项目说明文档

数据结构课程设计

——表达式计算

作 者 姓 名：杨鑫

学　　号：1950787

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

1.1 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。

人们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。

2 设计

2.1 数据结构设计

为了方便的将一个合法的中缀表达式转化为前缀式和后缀式，可以考虑将整个表达式存放在一个二叉树中，二叉树中每一个节点代表了一个数字或者运算符。这样在需要输出前缀，中缀，亦或是后缀表达式是，只需要分别前序，中序，后序遍历这棵二叉树即可。同时为了建立这棵二叉树，还需要用到一个栈，从而有利于问题的求解。

2.2 类结构设计

本程序最核心的类是一个二叉树类（BinaryTree），同时还有它的节点结构体（BinTreeNode），由于存放数字或者是运算符信息。为了构建表达式的二叉树，还设计了一个栈类（LinkedStzck），以及栈的节点结构体（LinkNode）。此外，程序还有一个表达式类，整合了二叉树以及对应的一些操作，使得程序更加规范完整。栈类和二叉树类都使用了模板，有利于程序的移植性和可拓展性。

2.3 成员与操作设计

节点结构体（LinkNode）：

```
1.  T data;
2.  LinkNode <T>* link;
3.  LinkNode(LinkNode<T>* ptr = NULL) : link(ptr) {}; // 构造函数
4.  LinkNode(const T& tem, LinkNode<T>* ptr = NULL) : data(tem), link
   (ptr) {}; // 构造函数
```

栈类（LinkedStack）：

私有成员：

```
1.  LinkNode<T>* top;  // 栈顶元素
```

公有操作：

```
1.  LinkedStack() : top(NULL) {} // 构造函数
2.  ~LinkedStack() { makeEmpty(); } // 析构函数
3.  void Push(const T& x); // 入栈
4.  bool Pop(T& x); // 出栈
5.  bool getTop(T& x) const; // 得到栈顶元素
6.  bool IsEmpty() const { return top == NULL; } // 判断是否栈空
7.  int getSize() const; // 返回栈中元素个数
8.  void makeEmpty(); // 栈置空
```

二叉树节点结构体（BinTreeNode）：

```
1.  T data;
2.  bool tag; // 增设一个标记，  用于标记是否需要进行括号打印
3.  BinTreeNode<T>* leftchild, * rightchild;
4.  BinTreeNode() : tag(0), leftchild(NULL), rightchild(NULL) {} //
   构造函数
5.  BinTreeNode(T x, BinTreeNode<T>* l = NULL, BinTreeNode<T>* r = NU
   LL) : tag(0), data(x), leftchild(l), rightchild(r) {} // 构造函数
```

二叉树类（BinTreeNode）：

私有成员：

```
1.  BinTreeNode<T>* root;
2.  void destroy(BinTreeNode<T>*& subTree); // 释放一个子树
3.  int Height(BinTreeNode<T>* subTree) const; // 返回子树高度
4.  int Size(BinTreeNode<T>*) const; // 返回子树节点个数
```

```
BinTreeNode<T>* Parent(BinTreeNode<T>* subTree, BinTreeNode<T>* current); // 返回父节点
void preOrder(BinTreeNode<T>*& subTree, void (*visit)(BinTreeNode<T>* p)); // 前序遍历
void inOrder(BinTreeNode<T>*& subTree, void (*visit)(BinTreeNode<T>* p)); // 中序遍历
void postOrder(BinTreeNode<T>*& subTree, void (*visit)(BinTreeNode<T>* p)); // 后序遍历
```

公有操作：

```
BinaryTree() : root(NULL) {} // 构造函数
~BinaryTree() { destroy(root); } // 析构函数
bool IsEmpty() { return root == NULL; } // 是否为空
BinTreeNode<T>* Parent(BinTreeNode<T>* current) { return (root == NULL || root == current ? NULL : Parent(root, current)); } // 得到父节点
BinTreeNode<T>* LeftChild(BinTreeNode<T>* current) { return current == NULL ? NULL : current->leftchild; } // 得到左子节点
BinTreeNode<T>* RightChild(BinTreeNode<T>* current) { return current == NULL ? NULL : current->rightchild; } // 得到右子节点
int Height() { return Height(root); } // 得到二叉树高度
int Size() { return Size(root); } // 得到二叉树节点个数
BinTreeNode<T>* getRoot() const { return root; } // 返回根节点点
void setRoot(BinTreeNode<T>* newRoot) { root = newRoot; } // 设置根节点
void preOrder(void (*visit) (BinTreeNode<T>* p)) { preOrder(root, visit); } // 前序遍历
void inOrder(void (*visit) (BinTreeNode<T>* p)) { inOrder(root, visit); } // 中序遍历
void postOrder(void (*visit) (BinTreeNode<T>* p)) { postOrder(root, visit); } // 后序遍历
```

表达式类（Expression）：

私有成员：

```
BinaryTree<string>* Tree; // 存放表达式的二叉树
string getItems(string& str); // 根据输入表达式每次取出一项
```

公有操作：

```
Expression(); // 构造函数
~Expression() { delete Tree; } // 析构函数
int CheckExp(string exp); // 检查项为数字还是符号，或者是非法字符串
int isp(string exp); // 返回符号对应的栈内优先数
int icp(string exp); // 返回符号对应的栈外优先数
bool BuildTree(); // 根据输入建立二叉树
void Polish(); // 通过前序遍历二叉树输出波兰表达式
void Infix(); // 通过中序遍历二叉树输出中序表达式
```

```
9.   void ReversePolish(); // 通过后序遍历二叉树输出逆波兰式
10.  void Loop(); // 主循环函数
```

2.4 系统设计

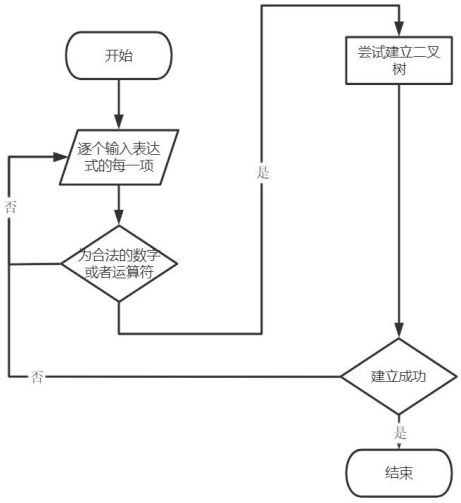
程序开始后，提示用户从键盘输入一个中缀表达式，然后存储到一个字符串中。然后程序根据输入的中缀表达式建立二叉树，如果是不合法的中缀表达式，将会提示用户输入有误；如果是正确的中缀表达式，将会建立二叉树，并且分别前序，中序，后序遍历二叉树，从而得到表达式的波兰式，中缀表达式和逆波兰式。

程序兼容了 windows 和 LINUX 平台，在双平台下均可以正常运行。

3 实现

3.1 建立二叉树功能的实现

3.1.1 建立二叉树功能的流程图

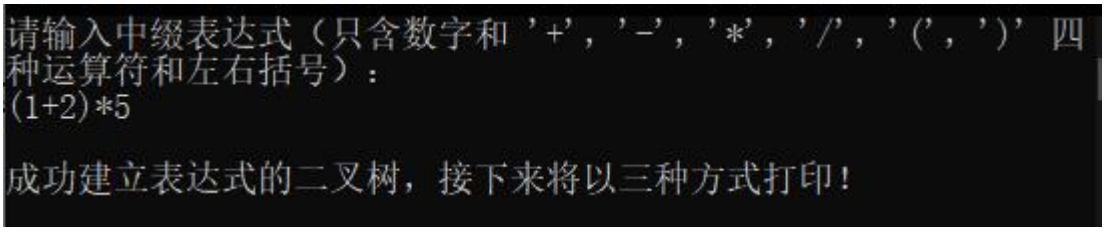


3.1.2 建立二叉树功能的核心代码

```
1.   // 根据出栈的符号建立节点
2.   temNode = new BinTreeNode<string>(op);
3.
4.   // 将符号节点和节点栈中的前两个节点链接起来
5.   BinTreeNode<string> * left, * right;
6.   if (nodeST->IsEmpty()) return false;
7.   else nodeST->Pop(right);
8.   if (nodeST->IsEmpty()) return false;
```

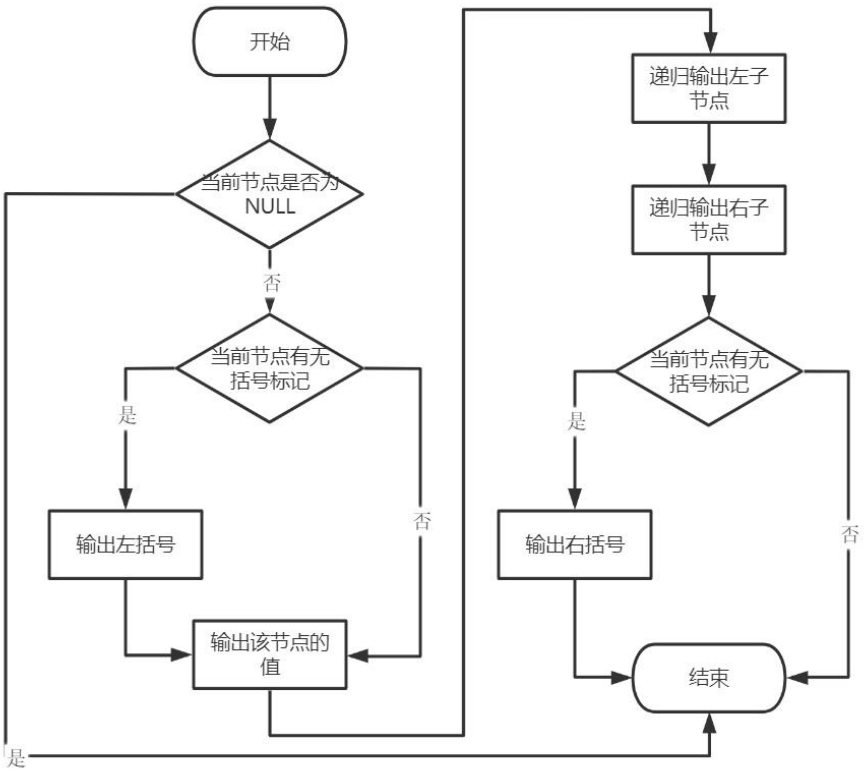
```
9.    else nodeST->Pop(left);
10.   temNode->leftchild = left;
11.   temNode->rightchild = right;
12.
13.   // 根据 next 标记是否为 true 决定其输出括号的标记是否为 true
14.   if (next) temNode->tag = true;
15.
16.   // 链接完成后重新入栈
17.   nodeST->Push(temNode);
```

3.1.3 建立二叉树功能的截屏示例



3.2 输出表达式功能的实现

3.2.1 输出表达式功能的流程图（以前序遍历为例）



3.2.2 输出表达式功能的核心代码

前序遍历（波兰表达式）：

```
if (subTree != NULL) {
    // 若打印括号标记为 true， 则先打印左括号
    if (subTree->tag) cout << "( ";

    // 先遍历该节点， 然后遍历左子树， 然后遍历右子树
    visit(subTree);
    preOrder(subTree->leftchild, visit);
    preOrder(subTree->rightchild, visit);

    // 若打印括号标记为 true， 则在最后打印右括号
    if (subTree->tag) cout << ") ";
}
```

中序遍历（中缀表达式）：

```
if (subTree != NULL) {
    // 若打印括号标记为 true， 则先打印左括号
    if (subTree->tag) cout << "( ";

    // 先遍历左子树， 然后遍历该节点， 然后遍历右子树
    inOrder(subTree->leftchild, visit);
    visit(subTree);
    inOrder(subTree->rightchild, visit);

    // 若打印括号标记为 true， 则在最后打印右括号
    if (subTree->tag) cout << ") ";
 }
```

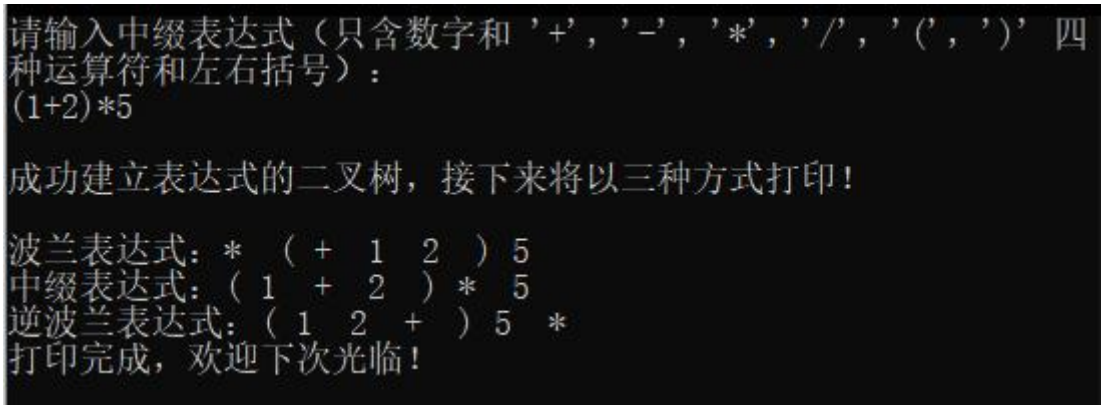
后序遍历（逆波兰表达式）：

```
if (subTree != NULL) {
    // 若打印括号标记为 true， 则先打印左括号
    if (subTree->tag) cout << "( ";

    // 先遍历左子树， 然后遍历右子树， 然后遍历该节点
    postOrder(subTree->leftchild, visit);
    postOrder(subTree->rightchild, visit);
    visit(subTree);

    // 若打印括号标记为 true， 则在最后打印右括号
    if (subTree->tag) cout << ") ";
}
```

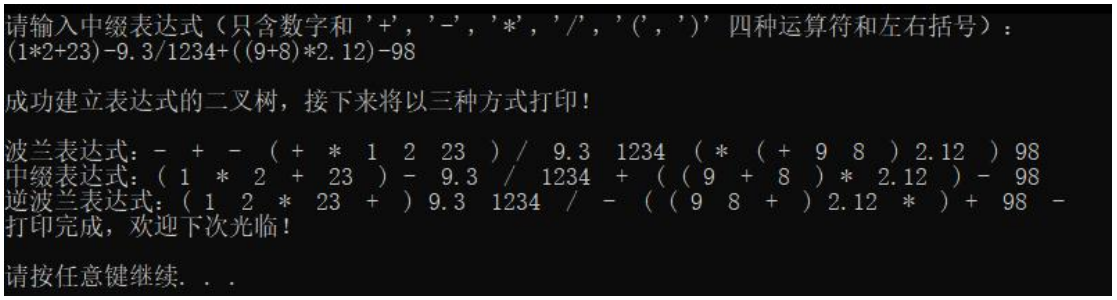
3.2.3 输出表达式功能的截屏示例



4 测试

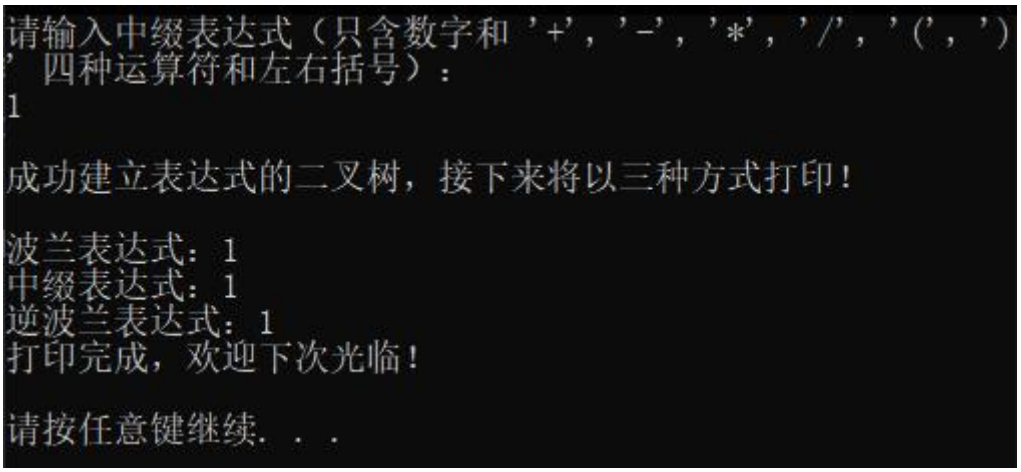
4.1 功能测试

测试用例：输入带有括号，小数点，多重括号的复杂表达式
预期结果：程序正常运行，输出正确结果
实验结果：



4.2 边界测试

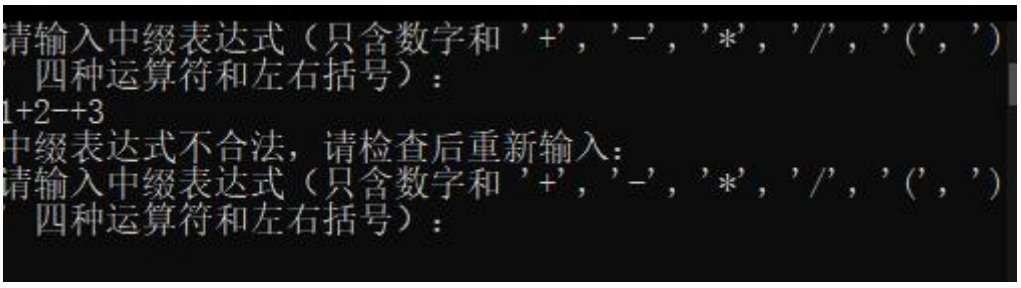
测试用例：只输入一个数字
预期结果：程序正常运行，输出对应结果
实验结果：



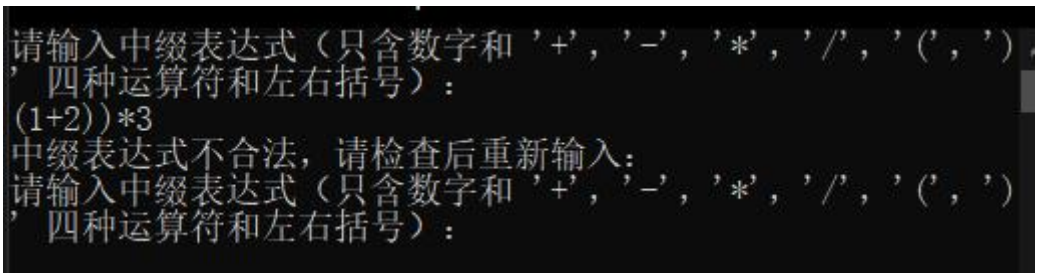
4.3 出错测试

4.3.1 表达式不合法测试

测试用例：输入不合法的表达式（运算符错误）
预期结果：程序提示用户输入有误，不会出现异常或者崩溃
实验结果：



测试用例：输入不合法的表达式（括号错误）
预期结果：程序提示用户输入有误，不会出现异常或者崩溃
实验结果：



4.3.2 输入不合法的项测试

测试用例：输入不合法的项

预期结果：程序提示用户输入有误，不会出现异常或者崩溃
实验结果：