|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 姓名 |  | 学号 |  | 实验成绩 |  |
| 专业班级 |  | 实验时间 | 2025-05-07 | 实验地点 | 计算中心 |

实验三：栈与队列综合实验

1实验目的

1. 熟练掌握栈与队列的逻辑结构和运算特点；
2. 熟练掌握栈与队列的顺序存储结构和链式存储结构。
3. 熟练掌握栈与队列的有关算法设计，并在顺序结构和链式结构上实现。
4. 根据具体给定的需求，合理设计并实现相关结构和算法。

2 实验要求

1. 对用到的栈和队列进行合理封装；
2. 在相应的存储结构上，实现基本运算；
3. 设计算法实现相应的实验任务；
4. 程序运行、测试正确；
5. 实验程序有较好可读性，各运算和变量的命名直观易懂，符合软件工程要求；
6. 程序有适当的注释，程序的书写要采用缩进格式。

3 实验任务

编写算法求解下列问题：

1. 对一个合法的数学表达式来说，其中的各大小括号“{”，“}”，“[”，“]”，“(”和“)”应是相互匹配的。设计算法对以字符串形式读入的表达式S，判断其中的各括号是否是匹配的。比如：“{[](){}}”是匹配的，“{[(})]”就是不匹配的。

4 算法设计与实现描述

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

用栈进行匹配即可实现需求，’(‘入栈，’)’出栈,’[‘匹配’]’,’{‘匹配’}’，其余字符跳过检查，最后遍历完成后，检查栈是否为空，为空则代表全部匹配成功，不为空则代表括号不匹配

Status isMatch(char left,char right) {

if(left == '(' && right == ')') {

return OK;

}

if(left == '[' && right == ']') {

return OK;

}

if(left == '{' && right == '}') {

return OK;

}

return ERROR;

}

Status isSuited(char str[]) {

Stack S;

InitStack(&S);

int ptr = 0;

Elemtype Top;

while(str[ptr] != '\0') {

if(

str[ptr] == '(' ||

str[ptr] == '[' ||

str[ptr] == '{'

) {

Push(&S,str[ptr]);

} else if(

str[ptr] == ')' ||

str[ptr] == ']' ||

str[ptr] == '}') {

if(Pop(&S,&Top) == ERROR || isMatch(Top,str[ptr]) == ERROR) {

return ERROR;

}

}

ptr++;

}

if(S.top == -1) {

return OK;

} else {

return ERROR;

}

}

5运行结果截图及说明



1. 假设栈的输入序列为1、2、3、...、n，设计算法求出所有可能的出栈序列。比如输入1、2、3、4、5，可能出栈的序列为12345、13452等42个。

4 算法设计与实现描述

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

使用回溯法可以实现此算法

**代码：**

int count = 0;

void BackTrack(int maxNum,int nextNum, Stack\* stack, Stack\* out)

{

// 已经完成出栈

if (Length(\*stack) == 0 && nextNum > maxNum) {

Get(out);

count++;

return;

}

// 可以出栈,遍历这一步进行出栈的情况

if(Length(\*stack) > 0) {

Elemtype E1,E2;

// 出栈

Pop(stack,&E1);

Push(out,E1);

// 遍历

BackTrack(maxNum, nextNum, stack, out);

// 恢复现场

Pop(out,&E2);

Push(stack,E2);

}

// 可以入栈，遍历这一步进行入栈的情况

if (nextNum <= maxNum) {

// 入栈

Push(stack, nextNum);

// 遍历

BackTrack(maxNum, nextNum + 1, stack, out);

// 恢复现场

Elemtype E;

Pop(stack,&E);

}

}

void AllStackOutput(int n) {

// 初始化

Stack stack;

Stack out;

Init(&stack);

Init(&out);

count = 0;

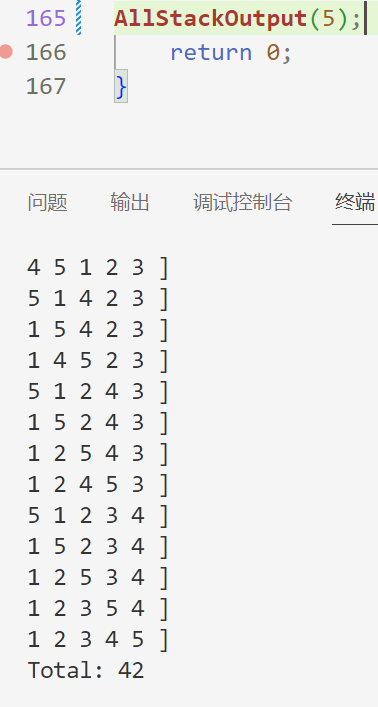
// 开始

BackTrack(n, 1, &stack, &out);

printf("Total: %d\n",count);

}

5运行结果截图及说明



1. 约瑟夫环问题(Josephus loop problem)

据说著名犹太历史学家Josephus有过以下的故事：在罗马人占领乔塔帕特后，39 个犹太人与Josephus及他的朋友躲到一个洞中，39个犹太人决定宁愿死也不要被敌人抓到，于是决定了一个自杀方式，41个人排成一个圆圈，由第1个人开始报数，每报数到第3人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。然而Josephus 和他的朋友并不想遵从。首先从一个人开始，越过k-2个人（因为第一个人已经被越过），并杀掉第k个人。接着，再越过k-1个人，并杀掉第k个人。这个过程沿着圆圈一直进行，直到最终只剩下一个人留下，这个人就可以继续活着。问题是，给定了和，一开始要站在什么地方才能避免被处决。Josephus要他的朋友先假装遵从，他将朋友与自己安排在第16个与第31个位置，于是逃过了这场死亡游戏。

17世纪的法国数学家加斯帕在《数目的游戏问题》中讲了这样一个故事：15个教徒和15 个非教徒在深海上遇险，必须将一半的人投入海中，其余的人才能幸免于难，于是想了一个办法：30个人围成一圆圈，从第一个人开始依次报数，每数到第九个人就将他扔入大海，如此循环进行直到仅余15个人为止。问怎样排法，才能使每次投入大海的都是非教徒。

我们可以把问题简单描述为：总共n个人，从1开始顺序编号，排成一个圆形队列。从1号开始报数，报数到k的人出圈。然后从出圈位置的下一个人继续从1报数，到k的人出圈。已经出圈的人不再参与报数，继续上面的做法，直到环形队列中最后只留下m个人。

即：n为初始总人数；k为出圈的报数；m为最后留下的人数。

请用队列求此问题。

测试用例：

输入：9 2 1 //总人数n； 报数数值k； 剩下人数m

输出：3 //剩下人的编号

输入：6 5 1

输出：1 //剩下人的编号

输入：11 3 2

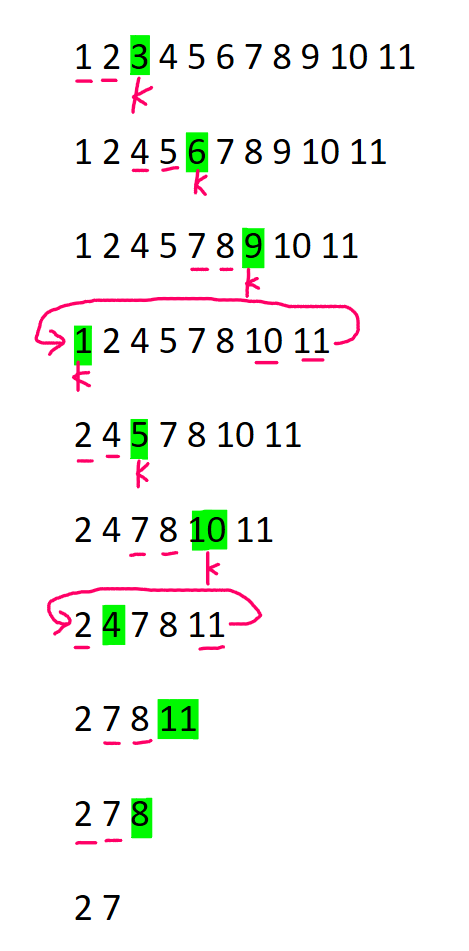
输出：2 7 //剩下人的编号

算法设计与实现描述

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

**分析：**

以11人，数到3出队，知道剩下2人为例，先进行分析：



可见借助循环链表或者循环队列能够更方便的实现算法，（3）用循环队列实现，（4）用血循环链表实现

循环队列实现：按上面推理过程，可发现规律，当currentPeople即剩余人数满足剩m人时结束代码，先出队，判断出队元素是否报数count为k，是则重置报数count即从头报数，再减去当前人数一名，若不是则把出队的放回队列队尾，即实现推理图中的向后遍历效果

**代码：**

void YueSeFu(int n, int k, int m) {

LinkQueue Q;

Init(&Q);

for (int i = 1; i <= n; i++) {

Add(&Q, i);

}

int count = 0;

int currentPeople = n;

while (currentPeople > m) {

ElemType temp;

DeAdd(&Q, &temp); // 出队

count++;

if (count == k) {

// 第k人出局

count = 0;

currentPeople--;

} else {

// 没出局的重新排到队尾

Add(&Q, temp);

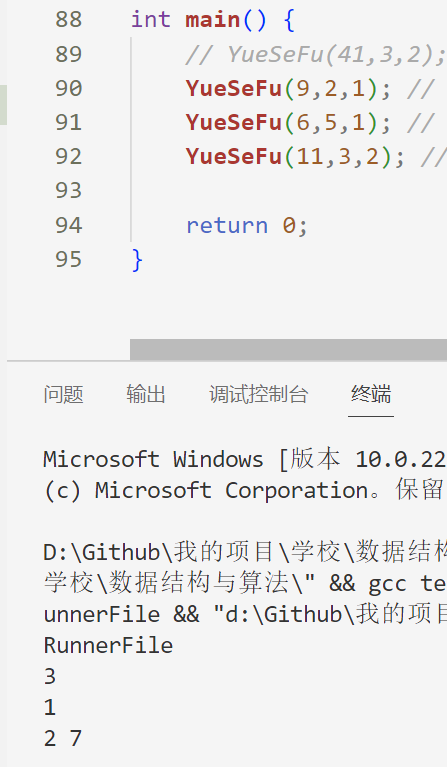
}

}

Get(&Q);

}

5运行结果截图及说明



1. 对上面约瑟夫环问题(Josephus loop problem)，尝试使用顺序表、链表等结构求解。

算法设计与实现描述

（书上给出的基本运算外，其它问题必须先给出算法思想或步骤，再给出算法描述）

基本思想上（3）已分析过，只不过链表实现将出队改为删除结点操作

**代码：**

void Count(LinkList \*L, int k, int m, int count) {

LinkList list = (\*L);

while (list->next != (\*L)) {

LinkList old = list->next;

if (count == k - 1) { // 第k个出圈

list->next = old->next;

free(old);

count = 0;

} else {

count++;

if (list->next != (\*L)) {

list = list->next;

} else {

count--; // 如果下一个是头节点，不移动且不加count

}

}

}

// 如果剩余人数 > m，继续递归

if (Length(L) > m) {

Count(L, k, m, count);

}

}

//总共n个人，从1开始顺序编号，排成一个圆形队列。从1号开始报数，报数到k的人出圈。然后从出圈位置的下一个人继续从1报数，到k的人出圈。已经出圈的人不再参与报数，继续上面的做法，直到环形队列中最后只留下m个人。

//递归 n为初始总人数；k为出圈的报数；m为最后留下的人数。

void YueSeFu(int n, int k, int m) {

Status I;

LinkList L;

I = Init(&L); // 初始化循环链表

for (int i = n; i > 0; i--) {

I = HeadAdd(&L, i); // 从大到小头插法

}

Count(&L, k, m, 0); // 启动递归，初始count为0

I = Get(L); // 输出/处理最后剩下的m个结点

}

5运行结果截图及说明



6总结、心得和建议

栈和队列是实际应用中常用的数据结构，栈的后进先出特性，可以用于一些需要原路返回的场景，例如在移动端APP交互，进入多级界面后，按返回键从即退栈，直到栈为空；而队列的先进先出特性，可以用于一些排队场景，例如消息队列使服务端在高并发场景下的网购APP将订单依次入队，然后按队列从队头处理。