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;
```

```
22          for (int[] row: chessArray1) {
23              for (int data : row) {
24                  if (data != 0){
25                      sum++;
26                  }
27              }
28          }
29          System.out.println("sum = " + sum);
30
31          //创建对应的稀疏数组
32          int sparseArray[][] = new int[sum + 1][3];
33          sparseArray[0][0] = 11;
34          sparseArray[0][1] = 11;
35          sparseArray[0][2] = sum;
36          int count = 0;
37          for (int i = 0; i < 11; i++) {
38              for (int j = 0; j < 11; j++) {
39                  if (chessArray1[i][j] != 0){
40                      count++;
41                      sparseArray[count][0] = i;
42                      sparseArray[count][1] = j;
43                      sparseArray[count][2] = chessArray1[i][j];
44                  }
45              }
46          }
47          //输出稀疏数组
48          System.out.println("输出稀疏数组");
49          for (int[] row: sparseArray) {
50              for (int data:row) {
51                  System.out.printf("%d\t",data);
52              }
53              System.out.println();
54          }
55          //稀疏数组转二维数组
56          System.out.println("--------------------------");
57          System.out.println("稀疏数组转二维数组:");
58          int chessArray2[][] = new int[sparseArray[0][0]][sparseArray[0][1]];
59          //遍历稀疏数组，把稀疏数组中存的信息还原回去
60          for (int i = 1; i <= sparseArray[0][2]; i++) {
61              chessArray2[sparseArray[i][0]][sparseArray[i][1]] = sparseArray[i][2];
62          }
63
64          for (int[] row: chessArray2) {
65              for (int data : row) {
66                  System.out.printf("%d\t",data);
67              }
68              System.out.println();
69          }
70      }
71  }
```

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。

6.8 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  }
```

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;
```

```
        System.out.println(binarySearch(arr,0,arr.length - 1,target));
        System.out.println(binarySearch1(arr,target));
        System.out.println(Arrays.toString(binarySearch2(arr,target)));
        System.out.println(insert(arr,target));
    }
    //递归版本
    public static int binarySearch(int[] arr,int lo,int hi,int value){
        if (lo > hi){
            return -1;
        }
        int mid = lo + ((hi - lo) >> 1);
        int midValue = arr[mid];
        if(midValue < value){
           return binarySearch(arr,mid + 1,hi,value);
        }else if(value <midValue){
            return binarySearch(arr,lo,mid - 1,value);
        }else{
            return mid;
        }
    }
    //迭代版本
    public static int binarySearch1(int[] arr,int value){
        int lo = 0;
        int hi = arr.length;
        while (lo <= hi){
            int mid = lo + ((hi - lo) >> 1);
            if(value < arr[mid]){
                hi = mid - 1;
            }else if(value > arr[mid]){
                lo = mid + 1;
            }else{
                return mid;
            }
        }
        return -1;
    }

    //寻找插入值
    //如果元素存在，返回该元素最后一个所在的位置
    //如果元素不存在，返回其插入位置的前一个位置
    public static  int insert(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{
                return findLast(arr,target);
```

```
58              }
59          }
60          return lo - 1;
61      }
62
63      //找到数组中所有的目标值元素，返回其范围
64      //没有则返回[-1,-1]
65      public static int[] binarySearch2(int[] arr,int target){
66          return new int[]{findFirst(arr,target),findLast(arr,target)};
67      }
68
69      private static int findLast(int[] arr, int target) {
70          int lo = 0;
71          int hi = arr.length;
72          while (lo <= hi){
73              int mid = lo + ((hi - lo) >> 1);
74              if(target < arr[mid]){
75                  hi = mid - 1;
76              }else if(target > arr[mid]){
77                  lo = mid + 1;
78              }else{
79                  if(mid == arr.length - 1 || arr[mid + 1] != target){
80                      return mid;
81                  }else{
82                      lo = mid + 1;
83                  }
84              }
85          }
86          return -1;
87      }
88
89      private static int findFirst(int[] arr, int target) {
90          int lo = 0;
91          int hi = arr.length;
92          while (lo <= hi){
93              int mid = lo + ((hi - lo) >> 1);
94              if(target < arr[mid]){
95                  hi = mid - 1;
96              }else if(target > arr[mid]){
97                  lo = mid + 1;
98              }else{
99                  if(mid == 0 || arr[mid - 1] != target){
100                     return mid;
101                 }else{
102                     hi = mid - 1;
103                 }
104             }
105         }
106         return -1;
107     }
108 }
```

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：
子树：
树的高度：

9.1.1　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。