数据结构课程设计

项目说明文档

**表达式计算**



同济大学

Tongji University

姓名： 林觉凯

学号： 2253744

指导老师： 张颖

学院专业： 软件学院 软件工程

**目录**

**1.项目分析-------------------------------------------------------------------------3**

**1.1 项目背景分析-------------------------------------------------------------------------3**

**1.2项目功能分析--------------------------------------------------------------------------3**

**1.2.1项目功能要求--------------------------------------------------------------------------------3**

**1.2.2项目输入要求--------------------------------------------------------------------------------3**

**1.2.3项目输出要求--------------------------------------------------------------------------------3**

**1.2.4项目示例--------------------------------------------------------------------------------------3**

**2.项目设计-------------------------------------------------------------------------4**

**2.1数据结构设计--------------------------------------------------------------------------4**

**2.2类设计-----------------------------------------------------------------------------------4**

**2.2.1 栈类(stack)-----------------------------------------------------------------------------------4**

**2.2.2 二叉树结点类(BitNode)-------------------------------------------------------------------5**

**2.2.3 二叉树类(BitTree)--------------------------------------------------------------------------5**

**2.3成员与操作设计-----------------------------------------------------------------------6**

**2.4系统流程设计--------------------------------------------------------------------------6**

**3.项目实现-------------------------------------------------------------------------7**

**3.1两个基础函数的实现-----------------------------------------------------------------7**

**3.2有关二叉树成员函数的实现--------------------------------------------------------7**

**3.3二叉树子树在栈中运算函数的实现-----------------------------------------------8**

**3.4 main函数中建立二叉树的实现 --------------------------------------------------9**

**4.项目测试------------------------------------------------------------------------11**

**4.1项目功能测试-------------------------------------------------------------------------11**

**4.2项目提示信息测试-------------------------------------------------------------------12**

**5.项目的心得与体会------------------------------------------------------------12**

**1.项目分析**

**1.1 项目背景分析**

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数的后面。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 项目功能分析**

**1.2.1项目功能要求**

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。

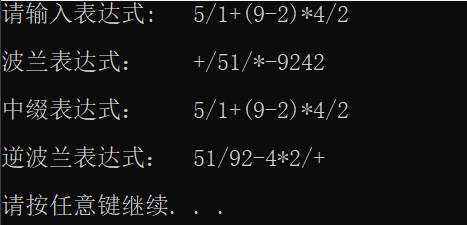
**1.2.2项目输入要求**

用户输入原始的中缀表达式（英文符号）。

**1.2.3项目输出要求**

按照要求输出波兰表达式、中缀表达式和逆波兰表达式。

**1.2.4项目示例**



**2.项目设计**

**2.1 数据结构设计**

此次项目首先需要实现根据中缀表达式构建二叉树，然后再对所构建的二叉树进行前序遍历和后序遍历得到它的前缀表达式和后缀表达式。所以，首先需要设计一个栈模板类，来达到根据中缀表达式构建二叉树所需的操作；其次，建立二叉树类，通过其前序遍历和后序遍历得到它的前缀表达式和后缀表达式。

**2.2类设计**

为了实现此次表达式的转换，我设计了三个类：栈类、二叉树结点类和二叉树类，栈的模板类根据老师的要求重新写了一遍，没有直接调用C++栈类；而后面两个类之间的耦合关系可以采用嵌套、继承等多种关系。本程序也构造了这两个基本的二叉树结点和二叉树类来进行相应的操作。

**2.2.1栈类**(stack)

class stack //栈的模板类

{

public:

stack(); //构造函数

~stack(); //析构函数

void Push(const T& item); //将元素压入栈中

T Pop(); //将栈顶元素移出

T GetTop(); //取得栈顶元素

bool is\_Empty() { return top == -1; }; //判断栈是否为空

bool is\_Full() { return top == size; }; //判断栈是否已满

private:

int top; //栈顶

int size = Max; //栈的大小

T\* element; //栈数组

};

template <typename T>

stack<T>::stack()

{

element = new T[size + 1]; //为栈数组分配空间

top = -1; //top置为-1

}

template <typename T>

stack<T>::~stack()

{

delete[] element; //删除栈数组

top = -1; //top置为-1

}

template <typename T>

void stack<T>::Push(const T& item)

{

if (is\_Full()) //如果栈满，直接return

return;

else

element[++top] = item; //将元素压入栈中，同时top加一

}

template <typename T>

T stack<T>::Pop()

{

if (top == -1) //如果栈空，退出

exit(-1);

else

return element[top--]; //将元素退出，同时top减1

}

template <typename T>

T stack<T>::GetTop()

{

if (top == -1) //如果栈空，退出

exit(-1);

else

return element[top]; //取得栈顶元素

}

**2.2.2 二叉树结点类**(BitNode)

class BitNode

{

public:

string data; //数据域（运算符或数值）

BitNode\* left; //左指针

BitNode\* right; //右指针

BitNode() { data = ""; left = right = NULL; }

};

**2.2.3 二叉树类**(BitTree)

class BitTree

{

public:

BitTree(BitNode\* node) { root = node; };

~BitTree() {};

void PreOrder(BitNode\* current); //前序遍历

void PostOrder(BitNode\* current); //后续遍历

BitNode\* root;

};

**2.3成员与操作设计**

栈类的私有成员有栈顶指针、栈数组指针和栈的最大长度；栈的公有操作有初始化栈、进栈、退栈、获得栈顶元素、判断栈空和判断栈满。二叉树结点类的公有成员为数据域、左指针和右指针；二叉树类的私有成员为根结点，公有操作为建立二叉树、二叉树的前序遍历和二叉树的后序遍历。

**2.4系统流程设计**

系统操作的整体流程大致如下：

**程序开始运行**

**输入中缀表达式**

**按照要求输出**

**波兰表达式、**

**中缀表达式、**

**逆波兰表达式**

**3.项目实现**

**3.1 两个基本函数的实现**

//判断输入的字符是否有价值（为数字或者运算符）

bool Is\_value(char ch)

{

if ((ch >= '0' && ch <= '9') || ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '(' || ch == ')')

return true; //有价值，return true

else

return false; //没价值，return false

}

//判断操作符的优先级

int Priority(char ch)

{

if (ch == '+' || ch == '-')

return 1; //'+'、'-'返回1

else if (ch == '\*' || ch == '/')

return 2; //'\*'、'/'返回2

else

return 0; //'('、')'返回0

}

这两个函数的作用分别是判断输入的字符是否有价值（为数字或者运算符）和判断操作符的优先级，前者在判断输入时是否存在非法字符中起作用（如果时非法字符，直接跳过，并给出一定的提示）；后者在判断运算符栈顶元素和当前运算符优先级时起作用，判断是否需要对number数组中的元素进行计算操作。

**3.2 有关二叉树成员函数的实现**

//二叉树的前序遍历

void BitTree::PreOrder(BitNode\* current)

{

if (current != NULL)

{

if (current != NULL)

cout << current->data;

PreOrder(current->left);

PreOrder(current->right);

}

}

//二叉树的后序遍历

void BitTree::PostOrder(BitNode\* current)

{

if (current != NULL)

{

PostOrder(current->left);

PostOrder(current->right);

if (current->left == NULL && current->right == NULL)

cout << current->data;

else

cout << current->data;

}

}

二叉树的前序遍历和后序遍历用于建立好表达式二叉树之后，输出相应的波兰表达式和逆波兰表达式。

**3.3 二叉树子树在栈中运算函数的实现**

//判断运算符的优先级之后调整二叉树子树栈中的元素（运算）

void Calculate(stack<BitNode\*> &number, stack<char> &oper)

{

BitNode\* anewnode = new BitNode; //结点anewnode为number栈中栈顶元素

anewnode = number.GetTop();

number.Pop(); //结点anewnode退栈，下一个结点bnewnode便为栈顶元素

BitNode\* bnewnode = new BitNode; //结点bnewnode为number栈中栈顶元素

bnewnode = number.GetTop();

number.Pop();

BitNode\* current = new BitNode; //新建结点current

current->left = bnewnode;

current->right = anewnode;

current->data = oper.GetTop(); //新结点current包括原先两个栈顶元素和操作符栈的运算符

number.Push(current); //将新结点压入number栈中

oper.Pop(); //刚刚做完运算的运算符栈栈顶元素退栈

}

该函数number栈中两个顶部元素的计算（计算后生成新的二叉树子树压入number数组）。先分别取栈顶的两个元素，再分别赋值给新的二叉树子树的左右指针（构建成新的子树），最后再压入原先的number栈中。

**3.4 main函数中建立二叉树的实现**

int main()

{

char expression[Max] = { 0 };

cout << "请输入表达式:\t";

cin.getline(&expression[1], Max); //得到英文的中缀表达式

cout << endl;

stack<BitNode\*>number; //number栈，存放二叉树子树

stack<char>oper; //oper栈，存放运算符

expression[0] = '(';

//此处在expression的前后各加一个括号，是为了防止当最后一个元素进number栈之后没有被插入到最后的二叉树中去

int length = strlen(expression);

expression[length] = ')';

int i = 0;

while (expression[i] != '\0') //循环结束标志

{

if (!Is\_value(expression[i])) //输入错误的检验与忽略

{

cout << "输入的第" << i++ << "位存在非法字符，计算时忽略" << endl << endl;

continue;

}

if (expression[i] >= '0' && expression[i] <= '9') //当前字符是数字，直接压到number栈中去

{

BitNode\* newnode = new BitNode;

newnode->data = expression[i];

newnode->left = NULL;

newnode->right = NULL;

number.Push(newnode);

}

else if (expression[i] == '+' || expression[i] == '-' || expression[i] == '\*' || expression[i] == '/' || expression[i] == '(' || expression[i] == ')') //如果是运算符

{

if (expression[i] == '(') //左括号压入运算符栈

oper.Push(expression[i]);

else if (expression[i] == ')')

//是右括号的话就要将括号里的元素进行计算了（合并为新的子树）

{

while (oper.GetTop() != '(') //循环条件为还没遇到前面一个左括号

Calculate(number, oper);

oper.Pop(); //最后将左括号退栈

}

else if (oper.is\_Empty() || Priority(expression[i]) > Priority(oper.GetTop()))

//如果栈为空或者当前运算字符的优先级大于运算符栈顶运算符优先级

oper.Push(expression[i]); //压入运算符栈

else if (!oper.is\_Empty() && Priority(expression[i]) <= Priority(oper.GetTop()))

//如果栈不为空或者当前运算字符的优先级小于等于运算符栈顶运算符优先级

{

while (!oper.is\_Empty() && Priority(expression[i]) <= Priority(oper.GetTop()))

//这时就要对括号里的元素进行计算（建立新的子树）

Calculate(number, oper);

oper.Push(expression[i]);

}

}

i++;

}

BitTree Exptree(number.GetTop()); //实例化表达式树

cout << "波兰表达式：\t";

Exptree.PreOrder(Exptree.root); //前序遍历输出波兰表达式

cout << endl << endl;

cout << "中缀表达式：\t";

for (int i = 1; i < length; i++) //中缀表达式

{

if (Is\_value(expression[i]))

cout << expression[i];

}

cout << endl << endl;

cout << "逆波兰表达式：\t";

Exptree.PostOrder(Exptree.root); //后序遍历输出逆波兰表达式

cout << endl << endl;

system("pause");

return 0;

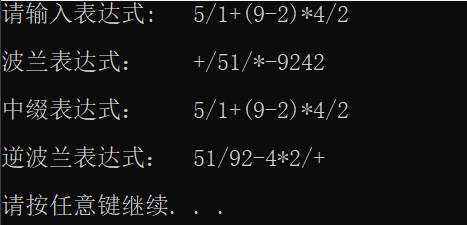
}

在main函数中主要功能的实现。首先需要读取输入的字符串。注意这里从expression的第二位expression[1]开始读取，原因时在读取字符串之后再在字符串的开头和结尾expression[0]和expression[length]分别加入左括号和右括号，原因是如果不加的话最后一个如果式数字字符会被直接压入至number栈中，不会加入到二叉树中去，注意栈顶元素就只有最后一个数字，无法对其遍历值。之后遍历整个字符串的每一个字符，根据数字还是运算符、运算符的优先级依次对应操作下去。最后依次输出波兰表达式、中缀表达式和波兰表达式。

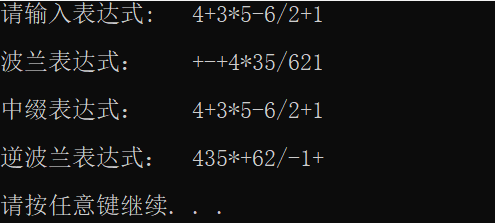
**4.项目测试**

**4.1 项目功能测试**

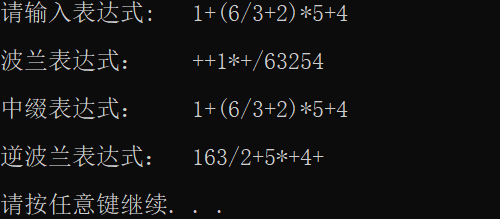
测试用例1： 5/1+(9-2)\*4/2

****

测试用例2： 4+3\*5-6/2+1

****

测试用例3： 1+(6/3+2)\*5+4

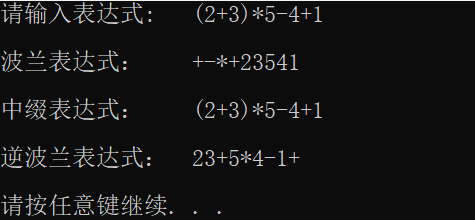
****

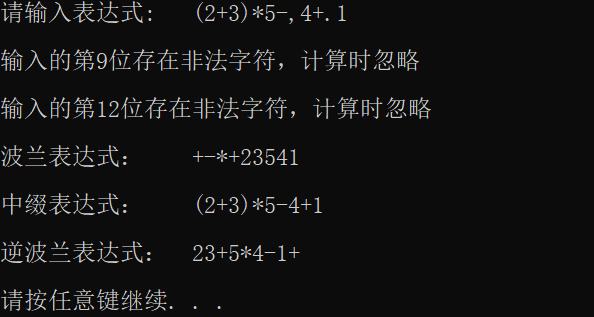
**4.2 项目提示信息测试**

测试用例：(2+3)\*5-,4+.1

即在输入字符串式中间参杂着非法字符，需要程序做出判断并且忽略，最后给出正确的波兰表达式、中缀表达式和逆波兰表达式，并给出错误提示信息。

输入(2+3)\*5-,4+.1的结果应该与输入(2+3)\*5-4+1的结果相同

****

****

**5.项目心得与体会**

本次项目我花了较多的时间完成，做了两种不同的解答方式。本次项目我按照老师的要求中间写了一遍栈的模板类再进行使用，这使我对栈的操作更加地熟悉。同时，巧妙地将前缀表达式、中缀表达式和后缀表达式与二叉树的前序遍历、中序遍历和后序遍历联系再一起，为解答此类问题提供了新的思路，而且加深了我对二叉树三种遍历方式的理解。栈的使用是这道题的精髓，通过比较运算符栈顶运算符和当前运算符的优先级做出相应的判断，再进行栈的一系列操作。

本次项目我一开始没有主要到需要用二叉树的前序遍历、中序遍历和后序遍历来完成，故我之前使用如下的解答方法完成(使用多个栈配合、原始栈、结果栈和运算符栈)，以下是我的另一种解法，不同于交上去的二叉树遍历的解法：

#include <iostream>

#include <cstring>

#define Max 100

using namespace std;

template <typename T>

class stack //栈的模板类

{

public:

stack(); //构造函数

~stack(); //析构函数

void Push(const T& item); //将元素压入栈中

T Pop(); //将栈顶元素移出

T GetTop(); //取得栈顶元素

bool is\_Empty(){ return top == -1; }; //判断栈是否为空

bool is\_Full(){ return top == size; }; //判断栈是否已满

private:

int top; //栈顶

int size = Max; //栈的大小

T\* element; //栈数组

};

template <typename T>

stack<T>::stack()

{

element = new T[size + 1]; //为栈数组分配空间

top = -1; //top置为-1

}

template <typename T>

stack<T>::~stack()

{

delete[] element; //删除栈数组

top = -1; //top置为-1

}

template <typename T>

void stack<T>::Push(const T& item)

{

if (is\_Full()) //如果栈满，直接return

return;

else

element[++top] = item; //将元素压入栈中，同时top加一

}

template <typename T>

T stack<T>::Pop()

{

if (top == -1) //如果栈空，退出

exit(-1);

else

return element[top--]; //将元素退出，同时top减1

}

template <typename T>

T stack<T>::GetTop()

{

if (top == -1) //如果栈空，退出

exit(-1);

else

return element[top]; //取得栈顶元素

}

bool Is\_value(char ch) //判断输入的字符是否有价值（为数字或者运算符）

{

if ((ch >= '0' && ch <= '9') || ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '(' || ch == ')')

return true; //有价值，return true

else

return false; //没价值，return false

}

int Priority(char ch) //判断操作符的优先级

{

if (ch == '+' || ch == '-')

return 1; //'+'、'-'返回1

else if (ch == '\*' || ch == '/')

return 2; //'\*'、'/'返回2

else

return 0; //'('、')'返回0

}

//中缀表达式转化为前缀表达式（波兰表达式）

char\* Transform\_Pre(char\* expression)

{

stack <char> Stack; //原始栈Stack

stack <char> result\_stack; //结果栈result\_stack

stack <char> operator\_stack; //运算符栈operator\_stack

int i = 0;

//将表达式中的元素压入原始栈中

while (expression[i] != '\0')

{

if (!Is\_value(expression[i])) //如果这个字符无价值，直接跳过，不压入栈中

{

i++;

continue;

}

Stack.Push(expression[i]); //字符有价值，压入栈中，同时i++

i++;

}

//获得相对应的结果栈

while (!Stack.is\_Empty()) //当原始栈Stack不为空时

{

char temp = Stack.GetTop(); //得到原始栈Stack顶的字符元素

Stack.Pop(); //原始栈Stack顶元素出栈

if (temp >= '0' && temp <= '9') //如果是数字，直接压入结果栈result\_stack中

result\_stack.Push(temp);

else if (temp == '+' || temp == '-' || temp == '\*' || temp == '/') //如果是'+'、'-'、'\*'、'/'运算符

{

if (operator\_stack.is\_Empty() || Priority(temp) > Priority(operator\_stack.GetTop()))

//如果运算符栈为空或者该运算符的优先级大于运算符栈顶运算符元素的优先级

operator\_stack.Push(temp); //将该运算符压入运算符栈

else if (Priority(temp) <= Priority(operator\_stack.GetTop()))

//如果该运算符的优先级小于等于运算符栈顶运算符元素的优先级

{

while (!operator\_stack.is\_Empty() && Priority(temp) <= Priority(operator\_stack.GetTop()))

//当循环条件为运算符栈不为空且要退出的元素的优先级小于等于该运算符

{

result\_stack.Push(operator\_stack.GetTop());

//此时就要将运算符栈中的元素退出到结果栈中

operator\_stack.Pop();

}

operator\_stack.Push(temp); //再将该运算符压入运算符栈中

}

}

else //如果是左括号或者右括号

{

if (temp == ')') //如果是右括号

operator\_stack.Push(temp); //直接将右括号压入栈中

else //如果是左括号，此时就要将运算符栈中先前的元素退出

{

while (operator\_stack.GetTop() != ')')

//退出的循环条件是退出的运算符不为右括号

{

result\_stack.Push(operator\_stack.GetTop());

//将运算符栈顶元素退出并压入结果栈中

operator\_stack.Pop();

}

operator\_stack.Pop(); //最后将右括号也出栈

}

}

}

while (!operator\_stack.is\_Empty()) //最后将运算符栈中剩余的元素压入结果栈中

{

result\_stack.Push(operator\_stack.GetTop());

operator\_stack.Pop();

}

//将结果栈中的元素转化为前缀表达式（波兰表达式）的字符串

int j = 0;

char\* retchar = new char[Max];

while (!result\_stack.is\_Empty()) //当结果栈不为空

{

retchar[j++] = result\_stack.GetTop(); //将栈中元素转化到字符数组相对应的位置

result\_stack.Pop(); //结果栈元素出栈

}

retchar[j] = '\0'; //最后加一个'\0'

return retchar;

}

//中缀表达式转化为后缀表达式（逆波兰表达式）

char\* Transform\_Post(char\* expression)

{

stack <char> Stack;

//原始栈Stack（中缀表达式转化为后缀表达式不需要再设一个结果栈和一个运算符栈）

char\* retchar = new char[Max];

int index = 0;

for (int i = 0; Is\_value(expression[i]) && expression[i] != '\0'; i++)

//遍历中缀表达式中有价值的字符串

{

if (expression[i] >= '0' && expression[i] <= '9') //当这个字符是数字

retchar[index++] = expression[i]; //就不用压入栈中了，直接赋值给结果字符串

else if (expression[i] == '+' || expression[i] == '-' || expression[i] == '\*' || expression[i] == '/')

//如果是'+'、'-'、'\*'、'/'运算符

{

if (Stack.is\_Empty() || Priority(Stack.GetTop()) < Priority(expression[i]))

//如果栈为空或者栈顶的运算符优先级小于当前运算符优先级

Stack.Push(expression[i]);

//将当前运算符压入栈中

else

//其它情况就要将栈中的运算符退出栈了

{

while (!Stack.is\_Empty() && Priority(Stack.GetTop()) >= Priority(expression[i]))

//循环条件为栈不为空而且要退出的运算符大等于当前运算符

{

retchar[index++] = Stack.GetTop();

//将栈顶运算符元素赋给结果字符串

Stack.Pop(); //栈Stack顶元素退栈

}

Stack.Push(expression[i]); //最后将该运算符压入栈中

}

}

else //如果是左括号或者右括号

{

if (expression[i] == '(') //如果是左括号的话，压入栈中

Stack.Push(expression[i]);

else //如果是右括号，则要有序地退栈

{

while (Stack.GetTop() != '(') //退栈的循环条件为还没有遇到左括号

{

retchar[index++] = Stack.GetTop(); //将栈中元素赋值给结果字符串

Stack.Pop(); //出栈

}

Stack.Pop(); //最后将左括号也出栈

}

}

}

while (!Stack.is\_Empty()) //将栈中剩余的元素赋值给结果字符串

{

retchar[index++] = Stack.GetTop();

Stack.Pop();

}

retchar[index] = '\0'; //最后加一个'\0'

return retchar;

}

int main()

{

char expression[Max];

cout << "请输入表达式:\t";

cin.getline(expression, Max); //得到英文的中缀表达式

cout << endl;

char\* Pre\_string = Transform\_Pre(expression); //中缀表达式转化为前缀表达式（波兰表达式）

cout << "波兰表达式：\t" << Pre\_string << endl << endl;

cout << "中缀表达式：\t" << expression << endl << endl;

char\* Post\_string = Transform\_Post(expression); //中缀表达式转化为后缀表达式（逆波兰表达式）

cout << "逆波兰表达式：\t" << Post\_string << endl << endl;

system("pause");

return 0;

}