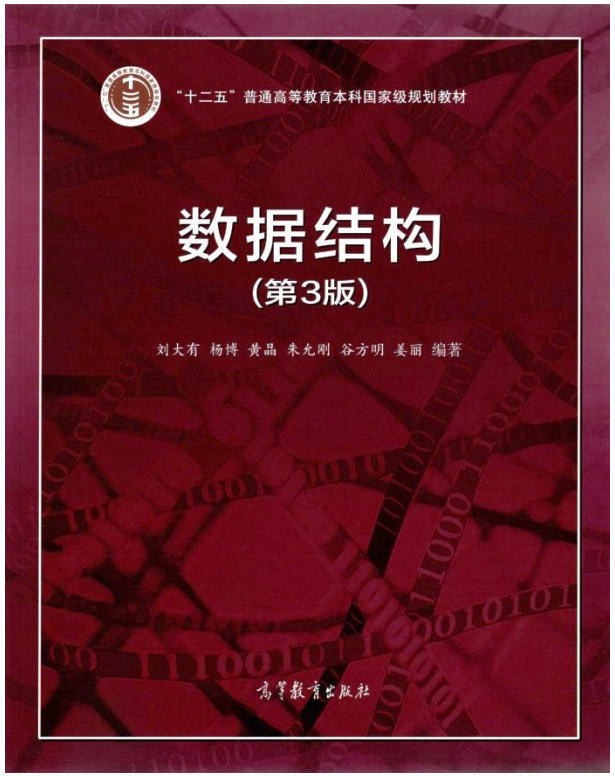
计算机学院王湘浩班

2024级



数据之法

结构之美

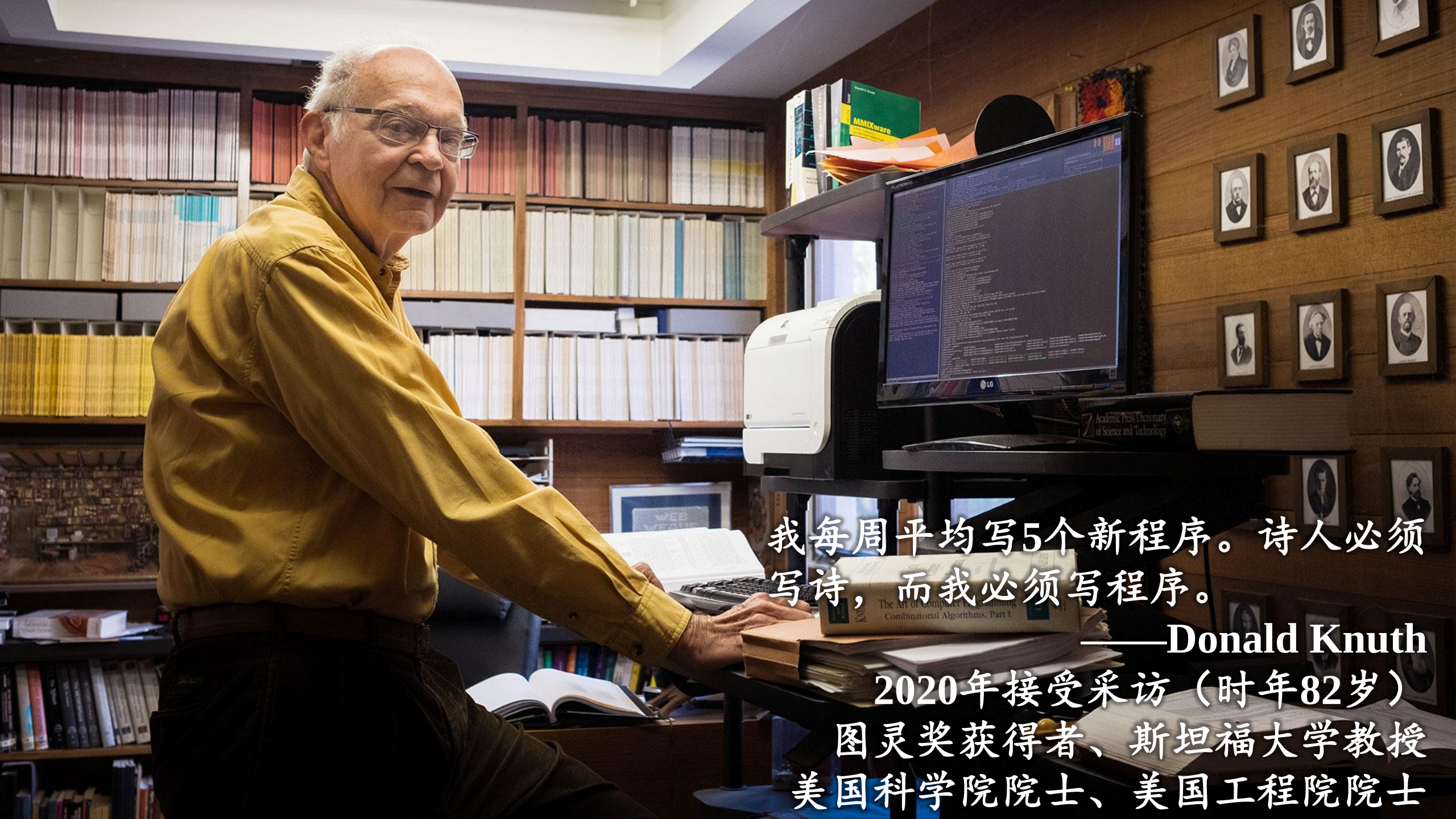
算法之道

二叉树的存储和操作

- 二叉树的存储结构
- 二叉树的遍历
- 二叉树的其他基本操作
- 二叉树的序列化/反序列化

zhuyungang@jlu.edu.cn

Last updated on 2025.4



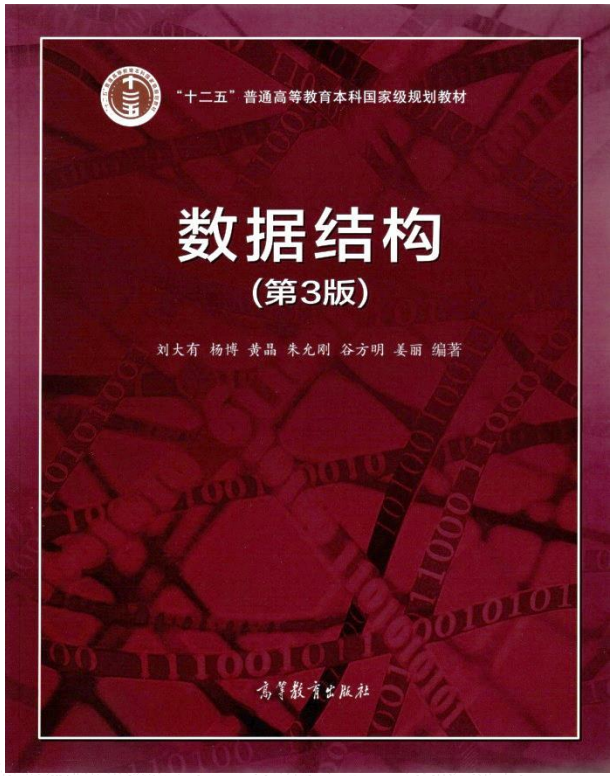
我每周平均写5个新程序。诗人必须写诗，而我必须写程序。

——Donald Knuth

2020年接受采访（时年82岁）

图灵奖获得者、斯坦福大学教授

美国科学院院士、美国工程院院士



二叉树的存储和操作

- **二叉树的存储结构**
- 二叉树的遍历
- 二叉树的其他基本操作
- 二叉树的序列化/反序列化

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2025.4

二叉树的顺序存储

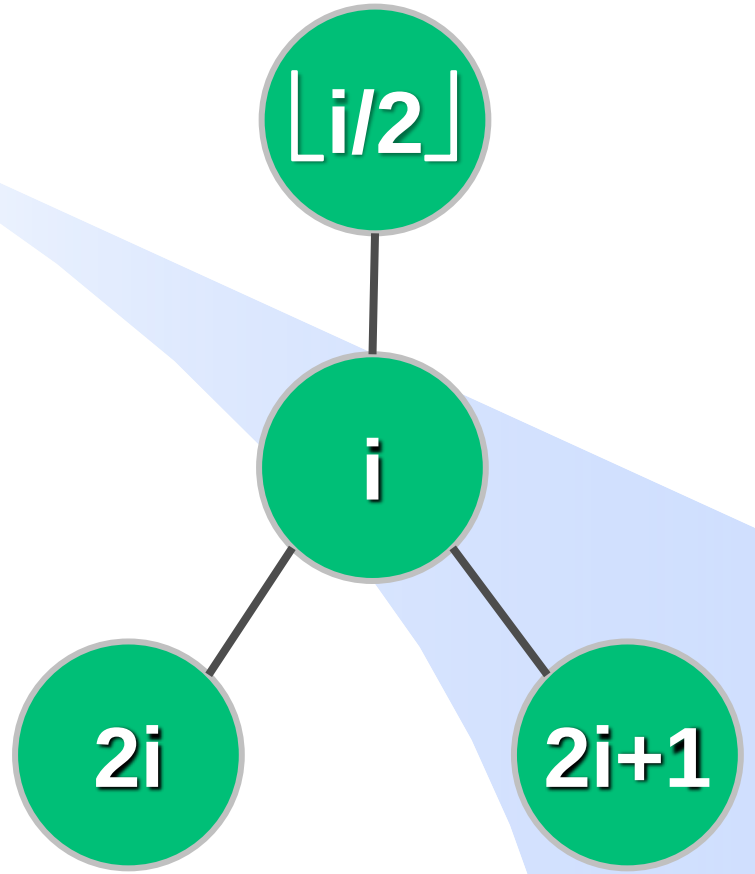
A

- 要存储一棵二叉树，必须存储其所有结点的数据信息、左孩子和右孩子地址，可用顺序结构存储，也可用链接结构存储。
- 二叉树的顺序存储是指将二叉树中所有结点存放在一块地址连续的存储空间中，同时反映出二叉树中结点间的逻辑关系。

二叉树的顺序存储

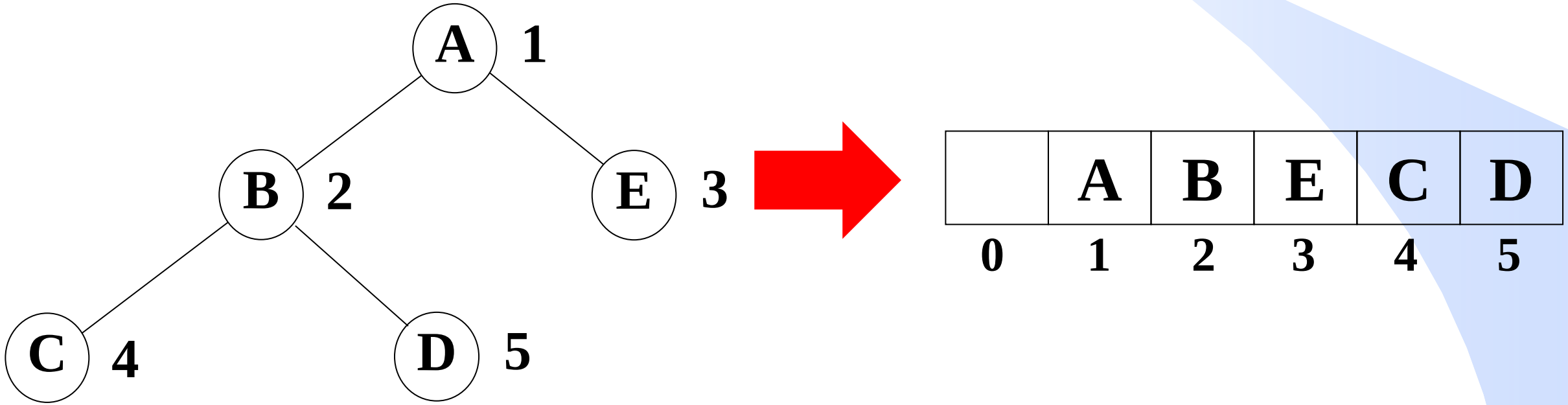
- 回顾：对于完全二叉树，可按层次顺序对结点编号，结点的编号恰好反映了结点间的逻辑关系。

- 借鉴上述思想，利用一维数组*T*存储二叉树，把编号作为数组下标，根结点存放在*T*[1]位置 。
- 结点*T*[*i*]的左孩子（若存在）存放在*T*[2*i*]处，而*T*[*i*]的右孩子（若存在）存放在*T*[2*i*+1]处。



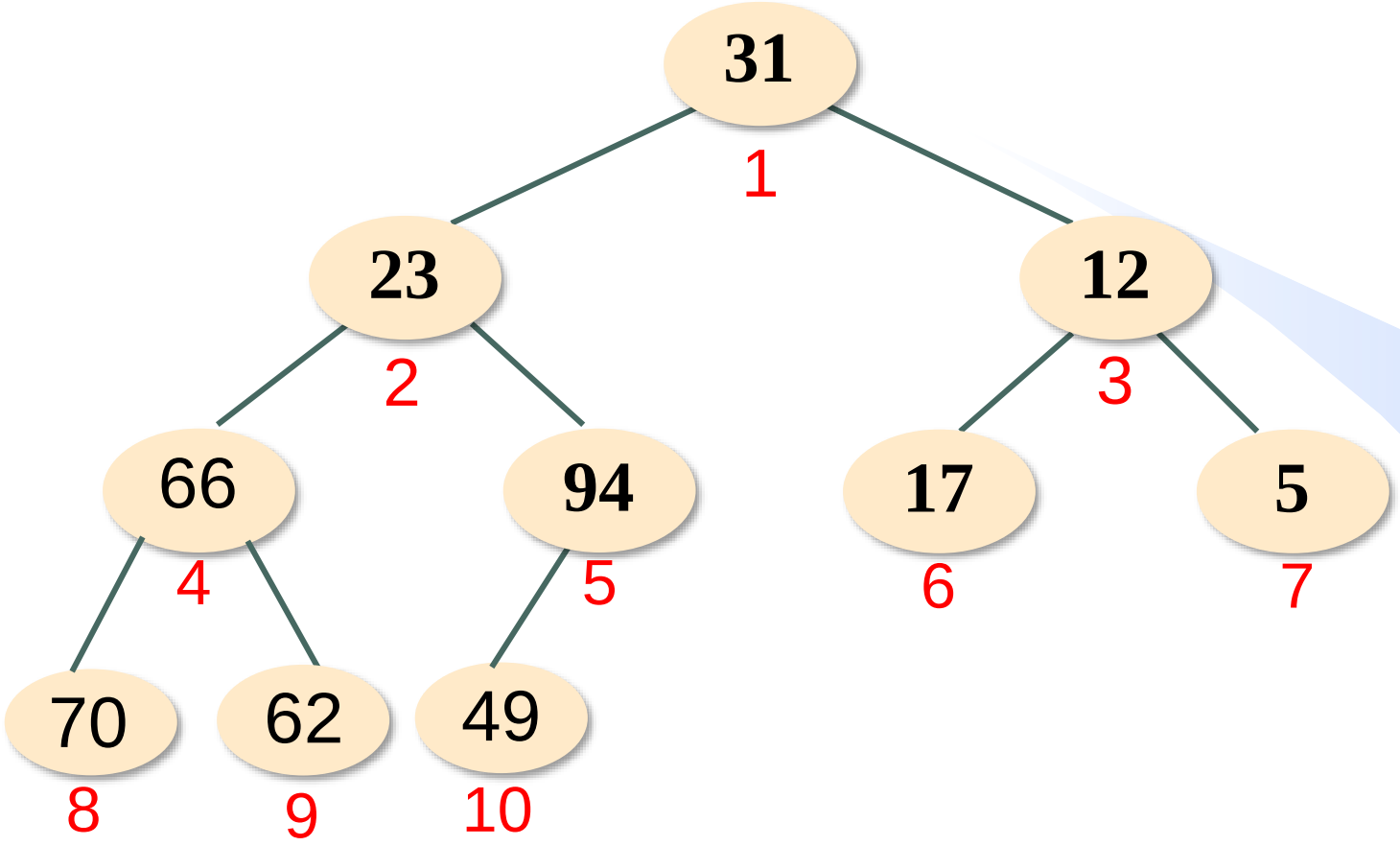
二叉树的顺序存储

若一个结点的下标是*i*，则其左孩子（若存在）存放在下标2*i*处，右孩子（若存在）存放在2*i*+1处



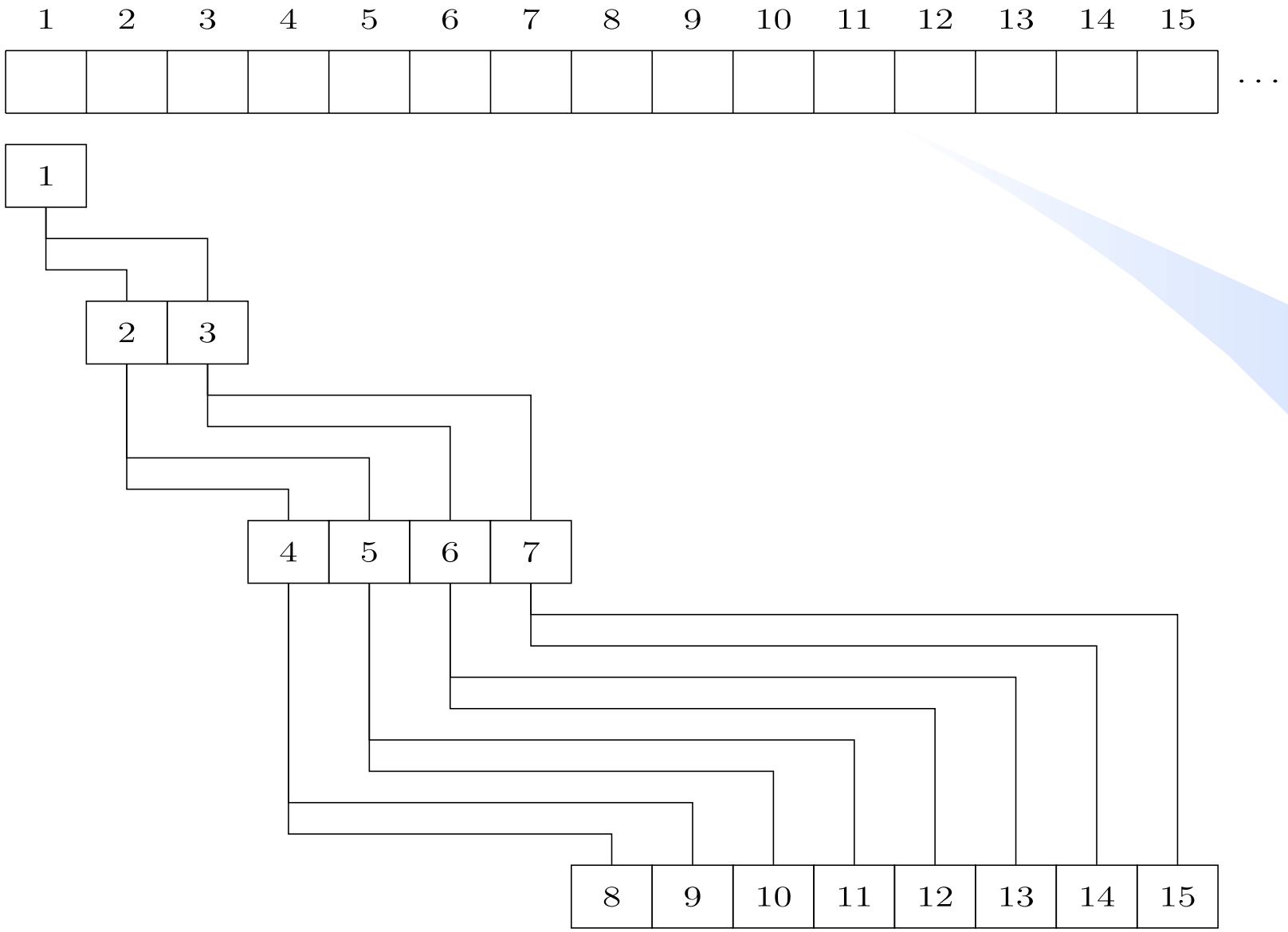
二叉树顺序存储结构

二叉树的顺序存储



T		31	23	12	66	94	17	5	70	62	49
	0	1	2	3	4	5	6	7	8	9	10

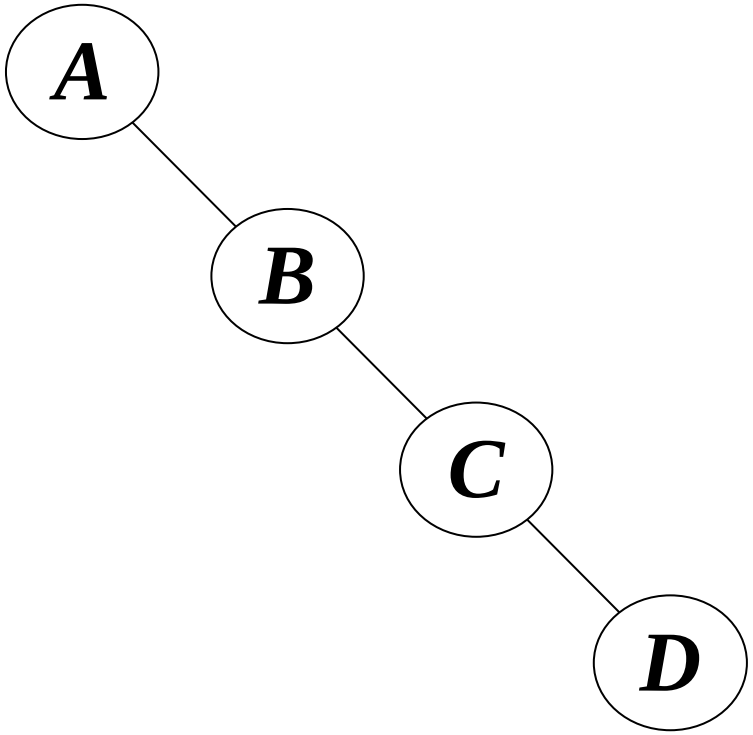
二叉树的顺序存储



二叉树的顺序存储

A

- 适合于完全二叉树。但应用到非完全二叉树时，将造成空间浪费。



	A	B	C	D
0	1	2	3	4

无法表达父子结点间的逻辑关系

T		A		B				C							D
		1		3				7							15

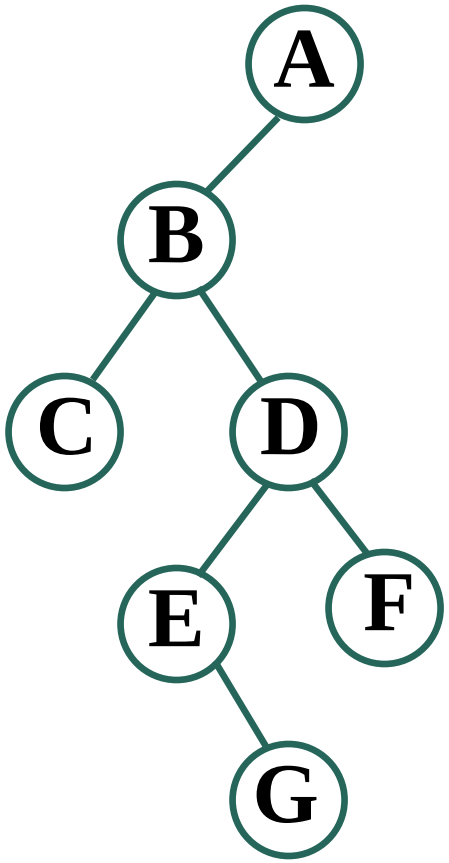
二叉树链接存储——二叉链表

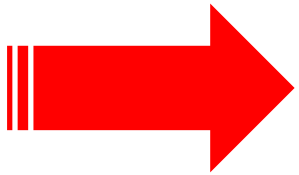
A

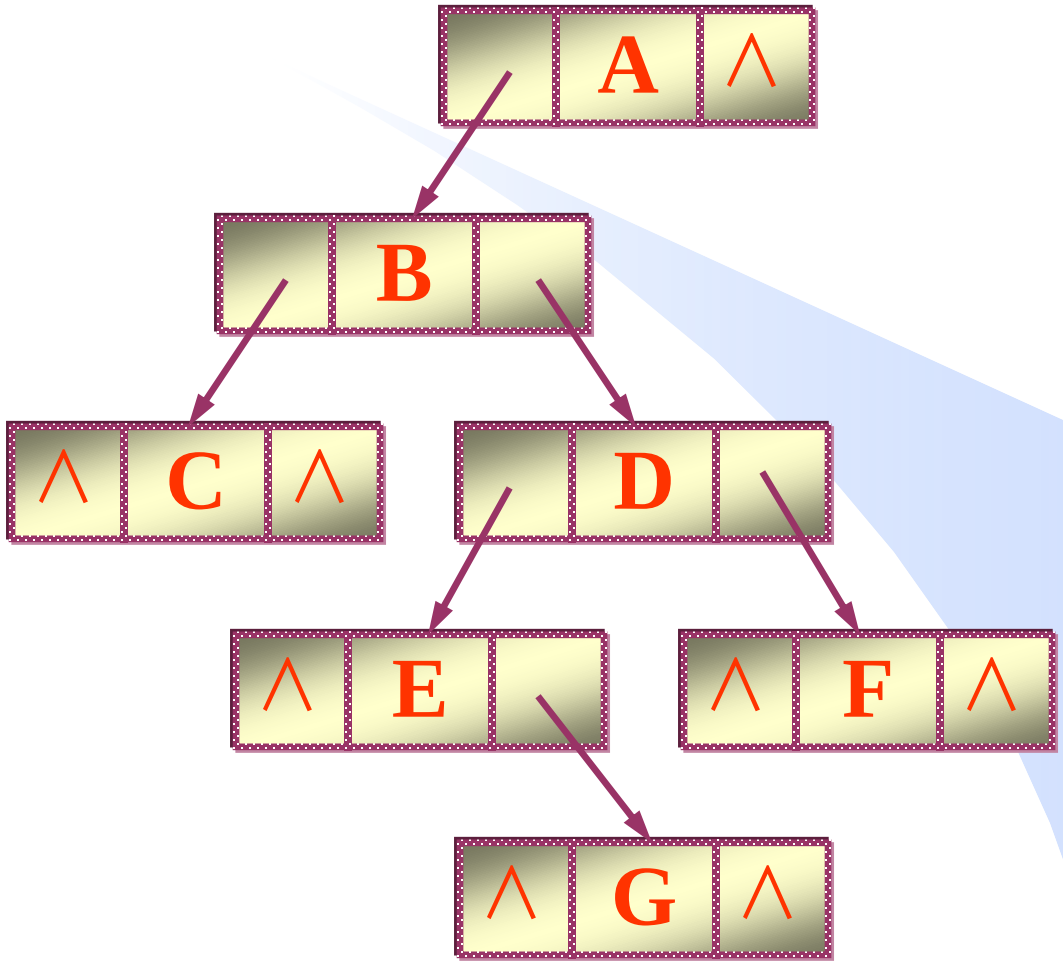
- 各结点被随机存放在内存空间中，结点间的关系用指针表达。
- 二叉树的结点结构：二叉树结点应包含三个域——数据域*data*、指针域*left*（称为左指针）和指针域*right*（称为右指针），其中左、右指针分别指向该结点的左、右子结点（亦称孩子结点）。


```
struct TreeNode{
    int data;
    TreeNode* left;
    TreeNode* right;
};
```

二叉树链接存储





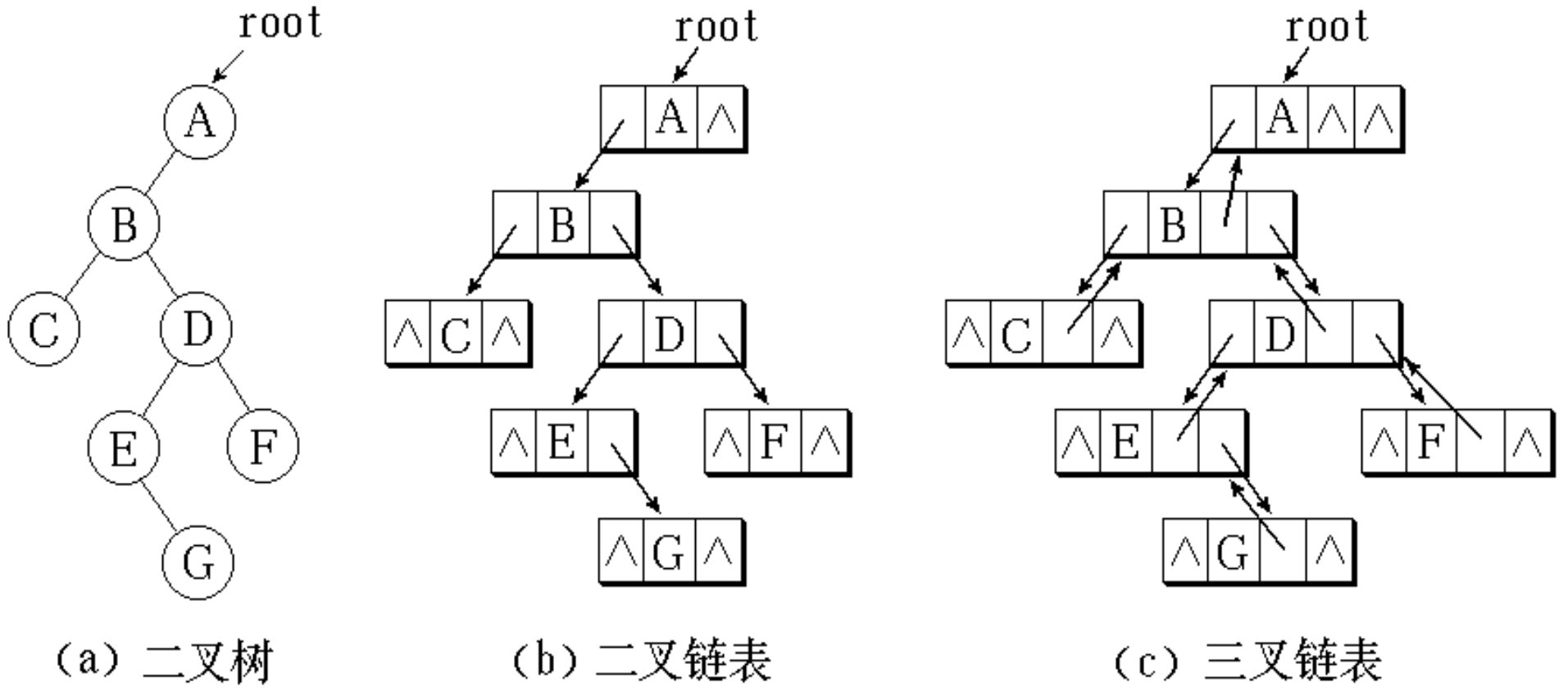


二叉树链接存储——三叉链表

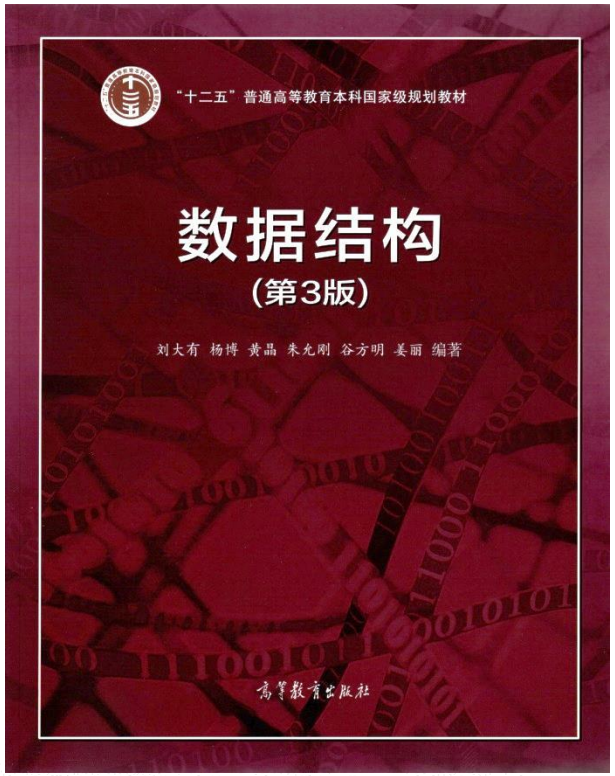
B

Left	Data	Parent	Right

Parent指针指向父结点



(a) 二叉树　(b) 二叉链表　(c) 三叉链表



二叉树的存储和操作

- 二叉树的存储结构
- **二叉树的遍历**
- 二叉树的其他基本操作
- 二叉树的序列化/反序列化

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2025.4

二叉树的遍历

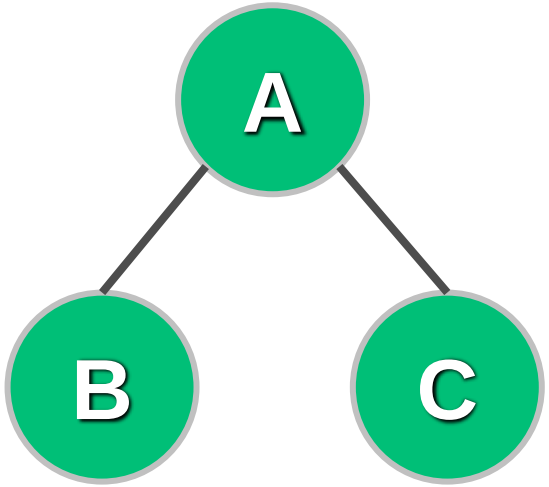
A

二叉树的遍历：按照一定**次序**访问二叉树中所有结点，并且每个结点仅被**访问一次**的过程。

当二叉树为空则什么都不做；否则遍历分三步进行：

A

遍历方法 / 步骤	先根遍历（前序遍历）
步骤一	访问根结点
步骤二	先根遍历左子树
步骤三	先根遍历右子树

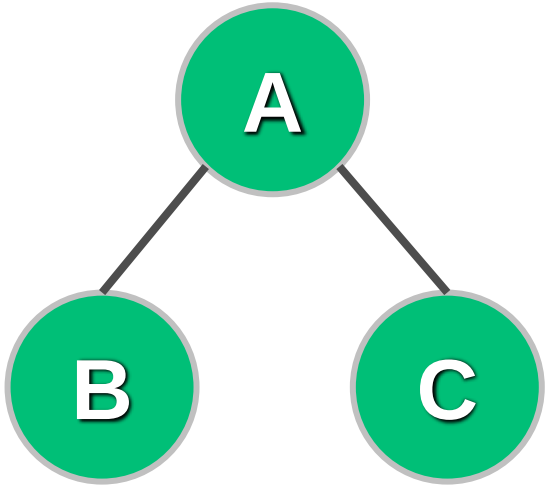


遍历方式 { 先根遍历：ABC

当二叉树为空则什么都不做；否则遍历分三步进行：

A

遍历方法 / 步骤	先根遍历（前序遍历）	中根遍历（中序遍历）
步骤一	访问根结点	中根遍历左子树
步骤二	先根遍历左子树	访问根结点
步骤三	先根遍历右子树	中根遍历右子树

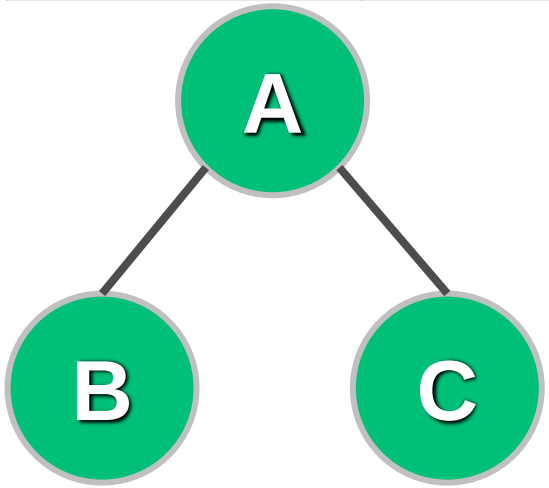


遍历方式
- 先根遍历：ABC
- 中根遍历：BAC

A

当二叉树为空则什么都不做；否则遍历分三步进行：

遍历方法 / 步骤	先根遍历（前序遍历）	中根遍历（中序遍历）	后根遍历（后序遍历）
步骤一	访问根结点	中根遍历左子树	后根遍历左子树
步骤二	先根遍历左子树	访问根结点	后根遍历右子树
步骤三	先根遍历右子树	中根遍历右子树	访问根结点



遍历方式
- 先根遍历：ABC
- 中根遍历：BAC
- 后根遍历：BCA

先根（中根、后根）遍历二叉树 T，得到 T 之结点的一个序列，称为 T 的先根（中根、后根）序列。

先根遍历 (Preorder Traversal，前/先序遍历)

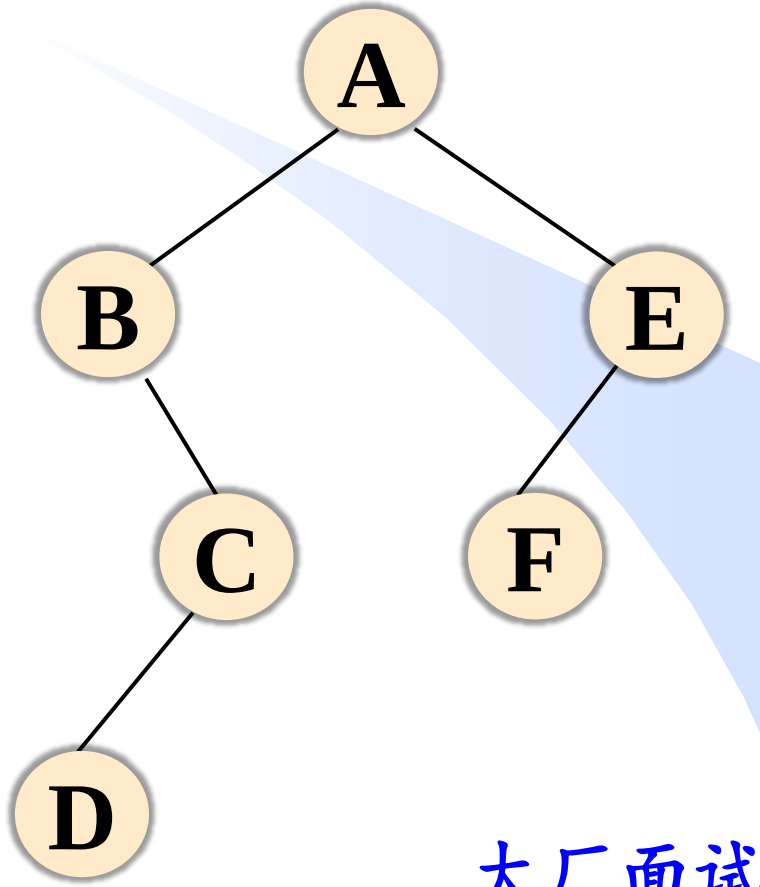
A

先根遍历二叉树算法的框架:

- 若二叉树为空，则空操作;
- 否则
  - ✓访问根结点；
  - ✓先根遍历左子树；
  - ✓先根遍历右子树。

遍历结果

A B C D E F



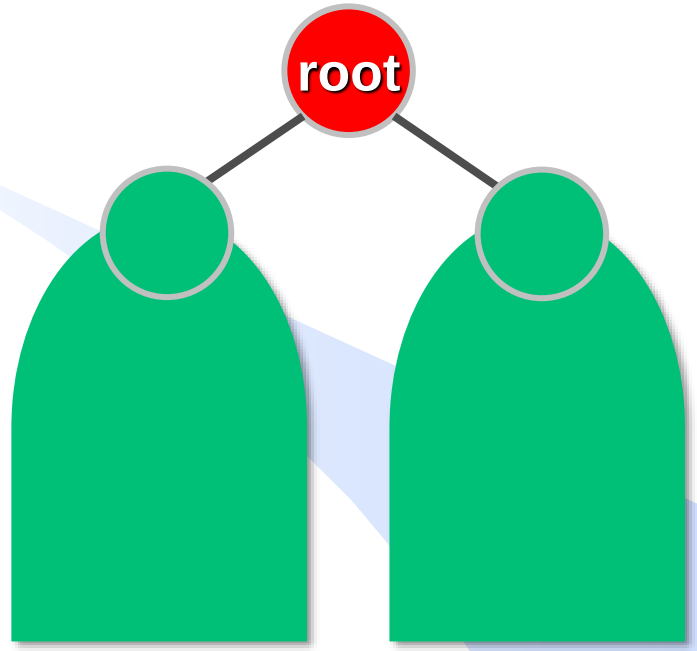
大厂面试题

LeetCode144

二叉树递归先根遍历的算法

A

```
void Preorder(TreeNode* root){
    if(root == NULL) return;
    visit(root->data);
    Preorder(root->left);
    Preorder(root->right);
}
```




```
void visit(int data){
    printf("%d ",data);
}
```

时间复杂度O(*n*)
空间复杂度O(*h*)
*n*为二叉树结点数
*h*为二叉树高度

二叉树递归先根遍历的算法

A

```
void Preorder(TreeNode* root){
    if(root == NULL) return;
    visit(root->data);
    Preorder(root->left);
    Preorder(root->right);
}
```

分治法
分而治之

大事化小
小事化了

大问题
访问整棵树
的所有结点

- 子问题1
  访问根结点
- 子问题2
  访问左子树的结点
- 子问题3
  访问右子树的结点

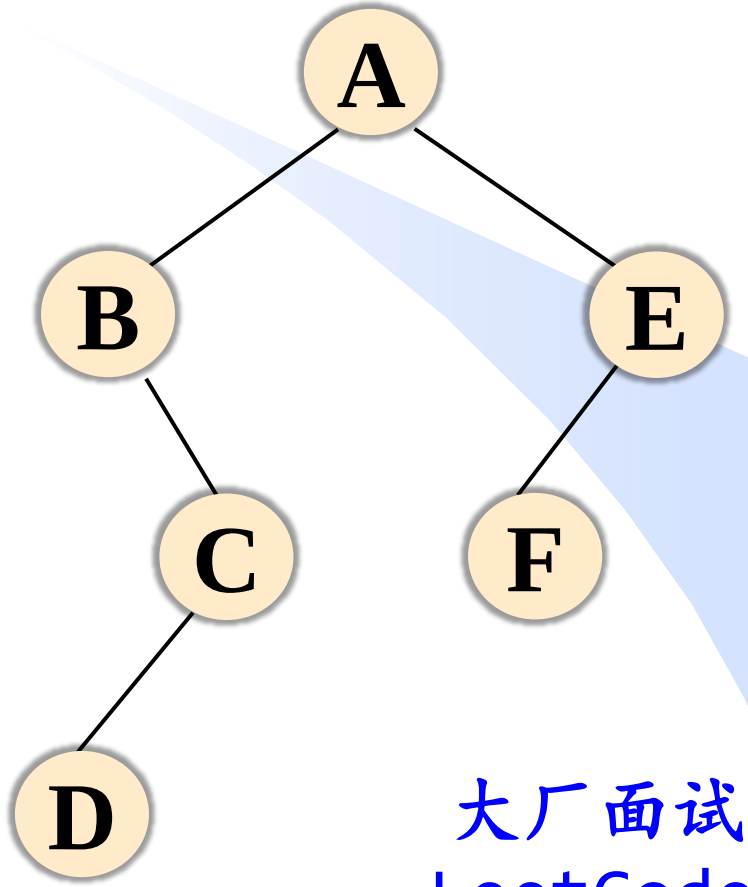
分治法	子问题相互独立 不重叠	自顶向下 递归实现
动态规划	子问题间往往具有重叠性，可将子问题的解存入表中，以后再遇到相同的子问题直接查表	自底向上 递推实现

中根遍历 (Inorder Traversal，中序遍历)

中根遍历二叉树算法的框架：

- 若二叉树为空，则空操作；
- 否则
  - ✓中根遍历左子树；
  - ✓访问根结点；
  - ✓中根遍历右子树。

遍历结果　**B D C A F E**



大厂面试题

LeetCode94

二叉树中根遍历的递归算法

A

```
void Inorder(TreeNode* root){
    if (root == NULL) return;
    Inorder(root->left);
    visit(root->data);
    Inorder(root->right);
}
```

时间复杂度O(*n*)
空间复杂度O(*h*)
*n*为二叉树结点数
*h*为二叉树高度

```
void visit (int data){
    printf("%d ",data);
}
```

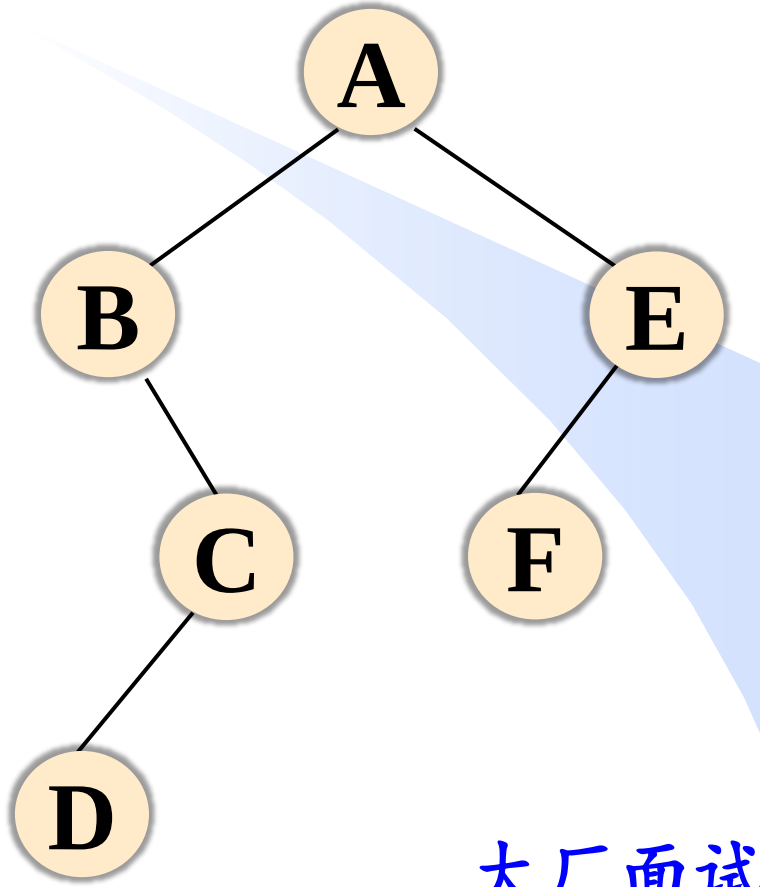
后根遍历 (Postorder Traversal，后序遍历)

A

后根遍历二叉树算法的框架:

- 若二叉树为空，则空操作；
- 否则
  - ✓后根遍历左子树；
  - ✓后根遍历右子树；
  - ✓访问根结点。

遍历结果 **D C B F E A**



大厂面试题

LeetCode145

二叉树后根遍历的递归算法

A

```
void Postorder(TreeNode* root){
    if (root == NULL) return;
    Postorder(root->left);
    Postorder(root->right);
    visit(root->data);
}
```

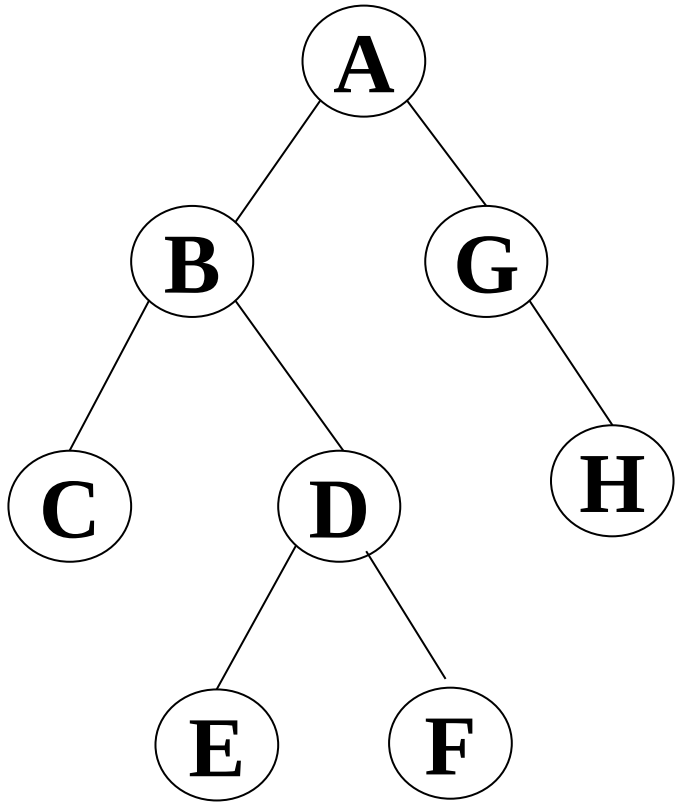
时间复杂度O(*n*)
空间复杂度O(*h*)
*n*为二叉树结点数
*h*为二叉树高度

```
void visit (int data){
    printf("%d ", data);
}
```

课下练习

A

下面二叉树的先根序列为_____，中根序列为_____，后根序列为_____。



练习

A

要使一棵非空二叉树的先根序列与中根序列相同，其所有非叶结点须满足的条件是(　B　)【2017年考研题全国卷】

A.只有左子树　　B.只有右子树

C.结点的度均为1　　D.结点的度均为2

先根序列：根 左 右

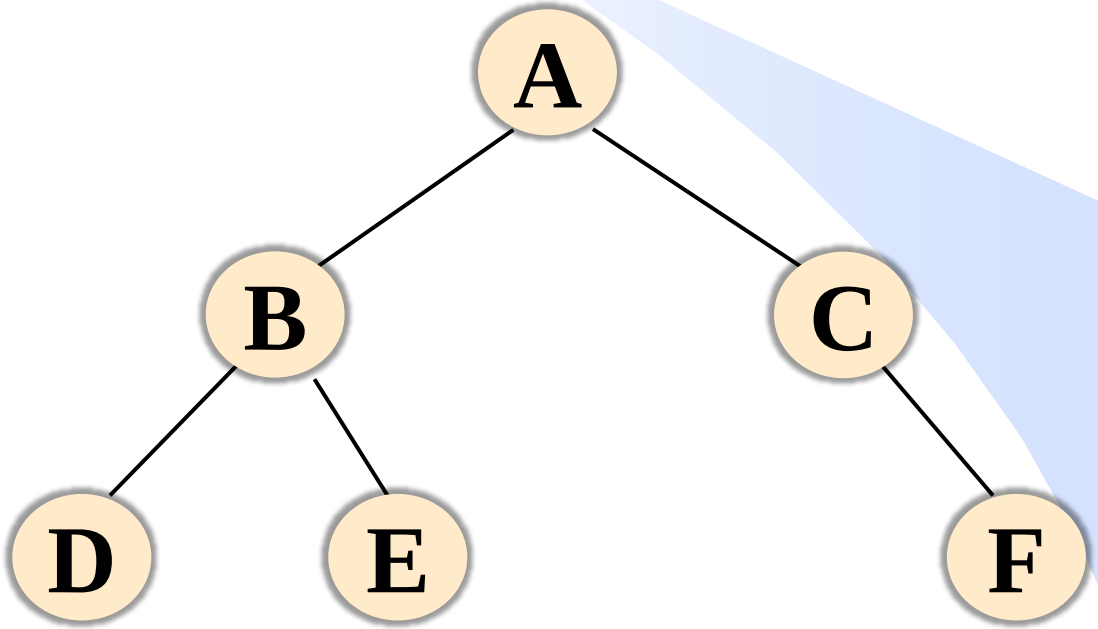
中根序列：左 根 右

层次遍历——定义

按层数由小到大，同层由左向右的次序访问结点。

遍历结果:

A B C D E F

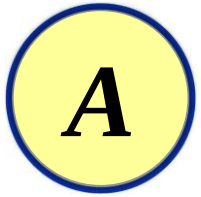


层次遍历——实现

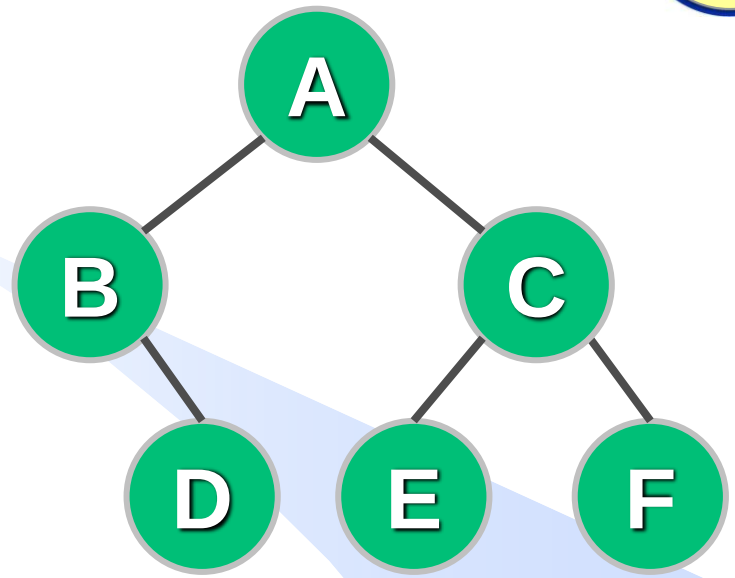
A

- 通过观察发现，在第i层上若结点x在结点y的左边，则x一定在y之前被访问。
- 并且，在第i+1层上，x的子结点一定在y的子结点之前被访问。
- 用一个队列来实现。

层次遍历——实现

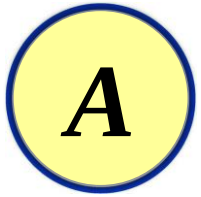


- 根结点入队。
- 重复下列步骤直至队列为空：
  - ✓出队一个结点并访问；
  - ✓若其有左孩子，将左孩子入队；
  - ✓若其有右孩子，将右孩子入队。

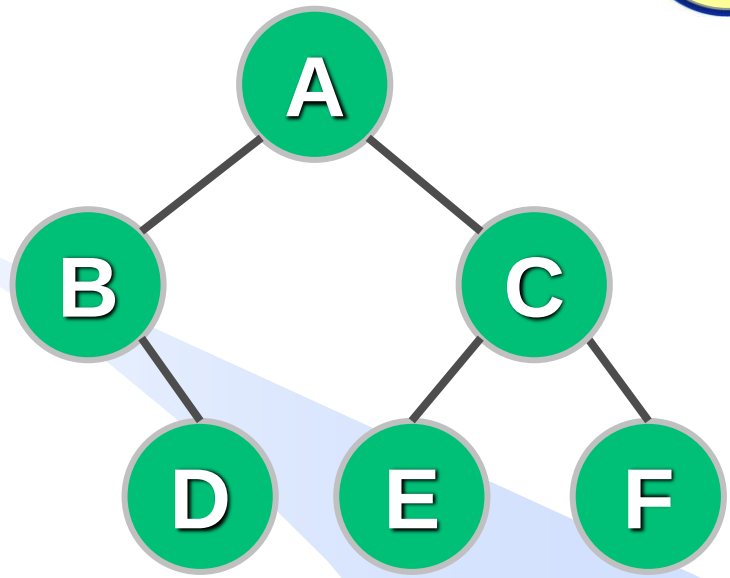


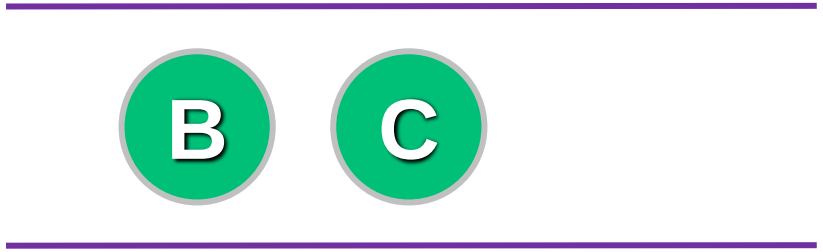
A入队

层次遍历——实现



- 根结点入队。
- 重复下列步骤直至队列为空：
  - ✓出队一个结点并访问；
  - ✓若其有左孩子，将左孩子入队；
  - ✓若其有右孩子，将右孩子入队。





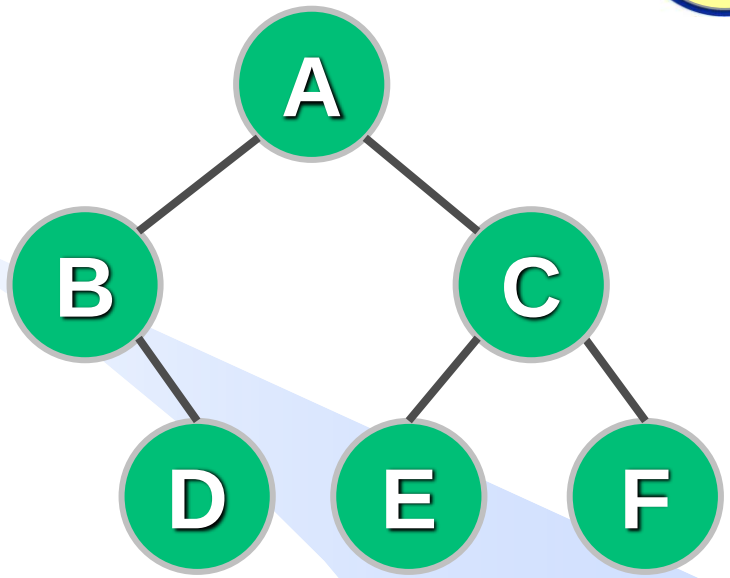
A出队

B、C入队

层次遍历——实现

A

- 根结点入队。
- 重复下列步骤直至队列为空：
  - ✓出队一个结点并访问；
  - ✓若其有左孩子，将左孩子入队；
  - ✓若其有右孩子，将右孩子入队。



C D

B出队

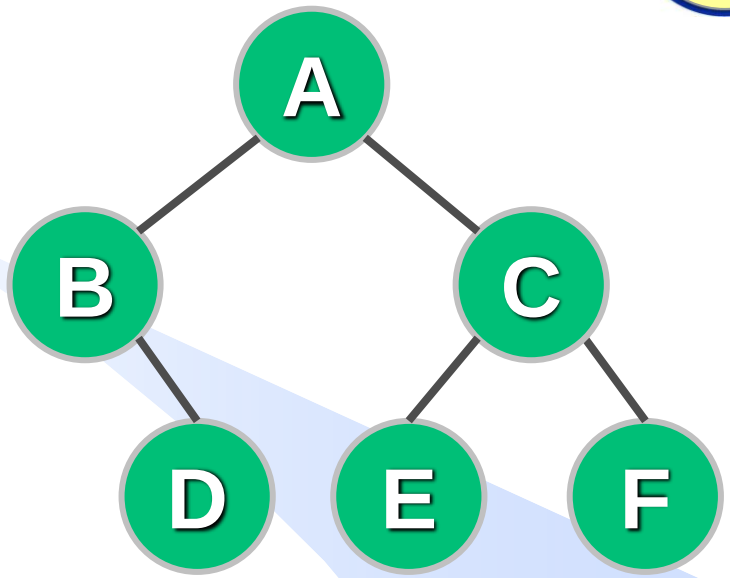
D入队

A B

层次遍历——实现

A

- 根结点入队。
- 重复下列步骤直至队列为空：
  - ✓出队一个结点并访问；
  - ✓若其有左孩子，将左孩子入队；
  - ✓若其有右孩子，将右孩子入队。



D E F

C出队

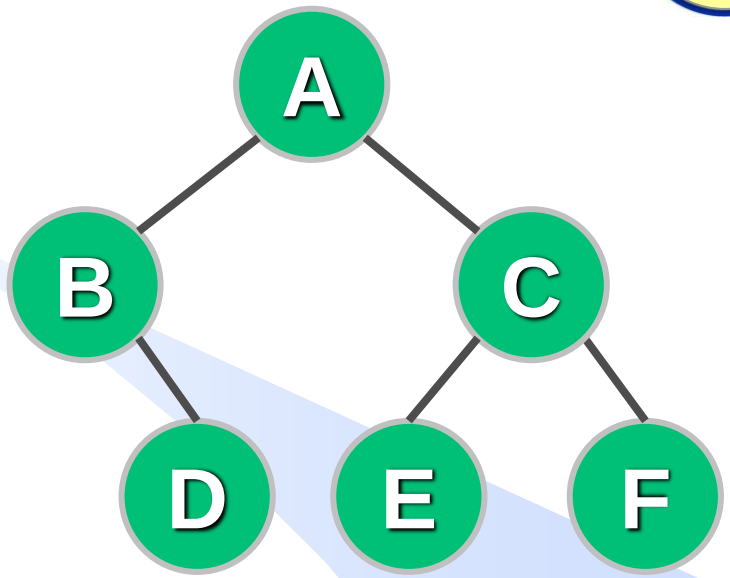
E、F入队

A B C

层次遍历——实现

A

- 根结点入队。
- 重复下列步骤直至队列为空：
  - ✓出队一个结点并访问；
  - ✓若其有左孩子，将左孩子入队；
  - ✓若其有右孩子，将右孩子入队。



E F

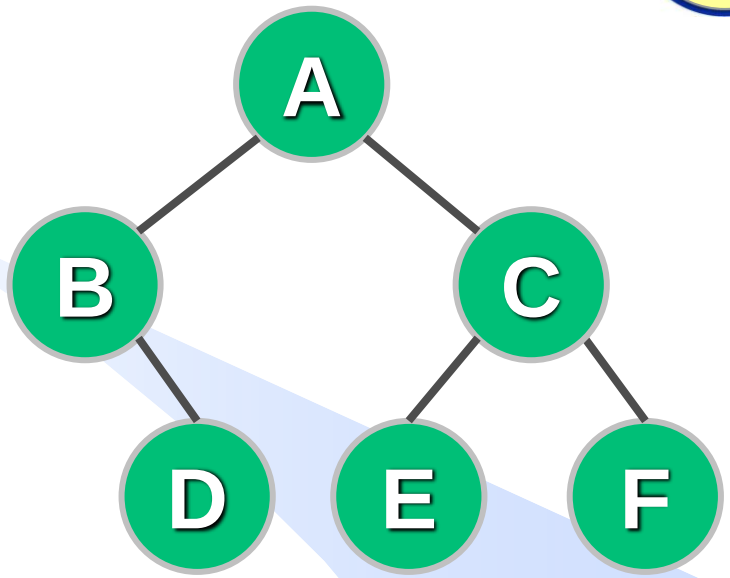
D出队

A B C D

层次遍历——实现

A

- 根结点入队。
- 重复下列步骤直至队列为空：
  - ✓出队一个结点并访问；
  - ✓若其有左孩子，将左孩子入队；
  - ✓若其有右孩子，将右孩子入队。



F

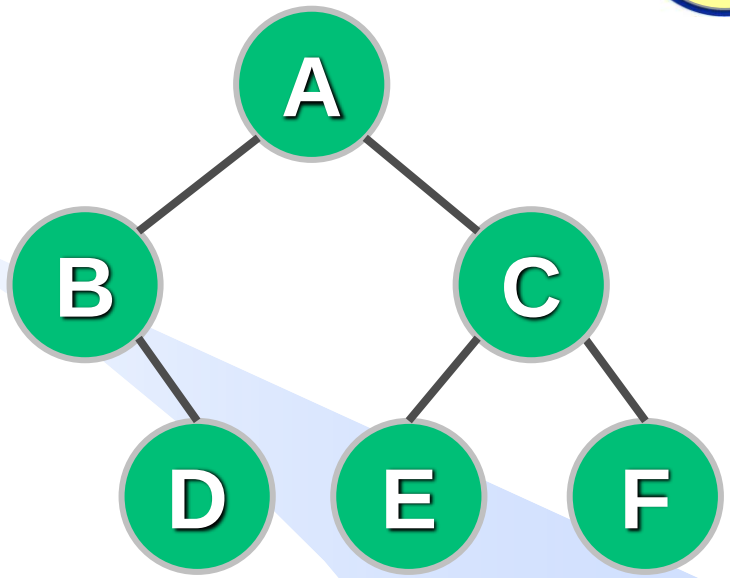
E出队

A B C D E

层次遍历——实现

A

- 根结点入队。
- 重复下列步骤直至队列为空：
  - ✓出队一个结点并访问；
  - ✓若其有左孩子，将左孩子入队；
  - ✓若其有右孩子，将右孩子入队。



F出队

A B C D E F

二叉树层次遍历——实现

A

```
void LevelOrder(TreeNode *root){
    Queue<TreeNode*> Q;
    if(root!=NULL) Q.enQueue(root);
    while(!Q.empty()){
        TreeNode* p = Q.deQueue();
        visit(p->data);
        if(p->left!=NULL) Q.enQueue(p->left);
        if(p->right!=NULL) Q.enQueue(p->right);
    }
}
```

总结

B

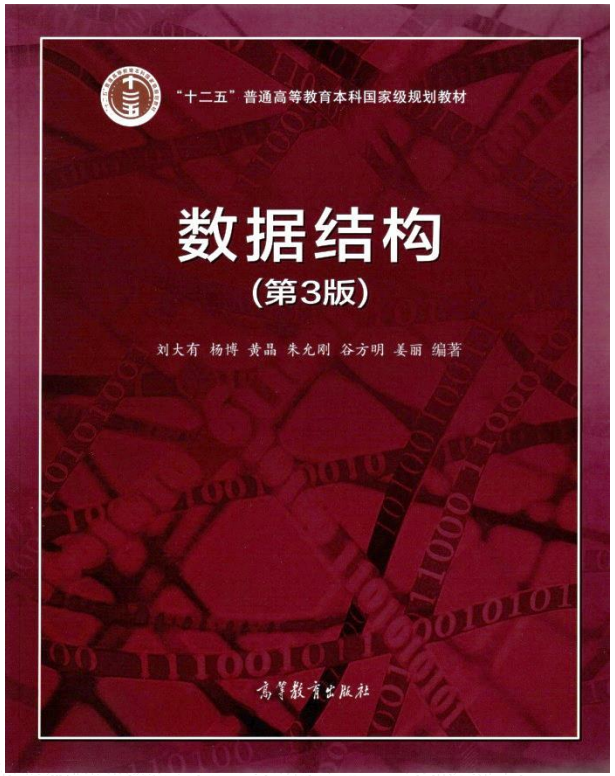
先根遍历

中根遍历

后根遍历

} 深度优先搜索

层次遍历 —— 广度优先搜索



二叉树的存储和操作

- 二叉树的存储结构
- 二叉树的遍历
- **二叉树的其他基本操作**
- 二叉树的序列化/反序列化

数据之法
结构之美
算法之道

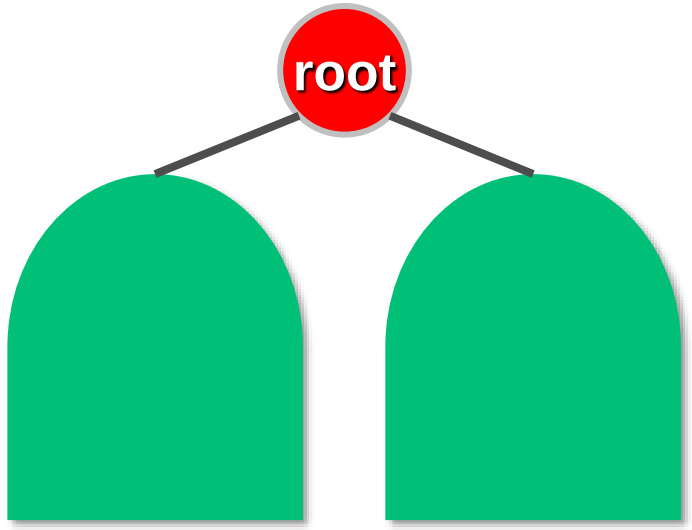
zhuyungang@jlu.edu.cn

Last updated on 2025.4

搜索二叉树中符合数据域条件的结点

A

```
TreeNode* Search(TreeNode *root, int K){
    //在以root为根的二叉树中找数据域为K的结点，返回指针
    if(root == NULL) return NULL;
    if(root->data == K) return root;          //root即为所求
    TreeNode* ans=Search(root->left, K);      //在左子树中查找
    if(ans!=NULL) return ans;
    return Search(root->right, K);            //在右子树中查找
}
```



分治法

遍历

回溯

深度优先搜索

在二叉树中搜索给定结点的父结点

A

```
TreeNode* FindParent(TreeNode *root, TreeNode *p){
    //在以root为根的二叉树中找p的父结点，返回指针
    if(root==NULL||p==root) return NULL;   //根空或p为根
    if(root->left==p||root->right==p)      //root即p的父结点
          return root;
    TreeNode *ans=FindParent(root->left,p);//在左子树找p父结点
    if(ans!=NULL) return ans;
    return FindParent(root->right, p);     //在右子树找p父结点
}
```

解决二叉树问题的一般框架

A

算法*f* (*root*)

可在此处处理根结点

递归处理左子树*f* (*root*->*left*).

可在此处处理根结点

递归处理右子树*f* (*root*->*right*).

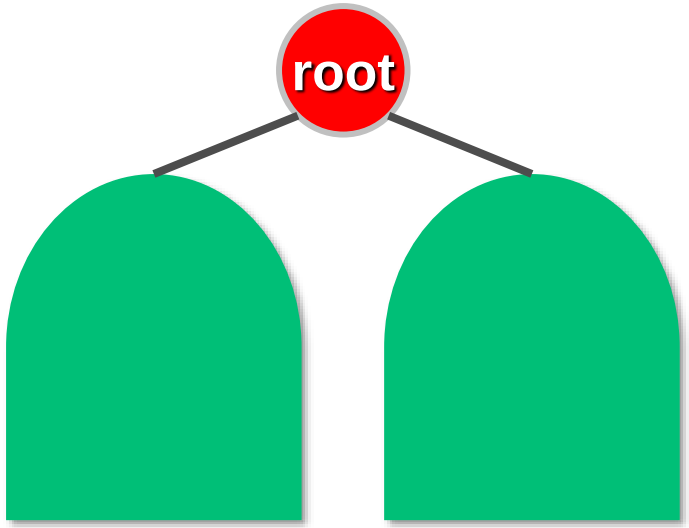
可在此处处理根结点

RETURN.

计算二叉树结点个数

A

```
int Count(TreeNode* root){
    if(root==NULL) return 0;
    return Count(root->left)+Count(root->right)+1;
}
```



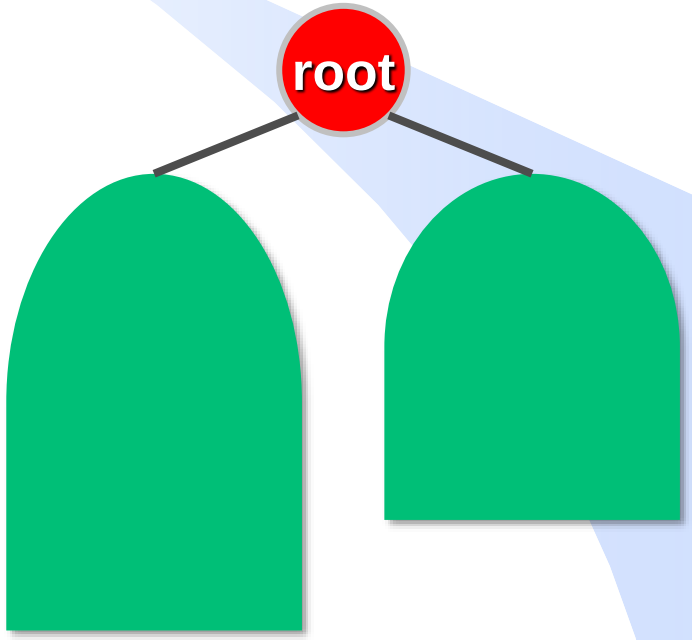
二叉树中结点总数 = 左子树结点总数+右子树结点总数+1（根结点）

计算二叉树高度

A

$$depth(t)=\begin{cases}-1 & 若t=NULL\\ \max\{depth(t->left),depth(t->right)\}+1 & 若t\neq NULL\end{cases}$$

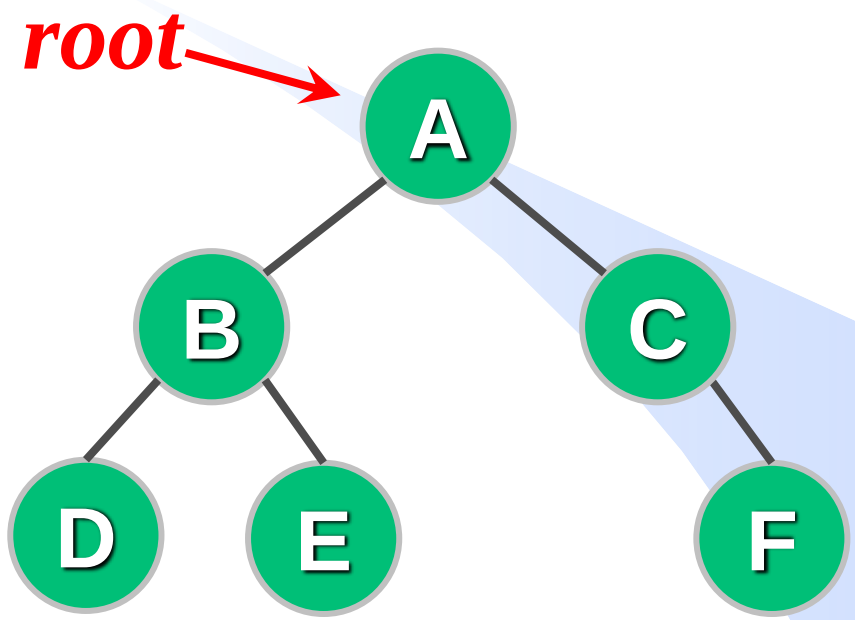
```
int depth(TreeNode* root){
    if(root==NULL) return -1;
    int d1 = depth(root->left);
    int d2 = depth(root->right);
    return(d1>d2)? d1+1:d2+1;
}
```



删除二叉树

A

```
void Del(TreeNode* &root){ //删除以root为根的二叉树
    if(root==NULL) return;
    Del(root->left);
    Del(root->right);
    delete root;
    root = NULL;
}
```

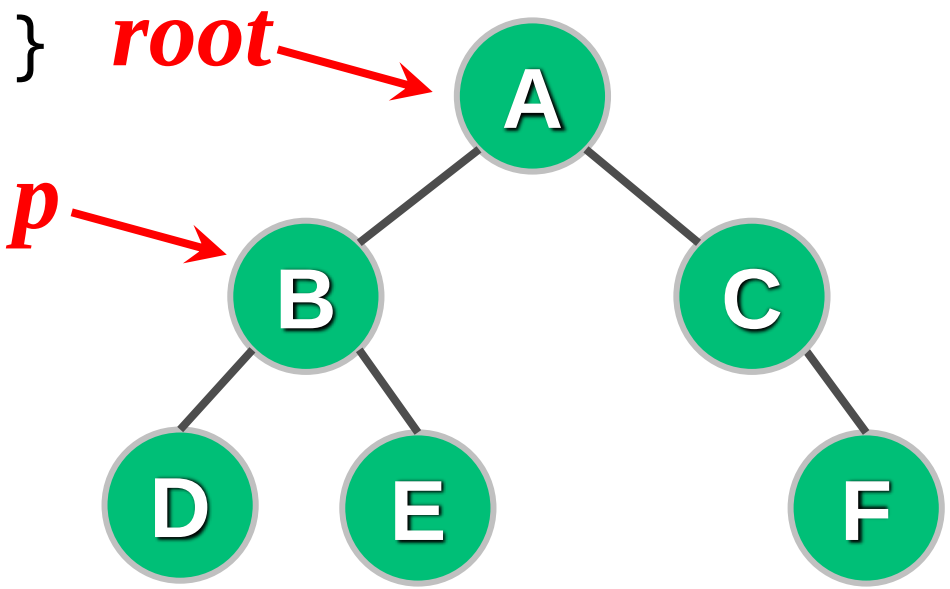


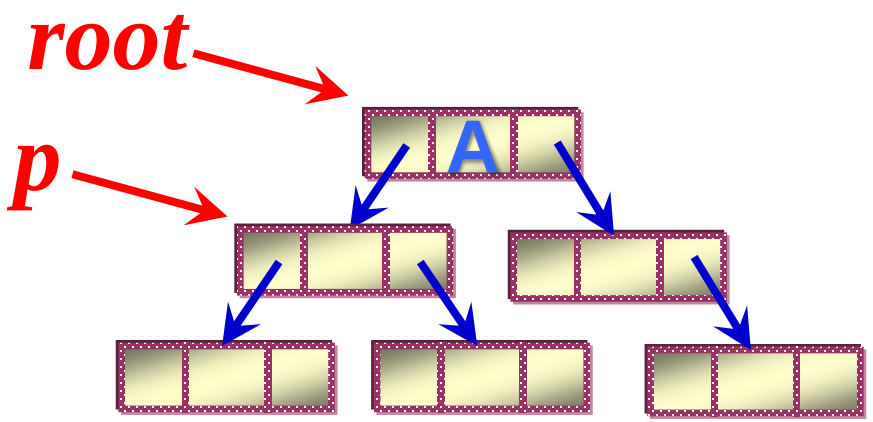
不同于查询操作，插入、删除操作将使二叉树的形态发生变化，在函数执行过程中二叉树的根指针可能被修改，在函数退出后，这种修改应得以继续保留。可将根指针作为函数的“引用参数”进行传递。

在二叉树中删除子树

A

```
bool DelSubTree(TreeNode *&root, TreeNode *p){
   //在以root为根的二叉树中删除p指向的子树,删除成功返回真,否则返回假
   if(root==NULL || p==NULL) return false;
   if(p==root){ Del(root); return true;}      //p是根
   if(DelSubTree(root->left,p)) return true; //在左子树中删p子树
   return DelSubTree(root->right, p);         //在右子树中删p子树
}
```

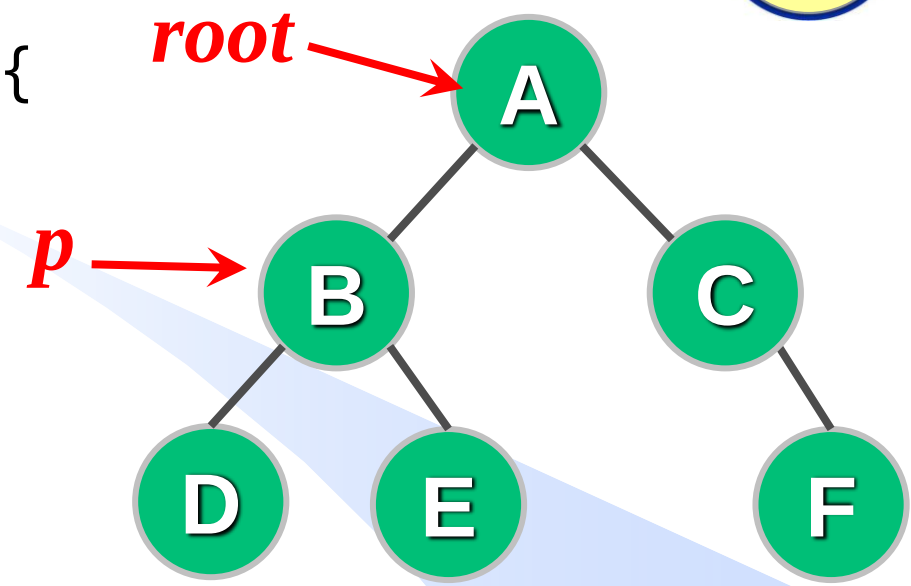




上述函数删除A的左子树后，会把A的左指针置为空么?

在二叉树中删除子树

```
bool DelSubTree(TreeNode *&root, TreeNode *p){
    if(root==NULL || p==NULL) return false;
    if(p==root){ Del(root); return true;}
    if(DelSubTree(root->left, p)) return true;
    return DelSubTree(root->right, p);
}
```



在二叉树中删除子树

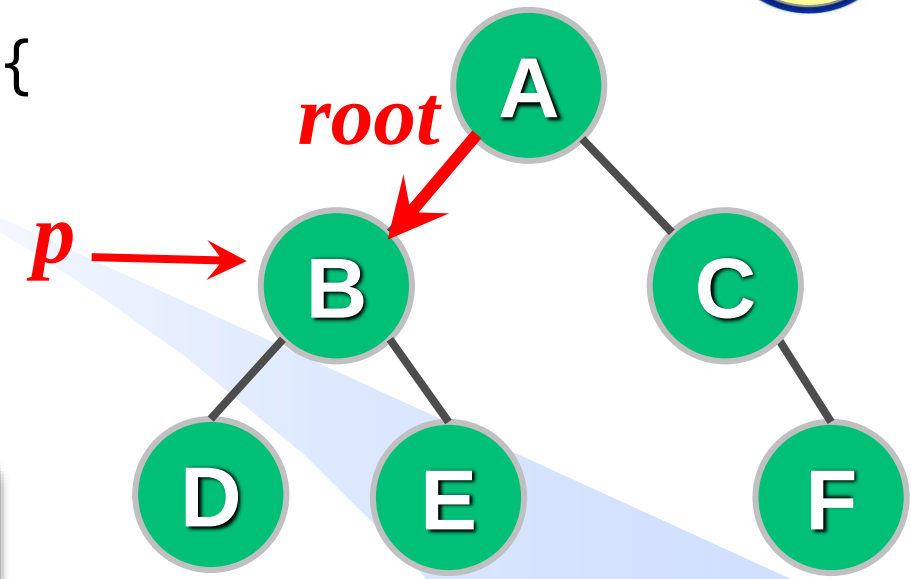
A

```
bool DelSubTree(TreeNode *&root, TreeNode *p){
    if(root==NULL || p==NULL) return false;
    if(p==root){ Del(root); return true;}
    if(DelSubTree(root->left, p)) return true;
    return DelSubTree(root->right, p);
}
```

进入下层递归

此处的root是上一层root->left的引用

```
bool DelSubTree(TreeNode *&root, TreeNode *p){
    if(root==NULL || p==NULL) return false;
    if(p==root){ Del(root); return true;}
    if(DelSubTree(root->left, p)) return true;
    return DelSubTree(root->right, p);
}
```




```
void Del(TreeNode* &root
    if(root==NULL) return;
    Del(root->left);
    Del(root->right);
    delete root;
    root = NULL;
}
```

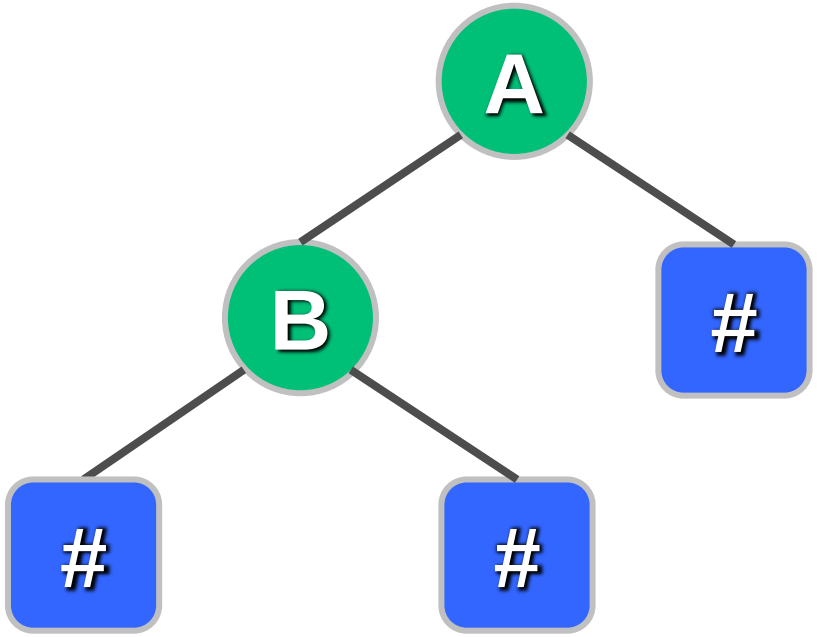
创建二叉树

- 通过一种遍历序列：先根序列？
- 先根序列不能唯一确定二叉树。因为在二叉树中，有的结点之左/右指针可能为空，这在先根序列中不能被体现，导致两棵不同的二叉树却可能有相同的先根序列。
- 在先根序列中加入特殊符号以表示空指针位置，不妨用#表示空指针位置。

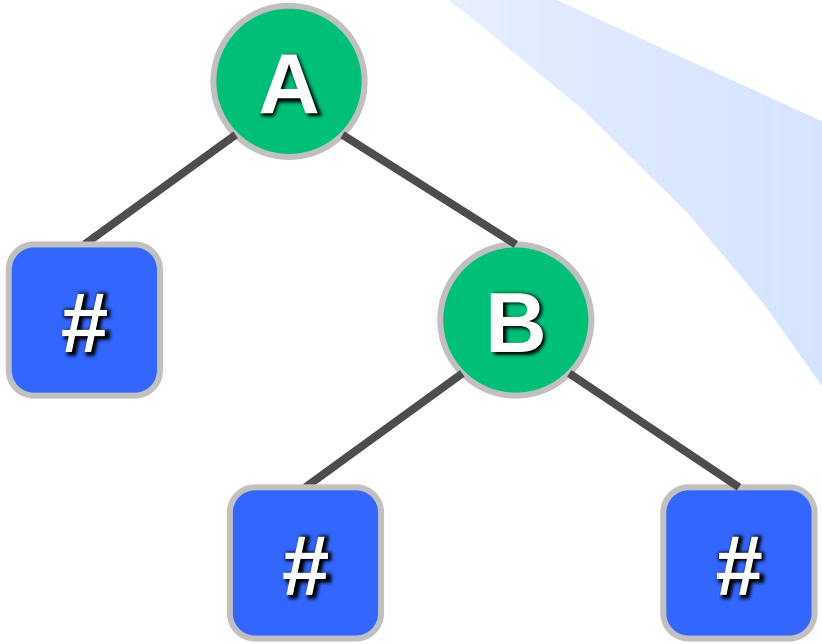
带空指针信息的先根序列

A

下面两棵不同的二叉树有相同的先根序列*AB*。在先根序列中加入特殊符号#以表示空指针位置后，先根序列则不同。



A B # # #

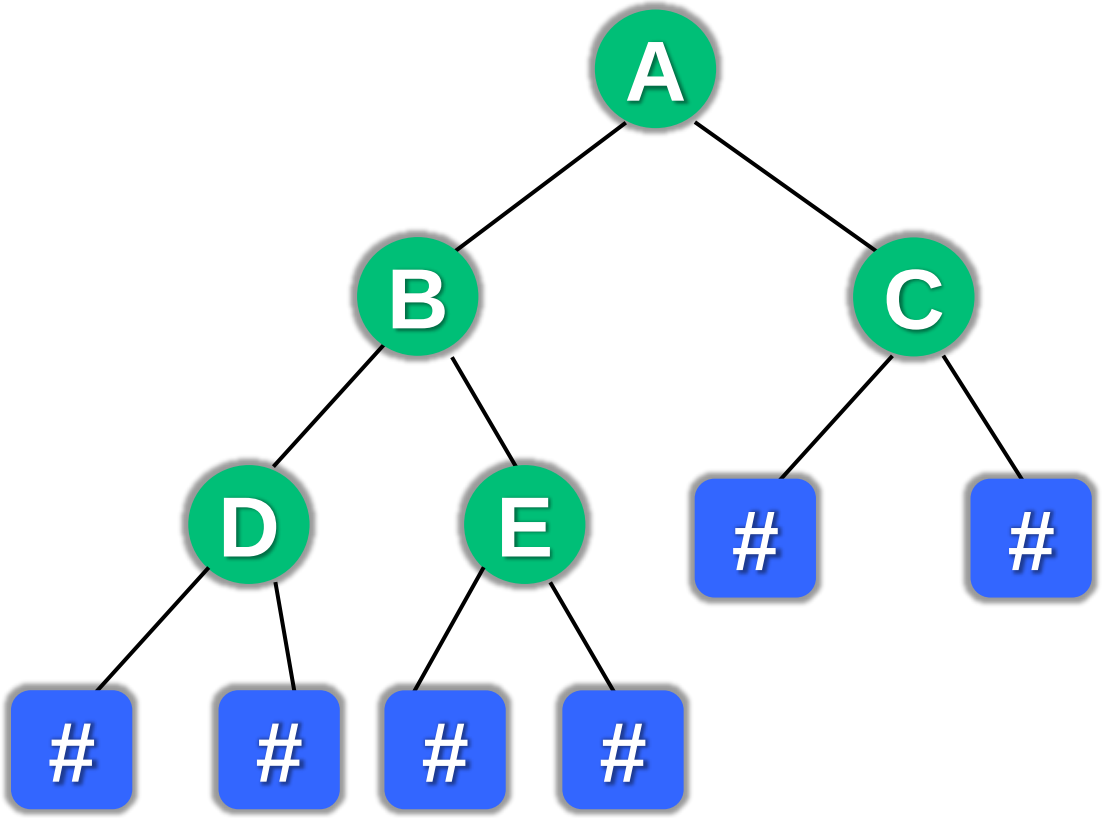


A # B # #

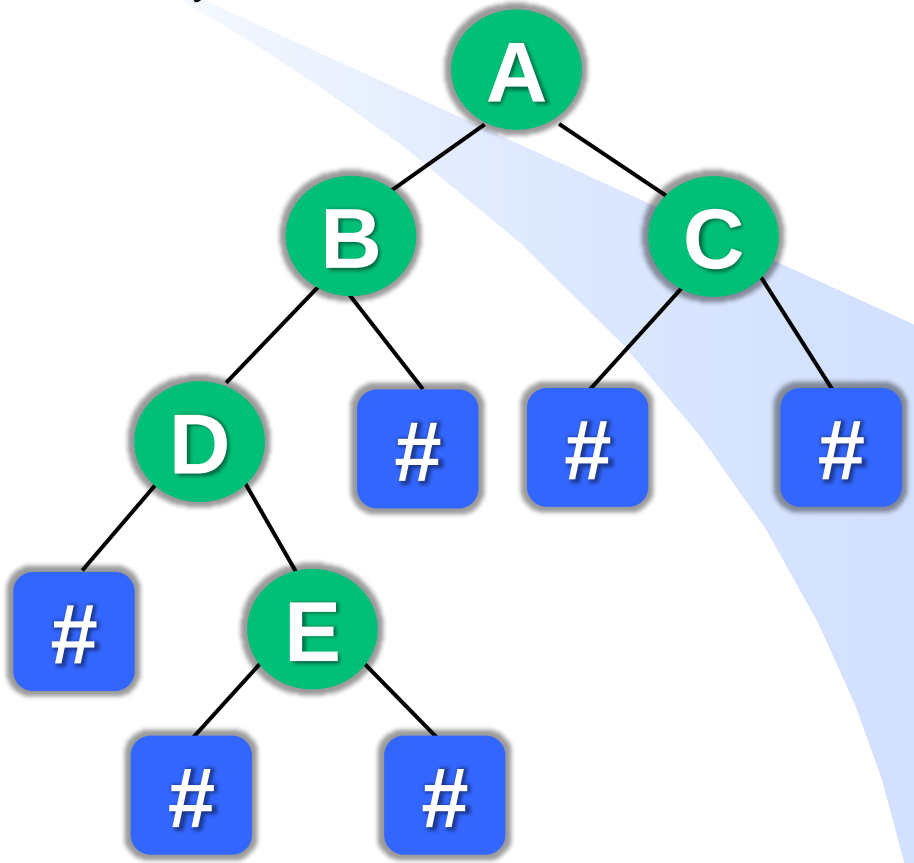
带空指针信息的先根序列

A

下面两棵不同的二叉树有相同的先根序列*ABDEC*。在先根序列中加入特殊符号#以表示空指针位置后，先根序列则不同。



A B D # # E # # C # #



A B D # E # # # C # #

根据增强先根序列创建二叉树

A

根据带空指针信息的先根序列创建二叉树，其中空指针信息用#表示，如*ABD##E##C##*【清华大学考研复试机试】

```
int k=0;
TreeNode* CreateBinTree(char preorder[]){
    char ch=preorder[k++];
    if(ch=='#') return NULL;
    TreeNode *root = new TreeNode;
    root->data=ch;
    root->left=CreateBinTree(preorder);
    root->right=CreateBinTree(preorder);
    return root;
}
```

```
struct TreeNode{
    char data;
    TreeNode *left=NUL;
    TreeNode *right=NULL;
};
```

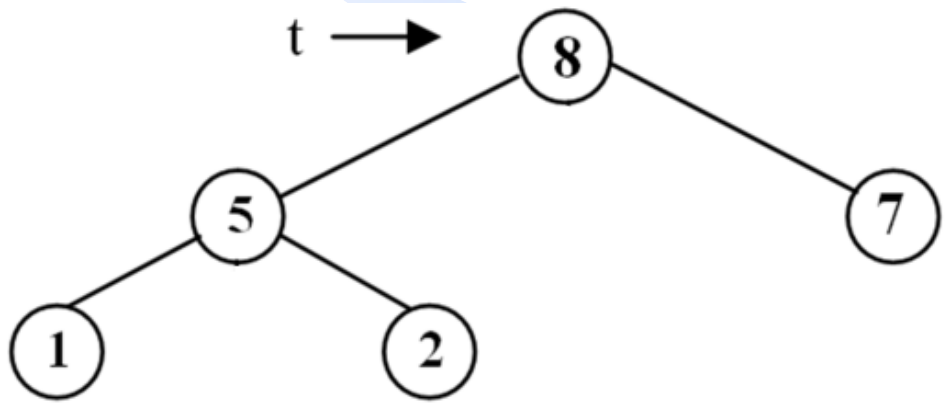
当读入#时，将其初始化为一个空指针，否则生成一个新结点

上机实验常见创建二叉树形式

A

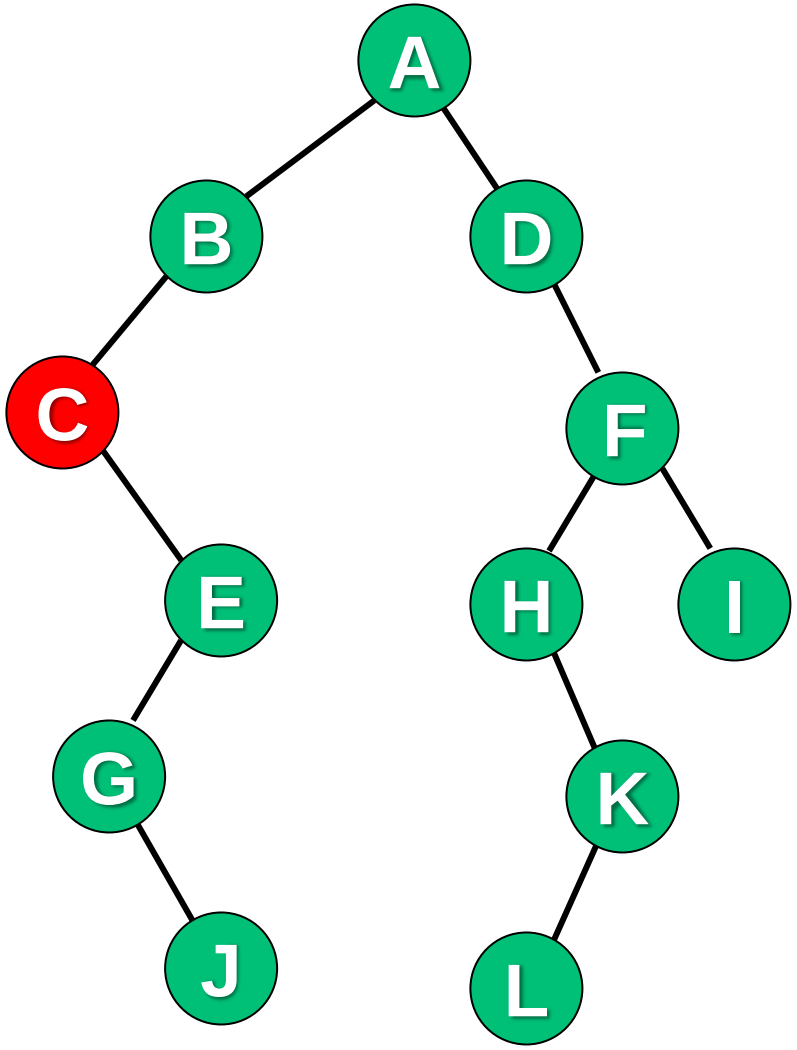
已知一棵非空二叉树结点的数据域为不等于0的整数，输入为一组用空格间隔的整数，表示带空指针信息的二叉树先根序列，其中空指针信息用0表示，如8 5 1 0 0 2 0 0 7 0 0表示如下二叉树。

```
TreeNode* CreateBinTree(){
    int k;
    scanf("%d", &k);
    if(k==0) return NULL;
    TreeNode *root = new TreeNode;
    root->data = k;
    root->left = CreateBinTree();
    root->right = CreateBinTree();
    return root;
}
```



二叉树中根序列的第一个结点

A

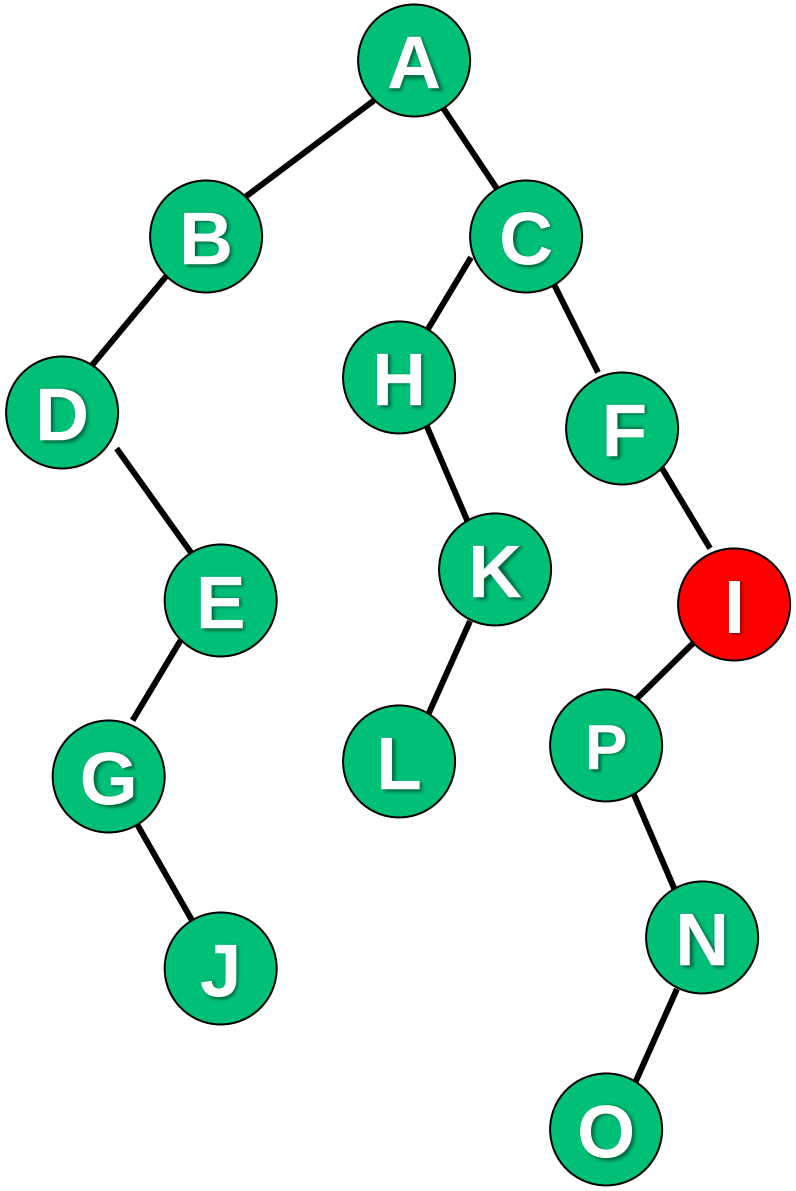



```
if(root==NULL) return NULL;
TreeNode* p=root;
while(p->left!=NULL)
    p=p->left;
return p;
```

时间复杂度O(*h*)
***h*为二叉树高度**

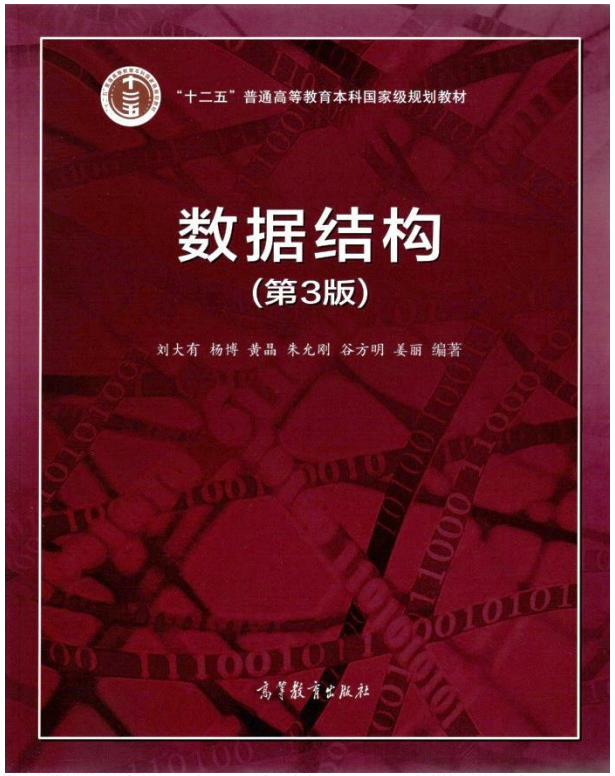
二叉树中根序列的最后一个结点

A




```
if(root==NULL) return NULL;
TreeNode* p=root;
while(p->right!=NULL)
    p=p->right;
return p;
```

时间复杂度O(*h*)
*h*为二叉树高度



二叉树的存储和操作

- 二叉树的存储结构
- 二叉树的遍历
- 二叉树的其他基本操作
- **二叉树的序列化/反序列化**

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2025.4

二叉树的重建

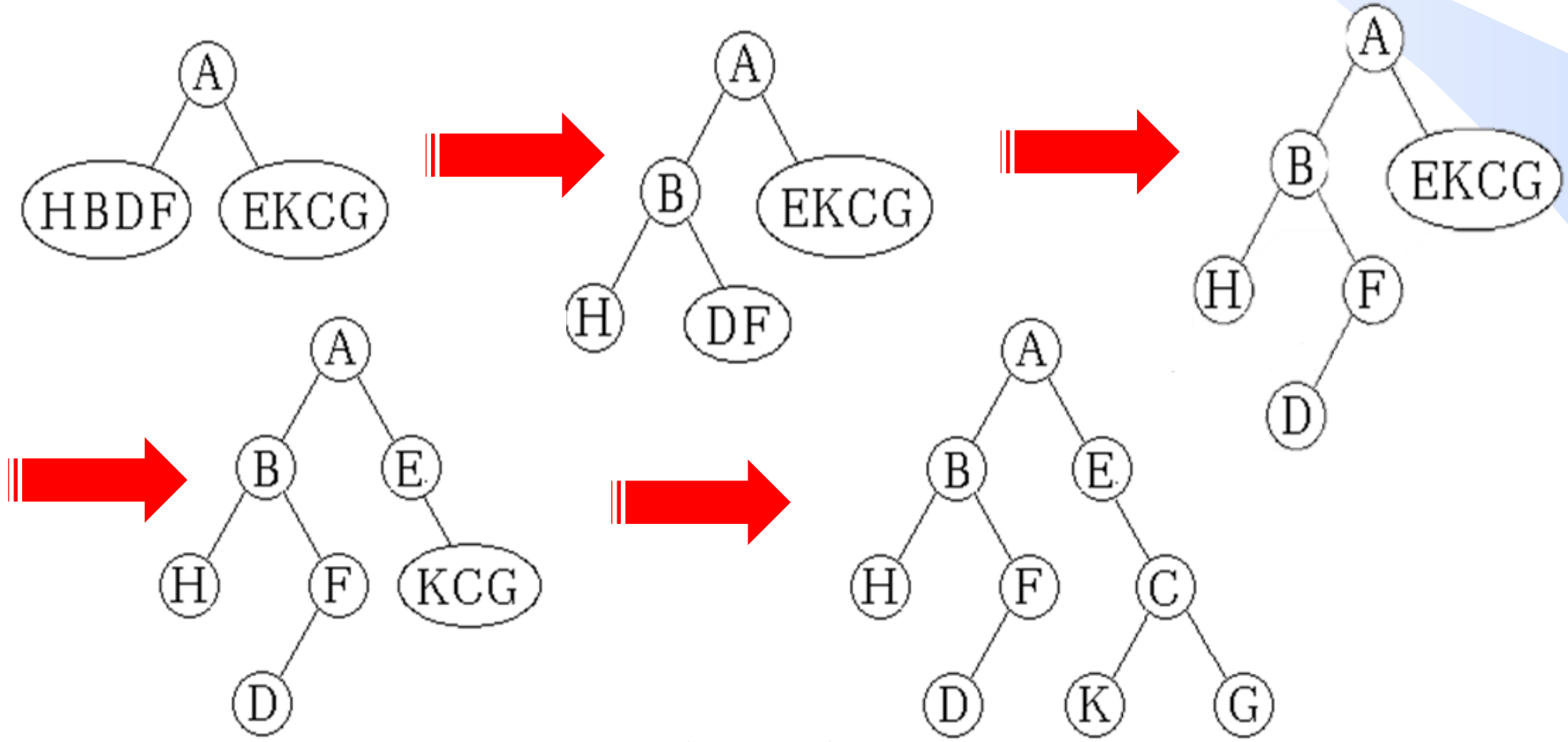
A

由先根序列和中根序列可否唯一确定一棵二叉树？

[例] 先根序列 A B H F D E C K G

中根序列 H B D F A E K C G

通过先根序列确定子树的根
通过中根序列确定左右子树



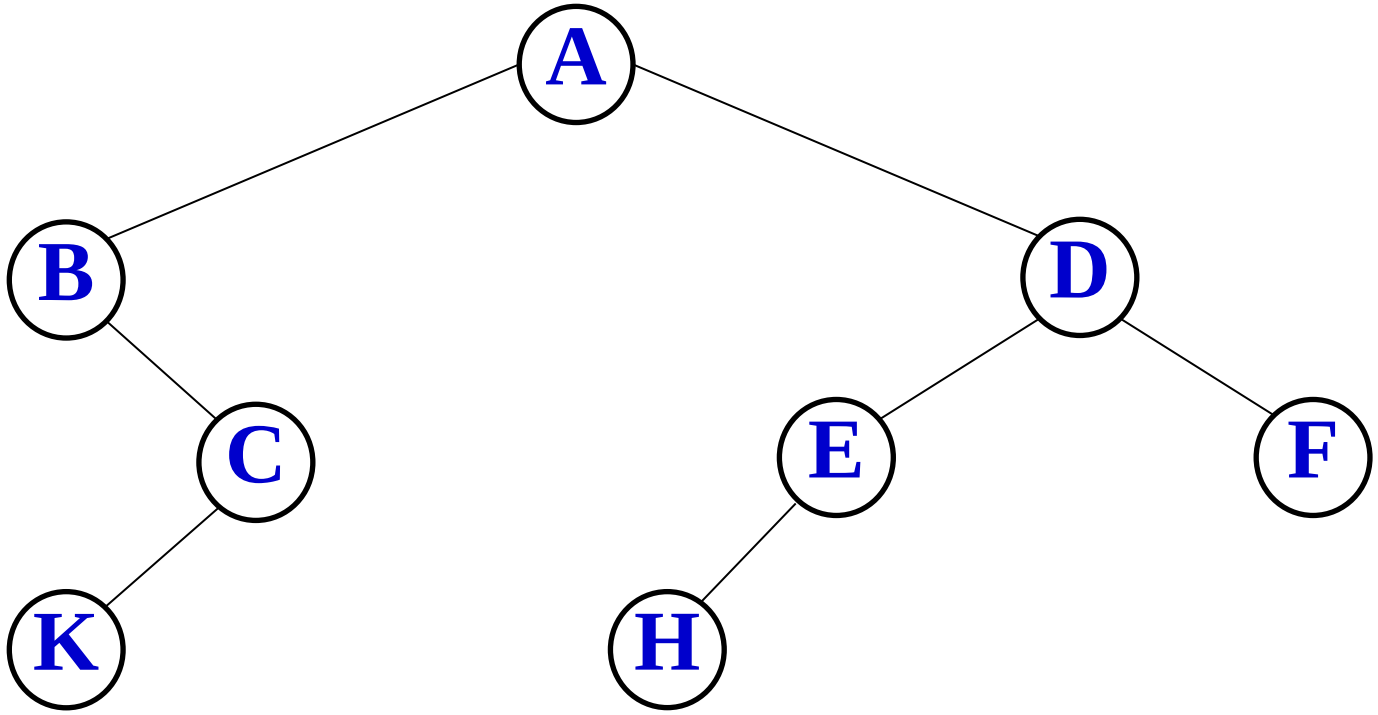
课下练习

A

由以下先根序列和中根序列确定一棵二叉树

先根序列　A B C K D E H F

中根序列　B K C A H E D F



课下练习

A

已知一棵二叉树的先序和中序遍历序列如下：

先序序列：A B C D E F G H I J

中序序列：C B A E F D I H J G

则其后序遍历序列为_____【阿里笔试题】

A. C B D E A G I H J F

B. C B D A E G I H J F

C. C E D B I J H G F A

D. C E D B I H J G F A

E. C B F E I J H G D A

F. C B F E I H J G D A

二叉树的重建——编程实现

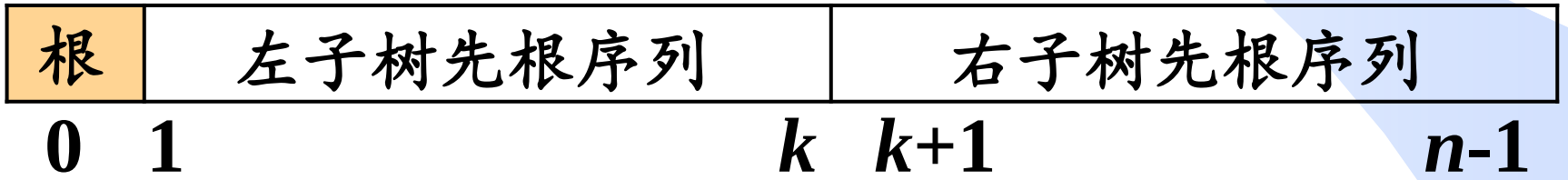
B

给定二叉树的先根序列和中根序列，编写程序构建二叉树。

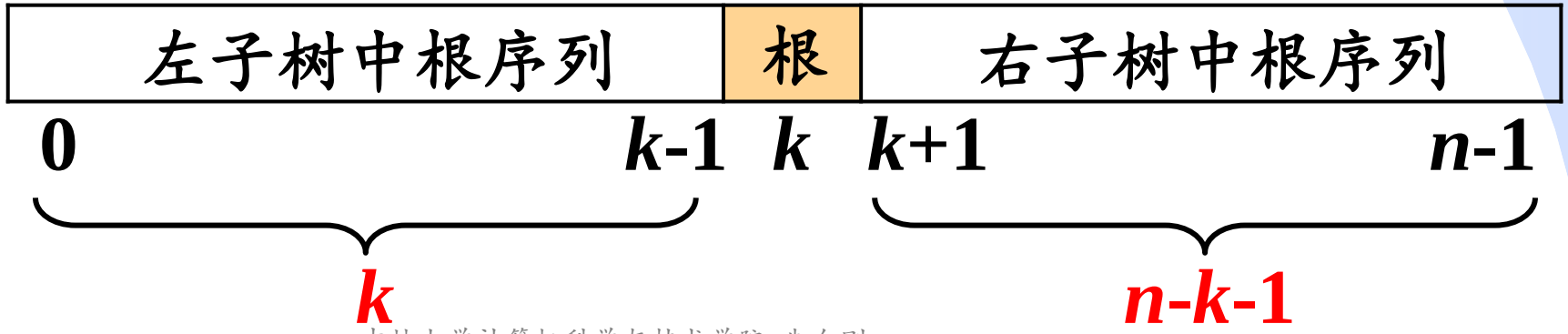
【大厂面试题、华中科技大学考研复试机试题、吉林大学保研复试机试题 OpenJudgeP0570、 LeetCode105】

```
TreeNode* buildTree(char *preorder, char *inorder, int n)
```

先根序列



中根序列



二叉树的重建——编程实现

B

```
TreeNode* buildTree(char *preorder, char *inorder, int n) {
    if (n <= 0) return NULL;
    TreeNode *root = new TreeNode;
    root->data = preorder[0]; //先根序列的根
    int k = findroot(inorder, n, root->data); //在中根序列找根
```

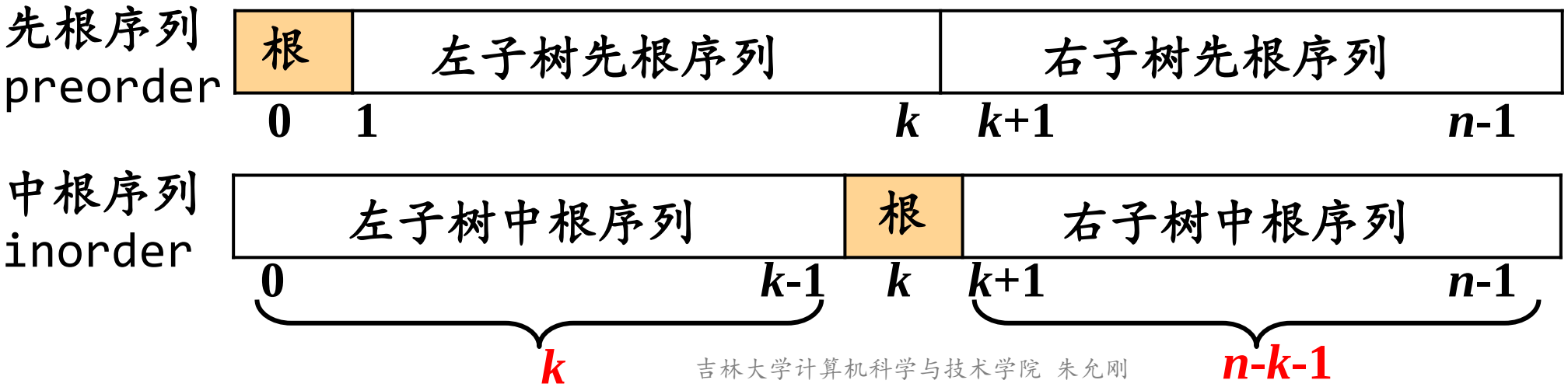
```
int findroot(char *inorder, int size, char val) {
    for(int i = 0; i<size; i++)
        if(inorder[i] == val) return i;
    return -1;
}
```

二叉树的重建——编程实现

B

```
TreeNode* buildTree(char *preorder, char *inorder, int n) {
    if (n <= 0) return NULL;
    TreeNode *root = new TreeNode;
    root->data = preorder[0]; //先根序列的根
    int k = findroot(inorder, n, root->data); //在中根序列找根
    root->left=buildTree(&preorder[1], &inorder[0], k);
    root->right=buildTree(&preorder[k+1],&inorder[k+1],n-k-1);
    return root;
}
```

最好/平均时间复杂度O(*n*log*n*)
最坏时间复杂度O(n^2)

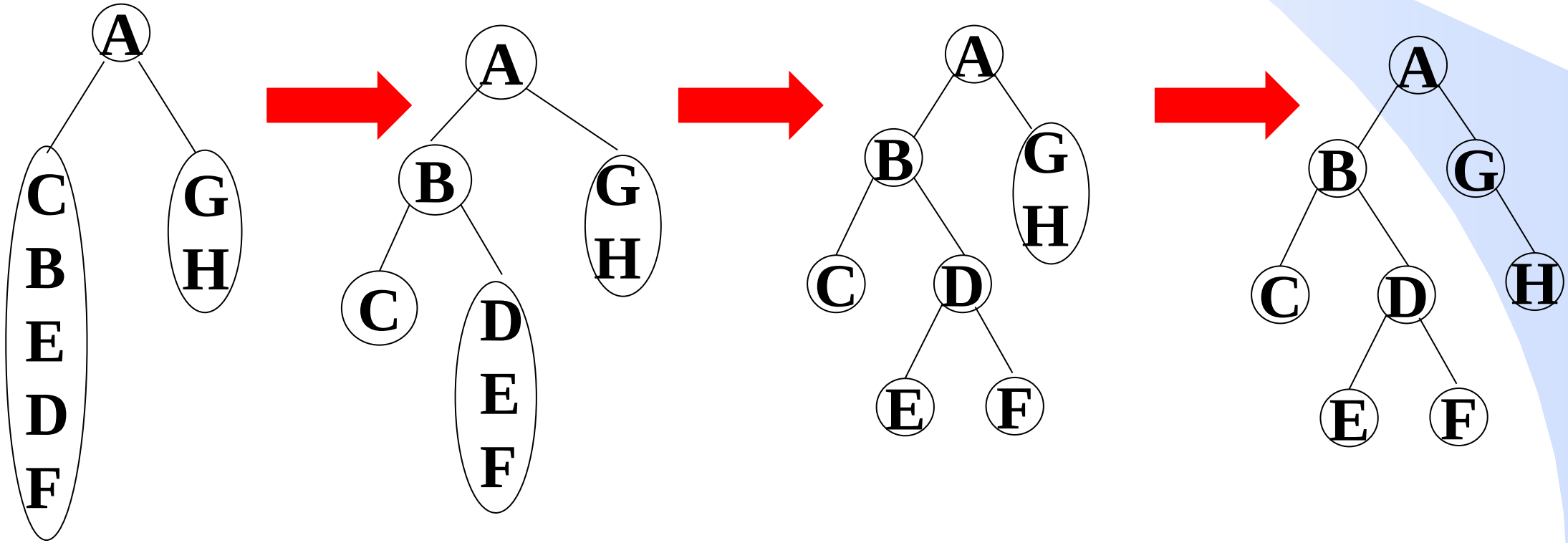


二叉树的重建

由后根序列和中根序列是否可以唯一地确定一棵二叉树？

后根序列　C E F D B H G A

中根序列　C B E D F A G H



A

- 由先根序列和后根序列是否可以唯一地确定一棵二叉树？

- 先根序列：A B
  后根序列：B A

A

- 由层次遍历序列可否唯一确定一棵二叉树？
- 层次序列：AB

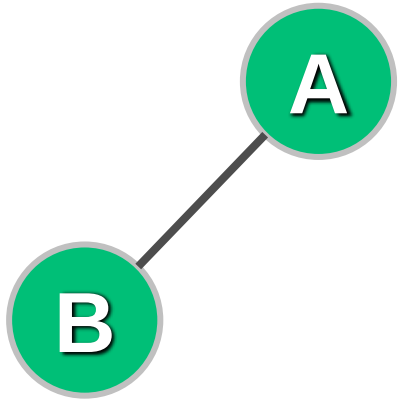
A

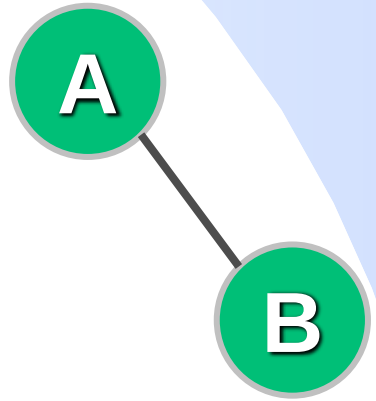
由先根序列和层次遍历序列可否唯一确定一棵二叉树？

- 先根序列：AB
- 层次序列：AB

由后根序列和层次遍历序列可否唯一确定一棵二叉树？

- 后根序列：BA
- 层次序列：AB



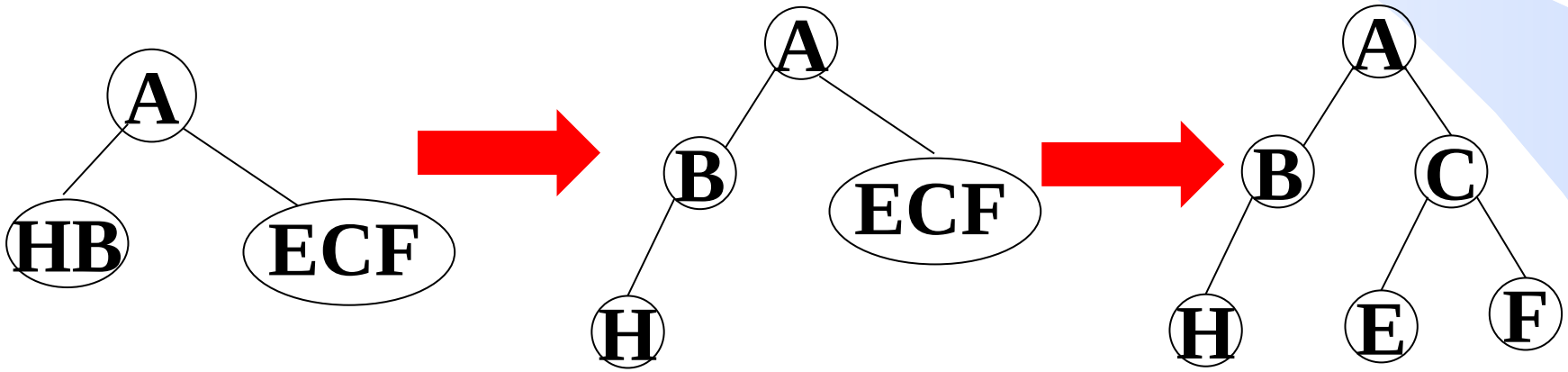


B

由中根序列和层次遍历序列可否唯一确定一棵二叉树？【清华大学考研题、吉林大学18级期末考试题】

中根序列　H B A E C F

层次序列　A B C H E F



中根序列和任意一种遍历序列都可以唯一地确定一棵二叉树

课下思考

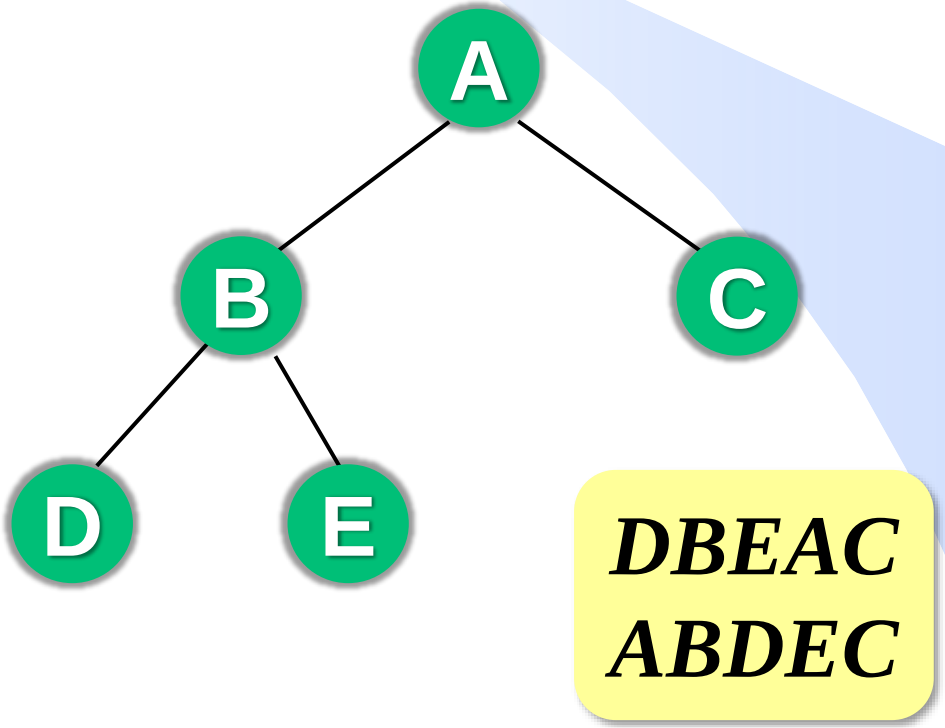
B

- 利用完全二叉树的先根序列能否唯一确定一棵完全二叉树?
- 利用完全二叉树的中根序列能否唯一确定一棵完全二叉树?
- 利用完全二叉树的后根序列能否唯一确定一棵完全二叉树?
- 利用完全二叉树的层次序列能否唯一确定一棵完全二叉树?

二叉树的序列化与反序列化

C

- 序列化：将数据结构转换为一个连续的序列，以便于存储和传输。
- 反序列化：将序列化的数据恢复、重建为原始的数据结构。
- 二叉树的序列化方法：
  - ✓中根序列+先根序列
  - ✓中根序列+后根序列
  - ✓中根序列+层次序列
  - ✓带空指针信息的先根序列
  - ✓……



自愿性质OJ练习题

- ✓ LeetCode 101（二叉树基础）
- ✓ LeetCode 654（二叉树基础）
- ✓ POJ 2499（二叉树基础）
- ✓ LeetCode 236（二叉树最近公共祖先）
- ✓ LeetCode 872（二叉树基础）
- ✓ LeetCode 543（二叉树基础）
- ✓ LeetCode 863（二叉树基础）
- ✓ HDU 1622（二叉树层次遍历）