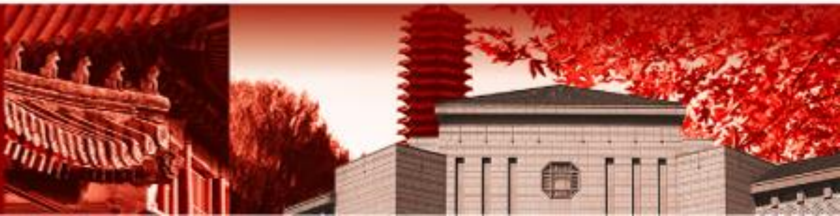
数据结构与算法B

09-树和森林

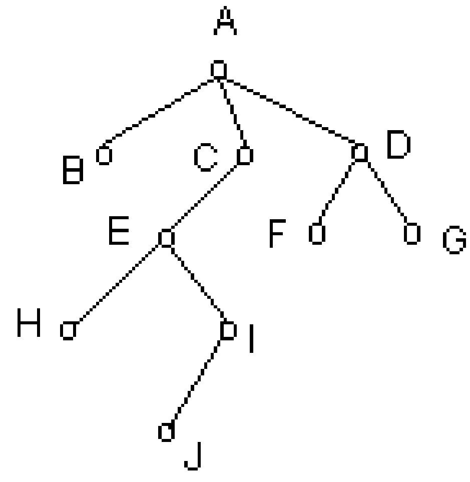


目录

9.1 树和森林的定义

树的定义

- 树的递归定义如下：
  - (1) 树是包含有限结点的非空集合，其中有且仅有一个根结点
  - (2) 仅包含一个结点的集合是一棵树，根就是该结点
  - (3) 在树中，根以外的其他结点，被分成若干个不相交的子集，每个子集都是一棵树，称为子树 (subtree)
- 特别地，允许不包括任何结点的树，把它称作空树。
  - 右图所示树的三棵子树分别为
  - T_1={B}，T_2={C，E，H，I，J}，T_3={D，F，G}



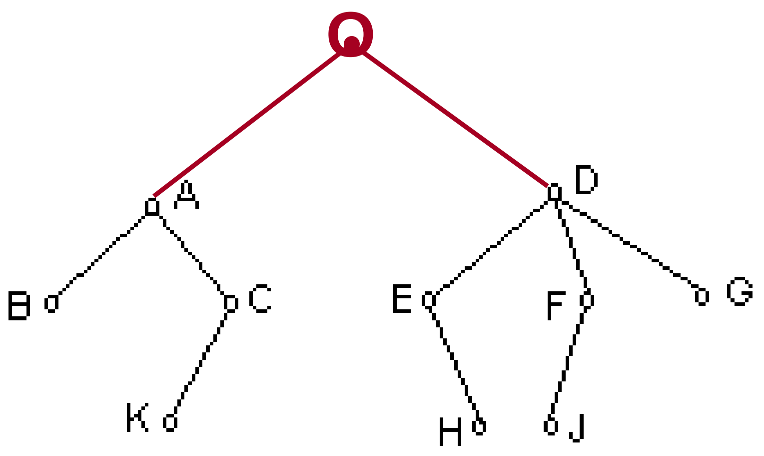
北京大學

树 & 二叉树

- 树中的边、高度、深度、度数、父子结点、层次等概念，和二叉树中的类似。
- 有序树与无序树
  - 树通常是有序的，即有第1子树、第2子树、……、第n子树之分
  - 如果删除第1子树，第2子树就顶替成为第1子树
  - 如果规定树的各个子树之间不存在顺序，则称该树是无序树
- 注意：二叉树并不是树
  - 不能将二叉树看作度数为2的树
  - 二叉树区分左子树和右子树。
  - 只有一棵子树的树，无法区分这棵子树是左子树或者右子树

森林的定义

- 森林是空集或者不相交的树的集合
- 森林也通常是有序的，即有第1棵树、第2棵树、第3棵树之分。
  - 森林中每棵树的根彼此称为“兄弟”；
  - 如果将两棵树中将根结点连接到一个父节点，便得到一个树。



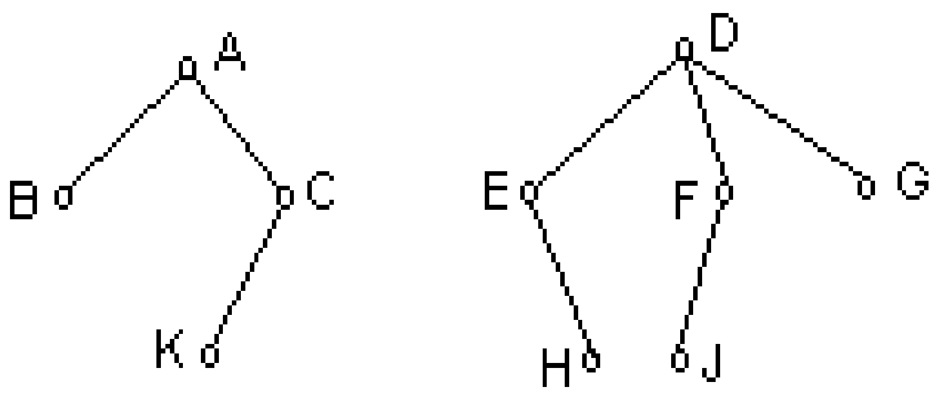
9.2 树和森林的存储表示

树和森林的存储表示

- 选择存储表示方法原则：结点本身+结点之间的关系
- 树和森林的存储表示（三种常见的结构）
  - 父指针表示法
  - 子表表示法
  - 长子-兄弟表示法（常用）

父指针表示法

- 树或森林组织成一个结点顺序表，其中每一结点包含父节点的下标。
  - 根节点不具有父节点，parent值设为-1
- 用这种方法表示的树是无序树，无法区分不同子树之间的顺序



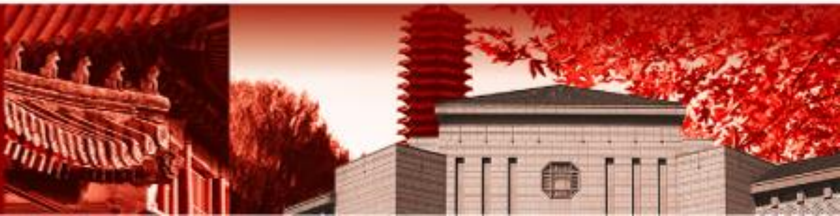
	info	parent
0	A	-1
1	B	0
2	C	0
3	K	2
4	D	-1
5	E	4
6	H	5
7	F	4
8	J	7
9	G	4

父指针表示法

```python
class ParTreeNode:
    def __init__(self, info):
        self.info = info
        self.parent = -1  # 使用索引表示父节点

class ParTree:
    def __init__(self, max_num):
        self.max_num = max_num
        self.nodelist = [None] * max_num
        self.n = 0
```

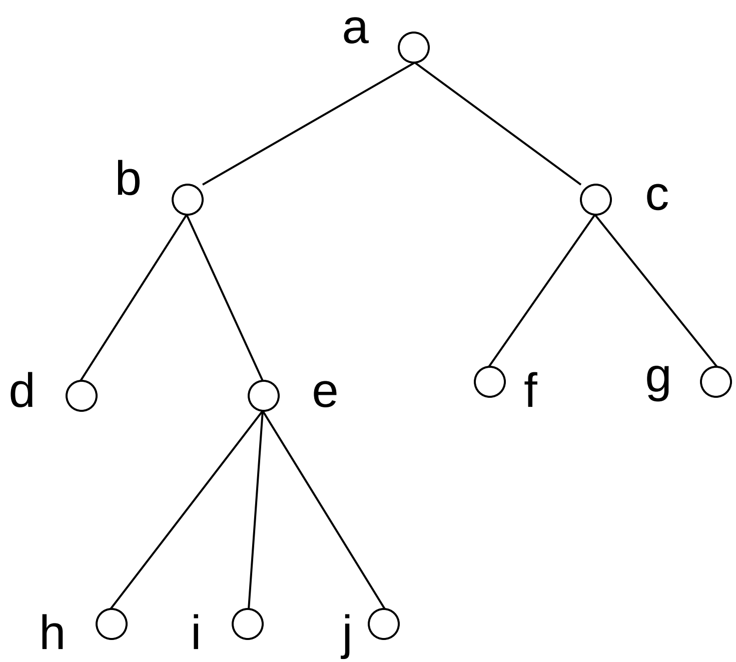
父指针表示法

- 如果两个结点到达同一根结点，它们一定在同一棵树中。
- 如果找到的根结点不同，那么两个结点就不在同一棵树中
- 优点
  - 容易找到父结点及其所有的祖先
  - 比较节省存储空间
- 缺点
  - 没有表示出结点之间的次序
  - 找结点的子女和兄弟比较费事（遍查整个数组）



父指针表示法的改进

- 为了表示有序树，可以按一种遍历次序在数组中存放结点
- 常见的一种方法是依次存放树的前序序列，如下图



	info	parent
0	a	-1
1	b	0
2	d	1
3	e	1
4	h	3
5	i	3
6	j	3
7	c	0
8	f	7
9	g	7

算法示例

- 在改进的父指针表示法的树中求右兄弟结点的位置
  - 即右侧的第一个兄弟

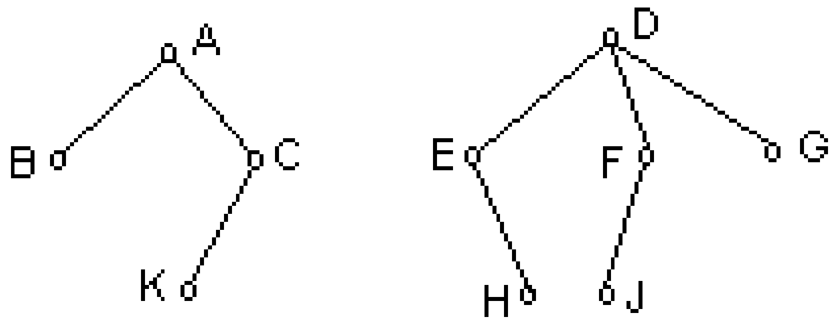
```
def right_sibling_partree(t, p):
    if 0 <= p < t.n:
        for i in range(p + 1, t.n):
            if t.nodelist[i].parent == t.nodelist[p].parent:
                return i
    return -1
```

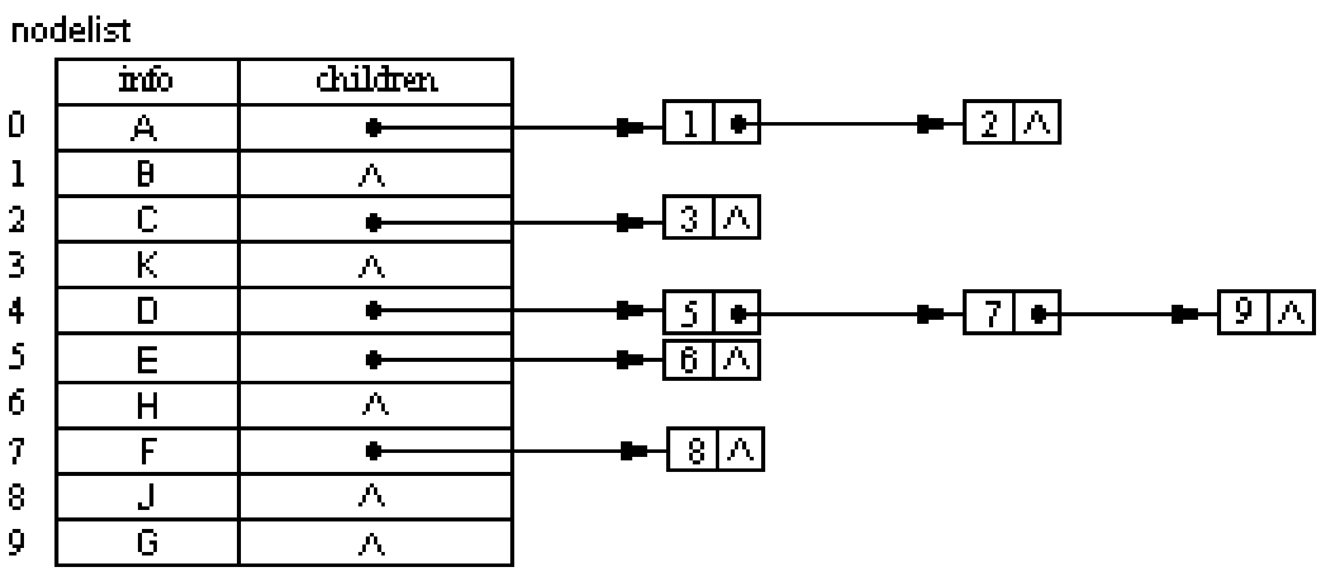
t.n表示当前树中实际存储的节点数量

从 **p+1** 开始，向右遍历树中的节点列表 **t.nodelist**

子表表示法

- 整棵树或森林组织成一个结点顺序表，其中每一结点使用数组或链表记录该结点的所有子节点





子表表示法

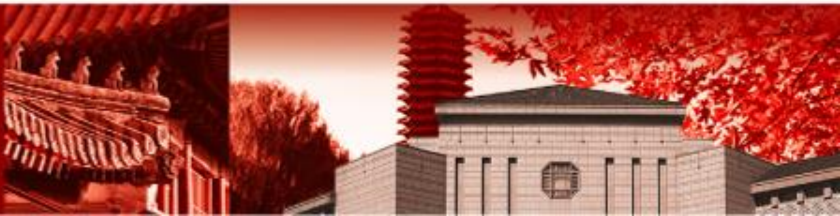
```python
# 定义子表中的节点
class EdgeNode:
    def __init__(self, nodeposition):
        self.nodeposition = nodeposition  # 节点在树中的位置
        self.link = None  # 指向下一个子节点的链接

# 定义子表表示法的树节点
class ChiTreeNode:
    def __init__(self, info):
        self.info = info  # 节点信息
        self.children = None  # 指向子表的指针

# 定义子表表示法的树
class ChiTree:
    def __init__(self, max_num):
        self.root = -1  # 根节点索引，-1 表示树为空
        self.n = 0  # 当前树中节点的数量
        self.nodelist = [None] * max_num  # 节点列表
```

算法：在子表表示法求右兄弟的位置

```python
# 获取某个节点的右兄弟节点
def right_sibling_chitree(t, p):
    for i in range(t.n):  # 遍历所有节点
        v = t.nodelist[i].children  # 获取当前节点的子表
        while v is not None:  # 遍历子表
            if v.nodeposition == p:  # 如果找到目标节点
                if v.link is None:  # 如果目标节点没有右兄弟
                    return -1
                else:
                    return v.link.nodeposition  # 返回右兄弟节点的位置
            v = v.link  # 移动到下一个子节点
    return -1  # 如果没有找到目标节点或其右兄弟，返回 -1
```

算法：在子表表示上求父结点的位置

```python
# 获取某个节点的父节点
def parent_chitree(t, p):
    for i in range(t.n):  # 遍历所有节点
        v = t.nodelist[i].children  # 获取当前节点的子表
        while v is not None:  # 遍历子表
            if v.nodeposition == p:  # 如果找到目标节点
                return i  # 返回目标节点的父节点索引
            v = v.link  # 移动到下一个子节点
    return -1  # 如果没有找到目标节点或其父节点，返回 -1
```

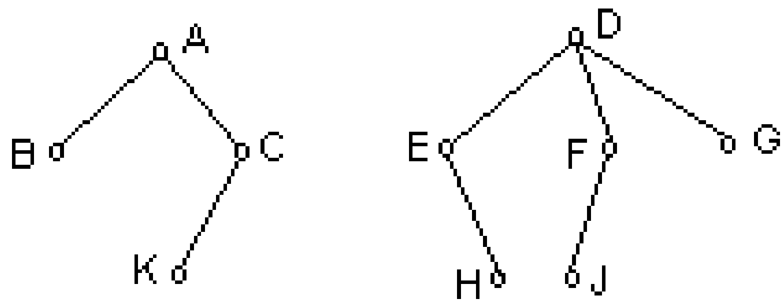



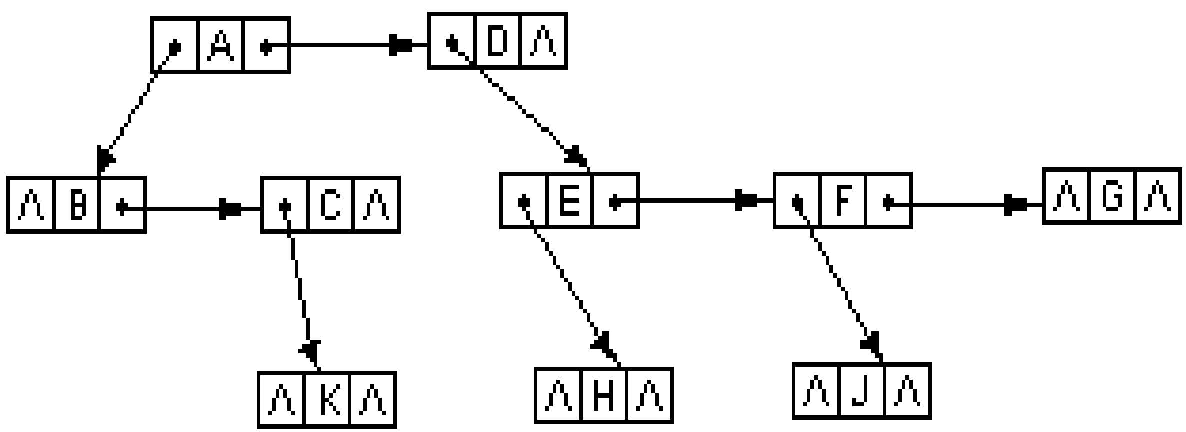
子表表示法

- 子表表示法的优点：
  - 方便找到父节点的所有子节点
- 缺点：
  - 由子节点找到父节点更困难
- 如果要将若干个子树合并成一个新树：
  - 子表表示法下，要考虑调整合并后树的子表
  - 父指针表示法下，只需调整子树的父指针即可

长子-兄弟表示法

- 每个结点存储：结点的值、最左子结点（长子）指针、右侧兄弟结点指针





长子-兄弟表示法

- 该表示法下，每个结点既包括了子结点（兄弟）信息，也包括了同层次结点（兄弟）的信息
- 比子表表示法空间效率更高
- 长子-兄弟表示法也称左子右兄表示法

长子-兄弟表示法

```python
# 定义长子-兄弟表示法的树节点
class CSNode:
    def __init__(self, info):
        self.info = info  # 节点信息
        self.lchild = None  # 指向最左子节点
        self.rsibling = None  # 指向右兄弟节点

# 定义长子-兄弟表示法的树
class CSTree:
    def __init__(self):
        self.root = None  # 根节点
```

长子-兄弟表示法

- 长子-兄弟表示法的优点：
  - 方便找子女、找兄弟运算
  - 找到全部子女很容易，先由lchild找到长子，再由rsibling字段逐个找右兄弟结点
- 缺点：
  - 找父结点比较麻烦
  - 也可以再为结点加一条指向父节点的指针

9.3 树和森林的遍历

树的遍历

- 定义：按某一规律系统地访问树的所有结点，并使每个结点恰好被访问一次。（又称为遍历）
- 遍历的结果：
  - 在遍历树的过程中，如果将各个结点按其被访问的先后顺序排列起来，则可得到一个包括所有结点的线性表

本质：将非线性结构转换为线性结构。

遍历方法

- 与二叉树的遍历类似，树的遍历方法同样包括：
  - 深度优先遍历[纵向遍历]
    - 前序遍历
    - 中序遍历
    - 后序遍历
  - 广度优先遍历[横向遍历]
    - 按层次顺序，逐层访问所有节点

前序遍历

- 先访问根结点
- 然后从左到右按前序次序遍历根结点的每棵子树
- 长子-兄弟表示法下的前序遍历：

```python
# 前序遍历（递归）
def pre_order(p):
    if p:  # 如果节点存在
        visit(p)  # 访问当前节点
        c = left_child(p)  # 获取当前节点的最左子节点
        while c:  # 遍历所有子节点
            pre_order(c)  # 递归前序遍历
            c = right_sibling(c)  # 移动到右兄弟节点
```

中序遍历

- 按中序次序遍历根结点的最左子树；
- 访问根结点；
- 从左到右按中序次序遍历根结点的其它各子树
- 长子-兄弟表示法下的中序遍历

```
# 中序遍历（递归）
def in_order(p):
    c = left_child(p)  # 获取当前节点的最左子节点
    if c is None:  # 如果没有子节点
        visit(p)  # 访问当前节点
        return
    in_order(c)  # 中序遍历左子树
    visit(p)  # 访问当前节点
    c = right_sibling(c)  # 移动到右兄弟节点
    while c:  # 遍历所有兄弟节点
        in_order(c)  # 中序遍历右兄弟子树
        c = right_sibling(c)  # 移动到下一个右兄弟节点
```

后序遍历

- 从左到右按后序次序遍历根结点的每棵子树； ·
- 访问根结点 ·
- 长子-兄弟表示法下的后序遍历：

```python
# 后序遍历（递归）
def post_order(p):
    c = left_child(p)  # 获取当前节点的最左子节点
    while c:  # 遍历所有子节点
        post_order(c)  # 递归后序遍历
        c = right_sibling(c)  # 移动到右兄弟节点
    visit(p)  # 访问当前节点
```

深度优先遍历的特点

- 相同点：在三种遍历序列中，兄弟结点的左右次序不变
- 不同点：只有祖先和子孙之间的相对次序可能有所不同
  - **在前序遍历序列中，结点的所有子孙都紧密排列在该结点的右边；**

    假定post(n)表示结点n在前序序列中的位置，desc(n)表示结点n的子孙个数，则结点x是结点n的子孙的充分必要条件为：

    $$post(n)+desc(n) \geq post(x) > post(n)$$

  - **在后序遍历序列中，结点的所有子孙都紧密排列在该结点的左边**

    假定post(n)表示结点n在后序序列中的位置，desc(n)表示结点n的子孙个数，则结点x是结点n的子孙的充分必要条件为：

    $$post(n)-desc(n) \leq post(x) < post(n)$$

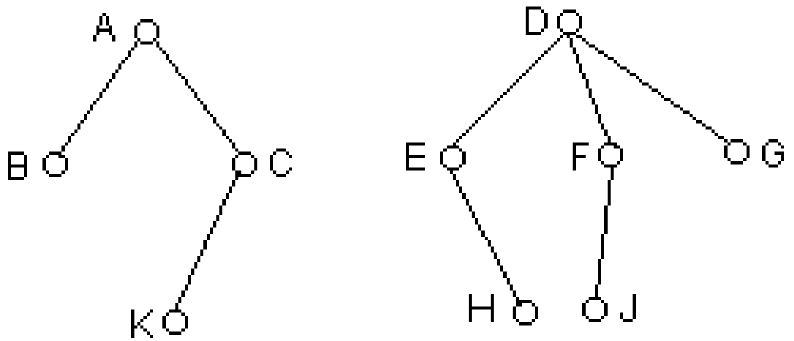
广度优先遍历

- 广度优先遍历：先访问层数为0的结点，然后从左到右逐个访问层数为1的结点，依此类推，直到访问完树中的全部结点。
- 按广度优先遍历所得到的线性表叫作树的层次序列
- 特点：
  - 在层次序列中，层数较低的结点总是排在层数较高的结点之前
  - 同层结点的左右次序还保持着，非同层结点的左右次序已被破坏

森林的遍历

- 森林的遍历方法有两种：前序遍历和后序遍历
- 前序遍历
  - 访问森林中第一棵树的根结点；
  - 前序遍历第一棵树的所有子树；
  - 前序遍历除去第一棵树之后的子森林。

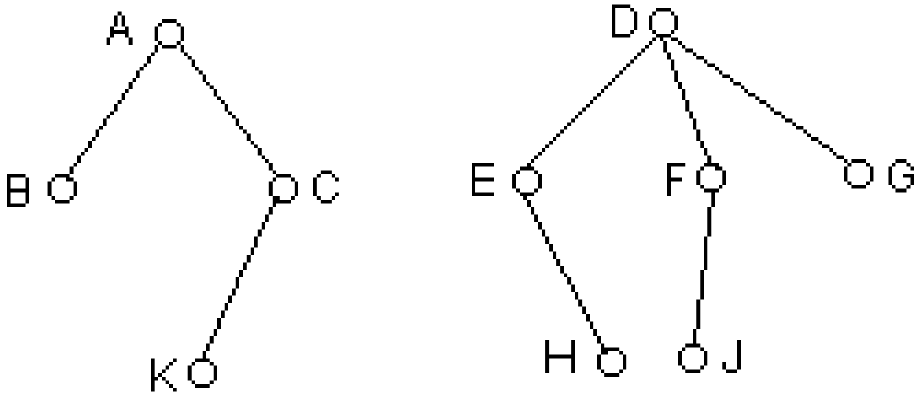
- 前序遍历序列：（A, B, C, K, D, E, H, F, J, G)



森林的遍历

- 后序遍历
  - 后序遍历第一棵树的所有子树；
  - 访问森林中第一棵树的根结点；
  - 后序遍历除去第一棵树之后的子森林。

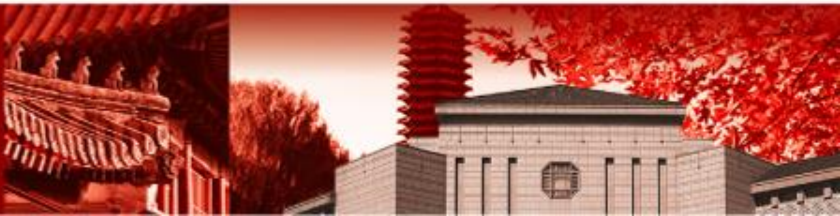
- 后序遍历序列：(B, K, C, A, H, E, J, F, G, D)



9.4 树和二叉树的转换

树、森林与二叉树的转换

- 森林与二叉树一一对应
  - 任何森林都唯一地对应到一棵二叉树；
  - 反过来，任何二叉树也都唯一地对应到一个森林
- 森林对应的二叉树中：
  - 一个结点的左子节点，是原来森林中的长子
  - 右子节点是原来森林中的下一个兄弟
  - 即：左孩子、右兄弟

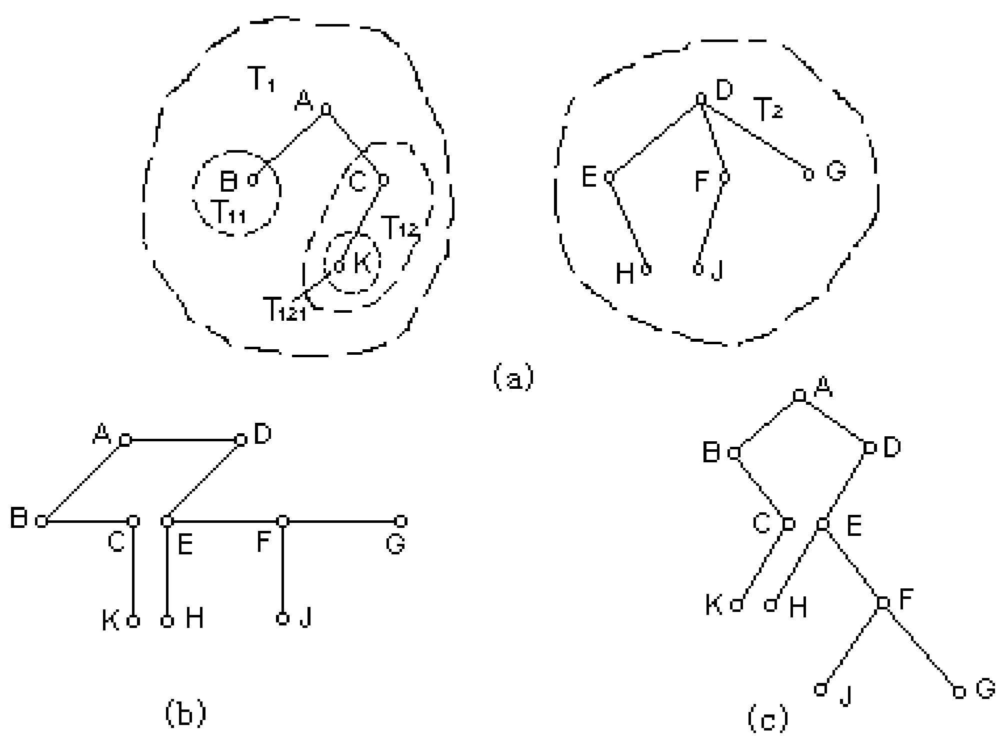


树、森林与二叉树的转换

- 森林($F=T_1, T_2, \ldots\ldots, T_n$)对应的二叉树B(F)：
- 若n = 0，则B(F)为空；
- 若n＞0，则：
  - B(F)的根是T_1的根
  - B(F)的左子树是$B(T_{11}, T_{12}, \ldots\ldots, T_{1m})$，其中$T_{11}, T_{12}, \ldots, T_{1m}$是$T_1$的子树；
  - B(F)的右子树是$B(T_2, \ldots\ldots, T_n)$
- 树对应的二叉树中，根节点的右子节点总是空的。
- 森林对应的二叉树中，根节点的右子节点是第2棵树的根

树、森林与二叉树的转换

- 加线：在树中所有相邻的兄弟之间加一连线
- 抹线：对树中每个结点，除其最左孩子外，抹去该结点与其余孩子间的连线
- 整理：以树的根结点为轴心，将整树顺时针转45°

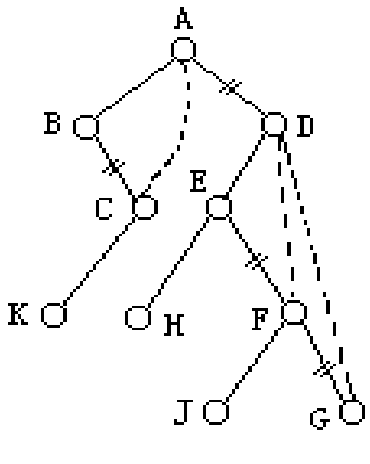


二叉树转换为树、森林

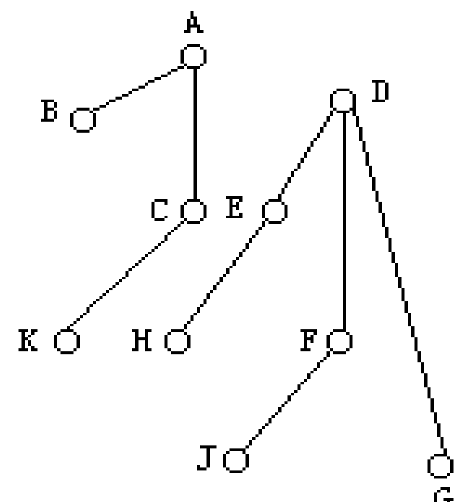
- 设B是一棵二叉树，r为根，L为左子树，R为右子树，则对应于B的森林F(B)的定义是：
  - 若B为空，则F(B)是空的森林；
  - 若B不为空，则F(B)是一棵树T_1加上森林F(R)，其中树T_1的根为r，r的子树为F(L)

二叉树转换为树、森林

- 方法
  - **若某结点是其父节点的左子节点**，则把该结点的右子节点，右子节点的右子节点……，都与该结点的父节点连起来，最后删除所有的父节点到右子节点的连线



(a)



(b)

9.5 并查集

并查集

- 并查集是用父结点表示法表示的树构成的森林
- 主要用于解决的问题：一些元素分布在若干个互不相交的集合(等价类)中，需要多次进行以下操作：
  - (1) 合并(Union) a, b两个元素所在的集合
  - (2) 查询(Find) 一个元素在哪个集合
  - (3) is_connected (a,b):判断两个元素a和b是否属于同一个集合。
- 例如：
  - 在社交网络中，判断两个人是否属于同一个社交圈。
  - 在图论中，判断两个节点是否连通。
  - 在一些算法竞赛中，处理动态连通性问题。

并查集

- 集合的表示
  - 并查集通过森林来表示多个集合，每个集合是一个树结构。每个节点存储其父节点的指针，根节点的父节点指向自身。
  - 如果两个节点的根节点相同，则它们属于同一个集合。
- 关键操作
  - 查找操作（Find）：通过不断向上查找父节点，直到找到根节点。
  - 合并操作（Union）：将一个集合的根节点连接到另一个集合的根节点上，作为另一个集合的根节点的子节点。

等价类的求解问题

- 等价关系具有自反性、对称性和传递性
  - **自反性**：任何元素必须和自身相关。

    $\forall a \in A, \Rightarrow (a, a) \in R$

    例子：在“等于”关系中，a=a恒成立。
  - **对称性**：如果 a和 b相关，则 b 和 a也必须相关。

    $(a, b)\in R,(b, c)\in R \Rightarrow (a, c)\in R$

    例子：在“同学关系”中，若甲是乙的同学，则乙也是甲的同学。
  - **传递性**：如果 a和 b相关，且 b和 c相关，则 a和 c必须相关。

    $(a, b)\in R,(b, c)\in R \Rightarrow (a, c)\in R$

    例子：在“平行线”关系中，若直线 $L1 \| L2$ 且 $L2 \| L3$，则 $L1 \| L3$。

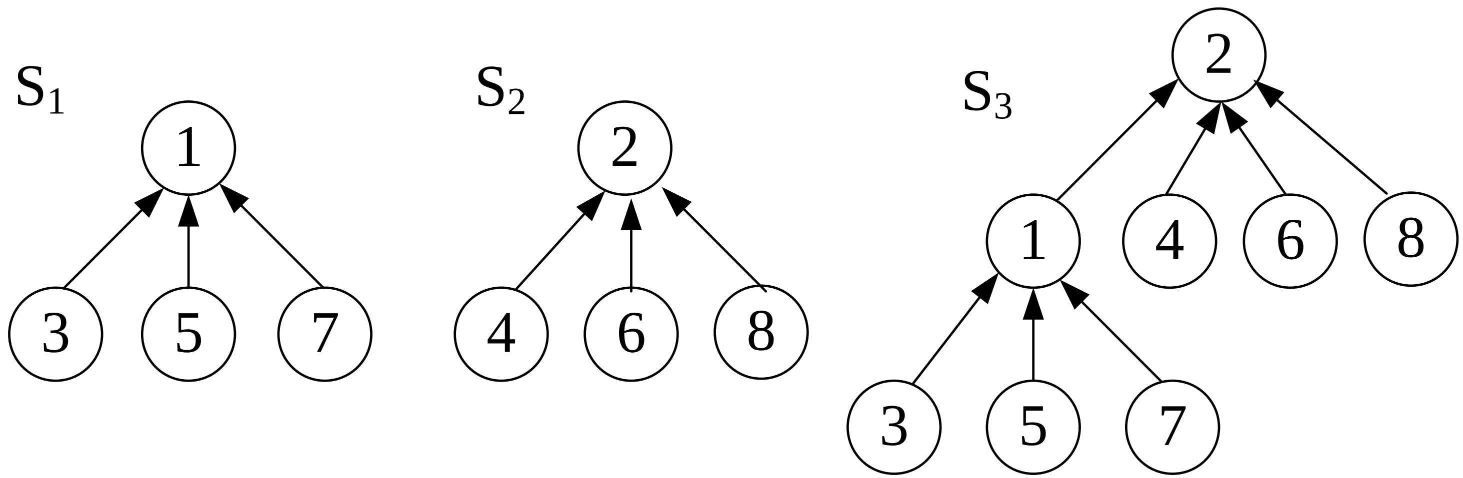
北京大學

等价类的求解问题

- 并查算法可以很容易地构建等价类
  - 开始时，每个元素都在独立的只包含一个结点的树中，而它自己就是根结点
  - 对于一个等价对中的两个元素，判断是否在同一棵树中：
    - 如果是，由于它们已经在同一个等价类中，不需要作变动
    - 否则两个等价类可以用 UNION 函数归并

等价类

- Union操作：合并S_1，S_2 等价类，将S_1根节点的父指针指向S_2根节点



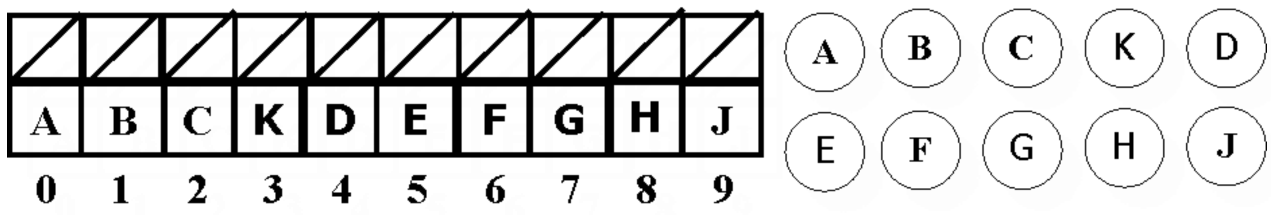
(a) (b) (c)

按秩合并

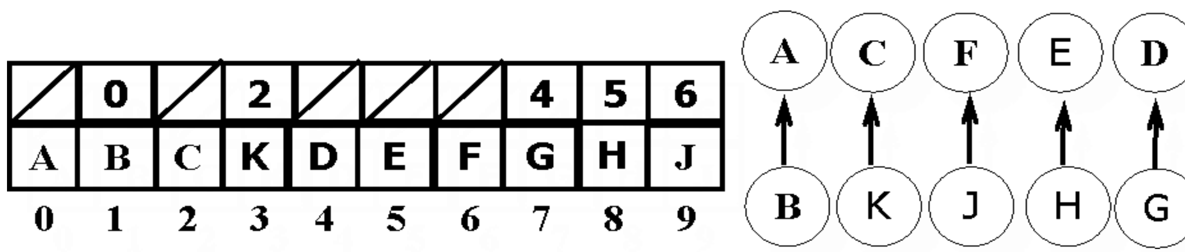
- Find、Union操作的代价都取决于目标结点到根节点的距离，与所有树的最大高度有关。
  - 最坏情况下，树退化为线性结构，复杂度将变成O(n)的
- 按秩合并规则
  - 合并两棵树时，将较矮的树合并到较高的树中去，将矮树的根节点的父指针指向高树的根
  - 即便如此，两棵一样高的树合并时仍然会导致新树的高度增加

示例

- 10个结点A、B、C、D、E、F、G、H、J、K和它们的等价关系（A,B）、(C,K) 、（F,J) 、(E,H) 、(D,G) 、(K,A) 、(E,G) 、(H,J)



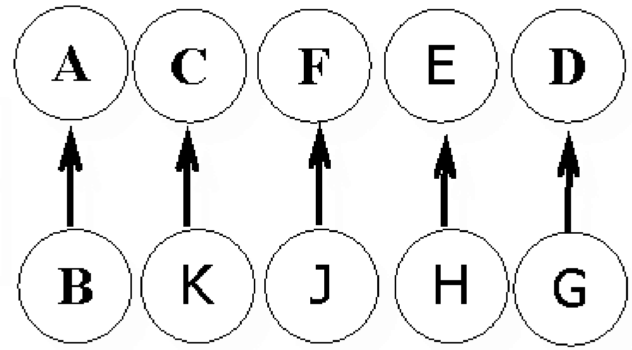
- 执行Union(A, B), Union(C, K), Union(F, J), Union(E, H), Union(D, G)



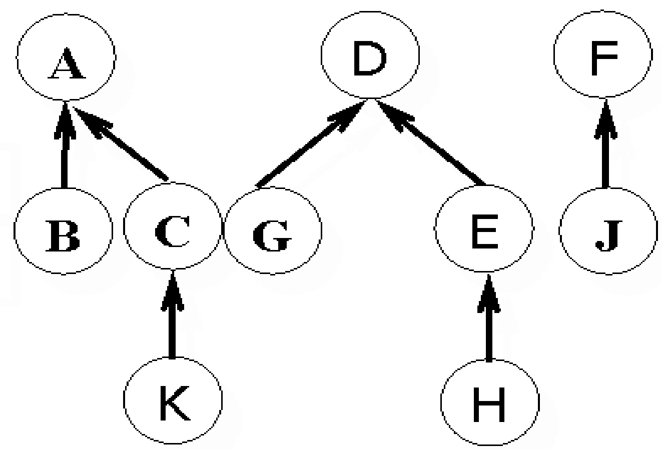
示例

- 执行Union(K, A)，Union(E, G)

/	0	/	2	/	/	/	4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9



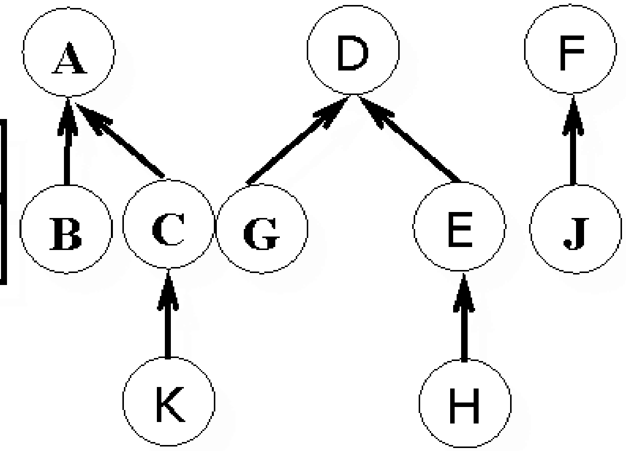
/	0	0	2	/	4	/	4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9



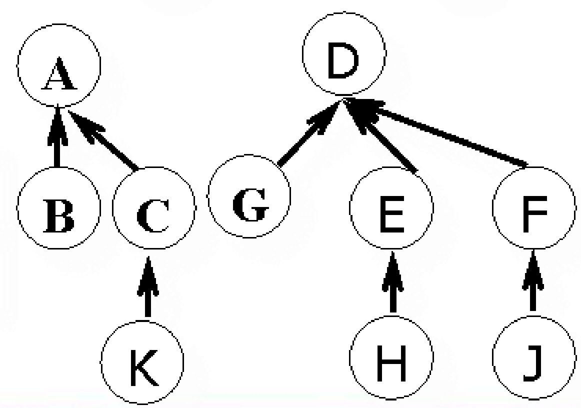
示例

- 使用按秩合并规则，执行Union(H, J)

	0	0	2		4		4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9



	0	0	2		4	4	4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9



北京大学

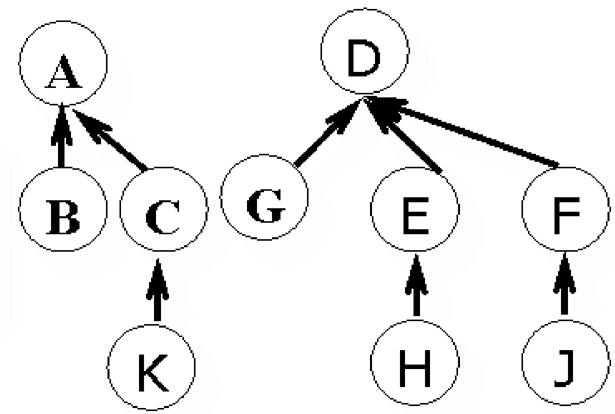
路径压缩

- 在查找过程中，可以将路径上的所有节点直接连到根节点。
- 这样可以显著减少树的高度，进一步优化查找效率。
- 在Find操作中递归实现路径压缩：
  - 对父节点递归调用，即查找父节点的根节点，并将路径上的所有结点的父指针都指向根，然后将自己父指针也指向根
  - 在根结点处达到递归的终点

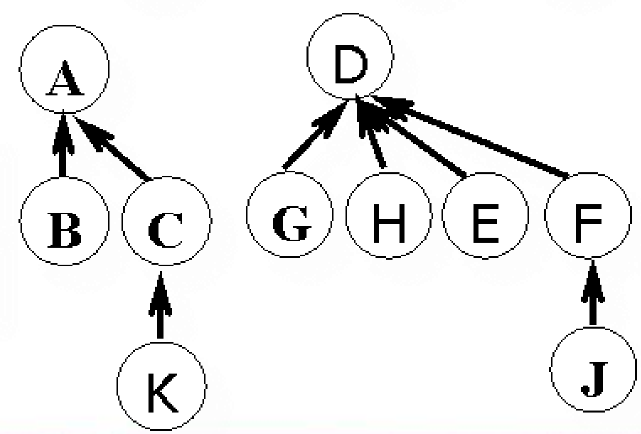
示例

- 使用路径压缩规则，执行Find(H)操作

/	0	0	2	/	4	4	4	5	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9



/	0	0	2	/	4	4	4	4	6
A	B	C	K	D	E	F	G	H	J
0	1	2	3	4	5	6	7	8	9



并查集——Python实现

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))  # 初始化每个节点的父节点为自身
        self.rank = [0] * n  # 初始化每个节点的秩（树的高度）

    def find(self, x):
        if self.parent[x] != x:  # 如果当前节点不是根节点
            self.parent[x] = self.find(self.parent[x])  # 路径压缩
        return self.parent[x]
```

并查集——Python实现

```python
def union(self, x, y):
    root_x = self.find(x)
    root_y = self.find(y)
    if root_x != root_y:  # 如果两个节点不在同一个集合
        if self.rank[root_x] < self.rank[root_y]:  # 按秩合并
            self.parent[root_x] = root_y
        elif self.rank[root_x] > self.rank[root_y]:
            self.parent[root_y] = root_x
        else:
            self.parent[root_x] = root_y
            self.rank[root_y] += 1

def is_connected(self, x, y):
    return self.find(x) == self.find(y)  # 判断两个节点是否连通
```

并查集的效率分析

- 使用路径压缩，在多次Find操作后，大部分的结点都变成了根的子节点，或者离根很近，Find操作的速度就会大大提升
- 可以证明单靠路径压缩，Find 操作的均摊复杂度是O(log n)
- 加上按秩合并规则，Find操作的均摊复杂度基本上可以做到O(1)
  - 更准确的复杂度是O(α(n))，其中α(n)为反阿克曼函数
  - α(n)增长速度极慢，α(n)不会超过5，可以视为常数

选择题

- 如果结点A有3个兄弟(不包括A本身),而且B是A的父结点,则B的度是（ ）。

A. 3

B. 4

C. 5

D. 1

选择题

- 设高度为h的二叉树上只有度为0和度为2的结点，则此二叉树中所包含的结点数至少为(   )。

A. 2h

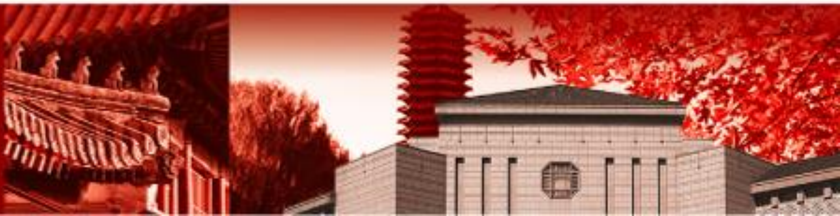
B. 2h-1

C. 2h +1

D. h+1

判断题

- 二叉树的前序遍历序列中,任意一个结点均处在其子女结点的前面。
- 由树转换成二叉树，其根结点的右子树总是空的。
- 哈夫曼树是带权路径长度最短的树，路径上权值较大的结点离根较近。



选择题

- 树最适合用来表示（）

A 有序的数据元素

B 无序的数据元素

C 元素之间具有分支层次关系的数据

D 元素之间无联系的数据

选择题

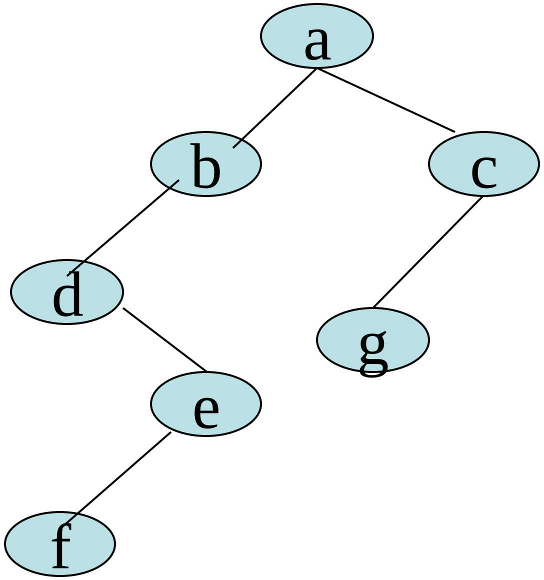
- 如图所示二叉树的中序遍历序列是（）

A abcdgef

B dfebagc

C dbaefcg

D defbagc



选择题

- 已知某二叉树的后序遍历序列是dabec,中序遍历序列是debac,它的前序遍历序列是（ ）。

A acbed

B decab

C deabc

D cedba

选择题

- 任何一棵二叉树的叶结点在先序、中序和后序的遍历序列中的相对次序（　　）

A 不发生改变

B 发生改变

C 不能确定

D 以上都不对

选择题

- 某二叉树的前序序列和后序序列正好相反，则该二叉树一定是（　　）二叉树。

A. 空或只有一结点

B. 树的高度等于其结点数减1

C. 任一结点都只有右子结点

D. 任一结点都只有左子结点

选择题

- 设n,m为一棵二叉树上的两个结点，在中序遍历时，n在m前的条件是（　）

A. n在m右方

B. n在m祖先

C. n在m左方

D. n在m子孙

选择题

- 在一非空二叉树的中序遍历序列中,根结点的右边(   )

A.只有右子树上的所有结点
B.只有右子树上的部分结点
C.只有左子树上的所有结点
D.只有左子树上的部分结点

选择题

- 某二叉树T有n个结点,设按某种顺序对T中的每个结点进行编号，编号值为1,2,...n.且有如下性质：T中任意结点v，其编号等于左子树上的最小编号减一，而v的右子树的结点中，其最小编号等于v左子树上结点的最大编号加一，这是按(　)编号的。

A. 中序遍历序列
B. 前序遍历序列
C. 后序遍历序列
D. 层次顺序