**第二次作业**

**23122721 冯俊佳**

**填空题：**

1. 一个栈的输入序列是a 、 b 、c ，则不可能的出栈序列是\_\_\_\_\_\_\_\_\_。

分析：通过应用题2的结论或是枚举法，都可以得出不可能的出栈序列是c、b、a。

答：c、b、a

1. 用S表示入栈操作，X表示出栈操作，若元素入栈的顺序为1，2，3，4，为了得到1，3，4，2的出栈顺序，相应的S和X操作串为\_\_\_\_\_\_\_\_\_。

分析：因为需要先入栈再出栈，因此1先入栈再直接出栈；而后2入栈，3入栈，3再出栈；接着4入栈；最后4出栈后2出栈。即对应操作为SXSSXSXX

答：SXSSXSXX

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

分析：按照运算顺序依次进行转换即可。

答：23 12 3 \* 2 – 4 / 34 5 \* 7 / + + 108 9/ +

**应用题：**

1. 试用反证法证明：若借助栈可由输入序列1 , 2，…，n得到一个输出序列p1，p2，…，pn（它是输入序列的某一种排列），则在输出序列中不可能出现以下情况，即存在i<j<k，使得pj<pk<pi。

分析：假设在输出序列中，存在i<j<k，满足pj<pk<pi。pi先于pj和pk出栈，因此在pi出栈时，pj和pk必须尚未出栈。为了使pk能够出栈，pj必须先于pk入栈。为了使pj和pk出栈前不被pi阻挡，pi必须比pj和pk更早入栈。但这种假设与栈的操作性质矛盾，即如果 pj<pk<pi，则pi必须在pj和pk的下面。然而，根据假设，pi是先出栈的，这与栈的后进先出规则矛盾。故得证。

1. 若以1 , 2 , 3 , 4作为双端队列的输入序列，试分别求出以下条件的序列。

(1）能由输入受限的双端队列得到，但不能由输出受限的双端队列得到的输出序列。

(2）能由输出受限的双端队列得到，但不能由输入受限的双端队列得到的输出序列。

答：（1）4、1、3、2

（2）4、2、1、3

1. 根据教材介绍的两个顺序栈共享一个存储空间的设计，完成其中入栈、出栈和判断栈空函数的实现。

答：核心代码如下：

4.1 入栈函数的实现

1. //Push方法用于向栈中添加元素，根据t的值决定是添加到栈顶还是栈底
2. **template**<**class** ElemType>
3. Status SeqStack<ElemType>::Push(ElemType t, **const** ElemType d)
4. //t是一个标识符，用于决定是向栈顶添加（t=0）还是向栈底方向添加（t=1）
5. //d是要添加到栈中的元素
6. {
7. **if** (top1 > top2) cout << "Stack is full!" << endl;
8. // 如果t为0，表示向栈顶添加元素。
9. **if** (t == 0)
10. {
11. a[top1] = d;        // 将元素d添加到top1指向的位置
12. top1++;             // 更新top1，指向下一个空位置
13. }
14. **else** **if** (t == 1)
15. {
16. a[top2] = d;        // 将元素d添加到top2指向的位置
17. top2--;             // 更新top2，指向下一个“空”位置
18. }
19. **return** SUCCESS;
20. }

4.2 出栈函数的实现

1. // Pop方法用于从栈中弹出元素，根据t的值决定是从栈顶还是栈底（以倒序方式理解）弹出
2. **template**<**class** ElemType>
3. Status SeqStack<ElemType>::Pop(**const** ElemType& t, ElemType d)
4. //t是一个标识符，用于决定是从栈顶（t=0）还是栈底方向（t=1）弹出元素
5. //d是用于接收弹出的元素值
6. {
7. // 如果t为0，表示从栈顶弹出元素。
8. **if** (t == 0)
9. {
10. **if** (!IsEmpty1())
11. {
12. d = a[top1];    // 将栈顶元素赋值给d
13. top1--;         // 更新top1，指向新的栈顶
14. }
15. }
17. // 如果t为1，表示从栈底方向弹出元素
18. **if** (t == 1)
19. {
20. **if** (!IsEmpty2())
21. {
22. d = a[top2];
23. top2++;
24. }
25. }
26. **return** SUCCESS;
27. }

4.3 判断栈空函数的实现

1. **template**<**class** ElemType>
2. **bool** SeqStack<ElemType>::IsEmpty1() **const**
3. {
4. **if** (top1 == 0)
5. **return** **true**;
6. **return** **false**;
7. }
9. **template**<**class** ElemType>
10. **bool** SeqStack<ElemType>::IsEmpty2() **const**
11. {
12. **if** (top2 == maxSize - 1)
13. **return** **true**;
14. **return** **false**;
15. }

4.4 主函数代码

1. **int** main()
2. {
3. SeqStack<**int**> S(8);       //创建栈
4. S.Push(0, 1);
5. S.Push(0, 2);
6. S.Push(0, 3);
7. S.Push(0, 4);
8. S.Push(1, 5);
9. S.Push(1, 6);
10. S.Push(1, 7);
11. S.Push(1, 8);
12. S.Print();              //打印原先的栈
13. S.ShowTop();            //输出原先两个栈的栈顶元素
15. cout << "删去Stack1的栈顶元素后的栈为：" << endl;
16. S.Pop(0, 0);
17. S.Print();
18. cout << "删去Stack2的栈顶元素后的栈为：" << endl;
19. S.Pop(1, 0);
20. S.Print();
21. **return** 0;
22. }

4.5 测试结果

