《数据结构》上机报告

2018 年 11 月 11 日

姓名： 赵得泽 学号： 1753642 班级： 电子 2 班 得分：

<table>
<tr><td>实验题目</td><td colspan="2">汉诺塔（栈）</td></tr>
<tr><td>问题描述</td><td colspan="2">汉诺塔（又称河内塔）是源于印度一个古老传说的益智玩具。
大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着 64 片黄金圆盘。
大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。</td></tr>
<tr><td rowspan="2">基本要求</td><td colspan="2">无</td></tr>
<tr><td>已完成基本内容（序号）：</td><td>无</td></tr>
<tr><td rowspan="2">选做要求</td><td colspan="2">1. 现在有 n（n>=2）根高为 mi（mi>=1，1<=i<=n）的柱子，一开始第一根柱子有 k（0<=k<=m1）个从下到上放置的从大到小、大小依次为 K1~Kk 的圆盘，现在给与你一些操作步骤，你依次按照操作步骤给出结果</td></tr>
<tr><td>已完成选做内容（序号）</td><td>1</td></tr>
<tr><td>数据结构设计</td><td colspan="2">

```
class  SqStack
{
public:
    SElemType *base;
    SElemType *top;
    int    stacksize;
    int    count;
    SqStack();
    ~SqStack();
    Status StackEmpty();
    Status GetTop();
    Status Push(SElemType e);
    Status Pop(SElemType&e);
 };
```

本实验是栈的实际应用，用到的的数据结构类型是顺序栈，即线性表的顺序存储结构；它是用一组地址连续的存储单元依次存放栈顶到栈底的元素。其中包括指针域 top 和 base，base 用来指向栈底，也就是用来判断栈是否为空，top 指针则是进行元素的入栈和出栈操作，还有一个 int 型数据 stacksize 是初始化设定的栈的大小。还有其他的成员函数则是栈的基本操作，包括初始化，销毁，清空，取栈顶元素，入栈出栈。通过栈的基本操作来实现来实现相应的功能。</td></tr>
</table>

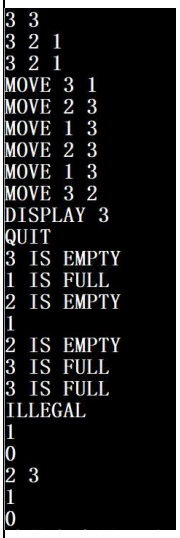
功能(函数)说明	/**
	函数功能：构造函数（初始化栈）
	说明：为base分配存储空间，栈顶等于栈底，栈的初始长度置零。
	**/
	SqStack::SqStack()
	{
	base = new SElemType[STACK_INIT_SIZE];
	if (base == NULL)
	exit(LOVERFLOW);
	top = base;
	stacksize = 0;
	count = 0;
	}
	/**
	函数功能：析构函数（销毁栈）
	说明：将base指针delete，top置空，栈长度置零。
	**/
	SqStack::~SqStack()
	{
	if (base)
	delete base;
	top = NULL;
	stacksize = 0;
	}
	/**
	函数功能：判断栈是否为空
	说明：当栈顶=栈底时，栈为空。
	**/
	Status SqStack::StackEmpty()
	{
	if (top == base)
	return TRUE;
	else
	return FALSE;
	}
	/**
	函数功能：取栈顶元素
	说明：如果实际大小为零，则不能取栈顶元素，返回0；否则返回栈顶元素即可。
	**/
	Status SqStack::GetTop()
	{
	if (count == 0)
	return 0;
	else

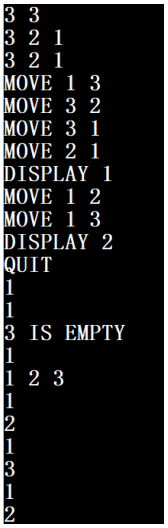
```
        return  *(top - 1);
}
/*************************************************************************
函数功能：出栈
说明：如果栈空，则不能出栈；若栈未空，则栈顶指针减1，输出栈顶元素。
Count记录栈的实际大小。
*************************************************************************/
Status SqStack::Pop(SElemType &e)
{
    if (top == base)
        return ERROR;
    e = *--top;
    count--;
    return OK;
}
/*************************************************************************
函数功能：入栈
说明：如果栈满，就不能执行入栈操作，但是可以在这个基础上进行对栈的扩充，否则直接入栈，将元素放到栈顶，然后指针栈顶指针上移一位。
*************************************************************************/
Status SqStack::Push(SElemType e)
{
    /*如果栈已满则扩充空间*/
    if (top - base >= stacksize)
    {
        SElemType *newbase;
        newbase = new SElemType[stacksize + STACK_INIT_SIZE];
        if (!newbase)
            return LOVERFLOW;
        memcpy(newbase, base, stacksize * sizeof(SElemType));
        delete base;
        base = newbase;
        top = base + stacksize;
        stacksize += STACKINCREMENT;
    }
    *top++ = e;
    count++;
    return OK;
}
/*************************************************************************
函数功能：移动圆盘
说明：按照题目要求，设置一个标志flag，将a柱上面的圆盘移动到b柱上面。若是a柱空，就让flag=0，表示这种情况考虑完了；如果b柱实际长度等于初始赋予的长度，说明b柱是满的，之后也让flag=0；如果a柱最上面圆盘的大小比b柱最上面的圆盘大小 大，并且b柱不空，那么这次移动是不合法的，也让
```

```
flag=0；如果flag=1，即在剩下的情况中，移动圆盘是合法的，就输出a柱顶的数据，然后让它出栈，接着再让它放到b柱上，即入b栈，这样就完成了一次移动。
**************************************************************************/
void MOVE_Disk(SqStack CS[], int a, int b)
{
    int c;
    int flag = 1;
    if (CS[a].StackEmpty())
    {
        cout << a << " " << "IS EMPTY" << endl;
        flag = 0;
    }
    if (CS[b].count == CS[b].stacksize)
    {
        cout << b << " " << "IS FULL" << endl;
        flag = 0;
    }
    if (CS[a].GetTop() > CS[b].GetTop() && !CS[b].StackEmpty())
    {
        cout << "ILLEGAL" << endl;
        flag = 0;
    }
    if (flag == 1)
    {
        cout << CS[a].GetTop() << endl;
        CS[a].Pop(c);
        CS[b].Push(c);
    }
}
/**************************************************************************
函数功能：显示某个柱子上面的圆盘
说明：如果柱子（栈）不空，那就新建立一个指针，然后从栈顶向栈底移动，每次输出一个元素，直到栈底，停止循环；若栈空，则输出0即可。
**************************************************************************/
void Display(SqStack CS[], int a)
{
    if (!CS[a].StackEmpty())
    {
        int *p = CS[a].top - 1;
        while (p >= CS[a].base)
        {
            cout << *p << " ";
            p--;
```

<table>
<tr><td></td><td>

```
        }
        cout << endl;
    }
    else
        cout << 0 << endl;
}
```

</td></tr>
<tr><td>开发环境</td><td>Win10 ，vs2017， C++高级程序语言设计</td></tr>
<tr><td>调试分析</td><td>1.
 


</td></tr>
<tr><td>心得体会</td><td>这次实验主要还是对栈的运用，将栈放到实际问题中去处理，让我对它的特点有了更加深切的体会。

按照题目的要求，进行入栈、出栈，判断栈空栈满等操作，算法最主要的环节还是在将 a 柱圆盘移动到 b 柱圆盘上条件的判断和情况的讨论，如果这个环节出现了问题，就会导致整个问题陷入一种困境。有可能漏掉其中的一种情况，或者栈空栈满的判断出现问题，就有可能让程序在中途异常停止，例如，如果 a 顶的圆盘比 b 柱上的圆盘大，就不能进行移动，这种情况下，我们首先要得到两个柱子顶端的元素才能进行判断，那么问题就来了，如果只考虑得到元素，却不考虑 b 柱是否为空，那何来柱顶元素之说？这是一个比较隐蔽的问题，所以应该值得注意。

再者，当 display 指令出现的时候，即显示某个柱子上面的所有圆盘，在此必须先建立一个指针指向栈顶，再去操作这个指针，依次输出所有元素。不能直接就去操作栈顶指针，这样就相当于直接将栈清空，那这个指令再次出现的时候，就会得到错误的结果，这也是本次算法设计中应当注意的问题。

总之，栈的应用，首先要对它的物理结构有清晰的认识，然后你每一步的操作也要十分谨慎，防止操作栈越界，造成程序崩溃。</td></tr>
</table>