第3章 栈和队列

3.1 栈

3.1.1 栈的基本概念

栈是一种特殊的线性表，其插入、删除操作只能在表的尾部进行。在栈中允许进行插入、删除操作的一端称为栈顶，另一端称为栈底。在栈$\{a_0,a_1,\cdots,a_{n-1}\}$中$a_0$称为栈底元素，$a_{n-1}$称为栈顶元素。通常，栈的插入操作叫入栈，栈的删除操作叫出栈。

由于栈的插入和删除操作只允许在栈顶进行，每次入栈的元素即成为栈顶元素，每次最先出栈的总是栈顶元素，所以栈是一种后进先出的线性表。就像一摞盘子，每次将一个盘子摞在最上面，每次从最上面取一只盘子，不能从中间插进或者抽出。

3.1.2 栈的抽象数据类型描述

栈中的数据元素和数据间的逻辑关系与线性表相同，是由n个具有相同数据类型的数据元素构成的有限序列，栈的抽象数据类型的Java描述如下：

```
1  package ch03;
2  public interface IStack {
3  public void clear();                    //将栈置空
4  public boolean isEmpty();               //判断栈是否为空
5  public int length();                    //返回栈的数据元素个数
6  public Object peek();                   //返回栈顶元素
7  public void push(Object x);             //将数据元素 x 入栈
8  public Object pop();                    //将栈顶元素出栈并返回
9  public void display();                  //输出栈中的所有元素
10 }
```

栈的抽象数据Java接口包含了栈的主要基本操作，如果要使用这个接口还需要具体的类来实现。栈的Java接口的实现方法主要有以下两种：

(1) 基于顺序存储的实现，为顺序栈；

(2) 基于链式存储的实现，为链栈。

3.1.3 顺序栈

1. 顺序栈类的描述

顺序栈用数组实现，因为入栈和出栈操作都是在栈顶进行，所以增加变量top来指示栈

顶元素的位置，top 指向栈顶元素存储位置的下一个存储单元的位置，空栈时 top＝0。

顺序栈类的 Java 语言描述如下：

```
1  package ch03;
2  public class SqStack implements IStack{

3  private Object[] stackElem;                    //顺序栈存储空间
4  private int top;                               //指向栈顶元素的下一个存储单元位置
5  private int maxSize;                           //栈的最大存储单元个数
6
7  //构造最大存储单元个数为 maxSize 的空栈
8  public SqStack(int maxSize){
9      top = 0;
10     this.maxSize = maxSize;
11     stackElem = new Object[maxSize];
12 }
13 //将栈置空
14 public void clear() {
15     top = 0;
16 }
17     //判断栈是否为空
18 public boolean isEmpty() {
19     return top == 0;
20 }
21     //返回栈中数据元素的个数
22 public int length() {
23
24     return top;
25 }
26     //返回栈顶元素
27 public Object peek() {
28     if(!isEmpty())
29         return stackElem[top - 1];
30     else
31         return null;
32 }
33     //入栈
34 public void push(Object x) {
35
36 }
37     //出栈
38 public Object pop() {
39
40 }
41 //输出栈中的所有数据元素
42 public void display(){
43     for(int i = top - 1;i >= 0;i -- ){
44         System.out.print(stackElem[i] + " ");
45     }
46 }
47 }
```

2. 顺序栈基本操作的实现

1) 入栈操作

入栈操作 push(x)是将数据元素 x 作为栈顶元素插入顺序栈中,主要操作如下:

(1) 判断顺序栈是否为满,若满则抛出异常。

(2) 将 x 存入 top 所指的存储单元位置。

(3) top 加 1。

图 3.1 显示了进行入栈操作时栈的状态变化。

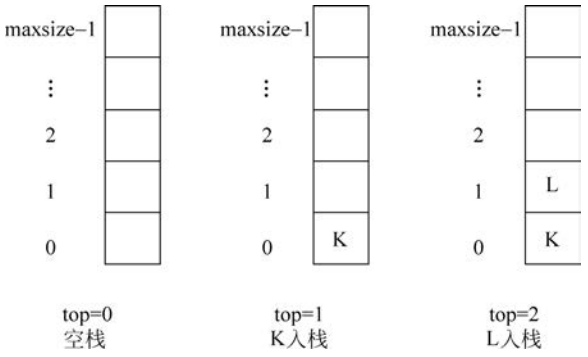

图 3.1 入栈操作

【算法 3.1】 入栈操作。

```
1  //入栈
2  public void push(Object x) {
3      if(top == maxSize)
4          throw new Exception("栈已满");
5      stackElem[top] = x;
6      top++;
7  }
```

2) 出栈操作

出栈操作 pop()是将栈顶元素从栈中删除并返回,主要步骤如下。

(1) 判断顺序栈是否为空,若空则返回 null。

(2) top 减 1。

(3) 返回 top 所指的栈顶元素的值。

图 3.2 显示了进行出栈操作时栈的状态变化。

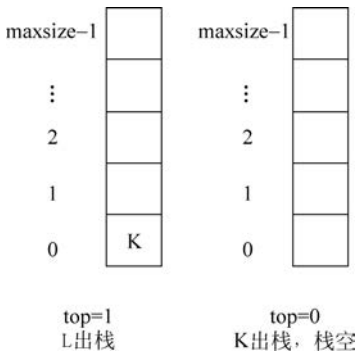

图 3.2 出栈操作

【算法 3.2】 出栈操作。

```
1  public Object pop() {
2      if(isEmpty())
3          return null;
4      top--;
5      return stackElem[top];
6  }
```

分析可得，入栈和出栈操作的实现为顺序表的尾插入和尾删除，时间复杂度为 $O(1)$。

【例 3.1】 利用顺序栈实现括号匹配的语法检查。

解：括号匹配是指程序中出现的括号，左、右括号的个数是相同的，并且需要先左后右依次出现。括号是可以嵌套的，一个右括号与其前面最近的一个左括号匹配，使用栈保存多个嵌套的左括号。

```
public void isMatched(String str) throws Exception{
      for(int i = 0;i < str.length();i++){
          if(str.charAt(i) == '('){                          //左括号入栈
              push('(');
          }
          else if(str.charAt(i) == ')'&&!isEmpty()){         //左括号出栈
              pop();
          }
          else if(str.charAt(i) == ')'&&isEmpty()){          //存在多余的右括号
              System.out.println("括号不匹配");
              return;
          }
      }
      if(isEmpty()){
          System.out.println("括号匹配");
      }
      else{                                                  //存在多余的左括号
          System.out.println("括号不匹配");
      }
  }
  public static void main(String []args) throws Exception{
      String str1 = "(a + b * (c + d))";
      String str2 = "(a + b * (c + d)))";
      SqStack q = new SqStack(str1.length());
      q.isMatched(str1);
      SqStack p = new SqStack(str2.length());
      p.isMatched(str2);
  }
```

3.1.4 链栈

1. 链栈类的描述

采用链式存储结构的栈称为链栈，由于入栈和出栈只能在栈顶进行，不存在在栈的任意位置进行插入和删除的操作，所以不需要设置头结点，只需要将指针 top 指向栈顶元素结点，每个结点的指针域指向其后继结点即可。

链栈的存储结构如图 3.3 所示。

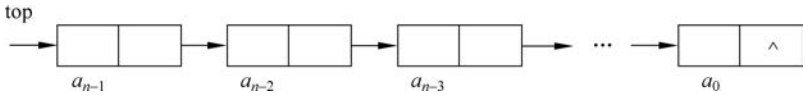

图 3.3 链栈的存储结构

实现 IStack 接口的链栈类的 Java 语言描述如下：

```
package ch03;
import ch02.Node;
public class LinkStack implements IStack{
    private Node top;                                    //栈顶元素

    //将栈置空
    public void clear() {
        top = null;
    }
    //判断栈是否为空
    public boolean isEmpty() {
        return top == null;
    }
    //返回栈中数据元素个数
    public int length() {
        Node p = top;
        int length = 0;
        while(p!= null){
            p = p.next;
            length++;
        }
        return length;

    }
    //返回栈顶元素
    public Object peek() {
        if(!isEmpty())
            return top.data;
        else
            return null;
    }
    //入栈
    public void push(Object x) throws Exception {
        Node s = new Node(x);
        s.next = top;
        top = s;
    }
    //出栈
    public Object pop() {
        if(isEmpty())
            return null;
        Node p = top;
        top = top.next;
        return p.data;
    }
```

```
46      //输出栈中的所有数据元素
47      public void display(){
48          Node p = top;
49          while(p!= null){
50              System.out.print(p.data + " ");
51              p = p.next;
52          }
53      }
54  }
```

2. 链栈基本操作的实现

1）入栈操作

入栈操作 push(x)是将数据域值为 x 的结点插入到链栈的栈顶，主要步骤如下。

(1) 构造数据值域为 x 的新结点。

(2) 改变新结点和首结点的指针域，使新结点成为新的栈顶结点。

链栈进行入栈操作后的状态变化如图 3.4 所示。

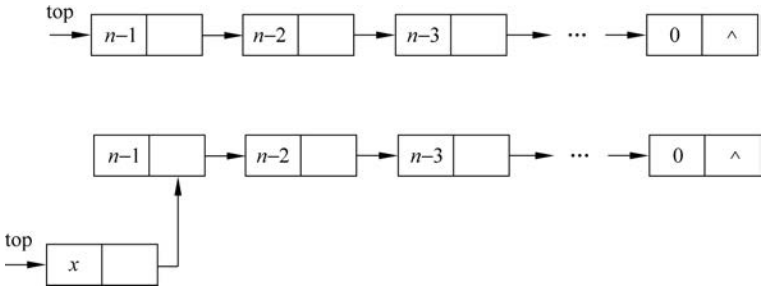

图 3.4　入栈操作

【算法 3.3】 入栈操作。

```
1   public void push(Object x) throws Exception {
2       Node s = new Node(x);
3       s.next = top;
4       top = s;
5   }
```

2）出栈操作

出栈操作 pop()是将栈顶元素从链栈中删除并返回其数据域的值，主要步骤如下。

(1) 判断链栈是否为空，若空则返回 null。

(2) 修改 top 指针域的值，返回被删结点的数据域值。

链栈进行出栈操作后的状态变化如图 3.5 所示。

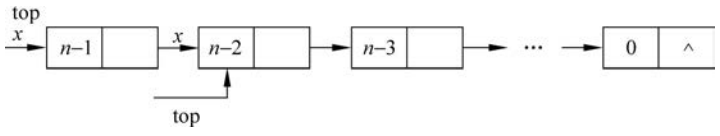

图 3.5　出栈操作

【算法 3.4】 出栈操作。

```
1  public Object pop() {
2      if(isEmpty())
3          return null;
4      Node p = top;
5      top = top.next;
6      return p.data;
7  }
```

分析可得,使用单链表实现栈,入栈和出栈操作的实现为单链表的头插入和头删除,时间复杂度为 $O(1)$。

【例 3.2】 设有编号为 1、2、3、4 的 4 辆列车顺序进入一个栈式结构的车站,具体写出这 4 辆列车开出车站的所有可能的顺序。

解:至少有 14 种。

全进之后再出的情况只有 1 种:4,3,2,1。

进 3 个之后再出的情况有 3 种:3,4,2,1; 3,2,4,1; 3,2,1,4。

进两个之后再出的情况有 5 种:2,4,3,1; 2,3,4,1; 2,1, 3,4; 2,1,4,3; 2,1,3,4。

进一个之后再出的情况有 5 种:1,4,3,2; 1,3,2,4; 1,3,4,2; 1, 2,3,4; 1,2,4,3。

【例 3.3】 在执行操作序列 push(1)、push(2)、pop、push(5)、push(7)、pop、push(6)之后栈顶元素和栈底元素分别是什么?

解:操作序列的执行过程如图 3.6 所示。

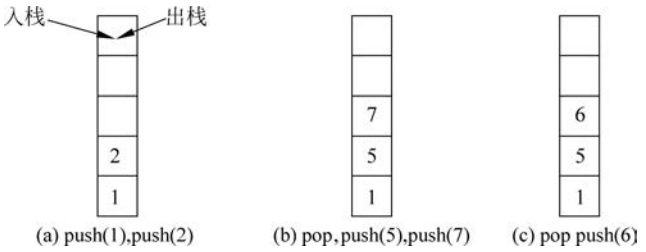

图 3.6 操作序列的执行过程

所以栈顶元素为 6、栈底元素为 1。

【例 3.4】 编程实现汉诺塔问题的求解。假设有 3 个分别命名为 x、y、z 的塔座,在塔座 x 上插有 n 个直径和序号均为 1、2、…、n 的圆盘。要求将塔座 x 上的 n 个圆盘借助塔座 y 移动到塔座 z 上,仍按照相同的序号排列,并且每次只能移动一个圆盘,任何时候都不能将一个较大的圆盘压在较小的圆盘之上。

解:分析问题可知,当 $n=1$ 时只要将序号为 1 的圆盘从 x 直接移动到 z 即可;当 $n>1$ 时则需要将序号小于 n 的 $n-1$ 个圆盘移动到 y 上,再将序号为 n 的圆盘移动到 z 上,然后将 y 上的 $n-1$ 个圆盘移动到 z 上。如何将 $n-1$ 个圆盘移动到 z 上是一个和原问题相似的问题,只是规模变小,可以用同样的方法求解。代码如下:

```
public static void hanoi(int n,char x,char y,char z){
        if(n == 1)
```

```
            move(1,x,z);
        else{
            hanoi(n-1,x,z,y);              //将 x 上序号为 1 至 n-1 的圆盘从 x 移动到 y
            move(n,x,z);                   //将序号为 n 的圆盘从塔座 x 移动到塔座 z
            hanoi(n-1,y,x,z);              //将 y 上序号为 1 至 n-1 的圆盘从 y 移动到 z
        }
    }
    //将序号为 n 的圆盘从塔座 x 移动到塔座 z
    public static void move(int n,char x,char z){
        System.out.println("将圆盘:" + n + "从塔座" + x + "移动到塔座" + z + " ");
    }
    public static void main(String[]args){
        hanoi(3,'x','y','z');
                    }
```

3.2 队　　列

3.2.1 队列的基本概念

队列是一种特殊的线性表，其插入操作只能在表的尾部进行，删除操作只能在表头进行。在队列中允许进行插入操作的一端称为队尾，允许进行删除操作的另一端称为队首。在队列$\{a_0,a_1,\cdots,a_{n-1}\}$中$a_0$称为队首元素，$a_{n-1}$称为队尾元素。通常，队列的插入操作叫入队，队列的删除操作叫出队。没有数据元素的队列称为空队列。

由于插入和删除操作分别在队尾和队首进行，最先入队的元素总是最先出队，因此队列具有先进先出的特点。

3.2.2 队列的抽象数据类型描述

队列中的数据元素和数据间的逻辑关系与线性表相同，是由 n 个具有相同数据类型的数据元素构成的有限序列，队列的抽象数据类型的 Java 描述如下：

```
1  package ch03;
2  public interface IQueue {
3  public void clear();                                  //将队列置空
4  public boolean isEmpty();                             //判断队列是否为空
5  public int length();                                  //返回队列的数据元素个数
6  public Object peek();                                 //返回队首元素
7  public void offer(Object x) throws Exception;         //将数据元素 x 插入到队列成为队尾元素
8  public Object poll();                                 //将队首元素删除并返回其值
9  public void display();                                //输出队列中的所有数据元素
10 }
```

队列的抽象数据 Java 接口包含了队列的主要基本操作，如果要使用这个接口，还需要具体的类来实现。队列的 Java 接口的实现方法主要有以下两种。

(1) 基于顺序存储的实现，为顺序队列。

(2) 基于链式存储的实现，为链队列。

3.2.3 顺序队列

1. 顺序队列类的描述及实现

顺序队列的存储结构与顺序栈类似,可用数组实现,因为入队和出队操作分别在队尾和队首进行,所以增加变量 front 来指示队首元素的位置,rear 指示队尾元素的下一个存储单元的位置。顺序队列进行入队操作的状态变化如图 3.7 所示,进行出队操作后的状态变化如图 3.8 所示。

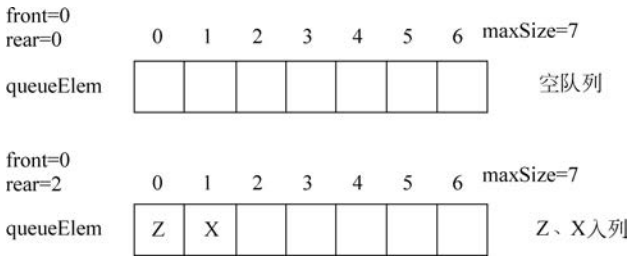

图 3.7 入队操作

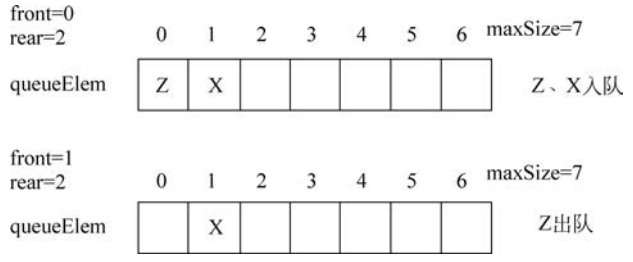

图 3.8 出队操作

顺序队列类的 Java 语言描述如下:

```
1  package ch03;
2  public class SqQueue implements IQueue {
3  private Object[] queueElem;                    //队列的存储空间
4  private int front;                             //指向队首元素
5  private int rear;                              //指向队尾元素的下一个存储单元
6  private int maxSize;                           //队列的最大存储单元个数
7
8  //构造最大存储单元个数为 maxSize 的空队列
9  public SqQueue(int maxSize){
10     front = rear = 0;
11     queueElem = new Object[maxSize];
12     this.maxSize = maxSize;
13 }
14 //将队列置空
15 public void clear() {
16     front = rear = 0;
17 }
18     //判断队列是否为空
```

```
19 public boolean isEmpty() {
20     return rear == front;
21 }

22 //返回队列的长度
23 public int length() {
24     return rear - front;
25 }

26 //读取队首元素并返回其值
27 public Object peek() {
28     if(isEmpty())
29         return null;
30     return queueElem[front];
31 }
32 //入队
33 public void offer(Object x) throws Exception {
34     if(rear == maxSize)
35         throw new Exception("队列已满");
36     queueElem[rear] = x;
37     rear++;
38 }
39 //出队
40 public Object poll() {
41     if(rear == front)
42         return null;
43     Object p = queueElem[front];
44     front++;
45     return p;
46 }
47 //输出队列中的所有数据元素
48 public void display() {
49     if(!isEmpty()){
50         for(int i = front;i < rear;i++){
51             System.out.print(queueElem[i] + " ");
52         }
53     }
54     else{
55         System.out.print("此队列为空");
56     }
57 }
58 }
```

2. 循环顺序队列类的描述及实现

分析发现，顺序队列的多次入队和出队操作会造成有存储空间却不能进行入队操作的“假溢出”现象，如图 3.9 所示。顺序队列之所以会出现“假溢出”现象是因为顺序队列的存储单元没有重复使用机制，为了解决顺序队列因数组下标越界而引起的“溢出”问题，可将顺序序列的首尾相连，形成循环顺序队列。循环顺序队列进行入队和出队操作后的状态变化如图 3.10 所示。

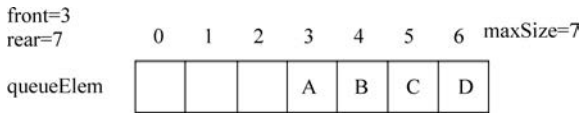

图 3.9 “假溢出”现象

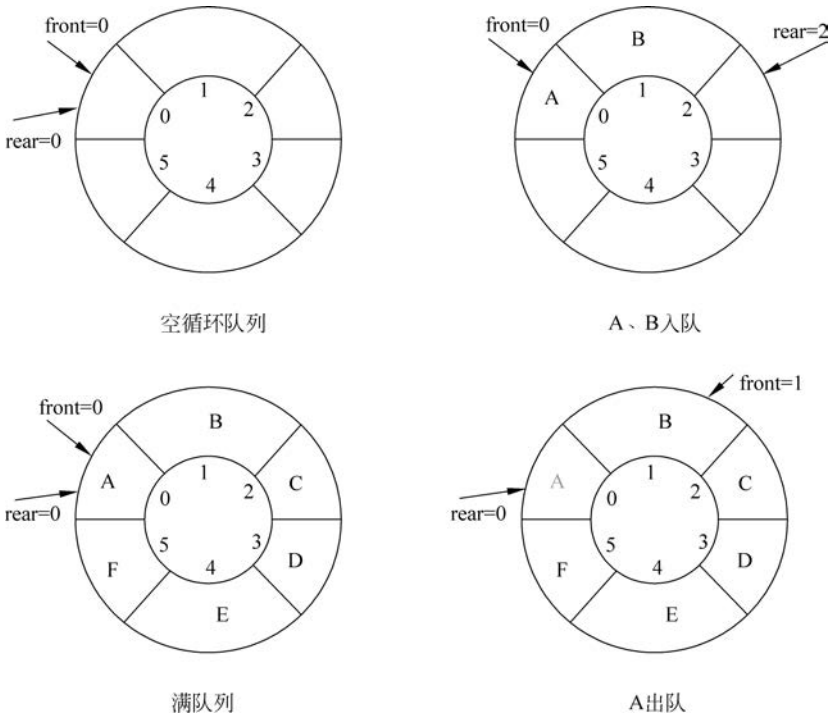

图 3.10 循环顺序队列入队和出队操作

有新的问题产生——队空和队满的判定条件都变为 front==rear,为了解决这一问题,可少利用一个存储单元,队列最多存放 maxSize-1 个数据元素,队空的判定条件为 front==rear,队满的判定条件为 front=(rear+1)%maxSize。

循环顺序队列类和顺序队列类的 Java 语言描述相似,仅是指示变量 front 和 rear 的修改以及队满的判定条件发生了变化。

循环顺序队列的 Java 语言描述如下:

```
1  package ch03;
2  public class CircleSqQueue {
3  private Object[] queueElem;                    //队列的存储空间
4  private int front;                             //指向队首元素
5  private int rear;                              //指向队尾元素的下一个存储单元
6  private int maxSize;                           //队列的最大存储单元个数
7
8  //构造最大存储单元个数为 maxSize 的空队列
9  public CircleSqQueue(int maxSize){
10     front = rear = 0;
11     queueElem = new Object[maxSize];
```

```
12      this.maxSize = maxSize;
13  }
14  //将队列置空
15  public void clear() {
16      front = rear = 0;
17  }
18  //判断队列是否为空
19  public boolean isEmpty() {
20      return rear == front;
21  }

22  //返回队列的长度
23  public int length() {
24      return (rear - front + maxSize) % maxSize;
25  }

26  //读取队首元素并返回其值
27  public Object peek() {
28      if(isEmpty())
29          return null;
30      return queueElem[front];
31  }
32  //入队
33  public void offer(Object x) throws Exception {
34      if((rear + 1) % maxSize == front)
35          throw new Exception("队列已满");
36      queueElem[rear] = x;
37      rear = (rear + 1) % maxSize;
38  }
39  //出队
40  public Object poll() {
41      if(rear == front)
42          return null;
43      Object p = queueElem[front];
44      front = (front + 1) % maxSize;
45      return p;
46  }
47  //输出队列中的所有数据元素
48  public void display() {
49      if(!isEmpty()){
50          for(int i = front;i < rear;i = (i + 1) % maxSize){
51              System.out.print(queueElem[i] + " ");
52          }
53      }
54      else{
55          System.out.print("此队列为空");
56      }
57  }
58  }
```

【例 3.5】 假定用于循环顺序存储一个队列的数组的长度为 N，队首和队尾指针分别

为 front 和 rear,写出求此队列长度(即所含元素个数)的公式。

解:当 rear 大于等于 front 时队列长度为 rear−front,也可以表示为(rear−front+N)%N;当 rear 小于 front 时队列被分成两个部分,前部分在数组尾部,其元素个数为 $N-1-$front,后部分在数组首部,其元素个数为 rear+1,两者相加为 rear−front+N。综上所述,在任何情况下队列长度的计算公式都为(rear−front+N)%N。

【例 3.6】 在执行操作序列 EnQueue(1)、EnQueue(3)、DeQueue、EnQueue(5)、EnQueue(7)、DeQueue、EnQueue(9)之后队首元素和队尾元素分别是什么? EnQueue(k)表示整数 k 入队,DeQueue 表示队首元素出队。

解:上述操作的执行过程如图 3.11 所示。

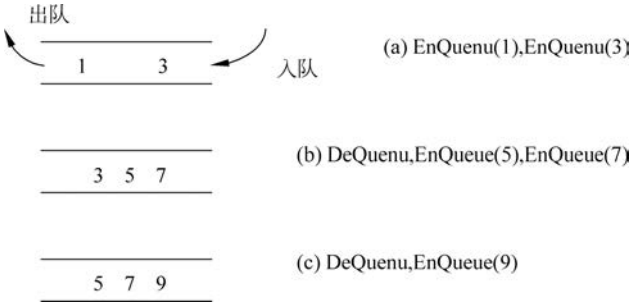

图 3.11 操作的执行过程

所以队首元素为 5,队尾元素为 9。

3.2.4 链队列

链队列用单链表实现,由于入队和出队分别在队列的队尾和队首进行,不存在在队列的任意位置进行插入和删除的情况,所以不需要设置头结点,只需要将指针 front 和 rear 分别指向队首结点和队尾结点,每个结点的指针域指向其后继结点即可。

图 3.12 所示为链队列进行入队操作后的状态变化。

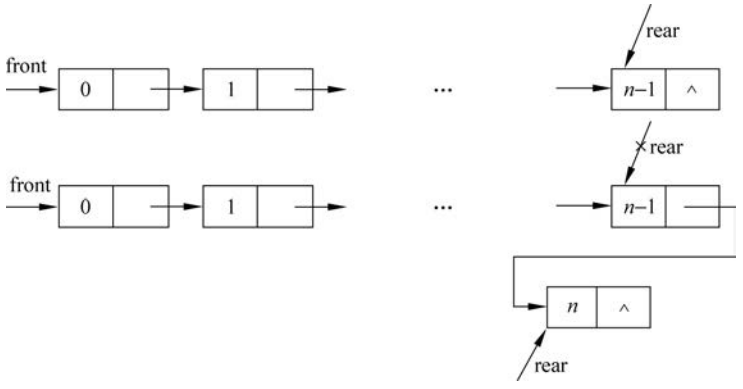

图 3.12 入队操作

图 3.13 所示为链队列进行出队操作后的状态变化。

利用 Node 类,链队列的 Java 语言描述如下:

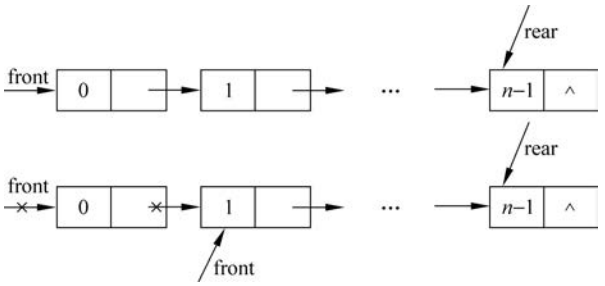

图 3.13　出队操作

```
1  package ch03;
2  import ch02.Node;
3  public class LinkQueue implements IQueue {
4  private Node front;                                    //队首指针
5  private Node rear;                                     //队尾指针
6
7  //构造空队列
8  public LinkQueue(){
9      front = rear = null;
10 }
11 //将队列置空
12 public void clear() {
13     front = rear = null;
14 }
15 //判断队列是否为空
16 public boolean isEmpty() {
17     return front == null;
18 }

19 //返回队列的长度
20 public int length() {
21     Node p = front;
22     int length = 0;
23     while(p!= null){
24         p = p.next;
25         length++;
26     }
27     return length;
28 }

29 //读取队首元素并返回其值
30 public Object peek() {
31     if(isEmpty())
32         return null;
33     return front.data;
34 }
35 //入队
```

```
36 public void offer(Object x) throws Exception {
37     Node s = new Node(x);
38     if(!isEmpty()){                              //如果队列非空
39         rear.next = s;
40         rear = s;
41     }
42     else{
43         front = rear = s;
44     }
45 }
46 //出队
47 public Object poll() {
48     if((front == null))
49         return null;
50     Node p = front;
51     front = front.next;
52     if(p == rear)                                //删除结点为队尾结点时需要修改 rear
53         rear = null;
54     return p.data;
55 }
56 //输出队列中的所有数据元素
57 public void display() {
58     if(!isEmpty()){
59         for(Node p = front;p!= null;p = p.next){
60             System.out.print(p.data + " ");
61         }
62     }
63     else{
64         System.out.print("此队列为空");
65     }
66 }
67 }
```

3.2.5 优先级队列

有些应用中的排队等待问题仅按照“先来先服务”原则不能满足要求,还需要将任务的重要程度作为排队的依据。例如操作系统中的进程调度管理,每个进程都有一个优先级值表示进程的紧急程度,优先级高的进程先执行,同级进程按照先进先出原则排队等待,因此操作系统需要使用优先级队列来管理和调度进程。又例如打印机的输出任务队列,对于先后到达的打印几百页和几页的任务将需要打印的页数较少的任务先完成,这样使得任务的总的等待时间最小。

优先级队列是在普通队列的基础之上将队列中的数据元素按照关键字的值进行有序排列。优先级队列在队首进行删除操作,但为了保证队列的优先级顺序,插入操作不一定在队尾进行,而是按照优先级插入到队列的合适位置。

和其他数据结构类似,优先级队列也可以采用顺序和链式两种存储结构。但为了快速地访问优先级高的元素以及快速地插入数据元素,通常使用链式存储结构。

1. 优先级队列结点类的描述

```
1  package ch03;
2  public class PriorityNode {
3  public Object data;                                    //结点的数据域
4  public int priority;                                   //结点的优先级
5  public PriorityNode(Object x,int priority){
6      this.data = x;
7      this.priority = priority;
8  }
9  }
```

2. 优先级队列类的描述及实现

```
1  package ch03;
2  public class PriorityQueue implements IQueue {
3  private PriorityNode front;
4  private PriorityNode rear;
5
6  public PriorityQueue(){
7      front = rear = null;
8  }
9  public void clear() {
10     front = rear = null;
11 }
12 //判断队列是否为空
13 public boolean isEmpty() {
14     return front == null;
15 }
16 //返回队列的长度
17 public int length() {
18     PriorityNode p = front;
19     int length = 0;
20     while(p!= null){
21         p = p.next;
22         length++;
23     }
24     return length;
25 }

26 //读取队首元素并返回其值
27 public Object peek() {
28     if(isEmpty())
29         return null;
30     return front.data;
31 }
32 //入队
33 public void offer(Object x,int priority) throws Exception {
34     PriorityNode s = new PriorityNode(x,priority);
```

```
35      if(!isEmpty()){                                    //如果队列非空
36          PriorityNode p = front;
37          PriorityNode q = front;
38          while(p!= null&&p.priority <= s.priority){ //按优先级寻找元素所在的位置
39              q = p;
40              p = p.next;
41          }
42          //元素位置的 3 种情况
43          if(p == null){                                 //队尾
44              rear.next = s;
45              rear = s;
46          }
47          else if(p == front){                           //队首
48              s.next = front;
49              front = s;
50          }
51          else{                                          //队中
52              q.next = s;
53              s.next = p;
54          }
55      }
56      else{                                              //队列为空
57          front = rear = s;
58      }
59  }
60  //出队
61  public Object poll() {
62      if((front == null))
63          return null;
64      PriorityNode p = front;
65      front = front.next;
66      if(p == rear)                                      //删除结点为队尾结点时需要修改 rear
67          rear = null;
68      return p.data;
69  }
70  //输出队列中的所有数据元素
71  public void display() {
72      if(!isEmpty()){
73          for(PriorityNode p = front;p!= null;p = p.next){
74              System.out.print(p.data + " " + p.priority + " ");
75          }
76      }
77      else{
78          System.out.print("此队列为空");
79      }
80  }
81  }
```

注意：在此优先队级列中，数据元素的优先级别依据优先数的大小进行判定，即优先数

越小优先级别越高。

3. 优先级队列类的应用

【例 3.7】 利用优先级队列模仿操作系统的进程管理问题,要求优先级高的进程先获得 CPU,优先级相同的进程先到的先获得 CPU。假设操作系统中的进程由进程号和进程优先级两部分组成。

解:

```
public static void main(String []args) throws Exception{
        PriorityQueue p = new PriorityQueue();
        p.offer(1,20);
        p.offer(2,30);
        p.offer(3,10);
        p.offer(4,50);
        System.out.println("进程服务的顺序为:");
        while(!p.isEmpty()){
            System.out.println(p.poll().toString());

        }
```

3.3 栈和队列的比较

栈和队列的比较如表 3.1 所示。

表 3.1 栈和队列的比较

相同点	(1) 均为线性结构,数据元素间具有“一对一”的逻辑关系。 (2) 都有顺序存储和链式存储两种实现方式。 (3) 操作受限,插入操作均在表尾进行(优先级队列除外)。 (4) 插入与删除操作都具有常数时间
不同点	(1) 栈删除操作在表尾进行,具有后进先出特性;队列的删除操作在表头进行,具有先进先出特性。 (2) 顺序栈可以实现多栈空间共享,而顺序队列则不同

3.4 实　　验

3.4.1 汉诺塔

相传在古印度圣庙中,有一种被称为汉诺塔(Hanoi)的游戏。该游戏是在一块铜板装置上放三根柱(编号 A、B、C),在 A 柱自下而上、由大到小按顺序放置 64 个金盘。游戏目标:把 A 柱上的金盘全部移到 C 柱上,并仍保持原有顺序叠好。操作规则:每次只能移动一个盘子,并且在移动过程中三根柱上都始终保持大盘在下、小盘在上,操作过程中盘子可以置于 A、B、C 任一柱上。

编程要求:

有 A、B、C 三根柱子,A 为起始柱,B 为辅助柱,C 为目标柱,以及若干圆盘。圆盘按上

小下大的顺序堆放在 A 柱上,从上到下依次为每一个圆盘标号为 1～n。输入圆盘数,要求将 A 柱的圆盘移动到 C 柱,一次只能移动一个,过程中可以任意使用 3 根柱子,但是要保证小圆盘始终在上。输出每个圆盘每一步移动的过程,如 1 号圆盘从 A→C。

输入:输入 n,1≤n≤20,表示黄金圆盘的个数。

输出:输出需要移动的圆盘和移动的具体方案,参见示例。

例如,1 A→C 表示将 1 号圆盘从 A 柱移到 C 柱。

示例:

输入:2

输出:1 A→B
2 A→C
1 B→C

```
import java.util.Scanner;

public class U3H1 {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("请输入圆盘的数量");
        int num = in.nextInt();
        hanoi(num, 'A', 'B', 'C');                    //起始柱、辅助柱、目标柱默认为 A、B、C
    }

    //汉诺塔问题实现
    //a 存放起始柱,b 存放辅助柱,c 存放目标柱
    public static void hanoi(int num, char a, char b, char c){
        if (num == 1) {
            System.out.println(num + " " + a + " -> " + c);
        }else{
            hanoi(num - 1, a, c, b);          //借助 c 把第 num 个以外的圆盘从 a 移动到 b
            System.out.println(num + " " + a + " -> " + c); //把第 num 个圆盘从 a 移动到 c
            hanoi(num - 1, b, a, c);          //借助 a 把第 num 个以外的圆盘从 b 移动到 c
        }
    }

}
```

3.4.2 吃巧克力

小 Q 的父母要出差 N 天,走之前给小 Q 留下了 M 块巧克力。小 Q 决定每天吃的巧克力数量不少于前一天吃的一半,但是他又不想在父母回来之前的某一天没有巧克力吃,请问他第一天最多能吃多少块巧克力?

```
import java.util.Scanner;

public class U3H2 {
    public static void main(String[] args) {
```

```
        System.out.println("请输入巧克力个数 M:");
        Scanner sc1 = new Scanner(System.in);
        int M = sc1.nextInt();
        System.out.println("请输入出差天数 N:");
        Scanner sc2 = new Scanner(System.in);
        int N = sc2.nextInt();
        //第 i 天吃 a[i]个巧克力
        //为了保证最后一天有至少巧克力吃
        if (M - Eat(N) > 0) {
            System.out.println("第一天最多吃" + (M - Eat(N)) + "巧克力");
        } else {
            System.out.println("巧克力不足,请重新输入");
        }

    }

    private static int Eat(int s) {
        int sum = 1;
        for (int i = 0; i < s - 2; i++) {
            sum += 2 * sum;
        }
        return sum;

    }

}
```

3.4.3 表达式求值


视频讲解

传入一个由整数、运算符和符合语法的括号组成的表达式,输出运算结果。

示例:

输入:(((10 * (6 / (9 + 3) * 11)) + 17) + 5)/9

输出:8.555555555555555

提示:利用栈将表达式转换为逆波兰表达式(后缀表达式)后进行运算。

```
import java.util.Collections;
import java.util.Stack;

public class U3H3 {
    private Stack<String> postfixStack = new Stack<>();      //后缀式栈
    private Stack<Character> opStack = new Stack<>();        //运算符栈
    private int[] operatPriority = new int[] {0,3,2,1,-1,1,0,2};      //运用运算符 ASCII
码 - 40 做索引的运算符优先级
    public static void main(String[] args) {
        U3H3 cal = new U3H3();
        String s = "5 + 12 * (3 + 5)/7";
        double result = cal.calculate(s);
        System.out.println(result);
    }
```

```
    /**
     * 按照给定的表达式计算
     * @param expression 要计算的表达式,如 5 + 12 * (3 + 5)/7
     * @return
     */
    public double calculate(String expression) {
        Stack<String> resultStack = new Stack<String>();
        prepare(expression);
        Collections.reverse(postfixStack);                    //将后缀式栈反转
        String firstValue ,secondValue,currentValue;          //参与计算的第一个值、第二个
值和算术运算符
        while(!postfixStack.isEmpty()) {
            currentValue = postfixStack.pop();
            if(!isOperator(currentValue.charAt(0))) {         //如果不是运算符则存入操作
数栈中
                resultStack.push(currentValue);
            } else {            //如果是运算符则从操作数栈中取两个值和该数值一起参与运算
                secondValue = resultStack.pop();
                firstValue = resultStack.pop();
                String tempResult = calculate(firstValue, secondValue, currentValue.charAt
(0));
                resultStack.push(tempResult);
            }
        }
        return Double.valueOf(resultStack.pop());
    }

    /**
     * 数据准备阶段将表达式转换为后缀式栈
     * @param expression
     */
    private void prepare(String expression) {
        opStack.push(',');                        //运算符放入栈底元素逗号,此符号优先级最低
        char[] arr = expression.toCharArray();
        int currentIndex = 0;                                 //当前字符的位置
        int count = 0;                           //上次算术运算符到本次算术运算符的字符长度
        char currentOp ,peekOp;                               //当前操作符和栈顶操作符
        for(int i = 0;i<arr.length;i++) {
            currentOp = arr[i];
            if(isOperator(currentOp)) {                       //如果当前字符是运算符
                if(count > 0) {
                    postfixStack.push(new String(arr,currentIndex,count));     //取两个运
算符之间的数字
                }
                peekOp = opStack.peek();
                if(currentOp == ')') {//遇到反括号则将运算符栈中的元素移除到后缀式栈中,
直到遇到左括号
                    while(opStack.peek() != '(') {
                        postfixStack.push(String.valueOf(opStack.pop()));
                    }
                    opStack.pop();
```

```
                } else {
                    while(currentOp != '(' && peekOp != ',' && compare(currentOp,peekOp) ) {
                        postfixStack.push(String.valueOf(opStack.pop()));
                        peekOp = opStack.peek();
                    }
                    opStack.push(currentOp);
                }
                count = 0;
                currentIndex = i + 1;
            } else {
                count++;
            }
        }
        if(count > 1 || (count == 1 && !isOperator(arr[currentIndex]))) {//最后一个字符不是
括号或者其他运算符则加入后缀式栈中
            postfixStack.push(new String(arr,currentIndex,count));
        }

        while(opStack.peek() != ',') {
            postfixStack.push(String.valueOf( opStack.pop())); //将操作符栈中的剩余元素添
加到后缀式栈中
        }
    }

    /**
     * 判断是否为算术符号
     * @param c
     * @return
     */
    private boolean isOperator(char c) {
        return c == '+' || c == '-' || c == '*' || c == '/' || c == '(' ||c == ')';
    }

    /**
     * 利用 ASCII 码 - 40 做下标的算术符号优先级
     * @param cur
     * @param peek
     * @return
     */
    public boolean compare(char cur,char peek) {// 如果是 peek 优先级高于 cur,返回 true,默认
都是 peek 优先级要低
        boolean result = false;
        if(operatPriority[(peek) - 40] >= operatPriority[(cur) - 40]) {
            result = true;
        }
        return result;
    }

    /**
     * 按照给定的算术运算符做计算
     * @param firstValue
```

```
     * @param secondValue
     * @param currentOp
     * @return
     */
    private String calculate(String firstValue,String secondValue,char currentOp) {
        String result = "";
        switch(currentOp) {
            case '+':
                result = String.valueOf(ArithHelper.add(firstValue, secondValue));
                break;
            case '-':
                result = String.valueOf(ArithHelper.sub(firstValue, secondValue));
                break;
            case '*':
                result = String.valueOf(ArithHelper.mul(firstValue, secondValue));
                break;
            case '/':
                result = String.valueOf(ArithHelper.div(firstValue, secondValue));
                break;
        }
        return result;
    }
}

class ArithHelper {

    // 默认除法运算精度
    private static final int DEF_DIV_SCALE = 16;

    // 这个类不能实例化
    private ArithHelper() {
    }

    /**
     * 提供精确的加法运算
     *
     * @param v1 被加数
     * @param v2 加数
     * @return 两个参数的和
     */

    public static double add(double v1, double v2) {
        java.math.BigDecimal b1 = new java.math.BigDecimal(Double.toString(v1));
        java.math.BigDecimal b2 = new java.math.BigDecimal(Double.toString(v2));
        return b1.add(b2).doubleValue();
    }

    public static double add(String v1, String v2) {
        java.math.BigDecimal b1 = new java.math.BigDecimal(v1);
        java.math.BigDecimal b2 = new java.math.BigDecimal(v2);
        return b1.add(b2).doubleValue();
```

```
    }

    /**
     * 提供精确的减法运算
     *
     * @param v1 被减数
     * @param v2 减数
     * @return 两个参数的差
     */

    public static double sub(double v1, double v2) {
        java.math.BigDecimal b1 = new java.math.BigDecimal(Double.toString(v1));
        java.math.BigDecimal b2 = new java.math.BigDecimal(Double.toString(v2));
        return b1.subtract(b2).doubleValue();
    }

    public static double sub(String v1, String v2) {
        java.math.BigDecimal b1 = new java.math.BigDecimal(v1);
        java.math.BigDecimal b2 = new java.math.BigDecimal(v2);
        return b1.subtract(b2).doubleValue();
    }

    /**
     * 提供精确的乘法运算
     *
     * @param v1 被乘数
     * @param v2 乘数
     * @return 两个参数的积
     */

    public static double mul(double v1, double v2) {
        java.math.BigDecimal b1 = new java.math.BigDecimal(Double.toString(v1));
        java.math.BigDecimal b2 = new java.math.BigDecimal(Double.toString(v2));
        return b1.multiply(b2).doubleValue();
    }

    public static double mul(String v1, String v2) {
        java.math.BigDecimal b1 = new java.math.BigDecimal(v1);
        java.math.BigDecimal b2 = new java.math.BigDecimal(v2);
        return b1.multiply(b2).doubleValue();
    }

    /**
     * 提供(相对)精确的除法运算。当发生除不尽的情况时,精确到小数点后 10 位,以后的数字
四舍五入
     *
     * @param v1 被除数
     * @param v2 除数
     * @return 两个参数的商
     */
```

```
    public static double div(double v1, double v2) {
        return div(v1, v2, DEF_DIV_SCALE);
    }

    public static double div(String v1, String v2) {
        java.math.BigDecimal b1 = new java.math.BigDecimal(v1);
        java.math.BigDecimal b2 = new java.math.BigDecimal(v2);
        return b1.divide(b2, DEF_DIV_SCALE, java.math.BigDecimal.ROUND_HALF_UP).
doubleValue();
    }

    /**
     * 提供(相对)精确的除法运算。当发生除不尽的情况时,由 scale 参数指定精度,以后的数字
四舍五入
     *
     * @param v1 被除数
     * @param v2 除数
     * @param scale 小数点后保留几位
     * @return 两个参数的商
     */

    public static double div(double v1, double v2, int scale) {
        if (scale < 0) {
            throw new IllegalArgumentException("The scale must be a positive integer or
zero");
        }
        java.math.BigDecimal b1 = new java.math.BigDecimal(Double.toString(v1));
        java.math.BigDecimal b2 = new java.math.BigDecimal(Double.toString(v2));
        return b1.divide(b2, scale, java.math.BigDecimal.ROUND_HALF_UP).doubleValue();
    }

    /**
     * 提供精确的小数位四舍五入处理
     *
     * @param v 需要四舍五入的数字
     * @param scale 小数点后保留几位
     * @return 四舍五入后的结果
     */

    public static double round(double v, int scale) {
        if (scale < 0) {
            throw new IllegalArgumentException("The scale must be a positive integer or
zero");
        }
        java.math.BigDecimal b = new java.math.BigDecimal(Double.toString(v));
        java.math.BigDecimal one = new java.math.BigDecimal("1");
        return b.divide(one, scale, java.math.BigDecimal.ROUND_HALF_UP).doubleValue();
    }

    public static double round(String v, int scale) {
        if (scale < 0) {
```

```
            throw new IllegalArgumentException("The scale must be a positive integer or
zero");
        }
        java.math.BigDecimal b = new java.math.BigDecimal(v);
        java.math.BigDecimal one = new java.math.BigDecimal("1");
        return b.divide(one, scale, java.math.BigDecimal.ROUND_HALF_UP).doubleValue();
    }
}
```

小　结

(1) 栈是一种特殊的线性表,它只允许在栈顶进行插入和删除操作,具有后进先出的特性,各种运算的时间复杂度为$O(1)$。栈采用顺序存储结构或者链式存储结构。

(2) 队列是一种特殊的线性表,它只允许在表头进行删除操作、在表尾进行插入操作,具有先进先出的特性,各种运算的时间复杂度为$O(1)$。队列采用顺序存储结构或者链式存储结构。

(3) 循环队列是将顺序队列的首尾相连,防止“假溢出”现象的发生。

(4) 优先级队列是在普通队列的基础之上将队列中的数据元素按照关键字的值进行有序排列。在表头进行删除操作,插入操作按照优先级插入到队列的合适位置。

习　题　3

一、选择题

1. 对于栈操作数据的原则是(　　)。

A. 先进先出　　B. 后进先出　　C. 后进后出　　D. 不分顺序

2. 在做入栈运算时应先判别栈是否(　①　),在做出栈运算时应先判别栈是否(　②　)。当栈中元素为n个,做进栈运算时发生上溢,则说明该栈的最大容量为(　③　)。

为了增加内存空间的利用率和减少溢出的可能性,由两个栈共享一片连续的内存空间时应将两栈的(　④　)分别设在这片内存空间的两端,这样当(　⑤　)时才产生上溢。

①、②: A. 空　　B. 满　　C. 上溢　　D. 下溢

③: A. $n-1$　　B. n　　C. $n+1$　　D. $n/2$

④: A. 长度　　B. 深度　　C. 栈顶　　D. 栈底

⑤: A. 两个栈的栈顶同时到达栈空间的中心点

B. 其中一个栈的栈顶到达栈空间的中心点

C. 两个栈的栈顶在栈空间的某一位置相遇

D. 两个栈均不空,且一个栈的栈顶到达另一个栈的栈底

3. 一个栈的输入序列为1,2,3,…,n,若输出序列的第一个元素是n,输出的第i($1\leqslant i\leqslant n$)个元素是(　　)。

A. 不确定　　B. $n-i+1$　　C. i　　D. $n-i$

4. 若一个栈的输入序列为1,2,3,…,n,输出序列的第一个元素是i,则第j个输出元

素是(　　)。

A. $i-j-1$　　B. $i-j$　　C. $j-i+1$　　D. 不确定的

5. 若已知一个栈的入栈序列是1,2,3,…,n,其输出序列为 $p_1,p_2,p_3,\cdots,p_n$,若 p_n 是 n,则 p_i 是(　　)。

A. i　　B. $n-i$　　C. $n-i+1$　　D. 不确定

6. 有6个元素6、5、4、3、2、1顺序入栈,下列不是合法的出栈序列是(　　)。

A. 5,4,3,6,1,2　　B. 4,5,3,1,2,6　　C. 3,4,6,5,2,1　　D. 2,3,4,1,5,6

7. 设栈的输入序列是1,2,3,4,则(　　)不可能是其出栈序列。

A. 1,2,4,3　　B. 2,1,3,4　　C. 1,4,3,2　　D. 4,3,1,2

E. 3,2,1,4

8. 一个栈的输入序列为1,2,3,4,5,则下列序列中不可能是栈的输出序列的是(　　)。

A. 2,3,4,1,5　　B. 5,4,1,3,2　　C. 2,3,1,4,5　　D. 1,5,4,3,2

9. 设一个栈的输入序列是1,2,3,4,5,则下列序列中是栈的合法输出序列的是(　　)。

A. 5,1,2,3,4　　B. 4,5,1,3,2　　C. 4,3,1,2,5　　D. 3,2,1,5,4

10. 某堆栈的输入序列为a,b,c,d,下面序列中不可能是它的输出序列的是(　　)。

A. a,c,b,d　　B. b,c,d,a　　C. c,d,b,a　　D. d,c,a,b

11. 设a、b、c、d、e、f以所给的次序入栈,若在入栈操作时,允许出栈操作,则下面得不到的序列为(　　)。

A. f,e,d,c,b,a　　B. b,c,a,f,e,d　　C. d,c,e,f,b,a　　D. c,a,b,d,e,f

12. 设有3个元素X、Y、Z顺序入栈(入的过程中允许出栈),下列得不到的出栈排列是(　　)。

A. X,Y,Z　　B. Y,Z,X　　C. Z,X,Y　　D. Z,Y,X

13. 输入序列为A,B,C,变为C,B,A时经过的栈操作为(　　)。

A. push,pop,push,pop,push,pop　　B. push,push,push,pop,pop,pop

C. push,push,pop,pop,push,pop　　D. push,pop,push,push,pop,pop

14. 若一个栈以向量V[1..n]存储,初始栈顶指针top为 $n+1$,则下面 x 入栈的正确操作是(　　)。

A. top=top+1;V[top]=x　　B. top=top-1; V[top]=x

C. V[top]=x; top=top+1　　D. V[top]=x; top=top-1

15. 若栈采用顺序存储方式存储,现两栈共享空间V[1..m],top[i]代表第 i 个栈(i=1,2)的栈顶,栈1的底在V[1]、栈2的底在V[m],则栈满的条件是(　　)。

A. |top[2]-top[1]|=0　　B. top[1]+1=top[2]

C. top[1]+top[2]=m　　D. top[1]=top[2]

二、填空题

1. 向量、栈和队列都是_____结构,可以在向量的_____位置插入和删除元素;对于栈只能在_____插入和删除元素;对于队列只能在_____插入和_____删除元素。

2. 栈是一种特殊的线性表，允许插入和删除运算的一端称为_____，不允许插入和删除运算的一端称为_____。

3. _____是被限定为只能在表的一端进行插入运算、在表的另一端进行删除运算的线性表。

4. 在一个循环队列中，队尾指针指向队首元素的_____位置。

5. 在具有 n 个单元的循环队列中，队满时共有_____个元素。

6. 向栈中压入元素的操作是先_____，后_____。

7. 从循环队列中删除一个元素，其操作是先_____，后_____。

8. 顺序栈用 data[1..n]存储数据，栈顶指针是 top，则值为 x 的元素入栈的操作是_____。

9. 表达式 23+((12*3-2)/4+34*5/7)+108/9 的后缀表达式是_____。

10. 引入循环队列的目的是为了克服_____。

三、算法设计题

1. 把十进制整数转换为二至九进制数并输出。

2. 堆栈在计算机语言的编译过程中用来进行语法检查，试编写一个算法检查一个 Java 语言程序中的大括号、方括号和圆括号是否配对，若能够全部配对则返回逻辑真，否则返回逻辑假。

3. 斐波那契(Fibonacci)数列的定义为，它的第 1 项和第 2 项分别为 0 和 1，以后各项为其前两项之和。若斐波那契数列中的第 n 项用 Fib(n)表示，则计算公式如下：

$$\mathrm{Fib}(n)=\begin{cases} n-1 & (n=1 \text{ 或 } 2) \\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2) & (n>2) \end{cases}$$

试编写出计算 Fib(n)的递归算法和非递归算法。