作业HW2报告

1. 涉及数据结构和相关背景

1.1 栈

后进先出LIFO，限定只在表的一端(表尾)进行插入和删除操作的线性表

允许插入和删除的一端称为栈顶(top)，另一端称为栈底(bottom)

- 基本操作：初始化、判空、push、pop、gettop、清空、返回长

- 顺序栈
  - 预先定义好栈的容量
  - 限定在表尾进行插入和删除操作的顺序表
  - base称为栈底指针，始终指向栈底；当base == NULL时，表明栈结构不存在
  - 空栈：当top=base时为栈空的标记
  - 当栈非空时，top的位置：指向当前栈顶元素的下一个位置
  - 当栈满时再做进栈运算必定产生空间溢出，简称“上溢”; 当栈空时,再做退栈运算也将产生溢出，简称为“下溢“

- 链栈
  - 无需预先定义好栈的容量
  - 不带头结点的单链表，其插入和删除操作仅限制在表头位置上进行。链表的头指针即栈顶指针
  - 栈空条件： s=NULL；栈满条件： 无 / 无Free Memory可申请

1.2 队列

队列是限定在表的一端进行删除，在表的另一端进行插入操作的线性表。

允许删除的一端叫做队头(front)，允许插入的一端叫做队尾(rear)。

特性：FIFO(First In First Out)

- 链队列
  - 实质是带头结点的线性链表

- 基本操作：初始化、销毁队列、enqueue、dequeue、判空、取队头元素
- 两个指针：
  - 队头指针Q.front指向头结点
  - 队尾指针Q.rear指向尾结点
- 初始态：队空条件
  - 头指针和尾指针均指向头结点

    Q.front = Q.rear

- 顺序队列
  - 用一组地址连续的存储单元依次存放

    从队列头到队列尾的元素
  - 头指针与尾指针
    - Q.front 指向**队列头元素**；
    - Q.rear 指向**队列尾元素的下一个位置**
  - 初始状态
    - Q.front = Q.rear=0
    - 队列的真满与假满

- 循环队列
  - 存储队列的数组被当作首尾相接的表处理。
  - 队头、队尾指针加1时从maxsize -1直接进到0，可用语言的取模(余数)运算实现
  - 队头指针进1: Q.front = (Q.front + 1)% MAXSIZE

    队尾指针进1: Q.rear = (Q.rear + 1)% MAXSIZE;
  - 队列初始化：Q.front = Q.rear = 0;
  - 队空条件：Q.front == Q.rear;
  - 队满条件：(Q.rear + 1) % MAXSIZE == Q.front
  - 队列长度：(Q.rear-Q.front+MAXSIZE)%MAXSIZE
  - 注意：
    - 不能用动态分配的一维数组来实现循环队列，初始化时必须设定一个最大队列长度。
    - 循环队列中要有一个元素空间浪费掉，约定队列头指针在队列尾指针的下一位置上为“满”的标志
    - 解决 Q.front=Q.rear不能判别队列“空”还是“满”的其他办法：
      - 计数器
      - 标志变量

- 非循环队列
  - 关键：修改队尾/队头指针Q.rear = Q.rear + 1; Q.front = Q.front+1;
  - 在判断时，有%MAXQSIZE为循环队列，否则为非循环队列
  - 队空条件： Q.front = Q.rear
  - 队满条件： Q.rear>= MAXQSIZE

- 注意“假上溢”的处理
- 长度： Q.rear - Q.front

2. 实验内容

2.01 链栈模板

2.0.1 抽象栈 AbstractStack.h

```
template<class dataType>


class MyStack
{
public:
    virtual bool isEmpty() = 0;  //判空
    virtual void push(const dataType& d) = 0;  //push
    virtual dataType pop() = 0;  //pop
    virtual dataType top() = 0;  //top

};
```

2.0.2 模板链栈Stack.hpp

```
#pragma once
#include "AbstractStack.h"
#include <iostream>
#include <stdio.h>
using namespace std;


template<class dataType>
struct node  //链栈节点
{
    node();  // 无参构造
    node(const dataType& d, node<dataType>* n = NULL)  //有参构造
    {
        data = d;
        next = n;
    }
    dataType data;
    node<dataType>* next;

};


template<class dataType>
class Stack : public MyStack<dataType>
{
public:
```

```
    Stack()  //无参构造
    {
        top_p = NULL;
        //cout << "New LinkedStack Created!" << endl;
    }

    bool isEmpty()  //判空
    {
        return top_p == NULL;
    }

    void push(const dataType& d)  //push
    {
        node<dataType>* add = new node<dataType>(d, top_p);
        top_p = add;
        //cout << "Push data successfully" << endl;
    }

    dataType pop()  //pop
    {
        if (isEmpty())
        {
            throw 0;
        }

        dataType d = top_p->data;
        node<dataType>* tmp = top_p;
        top_p = top_p->next;
        delete tmp;
        return d;
    }

    dataType top()  //取栈顶
    {
        if (isEmpty())
        {
            throw 0;
        }

        return top_p->data;
    }

    ~Stack()  //析构
    {
        while (top_p != NULL)
        {
            node<dataType>* tmp = top_p;
            top_p = top_p->next;
            delete tmp;
        }
    }


    node<dataType>* top_p;
```

```
};
```

实验中会使用到此模板

2.02 链队列模板

2.0.1 链队列存储

```
/*链式队列结点*/
typedef struct {
    ElemType data;
    struct LinkNode *next;
}LinkNode;
/*链式队列*/
typedef struct{
    LinkNode *front, *rear; //队列的队头和队尾指针
}LinkQueue;
```

2.0.2 链队列初始化

```
void InitQueue(LinkQueue *Q){
    Q->front = Q->rear = (LinkNode)malloc(sizeof(LinkNode));    //建立头结点
    Q->front->next = NULL;  //初始为空
}
```

2.0.3 链队列入队

```
Status EnQueue(LinkQueue *Q, ElemType e){
    LinkNode s = (LinkNode)malloc(sizeof(LinkNode));
    s->data = e;
    s->next = NULL;
    Q->rear->next = s;  //把拥有元素e新结点s赋值给原队尾结点的后继
    Q->rear = s;    //把当前的s设置为新的队尾结点
    return OK;
}
```

2.0.4 链队列入队

```
/*若队列不空，删除Q的队头元素，用e返回其值，并返回OK，否则返回ERROR*/
Status DeQueue(LinkQueue *Q, Elemtype *e){
    LinkNode p;
    if(Q->front == Q->rear){
        return ERROR;
    }
    p = Q->front->next; //将欲删除的队头结点暂存给p
    *e = p->data;   //将欲删除的队头结点的值赋值给e
    Q->front->next = p->next;   //将原队头结点的后继赋值给头结点后继
```

```
    //若删除的队头是队尾，则删除后将rear指向头结点
    if(Q->rear == p){
        Q->rear = Q->front;
    }
    free(p);
    return OK;
}
```

实验中会用到此模板

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，需要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。

2.1.2 基本要求

- 输入：一行字符串（纯括号）
- 输出：最长合法子串长度，及其起始位置

限制：字符串长度：小于1e5，空字符串输出0 0

2.1.3 数据结构设计

利用栈的数据结构，当遇到左括号时，就把这个左括号的位置push进栈，当遇到右括号时，先将栈顶元素出栈，再判断栈是否空，如果空（说明这时这个右括号没有匹配的左括号，是多出来的，是合法右括号的下一个右括号）则将这个右括号的位置入栈，否则判断当前右括号的位置和栈顶记录的位置之差是否大于最大值，是的话更新最大值。

- 相当于遇到右括号消去一个左括号，并且用右括号的下标减去左括号的下标表示有效括号的长度
- 特别注意在赋初值的时候最初栈里需要有一个-1，因为在做差比较长度大小的时候有两种情况
    - 栈顶是之前合法右括号右边多出来的右括号，是下一个合法括号组的开始前一位（当有多个右括号连在一起时，先出栈再判断栈是否空的机制可以保证这里的右括号是下一个合法字串开端的前一位）
    - 栈顶是合法括号组中上对应此右括号的左括号的上一位

  这两种情况都是当前合法串的上一位，所以在进行减法操作的时候直接将坐标相减即可，对应于此，当合法字串从0位置开始时，栈里应有一个准备元素-1保证减法的一致性

2.1.4 功能说明

```
#include <iostream>
#include ”Stack.h“
using namespace std;
int start;
```

```
int main()
{
    string s;
    cin >> s;

    int n = s.length();
    if (n == 1 || n == 0)
    {
        cout << 0 << " " << 0;
        return 0;
    }

    Stack<int> stk;

    stk.push(-1);

    int ret = 0;
    for (int i = 0; s[i]; ++i) {
        if (s[i] == '(')   //左括号位置入栈
        {
            stk.push(i);
        }
        else
        {
            stk.pop();  //先出栈
            if (stk.isEmpty()) stk.push(i);  //栈空直接push进去
            else
            {
                if (i - stk.top() > ret)  //更新最大值
                {
                    ret = i - stk.top();
                    start = stk.top() + 1;  //当最大值更新的时候才更新起始位置
                }
            }
        }
    }
    cout << ret << " " << start << endl;


    return 0;
}
```

> 注意要对于0 和 1进行一个特判
>
> 注意起始位置要+1，因为stk.top()指的是合法子串开头的前一个的位置

2.1.5 调试分析

- 在之前的尝试中，尝试过在栈里仅仅存放左括号的位置，右括号入栈相消，栈空判断长度，计数器置0。错误：当有多个合法的括号组合依次向右拼接，只能记录到第一组组合的值。
- 通过将右括号放入栈中的方式可以有效解决记录合法子串左端点位置的问题，使得问题中判断合法子串长度的方法更加统一
- 出栈的时机很重要，总是先出栈再判断，保证相减时总是减去合法长度的前一位

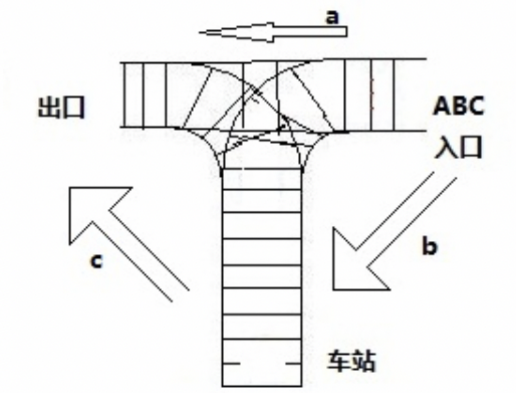
2.1.6 总结和体会

- 入栈相消是判断括号对数的基本方法，同时可以用于表达式求值当中
- 对于多种特殊情况，可能要改变技术方法来使得计算形式统一简洁

2.2 列车进站

2.2.1 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，判断是否能够通过上述的方式从出口出来。



2.2.2 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。

字符串长度范围没有明确给出

2.2.3 数据结构设计

主要是对系统进行模拟

遍历每一位入栈序列，

- 当入栈序列和出栈序列相同时，说明这一位入栈序列直接出栈，入栈序列和出栈序列同时往后移一位
- 当入栈序列和出栈序列不相同时
    - 判断栈顶元素是否和这一位出栈元素相同，如果相同，则栈顶元素出栈，观察出栈元素的下一位
    - 如果栈空，或者栈顶元素与这一位出栈元素不匹配，则将这一位进栈元素压入栈中

遍历完成，观察栈中是否有元素，如果有，依次出栈与剩下没有观察的出栈序列进行比对，如果相同就说明有这个出栈序列，否则此出栈序列不可以实现

2.2.4 功能说明

```
#include <iostream>
#include "Stack.h"
using namespace std;
string stand, s;
int main()
{
    cin >> stand;
    while (cin >> s)
    {
        Stack<char> st;
        int ptr = 0;
        /*i : stand , ptr : s*/
        /*ptr:遍历出栈序列的指针*/
        for (int i = 0; i < s.size(); i++)  //遍历每一个入栈序列
        {
            if (stand[i] == s[ptr]) //与出栈字符相同
            {
                ptr++;
                continue;
            }
            else
            {
                if (!st.isEmpty())  //栈不空
                {
                    if (s[ptr] == st.top())  //栈顶元素与出栈字符相同
                    {
                        st.pop();
                        ptr++;  //出栈序列指针后移
                        i--;  //入栈序列指针不变（因为下一次遍历会进行加一的操作）
                    }

                    else
                    {
                        st.push(stand[i]);  //与栈顶元素不同，入栈
                    }
                }
                else  //栈空直接push
                {
                    st.push(stand[i]);
                }
            }
        }
        if (ptr >= s.size()) ptr--;  //因为ptr一直都是指向出栈序列的下一个元素，所以有可能ptr指向了出栈元素最后一位的下一位，所以需要减一下

        int flg = 1;
        while (!st.isEmpty())  //栈不空，依次出栈比较
        {
            if (st.top() != s[ptr++])
            {
                flg = 0;
```

```
            }
            st.pop();
        }
        if (!flg) cout << "no" << endl;
        else cout << "yes" << endl;

    }
    return 0;
}
```

> 注意输入格式，while (cin >> s) 当空行输入的时候就跳出循环停止，或者可以写成while (scanf("%s", s) != EOF)，当需要读入字符串之间的空格时，可以用while (getline(cin, s))，想跳过空行可以在循环里边用输入字符串长度为0进行判断

2.2.5 调试分析

- 第一次尝试是另一种思路：遍历每一个元素，如果这个元素不是和原入栈序列一样，这个元素在入栈序列前面的字符（除了遍历过的）必须在出栈序列中都在这个字符的后面而且是逆序的，因为一个元素不在原位就说明它肯定有一些前边的元素入栈了，那这些元素在出栈时一定是逆序的（不一定紧挨着），因为在每一次观察的时候都要打好标记，所以不会重复的。但是此方法只过了一半的测试点
- 关键点是二者不同的情况，先要与栈顶元素进行比较，如果相等就出栈，不等或者为空才入栈，而不是不相等就直接入栈了，否则遗漏很多情况
- 根据简单测试用例来进行模拟，能模拟就先朴素模拟，根据每一步步骤（如何时出栈入栈，何时继续比较下一位，何时指针移动，何时终止循环等）

2.2.6 总结和体会

- 模拟可以解决就模拟进行解决，但是要注意在模拟的过程中的每一处细节，每一个步骤都不是理所当然的
- 不一样先比较栈顶再决定是否入栈是关键，否则会丢失很多情况
- 熟悉一下读到EOF就停止的这种形式

2.3 布尔表达式

2.3.1 问题描述

计算布尔表达式 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`）并且按照要求输出结果

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，同时所有测试数据中都可能穿插空格，不考虑空格空行的话会出现无法判断的WA或者RE
- 输出：真输出为V，假输出为F，格式为：

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，Expression和序号中间也有空格）

2.3.3 数据结构设计

- 类似于四则运算，将运算符分别赋值相应的等级，根据大压小出栈的规律进行运算，注意这里左括号和右括号的优先级要特别赋值
- 类似四则运算，将运算符映射到其优先级，遇到数入运算数栈，遇到运算符，先判断栈顶运算符是优先级否比要入栈这个小或等于入栈这个，如果是就入栈如果是大于就先让底下的运算符出栈，直到满足这个条件或者栈空。
- 要注意的是，左括号入栈前优先级最高，入栈后优先级最低，所以左括号无条件入栈，为保证后续涉及到左括号出栈时不出错，入栈判断时需要把左括号单独拿出来进行入栈，将左括号的优先级赋值为最低，这样以后有元素再压入之前让底下元素出栈时就不会误让左括号出栈了
- 对于右括号的判断就是运算符号栈中直到左括号出现之前的所有元素，最后将左括号消去，所以将右括号也赋值为最低等级，确保其栈下边到左括号之前的所有符号顺利出栈
- 因为数学运算会重复使用很多次，所以可以抽象成函数进行运算

2.3.4 功能说明

- doMath()函数实现

```
void doMath(char op, Stack<int>& stshu)
{
    /*
    * @param op : 运算符
    * @param stshu : 运算数栈
    */


    if (op == '!')  //取非
    {
        int temp = !stshu.top();
        stshu.pop();
        stshu.push(temp);
    }
    else if (op == '&')  //取与
    {
        int a = stshu.top();
        stshu.pop();
        int b = stshu.top();
        stshu.pop();
        stshu.push(a & b);
    }
    else if (op == '|')  //取或
    {
        int a = stshu.top();
        stshu.pop();
        int b = stshu.top();
        stshu.pop();
        stshu.push(a | b);
```

```
    }
}
```

> 注意取非是一元运算，与另外两个取两个操作数不相同
>
> 特别注意取top和pop的区别

- 主函数实现

```
int main()
{
    //map映射，可以用二维数组替代
    mp.insert({ '(', 0 });
    mp.insert({ '|', 1 });
    mp.insert({ '&', 2 });
    mp.insert({ '!', 3 });
    mp.insert({ ')', 0 });
    int cnt = 0;
    while (getline(cin,s))
    {
        if (!s.size()) continue;  // 空行判断
        Stack<int> stshu;
        Stack<char> stfuhao;
        for (int i = 0; i < s.size(); i++)
        {
            if (s[i] == ' ' || s[i] != 'V' && s[i] != 'F' && s[i] != '&' &&
s[i] != '|' && s[i] != '(' && s[i] != '!' && s[i] != ')') continue;  // 检查
是否有非法字符
            /*数字入栈*/
            if (s[i] == 'V')
            {
                stshu.push(1);
            }
            else if (s[i] == 'F')
            {
                stshu.push(0);
            }
            /*符号入栈*/
            else
            {
                /*左括号直接进栈*/
                if (s[i] == '(')
                {
                    stfuhao.push('(');
                    continue;
                }
                /*栈为空，直接进*/
                else if (stfuhao.isEmpty())
                {
                    stfuhao.push(s[i]);
                    continue;
                }
                else
                {
                    /*如果是右括号，直到遇到左括号之前，一直做运算，最后把左括号删掉*/
```

```
                    if (s[i] == ')')
                    {
                        while (stfuhao.top() != '(')
                        {
                            char tmp = stfuhao.top();
                            stfuhao.pop();
                            doMath(tmp, stshu);
                        }
                        stfuhao.pop();
                        continue;
                    }
                    else
                    {
                        while (!stfuhao.isEmpty() && mp[s[i]] <
mp[stfuhao.top()])  //短路运算 //小于
                        {
                            char fu = stfuhao.top();
                            doMath(fu, stshu);
                            stfuhao.pop();
                        }
                        stfuhao.push(s[i]);
                        continue;
                    }
                }
            }
        }

        while (!stfuhao.isEmpty())  //符号栈中存在运算符，需要依次出栈运算
        {
            char opr = stfuhao.top();
            stfuhao.pop();
            doMath(opr, stshu);
        }
        if (stshu.top() == 1) cout << "Expression " << ++cnt << ": " << 'V'
<< endl;
        else cout << "Expression " << ++cnt << ": " << 'F' << endl
    }
    return 0;
}
```

> 注意左括号需要单独判断是否入栈，以及运算符的取出，pop出的过程
>
> 注意输出格式

2.3.5 调试分析

- 第一次尝试在调试大部分表达式时并没有产生问题，但出现问题在两个取非运算，最初我设置的是当进栈运算符优先级比下边的要小或者等于才出栈，但是遇到!!并不正确，取非运算是一元运算，其作用域仅在后边，当取非运算按照平级也出栈的方式进行运算，其运算数还没有入运算数栈，所以这个非加到前一个运算数上去了，造成结果错误
- 另一个很有趣的问题是在doMath函数中的取非运算，VS中推荐使用~即取反，导致结果全部错误，后分析发现取非是取补码，取~是取反码，没有加1，所以1的取反是-2，造成结果错误

- 大部分时间是进行空白行以及空格处理，利用getline(cin, s)读入之后会将空白符也读入，需要进行一次判断，空白行的处理就是读入字符串的长度为0，直接返回即可
- 注意最后的输出格式，空格和计数器需要加一加

2.3.6 总结和体会

- 体会四则运算以及括号的运算规则，熟悉了计算机内部堆栈进行运算的过程
- 更为重要的是学会了debug的操作，以及对于空白行和非法输入的健壮性处理和输出格式的注意

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。

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。
- 输出：1行，代表区域数

限制：所有数据0 <= n, m <= 1000

2.4.3 数据结构设计

很明显本题是BFS（广度优先搜索），通过BFS搜索到每一块区域，通过排除一些特殊的区域进行计数即可

BFS：遍历一张图，找到终点就终止，在每一个岔路口探索该岔路口所有下一个节点的可能性。根据这一特点，考虑将一个队列中的头部节点出队，观察这一节点的所有分支，可行的下一个节点从队尾入列，如此就可以遍历完这一层节点的所有下一层节点，所以叫做广度优先搜索

伪代码：

```
init Q;
Q <- {Start};
Start->visited;
while (Q is not Emmpty)
{
    DeQueue Q -> u;
    if (u is equal to end status)
    {
        Enqueue (connected to u && not visited && certain criterion satisfied);
        u->visited
    }
}
```

2.4.4 功能说明

- check函数检查是否碰触边界
- check函数实现

```
bool check(int x, int y)
{
    if (x == n || x == 1 || y == m || y == 1) return 0;
    else return 1;
}
```

> 不是正常在边界的返回0
>
> 正常在边界的返回1

- 主函数BFS实现

```
const int N = 1000;
int vis[N][N];
int mp[N][N];
int xd[] = { 0, 0, -1, 1 };
int yd[] = { 1, -1, 0, 0 };
int flg;

int main()
{
    /*读入数据*/
    cin >> n >> m;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            cin >> mp[i][j];
        }
    }

    int cnt = 0;  //计数
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)  //遍历每一个元素，
        {
            if (mp[i][j] && !vis[i][j])  //如果是1而且还没有被访问过就访问一下
            {
                q.push({ i, j });  //把第一个节点入队
                vis[i][j] = 1;
                flg = 1;
                if (check(i,j)) flg = 0;

                while (q.size())  //队不空，将第一个元素出队
                {
                    pair<int, int> cod = q.front();
                    q.pop();
                    for (int i = 0; i < 4; i++)
                    {
                        int nx = cod.first + xd[i];  //遍历每一个方向
```

```
                        int ny = cod.second + yd[i];
                        if (nx <= n && nx >= 1 && ny <= m && ny >= 1 &&
mp[nx][ny] && !vis[nx][ny])
                        {
                            vis[nx][ny] = 1;  //遍历完的点打上访问过的标记
                            q.push({ nx, ny });  //符合条件的入队
                            if (check(nx, ny)) flg = 0;  //如果有一个点满足不在
边上，那这个区域就计入总数
                        }
                    }
                }
                if (!flg) cnt++;  //如果有一个点满足不在边上，那这个区域就计入总数
            }
        }
    }
    cout << cnt;
    return 0;
}
```

> 注意这里的四个方向的遍历是通过增减实现的，四个方向分别装在两个数组当中，进行遍历即可
>
> 注意这里计数的条件是只要有一个不在边上就可以计入总数

2.4.5 调试分析

- 主要在于题目的理解上，重点理解什么叫做仅在矩阵边缘联通，就是都贴在边上不行，对全称量词取反，就是存在一个单元不贴着边
- check在每一个区域访问到的元素上，如果不贴边那标志更新，计入总数

2.4.6 总结和体会

- 主要是考察BFS的应用，加一个限制条件判断是否有不在边上就可以了
- 值得注意的是这道题的测试用例给出的生成算法是DFS，经测试dfs也是可以跑通的，说明测试数据并不是很严格，时间限制没有卡的太紧
- 将待查元素放在队列中依次出队检查是一个经典方法

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

- 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

限制：

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
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%

2.5.3 数据结构设计

利用两个队列来实现存储元素和单调栈存储最大值

队列一：存储正常入队列、出队列的元素

队列二：进来元素和队尾进行比较，如果队尾元素比进来的元素要小的话就删除队尾元素，直到队空或者前边元素比进来的元素要大，让此元素入队（类似单调栈）

- 入队列正常操作，同时在最大值队列中比较一下
- 出队列如果正好出最大值，那最大值队列第一个元素也出队列，否则最大值队列不变化
- 当前队列最大值就是最大值队列的第一个元素

这样就保存了按顺序入队以来的单调元素

2.5.4 功能说明

```
int main()
{ // 这里两个栈用双端队列进行模拟，或者可以用单调栈更新最大值队列来实现
    cin >> maxa;  //队列最大长度
    while (1)
    {
        cin >> s;
        //退出
        if (s == "quit")
        {
            if (stEle.empty())
            {
                cout << "Queue is Empty" << endl;
                continue;
```

```
            }
            while (!stEle.empty())
            {
                cout << stEle.front() << " ";
                stEle.pop_front();
            }
            break;
        }

        //入队
        else if (s == "enqueue")
        {
            int e;
            cin >> e;


            if (stEle.size() >= maxa)
            {
                cout << "Queue is  Full" << endl;
                continue;
            }
            stEle.push_back(e);
            if (stMax.empty())
            {
                stMax.push_back(e);
                continue;
            }
            while (!stMax.empty() && stMax.back() < e)
            { // 与队尾元素进行比较，队尾元素小的话就出栈
                stMax.pop_back();
            }
            stMax.push_back(e);
        }

        //出队
        else if (s == "dequeue")
        {
            if (stEle.empty())
            {
                cout << "Queue is Empty" << endl;
                continue;
            }
            cout << stEle.front() << endl;
            if (stEle.front() == stMax.front())
            { // 如果出的是最大值，更新最大值栈，让其出队
                stMax.pop_front();
            }

            stEle.pop_front();

        }

        //取最大值
        else if (s == "max")
        {
```

```
            if (stEle.empty())
            {
                cout << "Queue is Empty" << endl;
                continue;
            }
            cout << stMax.front() << endl;  //取最大值队列第一个元素
        }
    }


    return 0;
}
```

2.5.5 调试分析

- 最初的想法是仅仅通过每次比较最大值，更新一个最大值就可以，但是通过测试用例发现，当删除最大值的时候，第二个大值已经被删除了，导致队列最大值还是保持原来的值不变
- 考虑到存储第二大值，可以联想到单调栈的特性，保证了在入队列方向的元素单调性，故存储最大值采用单调栈
- 考虑到是需要栈的性质，但是本质是一个队列，前端（也就是栈底）也要出元素，所以使用了双端队列进行相应的存储

2.5.6 总结和体会

其实这是一道栈和队列的结合题，通过最大值的方式考察了单调栈的使用，提供了一种动态存储最大值的方法，类似的可以求栈的最大值

3. 实验总结

本次实验的数据结构是栈和队列，其中涉及到了

- 栈的实现（此处实现的是链栈），利用栈进行模拟，单调栈的使用，栈的经典应用四则运算
- 队列的实现（此处实现的是链队列），队列经典应用BFS搜索，队列与栈的结合使用

其中BFS与单调栈都是十分经典的算法，值得深入探究