈顶对象而不移除它，返回栈顶对象
    public Object peek() {
        if (!isEmpty())// 栈非空
            return stackElem[top - 1]; // 栈顶元素
        else
            // 栈为空
            return null;
    }
```

```
        // 移除栈顶对象并作为此函数的值返回该对象
        public Object pop() {
            if (top == 0)// 栈为空
                return null;
            else {// 栈非空
                return stackElem[--top];// 修改栈顶指针，并返回栈顶元素
            }
        }
        // 把项压入栈顶
        public void push(Object o) throws Exception {
            if (top == stackElem.length)// 栈满
                throw new Exception("栈已满");// 输出异常
            else
                // 栈未满
                stackElem[top++] = o;// o 赋给栈顶元素后，top 增 1
        }
        // 打印函数，打印所有栈中的元素(栈顶到栈底)
        public void display() {
            for (int i = top - 1; i >= 0; i--)
                System.out.print(stackElem[i].toString() + " ");// 打印
        }
    public static void main(String[] args) throws Exception {
            SqStack S = new SqStack(100); // 初始化一个新的栈
            for (int i = 1; i <= 10; i++)
                // 初始化栈中的元素，其中元素个数为 10
                S.push(i);
            System.out.println("栈中各元素为(栈顶到栈底)：  ");
            S.display();// 打印栈中元素（栈低到栈顶）
            System.out.println();
            if (!S.isEmpty())// 栈非空，输出
                System.out.println("栈非空！");
            System.out.println("栈的长度为：  " + S.length());// 输出栈的长度
            System.out.println("栈顶元素为：  " + S.peek().toString());// 输出栈顶
元素
            System.out.println("去除栈顶元素后，栈中各元素为(栈顶到栈底)：
");
            S.pop();// 删除元素
            S.display();// 打印栈中元素
            System.out.println();
            System.out.println("去除栈中剩余的所有元素！  进行中。。。"); // 输
出
            S.clear(); // 清除栈中的元素
            if (S.isEmpty())// 栈空，输出
                System.out.println("栈已清空!");
```

```
    }
}
```

四、特别说明：

用 Object 类实现起来有困难的同学可以用整形数据作为操作对象来实现。具体算法参见课堂笔记及教材。

实验四 队列

一、实验目的：

1、掌握队列的存储结构（链式）

2、掌握队列的操作特性

3、掌握基于队列的基本操作的实现方法

二、实验内容：

1、建立一个空队列

2、对已经建立的队列实现入队、出队、取队头元素等基本操作

3、设停车场内只有一个可停放 n 辆汽车的狭长通道，且只有一个大门可供汽车进出。汽车在停车场内按车辆到达时间的先后顺序，依次由北向南排列（大门在最南端，最先到达的第一辆车停放在车场的最北端），若车场内已停满 n 辆汽车，则后来的汽车只能在门外的便道上等候，一旦有车开走，则排在便道上的第一辆车即可开入；当停车场内某辆车要离开时，在它之后开入的车辆必须先退出车场为它让路，待该辆车开出大门外，其它车辆再按原次序进入车场，每辆停放在车场的车在它离开停车场时必须按它停留的时间长短交纳费用。

试为停车场编制按上述要求进行管理的模拟程序。

三、实现提示：

首先，定义链队列类——LinkQueue 。

其次，设计构造函数和析构函数。

最后，设计算法完成实验内容要求的基本操作。

```
public class LinkQueue implements IQueue {
    private Node front;// 队头的引用

    private Node rear;// 队尾的引用，指向队尾元素

    // 链队列类的构造函数
    public LinkQueue() {
        front = rear = null;
    }

    // 将一个已经存在的队列置成空
    public void clear() {
        front = rear = null;

    }
```

```
    // 测试队列是否为空
    public boolean isEmpty() {
        return front == null;
    }

    // 求队列中的数据元素个数并由函数返回其值
    public int length() {
        Node p = front;
        int length = 0;// 队列的长度
        while (p != null) {// 一直查找到队尾
            p = p.getNext();
            ++length;// 长度增 1
        }
        return length;
    }

    // 把指定的元素插入队列
    public void offer(Object o) {
        Node p = new Node(o);// 初始化新的结点
        if (front != null) {// 队列非空
            rear.setNext(p);
            rear = p;// 改变队列尾的位置
        } else
            // 队列为空
            front = rear = p;
    }

    // 查看队列的头而不移除它，返回队列顶对象，如果此队列为空，则返回 null
    public Object peek() {
        if (front != null) // 队列非空
            return front.getData();// 返回队列元素
        else
            return null;
    }

    // 移除队列的头并作为此函数的值返回该对象，如果此队列为空，则返回 null
    public Object poll() {
        if (front != null) { // 队列非空
            Node p = front;// p 指向队列头结点
            front = front.getNext();
            return p.getData();// 返回队列头结点数据
        } else
```

```
            return null;
        }

        // 打印函数，打印所有队列中的元素(队列头到队列尾)
        public void display() {
            if (!isEmpty()) {
                Node p = front;
                while (p != rear.getNext()) {// 从对头到队尾
                    System.out.print(p.getData().toString() + " ");
                    p = p.getNext();
                }
            } else {
                System.out.println("此队列为空");
            }
        public static void main(String[] args) {
            LinkQueue Q = new LinkQueue();
            for (int i = 1; i <= 10; i++)
                // 初始化队列中的元素，其中元素个数为 10
                Q.offer(i);
            System.out.println("队列中各元素为(队首到队尾)：  ");
            Q.display();// 打印队列中元素（队首到队尾）
            System.out.println();
            if (!Q.isEmpty())
                System.out.println("队列非空！");
            System.out.println("队列的长度为：  " + Q.length());// 输出队列的长
度
            System.out.println("队首元素为：  " + Q.peek().toString());// 输出队首
元素
            System.out.println("去除队首元素后，队列中各元素为(队首到队尾)：
");
            Q.poll();// 删除元素
            Q.display();// 打印队列中元素
            System.out.println();
            System.out.println("去除队列中剩余的所有元素！ 进行中。。。"); // 输
出
            Q.clear(); // 清除队列中的元素
            if (Q.isEmpty())// 队列空，输出
                System.out.println("队列已清空!");
        }
    }
```

停车场管理

实现提示：

以栈模拟停车场，以队列模拟车场外的便道，按照从终端读入的输入数据序列进行模拟管理。每一组输入数据包括三个数据项：汽车"到达"或"离去"信

息、汽车牌照号码及到达或离去的时刻，对每一组输入数据进行操作后的输出数据为：若是车辆到达，则输出汽车在停车场内或便道上的停车位置；若是车离去；则输出汽车在停车场内停留的时间和应交纳的费用（在便道上停留的时间不收费）。栈以顺序结构实现，队列以链表实现。

需另设一个栈，临时停放为给要离去的汽车让路而从停车场退出来的汽车，也用顺序存储结构实现。输入数据按到达或离去的时刻有序。栈中每个元素表示一辆汽车，包含两个数据项：汽车的牌照号码和进入停车场的时刻。

设 n=2,输入数据为：('A'，1，5)，('A'，2，10)，('D'，1，15)，('A'，3， 20)， ('A'，4，25)，('A'，5，30)，('D'，2，35)，('D'，4，40)，('E'，0，0)。每一组输入数据包括三个数据项：汽车"到达"或"离去"信息、汽车牌照号码及到达或离去的时刻，其中，'A'表示到达；'D'表示离去，'E'表示输入结束。

四、特别说明：

用 Object 类实现起来有困难的同学可以用整形数据作为操作对象来实现。具体算法参见课堂笔记及教材。

实验五 树和二叉树

一、实验目的：

1、掌握二叉树的逻辑结构

2、掌握二叉树的二叉链表存储结构

3、掌握基于二叉链表存储的二叉树的遍历操作的实现

二、实验内容：

1、建立一棵含有 n 个结点的二叉树，采用二叉链表存储

2、先序（中序或者后序）遍历建立起来的二叉树

3、对已经建立的二叉树，求二叉树的叶子结点数

4、对已经建立的二叉树，求二叉树的高度

三、实现提示：

```
//二叉链式存储结构下的二叉树
public class BiTree {
    private BiTreeNode root;// 树的根结点

    public BiTree() {// 构造一棵空树
        this.root = null;
    }
    public BiTree(BiTreeNode root) {// 构造一棵树
        this.root = root;
    }
    // 由先根遍历的数组和中根遍历的数组建立一棵二叉树
    // 其中参数 preOrder 是整棵树的 先根遍历，inOrder 是整棵树的中根遍历，
```

```
preIndex 是
    // 先根遍历从 preOrder 字符串中的开始位置，inIndex 是中根遍历从字符串
inOrder 中的开始位置，count 表示树结点的个数
    public BiTree(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(new BiTree(preOrder, inOrder, preIndex + 1, inIndex,
                    i).root);// 建立树的左子树
            root.setRchild(new BiTree(preOrder, inOrder, preIndex + i + 1,
                    inIndex + i + 1, count - i - 1).root);// 建立树的右子树
        }
    }

    // 由标明空子树的先根遍历序列建立一棵二叉树
    private static int index = 0;// 用于记录 preStr 的索引值
    public BiTree(String preStr) {
        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#
            root = new BiTreeNode(c);// 建立树的根结点
            root.setLchild(new BiTree(preStr).root);// 建立树的左子树
            root.setRchild(new BiTree(preStr).root);// 建立树的右子树
        } else
            root = null;
    }
    // 先根遍历二叉树基本操作的递归算法
    public void preRootTraverse(BiTreeNode 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
        }
    }
    // 先根遍历二叉树基本操作的非递归算法
    public void preRootTraverse() {
        BiTreeNode T = root;
```

```
        if (T != null) {
            LinkStack S = new LinkStack();// 构造栈
            S.push(T);// 根结点入栈
            while (!S.isEmpty()) {
                T = (BiTreeNode) S.pop();// 移除栈顶结点，并返回其值
                System.out.print(T.getData()); // 访问结点
                while (T != null) {
                    if (T.getLchild() != null) // 访问左孩子
                        System.out.print(T.getLchild().getData()); // 访问结点
                    if (T.getRchild() != null)// 右孩子非空入栈
                        S.push(T.getRchild());
                    T = T.getLchild();
                }
            }
        }
    }
    // 中根遍历二叉树基本操作的递归算法
    public void inRootTraverse(BiTreeNode 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
        }
    }
    // 中根遍历二叉树基本操作的非递归算法
    public void inRootTraverse() {
        BiTreeNode T = root;
        if (T != null) {
            LinkStack S = new LinkStack();// 构造链栈
            S.push(T); // 根结点入栈
            while (!S.isEmpty()) {
                while (S.peek() != null)
                    // 将栈顶结点的所有左孩子结点入栈
                    S.push(((BiTreeNode) S.peek()).getLchild());
                S.pop(); // 空结点退栈
                if (!S.isEmpty()) {
                    T = (BiTreeNode) S.pop();// 移除栈顶结点，并返回其值
                    System.out.print(T.getData()); // 访问结点
                    S.push(T.getRchild());// 结点的右孩子入栈
                }
            }
        }
    }
    // 后根遍历二叉树基本操作的递归算法
```

```
public void postRootTraverse(BiTreeNode 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
    }
}
// 后根遍历二叉树基本操作的非递归算法
public void postRootTraverse() {
    BiTreeNode T = root;
    if (T != null) {
        LinkStack S = new LinkStack();// 构造链栈
        S.push(T); // 根结点进栈
        Boolean flag;// 访问标记
        BiTreeNode p = null;// p 指向刚被访问的结点
        while (!S.isEmpty()) {
            while (S.peek() != null)
                // 将栈顶结点的所有左孩子结点入栈
                S.push(((BiTreeNode) S.peek()).getLchild());
            S.pop(); // 空结点退栈
            while (!S.isEmpty()) {
                T = (BiTreeNode) S.peek();// 查看栈顶元素
                if (T.getRchild() == null || T.getRchild() == p) {
                    System.out.print(T.getData()); // 访问结点
                    S.pop();// 移除栈顶元素
                    p = T;// p 指向刚被访问的结点
                    flag = true;// 设置访问标记
                } else {
                    S.push(T.getRchild());// 右孩子结点入栈
                    flag = false;// 设置未被访问标记
                }
                if (!flag)
                    break;
            }
        }
    }
}
// 层次遍历二叉树基本操作的算法(自左向右)
public void levelTraverse() {
    BiTreeNode T = root;
    if (T != null) {
        LinkQueue L = new LinkQueue();// 构造队列
        L.offer(T);// 根结点入队列
        while (!L.isEmpty()) {
```

```
                    T = (BiTreeNode) L.poll();
                    System.out.print(T.getData()); // 访问结点
                    if (T.getLchild() != null)// 左孩子非空，入队列
                        L.offer(T.getLchild());
                    if (T.getRchild() != null)// 右孩子非空，入队列
                        L.offer(T.getRchild());
                }
            }
        }
        public BiTreeNode getRoot() {
            return root;
        }
        public void setRoot(BiTreeNode root) {
            this.root = root;
        }

}

// 统计叶结点数目
        public int countLeafNode(BiTreeNode T) {
            int count = 0;
            if (T != null) {
                if (T.getLchild() == null && T.getRchild() == null) {
                    ++count;// 叶结点数增 1
                } else {
                    count += countLeafNode(T.getLchild()); // 加上左子树上叶结点数
                    count += countLeafNode(T.getRchild());// 加上右子树上的叶结点
数
                }
            }
            return count;
        }
        // 统计结点的数目
        public int countNode(BiTreeNode T) {
            int count = 0;
            if (T != null) {
                ++count;// 结点数增 1
                count += countNode(T.getLchild()); // 加上左子树上结点数
                count += countNode(T.getRchild());// 加上右子树上的结点数
            }
            return count;
            }
         //求二叉树的高度
            public int getDepth(BiTreeNode T) {
```

```
        if (T != null) {
            int lDepth = getDepth(T.getLchild());// 左子树的高度
            int rDepth = getDepth(T.getRchild());// 右子树的高度
            return 1 + (lDepth > rDepth ? lDepth : rDepth);// 返回左子树的高
                                                    度和右子树的高度中的最大值加 1
        }
        return 0;
    }
```

四、特别说明：

本次实验难度较大。实验内容中的第 2、3 题要求必须完成。第 4 题同学们根据个人能力选做。

实验五二叉树的应用

一、实验目的：

1、掌握二叉树的逻辑结构

2、掌握二叉树的二叉链表存储结构

二、实验内容：

实现哈夫曼编码

假设某文本文档只由英文字母组成。应用哈夫曼算法对该文档进行压缩和解压缩操作。

三、实现提示：

1、文档内容从键盘输入

2、设计哈夫曼算法的存储结构

3、设计哈夫曼编码和解码算法

实验七 图的遍历

一、实验目的：

1、掌握图的逻辑结构

2、掌握图的邻接矩阵存储结构

3、掌握图在邻接矩阵存储结构上遍历算法的实现

二、实验内容：

1、建立无向图的邻接矩阵存储

2、对建立的无向图，进行深度优先遍历

3、对建立的无向图，进行广度优先遍历

三、实现提示：

本次实验采用图的邻接矩阵存储，即用一维数组存储图中顶点的信息，用二维数组存储图中边的信息（即各个顶点之间的邻接关系）。

假设图 G＝(V，E)有 n 个顶点，则邻接矩阵是一个 n×n 的方阵，定义为：

$$arcs[i][j]=\begin{cases}1 & \text{若}(vi, vj)\in E\\ 0 & \text{其它}\end{cases}$$

设计实验用邻接矩阵类 MGragh，包括遍历操作。

```
package graph;
import java.util.Scanner;
import linkQueue.LinkQueue;
//图的邻接矩阵表示法
public class MGraph implements IGraph {
    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
    private static boolean[] visited;// 访问标志数组
    public MGraph() {
        this( 0, 0, null, null);
    }
    public MGraph(int vexNum, int arcNum, Object[] vexs, int[][] arcs) {
        this.vexNum = vexNum;
        this.arcNum = arcNum;
        this.vexs = vexs;
        this.arcs = arcs;
    }
    // 创建一个无向图
    public void createGraph() {
        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] = 0;
        System.out.println("请输入各个边的顶点:");
        for (int k = 0; k < arcNum; k++) {
            int v = locateVex(sc.next());
            int u = locateVex(sc.next());
            arcs[v][u] = arcs[u][v] = 1;
        }

    }
```

```
    // 给定顶点的值 vex，返回其在图中的位置，如果图中不包含此顶点，则返回-1
    public int locateVex(Object vex) {
        for (int v = 0; v < vexNum; v++)
            if (vexs[v].equals(vex))
                return v;
        return -1;
    }

    // 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];
    }

    // 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!");

        for (int j = 0; j < vexNum; j++)
            if (arcs[v][j] != 0 && arcs[v][j] < 9999)
                return j;

        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!");

        for (int j = w + 1; j < vexNum; j++)
            if (arcs[v][j] != 0 && arcs[v][j] < 9999)
                return j;

        return -1;
    }

    // 对图 G 做广度优先遍历
    public void BFSTraverse(IGraph 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;
        System.out.print(" 对图进行广度优先遍历的结果是 ");
        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 void BFS(IGraph 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);
                }
        }
    }

    // 对图 G 做深度优先遍历
    public void DFSTraverse(IGraph 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;
        System.out.print(" 对图进行深度优先遍历的结果是 ");
        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
    private    void DFS(IGraph 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);
    }
```

```
//显示图的邻接矩阵
public void displayARCS(MGraph G){
    System.out.println("该无向图的邻接矩阵如下");
    for (int i = 0; i < G.getVexNum(); i++)
        System.out.print("\t" + G.getVexs()[i]);
    System.out.println();
    for (int i = 0; i < G.getVexNum(); i++) {
        try {
            System.out.print(G.getVex(i) +"\t");
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        for (int j = 0; j < G.getVexNum(); j++)
            System.out.print(G.getArcs()[i][j] +"\t");
        System.out.println();
    }
    }
}
```

四、特别说明：

用 Object 类实现起来有困难的同学可以用整形数据或字符型数据作为操作对象来实现。具体算法参见课堂笔记及教材。

实验八 图的应用

一、实验目的：

1、掌握图的逻辑结构

2、掌握图的邻接表存储结构

3、掌握图在邻接表存储结构上遍历算法的实现

二、实验内容：

1、建立一个有向图的邻接表存储

2、对建立的有向图，进行深度优先遍历

3、对建立的有向图，进行广度优先遍历

4、假设图 G 采用邻接表存储，设计一个算法，输出图 G 中从顶点 u 到顶点 v 的长度为 l 的所有简单路径。

5、实现拓扑序列。

三、实现提示：

本次实验采用图的邻接表存储。在邻接表中存在两种结点结构，分别是顶点表结点和边表结点。

定义边表结点类

```
public class ArcNode {
    private int adjVex;// 该弧所指向的顶点位置
```

```
        private int value;// 边的权值
        private ArcNode nextArc;// 指向下一条弧
    }
```

定义结点表结点类

```
    public 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
    }
```

设计实验用的邻接表类 ALGraph，包括遍历操作

```
package graph;
import java.util.Scanner;
import linkQueue.LinkQueue;
import graph.IGraph;
//有向图的邻接表存储表示
public class ALGraph implements IGraph {
    private int vexNum, arcNum;// 图的当前顶点数和边数
    private VNode[] vexs;// 顶点
    private static boolean[] visited;// 访问标志数组
    public ALGraph(int vexNum, int arcNum, VNode[] vexs) {
        this.vexNum = vexNum;
        this.arcNum = arcNum;
        this.vexs = vexs;
    }

    // 创建有向图
    public void createGraph() {
        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 VNode[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] = new VNode(sc.next());
        System.out.println("请输入各条边的顶点:");
        for (int k = 0; k < arcNum; k++) {
            int v = locateVex(sc.next());// 弧尾
            int u = locateVex(sc.next());// 弧头
            int value = 1;
            addArc(v, u, value);
        }
    }
    // 在位置为 v、u 的顶点之间，添加一条弧，其权值为 value
```

```
public void addArc(int v, int u, int value) {
    ArcNode arc = new ArcNode(u, value);
    arc.setNextArc(vexs[v].getFirstArc());
    vexs[v].setFirstArc(arc);
}

// 返回顶点数
public int getVexNum() {
    return vexNum;
}

// 返回边数
public int getArcNum() {
    return arcNum;
}

// 给定顶点的值 vex，返回其在图中的位置，如果图中不包含此顶点，则返回-1
public int locateVex(Object vex) {
    for (int v = 0; v < vexNum; v++)
        if (vexs[v].getData().equals(vex))
            return v;
    return -1;
}

public VNode[] getVexs() {
    return 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;
    }
    // 对图 G 做广度优先遍历
    public void BFSTraverse(IGraph 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;
        System.out.print(" 对图进行广度优先遍历的结果是 ");
        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 void BFS(IGraph 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);
```

```
                }
            }
        }


    // 对图 G 做深度优先遍历
    public void DFSTraverse(IGraph 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;
        System.out.print(" 对图进行深度优先遍历的结果是 ");
        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
    private    void DFS(IGraph 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);
    }
    //显示有向图的邻接表
    public void displayALink(ALGraph G) throws Exception{
        for (int i = 0; i < G.getVexNum(); i++) {
            System.out.print(G.getVex(i) + ": ");
            for(int w = G.firstAdjVex(i); w != -1; w = G.nextAdjVex(i, w)){
                System.out.print(G.getVex(w) + "\t");
            }

            System.out.println();
        }
    }
}
```

四、特别说明：

用 Object 类实现起来有困难的同学可以用整形数据或字符型数据作为操作对象来实现。具体算法参见课堂笔记及教材。

实验九 排序技术

一、实验目的：

1、掌握直接插入排序的基本思想和实现方法

2、掌握起泡排序的基本思想和实现方法
3、掌握快速排序的基本思想和实现方法
4、掌握简单选择排序的基本思想和实现方法

二、实验内容：
1、对一组数据进行直接插入排序
2、对一组数据进行起泡排序
3、对一组数据进行快速排序
4、对一组数据进行简单选择排序

三、实现提示：

```
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
    public RecordNode[] getRecord() {
        return r;
    }
    public void setRecord(RecordNode[] r) {
        this.r = r;
    }
    public SeqList(){}
        // 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
    }
    // 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
        }
        public void display() {       //输出数组元素
            for (int i = 0; i < this.curlen; i++) {
                System.out.print(" " + r[i].getKey().toString());
            }
            System.out.println();
        }
        //【算法 7.1】 不带监视哨的直接插入排序算法
        public void insertSort() {
            RecordNode temp;
            int i, j;
        //       System.out.println("直接插入排序");
            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--) { //将前面比 r[i]大的记录向后移动
                    r[j + 1] = r[j];
                }
                r[j + 1] = temp;                //r[i]插入到第 j+1 个位置
        //          System.out.println("第" + i + "趟: ");
           //         display();
            }
        }
        //【算法 7.2】带监视哨的直接插入排序算法
        public void insertSortWithGuard() {

            int i, j;
            System.out.println("带监视哨的直接插入排序");
            for (i = 1; i < this.curlen; i++) {//n-1 趟扫描
                r[0] = r[i]; //将待插入的第 i 条记录暂存在 r[0]中，同时 r[0]为监视
哨
                for (j = i - 1; r[0].getKey().compareTo(r[j].getKey()) < 0; j--) { //将前
面较大元素向后移动
                    r[j + 1] = r[j];
                }
                r[j + 1] = r[0];                // r[i]插入到第 j+1 个位置
          //      System.out.println("第" + i + "趟: ");
          //      display();
            }
        }
        //【算法 7.3】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();
        }
    }
    //【算法 7.4】 冒泡排序算法
    public void bubbleSort() {
    //      System.out.println("冒泡排序");
        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;
                }
            }
         //     System.out.print("第" + i + "趟: ");
         //     display();
        }
    }
    //【算法 7.5】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
    }
    //【算法 7.6】 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
        }
    }
    //【算法 7.7】顺序表快速排序算法
    public void quickSort() {
        qSort(0, this.curlen - 1);
    }
    //【算法 7.8】直接选择排序
    public void selectSort() {
     //    System.out.println("直接选择排序");
        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;
            }
        //      System.out.print("第" + (i + 1) + "趟: ");
        //      display();
        }
    }
    //【算法 7.9】 树形选择排序(锦标赛排序)
    //建立树的顺序存储数组 tree,并对其排序，并将结果返回到 r 中
    void tournamentSort() {
        TreeNode[] tree;     //胜者树结点数组
        int leafSize = 1;   //胜者树的叶子结点数
        //得到胜者树叶子结点(外结点)的个数，该个数必须是 2 的幂
        while (leafSize < this.curlen) {
            leafSize *= 2;
        }
        int TreeSize = 2 * leafSize - 1;     //胜者树的所有节点数
        int loadindex = leafSize - 1;          //叶子结点(外结点)存放的起始位置
        tree = new TreeNode[TreeSize];
        int j = 0;
        //把待排序结点复制到胜者树的叶子结点中
        for (int i = loadindex; i < TreeSize; i++) {
            tree[i] = new TreeNode();
            tree[i].setIndex(i);
            if (j < this.curlen) {          //复制结点
                tree[i].setActive(1);
                tree[i].setData( r[j++]);
            } else {
                tree[i].setActive(0) ;     //空的外结点
            }
        }
        int i = loadindex;       //进行初始比较查找关键码最小结点
        while (i > 0) {          //产生胜者树
            j = i;
            while (j < 2 * i) {     //处理各对比赛者
                if (tree[j + 1].getActive() == 0 ||
```

```
((tree[j].getData()).getKey().compareTo((tree[j + 1].getData()).getKey())) <= 0) {
                        tree[(j - 1) / 2] = tree[j];          //左孩子(胜者)赋值给父结点
                    } else {
                        tree[(j - 1) / 2] = tree[j + 1];   //右孩子(胜者)赋值给父结点
                    }
                    j += 2;        //下一对比赛者
                }
                i = (i - 1) / 2;       //处理上层结点
            }
            for (i = 0; i < this.curlen - 1; i++) {    //处理剩余的 n-1 个记录
                r[i] = tree[0].getData();                    //将胜者树的根(最小者)存入数组 r
                tree[tree[0].getIndex()].setActive(0);   //该元素相应外结点不再比赛
                updateTree(tree, tree[0].getIndex());     //调整胜者树
            }
            r[this.curlen - 1] = tree[0].getData();
        }
        //【算法 7.10】树形选择排序的调整算法，i 是当前最小关键字记录的下标
        void updateTree(TreeNode[] tree, int i) {
            int j;
            if (i % 2 == 0) { //i 为偶数，对手为左结点
                tree[(i - 1) / 2] = tree[i - 1];
            } else { //i 为奇数，对手为右结点
                tree[(i - 1) / 2] = tree[i + 1];
            }
            i = (i - 1) / 2;     //最小元素输出后，其对手上升到父结点
            while (i > 0) {                //直到 i==0
                if (i % 2 == 0) {     //i 为偶数，对手为左结点
                    j = i - 1;
                } else {                    //i 为奇数，对手为右结点
                    j = i + 1;
                }
                //比赛对手中有一个为空
                if (tree[i].getActive() == 0 || tree[j].getActive() == 0) {
                    if (tree[i].getActive() == 1) {
                        tree[(i - 1) / 2] = tree[i];   //i 可参选，i 上升到父结点
                    } else {
                        tree[(i - 1) / 2] = tree[j];   //否则，j 上升到父结点
                    }
                } else //双方都可参选
                //关键码小者上升到父结点
                if ((tree[i].getData()).getKey().compareTo((tree[j].getData()).getKey())
<= 0) {
```

```
                tree[(i - 1) / 2] = tree[i];
            } else {
                tree[(i - 1) / 2] = tree[j];
            }
            i = (i - 1) / 2;        //i 上升到父结点
        }
    }
    //【算法 7.11】将以筛选法调整堆算法
    //将以 low 为根的子树调整成小顶堆，low、high 是序列下界和上界
    public void sift(int low, int high) {
        int i = low;                                    //子树的根
        int j = 2 * i + 1;                              //j 为 i 结点的左孩子
        RecordNode temp = r[i];
        while (j < high) {   //沿较小值孩子结点向下筛选
            if (j < high - 1 && r[j].getKey().compareTo(r[j + 1].getKey()) > 0) {
                j++; //数组元素比较,j 为左右孩子的较小者
            }
            if (temp.getKey().compareTo(r[j].getKey()) > 0) { //若父母结点值较
大
                r[i] = r[j];                //孩子结点中的较小值上移
                i = j;
                j = 2 * i + 1;
            } else {
                j = high + 1;               //退出循环
            }
        }
        r[i] = temp;                        //当前子树的原根值调整后的位置
    //      System.out.print("sift    " + low + ".." + high + "    ");
    //      display();
    }
    //【算法 7.12】 堆排序算法
    public void heapSort() {
       // System.out.println("堆排序");
        int n = this.curlen;
        RecordNode temp;
        for (int i = n / 2 - 1; i >= 0; i--) {//创建堆
            sift(i, n);
        }
        for (int i = n - 1; i > 0; i--) {//每趟将最小值交换到后面，再调整成堆
            temp = r[0];
            r[0] = r[i];
            r[i] = temp;
            sift(0, i);
        }
```

```
    }
    //【算法 7.13】两个有序序列的归并算法
    //把 r 数组中两个相邻的有序表 r[h]-r[m]和 r[m+1]-r[t]归并为一个有序表 order[h]-order[t]
    public void merge(RecordNode[] r, RecordNode[] order, int h, int m, int t) {
        int i = h, j = m + 1, k = h;
        while (i <= m && j <= t) {//将 r 中两个相邻子序列归并到 order 中
            if (r[i].getKey().compareTo(r[j].getKey()) <= 0) {//较小值复制到 order 中
                order[k++] = r[i++];
            } else {
                order[k++] = r[j++];
            }
        }
        while (i <= m) {//将前一个子序列剩余元素复制到 order 中
            order[k++] = r[i++];
        }
        while (j <= t) {//将后一个子序列剩余元素复制到 order 中
            order[k++] = r[j++];
        }
    }
    //【算法 7.14】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];
            }
        }
    }
    //【算法 7.15】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;
            }
        }
        public static void main(String[] args) throws Exception {

                int[] d = {52, 39, 67, 95, 70, 8, 25, 52};
                int[] dlta = {5, 3, 1};     //希尔排序增量数组
                int maxSize = 20;                          //顺序表空间大小
                SeqList L = new SeqList(maxSize); //建立顺序表
                for (int i = 0; i < d.length; i++) {
                    RecordNode r = new RecordNode(d[i]);
                    L.insert(L.length(), r);
                }
                System.out.println("排序前： ");
                L.display();
                System.out.println("请选择排序方法： ");
                System.out.println("1-直接插入排序");
                System.out.println("2-希尔排序");
                System.out.println("3-冒泡排序");
                System.out.println("4-快速排序");
                System.out.println("5-直接选择排序");
                System.out.println("6-堆排序");
                System.out.println("7-归并排序");
                Scanner s = new Scanner(System.in);
                int xz = s.nextInt();
                switch (xz) {
                    case 1:
                        L.insertSort();
                        break;                  //直接插入排序
                    case 2:
                        L.shellSort(dlta);
                        break;              //希尔排序
                    case 3:
                        L.bubbleSort();
```

```
                    break;                          //冒泡排序
                case 4:
                    L.quickSort();
                    break;                      //快速排序
                case 5:
                    L.selectSort();
                    break;                       //直接选择排序
                case 6:
                    L.heapSort();           //堆排序
                    break;
                case 7:
                    L.mergeSort();     //归并排序
                    break;
            }
            System.out.println("排序后：");
            L.display();
        }
}
```

实验十 查找

一、实验目的：

1、掌握折半查找算法的思想及程序实现。

2、掌握散列存储结构的思想，能选择合适的散列函数，实现不同冲突处理方法的散列表的查找、建立。

二、实验内容：

1. 建立有序表，采用折半查找实现某一已知的关键字的查找。
2. 利用折半查找算法在一个有序表中插入一个元素，并保持表的有序性。
3. 建立顺序表，统计表中重复元素个数。
4. 哈希表类设计。要求：
   (1)哈希函数采用除留余数法，哈希冲突解决方法采用链地址法；
   (2)设计一个侧试程序进行侧试。

三、实现提示：

. 1.哈希表项设计

```
enum KindOfItem｛Empty,Active，Deleted｝;
struct HashItem
{
    DataType data;
    KindOfItem info;
```

```
    HashItem(Kind0fItem I=Empty)：info（I）{}
    HashItem（const DataType &D,KindOfItemi=Empty)：data（D)，info（I）{}
    int operator==（HashItem &a）
        {return data==a．data;}
    int operator！=（HashItem &a）
        {return data！=a．data;}
};
```

2.哈希表类设计

```
class HashTable
private：
    HashItein   ht；//哈希表数组
    int TableSize; //哈希表的长度（即 m）
    int currentSize；//当前的表项个数
public：
    HashTable（int m)；//构造函数
    -HashTable(void) //析构函数
        {delete[]ht;}

int Find（const DataType &x）const；//查找
int Insert（const DataType &x)；//插人
int Delete(const DataType &x) //删除
int IsIn(const DataType &x) //是否已存在
{int i=Find(x)；return i>=0？1:0；}
DataType GetValue(int i)const//取数据元素
{return ht[i].data;}
};
```

注意问题

1．注意理解折半查找的适用条件（链表能否实现折半查找？）。

2. 注意理解静态查找、动态查找概念。

3．比较各种查找算法的各自特点，能够根据实际情况选择合适的查找方法。