**武汉大 学计算机学院**

**本科生实验报告**

**数据结构实验报告**

**实验七：利用二叉树求解表达式的值**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：数据结构

指 导 教 师 ：安 扬

学 生 学 号 ：2017301500061

学 生 姓 名 ：彭 思 翔

学 生 班 级 ：计科二班

上 机 环 境 ：Visual Studio Code

二○一八 年 11 月

**一、实验题目**

实验七：利用二叉树求解表达式的值

【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。

**二、实验项目的**

深入掌握二叉树得应用并加深对表达式求值得理解。

1. **实验项目程序结构**

build

main

work

caculate

**四、实验项目中各文件函数功能描述**

void work(); //读取表达式并预处理和计算

double caculate(node\*); //计算表达式树

node\* build(int, int, char\*); //构建表达式树

**五、算法描述**

【数据结构】

表达式二叉树：下面得结构代表树得每个节点，其中lc为左子树地址，rc为右子树地址，当该节点为叶子节点时d中存放数字（字符形式），为非叶子节点时d中存放运算符。表达式树的运算规则是从叶子节点向根节点逐层计算。

typedef struct node{

char d;

struct node \*lc;

struct node \*rc;

} node;

【设计思路】

表达式树的构建：提前计算一个数组pair[i]，表示表达式第i位s[i]的匹配，预处理时只有当当前位为右括号时pair[i]为其对应左括号的位置，其他位pair[i]=i。这样做的好处是可以将括号内的部分作为新的表达式递归处理，其整体则作为上一个递归的一个数。在构建表达式中间这样维护pair，对于s[i]为运算符时pair是其左边（包括自己）最靠右的‘+’或‘-’的位置，即s[j]==‘+’或‘-’,且0<=j<=i。注意这里的左边的运算符不包括括号里的，因为一对()在这一层递归正如前面所说看作是一个数。这样做的好处一是满足乘法除法优先，二是防止递归时重复找。为了清晰看到pair的功能，我们做以下表格：

|  |  |
| --- | --- |
| **S[i]** | **Pair[i]** |
| ( | 指向自己，即本身i |
| ) | 指向匹配的左括号的位置 |
| + | 指向位置<=i的第一个加减（括号内不算），即本身i |
| - | 指向位置<=i的第一个加减（括号内不算），即本身i |
| \* | 指向位置<=i的第一个加减（括号内不算） |
| / | 指向位置<=i的第一个加减（括号内不算） |
| 0~9 | 指向自己，即本身i |

那么pair如何发挥功能的呢，我们看下面的递归。

在表达式构建时递归处理一段表达式字符串S[L~R]：

1. 边界条件α：L>R 表示当前表达式为0，构建节点‘0’，返回此节点。
2. 边界条件β：L==R 表示当前表达式为一个单独的数，构建节点S[L]，返回此节点。
3. 如果最右端运算符s[i]的pair[i]为本身，则从当前位向前找到第一个‘+’或‘-’，寻找时括号用pair跳过，找到的位置为addpos。将addpos后到i所有运算符的pair都赋值为addpos。

如果addpos不为-1：表示前面有加减，根据计算规则，应该先乘除后加减，所以乘除应作为子节点，加减应作为父节点。构建节点s[addpos]，递归调用S[L~addpos-1]作为左子树节点，递归调用S[addpos+1~R]作为右子树节点，返回此节点。

例如：3\*2+4\*5\*6应构建成

3\*2

4\*5\*6

+

如果addpos为-1：表示前面没有加减，所有运算同级别，根据计算规则，应该先计算前面的运算，所以前面的运算应作为子节点，当前位应作为父节点。构建节点s[i]，递归调用S[L~i-1]作为左子树节点，递归调用S[i+1~R]作为右子树节点，返回此节点。

例如：4\*5\*6应构建成

4\*5

6

\*

注意当前位s[i]是加减符号并不矛盾，因为找到的是自身。

例如：4\*5+6会构建成

4\*5

6

+

1. 如果最右端运算符s[i]的pair[i]不为本身，说明当前位运算符是乘除，并且已经在情况3中被作为右子树递归调用了，当前S[L~R]中没有‘+’或‘-’，所以可以放心直接按照前面运算符优先的方式处理而不必考虑加减。

例如：4\*5/6会构建成

4\*5

6

/

如此递归调用即可构建出整颗表达式二叉树，由于每个字符最多访问两次，即作为当前表达式最右的运算符访问一次，作为在寻找加减号addpos时访问一次，所以时间复杂度是O(N)的。对于预处理pair至于加入栈找到右括号匹配的左括号即可，比较简单不再赘述，时间复杂度也是O(N)。

时间复杂度O(N)

node\* build(int l, int r, char\* s) {

node \*x;

x = (node\*)malloc(sizeof(node));

if (l > r) { //边界条件

x->d = '0';

x->lc = x->rc = NULL;

return x;

}

if (l == r) {

x->d = s[l];

x->lc = x->rc = NULL;

return x;

}

if (s[r] == ')' && pair[r] == l)

return build(l + 1, r - 1, s);

if (pair[pair[r] - 1] == pair[r] - 1) { //未找过addpos

int addpos = -1;

for (int i = pair[r] - 1; i >= l; i = pair[i - 1] - 1)

if (s[i] == '+' || s[i] == '-') {

addpos = i;

break;

}

for (int i = pair[r] - 1; i > addpos; i = pair[i - 1] - 1)

pair[i] = addpos;

if (addpos != -1) { //找到了则先乘除后加减

x->d = s[addpos];

x->lc = build(l, addpos - 1, s);

x->rc = build(addpos + 1, r, s);

return x;

}

else { //未找到则优先前面的运算符

x->d = s[pair[r] - 1];

x->lc = build(l, pair[r] - 2, s);

x->rc = build(pair[r], r, s);

return x;

}

}

x->d = s[pair[r] - 1];

x->lc = build(l, pair[r] - 2, s);

x->rc = build(pair[r], r, s);

return x;

}

表达式树的计算：后序遍历表达式树，将左右两个子树的结果作为当前运算的两元，返回计算结果即可。由于每个节点访问一次，时间复杂度清晰明了。

时间复杂度O(N)

double caculate(node\* x) {

if (x->d >= '0' && x->d <= '9')

return (x->d - '0') \* 1.0;

double y = 0;

switch (x->d) {

case '+': y = caculate(x->lc)+caculate(x->rc); break;

case '-': y = caculate(x->lc)-caculate(x->rc); break;

case '\*': y = caculate(x->lc)\*caculate(x->rc); break;

case '/': y = caculate(x->lc)/caculate(x->rc); break;

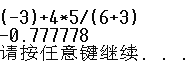
}

return y;

}

**六、实验数据和实验结果分析**

运行结果良好。



**七、实验体会**

这次实验在构建表达式树的时候挺暴力的，只有一个将括号递归的匹配，但是一想不能这么暴力，所以想办法利用递归构造。我发现了优先级和递归调用的关系，正如上文所描述的那样。然而每次找左边第一个加减号挺消耗时间的，所以充分利用了pair数组的剩余空间，记录了左边的第一个加减。这样既节省了空间，又节省了时间。所以pair其实是两个标记数组合二为一的，略微复杂。