**数据结构实验报告**

**实验七 利用二叉树求解表达式的值**

学 生 姓 名 ：杨仕振

学 号 ：2015301500349

二○一六年十二月

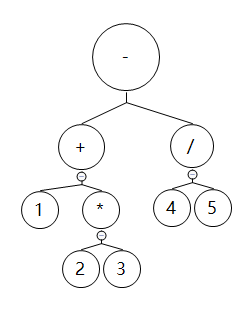
**一、简介**

【问题描述】设计一个程序，用二叉树来表示代数表达式，树的每一个节点包括一个运算符或运算数。代数表达式中只包含“+”，“-”，“\*”，“/”和一位整数且没有错误。按照先乘除后加减的原则构造二叉树，并求出表达式的值。

【提示】

建立运算符栈OPTR，来暂存已经扫描到的还未处理的运算符

表达式树栈EXPT，来暂存已建立好的表达式树的根结点，以便其作为另一个运算符结点的子树而被引用。



如图，该树就是代数表达式“1+2\*3-4/5”对应的二叉树。

**二、算法说明**

建立过程：从算式头部开始扫描，若字符串没有终止，则创建一个二叉树节点，

若为数字，则将数字存入节点，将指针计入表达式数栈EXPT。

若为加减，则将运算符存入节点，将其指针暂存至OPTR，直至遇到下一个加减符号或终止符号再取出，再将左、右孩子置为栈EXPT顶的两个节点，并将两个孩子节点出栈，将这个运算符节点的指针计入表达式数栈EXPT，再将其从OPTR出栈；

若为乘除，则将运算符存入节点，将其指针暂存至OPTR，将EXPT栈顶元素和其后的元素分别置为左右孩子节点，然后取出EXPT栈顶元素。再将这个运算符节点的指针存入EXPT，再将OPTR栈顶元素出栈。

如果OPTR栈中仍然存在元素，那么肯定是+或者-，则只需按照+或-对其进行处理即可。

至此二叉树建立完成。

运算方式：采用后序遍历的方法，对二叉树进行扫描，用变量result记录结果，并将临时结果用栈st存储，最终得到的栈顶元素就是结果

至此可以打印结果。

**三、测试结果**

由于代数表达式中只包含“+”，“-”，“\*”，“/”和一位整数且没有错误，进行测验的时候只需要对只含单个运算符和两个运算符的算式进行测试即可：

下面是测试的方案及运算结果：

|  |  |
| --- | --- |
| 算式 | 结果 |
| 1+1 | 2 |
| 1-1 | 0 |
| 3\*5 | 15 |
| 4/2 | 2 |
| 1+1-3 | -1 |
| 3\*4/6 | 2 |
| 1+2\*3 | 7 |
| 6/2-1 | 2 |

**四、分析与探讨**

由于题意前提是没有输入错误，并且只允许输入一位数字和四则运算符，所以相对来说比较简单，但是如果想要实现更为复杂的功能也是可以的，比如说对算式的正确性进行检测（顺序是否正确，除数不应为0），或者允许更大的整数，甚至是浮点数进行运算，也可以考虑引入括弧……

对于这个程序而言，其时间复杂度为O(n),因为扫描算式时需要将所有字符遍历一遍，空间复杂度为O(log2(n+1))。

**附录：源代码**

#include<stdio.h>

#include<stdlib.h>

#define Maxsize 12//其值减一为表达式的最大长度

int result;//表达式的结果

typedef struct

{

int top;

int value[Maxsize];

}stack;//运算时用来存储临时结果的栈

stack st;

typedef struct btnode

{

char data;

struct btnode \*lchild;

struct btnode \*rchild;

}BTNode;//声明二叉树节点结构

BTNode \*CreateBTNode(char \*str)

{

BTNode \* EXPT[Maxsize];//定义栈EXPT，用于存储未处理的算式

BTNode \* OPTR[Maxsize];//定义栈OPTR，用于处理未处理的算符

BTNode \*b;//定义二叉树的根节点

int topEXPT = -1;//EXPT栈顶指针

int topOPTR = -1;//OPTR栈顶指针

int j = 0;

char ch=str[j];//定义字符数组，记录表达式

while (ch != '\0')//遍历运算表达式的每一个字符

{

BTNode \*node;

node = (BTNode \*)malloc(sizeof(BTNode));//生成一个新的二叉树节点

switch (ch)

{

case'+':

case'-':node->data = ch;

if (topOPTR != -1)//处理操作符栈内的加减运算符

{

OPTR[topOPTR]->rchild = EXPT[topEXPT];

OPTR[topOPTR]->lchild = EXPT[topEXPT - 1];

topEXPT--;

EXPT[topEXPT] = OPTR[topOPTR];

topOPTR--;

}

topOPTR++;

OPTR[topOPTR] = node;//记录该节点的内容

break;

case'\*':

case'/':node->data = ch;

node->lchild = EXPT[topEXPT];

topOPTR++;

OPTR[topOPTR] = node;//记录该节点的内容

break;

default:node->data = ch;

node->lchild = node->rchild = NULL;

topEXPT++;

EXPT[topEXPT] = node;//记录该节点的内容

if (topOPTR>-1 &&(OPTR[topOPTR]->data == '\*' || OPTR[topOPTR]->data == '/'))

//处理乘除运算符前后的两个数

{

OPTR[topOPTR]->rchild = EXPT[topEXPT];

topEXPT--;

EXPT[topEXPT] = OPTR[topOPTR];

topOPTR--;

}

}

j++;

ch = str[j];

}

if (topOPTR != -1)//如果最后一个运算符是加减运算符，则进行相应的处理

{

OPTR[topOPTR]->rchild = EXPT[topEXPT];

OPTR[topOPTR]->lchild = EXPT[topEXPT - 1];

topEXPT--;

EXPT[topEXPT] = OPTR[topOPTR];

topOPTR--;

}

b = EXPT[topEXPT];

return b;

}

void PostOrder(BTNode \*&b)

//进行遍历和运算，找到需要进行运算的每一个元素，如果扫描的双亲节点为运算符，那么就根据运算符的内容对数值进行操作。

{

if (b != NULL)

{

PostOrder(b->lchild);

PostOrder(b->rchild);

switch (b->data)

{

case '+':result = st.value[st.top] + st.value[st.top - 1];

st.top--;

st.value[st.top] = result; break;

case '-':result = st.value[st.top-1] - st.value[st.top];

st.top--;

st.value[st.top] = result; break;

case '\*':result = st.value[st.top] \* st.value[st.top - 1];

st.top--;

st.value[st.top] = result; break;

case '/':result = st.value[st.top-1] / st.value[st.top];

st.top--;

st.value[st.top] = result; break;

default:st.top++;

st.value[st.top] = b->data-48;//将字符还原为对应的数值

}

}

}

void CalculateBTNode(BTNode \*b)//运算算法

{

st.top = -1;

result = 0;

PostOrder(b);

printf("\n该算式的运算结果是:%d\n",result);

}

int main()

{

BTNode \*b;

char str[Maxsize];

printf("下面是用二叉树表示的算式的运算结果\n");

printf("请输入运算表达式，只能包含加减乘除和数字,且不要试图使除数为0,长度不超过%d\n", Maxsize - 1);

scanf\_s("%s", str, Maxsize - 1);

b=CreateBTNode(str);//创建算式二叉树

CalculateBTNode(b);//进行运算

system("PAUSE");

return 0;

}