作业 HW2* 实验报告

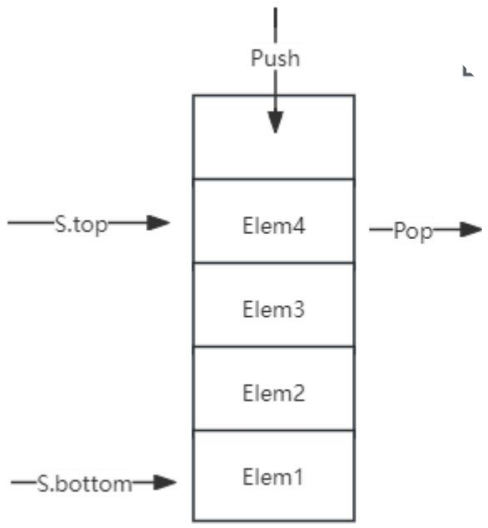
姓名：朱俊泽 学号：2351114 日期：2024 年 10 月 12 日

实验报告格式要求按照模板（使用 Markdown 等也请保证报告内包含模板中的要素）
对字体大小、缩进、颜色等不做强制要求（但尽量代码部分和文字内容有一定区分，可参考 vscode 配色）
实验报告要求在文字简洁的同时将内容表示清楚
报告内不要大段贴代码，尽量控制在 20 页以内

1. 涉及数据结构和相关背景

题目或实验涉及数据结构的相关背景
栈 队列的基本应用
① 栈特点：线性序列，先进后出。元素只能从栈顶进出。
② 队列特点：线性序列，先进先出，队尾只能进，队首只能出。存在特殊的双端队列，两端均可进出。

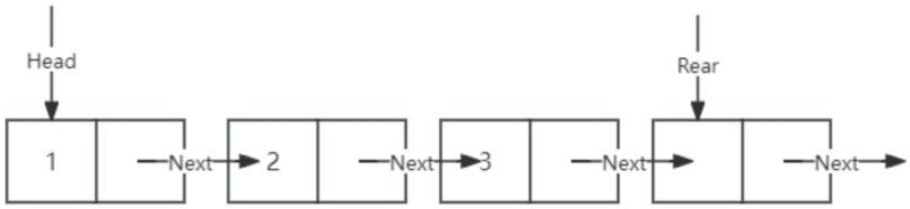
栈：示意图
1. push，往栈顶放入一个元素
2. Top 取出栈顶元素值，但是不消除该值
3. Pop 取出栈顶元素删除



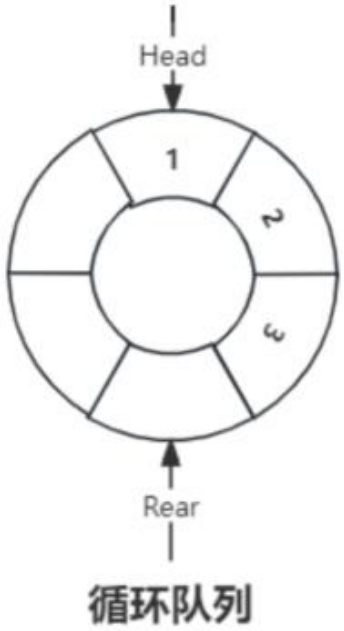
栈

队列：示意图
可以理解和链表有些相似的地方，至少部分代码可以直接借鉴。
要实现
1. Front 取出头元素
2. Back 取出尾元素

3. Push_front 从队头插入
4. Push_back 从队尾插入
5. Pop_back 删除队尾
6. Pop_front 删除队头



特殊*：循环队列

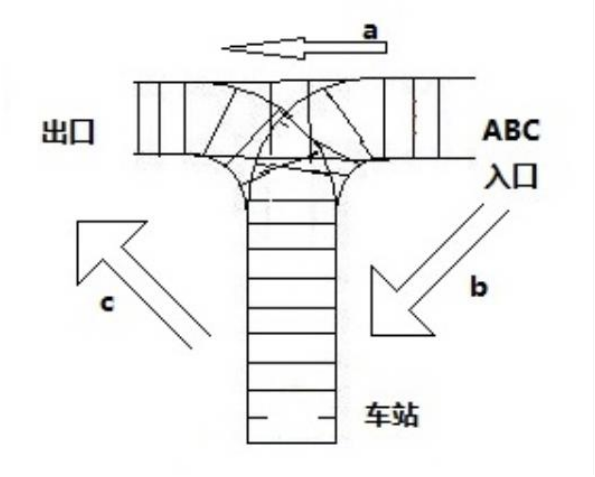


2. 实验内容

2.1 列车进栈

2.1.1 问题描述

每一时刻，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，然后给你多组出站序列，请你判断是否能够通过上述的方式从出口出来。



2.1.2 基本要求

输入第 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

使用了栈结构:
定义：栈顶，栈底，容量

```
typedef struct {
    SElemtype* base;              // 栈底指针，在栈构造之前和销毁之后，base的值为NULL
    SElemtype* top;               // 栈顶指针
    int stacksize;                // 栈的容量,即当前已分配的存储空间
} SqStack;
```

2.1.4 功能说明（函数、类）

初始化整个栈

```
int InitStack(SqStack* S) {
    (*S).base = (SElemtype*)malloc(STACK_INIT_SIZE * sizeof(SElemtype));
    if (!(*S).base) exit(OVERFLOW);
    (*S).top = (*S).base;
    (*S).stacksize = STACK_INIT_SIZE;
    return 1;
}
```

Push 插入栈

```
int Push(SqStack& S, SElemtype e) {
    if (S.top - S.base >= S.stacksize) {
        S.base = (SElemtype*)realloc(S.base, (S.stacksize + STACKINCREMENT) * sizeof(SElemtype));
        if (!S.base) exit(OVERFLOW);
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    *S.top++ = e;
    return 1;
}
```

Pop 弹出栈顶

```
char Pop(SqStack& S) {
    if (S.top == S.base) return '\0';
    return *(--S.top);
}
```

拿到栈顶值

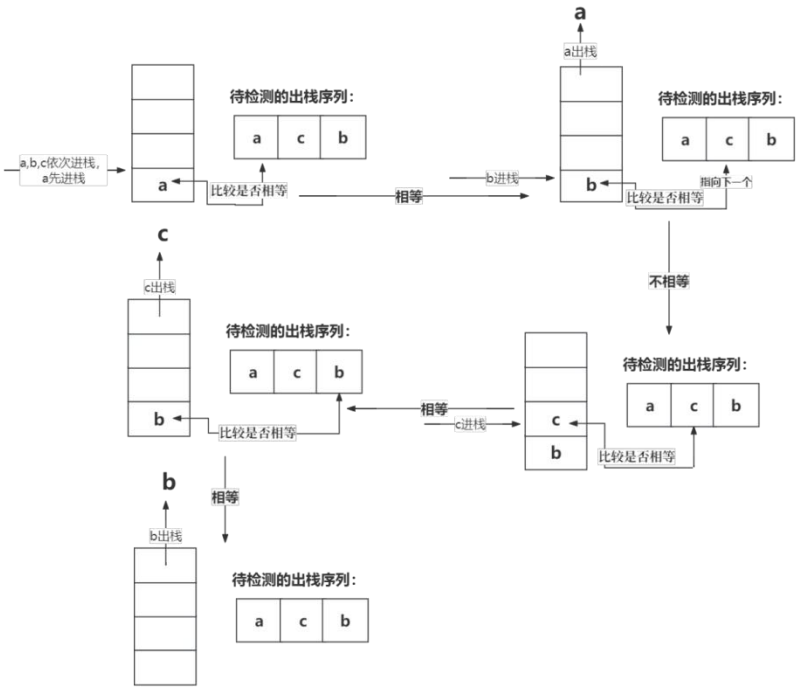
```
SElemtype GetTop(SqStack S) {
    if (S.top == S.base) return '\0';
    return *(S.top - 1);
}
```

问题核心判断是否能够实现图标方式的列车出站

```
int Judge(SqStack &S, int length, char c[], char b[]) {
    int n = 0;
    ClearStack(&S);
    for (int r = 0; r < length; r++) {
        Push(S, b[r]);
        while (GetTop(S) == c[n]) {
            Pop(S);
            n++;
        }
    }
    return StackEmpty(S);
}
```

其实就是暴力模拟一遍出站方式

如果合理的话，整个列车一定能清空



2.1.5 调试分析（遇到的问题和解决方法）

在最初的调试过程中，发现如果输入了一个错误的，则后面出现的正确的也会被判断为错误的。
每次判断前要保证是空栈的初始状态，若里面还有上一次判断未清空的值，
则会影响本次判断的正确性。因此需要在每次判断前清空辅助栈。

```
ClearStack(&S);
```

2.1.6 总结和体会

本题目的模拟进栈算法复杂度为 O（n），即只需要模拟一遍入栈和弹栈的过程即可，效率较高。
此外本题的关键难点在于对数据的读取和切割，需要谨慎注意边界值的划分，否则很容易出现漏数据、读取不完整等情况

2.2 题目二

2.2.1 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。

例 1：对字符串 "(()())()"来说，最长的子串是"(()())"，所以长度=6，起始位置是 0。

例 2：对字符串")()"来说，最长的子串是"()",子串长度=2，起始位置是 1。

例 3：对字符串""来说，最长的子串是"",子串长度=0，空串的起始位置规定输出 0。

字符串长度：0≤n≤1*105

对于 20%的数据：0<=n<=20

对于 40%的数据：0<=n<=100

对于 60%的数据：0<=n<=10000

对于 100%的数据：0<=n<=100000

下载并运行 p125_data.cpp 生成随机测试数据

提示：查找正确的括号子串可以用栈来实现，注意会有非法的右括号，比如例 2 中的第一个右括号。

2.2.2 基本要求

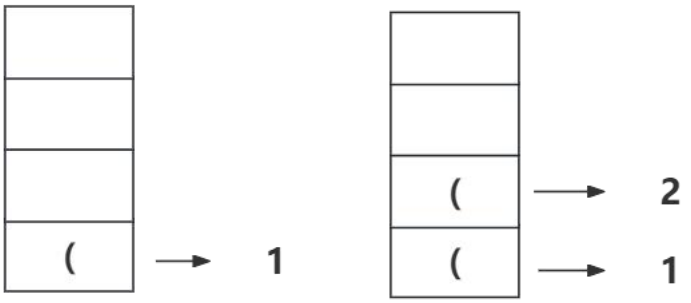
输入一行字符串。

输出子串长度，及起始位置

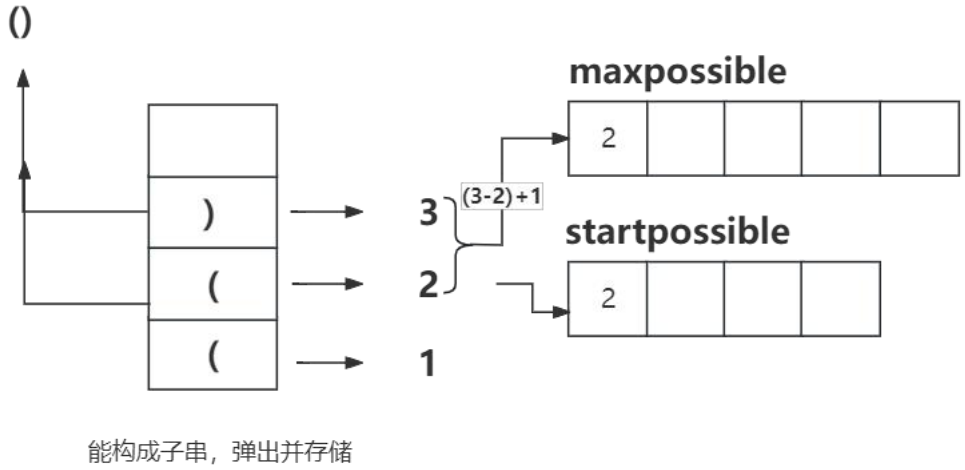
2.2.3 数据结构设计

模拟入栈过程：

① 括号依次入栈，每个元素入栈的时候，记录保存自己在栈中的次序。



② 当入栈为右括号且栈顶元素为左括号的时候（可构成子串），则返回位置差以及初始位置到数组中存储（数组中存储的是所有子串的长度以及初始位置），同时将括号都弹出。




```
int dp[100005];
```

只需要数组模拟一个栈结构

2.2.4 功能说明（函数、类）

```
int longestValidParentheses(string s, int &start) {

    memset(dp, 0, sizeof(dp));
    int maxValue = 0;
    int prev;


    for (int i = 1; i < s.size(); i++) {
        if (s[i] == ')') {
            prev = i - 1 - dp[i - 1];
            if (prev >= 0 && s[prev] == '(') {
                dp[i] = dp[i - 1] + 2;
                if (prev - 1 >= 0)
                    dp[i] += dp[prev - 1];
            }
        }

        if (dp[i] > maxValue) {
            start = i - dp[i] + 1;
            maxValue = dp[i];
        }
    }
    return maxValue;
}
```

这里执行了一个模拟入栈过程，再进行一个长度的选择，最后筛选最大值

① 将序列读入到 string s[]中：

② 定义一个数组 dp[]，其保存的是以第 i 个位置为终止右括号的子列的长度：

则有转移方程 dp[i]=dp[i-1]+2;

③ 判断某个右括号是否存在配对的左括号：

若存在配对左括号，则从 i 位置开始，往前推 dp[i]-2=dp[i-1]个位置应该是左括号，即 s[i-dp[i-1]-1]='('。

④ 合并并列子列：在该子列左侧并列的子列的最右端括号的位置为 i-dp[i-1]-2，因此：dp[i]+=dp[i-dp[i-1]-2]即可。

2.2.5 调试分析（遇到的问题和解决方法）

2.2.6 总结和体会

该题目综合性很强，解法也很多，对于不同的任务需求需要选择不同的算法完成。用动态规划算法设计到动态规划的思想，需要推导相应的关系式，但是时间复杂度低。用栈的算法思路简单，但涉及到排序的操作，很容易超时；

2.3 题目三 布尔表达式

2.3.1 问题描述

计算如下布尔表达式（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，所以表达式的总长度，即字符的个数，是未知的。

对于 20%的数据，有 A<=5，N<=20，且表达式中包含 V、F、&、|

对于 40%的数据，有 A<=10，N<=50，且表达式中包含 V、F、&、|、!

对于 100%的数据，有 A<=20，N<=100，且表达式中包含 V、F、&、|、!、(、)

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（注意冒号后面有空格）。

2.3.3 数据结构设计

①运算符栈：存放运算符号及其相关信息（优先级、左右结合性）

```
struct Elem {
    char fuhao;
    int priority;
    int L_R;
};
```

②VF 字符栈

```
typedef struct
{
    Elem* base;
    Elem* top;
    int stacksize;
}SqStack;
```

两个栈功能类似，下面以符号栈为例剖析其功能设计：

① 清空栈
② 判断栈是否为空
③ 弹出栈顶元素
④ 把元素加入栈顶
⑤ 获取栈 顶元素
⑥ 修改栈顶元素

2.3.4 功能说明（函数、类）

①VF 字符：读入 V、F 字符依次使之入参数栈，留待计算；
②运算符号：读入符号，如果：
a)优先级高于符号栈的栈顶的符号，则直接进栈；
b)优先级低于符号栈的栈顶的符号，则栈顶元素开始进行计算；计算完成后，再比较读入符

号 与当前栈顶计算符的优先级，规则与前相同。

c)优先级等于符号栈栈顶的符号，若符号是左结合（&、|、）），情况同 b），即需要使栈顶元素开始计算；若符号是右结合（!），则情况同 a），直接入栈。

以下表格展示了各个运算符的优先级、左右结合情况以及运算方法：

符号形式	优先级（相对顺序）	左/右结合	运算方式
“！”	2	右	取参数栈栈顶元素，进行取反的逻辑运算，再放回参数栈栈顶
“&”	3	左	依次从参数栈的栈顶取出两个元素，进行逻辑取并运算，再放回参数栈栈顶
“|”	4	左	依次从参数栈的栈顶取出两个元素，进行逻辑取或运算，再放回参数栈栈顶
“（”	进入前为 1，进入后为 2		运算符栈栈顶为“(”时，和“（”结合（即使“（”从栈中弹出）
“）”	5	左	

①取并(&)运算：

```
}
void calculate_and(SqStack2* VF_stack) {
    char VF1 = GetTop2(*VF_stack);
    Pop2(*VF_stack);
    char VF2 = GetTop2(*VF_stack);
    Pop2(*VF_stack);
    char VF;
    if (VF1 == 'V' && VF2 == 'V') {
        VF = 'V';
    }
    else {
        VF = 'F';
    }
    Push2(*VF_stack,VF);
}
```

② 取或(|)运算

```
void calculate_or(SqStack2* VF_stack) {
    char VF1 = GetTop2(*VF_stack);
    Pop2(*VF_stack);
    char VF2 = GetTop2(*VF_stack);
    Pop2(*VF_stack);
    char VF;
    if (VF1 == 'F' && VF2 == 'F') {
        VF = 'F';
    }
    else {
        VF = 'V';
    }
    Push2(*VF_stack, VF);
}
```

③ 取否(!)运算

```
void calculate_not(SqStack2* VF_stack) {
    char VF = GetTop2(*VF_stack);
    Pop2(*VF_stack);
    if (VF == 'V')
        VF = 'F';
    else
        VF = 'V';
    Push2(*VF_stack, VF);

}
```

2.3.5 调试分析（遇到的问题和解决方法）

2.3.6 总结和体会

本题难点重在理清运算符号栈的进栈的规则，即理清楚各个符号的运算优先级，注意左括号“（”在进栈前后会发生一个优先级变化。解决好“是否入栈”的问题，接下来就是解决“怎么算”的问题，即&，|，！的运算规则。此外，对于括号的处理要格外小心，因为括号并不参与实际的符号运算，但会变相改变其他运算符的运算顺序，因此这里也把括号当做运算符进栈，并赋予其优先级。

2.4 题目四 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

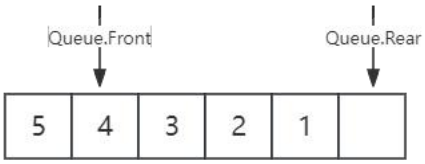
对于所有数据，0<=n,m<=1000。

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。输出：区域数

2.4.3 数据结构设计

使用队列




```
struct Point {
    int x;
    int y;
};

struct Queue {
    Point* data;
    int head;
    int tail;
};
```

定义队列元素存放每个节点的 x，y 坐标

队列的基础功能

① 在队尾添加元素

```
bool enqueue(Queue &q, Point p) {
    q.data[q.tail] = p;
    q.tail++;
    return true;
}
```

② 在队尾删除元素

```
bool dequeue(Queue &q, Point &p) {
    if (q.head < q.tail) {
        p = q.data[q.head];
        q.head++;
        return true;
    }
    return false;
}
```

③ 判断队列是否为空，是空返回 1，反之返回 0

```
bool isEmpty(Queue &q) {
    return q.head == q.tail;
}
```

2.4.4 功能说明（函数、类）

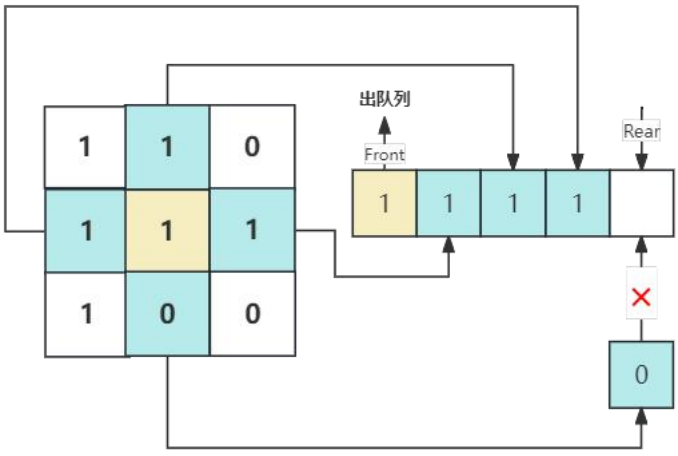
1）Input_Matrix()读入矩阵：

为了更好检测边缘，需要在原矩阵的外沿加上一圈“围墙”，围墙的值为-2.

```
void Grid(int (*grid)[1003], int &n, int &m) {
    int k;
    cin >> n >> m;
    for (int i = 0; i <= n + 1; i++) {
        for (int j = 0; j <= m + 1; j++) {
            if (i == 0 || i == n + 1 || j == 0 || j == m + 1) {
                grid[i][j] = -2;
            } else {
                cin >> k;
                grid[i][j] = k;
            }
        }
    }
}
```

2)搜索区域 Area_Search（）

功能：从起点开始，遍历一遍所有联通的 1 区域，每个被遍历过的位置都从 1 修改为-1.




```
int searchConnectedArea(int (*grid)[1003], Queue &q, int x0, int y0) {
    Point p;
    int connectedCount = 1;
    int edgeCount = 0;
    int x, y, dx, dy;
```

```
    int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    p.x = x0;
    p.y = y0;
    grid[x0][y0] = -1;

    if (grid[x0-1][y0] == -2 || grid[x0][y0-1] == -2 || grid[x0+1][y0] == -2 || grid[x0][y0+1] == -2) {
        edgeCount++;
    }

    enqueue(q, p);

    while (!isEmpty(q)) {
        dequeue(q, p);
        x = p.x;
        y = p.y;

        for (int i = 0; i < 4; i++) {
            dx = x + directions[i][0];
            dy = y + directions[i][1];

            if (checkPoint(grid, dx, dy) == 1) {
                p.x = dx;
                p.y = dy;
                enqueue(q, p);
                grid[dx][dy] = -1;
                connectedCount++;

                if (grid[dx-1][dy] == -2 || grid[dx][dy-1] == -2 || grid[dx+1][dy] == -2 || grid[dx][dy+1] ==
-2) {
                    edgeCount++;
                }
            }
        }
    }

    return (connectedCount == edgeCount) ? 0 : 1;
}
```

2.4.5 调试分析（遇到的问题和解决方法）

2.4.6 总结和体会

本题利用队列实现广度搜索算法，是仿照迷宫问题算法进行迁移。
同时本题也可以用栈来实现。栈相比于队列还有一大优势是栈更方便回溯，即对于迷宫问题需要返回路径的问题，利用栈会更方便。
实际上利用队列也可以解决迷宫问题，只需要对每个入队的元素设为结构体，包含横纵位置坐标的同时也包括了其上一个遍历结点的序号，由于出队并不是真正的释放内存空间，因此可以根据上一个结点标号去返回路径（类似树结构的子节点与父节点，只是这里是用迭代的方式写出来的）。

2.5 题目五 队列最大值

2.5.1 问题描述

（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

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。

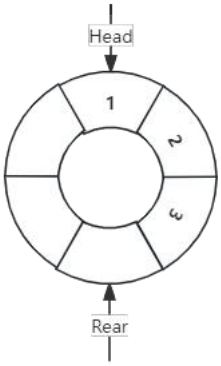
若输入"dequeue"，表示出队，当队空时，输出一行“Queue is Empty”;否则，输出出队元素；

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

2.5.3 数据结构设计



循环队列

```
struct ElemBox {//每个节点的元素域保存的是坐标值和父节点的索引
    long long value;
};
typedef ElemBox elem;
struct quene {
    elem* base;//初始化的动态分配存储空间
    int front;//头指针
    int rear;//尾指针
};
```

① 在队尾添加元素

```
        Q.base[Q.rear].value = e;
        Q.rear = (Q.rear+1)%n;
        P.base[P.rear].value = e;
        P.rear = (P.rear + 1) % n;
```

② 在队首/尾删元素

```
bool DeQueneFront(quene& Q, quene &P,long long &e,int n,int order) {//Q 为正常序列，P 为最大值单减序列
    static int empty=0;
    if (Is_Empty(Q) == 1) {
        if(order==0){      //对正常列而言
            cout << "Queue is Empty"<<endl;
        }
        return 0;
    }else{

        e = Q.base[Q.front].value;
        Q.front=(Q.front+1)%n;
        if (e == P.base[P.front].value) {    //出的数恰好是最大值时，记得要把最大值也出列
            P.front = (P.front + 1) % n;
        }
        cout << e << endl;
        return 1;
    }
```

```
}
bool DeQueneRear(quene& Q, long long& e,int n) {      //单减队列需要删除尾部操作
    if (Is_Empty(Q) == 1) {
        return 0;
    }
    else {
        e = Q.base[(Q.rear-1)%n].value;
        Q.rear = (Q.rear- 1 )%n;
        return 1;
    }
}
```

③ 判断是否为空

```
bool Is_Empty(quene Q) {//是空返回 1，不是空返回 0
    if (Q.front == Q.rear) {
        return 1;
    }
    else {
        return 0;
    }
}
```

④ 判断是否为满

```
bool Is_Full(quene Q,int n) {//是空返回 1，不是空返回 0
    if (Q.front == (Q.rear+1)%n) {
        return 1;
    }
    else {
        return 0;
    }
}
```

2.5.4 功能说明（函数、类）

获取队列中最大值元素：

方法一：从队列中遍历，时间复杂度 O（n），不建议使用，后续样例必定会超时。

方法二：构造最大值序列。

除了一个正常的保存入队元素的队列，再构建一个最大值元素一直在队首的最大值队列，则每次获得最大值的时候只需要访问该队列队首即可。

为了构建此最大值队列，可以如下操作：

一个元素入队的时候，如果队尾元素比该元素大，则先让队尾元素出队，直到队尾元素比待入队元素大或者队列为空时才允许入队。

这样可以保证越靠近队首的元素越大，队首元素为最大值。

该算法的时间复杂度为 O（1）

获得最大值 *Get_Max()*

```
void Get_Max(quene Q,quene P,int n) {
    if (Is_Empty(P) == 1) {
        cout << "Queue is Empty"<<endl;
    }else{
        cout << Q.base[Q.front].value << endl;
    }
}
```

2.5.5 调试分析（遇到的问题和解决方法）

2.5.6 总结和体会

本题设计的最大值队列算法的时间复杂度仅为 O（1），其中用到的思想是一边读一边对数据做整理，因此实际上只对所有数据进行了一次遍历。因此借助此思想，如果可以设计一个算法。在一开始数据读入过程中就将数据按照要求组织好，则可以避免二次遍历的整理。此外，本题运用了双端循环队列的数据结构。循环队列是一种原地数据结构，相比链式队列，空间利用效率更高。其中循环队列使用时，一旦出现首/尾只针的移动，一定要模 n；同时不要直接比较首位指针的绝对大小，要关注两者的相对位置关系

3.实验总结

本次上机实验运用到的主要线性数据结构式栈和队列。栈的特点是元素先进后出，队列的特点是先进先出。此外为了适应更多的任务要求，队列还包括双端队列，即在首位均可以进出的队列形式（更接近顺序表的形式，但是有两个指针，可以对首尾都进行访问）。
栈数据结构的应用十分广泛。因为其可以存储数据，并且根据栈先进后出的特性，其常见的特性是把数据存进栈中后，可以自后向前依次弹出数据再做使用（数据回溯）。根据这个特性，可以通过模拟进栈的方式对某个过程序列进行分析。如**列车出站**、**布尔表达式**以及**最长子串**，均可以借助模拟进栈的方式完成。**队列的应用**也可以用栈来完成。但是栈只能解决顺序相关的问题，如果是需要从中间增删的则需要用链表或顺序表。

队列结构的特点是先进先出。队列有两种形式，链式结构以及循环结构。其中链式结构可以采用链表或者顺序表两种方式实现，如果需要对队列中元素进行直接访问，则应该通过顺序表的方式实现，如借助队列解决迷宫问题，需要用到下标回溯检索，因此需要用到顺序表。事实上，链式队列可以当做链表和顺序表的退化形式，队列可以进一步进化成双端队列。如**队列最大值**则使用了双端队列。

此外关于本次实验对于算法的时间复杂度都提出了一定的要求。如在**最长子串**用栈解决时需要用到排序算法，如果使用复杂度为 O（n^2）的冒泡算法则效率太低会有很多样例点无法通过，因此可以考虑效率更高的快排或者堆排序。