作业 HW2 试验报告

姓名: 范潇　学号: 2254298

2023年 10月 15日

1　涉及数据结构和相关背景

本次实验主要涉及栈和队列这两个数据结构。

栈，即限定只在表的一段（表尾）进行插入和删除操作的线性表。我们可以用顺序表来实现顺序栈，或者用不带头结点的单链表来实现链栈，也可以用固定大小的数组来模拟栈。

队列，即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，允许插入的一端称为队头，允许插入的一端称为队尾。队列的存储结构有链队列和循环队列。其中链队列的本质即带头结点的线性链表。循环队列则是队列的顺序存储结构，只能用静态数组来实现。

下面给出的是利用静态分配整型指针的栈的功能实现。

```
//初始化栈
Status InitStack(SqStack &S)
{
    S.base = (ElemType*)malloc(sizeof(ElemType)*INIT_STACK_SIZE);
    if(!S.base)//检查是否申请成功
      return OVERFLOW;
    S.stacksize = INIT_STACK_SIZE;
    S.top = S.base;
    return OK;
}
```

```
//获取栈顶数据
Status GetTop (SqStack S,ElemType &e)
{
    if(S.top == S.base)//检查是否为空
      return ERROR;
```

```
    e = *(S.top -1);
    return OK;
}
```

```
//压栈
Status Push(SqStack &S,ElemType e)
{
    if(S.top-S.base >= S.stacksize){//检查是否上溢
        S.base = (ElemType*)realloc(S.base,sizeof(ElemType)*(S.stacksize+
            STACK_SIZE_INCREMENT));
        if(!S.base)
          return OVERFLOW;
        S.top = S.base + S.stacksize;
        S.stacksize += STACK_SIZE_INCREMENT;
    }
    *S.top++=e;
    return OK;
}
```

```
//出栈
Status Pop(SqStack &S,ElemType &e)
{
    if(S.base == S.top)//判断是否为空
      return ERROR;
    e = *--S.top;
    return OK;
}
```

```
//判断是否栈空
Status StackEmpty(SqStack S)
{
    return S.top ==S.base;
}
```

```
//重置栈
Status EmptyStack(SqStack &S)
{
    S.top = S.base;
    S.stacksize =0;
    return OK;
```

```
}
```

下面给出的是链队列的一些功能实现

```
//初始化队列
Status InitQueue(LinkQueue &Q)
{
    Q.front = Q.rear = (QueuePtr)malloc(sizeof(Node));
    if(!Q.front)
      return OVERFLOW;
    Q.front ->next = NULL;
    return OK;
}
```

```
//销毁队列
Status DestroyQueue(LinkQueue &Q)
{
    while(Q.front)
    {
        Q.rear = Q.front->next;
        free(Q.front);
        Q.front=Q.rear;
    }
    return OK;
}
```

```
//进入队列
Status EnQueue(LinkQueue&Q,ElemType e)
{
    Q.rear->next = (Node*) malloc(sizeof(Node));
    if(!Q.rear->next)
      return OVERFLOW;
    Q.rear = Q.rear->next;
    Q.rear->next = NULL;
    Q.rear->data = e;
    return OK;
}
```

```
//出队列
Status DeQueue(LinkQueue&Q,ElemType &e)
{
```

```
    if(Q.front == Q.rear)
      return ERROR;
    Node * temp = Q.front->next;
    Q.front ->next = temp->next;
    if(!temp->next)
      Q.rear = Q.front;
    e = temp->data;
    free(temp);
    return OK;
}
```

```
//判断队列是否为空
Status QueueEmpty(LinkQueue Q)
{
    return Q.front == Q.rear;
}
```

2 实验内容

2.1 列车进站

2.1.1 问题描述

问题描述： 有一队编号不同的火车依次入站。把火车站视为一个栈，请问哪些出站的顺序是合法的？

2.1.2 基本要求

输入要求： 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。

输出要求： 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

```
char in [LIST_INIT_SIZE];//存储进站次序
char out[LIST_INIT_SIZE];//存储出站顺序
typedef char ElemType;
typedef struct{
    ElemType* top;
    ElemType* base;
    int stacksize;
}SqStack;//利用静态分配指针来实现
SqStack station;//模拟火车站
```

2.1.4 功能说明

```
int main()
{
    InitStack(station);//初始化模拟栈
    scanf("%s",&in);//存储进栈顺序
    while(scanf("%s",&out)==1&&out[0]){//循环读取出栈顺序
    int i =0,j=0;
    char temp;
    EmptyStack(station);//重置栈
        while(in[i]!='\0'){//模拟进栈
            Push(station,in[i]);//进栈
            temp = in[i];//读取待出栈编号
            while(out[j] == temp){//循环判断当前的出栈编号是否等于待出栈编号
                Pop(station,temp);//出栈
             GetTop(station,temp);//读取下一个待出栈编号
                j++;
            }
            i++;//循环进栈
        }
        if(StackEmpty(station))//栈空则说明该出栈顺序是合法的。
          printf("yes\n");
        else
          printf("no\n");
    }
    return 0;
}
```

2.1.5 调试分析

在调试该程序时，我发现如果当一个出栈次序被判定为不合法后，后续的都被判为不合法，依次判断问题出在循环时没有重置变量，进行了针对性排查后发现是每次读入出栈次序时，没有将栈进行重置，在循环开头添加了对应语句后便成功通过。

2.1.6 总结与体会

在涉及处理多个输入时，要确保相互之间不会影响，即要在循环开头对于变量进行重置。

2.2 布尔表达式

2.2.1 问题描述

问题描述： 求解布尔表达式。

2.2.2 基本要求

输入要求： 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。

输出要求： 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

```
typedef struct {
    ElemType base[MAX_SIZE];//由于长度上限已知，且较小，所以用静态数组来实现
    int top;
}SqStack;
SqStack Operand;//存储VF
SqStack Opt;//存储其他运算符
char expression[MAX_SIZE];//存储表达式
```

2.2.4 功能说明

Algorithm 1 运算符优先级

输入: char in待进栈运算符，char cur栈顶运算符

输出: 比较结果

```
1: function COMPARE(char in,char cur)
2:     利用switch和if语句以及运算符之间的优先级和结合性来得出结果
3:     return 比较结果
4: end function
```

Algorithm 2 运算

输入: SqStack &Operand布尔值栈，SqStack &Opt运算符栈

输出: void

```
1: function CALC(SqStack &Operand,SqStack &Opt)
2:     if 运算符栈顶为!  then
3:         运算符栈出栈
4:         布尔值栈出栈
5:         相反的布尔值压进布尔值栈
6:     else
7:         运算符栈出栈
8:         布尔值栈出栈两次
9:         进行运算
10:        将结果对应的布尔值压入布尔值栈
11:    end if
12: end function
```

Algorithm 3 求解布尔表达式

输入: 布尔表达式

输出: 布尔值

```
1: function SolveExpression(char Expression[] 布尔表达式)
2:     while 遍历Expression中的每个字符,直至尾零 do
3:         if 是布尔值 then
4:             压入布尔值栈
5:         else
6:             if 布尔运算符栈非空 then
7:                 temp取运算符栈顶部
```

```
8:              while 该布尔运算符比栈顶布尔运算符temp优先级高 do
9:                  计算
10:                 if 运算符栈为空 then
11:                     break
12:                 else
13:                     temp取运算符栈顶部
14:                 end if
15:             end while
16:         end if
17:     end if
18: end while
19: while 运算符栈非空 do
20:     计算
21: end while
22: 布尔值栈出栈并打印
23: end function
```

2.2.5 调试分析

在调试该程序的过程中，我不仅仅打印出中间变量的值，还将中间所调用的函数名也给打印出来，方便查找问题的原因。

2.2.6 总结与体会

本题是求解涉及四则运算的表达式的变体。易错点在于取反运算符是右结合，而非左结合。同时，取反运算符是一元运算符，在实现时需要多加注意。

本例也体现出栈的实用性。

2.3 最长字串

2.3.1 问题描述

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。

2.3.2 基本要求

输入要求： 一行字符串。

输出要求： 子串长度，及起始位置。

2.3.3 数据结构设计

```
typedef struct {
    ElemType base[MAX_SIZE];//由于长度上限固定，且只有单一测试样例，所以选择使用静态数组实现。
    int top;
}SqStack;//模拟配对过程的栈
SqStack index;
int maxlen=0,maxpos=0;//最大长度，最大长度的子串开始位置
int len[MAX_SIZE]={0};//存储以对应位置为结尾的最长子串的长度
```

2.3.4 功能说明

```
int main()
{
    int n =0;
    int tempindex;
    char temp = getchar();
    InitStack(index);
    while(temp !='\r'&&temp !='\n'&&temp !=EOF){
        if(temp == '(')//读取的是左括号
          Push(index,n);//等待与之匹配的右括号
        if(temp ==')'&&!StackEmpty(index)){//读取的是右括号，且有与之匹配的左括号
            Pop(index,tempindex);
            len[n]= n-tempindex+1+len[tempindex-1];
            //计算匹配的左右括号之间的长度并加上在此之前与之相邻的子串长度，
            if(len[n]>maxlen){//检查是否要更新最大值，及其位置
                maxlen = len[n];
                maxpos =n-len[n]+1;//通过右括号的位置以及字串长度来求得左括号的位置
            }
        }//否则读取的是非法的右括号
```

```
19          temp = getchar();
20          n++;
21      }
22      printf("%d␣%d",maxlen,maxpos);
23      return 0;
24  }
```

2.3.5 调试分析

一开始提交测评后，反馈第1，4，7，8，10个样例没通过。后来将自己的输出和正确输出进行比较，发现一般数值十分接近。最后发现是自己对题目的理解有误，我一开始以为合法的字串的两端是一对相互配对的括号，而实际上只要是一串均能够相互配对的左右括号即可。

2.3.6 总结与体会

栈这一数据结构适合处理符合FILO的场景。

2.4 队列的应用

2.4.1 问题描述

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。

2.4.2 基本要求

输入要求： 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。

输出要求： 1行，代表区域数。

2.4.3 数据结构设计

```
typedef struct{
    int i;
    int j;
}ElemType,index;//二维数组中的下标
typedef struct Node{//元素结点
    ElemType data;
    struct Node * next;
}Node,*QueuePtr;//采用链队列
typedef struct {//特殊结点
    QueuePtr front;//队头指针
    QueuePtr rear;//队尾指针
}LinkQueue;
LinkQueue Queue;//用于搜索的队列
int mark[MAX_SIZE][MAX_SIZE]={0};//用于记录是否已被计数
int mat[MAX_SIZE][MAX_SIZE]={0};//用于存储输入矩阵
```

2.4.4 功能说明

Algorithm 4 查找相邻区域

输入: index cur搜索的起始下标，int &valid是否为合法区域

输出: void

```
function SEARCH(index cur,int &valid)
    cur进搜索队列
    while 搜索队列非空 do
        退队列，用cur来储存
        在将mark中的对应元素置1
        if mat中与cur相邻的位置为1，且在mark中为0 then
            将该位置进队列
            if 该位置不为矩阵边缘 then
                valid置1
            end if
        end if
    end while
end function
```

```
int main()
{
```

```
    scanf("%d%d",&n,&m);
    for(int i =1;i<=n;i++)//存储输入矩阵
      for(int j = 1;j<=m;j++)//下标起始为1
        scanf("%d",&mat[i][j]);
    for(int i = 1;i<=n;i++)
      for(int j=1;j<=m;j++)
        if(!mark[i][j]&&mat[i][j]){//未被搜索过且为1
            int valid=0;//需要有在中间的元素
            InitQueue(Queue);//初始化搜索队列
            index index{i,j};
            Search(index,valid);//进行搜索
            DestroyQueue(Queue);//销毁队列
            if(valid)//如果合法，则计数器+1
              num++;
        }
    printf("%d",num);
    return 0;
}
```

2.4.5 调试分析

程序结构较清晰，没有在调试过程中遇到困难。

2.4.6 总结与体会

可以利用循环和队列来代替一部分递归的使用。

这道题的一个易错点在于矩阵边缘部分的搜索，为了避免发生越界访问，以及为了统一处理，在存储输入二维矩阵时，下标应该从1开始，使得存储的二维矩阵有一圈为0的“边框”。

2.5 队列中的最大值问题

2.5.1 问题描述

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

输入要求： 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。

输出要求： 多行，分别是执行每次操作后的结果。

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。

2.5.3 数据结构设计

```
typedef long long Status,ElemType;
typedef struct Node{//元素结点
    ElemType data;
    struct Node * next;
}Node,*QueuePtr;
typedef struct {//特殊结点
    ElemType maxdata;//用于存储当前链表中的最大值
    long long max_size;
    QueuePtr front;//队头指针
    QueuePtr rear;//队尾指针
    long long Length;
}LinkQueue;
LinkQueue Q;
```

2.5.4 功能说明

Algorithm 5 初始化队列

输入: LinkQueue &Q待初始化队列，long long max_size队列最大值

输出: 操作状态

```
1: function InitQueue(LinkQueue &Q,long long max_size)
2:     为头尾指针申请内存空间，并判断是否成功
3:     将队列的长度置零
4:     将队列的最大长度置为max_size
5:     return OK
6: end function
```

Algorithm 6 数据进入队列

输入: LinkQueue &Q队列，ElemType e待进入元素

输出: 操作状态

```
1: function InitQueue(LinkQueue &Q,long long max_size)
2:     判断队列是否已满
3:     为普通结点申请内存空间，并判断是否成功
4:     if 队列为空 then
5:         将特殊结点中的maxdata置为e
6:     else
7:         if e大于maxdata then
8:             将maxdata置为e
9:         end if
10:    end if
11:    特殊结点的Length加一
12:    将新普通节点进入队列
13:    return OK
14: end function
```

Algorithm 7 搜寻队列中的最大值

输入: LinkQueue &Q队列

输出: 操作状态

```
1: function FindMax(LinkQueue &Q)
2:     判断队列是否为空
3:     将工作指针指向开始结点
```

```
    将max置为工作指针所指结点的数据
    while 工作指针遍历队列 do
        if 工作指针所指结点数据大于max then
            更新max
        end if
    end while
    将特殊节点中的maxdata置为max
    return OK
end function
```

Algorithm 8 出队列

输入: LinkQueue &Q队列，ElemType e

输出: 操作状态

```
function DeQueue(LinkQueue &Q,ElemType &e)
    判断队列是否为空
    将开始结点移出队列，对应的数据赋给e
    if e等于maxdata then
        FindMax
    end if
    释放移除结点的内存空间。
    return OK
end function
```

2.5.5 调试分析

采用输出中间变量以及调用函数的函数名的方式来辅助调试，过程较为顺利。

2.5.6 总结与体会

可以在特殊结点或头结点中存储关于表的额外信息，本题中我便在特殊结点中增加了maxdata这一数据域来记录表中的最大值。易错点在于初始化链队列时，不能将maxdata置1，因为后续的数据可能为负数。正确的操作应该是将第一个进入的元素赋给maxdata。有了这一数据域后，便能降低遍历整个队列来获取最大值的频率。

3 实验总结

栈和列表本质上是操作受限的线性表，它们将操作限制在了表的两端，但是仍然有很广泛的应用场景。操作的简化也使得栈和列表的实现过程相对简单。

根据使用场景中的数据是FIFO还是FILO，应该分别选用队列和栈。

在某些情况下，可以使用队列和循环的组合来代替递归函数。