作业 HW2* 实验报告

姓名　　学号　　日期：2024 年 10 月 16 日

1. 涉及数据结构和相关背景

本次作业的五道题均基于**栈和队列**的相关知识进行程序设计。

一、栈

1. 栈是一种后进先出（LIFO，Last In First Out）的数据结构。数据存储在内存的连续空间中，数据的插入和删除都发生在栈顶，栈顶是可变的。
2. 栈可以使用数组或链表来实现--数组实现的栈具有固定的容量，而链表实现的栈则可以动态调整容量。
3. 栈在本次作业中主要应用于列车进栈、最长字串匹配、布尔表达式求值等问题情境。

二、队列

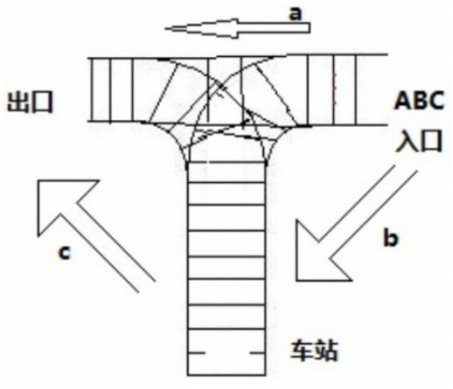
1. 队列是一种先进先出（FIFO，First In First Out）的数据结构。数据存储在内存的连续空间种，数据的插入发生在队列尾部，数据的删除发生在队列的头部。队列的头部和队列的尾部都是可变的。
2. 队列同样可以使用数组或链表来实现。但是，链表实现的队列需要一个指向队列头部的指针和一个指向队列尾部的指针。
3. 队列在本次作业中主要应用与队列的应用、队列中的最大值等问题情境。

2. 实验内容

2.1 列车进站

2.1.1 问题描述

每一时刻可以有一辆车沿着箭头 a 或 b 或 c 的方向行驶。现在有一些车在入口处等待，给出等待序列，然后再给出多组出站序列，判断多组出站序列是否能够从出口出来。



2.1.2 基本要求

输入要求：第 1 行，一个串，入站序列。后面多行，每行一个串，表示出栈序列；当输入=EOF 时结束。

输出要求：多行，若给定的出栈序列可以得到，输出 yes,否则输出 no。

2.1.3 数据结构设计

```
// Definition for a Stack.
struct Stack {
    int stacksize;          //栈当前可使用的最大容量
    SElemType    *base;     // 栈底指针
    SElemType    *top;      // 栈顶指针
}SqStack;
```

2.1.4 功能说明（函数、类）

```
//该函数判断给定的出站序列是否合法
bool isValidOutSequence(const sstring& inSequence, const string& outSequence)
{
    Stack<char>stk;
    Int i=0，j=0；
     //遍历入栈序列
    While(j<inSequence.length()) {
        压入栈，并将 j 递增以指向下一个元素；
         While（栈不空&&检查栈顶元素是否与当前出栈序列的元素匹配）{
            弹出栈；     //若匹配，则出栈，并将 i 递增以指向出栈序列的下一个元素
            i++；
           }
    }
    //如果 i 等于出栈序列长度，说明出栈序列完全匹配；否则不匹配
    return i==outSequence.length();
}
```

2.1.5 调试分析（遇到的问题和解决方法）

1.该程序整体实现较为简单，但调试过程中，输入输出的格式遇到了一些问题——比如如何一行一行读取输入。最开始我选用的是 getchar（）一个一个读取，但后来经过查找资料，选用 getline（）来读取每一行的输入，效率更高，也更不容易出错。

2.1.6 总结和体会

关于本题，其实很形象地体现了"栈"的定义以及应用场景。通过这道题，加深了我对栈的理解。这道题本身并没有很复杂，故不在此分析难点和易错点。）

2.2 布尔表达式

2.2.1 问题描述

计算类似布尔表达式（V|V)&F&(F|V)的值。其中 V 表示 True，F 表示 False，|表示 or，

&表示 and，！表示 not（运算符优先级 not> and > or）。

2.2.2 基本要求

输入：以文件形式输入，要能处理<=20 个且符号<=100 个的布尔表达式，并且忽略表达式内的空格，表达式字符数未知。

输出：有一定的格式输出要求，如--对测试用例中的每个表达式输出“Expression”，后面跟着序列号和“: ”，然后是相应的测试表达式的结果（V 或 F），每个表达式结果占一行（注意冒号后面有空格）。

2.2.3 数据结构设计

```
// Definition for a Stack.
struct Stack {
    int stacksize;          //栈当前可使用的最大容量
    SElemType    *base;     // 栈底指针
    SElemType    *top;      // 栈顶指针
}SqStack;

stack <char> Operation;      // 用于存储运算符的栈
stack <bool> Value;          // 用于存储值的栈
map <char, int> Priority;    // 用于存储运算符的优先级
```

2.2.4 功能说明（函数、类）

```
//该函数定义了逻辑运算
bool Calculation(char temp) {
    bool num_1 = 0, num_2 = 0;
        if (temp == '|') {  //或运算

            num_1 = Value.top();    // 获取栈顶元素
            Value.pop();            // 弹出栈顶元素
            num_2 = Value.top();    // 获取栈顶元素
            Value.pop();            // 弹出栈顶元素

            return (num_1 || num_2);

         //类似的进行&和！的运算，不再列出，格式和或元素类似

    return 0;
}
```

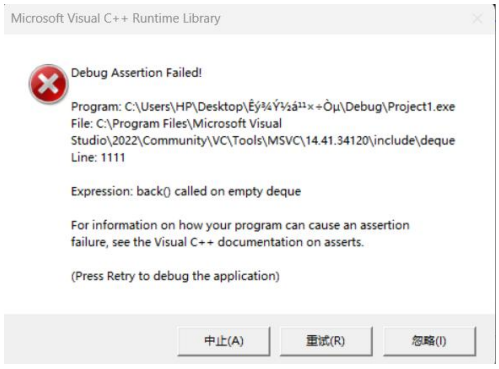
```
//该函数进行运算符入栈元素的判断及计算
```

```
void Judgement_1(char temp) {
        //判断运算符栈是否为空，空则退出函数
         if (Operation.empty() == 1) {
              Operation.push(temp);
              return;
          }
         //遇到“（”无条件入栈
          if（temp==‘(‘){
                Push;
                Return;
           }

           if(入栈的元素符优先级高于栈顶元素）{
               Push
           }
           else{
              While(1){
                 栈顶元素调用 Calcuulation 计算并弹出;
                 if (Operation.empty() == 1) {
                      Operation.push(temp);
                      break;
                  }                    //判断栈是否非空;
                  再次判断栈顶元素和入栈元素的优先级关系;
               }
           }
    return;
}
```

2.2.5 调试分析（遇到的问题和解决方法）

1. 第一次写的程序在运行时，如果文件输入的表达式个数稍微大一些，就会出现越界访问的情况。检查了很多地方仍无法解决这一问题，于是全部推翻重写，最终验证通过。



2.2.6 总结和体会

1.通过使用栈的这一数据结构，其 LIFO 的特性非常适合执行处理逻辑表达式的求值。

2.由于该题目无需判断布尔表达式的正误，于是在程序编写中可以不用编写判断布尔表达式正确与否的部分，大大降低了题目难度。

3.通过此程序的编写，我更加深刻地认识到了栈在处理需要逆序执行的问题时的优越性。

2.3 最长子串

2.3.1 问题描述

已知一个长度为 n，仅含有字符'('和')'的字符串，请计算出最长的正确的括号子串的长度及起始位置，若存在多个，取第一个的起始位置。（子串是指任意长度的连续的字符序列）

2.3.2 基本要求

输入：一行字符串。处理后输出：字串长度 起始位置（注意起始位置从 0 开始）

2.3.3 数据结构设计

继续使用栈作为本题的数据结构

```
// Definition for a Stack.
struct Stack {
    int stacksize;           //栈当前可使用的最大容量
    SElemType    *base;      // 栈底指针
    SElemType    *top;       // 栈顶指针
}SqStack;


Stack <int> stk;             // 存储索引的栈
```

2.3.4 功能说明（函数、类）

```
//该函数用于找出给定字符串中最长子串
pair<int, int> zuichangzichuan(const string& s) {
    for(int i=0;i<int(s.size());++i){
        if(s[i]==’(‘{
            压入栈;
        }
        else{        //如果是‘）’，则弹出，表示找到了一个与当前右括号匹配的左括号
           if (stk.empty()){  //栈空，则当前右括号没匹配的左括号
              当前索引 i 压入栈;
           }
           else{     //栈不空，则计算当前有效括号子串长度
             int length = i - stk.top();
```

```
            if (length > maxLength){      //如果当前长度大于已知的最长长度
              更新 maxLength 和 startIndex；//startIndex 是栈顶索引的下一个位置
            }
          }
    }
    return { maxLength, startIndex };
}
```

2.3.5 调试分析（遇到的问题和解决方法）

1.该题目比较简单，依旧重复之前两道题的知识点——栈的应用。有了前面两道题的铺垫，该题目在调试过程中并没有遇到什么大的问题。

2.但有一处需要注意，就是索引的计算。题目中要求起始位置从 0 开始，所以初始化栈时，是将一个-1 压入了栈。（最开始用的是 0，发现每次起始位置都多一个）

2.3.6 总结和体会

1.该题目依旧使用栈这一数据结构来解决问题，避免了递归处理带来的潜在问题，尤其是在处理大量数据时，比递归更加高效。（因为这个算法的时间复杂度是 O（n），每个字符只会被遍历一遍，大大降低了时间复杂度。）

2.通过该题目又一次加深了我对栈这一数据结构的理解和应用。并且通过与递归处理的对比，明确了针对某一问题选择正确的数据结构的重要性，认识到了学习数据结构的重要意义。

2.4 队列的应用

2.4.1 问题描述

输入一个 n*m 的 0 1 矩阵，1 表示该位置有东西，0 表示该位置没有东西。所有四邻域联通的 1 算作一个区域，仅在矩阵边缘联通的不算作区域。求区域数。对于所有数据，0<=n,m<=1000

2.4.2 基本要求

输入：第 1 行 2 个正整数 n，m，表示要输入的矩阵行数和列数；第 2—n+1 行为 n*m 的矩阵，每个元素的值为 0 或 1。能够处理 1000*10000 以内的矩阵.

输出：1 行，代表区域数。

2.4.3 数据结构设计

```
// Definition for a Queue.
struct QueueNode {
    int data;                //节点元素
    QueueNode*    next;      // 指向下一个节点的指针
    QueueNode(int x) : data(x), next(nullptr);
    ~QueueNode();
```

```
};
```

2.4.4 功能说明（函数、类）

```
//该函数用于寻找矩阵中为 1 的节点
void Seek(int x, int y) {
    queue<pair<int, int>> q;   //初始化一个队列
    q.enqueue((x, y))          // 将起始点 (x, y) 加入队列，并标记为已访问
    visited[x][y] = 真

    当 q 不为空 时 {                // 当队列不为空时，持续进行循环
        (cx, cy) = q.dequeue()     // 从队列中取出一个点 (cx, cy)


        对于 i 从 0 到 3 {          // 遍历四个可能的移动方向
            nx = cx + dx[i]
            ny = cy + dy[i]        // 根据当前点和方向计算新点的坐标 (nx, ny)

        if（ nx >= 0 且 nx < line 且 ny >= 0 且 ny < row 且 !visited[nx][ny] 且
region[nx][ny] == 1) {
            q.enqueue((nx, ny))    // 将新点加入队列，并标记为已访问
            visited[nx][ny] = 真
            }
        }
    }
}
```

2.4.5 调试分析（遇到的问题和解决方法）

1.经过查找相关资料，该题目有 BFS 和 DFS 两种算法去解决，其中 DFS 是基于栈对该问题进行求解。刚开始我使用栈进行求解，不太贴合题目要求，故改用 BFS 使用队列进行求解。

2.调试过程中，由于题目要求的“仅在矩阵边缘联通的不算作区域”，第一次写出来的代码没有很好地解决这一问题，导致有些矩阵求解的答案不对，后来通过检查代码，完善并增加了上、下、左、右四个边界数据的处理判断。

2.4.6 总结和体会

1.刚开始我使用栈进行求解，确实是可以做的。但栈主要是通过递归方法，当矩阵特别大的话容易引发栈溢出。而使用队列的 BFS 方法则不存在这个问题，因为它不会形成深度嵌套的调用栈。

2.因此，基于以上对栈和队列分别解决这一问题优劣的思考，进一步加深了我对栈和队列的理解，以及它们各自适合应用的问题情境。

2.5 队列中的最大值

2.5.1 问题描述

给定一个队列，有下列 3 个基本操作:

（1）Enqueue(v): v 入队

（2）Dequeue(): 使队首元素删除，并返回此元素

（3）GetMax(): 返回队列中的最大元素

请设计一种数据结构和算法，让 GetMax 操作的时间复杂度尽可能地低。

2.5.2 基本要求

运行时间不超过一秒;

输入：第 1 行 1 个正整数 n，表示队列的容量(队列中最多有 n 个元素）；接着读入多行，每一行执行一个动作---若输入"dequeue"，表示出队，当队空时，输出一行“Queue is Empty”;否则，输出出队的元素；若输入"enqueue m"，表示将元素 m 入队,当队满时(入队前队列中元素已有 n 个)，输出"Queue is Full"，否则，不输出；若输入"max",输出队列中最大元素，若队空，输出一行“Queue is Empty”。若输入"quit",结束输入，输出队列中的所有元素;

输出：多行，分别是执行每次操作后的结果。

2.5.3 数据结构设计

```
// Definition for a Queue.
struct QueueNode {
    int data;                //节点元素
    QueueNode*    next;      // 指向下一个节点的指针
    QueueNode(int x) : data(x), next(nullptr);
    ~QueueNode();
};

// Definition for a Deque.
struct DequeNode {
    int data;           // 存储的数据
    DequeNode* prev;     // 指向前一个节点的指针
    DequeNode* next;     // 指向下一个节点的指针
    DequeNode(int x = 0, DequeNode* p = nullptr, DequeNode* n = nullptr) ;
    ~DequeNode();
};
```

2.5.4 功能说明（函数、类）

```
//该函数用于返回当前队列的最大值
void Getmax() {
    if（队列 que 为空）{
        输出 "Queue is Empty"；
    }
    else  {
        Cout 队列 deq 的队首元素 ；
    }
Return;
}
```

```
//该函数用于在队列末尾添加一个新元素
void Enqueue(int value) {
    If( 队列 que 已满){
        输出 "Queue is Full";
    }
    Else{
        将 value 加入 队列 que;

        While(队列 deq 不为空 && 队列 deq 的队尾元素小于 value 时){
            移除 队列 deq 的队尾元素
        }
        将 value 加入 队列 deq;
        增加 total 的值;
     Return;
}
```

```
//该函数用于移除并返回队列的前面元素
void Dequeue() {
    If(队列 que 为空){
        输出 "Queue is Empty" ;
        Return;
     }
    设置 val 为 队列 que 的队首元素;

    If(val 等于 队列 deq 的队首元素){
        移除 队列 deq 的队首元素
     }
    输出 val;
    移除 队列 que 的队首元素;
```

```
    减少 total 的值;
    Return;
}
```

```
//该函数用于判读是否移除
void Quit() {
     While(队列 que 不为空 时){
        输出 队列 que 的队首元素;
        移除 队列 que 的队首元素;
    }
    输出 换行符 ;
    Return;
}
```

2.5.5 调试分析（遇到的问题和解决方法）

1.该题目的难点在于算法和数据结构的设计上，调试过程中没有什么特别需要解决的问题。

2.5.6 总结和体会

1.该问题明显可以直接暴力求解，遍历比较队列里每一个值找出最大值即可，但该方法的时间复杂度为 O（n），不满足题目中所说运行时间小于一秒的条件。故需要寻找另一种方法。

2.该题目的解决方法是额外使用了双向队列，对入队的数据进行“筛选”，确保进入主队列的都是递减的，最后可以直接取主队列队首的数据作为最大值。该算法时间复杂度为 O（1），大大降低了时间复杂度，但与此同时，额外队列的使用大大增加了空间复杂度。由此可以更加深刻地理解数据结构设计中“空间换时间，时间换空间”的思想。

3. 实验总结

通过此次实验作业，我更加深刻地理解了栈和队列在程序设计中的重要地位。而栈和队列作为比较相似又不同的两种线性结构，在实验过程中，我进一步深化了两种数据结构的适用场景——栈是一种后进先出（LIFO）的数据结构。它只允许在栈顶进行插入和删除操作。这种特性使得栈在处理具有递归性质的问题时非常有效；队列是一种先进先出（FIFO）的数据结构。它允许在队尾进行插入操作，在队头进行删除操作。这种特性使得队列在处理具有顺序性质的问题时非常有效。

同时经过查找相关资料，我学到了一种特别重要的设计思想——“空间换时间，时间换空间”，为我在之后降低时间复杂度提供了新思路。