作业 HW2 实验报告

姓名：刘彦 学号：2352018 日期：2024 年 10 月 19 日

1. 涉及数据结构和相关背景

栈是一种遵循后进先出（LIFO）原则的线性表，允许在一端进行插入和删除操作。

其基本操作包括**初始化、判空、压栈（push）、弹栈（pop）、获取栈顶元素（gettop）、清空栈**以及**返回栈内元素的数量**。栈可以分为**顺序栈**和**链栈**。

- 顺序栈使用固定大小的数组实现，需预先定义容量，容易出现上溢（栈满时尝试压栈）和下溢（栈空时尝试弹栈）的问题。
- 链栈则采用链表结构，允许动态扩展，无需预定义容量，栈顶指针指向链表的头部，操作更加灵活。

队列是一种遵循先进先出（FIFO）原则的线性表，允许在一端插入元素，在另一端删除元素。

其基本操作同样包括**初始化、销毁队列、入队（enqueue）、出队（dequeue）、判空**以及**获取队头元素**。队列也分为**顺序队列、循环队列**和**链队列**。

- 顺序队列通过一维数组实现，队头和队尾指针管理元素，当头尾指针相等时表示队列为空，队尾指针达到最大长度时表示队列已满。
- 循环队列则将数组视为首尾相接，利用取模运算来管理指针，减少空间浪费，并需留一个空位来区分队满和队空。
- 链队列则基于链表实现，动态扩展，队头和队尾各有指针，便于高效管理。

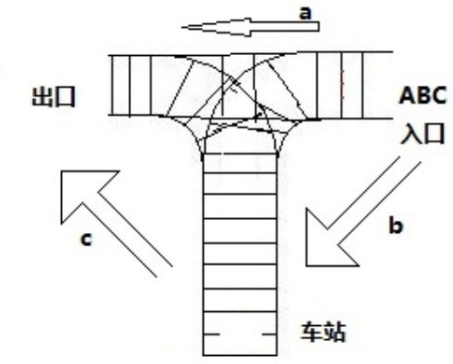
栈和队列广泛应用于计算机科学和编程中，例如栈用于函数调用管理、表达式求值等，而队列常用于任务调度、打印任务管理等。在实现时，注意处理溢出和下溢情况，动态数据结构在元素数量不确定时通常更为高效。

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

```
struct Stack {
    char arr[1000];
    int top;
    Stack() : top(-1) {} // 构造函数初始化栈顶指针
    void push(char c) {
        arr[++top] = c; // 压栈
    }
    void pop() {
        if (top >= 0)
            top--; // 弹栈
    }
    char topElement() {
        return arr[top]; // 获取栈顶元素
    }
    bool isEmpty() {
        return top < 0; // 检查栈是否为空
    }
};
```

2.1.4 功能说明（函数、类）

```
/**
 * @brief           检查一个字符串是否为另一个字符串的子序列
 * @param   stand   标准字符串
 * @param   s       待检测字符串
 * @return          如果 s 是 stand 的子序列，返回"yes"；否则返回"no"
 */
int main() {
    char stand[1000], s[1000];
    cin >> stand; // 读取标准字符串
    while (cin >> s) { // 读取待检测字符串
        Stack st; // 创建栈对象
        int ptr = 0; // 用于遍历待检测字符串
        for (i 从 0 到 stand[i] != '\0') {
            if (ptr < strlen(s) && stand[i] == s[ptr]) {
                ptr++;
            } else  压栈
```

```
            while (!st.isEmpty() && ptr < strlen(s) && st.topElement() ==
s[ptr])  弹栈  ptr++;
        }
        // 处理完标准字符串后，检查栈中剩余字符
        while (!st.isEmpty() && ptr < strlen(s) && st.topElement() ==
s[ptr])  弹栈  ptr++;
        cout << (ptr == strlen(s) ? "yes" : "no") << endl;
    }
    return 0;
}
```

2.1.5 调试分析（遇到的问题和解决方法）

2.1.5.1 栈顶元素比较不当

在比较当前字符和栈顶元素时，如果不相等，就直接将当前字符压入栈中，忽略了可能的出栈操作，导致遗漏了许多情况。最终在入栈之前，先与栈顶元素进行比较，如果相等则出栈，只有在不相等且栈为空时才入栈。

2.1.5.2 栈下溢错误

在弹出操作中，如果栈为空，程序可能会试图访问栈顶元素，从而导致下溢错误。最终在 pop() 方法中添加判断，确保在弹栈前栈不为空。

```
  void pop() {
        if (top >= 0)
            top--; // 弹栈
    }
```

2.1.5.3 逻辑错误导致不正确的输出：

在一开始的代码中，我在处理栈中剩余字符时，未能正确更新指针 ptr，导致结果输出错误。最终在检查栈中剩余元素时，确保同时更新 ptr。

```
// 处理完标准字符串后，检查栈中剩余字符
        while (!st.isEmpty() && ptr < strlen(s) && st.topElement() ==
s[ptr]) {
            st.pop(); // 弹栈
            ptr++;
        }
```

2.1.6 总结和体会

深入掌握了栈的后进先出特性，增强了对栈操作（压栈和弹栈）的理解。学会了通过栈判断字符顺序的问题，提高了字符串操作的技巧。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。

合理使用栈操作判断何时入栈、出栈是核心，需细致设计。在遍历待检测字符串时，首先需要判断当前字符是否与栈顶元素相等。如果相等，立即出栈，因为这个字符已经在正确的位置。当当前字符与栈顶不相等时，才能考虑入栈操作。此时需要确保将当前字符压入栈中，只有在栈为空或栈顶元素不等于当前字符的情况下才可以入栈。每次入栈后，应立即检查栈顶元素是否与待检测字符串的当前字符相等。如果相等，则执行出栈操作，并更新指针，继续检查下一个字符。在入栈后，如果栈顶元素可以继续匹配后续字符，应保持循环检查并出栈，直到不再匹配或栈为空。

2.2 最长子串

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

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

```
struct Stack {
    int arr[N]; // 栈的数组  int top;    // 栈顶指针
    Stack() : top(-1) {} // 构造函数，初始化栈顶为 -1
    void push(int value) { // 入栈
        arr[++top] = value;
    }
    void pop() { // 出栈
        if (top >= 0)
            --top;
    }
    int peek() { // 获取栈顶元素
        return arr[top];
    }
    bool isEmpty() { // 判断栈是否为空
        return top == -1;
    }
};
struct Result {
    int maxLength;   // 最长有效括号的长度
    int startIndex;  // 最长有效括号的起始索引
};
```

2.2.4 功能说明（函数、类）

```
/**
 * @brief            找出最长有效括号子串的起始索引和最大长度
 * @param   s        输入的字符数组，包含括号
 * @return           输出最大有效括号子串的长度和起始索引
 */
int main() {
    char s[N];
    cin >> s; // 读取字符数组
    int n = strlen(s); // 获取字符数组的长度
    if (长度小于 1) {
        cout << 0 << " " << 0;
        return 0;
    }
    Stack stk; // 创建栈
    stk.push(-1); // 初始化栈，先放一个 -1
    Result res = {0, 0}; // 初始化结果结构体
    for (int i = 0; s[i] != '\0'; ++i) { // 使用 '\0' 判断结束
        if (s[i] == '(') 入栈当前索引
        else {
            stk.pop(); // 出栈
            if (栈为空) 压入当前索引
            } else if (找到了一个更大的有效范围)  更新最大长度和起始索引
        }
    }
    cout << res.maxLength << " " << res.startIndex << endl;
    return 0;
}
```

2.2.5 调试分析（遇到的问题和解决方法）

2.2.5.1 起始索引计算错误

最初在更新最长有效括号的长度时，起始索引并没有正确更新，导致输出的起始位置错误。解决方法是在更新 `maxLength` 的同时，也更新 `startIndex` 为栈顶元素的下一个索引。

```
if (currentLength > maxLength) {
    maxLength = currentLength; // 更新最大长度
    // startIndex = stk.topElement() + 1; // 这行缺失，导致起始索引不正确
}
```

2.2.5.2 无效括号处理

最初的设计栈在处理右括号时未能正确更新无效括号位置，可能导致计算错误。解决方法是确保每次栈为空时，都将当前索引压入栈作为新的无效位置。

2.2.5.3 字符串长度处理

对于长度为 0 的字符串处理正常，但输入过长时，栈可能会溢出。解决方法是在压栈前增加栈容量检查，确保不会超出数组边界。

```
void push(int value) {
    arr[++top] = value; // 没有检查 top 是否超过 N-1
}
```

2.2.6 总结和体会

在这道题目中，我使用栈的解决方案具有 O(n)的时间复杂度，适合处理大规模输入。通过解决这个有效括号的问题，我对栈的使用和管理有了更深的理解。栈的后进先出（LIFO）特性在处理嵌套结构（如括号）时非常有用。应当确保在每次更新最大长度时也更新起始索引，容易被忽略。尤其是在复杂条件下，状态的同步是非常重要的。对于空字符串、只有一个括号或连续多个括号的情况，处理不当可能导致错误的输出。

2.3 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）。

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

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。

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

```
struct SqStack {
private:
    char* base;
    char* top;
    int stacksize;
public:
    SqStack();                  // 构造空栈
    ~SqStack();                 // 销毁已有的栈
    int ClearStack();           // 把现有栈置空栈
    int Top(char& e);           // 取栈顶元素
    int Pop(char& e);           // 弹出栈顶元素
    int Push(char e);           // 新元素入栈
    bool StackEmpty();          // 是否为空栈
};
```

2.3.4 功能说明（函数、类）

根据运算符对值栈进行操作的函数

```
/**
 * @brief           对运算符栈进行操作
 * @param Value      布尔值栈，存储'V'(true)或'F'(false)
 * @param Operator   运算符栈，存储逻辑运算符
 */
void Operate(SqStack& Value, SqStack& Operator) {
    char v1, v2, o; // 声明变量 v1, v2 为操作数，o 为运算符
    Operator.Pop(o); // 弹出栈顶运算符
    switch (o) {
        case '!': // 处理逻辑非运算
            弹出一个操作数，如果为 'V' 则压入 'F'，反之亦然
            break;
        case '&': // 处理逻辑与运算
            弹出两个操作数，两个都是 'V' 时压入 'V'
            break;
        case '|': // 处理逻辑或运算
            弹出两个操作数，只要有一个是 'V' 就压入 'V'
            break;
    }
}
```

根据输入的运算符更新值栈和运算符栈的函数

```
/**
 * @brief           处理输入的运算符
 * @param ch         输入的字符
 * @param Value      布尔值栈
 * @param Operator   运算符栈
 */
void processOperator(char ch, SqStack& Value, SqStack& Operator) {
    char get; // 用于临时存储栈顶运算符
    switch (ch) {
        case '(': // 遇到左括号时直接压入运算符栈
            Operator.Push(ch);
            break;
        case ')': // 遇到右括号时处理运算符
            弹出栈内运算符直到遇到左括号，弹出左括号
            break;
        case '!': // 处理逻辑非运算符
            Operator.Push(ch); // 压入运算符栈
            break;
        case '&': // 处理逻辑与运算符
            弹出栈内优先级高于或等于 '&' 的运算符
            Operator.Push(ch); // 压入当前运算符
            break;
```

```
        case '|': // 处理逻辑或运算符
            弹出栈内优先级高于或等于 '|' 的运算符
            Operator.Push(ch); // 压入当前运算符
            break;
    }
}
```

main 函数中关键部分如下

```
while ((ch = getchar()) != EOF) { // 逐字符读取输入直到 EOF
        if (遇到换行符) {
            输出当前表达式编号
            // 清空运算符栈，进行计算
            while (栈不空)  执行所有剩余运算
            char Answer; // 存储最终结果
            Value.Top(Answer); // 获取值栈顶的结果
            cout << Answer << endl; // 输出结果
            Value.ClearStack(); // 清空值栈以准备下一个表达式
            如果是操作数 'F' 或 'V'，压入值栈，否则调用函数处理运算符
        }
    }
int temp = !stshu.pop();
```

2.3.5 调试分析（遇到的问题和解决方法）

2.3.5.1 逻辑非（!）运算的实现

原本的设计在 doMath 函数中，int temp = !stshu.pop(); 这段代码会返回布尔值（0 或 1），而实际上 temp 应该是从栈中取出的布尔值。

```
if (op == '!') {
    int temp = !stshu.pop();
    stshu.push(temp);
}
```

最后我将将逻辑非的实现修改为 temp = stshu.pop();，然后再根据 temp 的值决定压入栈中的值。

2.3.5.2 优先级处理不准确

原本的设计在处理运算符时，如果当前运算符的优先级与栈顶运算符相同，可能会导致错误的顺序。应确保按照更严格的优先级规则执行。最终经检查将条件改为 >，以确保只有当前运算符的优先级高于栈顶运算符时才将其压入栈中。

```
while (!operators.empty() && precedence(s[i]) <= precedence(operators.top())) {
    char op = operators.pop();
    doMath(op, values);
}
```

2.3.6 总结和体会

通过实现这个布尔表达式求值的程序，我加深了对栈的使用和运算符优先级的理解。学习如何使用两个栈分别处理操作数和运算符。理解运算符的优先级和结合性是一个关键点，

特别是在复杂表达式中，如何正确处理相同优先级的运算符非常重要。需要处理栈空的情况时，以及在读取表达式时确保括号匹配和字符有效性，避免运行时错误。

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。

对于所有数据，0<=n,m<=1000

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

```
// 链表的一个节点
struct QNode {
    int data[2]; // 使用数组存储坐标 {x, y}
    QNode* next; // 指向下一个节点的指针
};
class myQueue {
private:
    QNode* front; // 队列头指针
    QNode* rear;  // 队列尾指针
public:
    myQueue();                  // 构造一个空队列
    ~myQueue();                 // 销毁队列
    int in_queue(int x, int y);    // 在队尾插入元素
    int out_queue(int& x, int& y);  // 弹出队首的元素
    int get_head(int& x, int& y);  // 取队首的元素（不弹出）
    bool if_empty();             // 判断队列是否为空
};
bool map[1000][1000]; // 存储地图的二维数组
bool vis[1000][1000]; // 存储访问标记的二维数组
```

2.4.4 功能说明（函数、类）

处理邻居坐标的函数

```
/**
 * @brief          处理邻居坐标的函数
 * @param newx     新的 x 坐标
 * @param newy     新的 y 坐标
 * @param map      地图数组，表示可走的位置为 true
 * @param vis      访问数组，表示已访问的位置为 true
```

```
 * @param Queue     队列，用于 BFS
 */
void processNeighbor(int newx, int newy, bool map[1000][1000], bool
vis[1000][1000], myQueue& Queue) {
    // 检查坐标是否在有效范围内，且该位置未被访问
    if (newx >= 0 && newx < 1000 && newy >= 0 && newy < 1000 &&
map[newx][newy] && !vis[newx][newy]) {
        vis[newx][newy] = true; // 标记为已访问
        Queue.in_queue(newx, newy); // 将新坐标入队
    }
}
```

广度优先搜索（BFS）函数

```
/**
 * @brief           广度优先搜索（BFS）函数
 * @param n         地图的行数
 * @param m         地图的列数
 * @param map       地图数组
 * @param vis       访问数组
 * @param x         起始 x 坐标
 * @param y         起始 y 坐标
 */
void bfs(int n, int m, bool map[1000][1000], bool vis[1000][1000], int x,
int y) {
    myQueue Queue; // 创建队列
    Queue.in_queue(x, y); // 将起始坐标入队
    vis[x][y] = true; // 标记起始坐标为已访问
    const int dx[] = { 0, 0, -1, 1 }; // x 方向的移动
    const int dy[] = { -1, 1, 0, 0 }; // y 方向的移动
    while (队列不为空) {
        int curx, cury;
        Queue.out_queue(curx, cury); // 获取当前坐标
        for (遍历四个方向) {
            int newx = curx + dx[i]; // 新的 x 坐标
            int newy = cury + dy[i]; // 新的 y 坐标
            processNeighbor(newx, newy, map, vis, Queue); // 处理邻居
        }
    }
}
```

2.4.5 调试分析（遇到的问题和解决方法）

2.4.5.1 队列实现的内存管理问题

一开始的代码中，在销毁队列时，没有删除 `rear` 节点，可能导致内存泄漏。

```
~LinkQueue() {
    while (front) {
```

```
        QNode* temp = front;
        front = front->next;
        delete temp;
    }
}
```

最终我在析构函数中确保删除 rear 节点，以及 front 节点的所有后续节点。

2.4.5.2 重复遍历的问题

第一次代码实现时，在队列未清空之前，可能会有重复的区域被访问，导致计数错误。

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
        if (mp[i][j] && !vis[i][j])
            // BFS ...
```

最终我增加了一个标记的逻辑确保在 BFS 完成后，正确地标记已经访问的节点，避免重复统计。

```
Queue.in_queue(x, y); // 将起始坐标入队
vis[x][y] = true; // 标记起始坐标为已访问
```

2.4.6 总结和体会

2.4.6.1 理解边界条件的重要性

明确了如何判断一个区域是否与矩阵的边界相连，理解“边缘联通”与“区域联通”的区别，确保能准确处理不同情况。题目要求确定的区域必须是“边缘联通”，这意味着所有单元都必须与矩阵的边界相接触。在判断联通时，特别注意全称量词的取反概念。题目可以理解为“对于所有在某区域内的单元，它们必须都在边缘”。用逻辑表达为：

$$\forall x, P(x)(\text{单元 } x \text{ 在边缘})$$

取反后，意思变为“存在至少一个单元不在边缘”，即：

$$\exists x, \neg P(x)$$

这意味着如果在 BFS 遍历过程中，发现有任何单元不在边缘，就可以判定该区域不满足“边缘联通”的条件。

2.4.6.2 增强了对图形遍历的理解

在使用 BFS 遍历时，强化了对队列操作的掌握，以及如何利用状态标志来跟踪区域的特性。并学会了将复杂问题拆分为简单的逻辑块，明确每个部分的责任。

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：

Enqueue(v)：v 入队 Dequeue()：使队首元素删除，并返回此元素 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。

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%

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：，分别是执行每次操作后的结果

2.5.3 数据结构设计

```
class MaxQueue {
private:
    int* mainQueue; // 主队列
    int* maxQueue;  // 辅助队列
    int front;      // 主队列头索引
    int rear;       // 主队列尾索引
    int maxFront;   // 辅助队列头索引
    int maxRear;    // 辅助队列尾索引
    int capacity;   // 队列容量
    int size;       // 当前元素数量
public:
    MaxQueue(int n);   // 构造函数
    ~MaxQueue();       // 析构函数
    void enqueue(int value); // 入队
    void dequeue();     // 出队
    void getMax();      // 获取最大值
    void printQueue();  // 打印队列
};
```

2.5.4 功能说明（函数、类）

构造函数

```
/**
 * @brief           构造函数，初始化最大值队列
 * @param n         队列的容量
 */
```

```
MaxQueue::MaxQueue(int n) {
    capacity = n; // 设置队列容量
    mainQueue = new int[capacity]; // 初始化主队列数组
    maxQueue = new int[capacity]; // 初始化辅助队列数组
    front = rear = 0; // 初始化队头和队尾指针
    maxFront = maxRear = 0; // 初始化辅助队列的队头和队尾指针
    size = 0; // 初始化队列大小
}
```

入队函数

```
/**
 * @brief           入队操作，将元素添加到队列
 * @param value     要入队的值
 */
void MaxQueue::enqueue(int value) {
    // 检查队列是否已满
    if (size >= capacity)  输出"Queue is Full"并退出函数
    mainQueue[rear] = value; // 将新元素添加到主队列
    rear = (rear + 1) % capacity; // 更新后指针，循环使用
    size++; // 增加队列大小
    // 更新辅助队列，保持其递减顺序
    while (maxRear > maxFront && maxQueue[maxRear - 1] < value)
        maxRear--; // 移除小于新值的元素
    maxQueue[maxRear++] = value; // 添加新元素到辅助队列
}
```

出队函数

```
/**
 * @brief           出队操作，移除队列的前端元素
 */
void MaxQueue::dequeue() {
    // 检查队列是否为空
    if (size == 0)  输出"Queue is Full"并退出函数
    // 记录出队的元素
    int value = mainQueue[front];
    front = (front + 1) % capacity; // 更新前指针，循环使用
    size--; // 减小队列大小
    // 如果出队的元素是当前最大值，移除它
    if (value == maxQueue[maxFront]) {
        maxFront++; // 更新最大值队列的前指针
    // 检查是否需要移除相同的最大值
        while (maxFront < maxRear && maxQueue[maxFront] == value)
            maxFront++; // 移除所有相同的最大值
    }
    输出出队的元素 value
}
```

2.5.5 调试分析（遇到的问题和解决方法）

2.5.5.1 删除最大值时导致错误

一开始的实现代码中，当最大值在队列的头部，而此时第二大值被移除的逻辑没有得到妥善处理。可能出现删除最大值的错误，错误代码主要集中在 dequeue()和 enqueue()方法中。

这个逻辑假设出队的值一定是最大值，如果队列中有相同的最大值，可能导致下一个最大值未被正确更新。

```
if (value == maxQueue[maxFront])
    maxFront++; // 更新最大值队列的前指针
```

在这里，如果入队的值是最大值，并且之前的最大值已被出队，可能导致辅助队列的管理不当。

```
while (maxRear > maxFront && maxQueue[maxRear - 1] < value) {
    maxRear--; // 移除小于新值的元素
}
```

最后修改主要是在添加新元素到辅助队列时，确保更新逻辑保持正确。

```
while (maxRear > maxFront && maxQueue[maxRear - 1] < value)
        maxRear--; // 移除小于新值的元素
    maxQueue[maxRear++] = value; // 添加新元素到辅助队列
```

在出队时，确保对相同最大值的处理。

```
  // 检查是否需要移除相同的最大值
while (maxFront < maxRear && maxQueue[maxFront] == value)
      maxFront++; // 移除所有相同的最大值
```

2.5.5.2 边界条件检查

一开始在入队和出队操作时，未正确处理队列满和空的边界条件，可能导致访问无效内存或程序崩溃。最终确保在队列满和队列空的情况下，程序能正确返回提示信息而不崩溃。

2.5.6 总结和体会

2.5.6.1 这段代码的时间复杂度分析

入队操作 (enqueue):最坏情况下，可能需要遍历 maxQueue 中的所有元素以维护其递减顺序，这个操作的时间复杂度为 O(n)。

出队操作 (dequeue):出队操作的时间复杂度为 O(1)，因为只涉及更新指针和条件检查。

获取最大值 (getMax):获取最大值的操作也只涉及指针的检查和返回，时间复杂度为 O(1)。

打印队列 (printQueue):遍历并打印队列中的所有元素，时间复杂度为 O(n)。

2.5.6.2

在实现这个最大值队列的过程中，我理解了如何利用辅助队列来高效地维护最大值。最大值队列的设计使我意识到在特定情况下，合理的选择数据结构可以显著提高操作的效率。

3. 实验总结

本次实验的目的是通过实现栈和队列这两种线性数据结构，解决一些实际问题，深入理解栈和队列的应用场景、时间复杂度以及各自的优缺点。

数据结构选择：

- 栈：遵循后进先出（LIFO）原则，适合于需要临时存储数据的场景。例如，函数调用管理、表达式求值等。栈的查找操作通常不如队列直接，但在特定应用中能提供有效的支持。
- 队列：遵循先进先出（FIFO）原则，适合于需要顺序处理数据的场景，如任务调度、广度优先搜索等。队列提供高效的入队和出队操作，尤其在需要保持处理顺序时非常有效。

注意事项：

- 栈：在实现栈时，确保入栈和出栈操作的正确性，特别是在栈满或栈空的情况下要有相应的处理机制。在动态栈的实现中，动态扩展数组的大小要谨慎，避免频繁的内存分配导致性能下降。
- 队列：实现队列时，要注意使用循环队列以避免数组的空间浪费。在链队列实现中，确保指针管理的正确性，避免内存泄漏。在出队时，需确保正确更新队列头指针。

实验收获：

- 操作选择的重要性：在解决实际问题时，选择合适的数据结构对操作效率至关重要。例如，在需要临时存储和回溯的场景中，栈是理想的选择；而在需要顺序处理任务的场景中，队列则显得尤为重要。
- 通过实践，掌握了栈和队列在实际算法中的应用，如深度优先搜索（DFS）和广度优先搜索（BFS），还有单调栈和循环队列（环形队列）的结构。这些算法在图和树的遍历中非常重要。
- 掌握了各类基本操作的时间复杂度，栈的入栈和出栈操作时间复杂度为 O(1)，而队列的入队和出队操作也是 O(1)。这强调了在需要频繁插入和删除的应用中，选择合适的结构可以提升效率。