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　　：202200400053	姓名： 王宇涵	班级： 22 级 2 班
实验题目：二叉树		
实验学时：2	实验日期：　　2023-11-08	

实验目的：

掌握二叉树的基本概念，链表描述方法；二叉树操作的实现。

软件开发环境：

VSCODE

1. 实验内容

1、题目描述：

创建二叉树类（采用链表存储），提供操作：前序遍历、中序遍历、后序遍历、层次遍历、计算二叉树结点数目、计算二叉树高度，其中前序遍历要求以递归方式实现，中序遍历、后序遍历要求以非递归方式实现。

输入输出格式：

输入：第一行输入一个数字 n（10≤n≤100000）表示二叉树的节点个数，

节点编号为 1～n。接下来 n 行，每行输入两个数，第 i 行的两个数 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，a 或 b 为－1 时

表示该位置没有节点。输入数据保证有效，根节点的编号为 1。

输出：输出六行，每行 n 个数。

第一行输出 n 个数，表示该树的前序遍历；

第二行输出 n 个数，表示该树的中序遍历；

第三行输出 n 个数，表示该树的后序遍历；

第四行输出 n 个数，表示该树的层次遍历；

第五行输出 n 个数，其中第 i 个数表示以节点 i 为根的子树的节点数目；

第六行输出 n 个数，其中第 i 个数表示以节点 i 为根的子树的高度。

2、题目描述：

根据二叉树的前序序列和中序序列（树中的元素各不相同），输出该二叉
树的后序序列。
输入输出格式：

输入：第一行输入一个数字 n 表示二叉树的节点个数；第二行输入 n 个

数表示二叉树的前序遍历；第三行输入 n 个数表示二叉树的中序遍历。

输出：在一行中输出 n 个数，表示该二叉树的后序序列。

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）

题目一

BinaryTreeNode 结构体

这是二叉树节点的定义。它包含以下成员：

element：存储节点的元素值。

leftChild：指向左子节点的指针。

rightChild：指向右子节点的指针。

LinkedBinaryTree 类

这是基于链表的二叉树类，继承自 BinaryTree 抽象类。

它包括了二叉树的各种操作，包括前序遍历、中序遍历、后序遍历、层次遍历等。此外，还包含了构建二叉树、判断是否为空、获取大小、获取高度等操作。

preOrder、inOrder、postOrder、leverOrder 方法：分别实现了前序、中序、后序、层次遍历二叉树的递归算法。

makeTree 方法：用于将两颗二叉树和一个元素合并成一颗新的二叉树。

output 方法：用于输出节点的元素值。

dispose 方法：用于销毁节点。

height 方法：用于计算树的高度。

中序遍历和后序遍历的非递归实现

在 inOrder 函数中，我们使用了一个 stack 数据结构来模拟递归遍历，确保我们按中序遍历的顺序访问节点。

首先，我们检查根节点是否为空，如果为空，直接返回，否则进行遍历。

在遍历的过程，我们首先沿着左子树一路向下，将遇到的节点入栈，直到到达最左边的叶子节点。这确保了我们首先访问左子树。

接着，我们弹出栈顶元素，访问它（根节点），然后切换到右子树，继续相同的过程。这确保了我们在左子树遍历完成后，访问根节点，然后再遍历右子树。

postOrder 函数同样使用了一个 stack 数据结构来模拟递归遍历，确保我们按后序遍历的顺序访问节点。

首先，我们检查根节点是否为空，如果为空，直接返回，否则进行遍历。

在遍历的过程中，首先从根节点开始，沿着左子树一路向下，将遇到的节点入栈，直到到达最左边的叶子节点。这一部分和中序遍历类似。

然后，我们在栈中弹出栈顶元素，如果它的右子树为空或者右子树已经被访问过（通过 pre 变量来判断），则访问它，否则，将它的右子树入栈，并继续向下遍历左子树。

这样，确保了在遍历完左子树和右子树后，访问根节点，实现了后序遍历的效果。

如何生成一个树

巧妙利用 makeTree 函数从编号最大的元素开始逐步从下到上构建,保证被置空的元素不会再被使用,默认元素的编号大小顺序是从上到下,从左到右的.

题目二

preOrder、inOrder、pos 数组

这些数组用于存储前序遍历、中序遍历和中序遍历中每个元素的位置。这些数组在构建二叉树时用于确定根节点和子树的边界。

buildTree 函数

这是主要的构建二叉树的函数。它使用前序遍历和中序遍历的信息来递归构建二叉树。具体步骤如下：

如果前序遍历的范围 pl 到 pr 或中序遍历的范围 il 到 ir 是空的，返回空指针表示空树。

否则，创建一个根节点，根节点的值为前序遍历的第一个元素 preOrder[pl]。

在中序遍历中找到根节点的位置 k，即 pos[preOrder[pl]] - il，其中 k 表示根节点在中序遍历中的位置，从而可以确定左子树和右子树的范围。

递归构建左子树，范围为 pl + 1 到 pl + k（前序遍历范围），il 到 il + k - 1（中序遍历范围）。

递归构建右子树，范围为 pl + k + 1 到 pr（前序遍历范围），il + k + 1 到 ir（中序遍历范围）。

返回根节点。

postOrderOutput

这是用于后序遍历输出二叉树的递归实现函数.

3. 测试结果（测试输入，测试输出）

题目一

输入

5

2 3

4 5

-1 -1

-1 -1

-1 -1

输出

1 2 4 5 3

4 2 5 1 3

4 5 2 3 1

1 2 3 4 5

5 3 1 1 1

3 2 1 1 1

题目二

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）

实验一

三个难点:中序遍历的非递归实现,后序遍历的非递归实现, 树如何进行构建

解决方法

非递归实现:通过栈来模拟递归的过程,在不同的位置进行元素的 visit 操作,中序遍历在遍历完左子树,取出栈顶元素时进行输出,后序遍历则需要满足右子树为空或右子树刚被访问过才能进行输出.

树的构建:使用 maketree 函数,初始化 root 的左右孩子为两个树的根结点,此时需要将根结点置空,否则出现 bug.再从下到上构建 maketree 函数即可

实验二

难点在于:如何模拟构建树的过程

解决方法

先通过前序遍历序列找到根节点,再通过中序遍历序列找到左右子树的范围,构建 root 结点,root 的左孩子和右孩子分别再进行递归,递归结束的条件为递归的左边界大于右边界即可.,最后返回 root 结点即可

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）

实验一

```
#pragma once

#pragma once

#include<iostream>

using namespace std;

template<class T>

struct BinaryTreeNode

{

    T element;

    BinaryTreeNode<T>* leftChild;

    BinaryTreeNode<T>* rightChild;

    BinaryTreeNode()

    {

        leftChild=rightChild=NULL;

    }
```

```
        BinaryTreeNode(const T& theElement)
        {
                element=theElement;
                leftChild=rightChild=NULL;
        }
        BinaryTreeNode(const T& theElement,BinaryTreeNode * theLeftChild,BinaryTreeNode* theRightChild)
        {
                element=theElement;
                leftChild=theLeftChild;
                rightChild=theRightChild;
        }
};
#pragma once
#include<iostream>
using namespace std;
template<class T>
class BinaryTree
{
        public:
                virtual ~BinaryTree(){}
                virtual bool empty()const =0;
                virtual int size()const =0;
                virtual void preOrder(void (*) (T*))=0;
                virtual void inOrder(void(*)(T*))=0;
                virtual void postOrder(void(*)(T*))=0;
                virtual void leverOrder(void(*)(T*))=0;
};
#pragma once
#include<iostream>
using namespace std;

template<class T>
```

```
class Queue
{
public:
	virtual ~Queue(){}
	virtual bool empty()const =0;
	virtual int size()const =0;
	virtual T front()=0;
	virtual T back()=0;
	virtual void pop()=0;
	virtual void push(const T& theElement)=0;
};



template<class T>
class ArrayQueue :public Queue<T>
{
public:
	ArrayQueue(int theCapacity=10);
	~ArrayQueue(){delete[] queue;}
	bool empty() const{return queueBack==queueFront;}
	bool full() const {return (queueBack+1)%arrayLength==queueFront;}
	int size()const {return (queueBack-queueFront+arrayLength)%arrayLength;}
	T front()
	{
		if(empty())
		{
			cout<<"empty"<<endl;
			return 0;
		}
		return queue[(queueFront+1)%arrayLength];
	}
```

```
        T back()
        {
            if(empty())
            {
                cout<<"empty"<<endl;
                return 0;
            }
            return queue[queueBack];
        }
        void pop()
        {
            if(empty())
            {
                cout<<"empty"<<endl;
                return;
            }
            queueFront=(queueFront+1)%arrayLength;
            queue[queueFront].~T();
        }
        void push(const T& theElement);
        void changeQueueLength();

    private:
        int queueFront;
        int queueBack;
        int arrayLength;
        T *queue;
};

template <class T>
inline ArrayQueue<T>::ArrayQueue(int theCapacity)
{
```

```
	if(theCapacity<1)
	{
		cout<<"capacity must >0"<<endl;
		return;
	}
	arrayLength=theCapacity;
	queue=new T[theCapacity];
	queueFront=queueBack=0;
}

template <class T>
inline void ArrayQueue<T>::push(const T &theElement)
{
	//扩容
	if(size()==arrayLength-1)
	{
		changeQueueLength();
	}
	queueBack=(queueBack+1)%arrayLength;
	queue[queueBack]=theElement;
}

template <class T>
inline void ArrayQueue<T>::changeQueueLength()
{
	T* newQueue=new T[2*arrayLength];

	int start=(queueFront+1)%arrayLength;
	//未形成环形
	if(start<2)
		copy(queue+start,queue+start+arrayLength-1,newQueue);
	//形成环形
```

```
        else
        {
            copy(queue+start,queue+arrayLength,newQueue);
            copy(queue,queue+queueBack+1,newQueue+arrayLength-start);
        }

        queueFront=2*arrayLength-1;
        queueBack=arrayLength-2;
        arrayLength=arrayLength*2;
        delete[]queue;
        queue=newQueue;
}

#include<stack>

template<class E>
class LinkedBinaryTree:public BinaryTree<BinaryTreeNode<E>>
{
    public:
        LinkedBinaryTree()
        {
            root=NULL;
            treeSize=0;
            visit=output;
        }
        ~LinkedBinaryTree()
        {
            erase();
        }
        void createTreeByLevel();
        bool empty() const
        {
```

```
        return treeSize==0;
    }
    int size()const
    {
        return treeSize;
    }
    //遍历函数,传入 visit 函数指针
    void preOrder(void (*theVisit) (BinaryTreeNode<E>*))
    {
        visit=theVisit;preOrder(root);
    }
    void inOrder(void (*theVisit) (BinaryTreeNode<E>*))
    {
        visit=theVisit;inOrder(root);
    }
    void postOrder(void (*theVisit) (BinaryTreeNode<E>*))
    {
        visit=theVisit;postOrder(root);
    }
    void leverOrder(void (*theVisit) (BinaryTreeNode<E>*))
    {
        visit=theVisit;leverOrder(root);
    }
    void preOrderOutput()
    {
        preOrder(output);
        cout<<endl;
    }
    void inOrderOutput()
    {
        inOrder(output);
        cout<<endl;
```

```
        }
        void postOrderOutput()
        {
            postOrder(output);
            cout<<endl;
        }
        void leverOrderOutput()
        {
            leverOrder(output);
            cout<<endl;
        }
        void erase()
        {
            postOrder(dispose);
            root=NULL;
            treeSize=0;
        }
        int height() const
        {
            return height(root);
        }

        void makeTree(const E& theElement,LinkedBinaryTree<E>&a,LinkedBinaryTree<E>&b);

    private:
        BinaryTreeNode<E>* root;
        int treeSize;
        //定义函数指针,规定如何访问元素
        static void (*visit)(BinaryTreeNode<E>*);
        static void preOrder(BinaryTreeNode<E>*t);
        static void inOrder(BinaryTreeNode<E>*t);
        static void postOrder(BinaryTreeNode<E>*t);
```

```
        static void leverOrder(BinaryTreeNode<E>*t);
        static void output(BinaryTreeNode<E>*t)
        {
            cout<<t->element<<" ";
        }
        static void dispose(BinaryTreeNode<E>*t)
        {
            delete t;
        }
        static int 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>::visit)(BinaryTreeNode<E>*);


template<class E>
BinaryTreeNode<E>* root;


template <class E>
inline void LinkedBinaryTree<E>::createTreeByLevel()
```

```
{
    int n;cin>>n;
    int a[n];
    for(int i=0;i<n;i++)
        cin>>a[i];

    ArrayQueue<BinaryTreeNode<E>*>q;

    int index=0;
    root=new BinaryTreeNode<E> (a[index++]);
    q.push(root);
    BinaryTreeNode<E>* p=NULL;
    while(!q.empty()&&index<n)
    {
        p=q.front();
        q.pop();
        //如果不空就创建一个节点
        BinaryTreeNode<E>* leftNode=new BinaryTreeNode<E>(a[index++]);
        p->leftChild=leftNode;
        q.push(leftNode);
        treeSize++;

        if(index<n)
        {
            BinaryTreeNode<E>* rightNode=new BinaryTreeNode<E>(a[index++]);
            p->rightChild=rightNode;
            q.push(rightNode);
            treeSize++;
        }
    }
}
//根左右
```

```
template <class E>
inline void LinkedBinaryTree<E>::preOrder(BinaryTreeNode<E> *t)
{    //非递归写法
     // stack<BinaryTreeNode<E>*>stk;
     // if(t==NULL)
     //      return;

     // //只有当栈为空且 t 为空的时候才停止
     // while(!stk.empty()||t)
     // {    //遍历左子树
     //      while(t)
     //      {
     //           stk.push(t);
     //           LinkedBinaryTree::visit(t);
     //           t=t->leftChild;
     //      }
     //      t=stk.top();
     //      stk.pop();
     //      t=t->rightChild;
     // }
     //递归写法
     if(t!=NULL)
     {
          LinkedBinaryTree::visit(t);
          preOrder(t->leftChild);
          preOrder(t->rightChild);
     }
}
//左根右
template <class E>
inline void LinkedBinaryTree<E>::inOrder(BinaryTreeNode<E>*t)
{
```

```
    stack<BinaryTreeNode<E>*>stk;
    if(t==NULL)
        return;

    //只有当栈为空且 t 为空的时候才停止
    while(!stk.empty()||t)
    {   //遍历左子树
        while(t)
        {
            stk.push(t);
            t=t->leftChild;
        }
        t=stk.top();
        stk.pop();
        LinkedBinaryTree::visit(t);
        t=t->rightChild;
    }
}

template <class E>
inline void LinkedBinaryTree<E>::postOrder(BinaryTreeNode<E>*t)
{
    stack<BinaryTreeNode<E>*>stk;
    if(t==NULL)
        return;

    BinaryTreeNode<E>* pre=NULL;
    BinaryTreeNode<E>* cur=t;
    while(cur)
    {
        stk.push(cur);
        cur=cur->leftChild;
```

```
        }
        //对于根结点的每个左孩子而言
        while(!stk.empty())
        {
            cur=stk.top();
            //如果右孩子为空,或者刚才被访问过
            if(cur->rightChild==NULL||cur->rightChild==pre)
            {
                LinkedBinaryTree::visit(cur);
                pre=cur;
                stk.pop();
            }
            else
            {
                cur=cur->rightChild;
                while(cur)
                {
                    stk.push(cur);
                    cur=cur->leftChild;
                }
            }
        }
}

template <class E>
inline void LinkedBinaryTree<E>::leverOrder(BinaryTreeNode<E>*t)
{
        ArrayQueue<BinaryTreeNode<E>*> q;
        while(t!=NULL)
        {
            visit(t);
```

```
            if(t->leftChild!=NULL)
                q.push(t->leftChild);
            if(t->rightChild!=NULL)
                q.push(t->rightChild);

            if(!q.empty())
            {
                t=q.front();
                q.pop();
            }
            else
                return;
        }
}

template <class E>
inline void LinkedBinaryTree<E>::makeTree(const E &theElement, LinkedBinaryTree<E> &a, LinkedBinaryTree<E> &b)
{
    BinaryTreeNode<E>* tmp=b.root;
    root=new BinaryTreeNode<E> (theElement,a.root,tmp);
    treeSize=a.treeSize+b.treeSize+1;

    a.root=b.root=NULL;
    a.treeSize=b.treeSize=0;
}

struct lr
{
    int left,right;
};

int n;
```

```
int main()
{
    cin>>n;
    LinkedBinaryTree<int>tree[n+1];
    LinkedBinaryTree<int> a;
    int height[n+1];
    int size[n+1];
    struct lr st[n+1];

    //先存下 left 和 right
    for(int i=1;i<=n;i++)
    {
        int left,right;
        cin>>left>>right;
        st[i].left=left;
        st[i].right=right;
    }
    //对于每一颗树,进行合并
    for(int i=n;i>=1;i--)
    {
        if(st[i].left!=-1&&st[i].right!=-1)
        {
            tree[i].makeTree(i,tree[st[i].left],tree[st[i].right]);
        }
        else if(st[i].left==-1&&st[i].right!=-1)
        {
            tree[i].makeTree(i,a,tree[st[i].right]);
        }
        else if(st[i].left!=-1&&st[i].right==-1)
        {
            tree[i].makeTree(i,tree[st[i].left],a);
        }
```

```
            else
            {
                tree[i].makeTree(i,a,a);
            }
            height[i]=tree[i].height();
            size[i]=tree[i].size();
        }

        //输出结果
        tree[1].preOrderOutput();
        tree[1].inOrderOutput();
        tree[1].postOrderOutput();
        tree[1].leverOrderOutput();
        for(int i=1;i<=n;i++)
        {
            cout<<size[i]<<" ";
        }
        cout<<endl;
        for(int i=1;i<=n;i++)
        {
            cout<<height[i]<<" ";
        }
}
```

实验二

```
#include<iostream>
#pragma once
#include<iostream>
using namespace std;
template<class T>
struct BinaryTreeNode
```

```
{
    T element;
    BinaryTreeNode<T>* leftChild;
    BinaryTreeNode<T>* rightChild;
    BinaryTreeNode()
    {
        leftChild=rightChild=NULL;
    }
    BinaryTreeNode(const T& theElement)
    {
        element=theElement;
        leftChild=rightChild=NULL;
    }
    BinaryTreeNode(const T& theElement,BinaryTreeNode * theLeftChild,BinaryTreeNode* theRightChild)
    {
        element=theElement;
        leftChild=theLeftChild;
        rightChild=theRightChild;
    }
};
using namespace std;
const int N=100010;
int preOrder[N];
int inOrder[N];
int pos[N];

BinaryTreeNode<int>* buildTree(int pl,int pr,int il,int ir)
{
    if(pl>pr||il>ir)
        return NULL;
    BinaryTreeNode<int>* root=new BinaryTreeNode<int>(preOrder[pl]);
    //找出划分的区域
```

```
        int k=pos[preOrder[pl]]-il;
        root->leftChild=buildTree(pl+1,pl+k,il,il+k-1);
        root->rightChild=buildTree(pl+k+1,pr,il+k+1,ir);
        return root;
}

void postOrderOutput(BinaryTreeNode<int>*t)
{
        if(t!=NULL)
        {
                postOrderOutput(t->leftChild);
                postOrderOutput(t->rightChild);
                cout<<t->element<<" ";
        }
}

int n;
int main()
{
        cin>>n;
        for(int i=0;i<n;i++)
        {
                cin>>preOrder[i];
        }
        for(int i=0;i<n;i++)
        {
                cin>>inOrder[i];
        }
        for(int i=0;i<n;i++)
        {
                pos[inOrder[i]]=i;
        }
```

```
    BinaryTreeNode<int>* root=buildTree(0,n-1,0,n-1);

    postOrderOutput(root);
}
```