CHAPTER 3

第3章 栈的应用

3.1 存储结构与基本运算的算法

1. 顺序栈

采用顺序存储结构的栈简称顺序栈。

(1) 顺序栈的 C 语言描述如下(存放于 seqstack.h 文件中):

```
typedef struct                    /*顺序栈定义*/
{
    DataType data[MAXNUM];        /*存放栈的数据元素*/
    int top;                      /*栈顶指针,用来存放栈顶元素在数组中的下标*/
}SeqStack;
```

(2) 基本运算的算法如下(存放于 seqstack.c 文件中):

① 置空栈。

```
void SStackSetNull(SeqStack *s)          /*置空栈*/
{
    s->top=-1;
}
```

② 判栈空。

```
int SStackIsEmpty(SeqStack *s)  /*判断栈S是否为空栈,为空栈时返回值为真,反之为假*/
{
    return(s->top<0?TRUE:FALSE);
}
```

③ 进栈。

```
int SStackPush(SeqStack *s,DataType x)
{
    if(s->top==MAXNUM-1)
    {
        printf("栈上溢出!\n");
        return FALSE;
    }
```

```
    else
    {
        s->top=s->top+ 1;
        s->data[s->top]=x;
        return TRUE;
    }
}
```

④ 出栈。

```
int SStackPop(SeqStack * s,DataType * x)
{
    if(s->top==-1)
    {
        printf("栈下溢出!\n");
        return FALSE;
    }
    else
    {
        * x=s->data[s->top];
        s->top--;
        return TRUE;
    }
}
```

⑤ 读栈顶。

```
DataType SStackGetTop(SeqStack * s)
{
    if(s->top==-1)
    {
        printf("栈下溢出!\n");
        return FALSE;
    }
    else
        return(s->data[s->top]);
}
```

⑥ 输出栈。

```
void SStackPrint(SeqStack * s)
{
    int p;
    if(SStackIsEmpty(s)==TRUE)
        printf("栈空!\n\n");
    else
    {
```

```
        printf("栈数据元素如下:\n\n");
        p=s->top;
        while(p>=0)
        {
            printf("%d\n",s->data[p]);
            p--;
        }
        printf("\n\n");
    }
}
```

⑦ 判断栈满。

```
int SStackIsFull(SeqStack * s)              /* 栈 S 为满栈时返回值为真,反之为假 */
{
    return(s->top==MAXNUM-1?TRUE:FALSE);
}
```

⑧ 顺序栈运算的综合实例。

存放于 31mainseqstack.c 文件中。

```
#include "consts.h"
typedef int DataType;
#define MAXNUM 100
#include "seqstack.h"
#include "seqstack.c"
int main(int argc,char * argv[])
{
    DataType x;
    SeqStack ss;
    int read=0;
    do
    {
        puts("        关于顺序栈的操作\n");
        puts("        ======================\n");
        puts("        1 ------置空栈");
        puts("        2 ------入栈");
        puts("        3 ------出栈");
        puts("        4 ------输出");
        puts("        0 ------退出");
        printf("      请选择代号(0-4)");
        scanf("%d",&read);
        printf("\n");
        switch(read)
        {
            case 1: SStackSetNull(&ss);
```

```
                break;
            case 2: printf("     输入入栈数据元素:");
                scanf("%d",&x);
                SStackPush(&ss,x);
                break;
            case 3: if(SStackPop(&ss,&x)!=FALSE)
                    printf("     出栈数据元素是: %d\n",x);
                break;
            case 4: SStackPrint(&ss);
                break;
            case 0: read=0 ;
                break;
        }
    }while(read!=0 );
    return 0;
}
```

2. 链栈

链栈是指采用链式存储结构实现的栈。为了便于操作,采用带头结点的单链表实现栈,将表头作为栈顶,则链表的表头指针即为栈顶指针。

(1) 链栈的 C 语言描述如下(存放于 linkstack.h 文件中):

```
typedef struct node
{
    DataType data;                          /*数据域*/
    struct node *next;                      /*指针域*/
}LinkStack;                                 /*链栈结点类型*/
```

(2) 基本运算的算法如下(存放于 linkstack.c 文件中):

① 置空栈。

```
LinkStack *LStackInit()                     /*初始化链栈*/
{
    LinkStack *h;
    h=(LinkStack *)malloc(sizeof(LinkStack));
    h->data=1;
    h->next=NULL;
    return h;
}
```

② 判栈空。

```
int LStackIsEmpty(LinkStack *ls)            /*判别空栈*/
{
    return (ls->next?FALSE:TRUE);
```

```
}
```

③ 进栈。

```
LinkStack * LStackPush(LinkStack * ls,DataType x)   /* 入栈 */
{
    LinkStack * p;
    p=(LinkStack * )malloc(sizeof(LinkStack));      /* 分配空间 */
    p->data=x;                                      /* 设置新结点的值 */
    p->next=ls;                                     /* 将新元素插入栈中 */
    ls=p;                                           /* 将新元素设为栈顶元素 */
    return ls;
}
```

④ 出栈。

```
DataType LStackGetTop(LinkStack * ls)               /* 取栈顶元素 */
{
    if(!ls)
    {
        printf("\n 栈是空的!");
        return ERROR;
    }
return ls->data;
}
```

⑤ 读栈顶。

```
LinkStack * LStackPop(LinkStack * ls,DataType * e)  /* 出栈 */
{
    LinkStack * p;
    if(!ls)                                         /* 判断是否为空栈 */
    {
        printf("\n 链栈是空的!");
        return NULL;
    }
    p=ls;                                           /* 指向被删除的栈顶 */
    * e=p->data;                                    /* 返回栈顶元素 */
    ls=ls->next;                                    /* 修改栈顶指针 */
    free(p);
    return ls;
}
```

⑥ 链栈运算实例。

存放于 31mainlinkstack.c 文件中。

```
#include "consts.h"
```

```
typedef int DataType;
#include "linkstack.h"
#include "linkstack.c"
int main(int argc,char * argv[])
{
    int a[5]={1,2,3,4,5},i,isEmtpy;
    LinkStack * linkStack;
    DataType tmp;
    linkStack =LStackInit();                          /* 初始化链栈 */
    printf("\n 依次向链栈内压入 1,2,3,4,5!\n" );
    for(i=1;i<5;i++)
        linkStack=LStackPush(linkStack,a[i]);
    printf("栈顶元素为:");
    printf("%d\n",LStackGetTop(linkStack));
    linkStack=LStackPop(linkStack,&tmp);              /* 栈顶元素出栈 */
    printf("出栈后的栈顶元素是:");
    printf("%d\n",LStackGetTop(linkStack));
    isEmtpy =LStackIsEmpty(linkStack);                /* 判断是否为空栈 */
    if(isEmtpy==0)
        printf("当前链栈为非空链栈!\n");
    else
        printf("当前链栈为空链栈!\n");
    return 0;
}
```

3.2 括号匹配

1. 问题描述

设某一算术表达式中包含圆括号、方括号或花括号三种类型的括号，编写一个算法判断其中的括号是否匹配。

2. 设计要求

(1) 程序对所输入的表达式能给出适当的提示信息，表达式中包含括号，括号分为圆括号、方括号和花括号三种类型。

(2) 允许使用四则混合运算(+、-、* 和/)，以及包含变量的算术表达式。

(3) 只验证表达式中的括号是否匹配(圆括号、方括号和花括号三种类型)，并给出验证结果。

3. 数据结构

本课程设计使用的数据结构是栈，利用顺序栈来实现。

4. 分析与实现

在算术表达式中，通常包含数字符号、运算符以及括号(圆括号、方括号和花括号三种类型)。本题的解决关键在于对各种括号符号的处理。本实例使用一个运算符栈 st，逐个读入字符，当遇到“(”、“[”或“{”时括号入栈，当遇到“}”、“]”或“}”时判断栈顶指针是否为匹配的括号，若不是则括号不匹配，算法结束；若是则退栈，继续读取下一个字符，直到所有字符读完为止，若栈是空栈，则说明括号是匹配的，否则括号不匹配。

具体代码如下：

```
#include "consts.h"
typedef char DataType;
#define MAXNUM 100
#include "seqstack.h"
#include "seqstack.c"
int IsCorrect(char * str)
{
    SeqStack st;
    char x;
    int i,flag=TRUE;
    SStackSetNull(&st);
    for(i=0;str[i]!='\0';i++)                    /* 从字符串的第一个字符开始,逐个判断 */
    {
        switch(str[i])
        {
            case '(': SStackPush(&st,'(');                    /* 如果是“(”,入栈 */
                    break;
            case '[': SStackPush(&st,'[');                    /* 如果是“[”,入栈 */
                    break;
            case '{': SStackPush(&st,'{');                    /* 如果是“{”,入栈 */
                    break;
            case ')': if(!(SStackPop(&st,&x) && x=='('))      /* 如果是“)”,则“(”出栈 */
                    flag=FALSE;
                    break;
            case ']': if(!(SStackPop(&st, &x) && x=='['))     /* 如果是“]”,则“[”出栈 */
                        flag=FALSE;
                    break;
            case '}': if(!(SStackPop(&st, &x) && x=='{'))     /* 如果是“}”,则“{”出栈 */
                        flag=FALSE;
                    break;
        }
        if(!flag)
            break;
```

```
    }
    if(SStackIsEmpty(&st)&&flag)                    /*如果最后栈为空,则括号匹配*/
        return TRUE;
    else
        return FALSE;
}
int main(int argc,char*argv[])
{
    char*str;
    str=(char*)malloc(100*sizeof(char));
    printf("请输入带括号(()、[]和{})的表达式:\n");
    while(scanf("%s",str)&&strcmp(str,"#"))
    {
        if(IsCorrect(str))
            printf("表达式括号匹配\n");
        else
            printf("表达式括号不匹配\n");
    }
    return 0;
}
```

注意：本课程设计的详细代码存放于光盘 32bracketmatch.c 文件中。

5. 运行与测试

```
请输入带括号(()、[]和{})的表达式:
34+(2+3)
表达式括号匹配
45-(3*4
表达式括号不匹配
54-([6+7)]
表达式括号不匹配
```

6. 总结与思考

括号匹配问题是栈的典型应用，算法简单易懂，读者应上机具体实现，才能提高编程能力。

本实例是采用顺序栈作为存储结构具体实现的，在实例中考虑了算术表达式中包含圆括号、方括号和花括号三种类型括号的情况，读者可以考虑简化到一种括号程序该怎样写；也可以考虑把问题进一步复杂化该如何实现。

本实例只给出了括号匹配的一种实现方式，读者可以考虑用其他方法实现，通过不同的实现方式练习可开阔思路，达到触类旁通、举一反三的目的。

3.3 汉诺塔问题

1. 问题描述

设有三个分别命名为 X、Y 和 Z 的塔座，在塔座 X 上插有 n 个直径各不相同，从小到大依次编号 1、2、…、n 的圆盘，现要求将 X 塔座上的 n 个圆盘移到塔座 Z 上，并插在 X、Y 和 Z 中任一塔座；任何时候都不允许将较大的圆盘放在较小的圆盘之上。

2. 设计要求

(1) 程序要求用户输入初始圆盘数。
(2) 输出所有的移动过程。

3. 数据结构

本课程设计使用的数据结构是栈，利用顺序栈来实现。

4. 分析与实现

汉诺塔(又称河内塔)问题是印度的一个古老传说。开天辟地的神勃拉玛在一个庙里留下了三根金刚石棒，第一根上面套着 64 个圆的金片，最大的一个在底下，其余一个比一个小，依次叠上去，庙里的众僧不倦地把它们一个个地从这根棒搬到另一根棒上，规定可利用中间的一根棒作为帮助，但每次只能搬一个，而且大的不能放在小的上面。解答结果请自己进行计算，面对庞大的数字(移动圆片的次数)，看来众僧们耗尽毕生精力也不可能完成金片的移动。

后来，这个传说就演变为汉诺塔游戏：

(1) 有三根杆子 X、Y 和 Z。X 杆上按从小到大依次放置若干大小不等的盘子；
(2) 每次只能移动一个盘子，大的不能放在小的上面；
(3) 把所有盘子从 X 杆全部移到 Z 杆上，可借助中间 Y 杆。

经过研究发现，汉诺塔的破解很简单，就是按照移动规则向一个方向移动盘子。

如 3 阶汉诺塔的移动：X→Z，X→Y，Z→Y，X→Z，Y→X，Y→Z，X→Z。

在这里采用一种非递归的算法实现。

实现本实例的栈的结构如下：

```
#include "consts.h"
#define MAXNUM 50
typedef struct                    /* hanoi 结构体 */
{
    int no;                       /* no 为 hanoi 标志 */
    int ns;                       /* ns 为 hanoi 盘子序号 */
    char x,y,z;                   /* x,y,z 分别为三个 hanoi 塔座 */
}Stack;
```

这里用结构体数组来模拟栈的结构，其中 no 为一种标记，当 no 值为 0 时表示可直接移动一个圆盘，当 no 值为 1 时表示需进一步分解；ns 表示当前圆盘数；X，Y 和 Z 表示三个塔座。由此得到 hanoi 函数，此函数将 X 塔座上的 n 个盘子借助于 Y 塔座按自下向上从大到小的顺序转移到 Z 塔座上。在此过程中用后进先出的数据结构——栈，模拟借助 Y 塔座向 Z 塔座转移的过程。

实现的具体程序如下：

```
void Hanoi(Stack st[],int n,char a,char b,char c)  /*hanoi 移动程序*/
{
    int top=1;                                     /*top 为栈顶指针*/
    int n1;                                        /*n1 为当前盘子序号*/
    char a1,b1,c1;                                 /*a1,b1,c1 为临时塔座*/
    st[top].no=1;
    st[top].ns=n;
    st[top].x=a;
    st[top].y=b;
    st[top].z=c;
    while(top>0)
    {
        if(st[top].no==1)
        {
            n1=st[top].ns;                         /*退栈 hanoi(n,x,y,z)*/
            a1=st[top].x;
            b1=st[top].y;
            c1=st[top].z;
            top--;
            top++;                                 /*将 hanoi(n-1,x,z,y)入栈*/
            st[top].no=1;
            st[top].ns=n1-1;
            st[top].x=b1;
            st[top].y=a1;
            st[top].z=c1;
            top++;                                 /*将第 n 个圆盘从 X 移到 Z*/
            st[top].no=0;
            st[top].ns=n1;
            st[top].x=a1;
            st[top].y=c1;
            top++;                                 /*hanoi(n-1,y,x,z)入栈*/
            st[top].no=1;
            st[top].ns=n1-1;
            st[top].x=a1;
            st[top].y=c1;
            st[top].z=b1;
        }
```