

# 哈尔滨工业大学计算机科学与技术学院

## 实验报告

课程名称：数据结构与算法分析

课程类型：必修

实验项目名称：树型结构的建立与遍历

实验题目：树型结构的建立与遍历

班级： 1303101

学号： 1130310128

姓名： 杨尚斌

设计成绩	报告成绩	指导老师
		张岩

## 一、 实验目的

树型结构的遍历是树型结构算法的基础,本实验要求编写程序演示二叉树的存储结构的建立和遍历过程。

## 二、 实验要求及实验环境

### 1.实验环境:

CodeBlocks

Windows 7 sp1 64x

Gcc 4.7.1

### 2.实验要求

(1) 至少采用两种方法,编写建立二叉树的二叉链表存储结构的程序,并(用广义表的形式)显示并保存二叉树;

(2) 采用二叉树的二叉链表存储结构,编写程序实现二叉树的先序、中序和后序遍历的递归和非递归算法以及层序遍历算法,并显示二叉树和相应的遍历序列;

(3) 在二叉树的二叉链表存储结构基础上,编写程序实现二叉树的先序或中序或后序线索链表存储结构建立的算法,并(用广义表的形式)显示和保存二叉树的相应的线索链表;

(4) 在二叉树的线索链表存储结构上,编写程序分别实现求一个结点的先序(中序、后序)的后继结点的算法;

(5) 在 (4) 基础上，编写程序实现对线索二叉树进行先序、中序和后序遍历的非递归算法，并显示线索二叉树和相应的遍历序列。实验内容：树型结构的建立与遍历

### 三、设计思想

#### 1. 物理设计

```
struct BTree
{
    BTree *lchild;
    char data;
    BTree *rchild;
    bool ltag;
    bool rtag;
};
```

定义 BTree 的结构，里面包含左子树，右字数，data 域，左标记，右标记。

```
char withList[100];
int i = 0;
```

用于广义表的储存，withList 数组用于广义表的储存。

```
int main()
```

主函数的开始

```
preCreateTree()
```

递归建立二叉树

```
printList(pre_tree)
```

广义表的输出,用于广义表开头，结尾，以及部分地区的‘，’

```
printWithList(pre_tree)
```

广义表的正常输出，与 printList()配合使用构成广义表

```
listTree(pre_tree)
```

开始树的遍历

```
listTreePre(BT)
```

树的递归先序遍历

`listTreeMid(BT)`

树的递归中序遍历

`listTreeBeh(BT)`

树的递归后序遍历

`listTreePreNode(BT)`

树的非递归先序遍历

`listTreeMidNode(BT)`

树的非递归中序遍历

`listTreeBehNode(BT)`

树的非递归后序遍历

`levCreateTree()`

输入树的结点位置建立树的结构

`toTreeList(pre_tree)`

树的线索化

`toTreeListPre(BT, HEAD)`

`preing(BTtree p)`

树的先序线索化

`toTreeListPrePrint(HEAD)`

树的先序线索化输出

`resetTreeList(HEAD, BT)`

重置树的结构，访问线索化后结构的紊乱

`toTreeListMid(BT, HEAD)`

`miding(BTtree p)`

树的中序线索化

`toTreeListMidPrint(HEAD)`

树的中序线索化之后的输出

`toTreeListBeh(BT, HEAD)`

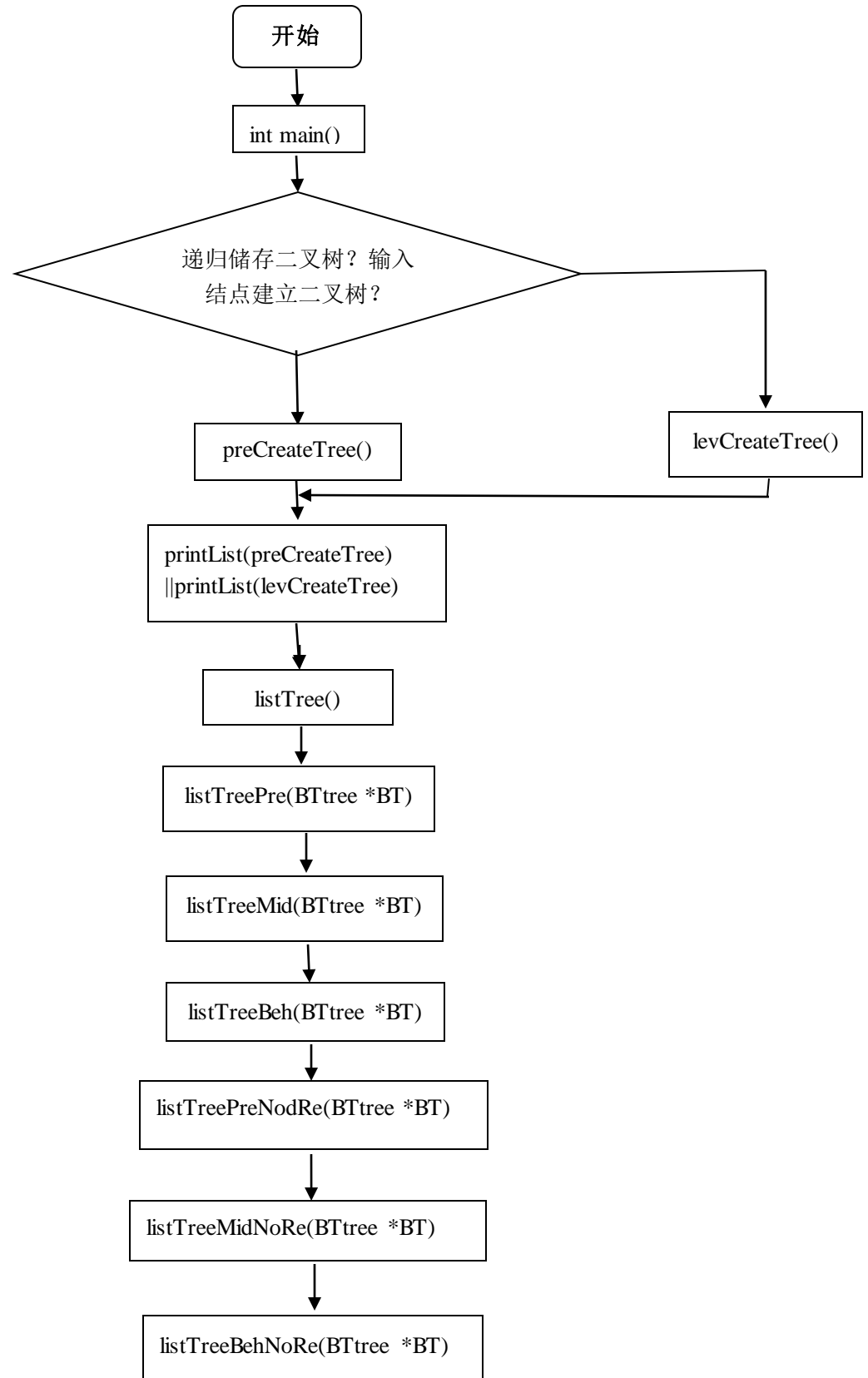
`behing(BTtree p)`

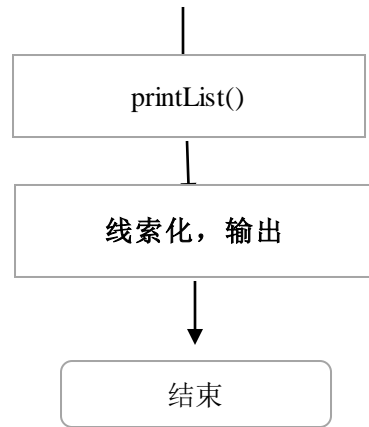
树的后序线索化

toTreeListBenPrint(HEAD)

树的后序线索化后的输出

## 2. 逻辑设计





### 三、测试结果

```
D:\cpp\kkk\bin\Debug\kkk.exe
输入数字选择菜单：
1.递归建立二叉树
2.输入结点的数建立二叉树
1
说明：#代表空
输入根节点
A
输入A的左子树
B
输入B的左子树
C
输入C的左子树
D
输入D的左子树
#
输入D的右子树
#
输入C的右子树
E
输入E的左子树
F
输入F的左子树
#
输入F的右子树
#
输入E的右子树
#
输入B的右子树
G
输入G的左子树
#
输入G的右子树
#
输入A的右子树
#
树的储存已经完成!

===广义表的输出===
(A(B(C(D,E(F,)),G),))
半：
```

```
D:\cpp\kkk\bin\Debug\kkk.exe
输入A的右子树
#
树的储存已经完成!

===广义表的输出===
<A<B<C<D,E<F,>>,G>,>>

===开始树的遍历操作===

=递归实现：=

先序遍历：
A B C D E F G
中序遍历：
D C F E B G A
后序遍历：
D F E C G B A
=非递归实现：=

先序遍历：
A B C D E F G
中序遍历：
D C F E B G A
后序遍历：
D F E C G B A ==树的线索化==

Process returned 0 (0x0)   execution time : 36.317 s
Press any key to continue.

半：
```



```
选定 D:\cpp\kkk\bin\Debug\kkk.exe
输入数字选择菜单：
1.递归建立二叉树
2.输入结点的数建立二叉树

2
输入结点数：1
输入data域：A
输入结点数：
2
输入data域：
B
输入结点数：
3
输入data域：
C
输入结点数：
4
输入data域：
D
输入结点数：
5
输入data域：
F
输入结点数：
半：
```

```
选定 D:\cpp\kkk\bin\Debug\kkk.exe
6
输入data域：
#
输入结点数：
#
输入data域：
树的储存已经完成！

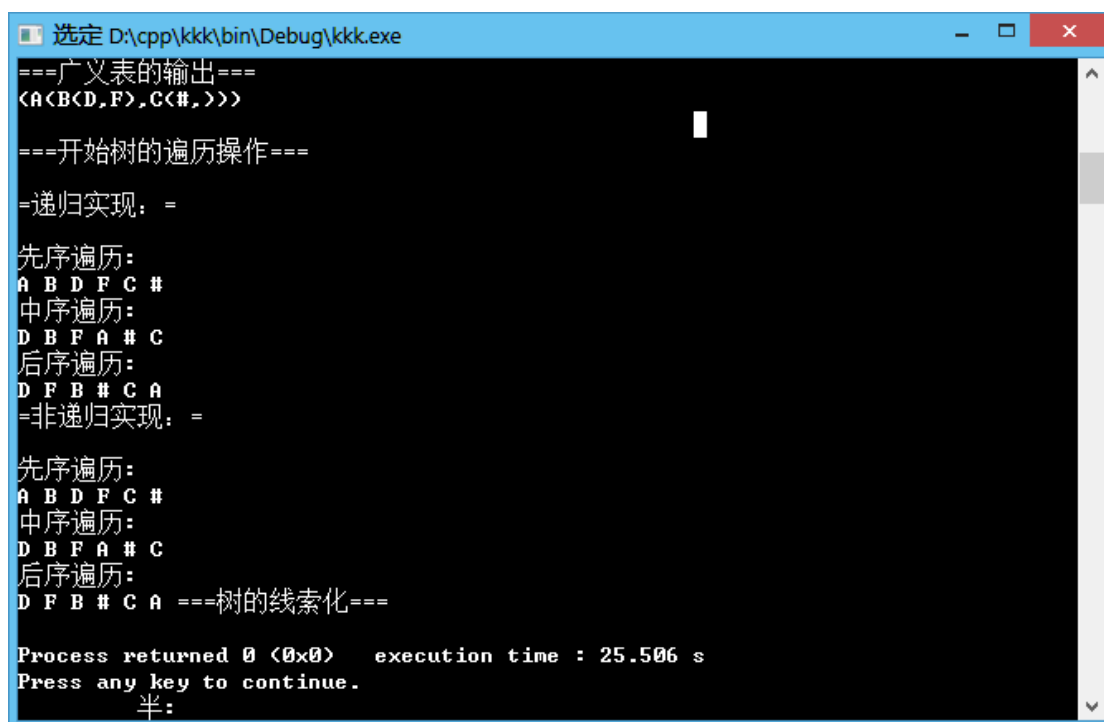
===广义表的输出===
(A(B(D,F),C(#,)))

===开始树的遍历操作===

-递归实现：=

先序遍历：
A B D F C #
中序遍历：
D B F A # C
后序遍历：
D F B # C A
-非递归实现：=

半：
```



```
选定 D:\cpp\kkk\bin\Debug\kkk.exe
===广义表的输出===
<A<B<D,F>,C<#,>>>

===开始树的遍历操作===

-递归实现: =
先序遍历:
A B D F C #
中序遍历:
D B F A # C
后序遍历:
D F B # C A
-非递归实现: =
先序遍历:
A B D F C #
中序遍历:
D B F A # C
后序遍历:
D F B # C A
===树的线索化===

Process returned 0 (0x0) execution time : 25.506 s
Press any key to continue.
半:
```

(注：线索化的运行暂时有问题，故没有截图)

## 四、系统不足与经验体会

1. 在初始阶段对树的认识不清，初期阶段难以建立一个能使用的二叉树。

2. 对后面的线索化完成度不是很好，思路正确，后部分的代码还无法编译通过。

3. 对指针的操作还不是很熟练，容易出现难以预料的BUG。

4. 在编码的过程中遇到许多问题，在一些开源社区完美得到了解决。

## 六、附录：源代码（带注释）

.cpp

```
#include <iostream>
#include <stdlib.h>
```

```

#include <string.h>
#define MaxSize 100
using namespace std;
/*
*定义树的结构，左儿子的指针和右儿子的指针，树的 data 域
*/
struct BTtree
{
    BTtree *lchild;
    char data;
    BTtree *rchild;
    bool ltag;
    bool rtag;
};
/*
*用于广义表的存储
*/
char withList[100];
int i = 0;
//刚访问过的节点
BTtree pre;
/*
*递归实现二叉树的储存
*/
BTtree * preCreateTree()
{
    char ch;
    BTtree *BT;
    cin >> ch;
    if(ch == '#')
    {
        BT = NULL;
    }
    else
    {
        BT = new BTtree;
        BT -> data = ch;
        BT -> ltag = false;
        BT -> rtag = false;
        cout << "输入" << ch << "的左子树" << endl;
        BT -> lchild = preCreateTree();
        cout << "输入" << ch << "的右子树" << endl;
        BT -> rchild = preCreateTree();
    }
}

```

```

        return BT;
    }
    /*
    *前序递归遍历
    */
    void listTreePre(BTtree * BT)
    {
        if(BT != NULL)
        {
            cout << BT ->data << " ";
            listTreePre(BT -> lchild);
            listTreePre(BT -> rchild);
        }
    }
    /*
    *中序递归遍历
    */
    void listTreeMid(BTtree * BT)
    {
        if(BT != NULL)
        {
            listTreeMid(BT -> lchild);
            cout << BT -> data << " ";
            listTreeMid(BT -> rchild);
        }
    }
    /*
    *后序递归遍历
    */
    void listTreeBeh(BTtree * BT)
    {
        if(BT != NULL)
        {
            listTreeBeh(BT -> lchild);
            listTreeBeh(BT -> rchild);
            cout << BT -> data << " ";
        }
    }
    /*
    *前序非递归遍历
    */
    void listTreePreNoRe(BTtree * BT)
    {
        BTtree *S[MaxSize];

```

```

int top = MaxSize;
while(BT != NULL || top != MaxSize)
{
    while(BT !=NULL)
    {
        //一直往做走
        cout << BT -> data << " ";
        S[--top] = BT;
        BT = BT -> lchild;
    }
    if(top != MaxSize)
    {
        //往上反一个
        BT = S[top++];
        BT = BT -> rchild;
    }
}

}
/*
*中序非递归遍历
*/
void listTreeMidNoRe(BTtree * BT)
{
    BTtree *S[MaxSize];
    int top = MaxSize;
    while(BT != NULL || top != MaxSize)
    {
        while(BT !=NULL)
        {
            S[--top] = BT;
            BT = BT -> lchild;
        }
        if(top != MaxSize)
        {
            //上一层
            BT = S[top++];
            cout << BT -> data << " ";
            BT = BT -> rchild;
        }
    }
}
/*
*后序非递归排序

```

```

*/
void listTreeBehNoRe(BTtree * BT)
{
    struct tep
    {
        BTtree *tree;
        int flag;
    } S[MaxSize];
    int top = MaxSize;

    BTtree * tepTree = BT;
    while(tepTree != NULL || top != MaxSize)
    {
        if(tepTree != NULL)
        {
            //一直往左走，保存入栈
            S[--top].tree = tepTree;
            S[top].flag = 1;
            tepTree = tepTree -> lchild;
        }
        else
        {
            if(S[top].flag == 2)
            {
                BT = S[top++].tree;
                cout << BT -> data << " ";
            }
            else
            {
                //没有左子树，去右子树看看
                S[top].flag = 2;
                tepTree = S[top].tree -> rchild;
            }
        }
    }
}

//遍历操作
void listTree(BTtree * BT)
{
    cout << "递归实现: =" << endl << endl;
    cout << "先序遍历:" << endl;
    listTreePre(BT);
    cout << endl;
}

```

```

    cout << "中序遍历:" << endl;
    listTreeMid(BT);
    cout << endl;
    cout << "后序遍历:" << endl;
    listTreeBeh(BT);
    cout << endl << "非递归实现: =" << endl << endl;
    cout << "先序遍历:" << endl;
    listTreePreNoRe(BT);
    cout << endl;
    cout << "中序遍历:" << endl;
    listTreeMidNoRe(BT);
    cout << endl;
    cout << "后序遍历:" << endl;
    listTreeBehNoRe(BT);
}
/*
*广义表的输出
*递归操作，所以对存广义表的数组和控制下标变量是全局变量
*/
void printWithList(BTtree * BT)
{
    if(BT != NULL)
    {
        if(BT -> lchild == NULL && BT -> rchild == NULL)
        {
            withList[i++] = BT -> data;
        }
        else
        {
            withList[i++] = BT -> data;
            withList[i++] = '(';
            printWithList(BT -> lchild);
            withList[i++] = ',';
            printWithList(BT -> rchild);
            withList[i++] = ')';
        }
    }
}
//按广义表输出，此函数主要用来给广义表的头尾添加括号
void printList(BTtree *BT)
{
    BTtree *tep = BT;
    //初始化 i
    i = 0;

```

```

        withList[i] = '(';
        i++;
        printWithList(tep);
        withList[i] = ')';
        i++;
        withList[i] = '\\0';
    }
    /*
    *按节点数建立相应二叉树
    */
    BTtree * levCreateTree()
    {
        int m, l;
        char ch;
        BTtree * S[MaxSize];
        BTtree * BT, * tep;
        cout << "输入结点数: ";
        cin >> m;
        cout << "输入 data 域:";
        cin >> ch;
        while(m != 0 || ch != '#')
        {
            tep = new BTtree;
            tep -> data = ch;
            tep -> lchild = NULL;
            tep -> rchild = NULL;
            S[m] = tep;
            if(m == 1)
                BT = tep;
            else
            {
                l = m/2;
                if(m % 2 == 0)
                {
                    S[l] -> lchild = tep;
                }
                else
                {
                    S[l] -> rchild = tep;
                }
            }
            cout << "输入结点数: " << endl;
            cin >> m;
            cout << "输入 data 域:" << endl;

```



```

        cin >> ch;
        cout << "m" << m << "ch:" << ch << endl;
    }
    return BT;
}
/*
*中序二叉树线索化辅助函数
*/
void miding(BTtree p)
{
    if(p != NULL)
    {
        miding(p -> lchild);
        //左子树线索化
        if(p -> lchild == NULL)
        {
            p -> lchild = pre;
            p -> ltag = true;
        }
        //右子树线索化
        if(pre -> rchild == NULL)
        {
            pre -> rchild = p;
            pre -> rtag = true;
        }
        pre = p;
        miding(p -> rchild);
    }
}
/*
*中序线索化的起始函数
*/
bool toTreeListMid(BTtree BT, BTtree &HEAD)
{
    miding(BTtree p);
    HEAD = new BTtree;
    HEAD -> ltag = false;
    HEAD -> rtag = true;
    HEAD -> rchild = HEAD;
    if(BT == NULL)
    {
        HEAD -> lchild = HEAD;
    }
    else

```

```

    {
        HEAD -> lchild = BT;
        pre = HEAD;
        miding(BT);
        pre -> rchild = HEAD;
        pre -> rtag = true;
        HEAD -> rchild = pre;
    }
    return 1;
}
//中序线索化的输出
void toTreeListMidPrint(BTtree HEAD)
{
    BTtree p = HEAD -> lchild;
    while(p != HEAD)
    {
        while(HEAD -> ltag == false)
        {
            p = p -> lchild;
        }
        cout << p -> data << "->";
        while(p -> rtag == true && p -> rchild != HEAD)
        {
            p = p -> rchild;
            cout << p -> data << endl << p -> data << "->";
        }
        p = p -> rchild;
    }
}
/*
*前序线索二叉树辅助函数
*/
void preing(BTtree p)
{
    if(p != NULL)
    {
        //前驱线索化
        if(p -> lchild == NULL)
        {
            p -> lchild = pre;
            p -> ltag = true;
        }
        //后继线索化
        if(pre -> rchild == NULL)

```

```

        {
            pre -> rchild = p;
            pre -> rtag = true;
        }
        pre = p;
        //左右子树线索化
        if(p -> ltag == false)
        {
            preing(p -> lchild);
        }
        if(p -> rtag == false)
        {
            preing(p -> rchild);
        }
    }
}
/*
*前序线索化二叉树的起始函数
*/
bool toTreeListPre(BTtree BT, BTtree &HEAD)
{
    preing(BTtree p);
    HEAD = new BTtree;
    HEAD -> ltag = false;
    HEAD -> rtag = true;
    HEAD -> rchild = HEAD;
    if(BT == NULL)
    {
        HEAD -> lchild = HEAD;
    }
    else
    {
        HEAD -> lchild = BT;
        pre = HEAD;
        preing(BT);
        pre -> rchild = HEAD;
        pre -> rtag = true;
        HEAD -> rchild = pre;
    }
    return 1;
}
/*
*前序线索化二叉树的输出函数
*/

```

```

void toTreeListPrePrint(BTtree HEAD)
{
    BTtree p = HEAD -> lchild;
    while(p != HEAD)
    {
        cout << p -> data << "->";
        while(p -> ltag == false)
        {
            p = p -> lchild;
            cout << p -> data << "->";
        }
        while(p -> rtag == true && p -> rtag != HEAD)
        {
            p = p -> rchild;
            cout << p -> data << "->";
        }
        if(p -> ltag == false)
        {
            p = p -> lchild;
        }
        else
        {
            p = p -> rchild;
        }
    }
}

/*
*后序线索化辅助函数
*/
void behing(BTtree p)
{
    if(p != NULL)
    {
        behing(p -> lchild);
        behing(p -> rchild);
        //前驱搜索
        if(p -> lchild == NULL)
        {
            pre -> rtag = true;
            pre -> lchild = p;
        }
        //后驱搜索
        if(pre -> rchild == NULL)
        {

```

```

        pre -> rtag = true;
        pre -> rchild = p
    }
    pre = p;
}
}
/*
*后序线索化二叉树的起始函数
*/
BTtree toTreeListBeh(BTtree &BT)
{
    BTtree beh;
    beh = new BTtree;
    beh -> ltag = false;
    beh -> rtag = true;
    beh -> rchild = beh;
    if(BT == NULL)
    {
        beh -> lchild = beh;
    }
    else
    {
        beh -> lchild = BT;
        pre = beh;
        behing(BT);
        beh -> rchild = pre;
    }
    return beh;
}
/*
*找爸爸
*/
BTtree Parent(BTtree & HEAD, BTtree & p)
{
    BTtree tep;
    tep = HEAD;
    //tep 是头节点
    if(tep -> lchild == p)
    {
        return tep;
    }
    else
    {
        tep = tep -> lchild;
    }
}

```

```

while(tep -> lchild != p && tep -> rchild != p)
{
    if(tep -> rtag == false)
    {
        //有右节点，往右走
        tep = tep -> rchild;
    }
    else
    {
        //左儿子，到前驱
        tep = tep -> lchild;
    }
}
return tep;
}
}
/*
*后序线索化二叉树的输出函数
*/
void toTreeListBehPrint(BTtree HEAD)
{
    BTtree p, parent;
    p = HEAD -> lchild;
    while(true)
    {
        while(p -> ltag == 0)
            p = p -> lchild;
        if(p -> rtag == 0)
            p = p -> rchild;
        else
            break;
    }
    while(HEAD != p)
    {
        cout << p -> data << "->";
        parent = Parent(HEAD, P);
        //如果 parent 是 HEAD,则 p 是根节点,没有后继
        if(HEAD == parent)
            p = HEAD;

        else if(parent -> rchild == p || parent -> rtag == true)
        {
            p = parent;
        }
    }
}

```

```

//如果 p 是双亲的右孩子，或者是独生左孩子，，则后继为双亲
else
{
    while(parent -> rtag == false)
    {
        //若 P 是有兄弟的左孩子，则后继为双亲的右子树上后序遍历访问的第
一个节点
        parent = parent -> rchild;
        while(parent -> ltag == false)
        {
            parent = parent -> lchild;
        }
        p = parent;
    }
    cout << p -> data << endl;
}
}
/*
*线索化之后 reset 一下 HEAD, BT 等东西，防止影响下一个线索化
*/
void resetTreeList(BTtree HEAD, BTtree &BT)
{
    BTtree p, post;
    p = HEAD -> lchild;
    while(p != HEAD)
    {
        while(p -> ltag == false)
            p = p -> lchild;
        p -> ltag = false;
        p -> lchild = NULL;
        while(p -> rtag == true && p -> rchild != HEAD)
        {
            p -> rtag = false;
            post = p -> rchild;
            p -> rchild = NULL;
            p = post;
        }
        p = p -> rchild;;
    }
    p = HEAD -> rchild;
    p -> rtag = false;
    p -> rchild = NULL;
    BT = HEAD -> lchild;
}

```

```

        delete(HEAD);
    }
    /*
    *线索化二叉树的主函数
    */
void toTreeList(BTtree * BT)
{
    //头节点
    BTtree * HEAD;
    cout << "==前序==" << endl;
    toTreeListPre(BT, HEAD);
    cout << "遍历前序线索化二叉树: " << endl;
    toTreeListPrePrint(HEAD);
    resetTreeList(HEAD, BT);
    cout << "==中序==" << endl;
    toTreeListMid(BT, HEAD);
    cout << "遍历中序线索化二叉树: " << endl;
    toTreeListMidPrint(HEAD);
    resetTreeList(HEAD, BT);
    cout << "==后序==" << endl;
    BTtree beh;
    //会改变 BT
    beh = toTreeListBehPrint(BT);
    cout << "遍历后序线索化二叉树: " << endl;
    toTreeListBehPrint(beh);
    resetTreeList(ben, BT);
}
//主函数
int main()
{
    BTtree *pre_tree = NULL;
    BTtree *lev_tree = NULL;
    int n;
    //菜单的建立
    cout << "输入数字选择菜单: " << endl << "1.递归建立二叉树" << endl <<
    "2.输入结点的数建立二叉树" << endl << endl;
    cin >> n;
    if(n == 1)
    {
        cout << "说明: #代表空" << endl << "输入根节点" << endl << endl;
        //递归建立二叉树
        pre_tree = preCreateTree();
        cout << "树的储存已经完成!" << endl << endl;
        cout << "===广义表的输出===" << endl;
    }
}

```



```

    printList(pre_tree);
    cout << withList << endl << endl;
    //开始树的遍历
    cout <<"===开始树的遍历操作==="<< endl << endl;
    listTree(pre_tree);
}
else if(n == 2)
{
    lev_tree = levCreateTree();
    cout << "树的储存已经完成! " << endl << endl;
    cout << "===广义表的输出===" << endl;
    printList(lev_tree);
    cout << withList << endl << endl;
    //开始树的遍历
    cout <<"===开始树的遍历操作==="<< endl << endl;
    listTree(lev_tree);
}
cout << "===树的线索化===" << endl;
toTreeList(pre_tree);
}

```