项目说明文档

数据结构课程设计

——表达式计算

作 者 姓 名： 曹晓慈

学 号： 2052844

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

## 分析

### 背景分析

表达式求值是程序设计语言编译中的一个最基本问题，就是将一个表达式转化为逆波兰表达式并求值。具体要求是以字符序列的形式从终端输入语法正确的，不含变量的整数表达式，并利用给定的优先关系实现对算术四则混合表达式的求值，并延时在求值过程中运算符栈，操作数栈，输入字符和主要操作变化过程。

要把一个表达式翻译成正确求值的一个机器指令序列，或者直接对表达式求值，首先要能正确解释表达式。任何一个表达式都是由操作符，运算符和界限符组成，我们称它们为单词。一般来说，操作数既可以是常数，又可以是被说明为变量或常量的标识符；运算符可以分成算术运算符，关系运算符和逻辑运算符3类；基本界限符有左右括号和表达式结束符等。为了叙述的简洁，我们仅仅讨论简单算术表达式的求值问题。这种表达式只包括加，减，乘，除4种运算符。

人民在书写表达式时通常采用的是“中缀”表达形式，也就是将运算符放在两个操作数中间，用这种“中缀”形式表示的表达式称为中缀表达式。但是，这种表达式表示形式对计算机处理来说是不大合适的。对于表达式的表示还有另一种形式，称之为“后缀表达式“，也就是将运算符紧跟在两个操作书的后面。这种表达式比较合适计算机的处理方式，因此要用计算机来处理，计算表达式的问题，首先要将中缀表达式转化成后缀表达式，又称为逆波兰表达式。

### 2.功能分析

为了实现表达式求值，本项目要求首先读入表达式（包括括号）并创建对应二叉树，其次对二叉树进行前序遍历，中序遍历，后续遍历，输出对应的逆波兰式，中序表达式和波兰表达式。

## 设计

### 数据结构设计

根据项目的功能分析可以知道，可以用二叉树的数据结构来表达表达式，用父节点作为符号位，子结点用来储存数字，即父节点的两个孩子的数据进行父节点所存储符号的计算。同时还要 创建两个栈，一个栈用来存储运算符，另一个栈用于存储中间结果。将中缀表达式的字符按照规则存入两个栈。

### 类结构设计

首先建立二叉树，故应设计树的结点（BiTNode），其中包含有储存符号的数据域，储存数值的数据域和左右两个孩子指针。同时因为该题目还要用到两个栈分别储存运算符和表达式，所以选择链式栈的抽象数据类型，所以还需要StackNode作为链式栈的结点已满足先进后出的规则。

### 成员与操作设计

typedef struct BiTNode

{

char data; //结点数据域（符号）

string combine\_data; //结点数据域（数值，可大于9）

BiTNode\* lchild, \* rchild; //左右孩子指针

}\*BiTree;

typedef struct StackNode

{

BiTree data\_tree; //存储的是二叉树

char data\_op; //存储的是符号

StackNode\* next;

}\*LinkStack;

int InitStack(LinkStack& S);

int Push\_tree(LinkStack& S, BiTree e);//二叉树入栈

int Push\_op(LinkStack& S, char e);//字符（运算符号）入栈

int Pop\_tree(LinkStack& S, BiTree& T1);//二叉树出栈

int Pop\_op(LinkStack& S, char& ch);//字符（运算符号）出栈

char GetTop\_op(LinkStack S);

char GetTop\_op(LinkStack S);//取栈顶元素

BiTree GetTop\_tree(LinkStack S);

void CreateExpTree(BiTree& T, BiTree a, BiTree b, char theta);//a是左孩子，b是右孩子,theta是数据域,创建以T为根结点的二叉树（存储符号）

void CreateExpTree\_str(BiTree& T, BiTree a, BiTree b, string theta);//创建以T为根结点的二叉树（存储数值，可大于9）a是左孩子，b是右孩子,theta是数据域

char Precede(char top, char ch);//比较运算符的优先级,top是栈顶元素，ch是需要比较的元素

void InitExpTree(char ep[], LinkStack& expt, LinkStack& optr);//创建表达式树,expt栈(根结点)，optr栈(运算符)

void PostOrder(BiTree T); //表达式树的后序遍历

void PreOrder(BiTree T);//表达式树的前序遍历

## 实现

### 符号优先级判断的实现

因为采用了栈来储存运算符，因此每次对一个符号进行判断时只需将其与栈顶符号top进行比较，当要入栈的符号优先级大于栈顶元素时，该符号直接入栈，当要入栈的符号优先级小于栈顶元素时栈顶元素出栈，对应的两个数值也出栈，再以运算符为根，将两个数值连接并存入表达式的栈中。当运算符优先级相同时优先进行前面的运算，故操作和处理与栈顶元素大于入栈的运算符一样，因此在处理时等同于大于处理。对于括号的处理，当栈顶为‘（’而要入栈的符号为‘）’，‘）’进入运算符栈，其他情况下只要入栈元素为‘）’，则同样重复栈顶元素大于入栈的运算符的操作。

核心代码：

char Precede(char top, char ch)//比较运算符的优先级,top是栈顶元素，ch是需要比较的元素

{

if (ch == ')' && top == '(') return '=';

else if (ch == ')') return '>';

if (top == ' ' || top == '(' || ch == '(') return '<';

if (ch == '#') return '>';

if (top == '+' || top == '-') {

if (ch == '+' || ch == '-') return '>';

else if (ch == '/' || ch == '\*') return '<';

}

else if (top == '\*' || top == '/') return '>';

}

switch (Precede(GetTop\_op(optr), ep[i])) //比较优先级

{

case '<':

Push\_op(optr, ep[i]);

i++;

break;

case '>':

Pop\_op(optr, ch); //弹出上一个字符

Pop\_tree(expt, T1); //弹出上两个数值（二叉树）

Pop\_tree(expt, T2);

CreateExpTree(T, T2, T1, ch); //以data\_tree为根，连接T1和T2两颗子树

Push\_tree(expt, T); //最后把T放进expt栈中

break;

case '=':

Pop\_op(optr, ch);

i++;

break;

default:

break;

}

### 链式栈的实现

链式栈的思想和创建一个带头指针的链式线性表的思想非常相似，头指针指向栈顶指针。本题目中用到的具体操作不多，只需要入栈出栈和取栈顶元素的操作。入栈的具体操作是创建一个新的链式栈结点，把入栈的data赋值给它，新结点的next指针指向原栈顶元素，并将栈的地址更新为栈顶元素的地址。出栈则是取栈顶元素，将它的next指针指向的结点作为栈顶结点，再将该结点释放掉。

核心代码：

int Push\_op(LinkStack& S, char e)//字符（运算符号）入栈

{

LinkStack p = new StackNode;

p->data\_op = e;

p->next = S;

S = p;

return 1;

}

int Pop\_op(LinkStack& S, char& ch)//字符（运算符号）出栈

{

if (S == NULL) return 0;

LinkStack p = S;

ch = p->data\_op;

S = S->next;

delete p;

return 1;

}

### 表达式树的创建实现

从表达式的开始遍历到结束，遇到数值就创建根结点数据域为该数值的二叉树，然后进栈；遇到运算符就比较优先级，比较结果是’<’的话运算符进栈，比较结果是’>’的话把上一个运算符弹出和二叉树栈弹出的树连接起来，然后再把结果进栈，比较结果是’=’的话弹出上一个运算符。

核心代码：

void InitExpTree(char ep[], LinkStack& expt, LinkStack& optr)//创建表达式树,expt栈(根结点)，optr栈(运算符)

{

int n = strlen(ep);

int i = 0;

BiTree T = NULL; //树根

BiTree T1 = NULL; //左子树

BiTree T2 = NULL; //右子树

char ch = ' '; //弹出的符号

string combine\_str = ""; //存储数值，可大于9

while (i != n)

{

if (ep[i] >= '0' && ep[i] <= '9') { //数值（二叉树），进入expt栈中

combine\_str += ep[i];

if (ep[i + 1] >= '0' && ep[i + 1] <= '9') { //下一位仍是数字，需连接

i++;

continue;

}

CreateExpTree\_str(T, NULL, NULL, combine\_str);

combine\_str = ""; //建完数值的二叉树后string要置空

Push\_tree(expt, T);

i++;

}

else {

switch (Precede(GetTop\_op(optr), ep[i])) //比较优先级

{

case '<':

Push\_op(optr, ep[i]);

i++;

break;

case '>':

Pop\_op(optr, ch); //弹出上一个字符

Pop\_tree(expt, T1); //弹出上两个数值（二叉树）

Pop\_tree(expt, T2);

CreateExpTree(T, T2, T1, ch); //以data\_tree为根，连接T1和T2两颗子树

Push\_tree(expt, T); //最后把T放进expt栈中

break;

case '=':

Pop\_op(optr, ch);

i++;

break;

default:

break;

}

}

}

}

### 波兰式和逆波兰式的实现

波兰式是二叉树前序遍历的结果，逆波兰式则是二叉树后序遍历的结果，因此得到表达式二叉树之后，只需要对其进行前序遍历和后序遍历即可。在这里为了代码的简便选择了递归的算法。

核心代码：

void PreOrder(BiTree T)//表达式树的前序遍历

{

if (T)

{

if (T->lchild == NULL && T->rchild == NULL) {

cout << T->combine\_data ;

}

else cout << T->data ;

PostOrder(T->lchild);

PostOrder(T->rchild);

}

}

void PostOrder(BiTree T) //表达式树的后序遍历

{

if (T)

{

PostOrder(T->lchild);

PostOrder(T->rchild);

if (T->lchild == NULL && T->rchild == NULL) {

cout << T->combine\_data;

}

else cout << T->data ;

}

}

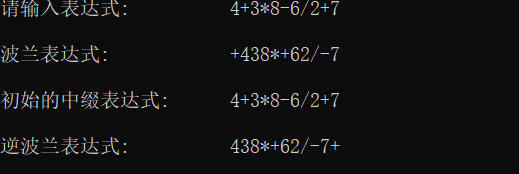
## 测试

### 1.

**测试用例：**个位数输入，不含括号

**预期结果：**程序给出正确结果

**实验结果：**

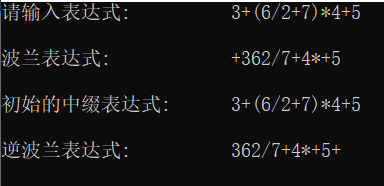


### 2.

**测试用例：**个位数输入，含括号

**预期结果：**程序给出正确结果

**实验结果：**

****

### 3.

**测试用例：**个位数以上输入，含括号

**预期结果：**程序给出正确结果

**实验结果：**

