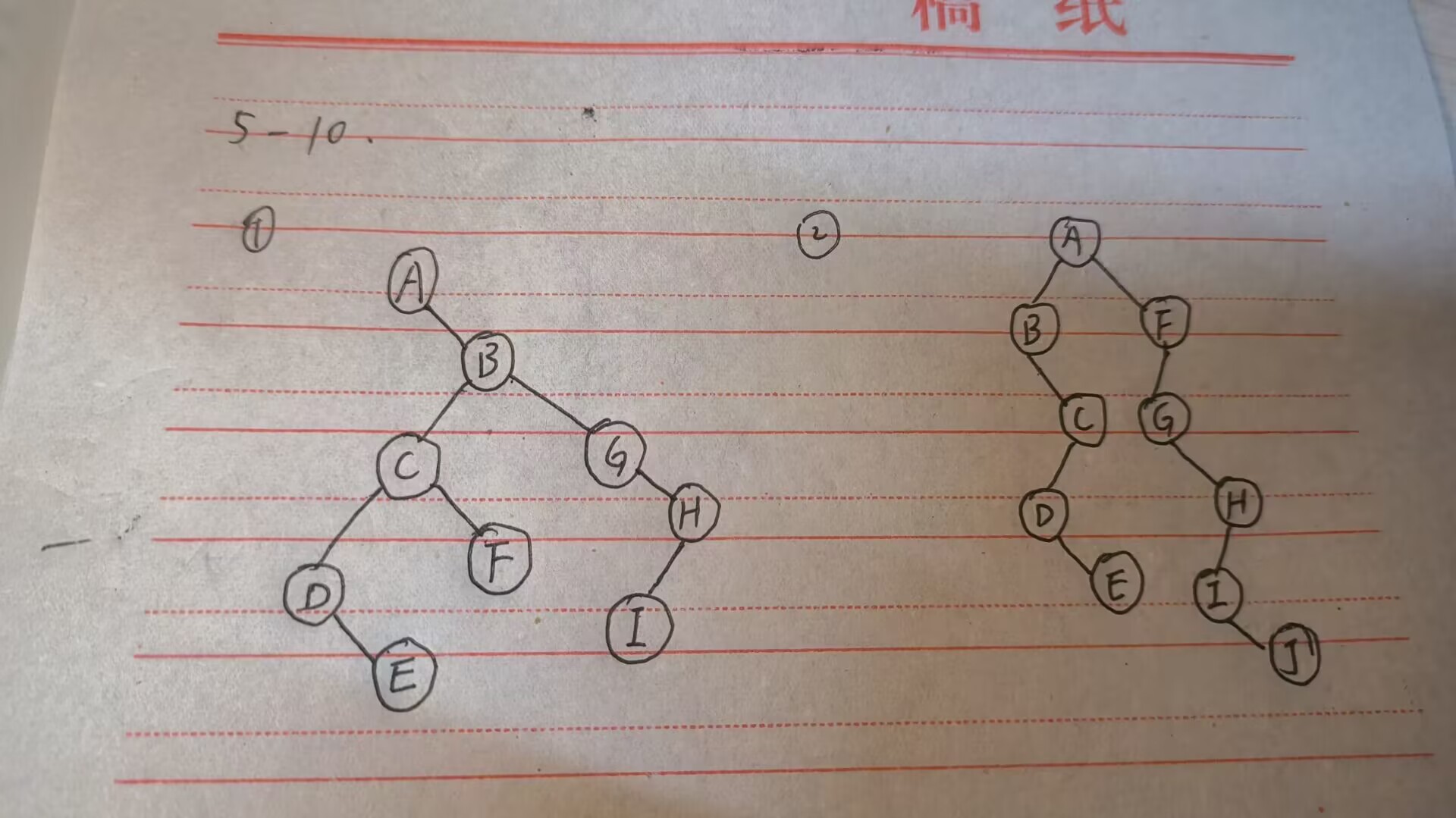
第6次作业

## 第5章习题 P158

**5.10证明：由二叉树的先序序列和中序序列能唯一确定一棵二叉树，并分别由下面的两个序列构造出相应的二叉树：**

①先序：ABCDEFGHI ②先序：ABCDEFGHIJ

中序：ADECFBGIH 中序：BDECAGIJHF

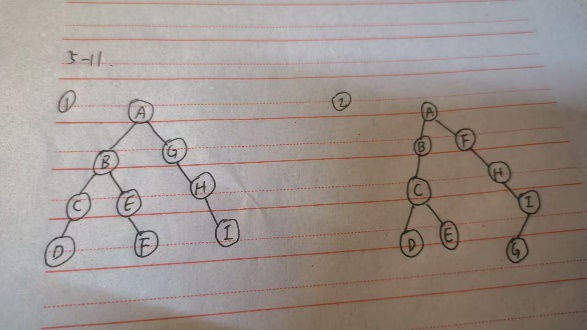


**5.11证明：由二叉树的后序序列和中序序列能唯一确定一棵二叉树，并分别由下面的两个序列构造出相应的二叉树：**

①后序：DCFEBIHGA ②后序：DECBGIHFA

中序：DCBFEAGHI 中序：DCEBAFHGI

后序序列中最后结点即为根结点。由根结点，在中序序列中就可以区分出左子树和右子 树结点。对左右子树做相同的处理，最终即可重建二叉树。



**5.12已知一棵二叉树的先序、中序和后序序列如下，其中各有一部分未给出其值，请构造出该二叉树。**

先序：A\_CDEF\_H\_J

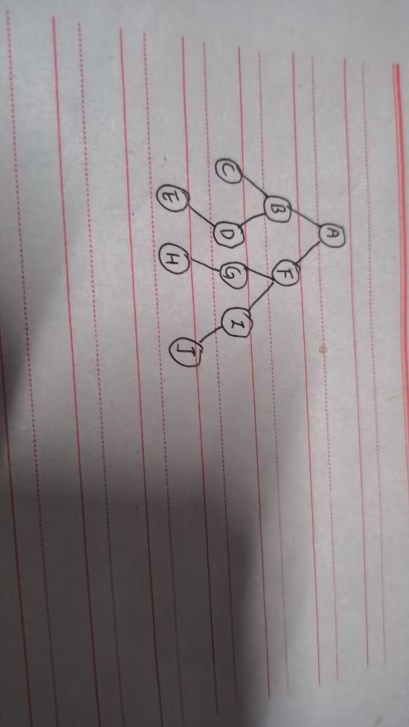
中序：C\_EDA\_GFI\_

后序：C\_ \_BHGJI\_ \_

解】 先序：A B C D E F G H I J

中序：C B E D A H G F I J

后序：C E D B H G J I F A



**5.16设计算法以输出每个结点到根结点之间的路径上的所有结点的值。**

【先序遍历输出路径算法描述】

void prePath( btNode \*T, seqStack S )

{

if(T)

{

push(S,T);

//遇到结点就入栈，栈中保存的即当前结点到根结点的路径

//输出结点路径

for( int i=S.top; i>=0; i-- )

cout<data<<" "; //输出栈中结点值，即路径

cout<lChild, S );

prePath( T->lChild, S ); //先序遍历左子树

prePath( T->rChild, S ); //先序遍历右子树

}

}

**5.18设计算法将一棵以顺序存储方式存储在数组A[]中的二叉树（已转换为完全二叉树，补充的虚拟结点为值为’#’）转换为二叉链表存储形式。**

算法思想：改造二叉树的一种遍历算法完成，这里改造先序遍历实现。算法包括两个整 型参数 i 和 num。i 为结点在完全二叉树上的编号，从 1 开始；num 为二叉树上最后有效结 点编号。

【算法描述】

void seq2BiTree(btNode \*&T,elementType A[],int i,int num)

{

//i 为当前结点编号，从 1 开始

//num 为最后有效结点编号

if(i<=num && A[i]) {

T=new btNode;

T->data=A[i];

T->lChild=NULL;

T->rChild=NULL;

seq2BiTree(T->lChild,A,2\*i,num);

seq2BiTree(T->rChild,A,2\*i+1,num);

}

}

**5.19分别设计出先序、中序和后序遍历二叉树的非递归算法。**

**//非递归：先序遍历**

void PreTraverseNR(BiNode\* pBT) {

BiNode\* p;

seqStack S;

initStack(S); //初始化栈

p=pBT;

while(p || !stackEmpty(S))

{

if(p)

{

cout<data<<", "; //访问根节点

pushStack(S, p); //p 指针入栈

p=p->lChild; //遍历左子树

}

else

{

popStack(S, p); //p 为空时，将上一层的根节点指针弹出

p=p->rChild; //遍历右子树

}

}

}

**//非递归：中序遍历**

void InTraverseNR(BiNode\* pBT)

{

BiNode\* p;

seqStack S;

initStack(S); //初始化栈

p=pBT;

while(p || !stackEmpty(S)) {

if(p) { pushStack(S,p); //根节点先行入栈，以便左子树遍历结束，返回访问根点

p=p->lChild; //遍历左子树

}

else //p 为空--访问根节点、遍历右子树

{

popStack(S, p); //某子树根节点出栈

cout<data<<", "; //访问某子树根节点

p=p->rChild; //遍历 p 的右子树

}

}

}

**//非递归：后序遍历**

void PostTraverseNR(BiNode\* pBT)

{

BiNode\* p;

seqStack S;

int tag[MaxLen]; //标记左子树、右子树

int n;

initStack(S); //初始化栈

p=pBT;

while(p || !stackEmpty(S))

{

if(p)

{

pushStack(S,p);

tag[S.top]=0; //标记遍历左子树

p=p->lChild; //循环遍历左子树

}

else //p==NULL 但是栈不空

{

stackTop(S,p); //取栈顶，但不退栈，以便遍历 p 的右子树

if(tag[S.top]==0) //说明 p 的左子树已经遍历，右子树尚未遍历

{

tag[S.top]=1; //设置当前结点遍历右子树标记

p=p->rChild; //遍历右子树

}

else //tag[S.top]==1, 说明 p 的左右子树皆已经遍历

{

popStack(S,p); //退栈

cout<data<<", "; //访问某子树根节点

p=NULL; //上面出栈的 p 已经没用，回去循环取栈顶的下一个元素

}

}

}

}