

二叉树

吉林大学计算机学院
谷方明

fmgu2002@sina.com



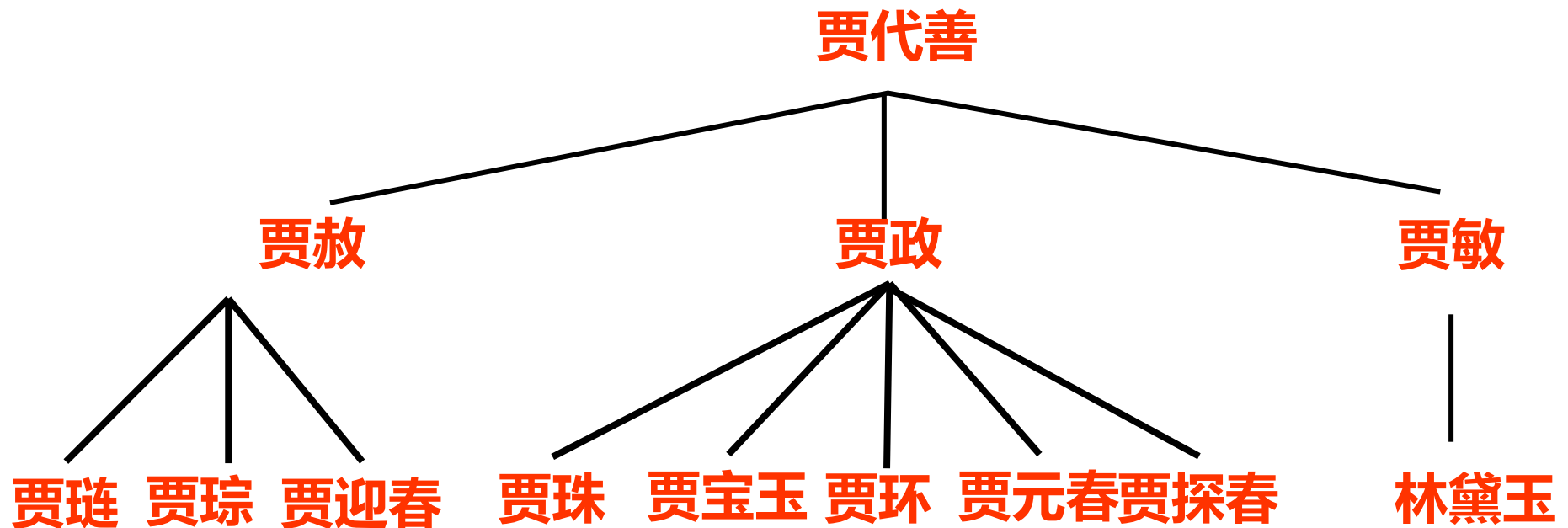
学习目标

- 掌握树的定义、术语和表示
- 掌握二叉树的定义
- 掌握二叉树的顺序存储和链接存储
- 掌握二叉树的遍历操作
- 掌握二叉树的创建等操作

5.1 树的基本概念

- 树是一种非常重要的非线性数据结构。可用来描述客观世界中广泛存在的具有分支或层次关系的对象。

例1 族谱(直系图)



例2 组织结构



例3 计算机领域的广泛应用



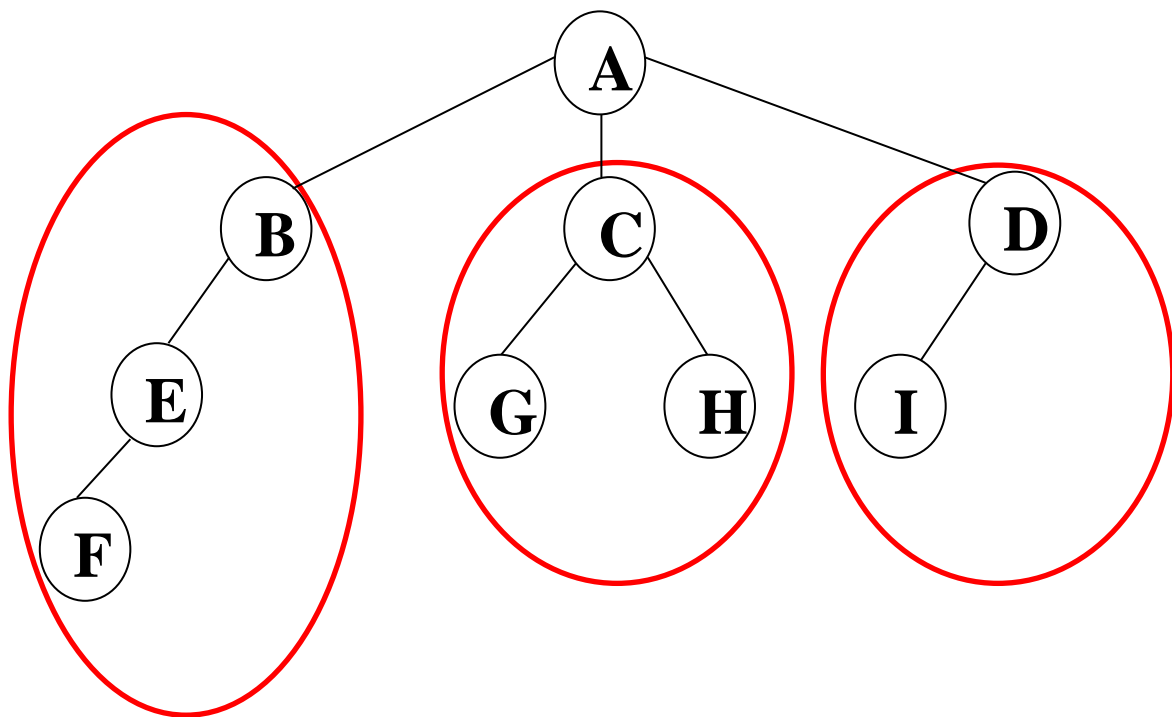
树的定义

定义5.1: 一棵树是一个有限的结点集合 T . 若 T 空, 则称为空树。若 T 非空, 则:

1. 有一个被称为根的结点, 记为 $root(T)$;
2. 其余结点被分成 $m(m \geq 0)$ 个不相交的非空集合 T_1, T_2, \dots, T_m , 且 T_1, T_2, \dots, T_m 又都是树。树 T_1, T_2, \dots, T_m 称作 $root(T)$ 的子树。

例4

Φ



(a)空树

(b)单个结点的树

(c)多个结点的树

有序树

- 如果树的定义(2)中，子树 T_1, T_2, \dots, T_m 的相对次序是重要的，则称该树为有序树，否则称为无序树。在有序树中，把 T_i 称作根的第*i*个子树。

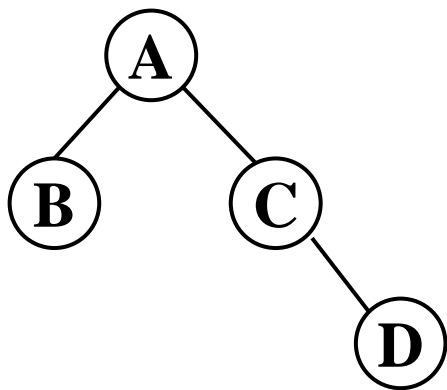


图5.2 (a) 树1

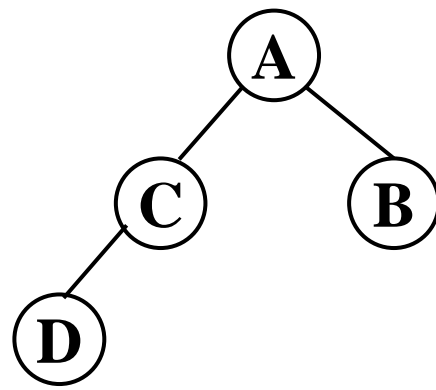
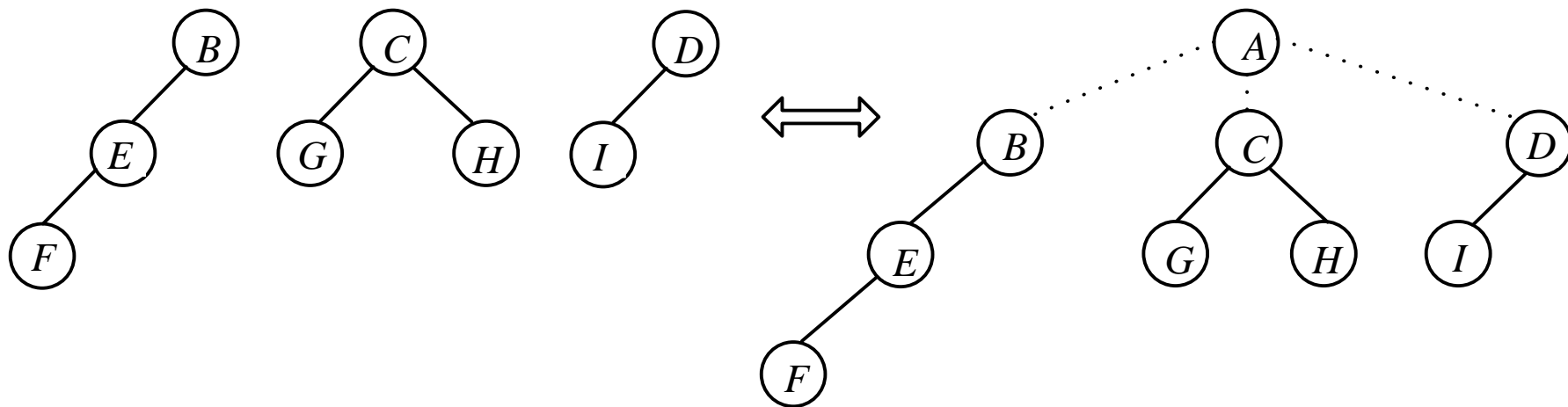


图5.2 (b) 树2

- 因为计算机表示定义了树的一种隐含次序，所以假定所讨论的树都是有序的。

森林

- 一个森林是若干个非空不相交的树的集合
- 抽象的森林和树的差别很小。如果删掉一个树的根，就得到一个森林；反过来，增加一个结点作为根结点到任何森林，就得到一个树。树和森林这两个词几乎可互换的使用。



树的定义说明

□ 本节定义的树在图论中称为有根树

- ✓ 有根树(**rooted tree**): 要求选出一个结点特别标识, 一般称为根(**root**)。
- ✓ 无根树(**unrooted tree**): 连通无环图, 也叫自由树(**free tree**)

□ 数据结构的树: 有限的带标号的有根有序 的树 (图论术语)

树的术语 I

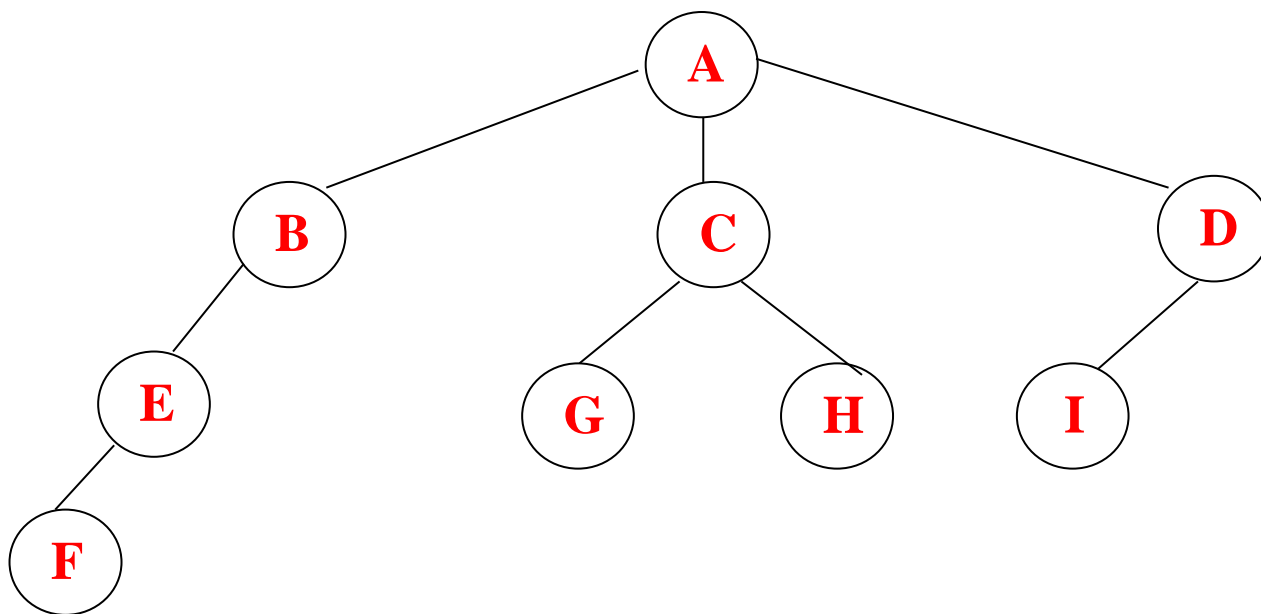
父亲、儿子、兄弟、祖先、后裔

1. 父亲、儿子、兄弟:

每个结点都是它的子树的根的父亲；反过来，每个结点都是它父亲的儿子；同一父亲的儿子是兄弟。

2. 祖先、后裔:

每个结点都是它的子树的所有结点的祖先；反过来，每个结点都是它祖先的后裔。



树的术语 II

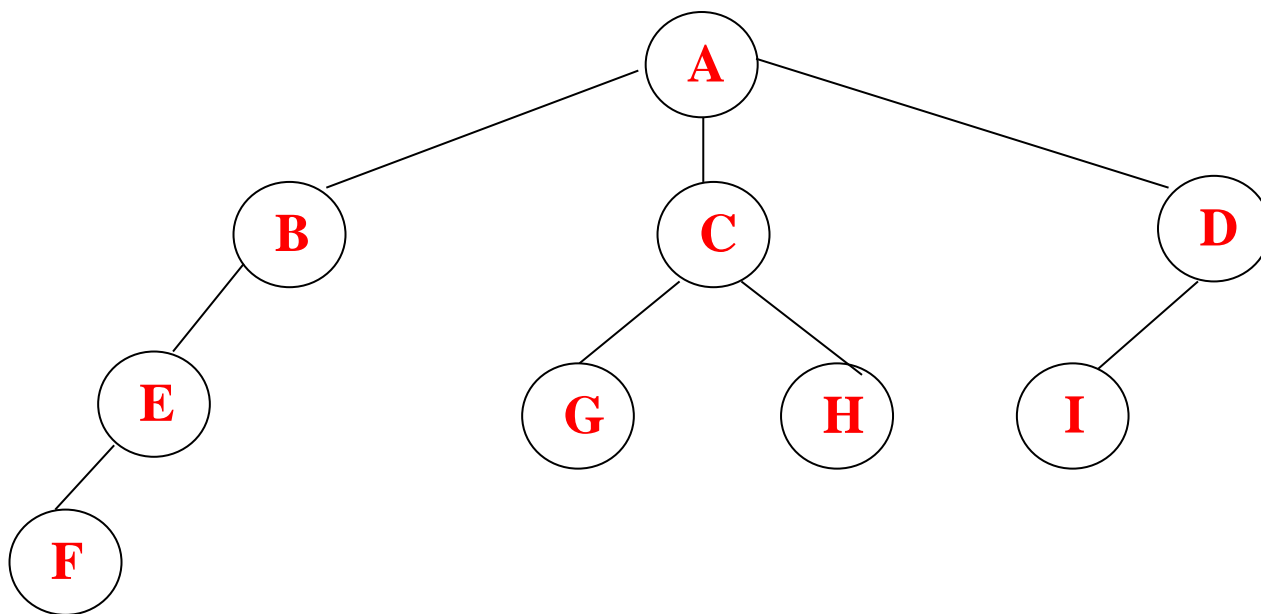
度、叶子、分枝

1. 度

一个结点的子树的个数，称为该结点的度或次数。
一棵树的度为 $\max_{i=1,\dots,n} D(i)$.

2. 叶结点、分支结点:

度为**0**的结点称为终端结点；非根的终端结点称为叶结点，有时不严格区分终端结点和叶结点；非终端结点称为分枝结点。



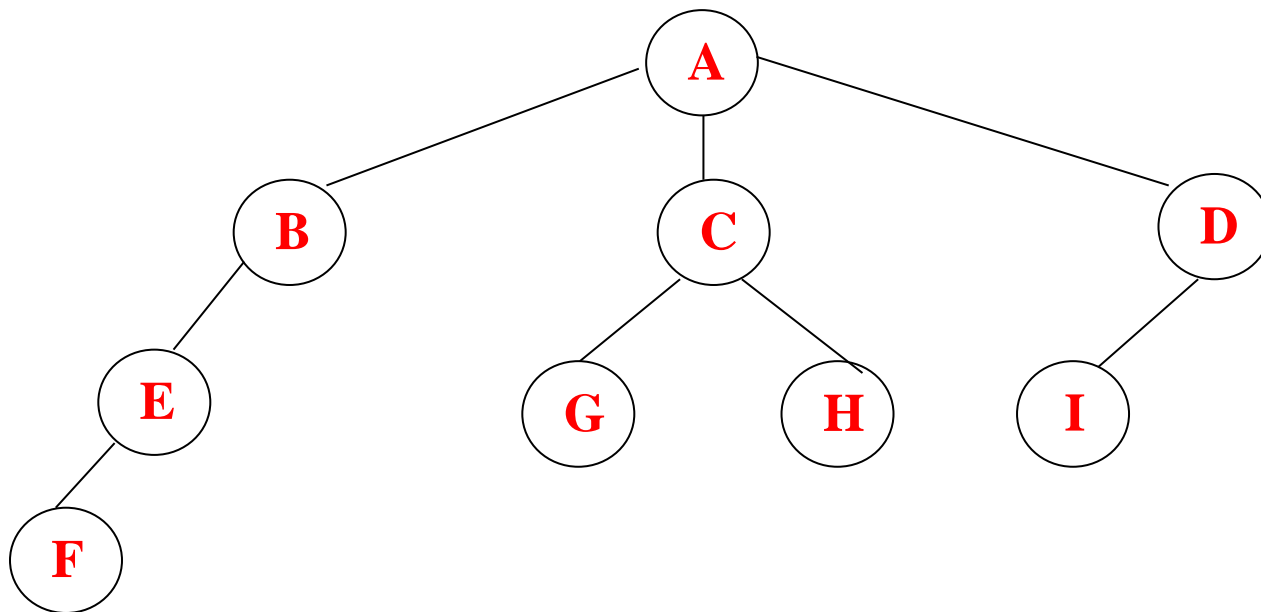
树的术语 III

结点的层数

1. 结点的层数

(1) $root(T)$ 层数为零;

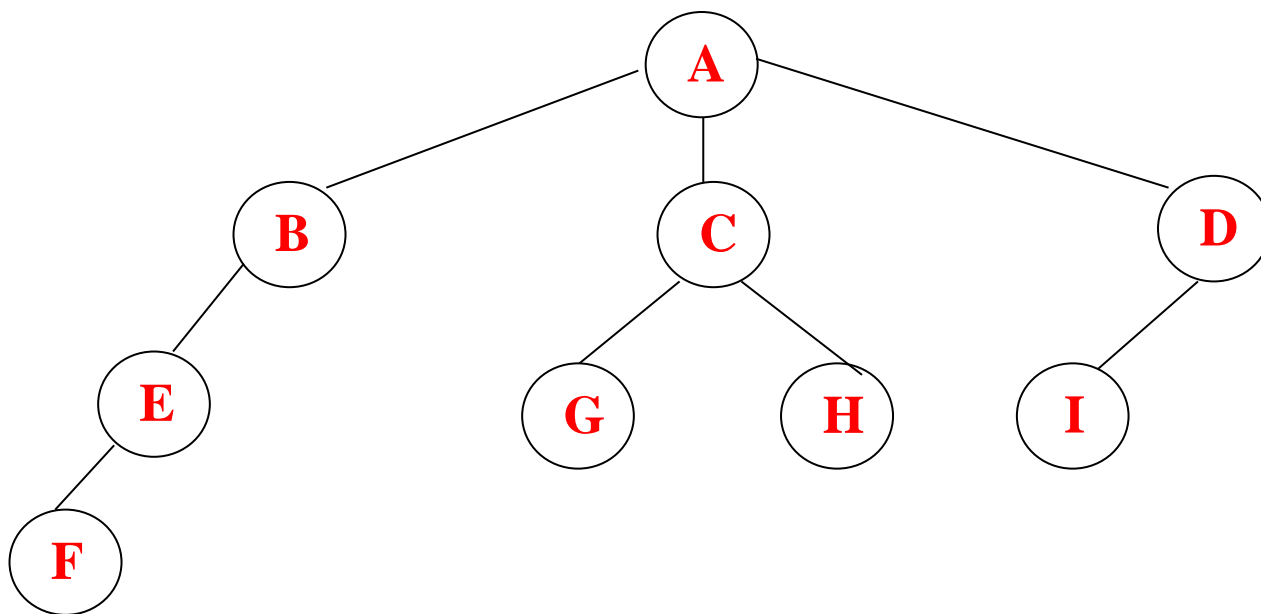
(2) 其余结点的层数为其父结点的层数加 1;



树的术语 IV

路径、深度、高度

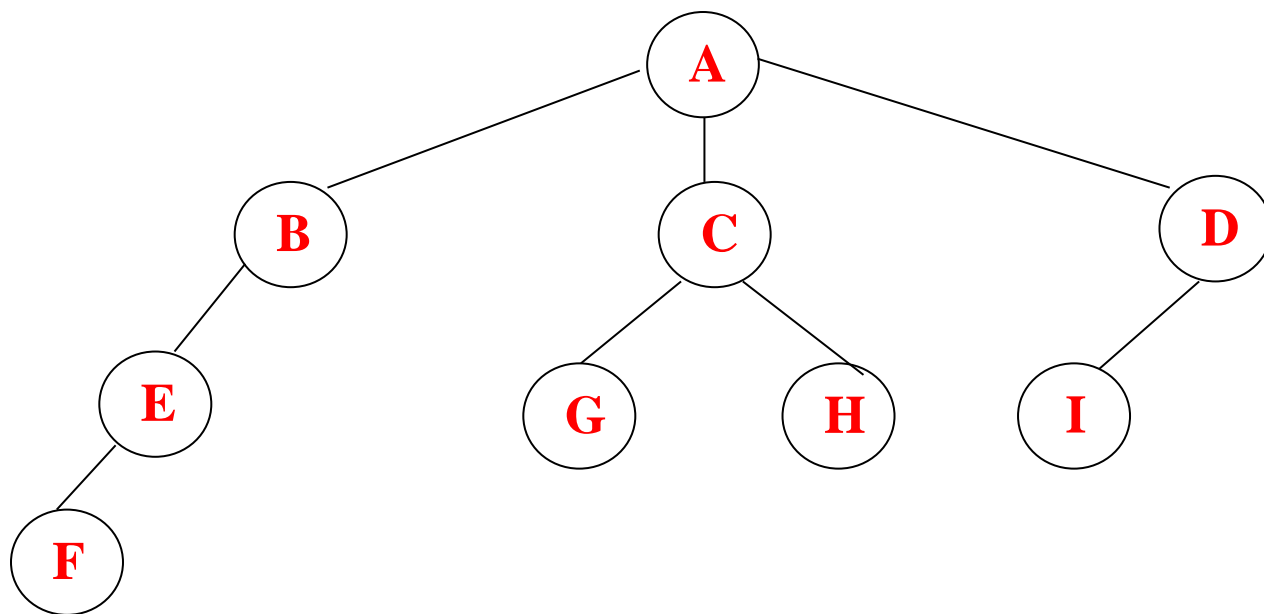
- 结点序列 v_1, v_2, \dots, v_k 称为结点 v_1 到结点 v_k 的路径，如果 v_i 是 v_{i+1} 的父亲 ($1 \leq i < k$)。路径经历的边数称为路径长度，即 $k-1$ 。
- 根到结点 v_i 的路径长度称为结点 v_i 的深度。设 n 为树的结点数， $D(i)$ 表树中第 i 个结点的深度，则一棵 树的深度 为 $\max_{i=1, \dots, n} D(i)$
- 结点 v_i 到叶结点的最长路径的长度称为结点 v_i 的高度。叶结点的高度为 **0**；一棵树的高度 定义为其根结点的高度。



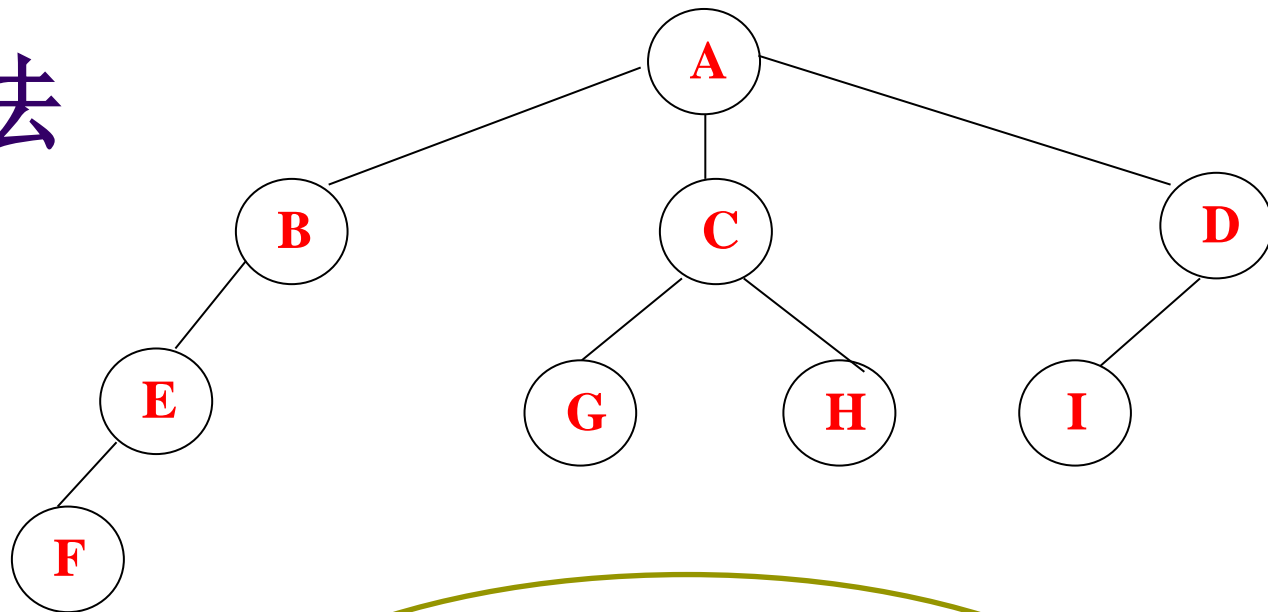
树的表示法

□ 树形表示法

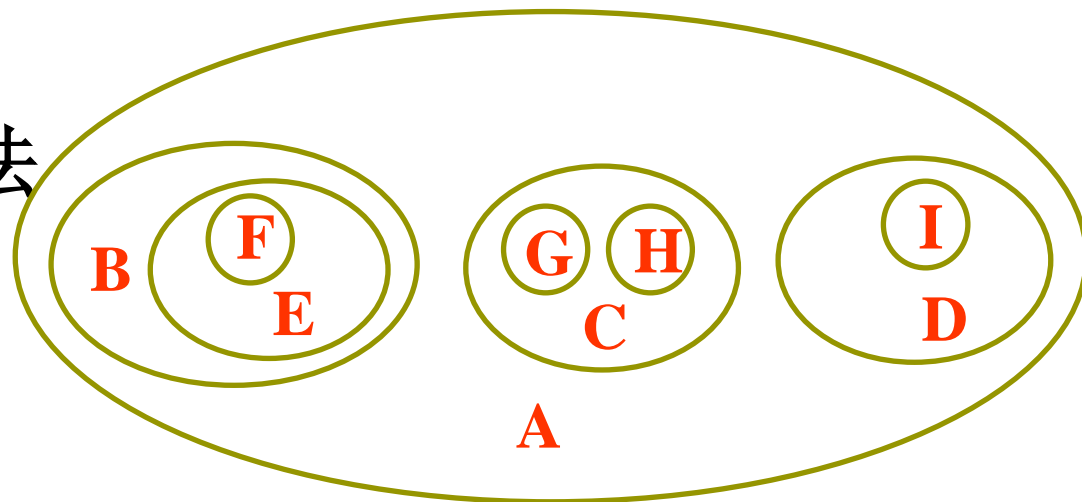
✓ 树根的位置



树的表示法

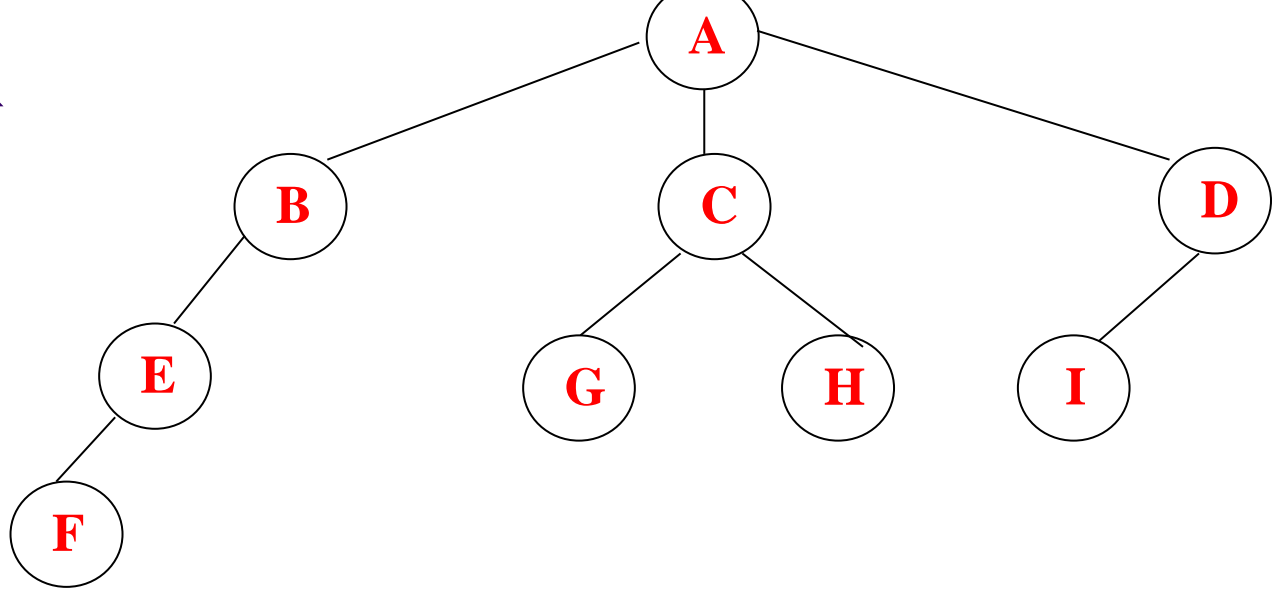


□ 嵌套集合表示法



$\{ A, \{ B, \{ E, \{ F \} \} \}, \{ C, \{ G \}, \{ H \} \}, \{ D, \{ I \} \} \}$

树的表示法

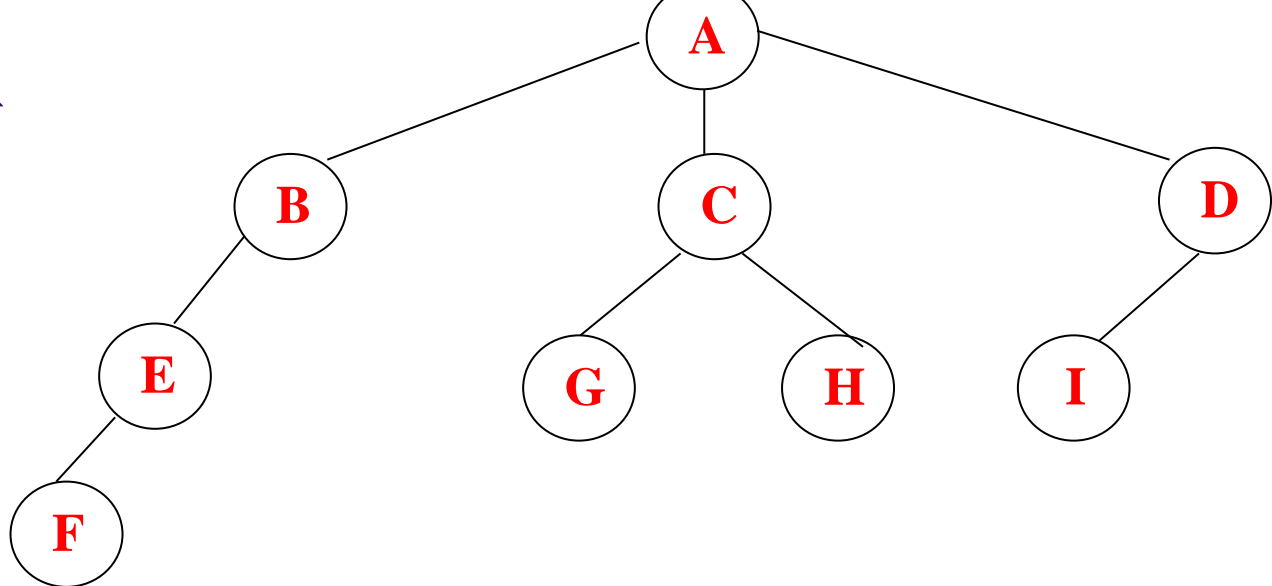


$\{ A, \{ B, \{ E, \{ F \} \} \}, \{ C, \{ G \}, \{ H \} \}, \{ D, \{ I \} \} \}$

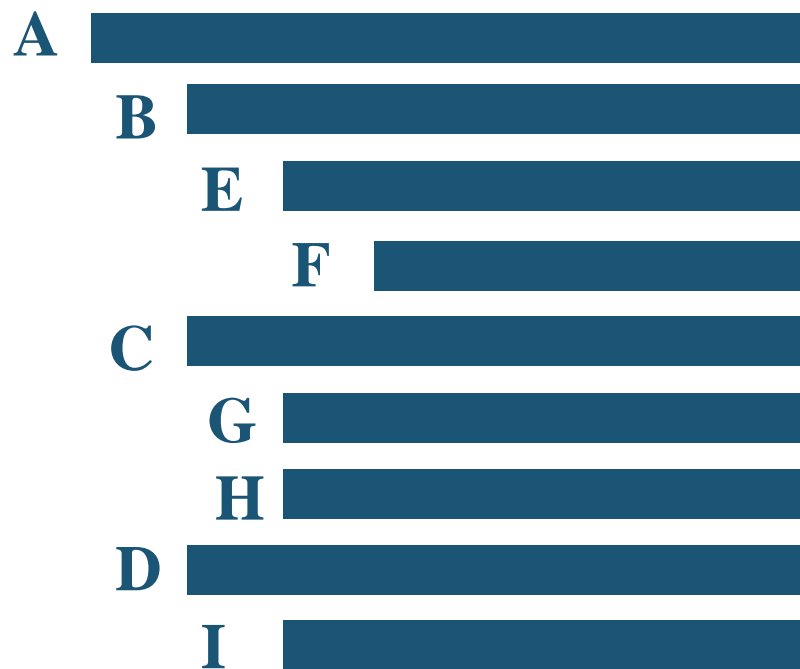
□ 嵌套括号表示法

✓ $(A (B(E(F)), C(G,H), D(I)))$

树的表示法



□ 凹入表示法



5.2 二叉树

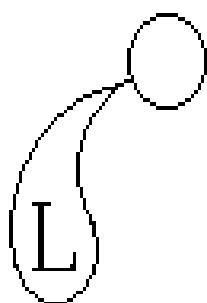
- **定义5.2:** 二叉树是结点的有限集合，它或者是空集，或者由一个根及左、右两个不相交的的二叉树构成。这两棵子树分别称为左、右子树。

\emptyset

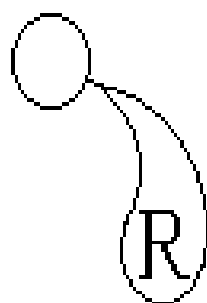
(a)



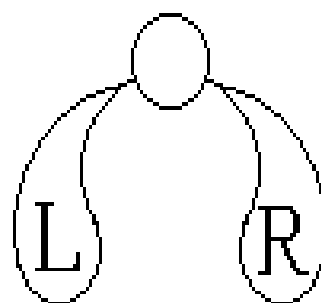
(b)



(c)



(d)

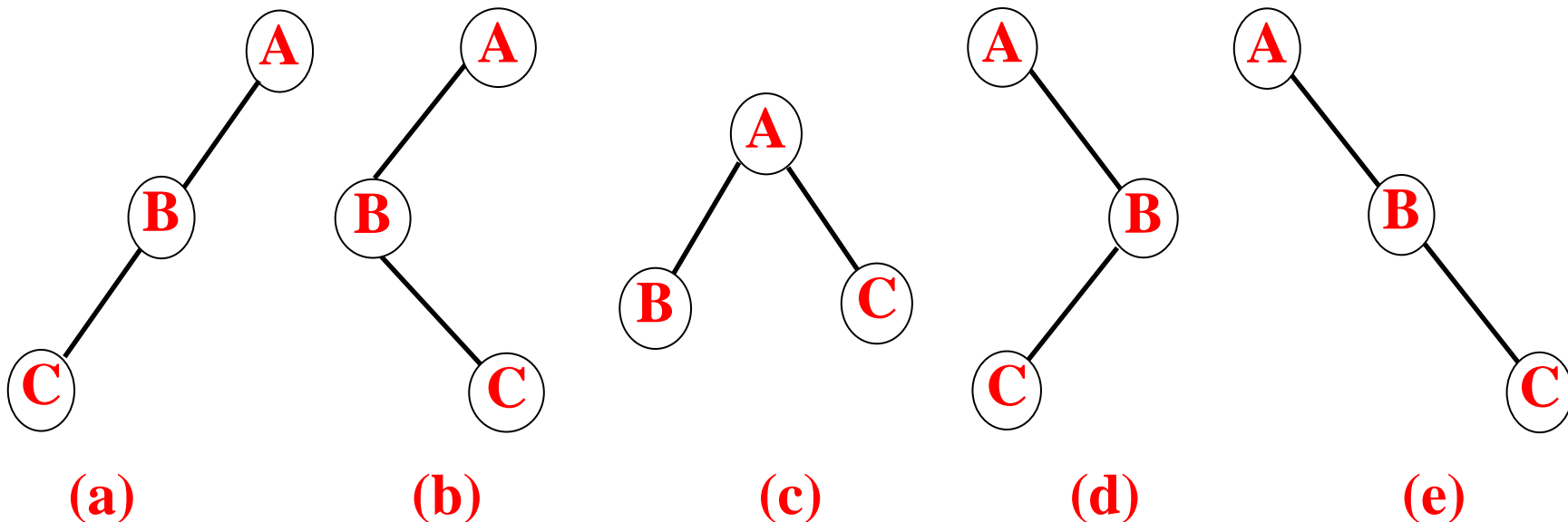


(e)

树与二叉树的主要区别

- I. 二叉树每个结点最多有 **2** 个子结点，树则无此限制；
- II. 二叉树中**结点的子树**分成左子树和右子树，即使某结点只有一棵子树，也要指明该子树是左子树，还是右子树，即**二叉树不仅是有序的**，而且是拓扑分叉的

例 含有3个结点的不同二叉树



含有3个结点的不同二叉树

引理5.1

二叉树中层数为 i 的结点至多有 2^i 个， $i \geq 0$ 。

引理5.2

高度为 k 的二叉树中至多有 $2^{k+1}-1$ ($k \geq 0$)个结点。

引理5.3

- 设T是由n个结点构成的二叉树，其中叶子结点个数为 n_0 ，度为2的结点个数为 n_2 ，则有： $n_0 = n_2 + 1$

证明： 设度为1的结点有 n_1 个，总结点个数为 n ，总边数为 e ，则：

$$n=n_0+n_1+n_2 \quad (1)$$

$$e=n-1 \quad (2)$$

$$e=2n_2+n_1 \quad (3)$$

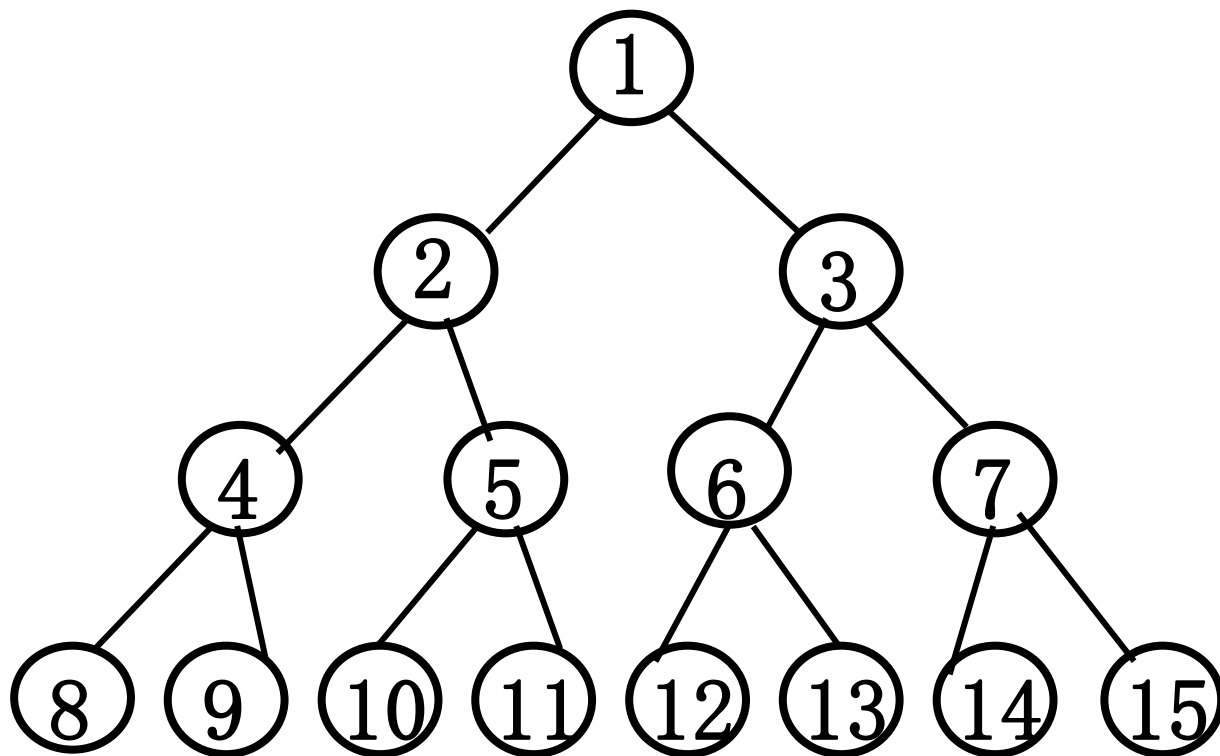
因此，有 $n_0+n_1+n_2-1=2n_2+n_1$

$$n_0=n_2+1 \quad .$$

证毕

满二叉树

- **定义5.3** 一棵非空高度为 k ($k \geq 0$)的满二叉树, 是有 $2^{k+1}-1$ 个结点的二叉树。

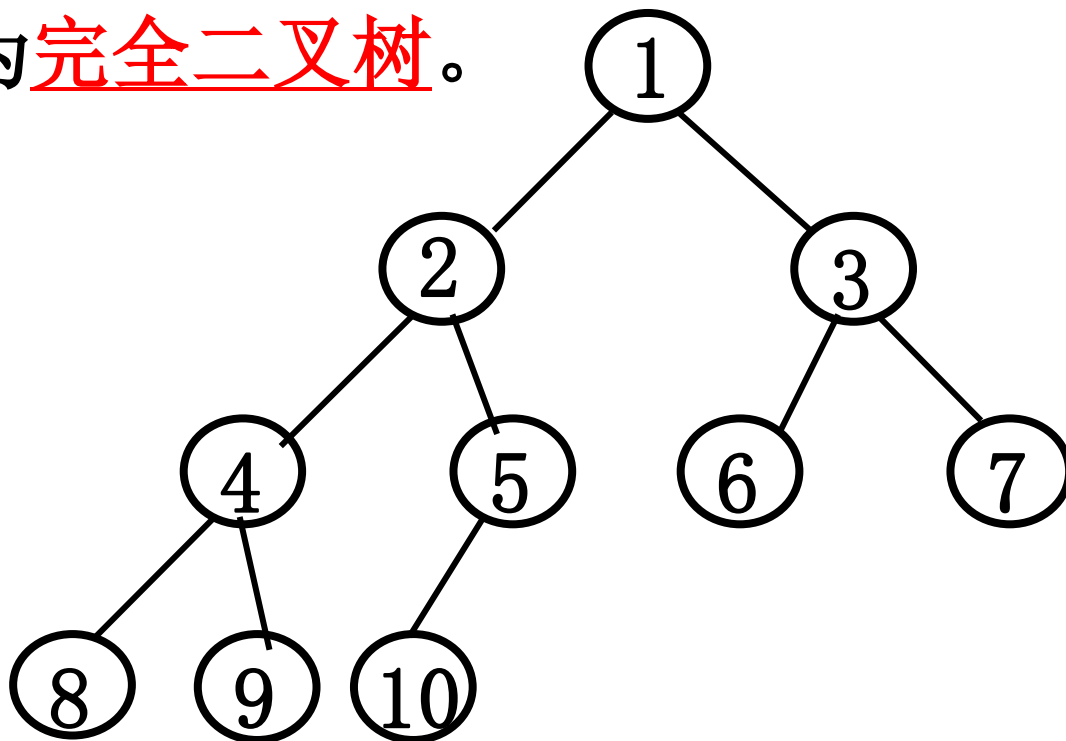


满二叉树的特点

- ① 叶结点都在第 k 层上;
- ② 每个分支结点都有两个子结点;
- ③ 叶结点的个数等于非叶结点个数加1 ;

完全二叉树

- **定义5.4** 一棵包含 n 个结点高为 k 的二叉树 T ，当按**层次顺序**编号 T 的所有结点，对应于一棵高为 k 的满二叉树中编号由1至 n 的那些结点时， T 被称为**完全二叉树**。



完全二叉树的特点

- 树中叶结点只能在层数最大的两层上出现；树中最下一层的结点都集中在该层最左边的若干位置上；
- 仅仅编号最大的分支结点可以没有右孩子，其余分支结点都有两个孩子；

引理5.4

- 若将一棵具有 n 个结点的**完全二叉树**按层次次序从1开始编号，则对编号为 i ($1 \leq i \leq n$)的结点有：
 - ① 若 $i \neq 1$ ，则编号为 i 的结点的父结点的编号为 $\lfloor i/2 \rfloor$ 。
 - ② 若 $2i \leq n$ ，则编号为 i 的结点的左孩子的编号为 $2i$ ，否则 i 无左孩子。
 - ③ 若 $2i+1 \leq n$ ，则 i 结点的右孩子结点编号为 $2i+1$ ，否则 i 无右孩子。

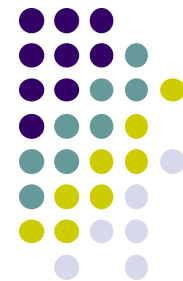
引理5.5

□ 具有 n 个结点的完全二叉树的高度是 $\lfloor \log_2 n \rfloor$

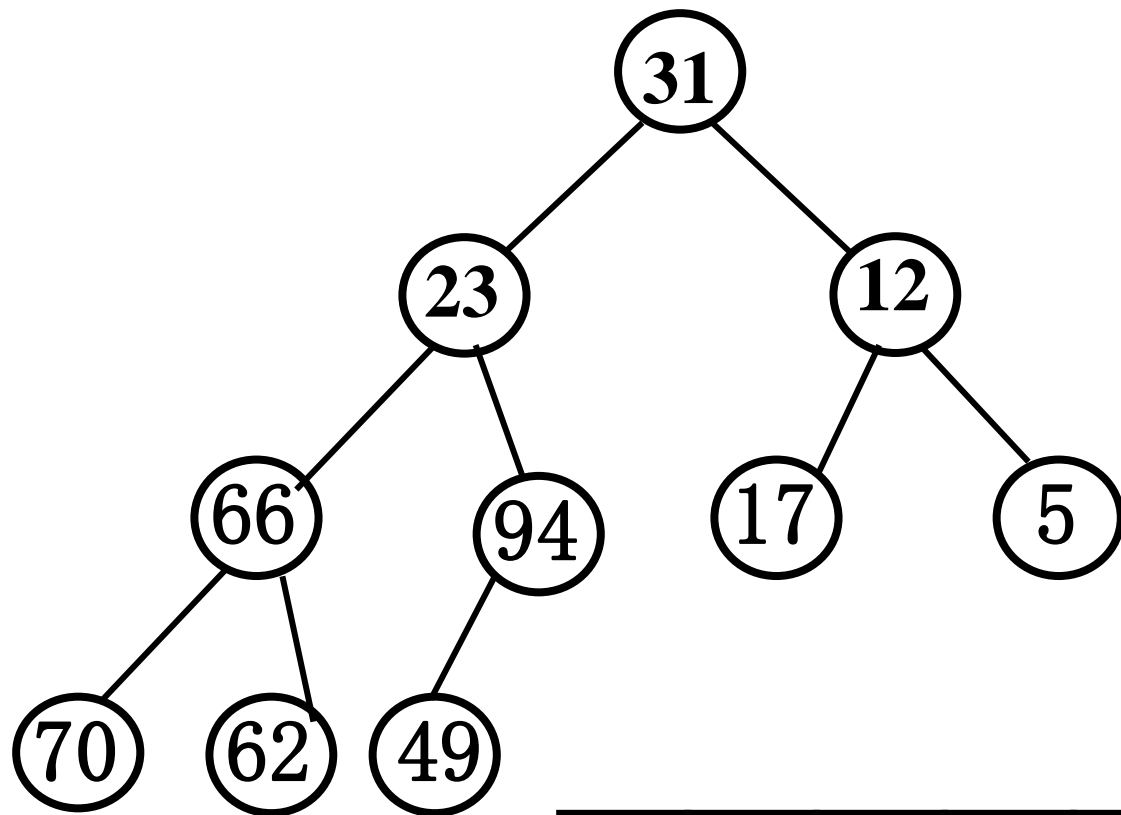


2 二叉树的顺序存储

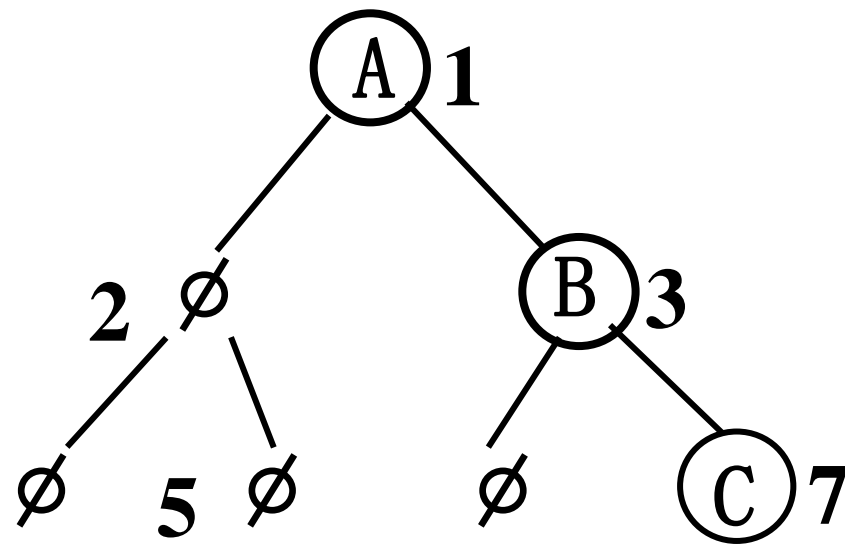
- 将二叉树中所有结点存放在一块地址连续的存储空间中，同时**反映出二叉树中结点间的逻辑关系**。



完全二叉树的顺序存储



31	23	12	66	94	17	5	70	62	49
----	----	----	----	----	----	---	----	----	----



a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
A	∅	B	∅	∅	∅	C

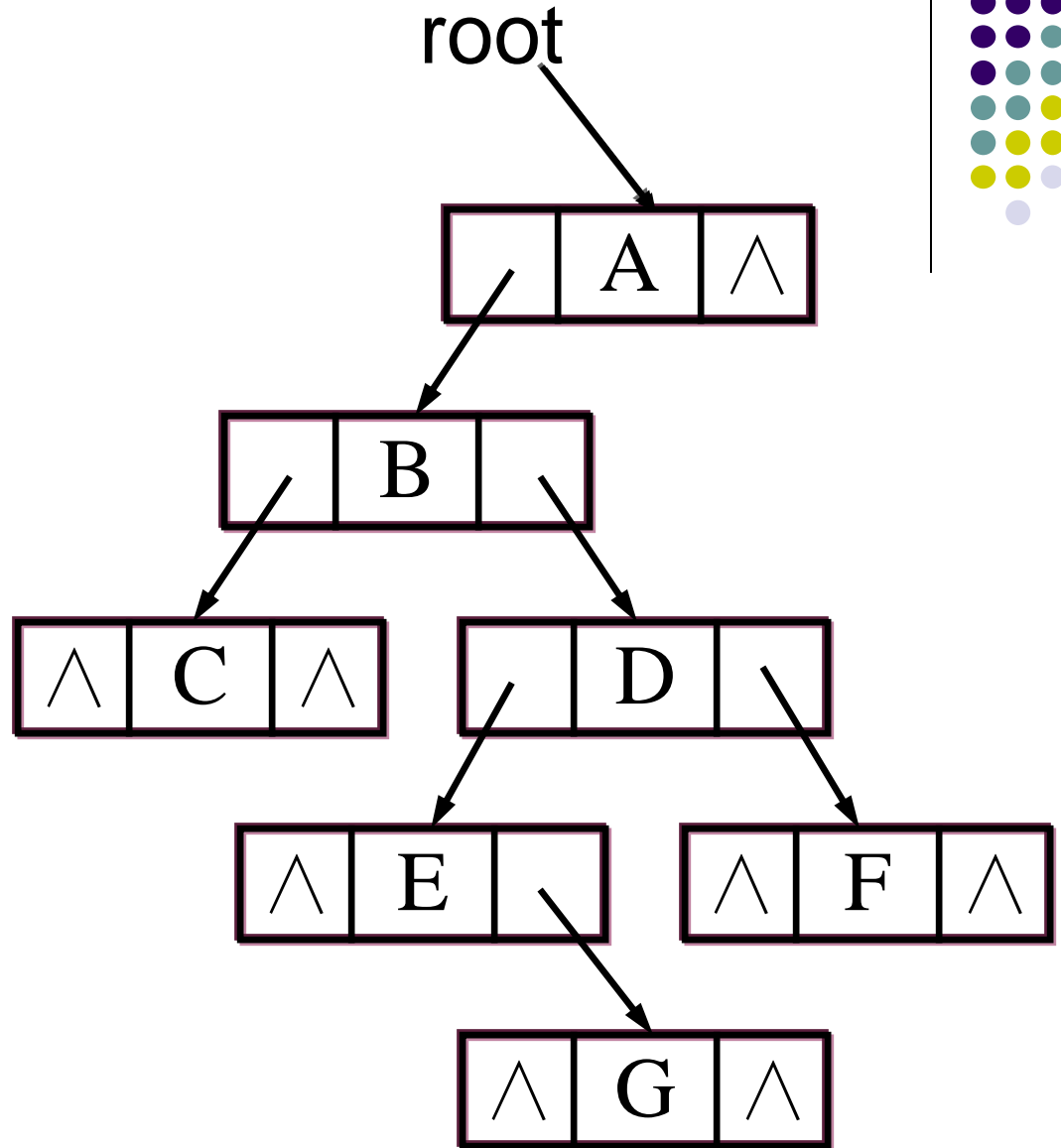
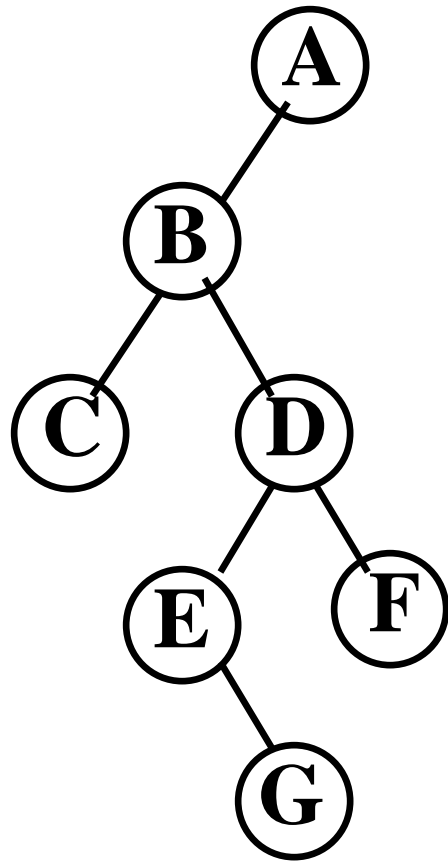
- 一般二叉树也可仿照完全二叉树那样存储。但可能会浪费很多存储空间。



二叉树的链接存储

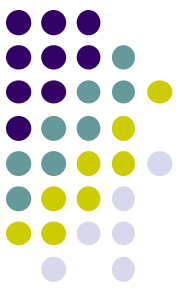
- 二叉树诸结点被随机存放在内存空间中，结点之间的关系用指针说明。
- 二叉树的结点结构包含三个域：数据域**data**、指针域**left**和指针域**right**，其中左、右指针分别指向该结点的左、右孩子结点。





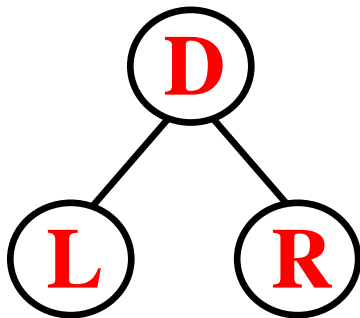


- 在二叉树的链接存储中，有一个指向根结点的指针，称为根指针。若二叉树为空，根指针为 **NULL**.
- 空指针域个数： **$2n - (n-1)$**



二叉树的遍历

- **二叉树的遍历**：按照一定次序访问树中所有结点，且使每个结点恰好被访问一次。



遍历方式

先根遍历：DLR

中根遍历：LDR

后根遍历：LRD

- **先根(中根、后根)序列**：以先根(中根、后根)次序遍历二叉树 T, 得到 T 之结点的一个序列

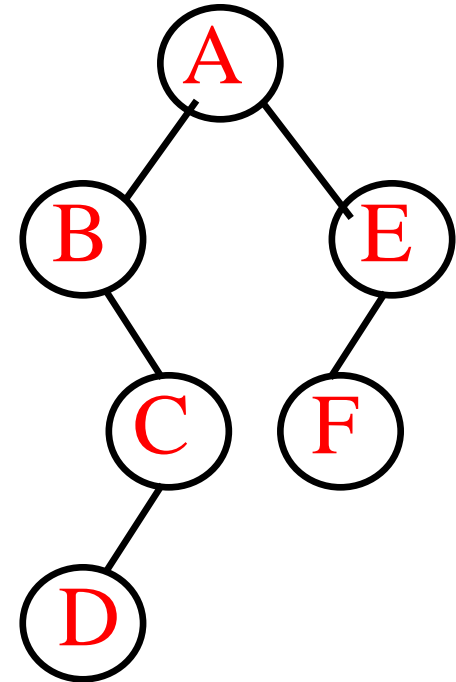


先根遍历 (前/先序遍历)

- 若二叉树为空，则返回空；
- 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

ABCDEF





先根遍历算法

算法PreOrder(t)

/* 先根遍历 t 指向的树*/

P1 [递归遍历]

```
if (t != NULL) {  
    printf("%d\n", t->data );  
    PreOrder(t->left);  
    PreOrder(t->right);  
}
```

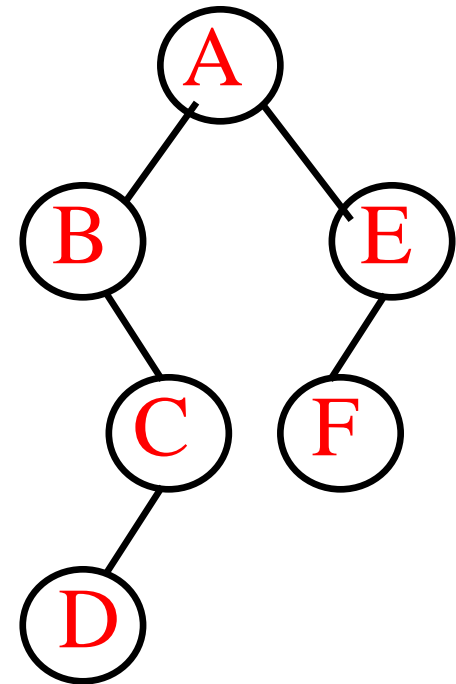


中根遍历(中序遍历)

- 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

BDCAFE





中根遍历算法

算法InOrder(t)

/* 中根遍历 t 指向的树*/

I1 [递归遍历]

```
if (t != NULL) {  
    InOrder(t->left);  
    printf("%d\n", t->data );  
    InOrder(t->right);  
}
```

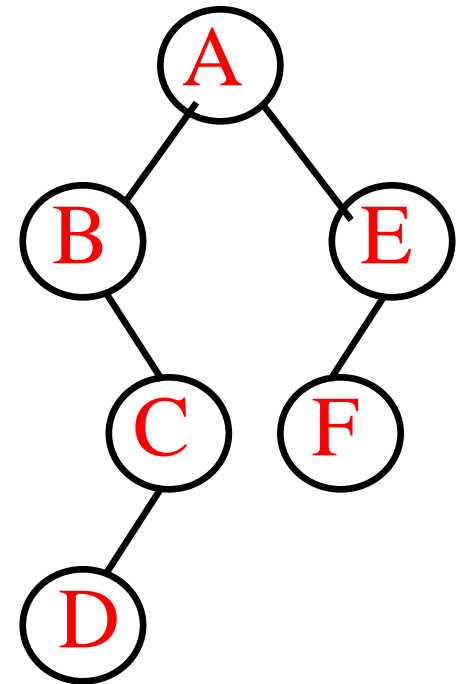


后根遍历 (后序遍历)

- 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

DCBF EA





后根遍历算法

算法PostOrder(t)

/* 后根遍历 t 指向的树*/

I1 [递归遍历]

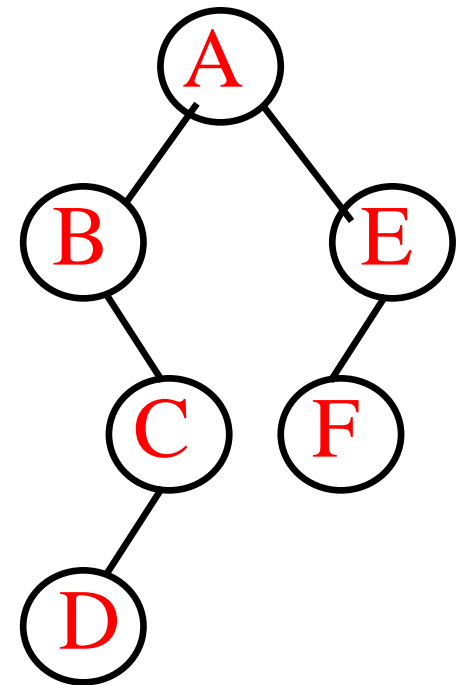
```
if (t != NULL) {  
    PostOrder(t->left);  
    PostOrder(t->right);  
    printf("%d\n", t->data );  
}
```




层次遍历

□ 层次遍历：按层数由小到大，即从第 0 层开始逐层向下，同层中由左到右的次序访问二叉树的所有结点。

□ 例：层次遍历序列为
ABECFD





层次遍历的实现

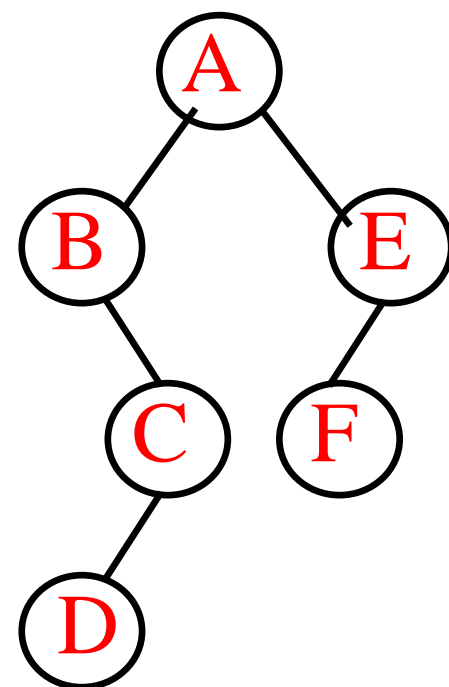
□ 需要一个队列辅助

1. 根结点入队。
2. 若队不空，重复本步骤：

取队头结点并访问；

若其左指针不空，将其左孩子入队；

若其右指针不空，将其右孩子入队。





算法LevelOrder (t)

LevelOrder1. [初始化]

CREATEQuene Q ;

$p = t$; if ($p \neq \text{NULL}$) $Q \Leftarrow p$;

LevelOrder 2. [层次遍历]

while (! $Q.empty()$) {

$p \Leftarrow Q$.

 printf("%d\n", $p \rightarrow \text{data}$);

 if ($p \rightarrow \text{left} \neq \text{NULL}$) $Q \Leftarrow p \rightarrow \text{left}$;

 if ($p \rightarrow \text{right} \neq \text{NULL}$) $Q \Leftarrow p \rightarrow \text{right}$; }



创建二叉树

- 先根序列不能唯一确定二叉树之结构，两棵不同的二叉树却可能有相同的先根序列。





带空指针的先根序列

- 用“#”表示空指针，对下图所示二叉树的先根序列是 $ABDEC$ ，加入‘#’表示空指针位置后变为 $ABD##E##C##$ 。对图4.15 (b)所示二叉树之先根序列 $ABDEC$ ，加入‘#’表示空指针位置后变为 $ABD#E####C##$ 。





- ▣ 递归算法 **CreateBinTree**（简记为**CBT**）
以包含空指针信息的先根序列为输入序列，
以根指针 t 为输出参数.
- ▣ 当读入 ‘#’ 字符时，将其初始化为一个空指针；否则生成一个新结点并初始化其父结点之左、右指针.



算法CBT (*tostop* . *t*)

/* 构造以结点*t*为根的二叉树； *tostop* = '#' */

CBT1. [读数据]

scanf("%c",&ch); /* 顺序读入序列中的一个符号 */

CBT2. [*ch* = *tostop*?]

if (*ch* == *tostop*) { *t* = Λ ; return; }

t \leftarrow **AVAIL** ; *t*->*data* = *ch* ;

CBT3. [构造左子树]

CBT (*tostop* , *t* ->*left*);

CBT4. [构造右子树]

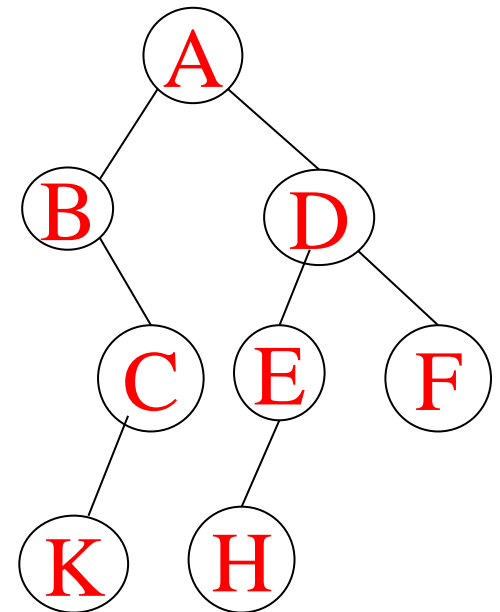
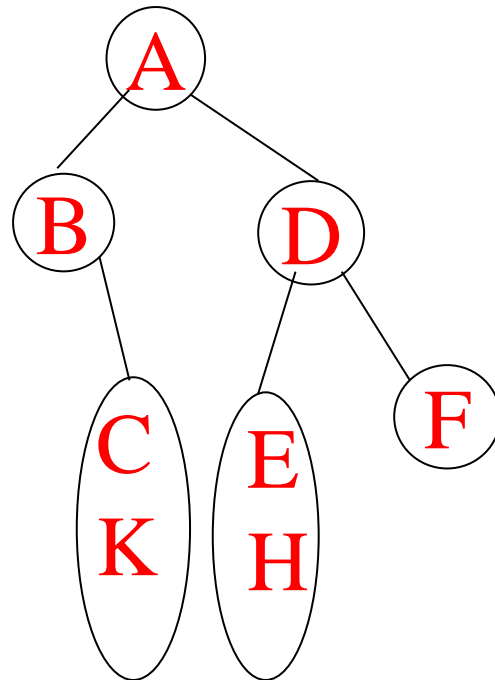
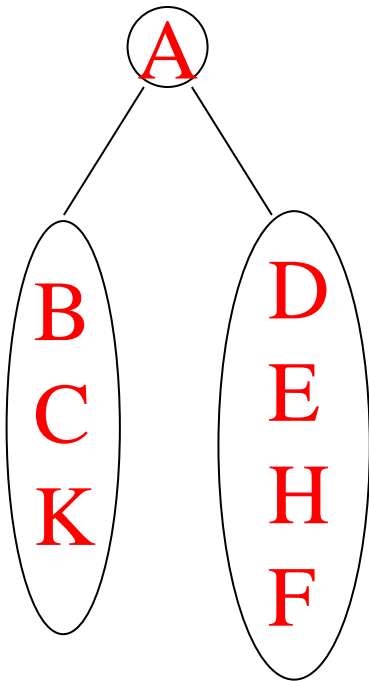
CBT (*tostop*, *t* -> *right*);



□ 先根序列和中根序列可以唯一确定一棵二叉树。

[例] 先根序列 **A** B C K D E H F

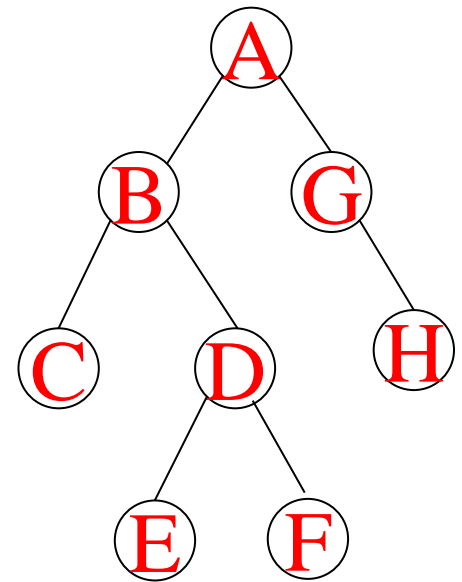
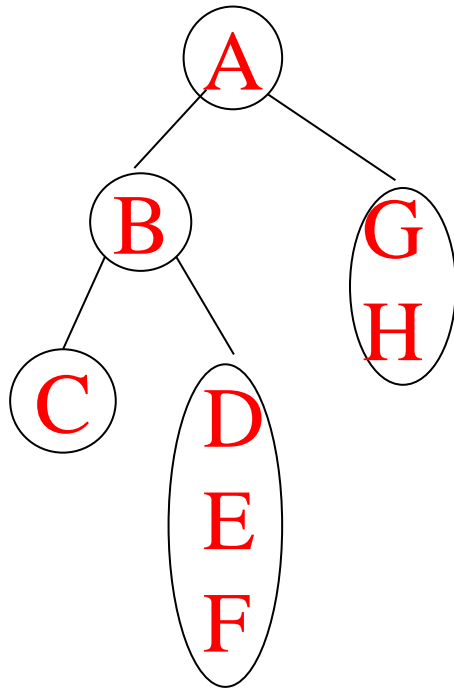
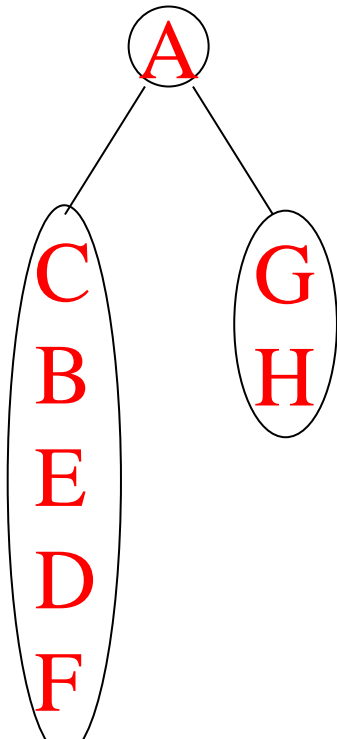
中根序列 B K C **A** H E D F





□ 后根序列和中根序列可以唯一确定一棵二叉树。

[例] 后根序列 C E F D B H G **A**
中根序列 C B E D F **A** G H





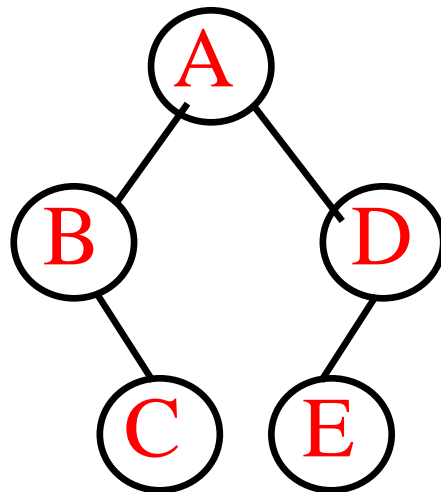
课后思考

1. 给定一棵二叉树 T 的先根序列和后根序列，那么能否由此确定出 T 之结构？
2. 尝试设计扩展先根序列序列表示树的其它方式



复制二叉树

- 可以按先根遍历、中根遍历或后根遍历的方式复制二叉树。以后根遍历为例进行复制。
- 复制过程：先复制子结点，再复制父结点，将父结点与子结点连接起来。





算法 CopyTree (t . p)

CopyTree1. [递归出口]

if ($t == \text{NULL}$) RETURN NULL;

CopyTree2. [复制左子树]

if ($t \rightarrow \text{left} \neq \text{NULL}$)

CopyTree($t \rightarrow \text{left}$, newlptr); else $\text{newlptr} = \text{NULL}$;

CopyTree3. [复制右子树]

if ($t \rightarrow \text{right} \neq \text{NULL}$)

CopyTree($t \rightarrow \text{right}$, newrptr); else $\text{newrptr} = \text{NULL}$;

CopyTree4. [生成结点]

$p \leftarrow \text{AVAIL}$.

$p \rightarrow \text{data} = t \rightarrow \text{data}$; $p \rightarrow \text{left} = \text{newlptr}$; $p \rightarrow \text{right} = \text{newrptr}$;

RETURN p ; ■



搜索父结点

算法Father (t, p . q)

F1 [判断t是否存在及p是否为根结点]

if (t==NULL || p==NULL || p==t) return q = NULL;

F2 [若t为所求]

if (t->Left(t)==p || t->Right(t)==p) return q = t;

F3 [递归调用]

Father(t->left, p , qL);

if (qL!=NULL) return q=qL ;

Father(t-> right, p . qR);

return q=qR; ■



三叉链

- 另一种常用的结点结构包括三个指针域，**parent**域中指针指向其父结点

left	data	parent	right
-------------	-------------	---------------	--------------



搜索数据

算法**Find**($t, item . q$)

Find1. [判断 t 是否为空或为所求]

if ($t == \text{NULL}$) return $q = \text{NULL}$;

if ($t \rightarrow \text{data} == \text{item}$) return $q = t$;

Find2. [递归]

Find ($t \rightarrow \text{left}, \text{item}, p$);

if($p \neq \text{NULL}$) return $q = p$;

Find ($t \rightarrow \text{right}, \text{item}, p$);

return $q = p$; ■



插入结点作为某结点的左儿子

算法 **InsertLeft** ($t, item, p$)

I1. [特判]

if ($t == NULL$) return ;

I2. [结点p]

Find($t, item, q$);

if($q == NULL$) return;

I3. [插入结点]

$p \rightarrow left = q \rightarrow left$;

$q \rightarrow left = p$; ■



释放二叉树

算法Del(p)

/* 删除结点 p 及其左右子树 */

Del1. [递归删除]

```
    if(  $p \neq \text{NULL}$  ) {  
        Del(  $p \rightarrow \text{left}$  );  
        Del(  $p \rightarrow \text{right}$  );  
        AVAIL  $\leftarrow p$  ;  
    } ■
```



删除给定结点及其左右子树

算法**DST** (t)

DST1. [特判]

if ($t == \text{NULL}$) return ;

if ($t == \text{root}$) { Del(t) ; $\text{root} = \text{NULL}$; return ; }

DST2. [找 t 的父结点 q]

$p = t$; Father(root , p , q).

DST3. [修改 q 的指针域]

if ($q \neq \text{NULL} \ \&\& \ q \rightarrow \text{left} == p$) $q \rightarrow \text{left} = \text{NULL}$;

if ($q \neq \text{NULL} \ \&\& \ q \rightarrow \text{right} == p$) $q \rightarrow \text{right} = \text{NULL}$;

DST4. [删除 p 及其子树]

Del(p);



非递归的中根遍历算法

算法NIO(t)

NIO1. [初始化]

CREATEStack(S) ; $p = t$.

NIO2. [入栈]

while($p \neq \text{NULL}$) { $S.\text{push}(p)$; $p = p \rightarrow \text{Left}$; }

NIO3. [栈为空?]

if ($S.\text{empty}()$) return; else $p = S.\text{pop}()$;

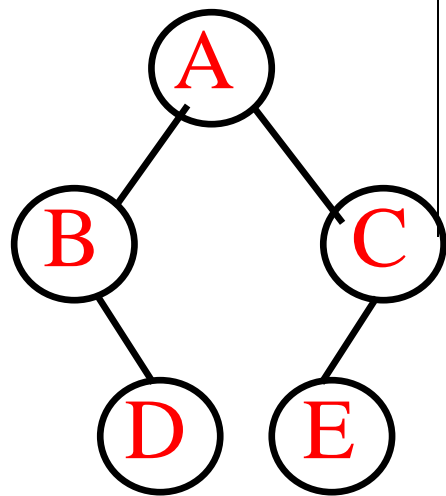
NIO4. [访问 p , 更新 p]

printf(“%c ”, $p \rightarrow \text{Data}$) ; $p = p \rightarrow \text{Right}$;

NIO5. [返回]

goto NIO2 ; ■

运行及证明



□ 定理5.1：正确性证明

设算法NIO从步骤NIO2开始， p 指向一棵有 n 个结点之二叉树 T^* 的根，此时栈 S 中有 $S[1]$, $S[2]$, ..., $S[m]$, $m \geq 0$ ，则步骤NIO2至NIO5将以中根序遍历 T^* ，并最后到达步骤NIO3，同时栈 S 也恢复到原来值。



非递归的后根遍历算法

- 先根和中根遍历的非递归算法，一个结点仅进栈出栈一次，我们能够判断其输出语句的位置，分别为结点进栈前及出栈后。
- 而后根遍历输出结点的位置应为处理完右子树之后，如果每个结点还是进栈、出栈一次，则无法确定何时输出结点，即其左右子树是否已处理完。



工作栈结点

结点	结点状态 i
----	----------

结点所处状态 $i = 0, 1$ 或 2 :

0: 结点及左右子树均未被访问;

1: 遍历左子树;

2: 遍历右子树。

- 树中任一结点 q 都需进栈三次，出栈三次。第一次出栈是为遍历结点 q 的左子树，第二次出栈是为遍历结点 q 的右子树，第三次出栈是为访问结点 q 。

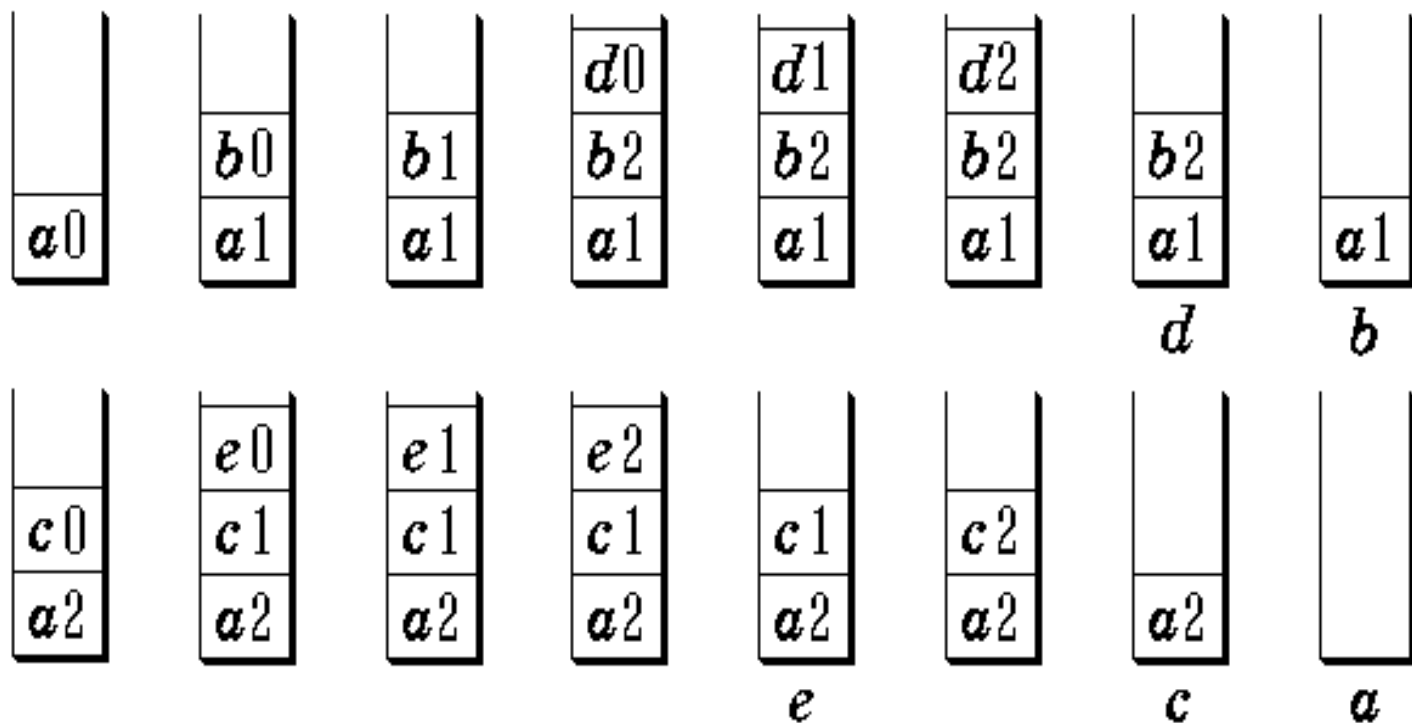
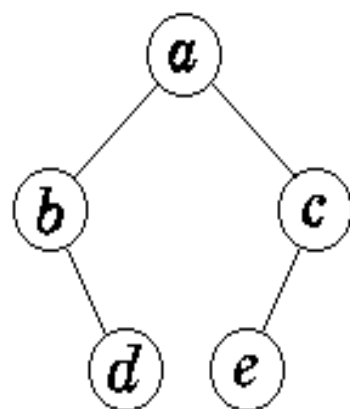


算法思想

1) 将(根结点,0)压入堆栈。

2) 弹栈，对出栈元素(p, i)中标号 i 进行判断，

- ✓ 若 $i=0$ ，则将($p,1$)压入堆栈；若结点 p 的左指针不为空，则将($Left(p), 0$)压入堆栈，准备遍历其左子树。
- ✓ 若 $i=1$ ，此时已遍历完结点 p 的左子树，则将($p,2$)压入堆栈；若右指针不为空，则将($Right(p), 0$)压入堆栈，准备遍历其右子树。
- ✓ 若 $i=2$ ，此时已遍历完结点 p 的右子树，访问结点 p 。





算法NPostOrder(t)

NPO1[建立堆栈]

CREATStack(S); $S \leftarrow (t, 0)$;

NPO3[利用栈实现递归]

while(! s.empty()) {

(P,i) \leftarrow S;

if (i==0) { S \leftarrow (P,1);

if (p->Left != NULL) S \leftarrow (p->Left,0) ; }

if (i==1) { S \leftarrow (P,2);

if (p->Right != NULL) S \leftarrow (p->Right,0) ; }

if (i==2) printf(“%c ”, p->Data) ;

}



- 非递归的**先根遍历**算法与非递归遍历类似;
- 非递归的后根遍历有多种实现方案



总结

- 树的定义、术语（**4组**）和表示法（**4种**）
- 二叉树的定义和性质（**5个引理**）
- 二叉树的存储（顺序、链接）
- 二叉树的操作（遍历、创建、.....）
- 遍历的迭代算法（中根、后根、先根）



第5章 任务

□ 慕课

- ✓ 在线学习/预习 第 5 章 视频
- ✓ 在线学习/预习 7.4.2 堆排序

□ 作业

- ✓ P162: 5-2, 5-3, 5-4, 5-5, 5-6,
5-10, 5-12, 5-14
- ✓ 在线提交