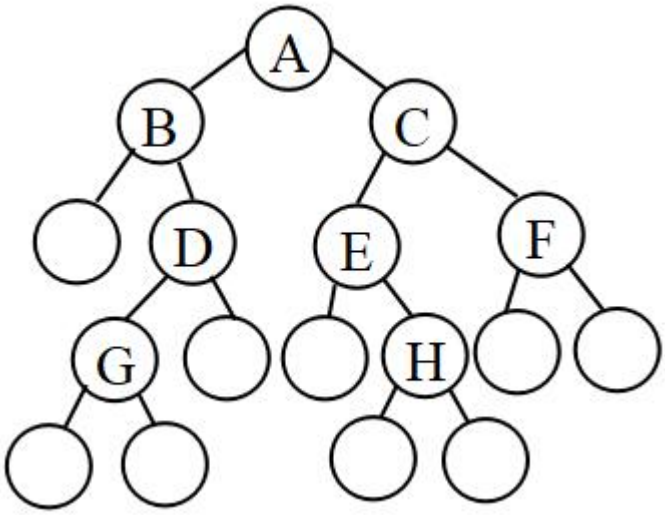
数据结构实验 03

一．实验要求

1. 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。

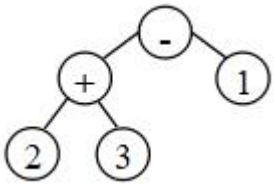
构建这颗二叉树(不包含图中的虚结点)，并增加左右标志域，将二叉树后序线索化



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。

2. 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。

对应的表达式树如下:



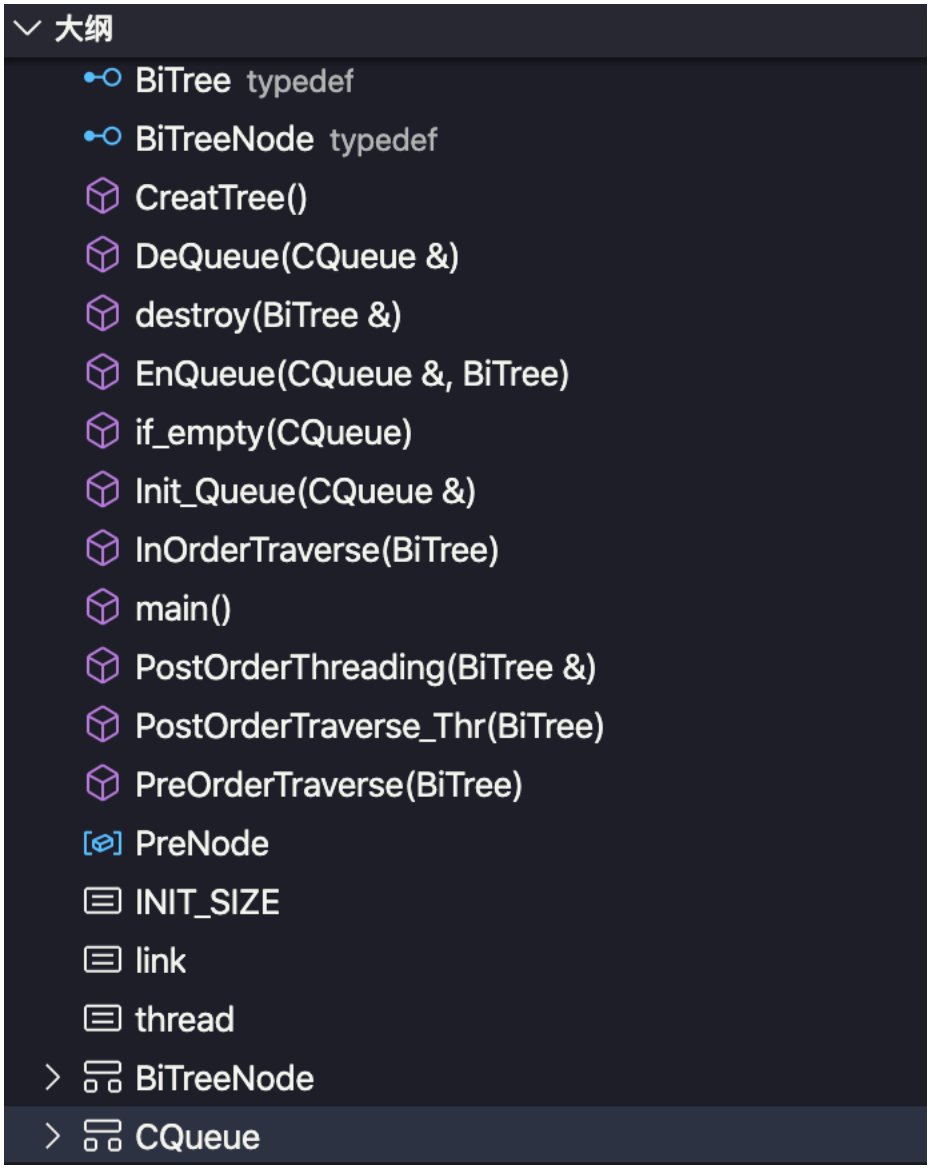
选做：

输出的中缀表达式中不含有多余的括号。

二．设计思路

1. 二叉树的创建与遍历：

大纲如下：



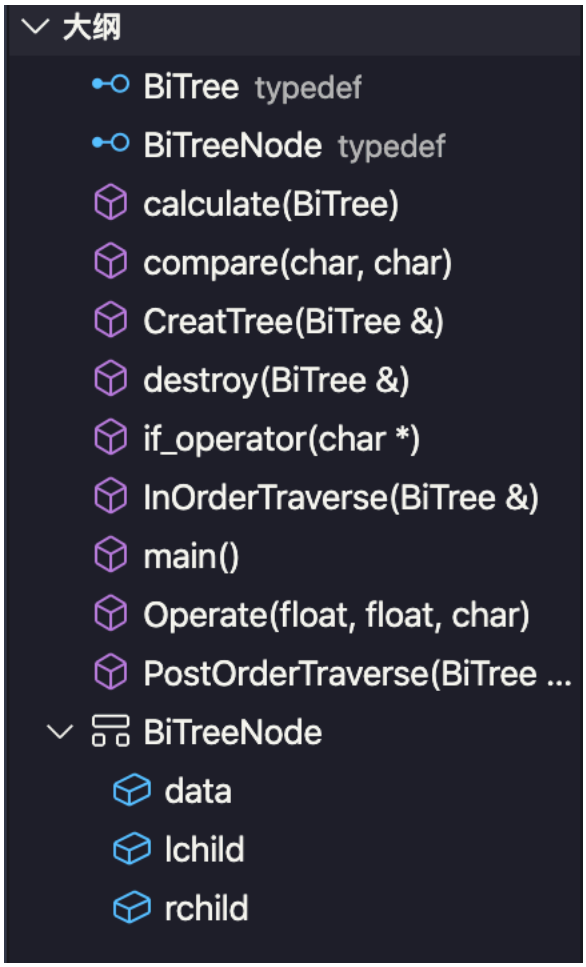
其中 Init_Queue、EnQueue、DeQueue、if_empty 实现队列的基本操作：初始化、进队列、出队列、判断队列是否为空；

CreatTree 利用队列按层次建立二叉树，并后序线索化该二叉树；

遍历时，先序、中序使用递归算法，后序利用线索化的性质，采用非递归方式遍历；

最后调用 destroy 函数释放申请的内存空间。

2. 表达式树：



其中 data 为 char 型数组，可以存放运算符（仅占用第一个空间）、多位浮点数、负数：

```
typedef struct BiTreeNode{
    char data[10];
    BiTreeNode *lchild;
    BiTreeNode *rchild;
}BiTreeNode,*BiTree;
```

if_operator 函数用于判断该结点数据是否为运算符；
compare 函数用于比较两个运算符的优先级；
CreatTree 用波兰式先序建树；InOrder 与 PostOrder 按中序、后序遍历输出中缀、后缀表达式，calculate 函数用于求表达式的值

三．关键代码讲解

1. 二叉树的创建与遍历：

   CreatTree 建树时用

```
BiTree CreatTree(){
   char c1;
   cin>>c1;
   if(c1 == '#')
       return(NULL);
   BiTree bt = (BiTree)malloc(sizeof(BiTreeNode));
   bt->data = c1;
   bt->lchild  = NULL;
   bt->rchild  = NULL;
   bt->parent  = NULL;
   CQueue Q;
   Init_Queue(Q);
   EnQueue(Q,bt);  //根结点进队列
   while(!if_empty(Q))
   {
       BiTree p = DeQueue(Q);  //下一个结点出队列，等待判断其左右孩子的实/虚
       cin>>c1;
       if(c1 != '#')
       {//p 左孩子为实结点
           p->lchild = (BiTree)malloc(sizeof(BiTreeNode));
           p->lchild->data = c1;
           p->lchild->parent = p;
           p->lchild->lchild = NULL;
           p->lchild->rchild = NULL;
           EnQueue(Q,p->lchild);   //p 的左孩子进队列，等待被赋为实/虚结点
       }
       cin>>c1;
       if(c1 != '#')
       {//p 右孩子为实结点
```

```
            p->rchild = (BiTree)malloc(sizeof(BiTreeNode));
            p->rchild->data = c1;
            p->rchild->parent = p;
            p->rchild->lchild = NULL;
            p->rchild->rchild = NULL;
            EnQueue(Q,p->rchild);    //p 的右孩子进队列，等待被赋为实/虚结点
        }
    }
    return(bt);
}//CreatTree
```

当结点为实结点时，为结点申请空间，并将其压入队列，按顺序等待判断其左/右孩子是否为实结点，并依次进行赋值；当队列为空时二叉树建立完毕。由此建立二叉树。

后序线索化的算法：

```
BiTree PreNode = NULL;  //全局变量 PreNode 记录前驱结点的位置
void PostOrderThreading(BiTree  &bt){
    if(!bt)
        return;
    PostOrderThreading(bt->lchild); //线索化左右子树
    PostOrderThreading(bt->rchild);

    if(bt->lchild == NULL)
    {
        bt->lchild = PreNode;
        bt->ltag = thread;
    }
    else
        bt->ltag = link;
    if(PreNode != NULL && PreNode->rchild == NULL)
    {
        PreNode->rchild = bt;
        PreNode->rtag = thread;
```

```
    }
    else if(PreNode != NULL)
        PreNode = link;

    PreNode =  bt;
    return;
}//PostOrderThreading
```

设置了全局变量 PreNode 来记录上一个被 visit 的结点的地址，利用递归完成线索化

后序非递归遍历二叉树（利用线索化）：

```
void PostOrderTraverse_Thr(BiTree  bt){
    //非递归遍历后序线索二叉树
    BiTree p;
    p = bt;
    while(p->lchild)
        p = p->lchild;       //找到后序遍历第一个被 visit 的结点
    for(;;)
    {
        cout<<p->data;
        if(p == bt)
        //当 p 为根结点时，不再有后继，结束循环
            break;
        else if(p->rtag == thread)
        {//若 p 的右孩子已被线索化，则 p 直接前往它的后继（它的右孩子）
            p = p->rchild;
            continue;
        }
        else if(( p->parent->rtag == link && p == p->parent->rchild )//若 p 为
右孩子
                ||             p->parent->rtag == thread          )//或者 p 为
左孩子且 p 的父结点无右孩子
```

```
            p = p->parent;                                          //此时 p 的
后继为其父结点
        else
        {//若 p 为父结点的左孩子且其父结点有右孩子，则 p 的后继为其父结点右子树中第一个被
visit 的结点
            p = p->parent->rchild;
            while(!(p->ltag && p->rtag))
            {
                if(p->ltag == link)
                    p = p->lchild;
                else
                    p = p->rchild;
            }
        }
    }
    return;
}//PostOrderTraverse_Thr
```

设立一个指向结构体的指针 p，先用 p 找到二叉树后序遍历过程中第一个被 visit 的结点，之后始终用 p 来指向下一个结点的位置。p 的后继的求法：

若 p 指向根结点，则 p 无后继；

若 p 右子树被线索化，则 p 后继为其 rchild 所指向的结点；

若 p 为其 parent 的右孩子 或 为左孩子且 parent 无右孩子，则 p 的后继为其 parent；

若以上都不成立，则 p 后继为其 parent 结点右子树中第一个被 visit 的结点。

由此完成二叉树的遍历。

2. 表达式求值：

   InOrderTraverse 输出中缀表达式：

```
void InOrderTraverse(BiTree &bt){
    int status = 0; //status 用于判断是否需要匹配括号
    if(bt != NULL)
```

```
    {
        if(if_operator(bt->data) && if_operator(bt->lchild->data))
            if(compare(*bt->data,*bt->lchild->data) == 1)
            {
                cout<<"(";
                status = 1;
            }
        InOrderTraverse(bt->lchild);
        if(status == 1)
        {
            cout<<")";
            status = 0;
        }
        atof(bt->data)<0 ? cout<<"("<<bt->data<<")":cout<<bt->data;
        if(if_operator(bt->data) && if_operator(bt->rchild->data))
            if(compare(*bt->data,*bt->rchild->data) != -1)
            {
                cout<<"(";
                status = 1;
            }
        InOrderTraverse(bt->rchild);
        if(status == 1)
            cout<<")";
    }
    return;
}//InOrderTraverse
```

在三种情况下需要匹配括号：

（1） 该结点与左孩子结点 data 都为运算符且左孩子的优先级低

（2） 该结点与右孩子结点 data 都为运算符且右孩子优先级不高于该结点

（3） 即将输出的 data 是操作数且为负数

caiculate 函数计算表达式树的值：

```
float calculate(BiTree bt){
    float a,b;
    if(if_operator(bt->lchild->data))
        a = calculate(bt->lchild);
    else
        a = atof(bt->lchild->data);
    if(if_operator(bt->rchild->data))
        b = calculate(bt->rchild);
    else
        b = atof(bt->rchild->data);
    float result = Operate(a,b,*bt->data);
    return(result);
}//calculate
```

利用递归算法，若孩子存储的是运算符则进行递归得到子树的值，否则孩子的值就是它们存储的操作数。得到左右子树的值后，调用 Operator 函数返回左右子树值与根结点运算符的计算结果

四．调试分析

1. 二叉树的创建与遍历：
   二叉树递归遍历的时间复杂度为 O(N),因为共 N 个元素且每个元素均被 visit 一次
   空间复杂度的最坏情况是 O(N) ,即对于左/右斜二叉树，使用的递归栈空间是 O(N)　　而由二叉树的性质，一般情况下的平均空间复杂度是 O(logN)

   对于线索化后的后序非递归遍历，其时间复杂度为 O(N),空间复杂度为 O(1)

2. 表达式求值：

遍历函数使用的空间与上一题类似。

Calculate 函数的时间复杂度为 O(N)，其递归栈所占用的空间最大为树的深度，最坏情况下与 N 成线性关系，平均情况下与 logN 成线性关系

五．代码测试

1.第一题：测试如下：

```
A#BC##DE##F##
先序遍历如下:
ABCDEF
中序遍历如下:
ACEFDB
后序遍历如下:
FEDCBA
```

```
ABCD##E##F###
先序遍历如下:
ABDCEF
中序遍历如下:
DBACFE
后序遍历如下:
DBFECA
```

2.第二题（表达式求值）：

主页上给出测试用例如右：

```
/ + 15 * 5 + 2 18 5
(15+5*(2+18))/5
15 5 2 18 + * + 5 /
23
```

将 5 改为-5，中缀表达式会为其匹配括号

```
/ + 15 * 5 + 2 18 -5
(15+5*(2+18))/(-5)
15 5 2 18 + * + -5 /
-23
```

若将 5 改为小数，也可运算：

```
/ + 15 * 5 + 2 18 5.77
(15+5*(2+18))/5.77
15 5 2 18 + * + 5.77 /
19.9307
```

六．实验总结

(1)通过本次实验，我以二叉树的链式表示、建立和应用为基础，深入了解二叉树的存储表示特征以及遍历次序与二叉树的存储结构之间的关系，进一步掌握利用遍历思想解决二叉树中相关问题的方法。

(2)通过思考、上机实践与分析总结，理解计算机进行算术

表达式解析、计算的可能方法，初步涉及一 些编译技术，增加自己今后学习编译原理的兴趣，并奠定一些学习的基础。

(3)在按层次创建二叉树时，回顾并使用了队列这一数据结构

(4)通过线索二叉树的遍历，我体会到了线索二叉树遍历时空间复杂度小的优势

七．附录

“PB18071496_李昱祁_03.pdf”

“PB18071496_李昱祁_03_1.cpp”

“PB18071496_李昱祁_03_2.cpp”