广州大学学生实验报告

开课实验室： 电子楼 416B　　2018 年 5 月 6 日

学院	计算机科学与网络工程学院	年级、专业、班	软件 171	姓名	谢金宏	学号	1706300001
实验课程名称	数据结构					成绩	
实验项目名称	实验二　树和二叉树的实现					指导老师	杜娇

一、实验目的

熟练掌握二叉树的二叉链表表示方法及二叉树遍历算法。

二、使用仪器、器材

1. 使用 Windows 操作系统的微型计算机一台。
2. 符合 C++11 标准的 C++编程软件。

三、实验内容及原理

（一）利用二叉树的先序遍历和中序遍历建立二叉树。

例如：先根序列为 ABDGCEF#， 中根序列为 DGBAECF#（#表示结束）。然后用程序构造一棵二叉树。注意程序的通用性（也就是说上述只是一个例子，你的程序要接受两个序列（先根和中根序列），然后构造相应的二叉树）。

（二）将中缀表达式转换为表达式树，并通过后序遍历计算表达式的值。

例如：中缀表达式为(a+b)*(c+d)#（#表示结束），将之转换成一棵二叉树，然后通过后序遍历计算表达式的值，其中 abcd 都是确定的值。注意程序的通用性（也就是说上述只是一个例子，你的程序要接受一个序列，然后构造相应的二叉树，最后通过后序遍历计算出值（注意不是根据中缀表达式计算出值，而是通过后序遍历所构造出的二叉树计算出值。））。

四、实验过程原始数据记录

实验使用的源代码如下：


Tree.cpp


ExpressionTree.cpp

（一）利用二叉树的先序遍历和中序遍历建立二叉树。

算法

对于给定的二叉树的先序遍历和中序遍历，定义建树算法如下：

1. 如果序列为空，那么返回空结点；如果序列只有一个结点，那么返回这个结点。
2. 其他情况下，确定先序遍历的第一个结点为树的根节点。依据此结点为界，将中序遍历划分为左子树部分和右子树部分，并确定左子树部分的结点数量；根据左子树部分的结点数量可以确定先序遍历中左子树部分和右子树部分。这样就确定了左子树和右子树对应的先序遍历和中序遍历序列，可以递归地调用建树算法。

基础结构

```
// 结点定义
struct Node
{
    char value;  // 结点的值
    Node * left_child, * right_child;  // 结点的左孩子、右孩子指针
};

// 二叉树定义
using Tree = Node*;
```

工具函数

```
// 求二叉树的深度
int TreeDepth(Tree & tree)
{
    if (tree) {
        return max(TreeDepth(tree->left_child), TreeDepth(tree->right_child)) + 1;
    }
    return 0;
}
```

```
// 在标准输出中打印二叉树
// 打印格式形如：
//       F
//    Q      B
//  D   E   A   F
// # F # # # # # #
void TreePrint(Tree & tree)
{
    int tree_depth = TreeDepth(tree);  // 待打印二叉树的深度
    function<void(int)> print_space = [](int space_count){
        while (space_count--) cout << ' ';
    };  // 输出 space_count 个空格的函数

    queue<Node *> que;
    que.push(tree);  // 树根结点入队

    // 按层打印二叉树
    for (int current_depth = 1; current_depth <= tree_depth; ++current_depth)
    {
        int next_level_node_count = 0;  // 下一层非空结点的个数
        int space_count = (1 << (tree_depth - current_depth)) - 1;  // 结点字符前后空格的数量
        queue<Node *> nxt;  // 下一层待打印的结点
        while (!que.empty())  // 当本层还有待打印的结点
        {
            print_space(space_count);  // 打印前导空格
            Node * front = que.front();  // 取结点
            if (front != nullptr) {  // 如果结点不是空结点
                cout << front->value;  // 打印结点
                nxt.push(front->left_child);  // 将左右孩子结点加入队列
                nxt.push(front->right_child);
                next_level_node_count += int(front->left_child != nullptr);  // 计算下一层待打印的结点的个数
                next_level_node_count += int(front->right_child != nullptr);
            }
            else {  // 如果结点是空结点
                cout << '#';  // 打印空结点符号
                nxt.push(nullptr);  // 放入表示空结点的值
                nxt.push(nullptr);
            }
            que.pop();  // 弹出当前结点
            que.empty() ? print_space(0) : print_space(space_count + 1);  // 打印后导空格
        }


        cout << '\n';  // 换行
        que = nxt;
        if (next_level_node_count == 0) break;
    }
}
```

建立二叉树的算法

```
// 根据先序序列和中序序列创建二叉树
// tree: 树根
// pre_order_string: 树对应的先序序列
// mid_order_string: 树对应的中序序列
void CreateTree(Tree& tree, const string& pre_order_string, const string&
mid_order_string)
{
    if (pre_order_string.length() == 0) {
        tree = nullptr;
        return;
    }
    tree = new Node{pre_order_string[0], nullptr, nullptr};
    unsigned pos = mid_order_string.find_first_of(pre_order_string[0]);  // 中序遍历中
根结点的位置
    CreateTree(tree->left_child, pre_order_string.substr(1, pos),
mid_order_string.substr(0, pos));  // 递归创建左子树
    CreateTree(tree->right_child, pre_order_string.substr(pos + 1),
mid_order_string.substr(pos + 1));  // 递归创建右子树
}
```

入口函数

```
int main()
{
    cout << "根据先序遍历和中序遍历构建二叉树。" << endl;

    string pre_order_string, mid_order_string;  // 先序序列和中序序列
    cout << "请输入先序序列："; cin >> pre_order_string;
    cout << "请输入中序序列："; cin >> mid_order_string;

    if (*pre_order_string.rbegin() != '#' || *mid_order_string.rbegin() != '#')
    {  // 检查是否先序序列和中序序列是否以“#”标志结尾
        cout << "请以#作为序列的结束标志。" << endl;
        return -1;
    }
    pre_order_string.pop_back(), mid_order_string.pop_back();  // 在序列中移除末尾的“#”

    Tree tree;
    CreateTree(tree, pre_order_string, mid_order_string);
    TreePrint(tree);
}
```

（二）将中缀表达式转换为表达式树，并通过后序遍历计算表达式的值。

算法

对于给出的中缀表达式，定义转换为表达式树的建树算法如下：

1. 如果表达式为空，返回数值为 0 的结点；如果表达式只有一个数值，返回将这个结点作为数值返回。
2. 如果表达式两端有匹配的括号，将表达式两端的括号去除后，递归调用建树算法。
3. 寻找表达式中运算优先级最低的操作符，将这个操作符作为根节点；以此操作符为界，将表达式分为左右两部分，对这两部分分别调用建树算法，将返回的结点作为左子结点和右子结点。

对于一棵表达式树，使用后序遍历求值的算法定义如下：

1. 如果表达式树根节点为数值结点，那么返回这个结点的数值。
2. 如果表达式树根节点为操作符结点，那么使用求值算法分别对左右子树求值，并对左右子树求值的结果应用操作符进行计算，并返回计算的结果。

基础结构

```
// 结点定义
struct Node
{
    string value;  // 结点的值
    Node * left_child, * right_child;  // 结点的左孩子、右孩子指针
};


// 二叉树定义
using Tree = Node*;
```

根据中缀表达式建树函数

```
// 根据中缀表达式建立表达式树
void BuildTree(vector<string> vec, Tree& tree)
{
    if (vec.empty()) {  // 如果表达式为空返回 0
        tree = nullptr;
        return;
    }
    if (vec.size() == 1) {  // 如果表达式只有一个数，则返回这个数
        tree = new Node{vec[0], nullptr, nullptr};
        return;
    }
```

```
// 若表达式以左括号开始且以右括号结束，并且开头的左括号与此右括号匹配，
// 将表达式两端的括号去掉，并递归建树
if (*vec.begin() == "(" && *vec.rbegin() == ")") {
    int unclosed_parentheses = 1;
    for (auto it = vec.begin() + 1; it != vec.end() - 1; ++it)
    {
        if (*it == "(") ++unclosed_parentheses;
        if (*it == ")") --unclosed_parentheses;
        if (unclosed_parentheses == 0) break;
    }
    if (unclosed_parentheses == 1) {
        BuildTree(vector<string>(vec.begin()+1, vec.end()-1), tree);
        return;
    }
}


// 寻找优先级最低的运算符
auto pos = vec.end();  // 优先级最低的运算符的位置
string lowerest_operator = "*";  // 假设优先级最低的运算符为*
for (auto it = vec.begin(); it != vec.end(); ++it) {
    if (*it == "(") {  // 跳过括号
        int count = 1;  // 进入的括号层数
        while (count != 0) {
            ++it;
            if (*it == "(") ++count;
            if (*it == ")") --count;
        }
        continue;
    }
    if ((*it == "*" || *it == "/") && (lowerest_operator == "*" || lowerest_operator
== "/")) {
        pos = it;
        lowerest_operator = *it;
    }
    if (*it == "+" || *it == "-") {
        pos = it;
        lowerest_operator = *it;
    }
}
```

```
    // 以最低优先级的运算符为界，将表达式划分为两部分，递归建树
    tree = new Node{*pos, nullptr, nullptr};
    BuildTree(vector<string>(vec.begin(), pos), tree->left_child);
    BuildTree(vector<string>(pos+1, vec.end()), tree->right_child);
}
```

后序遍历求值函数

```
// 使用后续遍历计算表达式树的值
double Calculate(Tree tree)
{
    if (tree == nullptr) {
        return 0;
    }
    string value = tree->value;  // 取根节点的值
    switch (value[0]) {
        case '+':
            return Calculate(tree->left_child) + Calculate(tree->right_child);
        case '-':
            return Calculate(tree->left_child) - Calculate(tree->right_child);
        case '*':
            return Calculate(tree->left_child) * Calculate(tree->right_child);
        case '/':
            return Calculate(tree->left_child) / Calculate(tree->right_child);
        default:
            return stof(value);
    }
}
```

入口函数

```
int main()
{
    cout << "将中缀表达式转换成表达式树，" << endl;
    cout << "然后使用后序遍历计算表达式的值。" << endl;

    cout << "请输入中缀表达式，并以#结尾" << endl;
    string expression;  // 待处理的中缀表达式
```

```
{  // 处理用户输入
    getline(cin, expression);
    if (*expression.rbegin() == '#')
        expression.pop_back();  // 删除表达式末尾的“#”
}

vector<string> vec;  // 中缀表达式中运算数和运算符的队列
Tree tree;  // 表达式树

{  // 解析中缀表达式中的符号
    for (unsigned index = 0; index < expression.length(); ++index)
    {
        string token;
        if (isspace(expression[index])) continue;
        if (isdigit(expression[index])) {  // 处理运算数
            while (isdigit(expression[index])) {
                token += expression[index];
                ++index;
            }
            vec.push_back(token);
            --index;
            continue;
        }
        else {  // 处理运算符
            token = expression[index];
            vec.push_back(token);
            continue;
        }
    }
}

    BuildTree(vec, tree);
    TreePrint(tree);
    cout << "表达式的计算结果为：" << Calculate(tree) << endl;
}
```

五、实验结果及分析

本次实验整体完成度较好，下面是程序的测试情况。

（一）利用二叉树的先序遍历和中序遍历建立二叉树。

程序输入	程序输出	是否符合预期?
# # 	（空）	√
a# a#	a	√
abc# cba#	a b # c # # #	√
abc# bca#	a b # # c # #	√
abcefg# becagf#	a b f # c g # # # e # # # # #	√

程序输出符合预期，完成实验要求。

（二）将中缀表达式转换为表达式树，并通过后序遍历计算表达式的值。

程 序 输入	程序输出	是否符合预期?
#	0 表达式的计算结果为：0	√
34#	34 表达式的计算结果为：34	√
1+3+4 #	+ + 4 1 3 # # 表达式的计算结果为：8	√
1*4-7 #	- * 7 1 4 # # 表达式的计算结果为：-3	√
1+2*3 -4#	- + 4 1 * # # # # 2 3 # # # # 表达式的计算结果为：3	√
(1+3) *(2-6)#	* + - 1 3 2 6 表达式的计算结果为：-16	√
((3-2)*8+1	+ * 12	√

2)#	- 8 # # 3 2 # # # # # # 表达式的计算结果为：20	
(1)+(2)-(3)*(4)/(5)#	- + / 1 2 * 5 # # # # 3 4 # # 表达式的计算结果为：0.6	√
1+(((3+8)-9)*6)-12)#	+ 1 - # # * 12 # # # # - 6 # # # # # # # # # # # # # # # # # 3 8 # # # # # # # # # # # # # # # 表达式的计算结果为：1	√

程序的输出符合预期，完成了实验要求。