作业 HW2 实验报告

姓名：钱子贤　学号：2065170 日期：2021 年 11 月 7 日

实验报告格式按照模板来即可，对字体大小、缩进、颜色等不做要求

实验报告要求在文字简洁的同时将内容表示清楚

1. 涉及数据结构和相关背景

栈和队列。

在数据结构中的堆栈，实际上堆栈是两种数据结构：堆和栈。堆和栈都是一种数据项按序排列的数据结构。

1.栈就像装数据的桶或箱子

它是一种具有后进先出性质的数据结构，也就是说后存放的先取，先存放的后取。这就如同我们要取出放在箱子里面底下的东西（放入的比较早的物体），我们首先要移开压在它上面的物体（放入的比较晚的物体）。

2.堆像一棵倒过来的树

而堆就不同了，堆是一种经过排序的树形数据结构，每个结点都有一个值。通常我们所说的堆的数据结构，是指二叉堆。堆的特点是根结点的值最小（或最大），且根结点的两个子树也是一个堆。由于堆的这个特性，常用来实现优先队列，堆的存取是随意，

2. 实验内容

2.1

HW2 栈和队列 > 2-1 栈的基本操作
栈
描述
栈是限制仅在表的一端插入和删除的线性表。栈的操作简单，重点掌握栈具有后进先出（LIFO）的特性。顺序栈是栈的顺序存储结构的实现。链栈是栈的链式存储结构的实现。 本题练习顺序栈的基本操作，包括入栈、出栈、判栈空、判栈满、取栈顶元素、栈的遍历。

输入
第 1 行 1 个正整数 n，表示栈的容量
接着读入多行，每一行执行一个动作。
若输入"pop"，表示出栈，当栈空时，输出一行“Stack is Empty”;否则，输出出栈的元素；
若输入"push 10"，表示将元素 10 入栈,当栈满时，输出"Stack is Full"，否则，不输出；
若输入"quit",输出栈中所有元素，以空格分割。

假定全部数值 <= 10000
输出
多行，分别是执行每次操作后的结果

2.1.4 功能说明（函数、类）

```
typedef struct {
    int* base;
    int* top;
    int length;
}STACK;


int initial(STACK& stack,int length)//初始化
{
    stack.base = (int*)calloc(length, sizeof(int));
    stack.top = stack.base;
    stack.length = length;

    return 0;
}

int push(STACK& stack, int length, int e)//入栈
{

    if (stack.top - stack.base >= length)
    {
        cout << "Stack is Full" << endl;
    }
    else
    {
        *stack.top++ = e;
    }
    return 0;
}

int pop(STACK& stack, int length)//出栈
{
    if (stack.base == stack.top)
        cout << "Stack is Empty" << endl;
    else
    {
        cout << *--stack.top << endl;
    }
    return 0;
}

int show(STACK& stack, int length)//遍历
{
    int* p;
```

```
    if (stack.base != stack.top)
        for (p = stack.top - 1; p >= stack.base; p--)
        {
            cout << *p << " ";
        }
    else
        ;
    cout << endl;
    return 0;
}

int destory(STACK& stack)//销毁
{
    free(stack.base);
    stack.base = NULL;
    stack.top = NULL;
    stack.length = 0;
```

} }**2.1.5 调试分析（遇到的问题和解决方法）**

```
4
pop
Stack is Empty
push 10
push 2
push 3
pop
3
pop
2
push 1
push 2
push 3
push 4
Stack is Full
quit
3 2 1 10
```

无特殊问题，此题比较简单

2.1.6 总结和体会

由于顺序栈和顺序表一样， 受到最大空间容量的限制， 虽然可以在“满员”时重新分配空

间扩大容量， 但工作量较大，因此在应用程序无法预先估计栈可能达到的最大容量时，应该尽量避免使用顺序栈。

2.2 题目二

列车进站
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。

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。

```
typedef struct {
    char* base;
    char* top;
    int length;
}STACK;


int initial(STACK& stack, int length)//初始化
{
    stack.base = (char*)calloc(length, sizeof(char));
    stack.top = stack.base;
    stack.length = length;

    return 0;
}

int push(STACK& stack, int e)//进栈
{
    *stack.top = e;
    stack.top++;
    return 0;
}

int pop(STACK& stack)//出栈
{
```

```
    stack.top--;
    return 0;
}

char get_top(STACK& stack)//找出栈顶
{
    if (stack.top != stack.base)
        return *(stack.top - 1);
    else
        return 0;
}

int destory(STACK& stack)//销毁
{
    free(stack.base);
    stack.base = NULL;
    stack.top = NULL;
    stack.length = 0;
    return 0;
}
```

```
abc
abc
yes
acb
yes
bac
yes
bca
yes
cab
no
cba
yes
```

一个栈的进栈序列为 1,2,3...n，有多少种不同的出栈序列呢?

首先假设 F(n) 为序列数量为 n 的出栈序列情况

假定最后出栈的元素为 k ,则 k 取不同的值的情况是相互独立的，也就是求出每种 k 最后的出栈情况后可以用加法原理，由于 k 最后出栈，则在 k 入栈之前所有比他小的都出栈了，这里的情况为 F（k-1），所有比 k 大的元素都在 k 之前出栈，为 F（n-k）种情况，两种出栈是相互独立的，所以最后出栈元素为 k 的所有出栈序列情况为 F（k-1）*F（n-k）

2.3 题目三

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）

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

下载并运行 96.py 生成随机测试数据

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）。

样例输入　　样例输出
(V|V)&F&(F|V)Expression 1: F
!V|V&V&!F&(F|V)&(!F|F|!V&V)　　Expression 2: V
(F&F|V|!V&!F&!(F|F&V))Expression 3: V
提示：

利用栈对表达式求值

```
memset(s, 0, sizeof s);
    while (cin.getline(s, 10000000))
    {
        cases++;
        while (!q.empty())q.pop();
        while (!p.empty())p.pop();
```

```
        work(); calcu();
        printf("Expression %d: %c\n", cases, p.top());
    }
    return 0;
}
int advance(char ch)//优先级
{
    if (ch == '!')return 3;
    else if (ch == '&')return 2;
    else if (ch == '|')return 1;
    return 0;
}
void work() //做运算
{
    int length = strlen(s);
    for (int i = 0; i < length; i++)
    {
        char ch = s[i];
        if (ch != ' ')
        {
            if (ch == 'V' || ch == 'F')
                q.push(ch);
            else if (advance(ch))
            {
                if (p.empty())
                {
                    p.push(ch);
                    continue;
                }
                if (ch == '!' && p.top() == '!')
                {
                    p.pop();
                    continue;
                }
                while (!p.empty() && p.top() != '(' && advance(p.top()) >= advance(ch))
                {
                    q.push(p.top()); p.pop();
                }
                p.push(ch);
            }
            else if (ch == '(')
                p.push(ch);
            else if (ch == ')')
            {
```

```
                while (p.top() != '(')
                {
                    q.push(p.top()); p.pop();
                }
                p.pop();
            }
        }
    }
    while (!p.empty())
    {
        q.push(p.top()); p.pop();
    }

    return;
}
void calcu()//计算结果
{
    char a, b, ch;
    while (!q.empty())
    {
        ch = q.front();
        if (ch == '!')
        {
            int a = p.top();
            if (a != '!')
            {
                p.pop();
                p.push('V' + 'F' - a);
            }
        }
        else if (ch == '&') {

            a = p.top(); p.pop(); b = p.top(); p.pop();
            if (b != a)p.push('F');
            else if (b == a)p.push(a);
            else { p.push(b); p.push(a);
            }
        }
        else if (ch == '|')
        {
            a = p.top(); p.pop(); b = p.top(); p.pop();
            if (b != a)p.push('V');
            else if (b == a)p.push(a);
            else { p.push(b); p.push(a);
```

```
            }
        }
        else p.push(ch);
        q.pop();
    }
```

```
(V|V)&F&(F|V)
Expression 1: F
(V|V)&F&(F|V)
Expression 2: F
(V|V)&F&(F|V)
Expression 3: F
!V|V&V&!F&(F|V)&(!F|F|!V&V)
Expression 4: V
```

C++的短路求值 对于&&或者||运算，多个表达式时，按照计算规则，C++在判断左边的表达式后，如果可以得到结果，后面的表达式便不再计算。

枚举类型 enum 其值是由一系列 int 类型的常量来定义的一种类型。枚举类型和已经声明常量的一个列表非常相似。定义枚举类型时可以使用任何 int 值，并可在枚举类型中定义任何数量的常量。例如：

enum MonthLength{ JAN_LENGTH=31, FEB_LENGTH=28, MAR_LENGTH=31, APR_LENGTH=30, MAY_LENGTH=31, JUN_LENGTH=30, JUL_LENGTH=31, AUG_LENGTH=31, SET_LENGTH=30, OCT_LENGTH=31, NOV_LENGTH=31, DEC_LENGTH=31};

枚举类型返回值 在一个枚举类型中，已命名的常量可接受同一个 int 值。如果不指定任何数值，枚举类型中的标识符就会自动指派一系列连续的值，这些值从 0 开始，依次递增。

如果只是指派了部分值，则未指派值的量的值为上一个值加 1。例如：

enum MyEnum{ONE=17, TWO, THREE, FOUR=-3, FIVE};

那么，ONE 值为 17，TWO 为 18，THREE 为 19，FOUR 为-3，FIVE 为-2。

2.4 题目四

队列的应用

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。

对于所有数据，0<=n,m<=1000

下载编译并运行 p44.cpp 生成随机测试数据

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
1 行，代表区域数

```
void check(int x1, int y1, int value)//遍历寻找
{
    if (x1 < 0 || y1 >= m)
        return;
    if (x1 - 1 >= 0 && a[x1 - 1][y1] == value)
    {
        a[x1 - 1][y1] = -1;
        check(x1 - 1, y1, value);
    }
    if (y1 - 1 >= 0 && a[x1][y1 - 1] == value)
    {
        a[x1][y1 - 1] = -1;
        check(x1, y1 - 1, value);
    }
    if (x1 + 1 < n && a[x1 + 1][y1] == value)
    {
        a[x1 + 1][y1] = -1;
        check(x1 + 1, y1, value);
    }
    if (y1 + 1 < m && a[x1][y1 + 1] == value)
    {
        a[x1][y1 + 1] = -1;
        check(x1, y1 + 1, value);
    }
}
int main()
{
    int count = 0;
    cin >> n >> m;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            cin >> a[i][j];
    for (int i = 1; i < n-1; i++)
        for (int j = 1; j < m-1; j++)
        {
            if (a[i][j] != -1)
            {
                if (a[i][j] == 1)
                {
```

```
                    check(i, j, a[i][j]);
                    a[i][j] = -1;
                    count++;
                }
            }
        }
    cout << count;
```

```
9 9
0 0 0 1 1 0 0 0 0
0 1 1 0 0 1 1 1 0
0 1 1 0 0 1 1 1 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 0
0 0 1 0 1 0 1 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
}3
```

四连通线和 8 连通线上的点是怎么确定的，查到 opencv 中这两种直线都是由 Bresenham 算法插值计算出来的，所以又去找了算法的原理，但并没有看出两种线型上的点到底如何确定的。比较焦灼，不想算，后来找到这两个画线的代码，记录一下，以后再研究。

给定一个方阵，定义连通性：上下左右相邻，并且值相同。可以想象一张地图，不同的区域被染成了不同的颜色。现在我们需要判断图中任意两点是否在同一个连通区间中。

2.5 题目五

队列中的最大值问题
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。

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
对于20%的数据，有0<=n<=100，|max(m) - min(m)|<=1e4；
对于40%的数据，有0<=n<=6000，|max(m) - min(m)|<=1e6；（未优化O(nm)程序运行时间略微超过一秒）
对于100%的数据，有0<=n<=10000,|max(m) - min(m)|<=1e8；
对于每个测试点，0x80000000 <= min(m) < max(m) <= 0x7fffffff
对于每个测试点，操作的个数约为队列大小的10倍左右
事实上，测试数据保证对于后80%左右的操作，队列内空位不会超过15%
具体测试数据生成见p101.py，下载并运行p101.py生成随机测试数据
输出
多行，分别是执行每次操作后的结果

```
class MaxQueue {
    queue<int> que;
    deque<int> deq;
public:
    MaxQueue() { }
    int max_value() {
        return deq.empty() ? -1 : deq.front();
    }
```

```
    void push_back(int value)//入栈
    {
        que.push(value);
        while (!deq.empty() && deq.back() < value)
            deq.pop_back();
        deq.push_back(value);
    }
    int pop_front() //出栈
    {
        if (que.empty()) return -1;
        int val = que.front();
        if (val == deq.front())
            deq.pop_front();
        que.pop();
        return val;
    }
    int queuelength()//队列长度
    {
       return que.size();
    }
    int pop_all()//全出
    {
        if (que.empty()) return -1;
        while (!que.empty())
        {
            cout << pop_front() << " ";
        }
        cout << endl;
        return 0;
    }
    int destory()//销毁
    {
        deq.clear();
        return 0;
    }
};
```

```
4
dequeue
Queue is Empty
enqueue 10
enqueue 2
enqueue 3
max
10
dequeue
10
max
3
dequeue
2
enqueue 1
enqueue 2
enqueue 3
enqueue 4
Queue is Full
max
3
quit
3 1 2 3
```

这个题目需要去搜索才能得到最优解。
思路

本算法基于问题的一个重要性质：当一个元素进入队列的时候，它前面所有比它小的元素就不会再对答案产生影响。

举个例子，如果我们向队列中插入数字序列 1 1 1 1 2，那么在第一个数字 2 被插入后，数字 2 前面的所有数字 1 将不会对结果产生影响。因为按照队列的取出顺序，数字 2 只能

在所有的数字 1 被取出之后才能被取出，因此如果数字 1 如果在队列中，那么数字 2 必然也在队列中，使得数字 1 对结果没有影响。

按照上面的思路，我们可以设计这样的方法：从队列尾部插入元素时，我们可以提前取出队列中所有比这个元素小的元素，使得队列中只保留对结果有影响的数字。这样的方法等价于要求维持队列单调递减，即要保证每个元素的前面都没有比它小的元素。

那么如何高效实现一个始终递减的队列呢？我们只需要在插入每一个元素 value 时，从队列尾部依次取出比当前元素 value 小的元素，直到遇到一个比当前元素大的元素 value' 即可。

上面的过程保证了只要在元素 value 被插入之前队列递减，那么在 value 被插入之后队列依然递减。
而队列的初始状态（空队列）符合单调递减的定义。
由数学归纳法可知队列将会始终保持单调递减。
上面的过程需要从队列尾部取出元素，因此需要使用双端队列来实现。另外我们也需要一个辅助队列来记录所有被插入的值，以确定 pop_front 函数的返回值。

保证了队列单调递减后，求最大值时只需要直接取双端队列中的第一项即可。

3．实验总结

先来简单的了解一下栈

1.栈：一种特殊的线性表，其实只允许在固定的一端进行插入或删除操作。进行数据插入和删除的一端称为栈顶，另一端称为栈底。不含任何元素的栈称为空栈，栈又称为 后进先出的线性表。

特性栈：后进先出（LILO）特殊线性表

栈功能：将数据从一种序列改变为另一种序列

2.顺序栈和顺序表数据成员相同，不同之处： 顺序栈的入栈和出栈操作只允许对当前栈顶进行操作!

顺序栈所有的的操作时间复杂度为 O（1）

注意：面试中如果需要用到栈，封装静态栈

队列(Queue)是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。允许插入的端是队尾，允许删除的端是队头。

所以说队列是一个先进先出的线性表，相应的也有顺序存储和链式存储两种方式。

顺序存储就是用数组实现，比如有一个 n 个元素的队列，数组下标 0 的一端是队头，入队操作就是通过数组下标一个个顺序追加，不需要移动元素，但是如果删除队头元素，后面的元素就要往前移动，对应的时间复杂度就是 O(n)，性能自然不高。

同时规定，当队列为空时，front 和 rear 相等，那么队列什么时候判断为满呢？按照循环操作 rear 依次后移，然后再从头开始，也是出现 rear 和 front 相等时，队列满。这样跟队列空的情况就相同了，为了区分这种情况，规定数组还有一个空闲单元时，就表示队列已满，因为 rear 可能在 front 后面，也可能循环到 front 前面，所以队列满的条件就变成了(rear+1)%maxsize = front ,同时队列元素个数的计算就是(rear -

front+maxsize)%maxsize。

为了提高出队的性能，就有了循环队列，什么是循环队列呢？就是有两个指针，front 指向队头，rear 指向对尾元素的下一个位置，元素出队时 front 往后移动，如果到了对尾则转到头部，同理入队时 rear 后移，如果到了对尾则转到头部，这样通过下标 front 出队时，就不需要移动元素了。