目 录

第一章　内容介绍和授课方式

1.1　数据结构和算法内容介绍

1.1.1　经典算法面试题

1. 字符串匹配问题：KMP 算法
2. 汉诺塔移动问题：分治思想。
3. 八皇后问题：回溯算法、分治算法。
4. 马踏棋盘：深度优化遍历算法（DFS）+ 贪心算法

第二章　稀疏 sparsearray 数组和队列

2.1　稀疏数组

2.1.1　应用场景

编写的五子棋程序中，有存盘退出和续上盘的功能
该二维数组很多值为默认值 0，记录了很多没有意义的数据，使用稀疏数组。

2.1.2　稀疏数组的处理方法

（1）第一行记录数组一共有几行几列，有多少个不同的值
（2）把具有不同值的元素的行列及值记录在一个小规模的数组中，从而缩小程序的规模。

2.1.3　二维数组转稀疏数组

1. 遍历原始数组，得到有效数据的个数 `sum`.
2. 根据 `sum` 就可以创建稀疏数组 `sparseArr`，其规模为 `int[sum + 1][3]`
3. 稀疏数组的第一行存入原始数组的行数、列数、有效数据个数。
4. 将二维数组的有效数据存入到稀疏数组其余行中。

2.1.4　稀疏数组转二维数组

1. 先读取稀疏数组的首行，根据该数据创建原始数组。
2. 在读取稀疏数组后几行的数据，并赋给原始的二维数组。

2.1.5　代码实现

```
package com.atWSN.sparsearray;

public class SparseArray {
    public static void main(String[] args) {
        //创建一个原始的二维数组 11 * 11
        //0表示没有棋子，1表示黑子，2表示蓝子
        int chessArray1[][] = new int[11][11];
        chessArray1[1][2] = 1;
        chessArray1[2][3] = 2;
        chessArray1[7][8] = 2;
        System.out.println("原始的二维数组: ");
        for (int[] row: chessArray1) {
            for (int data : row) {
                System.out.printf("%d\t",data);
            }
            System.out.println();
        }

        //把二维数组转为稀疏数组
        //1.遍历二维数组，得到非零数据的个数
        int sum = 0;
```

```
        for (int[] row: chessArray1) {
            for (int data : row) {
                if (data != 0){
                    sum++;
                }
            }
        }
        System.out.println("sum = " + sum);

        //创建对应的稀疏数组
        int sparseArray[][] = new int[sum + 1][3];
        sparseArray[0][0] = 11;
        sparseArray[0][1] = 11;
        sparseArray[0][2] = sum;
        int count = 0;
        for (int i = 0; i < 11; i++) {
            for (int j = 0; j < 11; j++) {
                if (chessArray1[i][j] != 0){
                    count++;
                    sparseArray[count][0] = i;
                    sparseArray[count][1] = j;
                    sparseArray[count][2] = chessArray1[i][j];
                }
            }
        }
        //输出稀疏数组
        System.out.println("输出稀疏数组");
        for (int[] row: sparseArray) {
            for (int data:row) {
                System.out.printf("%d\t",data);
            }
            System.out.println();
        }
        //稀疏数组转二维数组
        System.out.println("--------------------------");
        System.out.println("稀疏数组转二维数组:");
        int chessArray2[][] = new int[sparseArray[0][0]][sparseArray[0][1]];
        //遍历稀疏数组，把稀疏数组中存的信息还原回去
        for (int i = 1; i <= sparseArray[0][2]; i++) {
            chessArray2[sparseArray[i][0]][sparseArray[i][1]] = sparseArray[i][2];
        }

        for (int[] row: chessArray2) {
            for (int data : row) {
                System.out.printf("%d\t",data);
            }
            System.out.println();
        }
    }
}
```

2.2 队列

2.2.1 介绍

队列是一个有序列表，可以使用数组或链表来实现。

遵循先进先出原则

2.2.2 数组模拟队列

入队操作：`addQueue`

1. 尾指针向后移:`rear + 1`，当`front == rear`时，队列为空。

2. 若尾指针`rear`小于队列的最大下标`maxSize - 1`,则将数组存入`rear`所指的数组元素中，否则无法存入数据.`rear == maxSize - 1`时，队列满。

```
package com.atWSN.queue;

import java.util.Scanner;

public class ArrayQueueDemo {
    public static void main(String[] args) {
        //测试
        ArrayQueue arrayQueue = new ArrayQueue(3);
        char key = ' ';//接收用户输入
        Scanner scanner = new Scanner(System.in);
        boolean loop = true;
        while(loop){
            System.out.println("s(show): 显示队列");
            System.out.println("e(exit): 退出程序");
            System.out.println("a(add): 添加数据到队列");
            System.out.println("g(get): 从队列取出数据");
            System.out.println("h(head): 查看队列头的数据");
            key = scanner.next().charAt(0);
            switch(key){
                case 'a':
                    System.out.println("请输入一个数据: ");
                    arrayQueue.addQueue(scanner.nextInt());
                    break;
                case 's':
                    arrayQueue.showQueue();
                    break;
                case 'e':
                    scanner.close();
                    loop = false;
                    System.out.println("程序退出! ");
                    break;
                case 'g':
                    try{
                        int ret =arrayQueue.getQueue();
                        System.out.println("取出的数据是: " + ret);;
                    }catch(Exception e){
```

```
37                          System.out.println(e.getMessage());
38                      }
39                      break;
40                  case 'h':
41                      try{
42                          int ret =arrayQueue.headQueue();
43                          System.out.println("头部的数据是: " + ret);;
44                      }catch(Exception e){
45                          System.out.println(e.getMessage());
46                      }
47                      break;
48                  default:
49                      break;
50              }
51          }
52      }
53  }
54
55  //使用数组模拟队列-编写一个ArrayQueue类
56  class ArrayQueue{
57      //表示数组的最大容量
58      private int maxSize;
59      //指向队首的指针
60      private int front;
61      //指向队尾的指针
62      private int rear;
63      private int[] arr;
64      public ArrayQueue(int maxSize){
65          this.maxSize = maxSize;
66          arr = new int[maxSize];
67          front = -1;//指向队列头部，指向队列头的前一个位置
68          rear = -1;//指向队列尾，指向队列的最后一个数据
69      }
70      //判断队列是否为满
71
72      public boolean isFull(){
73          return rear == maxSize - 1;
74      }
75      //判断队列是否为空
76      public boolean isEmpty(){
77          return rear == front;
78      }
79
80      //添加数据到队列
81      public void addQueue(int n){
82          //判断队列是否满
83          if (isFull()){
84              System.out.println("队列满！不能加入！");
85              return;
86          }
87          arr[++rear] = n;
```

```
    }
    //获取队列的数据，出队列
    public int getQueue(){
        if (isEmpty()){
            throw new RuntimeException("队列空！不能取数据！");
        }
        return arr[++front];
    }
    public void showQueue(){
        if(isEmpty()){
            System.out.println("队列为空！");
            return;
        }
        System.out.print("[");
        for (int i = front + 1; i <= rear; i++) {
            System.out.print(arr[i]);
            if (i!=rear){
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }

    //显示队列头部的数据，不是取数据
    public int headQueue(){
        if (isEmpty()){
            throw new RuntimeException("队列为空！");
        }
        return arr[front + 1];
    }
}
```

上述队列的问题分析及优化：

1. 目前数组使用一次就不能复用了，没有达到复用的效果。
2. 将这个数组使用算法，改进成一个环形的队列。（取模）

2.2.3 数组模拟环形队列

思路：

1. `front` 变量的含义做一个调整：`front` 指向队列的第一个元素。即 `arr[front]` 是队列的第一个元素。`front` 初始值为 0 。2. `rear` 变量的含义做一个调整：`rear` 指向队列的最后一个元素的后一个位置。因为希望空出一个空间作为约定。`rear` 初始值为 0

3. 队列满时：条件是 `(rear + 1)% maxSize == front`
4. 队列空：`rear == front`
5. 队列中有效数据的个数：`(rear - front + maxSize)%maxSize`

2.2.4 环形队列的代码实现

```
class CircleArrayQueue{
```

```
    //表示数组的最大容量
    private int maxSize;
    //指向队首的指针
    private int front;
    //指向队尾的指针
    private int rear;
    private int[] arr;
    public CircleArrayQueue(int maxSize){
        this.maxSize = maxSize + 1;//要空出一个空间，所以要+1
        arr = new int[this.maxSize];
        front = 0;//指向队列头部，指向队列第一个元素
        rear = 0;//指向队列尾，指向队列的最后一个数据的下一个位置
    }

    //判断队列是否为满
    public boolean isFull(){
        //数组的一个空间是不存元素的
        //假设front在开头，即下标为0；rear在队列的最后，也就是数组的最后一个位置，
            下标为maxSize - 1
        //此时队列是满的，有(rear + 1) % maxSize == front
        return (rear + 1) % maxSize  == front;
    }

    //判断队列是否为空
    public boolean isEmpty(){
        return rear == front;
    }
    //添加数据到队列
    public void addQueue(int n){
        //判断队列是否满
        if (isFull()){
            System.out.println("队列满！不能加入！");
            return;
        }
        arr[rear] = n;
        rear = (rear + 1) % maxSize;
    }

    //获取队列的数据，出队列
    public int getQueue(){
        if (isEmpty()){
            throw new RuntimeException("队列空！不能取数据！");
        }
        //1.先把front对应的值保留到临时变量中
        //2.把front后移
        //3.返回临时变量保存的值
        int ret =  arr[front];
        front = (front + 1) % maxSize;
        return ret;
    }

```

```
52      public void showQueue(){
53          if(isEmpty()){
54              System.out.println("队列为空！");
55              return;
56          }
57          System.out.print("[");
58          for (int i = front; i < front + getSize(); i++) {
59              System.out.print(arr[i % maxSize]);
60              if (i!=front + getSize() - 1){
61                  System.out.print(", ");
62              }
63          }
64          System.out.println("]");
65      }
66      //求出当前队列有效数据的个数
67
68
69      public int getSize() {
70          return (rear + maxSize - front) % maxSize;//rear可能会位于front的前边，此
                时为负数，所以要加上maxSize
71      }
72
73      //显示队列头部的数据，不是取数据
74      public int headQueue(){
75          if (isEmpty()){
76              throw new RuntimeException("队列为空！");
77          }
78          return arr[front];
79      }
80  }
```

第三章　链表

链表：

1. 以节点的方式来存储；
2. 每个节点包含 data 域，next 域（指向下一个节点）

3.1　单链表

3.1.1　代码实现

```
public class SingleLinkedListDemo {
    public static void main(String[] args) {
        HeroNode heroNode1 = new HeroNode(1, "宋江", "及时雨");
        HeroNode heroNode2 = new HeroNode(2, "卢俊义", "玉麒麟");
        HeroNode heroNode3 = new HeroNode(3, "吴用", "智多星");
        HeroNode heroNode4 = new HeroNode(4, "林冲", "豹子头");


        SingleLinkedList singleLinkedList1 = new SingleLinkedList();
//          singleLinkedList1.list();
        singleLinkedList1.addByOrder(heroNode1);
//          singleLinkedList1.list();
        singleLinkedList1.addByOrder(heroNode2);
        singleLinkedList1.addByOrder(heroNode4);
//          singleLinkedList1.list();
        singleLinkedList1.addByOrder(heroNode3);
        singleLinkedList1.list();
        HeroNode newHeroNode = new HeroNode(2, "李逵", "黑旋风");
        singleLinkedList1.update(newHeroNode);
        singleLinkedList1.list();
        singleLinkedList1.delete(heroNode1);
        singleLinkedList1.list();
        singleLinkedList1.delete(heroNode2);
        singleLinkedList1.list();
        singleLinkedList1.delete(heroNode3);
        singleLinkedList1.list();
        singleLinkedList1.delete(heroNode1);
        singleLinkedList1.delete(heroNode4);
        singleLinkedList1.list();
        singleLinkedList1.delete(heroNode1);
        singleLinkedList1.list();
    }
}

//定义SingleLinkedList 管理我们的英雄
class SingleLinkedList{
    //先初始化一个头结点
    private HeroNode head =  new HeroNode();

```

```
40      //尾插
41      public void add(HeroNode heroNode){
42          HeroNode curNode = head;
43          while(true){
44              if (curNode.next == null) {
45                  break;
46              }
47              curNode = curNode.next;
48          }
49          curNode.next = heroNode;
50      }
51
52      //方式2: 按编号的顺序添加
53      //如果编号存在, 抛出异常
54      public void addByOrder(HeroNode heroNode){
55          HeroNode curNode = head;
56          boolean flag = false;//标识添加的编号是否存在, 默认不存在
57          while(true){
58              if (curNode.next == null){//说明curNode在链表的最后
59                  break;
60              }
61              if(curNode.next.getNum() > heroNode.getNum() ) {//位置找到了, 就在
                    curNode的后边
62                  break;
63              }else if (curNode.next.getNum() == heroNode.getNum()){//说明编号存在
64                  System.out.println("节点" + heroNode.getNum() + "已存在在链表中! "
                        );
65                  System.out.println("添加失败! ");
66                  return;
67              }
68              curNode = curNode.next;
69          }
70          heroNode.next = curNode.next;
71          curNode.next = heroNode;
72      }
73      //显示链表
74      public void list(){
75          if (head.next == null){
76              System.out.println("[]");
77              return;
78          }
79          HeroNode curNode = head.next;
80          System.out.print("[");
81          while (true){
82              if (curNode == null){
83                  break;
84              }
85              System.out.print(curNode.toString());
86              if(curNode.next != null){
87                  System.out.print(",");
88              }
```

```
89              curNode = curNode.next;
90          }
91          System.out.println("]");
92      }
93
94      //修改链表中节点的信息：根据编号来改变名字和昵称
95      public void update(HeroNode heroNode){//根据heraNode的num来修改
96          if (head.next == null){
97              System.out.println("链表中无节点，无法修改！");
98              System.out.println("修改失败！");
99              return;
100         }
101         HeroNode curNode = head.next;
102         while (curNode != null){
103             if (curNode.getNum() == heroNode.getNum()){
104                 curNode.setName(heroNode.getName());
105                 curNode.setNickName(heroNode.getNickName());
106                 return;
107             }
108             curNode = curNode.next;
109         }
110         System.out.println("没有找到相关节点，无法修改！");
111         System.out.println("修改失败！");
112     }
113
114     //删除节点
115     public void delete(HeroNode heroNode){
116         if (head.next == null){
117             System.out.println("链表中无节点，无法删除！");
118             System.out.println("删除失败！");
119             return;
120         }
121         HeroNode curNode = head;
122         while (curNode.next != null){
123             if (curNode.next.getNum() == heroNode.getNum()){
124                 curNode.next = curNode.next.next;
125                 return;
126             }
127             curNode = curNode.next;
128         }
129         System.out.println("链表中无该节点，无法删除！");
130         System.out.println("删除失败！");
131     }
132 }
133
134 //定义HeroNode，每个HeroNode对象就是一个节点
135 class HeroNode{
136     private int num;
137     private String name;
138     private String nickName;
139     public HeroNode next;
```

```
140
141     //构造器
142     //无参版
143     public HeroNode(){
144
145     }
146     //有参版
147     public HeroNode(int num,String name,String nickName){
148         this.num = num;
149         this.name = name;
150         this.nickName = nickName;
151     }
152
153     public int getNum() {
154         return this.num;
155     }
156
157     public String getName() {
158         return name;
159     }
160
161     public String getNickName() {
162         return nickName;
163     }
164
165     public void setNum(int num) {
166         this.num = num;
167     }
168
169     public void setName(String name) {
170         this.name = name;
171     }
172
173     public void setNickName(String nickName) {
174         this.nickName = nickName;
175     }
176
177     @Override
178     public String toString() {
179         return "HeroNode{" +
180                 "num=" + getNum() +
181                 ", name='" + name + '\'' +
182                 ", nickName='" + nickName + '\'' +
183                 '}';
184     }
185 }
```

3.1.2 单链表的面试题

```java
import java.util.Random;
import java.util.Scanner;
import java.util.Stack;

public class InterviewTest {
    public static void main(String[] args) {
        Node head = new Node();
        Node cur = head;
        for (int i = 1; i <= 3; i++) {
            cur.next = new Node(i);
            cur = cur.next;
        }
        //求链表的有效节点长度
        //带傀儡节点版
        System.out.println("-------------------------------------");
        System.out.println(getLength1(head));
        //不带傀儡节点版
        System.out.println("-------------------------------------");
        System.out.println(getLength2(head.next));
        System.out.println("-------------------------------------");
//        System.out.println("您要求倒数第几个节点？");
//        Scanner scanner = new Scanner(System.in);
        int num = 1;
        System.out.println("求倒数第" + num +"个节点：方法一（头结点为有效节点的前一个）");
        Node ret = findLastIndexNode1(head,num);
        if (ret != null) {
            System.out.println(ret.num);
        }else{
            System.out.println("ret指向为null");
        }
        System.out.println("-------------------------------------");
        System.out.println("求倒数第" + num +"个节点：方法一（头结点为有效节点）");
        Node ret1 = findLastIndexNode2(head.next,num);
        if (ret1 != null) {
            System.out.println(ret1.num);
        }else{
            System.out.println("ret1指向为null");
        }

        System.out.println("-------------------------------------");
        System.out.println("求倒数第" + num +"个节点：方法二（头结点下一个为有效节点）快慢指针1");
        Node ret3 = findLastIndexNode3(head,num);
        if (ret3 != null) {
            System.out.println(ret3.num);
        }else{
            System.out.println("ret3指向为null");
        }

```

```
49          System.out.println("-----------------------------------");
50          System.out.println("求倒数第" + num +"个节点：方法二（头结点下一个为有效节
                点）快慢指针2");
51          Node ret4 = findLastIndexNode4(head,num);
52          if (ret4 != null) {
53              System.out.println(ret4.num);
54          }else{
55              System.out.println("ret4指向为null");
56          }
57
58          System.out.println("----------------------------");
59          head = reverse1(head);
60          printLinkedList(head.next);
61          System.out.println("----------------------------");
62          head = reverse2(head.next);
63          printLinkedList(head);
64          System.out.println("----------------------------");
65          reversePrint(head);
66          System.out.println("----------------------------");
67          System.out.println("合并有序链表");
68          Node head1 = new Node();
69          Node cur1 = head1;
70          Scanner scanner = new Scanner(System.in);
71          Random random = new Random();
72          System.out.println("请输入head1的长度");
73          int i = scanner.nextInt();
74          System.out.println("请输入head1的初始值");
75          int number = scanner.nextInt();;
76          Node tmp = null;
77          head1.num = number;
78
79          while (i > 0){
80              cur1.next = new Node(cur1.num + random.nextInt(10) + 1);
81              cur1 = cur1.next;
82              i--;
83          }
84          Node head2 = new Node();
85          cur1 = head2;
86          System.out.println("请输入head2的长度");
87          i = scanner.nextInt();
88          System.out.println("请输入head2的初始值");
89          number = scanner.nextInt();;
90          head2.num = number;
91          while (i > 0){
92              cur1.next = new Node(cur1.num + random.nextInt(10) + 1);
93              cur1 = cur1.next;
94              i--;
95          }
96          System.out.println("打印链表1: ");
97          printLinkedList(head1);
98          System.out.println("打印链表2: ");
```

```
99          printLinkedList(head2);
100         Node mergeNode = mergeLinkedList(head1,head2);
101         System.out.println("打印合并后的链表: ");
102         if (mergeNode != null){
103             printLinkedList(mergeNode);
104         }else{
105             System.out.println("合并的链表为空");
106         }
107     }
108     //1.求单链表中有效节点的个数
109     //如果是带头结点的, 头结点要去掉
110
111     /**
112      *
113      * @param head 是链表的头结点（第一个有效节点前的那个节点）
114      * @return 返回有效节点的个数
115      */
116     public static int getLength1(Node head){
117         if (head.next == null){
118             return 0;
119         }
120         int sum = 0;
121         Node curNode = head.next;
122         while (curNode != null){
123             sum++;
124             curNode = curNode.next;
125         }
126         return sum;
127     }
128
129     /**
130      *
131      * @param head 是链表第一个有效节点
132      * @return 返回有效节点的个数
133      */
134     public static int getLength2(Node head){
135         if (head == null){
136             return 0;
137         }
138         int sum = 0;
139         Node curNode = head;
140         while (curNode != null){
141             sum++;
142             curNode = curNode.next;
143         }
144         return sum;
145     }
146
147     //查找单链表的倒数第K个节点
148
149     //方法一:
```

```
150     //1.写一个方法接收head好项目和K
151     //2.先遍历链表得到链表的长度sum
152     //3.从链表的头部开始遍历找链表的第sum - k个节点即可
153     //4.找到后就依题意返回，这里返回对应节点的引用
154
155     /**
156      *
157      * @param head 是链表的头结点（第一个有效节点前的那个节点）
158      * @return 返回对应节点的引用,没有则返回null
159      */
160     public static Node findLastIndexNode1(Node head,int k){
161         if (head.next == null){
162             return null;
163         }
164         int sum  = getLength1(head);
165         if(k <= 0 || k > sum){
166             return null;
167         }
168         Node curNode = head.next;
169         for (int i = 0; i < sum - k; i++) {
170             curNode = curNode.next;
171         }
172         return curNode;
173     }
174     /**
175      *
176      * @param head 是链表的第一个有效节点
177      * @return 返回对应节点的引用,没有则返回null
178      */
179     public static Node findLastIndexNode2(Node head,int k){
180         if (head == null){
181             return null;
182         }
183         int sum  = getLength2(head);
184         if(k <= 0 || k > sum){
185             return null;
186         }
187         Node curNode = head;
188         for (int i = 0; i < sum - k; i++) {
189             curNode = curNode.next;
190         }
191         return curNode;
192     }
193
194     //方法二：使用快慢指针1
195     //不求链表的长度
196     //快慢指针同时指向第一个有效数据节点，快指针向后移动k - 1步，走到第k个节点。
197     //快慢指针同时向后走，直到慢指针走到链表最后一个节点
198
199     /**
200      *
```

```
201         * @param head
202         * @param k
203         * @return
204         */
205        public static Node findLastIndexNode3(Node head,int k){
206            if (head.next == null){
207                return null;
208            }
209            if(k <= 0){
210                return null;
211            }
212            Node fast = head.next;
213            Node slow = head.next;
214            for (int i = 0; i < k - 1; i++) {
215                if(fast == null){
216                    return null;
217                }
218                fast = fast.next;
219            }
220            if (fast == null){
221                return null;
222            }
223            while(fast.next != null){
224                fast = fast.next;
225                slow = slow.next;
226            }
227            return slow;
228        }
229
230        //方法二：使用快慢指针1
231        //不求链表的长度
232        //快慢指针同时指向头节点（有效数据的前一个节点），快指针移动k步。
233        //快慢指针同时向后走，直到快指针指向为null
234
235        /**
236         *
237         * @param head
238         * @param k
239         * @return
240         */
241        public static Node findLastIndexNode4(Node head,int k){
242            if (head.next == null){
243                return null;
244            }
245            if(k <= 0){
246                return null;
247            }
248            Node fast = head;
249            Node slow = head;
250            for (int i = 0; i < k; i++) {
251                if(fast == null){
```

```
252                 return null;
253             }
254             fast = fast.next;
255         }
256         if (fast == null){
257             return null;
258         }
259         while(fast != null){
260             fast = fast.next;
261             slow = slow.next;
262         }
263         return slow;
264     }
265 
266     //单链表翻转
267     /**
268      * 方法一：头插法
269      * @author 王松年
270      * @param head 的下一个节点是有效节点
271      * @return 新的头结点
272      */
273     public static Node reverse1(Node head){
274         if(head.next == null || head.next.next == null){
275             return head;
276         }
277         Node newHead = new Node();
278         Node curNode = head.next;
279         Node nextNode = curNode.next;
280         while (curNode != null){
281             nextNode = curNode.next;
282             curNode.next = newHead.next;
283             newHead.next = curNode;
284             curNode = nextNode;
285         }
286         return newHead;
287     }
288 
289     /**
290      * 方法一：头插法
291      * @author 王松年
292      * @param head 是有效节点
293      * @return 新的头结点
294      */
295     public static Node reverse2(Node head){
296         if(head == null || head.next == null){
297             return head;
298         }
299         Node newHead = new Node();
300         Node curNode = head;
301         Node nextNode = curNode.next;
302         while (curNode != null){
```

```
303             nextNode = curNode.next;
304             curNode.next = newHead.next;
305             newHead.next = curNode;
306             curNode = nextNode;
307         }
308         return newHead.next;
309     }
310 
311     /**
312      * 方法二：三指针
313      * @author 王松年
314      * @param head 是有效节点
315      * @return 新的头结点
316      */
317     public static Node reverse3(Node head){
318         if(head == null || head.next == null){
319             return head;
320         }
321         Node preNode = null;
322         Node curNode = head;
323         Node nextNode = curNode.next;
324         while(curNode != null){
325             nextNode = curNode.next;
326             curNode.next = preNode;
327             preNode = curNode;
328             curNode = nextNode;
329         }
330         return preNode;
331     }
332 
333     //从尾到头打印单链表
334 
335     /**
336      * head为有效数据
337      */
338     //注意不要改变原有链表的结构（即不能翻转链表）
339     //方法：使用栈
340     public static void reversePrint(Node head){
341         if (head == null){
342             System.out.println("[]");
343             return;
344         }
345         Stack<Integer> stack = new Stack<>();
346         while (head != null){
347             stack.add(head.num);
348             head = head.next;
349         }
350         System.out.print("[");
351         while (!stack.isEmpty()){
352             System.out.print(stack.pop());
353             if ((stack.size()>=1)){
```

```
354                 System.out.print(",");
355             }
356         }
357         System.out.println("]");
358     }
359 
360     //合并两个有序的链表，要求合并后的链表依旧有序
361 
362     /**
363      *
364      * @param head1 第一个有效节点
365      * @param head2 第一个有效节点
366      * @return
367      */
368     public static Node mergeLinkedList(Node head1,Node head2){
369         Node newHead = new Node();
370         Node cur1 = head1;
371         Node cur2 = head2;
372         Node cur = newHead;
373         if (head1 == null && head2 == null){
374             return null;
375         }
376         if (cur1 == null){
377             return head2;
378         }
379         if(cur2 == null){
380             return head1;
381         }
382         while(cur1 != null && cur2 != null){
383             if (cur1.num <= cur2.num){
384                 cur.next = cur1;
385                 cur = cur.next;
386                 cur1 = cur1.next;
387             }else{
388                 cur.next = cur2;
389                 cur = cur.next;
390                 cur2 = cur2.next;
391             }
392         }
393         if (cur1 != null){
394             cur.next = cur1;
395         }
396         if (cur2 != null){
397             cur.next = cur2;
398         }
399         return newHead.next;
400     }
401 
402 
403     //显示链表:第一个节点为有效节点
404     public static void printLinkedList(Node head) {
```

```
405         System.out.print("[");
406         while (head != null){
407             System.out.print(head.num);
408             if (head.next != null){
409                 System.out.print(",");
410             }
411             head = head.next;
412         }
413         System.out.println("]");
414     }
415 }
416
417
418
419
420 class Node{
421     public int num;
422     public Node next;
423
424     public Node(int num) {
425         this.num = num;
426     }
427     public Node() {
428         //TODO
429     }
430 }
```

3.2 双向链表

3.2.1 特点

既有 `next`（指向下一个节点），也有 `pre`（指向前一个节点）

可以自我删除。

遍历：和单链表一致，只不过可以向前查找，也可以向后查找

添加：末尾添加

先找到双向链表的最后一个节点，把新节点的 `pre` 指向最后一个节点，把最后一个节点的 `next` 指向新的节点

添加：中间添加，添加到某个节点的后边

把新节点的 `pre` 指向这个节点，新节点的 `next` 指向该节点的 `next`，把该节点后边那个节点的 `pre` 指向新节点，该节点的 `next` 指向新节点

修改：与单向链表一致

删除：中间节点

因为是双向链表，可以直接找到待删除节点。把该节点后边的那个节点的 `pre` 指向该节点前边的那个节点：`delNode.next.pre = delNode.pre`，把该节点前的节点的 `next` 指向该节点的下一个节点：`delNode.pre.next = delNode.n`

3.2.2 代码实现

```
public class DoubleLinkedListDemo {
    public static void main(String[] args) {
        System.out.println("--------------------------------------");
        System.out.println("双向链表的测试: ");
        HeroNode2 node1 = new HeroNode2(1,"宋江","及时雨");
        HeroNode2 node2 = new HeroNode2(2,"卢俊义","玉麒麟");
        HeroNode2 node3 = new HeroNode2(3,"吴用","智多星");
        HeroNode2 node4 = new HeroNode2(4,"林冲","豹子头");
        DoubleLinkedList doubleLinkedList1 = new DoubleLinkedList();
        //添加
        doubleLinkedList1.add(node1);
        doubleLinkedList1.add(node2);
        //doubleLinkedList1.add(node3);
        doubleLinkedList1.add(node4);
        doubleLinkedList1.list();
//        //修改
//        HeroNode2 newNode = new HeroNode2(2, "公孙胜", "入云龙");
//        doubleLinkedList1.update(newNode);
//        System.out.println("--------------------------------------");
//        System.out.println("双向链表的测试: (修改链表内容)");
//        doubleLinkedList1.list();
//        //删除
//        doubleLinkedList1.delete(node2);
//        doubleLinkedList1.list();
//        doubleLinkedList1.delete(node4);
//        doubleLinkedList1.list();
        //
        System.out.println("--------------------------------------");
        System.out.println("测试: 添加相同元素");
        HeroNode2 addNode1 = new HeroNode2(4,"林冲","豹子头");
        doubleLinkedList1.addByOrder(addNode1);
        doubleLinkedList1.list();
        //HeroNode2 addNode1 = new HeroNode2(4,"林冲","豹子头");
        System.out.println("--------------------------------------");
        System.out.println("往中间添加元素: ");
        doubleLinkedList1.addByOrder(node3);
        doubleLinkedList1.list();
        System.out.println("--------------------------------------");
        System.out.println("往末尾添加元素: ");
        HeroNode2 node5 = new HeroNode2(5,"鲁智深","花和尚");
        doubleLinkedList1.addByOrder(node5);
        doubleLinkedList1.list();
        System.out.println("--------------------------------------");
        System.out.println("往开头添加元素: ");
        HeroNode2 node0 = new HeroNode2(0,"0","0");
        doubleLinkedList1.addByOrder(node0);
        doubleLinkedList1.list();
    }
}

```

```
51 //双向链表类
52 class DoubleLinkedList {
53     //初始化
54     private HeroNode2 head = new HeroNode2();
55 
56     public HeroNode2 getHead() {
57         return head;
58     }
59 
60     //遍历双向链表
61     //显示链表
62     public void list() {
63         if (head.next == null) {
64             System.out.println("[]");
65             return;
66         }
67         HeroNode2 curNode = head.next;
68         System.out.print("[");
69         while (true) {
70             if (curNode == null) {
71                 break;
72             }
73             System.out.print(curNode.toString());
74             if (curNode.next != null) {
75                 System.out.print(",");
76             }
77             curNode = curNode.next;
78         }
79         System.out.println("]");
80     }
81 
82     //尾插
83     public void add(HeroNode2 heroNode) {
84         HeroNode2 curNode = head;
85         while (curNode.next != null) {
86             curNode = curNode.next;
87         }
88         curNode.next = heroNode;
89         heroNode.pre = curNode;
90     }
91 
92     //方式2: 按编号的顺序添加
93     //如果编号存在, 抛出异常
94     public void addByOrder(HeroNode2 heroNode){
95         HeroNode2 curNode = head.next;
96         while(curNode != null && curNode.getNum() < heroNode.getNum()){
97             curNode = curNode.next;
98         }
99         if(curNode == null){
100             add(heroNode);
101             return;
```

```
102         }
103         if (curNode.getNum() == heroNode.getNum()){
104             System.out.println("链表中已有数据，无法添加。");
105             return;
106         }
107         heroNode.next = curNode;
108         heroNode.pre = curNode.pre;
109         curNode.pre.next = heroNode;
110         curNode.pre = heroNode;
111     }
112 
113     //修改链表中节点的信息：根据编号来改变名字和昵称
114     public void update(HeroNode2 heroNode) {//根据heraNode的num来修改
115         if (head.next == null) {
116             System.out.println("链表中无节点，无法修改！");
117             System.out.println("修改失败！");
118             return;
119         }
120         HeroNode2 curNode = head.next;
121         while (curNode != null) {
122             if (curNode.getNum() == heroNode.getNum()) {
123                 curNode.setName(heroNode.getName());
124                 curNode.setNickName(heroNode.getNickName());
125                 return;
126             }
127             curNode = curNode.next;
128         }
129         System.out.println("没有找到相关节点，无法修改！");
130         System.out.println("修改失败！");
131     }
132 
133     public void delete(HeroNode2 heroNode) {
134         if (head.next == null) {
135             System.out.println("链表中无节点，无法删除！");
136             System.out.println("删除失败！");
137             return;
138         }
139         HeroNode2 curNode = head.next;
140         while (curNode != null) {
141             if (curNode.getNum() == heroNode.getNum()) {
142                 curNode.pre.next = curNode.next;
143                 if (curNode.next != null) {
144                     curNode.next.pre = curNode.pre;
145                 }
146                 return;
147             }
148             curNode = curNode.next;
149         }
150         System.out.println("链表中无该节点，无法删除！");
151         System.out.println("删除失败！");
152     }
```

```
153 }
154
155 class HeroNode2 {
156     private int num;
157     private String name;
158     private String nickName;
159     public HeroNode2 pre;
160     public HeroNode2 next;
161
162
163     //构造器
164     //无参版
165     public HeroNode2() {
166
167     }
168
169     //有参版
170     public HeroNode2(int num, String name, String nickName) {
171         this.num = num;
172         this.name = name;
173         this.nickName = nickName;
174     }
175
176     public int getNum() {
177         return this.num;
178     }
179
180     public String getName() {
181         return name;
182     }
183
184     public String getNickName() {
185         return nickName;
186     }
187
188     public void setNum(int num) {
189         this.num = num;
190     }
191
192     public void setName(String name) {
193         this.name = name;
194     }
195
196     public void setNickName(String nickName) {
197         this.nickName = nickName;
198     }
199
200     @Override
201     public String toString() {
202         return "HeroNode{" +
203                 "num=" + getNum() +
```

```
                ", name='" + name + '\'' +
                ", nickName='" + nickName + '\'' +
                '}';
    }
}
```

3.3 单向环形链表

3.3.1 应用场景-约瑟夫问题

设编号为 1-n 的 n 个人围坐一圈，约定编号为 k 的人开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，知道所有人出列为止，由此产生一个出队编号的序列。

用一个不带头节点的循环列表来处理该问题。

3.3.2 代码实现

构建一个单向环形链表的思路：

1. 先创建第一个节点，让 `first` 指向该节点
2. 每创建一个新的节点 `newNode`，就把该节点加入到已有的环形链表中。

遍历环形链表：

1. 先让辅助指针 `curNode`，指向 `first` 节点
2. 通过 `while` 循环遍历环形链表即可
3. 链表剩余一个节点的条件：`curNode.next == first`

```
public class Josephu {
    public static void main(String[] args) {
        CircleSingleLinkedList circleSingleLinkedList = new CircleSingleLinkedList
            ();
        circleSingleLinkedList.add(125);
        circleSingleLinkedList.list();
        System.out.println(circleSingleLinkedList.size());
        circleSingleLinkedList.josephu(10,20,circleSingleLinkedList.size());
    }
}

//创建环形单向链表
class CircleSingleLinkedList{
    //创建first节点
    private BoyNode first;

    //添加新的节点，构建一个环形链表
    public void add(int nums) {
        if (nums < 1){
            return;
        }
        //使用for循环创建环形链表
        BoyNode curNode = null;
```

```
23          for (int i = 0; i < nums; i++) {
24              BoyNode newNode = new BoyNode(i + 1);
25              if (i == 0){
26                  first = newNode;
27                  first.next =  first;
28                  curNode = first;
29              }else {
30                  newNode.next = curNode.next;
31                  curNode.next = newNode;
32                  curNode = curNode.next;
33              }
34          }
35      }
36
37      //遍历当前的环形链表
38      public void list(){
39          if (first == null){
40              System.out.println("[]");
41              return;
42          }
43          BoyNode curNode = first;
44          System.out.print("[");
45          while(curNode.next != first){
46              System.out.print(curNode.getNum());
47              System.out.print(",");
48              curNode = curNode.next;
49          }
50          System.out.print(curNode.getNum());
51          System.out.println("]");
52      }
53
54      //节点个数
55      public int size(){
56          if (first == null){
57              return 0;
58          }
59          int sum = 0;
60          BoyNode curNode = first;
61          while(true){
62              sum++;
63              if (curNode.next == first){
64                  break;
65              }
66              curNode = curNode.next;
67          }
68          return sum;
69      }
70
71      //约瑟夫问题
72      /**
73       *
```

```
74          * @param start 从哪个小孩开始数
75          * @param m 每次数几个
76          * @param sum 表示最初共有几个小孩
77          */
78         public void josephu(int start, int m, int sum) {
79             if (first == null || start < 1 || start > sum) {
80                 System.out.println("参数输入有误, 请重新输入: ");
81                 return;
82             }
83             BoyNode preNode = first;
84             BoyNode curNode = first;
85             while(preNode.next != first){
86                 preNode = preNode.next;
87             }
88             //移动到第start个小孩
89             for (int i = 0; i < start - 1; i++) {
90                 first = first.next;
91                 preNode = preNode.next;
92             }
93             System.out.print("[");
94             while(first.next != first){
95                 for (int i = 0; i < m - 1; i++) {
96                     first = first.next;
97                     preNode = preNode.next;
98                 }
99                 System.out.print(first.getNum());
100                System.out.print(",");
101                preNode.next = first.next;
102                first = first.next;
103            }
104            System.out.print(first.getNum());
105            System.out.println("]");
106        }
107
108        public BoyNode getFirst() {
109            return first;
110        }
111    }
112    //创建一个节点
113    class BoyNode{
114        private int num;
115        public BoyNode next;
116
117        public BoyNode() {
118        }
119        public BoyNode(int num) {
120            this.num = num;
121        }
122
123        public int getNum() {
124            return num;
```

```
125     }
126
127     public void setNum(int num) {
128         this.num = num;
129     }
130 }
```

第四章　栈

4.1　栈的知识

1. 栈的英文是 stack；
2. 栈的元素先进后出
3. 出栈：pop
4. 入栈：push

4.2　栈的应用场景

1. 子程序的调用；
2. 处理递归调用
3. 表达式的转换（中缀表达式转后缀表达式）与求值（使用后缀表达式求值）
4. 二叉树的遍历
5. 图形的深度优先搜索算法

4.3　代码实现

4.3.1　思路分析

1. 使用数组模拟栈；
2. `top` 表示栈顶，初始化为 `-1` 。
3. 入栈： `arr[++top] = num;`
4. 出栈： `return arr[top--]`

4.3.2　数组实现

```
import java.util.Random;
import java.util.Scanner;

public class ArrayStackDemo {
    public static void main(String[] args) {
//栈的测试室代码:
        Scanner scanner = new Scanner(System.in);
        System.out.println("栈的测试: ");
        System.out.println("请输入栈的规模");
        int capacity = scanner.nextInt();
        ArrayStack arrayStack = new ArrayStack(capacity);
        String key = "";
        boolean loop = true;
        while (loop) {
            System.out.println("-----------------------------------------");
            System.out.println("s(Show):显示栈");
            System.out.println("e(Exit):退出栈");
            System.out.println("p(Pop):弹出栈顶元素");
            System.out.println("pu(Push):往栈中加入元素");
```

```
20              System.out.println("t(TopValue): 查看栈顶元素");
21              System.out.println("请输入你的选择: ");
22              key = scanner.next();
23              switch (key) {
24                  case "t":
25                      try {
26                          int num = arrayStack.topValue();
27                          System.out.println(num);
28                      } catch (Exception e) {
29                          System.out.println(e.getMessage());
30                      }
31                      break;
32                  case "pu":
33                      int num = randomNum();
34                      System.out.println("要加入栈的元素是: " + num);
35                      arrayStack.push(num);
36                      break;
37                  case "p":
38                      try {
39                          int num1 = arrayStack.pop();
40                          System.out.println(num1);
41                      } catch (Exception e) {
42                          System.out.println(e.getMessage());
43                      }
44                      break;
45                  case "s":
46                      arrayStack.list();
47                      break;
48                  case "e":
49                      scanner.close();
50                      loop = false;
51                      System.out.println("程序退出");
52                      break;
53              }
54          }
55      }
56
57      public static int randomNum() {
58          Random random = new Random();
59          return random.nextInt(1000) + 1;
60      }
61  }
62
63  class ArrayStack {
64      private final int capacity;
65      private int size;
66      private int[] stack;
67      private int top = -1;
68      private int bottom = -1;
69
70      public ArrayStack(int capacity) {
```

```
71          this.capacity = capacity;
72          stack = new int[capacity];
73      }
74
75      //判断栈空
76      public boolean isEmpty() {
77          return top == -1;
78      }
79
80      //判断栈满
81      public boolean isFully() {
82          return top == capacity - 1;
83      }
84
85      //栈的元素
86      public int size() {
87          return top + 1;
88      }
89
90      //入栈
91      public void push(int data) {
92          if (isFully()) {
93              System.out.println("栈满，无法添加！");
94              return;
95          }
96          stack[++top] = data;
97      }
98
99      //出栈
100     public int pop() {
101         if (isEmpty()) {
102             throw new RuntimeException("栈空，没有元素可以出栈！");
103         }
104         return stack[top--];
105     }
106
107     //遍历栈：从栈顶向栈底显示元素
108     public void list() {
109         if (isEmpty()) {
110             System.out.println("[]");
111             return;
112         }
113         System.out.print("[");
114         for (int i = top; i > bottom; i--) {
115             System.out.print(stack[i]);
116             if (i != 0) {
117                 System.out.print(",");
118             }
119         }
120         System.out.println("]");
121     }
```

```
122
123     //显示栈顶元素
124     public int topValue() {
125         if (isEmpty()) {
126             throw new RuntimeException("栈空，没有元素可以出栈！");
127         }
128         return stack[top];
129     }
130 }
```

4.3.3 单链表实现

```
1  import java.util.Random;
2  import java.util.Scanner;
3
4  public class SingleLinkedListStackDemo {
5
6      public static int randomNum() {
7          Random random = new Random();
8          return random.nextInt(1000) + 1;
9      }
10
11     public static void main(String[] args) {
12         //栈的测试代码:
13         Scanner scanner = new Scanner(System.in);
14         System.out.println("栈的测试: ");
15         System.out.println("请输入栈的规模");
16         int capacity = scanner.nextInt();
17         SingleLinkedListStack stack = new SingleLinkedListStack(capacity);
18         String key = "";
19         boolean loop = true;
20         while (loop) {
21             System.out.println("-----------------------------------------");
22             System.out.println("s(Show):显示栈");
23             System.out.println("e(Exit):退出栈");
24             System.out.println("p(Pop):弹出栈顶元素");
25             System.out.println("pu(Push):往栈中加入元素");
26             System.out.println("t(TopValue): 查看栈顶元素");
27             System.out.println("请输入你的选择: ");
28             key = scanner.next();
29             switch (key) {
30                 case "t":
31                     try {
32                         int num = stack.topValue();
33                         System.out.println(num);
34                     } catch (Exception e) {
35                         System.out.println(e.getMessage());
36                     }
37                     break;
38                 case "pu":
```

```
39                      int num = randomNum();
40                      System.out.println("要加入栈的元素是: " + num);
41                      stack.push(num);
42                      break;
43                  case "p":
44                      try {
45                          int num1 = stack.pop();
46                          System.out.println(num1);
47                      } catch (Exception e) {
48                          System.out.println(e.getMessage());
49                      }
50                      break;
51                  case "s":
52                      stack.list();
53                      break;
54                  case "e":
55                      scanner.close();
56                      loop = false;
57                      System.out.println("程序退出");
58                      break;
59              }
60          }
61      }
62
63      static class SingleLinkedListStack {
64          private Node top;
65          private final int capacity;
66
67          public SingleLinkedListStack(int capacity) {
68              this.capacity = capacity;
69          }
70
71          public boolean isEmpty() {
72              return top == null;
73          }
74
75          public boolean isFull() {
76              return size() == capacity;
77          }
78
79          public int size() {
80              int sum = 0;
81              Node curNode = top;
82              while (curNode != null) {
83                  sum++;
84                  curNode = curNode.next;
85              }
86              return sum;
87          }
88
89          public void push(int num) {
```

```
90              if (isFull()) {
91                  System.out.println("栈满，无法加入！");
92                  return;
93              }
94              Node node = new Node(num);
95              node.next = top;
96              top = node;
97          }
98
99          public int pop() {
100             if (isEmpty()) {
101                 throw new RuntimeException("栈空，无法取出元素！");
102             }
103             int value = top.getNum();
104             top = top.next;
105             return value;
106         }
107
108         public int topValue() {
109             return top.getNum();
110         }
111
112         public void list() {
113             Node curNode = top;
114             System.out.print("[");
115             while (curNode != null) {
116                 System.out.print(curNode.getNum());
117                 if (curNode.next != null) {
118                     System.out.print(",");
119                 }
120                 curNode = curNode.next;
121             }
122             System.out.println("]");
123         }
124     }
125
126     static class Node {
127         private int num;
128         public Node next;
129
130         public Node(int num) {
131             this.num = num;
132         }
133
134         public int getNum() {
135             return num;
136         }
137     }
138 }
```

4.4 栈实现综合计算器

数栈：用于存放数字

符号栈：用于存放符号

使用一个指针扫描字符串：

如果是数字就入数栈；

如果是符号：如果符号栈为空，该符号就直接入栈；如果符号栈有操作符，就要进行比较，如果该符号的优先级小于或者等于栈顶的操作符，需要从数栈弹出两个数字，从符号栈弹出一个操作符，进行运算，得到的结果入数栈，让该符号入操作符栈。如果该符号优先级大于符号栈栈顶的操作符，直接入栈。

扫描完毕后，顺序的从符号栈和数栈弹出相应的符号和数进行运算。

最后数栈只有一个数字，就是最终的运算结果。

4.4.1 代码实现

```java
public class Calculator {
    public static double cal(double x, double y, char ope) {
        double ret = 0;
        switch (ope) {
            case '/':
                if (y == 0) {
                    throw new RuntimeException("除数为0");
                }
                ret = x / y;
                break;
            case '*':
                ret = x * y;
                break;
            case '-':
                ret = x - y;
                break;
            case '+':
                ret = x + y;
                break;
            case '^':
                ret = Math.pow(x,y);
                break;
        }
        return ret;
    }

    public static int[] readNumber(String str, int index) {
        int ret = 0;
        while (index < str.length() && isNum(str.charAt(index))) {
            int x = str.charAt(index) - '0';
            ret = ret * 10 + x;
            index++;
        }
        return new int[]{ret, index};
    }
```

```
36
37      //判断是否为运算符
38      public static boolean isNum(char val) {
39          return val >= '0' && val <= '9';
40      }
41
42      public static int opeNum(char ch) {
43          int ret = -1;
44          switch (ch) {
45              case '+':
46                  ret = 0;
47                  break;
48              case '-':
49                  ret = 1;
50                  break;
51              case '*':
52                  ret = 2;
53                  break;
54              case '/':
55                  ret = 3;
56                  break;
57              case '^':
58                  ret = 4;
59                  break;
60              case '!':
61                  ret = 5;
62                  break;
63              case '(':
64                  ret = 6;
65                  break;
66              case ')':
67                  ret = 7;
68                  break;
69              case'\0':
70                  ret = 8;
71                  break;
72          }
73          return ret;
74      }
75
76      //
77      public static char orderBetween(char ch1, char ch2) {
78          int num1 = opeNum(ch1);
79          int num2 = opeNum(ch2);
80          char pri[][] =  //运算符优先等级[栈顶][当前]
81                  {       //      |----------当前运算符-----------|
82                          //竖的为栈顶运算符
83                          //                +    -    *    /    ^    !    (    )
84                          /* + */        {'>', '>', '<', '<', '<', '<', '<', '>','>'
                                },
85                          /* - */        {'>', '>', '<', '<', '<', '<', '<', '>','>'
```

```
                            },
86                      /* * */        {'>', '>', '>', '>', '<', '<', '<', '>','>'
                            },
87                      /* / */        {'>', '>', '>', '>', '<', '<', '<', '>','>'
                            },
88                      /* ^ */        {'>', '>', '>', '>', '>', '<', '<', '>','>'
                            },
89                      /* ! */        {'>', '>', '>', '>', '>', '>', ' ', '>','>'
                            },
90                      /* ( */        {'<', '<', '<', '<', '<', '<', '<', '=',' '
                            },
91                      /* ) */        {' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',' '
                            },
92                      /* \0 */       {'<', '<', '<', '<', '<', '<', '<', ' ','='
                            }
93              };
94          return pri[num1][num2];
95      }
96
97      public static void main(String[] args) {
98          String expression = "13+2*(6-2)+5/2";
99          ArrayStack<Double> numStack = new ArrayStack<>(100);
100         ArrayStack<Character> opeStack = new ArrayStack<>(100);
101         int index = 0;
102         char ope = '0';
103         char ch = ' ';//将每次扫描得到的char保存到ch中
104         opeStack.push('\0');
105         while (!(opeStack.isEmpty())) {
106             if(index < expression.length()){
107                 ch = expression.charAt(index);
108             }else{
109                 ch = '\0';
110             }
111             if (isNum(ch)) {
112                 int[] arr = readNumber(expression, index);
113                 double x = (double)arr[0];
114                 numStack.push(x);
115                 index = arr[1];
116             } else {
117                 if (opeStack.isEmpty()) {
118                     opeStack.push(ch);
119                     index++;
120                 } else {
121                     switch (orderBetween(opeStack.topValue(), ch)) {
122                         case '>':
123                             ope = opeStack.pop();
124                             if (ope == '!') {
125                                 double x = numStack.pop();
126                                 numStack.push((double)fac((int)x));
127                             }else{
128                                 double y = numStack.pop();
```

```
                                double x = numStack.pop();
                                double value = cal(x,y,ope);
                                numStack.push(value);
                            }
                            break;
                        case '<':
                            opeStack.push(ch);
                            index++;
                            break;
                        case '=':
                            opeStack.pop();
                            index++;
                            break;
                    }
                }
            }
        }
        System.out.println(numStack.pop());
    }

    private static int fac(int num) {
        if (num == 0 || num == 1){
            return 1;
        }
        return num * fac(num - 1);
    }

    public static void print(ArrayStack arrayStack) {
        arrayStack.list();
    }
}
```

4.5 栈应用–逆波兰表达式

前缀表达式又称波兰式，表达式的运算符位于操作数之前。

逆波兰表达式求值：从左至右扫描，遇数入栈，遇到符号则弹出栈顶元素进行计算。

4.5.1 逆波兰表达式代码的实现

```
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

public class PolandNotation {
    public static void main(String[] args) {
        String suffixExpression = "30 4 + 5 * 6 -";
        List<String> list = getList(suffixExpression);
        System.out.println(suffixExpression + " = " + cal(list));
    }
```

```
11      //将逆波兰表达式放入顺序表中
12      public static List<String> getList(String str){
13          List<String> list = new ArrayList<>();
14          int index = 0;
15          String str1[] = str.split(" ");
16          for (String s:str1) {
17              list.add(s);
18          }
19          return list;
20      }
21      //完成对逆波兰表达式的运算
22      public static int cal(List<String> list){
23          Stack<String> stack = new Stack<>();
24          for (String ch: list) {
25              if (ch.matches("\\d+")){//匹配多位数
26                  stack.push(ch);
27              }else{
28                  if ("!".equals(ch)){
29                      int num = Integer.parseInt(stack.pop());
30                      stack.push("" + fac(num));
31                  }else{
32                      int num2 = Integer.parseInt(stack.pop());
33                      int num1 = Integer.parseInt(stack.pop());
34                      int ret = 0;
35                      switch (ch){
36                          case "+":
37                              ret = num1 + num2;
38                              break;
39                          case "-":
40                              ret = num1 - num2;
41                              break;
42                          case "*":
43                              ret = num1 * num2;
44                              break;
45                          case "/":
46                              ret = num1 / num2;
47                              break;
48                          case "^":
49                              ret = (int)Math.pow((double)num1,(double)num1);
50                              break;
51                          default:
52                              throw new RuntimeException("运算符有误");
53                      }
54                      stack.push("" + ret);
55                  }
56              }
57          }
58          return Integer.parseInt(stack.pop());
59      }
60
61      private static int fac(int num) {
```

```
62          if (num == 0 || num == 1){
63              return 0;
64          }
65          return num * fac(num -1);
66      }
67  }
```

4.5.2 中缀表达式转后缀表达式

4.5.2.1 思路

1. 初始化两个栈 `stack1`、`stack2`。
2. 从左至右扫描中缀表达式。
3. 遇到操作数，压入 `stack2`
4. 遇到运算符，比较该符号与 `stack1` 运算符的优先级：

- 若 `stack1` 为空，或栈顶运算符为 `(`，将该运算符压入 `stack1`。
- 若优先级比栈顶的运算符高，也将运算符压入 `stack1`。
- 否则，将 `stack1` 栈顶元素弹出压入到 `stack2` 中，再次跳转到 4-1 步与 `stack1` 栈顶运算符相比较。

5. 遇到括号：

- `(`：直接入 `stack1`
- `)`：依次弹出 `stack1` 运算符将其压入 `stack2`，遇到 `(`，将这一对括号丢弃。

6. 重复 2-5 步，直到中缀表达式扫描结束。
7. 将 `stack1` 剩余符号全部弹出压入 `stack2`。
8. `stack2` 从栈顶至栈底则是逆波兰表达式。

```
1   import java.util.ArrayList;
2   import java.util.List;
3   import java.util.Stack;
4
5   //中缀表达式转后缀表达式
6   public class ToRPN {
7       public static void main(String[] args) {
8           String expression = "1+((2+3)*4)-5";
9           List<String> list = toList(expression);
10          System.out.println(list);
11          System.out.println(parsrSuffixExpreesionToRPN(list));
12      }
13
14      //中缀表达式转成对应的list
15      public static List<String> toList(String str) {
16          List<String> list = new ArrayList<>();
17          int index = 0;
18          StringBuilder stringBuilder = new StringBuilder();
19          while (index < str.length()) {
20              if (str.charAt(index) == ' ') {
```

```
21                  index++;
22              } else if (str.charAt(index) >= '0' && str.charAt(index) <= '9') {
23                  while (index < str.length() && str.charAt(index) >= '0' && str.
                        charAt(index) <= '9') {
24                      stringBuilder.append(str.charAt(index));
25                      index++;
26                  }
27                  list.add(stringBuilder.toString());
28                  stringBuilder.delete(0, stringBuilder.length());
29              } else {
30                  list.add("" + str.charAt(index));
31                  index++;
32              }
33          }
34          return list;
35      }
36
37      //中缀表达式转为逆波兰表达式
38      public static List<String> parsrSuffixExpreesionToRPN(List<String> list) {
39          List<String> rpn = new ArrayList<>();//存储中间结果和
40          Stack<String> stack1 = new Stack<>();//符号栈
41          for (String str : list) {
42              if (str.matches("\\d+")) {
43                  rpn.add(str);
44              } else if ("(".equals(str)) {
45                  stack1.push(str);
46              } else if (")".equals(str)) {
47                  while (!stack1.peek().equals("(")) {
48                      rpn.add(stack1.pop());
49                  }
50                  stack1.pop();
51              } else {
52                  //符号栈不为空或者符号栈栈顶不是(或者该符号优先级不大于栈顶符号
53                  while (!(stack1.size() == 0 || stack1.peek().equals("(")) && !(
                        getValue(str) > getValue(stack1.peek()))) {
54                      rpn.add(stack1.pop());
55                  }
56                  //否则符号入栈
57                  stack1.push(str);
58              }
59          }
60          while (!stack1.isEmpty()) {
61              rpn.add(stack1.pop());
62          }
63          return rpn;
64      }
65
66      //优先级
67      public static int getValue(String str) {
68          int add = 1;
69          int sub = 1;
```

```
70          int mul = 2;
71          int div = 2;
72
73          int ret = 0;
74          switch (str) {
75              case "+":
76                  ret = add;
77                  break;
78              case "-":
79                  ret = sub;
80                  break;
81              case "*":
82                  ret = mul;
83                  break;
84              case "/":
85                  ret = div;
86                  break;
87          }
88          return ret;
89      }
90  }
```

第五章　递归

可以解决的问题：

1. 八皇后问题、汉诺塔、阶乘问题、迷宫问题、球和篮子问题。
2. 快速排序、归并排序、二分查找、分治算法等等。
3. 将用栈解决的问题

递归遵守的规则：

1. 执行一个方法时，就创建一个新的受保护的独立栈空间
2. 方法的局部变量是独立的，不会相互影响，如果是引用类型的变量，就会共享该引用类型的数据。
3. 递归必须向递归基逼近
4. 当一个方法执行完毕，或者遇到`return`，就会返回，谁调用就返回给谁。

5.1　迷宫问题

```
public class Maze {
    public static void main(String[] args) {
        //先创建一个二维数组模拟迷宫
        final int row = 8;
        final int col = 8;
        //迷宫地图
        int[][] map = new int[row][col];
        //使用1来表示迷宫的墙壁
        for (int i = 0; i < col; i++) {
            map[0][i] = 1;
            map[row - 1][i] = 1;
        }
        for (int i = 1; i < row - 1; i++) {
            map[i][0] = 1;
            map[i][col - 1] = 1;
        }
        //地图中的挡板
        map[3][1] = 1;
        map[3][2] = 1;
        map[1][2] = 1;
//          map[2][2] = 1;
        //输出地图
        printMaze(map);

        //使用递归回溯来给小球找路
        System.out.println("----------------------------");
        setWay(map,1,1);
        printMaze(map);
    }
    //打印迷宫地图
    public static void printMaze(int[][] map){
        for (int[] arr: map) {
            for (int i = 0; i < arr.length; i++) {
                System.out.print(arr[i]);
```

```
                if (i != arr.length-1){
                    System.out.print(" ");
                }
            }
            System.out.println();
        }
    }
    //小球找路
    //map表示地图
    //出口为地图的右下角，即map[clo - 1 - 1][row - 1 - 1]
    //当map[i][j]为0，表示该点没有走过
    //当map[i][j]为1，表示该点为墙
    //当map[i][j]为2，表示该点走过，且是通路
    //当map[i][j]为3，表示该点走过，但是不是通路
    //走迷宫时，需要确定一个策略：先下再右再上再左。如果该点走不通，再回溯
    /**
     *
     * @param map 表示地图
     * @param i 表示开始位置的横坐标
     * @param j 表示开始位置的横坐标
     * @return 找路结果，找到为true
     */
    public static boolean setWay(int[][] map,int i,int j){
        int x = map.length;
        int y = map[0].length;
        if(map[x - 1 - 1][y - 1 - 1] == 2){//通路已找到
            return true;
        }else{
            if (map[i][j] == 0){//当前的点还没走过
                //按照策略玩
                map[i][j] = 2;//假定该点可以走通
                if(setWay(map,i+1,j)){//向下
                    return true;
                }else if(setWay(map,i,j+1)){//向右
                    return true;
                }else if(setWay(map,i-1,j)){//向上
                    return true;
                }else if(setWay(map,i,j-1)){//向左
                    return true;
                }else{//向四个方向都没走通，说明该点是死路
                    map[i][j] = 3;
                    return false;
                }
            }else{//某点不等于0，可能为1（墙）、2（通路）、3（死路）
                    return false;
            }
        }
    }
}
```

5.2 八皇后问题

问题描述：在 8*8 的棋盘上摆放八颗棋子，任意两颗棋子不能处于同一行、同一列或同一斜线上，问共有多少种摆法。

5.2.1 思路分析

1. 第一个皇后先放到第一行第一列。

2. 第二个皇后放在第二行第一列，判断是否 OK，如果不 OK，继续放在第二列、第三列……依次把所有的列放完后，找到一个合适的位置

3. 下一个皇后同上边第二步

4. 当得到一个正确解时，在退回到上一个栈，就会开始回溯，即将第一个皇后放到第一列的所有正确解全部得到

5. 继续第一个皇后放到第二列，继续循环执行 1-4 步。

5.2.2 代码实现

```
import java.util.Arrays;

public class QueenEight {
    //皇后的个数
    final static int capacity = 8;
    //定义一个一维数组存放皇后的位置
    int[] pos = new int[capacity];
    //解法
    static int count;
    public static void main(String[] args) {
        new QueenEight().check(capacity);
    }
    //将皇后摆放的位置输出
    private  void print(){
        System.out.println(Arrays.toString(pos));
    }

    //放置皇后
    private void check(int n){
        int x = capacity - n;
        if (x == capacity){//共8个皇后，x从0开始的，n等于8说明全部放完了
            System.out.print("第" + (++count) + "种解法：");
            print();
            return;
        }
        //依次放入皇后，并判断是否冲突
        for (int i = 0; i < capacity; i++) {
            //先把当前这个皇后x，放到该行的第1列
            pos[x] = i;
            //放置后判断是否冲突
            if(judge(x)){//不冲突就放下一个皇后
                check(n - 1);
```

```
33              }
34              //冲突就把该皇后放到下一个位置
35          }
36      }
37
38      //当放第n个皇后，检测是否和前面已经摆放的皇后冲突
39      /**
40       *
41       * @param n 第n个皇后,n从0开始
42       * @return 冲突: false, 不冲突: true
43       */
44      private boolean judge(int n){
45          for (int i = 0; i < n; i++) {
46              //pos[i] == pos[n]表示第n个皇后和第i个皇后位于同一列
47              //Math.abs(n - i) == Math.abs(pos[n] - pos[i])表示第n个皇后和第i个皇后
                    位于同一斜线
48              if (pos[i] == pos[n] || Math.abs(n - i) == Math.abs(pos[n] - pos[i])){
49                  return false;
50              }
51          }
52          return true;
53      }
54  }
```

第六章　排序算法

6.1　冒泡排序

1. 一共要进行数据规模- 1 次排序

6.1.1　代码实现

```
package com.atWSN.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
import java.util.Random;
import java.util.Scanner;

public class BubbleSort {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入问题的规模: ");
        int num = scanner.nextInt();
        int[] arr = new int[num];
        //初始化数组
        initArr(arr);
        //排序前数组
//        System.out.println("=================");
//        System.out.print("排序前: ");
//        print(arr);
        //排序
        Date date1 = new Date();
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm
            :ss");
        String date1Str = simpleDateFormat.format(date1);
        System.out.println("排序前的时间: " + date1Str);
        bubbleSort(arr);
        Date date2 = new Date();
        String date2Str = simpleDateFormat.format(date2);
        System.out.println("排序后的时间: " + date2Str);
        //排序后数组
//        System.out.println("=================");
//        System.out.print("排序后: ");
//        print(arr);
    }

    private static void bubbleSort(int[] arr) {
        boolean flag = true;
        for (int i = 0; i < arr.length - 1; i++) {
            flag = true;
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if(arr[j] > arr[j + 1]){
```

```
                    flag = false;
                    int tmp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = tmp;
                }
            }
            // System.out.println(Arrays.toString(arr));
            if (flag){
                break;
            }
        }
    }

    private static void print(int[] arr) {
        System.out.println(Arrays.toString(arr));
    }

    private static void initArr(int[] arr) {
        Random random = new Random();
        for (int i = 0; i < arr.length; i++) {
            arr[i] = random.nextInt(2000) - 1000;
        }
    }
}
```

6.2 选择排序

从前向后遍历数组，从未排序数组中选择最小的数据与未排序数组中的第一个元素进行交换。

或者从后向前遍历数组，从未排序的数组中选择最大的数据与未排序数组中的最后一个元素进行交换。

6.2.1 代码实现

```
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;

public class SelectSort {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入问题的规模: ");
        int num = scanner.nextInt();
        int[] arr = new int[num];
        //初始化数组
        initArr(arr);
        //排序前数组
        System.out.println("=================");
```

```
16          System.out.print("排序前: ");
17          print(arr);
18          //排序
19          selectSort(arr);
20          //排序后数组
21          System.out.println("=================");
22          System.out.print("排序后: ");
23          print(arr);
24      }
25
26      private static void selectSort(int[] arr) {
27          for (int i = 0; i < arr.length - 1; i++) {
28              int min = arr[i];
29              int index = i;
30              for (int j = i + 1; j < arr.length; j++) {
31                  if(arr[j] == min){
32                      break;
33                  }else if (arr[j] < min){
34                      min = arr[j];
35                      index = j;
36                  }
37              }
38              if (!(i == index)){
39                  int tmp = arr[i];
40                  arr[i] = arr[index];
41                  arr[index] = tmp;
42              }
43              System.out.println("第" + (i + 1) +"轮排序");
44              print(arr);
45          }
46      }
47
48      private static void print(int[] arr) {
49          System.out.println(Arrays.toString(arr));
50      }
51
52      private static void initArr(int[] arr) {
53          Random random = new Random();
54          for (int i = 0; i < arr.length; i++) {
55              arr[i] = random.nextInt(10) + 1;
56          }
57      }
58  }
```

6.3 插入排序

基本思想：把 n 个待排序的元素看成一个有序表和无序表，开始时有序表只包含一个元素，无序表中包含有 n-1 个元素，排序过程中每次从无序表中取出第一个元素，从有序中查找到插入的位置，将其插入到有序表的适当位置，使之成为新的有序表。

6.3.1 代码实现

未排序数组类

```java
import java.util.Arrays;
import java.util.Random;

public class unSortArray {
    private final int capacity;
    public int[] arr;

    public unSortArray(int capacity){
        this.capacity = capacity;
        arr = new int[this.capacity];
        Random random = new Random();
        for (int i = 0; i < capacity; i++) {
            arr[i] = random.nextInt(capacity) + 1;
        }
    }
    public void print(){
        System.out.println(Arrays.toString(arr));
    }
}
```

插入排序

```java
import java.util.Scanner;

//插入排序
public class InsertSort {
    public static void main(String[] args) {
        System.out.println("请输入数据的规模: ");
        int capacity = new Scanner(System.in).nextInt();
        unSortArray unSortArray = new unSortArray(capacity);
        System.out.println("------------------");
        System.out.println("排序前: ");
        unSortArray.print();
        //排序
        insertSort(unSortArray.arr);
        System.out.println("------------------");
        System.out.println("排序后: ");
        unSortArray.print();
    }
    //插入排序
    private static void insertSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            int insertVal = arr[i];
            int index = i;
            for (int j = 0; j < i; j++) {
                if (arr[j] > insertVal){
                    index = j;
```

```
                    break;
                }
            }
            if (!(index == i)) {
                for (int j = i; j > index; j--) {
                    arr[j] = arr[j - 1];
                }
                arr[index] = insertVal;
            }
        }
    }
    //版本2
    public static void insertSort1(int[] arr){
        if (arr.length <= 1){
            return;
        }
        for (int i = 1; i < arr.length; i++) {
            int index = i - 1;
            int insertValue = arr[i];
            while(index >= 0 && insertValue < arr[insertValue]){
                arr[index + 1] = arr[index];
                index--;
            }
            if (!(index == i - 1)) {
                arr[index] = insertValue;
            }
        }
    }
}
```

6.4 希尔排序

6.4.1 代码实现

先分组再进行选择排序。

```
import java.util.Scanner;

public class ShellSort {
    public static void main(String[] args) {
        System.out.println("请输入问题的规模: ");
        int capacity = new Scanner(System.in).nextInt();
        unSortArray unSortArray = new unSortArray(capacity);
        System.out.println("排序前: ");
        unSortArray.print();
        //排序
        shellSort2(unSortArray.arr);
        System.out.println("------------------");
        System.out.println("排序后: ");
        unSortArray.print();
```

```
15      }
16
17      private static void shellSort1(int[] arr) {
18          if (arr.length < 2){
19              return;
20          }
21          for (int step = arr.length/2; step > 0 ; step /= 2) {
22              //遍历各组中所有元素，每组元素有step个
23              for (int i = step; i < arr.length; i++) {
24                  //交换法：从每组的倒数第二个元素开始，与它的后一个元素比较，若大于
                        就交换，每次向前移动
25                  for (int j = i - step; j >= 0; j -= step) {
26                      if (arr[j] > arr[j + step]){
27                          int tmp = arr[j];
28                          arr[j] = arr[j + 1];
29                          arr[j + 1] = tmp;
30                      }
31                  }
32              }
33          }
34      }
35
36      //优化shell排序：移位法
37      public static void shellSort2(int[] arr){
38          if(arr.length < 2){
39              return;
40          }
41          for (int step = arr.length/2; step > 0; step /= 2) {
42              //step表示组数
43              //从第step个元素开始，对其所在的组进行插入排序
44              for (int i = step; i < arr.length; i++) {
45                  int j = i - step;
46                  int value = arr[i];
47                  while (j >= 0 && arr[j] > value){
48                      arr[j + step] = arr[j];
49                      j -= step;
50                  }
51                  if(!(j == i - step)){
52                      arr[j + step] = value;
53                  }
54
55              }
56          }
57      }
58  }
```

6.5 快速排序

基本思想：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，再按此方法对这两部分进行快速排序，整个排序过程可以递归进行。

关键在于轴点的构造。

6.5.1 代码实现

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Scanner;

public class QuickSort {
    public static void main(String[] args) {
        System.out.println("请输入数据的规模：");
        int capacity = new Scanner(System.in).nextInt();
        unSortArray unSortArray = new unSortArray(capacity);
//        System.out.println("------------------");
//        System.out.println("排序前：");
//        unSortArray.print();
        //排序\
        Date date1 = new Date();
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm
            :ss");
        String date1Str = simpleDateFormat.format(date1);
        System.out.println("排序前的时间：" + date1Str);
        quickSort(unSortArray.arr,0,unSortArray.arr.length - 1);
        Date date2 = new Date();
        String date2Str = simpleDateFormat.format(date2);
        System.out.println("排序后的时间：" + date2Str);
//        System.out.println("------------------");
//        System.out.println("排序后：");
//        unSortArray.print();
    }

    public static void quickSort (int[] arr, int lo,int hi){
        if(hi - lo < 1){
            return;
        }
        int mid = partition(arr,lo, hi);//mid位置处的元素已就位，即轴点。
        quickSort(arr,lo,mid - 1);//前缀排序
        quickSort(arr,mid + 1,hi);//后缀排序
    }
    //构造轴点
    private static int partition(int[] arr,int lo, int hi) {
        int left = lo;//左指针
        int right = hi;//右指针
        int value = arr[lo];//轴点的值，一般取区间第一个
        while (left < right) {
            //如果右边的值不小于轴点，右指针左移
```

```
            while((left < right) && value <= arr[right]){
                right--;
            }
            //右边的值小于轴点的值，取出该值放入左指针所指的值，同时左指针右移一个单位
            if(left < right){
                arr[left] = arr[right];
                left++;
            }
            //如果左指针指的值不大于轴点值，左指针右移
            while((left < right) && value >= arr[left]){
                left++;
            }
            //左指针的值大于轴点的值，取出该值放入右指针所指的位置，同时右指针左移
            if(left < right){
                arr[right] = arr[left];
                right--;
            }
        }
        //循环退出的条件是左指针右指针重合，
        // 其含义表示该位置左边的值都小于轴点，
        // 该位置右边的值都大于轴点
        //把轴点的值放入该位置，同时返回该位置。
        arr[left] = value;
        return left;
    }
}
```

6.6 归并排序

利用归并的思想实现排序的方法，采用经典的分治算法。

6.6.1 代码实现

```
import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
import java.util.Scanner;

public class MergeSort {
    public static void main(String[] args) {
        System.out.println("请输入数据的规模：");
        int capacity = new Scanner(System.in).nextInt();
        unSortArray unSortArray = new unSortArray(capacity);
//        System.out.println("------------------");
//        System.out.println("排序前：");
//        unSortArray.print();
        //排序
        Date date1 = new Date();
```

```
16          SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm
                :ss");
17          String date1Str = simpleDateFormat.format(date1);
18          System.out.println("排序前的时间: " + date1Str);
19          mergeSort(unSortArray.arr,0,unSortArray.arr.length - 1);
20          Date date2 = new Date();
21          String date2Str = simpleDateFormat.format(date2);
22          System.out.println("排序后的时间: " + date2Str);
23  //          System.out.println("------------------");
24  //          System.out.println("排序后: ");
25  //          unSortArray.print();
26  //          int[] arr = {1, 8, 4, 7, 3, 6, 5, 2};
27  //          mergeSort(arr, 0,  arr.length - 1);
28      }
29
30      private static void mergeSort(int[] arr, int left, int right) {
31
32          if (right - left == 0) {
33              return;
34          }
35          int mid = left + ((right - left) >> 1);
36          mergeSort(arr, left, mid);
37          mergeSort(arr, mid + 1, right);
38          merge(arr, left, mid, right);
39      }
40
41      /**
42       * @param arr   待排序的数组
43       * @param left  左边有序的左边界
44       * @param mid   左边有序的右边界, 右边有序的左边界的前一个位置
45       * @param right 右边有序的右边界
46       */
47      private static void merge(int[] arr, int left, int mid, int right) {
48          int[] tmp = Arrays.copyOfRange(arr, left, mid + 1);
49          int lo1 = 0;
50          int lo2 = mid + 1;
51          int i = left;
52          while (lo1 < tmp.length && lo2 <= right) {
53              if (tmp[lo1] <= arr[lo2]) {
54                  arr[i++] = tmp[lo1++];
55              } else {
56                  arr[i++] = arr[lo2++];
57              }
58          }
59          while (i <= right) {
60              if (lo1 < tmp.length) {
61                  arr[i++] = tmp[lo1++];
62              }
63              if (lo2 <= right) {
64                  break;
65              }
```

```
        }
    }
}
```

6.7 基数排序

基数排序属于分配式排序，又称桶子法或 bin sort。属于稳定排序。

思想：

1. 将所有待比较数值统一为同样的长度，数位较短的数前补 0，从最低位开始，依次进行依次排序。从最低位排序一直到最高位排序完成以后，数列变成一个有序序列。

6.7.1 代码实现

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Scanner;

public class RadixSort {
    public static void main(String[] args) {
        System.out.println("请输入数据的规模：");
        int capacity = new Scanner(System.in).nextInt();
        unSortArray unSortArray = new unSortArray(capacity);
//        System.out.println("------------------");
//        System.out.println("排序前：");
//        unSortArray.print();
        //排序
        Date date1 = new Date();
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm
            :ss");
        String date1Str = simpleDateFormat.format(date1);
        System.out.println("排序前的时间：" + date1Str);
        radixSort(unSortArray.arr);
        Date date2 = new Date();
        String date2Str = simpleDateFormat.format(date2);
        System.out.println("排序后的时间：" + date2Str);
//        System.out.println("------------------");
//        System.out.println("排序后：");
//        unSortArray.print();
    }
    public static void radixSort(int[] arr){
        //得到数组中最大的数
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max){
                max = arr[i];
            }
        }
        //得到最大位数
```

```
35          int maxLength = ("" + max).length();
36          //定义桶
37          int[][] bucket = new int[10][arr.length];
38          //记录每个桶中元素的个数
39          int[] bucketElementCount = new int[10];
40          for (int i = 0,n = 1; i < maxLength; i++ , n*=10) {
41              for (int j = 0; j < arr.length; j++) {
42                  int digitOfElement = arr[j] / n % 10;//找到所在的桶
43                  //第一个为所在桶的位置
44                  //第二个为所在桶的指针位置
45                  bucket[digitOfElement][bucketElementCount[digitOfElement]++] = arr
                        [j];
46              }
47
48              //把桶中的数据取出放入原数组
49              int index = 0;
50              for (int j = 0; j < bucketElementCount.length; j++) {
51                  if (!(bucketElementCount[j] == 0)){
52                      for (int k = 0; k < bucketElementCount[j]; k++) {
53                          arr[index++] = bucket[j][k];
54                      }
55                      //从桶中取出数据后，把该桶元素的个数归0
56                      bucketElementCount[j] = 0;
57                  }
58              }
59          }
60      }
61  }
```

6.8 堆排序

6.8.1 堆

堆是具有以下性质的完全二叉树：
每个节点的值都大于等于孩子节点的值，称为大顶堆。
每个节点的值都小于等于左右孩子节点的值称为小顶堆。

6.8.2 堆排序的基本思想

1. 将待排序的序列构造成一个大顶堆；
2. 此时，整个序列的最大值就是堆顶的根节点。
3. 将该节点与末尾元素进行交换，此时末尾就为最大值。
4. 剩余节点按照上述步骤进行。

6.8.3 批量构造二叉堆

```
1  //构建堆
2
```

```
/**
 * @param arr    要构建的数组
 * @param i      要调整的最后一个非叶子节点
 * @param length 要调整的数组长度
 */

public static void adjustHeap(int[] arr, int i, int length) {
//先取出当前元素的值，保存在临时变量中
    int tmp = arr[i];
    //说明
    //1.k = i * 2 + 1是i节点的左子节点，
    for (int j = i * 2 + 1; j < length; j = j * 2 + 1) {
        if (j + 1 < length && arr[j] < arr[j + 1]) {//判断右子节点是否存在
            //右子节点的值大于左子节点，把指针指向右子节点
            j++;
        }
        if (arr[j] > tmp) {//孩子大于该值，就下滤
            arr[i] = arr[j];
            i = j;//让i指向j继续比较
        } else {
            break;
        }
    }
    //for结束后，已经将以i为父节点的树的最大值放在了最顶上
    arr[i] = tmp;
}
```

6.8.4 代码实现

```
public class HeapSort {
    public static void main(String[] args) {
        int[] arr = new int[20];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = new Random().nextInt(100) + 1;
        }
        System.out.println(Arrays.toString(arr));
        heapSort(arr);
        System.out.println(Arrays.toString(arr));
    }

    public static void heapSort(int[] arr) {
        //先构建堆
        //从第一个叶子节点开始往前，直到根节点
        for (int j = (arr.length - 1 - 1) / 2; j >= 0; j--) {
            adjustHeap(arr, j, arr.length);
        }

        for (int i = arr.length - 1; i > 0; i--) {
            //上边构建堆后堆顶元素必然是最大，将其与尾元素进行交换
            int tmp = arr[0];
```

```
22              arr[0] = arr[i];
23              arr[i] = tmp;
24              //交换后不满足堆序性，重新建堆
25              adjustHeap(arr, 0, i);
26          }
27      }
28  }
```

第七章　查找相关算法

7.1　顺序查找

7.1.1　代码实现

```java
import java.util.Arrays;
import java.util.Scanner;

public class SeqSearch {
    public static void main(String[] args) {
        int[] arr = new Array(10).arr;
        System.out.println(Arrays.toString(arr));
        System.out.println("请输入要查找的数: ");
        int num = new Scanner(System.in).nextInt();
        int ret = seqSearch(arr,num);
        if (ret == -1){
            System.out.println("未找到! ");
        }else {
            System.out.println("该元素下标为" + ret);
        }

    }
    //找到第一个满足条件的下标
    public static  int seqSearch(int[] arr,int value){
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == value){
                return i;
            }
        }
        return -1;
    }
}
```

7.2　二分查找

7.2.1　要求

有序数组。

7.2.2　代码实现

```java
import java.util.Arrays;

public class BinarySearch {
    public static void main(String[] args) {
        int[] arr = {2,8,10,89,89,89,89,1000,1234};
        int target =  1;
```

```
7           System.out.println(binarySearch(arr,0,arr.length - 1,target));
8           System.out.println(binarySearch1(arr,target));
9           System.out.println(Arrays.toString(binarySearch2(arr,target)));
10          System.out.println(insert(arr,target));
11      }
12      //递归版本
13      public static int binarySearch(int[] arr,int lo,int hi,int value){
14          if (lo > hi){
15              return -1;
16          }
17          int mid = lo + ((hi - lo) >> 1);
18          int midValue = arr[mid];
19          if(midValue < value){
20             return binarySearch(arr,mid + 1,hi,value);
21          }else if(value <midValue){
22              return binarySearch(arr,lo,mid - 1,value);
23          }else{
24              return mid;
25          }
26      }
27      //迭代版本
28      public static int binarySearch1(int[] arr,int value){
29          int lo = 0;
30          int hi = arr.length;
31          while (lo <= hi){
32              int mid = lo + ((hi - lo) >> 1);
33              if(value < arr[mid]){
34                  hi = mid - 1;
35              }else if(value > arr[mid]){
36                  lo = mid + 1;
37              }else{
38                  return mid;
39              }
40          }
41          return -1;
42      }
43
44      //寻找插入值
45      //如果元素存在，返回该元素最后一个所在的位置
46      //如果元素不存在，返回其插入位置的前一个位置
47      public static  int insert(int[] arr,int target){
48          int lo = 0;
49          int hi = arr.length;
50          while (lo <= hi){
51              int mid = lo + ((hi - lo) >> 1);
52              if(target < arr[mid]){
53                  hi = mid - 1;
54              }else if(target > arr[mid]){
55                  lo = mid + 1;
56              }else{
57                  return findLast(arr,target);
```

```
            }
        }
        return lo - 1;
    }

    //找到数组中所有的目标值元素，返回其范围
    //没有则返回[-1,-1]
    public static int[] binarySearch2(int[] arr,int target){
        return new int[]{findFirst(arr,target),findLast(arr,target)};
    }

    private static int findLast(int[] arr, int target) {
        int lo = 0;
        int hi = arr.length;
        while (lo <= hi){
            int mid = lo + ((hi - lo) >> 1);
            if(target < arr[mid]){
                hi = mid - 1;
            }else if(target > arr[mid]){
                lo = mid + 1;
            }else{
                if(mid == arr.length - 1 || arr[mid + 1] != target){
                    return mid;
                }else{
                    lo = mid + 1;
                }
            }
        }
        return -1;
    }

    private static int findFirst(int[] arr, int target) {
        int lo = 0;
        int hi = arr.length;
        while (lo <= hi){
            int mid = lo + ((hi - lo) >> 1);
            if(target < arr[mid]){
                hi = mid - 1;
            }else if(target > arr[mid]){
                lo = mid + 1;
            }else{
                if(mid == 0 || arr[mid - 1] != target){
                    return mid;
                }else{
                    hi = mid - 1;
                }
            }
        }
        return -1;
    }
}
```

7.3 插值查找算法

插值查找算法的 `mid` 公式：

$$mid = low + \frac{key - a[low]}{a[high] - a[low]}(high - low)$$

7.3.1 注意事项

1. 对于数据量大，关键字分布比较均匀的查找表来说，采用插值查找速度较快。
2. 关键字分布不均匀的情况下，该方法不一定比折半查找要好。

7.3.2 代码实现

```java
public class InsertSearch {
    static int count;
    public static void main(String[] args) {
        int[] arr = new int[1000];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i + 1;
        }
        int ret = insertSearch(arr,7);
        System.out.println(ret);
        System.out.println(count);
    }

    public static int insertSearch(int[] arr,int target){
        if (target < arr[0] || target > arr[arr.length - 1]){
            return -1;
        }
        int lo = 0;
        int hi = arr.length - 1;
        while(lo <= hi){
            int mid  = lo + (hi - lo)*(target - arr[lo])/(arr[hi] - arr[lo]);
            count++;
            if(target < arr[mid]){
                hi = mid - 1;
            }else if(arr[mid] < target){
                lo = mid + 1;
            }else{
                return mid;
            }
        }
        return -1;
    }
}
```

7.4 斐波那契（黄金分割）查找算法

$$mid = lo + F(k-1) - 1$$

顺序表的长度不一定刚好等于 F(k) - 1，所以需要将原来的顺序表长度增加至 F(k) - 1。这里 k 值只要能使得 F(k) - 1，恰好大于或等于顺序表的长度即可。

```
while(F(k) - 1 < arr.length){
    k++;
}
```

7.4.1 代码实现

```
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;

public class FibSearch {
    public static void main(String[] args) {
//        int[] arr = {1,8,10,89,1000,1234,1235};
        int num = new Random().nextInt(100)+ 10;
        int[] arr = new int[num];
        for (int i = 0; i < num; i++) {
            arr[i] = new Random().nextInt(100)+ 10;
        }
        radixSort(arr);
        System.out.println(Arrays.toString(arr));
        System.out.println("请输入要查找的数: ");
        int target = new Scanner(System.in).nextInt();
        int ret = fibSearch(arr,target);
        System.out.println(ret);
    }

    //mid = low + fib(k - 1) - 1;
    public static int fibSearch(int[] arr,int target){
        int lo = 0;
        int hi = arr.length - 1;
        int len = arr.length;
        int k = 0;//表示斐波那契数列分割数值下标
        int mid = 0;
        int[] fib = fib(arr.length);//获取斐波那契数列
        //获取斐波那契数列分割数值的下标
        while (fib[k] - 1 < arr.length){
            k++;
        }
        //如果数组长度小于斐波那契数列数 - 1的值，把数组扩容至斐波那契数列数 - 1
        if (!(fib[k] - 1 == arr.length)){
            int[] arrTmp = Arrays.copyOf(arr,fib[k] - 1);
            for (int i = arr.length; i < fib[k] - 1; i++) {
```

```
37                  //扩容部分的数为原数组最后一个元素
38                  arrTmp[i] = arr[arr.length - 1];
39              }
40              arr = arrTmp;
41          }
42          while (lo <= hi) {
43              //数组长度为fib(k) - 1
44              //mid把数组分为左边是fib(k - 1) - 1
45              //右边长度是fib(k - 2) - 1
46              //fib(k) - 1 = fib(k - 1) - 1 + fib(k - 2) - 1 + 1
47              mid = lo + fib[k - 1] - 1;
48
49              if (target < arr[mid]) {
50                  hi = mid - 1;
51                  k--;
52              } else if(arr[mid] < target) {
53                  lo = mid + 1;
54                  k -= 2;
55              } else{
56                  if(mid >= len){
57                      return len - 1;
58                  }
59                  return mid;
60              }
61          }
62          return -1;
63      }
64
65      public static int[] fib(int n){
66          int[] arr = new int[n];
67          if (n == 1){
68              arr[0] = 1;
69              return arr;
70          }
71          arr[0] = 1;
72          if (n == 2){
73              arr[1] = 1;
74              return arr;
75          }
76          arr[1] = 1;
77          for (int i = 2; i < n; i++) {
78              arr[i] = arr[i - 1] + arr[i - 2];
79          }
80          return arr;
81      }
82  }
```

第八章　哈希表（散列表）hash

一个上机题：

某公司，当员工来报道时，要求将该员工的信息加入（id，姓名），当输入该员工的 id 时，要求查找到该员工所有信息。

要求：不使用数据库，速度越快越好（可以使用哈希表）

添加时，保证按照 id 从低到高的顺序插入。

8.0.1　散列函数

决定值对应哪个桶

哈希表是一个链表数组，即数组中的每个元素都是一个链表。该链表的引用指向的是有效节点。

8.1　代码实现

```
import java.util.Scanner;

public class HashTableDemo {
    public static void main(String[] args) {
        // 测试
        // 创建一个hash表
        HashTable hashTable = new HashTable(7);

        boolean flag = true;
        while (flag){
            System.out.println("----------------------------------------");
            System.out.println("a(add): 添加雇员！");
            System.out.println("l(list): 显示雇员！");
            System.out.println("f(find): 查找雇员！");
            System.out.println("d(del): 删除雇员");
            System.out.println("r(remove): 清空全部信息！");
            System.out.println("e(exit): 退出程序！");
            System.out.println("请输入选择：");
            String select = new Scanner(System.in).next();
            switch (select){
                case "e":
                    System.out.println("程序退出！");
                    flag = false;
                    break;
                case "a":
                    System.out.println("请输入员工id：");
                    int id = new Scanner(System.in).nextInt();
                    System.out.println("请输入员工姓名：");
                    String name = new Scanner(System.in).next();
                    Employee employee = new Employee(id,name);
                    hashTable.add(employee);
                    break;
                case "l":
```

```
                    hashTable.list();
                    break;
                case "f":
                    System.out.println("请输入要查找的雇员ID: ");
                    int findId = new Scanner(System.in).nextInt();
                    Employee findEmployee = hashTable.findEmployeeById(findId);
                    if (findEmployee == null){
                        System.out.println("未找到该id对应的员工");
                    }else{
                        System.out.println(findEmployee);
                    }
                    break;
                case "d":
                    System.out.println("请输入要删除的雇员ID: ");
                    int delId = new Scanner(System.in).nextInt();
                    hashTable.delEmployeeById(delId);
                    break;
                case "r":
                    hashTable.removeAll();
                    break;
                default:
                    System.out.println("输入错误, 请重新输入: ");
                    break;
            }
        }
    }
}
//一个雇员
class Employee{
    private int id;
    private String name;
    public Employee next;
    public Employee(){

    }
    public Employee(int id,String name){
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    @Override
    public String toString() {
        return "(id:" + id +
                ", name=" + name + ")";
    }
}

```

```
85 //创建employeeLinkedList，表示一条链表
86 class EmployLinkedList{
87     private Employee head;
88 
89     //添加雇员到链表
90     //添加雇员时，id总是增长的
91     //所以把雇员添加在链表的末尾
92     public void add(Employee employee){
93         if (head == null){
94             head = employee;
95             return;
96         }
97         Employee preEmployee = head;
98         while(preEmployee.next != null){
99             preEmployee = preEmployee.next;
100        }
101        preEmployee.next = employee;
102    }
103 
104    //删除雇员
105    public void delEmployeeById(int id){
106        Employee preEmployee = new Employee();
107        Employee tmpEmployee;
108        preEmployee.next = head;
109        tmpEmployee = preEmployee;
110        while(preEmployee.next != null){
111            if (preEmployee.next.getId() == id){
112                System.out.println("要删除的员工为：");
113                System.out.println(preEmployee.next);
114                preEmployee.next =  preEmployee.next.next;
115                head = tmpEmployee.next;
116                System.out.println("系统提示：删除成功");
117                return;
118            }
119            preEmployee = preEmployee.next;
120        }
121        System.out.println("该雇员不在此表中，无法删除！");
122        System.out.println("系统提示：删除失败");
123    }
124 
125    public Employee getHead() {
126        return head;
127    }
128 
129    public void setHead(Employee head) {
130        this.head = head;
131    }
132 
133    //遍历链表的信息
134    public void list(){
135        if (head == null){
```

```
136             System.out.print("[]");
137             return;
138         }
139 
140         Employee curEmployee = head;
141         System.out.print("[");
142         while (curEmployee != null){
143             System.out.print(curEmployee.toString());
144             if (curEmployee.next != null){
145                 System.out.print(",");
146             }
147             curEmployee = curEmployee.next;
148         }
149         System.out.print("]");
150     }
151 
152     //查找某个员工
153     public Employee findEmployeeById(int id){
154         Employee curEmployee = head;
155 
156         while (curEmployee != null && curEmployee.getId() != id){
157             curEmployee = curEmployee.next;
158         }
159         if (curEmployee != null){
160             return curEmployee;
161         }
162         return curEmployee;
163     }
164 }
165 
166 //创建哈希表
167 class HashTable{
168     private final int capacity;
169     EmployLinkedList[] employLinkedLists;
170     //构造方法
171     public HashTable(int capacity){
172         this.capacity = capacity;
173         employLinkedLists = new EmployLinkedList[this.capacity];
174         //此时该数组的每个元素都是null;要进行初始化
175         for (int i = 0; i < capacity; i++) {
176             employLinkedLists[i] = new EmployLinkedList();
177         }
178     }
179     public void add(Employee employee){
180         //根据员工的id, 得到该员工应当添加到哪条链表
181         employLinkedLists[hash(employee.getId())].add(employee);
182 
183     }
184 
185     public void list(){
186         System.out.print("{");
```

```
187         for (int i = 0; i < capacity; i++) {
188             employLinkedLists[i].list();
189             if (i != capacity - 1){
190                 System.out.print(",");
191             }
192         }
193         System.out.println("}");
194     }
195 
196     public Employee  findEmployeeById(int id){
197         return employLinkedLists[hash(id)].findEmployeeById(id);
198     }
199 
200     public void delEmployeeById(int id){
201         employLinkedLists[hash(id)].delEmployeeById(id);
202     }
203 
204     public void removeAll(){
205         System.out.println("系统提示：清空所有信息，此操作不可撤销，是否真的要删
                除？（Y:删除，N：取消）");
206         String  str = new Scanner(System.in).next();
207         if ("Y".equalsIgnoreCase(str)){
208             System.out.println("请再次确认是否要清空所有信息：（Y:确认，N：取消）"
                    );
209             str = new Scanner(System.in).next();
210             if (!("Y".equalsIgnoreCase(str))){
211                 return;
212             }
213             for (int i = 0; i < capacity; i++) {
214                 employLinkedLists[i].setHead(null);
215             }
216             System.out.println("系统提示：全部员工信息已删除！");
217         }else{
218             return;
219         }
220 
221     }
222 
223     //编写散列函数：使用取模法来完成
224     public int hash(int id){
225         return id % this.capacity;
226     }
227 
228 }
```

第九章　树

9.1　常用术语

节点：

根节点：没有父节点的节点

子节点：

叶子结点：没有孩子的节点

节点的权：

路径：从根节点找到该节点的路线

层：根节点为第一层，孩子为第二层，孙子为第三层，依次类推。

子树：

树的高度：

路径：一棵树中，从一个节点往下可以达到的孩子或孙子节点之间的通路称为路径。

路径长度：通路中分支数目称为路径长度，若规定根节点层数为 1，则从根节点到第 L 层节点的路径长度为 L-1.

节点的权：若将树中节点的赋予一个含有某种含义的数值，称该数值为该节点的权。

节点带权路径长度：从根节点到该节点之间的路径长度与该节点的权的乘积。

树的带权路径长度:树的带权路径长度规定为所有叶子节点的带权路径长度之和,记为 WPL（Weighted Path Length）, 权值越大的节点离根节点越近的二叉树才是最优二叉树。

9.2　二叉树

每个节点最多只能有两个孩子。

满二叉树：所有叶子节点都在最后一层。节点总数为 $2^n - 1$，n 为层数。

完全二叉树：所有叶子结点都在最后一层或倒数第二层，且最后一层的叶子节点在左边连续，倒数第二层的叶子结点在右边连续。

一棵深度为 k 的有 n 个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为 i（1 i n）的结点与满二叉树中编号为 i 的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。

9.2.1　二叉树的遍历

9.2.1.1　前序遍历

前序遍历即先遍历父节点，然后左子树，然后右子树。

9.2.1.2　中序遍历

先遍历左子树，再遍历父节点，再遍历右子树。

9.2.1.3　后序遍历

先遍历左子树，再遍历右子树，再遍历父节点。

9.2.1.4 代码实现

```
public class BinaryTreeDemo {
    public static void main(String[] args) {
        BinaryTree binaryTree = new BinaryTree();
        TreeNode treeNodeRoot = new TreeNode(1, "宋江");
        TreeNode treeNode1 = new TreeNode(2, "吴用");
        TreeNode treeNode2 = new TreeNode(3, "卢俊义");
        TreeNode treeNode3 = new TreeNode(4, "林冲");
        TreeNode treeNode4 = new TreeNode(5, "关胜");
        binaryTree.setRoot(treeNodeRoot);
        addLeft(treeNodeRoot, treeNode1);
        addRight(treeNodeRoot, treeNode2);
        addRight(treeNode2, treeNode3);
        addLeft(treeNode2, treeNode4);

        System.out.println("--------------------");
        binaryTree.preOrder();
        System.out.println("--------------------");
        binaryTree.infixOrder();
        System.out.println("--------------------");
        binaryTree.postOrder();

    }

    //插入左孩子
    public static void addLeft(TreeNode parentNode, TreeNode leftNode) {
        if (parentNode.leftChild != null) {
            System.out.println("该节点有左孩子, 无法插入");
            return;
        }
        parentNode.leftChild = leftNode;
        leftNode.parent = parentNode;
    }

    //插入右孩子
    public static void addRight(TreeNode parentNode, TreeNode rightNode) {
        if (parentNode.rightChild != null) {
            System.out.println("该节点有左孩子, 无法插入");
            return;
        }
        parentNode.rightChild = rightNode;
        rightNode.parent = parentNode;
    }
}

class BinaryTree {
    private TreeNode root;

    //前序遍历
```

```
49      public void preOrder() {
50          System.out.println("前序遍历的结果为: ");
51          if (this.root == null) {
52              System.out.println("{}");
53              return;
54          }
55          this.root.preOrder();
56      }
57
58      //中序遍历
59      public void infixOrder() {
60          System.out.println("中序遍历的结果为: ");
61          if (this.root == null) {
62              System.out.println("{}");
63              return;
64          }
65          this.root.infixOrder();
66      }
67
68      //后序遍历
69      public void postOrder() {
70          System.out.println("后序遍历的结果为: ");
71          if (this.root == null) {
72              System.out.println("{}");
73              return;
74          }
75          this.root.postOrder();
76      }
77
78
79      public TreeNode getRoot() {
80          return root;
81      }
82
83      public void setRoot(TreeNode root) {
84          this.root = root;
85      }
86  }
87
88  class TreeNode {
89      private int num;
90      private String name;
91      public TreeNode leftChild;
92      public TreeNode rightChild;
93      public TreeNode parent;
94
95      public TreeNode() {
96
97      }
98
99      public TreeNode(int num, String name) {
```

```
100         this.num = num;
101         this.name = name;
102     }
103 
104     public int getNum() {
105         return num;
106     }
107 
108     public void setNum(int num) {
109         this.num = num;
110     }
111 
112     @Override
113     public String toString() {
114         return "TreeNode{" +
115                 "num=" + num +
116                 ", name='" + name + '\'' +
117                 '}';
118     }
119 
120     //前序遍历的方法,迭代法
121     public void preOrder() {
122         System.out.println(this);
123         if (!(this.leftChild == null)) {
124             this.leftChild.preOrder();
125         }
126         if (!(this.rightChild == null)) {
127             this.rightChild.preOrder();
128         }
129     }
130 
131     //中序遍历: 递归法
132     public void infixOrder() {
133         if (!(this.leftChild == null)) {
134             this.leftChild.infixOrder();
135         }
136         System.out.println(this);
137         if (!(this.rightChild == null)) {
138             this.rightChild.infixOrder();
139         }
140     }
141 
142     //后序遍历
143     public void postOrder() {
144         if (!(this.leftChild == null)) {
145             this.leftChild.postOrder();
146         }
147 
148         if (!(this.rightChild == null)) {
149             this.rightChild.postOrder();
150         }
```

```
151         System.out.println(this);
152     }
153 }
```

9.2.2 查找指定节点

要求：编写前序、中序、后序查找方法。
分别使用三中查找方式，查找指定节点。
分析各种查找方式分别比较了多少次。
前序查找思路：

- 先判断当前节点的 num 是否等于要查找的。
- 如果相等，返回当前节点。
- 不相等，则判断当前左子节点是否为空，如果不为空，则递归前序查找。
- 左递归找到目标节点，返回目标节点，找不到就右子树递归查找。

9.2.2.1 代码实现

```
1  public class BinaryTreeDemo {
2      public static void main(String[] args) {
3          BinaryTree binaryTree = new BinaryTree();
4          TreeNode treeNodeRoot = new TreeNode(1, "宋江");
5          TreeNode treeNode1 = new TreeNode(2, "吴用");
6          TreeNode treeNode2 = new TreeNode(3, "卢俊义");
7          TreeNode treeNode3 = new TreeNode(4, "林冲");
8          TreeNode treeNode4 = new TreeNode(5, "关胜");
9          binaryTree.setRoot(treeNodeRoot);
10         addLeft(treeNodeRoot, treeNode1);
11         addRight(treeNodeRoot, treeNode2);
12         addRight(treeNode2, treeNode3);
13         addLeft(treeNode2, treeNode4);
14
15         System.out.println("--------------------");
16         binaryTree.preOrder();
17         System.out.println("--------------------");
18         binaryTree.infixOrder();
19         System.out.println("--------------------");
20         binaryTree.postOrder();
21         System.out.println("--------------------");
22         int target = 0;
23         System.out.println(binaryTree.preOrderSearch(target));;
24         System.out.println("--------------------");
25         System.out.println(binaryTree.infixOrderSearch(target));;
26         System.out.println("--------------------");
27         System.out.println(binaryTree.postOrderSearch(target));;
28
29     }
```

```
30
31      //插入左孩子
32      public static void addLeft(TreeNode parentNode, TreeNode leftNode) {
33          if (parentNode.leftChild != null) {
34              System.out.println("该节点有左孩子，无法插入");
35              return;
36          }
37          parentNode.leftChild = leftNode;
38          leftNode.parent = parentNode;
39      }
40
41      //插入右孩子
42      public static void addRight(TreeNode parentNode, TreeNode rightNode) {
43          if (parentNode.rightChild != null) {
44              System.out.println("该节点有左孩子，无法插入");
45              return;
46          }
47          parentNode.rightChild = rightNode;
48          rightNode.parent = parentNode;
49      }
50  }
51
52  class BinaryTree {
53      private TreeNode root;
54
55      //前序遍历
56      public void preOrder() {
57          System.out.println("前序遍历的结果为：");
58          if (this.root == null) {
59              System.out.println("{}");
60              return;
61          }
62          this.root.preOrder();
63      }
64
65      public TreeNode preOrderSearch(int target) {
66          return root.preOrderSearch(target);
67      }
68
69      //中序遍历
70      public void infixOrder() {
71          System.out.println("中序遍历的结果为：");
72          if (this.root == null) {
73              System.out.println("{}");
74              return;
75          }
76          this.root.infixOrder();
77      }
78
79      public TreeNode infixOrderSearch(int target) {
80          return root.infixOrderSearch(target);
```

```
81      }
82
83      //后序遍历
84      public void postOrder() {
85          System.out.println("后序遍历的结果为: ");
86          if (this.root == null) {
87              System.out.println("{}");
88              return;
89          }
90          this.root.postOrder();
91      }
92
93      public TreeNode postOrderSearch(int target) {
94          return root.postOrderSearch(target);
95      }
96
97      public TreeNode getRoot() {
98          return root;
99      }
100
101     public void setRoot(TreeNode root) {
102         this.root = root;
103     }
104 }
105
106 class TreeNode {
107     private int num;
108     private String name;
109     public TreeNode leftChild;
110     public TreeNode rightChild;
111     public TreeNode parent;
112
113     public TreeNode() {
114
115     }
116
117     public TreeNode(int num, String name) {
118         this.num = num;
119         this.name = name;
120     }
121
122     public int getNum() {
123         return num;
124     }
125
126     public void setNum(int num) {
127         this.num = num;
128     }
129
130     @Override
131     public String toString() {
```

```
132         return "TreeNode{" +
133                 "num=" + num +
134                 ", name='" + name + '\'' +
135                 '}';
136     }
137 
138     //前序遍历的方法,迭代法
139     public void preOrder() {
140         System.out.println(this);
141         if (!(this.leftChild == null)) {
142             this.leftChild.preOrder();
143         }
144         if (!(this.rightChild == null)) {
145             this.rightChild.preOrder();
146         }
147     }
148 
149     //前序遍历查找指定节点
150 
151     /**
152      * @param target 目标编号
153      * @return 返回对应节点
154      */
155     public TreeNode preOrderSearch(int target) {
156         if (this == null || this.num == target) {
157             return this;
158         }
159         TreeNode targetNode = null;
160         if (!(this.leftChild == null)) {
161             targetNode = this.leftChild.preOrderSearch(target);
162         }
163         if (targetNode != null) {
164             return targetNode;
165         }
166         if (!(this.rightChild == null)) {
167             targetNode = this.rightChild.preOrderSearch(target);
168         }
169         return targetNode;
170     }
171 
172     //中序遍历: 递归法
173     public void infixOrder() {
174         if (!(this.leftChild == null)) {
175             this.leftChild.infixOrder();
176         }
177         System.out.println(this);
178         if (!(this.rightChild == null)) {
179             this.rightChild.infixOrder();
180         }
181     }
182 
```

```
183     //中序遍历查找
184     public TreeNode infixOrderSearch(int target) {
185         if (this == null) {
186             return this;
187         }
188         TreeNode targetNode = null;
189         if (!(this.leftChild == null)) {
190             targetNode = this.leftChild.infixOrderSearch(target);
191         }
192         if (targetNode != null) {
193             return targetNode;
194         }
195         if (this.num == target) {
196             return this;
197         }
198         if (!(this.rightChild == null)) {
199             targetNode = this.rightChild.infixOrderSearch(target);
200         }
201         return targetNode;
202     }
203 
204     //后序遍历
205     public void postOrder() {
206         if (!(this.leftChild == null)) {
207             this.leftChild.postOrder();
208         }
209 
210         if (!(this.rightChild == null)) {
211             this.rightChild.postOrder();
212         }
213         System.out.println(this);
214     }
215 
216     public TreeNode postOrderSearch(int target) {
217         if (this == null) {
218             return this;
219         }
220         TreeNode targetNode = null;
221         if (!(this.leftChild == null)) {
222             targetNode = this.leftChild.postOrderSearch(target);
223         }
224         if (!(targetNode == null)) {
225             return targetNode;
226         }
227         if (!(this.rightChild == null)) {
228             targetNode = this.rightChild.postOrderSearch(target);
229         }
230         if (!(targetNode == null)) {
231             return targetNode;
232         }
233         if (this.num == target) {
```

```
234                 return this;
235             }
236             return targetNode;
237         }
238 }
```

9.2.3 删除节点

要求：

- 如果删除的是叶子节点，则删除该节点
- 如果删除的是非叶子节点，则删除该子树

思路：

- 对于单向的二叉树，应当判断当前节点的的子节点是否为待删除节点
- 如果树是空树，不用删除，如果只有一个 root 节点，且是目标元素，不用删除，是目标元素，等价于将树置空。
- 当前节点左子节点不为空，判断该左子节点是否为要删除的节点如果是，将当前节点的左孩子引用置为 null, 并且返回。
- 否则，如果当前节点的右子节点不为空，判断该右子节点是否为要删除的节点，如果是，将当前节点的右孩子引用置为空，并且返回。
- 如果上边两步没有删除节点，则需要向左子树进行递归删除。
- 否则就向右子树递归删除。

9.2.3.1 代码实现

```
1  public class BinaryTreeDemo {
2      public static void main(String[] args) {
3          BinaryTree binaryTree = new BinaryTree();
4          TreeNode treeNodeRoot = new TreeNode(1, "宋江");
5          TreeNode treeNode1 = new TreeNode(2, "吴用");
6          TreeNode treeNode2 = new TreeNode(3, "卢俊义");
7          TreeNode treeNode3 = new TreeNode(4, "林冲");
8          TreeNode treeNode4 = new TreeNode(5, "关胜");
9          binaryTree.setRoot(treeNodeRoot);
10         addLeft(treeNodeRoot, treeNode1);
11         addRight(treeNodeRoot, treeNode2);
12         addRight(treeNode2, treeNode3);
13         addLeft(treeNode2, treeNode4);
14
15         System.out.println("--------------------");
16         System.out.println("删除前: 前序遍历");
17         binaryTree.preOrder();
18         //删除
19         int del = 5;
```

```
20          binaryTree.delNode2(del);
21          System.out.println("--------------------");
22          System.out.println("删除后：前序遍历");
23          binaryTree.preOrder();
24      }
25
26      //插入左孩子
27
28      //插入右孩子
29
30  }
31
32  class BinaryTree {
33      private TreeNode root;
34
35      //删除1
36      public void delNode1(int del){
37          if (root == null){
38              System.out.println("系统提示：空树，无法删除！");
39              System.out.println("删除失败！");
40              return;
41          }
42          if (root.getNum() == del){
43              System.out.println("删除成功！");
44              root = null;
45              return;
46          }
47          root.delTreeNode(del);
48
49      }
50      //删除2
51      public void delNode2(int del){
52          TreeNode tmpNode = new TreeNode();
53          tmpNode.leftChild = root;
54          tmpNode.delTreeNode(del);
55          root = tmpNode.leftChild;
56
57      }
58      //前序遍历
59
60
61      //中序遍历
62
63      //后序遍历
64
65  }
66
67  class TreeNode {
68      private int num;
69      private String name;
70      public TreeNode leftChild;
```

```
71      public TreeNode rightChild;
72      public TreeNode parent;
73
74
75
76      //删除节点
77      //假设不是空树
78      public void delTreeNode(int del){
79          if (this.leftChild != null){
80              if (this.leftChild.num == del){
81                  this.leftChild = null;
82                  return;
83              }
84          }
85          if (this.rightChild != null){
86              if (this.rightChild.num == del){
87                  this.rightChild = null;
88                  return;
89              }
90          }
91          if (this.leftChild != null) {
92              this.leftChild.delTreeNode(del);
93          }
94          if (this.rightChild != null) {
95              this.rightChild.delTreeNode(del);
96          }
97      }
98
99
100     //前序遍历的方法,迭代法
101
102     //前序遍历查找指定节点
103
104     //中序遍历: 递归法
105
106     //中序遍历查找
107
108     //后序遍历
109
110 }
```

9.3 顺序存储二叉树

从数据存储来看，数组的存储方式和树的存储方式可以相互转换，即数组可以转换成树，树也可以转换成数组。

要求：

- 以数组的方式来存放数据。
- 要求遍历数组时，仍然可以以前序、中序、后序遍历的方式完成节点的遍历。

9.3.1 顺序存储二叉树的特点

1. 该二叉树是完全二叉树。
2. 第 n 个元素的左子节点为 $2n+1$，右子节点为 $2n+2$
3. 第 n 个元素的父节点为 $\frac{n-1}{2}$

4.n 表示二叉树中的第几个元素（从 0 开始）

9.3.2 代码实现

```
public class ArrBinaryTreeDemo {
    public static void main(String[] args) {
int[] arr = {1,2,3,4,5,6,7,8};
        ArrBinaryTree arrBinaryTree = new ArrBinaryTree(arr);
        int index = 0;
        System.out.println("先序");
        arrBinaryTree.preOrder();
        System.out.println("中序");
        arrBinaryTree.infixOrder();
        System.out.println("后序");
        arrBinaryTree.postOrder();
    }
}
class ArrBinaryTree{
    private int[] arr;

    public ArrBinaryTree(int[] arr){
        this.arr = arr;
    }

    //编写一个方法，完成顺序存储二叉树的遍历
    //先序
    public void preOrder(){
        preOrder(0);
    }
    public void preOrder(int index){
        if (arr == null || arr.length == 0){
            System.out.println("[]");
            return;
        }
        System.out.println(arr[index]);
        int l = 2*index + 1;
        if(l < arr.length){
            preOrder(l);
        }
        int r = 2*index+2;
        if (r < arr.length){
            preOrder(r);
        }
    }
    //中序
```

```
42      public void infixOrder(){
43          infixOrder(0);
44      }
45      public void infixOrder(int index){
46          if (arr == null || arr.length == 0){
47              return;
48          }
49          if (2*index + 1<arr.length){
50              infixOrder(2*index+1);
51          }
52          System.out.println(arr[index]);
53          if (2*index + 2<arr.length){
54              infixOrder(2*index+2);
55          }
56      }
57
58      //后序
59      public void postOrder(){
60          postOrder(0);
61      }
62      public void postOrder(int index){
63          if (arr == null || arr.length == 0){
64              return;
65          }
66          if (2*index + 1<arr.length){
67              postOrder(2*index+1);
68          }
69          if (2*index + 2<arr.length){
70              postOrder(2*index+2);
71          }
72          System.out.println(arr[index]);
73      }
74  }
```

9.4 线索化二叉树

9.4.1 基本介绍

1.n 个节点的二叉链表（完全二叉树）中含有 n+1 个空指针域，利用二叉链表的空指针域，存放指向该节点在某种遍历次序下的前驱和后继节点的指针，这种附加的指针称为线索。

2. 根据线索性质得不同，线索二叉树可以分为前序线索二叉树、中序线索二叉树、后序线索二叉树。

9.4.2 中序线索二叉树

线索化二叉树后，节点的 left 和 right，有如下两种情况：

- left 指向的是左子树，也可能指向前驱结点。
- right 指向的是右子树，也可能指向后继节点。

- left 为空，则该节点一定是遍历结果的第一个节点。
- right 为空，则该节点一定是遍历结果的最后一个节点。

9.4.3 代码实现

```
package com.atWSN.tree;

public class ThreadedBinaryTreeDemo {
    public static void main(String[] args) {
        TreeNode1 root = new TreeNode1(1, "tom");
        TreeNode1 node1 = new TreeNode1(3, "jack");
        TreeNode1 node2 = new TreeNode1(6, "smith");
        TreeNode1 node3 = new TreeNode1(8, "marry");
        TreeNode1 node4 = new TreeNode1(10, "king");
        TreeNode1 node5 = new TreeNode1(14, "dim");
        root.leftChild = node1;
        node1.leftChild = node3;
        node1.rightChild = node4;
        root.rightChild = node2;
        node2.leftChild = node5;
        ThreadedBinaryTree treeNode1 = new ThreadedBinaryTree();
        treeNode1.setRoot(root);
        treeNode1.threadedNodes();
        System.out.println(node4.leftChild);
        System.out.println(node4.rightChild);
    }
}

class ThreadedBinaryTree{
    private TreeNode1 root;
    //为了实现线索化，需要一个指向当前节点前驱结点的引用
    //在递归过程中，preNode始终指向该节点的前驱结点
    private TreeNode1 preNode = null;
    //删除1
    public void delNode1(int del){
        if (root == null){
            System.out.println("系统提示：空树，无法删除！");
            System.out.println("删除失败！");
            return;
        }
        if (root.getNum() == del){
            System.out.println("删除成功！");
            root = null;
            return;
        }
        root.delTreeNode(del);

    }
    //删除2
    public void delNode2(int del){
```

```
46          TreeNode1 tmpNode = new TreeNode1();
47          tmpNode.leftChild = root;
48          tmpNode.delTreeNode(del);
49          root = tmpNode.leftChild;
50
51      }
52      //前序遍历
53      public void preOrder() {
54          System.out.println("前序遍历的结果为: ");
55          if (this.root == null) {
56              System.out.println("{}");
57              return;
58          }
59          this.root.preOrder();
60      }
61
62      public TreeNode1 preOrderSearch(int target) {
63          return root.preOrderSearch(target);
64      }
65
66      //中序遍历
67      public void infixOrder() {
68          System.out.println("中序遍历的结果为: ");
69          if (this.root == null) {
70              System.out.println("{}");
71              return;
72          }
73          this.root.infixOrder();
74      }
75
76      public TreeNode1 infixOrderSearch(int target) {
77          return root.infixOrderSearch(target);
78      }
79
80      //后序遍历
81      public void postOrder() {
82          System.out.println("后序遍历的结果为: ");
83          if (this.root == null) {
84              System.out.println("{}");
85              return;
86          }
87          this.root.postOrder();
88      }
89
90      public TreeNode1 postOrderSearch(int target) {
91          return root.postOrderSearch(target);
92      }
93
94      public TreeNode1 getRoot() {
95          return root;
96      }
```

```
97
98      public void setRoot(TreeNode1 root) {
99          this.root = root;
100     }
101
102     public void threadedNodes(){
103         threadedNodes(root);
104     }
105
106     //对二叉树进行中序线索化的方法
107     public void threadedNodes(TreeNode1 node) {
108         //如果当前节点为空，不能进行线索化
109         if (node == null) {
110             return;
111         }
112         //先线索化左子树
113         threadedNodes(node.leftChild);
114         //再线索化当前节点
115         //先处理当前节点的前驱节点
116         if (node.leftChild == null) {
117             //如果当前节点的左孩子为空，就让其左孩子的引用指向其前驱节点
118             //将当前节点的leftType改为1
119             node.leftChild = preNode;
120             node.leftType = 1;
121         }
122         //处理后继节点，下次处理的
123         if (preNode != null && preNode.rightChild == null) {
124             preNode.rightChild = node;
125             preNode.rightType = 1;
126         }
127         //每处理一个节点后，让当前节点的前驱节点是下一个节点的前驱结点
128         preNode = node;
129         //再线索化右子树
130         threadedNodes(node.rightChild);
131     }
132 }
133
134 //创建节点
135 class TreeNode1 {
136
137     //leftType == 0 ，表示指向左子树
138     //leftType == 1，表示指向前驱结点
139     //rightType == 0 ，表示指向右子树
140     //rightType == 1，表示指向后继结点
141     public int leftType;
142     public int rightType;
143     private int num;
144     private String name;
145     public TreeNode1 leftChild;
146     public TreeNode1 rightChild;
147     public TreeNode1 parent;
```

```
148
149
150     public TreeNode1() {
151
152     }
153
154     public TreeNode1(int num, String name) {
155         this.num = num;
156         this.name = name;
157     }
158
159     public int getNum() {
160         return num;
161     }
162
163     public void setNum(int num) {
164         this.num = num;
165     }
166
167     @Override
168     public String toString() {
169         return "TreeNode{" +
170                 "num=" + num +
171                 ", name='" + name + '\'' +
172                 '}';
173     }
174
175     //删除节点
176     //假设不是空树
177     public void delTreeNode(int del) {
178         if (this.leftChild != null) {
179             if (this.leftChild.num == del) {
180                 this.leftChild = null;
181                 return;
182             }
183         }
184         if (this.rightChild != null) {
185             if (this.rightChild.num == del) {
186                 this.rightChild = null;
187                 return;
188             }
189         }
190         if (this.leftChild != null) {
191             this.leftChild.delTreeNode(del);
192         }
193         if (this.rightChild != null) {
194             this.rightChild.delTreeNode(del);
195         }
196     }
197
198
```

```
199     //前序遍历的方法,迭代法
200     public void preOrder() {
201         System.out.println(this);
202         if (!(this.leftChild == null)) {
203             this.leftChild.preOrder();
204         }
205         if (!(this.rightChild == null)) {
206             this.rightChild.preOrder();
207         }
208     }
209 
210     //前序遍历查找指定节点
211 
212     /**
213      * @param target 目标编号
214      * @return 返回对应节点
215      */
216     public TreeNode1 preOrderSearch(int target) {
217         if (this == null || this.num == target) {
218             return this;
219         }
220         TreeNode1 targetNode = null;
221         if (!(this.leftChild == null)) {
222             targetNode = this.leftChild.preOrderSearch(target);
223         }
224         if (targetNode != null) {
225             return targetNode;
226         }
227         if (!(this.rightChild == null)) {
228             targetNode = this.rightChild.preOrderSearch(target);
229         }
230         return targetNode;
231     }
232 
233     //中序遍历: 递归法
234     public void infixOrder() {
235         if (!(this.leftChild == null)) {
236             this.leftChild.infixOrder();
237         }
238         System.out.println(this);
239         if (!(this.rightChild == null)) {
240             this.rightChild.infixOrder();
241         }
242     }
243 
244     //中序遍历查找
245     public TreeNode1 infixOrderSearch(int target) {
246         if (this == null) {
247             return this;
248         }
249         TreeNode1 targetNode = null;
```

```
250         if (!(this.leftChild == null)) {
251             targetNode = this.leftChild.infixOrderSearch(target);
252         }
253         if (targetNode != null) {
254             return targetNode;
255         }
256         if (this.num == target) {
257             return this;
258         }
259         if (!(this.rightChild == null)) {
260             targetNode = this.rightChild.infixOrderSearch(target);
261         }
262         return targetNode;
263     }
264 
265     //后序遍历
266     public void postOrder() {
267         if (!(this.leftChild == null)) {
268             this.leftChild.postOrder();
269         }
270 
271         if (!(this.rightChild == null)) {
272             this.rightChild.postOrder();
273         }
274         System.out.println(this);
275     }
276 
277     public TreeNode1 postOrderSearch(int target) {
278         if (this == null) {
279             return this;
280         }
281         TreeNode1 targetNode = null;
282         if (!(this.leftChild == null)) {
283             targetNode = this.leftChild.postOrderSearch(target);
284         }
285         if (!(targetNode == null)) {
286             return targetNode;
287         }
288         if (!(this.rightChild == null)) {
289             targetNode = this.rightChild.postOrderSearch(target);
290         }
291         if (!(targetNode == null)) {
292             return targetNode;
293         }
294         if (this.num == target) {
295             return this;
296         }
297         return targetNode;
298     }
299 
300 
```

```
}
```

9.4.4 线索化二叉树的遍历

线索化二叉树后，二叉树的引用有所改变，不能使用原来的遍历方式来进行遍历。

9.4.4.1 代码实现

```
//线索化二叉树后的遍历
public void threadedList(){
    //定义一个临时变量来存储当前访问的节点
    //遍历是从root开始的
    TreeNode1 tmpNode =root;
    while(tmpNode != null){
        //循环遍历找到第一个leftType为1的节点
        //node随着遍历会变化
        //当leftType为1的时候，该节点是按照线索化处理后的有效节点
        while(tmpNode.leftType == 0){
            tmpNode = tmpNode.leftChild;
        }
        //打印当前节点
        System.out.println(tmpNode);
        //如果当前节点的右指针指向后继节点
        while (tmpNode.rightType == 1){
            tmpNode = tmpNode.rightChild;
            System.out.println(tmpNode);
        }
        //如果当前节点的右指针指向的是树
        //把当前节点替换为它的右孩子
        tmpNode = tmpNode.rightChild;
    }
}
```

第十章　赫夫曼树

给定 n 个权值作为 n 个叶子结点，构造一棵二叉树。若该树的带权路径长度达到最小，称这样的树为最优二叉树，也称赫夫曼树。

10.1　赫夫曼树实例

给定一个数列，要求转成一棵赫夫曼树。

思路分析：

1. 从小到大进行排序，将每个数据看成一个节点，每个节点都可以看做是一棵最简单的二叉树。

2. 取出根节点权值最小的两颗二叉树。

3. 将 2 中的的两颗二叉树组成一棵新的二叉树，该新的二叉树的根节点的权值是前面两棵二叉树根节点权值的和

4. 将新的二叉树，以根节点的权值大小再次排序，不断重复步骤 1-4, 直到数列中所有的数据都被处理。就得到一棵赫夫曼树。

10.2　代码实现

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class HuffmanTreeDemo {
    public static void main(String[] args) {
        int arr[] = {13, 7, 8, 3, 29, 6, 1};
        creatHuffmanTree(arr);
    }

    //创建赫夫曼树的方法
    public static Node creatHuffmanTree(int[] arr) {
        //第一步，为了操作方便，遍历arr
        //将arr的每一个元素构成一个node
        //将Node放入到Arraylist中
        List<Node> arrayList = new ArrayList<>();
        for (int i = 0; i < arr.length; i++) {
            arrayList.add(new Node(arr[i]));
        }
        while (arrayList.size() > 1) {
            //排序:权值从小到大
            Collections.sort(arrayList);
            //System.out.println(arrayList);
            //取出根节点权值最小的两颗二叉树

            Node left = arrayList.get(0);
            Node right = arrayList.get(1);
            //构建出一棵新的二叉树
            Node root = new Node(left.value + right.value);
```

```
            root.left = left;
            root.right = right;
            //从链表移除已构建的两个节点
            arrayList.remove(left);
            arrayList.remove(right);
            //把root添加到链表中
            arrayList.add(root);
        }
        return arrayList.get(0);
    }
}

class Node implements Comparable<Node> {
    int value;//节点的权值
    Node left;
    Node right;

    public Node() {

    }

    public Node(int value) {
        this.value = value;
    }

    @Override
    public String toString() {
        return "Node{" +
                "value=" + value +
                '}';
    }

    @Override
    public int compareTo(Node o) {
        //表示从小到大进行排序
        return this.value - o.value;
    }
}
```

10.3 赫夫曼编码

字符编码都不能是其他字符编码的前缀，符合此要求的编码焦作前缀编码，即不能匹配到重复的编码。思路：

- 给定字符串，统计各个字符对应的个数
- 按照字符出现的次数构建一棵赫夫曼树，次数作为权值。
- 根据赫夫曼树，给各个字符规定编码，向左的路径为 0，向右的路径为 1. 该编码为前缀编码。
- 按照赫夫曼编码，把字符串翻译为赫夫曼编码（这里使用的是无损压缩）

- 赫夫曼树不唯一，所以编码也不唯一，但最终压缩的长度是唯一的

10.3.1 数据压缩

功能：把字符串对应的赫夫曼树创建起来。

思路：

1. 构建新的节点：数据（指文本的字符/该字符对应的 ASCII 码）+ 权重（该字符总共出现的次数）+left+right。

2. 得到字符串对应的 `byte[]` 数组

3. 编写方法：将准备构建赫夫曼树的节点放入 `List` 中

4. 通过 3 中的 `List` 创建对应的赫夫曼树。

```
package com.atWSN.huffmantree.huffmancode;

import java.util.*;

public class HuffmanCode {
    public static void main(String[] args) {
        String content = "i like like like java do you like a java";
//        //得到字节数组
//        byte[] contentBytes = content.getBytes();
//
//        System.out.println(Arrays.toString(contentBytes));
//        List<Node> list = getNodes(contentBytes);
//        Collections.sort(list);
//        System.out.println(list);
//        System.out.println("-----------------------------------------------------");
    ;
//        Node root = huffmanTreeNode(list);
//        root.preOrder();
//
//        Map<Byte, String> huffmanCode = getCodes(root);
//        System.out.println(huffmanCode);
//
//        byte[] zip = zip(contentBytes,huffmanCode);
//        System.out.println(Arrays.toString(zip));
//        System.out.println((float)(contentBytes.length - zip.length)/
    contentBytes.length * 100);
        byte[] zip = huffmanZip(content);
        System.out.println(Arrays.toString(zip));
    }


    //使用一个方法，将前面的方法封装到一起，便于调用
    //直接处理字符串
    //返回经赫夫曼编码处理后的字节数组
    public static byte[] huffmanZip(String content){
//        //生成字节数组
//        byte[] contentBytes = content.getBytes();
//        //统计各字符的权值并创建字符对应的叶子节点
```

```
37 //         //将叶子节点放入到ArrayList中
38 //         List<Node > list =getNodes(contentBytes);
39 //         //生成哈夫曼树
40 //         Node root = huffmanTreeNode(list);
41 //         //通过哈夫曼树生成哈夫曼编码表
42 //         Map<Byte, String> huffmanCode = getCodes(root);
43 //         //返回经哈夫曼编码处理后的字节数组
44 //         return zip(contentBytes,huffmanCode);
45          //上边全部的代码可以写成下边这个
46          return zip(content.getBytes(),getCodes(huffmanTreeNode(getNodes(content.
                getBytes()))));
47      }
48      //处理字符串对应的字节数组
49      public static byte[] huffmanZip(byte[] bytes){
50          List<Node > list =getNodes(bytes);
51          Node root = huffmanTreeNode(list);
52          Map<Byte, String> huffmanCode = getCodes(root);
53          return zip(bytes,huffmanCode);
54      }
55
56
57      /**
58       * @param bytes 接收字节数组
59       * @return
60       */
61      public static List<Node> getNodes(byte[] bytes) {
62          List<Node> list = new ArrayList<>();
63          //遍历bytes, 统计每一个byte出现的次数, 使用map[key,value]
64          HashMap<Byte, Integer> counts = new HashMap<>();
65          for (byte b : bytes) {
66              Integer count = counts.get(b);
67              if (count == null) {//map中没有该数据
68                  counts.put(b, 1);
69              } else {
70                  counts.put(b, count + 1);
71              }
72          }
73          //构建Node
74          //把每一个键值对转成Node对象
75          for (Map.Entry<Byte, Integer> entry : counts.entrySet()) {
76              list.add(new Node(entry.getKey(), entry.getValue()));
77          }
78          return list;
79      }
80
81      //通过List生成赫夫曼树
82      public static Node huffmanTreeNode(List<Node> list) {
83          while (list.size() > 1) {
84              Collections.sort(list);
85              Node leftNode = list.get(0);
86              Node rightNode = list.get(1);
```

```
87              Node root = new Node(leftNode.weight + rightNode.weight);
88              root.right = rightNode;
89              root.left = leftNode;
90              list.add(root);
91              list.remove(leftNode);
92              list.remove(rightNode);
93          }
94          return list.get(0);
95      }
96
97      //通过赫夫曼树生成赫夫曼编码
98      //将赫夫曼编码表放入到map中去Map<Byte,String>形式
99      //32->01 97->100
100     static Map<Byte, String> huffmanCodes = new HashMap<>();
101
102     /**
103      * 功能：将所有传入的node节点的所有叶子结点的赫夫曼编码得到，并放入到
            huffmanCodes中
104      *
105      * @param node          传入的赫夫曼树节点
106      * @param code          路径：左子节点是0，右子节点是1
107      * @param stringBuilder 用于拼接路径的
108      */
109     private static void getCodes(Node node, String code, StringBuilder
            stringBuilder) {
110         //
111         StringBuilder stringBuilder2 = new StringBuilder(stringBuilder);
112         //将传入的code加入到stringBuilder2中
113         stringBuilder2.append(code);
114         if (node != null) {//node为空不处理
115             //判断当前节点是叶子还是非叶子
116             if (node.data == null) {//非叶子节点
117                 //递归处理
118
119                 //向左
120                 getCodes(node.left, "0", stringBuilder2);
121                 //向右递归
122                 getCodes(node.right, "1", stringBuilder2);
123             } else {//叶子节点
124                 //表示找到了某个叶子结点的最后
125                 huffmanCodes.put(node.data, stringBuilder2.toString());
126             }
127         }
128
129     }
130
131     //为了调用方便，重载该方法
132     private static Map<Byte, String> getCodes(Node root) {
133         if (root == null) {
134             System.out.println("哈夫曼数为空，无法构建哈夫曼码");
135             return null;
```

```
136         }
137         //生成赫夫曼编码表时，需要拼接路径，使用StringBuilder
138         StringBuilder stringBuilder = new StringBuilder();
139         getCodes(root, "", stringBuilder);
140         return huffmanCodes;
141     }
142 
143     //编写一个方法，将字符串对应的byte[]数组，通过生成的赫夫曼编码表，返回一个赫夫
        曼编码压缩后的byte[]
144 
145     /**
146      * @param bytes         原始字符串对应的byte[]数组
147      * @param huffmanCodes 由原始字符串生成的哈夫曼编码表
148      * @return 返回的是哈夫曼编码后转成的对应的byte数组
149      */
150     public static byte[] zip(byte[] bytes, Map<Byte, String> huffmanCodes) {
151         //先利用赫夫曼编码表将原始的byte[]数组转换为赫夫曼编码字符串
152         StringBuilder stringBuilder = new StringBuilder();
153         for (byte elem : bytes) {
154             stringBuilder.append(huffmanCodes.get(elem));
155         }
156         //这句是调试代码查看二进制码的，实际功能中不要
157         //System.out.println(stringBuilder.toString());
158 
159         //将字符串转为byte数组
160         //首先计算字节数组的长度，即stringBuilder的长度/8
161         int len = 0;
162         //为了便于统一，字节数组的最后一个用于存放赫夫曼编码后的最后那个字节的位长
            度
163         if (stringBuilder.length() % 8 == 0) {
164             len = stringBuilder.length() / 8 + 1;
165         } else {
166             len = stringBuilder.length() / 8 + 1 + 1;
167         }
168         int index = 0;//记录放入byte数组的位置
169         //创建存储压缩后的byte数组
170         byte[] tmp = new byte[len];
171         for (int i = 0; i < stringBuilder.length(); i += 8) {
172             String strByte;
173             if (i + 8 <= stringBuilder.length()) {
174                 strByte = stringBuilder.substring(i, i + 8);
175                 if (i + 8 == stringBuilder.length()) {
176                     tmp[len - 1] = 8;
177                 }
178             } else {
179                 strByte = stringBuilder.substring(i);
180                 tmp[len - 1] =  (byte) (stringBuilder.length() - i);
181             }
182             //将strByte转为byte数组
183             tmp[index++] = (byte) Integer.parseInt(strByte, 2);
184         }
```

```
185         return tmp;
186     }
187 }
188
189 class Node implements Comparable<Node> {
190     Byte data;//用于存放字符
191     int weight;//权值，表示data出现的次数
192     Node left;
193     Node right;
194
195     public Node() {
196
197     }
198
199     public Node(int weight) {
200         this.weight = weight;
201     }
202
203     public Node(Byte data, int weight) {
204         this.data = data;
205         this.weight = weight;
206     }
207
208     @Override
209     public int compareTo(Node o) {
210         return this.weight - o.weight;
211     }
212
213     @Override
214     public String toString() {
215         return "Node{" +
216                 "data=" + data +
217                 ", weight=" + weight +
218                 '}';
219     }
220
221     //前序遍历
222     public void preOrder() {
223         System.out.println(this);
224         if (this.left != null) {
225             this.left.preOrder();
226         }
227         if (this.right != null) {
228             this.right.preOrder();
229         }
230     }
231 }
```

10.3.2 解压缩

将数据通过赫夫曼编码压缩后，使用赫夫曼编码将压缩后的数据进行解码，重新得到原来的数据。解码的过程实际就是编码的逆向过程。

10.3.2.1 思路

1. 将压缩后的字节数组重新转为二进制码字符串；
2. 将该字符串对照赫夫曼编码重新转为字符串

10.3.2.2 代码实现

```
//解码

    //编写一个方法，完成对压缩数据的解码

    /**
     * @param huffmanCodes 哈夫曼编码表
     * @param huffmanBytes 压缩后得到的数组
     * @return 原字符串对应的字节数组
     */
    public static byte[] decode(Map<Byte, String> huffmanCodes, byte[]
        huffmanBytes) {
        //1.先得到huffmanBytes对应的二进制字符串
        StringBuilder stringBuilder = new StringBuilder();
        //处理huffmanBytes的前边字节，后两个单独处理
        for (int i = 0; i < huffmanBytes.length - 2; i++) {
            stringBuilder.append(byteToBitString(huffmanBytes[i]));
        }
        //后两个单独处理：哈夫曼编码处理的最后一个字节记录的是最后一个字节的长度
        int l = huffmanBytes[huffmanBytes.length - 1];
        String str  = Integer.toBinaryString((huffmanBytes[huffmanBytes.length -
            2]|256));//要和1 0000 0000(十进制为256)的或一下，正数补高位
        stringBuilder.append(str.substring(str.length() - l));
        //把字符串按照指定的赫夫曼编码进行解码
        //把赫夫曼编码表进行调换
        // 例如之前是97->100，即把97对应的字符转为赫夫曼编码100
        //现在要把100转为对应的字符
        //相当于是反向查询
        Map<String, Byte> map = new HashMap<>();
        for (Map.Entry<Byte, String> entry : huffmanCodes.entrySet()) {
            map.put(entry.getValue(), entry.getKey());
        }

        //创建一个集合存放byte
        List<Byte> list = new ArrayList<>();
        for (int i = 0; i < stringBuilder.length(); ) {
            int count = 1;
            boolean flag = true;
            Byte b = null;
```

```
            while (flag) {
                //取出一个'0'或'1'
                String key = stringBuilder.substring(i, i + count);
                b = map.get(key);
                if (b != null) {
                    flag = false;
                } else {//b为空，说明没有匹配到
                    count++;
                }

            }
            list.add(b);
            i += count;
        }
        byte[] b = new byte[list.size()];
        int index = 0;
        for (Byte by : list) {
            b[index++] = by;
        }
        return b;
    }


    /**
     * 功能：把压缩后的字节数组中的元素转为二进制字符字符串
     *
     * @param bt    传入的byte
     * @return 是bt对应的二进制字符串，是按补码返回的。
     */
    private static String byteToBitString(byte bt) {
        int tmp = bt;
            //对于正数，还存在补高位
        tmp |= 256;//256的二进制码是1 0000 0000；后八位刚好是0

        String str = Integer.toBinaryString(tmp);
        //int对应的字节是32位，要截取最后八位
            return str.substring(str.length() - 8);
    }
```

10.3.3 文件的压缩与解压缩

```
public static void main(String[] args) throws IOException {
//        String srcFile = "E://Desktop//数据结构.pdf";
//        String dstFile = "E://Desktop//KMP算法图压缩后.zip";
//        zipFile(srcFile, dstFile);
//        System.out.println("压缩文件成功");

        String zipFile = "E://Desktop//KMP算法图压缩后.zip";
        String dstFile = "E://Desktop//KMP算法图解压后.png";
unZipFile(zipFile,dstFile);
```

```
        System.out.println("解压缩文件成功");
    }

    //编写一个方法，将文件解压缩

    /**
     * @param zipFile 待解压文件
     * @param dstFile 文件解压到的路径
     */
    public static void unZipFile(String zipFile, String dstFile) throws
        IOException {
        //定义文件输入流
        InputStream is = null;
        //定义一个与文件输入关联的对象输入流
        ObjectInputStream ois = null;
        //定义文件的输出流
        OutputStream os = null;
        try {
            //创建文件输入流
            is = new FileInputStream(zipFile);
            //创建一个和is关联的对象输入流
            ois = new ObjectInputStream(is);
            //读取byte数组（哈夫曼编码处理后的那个数组）
            byte[] huffmanBytes = (byte[]) ois.readObject();
            Map<Byte, String> map = (Map<Byte, String>) ois.readObject();
            //解码
            byte[] bytes = decode(map, huffmanBytes);
            //将bytes写入到目标文件
            os = new FileOutputStream(dstFile);
            os.write(bytes);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                os.close();
                ois.close();
                is.close();
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }

    //编写方法，将一个文件进行压缩

    /**
     * @param srcFile 被压缩文件的全路径
     * @param dstFile 压缩后的文件存放的目录
     */
    public static void zipFile(String srcFile, String dstFile) {
        //创建输出流
```

```
60          OutputStream os = null;
61          ObjectOutputStream oos = null;
62          //创建文件的输入流
63          FileInputStream fileInputStream = null;
64          try {
65              fileInputStream = new FileInputStream(srcFile);
66              //创建一个和源文件一样的byte数组
67              byte[] b = new byte[fileInputStream.available()];//available()返回源文
                    件的大小
68              //读取文件
69              fileInputStream.read(b);//把文件的内容读到字节数组中
70              //获取到文件对应的赫夫曼编码
71
72              //直接对源文件进行了压缩
73              byte[] huffmanZip = huffmanZip(b);
74              //创建文件输出流，存放压缩文件
75              os = new FileOutputStream(dstFile);
76              //创建一个和文件输出流关联的ObjectOutStream
77              oos = new ObjectOutputStream(os);
78              //把赫夫曼编码后的字节数组写入压缩文件
79              oos.writeObject(huffmanZip);
80              //以对象流的方式写入赫夫曼编码，为了恢复源文件时使用
81              //注意一定要把赫夫曼编码写入压缩文件，否则无法恢复
82              oos.writeObject(huffmanCodes);
83          } catch (Exception e) {
84
85              System.out.println(e.getMessage());
86          } finally {
87              try {
88                  fileInputStream.close();
89                  oos.close();
90                  os.close();
91              } catch (Exception e) {
92                  System.out.println(e.getMessage());
93              }
94          }
95      }
```

第十一章　二叉排序树

二叉排序树 BST(Binaray Sort Tree)：对于二叉排序树的任何一个非叶子结点，要求左子节点的值小于当前节点的值，右子节点的值大于当前节点的值。

如果有相同的值，可将该值放入左子节点或右子节点。

11.1　二叉排序树的创建和遍历

把一个数组创建成对应的二叉排序树，并使用中序遍历二叉排序树。

11.2　代码实现

```
public class BinarySortTreeDemo {
    public static void main(String[] args) {
        int[] arr = {7, 3, 2,10, 1,12, 5, 1, 9};
        BinarySortTree binarySortTree = new BinarySortTree();
        for (int num :
                arr) {
            binarySortTree.add(new Node(num));
        }
        binarySortTree.infixOrder();
    }

}

//创建二叉排序树
class BinarySortTree {
    Node root;

    public void add(Node node) {
        if (root == null) {
            root = node;
            return;
        }
        root.add(node);
    }

    public void infixOrder() {
        if (root == null) {
            System.out.println("[]");
            return;
        }
        root.infixOrder();
    }

    public BinarySortTree() {

    }
```

```
}

//一个节点
class Node {
    public int num;
    public Node left;
    public Node right;

    public Node() {

    }

    public Node(int num) {
        this.num = num;
    }

    //添加节点的方法
    //递归的形式添加节点
    //必须满足二叉排序树
    public void add(Node node) {
        if (node == null) {
            System.out.println("节点指向为空, 无法添加");
            return;
        }
        //传入节点的值和当前子树根节点的值的关系
        if (this.num >= node.num) {
            if (this.left == null) {
                this.left = node;
                return;
            }
            this.left.add(node);
        } else {
            if (this.right == null) {
                this.right = node;
                return;
            }
            this.right.add(node);
        }

    }

    //中序遍历
    public void infixOrder() {
        if (this.left != null) {
            this.left.infixOrder();
        }

        System.out.println(this.num);

        if (this.right != null) {
            this.right.infixOrder();
```

```
88          }
89      }
90
91      @Override
92      public String toString() {
93          return "Node{" +
94                  "num=" + num +
95                  '}';
96      }
97  }
```

11.3 二叉排序树节点的删除

1. 删除叶子结点
2. 删除非叶子结点：该节点只有一棵子树，该节点有两棵子树。

11.3.1 思路

删除叶子结点：

- 找到要删除的节点 `targetNode`
- 应当找到待删除节点的父节点 `parentNode` (还应考虑是否存在父节点)
- 确定该节点是父节点的左子节点还是右子节点。
- 根据前面的情况来对应删除。

删除非叶子结点：只有一颗子树

- 找到要删除的节点 `targetNode`
- 应当找到待删除节点的父节点 `parentNode` (还应考虑是否存在父节点)
- 确定 `targetNode` 的子节点是左子节点还是右子节点
- 判断 `targetNode` 是 `parentNode` 的左子节点还是右子节点
- 如果 dmtargetNode 是 `parentNode` 的左子节点:dmtargetNode 的子树是左子树，`parentNode.left = targetNode.`
  如果 dmtargetNode 的子树是右子树 `parentNode.left = targetNode.right`
- 如果 dmtargetNode 是 `parentNode` 的右子节点:dmtargetNode 的子树是左子树，`parentNode.right = targetNode`
  如果 dmtargetNode 的子树是右子树 `parentNode.right = targetNode.right`

删除非叶子节点：有两棵子树
方式一：

- 找到要删除的节点 `targetNode`
- 应当找到待删除节点的父节点 `parentNode` (还应考虑是否存在父节点)
- 从 `targetNode` 的右子树找到最小的节点。
- 使用临时变量将该最小节点的值保存， `tmp`

- 删除最小值
- `targetNode = tmp`

方式二：

- 找到要删除的节点`targetNode`
- 应当找到待删除节点的父节点`parentNode`(还应考虑是否存在父节点)
- 从`targetNode`的左子树找到最大的节点。
- 使用临时变量将该最大节点的值保存，`tmp`
- 删除最大值
- `targetNode = tmp`

11.3.2 代码实现

```
public class BinarySortTreeDemo {
    public static void main(String[] args) {
        int[] arr = {7, 3, 6, 5, 1, 2};
        BinarySortTree binarySortTree = new BinarySortTree();
        for (int num :
                arr) {
            binarySortTree.add(new Node(num));
        }
        binarySortTree.infixOrder();
        System.out.println("----------------");
        System.out.println("删除节点");
        int del = new Scanner(System.in).nextInt();
        binarySortTree.delNode(del);
        binarySortTree.infixOrder();
    }

}

//创建二叉排序树
class BinarySortTree {
    Node root;

    public void add(Node node) {
        if (root == null) {
            root = node;
            return;
        }
        root.add(node);
    }

    //查找要删除的节点
    public Node search(int target) {
        if (root == null) {
```

```
34              return null;
35          }
36          return root.search(target);
37      }
38
39      //查找删除节点的父节点
40      public Node searchParent(int target) {
41          if (root == null || root.num == target) {
42              return null;
43          }
44          return root.searchParent(target);
45      }
46
47      //删除节点
48      public void delNode(int del) {
49          //空树
50          if (root == null) {
51              System.out.println("树为空，无法删除");
52              return;
53          }
54          Node delNode = search(del);
55          Node parentNode = searchParent(del);
56          //没有找到节点
57          if (delNode == null) {
58              System.out.println("未找到节点！");
59              return;
60          }
61
62          //待删除节点为叶子结点
63          if (delNode.left == null && delNode.right == null) {
64              //如果该树只有一个节点，且该节点为待删除节点
65              //此时该节点的父节点为null
66              if (parentNode == null) {
67                  root = null;
68                  return;
69              }
70              if (delNode == parentNode.left) {
71                  parentNode.left = null;
72                  return;
73              }
74              if (delNode == parentNode.right) {
75                  parentNode.right = null;
76                  return;
77              }
78          }
79          //待删除节点有两棵子树
80          if (delNode.right != null && delNode.left != null) {
81              //随机填入该节点左树的最大值或右树的最小值。
82              int random = new Random().nextInt(2);
83              switch (random) {
84                  case 0:
```

```
85                      int min = delRightTreeMin(delNode);
86                      delNode.num = min;
87                      break;
88                  case 1:
89                      int max = delLeftTreeMin(delNode);
90                      delNode.num = max;
91                      break;
92              }
93              return;
94          }
95          //删除节点只有一个子树
96          if (delNode.left == null) {
97              if (parentNode == null){
98                  root = root.right;
99                  return;
100             }
101             if (delNode == parentNode.left) {
102                 parentNode.left = delNode.right;
103                 return;
104             }
105             if (delNode == parentNode.right) {
106                 parentNode.right = delNode.right;
107                 return;
108             }
109         }
110         if (delNode.right == null) {
111             if (parentNode == null){
112                 root = root.left;
113                 return;
114             }
115             if (delNode == parentNode.left) {
116                 parentNode.left = delNode.left;
117                 return;
118             }
119             if (delNode == parentNode.right) {
120                 parentNode.right = delNode.left;
121                 return;
122             }
123         }
124     }
125
126     /**
127      * 功能:
128      * 1.以node为根节点的右子树的最小值
129      * 2.同时删除这个最小节点
130      *
131      * @param node 二叉排序树的根节点
132      * @return 以node为根节点的右子树的最小值
133      */
134     public int delRightTreeMin(Node node) {
135         int tmp = 0;
```

```
136         Node curNode = node;
137         if (node.right != null) {
138             node = node.right;
139         }
140         while (node.left != null) {
141             node = node.left;
142         }
143         tmp = node.num;
144         delNode(tmp);
145         return tmp;
146     }
147 
148     /**
149      * 功能:
150      * 1.以node为根节点的左子树的最大值
151      * 2.同时删除这个最大节点
152      *
153      * @param node 二叉排序树的根节点
154      * @return 以node为根节点的左子树的最大值
155      */
156     public int delLeftTreeMin(Node node) {
157         int tmp = 0;
158         Node curNode = node;
159         if (node.left != null) {
160             node = node.left;
161         }
162         while (node.right != null) {
163             node = node.right;
164         }
165         tmp = node.num;
166         delNode(tmp);
167         return tmp;
168     }
169 
170     public void infixOrder() {
171         if (root == null) {
172             System.out.println("[]");
173             return;
174         }
175         root.infixOrder();
176     }
177 
178     public BinarySortTree() {
179 
180     }
181 }
182 
183 //一个节点
184 class Node {
185     public int num;
186     public Node left;
```

```
187     public Node right;
188
189     public Node() {
190
191     }
192
193     public Node(int num) {
194         this.num = num;
195     }
196
197     //添加节点的方法
198     //递归的形式添加节点
199     //必须满足二叉排序树
200     public void add(Node node) {
201         if (node == null) {
202             System.out.println("节点指向为空，无法添加");
203             return;
204         }
205         //传入节点的值和当前子树根节点的值的关系
206         if (this.num >= node.num) {
207             if (this.left == null) {
208                 this.left = node;
209                 return;
210             }
211             this.left.add(node);
212         } else {
213             if (this.right == null) {
214                 this.right = node;
215                 return;
216             }
217             this.right.add(node);
218         }
219
220     }
221
222     //传入要删除的节点
223
224     /**
225      * @param target 希望删除节点的值
226      * @return 找到返回对应的节点，找不到则返回null
227      */
228     public Node search(int target) {
229         if (target == this.num) {//找到
230             return this;
231         } else if (target < this.num) {
232             if (this.left != null) {
233                 return this.left.search(target);
234             }
235         } else {
236             if (this.right != null) {
237                 return this.right.search(target);
```

```
238             }
239         }
240         return null;
241     }
242     //要删除节点的父节点
243
244     /**
245      * @param target 要找的节点的值
246      * @return 目标节点的父节点
247      */
248     public Node searchParent(int target) {
249         if ((this.left != null && this.left.num == target) || (this.right != null
                && this.right.num == target)) {
250             return this;
251         }
252         //如果查找的值小于当前节点的值，并且当前节点的左子树存在，递归的向左查找
253         if (target < this.num && this.left != null) {
254             return this.left.searchParent(target);
255         }
256         //如果查找得值大于当前节点的值，并且当前节点的右子树存在，递归的向右查找
257         if (target > this.num && this.right != null) {
258             return this.right.searchParent(target);
259         }
260         return null;
261     }
262
263     //中序遍历
264     public void infixOrder() {
265         if (this.left != null) {
266             this.left.infixOrder();
267         }
268
269         System.out.println(this.num);
270
271         if (this.right != null) {
272             this.right.infixOrder();
273         }
274     }
275
276     @Override
277     public String toString() {
278         return "Node{" +
279                 "num=" + num +
280                 '}';
281     }
282 }
```

第十二章　平衡二叉树：AVL 树

二叉排序树存在的问题，在某些情况下，二叉排序树会退化为链表，此时的查询速度明显降低（比单链表查询速度还低，因为相比单链表，二叉排序树每次都要判断另一个方向是否为空）。

解决方案：平衡二叉树。

12.1　概念

1. 平衡二叉树也叫平衡二叉搜索树，又被称为 AVL 树，可以保证查询效率较高。平衡二叉树必须满足二叉排序树。

2. 特点

- 它是一棵空树或它的左右两个子树的高度差的绝对值不超过 1.
- 并且左右两棵子树也都是一棵平衡二叉树。
- 实现方法：红黑树、AVL 树、替罪羊树、Treap 树、伸展树

12.2　左旋-zag

12.2.1　思路

1. 创建一个新的节点，该新节点的值为根节点的值
2. 新节点的左子树设为当前节点的左子树
3. 新节点的右子树设为当前节点右子树的左子树
4. 当前节点的值换为其右子节点的值
5. 把当前节点的右子树设为右子树的右子树
6. 当前节点的左子树设置为新的节点。

12.2.2　代码实现

```
public class AVLTreeDemo {
    public static void main(String[] args) {
        int[] arr = {4, 3, 6, 5, 7, 8};
        AVLTree avlTree = new AVLTree();
        for (int a : arr
        ) {
            avlTree.add(new Node(a));
        }
        avlTree.infixOrder();
        System.out.println(avlTree.root.height());
        System.out.println("--------------------");
        System.out.println(avlTree.root.leftHeight());
        System.out.println(avlTree.root.rightHeight());
    }
}

//创建AVL树
```

```
18 class AVLTree {
19     Node root;
20 
21     public void add(Node node) {
22         if (root == null) {
23             root = node;
24             return;
25         }
26         root.add(node);
27     }
28 
29     //查找要删除的节点
30     public Node search(int target) {
31         if (root == null) {
32             return null;
33         }
34         return root.search(target);
35     }
36 
37     //查找删除节点的父节点
38     public Node searchParent(int target) {
39         if (root == null || root.num == target) {
40             return null;
41         }
42         return root.searchParent(target);
43     }
44 
45     //删除节点
46     public void delNode(int del) {
47         //空树
48         if (root == null) {
49             System.out.println("树为空, 无法删除");
50             return;
51         }
52         Node delNode = search(del);
53         Node parentNode = searchParent(del);
54         //没有找到节点
55         if (delNode == null) {
56             System.out.println("未找到节点! ");
57             return;
58         }
59 
60         //待删除节点为叶子结点
61         if (delNode.left == null && delNode.right == null) {
62             //如果该树只有一个节点, 且该节点为待删除节点
63             //此时该节点的父节点为null
64             if (parentNode == null) {
65                 root = null;
66                 return;
67             }
68             if (delNode == parentNode.left) {
```

```
69                  parentNode.left = null;
70                  return;
71              }
72              if (delNode == parentNode.right) {
73                  parentNode.right = null;
74                  return;
75              }
76          }
77          //待删除节点有两棵子树
78          if (delNode.right != null && delNode.left != null) {
79              //随机填入该节点左树的最大值或右树的最小值。
80              int random = new Random().nextInt(2);
81              switch (random) {
82                  case 0:
83                      int min = delRightTreeMin(delNode);
84                      delNode.num = min;
85                      break;
86                  case 1:
87                      int max = delLeftTreeMin(delNode);
88                      delNode.num = max;
89                      break;
90              }
91              return;
92          }
93          //删除节点只有一个子树
94          if (delNode.left == null) {
95              if (parentNode == null) {
96                  root = root.right;
97                  return;
98              }
99              if (delNode == parentNode.left) {
100                 parentNode.left = delNode.right;
101                 return;
102             }
103             if (delNode == parentNode.right) {
104                 parentNode.right = delNode.right;
105                 return;
106             }
107         }
108         if (delNode.right == null) {
109             if (parentNode == null) {
110                 root = root.left;
111                 return;
112             }
113             if (delNode == parentNode.left) {
114                 parentNode.left = delNode.left;
115                 return;
116             }
117             if (delNode == parentNode.right) {
118                 parentNode.right = delNode.left;
119                 return;
```

```
            }
        }
    }

    /**
     * 功能:
     * 1.以node为根节点的右子树的最小值
     * 2.同时删除这个最小节点
     *
     * @param node 二叉排序树的根节点
     * @return 以node为根节点的右子树的最小值
     */
    public int delRightTreeMin(Node node) {
        int tmp = 0;
        Node curNode = node;
        if (node.right != null) {
            node = node.right;
        }
        while (node.left != null) {
            node = node.left;
        }
        tmp = node.num;
        delNode(tmp);
        return tmp;
    }

    /**
     * 功能:
     * 1.以node为根节点的左子树的最大值
     * 2.同时删除这个最大节点
     *
     * @param node 二叉排序树的根节点
     * @return 以node为根节点的左子树的最大值
     */
    public int delLeftTreeMin(Node node) {
        int tmp = 0;
        Node curNode = node;
        if (node.left != null) {
            node = node.left;
        }
        while (node.right != null) {
            node = node.right;
        }
        tmp = node.num;
        delNode(tmp);
        return tmp;
    }

    public void infixOrder() {
        if (root == null) {
            System.out.println("[]");
```

```
171             return;
172         }
173         root.infixOrder();
174     }
175 
176     public AVLTree() {
177 
178     }
179 }
180 
181 //一个节点
182 class Node {
183     public int num;
184     public Node left;
185     public Node right;
186 
187     public Node() {
188 
189     }
190 
191     public Node(int num) {
192         this.num = num;
193     }
194 
195     //左旋-zag
196     public void zag() {
197         Node newNode = new Node(this.num);
198         //新节点的左子树为当前节点的左子树
199         newNode.left = this.left;
200 
201         if (this.right != null) {
202             //新节点的右子树为当前节点右子树的左子树
203             newNode.right = this.right.left;
204             //当前节点的值置为右子节点的值
205             this.num = this.right.num;
206             //把当前节点的右子树设为当前节点右子树右子树的右子树
207             this.right = this.right.right;
208         }
209         //当前节点的左子树设置为新的节点
210         this.left = newNode;
211 
212     }
213 
214     //右旋-zig
215     public void zig() {
216         Node newNode = new Node(this.num);
217         //新节点的右子树为当前节点的右子树
218         newNode.right = this.right;
219 
220         if (this.left != null) {
221             //新节点的左子树为当前节点左子树的右子树
```

```
222             newNode.left = this.left.right;
223             //当前节点的值置为左子节点的值
224             this.num = this.left.num;
225             //把当前节点的左子树设为当前节点左子树左子树的右子树
226             this.left = this.left.left;
227         }
228         //当前节点的右子树设置为新的节点
229         this.right = newNode;
230 
231     }
232 
233     public int leftHeight() {
234         return left == null ? 0 : left.height();
235     }
236 
237     public int rightHeight() {
238         return right == null ? 0 : right.height();
239     }
240 
241     /**
242      * 功能:  返回以node为根节点的树的高度
243      *
244      * @return
245      */
246     public int height() {
247         return Math.max(left == null ? 0 : left.height(), right == null ? 0 :
                 right.height()) + 1;
248     }
249 
250     //添加节点的方法
251     //递归的形式添加节点
252     //必须满足二叉排序树
253     public void add(Node node) {
254         if (node == null) {
255             System.out.println("节点指向为空, 无法添加");
256             return;
257         } else if (this.num >= node.num) {//传入节点的值和当前子树根节点的值的关系
258             if (this.left == null) {
259                 this.left = node;
260             } else {
261                 this.left.add(node);
262             }
263         } else {
264             if (this.right == null) {
265                 this.right = node;
266             } else {
267                 this.right.add(node);
268             }
269         }
270 //当右子树高度高于左子树时: 即(rightHeight - leftHeight)>1, 要左旋
271         if ((this.rightHeight() - this.leftHeight()) > 1) {
```

```
272             if (this.right != null && this.right.rightHeight() < this.right.
                    leftHeight()) {
273                 this.right.zig();
274             }
275             zag();
276             return;
277         }
278         //顺时针旋转
279         if ((this.leftHeight() - this.rightHeight()) > 1) {
280             if (this.left != null && this.left.right.rightHeight() > this.left.
                    leftHeight()) {
281                 this.left.zag();
282             }
283             zig();
284         }
285     }
286 
287     //传入要删除的节点
288 
289     /**
290      * @param target 希望删除节点的值
291      * @return 找到返回对应的节点，找不到则返回null
292      */
293     public Node search(int target) {
294         if (target == this.num) {//找到
295             return this;
296         } else if (target < this.num) {
297             if (this.left != null) {
298                 return this.left.search(target);
299             }
300         } else {
301             if (this.right != null) {
302                 return this.right.search(target);
303             }
304         }
305         return null;
306     }
307     //要删除节点的父节点
308 
309     /**
310      * @param target 要找的节点的值
311      * @return 目标节点的父节点
312      */
313     public Node searchParent(int target) {
314         if ((this.left != null && this.left.num == target) || (this.right != null
                && this.right.num == target)) {
315             return this;
316         }
317         //如果查找的值小于当前节点的值，并且当前节点的左子树存在，递归的向左查找
318         if (target < this.num && this.left != null) {
319             return this.left.searchParent(target);
```

```
320             }
321             //如果查找得值大于当前节点的值，并且当前节点的右子树存在，递归的向右查找
322             if (target > this.num && this.right != null) {
323                 return this.right.searchParent(target);
324             }
325             return null;
326         }
327 
328         //中序遍历
329         public void infixOrder() {
330             if (this.left != null) {
331                 this.left.infixOrder();
332             }
333 
334             System.out.println(this.num);
335 
336             if (this.right != null) {
337                 this.right.infixOrder();
338             }
339         }
340 
341         @Override
342         public String toString() {
343             return "Node{" +
344                     "num=" + num +
345                     '}';
346         }
347 }
```

第十三章　多路查找树-B 树

13.1　B 树

B 树某个节点为 m，元素不超过 m-1，不低于 $\left\lceil \frac{m}{2} \right\rceil$,B 树又称为 $\left\lceil \frac{m}{2} \right\rceil, m$ 。

插入元素时，如果不满足 B 树规定，则发生分裂：将中间的元素移至父亲，同时该节点分为 2 个，父亲节点如果上溢，同样操作，直到根节点处。

删除元素时：如果该节点处不满足 B 树规定的个数，首先从左顾右盼，如果有兄弟，且兄弟的元素个数比 B 树规定的最小的节点个数还多就从兄弟那里借（注意要满足中序性的要求），如果没有符合条件的兄弟就从父亲那里拿一个跟兄弟拼接。从父亲那里拿一个后如果父亲不满足条件也按上述执行，直到根节点。

B 树所有的关键字存放在叶子或非叶子节点中，B+ 树搜索与 B 树基本相同，但所有的关键字都处于叶子节点的链表中。非叶子节点相当于是叶子结点的索引

B* 树是 B+ 树的变体，在 B+ 树的非根非叶子节点再增加指向兄弟的指针。

13.2　2-3 树

1.2-3 树的所有叶子节点都在同一层（只要是 B 树都满足这个条件）

2. 有两个子节点的节点叫二节点。二节点要么没有子节点，要么有两个子节点。

3. 有三个子节点的节点叫三节点，三节点要么没有子节点，要么有三个子节点。

4.2-3 树是由二节点和三节点构成的树。

第十四章　图

处理多对多的结构。

14.1　常用概念

1. 顶点：vertex
2. 边：edge
3. 路径
4. 无向图：顶点之间的连接没有方向
5. 有向图
6. 带权图

14.2　图的表示方式

二维数组（邻接矩阵），链表表示：邻接表。

邻接矩阵需要为每个顶点都分配 n 个边的空间。实际中有的边不存在，会造成一定的空间损失。

邻接表的实现只关心存在的边，不关心不存在的边，因此没有空间浪费。邻接表通常由链表 + 数组组成。

14.3　图的代码实现

对于邻接矩阵，1 表示直连，0 表示不能直接连接。

1. 要存储图中各顶点的信息。使用 String ArrayList 集合
2. 保存矩阵，使用二维数组表示边的关系。

```
import java.util.ArrayList;
import java.util.List;

public class GraphDemo {
    //存储顶点的信息，使用ArrayList
    public List<String> vertexList;
    private int[][] edges;//存储各条边的信息
    private int numEdges;//表示边的条数

    public static void main(String[] args) {
        int n = 5;
        String[] vertexValue = {"A", "B", "C", "D", "E"};
        GraphDemo graph = new GraphDemo(n);
        for (String vertex :
                vertexValue) {
            graph.insertVertex(vertex);
        }
        graph.showGraph();
        //添加边
        graph.insertEdge(0,1,1);
        graph.insertEdge(0,2,1);
```

```
22          graph.insertEdge(2,1,1);
23          graph.insertEdge(3,1,1);
24          graph.insertEdge(4,1,1);
25          System.out.println("-----------------------------------");
26          graph.showGraph();
27      }
28
29      //构造器
30      public GraphDemo(int n) {
31          //初始化矩阵和ArrayList
32          vertexList = new ArrayList<>();
33          edges = new int[n][n];
34      }
35
36      //插入顶点
37      public void insertVertex(String vertex) {
38          this.vertexList.add(vertex);
39      }
40      //添加边
41
42      /**
43       * @param v1      点1的下标
44       * @param v2      点2的下标
45       * @param weight  边的权值, 0或1
46       */
47      public void insertEdge(int v1, int v2, int weight) {
48          this.edges[v1][v2] = weight;
49          this.edges[v2][v1] = weight;
50          this.numEdges++;
51      }
52
53      //图的常用方法
54      //返回节点的个数
55      public int getNumOfVertex() {
56          return vertexList.size();
57      }
58
59      //返回边的数目
60      public int getNumOfEdges() {
61          return numEdges;
62      }
63
64      //返回节点i对应的数据,i指的是在节点顺序表中的下标
65      public String getVertexByIndex(int i) {
66          return vertexList.get(i);
67      }
68
69      //返回v1和v2的权值
70      public int getWeight(int v1, int v2) {
71          return edges[v1][v2];
72      }
```

```
73
74      public void showGraph() {
75          System.out.print("    ");
76          for (int i = 0; i < vertexList.size(); i++) {
77              System.out.print(vertexList.get(i));
78              System.out.print(" ");
79          }
80          System.out.println();
81          for (int i = 0; i < vertexList.size(); i++) {
82              System.out.print(vertexList.get(i));
83              System.out.print(": ");
84              for (int j = 0; j < vertexList.size(); j++) {
85                  System.out.print(edges[i][j]);
86                  System.out.print(" ");
87              }
88              System.out.println();
89          }
90
91      }
92  }
```

14.4 图的遍历

14.4.1 深度优先遍历

从初始访问节点出发，初始访问节点可能有多个邻接顶点，深度优先遍历的策略就是首先访问第一个邻接顶点，然后在以这个被访问的邻接顶点作为初始节点，访问它的第一个邻接顶点。相当于是每次都在访问完当前节点后首先访问当前节点的第一个邻接顶点。

步骤：

1. 访问初始节点 v，并标记节点 v 已访问。
2. 查找节点 v 的第一个邻接节点 w。
3. 若 w 存在，则继续执行 2，如果 w 不存在，则回到第一步，将从 v 的下一个节点继续。
4. 若 w 未被访问，对 w 进行深度优先遍历递归（即把 w 当做新的 v）。
5. 查找节点 v 的 w 邻接节点的下一个邻接节点，转到步骤 3.

14.4.2 代码实现

```
1  //类中添加一个boolean数组来标记各节点是否被访问
2
3  //节点访问状态
4      public boolean visit[];
5
6  //在构造器中对该数组进行初始化
7
8  //得到第一个邻接节点的下标
9      public int getFirstNeighbor(int index) {
10         for (int j = 0; j < vertexList.size(); j++) {
11             if (edges[index][j] > 0) {
```

```
12                  return j;
13              }
14          }
15          //没有就返回-1
16          return -1;
17      }
18      //根据前一个邻接节点的下标来获取下一个邻接节点
19
20      /**
21       * @param v1 当前节点
22       * @param v2 当前节点的邻接节点
23       * @return 下一个邻接节点
24       */
25      public int getNextNeighbor(int v1, int v2) {
26          for (int i = v2 + 1; i < vertexList.size(); i++) {
27              if (edges[v1][i] > 0) {
28                  return i;
29              }
30          }
31          return -1;
32      }
33
34      //深度优先遍历算法
35
36      /**
37       * @param index 从第index个节点开始访问
38       * @param visit 节点的访问状态
39       */
40      public void dfs(int index, boolean visit[]) {
41          //访问当前节点
42          System.out.println(vertexList.get(index));
43          //把当前节点访问状态变为已访问
44          this.visit[index] = true;
45          //获取当前节点的邻接节点
46          int w = getFirstNeighbor(index);
47          //如果该邻接节点存在
48          while (w != -1) {//说明找到了邻接顶点
49              //判断该邻接顶点是否被访问
50              if (!this.visit[w]) {//没有被访问，就以该节点为新的节点进行搜索
51                  dfs(w, this.visit);
52              } else {//如果w被访问过，就找index的下一个邻接节点
53                  w = this.getNextNeighbor(index, w);
54              }
55          }
56          //邻接节点不存在,如何访问全部的节点
57          //通过重载，对所有的节点进行dfs
58          //方法如下
59      }
60
61      public void dfs() {
62          //遍历所有的节点，进行dfs
```

```
63          for (int i = 0; i < vertexList.size(); i++) {
64              if (!visit[i]) {
65                  dfs(i, this.visit);
66              }
67          }
68      }
```

14.5 广度优先遍历 BFS(BroadFirst Search)

类似于分层搜索，广度优先遍历需要使用一个队列以保持访问过的节点的顺序。以便按这个顺序来访问这些节点的邻接节点。

14.5.1 算法步骤

1. 访问初始节点 v 并标记节点已被访问。
2. 节点入队列。
3. 队列非空时，继续执行，否则算法结束。
4. 出队列，取得队头节点 u。
5. 查找 u 的第一个邻接节点 w。
6. 如果节点 u 的邻接节点 w 不存在，转入步骤 3，否则：

- 若节点 w 尚未被访问，访问节点 w 同时，标记为已访问
- 节点 w 入队列。
- 查找 u 的继 w 邻接节点的下一个邻接节点 w, 转到步骤 6。

14.5.2 代码实现

```
1  //广度优先遍历
2      //对一个节点进行广度优先遍历
3      private void bfs(int index, boolean[] visit) {
4          //取出队列的头结点
5          int u;
6          //u的邻接点的下标w
7          int w;
8          //队列，用于记录节点的访问顺序
9          Queue<Integer> queue = new LinkedList<>();
10         //访问当前节点
11         System.out.println(vertexList.get(index));
12         //更改当前节点的访问状态
13         visit[index] = true;
14         //当前节点入队
15         queue.add(index);
16         //队列不为空时，进行访问
17         while (!queue.isEmpty()) {
18             //取出队首元素进行bfs
19             u = queue.poll();
20             //得到队首的第一个邻接节点
```

```
            w = getFirstNeighbor(u);
            while(w != -1){
                if (!visit[w]) {
                    //当前邻接节点没被访问
                    //则进行访问，同时入队列
                    System.out.println(vertexList.get(w));
                    visit[w] = true;
                    queue.add(w);
                }else{
                    w = getNextNeighbor(u,w);//广度优先
                }
            }
        }
    }

    //遍历所有的节点都进行广度优先搜索
    public void bfs(){
        visit = new boolean[vertexList.size()];
        for (int i = 0; i < vertexList.size(); i++) {
            if (!this.visit[i]){
                bfs(i,this.visit);
            }
        }
    }
```

14.6 图的全部代码汇总

```
package com.atWSN.graph;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class GraphDemo {
    //存储顶点的信息，使用ArrayList
    public List<String> vertexList;
    private int[][] edges;//存储各条边的信息
    private int numEdges;//表示边的条数

    //节点访问状态
    public boolean visit[];

    public static void main(String[] args) {
        int n = 5;
        String[] vertexValue = {"A", "B", "C", "D", "E"};
        GraphDemo graph = new GraphDemo(n);
        for (String vertex :
                vertexValue) {
```

```
23              graph.insertVertex(vertex);
24          }
25          graph.showGraph();
26          //添加边
27          graph.insertEdge(0, 1, 1);
28          graph.insertEdge(0, 2, 1);
29          graph.insertEdge(2, 1, 1);
30          graph.insertEdge(3, 1, 1);
31          graph.insertEdge(4, 1, 1);
32          System.out.println("-----------------------------------");
33          graph.showGraph();
34          graph.dfs();
35          System.out.println("-----------------------------------");
36          graph.bfs();
37      }
38
39      //构造器
40      public GraphDemo(int n) {
41          //初始化矩阵和ArrayList
42          vertexList = new ArrayList<>();
43          edges = new int[n][n];
44
45      }
46
47      //得到第一个邻接节点的下标
48      public int getFirstNeighbor(int index) {
49          for (int j = 0; j < vertexList.size(); j++) {
50              if (edges[index][j] > 0) {
51                  return j;
52              }
53          }
54          //没有就返回-1
55          return -1;
56      }
57      //根据前一个邻接节点的下标来获取下一个邻接节点
58
59      /**
60       * @param v1 当前节点
61       * @param v2 当前节点的邻接节点
62       * @return 下一个邻接节点
63       */
64      public int getNextNeighbor(int v1, int v2) {
65          for (int i = v2 + 1; i < vertexList.size(); i++) {
66              if (edges[v1][i] > 0) {
67                  return i;
68              }
69          }
70          return -1;
71      }
72
73      //深度优先遍历算法
```

```
74
75      /**
76       * @param index 从第index个节点开始访问
77       * @param visit 节点的访问状态
78       */
79      public void dfs(int index, boolean visit[]) {
80          //访问当前节点
81          System.out.println(vertexList.get(index));
82          //把当前节点访问状态变为已访问
83          this.visit[index] = true;
84          //获取当前节点的邻接节点
85          int w = getFirstNeighbor(index);
86          //如果该邻接节点存在
87          while (w != -1) {//说明找到了邻接顶点
88              //判断该邻接顶点是否被访问
89              if (!this.visit[w]) {//没有被访问，就以该节点为新的节点进行搜索
90                  dfs(w, this.visit);
91              } else {//如果w被访问过，就找index的下一个邻接节点
92                  w = this.getNextNeighbor(index, w);
93              }
94          }
95          //邻接节点不存在,如何访问全部的节点
96          //通过重载，对所有的节点进行dfs
97          //方法如下
98      }
99
100     public void dfs() {
101         visit = new boolean[vertexList.size()];
102         //遍历所有的节点，进行dfs
103         for (int i = 0; i < vertexList.size(); i++) {
104             if (!visit[i]) {
105                 dfs(i, this.visit);
106             }
107         }
108     }
109
110     //广度优先遍历
111     //对一个节点进行广度优先遍历
112     private void bfs(int index, boolean[] visit) {
113         //取出队列的头结点
114         int u;
115         //u的邻接点的下标w
116         int w;
117         //队列，用于记录节点的访问顺序
118         Queue<Integer> queue = new LinkedList<>();
119         //访问当前节点
120         System.out.println(vertexList.get(index));
121         //更改当前节点的访问状态
122         visit[index] = true;
123         //当前节点入队
124         queue.add(index);
```

```
125         //队列不为空时，进行访问
126         while (!queue.isEmpty()) {
127             //取出队首元素进行bfs
128             u = queue.poll();
129             //得到队首的第一个邻接节点
130             w = getFirstNeighbor(u);
131             while(w != -1){
132                 if (!visit[w]) {
133                     //当前邻接节点没被访问
134                     //则进行访问，同时入队列
135                     System.out.println(vertexList.get(w));
136                     visit[w] = true;
137                     queue.add(w);
138                 }else{
139                     w = getNextNeighbor(u,w);//广度优先
140                 }
141             }
142         }
143     }
144 
145     //遍历所有的节点都进行广度优先搜索
146     public void bfs(){
147         visit = new boolean[vertexList.size()];
148         for (int i = 0; i < vertexList.size(); i++) {
149             if (!this.visit[i]){
150                 bfs(i,this.visit);
151             }
152         }
153     }
154 
155     //插入顶点
156     public void insertVertex(String vertex) {
157         this.vertexList.add(vertex);
158     }
159     //添加边
160 
161     /**
162      * @param v1      点1的下标
163      * @param v2      点2的下标
164      * @param weight 边的权值，0或1
165      */
166     public void insertEdge(int v1, int v2, int weight) {
167         this.edges[v1][v2] = weight;
168         this.edges[v2][v1] = weight;
169         this.numEdges++;
170     }
171 
172     //图的常用方法
173     //返回节点的个数
174     public int getNumOfVertex() {
175         return vertexList.size();
```

```
176     }
177
178     //返回边的数目
179     public int getNumOfEdges() {
180         return numEdges;
181     }
182
183     //返回节点i对应的数据,i指的是在节点顺序表中的下标
184     public String getVertexByIndex(int i) {
185         return vertexList.get(i);
186     }
187
188     //返回v1和v2的权值
189     public int getWeight(int v1, int v2) {
190         return edges[v1][v2];
191     }
192
193     public void showGraph() {
194         System.out.print("    ");
195         for (int i = 0; i < vertexList.size(); i++) {
196             System.out.print(vertexList.get(i));
197             System.out.print(" ");
198         }
199         System.out.println();
200         for (int i = 0; i < vertexList.size(); i++) {
201             System.out.print(vertexList.get(i));
202             System.out.print(": ");
203             for (int j = 0; j < vertexList.size(); j++) {
204                 System.out.print(edges[i][j]);
205                 System.out.print(" ");
206             }
207             System.out.println();
208         }
209
210     }
211 }
```

第十五章　常用的算法

15.1　二分查找算法–非递归实现

只适用于有序数列查找，时间复杂度：O(log n)。
前边学过递归的方式，这里要求非递归的方式完成。

15.1.1　代码实现

```
package com.atWSN.Algorithm;

/**
 * 二分查找的算法
 * <p>
 * 要求：使用迭代的方式来完成
 */
public class BinarySearch {
    public static void main(String[] args) {

    }
    //二分查找的非递归实现

    /**
     * 功能：二分查找
     * 要求：迭代完成
     *
     * @param arr    从arr中查找,有序数组
     * @param target 查找元素target
     * @return 找到返回下标，没找到返回-1
     * @author 王松年
     */
    public static int binarySearch(int[] arr, int target) {
        int lo = 0;
        int hi = arr.length - 1;
        while (lo <= hi){
            int mid = lo + ((hi - lo) >> 1);
            if (arr[mid] == target){
                return mid;
            }else if(target < arr[mid]){
                hi = mid - 1;
            }else{
                lo = mid + 1;
            }
        }
        return -1;
    }

    //二分查找递归版
    public static int binarySearch(int arr[],int target,int lo,int hi){
```

```
41          if (lo > hi){
42              return -1;
43          }
44          int mid = lo + ((hi - lo)>>1);
45          if (arr[mid] == target){
46              return mid;
47          }else if(target < arr[mid]){
48              return binarySearch(arr,target,lo,mid -1);
49          }else{
50              return binarySearch(arr,target,mid + 1,hi);
51          }
52
53      }
54  }
```

15.2 分治算法

分治算法可以求解的经典问题：二分搜索、大整数乘法、棋盘覆盖、归并排序、快速排序、线性时间选择、最接近点对问题、循环赛日程表、汉诺塔。

15.2.1 分治算法的步骤

1. 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题。
2. 解决：若子问题的规模较小而容易被解决则直接解决，否则递归放入解各个子问题。
3. 合并：将各个子问题的解合并得到原问题的解。

15.2.2 设计模式

当问题的规模不超过某一阈值时，问题已容易解出，不必继续分解。（递归基）

```
1  if p < n0
2      then return (ADHOC(P))
3  //将P分解为较小的子问题P1、P2……
4  for i = 1 to k
5      do yi Divide and Conquer //递归的解决Pi
6  //合并子问题
7  MERGE（y1,y2,...）
8  return T
```

15.2.3 分治算法案例–汉诺塔

(应该是减治)

思路分析：

1. 只有一个盘：A->C
2. 盘数超过一个：

可以总是把盘分成两部分: 最下面一个盘和上边的那部分盘。

三步完成：把最上面的盘从 A 移动到 B，把最下面的盘从 A 移动到 C，把 B 的盘移动到 C。

15.2.3.1 代码实现

```
/**
 * 分治算法
 * 以求解汉诺塔为例
 */
public class HanoiTower {
    public static void main(String[] args) {
        hanoiTower(10, 'A', 'B', 'C');
    }

    /**
     * 使用分治算法
     *
     * @param n 汉诺塔的盘子数
     * @param a 盘子所在的柱子
     * @param b 辅助柱子
     * @param c 盘子要移动的柱子
     */
    public static void hanoiTower(int n, char a, char b, char c) {
        if (n == 1) {
            //如果只有一个盘，就移动这一个盘
            System.out.println("第1个盘从" + a + "移动到" + c+"上");
            return;
        }
        //把上边的n-1个盘从a移动到b上（借助c盘移动）
        hanoiTower(n - 1, a, c, b);
        //把最下边那个盘从a移动到c上
        System.out.println("第"+n+"个盘从" + a + "移动到" + c + "上");
        //把b上的盘借助a移动到c上
        hanoiTower(n - 1, b, a, c);
    }
}
```

15.3 动态规划算法

15.3.1 算法介绍

1. 将大问题划分为小问题进行解决，从而一步步获取最优解的处理算法。

2. 与分治算法不同的是，适合于用动态规划求解的问题，经分解得到的子问题往往不是相互独立的。（下一个子阶段的求解是建立在上一个子阶段求解的基础上）

3. 动态规划可以用填表的方式来逐步推进，得到最优解。

15.3.2 经典案例–背包问题

有一个背包，容量为 4 磅，现有如下物品：吉他（G）：1 磅，价值 1500；音响（S）：4 磅，价值 3000；电脑（L）：3 磅，价值 2000.

15.3.2.1 要求

1. 物品装入背包的总价值最大，且重量不能超过背包的总重。
2. 要求装入的物品不能重复。(01 背包，还有完全背包，完全背包指的是每种物品都有无限件可用)。

15.3.2.2 思路

对于给定的 n 个物品，设 `v[i] w[i]` 分别为第 `i` 个物品的价值和容量，`C` 为背包容量。令 `v[i][j]` 表示前 `i` 个物品能够装入到容量为 `j` 的背包中的最大价值。

动态规划的思想是每次遍历到第 `i` 个物品，根据 `w[i]` 和 `v[i]` 来确定是否需要将该物品放入到背包中。

公式：

```
1.v[i][0] = v[0][i]//表示填入的那张表的第一行第一列是0

2.if w[i] > j, then v[i][j] = v[i - 1][j]//增加一个商品，容量大于当前背包的容量时，就直接使用上一行的结果

3.if w[i] <= j,then v[i][j] = max(v[i-1][j],v[i-1][j-w[j]]+v[i])//增加一个商品，容量小于等于当前背包的容量时，取最大值：v[i-1][j]表示上一行的装入策略。v[i-1][j-w[j]]+v[i]表示把当前商品放入背包的同时，上一行中放入当前背包剩余空间的物品对应的价值。
```

15.3.2.3 代码实现

```java
package com.atWSN.Dynamic;

/**
 * 使用动态规划求解背包问题
 */
public class KnapsackProblem {
    public static void main(String[] args) {
        int[] weight = {4, 4, 4, 3};//用于保存物品的重量
        int[] value = {3000, 1000, 3000, 2000};//用于保存物品的价值
        final int capacity = 7;//背包的容量
        final int numOfGoods = value.length;

        int[][] valueOfKnapsack = new int[numOfGoods + 1][capacity + 1];

        //为了记录放入商品的情况，定义一个二维数组
        int[][] path = new int[numOfGoods + 1][capacity + 1];
        //初始化第0行
        for (int i = 0; i < capacity; i++) {
            valueOfKnapsack[0][i] = 0;
        }
        //初始化第0列
        for (int i = 0; i < value.length; i++) {
            valueOfKnapsack[i][0] = 0;
        }

        for (int i = 1; i <= value.length; i++) {//表示第i个物品
```

```
27              for (int j = 1; j <= capacity; j++) {//表示当前背包容量
28                  if (weight[i - 1] > j) {//当前物品大于背包容量
29                      valueOfKnapsack[i][j] = valueOfKnapsack[i - 1][j];
30                  } else {
31                      if (valueOfKnapsack[i - 1][j] > value[i - 1] + valueOfKnapsack
                            [i - 1][j - weight[i - 1]]) {
32                          valueOfKnapsack[i][j] =valueOfKnapsack[i - 1][j];
33                      } else {
34                          valueOfKnapsack[i][j] = value[i - 1] + valueOfKnapsack[i -
                                1][j - weight[i - 1]];
35                          path[i][j] = 1;
36                      }
37                  }
38              }
39          }
40          for (int i = 1; i < valueOfKnapsack.length; i++) {
41              for (int j = 1; j < valueOfKnapsack[0].length; j++) {
42                  System.out.print(valueOfKnapsack[i][j]);
43                  System.out.print(" ");
44              }
45              System.out.println();
46          }
47          System.out.println("背包放入的物品为: ");
48          int i = path.length - 1;
49          int j = path[0].length - 1;
50          while (i > 0 && j > 0) {
51              if (path[i][j] == 1) {
52                  System.out.printf("第%d个物品放入背包\n", i);
53                  j -= weight[i - 1];
54              }
55              i--;
56          }
57      }
58  }
```

15.4 串匹配算法

15.4.1 暴力匹配算法

15.4.2 KMP 算法

KMP 算法

15.5 贪心算法

15.5.1 算法介绍

1. 进行问题求解时，在每一步选择中都采取最好或者最优的选择，从而希望能够导致结果最好或最优的算法。

2. 贪心算法所得到的不一定是最优的结果，但是都是相对接近最优的结果

15.5.2 应用场景

集合覆盖问题：假设存在需要付费的广播电台，以及广播台信号可以覆盖的地区，如何选择最少的广播台，让所有的地区都可以接受信号。

k1: 北京，上海，天津
k2: 广州，北京，深圳
k3: 成都，上海，杭州
k4: 上海，天津
k5: 杭州，大连

15.5.2.1 思路分析

1. 遍历所有的广播台，找到一个覆盖了最多未覆盖的地区
2. 将这个电台加入到一个集合中，想办法把该电台覆盖的地区在下次比较时去掉
3. 重复第一步直到覆盖了全部的地区。

15.5.2.2 代码实现

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;

/**
 * 贪心算法求解集合覆盖问题
 */
public class GreedyAlgorithm {
    public static void main(String[] args) {
        //创建广播电台，放入到map
        HashMap<String, HashSet<String>> broadCasts = new HashMap<>();
        HashSet<String> strings1 = new HashSet<>();
        strings1.add("北京");
        strings1.add("上海");
        strings1.add("天津");
        broadCasts.put("k1", strings1);
        HashSet<String> strings2 = new HashSet<>();
        strings2.add("北京");
        strings2.add("广州");
        strings2.add("深圳");
        broadCasts.put("k2", strings2);
        HashSet<String> strings3 = new HashSet<>();
        strings3.add("成都");
        strings3.add("上海");
        strings3.add("杭州");
        broadCasts.put("k3", strings3);
        HashSet<String> strings4 = new HashSet<>();
//        strings1.add("北京");
        strings4.add("上海");
```

```
30          strings4.add("天津");
31          broadCasts.put("k4", strings4);
32          HashSet<String> strings5 = new HashSet<>();
33          strings5.add("杭州");
34          strings5.add("大连");
35  //          strings1.add("天津");
36          broadCasts.put("k5", strings5);
37          //存放所有的地区
38          HashSet<String> allAreas = new HashSet<>();
39          allAreas.add("北京");
40          allAreas.add("上海");
41          allAreas.add("天津");
42          allAreas.add("广州");
43          allAreas.add("深圳");
44          allAreas.add("杭州");
45          allAreas.add("成都");
46          allAreas.add("大连");
47          //存放所有的电台集合
48          //存放的是最终的电台
49          ArrayList<String> selects = new ArrayList<>();
50          //定义临时的集合，保存在遍历的过程中，存放遍历过程中电台覆盖的地区与当前没
                有覆盖地区的交集
51          HashSet<String> tmpSet = new HashSet<>();
52          //定义maxKey，用于记录在一次遍历过程中，能够覆盖最大未覆盖地区的那个电台
53          //如果maxKey不为空，则加入到select中
54          String maxKey = null;
55          while (!allAreas.isEmpty()) {//不为空，表示还没覆盖完
56              maxKey = null;
57              tmpSet.clear();
58              //遍历broadcasts
59              for (String key : broadCasts.keySet()) {
60                  HashSet<String> areas = broadCasts.get(key);//当前的k能够覆盖的地
                        区
61                  tmpSet.addAll(areas);//还未被覆盖的地区
62                  //和未覆盖地区取交集
63                  tmpSet.retainAll(allAreas);
64                  if (!tmpSet.isEmpty() && (maxKey == null || tmpSet.size() >
                        broadCasts.get(maxKey).size())) {
65                      //后边的那个条件体现了贪心算法
66                      //每次都选择最优的
67                      maxKey = key;
68                  }
69              }
70              if (maxKey != null) {
71                  selects.add(maxKey);
72                  //将maxKey指向的那个广播电台覆盖的地区从select清空
73                  allAreas.removeAll(tmpSet);
74              }
75          }
76      }
77  }
```

15.6 最小生成树相关算法

15.6.1 最小生成树

最小生成树 (Minimum Cost Spanning Tree)，简称 MST。

1. 给定一个带权的无向连通图，如何选取一棵生成树，使树上所有边上权的总和为最小，这叫最小生成树。

2.N 个顶点，则一定有 N-1 条边

3. 包含全部的顶点

4.N-1 条边全部都在图中。

求最小生成树的算法主要是普利姆算法和克鲁斯卡尔算法。

15.6.2 应用场景

有七个村庄 A、B、C、D、E、F、G，现在需要修路把七个村庄连通。各个村庄的距离用边线表示（权）。

A-B：5 公里；A-C：7 公里；A-G：2 公里；B-D：9 公里；B-G：3 公里；

C-E：8 公里；D-F：4 公里；E-F：5 公里；E-G：4 公里；F-G：6 公里

问：如何修路保证各个村庄都能联通，并且总的修建公路里程最短？

15.6.2.1 思路分析

尽可能选择少的路线，并且每条路线最小。

15.6.3 普利姆算法

普里姆算法求最小生成树，也就是在包含 n 个顶点的连通图中，找出只有 (n-1) 条边包含有 n 个顶点的连通子图，也就是所谓的极小连通子图。

1. 设 G=(V,E) 是联通网，T=(U,D) 是最小生成树，V，U 是顶点集合，E,D 是边集合。

2. 若从顶点 u 开始构造最小生成树,则从集合 v 中取出顶点 u 放入集合 U 中,标记顶点 v 的 `visited[u] = 1`

3. 若集合 U 中顶点 ui 与集合 V-E 中的顶点 vj 之间存在边，则寻找这些边中权值最小的边，但不能构成回路，将顶点 vj 加入集合 U 中，将边 (ui,vj) 加入集合 D 中，标记 `visited[vi] = 1`

4. 重复步骤 2，直到 U 与 V 相等，即所有顶点都被标记为访问过，此时 D 中有 n-1 条边

15.6.3.1 代码实现

```
1 package com.atWSN.Algorithm.Prim;
2
3 import java.util.Arrays;
4
5 /**
6  * 最小生成树问题
7  */
8 public class Prim {
9     public static void main(String[] args) {
```

```
10          char[] data = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
11          int capacity = data.length;
12          //邻接矩阵
13          int[][] weight =
14                  {
15                          //用0表示不连通
16                      //    A   B   C   D   E   F   G
17                          {0, 5, 7, 0, 0, 0, 2},
18                          {5, 0, 0, 9, 0, 0, 3},
19                          {7, 0, 0, 0, 8, 0, 0},
20                          {0, 9, 0, 0, 0, 4, 0},
21                          {0, 0, 8, 0, 0, 5, 4},
22                          {0, 0, 0, 4, 5, 0, 6},
23                          {2, 3, 0, 0, 4, 6, 0}
24                  };
25          //创建一个mGraph对象
26          MGraph mGraph = new MGraph(capacity);
27          MinTree minTree = new MinTree();
28          minTree.creatGraph(mGraph,data,weight);
29          minTree.showGraph(mGraph);
30          int[][] ret =  minTree.prim(mGraph,0);
31          System.out.println("最小生成树为: ");
32          for (int i = 0; i < ret.length; i++) {
33              System.out.println(""+data[ret[i][0]] + "<-"+weight[ret[i][0]][ret[i
                ][1]]+"->" +data[ret[i][1]]);
34          }
35      }
36  }
37
38  //创建最小生成树
39  class MinTree {
40      //创建邻接矩阵
41
42      /**
43       * @param mGraph    图的对象
44       * @param data      图中各节点的名字
45       * @param weight    图中各节点之间的权值
46       */
47      public void creatGraph(MGraph mGraph, char[] data, int[][] weight) {
48          for (int k = 0; k < mGraph.capacity; k++) {
49              mGraph.data[k] = data[k];
50              for (int i = 0; i < mGraph.capacity; i++) {
51                  mGraph.weight[k][i] = weight[k][i];
52              }
53          }
54      }
55
56      //显示图的方法:显示图的邻接矩阵
57      public void showGraph(MGraph mGraph) {
58          System.out.print("    ");
59          for (int i = 0; i < mGraph.capacity; i++) {
```

```
60              System.out.print(mGraph.data[i]);
61              System.out.print(" ");
62          }
63          System.out.println();
64          for (int i = 0; i < mGraph.capacity; i++) {
65              System.out.print(mGraph.data[i]);
66              System.out.print(": ");
67              for (int j = 0; j < mGraph.capacity; j++) {
68                  System.out.print(mGraph.weight[i][j]);
69                  System.out.print(" ");
70              }
71              System.out.println();
72          }
73      }
74
75      //编写prim算法，以生成最小生成树
76
77      /**
78       *
79       * @param mGraph 要创建最小生成树的那个图
80       * @param v 开始生成的顶点
81       */
82      public int[][] prim(MGraph mGraph,int v){
83          //创建一个数组来标记顶点是否被访问
84          boolean[] visit = new boolean[mGraph.capacity];
85          //将这些顶点初始标记为false
86          for (int i = 0; i < mGraph.capacity; i++) {
87              visit[i] = false;
88          }
89          //把当前节点标记为已访问
90          visit[v] = true;
91          //用于记录两个顶点的下标
92          int[][] edge = new int[mGraph.capacity - 1][2];
93          int minWeight = 0;
94          //找到最大的权值
95          for (int i = 0; i < mGraph.capacity; i++) {
96              for (int j = i; j < mGraph.capacity; j++) {
97                  if (mGraph.weight[i][j] > minWeight) {
98                      minWeight = mGraph.weight[i][j];
99                  }
100             }
101         }
102         //记录最大的权值
103         int max = minWeight;//
104
105         int h1 = 0;
106         int h2 = 0;
107         for (int i = 1; i < mGraph.capacity; i++) {//最小生成树有capacity-1条边
108             //确定每一次生成的子图，和哪个节点的距离最小
109             for (int j = 0; j < mGraph.capacity; j++) {
110                 //j表示被访问过的节点
```

```
111                 for (int k = 0; k < mGraph.capacity; k++) {
112                     //k表示未被访问的节点
113                     //寻找访问节点和未访问节点之间的最小权值
114                     //mGraph.weight[j][k]: 访问节点和未访问节点之间的权值
115                     if (visit[j] && !visit[k] && mGraph.weight[j][k] != 0 &&
                            mGraph.weight[j][k] < minWeight) {
116                         minWeight = mGraph.weight[j][k];
117                         //第i条边
118                         h1 = j;
119                         h2 = k;
120                     }
121                 }
122             }
123             //把权值最小的那个节点记录到最小生成树里
124             edge[i - 1][0] = h1;
125             edge[i - 1][1] = h2;
126             //将节点标记为已访问
127             visit[edge[i - 1][1]] = true;
128             minWeight = max;
129         }
130
131         return edge;
132     }
133 }
134
135 class MGraph {
136     final int capacity;//表示图的节点的个数
137     char[] data;//用于表示节点的名称
138     int[][] weight;//用于存放权值
139
140     public MGraph(int capacity) {
141         this.capacity = capacity;
142         data = new char[this.capacity];
143         weight = new int[this.capacity][this.capacity];
144     }
145 }
```

15.6.4 克鲁斯卡尔算法

15.6.4.1 公交站问题

某城市新增 7 个站点（A，B，C，D，E，F，G），现在需要修路把七个站点连通。各个站点的距离用边线来表示，如何修路保证各个站点都能联通，并且修建的公路总里程最短。

A<-12->B　A<-16->F　A<-14->G　B<-10->C　B<-7->F　C<-3->D
C<-5->E　C<-6->F　D<-4->E　E<-2->F　E<-8->G　F<-9->G

15.6.4.2 克鲁斯卡尔 (Kruskal) 算法介绍

用来求加权连通图的最小生成树的算法。

基本思想：按权值从小到大的顺序选择 n-1 条边，并保证这 n-1 条边不构成回路。

具体做法：首先构造一个只含有 n 个顶点的森林，然后依权值从小到大从联通网中选择边加入到森林中，并使森林中不产生回路，直到森林变成一棵树为止。

15.6.4.3 步骤

1. 将全部边的权值进行升序排序。
2. 依次选取最小权值的边及其对应的顶点加入至最小生成树中，注意不要构成回路。

终点：将所有顶点按从小到大的顺序排列好之后，某个顶点的终点就是与它连通的最大顶点。

回路：记录某顶点在最小生成树的终点，每次需要将一条边添加到最小生成树时，判断该边的两个顶点的终点是否重合，重合的话则会构成回路。

15.6.4.4 代码实现

```
package com.atWSN.Algorithm.Kruskal;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class Kruskal {
    private int edgeCapacity;//边的条数
    private char[] vertexs;//顶点的集合
    final private int vertexCapacity;//记录顶点的个数
    private int[][] matrix;
    private static final int max = Integer.MAX_VALUE;//用此值表示不能连通

    public static void main(String[] args) {
        char[] vertexs = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
        int[][] matrix = new int[][]
                {       //0表示自己与自己连通
                        {0, 12, max, max, max, 16, 14},
                        {12, 0, 10, max, max, 7, max},
                        {max, 10, 0, 3, 5, 6, max},
                        {max, max, 3, 0, 4, max, max},
                        {max, max, 5, 4, 0, 2, 8},
                        {16, 7, 6, max, 2, 0, 9},
                        {14, max, max, max, 8, 9, 0}
                };
        //创建Kruskal实例对象
        Kruskal graph = new Kruskal(vertexs, matrix);
        graph.print();
        List<EData> eData = new ArrayList<>();
        eData = graph.getEdges();
        System.out.println(eData);
        System.out.println("-----------------------------------");
        Collections.sort(eData);
        System.out.println(eData);
        List<EData> miniTree = graph.kruskal();
        System.out.println(miniTree);
```

```
38      }
39
40      public Kruskal(char[] vertexs, int[][] matrix) {
41          //初始化顶点的个数
42          this.vertexCapacity = vertexs.length;
43          //初始化顶点
44          this.vertexs = new char[this.vertexCapacity];
45          for (int i = 0; i < this.vertexCapacity; i++) {
46              this.vertexs[i] = vertexs[i];
47          }
48          //初始化边
49          this.matrix = new int[this.vertexCapacity][this.vertexCapacity];
50          for (int i = 0; i < this.vertexCapacity; i++) {
51              for (int j = 0; j < this.vertexCapacity; j++) {
52                  this.matrix[i][j] = matrix[i][j];
53              }
54          }
55          //统计边
56          for (int i = 0; i < this.vertexCapacity; i++) {
57              for (int j = i; j < this.vertexCapacity; j++) {
58                  if (this.matrix[i][j] < max) {
59                      this.edgeCapacity++;
60                  }
61              }
62          }
63      }
64
65      //打印邻接矩阵
66      public void print() {
67          System.out.println("邻接矩阵: ");
68          System.out.print("    ");
69
70          for (int i = 0; i < this.vertexCapacity; i++) {
71              System.out.printf("%3c", this.vertexs[i]);
72              System.out.print(" ");
73          }
74          System.out.println();
75          for (int i = 0; i < this.vertexCapacity; i++) {
76              System.out.print(this.vertexs[i]);
77              System.out.print(": ");
78              for (int j = 0; j < this.vertexCapacity; j++) {
79                  if (matrix[i][j] == max) {
80                      System.out.print("inf");
81                  } else {
82                      System.out.printf("%3d", matrix[i][j]);
83                  }
84                  System.out.print(" ");
85              }
86              System.out.println();
87          }
88      }
```

```
89
90      /**
91       * @param ch 顶点的值
92       * @return 返回该顶点的下标，找不到返回-1
93       */
94      private int getPosition(char ch) {
95          for (int i = 0; i < this.vertexCapacity; i++) {
96              if (vertexs[i] == ch) {
97                  return i;
98              }
99          }
100         return -1;
101     }
102
103     /**
104      * 功能：获取图中的边，放到EData顺序表中
105      * 通过权值数组(邻接矩阵)来遍历
106      */
107     public List<EData> getEdges() {
108         List<EData> getEdges = new ArrayList<>();
109         for (int i = 0; i < this.vertexCapacity; i++) {
110             for (int j = i + 1; j < this.vertexCapacity; j++) {
111                 if (this.matrix[i][j] > 0 && this.matrix[i][j] < max) {
112                     getEdges.add(new EData(this.vertexs[i], this.vertexs[j], this.
                            matrix[i][j]));
113                 }
114             }
115         }
116         return getEdges;
117     }
118
119     /**
120      * 功能：获取下标为i的顶点的终点
121      * 用于后面判断两个顶点的终点是否相同
122      *
123      * @param ends 记录了各个顶点对应的终点是哪个，该数组是在遍历的过程中逐步形成
             的
124      * @param i    传入的顶点对应的下标
125      * @return 下标为i的顶点对应的终点的下标
126      */
127     private int getEnd(int[] ends, int i) {
128         while (ends[i] != 0) {
129             i = ends[i];
130         }
131         return i;
132     }
133
134
135     /**
136      * Kruskal主算法
137      */
```

```
138     public List<EData> kruskal() {
139         //表示最后结果的顺序表的索引
140         int index;
141         //用于存放某已有最小生成树中的每个顶点在最小生成树中的数组
142         int[] ends = new int[this.edgeCapacity];
143         //存放最小生成树的集合
144         List<EData> miniTree = new ArrayList<>();
145 
146         //获取图中所有边的集合
147         List<EData> edges = getEdges();
148         //按边的权值进行排序
149         Collections.sort(edges);
150         //遍历所有的边，将权值最小的边添加到生成树时
151         //判断是否形成了回路：没有构成回路就加入最小生成树
152         for (EData edge :
153                 edges) {
154             //获取该边的两个端点
155             int p1 = getPosition(edge.start);
156             int p2 = getPosition(edge.end);
157             //获取p1这个顶点在已有的最小生成树中对应的终点
158             int m1 = getEnd(ends, p1);
159 
160             int m2 = getEnd(ends, p2);
161             //m1!=m2，说明不构成回路
162             if (m1 != m2) {
163                 ends[m1] = m2;//设置m1在已有最小生成树的终点
164                 miniTree.add(edge);
165             }
166         }
167         return miniTree;
168     }
169 }
170 
171 //创建一个边类，用于表示两点和这两点之间的边（权值）
172 class EData implements Comparable<EData> {
173     char start;
174     char end;
175     int weight;
176 
177     public EData(char start, char end, int weight) {
178         this.start = start;
179         this.end = end;
180         this.weight = weight;
181     }
182 
183     @Override
184     public String toString() {
185         return this.start + "<-" + this.weight + "->" + this.end;
186     }
187 
188     @Override
```

```
189     public int compareTo(EData o) {
190         return this.weight - o.weight;
191     }
192 }
```

15.7 最短路径相关算法

15.7.1 迪杰斯特拉 (Dijkstra) 算法

15.7.1.1 算法介绍

迪杰斯特拉 (Dijkstra) 算法是典型最短路径计算算法，用于计算一个节点到其他节点的最短路径。它的主要特点是以起始点为中心向外层层扩展（广度优先思想），直到扩展到终点为止。

15.7.1.2 算法过程

设置出发顶点 v，顶点集合 V，v 到 V 中个顶点的距离构成距离集合 Dis，Dis 集合记录着 v 到图中各顶点的距离（到自身的距离为 0）

1. 从 Dis 集合中选择值最小的 di 并移出 Dis 集合，同时移出 V 集合中对应的顶点 vi，此时 v 到 vi 即为最短路径。

2. 更新 Dis 集合，更新规则为：比较 v 到 V 集合中顶点的距离值，与 v 通过 vi 到 V 集合中顶点的距离值，保留较小的一个（同时更新顶点的前驱结点为 vi，表面是通过 vi 到达的）

3. 重复 1、2，直到最短路径顶点为目标顶点即可结束。

15.7.1.3 问题描述

有七个村庄，现有六个邮差，从村庄 G 出发，需要分别把邮件送到其他村庄。用边线的权表示各村庄之间的距离。

A-B：5 公里；A-C：7 公里；A-G：2 公里；B-D：9 公里；B-G：3 公里；

C-E：8 公里；D-F：4 公里；E-F：5 公里；E-G：4 公里；F-G：6 公里

问：如何计算出村庄 G 到其他各个村庄最短的距离？

如果从其他点出发到各个点的最短的距离又是多少？

15.7.1.4 代码实现

```
1  package com.atWSN.Algorithm.Dijkstra;
2
3  import java.util.Arrays;
4
5  public class Dijkstra {
6      public static void main(String[] args) {
7          char[] data = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
8  //         int capacity = data.length;
9          final int max = 65535;
10         //邻接矩阵
11         int[][] weight =
12                 {
13                         //用0表示不连通
```

```
                        //   A  B  C  D  E  F  G
                        {max, 5, 7, max, max, max, 2},
                        {5, max, max, 9, max, max, 3},
                        {7, max, max, max, 8, max, max},
                        {max, 9, max, max, max, 4, max},
                        {max, max, 8, max, max, 5, 4},
                        {max, max, max, 4, 5, max, 6},
                        {2, 3, max, max, 4, 6, max}
                };
        Graph graph = new Graph(data, weight);
        graph.print();
        graph.dijkstra(6);
    }
}

class Graph {
    private char[] vertex;//存放顶点的数组
    private int[][] weight;//邻接矩阵
    private VisitedVertex visitedVertex;

    public Graph(char[] vertex, int[][] weight) {
        this.vertex = vertex;
        this.weight = weight;
//        this.visitedVertex = new VisitedVertex(vertex.length,);
    }

    public void print() {
        for (int[] w : weight
        ) {
            System.out.println(Arrays.toString(w));
        }
    }

    /**
     * @param index 出发顶点对应的下标
     */
    public void dijkstra(int index) {
        visitedVertex = new VisitedVertex(vertex.length, index);
        update(index);//更新index顶点都周围顶点的距离和前驱
        for (int i = 0; i < vertex.length; i++) {
            index = visitedVertex.updateArr();//选择并返回新的访问顶点
            update(index);
        }
    }

    //更新index下标顶点到周围顶点的距离和周围顶点的前驱结点
    private void update(int index) {
        int len = 0;
        //遍历当前节点对应的邻接矩阵
        //更新顶点的前驱和距离
        for (int i = 0; i < weight[index].length; i++) {
```

```
65              //len: 出发顶点到index顶点的距离+index到顶点i的距离
66              len = visitedVertex.getDistance(index) + weight[index][i];
67              //如果index没有被访问过并且len小于出发顶点到i顶点的距离
68              if (!visitedVertex.in(i) && len < visitedVertex.getDistance(i)) {
69                  this.visitedVertex.updatePre(i, index);//更新i的前驱节点为index顶
                        点
70                  visitedVertex.upDateDis(i, len);//更新出发顶点到顶点i的距离
71              }
72          }
73      }
74  }
75
76  class VisitedVertex {
77      //记录顶点是否被访问
78      public boolean[] already_arr;
79      //记录该顶点的前驱顶点的下标（通过哪个节点访问的），动态更新
80      public int[] pre_visited;
81      //记录出发点到其他所有顶点的距离，动态更新
82      public int[] distance;
83
84      //index表示从哪个顶点开始构造
85      //capacity表示点的个数
86      public VisitedVertex(int capacity, int index) {
87          this.already_arr = new boolean[capacity];
88          this.already_arr[index] = true;//设置出发顶点被访问过
89          this.pre_visited = new int[capacity];
90          this.distance = new int[capacity];
91          //初始化各数组
92          Arrays.fill(this.pre_visited, -1);
93          this.pre_visited[index] = index;
94
95          Arrays.fill(this.distance, 65535);
96          this.distance[index] = 0;
97      }
98
99      /**
100      * 判断index是否被访问
101      *
102      * @param index
103      * @return 如果访问过，返回true，否则返回false
104      */
105     public boolean in(int index) {
106         return already_arr[index];
107     }
108
109     /**
110      * 更新出发顶点到index顶点的距离
111      *
112      * @param index
113      * @param len
114      */
```

```
115     public void upDateDis(int index, int len) {
116         this.distance[index] = len;
117     }
118 
119     /**
120      * 更新index顶点的前驱为pre
121      *
122      * @param pre
123      * @param index
124      */
125     public void updatePre(int index, int pre) {
126         pre_visited[index] = pre;
127     }
128 
129     /**
130      * 返回出发顶点到index顶点的距离
131      *
132      * @param index
133      * @return
134      */
135     public int getDistance(int index) {
136         return this.distance[index];
137     }
138 
139     //选择并返回新的访问节点，比如这里的G访问完后，就是A作为新的访问顶点
140     public int updateArr() {
141         int min = 65535;
142         int index = 0;
143         for (int i = 0; i < already_arr.length; i++) {
144             if (!already_arr[i] && distance[i] < min) {
145                 min = distance[i];
146                 index = i;
147             }
148         }
149         already_arr[index] = true;
150         return index;
151     }
152 }
```

15.7.2 佛洛依德算法

计算各个顶点之间的最短路径

15.7.2.1 代码实现

```
1 package com.atWSN.Algorithm.Floyd;
2 
3 public class Floyd {
4     public static void main(String[] args) {
5         char[] vertex = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
```

```
        final int max = 65535;
        //邻接矩阵
        int[][] weight =
                {
                        //用0表示不连通
                        //   A  B  C  D  E  F  G
                        {0, 5, 7, max, max, max, 2},
                        {5, 0, max, 9, max, max, 3},
                        {7, max, 0, max, 8, max, max},
                        {max, 9, max, 0, max, 4, max},
                        {max, max, 8, max, 0, 5, 4},
                        {max, max, max, 4, 5, 0, 6},
                        {2, 3, max, max, 4, 6, 0}
                };
        Graph graph = new Graph(vertex.length, weight, vertex);
        graph.show();
        System.out.println("-----------------------------------");
        graph.floyd();
        graph.show();
    }
}

class Graph {
    private char[] vertex;
    private int[][] dis;//保存从各个顶点出发到其他顶点的距离
    private char[][] pre;//保存到达目标顶点的前驱结点

    public Graph(int capacity, int[][] weight, char[] vertexs) {
        this.vertex = vertexs;
        this.dis = weight;
        this.pre = new char[vertexs.length][vertexs.length];
        for (int i = 0; i < vertexs.length; i++) {
            for (int j = 0; j < vertexs.length; j++) {
                if (i == j) {
                    pre[i][j] = vertex[i];
                } else {
                    pre[i][j] = '-';

                }
            }
        }
    }

    public void show() {
        System.out.print("   ");
        for (int i = 0; i < vertex.length; i++) {
            System.out.printf("%3c",vertex[i]);
            System.out.print(" ");
        }
        System.out.println();
        for (int i = 0; i < dis.length; i++) {
```

```
57              System.out.print(vertex[i]);
58              System.out.print(": ");
59              for (int j = 0; j < dis[0].length; j++) {
60                  if (dis[i][j] == 65535) {
61                      System.out.print("  n");
62                  } else {
63                      System.out.printf("%3d",dis[i][j]);
64                  }
65                  System.out.print(" ");
66              }
67              System.out.println();
68          }
69          System.out.println("-------------------------");
70          System.out.print("   ");
71          for (int i = 0; i < vertex.length; i++) {
72              System.out.print(vertex[i]);
73              System.out.print(" ");
74          }
75          System.out.println();
76          for (int i = 0; i < pre.length; i++) {
77              System.out.print(vertex[i]);
78              System.out.print(": ");
79              for (int j = 0; j < pre[0].length; j++) {
80                  System.out.print(pre[i][j]);
81                  System.out.print(" ");
82              }
83              System.out.println();
84          }
85      }
86
87      public void floyd() {
88          int len;
89          for (int i = 0; i < vertex.length; i++) {//中间顶点
90              for (int j = 0; j < vertex.length; j++) {//起点
91                  for (int k = 0; k < vertex.length; k++) {//终点
92                      len = dis[i][j] + dis[i][k];
93                      if (len < dis[j][k]) {
94                          pre[j][k] = vertex[i];
95                          dis[j][k] = len;
96                      }
97                  }
98              }
99          }
100         for (int i = 0; i < vertex.length; i++) {
101             for (int j = 0; j < vertex.length; j++) {
102                 if (pre[i][j] == '-'){
103                     pre[i][j] = vertex[i];
104                 }
105             }
106         }
107     }
```

```
108 }
```