项目说明文档

数据结构课程设计

——表达式计算

作 者 姓 名： 刘畅

学 号： 2054164

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 分析 1](#_Toc88064327)

[1.1 背景分析 1](#_Toc88064328)

[1.2 功能分析 1](#_Toc88064329)

[2 设计 2](#_Toc88064330)

[2.1 数据结构设计 2](#_Toc88064331)

[2.2 类结构设计 2](#_Toc88064332)

[2.3 成员与操作设计 2](#_Toc88064333)

[2.4 系统设计 2](#_Toc88064334)

[3 实现 3](#_Toc88064335)

[3.1 表达式转换为二叉树的实现 3](#_Toc88064336)

[3.1.1 操作数、操作符的读入 3](#_Toc88064337)

[3.1.2 运算符优先级 3](#_Toc88064338)

[3.1.3 二叉树的构建 4](#_Toc88064339)

[3.1.4 核心代码 4](#_Toc88064340)

[3.2 前缀、中缀、后缀表达式的输出 5](#_Toc88064341)

[3.2.1 实现方法 5](#_Toc88064342)

[3.2.2 核心代码 6](#_Toc88064343)

[3.3 二叉树模板类的实现 6](#_Toc88064344)

[3.2.1 基础功能 6](#_Toc88064345)

[3.2.2 添加子女功能 6](#_Toc88064346)

[3.2.3 生长树功能 7](#_Toc88064347)

[3.2.3 删除结点功能 7](#_Toc88064348)

# 1 分析

## 1.1 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。

## 1.2 功能分析

为了实现表达式求值，本项目首先读入表达式（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。关键在于如何根据表达式的中缀形式（中缀形式的表达式可以带有括号）转换为对应的二叉树。

# 2 设计

## 2.1 数据结构设计

中缀表达式转换为二叉树的过程中使用栈结构，因此定义一个模板类来封装栈的实现。在生成二叉树的过程中需要不断将二叉树进行扩充，具体可以分为向左上扩充和向右上扩充，因此在二叉树模板类中添加相关函数。

## 2.2 类结构设计

## 2.3 成员与操作设计

**栈类（LinkedStack）**

int getSize();

void Push(const T& data);//添加元素

bool Pop(T& data);//弹出栈顶元素

bool getTop(T& data)const;//获取栈顶元素

bool empty()const;//判断栈是否为空

void erase();//清空栈

**二叉树类（BinaryTree）**

bool setRoot(T& data);

bool IfEmpty();

BinaryTreeNode<T>\* getRoot()const { return \_root; };

bool InsertLeftChild(BinaryTreeNode<T>\* tar, T&data);//插入左孩子

bool InsertRightChild(BinaryTreeNode<T>\* tar, T&data);//插入右孩子

bool UpLeftGrow(T&data\_parent, T&data\_leftchild);//向左上生长

bool UpRightGrow(T&data\_parent, T& data\_rightchild); //向右上生长

void Remove(BinaryTreeNode<T>\* tar);//删除结点

void preOrder(BinaryTreeNode<T>\* tar, void(\*visit)(BinaryTreeNode<T>\* p));//前序遍历

void inOrder(BinaryTreeNode<T>\* tar, void(\*visit)(BinaryTreeNode<T>\* p));//中序遍历

void postOrder(BinaryTreeNode<T>\* tar, void(\*visit)(BinaryTreeNode<T>\* p));//后序遍历

## 2.4 系统设计

系统首先读入用户输入的中缀表达式（默认为正确，并且可以带有括号），扫描一遍之后直接将其转换为二叉树，复杂度可以近似认为是O(n)，而后直接进行二叉树的前中后序遍历，将其结果输出即可。

# 3 实现

## 3.1 表达式转换为二叉树的实现

### 3.1.1 操作数、操作符的读入

中缀表达式可以认为是由操作数和操作符组成，操作数代表0-9的10个数字，操作符包括+-\*/()六种。在读取过程中设置两个栈，分别存储操作数和操作符：当读到操作数的时候，直接将其入栈，当读到操作符时，先判断其与栈顶元素的优先级大小，如果栈顶元素的优先级低，则将读取到的操作符直接压入操作符栈，如果栈顶元素的优先级高，则将其出栈，并进行运算；如果两者相等，则退栈但不运算，若退出的时“（”则读入下一字符。

### 3.1.2 运算符优先级

为了使得读取操作数操作符的过程中能按照我们约定的计算法则、运算符优先级运算，设置两个函数分别表示不同运算符在栈内栈外的优先级，即isp()和icp()函数。下面列举几个例子来说明程序中isp和icp设置的合理性。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 操作符c | # | ( | \* / | + - | ) |
| isp | 0 | 1 | 5 | 3 | 6 |
| icp | 0 | 6 | 4 | 2 | 1 |

例如#，我们在输入的中缀表达式末尾添加一个#，其运算符无论在栈内还是栈外都是最低的，因此当栈顶操作符为#时，必入栈，此时即刚读入第一个操作符；例如“(”，其栈内优先级很低，因此会让左括号右边的操作符（即括号内部运算）先运算，而其栈外操作符很高，因此当读到左括号的时候会优先将其入栈并计算括号内部运算；例如“)”，其栈内优先级最高，因此当栈顶为右括号时必退栈，说明括号内运算已经完成，而其栈外优先级很低，因此当操作符为右括号时也必退栈。

再例如对于加减乘除运算，可以看到加减运算总比乘除运算优先级低，因此当栈顶为“+-”而读到“\*/”时，会进栈，反之则会退栈；而当加减或乘除自身相遇时，先进行左边的运算（即栈顶运算符），因此他们的栈内优先级高于栈外优先级。

### 3.1.3 二叉树的构建

对于一个表达式而言，其最简单的二叉树应该以运算符为根节点，左右是两个数字，如左1根+右2，表明1+2；二叉树应当从最下层开始运算，运算结果再作为上层树的运算对象。因此，二叉树的构建也应当根据谁先运算、自下而上地构建。

由3.1.1的读取方式我们可以知道，每次操作符出栈时，操作数栈的最外面两个数字必定是该操作符的操作对象，并且满足我们日常约定的运算顺序。因此可以在每次退栈的时候，利用这三个元素构建一颗二叉树，并将其添加到总二叉树上。

### 3.1.4 核心代码

Main.cpp

cin >> infix\_exp;

infix\_exp += '#';

string::const\_iterator citer = infix\_exp.cbegin();

while (citer + 1 != infix\_exp.cend() || !stack\_opt.empty()) {

if ('0' <= (\*citer) && '9' >= (\*citer)) {

//如果是操作数，直接存入数字栈

stack\_num.Push(\*citer);

//cout << \*citer;

citer++;

}

else {

//如果是操作符，先看操作符栈顶元素，并判断优先级

stack\_opt.getTop(cnew);

if (isp(cnew) < icp(\*citer)) {

//如果栈顶元素的优先级低，则将读到的元素直接入栈

stack\_opt.Push(\*citer++);

}

else if (isp(cnew) > icp(\*citer)) {

//如果栈顶元素的优先级高，则将其出栈

stack\_opt.Pop(opt);

//以opt为顶点，stack\_num外侧的两个为左右儿子建树

//#表示上一次运算的结果，若取出来的数为#的话，说明要长树

char leftchild, rightchild;

stack\_num.Pop(rightchild);

stack\_num.Pop(leftchild);

if (leftchild == '#') {

//如果左儿子是#，向右上方长树

tree\_exp.UpRightGrow(opt, rightchild);

stack\_num.Push('#');

}

else if (rightchild == '#') {

//如果右儿子是#，向左上方长树

tree\_exp.UpLeftGrow(opt, leftchild);

stack\_num.Push('#');

}

else {

//都是纯数字，构建子树，并用#代表本次运算的结果子树

tree\_exp.setRoot(opt);

tree\_exp.InsertLeftChild(tree\_exp.getRoot(), leftchild);

tree\_exp.InsertRightChild(tree\_exp.getRoot(), rightchild);

stack\_num.Push('#');

}

}

else {

stack\_opt.Pop(opt);

if (opt == '(') {

citer++;

}

}

}

}

int isp(char c) {

if (c == '(')return 1;

else if (c == '\*' || c == '/')return 5;

else if (c == '+' || c == '-')return 3;

else if (c == ')')return 6;

else if (c == '#')return 0;

else { return -1; }

}

int icp(char c) {

if (c == '(')return 6;

else if (c == '\*' || c == '/')return 4;

else if (c == '+' || c == '-')return 2;

else if (c == ')')return 1;

else if (c == '#')return 0;

else { return -1; }

}

## 3.2 前缀、中缀、后缀表达式的输出

### 3.2.1 实现方法

当二叉树构建好了之后，直接对二叉树进行前、中、后序遍历即可输出对应的表达式，因此直接使用递归遍历即可。（此处也可以使用非递归遍历算法，但由于对于性能要求不高，为了省事就直接采用递归算法）

### 3.2.2 核心代码

void preOrder(BinaryTreeNode<T>\* tar, void(\*visit)(BinaryTreeNode<T>\* p)) {

if (tar) {

visit(tar);

preOrder(tar->\_leftChild, visit);

preOrder(tar->\_rightChild, visit);

}

}

void inOrder(BinaryTreeNode<T>\* tar, void(\*visit)(BinaryTreeNode<T>\* p)) {

if (tar) {

inOrder(tar->\_leftChild, visit);

visit(tar);

inOrder(tar->\_rightChild, visit);

}

}

void postOrder(BinaryTreeNode<T>\* tar, void(\*visit)(BinaryTreeNode<T>\* p)) {

if (tar) {

postOrder(tar->\_leftChild, visit);

postOrder(tar->\_rightChild, visit);

visit(tar);

}

}

cout << "前缀表达式：";

tree\_exp.preOrder(tree\_exp.getRoot(), print);

cout << endl << "中缀表达式：";

tree\_exp.inOrder(tree\_exp.getRoot(), print);

cout << endl << "后缀表达式：";

tree\_exp.postOrder(tree\_exp.getRoot(), print);

## 3.3 二叉树模板类的实现

### 3.2.1 基础功能

BinaryTree() { \_root = NULL; }

BinaryTree(T& rootdata) { \_root = new BinaryTreeNode<T>(rootdata); }

~BinaryTree() {Remove(\_root);}

bool IfEmpty() { return \_root == NULL; }

BinaryTreeNode<T>\* getRoot()const { return \_root; }

### 3.2.2 添加子女功能

/\*对于目标结点tar，添加其左子女或右子女\*/

bool InsertLeftChild(BinaryTreeNode<T>\* tar, T& data) {

if (tar == NULL)return false;

BinaryTreeNode<T>\* leftchild = new BinaryTreeNode<T>(data);

tar->\_leftChild = leftchild;

return true;

}

bool InsertRightChild(BinaryTreeNode<T>\* tar, T& data) {

if (tar == NULL)return false;

BinaryTreeNode<T>\* rightchild = new BinaryTreeNode<T>(data);

tar->\_rightChild = rightchild;

return true;

}

### 3.2.3 生长树功能

/\*传入parent和child，不妨设为左子女，则原树的根结点为右子女，向上生长\*/

bool UpLeftGrow(T& data\_parent, T& data\_leftchild) {

BinaryTreeNode<T>\* leftchild = new BinaryTreeNode<T>(data\_leftchild);

BinaryTreeNode<T>\* parent = new BinaryTreeNode<T>(data\_parent);

if (rightchild == NULL || parent == NULL) { return false; }

parent->\_leftChild = leftchild;//设置父结点的左孩子

parent->\_rightChild = \_root;//设置父节点的右孩子为根结点

this->\_root = parent;//设置根结点

return true;

}

bool UpRightGrow(T& data\_parent, T& data\_rightchild) {

BinaryTreeNode<T>\* rightchild = new BinaryTreeNode<T>(data\_rightchild);

BinaryTreeNode<T>\* parent = new BinaryTreeNode<T>(data\_parent);

if (rightchild == NULL || parent == NULL) { return false; }

parent->\_rightChild = rightchild;//设置父结点的右孩子

parent->\_leftChild = \_root;//设置父结点的左孩子为根结点

this->\_root = parent;//设置根结点

return true;

}

### 3.2.3 删除结点功能

/\*递归删除结点\*/

void Remove(BinaryTreeNode<T>\* tar) {

if (!tar) {

Remove(tar->\_leftChild);//nihao

Remove(tar->\_rightChild);

delete tar;

}

}