**作业HW2实验报告**

日期：2024年10月18日

1. **涉及数据结构和相关背景**

**1.1 栈**

* 后进先出LIFO
* 只能在表的一端(栈顶)进行插入和删除操作
* 允许插入和删除的一端为栈顶(top)，另一端为栈底(bottom)
* 基本操作：初始化、判空、push、pop、gettop、清空、返回长

**1.1.1 顺序栈**

* 需要预先定义栈的容量
* 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“

**1.1.2 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：无空间可动态申请时

**1.2 队列**

* 先进后出FIFO(FirstInFirstOut)
* 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)。

**1.2.1 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

* + 1. **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

**1.2.3 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，可用取模运算实现
* 队头指针加1:Q.front=(Q.front+1)%MAXSIZE

队尾指针加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
* 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”的办法：
* 计数器
* 标志变量

**1.2.4 非循环队列**

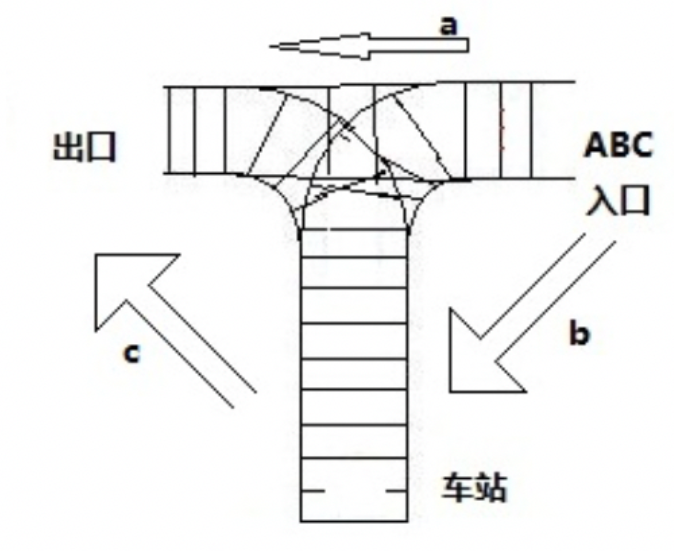
* 关键：修改队尾/队头指针Q.rear=Q.rear+1;Q.front=Q.front+1;
* 队空条件：Q.front=Q.rear
* 队满条件：Q.rear=MAXQSIZE
* 长度：Q.rear-Q.front

**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**

为了模拟列车的进站与出站过程，使用栈数据结构存储暂时待出站的列车。

// 栈结构用于临时存储等待出站的列车

class Stack {

private:

char\* stackArray; // 存储栈元素的数组

int top; // 栈顶指针

int capacity; // 栈的容量

public:

Stack(int size) {

stackArray = new char[size];

top = -1; // 初始化栈顶指针

capacity = size; // 设置栈的容量

}

void push(char item); // 压入元素

char pop(); // 弹出元素

char peek(); // 获取栈顶元素

bool isEmpty(); // 判断栈是否为空

~Stack(); // 析构函数释放内存

};

**2.1.4功能说明（函数、类）**

以下是主要的功能实现，采用伪代码形式展示关键逻辑

/\*\*

\* @brief 主程序逻辑

\* @param the\_input\_stack 入站序列

\* @param the\_out\_stack 当前的出站序列

\* @return 是否能够生成出站序列

\*/

function main() {

read the\_input\_stack // 读取入站序列

while (there are more out stacks) {

read the\_out\_stack // 读取出站序列

initialize temporary stack // 初始化临时栈

initialize pointers for input and output

while (input not finished and output not finished) {

if (input matches output) {

move to next input and output

}

else if (top of stack matches output) {

pop from stack and move to next output

}

else {

push current input onto stack

move to next input

}

}

// 检查栈中剩余元素是否可以按顺序出站

while (stack is not empty) {

if (top of stack does not match current output) {

output "no"

break

}

else {

pop from stack

move to next output

}

}

output "yes" // 如果所有元素匹配则输出yes

}

}

**2.1.5调试分析（遇到的问题和解决方法）**

* 在第一次尝试中，我采用了一种不同的思路：遍历每一个元素，如果当前元素与入栈序列中的元素不同，那么它之前的所有元素必须已经按照逆序出栈。因为当一个元素不按顺序出栈时，说明其前面的部分元素已经被入栈，出栈时它们必须是逆序出栈的。这个思路听起来可行，但在实际操作中只通过了一部分测试点。
* 问题的关键在于，如何处理入栈序列和出栈序列不同时的情况。我最初的逻辑是在两者不相同时，直接将元素入栈，这导致漏掉了很多情况。后来通过反思，我发现正确的做法是：当入栈序列与出栈序列不同步时，应该先比较栈顶元素是否与当前出栈元素相同。如果相同，就出栈，否则才入栈。这个改动显著提高了程序的准确性。
* 通过模拟简单的测试用例（如 "abc -> cba"），我一步步分析入栈和出栈的过程，详细推敲每一步的操作，包括什么时候该出栈、什么时候继续比较下一个元素，以及指针该如何移动等细节。最终，在反复模拟中，我逐步优化了逻辑，使得程序能够正确处理所有测试用例。

**2.1.6总结和体会**

* 本次实验让我深刻体会到模拟问题时的关键在于对细节的把握。通过模拟列车进站出站的过程，我发现每一个操作（无论是入栈还是出栈）都需要精确执行，特别是在面对不同的出栈序列时，栈顶元素的比较决定了接下来的操作流程。正是通过这些细致的分析与调试，我得以完善解决方案。
* 这次实验让我认识到，先对栈顶元素进行比较再决定是否入栈是至关重要的，否则可能会遗漏很多特殊情况。此外，熟悉像EOF这样的特殊输入形式也对编写健壮的程序有很大帮助。通过不断的模拟和测试，我学会了如何逐步改进算法，并体会到了模拟问题中每一步都不是理所当然的，需要在反复实践中寻找最优解。
* 总之，这次实验帮助我更加熟练地掌握了栈的应用，并让我认识到模拟类问题不仅仅是简单的实现，还需要高度关注每一个步骤的合理性和准确性。

**2.2布尔表达式**

**2.2.1问题描述**

本题要求计算一系列布尔表达式。布尔表达式的符号如下：

* V 表示 True
* F 表示 False
* | 表示 or
* & 表示 and
* ! 表示 not

运算符的优先级为 ! > & > |。输入若干布尔表达式，按顺序输出每个表达式的计算结果。

**2.2.2基本要求**

 **输入**：若干布尔表达式，每行一个表达式。表达式符号之间可能有空格或没有空格，总长度最多 100 个字符。

 **输出**：对于每个表达式，输出 Expression 序号: 结果，其中结果为 V 或 F。

**2.2.3数据结构设计**

为了实现布尔表达式的计算，我们需要两个栈：

操作数栈：存储 V 和 F，即布尔值

运算符栈：存储运算符 !、&、| 以及括号 ( 和 )

// 布尔运算符栈与操作数栈

class Stack {

private:

char\* stackArray; // 用于存储栈元素

int top; // 栈顶指针

int capacity; // 栈容量

public:

Stack(int size) {

stackArray = new char[size]; // 初始化栈数组

top = -1; // 栈顶初始化

capacity = size;

}

void push(char item); // 压入元素

char pop(); // 弹出元素

char peek(); // 查看栈顶元素

bool isEmpty(); // 判断栈是否为空

~Stack(); // 析构函数，释放内存

};

**2.2.4功能说明（函数、类）**

该算法的核心思路是模拟运算符优先级，并使用栈来实现中缀表达式的运算。在处理每一个输入字符时，如果是布尔值（V 或 F），则直接入栈；如果是运算符，则根据其优先级与栈顶运算符比较，决定是将当前运算符入栈还是先进行计算。

function evaluateBooleanExpression(expression) :

initialize operatorStack and valueStack

for each character in expression :

if character is 'V' or 'F' :

push value onto valueStack

else if character is '(' :

push onto operatorStack

else if character is ')' :

while top of operatorStack is not '(' :

pop operator and perform calculation

pop '(' from operatorStack

else if character is an operator ('!', '&', '|') :

while operatorStack is not empty and precedence of top operator is greater or equal to current operator :

pop operator and perform calculation

push current operator onto operatorStack

while operatorStack is not empty :

pop operator and perform calculation

return top of valueStack

**2.2.5调试分析（遇到的问题和解决方法）**

* 在调试过程中，我一开始采用了直接匹配运算符并逐个执行计算的方式，但在处理两个连续的 ! 运算符时出现了问题。由于 ! 是一元运算符，而其他运算符是二元的，直接按普通运算符规则进行栈处理会导致计算错误。
* 为了解决这个问题，我重新设计了运算符的处理逻辑。在遇到 ! 时，只弹出一个操作数进行运算，而在 & 和 | 时则弹出两个操作数。通过对每种运算符进行细化处理，我成功通过了多个复杂的测试用例，特别是在处理括号和多个不同优先级运算符时，确保了表达式的正确性。
* 此外，在处理空格和非法字符时，我增加了额外的判断，确保输入表达式格式不影响结果。

**2.2.6总结和体会**

* 这次实验让我深入理解了栈在表达式计算中的作用，尤其是如何利用栈处理优先级和括号的嵌套问题。在实现过程中，最难的部分是确保所有运算符按优先级正确执行，特别是在遇到连续多个 ! 运算符时，调试了一些边界情况。
* 通过这次实验，我体会到调试时需要充分考虑不同输入的情况，尤其是在复杂的表达式中如何保证算法的正确性和健壮性。掌握布尔表达式的运算让我更加熟悉了栈的运用，也对符号运算有了更深的理解。

**2.3最长子串**

**2.3.1问题描述**

给定一个仅由字符 ( 和 ) 组成的字符串，计算该字符串中最长的有效括号子串的长度以及起始位置。若存在多个长度相同的子串，返回第一个子串的起始位置。如果输入为空字符串，输出 0 0

**2.3.2基本要求**

* 输入：一个由 ( 和 ) 组成的字符串，长度不超过 1e5。
* 输出：最长的有效括号子串的长度及其起始位置。

**2.3.3数据结构设计**

为了实现括号匹配并找到最长的有效子串，我们使用栈来保存括号的位置。栈中存储的是左括号的索引，当遇到右括号时，会通过栈来消去对应的左括号，并计算此时形成的有效括号子串的长度。

Stack: 用于保存左括号的索引，确保括号匹配。

left\_no[] : 保存左括号的位置。

right\_no[] : 保存无法匹配的右括号位置。

last\_not\_0[] : 保存所有不为0的位置，以便最后计算有效括号长度。

栈的主要操作包括：

* push：遇到左括号时，将其索引压入栈。
* pop：遇到右括号时，栈顶元素弹出，并计算当前括号匹配形成的子串长度。

**2.3.4功能说明（函数、类）**

主程序逻辑:

function main :

initialize input\_stack and helper arrays(left\_no, right\_no)

for each character in the input string :

if character is '(' :

push its index onto left\_no stack

if character is ')' :

if left\_no is not empty and can match a left parenthesis :

remove the matching left parenthesis and calculate valid substring

else :

add the position of ')' into right\_no

After processing the string :

Traverse the left\_no and right\_no stacks to find the largest gap between consecutive non - zero positions.

Output the longest valid substring length and its starting position.

伪代码实现:

initialize input\_stack[]

initialize left\_no[], right\_no[], and other helper variables

for each character in input\_stack:

if character is '(' :

add its index to left\_no[]

if character is ')' :

if left\_no[] is not empty and can match :

pop from left\_no[], mark matching parentheses as valid

else :

push the index of ')' onto right\_no[]

calculate the longest valid substring by traversing left\_no and right\_no

output max\_len and the\_start

**2.3.5调试分析（遇到的问题和解决方法）**

在最初的尝试中，我只在栈中存放了左括号的位置并通过匹配右括号来进行出栈操作。但是，当多个合法的括号组合依次拼接时，无法记录正确的子串长度。通过修改栈的结构，将未匹配的右括号位置也记录下来，并遍历两边的括号位置来计算最终的长度，解决了这个问题。

另一个问题出现在计算子串长度的逻辑中。在程序中需要注意的是，子串的长度需要通过索引差计算，而每次匹配完后必须检查栈是否为空，从而确保所有的括号都被正确处理。

**2.3.6总结和体会**

本题的关键是使用栈来处理括号匹配的问题。栈结构特别适合这种左右匹配的情况，因为它可以方便地处理嵌套的括号组合。在实现过程中，需要特别注意出栈和入栈的时机，以及如何计算匹配子串的长度。通过这次实验，我进一步加深了对栈数据结构的理解，并学会了如何处理边界情况（如空字符串或全为右括号的情况）。此外，本题让我认识到调试过程中对于细节处理的重要性，特别是在括号匹配和子串起始位置的计算上。

栈结构的使用不仅提高了问题的处理效率，还让代码更加简洁清晰。总的来说，本次实验让我更加熟练掌握了栈的应用及其在实际问题中的重要性。

**2.4队列的应用**

**2.4.1问题描述**

定一个 n\*m 的矩阵，其中 1 表示某位置有东西，0 表示该位置为空。要求计算矩阵中四邻域联通的 1 构成的区域数量，但仅在矩阵边缘联通的区域不计入。这个问题类似于细胞计数问题，要求找出所有的内嵌区域。

**2.4.2基本要求**

* 输入：第一行输入两个正整数 n 和 m，表示矩阵的行数和列数。接下来的 n 行是一个 n\*m 的矩阵，矩阵中的元素为 0 或 1。
* 输出：输出一个整数，表示区域的数量。

**2.4.3数据结构设计**

该问题可以使用广度优先搜索（BFS）或深度优先搜索（DFS）来遍历所有的区域。我们通过标记矩阵中的每一个区域（即相连的 1）来避免重复计算，同时通过队列或递归来实现区域的扩展。

bool matrix[n + 2][m + 2]; // 定义一个大小为 (n+2)\*(m+2) 的矩阵，用来防止边界越界

bool visited[n + 2][m + 2]; // 用于记录已经访问过的区域

queue<pair<int, int>> q; // BFS使用的队列，用于存储待访问的坐标

每个区域的探索从一个未访问的 1 开始，将其加入队列中，然后逐步检查其四个方向的相邻位置（上、下、左、右）。对于所有联通的 1，将其标记为已访问，直到队列为空为止。若该区域不在矩阵的边缘联通，则该区域计数。

**2.4.4功能说明（函数、类）**

主程序：

1. 输入矩阵：从输入中读取矩阵的尺寸 n 和 m，以及矩阵的内容。
2. BFS或DFS遍历：使用队列（BFS）或递归（DFS）遍历矩阵，找到每个独立的区域，并判断该区域是否仅靠边缘联通。
3. 区域计数：对所有符合条件的区域进行计数，并输出结果。

**伪代码实现**

function main :

initialize matrix and visited array

input n and m(number of rows and columns)

for each cell in matrix :

if cell contains 1 and is not visited :

perform BFS / DFS from this cell to explore the entire connected region

if region is not connected to the edge, increment the region count

output the region count

BFS实现

function BFS(x, y) :

initialize queue q

q.push((x, y))

mark(x, y) as visited

while q is not empty :

u = q.pop()

for each direction(up, down, left, right) :

if the neighbor cell is 1 and not visited :

push the neighbor into q

mark it as visited

**2.4.5调试分析（遇到的问题和解决方法）**

在调试过程中，我首先通过DFS方式进行遍历，但在处理边缘联通区域时，出现了遗漏和重复计算的错误。为了修正这些问题，我将边缘检查函数独立出来，确保每次探索到边缘时，都能正确地标记为“无效区域”。

此外，矩阵边缘越界的问题通过增加边界扩展解决。在初始定义矩阵时，将 n 行和 m 列扩大到 n+2 和 m+2，这样在访问矩阵边缘时不会出现访问越界的情况，从而减少了额外的边界判断逻辑。

最终，通过BFS方法替代DFS，简化了递归深度控制，同时保证了区域遍历的完整性。

**2.4.6总结和体会**

这道题目让我深入理解了广度优先搜索（BFS）在图遍历中的应用。特别是在处理大规模矩阵时，BFS通过队列实现逐层扩展，避免了深度优先搜索可能带来的递归栈溢出问题。

另外，边界处理是这道题的一个难点。通过构造一个带有虚拟边界的矩阵，我有效地避免了边界条件的特殊处理，使得程序更加简洁和高效。对于这种边界处理问题，这是一种很实用的技巧。

总的来说，这次实验让我更加熟悉了BFS的应用场景，也加深了对队列和矩阵遍历的理解。

**2.5队列中的最大值**

**2.5.1问题描述**

设计一个支持以下三种操作的队列，并且要求 GetMax 操作的时间复杂度尽可能低：

1. Enqueue(v)：将元素 v 入队。
2. Dequeue()：删除队首元素并返回该元素。如果队列为空，输出 "Queue is Empty"。
3. GetMax()：返回队列中的最大元素。如果队列为空，输出 "Queue is Empty"。

**2.5.2基本要求**

输入：第一行输入队列的容量 n。接下来多行执行操作：

* "dequeue"：出队，输出队首元素或队列为空时输出 "Queue is Empty"。
* "enqueue m"：将元素 m 入队，若队列已满，输出 "Queue is Full"。
* "max"：输出队列中的最大元素，若队列为空，输出 "Queue is Empty"。
* "quit"：结束输入，输出队列中的所有元素。

输出：每次操作后的结果。

**2.5.3数据结构设计**

为了高效实现 GetMax 操作，我们使用两个队列：

* 元素队列：用于正常存储入队和出队的元素。
* 最大值队列：用于存储当前队列中的最大元素，保持单调递减的顺序。每当有元素入队时，与最大值队列尾部的元素进行比较，如果新元素大于等于队尾元素，则将队尾元素出队，直到队尾元素大于新元素或队列为空。

class Queue :

queue<int> elementQueue; // 正常存储入队、出队元素的队列

deque<int> maxQueue; // 维护单调递减队列，用于存储最大值

function Enqueue(v) :

push v into elementQueue

while maxQueue is not empty and maxQueue.back() < v :

pop from maxQueue

push v into maxQueue

function Dequeue() :

if elementQueue is empty :

return "Queue is Empty"

else :

if elementQueue.front() == maxQueue.front() :

pop from maxQueue

return elementQueue.front(), then pop from elementQueue

function GetMax() :

if elementQueue is empty :

return "Queue is Empty"

else :

return maxQueue.front()

**2.5.4功能说明（函数、类）**

主函数：

* Enqueue(v)：将元素 v 入队，同时更新最大值队列，保持最大值队列中的元素是单调递减的。
* Dequeue()：出队时，如果出队的元素是当前的最大值，则需要同时更新最大值队列。
* GetMax()：直接返回最大值队列的队首元素。

伪代码实现

function main :

input n(队列容量)

while True :

read operation :

if operation is "enqueue v" :

if elementQueue.size() == n :

print "Queue is Full"

else :

Enqueue(v)

if operation is "dequeue" :

if elementQueue is empty :

print "Queue is Empty"

else :

print Dequeue()

if operation is "max" :

if elementQueue is empty :

print "Queue is Empty"

else :

print GetMax()

if operation is "quit" :

break

print all elements in elementQueue

**2.5.5调试分析（遇到的问题和解决方法）**

在最初的尝试中，我尝试通过在出队时重新计算队列的最大值，但在测试中发现，若当前最大值被出队，且下一个最大值已经被删除，导致队列中最大值错误。为了解决这个问题，我使用了**单调队列**的思路，保证队列中的元素单调递减，从而可以在 O(1) 的时间内获取最大值。

此外，测试用例中队列已满时的边界条件也需要特别处理，确保当队列容量达到 n 时，新的元素不能再入队。

**2.5.6总结和体会**

这道题目让我深入理解了如何在队列中维护动态最大值，并且通过**单调队列**的思想，能够在保持队列正常操作的同时，以 O(1) 的时间复杂度高效地实现 GetMax 操作。结合栈与队列的特性，单调栈是一种非常实用的算法技巧。

在调试过程中，我发现维护单调队列的顺序是解决动态最大值问题的关键。通过这次实验，我对队列、栈的结合使用有了更深的理解，也掌握了如何在复杂操作中保证时间复杂度的优化。

**3.实验总结**

在本次实验中，我们主要实现并探索了栈和队列两种数据结构，尤其是通过这些结构解决动态最大值和广度优先搜索（BFS）相关问题。这些操作涵盖了数据结构的核心应用，并引入了常见的高级算法思想，如单调栈和BFS。

* 栈的实现与应用

本实验使用链式栈（链栈）来实现栈操作。栈是一种后进先出（LIFO）的数据结构，适合处理具有递归性质或需要逆序处理的数据。在这次实验中，栈被用来模拟与四则运算类似的操作，如括号匹配、表达式求值等。此外，栈还在解决队列中的最大值问题时，和队列结合使用，以支持动态最大值的高效计算。

其中，单调栈的使用是本实验的亮点。通过保持栈内元素的单调递减，我们能够在 `O(1)` 时间复杂度下获得队列中的最大值。这种思想不仅优化了算法效率，也展示了如何通过栈实现复杂的动态查询操作。

* 队列的实现与应用

实验中使用了链队列来模拟标准的队列操作。队列是一种先进先出（FIFO）的数据结构，通常用于按顺序处理任务。在本实验的队列实现中，我们结合了广度优先搜索（BFS）的思想。队列广泛用于图的遍历操作，尤其是在处理层次结构、最短路径问题等情景时，BFS是一种非常经典的算法。

通过这次实验，我们实现了队列与栈的结合使用，例如在实现队列最大值时，将队列和单调栈结合，用队列保存正常顺序的元素，用栈维护最大值。这种结合有效地优化了查询最大值的时间复杂度。

* 总结

通过本次实验，我加深了对栈和队列这两种基础数据结构的理解，尤其是如何通过它们解决实际问题。单调栈和广度优先搜索都是非常经典的算法，它们不仅高效，而且在多个领域有着广泛的应用。栈和队列在本质上具有互补的特性，在一些复杂问题中可以通过它们的结合使用，获得更高效的解法。

通过这次实验，我不仅掌握了栈与队列的基本操作，还学习了如何优化操作的时间复杂度，并将理论与实践相结合，解决了复杂的动态查询问题。这些技能和经验将对未来的数据结构学习和算法优化提供宝贵的帮助。