**作业 HW02 实验报告**

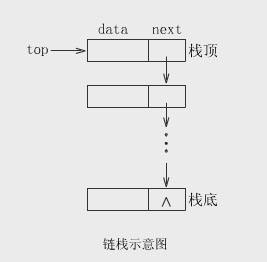
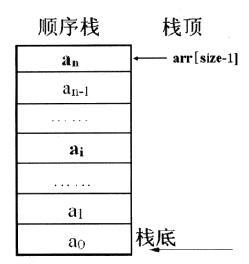
姓名：刘禹锡 学号：2256666 日期：2023年10月18日

1. **涉及数据结构和相关背景**

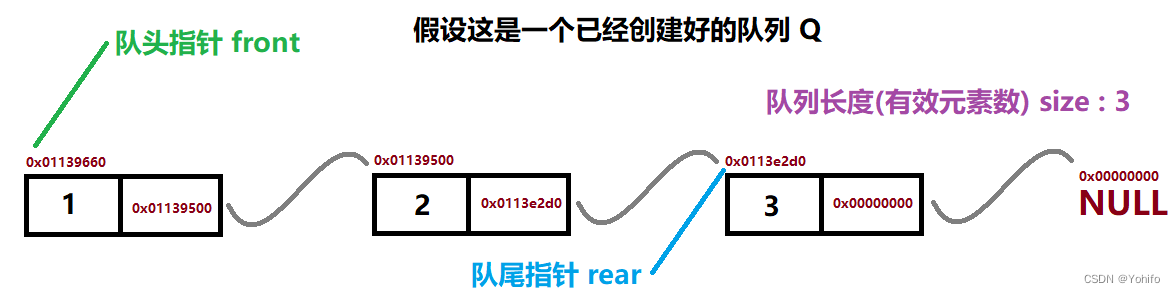
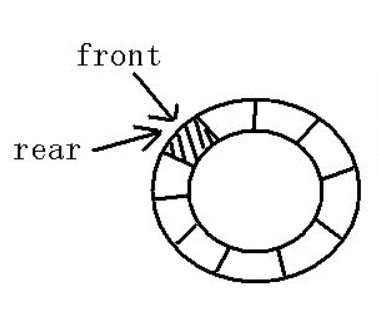
这次作业主要涉及了两个数据结构，栈和队列。

**栈**：最大特点是"后进先出"（LIFO，Last-In-First-Out），并且只可以操作栈顶元素。

栈通常用于解决需要按照相反顺序处理数据的问题。栈又可分为顺序栈（使用数组实现）和链栈（使用链表实现）。



**队列**：最大特点是"先进先出"（FIFO，First-In-First-Out），并且只可在队首弹出元素，在队尾加入元素。队列又可分为循环队列（队首队尾相接的顺序存储结构）和链队列（使用链表实现）。



**2. 实验内容**

**2.1 列车进站**

**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**

*/\*description：*

*由于入站列车满足后进先出，所以使用栈来模拟火车序列*

*\*/*

class Train *//火车序列*

{

private:

    char data[1000]; *//储存栈*

    int top; *//栈顶指针*

public:

    Train() { top = -1; } *//初始化栈顶指针*

    void push(char c) { *//入栈*

        data[++top] = c;

    }

    char pop() { *//出栈*

        return data[top--];

    }

    char peek() { *//取栈顶值*

        return data[top];

    }

    bool isEmpty() { *//判空*

        return top == -1;

    }

};

**2.1.4功能说明（函数、类）**

/\*

 \* @brief           判断火车是否可按照指定序列驶出

 \* @param   entrance 进栈序列

 \* @param   exit   出站序列

 \* @return           true / false

 \*/

bool canExit(const string& entrance, const string& exit)

{

    Train train;

    int entranceIndex = 0; *//初始化入栈序列索引*

    for (char c : exit){ *//c为出站序列索引*

        if (栈非空 && train.peek() == c)

            列车顶部出栈;

        else {

           while (入栈序列索引在合法范围内&&当前入栈序列元素与出站序列元素c不同)

            {

                train.push(entrance[entranceIndex]);

                入栈序列索引+1;

            }

            if (entranceIndex >= entrance.length()) {

                return false;

            }

            入栈序列索引+1;

        }

    }

    return true;

}

该函数使用栈来模拟火车进出站的过程。当遇到出站的字符时，首先检查栈顶元素是否与该字符匹配，如果匹配则将栈顶元素弹出；否则，将入站的字符加入栈中。通过这个过程，可以判断火车的进站顺序是否能够满足出站顺序。

时间复杂度：设m为出站序列的长度。在最坏情况下，每个元素都需要进行一次入栈操作和一次出栈操作。因此，总体时间复杂度为 O(m)。

空间复杂度：取决于栈的空间，O(m)

**2.1.5 调试分析（遇到的问题和解决方法）**

1. **边界操作**

忘记进行循环条件约束。比如应该加上对于入栈序列索引的判断，避免其超范围。

1. **栈顶指针的构造**

刚开始把栈顶指针实指和虚指搞混，出现错误。后来改成实指。

**2.1.6 总结和体会**

本道题主要考察栈的使用，栈最大的特点是”后进先出“，在本题中，由于车站是单通道，也符合先进后出的特点，因而适用栈。通过这个题，熟悉了栈的结构特点，以及一些基本的操作。

**2.2 布尔表达式**

**2.2.1 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）

**2.2.2 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。

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）。

**2.2.3 数据结构设计**

*/\*description：*

*参考课本上的算数运算，采用栈的结构来进行布尔表达式运算。相比课本，布尔表达式较为简便因为数值只有V or F.*

*\*/*

class Stack {

private:

    char data[100]; *//栈内元素*

    int top; *//栈顶指针*

public:

    Stack() : top(-1) {} *//初始化栈顶指针*

    void push(char c) { *//入栈*

        data[++top] = c;

    }

    char pop() { *//出栈*

        return data[top--];

    }

    char peek() { *//取栈顶元素*

        return data[top];

    }

    bool isEmpty() { *//判空*

        return top == -1;

    }

};

**2.2.4功能说明（函数、类）**

函数一

*/\**

\* @brief 用于返回运算符的优先级

\* @param op 运算符

\* @return 优先级 3>2>1

\*/

int getPrecedence(char op) {

    if (op == '|') return 1;

    if (op == '&') return 2;

    if (op == '!') return 3;

    return 0;

}

由于布尔表达式中有多种运算符，在计算时必须要按照优先级运算，而字符的比较不太方便，因此采用此方法将运算符的优先级返回。

函数二

*/\**

\* @brief 用于计算逻辑表达式

\* @param expression 运算符串

\* @return true or false

\*/

bool evaluateExpression(const char\* expression) {

Stack operandStack ,operatorStack; *// 创建两个栈，一个用于操作数，一个用于操作符*

while (expression[i] != '\0') { *// 从表达式的第一个字符开始，逐个处理字符*

char ch = expression[i];

if (ch == 空格){ 跳过 }

else if (ch == '(') {

operatorStack.push(ch); *//遇到左括号就入栈*

i++; }

else if (ch == 'V' || ch == 'F') {

将 'F' 转换为 0，将 'V' 转换为 1

继续处理连续的 'V' 或 'F' 字符 }

while (操作符栈非空 && 操作符栈顶为 非)

处理前置的逻辑非操作符 '!'

operandStack.push(t); *// 将结果压入操作数栈*

}

else if (ch == '|' || ch == '&') {

while (操作数栈非空 && 栈中优先级更高) { *// 处理具有较高优先级的操作符*

弹出两个操作数和一个操作符

*// 根据操作符计算结果并将其压入操作数栈*

if (oper == '&') { 与操作 }

else if (oper == '|') { 或操作 }

}

operatorStack.push(ch); *//操作符入栈*

i++;

}

else if (ch == '!') { *// 逻辑非 '!'入栈*

operatorStack.push(ch);

i++;

}

else if (ch == ')') { *// 处理右括号*

*// 处理与右括号匹配的操作符，直到遇到左括号*

while (操作符栈非空 && 未遇到 '(' ) {

char op1 && op2 = operandStack.pop(); *// 弹出两个操作数和一个操作符*

char oper = operatorStack.pop();

if (oper == '&') { 与操作 }

else if (oper == '|') { 或操作 }

}

operatorStack.pop(); *// 弹出左括号*

处理前置的逻辑非操作符 '!'

i++;

}

}

while (操作符栈非空) { *// 处理剩余的操作符*

弹出两个操作数和一个操作符

if (oper == '&') { 与操作 }

else if (oper == '|') { 或操作 }

}

return operandStack.peek() == 1;

}

这个函数用于计算逻辑表达式。通过两个栈（操作数栈和操作符栈）来管理表达式的计算过程，首先进行些前提操作，例如将‘V’‘F’转化为1和0，便于计算，并处理前置“!”，然后按照优先级顺序处理与非操作符，并将结果存放在操作数栈中，同时处理右括号与左括号，最后将剩余操作符运算完毕，返回结果。

时间复杂度：主要受表达式长度影响，记为n。输入表达式为O(n),在循环处理栈时，最坏下每个操作符都要操作，复杂度为O(n)，因此总体复杂度为O(n)。

空间复杂度：主要受栈的使用和表达式长度的影响，总体为O(n).

**2.2.5 调试分析（遇到的问题和解决方法）**

1. **数据结构设计有问题**

对于逻辑表达式的计算，应该如同算数操作一样，将操作数和操作符放在不同的栈中，所以应该开辟两个栈，operatorStack和oprandStack。

1. **测试数据中无论什么表达式，都输出F**

优先级运算有误，新建了一个函数，用于返回优先级，并在操作符入栈时进行优先级判断。

1. **爆栈**

第二个测试数据正常但是第三个爆栈，通过比较，猜测是因为之前的算法仅考虑了括号中至多一个操作符，以及进行运算后未及时把操作符（例如!）弹出。优化了右括号判断，以及及时将操作符出栈。

**2.2.6 总结和体会**

本道题主要考察栈的综合应用，在逻辑表达式计算时，需要把操作数和操作符分开计算，并且需要考虑操作符的优先级，以及括号的处理。体会到了，对于一些问题，还可以构造多个栈，彼此配合；来使用。

**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。

**2.3.2 基本要求**

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。

**2.3.3 数据结构设计**

*/\*description：*

*读题后发现，测试数据长度会很大，因此采用链表实现栈，同时由于要求索引，因而在栈节点中就保留下索引*

*\*/*

class Stack { *//链表栈*

private:

    struct Node *//栈节点*

    {

        int index; *//索引*

        Node \*next; *//指向下个节点*

        Node(int i) : index(i), next(NULL) {} *//初始化*

    };

    Node \*top; *//栈顶指针*

public:

    Stack() : top(NULL) {} *//初始化栈顶指针*

    void push(int index) {}*//入栈*

    void pop() {} *//出栈*

    bool empty() {} *//判空*

    int getTopIndex() {} *//获取索引值*

};

**2.3.4功能说明（函数、类）**

*/\**

\* @brief 用于返回最长匹配序列的长度与位置

\* @param s 用于传入序列

\* @return maxLen maxStart 最长匹配序列的长度与位置

\*/

pair<int, int> longestValidParentheses(string s){

Stack stk; *// 创建一个栈用于跟踪括号的索引*

int maxLen = 0，maxStart = 0; *// 用于记录最长有效括号序列的长度和起始位置*

int lastRight = -1; *// 用于记录上一个未匹配的右括号的位置*

int start = -1; *// 用于记录当前有效括号序列的起始位置*

for (int i = 0; i < s.size(); i++)

{

如果s[i]是左括号：

if (start == -1) *// 并且为起始左括号*

start = i; *// 记录当前有效括号序列的起始位置*

stk.push(i); *// 将左括号的索引入栈*

否则：

如果栈为空:

lastRight = i; *// 更新上一个未匹配的右括号位置*

start = -1; *// 重置当前有效括号序列的起始位置*

否则:

stk.pop(); *// 匹配一个右括号，弹出栈顶的左括号索引*

if (栈空){

int currentLen = i - lastRight;

if (currentLen > maxLen) { 更新最长有效括号序列的起始位置 }

}

else{

int currentLen = i - stk.getTopIndex();

if (currentLen > maxLen) { 更新最长有效括号序列的起始位置 }

}

}

return {maxLen, maxStart}; *// 返回最长有效括号序列的长度和起始位置*

}

这个函数用于找到最长子串的长度和位置。函数类型为pair，用于返回多个值。

首先，函数创建了一个名为stk的栈，用于跟踪括号的索引，以备后续匹配使用。同时，它初始化了一些变量来记录最长长度、位置，以及上一次未匹配右括号的位置。接下来，函数遍历序列 s，将特定的左括号索引入栈。当遇到右括号时，函数检查栈中是否有可以与之匹配的左括号，如果有匹配的左括号，则将其弹出栈，并更新相关的变量。通过这个过程，函数最终能够找到最长子串的长度和位置。

时间复杂度：主要时间开销为遍历字符串s，其他大部分操作，如push，pop都是常数时间，因此时间复杂度为O(n).

空间复杂度：主要空间开销是用来跟踪括号索引的栈stk，最坏情况下，s中所有括号都是匹配的，因此站的大小会达到字符串长O(n/2)，其他变量只占用常数空间，因此空间复杂度为O(n)。

**2.3.5 调试分析（遇到的问题和解决方法）**

1. **没有存索引位置**

第一版只输出最大长度，没有起始位置，原因是没有如同最大长度一样同步更新最大子串的起始位置。修改方法，添加了一个变量maxStart，跟随maxLen同步更新。

1. **数组越界**

没有认真读题，测试数据可以达到很大的规模，使用静态栈不仅效率慢，而且不可以动态增加空间，因此考虑使用链表模拟栈，可以适应不同规模的数据，并且提高了效率。

1. **非法右括号的处理**

刚开始因为全是合法的一对一括号对，所以没有考虑非法右括号的处理，但是实际测试数据存在非法的右括号。

修改方法：在遇到右括号时，先检查stk是否为空，如果为空，则表示没有与当前右括号匹配的左括号，执行以下操作：更新lastRight为当前右括号位置’i’用于记录上一个未匹配的右括号，同时重置start为-1，表示当前有效括号序列的起始位置无效。

**2.3.6 总结和体会**

本道题考察了栈的链表形式，链表栈相比于静态栈有以下优点：动态大小，不易溢出，但是空间开销大，时间复杂度较高，这是因为链表通过指针进行链接。同时，需要注意，链表栈涉及了动态内存申请，需要在出栈操作中将申请的内存释放回收。

通过这道题，我更深刻地理解了栈的不同形式的优点与不足，以及如何根据问题选择适当的数据结构。同时，学会了要对特殊情况进行必要的考虑。

**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

**2.4.3 数据结构设计**

*/\*description：*

*创建链表节点，作为构造链表队列的组成。*

*\*/*

struct Node

{

    int data; *//队列节点的数据*

    Node\* next; *//指向下一个队列节点*

};

*/\*description：*

*创建链表队列，用于*

*\*/*

class Queue {

private:

    Node\* front; *//队首指针*

    Node\* rear; *//队尾指针*

public:

    Queue() : front(NULL), rear(NULL) {} *//初始化*

    void enqueue(int value) {} *//入队操作*

    int dequeue() {}  *//出队操作*

    bool isEmpty() {}  *//判空*

};

**2.4.4功能说明（函数、类）**

**函数一 BFS算法**

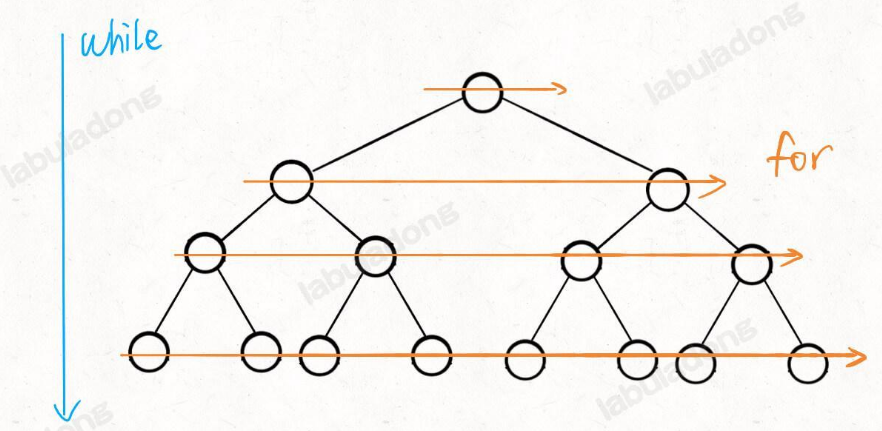


图 1 BFS算法流程图

由上图可以知道，BFS算法可以由一个中心点对地图进行扩散性筛查，更符合我们本题计算连通域的思路。

**实现方法：**

由于BFS会在每个岔路口首先储存所有岔道的特性，通常BFS会和Queue（先进先出）一同使用。步骤如下：

1. 将根节点加入到Queue中
2. 使用while循环，当Queue为空时，结束循环。
3. 将Queue中的节点依次poll出来检查。
4. 如果没有找到要找的值，就将poll出来的节点的子节点再加入到Queue中（如果没有子节点就不加）。

**常见使用场景：**

1. 连通块问题（Connected Component）

连通块问题是指通过一个点找到图中所联通的所有点的问题。这类问题我们可以用经典的BFS加上HashSet来解决。

1. 分层遍历（Level Order Traversal）

分层遍历问题要求我们在图中按层次进行遍历，常见的分层遍历问题有简单图的最短路径问题(Simple Graph Shortest Path) *Note: 简单图指的是每条边的距离相等，或者都视作长度为单位1的图。*

1. 拓扑排序（Topological Sorting）

**本题中算法实现：**

*/\**

\* @brief 以BFS方式遍历矩阵中的连通区域

\* @param i j 起始点坐标

\* @param m n 矩阵行数和列数

\* @param visited 用于记录已访问节点的二维布尔数组

\* @param matrix 表示矩阵的二维整数数组

\* @param queue 存储待处理节点的队列

\* @return 采用指针形式修改实参，返回空

\*/

void bfs(int i, int j, int m, int n, bool \*\*visited, int \*\*matrix, Queue &queue){

    int dx[] = {-1, 1, 0, 0};

    int dy[] = {0, 0, -1, 1}; *//定义偏移量*

    visited[i][j] = true; *//标记起点为visited*

    queue.enqueue(i \* n + j); *//将起点入队*

    while (队列非空){

        int cell = queue.dequeue();

        int x = cell / n;int y = cell % n; *//从出队元素获取坐标*

        for (int k = 0; k < 4; k++){ *//遍历相邻的四个端点*

            int newX = x + dx[k];int newY = y + dy[k];*//根据偏移量表示四个端点坐标*

            if (四个端点在地图范围内&& 端点未被访问 && 地图上该点为1){

                visited[newX][newY] = true; *//标记该端点*

                queue.enqueue(newX \* n + newY); *//将新标记的端点入队*

            }

        }

    }

}

这个函数整个逻辑思路是使用BFS算法从起始点出发，不断遍历矩阵中与当前节点相邻且未被访问的节点，直到无法再访问到新的节点为止，这个过程用于下面计算矩阵中的连通区域。

时间复杂度：遍历矩阵中的每个节点，因此时间复杂度是O(m\*n+4m\*n)其中m为行数，n 为列数，简化为O(N)，其中N代表节点个数。

空间复杂度：主要为visited矩阵和queue的空间占用，总体为O(m\*n)

**函数二 计算连通域**

*/\**

\* @brief 以BFS方式计算连通域个数

\* @param m n 矩阵行数和列数

\* @param matrix 表示矩阵的二维整数数组

\* @return count 返回连通域个数

\*/

int countRegions(int \*\*matrix, int m, int n)

{

*//动态申请一个二维bool数组用来标记节点是否被访问过，避免回走*

bool \*\*visited = new bool \*[m];

int count = 0; *// 用于计数连通块的数量*

Queue queue; *// 创建一个队列用于BFS*

for (int i = 0; i < m; ++i)

初始化 visited 数组，将所有节点标记为未访问

*// 遍历矩阵中的每个节点*

for (int i = 0; i < m; i++){

for (int j = 0; j < n; j++){

*// 如果当前节点是一个已访问的连通块的一部分，跳过*

if (节点为1并且未被访问){

if (当前节点位于矩阵的边界上)

continue; *// 不处理*

*// 使用BFS从当前节点开始探索一个新的连通块*

bfs(i, j, m, n, visited, matrix, queue);

count++; *// 增加连通块的计数*

}

}

}

释放 visited 数组的内存

return count;

}

这段代码的主要目标是通过BFS算法计算给定矩阵中的连通块数量。首先，通过一个二维bool数组visited标记已访问的节点，以避免重复访问。然后，通过遍历矩阵的每个节点，检查是否存在未访问的连通块的起始点。对于每个发现的起始点，使用BFS算法来遍历并标记整个连通块，并增加计数以记录发现的连通块数量。

时间复杂度：初始化visited矩阵，复杂度为O(m\*n)，对于每个连通块，BFS的时间复杂度为O(N + 4N)，其中N为节点的数量。遍历每个连通块造成count \* (N + 4N) 的时间复杂度。因此，总的时间复杂度为 O(m \* n + count \* N)，其中 count 通常远小于 m \* n，因此最终可以近似表示为 O(m \* n)。

空间复杂度：主要为visited矩阵和queue的空间占用，总体为O(m\*n)

**2.4.5 调试分析（遇到的问题和解决方法）**

1. **区域计算有误**

题目要求：位于矩阵边缘的不算做区域。第一版未考虑到。修改方法：在查找连通块中心时，如果遇到位于边界的点，continue跳过。

1. **内存越界**

尝试过使用循环队列，但是发现测试数据中后几个矩阵规模很大，使用静态循环队列必定有问题。

修改方法：修改成链表队列，优点是可以动态修改队列大小。

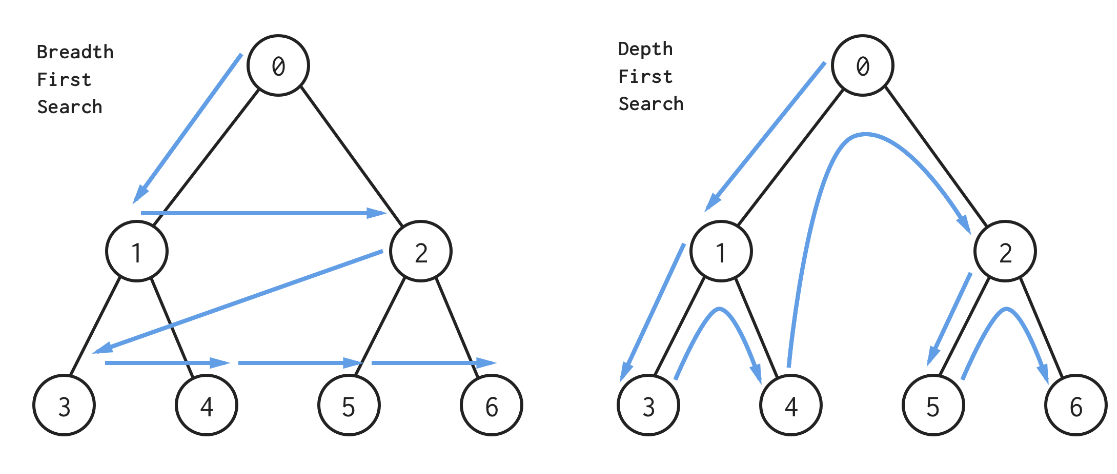
1. **死循环或超时**

原因是因为在BFS算法中没有标记端点是否走过，有可能陷入死循环或重复访问。

修改方法：申请一个bool二维数组visited用于标记地图上各个点有没有走过。

1. **使用DFS算法造成超时及爆栈**

原因是因为DFS算法通常是递归调用来遍历矩阵，每次递归调用都会将当前节点入栈，当矩阵规模很大时，递归深度很深，导致栈溢出。



修改方法：改用BFS算法配合队列来管理节点遍历顺序，因此不会出现递归深度问题。而且BFS通常在许多情况下，尤其是在找到最短路径或最短步骤的问题中，是更为有效的选择，因为它会逐层遍历，保证首次到达目标的路径就是最短的。

**2.4.6 总结和体会**

本道题考察了队列的使用，队列是一种先进先出，只可在头尾进行操作的数据结构。同时，也学习了BFS和DFS算法，BFS相较于DFS盲目搜索，更有一定的层次性，BFS按层搜索的方式可以避免栈溢出，类似于辐射式扩散，对于图像，矩阵处理有其优势。

一个好的算法要求时间复杂度和空间复杂度低，如何选择适当的算法，我还需要继续深入学习。

**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：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**

*/\*description：*

*使用双向链表构成队列的节点*

*\*/*

class Node { *//双向链表节点*

public:

    int value; *//队列节点的值*

    Node\* next; *//指向下一个节点*

    Node\* prev; *//指向前一个节点*

    Node(int val) : value(val), next(NULL), prev(NULL) {} *//初始化*

};

*/\*description：*

*\* MaxQueue 是一个基于双向链表的队列数据结构，支持入队、出队和最大值查询操作。*

*\*/*

class MaxQueue {

private:

int n; *// 队列的容量*

int count; *// 队列中当前元素数量*

Node\* front; *// 队首指针*

Node\* rear; *// 队尾指针*

Node\* maxNode; *// 用于跟踪当前队列中的最大值节点*

public:

MaxQueue(int capacity) : n(capacity), count(0), front(NULL), rear(NULL), maxNode(NULL) {}

void enqueue(int value) {} *//入队*

int dequeue() {} *//出队*

int getMax() {} *//获取最大值*

bool isFull() {} *//判满*

bool isEmpty() {} *//判空*

};

**2.5.4功能说明（函数、类）**

函数一 入队

*/\**

\* @brief 将一个元素入队，同时维护最大值

\* @param value 待入队的元素值

\* @return 无返回值

\*/

void enqueue(int value) {

判满

Node\* newNode = new Node(value);*// 创建一个新的节点，存储待入队的元素值*

if (!front) {

front=rear=maxNode=newNode;*//如果队列为空，将新节点设置为队列的唯一节点*

}

else {

rear->next = newNode;*// 队列非空，将新节点添加到队列尾部*

newNode->prev = rear;*//同时修改前驱后驱指针*

rear = newNode;

if (value >= maxNode->value) {

maxNode=newNode;*//如果新元素的值大于等于当前最大值，更新最大值节点*

}

}

*// 增加队列中的元素计数*

count++;

}

这个函数通过双向链表的操作将新元素加入到队尾，并在每一次入队时，追踪元素值，如果超过最大元素，则更新最大元素。

时间复杂度：创建新节点时间复杂度为O(1)，因为只涉及分配内存与初始化。添加新节点也为O(1)，只涉及指针修改操作。更新最大值也为O(1)，因为有现成的最大元素节点，同其比较一次即可。所以总体时间复杂度为O(1)。

空间复杂度：主要涉及新节点的内存分配和存储额外指针，总体空间复杂度为O(1)

函数二 出队

*/\**

\* @brief 出队操作，返回出队的元素值

\* @return value 出队的元素值

\*/

int dequeue() {

判空

Node\* removed = front; *// 从队列的前端移除一个节点*

int value = removed->value; *// 获取被出队节点的元素值并保存在变量value中*

front = front->next; *// 更新队列的front指针，使其指向下一个节点*

if (被出队的节点removed是队列中的最大值节点maxNode) {

maxNode = front; *// 先假定当前front为最大值节点*

Node\* current = front;

while (current) { *// 找到新的最大值节点，并将其赋给maxNode*

if (current->value >= maxNode->value)

maxNode = current;

current = current->next;

}

}

delete removed; count--;*// 删除被出队的节点以释放内存，队列元素计数--*

return value; *// 返回被出队的元素值*

}

这个函数通过通过双向链表操作将队首元素出队，同时，由于需要维护最大值节点，所以如果出队元素恰为出队元素时，需要重新遍历队列找出最大元素。

时间复杂度：一般情况下，只需要删除节点即可，时间复杂度为O(1)。当出队元素恰为出队元素时，需要重新遍历队列找出最大元素，时间复杂度为O(n)。

空间复杂度：主要受节点的内存消耗影响，总体空间复杂度为O(1)

函数三 其他操作

*/\**

\* @brief 包括返回最大值，判空判满操作

\*/

int getMax() { //返回最大值节点元素

  判空

    return maxNode->value;

}

bool isFull() { //判满

     return count == n;

}

bool isEmpty() { //判空

     return front == NULL;

}

由于队列中已存在储存最大元素值的节点，因此要求返回最大元素，只需要返回最大值元素节点的值即可；其他判满判空，也只需要比较指针是否非空，元素是否超限即可。

因此三个的时间复杂度都为O(1)

**2.5.5 调试分析（遇到的问题和解决方法）**

1. **超时TimeOut**

第一版代码在矩阵大时超时，这是因为使用了countElements函数用于计算队列中元素数量时，该函数的时间复杂度为O(n)。在每次判断队列是否已满时，都需要调用countElements函数，导致时间复杂度较高。超时的部分代码如图2

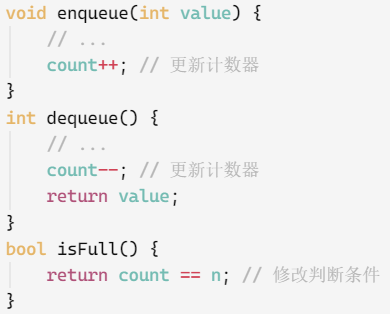
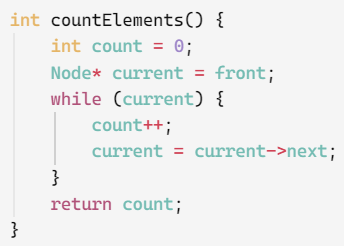


图 2 超时代码 图 3 修改代码

解决办法：如图3，在队列中维护一个计数器(count)，每次enqueue和dequeue操作时，根据情况更新计数器的值。这样，在判断队列是否已满时，只需要比较计数器的值与队列容量n的关系即可，时间复杂度为O(1)。

1. **未考虑删除节点为最大值节点**

第一版代码在删除时，没有考虑到删除的节点可能为最大值元素，最大值节点应当在出队操作时进行更新。解决办法：在出队时，将出队元素与最大值元素比较，如果相等，应当重新遍历队列，更新最大值。

**2.5.6 总结和体会**

本道题主要收获就是，深刻认识到了算法的重要性，本着封装函数的思路，想要把一些语句都封装成函数，但是忘记了函数调用会消耗时间，同时，一些简单功能，不需要函数来实现，例如上面的cout计数器，使用函数反复调用反而超时，不如使用计数器，在需要时进行更新。

**3. 实验总结**

在本次的作业中，我学习了栈和队列两种数据结构，并进行了一些练习。通过这些题目，我更好地认识了栈（后进先出）和队列（先进先出）的特点，以及它们的适用情况，并了解到使用时应当注意的细节。

同时，我也学习到了两个新的搜索算法，DFS和BFS，并且在实际体验中比较了两者的区别。BFS更适用于本次作业中的辐射式搜索。

最后，我也意识到一个好的算法要求时间复杂度和空间复杂度低，如何选择适当的算法，我还需要继续深入学习。