作业 HW2 实验报告

姓名：郑博远 学号：2154312 日期：2022 年 9 月 28 日

1. 涉及数据结构和相关背景

本次作业涉及的数据结构为栈与队列这两种基础的线性结构。其中，第 1-3 题涉及括号匹配、列车入站、表达式计算，考察栈的相关知识；第 4 题为队列知识的应用，应用队列进行广度优先搜索；后者是栈、队列知识的综合，使用栈来实现队列，使其查询的时间复杂度接近于 O(1)。

栈是限定仅在表尾进行插入或删除操作的一种线性表。其支持在栈顶插入（入栈）、删除元素（出栈），以及初始化、判空、取栈顶元素等操作。其显著的特点为先进后出（last in first out，简称为 LIFO 结构）。

与栈相反，队列是一种先进先出（first in first out，缩写为 FIFO）的线性表。在队列中，允许插入的一端称为队尾，允许删除的一端称为队头。只允许在表的队尾进行插入元素，队头进行删除元素。即，最早进入队列的元素最先离开。

2. 实验内容

2.1 最长子串

2.1.1 问题描述

已知一个长度为 n，仅含有字符'('和')'的字符串。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，则取第一个的起始位置）。

其中，子串是指任意长度的连续的字符序列。

2.1.2 基本要求

输入：一行字符串，如题目描述。字符串的长度 n 对于 100% 的数据均有：0<=n<=100000，存在非法的右括号。

输出：字符串中最长子串的长度。若存在多个最长字串，输出第一个子串的起始位置。

2.1.3 数据结构设计

```
typedef int SElemType;   //栈的元素类型int, 用于记录左括号的下标

struct SqStack{
private:
    SElemType* base;         //栈的底部
    SElemType* top;          //栈的顶部
    int stacksize;           //栈的空间大小
};
```

2.1.4 功能说明（函数、类）

以下均为 SqStack 栈类的成员函数：

```
/*
* SqStack()
*/
SqStack();                   //空栈的建立（构造函数）

/*
* SqStack()
*/
~SqStack();                  //销毁已有的栈（析构函数）

/*
* Status ClearStack()
*/
Status ClearStack();         //把现有栈置空栈

/*
* Status Top(SElemType& e)
* @param e 取到的栈顶元素
*/
Status Top(SElemType& e);       //取栈顶元素

/*
* Status Pop(SElemType& e)
* @param e 取出的栈顶元素
*/
Status Pop(SElemType& e);       //弹出栈顶元素
```

```
/*
* Status Push(SElemType e)
* @param e 入栈的元素
*/
Status Push(SElemType e);       //新元素入栈

/*
* bool StackEmpty()
*/
bool StackEmpty();              //当前栈是否为空栈
```

2.1.5 调试分析

作为本次的第一个栈实验作业，在编写本题代码时犯了不少小错误，导致调试的时间比较长，现将主要犯的错误总结如下：

1. 起初没有构思好计算子串长度的方法，没办法很好的解决单个左括号未消除完全，以及存在非法右括号的情况。改进之后采用了将合法子串标记为 1，再在最后 O(n) 遍历整个字符串求最长子串的方法解决了问题；

2. 本题使用 C++方式的类来实现，因此在动态内存申请时，使用的是 new 进行申请空间，delete 进行释放空间，没有 C 方式的 realloc 来扩大已有的空间。在替代 C 方式的 realloc 函数时，我忘记将旧指针所指空间的内存数据 memcpy 到新指针所指的内存空间，导致之后的判断出错；

3. 初始化构造栈时，错将申请 STACK_INIT_SIZE 个 SElemType 的空间写成 STACK_INCREMENT，导致后续 push 造成越界写，程序卡死；

4. 使用 cin.getline 读入完整的字符串，然后再一个个遍历。但 for 循环的条件中使用了复杂度为 O(n)的 strlen，导致时间复杂度从 O(n)变为 $O(n^2)$，多个点 TLE 如图 1。后改为 getchar 边读入边判断解决问题。也可以考虑先用变量保存 strlen 的长度，避免每次循环遍历都执行 strlen。

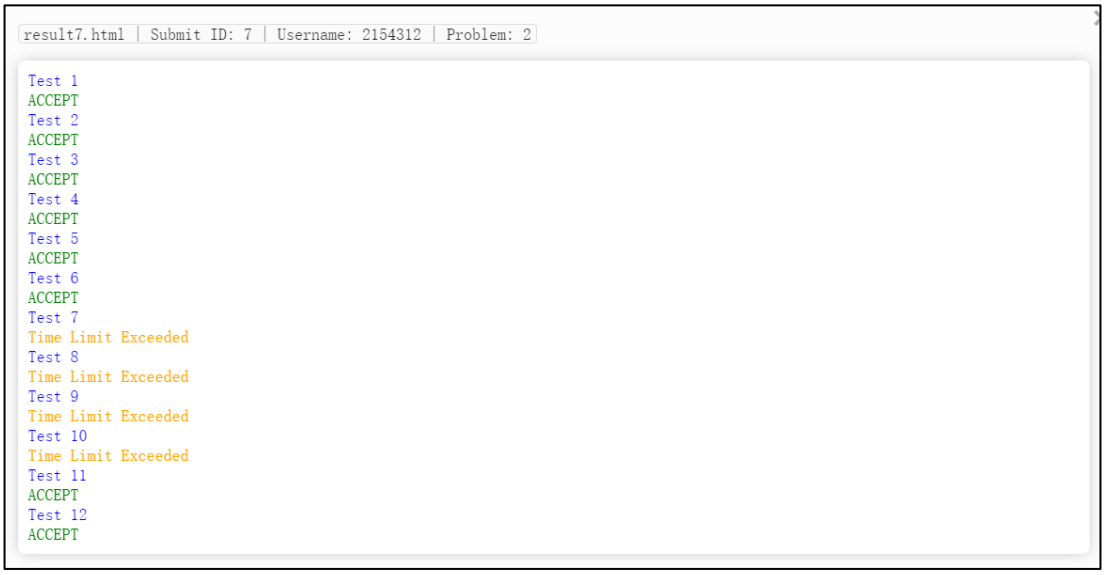

图 1　strlen 使时间复杂度变为 $O(n^2)$导致的超时

2.1.6 总结和体会

在本题中主要体会了栈的基本写法，包括建立、销毁、在栈顶插入元素（入栈）、删除栈顶元素（出栈）、取栈顶元素值等基础函数的编写。

左右括号匹配、表达式求值等是栈这一数据结构的最基础应用。此题即在括号匹配的基础上求最长的字串长度；因此题目比较简单，没有太多难点，主要考察对栈的基本掌握。与上次作业不同的是，由于有数据范围以及时间限制，加入了对时间复杂度的考量，是算法尽可能的优化。

本次调试过程中，在写栈的基础操作函数时，由于很久没有写栈导致不太熟练，遇到不少小错误耽搁了时间，希望之后在写代码时能够更加细致。在对栈的使用上则比较顺利，能较快实现期望的功能。

2.2 列车进站

2.2.1 问题描述

有一车站如图 2 所示，列车可以从入口进车站或直接从入口进入出口，再或者从车站进入出口。即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，给出该序列和多组出站序列，要求判断是否能够通过上述的方式从出口出来。

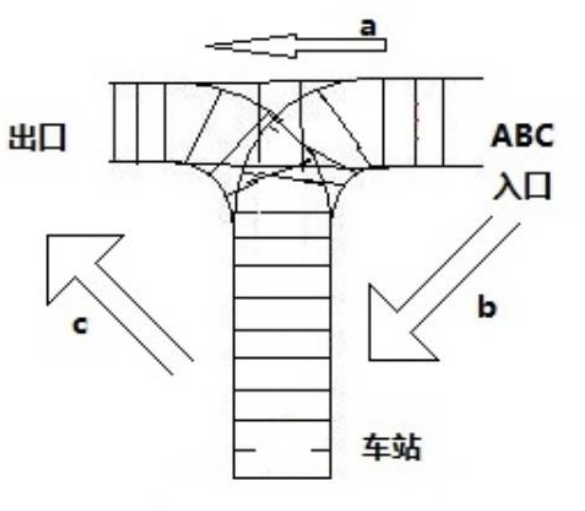

图 2　列车出入口与车站示意图

2.2.2 基本要求

输入：第一行为一个串，表示入站的序列。

后面多行，每行一个串，依次表示希望得到的出站序列。

当输入为 EOF 时结束。

输出：多行，若给定的出站序列可以得到，输出 yes；否则，输出 no。

2.2.3 数据结构设计

```
typedef char SElemType;
//栈的元素类型char，用于存储进入车站的列车编号

struct SqStack{
private:
    SElemType* base;          //栈的底部
    SElemType* top;           //栈的顶部
```

```
    int stacksize;          //栈的空间大小
};
```

2.2.4 功能说明（函数、类）

1. SqStack 栈类的成员函数:

```
/*
* SqStack()
*/
SqStack();                  //空栈的建立（构造函数）

/*
* SqStack()
*/
~SqStack();                 //销毁已有的栈（析构函数）

/*
* Status ClearStack()
*/
Status ClearStack();        //把现有栈置空栈

/*
* Status Top(SElemType& e)
* @param e 取到的栈顶元素
*/
Status Top(SElemType& e);       //取栈顶元素

/*
* Status Pop(SElemType& e)
* @param e 取出的栈顶元素
*/
Status Pop(SElemType& e);       //弹出栈顶元素

/*
* Status Push(SElemType e)
* @param e 入栈的元素
*/
Status Push(SElemType e);       //新元素入栈

/*
* bool StackEmpty()
*/
bool StackEmpty();              //当前栈是否为空栈
```

2. 其他函数：

```
/*
* int GetIndex(char* given, int len, char ch)
* @param given 被查找的字符串
* @param len 字符串的长度
* @param ch  要查找的元素
*/
int GetIndex(char* given, int len, char ch)
//获取 ch 元素在 given 字符串中的下标

/*
* void solve(char* given, int len)
* @param given 给定的出站序列字符串
* @param len 字符串的长度
*/
void solve(char* given, int len)
//针对每一个希望的出站序列，输出 yes 或 no
```

2.2.5 调试分析

1. 我思考的程序逻辑如下：若栈顶元素是下一个要出站的元素则将其弹出。否则，如果下一个要出站的元素在入站处，则将其前方的元素（可以包括自身）全部入栈；如果下一个要出站的元素在栈内（非栈顶），则输出“no”。但实际操作中，将前方元素（包括自身）入栈后，我忘记将栈顶要出站的元素弹出，导致出错。因此，在批量入栈后，也要记得弹出栈顶元素；

2. 对于给定的入站序列，使用 scanf 一次性读入；此后的每一个出站序列都用 getchar 一边读入一边处理。但 scanf 后的 '\n' 忘记处理，导致此后的 getchar 读到的出站序列错误。解决方法是，在使用 scanf 时写作 scanf(“%s\n”, given)形式，即可让字符串读入后的换行符不停留在缓冲区中；

3. 根据题目描述，若干行的出栈序列以 EOF 结束。因此我在每读入一组数据后判断 getchar 得到的是否是换行符，若不是则结束进程。但实际的测试数据

是一个换行后再跟 EOF，会导致空读一次序列，使得结果出错。处理之后，对每一个 getchar 到的字符进行判断，即可解决该问题。

2.2.6 总结和体会

与上题类似，本题涉及到的栈操作与应用也较为基础，包括了建立、销毁、在栈顶插入元素（入栈）、删除元素（出栈）、取栈顶元素值等基础操作。实际上，本题与上课时所讲的例题思路基本一致，是其在生活背景下的实例化。

本题也是对栈“先进后出”特点的经典应用。在解决该问题时，我的思路比较清晰。即，对于某个期望的出站序列，依次遍历它的每个元素。若期望出站的元素在栈顶（车站最后），则立即将其出栈；否则，若期望出站的元素在栈中则不可能实现该期望的序列；若其在入站处，则将前方所有元素入栈后，将该元素出栈。如此往复直至所有元素入栈，输出“yes”；或有元素不满足条件，输出“no”。再进行下一组出站队列的判断。

2.3 布尔表达式

2.3.1 问题描述

计算格式如下的布尔表达式：（V | V) & F & (F | V ）。

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 ）

2.3.2 基本要求

输入：文件输入，有若干（A<=20）个表达式，其中每一行为一个表达式。表达式有（N<=100）个符号，符号间可以用任意空格分开，或者没有空格。表达式的总长度，即字符的个数，是未知的。

对于 100%的数据，有 A<=20，N<=100，且表达式中包含 V、F 两个值，以

及&、|、!、(、)上述符号。

所有测试数据中都可能穿插空格。

输出：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。

2.3.3 数据结构设计

```
//栈的元素类型char
typedef char SElemType;

struct SqStack{
private:
    SElemType* base;          //栈的底部
    SElemType* top;           //栈的顶部
    int stacksize;            //栈的空间大小
};

SElemType Value, Operator;
//分别用两个栈来存储操作数与操作符
```

2.3.4 功能说明（函数、类）

1. SqStack 栈类的成员函数：

```
/*
* SqStack()
*/
SqStack();                    //空栈的建立（构造函数）

/*
* SqStack()
*/
~SqStack();                   //销毁已有的栈（析构函数）

/*
* Status ClearStack()
*/
Status ClearStack();          //把现有栈置空栈

/*
* Status Top(SElemType& e)
```

```
* @param e 取到的栈顶元素
*/
Status Top(SElemType& e);       //取栈顶元素

/*
* Status Pop(SElemType& e)
* @param e 取出的栈顶元素
*/
Status Pop(SElemType& e);       //弹出栈顶元素

/*
* Status Push(SElemType e)
* @param e 入栈的元素
*/
Status Push(SElemType e);       //新元素入栈

/*
* bool StackEmpty()
*/
bool StackEmpty();              //当前栈是否为空栈
```

2. 其他函数:

```
/*
* void Operate(SqStack& Value, SqStack& Operator)
* @param Value 存放操作数的栈
* @param Operator 存放操作符的栈
*/
void Operate(SqStack& Value, SqStack& Operator)
//弹出[操作符栈]的栈顶操作符 根据其用法取出操作数进行操作
```

2.3.5 调试分析

1. 右括号 ')' 入栈时，应该弹出所有操作符进行相应操作，直至遇到左括号停止出栈，继续向后进行入栈分析。但起初的程序在弹出所有的操作符并进行相应操作之后，忘记将左括号从存放操作符的栈中弹出，导致此后的括号匹配错误。每次匹配后将左括号 Pop 弹出即可解决;

2. 当新的操作符准备入栈时，应该依次比较存放操作符栈顶的符号。若当前符号优先级低于栈顶符号，则将其入栈; 否则，弹出栈顶符号并执行其对应操作，

直至栈顶符号优先级低于入栈符号。执行之后，忘记将此时准备入栈的操作符入栈，导致此后的答案计算错误；

3. 没有注意到一组测试数据中有多个样例，输出时要分别对各个 Expression 作单行的输出，修改之后又没有观察到输出格式中的空格，导致所有测试点 Wrong Answer。注意题目表述改正后便可通过所有测试点。

2.3.6 总结和体会

表达式的计算，是对栈这一数据结构的经典应用。本题则是针对操作数 F、V，操作符与、或、非、括号的表达式计算，在操作上比较基础。

```
void Operate(SqStack& Value, SqStack& Operator)
{
    SElemType v1, v2, o;
    Operator.Pop(o);
    switch (o) {
        case '!':
            Value.Pop(v1);
            v1 = 'F' + 'V' - v1;
            Value.Push(v1);
            break;
        case '&':
            Value.Pop(v1);
            Value.Pop(v2);
            v1 = (v1 == 'V' && v2 == 'V') ? 'V' : 'F';
            Value.Push(v1);
            break;
        case '|':
            Value.Pop(v1);
            Value.Pop(v2);
            v1 = (v1 == 'V' || v2 == 'V') ? 'V' : 'F';
            Value.Push(v1);
            break;
    }
}
```

图 3　执行栈顶操作符相应操作的函数

我对这道题的整体思路如下：分别用两个栈记录操作数与操作符。读取到操作数时直接入栈；读取到操作符时，依次比较存放操作符栈顶的符号。若当前读取到的符号优先级低于栈顶的符号，则将其入栈；否则，不断弹出栈顶符号并执行其对应操作，直至栈顶符号优先级低于入栈符号或栈为空。我将这一步骤的具

体对应操作整合成函数（如图 3 所示），使代码更简洁。需特别注意的是，所有操作数、操作符入栈之后，要依次弹出栈中所有操作符，否则表达式可能未计算完整，从而导致计算表达式的值错误。

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。此算法在细胞计数上会经常用到。

2.4.2 基本要求

输入：第 1 行为 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。

输出：一行，代表计算出的区域数量。

2.4.3 数据结构设计

```
//队列的数据元素 - 存储单元格的坐标
typedef struct{
    int x;     //横坐标
    int y;     //纵坐标
}QElemtype;

//链表的一个节点
typedef struct QNode{
    QElemtype data;  //节点存储的数据
    QNode* next;     //指向下一节点的指针
}QNode, * QueuePtr;

//用链表实现的队列
class LinkQueue {
private:
    QueuePtr front;  //队列的头部
    QueuePtr rear;   //队列的尾部
```

2.4.4 功能说明（函数、类）

1. LinkQueue 的成员函数

```
/*
* LinkQueue()
*/
LinkQueue();                    //空队列的建立（构造函数）

/*
* LinkQueue()
*/
~LinkQueue();                   //销毁已有的队列（析构函数）

/*
* Status GetHead(QElemType& e)
* @param e 取到的队首元素
*/
Status GetHead(QElemType& e);       //取队首元素

/*
* Status Dequeue(QElemType& e)
* @param e 取出的队首元素
*/
Status Dequeue(QElemType& e);       //弹出队首

/*
* Status Enqueue(QElemType e)
* @param e 入队的元素
*/
Status Enqueue(QElemType e);        //新元素入队尾

/*
* bool QueueEmpty()
*/
bool QueueEmpty();                  //当前队是否为空队列
```

2. 广度优先搜索（BFS）的函数

```
/*
* void bfs(int n, int m, bool map[1010][1010], bool
vis[1010][1010], int x, int y)
* @param n 矩阵的长
* @param m 矩阵的宽
```

```
* @param map 存放读入矩阵信息的二维数组
* @param vis 存放该点是否访问过的二维数组
* @param x 当前要入队搜索的点的横坐标
* @param y 当前要入队搜索的点的纵坐标
*/
void bfs(int n, int m, bool map[1010][1010], bool vis[1010][1010],
int x, int y)  //广度优先搜索
```

2.4.5 调试分析

1. 在程序运行时陷入死循环。原因是，忘记对遍历过的所有点的 vis 数组进行标记，导致已经入队过的点反复重新入队，队列始终不能为空，程序陷入死循环。需注意出队后的点要标记 vis[x][y] = true;

2. 计算出的区域数量与预期不符合。这是因为元素入队时忘记考虑其值是否为“1”，导致与其相连的“0”也被纳入联通块，得到错误的结果;

3. 忽略了“仅在矩形边缘联通的不算作区域”这一条件，导致计数偏多。理解题意后，在处理该条件时，开始的处理方式是将靠近边缘一周的点全部置零从而达到忽略效果，但这样会导致本可以通过边缘联通的区域被分别计算、重复计数。最终仅将非边缘点作为遍历区域第一个入队，解决了该问题。

2.4.6 总结和体会

在本题中，主要体会了队列的基本写法，包括建立、销毁、在队列尾部插入元素（入队）、删除队列头部元素（出队）、取队列头部元素值等基础函数的编写。

本题是典型的搜索题。经典的搜索方式有深度优先搜索（DFS）与广度优先搜索（BFS）。前者往往借助递归（即栈方式）实现，而后者则是队列这一数据结构的经典应用。掌握队列的基本写法后，本题通过广度优先搜索，应用队列的基础知识，完成地比较顺畅。调试过程中，在入队条件的细节判断上出了一些小错误，但仔细阅读题意后比较迅速的得到了解决。

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;

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

输入：第 1 行为 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：有若干行，分别是每次执行操作后的结果。

时间：运行时间不得超过一秒。

2.5.3 数据结构设计

本题使用记录最大值的栈来实现队列，具体解释见 2.5.6。共涉及普通栈、记录最大值的栈、记录最大值的队列三种数据结构，具体如下:

```
//栈的元素类型int 记录所存放数据的值
typedef int ElemType;
```

```
//普通栈
struct SqStack{
private:
    ElemType* base;          //栈的底部
    ElemType* top;           //栈的顶部
    int stacksize;           //栈的空间大小
};

//记录当前最大值的栈
class SqStack_WithMax {
private:
    SqStack Value, Max;      //存放元素 及 最大值的栈
};

//记录当前最大值的队列（用栈实现）
class SqQueue_WithMax {
private:
    SqStack_WithMax a, b;    //用于实现队列的两个栈
    int ElemNum = 0;         //队列的元素个数
};
```

2.5.4 功能说明（函数、类）

1. SqStack 普通栈的成员函数

```
/*
* SqStack()
*/
SqStack();                   //空栈的建立（构造函数）

/*
* SqStack()
*/
~SqStack();                  //销毁已有的栈（析构函数）

/*
* Status ClearStack()
*/
Status ClearStack();         //把现有栈置空栈

/*
* Status Top(ElemType& e)
* @param e 取到的栈顶元素
*/
Status Top(ElemType& e);        //取栈顶元素
```

```
/*
* Status Pop(ElemType& e)
* @param e 取出的栈顶元素
*/
Status Pop(ElemType& e);        //弹出栈顶元素

/*
* Status Push(ElemType e)
* @param e 入栈的元素
*/
Status Push(ElemType e);        //新元素入栈

/*
* bool StackEmpty()
*/
bool StackEmpty();              //当前栈是否为空栈
```

2. SqStack_WithMax 记录最大值的栈的成员函数

```
/*
* SqStack_WithMax()
*/
SqStack_WithMax();                  //空栈的建立（构造函数）

/*
* SqStack_WithMax()
*/
~SqStack_WithMax();                 //销毁已有的栈（析构函数）

/*
* Status Top(ElemType& e)
* @param e 取到的栈顶元素
*/
Status Top(ElemType& e);        //取栈顶元素

/*
* Status Pop(ElemType& e)
* @param e 取出的栈顶元素
*/
Status Pop(ElemType& e);        //弹出栈顶元素

/*
* Status GetMax(ElemType& e)
* @param e 取到的最大值元素
*/
```

```
Status GetMax(ElemType& e);    //取栈中元素最大值

/*
* Status Push(ElemType e)
* @param e 入栈的元素
*/
Status Push(ElemType e);       //新元素入栈

/*
* bool StackEmpty()
*/
bool StackEmpty();             //当前栈是否为空栈
```

3. SqQueue_WithMax 记录最大值的队列的成员函数

```
/*
* SqQueue_WithMax()
*/
SqQueue_WithMax();                  //空队列的建立（构造函数）

/*
* SqQueue_WithMax()
*/
~SqQueue_WithMax();                 //销毁已有的队列（析构函数）

/*
* Status GetMax(ElemType& e)
* @param e 取到的队首元素
*/
Status GetMax(ElemType& e);         //取队列元素最大值

/*
* Status Dequeue(ElemType& e)
* @param e 取出的队首元素
*/
Status Dequeue(ElemType& e);        //弹出队首

/*
* Status Enqueue(ElemType e)
* @param e 入队的元素
*/
Status Enqueue(ElemType e);    //新元素入队尾

/*
* bool QueueEmpty()
*/
```

```
bool QueueEmpty();                    //当前队是否为空队列
```

2.5.5 调试分析

1. 取队列当前最大值时，在 a、b 两个栈中分别取最大值并进行比较。但有时 a 或 b 栈为空，空读到的元素做 max 比较会得到预期外的结果，导致取到的最大值错误。后来在 GetMax 时先将元素 e 赋值为 0x80000000（即 INT_MIN），在比较时加上了栈不为空的条件判断，使得取到的最大值正确；

2. 在判断队列是否满的时候，错用 a、b 两个栈是否满作为判断条件。但若 a、b 两栈的大小均为 n，二者均满时代表队列大小为 2n 与题意不符。而两栈中的元素个数相加又没有直接的方法得到，因此在类中增设变量 ElemNum 来记录当前队列内的元素个数判断队列的空或满。

2.5.6 总结和体会

本题是对栈、队列两种数据结构的综合，有比较高的挑战性。在未经过优化的队列中，找到最大值的复杂度是 O(n)，显然无法满足题目的时间要求。

首先，考虑栈这样的数据结构是否能满足 O(1) 查找栈内最大元素的时间复杂度。显然，另开一个栈存储当前最大值，当有更大的值入栈时，最大值栈也将该元素入栈；当当前最大值出站时，最大值也进行元素的出栈便可以维护。为了方便之后队列类的使用，我构造了一个类来封装这样的结构。这样便可以调用其成员函数进行数据的操作，十分方便。

其次，考虑用栈来实现队列。这可以通过使用两个栈（记为 a 栈、b 栈）来解决（这里的栈指前文中构造的能够维护最大值 O(1)查询的特殊栈）：入队列时均往 a 栈入栈；元素出队列时，若 b 栈非空，则将其栈顶元素出栈；若 b 栈为空，则将 a 栈所有元素出栈进入 b 栈后，再将其栈顶元素出栈。这样就能用栈的

“先进后出”实现队列的“先进先出”。同时，由于前文中构造的能够 O(1)查询最大值的栈的设计，这样的队列也能支持近似于 O(1) 的查找最大值。

3. 实验总结

这次作业中，我巩固了对栈与队列这两种数据结构的了解与使用，在不同的题目中分别应用了栈与队列的相关知识来解决问题，操作还算熟练。特别地，队于第 5 题的解决，我使用了栈来实现链表，使其最大值查询时间复杂度近似 O（1）。在这几题的操练中，我对栈和队列基本函数的编写有了更为熟练的掌握。

通过这次作业，我在不同的题目之中体会了栈“先进后出”的特点以及队列“先进先出”的特点，比较了二者的异同。通过括号匹配、表达式计算以及深度优先搜索等题目，我接触了栈与队列这两种数据结构的经典应用，也更为熟练地掌握了对它们的使用，总体的编程过程比较顺畅。

总的来说，通过本次作业，我加深了对栈、队列这两种特殊的线性表的理解，也复习了各种基础操作。我希望通过这次作业的练习在未来能更加熟练掌握使用已学过的这些数据结构。