山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：201700130033	姓名： 武学伟	班级： 2017 级 2 班
实验题目：二叉树操作		
实验学时：4	实验日期： 2018.11.25	

实验目的：

1.掌握二叉树的基本概念，链表描述方法。

2.掌握二叉树操作的实现。

软件环境：

Win10home，sublime

1. 实验内容（题目内容，输入要求，输出要求）

1）二叉树的层次遍历、节点数目与高度；

求出一个输入的二叉树的层次遍历，求出每个节点为根的子树的节点数目以及高度。

输入描述：

第一行为一个数字 n，表示有这棵树有 n 个节点。 编号为 1-n

之后 n 行每行两个数字，第 i 行的两个数字 a、b 表示编号为 i 的节点的左孩子节点为 a，右孩子节点为 b，-1 表示该位置没有节点。

保证数据有效，根节点为 1。

输出描述：

第一行，n 个数字，表示该树的层次遍历

第二行，n 个数字，第 i 个数字表示以 i 节点为根的子树的节点数目

第三行，n 个数字，第 i 个数字表示以 i 节点为根的子树的高度

2）利用二叉树的前序遍历和中序遍历来求得后序遍历。

输入描述：

输入有三行

第一行为数字 n

第二行有 n 个数字，表示二叉树的前序遍历

输出描述：

输出一行，表示该二叉树的后序遍历序列

第三行有 n 个数字，表示二叉树的中序遍历

2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

数据结构：

二叉树、链表。

算法：

1） 建树

a） 因为无法在构造函数中直接建树，所以单独写一个函数，进行建树操作。将所有的节点值保存在一个数组中，数组的大小是传入的参数 n 的大小。

b） 对数组的元素进行赋值，因为根节点的值为 1，为了利用所有的空间，所以在赋值的时候注意加一。

c) 将数据成员 root 的值设定为数组的第 0 个元素。
d) 定义两个 int 型变量，表示节点的左孩子与右孩子的位置。
e) 遍历数组，根据左孩子与右孩子的位置，确定他们的值，注意数组的下标值应为左孩子与右孩子的位置减一。

2) 层次遍历

a) 建立一个链表描述的队列类，并提供判断是否为空，返回队首，返回队尾，出队，入队等操作。
b) 建立一个队列对象，用于存放节点。
c) 定义一个二叉树节点对象 t 用于遍历，t 的初始值为书的根节点。
d) 当 t 的值不为空的时候，输出 t 的值，然后将 t 的左孩子和右孩子分别入队，跟新 t 的值为队首元素，再出队一次，直至队列为空。

3) 计算高度

a) 先计算一个节点作为根的子树的高度。传入的参数是根节点 t
b) 在类中加入 hei[]数组用于存储节点的子树的高度。
c) 利用的递归的思想，定义两个变量 hl 和 hr，分别为节点 t 的左子树的高度和右子树的高度，取其中的最大值加一作为当前节点的高度，并返回这个结果。
d) 然后用另外一个函数对存储在数组中的每个节点调用这个函数，就得到了所有节点作为根的子树的高度值。

4) 计算节点数量

a) 先计算一个节点作为根的子树的节点数。传入的参数是根节点 t
b) 在类中加入 hei[]数组用于存储节点的子树的节点数。
c) 利用的递归的思想，定义两个变量 hl 和 hr，分别为节点 t 的左子树的节点数和右子树的节点数，取两数之和加一作为当前节点的节点数，并返回这个结果。
d) 然后用另外一个函数对存储在数组中的每个节点调用这个函数，就得到了所有节点作为根的子树的节点数。

5) 计算后序遍历

a) 实验二比较简单，且题目要求仅为输出后序遍历，所以不用建树
b) 利用递归的思想，函数的参数是前序数组，中序数组，以及它们的大小。
c) 新建一个指向前序数组首元素的指针，和一个整型变量作为索引值，然后遍历中序数组，找到与前序数组首元素相同的元素值，这就是根节点。
d) 改变前序中序数组以及大小，左子树的前序数组首元素加一，中序数组不变，大小为当前索引值；右子树的前序数组为首元素加索引值加一，中序为首元素加索引值加一，大小为原大小减索引值再减一。对左子树和右子树进行递归调用。
e) 然后输出当前节点值。这样就实现了后序遍历的输出。

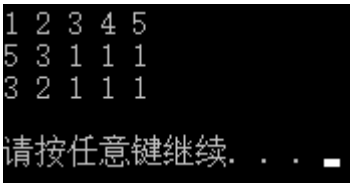
3. 测试结果（测试输入，测试输出，结果分析）

测试一（实验一）：

输入：

5
2 3

4 5
-1 -1
-1 -1
-1 -1
输出：
1 2 3 4 5
5 3 1 1 1
3 2 1 1 1



测试二（实验一）：
输入：
5
3 2
-1 -1
4 5
-1 -1
-1 -1
输出：
1 3 2 4 5
5 1 3 1 1
3 1 2 1 1

测试三（实验二）：
输入：
5
1 2 4 5 3
4 2 5 1 3
输出：
4 5 2 3 1

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

结果分析：

数据正确。

实验一的计算每个节点作为根节点的子树的高度和节点数时间复杂度为O(n)。因为这一步是在递归函数中完成的，对于每个节点的遍历的时间复杂

度为Θ(1)，所以总计为O(n)。

实验一在一开始写的时候复杂度较高，因为在计算每个节点作为根节点的子树的高度或者节点数的时候，采用了层次遍历，这样就相当于每个节点的复杂度就为$O(n^2)$，在最后一个测试数据中就超时了，显然这么高的复杂度不符合要求，所以将层次遍历所有节点改为了用数组存储所有的节点。这样就有效降低了复杂度。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
/*Ex9_1_up.cpp*/
#include <iostream>
#include <string>
using namespace std;

/*二叉树结点*/
template <class E>
struct binaryTreeNode
{
    E element;          //节点元素
    binaryTreeNode<E>* leftChild;   //左儿子
    binaryTreeNode<E>* rightChild;  //右儿子
    /*构造函数*/
    binaryTreeNode()
    {rightChild = leftChild = NULL;}
    binaryTreeNode(const E& theElement):element(theElement)
    {rightChild = leftChild = NULL;}
    binaryTreeNode(const E& theElement, binaryTreeNode<E>*
theLeftChild, binaryTreeNode<E>* theRightChild):element(theElement)
    {
        rightChild = theRightChild;
        leftChild = theLeftChild;
    }
};

/*链表节点*/
template <class T>
struct chainNode
{
    T element;
    chainNode<T>* next;

    chainNode(){}
    chainNode(const T& theElement)
    {
        element = theElement;
```

```
        next = NULL;
    }
    chainNode(const T& theElement, chainNode<T>* theNext)
    {
        element = theElement;
        next = theNext;
    }
};

/*链表型队列*/
template <class T>
class chainQueue
{
    public:
        chainQueue()
        {
            queueFront = NULL;
            queueSize = 0;
        }
        ~chainQueue()
        {
            while (queueFront != NULL)
            {
                chainNode<T>* nextNode = queueFront->next;
                delete queueFront;
                queueFront = nextNode;
            }
        }
        bool empty() const {return queueSize == 0;}
        int size() const {return queueSize;}
        T& front()
        {
            if (queueSize == 0)
            {
                string error = "Empty queue";
                throw error;
            }
            return queueFront->element;
        }
        T& back()
        {
            if (queueSize == 0)
            {
                string error = "Empty queue";
```

```
            throw error;
        }
        return queueBack->element;
    }
    void pop()
    {
        if (queueSize == 0)
        {
            string error = "Empty queue";
            throw error;
        }
        chainNode<T>* nextNode = queueFront->next;
        delete queueFront;
        queueFront = nextNode;
        queueSize--;
    }
    void push(const T& theElement)
    {
        chainNode<T>* newNode = new chainNode<T>(theElement);
        if (queueSize == 0)
            queueFront = newNode;
        else
            queueBack->next = newNode;
        queueBack = newNode;
        queueSize++;
    }
private:
    chainNode<T>* queueFront;
    chainNode<T>* queueBack;
    int queueSize;
};

/*二叉树*/
template <class E>
class linkedBinaryTree
{
public:
    linkedBinaryTree()
    {
        root = NULL;
        treeSize = 0;
    }
    ~linkedBinaryTree(){;}
    int size() const {return treeSize;}
```

```
        void makeTree(int n);          //建树
        void leverOrder();             //层次遍历
        void pheight();                //输出节点作为根节点的子树的高度
        void pcount();                 //输出节点作为根节点的子树的节点
数
    protected:
        binaryTreeNode<E>* root;    //二叉树的根节点
        int treeSize;              //树的节点的个数
        int* hei;                  //记录每个节点作为根节点的子树的高度
        int* cou;                  //记录每个节点作为根节点的子树的节点
个数
        int height(binaryTreeNode<E>*);//计算某个节点作为根节点的子
树的高度
        int count(binaryTreeNode<E>*); //计算某个节点作为根节点的子
树的节点个数
};

/*建树过程*/
template <class E>
void linkedBinaryTree<E>::makeTree(int n)
{
    treeSize = n;
    hei = new int[treeSize];     //定义 hei 数组
    cou = new int[treeSize];     //定义 cou 数组
    for(int i=0; i<treeSize; i++)
    {
        hei[i] = 0;
        cou[i] = 0;
    }
    int theLeftChild;              //定义左孩子
    int theRightChild;                 //定义右孩子
    binaryTreeNode<E>* childs[n];     //创建一个树结点数组保存树中的
其他节点
    for (int i=0; i<n; i++)     //为每个节点赋值
        childs[i] = new binaryTreeNode<E>(i+1);          //节点元素
的赋值，为下标值加一
    root = childs[0];
    for (int i=0; i<n; i++)
    {
        cin >> theLeftChild;
        cin >> theRightChild;
        if (theLeftChild != -1)
            childs[i]->leftChild = childs[theLeftChild-1];  //左孩子
节点的建立
```

```
        else
            childs[i]->leftChild = NULL;
        if (theRightChild != -1)
            childs[i]->rightChild = childs[theRightChild-1];//右孩子
节点的建立
        else
            childs[i]->rightChild = NULL;
    }

}

/*层次遍历*/
template <class E>
void linkedBinaryTree<E>::leverOrder()
{
    chainQueue<binaryTreeNode<E>*> q;  //记录节点
    binaryTreeNode<E>* t = root;
    while(t != NULL)
    {
        cout << t->element << " ";
        if (t->leftChild != NULL)       //入队左孩子
            q.push(t->leftChild);
        if (t->rightChild != NULL)      //入队右孩子
            q.push(t->rightChild);
        try {t = q.front();}
        catch (string error) {break;}
        q.pop();
    }
    cout << endl;
}

/*计算高度*/
template <class E>
int linkedBinaryTree<E>::height(binaryTreeNode<E>* t)
{
    if (t == NULL)
        return 0;
    int hl = height(t->leftChild); //左子树高度
    int hr = height(t->rightChild);//右子树高度
    if (hl>hr)                     //取其中较大值
    {
        hei[t->element-1] = hl+1;
        return ++hl;
    }
```

```
    else
    {
        hei[t->element-1] = hr+1;
        return ++hr;
    }
}

template <class E>
void linkedBinaryTree<E>::pheight()
{
    height(root);
    for (int i=0; i<treeSize; i++)
        cout << hei[i] << " ";
    cout << endl;
}

/*计算节点数*/
template <class E>
int linkedBinaryTree<E>::count(binaryTreeNode<E>* t)
{
    if (t == NULL)
        return 0;
    int cl = count(t->leftChild);       //左子树节点数
    int cr = count(t->rightChild);      //右子树节点数
    cou[t->element-1] = cl + cr + 1;    //左子树节点数加右子树节点数
加一
    return cl + cr + 1;
}

template <class E>
void linkedBinaryTree<E>::pcount()
{
    count(root);
    for (int i=0; i<treeSize; i++)
        cout << cou[i] << " ";
    cout << endl;
}

int main()
{
    int n;     //二叉树节点数量
    cin >> n;
    linkedBinaryTree<int> b;
    b.makeTree(n);
```

```
    b.leverOrder();
    b.pcount();
    b.pheight();
    cout << endl;
    return 0;
}
/*Ex9_2.cpp*/
#include <iostream>
using namespace std;

template <class E>
struct binaryTreeNode      //二叉树节点
{
    E element;        //节点元素
    binaryTreeNode<E>* leftChild;   //左儿子
    binaryTreeNode<E>* rightChild;  //右儿子
    /*构造函数*/
    binaryTreeNode()
    {
        rightChild = leftChild = NULL;
    }
    binaryTreeNode(const E& theElement):element(theElement)
    {
        rightChild = leftChild = NULL;
    }
    binaryTreeNode(const E& theElement, binaryTreeNode<E>*
theLeftChild, binaryTreeNode<E>* theRightChild):element(theElement)
    {
        rightChild = theRightChild;
        leftChild = theLeftChild;
    }
};

template <class E>
class linkedBinaryTree
{
    public:
        linkedBinaryTree()
        {
            root = NULL;
            treeSize = 0;
        }
        ~linkedBinaryTree(){;}
        void transPost(E* pre, E* in, int theSize);
```

```
    private:
        binaryTreeNode<E>* root;    //二叉树的根节点
        int treeSize;               //数的节点的个数
};

template <class E>
void linkedBinaryTree<E>::transPost (E* pre, E* in, int theSize)
{
    if (theSize == 0)    //若长度为零，直接返回空指针
    {
        return;
    }
    binaryTreeNode<E>* newNode = new binaryTreeNode<E>;
    newNode->element = *pre;     //定位到前序的根节点
    int index = 0;
    for (; index<theSize; index++)
    {
        if (in[index] == *pre)     //前序遍历的第一个元素肯定是根节点，定位其在中序遍历中的位置
        break;
    }
    transPost(pre+1,in,index);  //左子树肯定在中序根节点的左边
    transPost(pre+index+1,in+index+1,theSize-(index+1));  //右子树肯定在中序根节点的右边
    cout << newNode->element << " ";
    delete newNode;
}



int main()
{
    int n;     //二叉树节点数量
    cin >> n;
    int pre[n];    //存储前序遍历的数组
    int in[n]; //存储中序遍历的数组
    for(int i=0; i<n; i++)
        cin >> pre[i];
    for(int i=0; i<n; i++)
        cin >> in[i];
    linkedBinaryTree<int> test;
    test.transPost(pre,in,n);
    return 0;
}
```

附：
复杂度较高的错误写法：

```
/*计算高度*/
template <class E>
int linkedBinaryTree<E>::height(binaryTreeNode<E>* t)
{
    if (t == NULL)
        return 0;
    int hl = height(t->leftChild);
    int hr = height(t->rightChild);
    if (hl>hr)
        return ++hl;
    else
        return ++hr;
}

template <class E>
void linkedBinaryTree<E>::pheight()
{
    chainQueue<binaryTreeNode<E>*> q;
    binaryTreeNode<E>* t = root;
    while(t != NULL)
    {
        cout << height(t) << " ";
        if (t->leftChild != NULL)
            q.push(t->leftChild);
        if (t->rightChild != NULL)
            q.push(t->rightChild);
        try {t = q.front();}
        catch (string error) {break;}
        q.pop();
    }
    cout << endl;
}
/*计算节点数*/
template <class E>
int linkedBinaryTree<E>::count(binaryTreeNode<E>* t)
{
    if (t == NULL)
        return 0;
    int cl = count(t->leftChild);
    int cr = count(t->rightChild);
    return cl + cr + 1;
}
```

```
template <class E>
void linkedBinaryTree<E>::pcount()
{
    chainQueue<binaryTreeNode<E>*> q;
    binaryTreeNode<E>* t = root;
    while(t != NULL)
    {
        cout << count(t) << " ";
        if (t->leftChild != NULL)
            q.push(t->leftChild);
        if (t->rightChild != NULL)
            q.push(t->rightChild);
        try {t = q.front();}
        catch (string error) {break;}
        q.pop();
    }
    cout << endl;
}
```