作业 2

第 3 章：栈和队列

龚舒凯 2022202790 应用经济-数据科学实验班

2023 年 10 月 7 日

运行代码链接：
`https://github.com/GONGSHUKAI/Data_Structure/tree/main/Homework_Code/Assignment_2`

1. 设一个栈的输入序列为 $1,2,...,n$，编写一个算法，判断一个序列 $p_1,p_2,...,p_n$ 是否是一个合理的栈输出序列。

解.

通过模拟入栈的方法，判断终端输入的出栈序列`input[N]`是否合理。以一个栈输出队列 465123 为例：

1. 先找到栈输出序列的第一个元素 x，并入栈 $1,2,\ldots,x$，再将 x 出栈。例如此处先入栈 $1,2,3,4$，再出栈 4。
2. 观察此时的栈顶元素，有两种情况：
   (a) 如果当前栈顶元素 y 比此前出栈元素 x 大，则将 $x+1,\ldots,y$ 依次入栈，再出栈 y。例如此处 $6>4$，则将 $4,5,6$ 入栈，再出栈 6
   (b) 如果当前栈顶元素 y 比此前出栈元素 x 小，则：
      i. 如果当前栈顶元素是出栈序列`input[N]`里相应位置的元素，则将当前栈顶元素 y 出栈。例如此处 $5<6$，则出栈 5
      ii. 如果当前栈顶元素不是出栈序列`input[N]`里相应位置的元素，则说明栈输出序列不合法！

回到本例，出栈序列 465123 可由如下操作得到：1234 入栈，4 出栈，56 入栈，65 出栈，1 入栈，1 出栈，2 入栈，2 出栈，3 入栈，3 出栈。因此 465123 是一个合法的栈输出序列。

时间复杂度分析：设出栈序列的长度为 n。`while`循环从下标 0 遍历到下标 $n-1$，因此时间复杂度为 $O(n)$

空间复杂度分析：该算法用到一个栈作为储存容器，故空间复杂度为 $O(n)$

```
bool ValidOutputSequence(int input[N],int n, stack *MyInput){
        //i 记录出栈序列下标
        int i = 0;
        //设出栈序列第一个元素为 x，则先入栈 1,2,...,x
        for (int j = 1 ; j <= input[i] ; j++){
                StackPush(MyInput, j);
        }
        //flag 是标识变量，current 记录出栈元素
        int flag = StackTop(MyInput);
        int current = StackTop(MyInput);
        //将第一个元素 x 出栈
        StackPop(MyInput);
        i++;
        while (i < n){
                //如果当前出栈元素 input[i] 比此前的栈顶元素大
```

```
                //那么将 current+1,...,input[i] 全部入栈
                if (input[i] > flag){
                        for (int j = current + 1 ; j <= input[i] ; j++){
                                StackPush(MyInput, j);
                        }
                        flag = StackTop(MyInput);
                        current = StackTop(MyInput);
                        StackPop(MyInput);
                        i++;
                }
                //如果当前出栈元素 input[i] 比此前的栈顶元素小，那么现在应该出栈
                else{
                        //如果出栈元素不是 input[i]，说明出栈序列不合法!
                        if (StackTop(MyInput) != input[i]){
                                cout << "Invalid" << endl;
                                return false;
                        }
                        //否则将当前栈顶元素出栈，flag 记住出栈的元素
                        else{
                                flag = StackTop(MyInput);
                                StackPop(MyInput);
                                i++;
                        }
                }
        }
        cout << "Valid" << endl;
        return true;
}
```

2. 设计一个算法，借助栈判断存储在单链表中的数据是否是中心对称。例如，单链表中数据序列 $\{12, 21, 27, 21, 12\}$ 或 $\{13, 20, 38, 38, 20, 13\}$ 即为中心对称。

解.

首先获取单链表的长度 n。

1. 如果 n 为偶数，则中心对称点为 $\frac{n}{2}$ 和 $\frac{n}{2}+1$。将单链表的第 1 个到第 $\frac{n}{2}$ 个元素入栈，再逐一出栈与单链表剩余元素对比。如果出栈元素和剩余元素一一对应相等，则说明该出栈序列是中心对称的。
2. 如果 n 为奇数，则中心对称点为 $\frac{n+1}{2}$。将单链表的第 1 个到第 $\frac{n-1}{2}$ 个元素入栈，再逐一出栈与单链表第 $\frac{n+3}{2}$ 个开始的剩余元素对比。如果出栈元素和剩余元素一一对应相等，则说明该出栈序列是中心对称的。

时间复杂度分析：获取链表长度的算法时间复杂度为 $O(n)$，出栈元素与单链表剩余元素对比的算法时间复杂度为 $O(\frac{n}{2})$，故该算法的时间复杂度为 $O(n)$

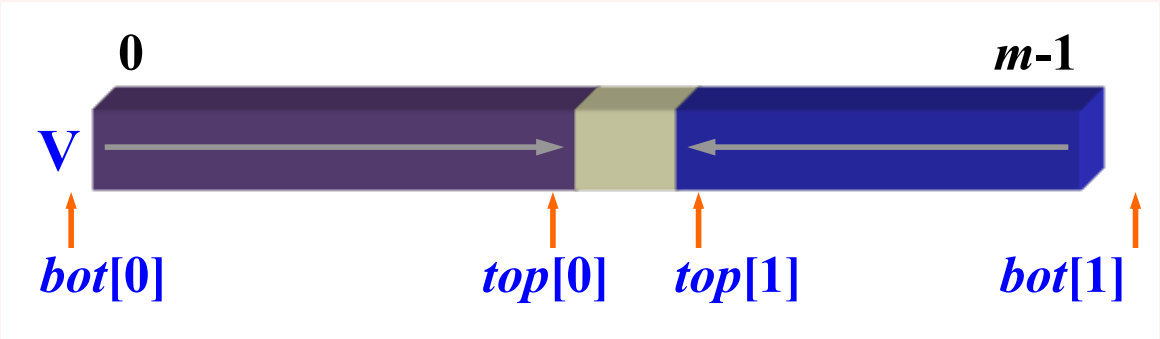
空间复杂度分析：该算法用到一个栈作为储存容器，故空间复杂度为 $O(n/2) = O(n)$

```
void CentroSymmetry(LNode *head, stack *s){
        int length = ListLength(head);//先求链表的长度
        int loop;//链表需要入栈的长度
        if (length % 2 == 0) loop = length / 2;//链表长度为奇数的情况
        else loop = (length + 1)/2;//链表长度为偶数的情况
        LNode *p = head->next;//将一半的链表数据入栈
        for (int i = 0 ; i < loop ; i++){
                StackPush(s, p->data);
                p = p->next;
        }
        if (length % 2 == 1) StackPop(s);//如果链表长度为奇数，则将栈顶元素弹出
        while (p != NULL){
                if (p->data != StackTop(s)){
                        cout << "Not Centro-symmetric!";//不中心对称
                        return;
                }
                else{
                        StackPop(s);
                        p = p->next;
                }
        }
        cout << "Centro-symmetric";//中心对称
}
```

3. 试在一个长度为 n 的数组中实现两个栈，使得二者在元素的总数目为 n 之前都不溢出，并保证 push 和 pop 操作的时间代价为 $O(1)$。请给出实现以下五个基本操作的算法：初始化双栈、是否栈空、是否栈满、进栈和出栈。

解.

双栈的工作原理和初始化机制如下图所示：



设编号为 0 和 1 的两个栈存放于一个长度为`length`的数组空间`*data`中，栈底分别处于数组的两端。第 0 号栈的栈顶指针 `top[0]`等于 -1 时该栈为空，第 1 号栈的栈顶指针`top[1]`等于`length`时该栈为空。两个栈均从两端向中间增长。双栈道初始化、判断栈空、判断栈满、进栈和出栈操作如以下代码所示。

时间复杂度分析：初始化栈 $O(1)$，判断栈空 $O(1)$，判断栈满 $O(1)$，进栈/出栈 $O(1)$
空间复杂度分析：初始化栈 $O(n)$，判断栈空 $O(1)$，判断栈满 $O(1)$，进栈/出栈 $O(1)$

```
typedef struct DblStack{
        int top[2];//两个栈顶指针
        int *data;//栈
        int length;//数组长度
}DblStack;

DblStack* InitDblStack(int n){
        DblStack *S = new DblStack;
        S->data = new int(n);
        //数组下标: 0,1,...,n-1
        S->length = n;
        S->top[0] = -1;//0 号栈初始栈顶位于-1
        S->top[1] = S->length;//1 号栈初始栈顶位于 n
        return S;
}
```

```
bool IsEmpty(DblStack *S, int i){
        //判断 i 号栈栈空与否，i 表示双号栈中的第 i 个 (i=0,1)
        if (i == 0){
                if (S->top[i] == -1) return true;
                else return false;
        }
        else{
                if (S->top[i] == S->length) return true;
                else return false;
        }
}

bool IsFull(DblStack *S){
        //判断整个数组是否已满
        if (S->top[0] + 1 == S->top[1]) return true;
        else return false;
}

void DblStackPush(DblStack *S, int value, int i){
        //向 i 号栈入栈元素 value
        if (i == 0){
                if (IsFull(S)) return;
                else S->data[++S->top[i]] = value;
        }
        else{
                if (IsFull(S)) return;
                else S->data[--S->top[i]] = value;
        }
}

void DblStackPop(DblStack *S, int i){
        //弹出 i 号栈元素
        if (i == 0){
                if (IsEmpty(S, i)) return;
                else S->top[i]--;
        }
        else{
                if (IsEmpty(S, i)) return;
                else S->top[i]++;
        }
}
```

4. 在顺序存储结构上实现输出受限的双端循环队列的入列和出列（只允许队头出列）算法。设每个元素表示一个待处理的作业，元素值表示作业的预计时间。入队列采取简化的短作业优先原则，若一个新提交的作业的预计执行时间小于队头和队尾作业的平均时间，则插入在队头，否则插入在队尾。

解.

在判断了入队列作业的插入位置后（队头/队尾），对于循环队列而言，无论是插入队头还是插入队尾都通过取队列长度的MAXQSIZE同余确定指针（数组下标）front，rear的位置：

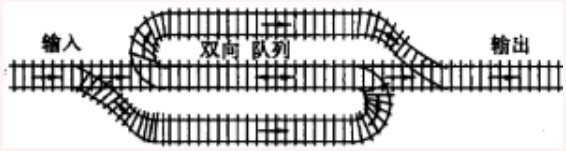
1. 由于插入队头的操作通过front-1实现，需要单独处理这样操作导致数组下标front为负的情况。
2. 插入队尾的操作与普通循环队列的入队操作无异。

时间复杂度分析：判断队满、队空、入队、出队的时间复杂度均为 $O(1)$，故该算法的时间复杂度为 $O(1)$
空间复杂度分析：判断队满、队空、入队、出队的空间复杂度均为 $O(1)$，故该算法的空间复杂度为 $O(1)$

```
void MyEnqueue(SqQueue *Q, int value){
        int ave = (Q->base[Q->front] + Q->base[Q->rear-1]) / 2;
        if (QueueFull(Q)) return;
        else{
                if (value < ave){
                        if ((Q->front - 1) % MAXQSIZE < 0){
                                Q->front = MAXQSIZE + (Q->front - 1) % MAXQSIZE;
                                Q->base[Q->front] = value;
                        }
                        else{
                                Q->front = (Q->front - 1) % MAXQSIZE;
                                Q->base[Q->front] = value;
                        }
                }
                else{
                        Q->base[Q->rear] = value;
                        Q->rear = (Q->rear + 1) % MAXQSIZE;
                }
        }
}
void DeQueue (SqQueue *Q){
        if (QueueEmpty(Q)) return;//队空，无法出列
        else{
                Q->front = (Q->front + 1) % MAXQSIZE;
        }
}
```

5. 假设在铁道转轨网的输入端有 n 节车厢：硬座、硬卧和软卧（分别以 P，H 和 S 表示）等待调度，要求这三种车厢在输出端铁道上的排列次序为：硬座在前，软卧在中，硬卧在后。试利用输出受限的双端队列对这 n 节车厢进行调度，编写算法输出调度的操作序列：分别以字符'E'和'D'表示对双端队列的头端进行入队列和出队列的操作；以字符'A'表示对双端队列的尾端进行入队列的操作。

解.



由于最后输出端铁道上的排列次序必须为：硬座在前、软卧在中、硬卧在后，且考虑到该双端队列在队头入列和出列、在队尾只能入列，因此考虑如下算法：从输入端逐一的扫入需要调度的车厢

1. 如果扫到硬座 (P)，就直接从输入端送到输出端（实现方法：先在头端入列 (输出'E')、马上再在头端出列 (输出'D')
2. 如果扫到硬卧 (H)，就从尾端入列 (输出'A')
3. 如果扫到软卧 (S)，就从头端入列 (输出'E')

等到输入端待调度车厢已经全部遍历完毕，此时硬座 (P) 已经完全输出，排在双端队列头端的是全部的软卧，排在双端队列尾端的是全部的硬卧。此时双端队列中全部元素从头端出队 (输出'D') 即完成了调度操作。

时间复杂度分析：设输入端待调度序列的长度为 n。由于需要全部扫描一遍待调度序列再进行 $O(1)$ 的入列/出列操作，故算法的时间复杂度为 $O(n)$
空间复杂度分析：该算法用到了一个最大长度为 n 的双端队列，因此算法的空间复杂度为 $O(n)$

```
void TrainArrange(SqQueue *Q){
        int n;
        cin >> n;
        char input[n];
        for (int i = 0 ; i < n ; i++){
                cin >> input[i];
        }
        for (int i = 0 ; i < n ; i++){
                //如果扫到 P，因为 P 要排在最前面，可以直接输出
                //具体操作符：头部入队列"E"，再出队列"D"
                if (input[i] == 'P') cout << "E" << " " << "D" << " ";
                //如果扫到 S，因为 S 要排中间，所以从头部入队列储存起来
```

```
                else if (input[i] == 'S'){
                        cout << "E" << " ";
                        HeadEnQueue(Q, input[i]);
                }
                //如果扫到 H，因为 H 要排最后，所以从尾部入队列储存起来
                else{
                        cout << "A" << " ";
                        TailEnQueue(Q, input[i]);
                }
        }
        //此时双端队列中车厢的排布方式必然是 H...H...S...S
        //接下来将双端队列中的车厢一个个输出出来
        while (!QueueEmpty(Q)){
                if (Q->base[Q->front] == 'S') cout << "D" << " ";
                if (Q->base[Q->front] == 'H') cout << "D" << " ";
                Q->front = (Q->front + 1) % MAXQSIZE;
        }
}
```