实验二：将算术表达式转换成语法树形式

18340166　　王若琪

1. 算法描述

我的实现方法分为 3 个步骤，分别如下：

1.1 首先，将算数表达式转化为后缀表达式 (以上一次实验为基础)

- 首先初始化一个空栈。
- 从左到右遍历中缀表达式中的每一个字符，如果遇到数，就将他们直接添加进要输出的后缀表达式中，在实现过程中，还需考虑几个字符组成一个数，根据每次遇到的数字字符的下一个是否还为数字字符来确定数字是否结束，输出的后缀表达式用空格分隔开数字和运算符。
- 如果遇到左括号 '(' ，就直接进栈。
- 如果遇到右括号 ')' ，就一直将栈里的符号出栈，并将这些符号添加进要输出的后缀表达式中，直到遇到左括号 '(' 为止，其中这个左括号也要出栈，但不添加进要输出的后缀表达式中。
- 如果遇到操作符 '+' , '-' , '*', '/'，那么继续进行如下判断：
    - 先判断栈是不是空的，如果是空栈，就直接进栈。
    - 如果不是空栈，就将此操作符与栈顶符号比较优先级：
        - 如果此操作符的优先级小于或等于栈顶操作符的优先级时，那么将该栈顶符号放入后缀表达式，并且弹出该栈顶符号。反复执行此过程直到当前的操作符的优先级大于栈顶操作符的优先级。
        - 如果此操作符优先级大于栈顶操作符，那么将此操作符进栈。
- 重复以上步骤直到中缀表达式遍历完成。
- 当遍历完中缀表达式后，如果栈中还有剩余的操作符，就依次弹出并放入后缀表达式中，直到栈空。

1.2 再用后缀表达式构造表达式树

- 新建一个空栈，并从上一步生成的后缀表达式的第一个符号或者数字开始逐个判断并操作：
    - 如果为数字，就新建一个树的节点，节点的值为这个数，并将这个节点的指针压入栈中。
    - 如果是操作符，就新建一个树的节点，并且从栈中弹出两个树的指针，先弹出的作为这个新树的右节点，后弹出的作为这个树的左节点。然后将这个新树根节点的指针压入栈中。
    - 按上面所述步骤循环，直到操作完最后一个符号。
- 循环结束后，最终，指向树的根节点指针就是最终的栈顶。

1.3 树的可视化输出

由于要输出左对齐的树的可视化形式，所以需要先将整个树的结构先储存在一个二维数组里，便于输出，具体方法如下：

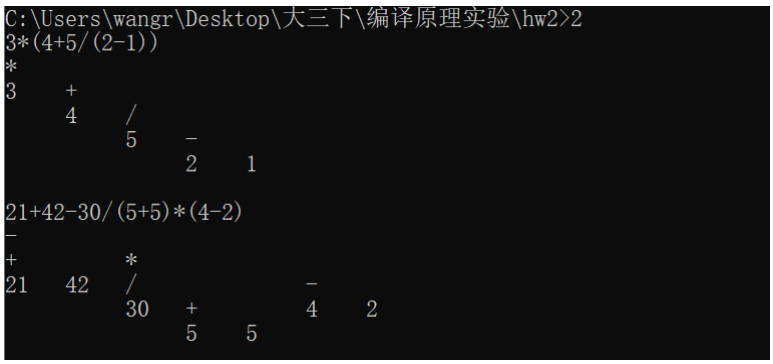
- 用递归方法求得二叉树的高度 $height$。
- 分配二维数组，数组大小为 $height \times (height^2 - 1)$.
- 用递归的方法对二叉树进行深度优先遍历，确定每一个节点的输出位置（即在该二维数组中的坐标）。具体方法如下：

    - 根节点的坐标先确定为 $(0,0)$，整个树的节点在数组中的左右范围设为 $(left, right) = (0, (height^2 - 1))$
    - 根节点的左儿子的坐标为 $(1, left)$，右儿子的坐标为 $(1, (left + right)/2)$，左子树的范围为.
    - 以此类推，用递归求出所有节点的位置坐标，并将各节点的值写入数组中的对应位置。
- 简化求得的二维数组。由于之前的二维数组是按照宽度为 $(height^2 - 1)$ 来设计的，所以在实际操作时，该二维数组中会有很多空列，导致可视化输出时显得树比较松散。针对这个问题，在输出前需要将二维数组中的空列去除，从而达到要求的目标效果。

---

2. 结果展示

运行程序 2.exe，输入算数表达式，得到响应结果输出如下：



---

3. 附录 (源代码)

```
#include <iostream>
#include <stack>
#include <vector>
#include <iomanip>
#include <math.h>
using namespace std;

struct Node
{
    string val;
    struct Node *left;
    struct Node *right;
};

class BinaryTree
{
public:
    BinaryTree(struct Node *r = NULL) : root(r) {}
    BinaryTree(vector<string> &v);
    ~BinaryTree();
    int getDepth(Node *root);
    void write(vector<vector<string>> &print, Node *root, int cur_depth,
int left, int right);
    vector<vector<string>> get_tree_vec(Node *root);
    void print_tree();
    void deleteNode(struct Node *pNode);

private:
```

```
    stack<struct Node *> st;
    struct Node *root;
    int width;
};

//构造函数
BinaryTree::BinaryTree(vector<string> &v)
{
    for (unsigned i = 0; i < v.size(); ++i)
    {
        // 新建一个节点，并赋值
        struct Node *pNode = new Node;
        pNode->val = v[i];
        pNode->left = NULL;
        pNode->right = NULL;

        // 如果是运算符，就弹出并且作为刚才新建的节点的左右孩子
        if ((v[i] == "+") || (v[i] == "-") || (v[i] == "*") || (v[i] == "/"))
        {
            pNode->right = st.top();
            st.pop();
            pNode->left = st.top();
            st.pop();
        }
        // 将新树压栈
        st.push(pNode);
    }
    //循环结束后，最后根节点指针会是栈顶指针
    root = st.top();
}

//析构函数
BinaryTree::~BinaryTree()
{
    deleteNode(root);
}

void BinaryTree::deleteNode(struct Node *pNode)
{
    if (pNode != NULL)
    {
        delete pNode;
        deleteNode(pNode->left);
        deleteNode(pNode->right);
    }
}

int BinaryTree::getDepth(Node *root)
{
    if (!root)
    {
        return 0;
    }
    return max(getDepth(root->left), getDepth(root->right)) + 1;
}

```

```cpp
void BinaryTree::write(vector<vector<string>> &print, Node *root, int cur_depth, int left, int right)
{
    if (!root)
    {
        return;
    }
    print[cur_depth][left] = root->val;
    //递归深度优先遍历
    write(print, root->left, cur_depth + 1, left, (left + right) / 2);
    write(print, root->right, cur_depth + 1, (left + right) / 2, right);
}

vector<vector<string>> BinaryTree::get_tree_vec(Node *root)
{
    int row = getDepth(root);
    int col = 2 << row - 1;
    vector<vector<string>> print(row, vector<string>(col, ""));
    write(print, root, 0, 0, col);
    return print;
}

void BinaryTree::print_tree()
{
    vector<vector<string>> ans = get_tree_vec(root);
    vector<int> empty(32, 0);
    for (int i = 0; i > empty.size(); i++)
        cout << empty[i] << endl;
    for (int i = 0; i < ans.size(); i++)
    {
        for (int j = 0; j < ans[i].size(); j++)
        {
            if (ans[i][j] != "")
                empty[j] = 1;
        }
    }
    for (int i = 0; i < ans.size(); i++)
    {
        for (int j = 0; j < ans[i].size(); j++)
        {
            //cout << '(' << ans[i][j]<<')'<<' ';
            if (empty[j] != 0)
                cout << setiosflags(ios::left) << setw(5) << ans[i][j];
        }
        cout << endl;
    }
}

int get_priority(char ch) //判断符号的优先级
{
    if (ch == '*' || ch == '/')
        return 2;
    else if (ch == '+' || ch == '-')
        return 1;
    else
        return 0;
}

```

```
string get_postfix(string infix)
{
    stack<char> char_stack;
    string postfix;
    int lenth = infix.size();
    for (int i = 0; i < lenth; i++) //从左往右遍历中缀表达式
    {
        if (isdigit(infix[i])) //如果是数字
        {
            postfix += infix[i];
            //如果这个数字结束了，后缀表达式中加空格
            if (i == lenth - 1)
                postfix += ' ';
            else if (!isdigit(infix[i + 1]))
                postfix += ' ';
        }
        else if (infix[i] == '(') //如果左括号直接进栈
        {
            char_stack.push(infix[i]);
        }
        else if (infix[i] == ')') //如果右括号，一直出栈直到遇到左括号
        {
            while (char_stack.top() != '(')
            {
                postfix = postfix + char_stack.top() + ' ';
                char_stack.pop();
            }
            char_stack.pop();
        }
        else if (infix[i] == '+' || infix[i] == '-' || infix[i] == '*' || infix[i] == '/')
        {
            if (char_stack.empty()) //如果空栈就push进栈
            {
                char_stack.push(infix[i]);
            }
            else //如果不空，与栈顶符号比较优先级
            {
                while (!char_stack.empty())
                {
                    if (get_priority(infix[i]) <= get_priority(char_stack.top())) //如果优先级低于等于栈顶的符号，栈顶元素就出栈直到不满足条件
                    {
                        postfix = postfix + char_stack.top() + ' ';
                        char_stack.pop();
                    }
                    else
                        break;
                }
                char_stack.push(infix[i]); //进栈
            }
        }
    }
    while (!char_stack.empty())
    {
        postfix = postfix + char_stack.top() + ' ';
        char_stack.pop();
```

```
    }
    return postfix;
}

// 将后缀表达式分割成 vector 中的元素
int splitStringToVect(const string &srcStr, vector<string> &destVect,
const string &strFlag)
{
    int pos = srcStr.find(strFlag, 0);
    int startPos = 0;
    int splitN = pos;
    string lineText(strFlag);

    while (pos > -1)
    {
        lineText = srcStr.substr(startPos, splitN);
        startPos = pos + 1;
        pos = srcStr.find(strFlag, pos + 1);
        splitN = pos - startPos;
        destVect.push_back(lineText);
    }

    lineText = srcStr.substr(startPos, srcStr.length() - startPos);
    return destVect.size();
}

int main()
{
    while (1)
    {
        string infix, postfix; //中缀，后缀
        cin >> infix;
        if (infix == "EXIT")
            break;
        postfix = get_postfix(infix);
        vector<string> v;
        splitStringToVect(postfix, v, " ");
        BinaryTree tree(v);
        tree.print_tree();
        cout << endl;
    }
}
```