数据结构课程设计

项目说明文档

表达式计算

|  |  |
| --- | --- |
| 作者姓名： | 高逸轩 |
| 学 号： | 2053385 |
| 指导教师： | 张 颖 |
| 学院专业： | 软件学院 软件工程 |



同济大学

Tongji University

# 1项目分析

## 1.2 项目需求分析

针对于算术表达式求值问题，本项目在设计过程中，应当考虑到满足以下需求：

* 功能完善

系统需要满足所要求的各种运算，包括：加减乘除四则运算，同时也需要能够处理包括正负在内的单目运算符、括号的存在等。

* 执行效率高

针对表达式相对比较复杂的情况，本系统也应该具有在较短时间内求解出正确答案的能力。

## 1.3 项目要求

### 1.3.1 功能要求

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。

程序需支持：包括加减乘除四则运算，能够**实现多位整数、小数的存储**，而不仅仅是简单的个位正整数数字的运算。

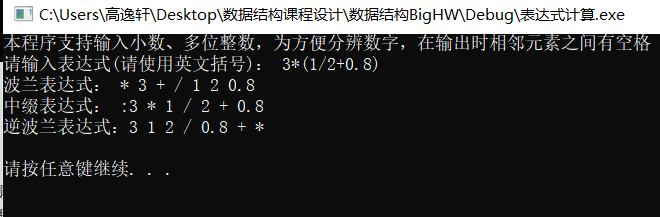
### 1.3.2 输入格式

中缀算数表达式，包括括号和上述运算符。

### 1.3.3 输出格式

表达式的前缀、中缀、后缀表达式

### 1.3.4 项目示例



# 2 项目设计

## 2.1 数据结构设计

如上述功能分析所示，本项目实际需要由一个中缀表达式建立二叉树，并实现二叉树的前序、中序、后序遍历。

本题采用的思路是，采用一个运算符号栈，遍历整个表达式，先将中缀表达式转换为逆波兰表达式。然后再遍历逆波兰表达式，将其转换为二叉树储存。之后再将表达式二叉树进行前序、中序、后序遍历。

本系统各种操作的时间复杂度如下：

* 计算表达式结果：O(n)

## 2.2 类设计

在本项目中使用到了栈（Stack）这一数据结构，在STL中，栈的底层数据类型采用了链表进行实实现。在本题中，由于栈只为辅助工具，采取了数组简单模拟。

而二叉树的结构则分别由节点类（存储了当前节点的数据、左右儿子地址等数据）、二叉树类共同构成。部分代码如下：

### 2.2.1 节点类

// 记录节点记录的是数字还是运算符

enum { isNumber, isOperation };

// 每个节点的信息

struct info

{

    info()

    {

        value = 0.0;

        operation = '\0';

        infoType = 0;

    }

    double value;                                    // 数字的值

    char operation;                                  // 运算符

    bool infoType;                                   // 记录节点信息

};

// 二叉树节点

struct BinaryTreeNode

{

    // 构造函数，为左右儿子赋值

    BinaryTreeNode(BinaryTreeNode\* lc = NULL, BinaryTreeNode\* rc = NULL) :leftChild(lc), rightChild(rc) {};

    // double to BinaryTreeNode 显式转换

    BinaryTreeNode(double v) { data.value = v; data.infoType = isNumber; }

    // char to BinaryTreeNode 显式转换

    BinaryTreeNode(char o) { data.operation = o; data.infoType = isOperation; }

    BinaryTreeNode\* leftChild = NULL, \* rightChild = NULL;// 左右儿子

    info data;                                            // 节点存储信息

};

// 重载输出运算符

ostream& operator<<(ostream& os, const BinaryTreeNode& x)

{

    // 根据该节点存储的数据类型进行输出

    if (x.data.infoType == isOperation) os << x.data.operation;

    if (x.data.infoType == isNumber) os << x.data.value;

    return os;

}

可以注意到的是，由于节点中存储的信息可能是一个运算符，也有可能是一个数字，所以在信息Info中，采用了bool 类型的infoType 变量以及enum{isNumber,isOperation}来辅助判断。

### 2.2.2 二叉树

/ 二叉树

class BinaryTree

{

public:

    // 构造函数：利用输入的字符串表达式s建立表达式树

    // 思路：先将原表达式转化为逆波兰式，再将逆波兰式转化为表达式树

    BinaryTree(string& s)

// 析构函数

~BinaryTree()

  // 以下为三种遍历输出的方式，为保证输出后可以分辨小数、多位整数，每个元素之间多输出了一个空格以分割

    // 前序遍历

    void preOrder(BinaryTreeNode\* current)

    // 中序遍历

    void inOrder(BinaryTreeNode\* current)

    // 后序遍历

    void postOrder(BinaryTreeNode\* current)

    BinaryTreeNode\* Root() { return root; }

private:

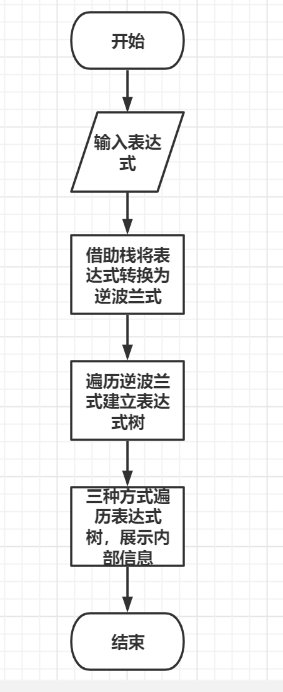
    BinaryTreeNode\* root;

    // 运算符优先级比较

    bool CompareOperator(const char a, const char b)

};

## 2.3 项目流程图



# 3 核心代码介绍

## 3.1 运算符优先级比较

// 运算符优先级比较

    bool CompareOperator(const char a, const char b)

    {

        if (a == '\*' || a == '/')

        {

            if (b == '(') return 0;

            else return 1;

        }

        else if (a == '+' || a == '-')

        {

            if (b == '+' || b == '-' || b == ')') return 1;

            else return 0;

        }

        else if (a == '(')  return 0;

    }

在进行操作前，由于运算符出栈入栈时需要对运算符的优先级进行比较，所以前置编写了CompareOperator()函数，用来比较各运算符之间的优先级高低。由于右括号不会入栈，所以仅需要比较加减乘除、左括号之间关系即可。分别是左括号 > 除法 = 乘法 > 加法 = 减法。本函数的参数a表示栈顶元素，b表示将要入栈的元素。当返回值为0时，代表b入栈；当返回值为1时，代表a出栈。

## 3.2 构建二叉树

构造函数会利用输入的字符串表达式s建立表达式树。

思路为：先将原表达式转化为逆波兰式，再将逆波兰式转化为表达式树

### 3.2.1 将中缀表达式转换为逆波兰式

    BinaryTree(string& s)

    {

        /\*将中缀表达式转换为逆波兰表达式\*/

        char stack1[1024];                // 存放运算符的栈

        int top1 = 0;                     // 栈顶

        int lengthRPN = 0;              // 逆波兰表达式长度

        BinaryTreeNode RPN[1024];         // 存放逆波兰表达式

        // temp用以提取expression中的高位数字和小数

        string temp = "";

        // 将表达式整体套上括号

        s += ')';

        stack1[top1++] = '(';

        // 遍历表达式

        for(int i=0;i<s.length();i++)

        {

            // 表达式该位置为一个数字，记录进temp

            if (isdigit(s[i]))

                temp += s[i];

            // 小数点直接记入temp

            else if (s[i] == '.')

                temp += s[i];

            // 表达式该位置为运算符

            else

            {

//若temp不为空，需要先将现在temp转化为数字节点后加入逆波兰式，并将temp重置

                if (temp != "")

                {

// 利用atof函数和c\_str 实现 char\* to double 与 string to char\*

                    RPN[lengthRPN++] = BinaryTreeNode(atof(temp.c\_str()));

                    temp = "";

                }

                // 遇到左括号直接入栈

                if (s[i] == '(') stack1[top1++] = s[i];

                // 遇到右括号则弹出栈中运算符并进行运算，直到遇到左括号为止

                else if (s[i] == ')')

                {

                    // 栈为空，则直接进入下一次循环，不必进行左括号匹配

                    if (!top1) continue;

                    // 弹出栈内运算符直到匹配到左括号

                    while (stack1[top1 - 1] != '(')

                    // 将栈顶运算符转换为节点后加入逆波兰式，同时弹出栈顶元素

                        RPN[lengthRPN++] = BinaryTreeNode(stack1[--top1]);

                    top1--; // 左括号出栈

                }

                else if (CompareOperator(stack1[top1 - 1], s[i]) == 0) stack1[top1++] = s[i];// 栈顶元素优先级小于当前元素，当前元素入栈

                else if (CompareOperator(stack1[top1 - 1], s[i]) == 1)             // 栈顶元素优先级大于当前元素，出栈

                {

                    RPN[lengthRPN++]；BinaryTreeNode(stack1[--top1]);                     // 将栈顶运算符转换为节点后加入逆波兰式，同时弹出栈顶元素

                    i--;                                                                     // i--，重新对此运算符重复上述操作，直到遇到发现更低优先级的 元素(或者栈为空)为止

                }

            }

        }

在将中缀表达式s转化为逆波兰式RPN的过程中，我们借助了数组模拟的符号栈stack1和栈顶指针top1，用来暂时存放运算符。

为了达到能够实现存储多位整数和小数的功能，我们采用字符串temp存储由字符串暂时转换而来的数字。在遍历表达式s时，每当遇到一个运算符，则表明生成了一个完整的数字。我们利用c\_str()函数，将temp类型从string 转换为char\* ，再利用atof()函数将char\* 转换为double 型的数字，这样就成功生成了一个存储数字的节点。在将其存储为数字节点时，要将节点的data中的infoType记录为isNumber。

在遍历表达式s之前，将其整体套上一个括号，可以标志为遍历结束的标志。在遍历时，遇到一个新的运算符时，首先利用Compare()函数比较其与栈顶部元素的优先级高低。根据判断情况执行操作：将新的运算符入栈，或者是不断弹出栈顶运算符，直到栈为空或者遇到更低优先级的运算符，再将其入栈。

### 3.2.2 将逆波兰式转换为二叉树

/\*将逆波兰式转换为表达式树\*/

        BinaryTreeNode stack2[1024];     // 存放节点的栈

        int top2 = 0;                    // 栈顶

        // 遍历逆波兰式

        for(int i=0;i<lengthRPN;i++)

        {

            // 遇到数字，直接入栈

            if (RPN[i].data.infoType == isNumber) stack2[top2++] = RPN[i];

            // 遇到运算符

            if (RPN[i].data.infoType == isOperation)

            {

  BinaryTreeNode\* rc = new BinaryTreeNode(stack2[--top2]);  // 取栈顶元素为rightChild

                BinaryTreeNode\* lc = new BinaryTreeNode(stack2[--top2]);  // 取次顶元素为leftChild

                BinaryTreeNode\* parent = new BinaryTreeNode(RPN[i]);      // 记录父节点为当前运算符节点

        MergeTree(\*lc, \*rc, \*parent);       // 构建父子关系

        stack2[top2++] = \*parent;           // parent入栈

         root = parent;                      // 更新根节点

            }

        }

    }

在将逆波兰表达式转换为表达式二叉树时，需要借助栈stack2和栈顶指针top2，暂时存储各节点。

逐个遍历逆波兰式RPN，遇到数字后直接入栈，遇到运算符则取出栈顶两个元素，分别作为运算符的左右子树，构建为二叉树的父子关系，再将运算符节点入栈。同时，每次建立父子关系后，需要将root的指针指向最先产生的父亲节点，这样最终遍历结束后，root即指向最后的父节点。

## 3.3 二叉树的遍历

// 前序遍历

    void preOrder(BinaryTreeNode\* current)

    {

        if (current == NULL) return;

        cout << \*current << ' ';

        preOrder(current->leftChild);

        preOrder(current->rightChild);

    }

    // 中序遍历

    void inOrder(BinaryTreeNode\* current)

    {

        if (current == NULL) return;

        inOrder(current->leftChild);

        cout << \*current << ' ';

        inOrder(current->rightChild);

    }

    // 后序遍历

    void postOrder(BinaryTreeNode\* current)

    {

        if (current == NULL) return;

        postOrder(current->leftChild);

        postOrder(current->rightChild);

        cout << \*current << ' ';

    }

在建立完表达式二叉树后，需要以三种方式输出其内部信息。这里采取递归的方式，前、中、后三种遍历方式取决于当前节点与向下递归左右儿子的顺序关系。

**注：为体现二叉树的中序遍历，在中缀表达式中也将括号去除，以保证中序遍历时输出的是二叉树中的信息。**

# 4 项目测试

为了方便老师进行测试，提供了文件7\_test.txt，内含有一组测试数据。

# 5. 心得体会

计算机科学中，除了栈以外，[二叉树](https://so.csdn.net/so/search?from=pc_blog_highlight&q=%E4%BA%8C%E5%8F%89%E6%A0%91)也是处理表达式的常用工具，为了处理表达式而遵循相应规则构造的树被称为表达式树。为处理我们输入的表达式，计算机也将其转换为逆波兰表达式再计算。在本题的任务中，我学会了如何借助栈将中缀表达式转换为逆波兰表达式，也学会了借助栈将逆波兰表达式转换为表达式树。

由于篇幅原因，报告内还有很多内容与解释没有展示，请老师和助教老师再移步源程序，在其中的注释写了每一步过程的详解。