实验报告

实验题目：二叉树及其应用

计算机学院3班，雷雨轩，PB18111791

完成日期：2019年11月20日

实验要求

二叉树的创建与遍历

基本要求：

- 通过添加虚结点，为二叉树的每一实结点补足其孩子，再对补足虚结点后的二叉树按层次遍历的次序输入。 例如： ABC#DEFG##H######.
- 构建这颗二叉树（不包含图中的虚结点），并增加左右标志域，将二叉树后序线索化。
- 完成后序线索化树上的遍历算法，依次输出该二叉树先序遍历、中序遍历和后序遍历的结果。

输入输出样例：

```
Input:
ABC#DEFG##H######
Output:
ABDGCEHF //先序遍历
BGDAEHCF //中序遍历
GDBHEFCA //后序遍历
```

表达式树

基本要求:

输入合法的波兰式(仅考虑运算符为双目运算符的情况)，构建表达式树，分别输出对应的中缀表达式（可含有多余的括号）、逆波兰式和表达式的值，输入的运算符与操作数之间会用空格隔开。

选做要求（二选一即可）:(本次实验选择第一个选做要求)

1.输出的中缀表达式中不含有多余的括号。 例如在上面的样例中，期望的输出结果应该是 2+3-1

2. 输入逆波兰式，输出波兰式、中缀表达式（可含有多余的括号）和表达式的值。

输入输出样例:

```
Input:
- + 2 3 1   //波兰式
Output:
(2+3)-1     //中缀表达式
2+3-1       //不含多余括号的中缀表达式
2 3 + 1 -   //逆波兰式
4           //求值
```

设计思路

二叉树的创建与遍历

- **由层次遍历输入构建二叉树**:构建关键是要理解层序遍历的顺序，所以考虑利用循环队列作为辅助，入队顺序即为层序遍历顺序（虚节点则跳过），且按层出队，使得每次出队元素则正好为接下来读入数据的父母结点
- **二叉树后续线索化**:利用好递归的思想:后序遍历顺序即:访问左子树，访问右子树，访问根节点，并在访问过程中用pre记录当前访问节点的前驱，按后序遍历顺序为每个节点添加前驱后继线索(如果该节点相应左右子树不存在)
- **先序遍历和中序遍历**：均采用递归算法，但可通过对当前节点LTag,RTag的检查来判断当前节点左右子树是否存在(不存在，则不再遍历其左/右子树)，从而减少递归层数
- **后序遍历**:采用非递归算法，关键即如何来判断当前访问节点的后继，分三种情况
  - 该节点无右子树，其后继可由后继线索找到
  - 该节点有右子树
    - 该节点是双亲的右孩子或者是左孩子且双亲无右孩子，则后继为双亲节点
  - 该节点是双亲左孩子且双亲有右孩子，则接下来后序遍历双亲右孩子
- 代码分为五个模块：
  - **数据结构定义**:带线索的三叉链表、队列及其操作实现
  - **按层次遍历次序的输入构建二叉树**：`void CreateBiThrTree(BiThrTree& T)`
  - **将二叉树后续线索化**:

    `void PostOrderThreading(BiThrTree &Thrt,BiThrTree T)`

    `void PostThreading(BiThrTree T, BiThrTree &pre)`
  - **二叉树的先序遍历、中序遍历递归实现及后序遍历非递归实现**:

    ```
    void PreOrderTraverse(BiThrTree T, int (* Visit)(char))
    void InOrderTraverse(BiThrTree T, int (*Visit)(char))
    void PostOrderTraverse(BiThrTree Thrt, int (*Visit)(char))
    ```

  - **主函数**:调用各个模块的函数实现实验要求的操作并输出

表达式树

- **以先序次序的输入来构造二叉链表**:首先对每个读入的字符，若是运算数，则其就是叶节点。若是运算符，则还要对其左右子树进行构造。因为要考虑到完成无多余括号的中缀表达式的输出，所以需要对符号做一个优先级划分，运算数为3级，*，/为2级，+，-为1级，优先级决定了是否加括号。
- **中序遍历表达式树**：
  - 有多余括号:只要左子树或右子树根节点仍为运算符，则为相应子树的运算式加一对括号
  - 无多余括号:需判断优先级:若当前节点为运算符，则需考虑左子树及右子树是否要加括号:
    - 左子树根节点优先级低于当前节点优先级，则加括号
    - 右子树根节点优先级低于或等于当前节点优先级，则加括号
    - 注意若左子树或右子树为运算数，即叶节点时，是不需要加括号的，所以把运算数的优先级设置为最高的3，来避免此类错误发生。
- **后序遍历输出并求值**：

- 只要求输出则很简单，只需要递归调用，且按遍历左子树、遍历右子树，根节点的顺序来输出即可。
- 本实验考虑采用以逆波兰式求值:利用操作数栈作为辅助，遇到操作数则入栈，遇到运算符则弹出两个数并在做相应计算后重新入栈，如此一来最终栈底的数即为所求表达式的值

- **代码分为五个模块**
  - **数据结构定义**:节点含字符及相应优先级，以及标准的二叉链表，并利用C++栈模块来辅助
  - **以先序次序的输入来构造二叉链表**:`void createBiTree(BiTree& T)`
  - **中序遍历表达式树**并输出:

    ```
    void InOrderTraverse(BiTree T, void (*Visit)(string))//有多余括号
    void InOrderTraverse1(BiTree T, void (*Visit)(string))//无多余括号
    ```

  - **后序遍历输出并求值**:

    ```
    void PostOrderTraverse(BiTree T, void (*Visit)(string))
    void PrintEle(string S)   //打印树节点的值并进行相应栈操作
    ```

  - **主函数**:调用各个模块的函数实现实验要求的操作并输出

关键代码讲解

二叉树的创建与遍历

```
//数据定义
typedef enum PointerTag { Link, Thread };        //Link = 0, 指针;thread=1,线索
typedef struct BiThrNode {                       //线索化的三叉链表
    char data;
    struct BiThrNode* lchild, * rchild, * parent;
    PointerTag LTag, RTag;
}BiThrNode, * BiThrTree;

typedef struct {
    BiThrTree * base;
    int front;
    int rear;
}SqQueue;                                    //构造循环队列作为层序遍历构建树的辅助
```

基本要求:

```
//由层次遍历输入构建二叉树
void CreateBiThrTree(BiThrTree& T) {     //由层序遍历构建出来的二叉树，
                                         //左右指针指向左右孩子，若没左右孩子，相应指
针为NULL
    char ch;
    SqQueue Q;                           //构建辅助队列
    InitQueue(Q);
    cin >>noskipws>> ch;
    if (ch == ' ') {                     //第一个节点为空，则返回空树
        T = NULL;
    }
    else {                               //创建根节点并使根节点入队
```

```
        T = new BiThrNode;
        T->data = ch;
        T->parent = NULL;
        T->LTag = Link;
        T->RTag = Link;
        enqueue(Q, T);
    }
    while (Q.rear != Q.front) {          //当前队列不为空
        BiThrTree p = dequeue(Q);    //队头元素出队，此为接下来层序遍历输入数据的父母
        cin >> noskipws >> ch;
        if (ch == ' ') {                 //左孩子为空
            p->lchild = NULL;
            p->LTag = Thread;
        }
        else {                           //左孩子不为空
            p->lchild = new BiThrNode;
            p->lchild->data = ch;
            p->lchild->parent = p;
            p->LTag = Link;
            enqueue(Q, p->lchild);       //左孩子入队
        }
        cin >> noskipws >> ch;
        if (ch == ' ') {                 //右孩子为空
            p->rchild = NULL;
            p->RTag = Thread;
        }
        else {                           //右孩子不为空
            p->rchild = new BiThrNode;
            p->rchild->data = ch;
            p->rchild->parent = p;
            p->RTag = Link;
            enqueue(Q, p->rchild);       //右孩子入队
        }
    }
}
```

```
//将二叉树后续线索化
void PostThreading(BiThrTree T, BiThrTree &pre) {           //实现后序线索化的函数
    if (T) {
        PostThreading(T->lchild,pre);                       //对左子树后序线索化
        PostThreading(T->rchild,pre);                       //对右子树后序线索化
        if (!T->lchild) { T->LTag = Thread; T->lchild = pre; }      //当前节点T的
前驱线索
        if (!pre->rchild) { pre->RTag = Thread; pre->rchild = T; }  //pre的后继线
索
        pre = T;                                                    //当前节点完成
线索化，则                                                                  //前
驱变为当前节点
    }
}

void PostOrderThreading(BiThrTree &Thrt,BiThrTree T) {
    //后序遍历二叉树T,并将其后续线索化
    //在线索链表上添加一个头结点，其lchild
//指向二叉树根节点，
    //rchild指向后序遍历中访问的最后一个节点（也为根节点）；
    //相应的，二叉树后序序列第一个节点lchild指向头结点；最后一个节点(根)视情况而定
```

```
    Thrt = new BiThrNode;                       //建头结点
    Thrt->LTag = Link; Thrt->RTag = Thread;
    Thrt->rchild = Thrt;                        //开始为空树，右指针回指
    if(!T)  Thrt->lchild = Thrt;                //若二叉树为空，则左指针回指
    else {
        Thrt->lchild = T;
        BiThrTree pre = Thrt;                   //pre指向遍历过程中当前节点的前驱
        PostThreading(T,pre);
        if (!pre->rchild) {                     //尾结点(即根节点若无右子树，则其后继指向头
结点Thrt)
            pre->rchild = Thrt;
            pre->RTag = Thread;
        }
        Thrt->rchild = pre;
    }
}
```

```
//二叉树的先序遍历、中序遍历递归实现及后序遍历非递归实现
int PrintEle(char a) {                                                  //遍历的访问函
数
    cout << a;
    return 1;
}

void PreOrderTraverse(BiThrTree T, int (* Visit)(char)) {
                                                                        //先序遍历后续线索化树的
递归算法
    if (T) {
        Visit(T->data);                                                 //访问根节点
        if (T->LTag == 0) {                                             //左子树存在，先序遍历左
子树
            PreOrderTraverse(T->lchild, Visit);
        }
        if (T->RTag == 0) {                                             //右子树存在，先序遍历右
子树
            PreOrderTraverse(T->rchild, Visit);
        }
    }
}

void InOrderTraverse(BiThrTree T, int (*Visit)(char)) {
                                                                        //中序遍历后续线索化树的
递归算法
    if (T) {
        if (T->LTag == 0) {                                             //左子树存在，中序遍历左
子树
            InOrderTraverse(T->lchild, Visit);
        }
        Visit(T->data);                                                 //访问根节点
        if (T->RTag == 0) {                                             //右子树存在，中序遍历右
子树
            InOrderTraverse(T->rchild, Visit);
        }
    }
}
```

```
void PostOrderTraverse(BiThrTree Thrt, int (*Visit)(char)) {
                                                        //后序遍历后续线索化树的非递归算法
    int flag = 0;
    BiThrTree p = Thrt->lchild;
    if (p == Thrt) return;                              //空树
    do {
        while (p->LTag == Link) p = p->lchild;          //走到树的最左端
        if (p->RTag == Link) { p = p->rchild; continue; }   //最左端节点有右孩子，结束当前循                                                    //环，开始后序遍历以该右孩子为根的子树
        while (p->RTag == Thread) { Visit(p->data); p = p->rchild; }//没有右孩子，则访问当前                                                                //节点并指向其后继
        Visit(p->data);         //退回到有右孩子的节点，此时以该节点为根的子树的左右子树已后序遍历完                                    //毕，所以访问该节点
        while ((p->parent->rchild == p) ||
               (p->parent->lchild == p && p->parent->RTag == Thread)) {
            //找该根节点的后继:是双亲右孩子或是左孩子且双亲无右孩子，则后继为双亲
            p = p->parent;
            if (p == Thrt->lchild) {                    //若回到根节点，则退出循环
                flag = 1;
                break;
            }
            Visit(p->data);
        }
        if (flag) break;
        if(p->parent->lchild == p && p->RTag == Link){  //若是双亲左孩子且双亲有右孩子，则定位到                                                    //双亲的右孩子，对以其为根的子树后序遍历
            p = p->parent->rchild;
        }
    } while (p != Thrt->lchild);
    Visit(p->data);                                                 //访问根节点
}
```

表达式树

```
//数据结构定义
typedef struct {  //每个叶节点存放字符与相应等级，常数为3；+,-为1；  * /为2
    string s;
    int grade;
}node;

typedef struct BiTNode {
    node data;
    struct BiTNode* lchild, * rchild;
}BiTNode, *BiTree;

stack<int> stk;
```

基本要求：

```
//按先序次序输入二叉树中节点的值
//构造二叉链表表示的二叉树T
void createBiTree(BiTree& T) {
    node ch;
    cin >> ch.s;
    if ((ch.s == "+") || (ch.s == "-") || (ch.s == "/") || (ch.s == "*")) {
        //若是运算符，则除了构造此节点外，还要递归的构造该节点的左孩子和右孩子
        T = new BiTNode;
        T->data = ch;
        if ((ch.s == "+") || (ch.s == "-")) T->data.grade = 1;
        else if ((ch.s == "/") || (ch.s == "*")) T->data.grade = 2;
        createBiTree(T->lchild);
        createBiTree(T->rchild);
    }
    else {      //若是运算数，则只需要构造该节点即可
        T = new BiTNode;
        T->data = ch;
        T->data.grade = 3;
        T->lchild = NULL;
        T->rchild = NULL;
    }
}
```

```
//中序遍历表达式树，输出中缀表达式,有多余括号
void PrintEle1(string S) {
    cout << S;
}
void InOrderTraverse(BiTree T, void (*Visit)(string)) {
    if (T) {
        if (T->lchild && T->lchild->lchild) cout << '(';//若当前节点的左子树仍至少2
的深度，相当于还有计算式，则添加括号
        InOrderTraverse(T->lchild , Visit);
        if(T->lchild && T->lchild->lchild) cout << ')';
        Visit(T->data.s);
        if (T->rchild && T->rchild->rchild) cout << '(';//若当前节点的右子树仍至少2
的深度，相当于还有计算式，则添加括号
        InOrderTraverse(T->rchild, Visit);
        if (T->rchild && T->rchild->rchild) cout << ')';
    }
}
```

```
//后序遍历表达式树，输出逆波兰式，并求值，计算结果放在栈stk中
void PostOrderTraverse(BiTree T, void (*Visit)(string)) {
    if (T) {
        PostOrderTraverse(T->lchild, Visit);
        PostOrderTraverse(T->rchild, Visit);
        Visit(T->data.s);
    }
}

void PrintEle(string S) {  //打印树节点的值并进行相应栈操作
    cout << S<<' ';
    int a1, b1;
    if (S == "+") {
        a1 = stk.top(); stk.pop();
        b1 = stk.top(); stk.pop();
```

```
            stk.push(b1 + a1);
        }
        else if (S == "-") {
            a1 = stk.top(); stk.pop();
            b1 = stk.top(); stk.pop();
            stk.push(b1 - a1);
        }
        else if (S == "*") {
            a1 = stk.top(); stk.pop();
            b1 = stk.top(); stk.pop();
            stk.push(b1 * a1);
        }
        else if (S == "/") {
            a1 = stk.top(); stk.pop();
            b1 = stk.top(); stk.pop();
            stk.push(b1/a1);
        }
        else {
            istringstream(S) >> a1;
            stk.push(a1);
        }
    }
```

选做要求

```
void InOrderTraverse1(BiTree T, void (*Visit)(string)) {
    //中序遍历表达式树，输出中缀表达式,无多余括号
    int flag1 = 0, flag2 = 0;
    if (T->lchild) {            //若左孩子存在(则该节点为运算符)
        if (T->lchild->data.grade < T->data.grade) {  //左孩子运算符优先级低于子树根节点，加括号
            flag1 = 1; cout << '(';
        }
        InOrderTraverse1(T->lchild, Visit);     //若节点为操作数，则没有孩子，不需要再进行对以该                                                //操作数为根的子树中序遍历
        if (flag1) cout << ')';
    }
    Visit(T->data.s);
    if (T->rchild) {                //若右孩子存在(则该节点为运算符)
        if (T->rchild->data.grade <= T->data.grade) { //右孩子运算符优先级低于或等于子树根节点，                                                                //加括号
            flag2 = 1; cout << '(';
        }
        InOrderTraverse1(T->rchild, Visit);
        if (flag2) cout << ')';
    }
}
```

调试分析

二叉树的创建与遍历

时间复杂度:

CreateBiThrTree函数为O(n)（每有一个节点则开辟空间一次，即循环一次）；PostThreading函数为O(n)（对每个节点都有操作）；PreOrderTraverse函数为O(n)；InOrderTraverse函数为O(n); PostOrderTraverse函数为O(n)。均为每个节点循环一次，每次循环为有限操作

空间复杂度：

CreateBiThrTree函数为O(n)（每有一个节点则开辟空间一次）；PostThreading函数为O(depth)(depth为树的深度，亦即栈的深度)；PreOrderTraverse函数为O(depth)；InOrderTraverse函数为O(depth); PostOrderTraverse函数为O(1)，没有额外空间消耗

遇到的问题

- 如何根据层序遍历次序的输入来构建二叉树。这个点与书中所讲的先序、中序、后序遍历不同，不是一个可以递归来简化的问题。后来想到关键还是在于模拟层序输入过程中父母节点与孩子节点的位置关系。通过队列入队和出队，能使每次读入的值为当前所指节点的孩子节点。
- 解决后序线索化：巧妙利用书中所给中序线索化的递归算法，通过对遍历顺序的改变，很容易便实现后序线索化
- 如何实现非递归的后序遍历:这里涉及到对后序遍历顺序上节点的依次访问，需要许多判断条件来决定当前节点的后继是什么，所以在完成时自己先把所有可能情况列出，然后以一个例子为例，按后序流程走，看代码判断条件是否正确。清晰地条件分类有助于高效完成代码，也为debug带来很大的帮助

表达式树

时间复杂度:

createBiTree函数为O(n)（每有一个节点则循环一次），PostOrderTraverse函数为O(n)（对每个节点遍历操作），InOrderTraverse，InOrderTraverse1函数为也O(n)（对每个节点遍历操作）

空间复杂度:

createBiTree函数为O(n)（每有一个节点则开辟一个空间），PostOrderTraverse，InOrderTraverse，InOrderTraverse1函数均为O(depth)(depth为树的深度，亦即递归栈的深度)

遇到的问题:

- 如何来对括号添加与否做一个判断:从运算符优先级入手，一个运算符左边的式子的根运算符优先级低，则加括号；一个运算符右边的式子的根运算符优先级低或等，则加括号
- 如何利用好表达式树上操作数均在叶节点，运算符均在非叶节点
- 如何通过逆波兰式求值:关键是辅助栈的利用。

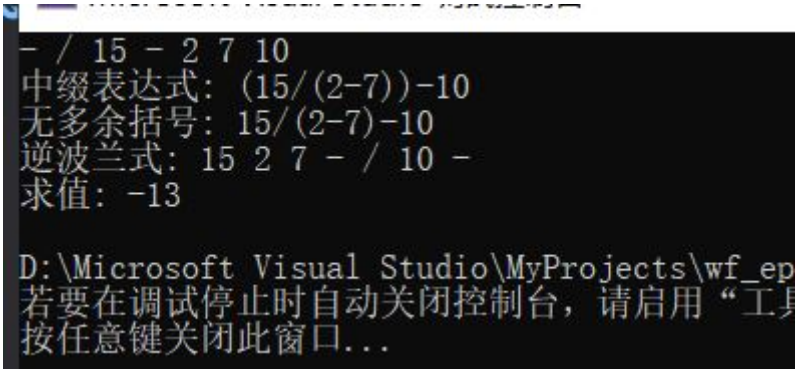
代码测试

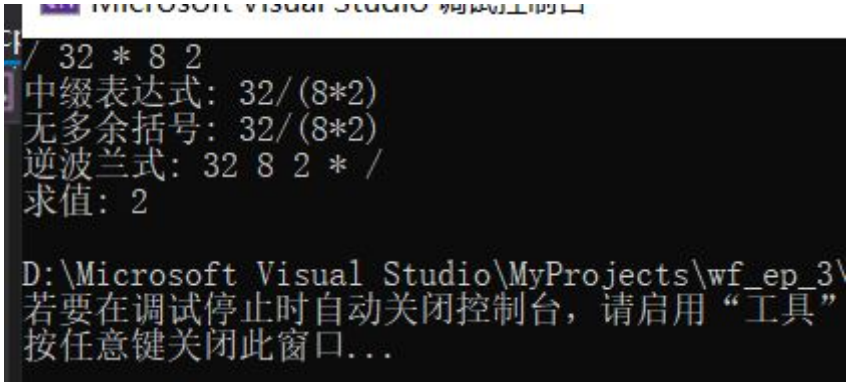
二叉树的创建与遍历

按输入样例的格式运行程序并输入即可,虚节点用空格代替

表达式树

按样例输入即可，输入的每个运算符、操作数之间都会有空格隔开





实验总结

通过本次实验，自己对于二叉树的构建以及先序中序后序层序的概念有了更深刻地理解，加强了对二叉树存储表示特征与各次序遍历的联系的认知，对于各种顺序的构建树、遍历树的递归算法和非递归算法更加熟悉，深刻认识到代码其实也就是对现实生活中计算的模拟，关键把真实求解过程的条件分清楚，把递归或循环的层次弄懂，弄透彻，那么实现相应功能的代码便自然能很容易实现。

此外，在表达式树实验中，在亲身编写相应算法后，对前缀式，中缀式，后缀式的求值方式、以及与表达式树的对应有了更深层次的理解。学会通过分析以及算术表达式特点的解析，找出合适的数据结构来计算算术表达式的思想也让我受益匪浅。在二叉树一节中，也要用到栈、队列的相关的知识，恰是说明所有知识的融汇贯通才是掌握一门课程的核心所在，也让我明白后期学习的方向，温故而知新。

另外，在写代码之前现在草稿纸上理清函数逻辑，并在敲代码时多推敲每行代码的逻辑正确性，能有效提高代码的正确性，大大减少了自己debug时间。