第五章 树

5.1 树的基本概念

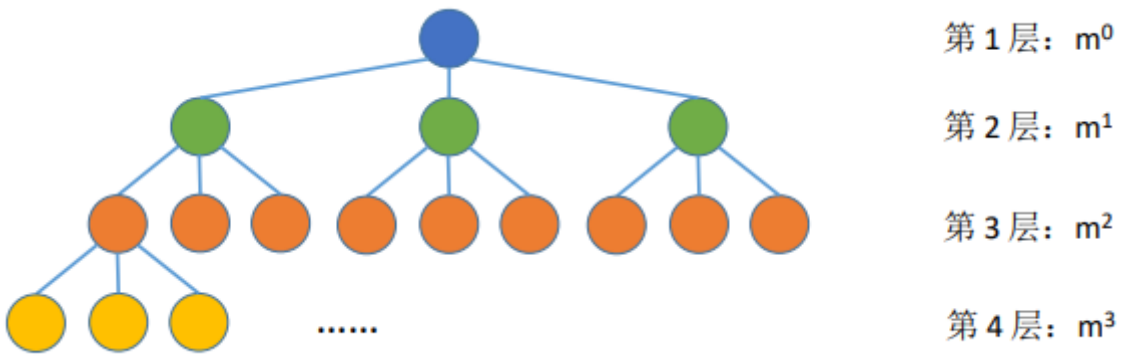
1. 树是n（n≥0）个结点的有限集合，n = 0时，称为**空树**。
2. 空树中应满足：
    1. 有且仅有一个特定的称为根的结点。
    2. 当n > 1时，其余结点可分为m（m>0）个互不相交的有限集合T1，T2，...，Tm，其中每个集合本身又是一棵树，并且称为根结点的子树。
3. 度：树中一个结点的孩子个数称为该结点的度。**所有结点的度的最大值是树的度。**
4. 度大于0的结点称为分支结点，度为0的结点称为叶子结点。
5. 结点的层次（深度）：从上往下数。
6. 结点的高度：从下往上数。
7. 树的高度（深度）：树中结点的层数。
8. 有序树：逻辑上看，树中结点的各子树从左至右是有次序的，不能互换。
9. 若树中结点的各子树从左至右是有次序的，不能互换，则该树称为有序树，否则称为无序树。
10. 树中两个结点之间的路径是由这两个结点之间所经过的结点序列构成的，而路径长度是路径上所经过的边的个数。
11. 森林：森林是m（m≥0）棵互不相交的树的集合。

5.1.2 树的常考性质

1. 结点数 = 总度数 + 1
2. 度为 m 的树、m 叉树的区别：

度为m的树	m叉树的区别
任意结点的度≤m（最多m个孩子）	任意结点的度≤m（最多m个孩子）
至少有一个结点度=m（有m个孩子）	允许所有结点的度都 < m
一定是非空树，至少有m+1个结点	可以是空树

3. 度为 m 的树第 i 层至多有**m^(i-1)**个结点（i≥1）；m 叉树第 i 层至多有**m^(i-1)**个结点（i≥1）。



4. 高度为 h 的 m 叉树至多有

$$m^h - 1/m - 1$$

个结点。（等比数列求和）

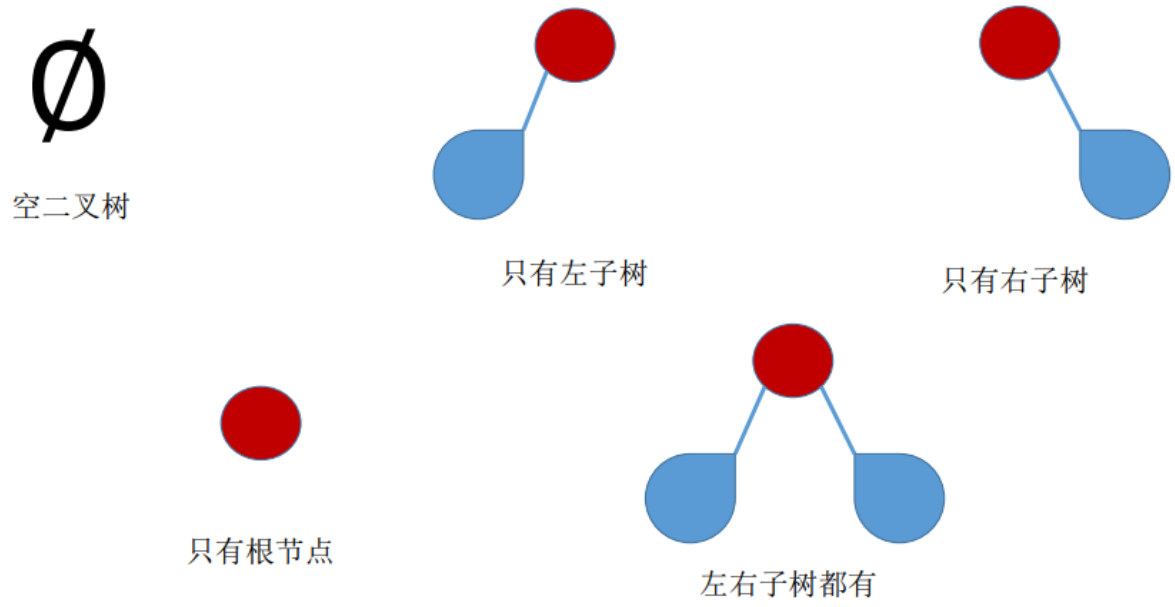
5. 高度为 h 的 m 叉树至少有 h 个结点；高度为 h、度为 m 的树至少有（h+m-1）个结点。
6. 具有 n 个结点的 m 叉树的最小高度为

$$\lceil log_m[n(m-1)+1] \rceil$$

5.2 二叉树

5.2.1. 二叉树的定义

1. 二叉树是 n（n≥0）个结点的有限集合：
    1. 或者为空二叉树，即 n = 0。
    2. 或者由一个根结点和两个互不相交的被称为根的左子树和右子树组成，左子树和右子树又分别是一棵二叉树。
2. 二叉树的特点：
    1. 每个结点至多只有两棵子树。
    2. 左右子树不能颠倒（二叉树是有序树）。
3. 二叉树的五种状态：
    1. 空二叉树
    2. 只有左子树
    3. 只有右子树
    4. 只有根节点
    5. 左右子树都有



5.2.2 几个特殊的二叉树

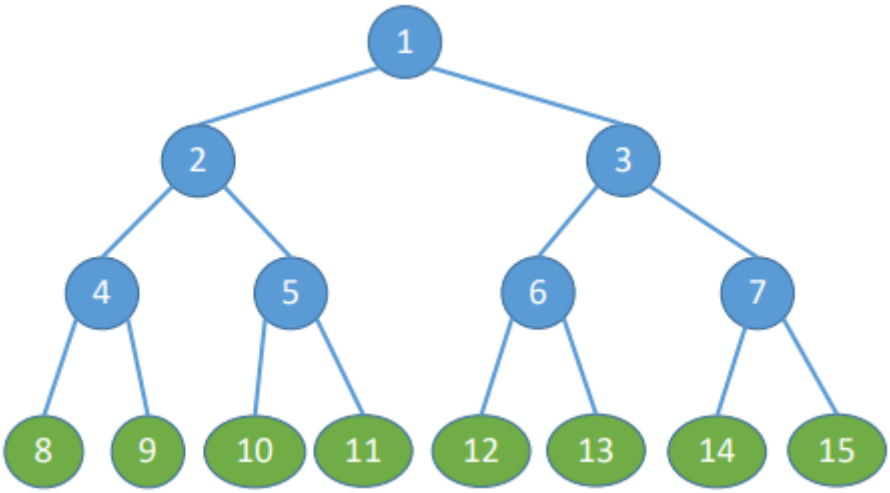
1、满二叉树

一棵高度为h，结点数为`2^h-1`的二叉树称为满二叉树，即 **树中的每层都含有最多的结点。**

特点：

- 满二叉树的叶子结点都集中在二叉树的最下一层
- 除了叶子结点外的每个节点的度都为2。

- 可以对满二叉树按照层序编号，约定编号从根节点（编号为1）起，自上而下，自左向右。这样每个结点对应一个编号，对于编号为i的结点，若有双亲，则其双亲为⌊i/2⌋，若有左孩子，则其左孩子为2i，若有右孩子，则其右孩子为2i+1.

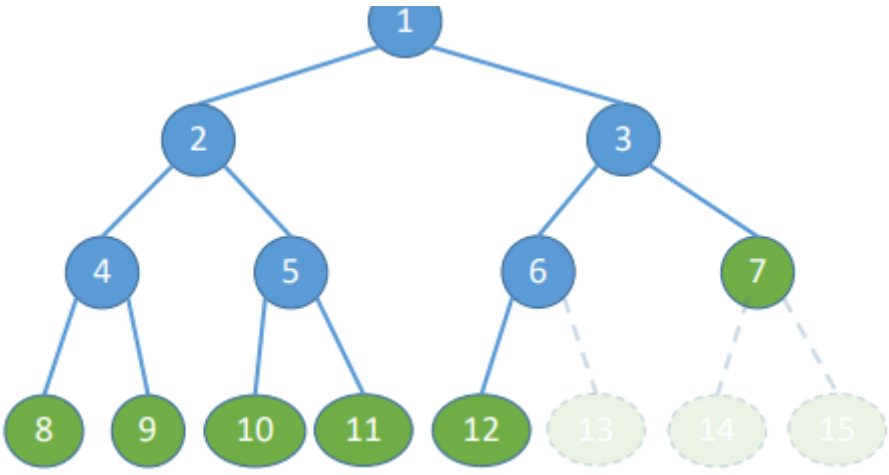


2、完全二叉树

高度为h，有n个结点的二叉树，当且仅当每个节点都与高度为h的满二叉树中编号为1~n的结点一一对应时，称为完全二叉树。

其有如下特点：

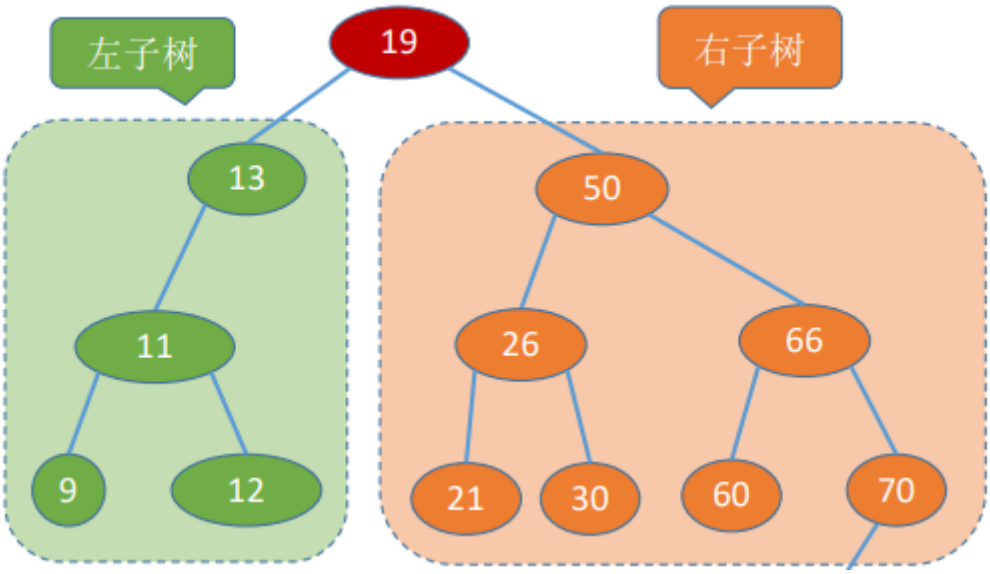
- 若i≤⌊i/2⌋，则结点i为分支节点，否则为叶子结点。
- 叶子结点只可能在层次最大的两层上出现。
- 若有度为1的结点，则只可能有一个，且该结点只有左孩子。
- 按照层序编号后，一旦出现某结点（编号为i）为叶子结点或只有左孩子，则编号大于i的结点均为叶子结点。
- 若n为奇数，则每个分支节点都有左右孩子，若n为偶数，则编号最大的分支结点（n/2）只有左孩子，没有右孩子。其余分支节点左右孩子都有。



> 满二叉树一定是完全二叉树，完全二叉树不一定是满二叉树。

3、二叉排序树

左子树上的所有结点的关键字均小于根节点的关键字；右子树上的所有结点的关键字均大于根节点的关键字；左子树和右子树又各自是一颗二叉排序树。



4、平衡二叉树

树上的任一结点的左子树和右子树的深度之差不超过1.

5.2.3 二叉树的性质

1. 设非空二叉树中度为 0、1 和 2 的结点个数分别为 n_0、n_1 和 n_2，则 $n_0 = n_2 + 1$。

   推导过程：设树中结点总数为 n，则：$n = n_0 + n_1 + n_2$ ，$n = n_1 + 2n_2 + 1$。

2. 二叉树第 i 层至多有 2^{i-1} 个结点（i≥1）。

3. 高度为 h 的二叉树至多有 $2^h - 1$ 个结点（满二叉树）。

4. 具有 n 个（n>0）结点的完全二叉树的高度 h 为 $\lceil \log_2(n+1) \rceil$ 或 $\lfloor \log_2 n \rfloor + 1$。

5. 对于完全二叉树，可以由总结点数 n 推出度为 0、1 和 2 的结点个数 n_0、n_1 和 n_2。

   推导过程：n_1 = 0 或 1，$n_0 = n_2 + 1$，则 $n_0 + n_2$ 一定是奇数。

   若完全二叉树有 $2k$（偶数）个结点，则有 $n_1 = 1$，$n_0 = k$，$n_2 = k - 1$；

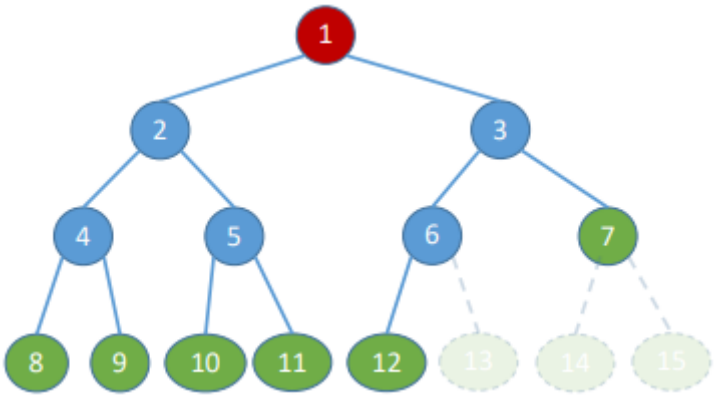
   若完全二叉树有 $2k - 1$（奇数）个结点，则有 $n_1 = 0$，$n_0 = k$，$n_2 = k - 1$；

5.2.4 二叉树的存储结构

1、顺序存储

> 包含的结点个数有上限
>
> 顺序存储完全二叉树：定义一个长度为 MaxSize 的数组 t，按照从上至下、从左至右的顺序依次存储完全二叉树中的各个结点。让第一个位置空缺，保证数组中下标和结点编号一致。

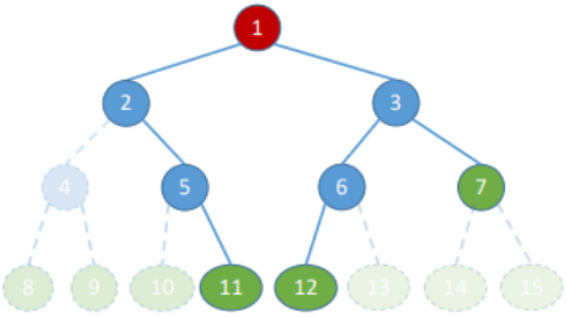
根据二叉树的性质，**完全二叉树和满二叉树采用顺序存储比较合适**，树中结点的序号可以唯一反映结点之间的逻辑关系，这样既能最大程度上的节省空间，又能根据数组元素的下标来确定结点在二叉树中的位置以及结点间的关系。



这样可以使用二叉树的性质求一些问题：

> - 结点 i 的左孩子：$2i$
> - 结点 i 的右孩子：$2i+1$
> - 结点 i 的父节点：$\lfloor i/2 \rfloor$
> - 结点 i 的层次：$\lceil \log_2(i+1) \rceil$ 或 $\lfloor \log_2 i \rfloor + 1$

而对于一般的二叉树而言，若使用顺序存储，则只能添加一些并不存在的空结点，让每个结点与二叉树上的结点相对照，再存储到一维数组的相应分量中。这样存在着空间的浪费，不建议使用顺序存储。**因此，二叉树的顺序存储结构，只适合存储完全二叉树。**



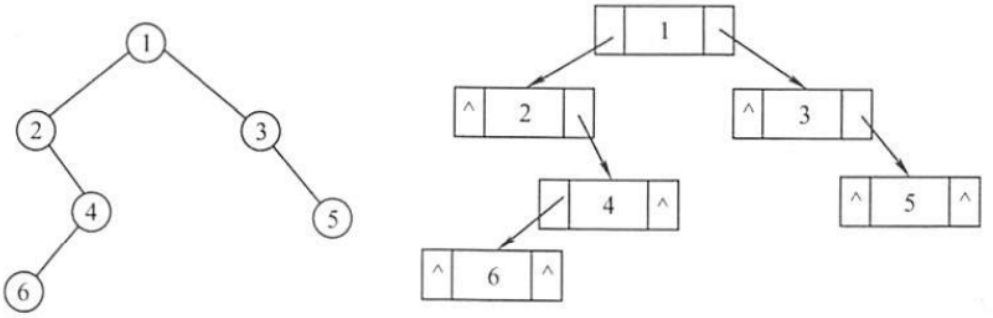
顺序存储的结构描述如下：

```
#define MaxSize 100
// 二叉树的顺序存储
struct TreeNode {
    int data;   // 结点中的数据元素
    bool isEmpty;   // 结点是否为空
};
TreeNode t[MaxSize];    // 定义一个长度为MaxSize的数组t，按照从上到下，从左到右的顺序依次存储完全二叉树的各个节点
```

2、链式存储

为了解决存储一般二叉树的空间浪费问题，一般二叉树的存储使用链式存储结构。使用链表结点来存储二叉树中的各个结点。在二叉树中，结点的结构通常包括若干数据域以及若干指针域。

二叉链表的存储结构如下：



其结构描述如下：

```
typedef struct BiTNode {
    ElemType data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```

> 【重要】在含有n个结点的二叉链表中，含有n+1个空链域。

> 为了找到指定结点的父结点，一般要从根节点开始遍历，可在BiTNode中设置一个新的指针存储父结点来解决此问题。

5.3 二叉树的遍历与线索二叉树

> 二叉树的遍历类似：
>
> 先序遍历：前缀表达式
>
> 中序遍历：中缀表达式（需添加界限符）
>
> 后序遍历：后缀表达式

5.3.1 二叉树的遍历

二叉树的遍历是 **按照某条搜索路径访问树中的每个结点，使得每个节点均被访问一次，而且仅被访问一次。**

1、先序遍历（PreOrder）

先序遍历的操作过程如下：

若二叉树为空，则什么都不做，否则：

1. 访问根节点
2. 先序遍历左子树
3. 先序遍历右子树

对应的递归算法如下：

```
void PreOrder(BiTree T) {
    if (T == NULL) return;
    visit(T);
    PreOrder(T->lchild);
    PreOrder(T->rchild);
}
```

2、中序遍历(InOrder)

中序遍历的操作过程如下：

若二叉树为空，则什么也不做，否则：

1. 中序遍历左子树;
2. 访问根结点;
3. 中序遍历右子树。

对应的递归算法如下：

```
void InOrder(BiTree T) {
    if (T == NULL) return;
    InOrder(T->lchild);
    visit(T);
    InOrder(T->rchild);
}
```

3、后序遍历（PostOrder）

后序遍历的操作过程如下：

若二叉树为空，则什么也不做，否则：

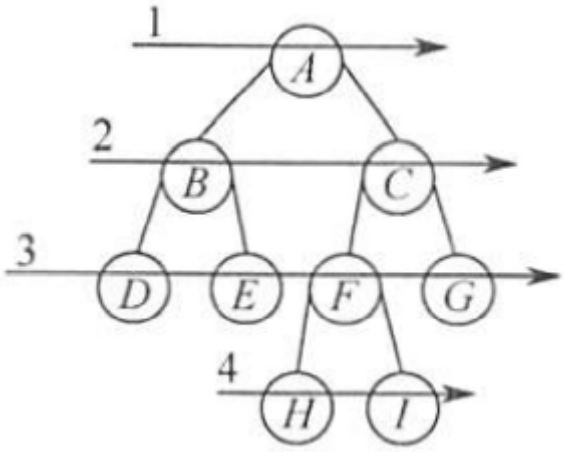
1. 后序遍历左子树;
2. 后序遍历后子树;
3. 访问根结点;

对应的递归算法如下：

```
void PostOrder(BiTree T) {
    if (T == NULL) return;
    PostOrder(T->lchild);
    PostOrder(T->rchild);
    visit(T);
}
```

4、层序遍历

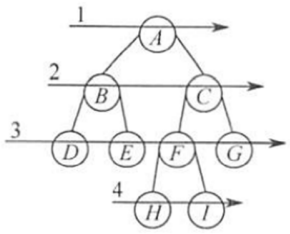
按照层序来进行遍历，如下图所示：



要进行层次遍历，需要借助一个队列。先将二叉树根结点入队，然后出队，访问出队结点，若它有左子树，则将左子树根结点入队;若它有右子树，则将右子树根结点入队。然后出队，访问出队结点……如此反复，直至队列为空。

其示例如下：

A	根节点A入队
A B C	A出队，A有左子树B，B入队 A有右子树C，C入队
A B C D E	B出队，访问出队结点，B有左子树D，D入队，B有右子树E，E入队
A B C D E F G	C出队，访问出队结点，C的左右子树根节点F、G入队
A B C D E F G	D出队，访问出队结点，D无左右子树，继续
A B C D E F G	E出队，E无左右子树，继续
A B C D E F G H I	F出队，访问出队结点，F的左子树H入队，F的右子树I入队



A B C D E F G H I	G出队，G无左右子树，继续
A B C D E F G H I	H出队，H无左右子树，继续
A B C D E F G H I	I出队，I无左右子树，此时，队列为空，层序遍历完成

5、由遍历序列构造二叉树

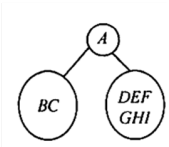
1. 一个前序遍历序列可能对应多种二叉树形态。同理，一个后序遍历序列、一个中序遍历序列、一个层序遍历序列也可能对应多种二叉树形态。即：**若只给出一棵二叉树的 前/中/后/层序遍历序列 中的一种，不能唯一确定一棵二叉树。**
2. 由二叉树的遍历序列构造二叉树：
    1. 前序+中序遍历序列
    2. 后序+中序遍历序列
    3. 层序+中序遍历序列
3. 由 前序+中序遍历序列 构造二叉树：由前序遍历的遍历顺序（根节点、左子树、右子树）可推出根节点，由根节点在中序遍历序列中的位置即可推出左子树与右子树分别有哪些结点。
4. 由 后序+中序遍历序列 构造二叉树：由后序遍历的遍历顺序（左子树、右子树、根节点）可推出根节点，由根节点在中序遍历序列中的位置即可推出左子树与右子树分别有哪些结点。

5. 由 层序+中序遍历序列 构造二叉树：由层序遍历的遍历顺序（层级遍历）可推出根节点，由根节点在中序遍历序列中的位置即可推出左子树与右子树分别有哪些结点。

示例：

求先序遍历序列 ABCDEFGHI 和中序序列 BCAEDGHFI所确定的二叉树。

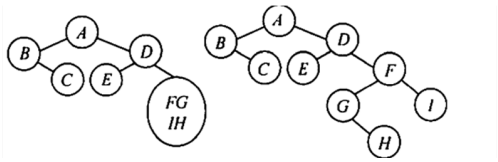
A B C D E F G H I
B C A E D G H F I



1、先序遍历的第一个元素一定是二叉树的根节点，其在中序遍历序列中，左侧为左子树的结点，右侧为右子树的结点，此时，得到了两个新的遍历序列BC和EDGHFI

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

2、分析可知，在先序遍历序列中，BC序列的B靠前，故B就是左子树的根节点，而根据中序遍历序列，C在B的后面，故C是B的右子树的根节点。
同理，EDGHFI序列中，D在先序遍历的第一位，故右子树的根节点为D，而中序遍历中，D的左侧就是D作为根节点的子树的左子树，D的右侧就是相应的右子树序列，即可再次进行划分



3、同理，可继续划分，最终得到左侧的二叉树

5.3.2 线索二叉树

> 线索二叉树是一种**物理结构**！

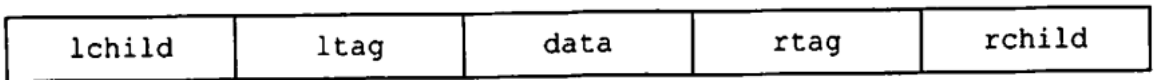
1、线索二叉树的基本概念

传统的二叉链表只能体现一种父子关系，**不能直接得到结点在遍历中的前驱和后继。**而考虑到在含有n个结点的二叉树中，**有n+1个空指针。**考虑能否利用这些空指针来存放指向其前驱或后继的指针？这样就可以更加方便地遍历二叉树。

> 故**含n个结点的线索二叉树共有n+1个线索**

引入线索二叉树正是为了加快查找结点前驱和后继的速度。

线索二叉树的结点结构如下：



规定：

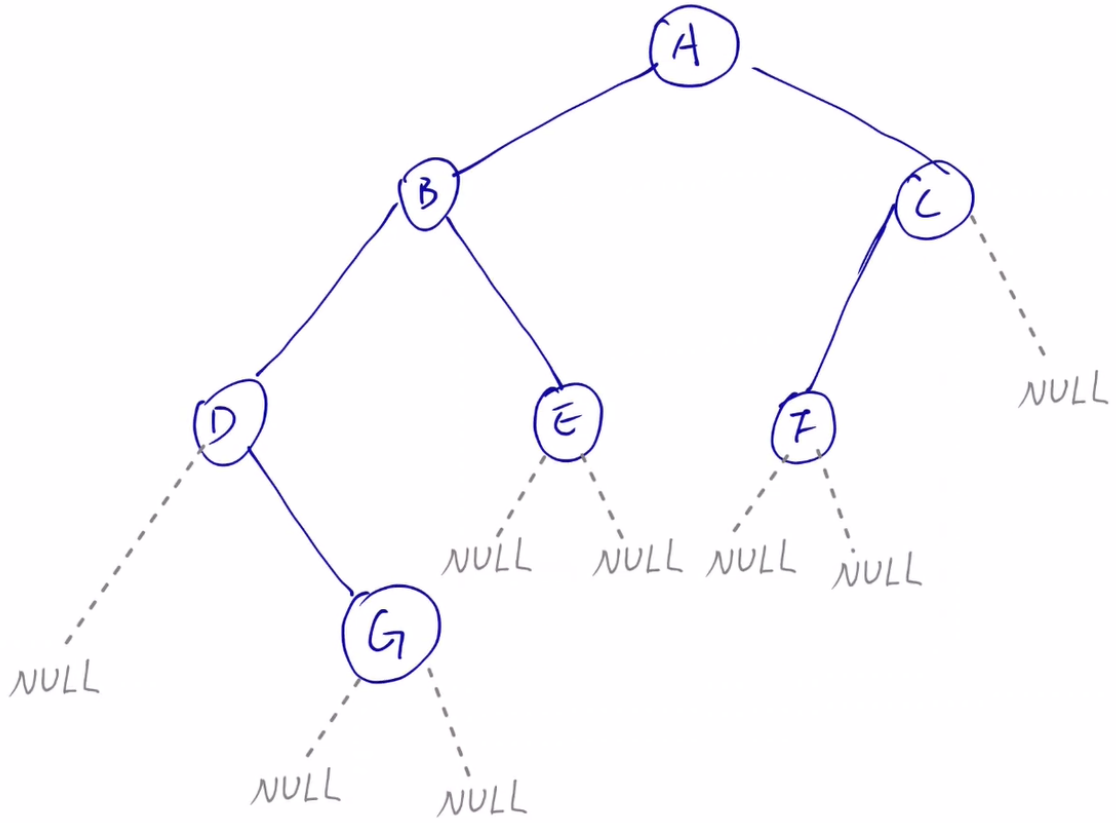
- 若无左子树，则lchild指向其前驱节点，ltag为1，否则lchild指向左孩子，ltag为0
- 若无右子树，则rchild指向其后继结点，rtag为0，否则rchild指向左孩子，rtag为0

其存储结构描述如下：

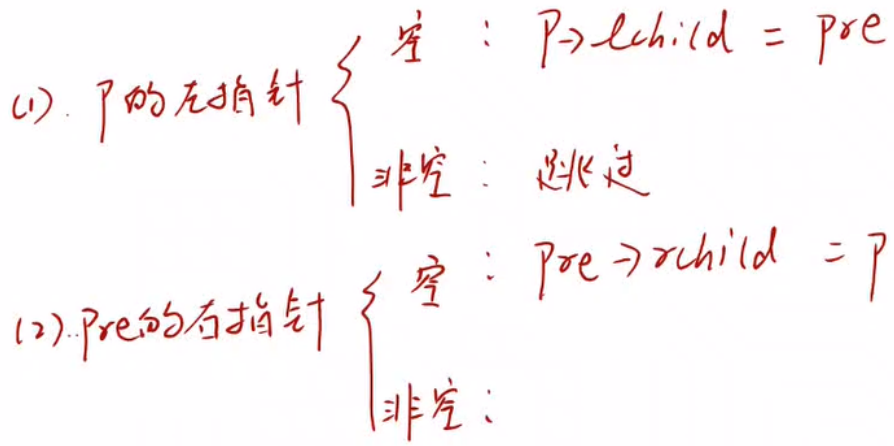
```
typedef struct ThreadNode {
    int data;                               // 数据域
    struct ThreadNode *lchild, *rchild; // 左右孩子指针
    int ltag, rtag;                         // 左右线索标志
} ThreadNode, *ThreadBiTree;
```

以这种结点结构构成的二叉链表作为二叉树的存储结构，称为**二叉链表**。其中指向结点前驱和后继的指针称为**线索**，加上线索的二叉树称为**线索二叉树**。

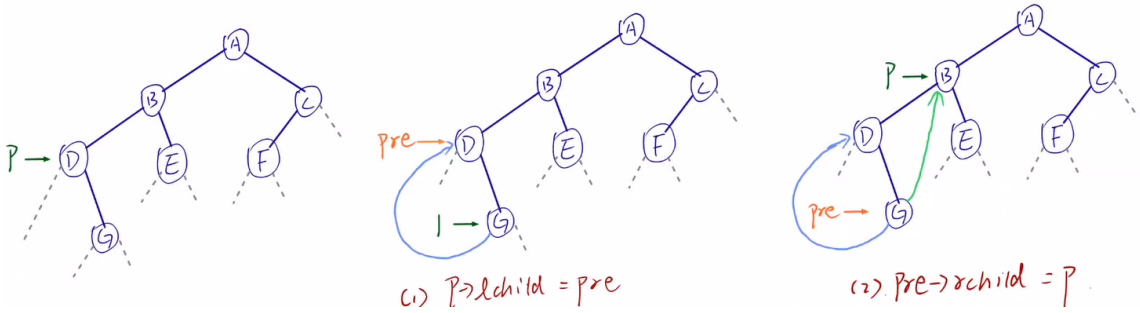
2、中序线索二叉树的构造

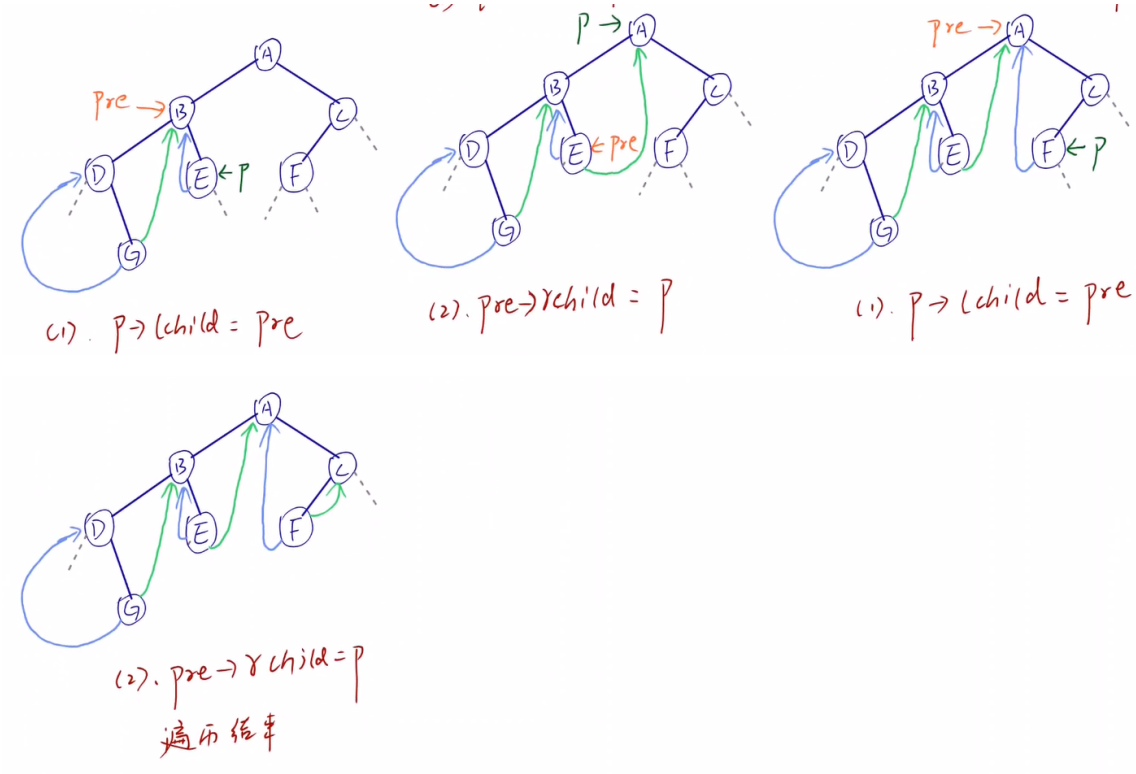


二叉树的线索化是将二叉链表中的空指针改为指向前驱或者后继的线索。而前驱或后继的信息只有在遍历时才能够得到，因此**二叉树的线索化的本质就是遍历一次二叉树。**



以下是二叉树中序线索化的一个示例过程：

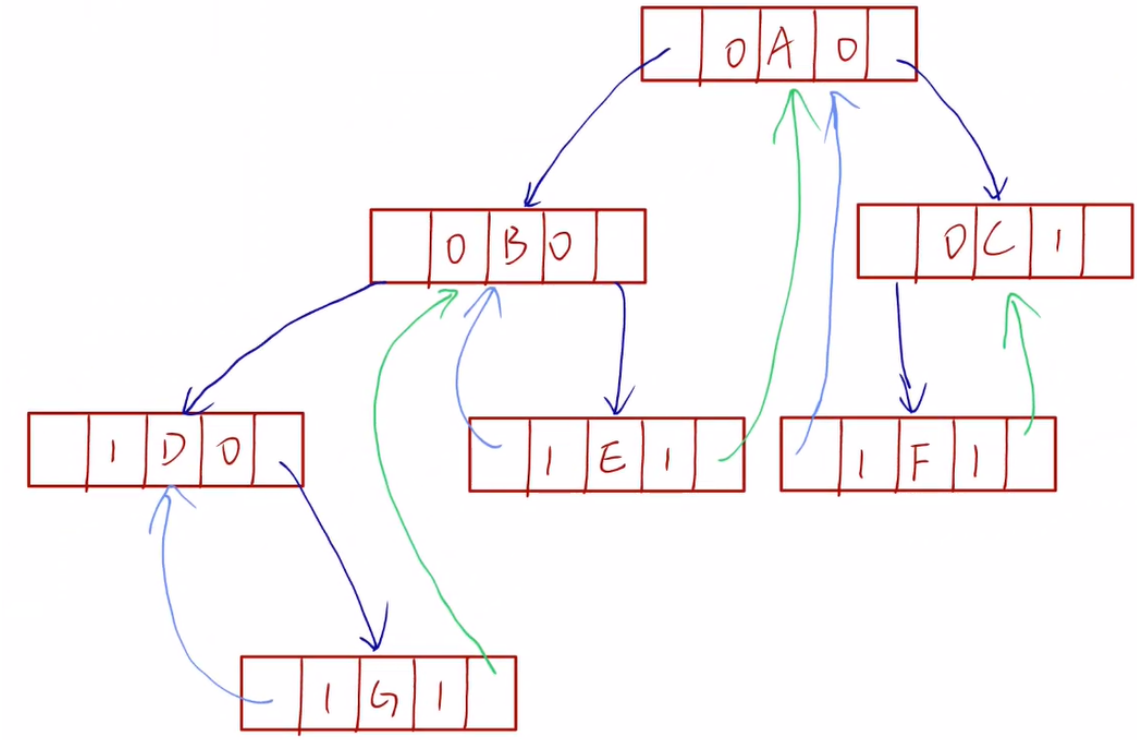




其代码实现如下：

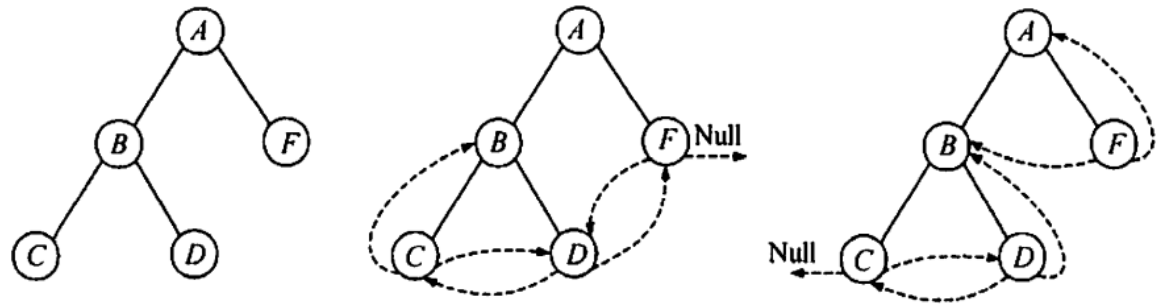
```
// 中序线索化二叉树
void InThread(ThreadBiTree &p, ThreadBiTree &pre) {
    if (p != NULL) {         // 若p非空，结点没有全部遍历
        InThread(p->lchild, pre);    // 递归调用
        if (p->lchild == NULL) {     // 若p的左孩子为空
            p->lchild = pre;     // p的左孩子指向前驱
            p->ltag = 1;         // 标记为线索
        }
        if (pre != NULL && pre->lchild == NULL) {    // pre存在且右孩子为空
            pre->lchild = p;     // pre的右孩子指向后继
            pre->rtag = 1;       // 标记为线索
        }
        pre = p;                 // pre指向p的上一个位置
        InThread(p->rchild, pre);    // 对右孩子建立线索
    }
}
```

线索化后，存储结构如下：



4、先序和后序遍历

先序和后序遍历的方法类似中序遍历，这里不再给出具体流程。



(a) 一棵二叉树　　(b) 先序线索二叉树　　(c) 后序线索二叉树

5、在线索二叉树中查找前驱、后继

中序线索二叉树找到指定结点 *p 的中序后继 next：

1. 若 `p->rtag==1`，则 `next = p->rchild`；
2. 若 `p->rtag==0`，则 next 为 p 的右子树中最左下结点。

中序线索二叉树找到指定结点 *p 的中序前驱 pre：

1. 若 `p->ltag==1`，则 `pre = p->lchild`；
2. 若 `p->ltag==0`，则 next 为 p 的左子树中最右下结点。

先序线索二叉树找到指定结点 * p 的先序后继 next：

1. 若 `p->rtag==1`，则 `next = p->rchild`；
2. 若 `p->rtag==1`，则 `next = p->rchild`；
    1. 若 p 有左孩子，则先序后继为左孩子；
    2. 若 p 没有左孩子，则先序后继为右孩子。

先序线索二叉树找到指定结点 *p 的先序前驱 pre：

1. 前提：改用三叉链表，可以找到结点 * p 的父节点。
2. 如果能找到 p 的父节点，且 p 是左孩子：p 的父节点即为其前驱；
3. 如果能找到 p 的父节点，且 p 是右孩子，其左兄弟为空：p 的父节点即为其前驱；
4. 如果能找到 p 的父节点，且 p 是右孩子，其左兄弟非空：p 的前驱为左兄弟子树中最后一个被先序遍历的结点；
5. 如果 p 是根节点，则 p 没有先序前驱。

> **先序遍历中，每个子树的根节点是最先遍历到的，若根节点有左右孩子，则其左右指针都指向了孩子，这种情况下，没有办法直接找到子树根节点的前驱。**

后序线索二叉树找到指定结点 *p 的后序前驱 pre：

1. 若`p->ltag==1`，则`pre = p->lchild`；
2. 若`p->ltag==0`：
    1. 若 p 有右孩子，则后序前驱为右孩子；
    2. 若 p 没有右孩子，则后续前驱为右孩子。

后序线索二叉树找到指定结点 *p 的后序后继 next：

1. 前提：改用三叉链表，可以找到结点 * p 的父节点。
2. 如果能找到 p 的父节点，且 p 是右孩子：p 的父节点即为其后继；
3. 如果能找到 p 的父节点，且 p 是左孩子，其右兄弟为空：p 的父节点即为其后继；
4. 如果能找到 p 的父节点，且 p 是左孩子，其右兄弟非空：p 的后继为右兄弟子树中第一个被后序遍历的结点；
5. 如果 p 是根节点，则 p 没有后序后继。

> **后序遍历中，每个子树的根节点是最后遍历到的，若根节点有左右孩子，则其左右指针都指向了孩子，这种情况下，没有办法直接找到子树根节点的后继。**

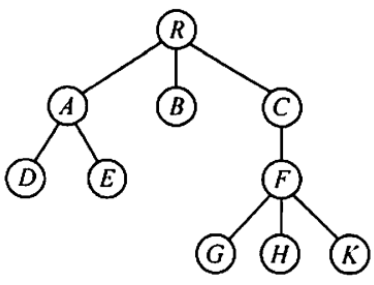
> 【考点】二叉树线索化之后，仍不能有效求解的问题：
>
> 1. 查找后序线索二叉树的后续后继
> 2. 查找先序线索二叉树的先序前驱

5.4 树和森林

5.4.1 树的存储结构

1、双亲表示法（顺序存储）

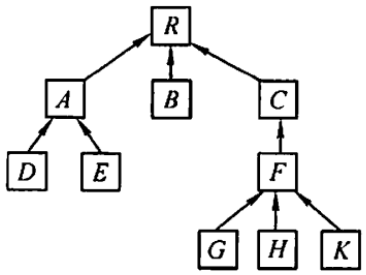
采用一组**连续空间**来存储每个节点，同时在每个节点中设置一个**伪指针**，指示其**双亲结点**在数组中的位置。



(a) 一棵树

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

(b) 双亲表示



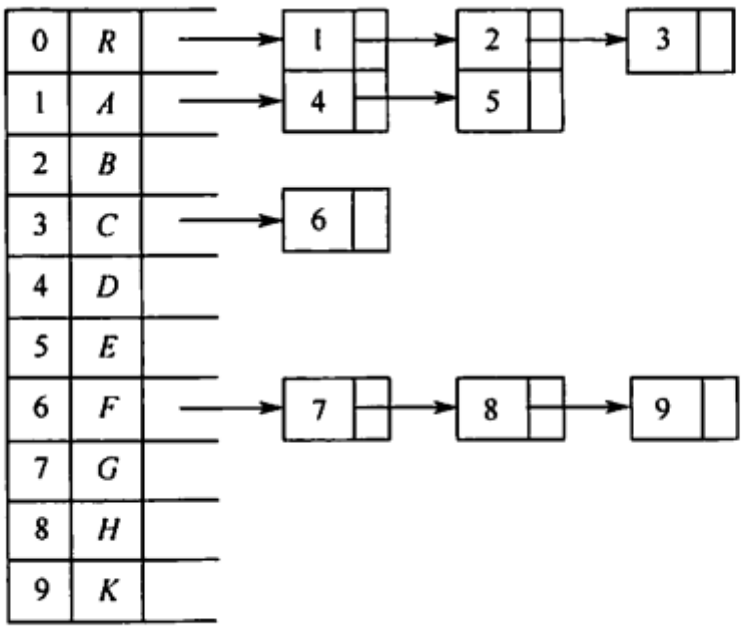
(c) 双亲指针图示

- 根结点固定存储在0号位置，-1表示其没有双亲。
- 插入结点时只需在空白位置添加一行即可。（与二叉树的顺序存储不同）
- 树的顺序存储结构中，数组下标只代表结点的编号，不表示各个结点间的关系。

> 优点：查找指定节点的**双亲**很方便
>
> 缺点：
>
> 1、查找指定节点的**孩子**只能从头开始遍历；
>
> 2、空数据导致结点的遍历更慢。

2、孩子表示法（顺序+链式存储）

- 孩子表示法中，每个结点的孩子都使用了单链表链接起来形成一个线性结构，这时n个结点就有n个孩子链表（叶节点的孩子链表为空表）。
- 这种存储方式寻找子女的操作非常直接，而寻找双亲的操作需要遍历n个结点中孩子链表指针域所指向的n个孩子链表。

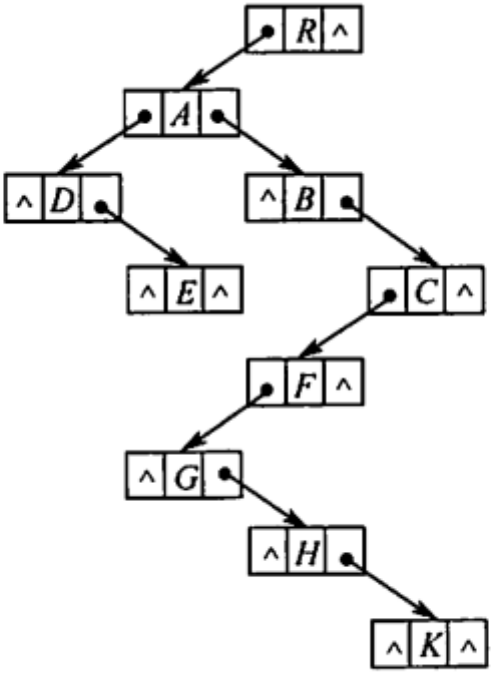


(a) 孩子表示法

3、孩子兄弟表示法（链式存储）

孩子兄弟表示法又称二叉树表示法，即以二叉链表作为树的存储结构。孩子兄弟表示法使每个结点包括三部分内容:结点值、指向结点第一个孩子结点的指针，及指向结点下一个兄弟结点的指针（沿此域可以找到结点的所有兄弟结点)。

这种存储表示法比较灵活，其最大的优点是**可以方便地实现树转换为二叉树的操作**，**易于查找结点的孩子**等,但缺点是**从当前结点查找其双亲结点比较麻烦**。若为每个结点增设一个parent域指向其父结点，则查找结点的父结点也很方便。



(b) 孩子兄弟表示法

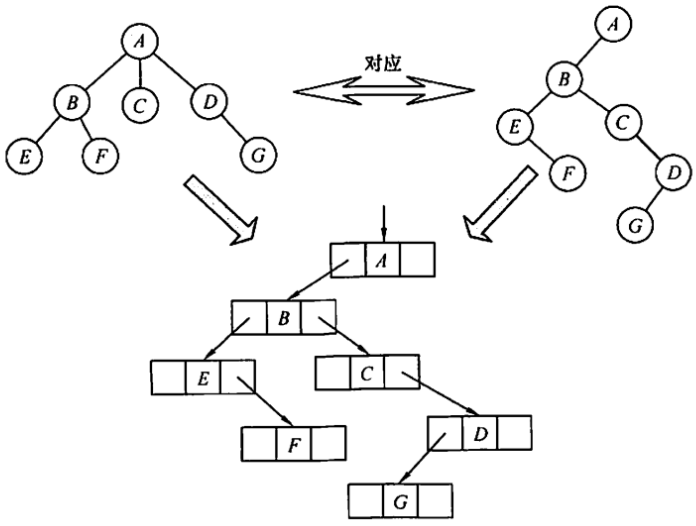
5.4.2 树、森林和二叉树的转换

1、树和二叉树的转化

树转换二叉树的原则：每个结点的左指针指向它的第一个孩子，右指针指向它在树中的相邻右兄弟。

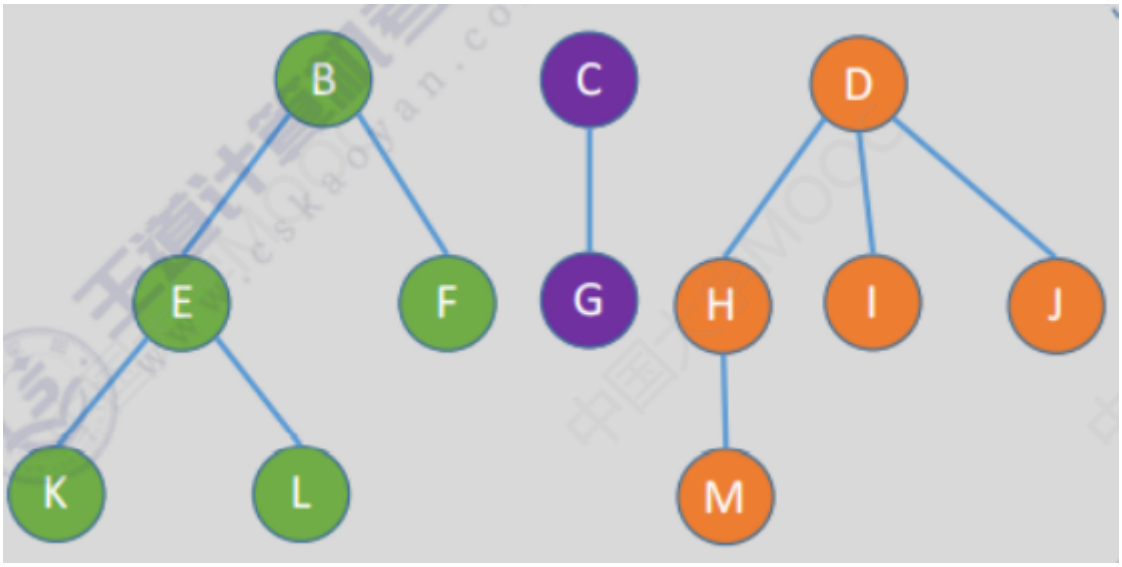
> 记忆："**左孩子右兄弟**"

由于根节点没有兄弟，**所以树转化成的二叉树没有右子树。**

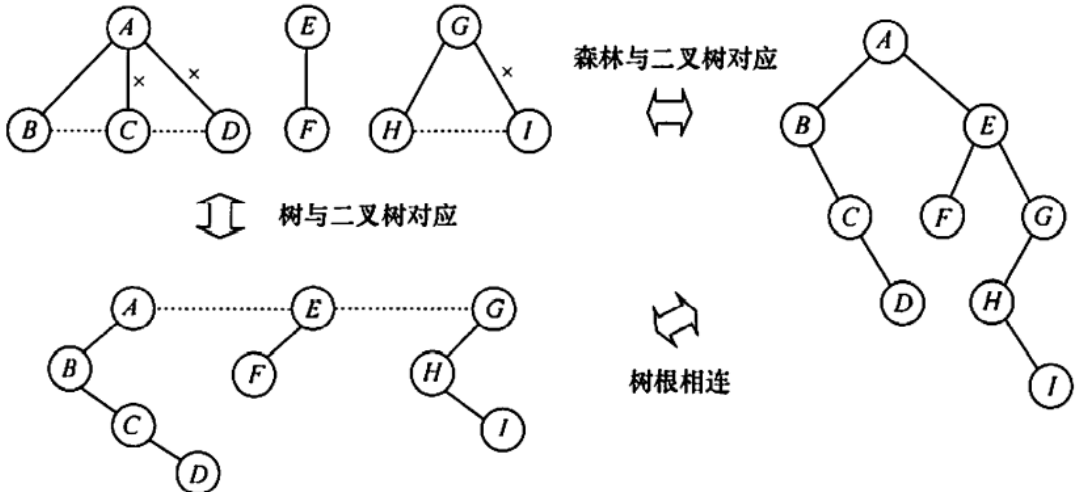


2、树和森林

森林是m (m≥0）棵互不相交的树的集合。



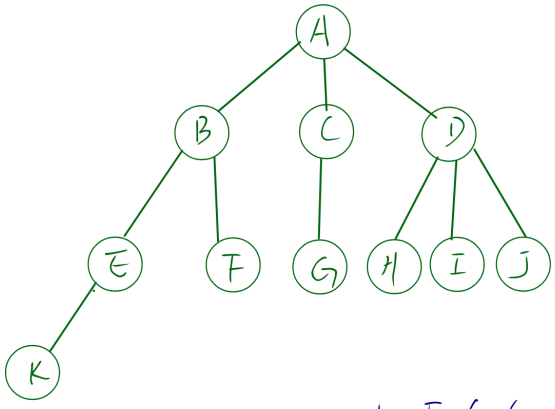
将森林转化成二叉树：先将森林中的每棵树转换为二叉树，由于任何一棵和树对应的二叉树的**右子树必空**，若**把森林中第二棵树根视为第一棵树根的右兄弟**，即将第二棵树对应的二叉树当作第一棵二叉树根的右子树，将**第三棵树对应的二叉树当作第二棵二叉树根的右子树**，以此类推，就可以将森林转换为二叉树。



5.4.3 树和森林的遍历

1、树的遍历

- **先根遍历。**若树非空，先访问根结点，再依次遍历根结点的每棵子树，遍历子树时仍遵循先根后子树的规则。**其遍历序列与这棵树相应二叉树的先序序列相同。**
- **后根遍历。**若树非空，先依次遍历根结点的每棵子树，再访问根结点，遍历子树时仍遵循先子树后根的规则。**其遍历序列与这棵树相应二叉树的中序序列相同。**
- **层序遍历。**按照层序依次访问各个结点。



先根遍历：A B E K F C G D H I J

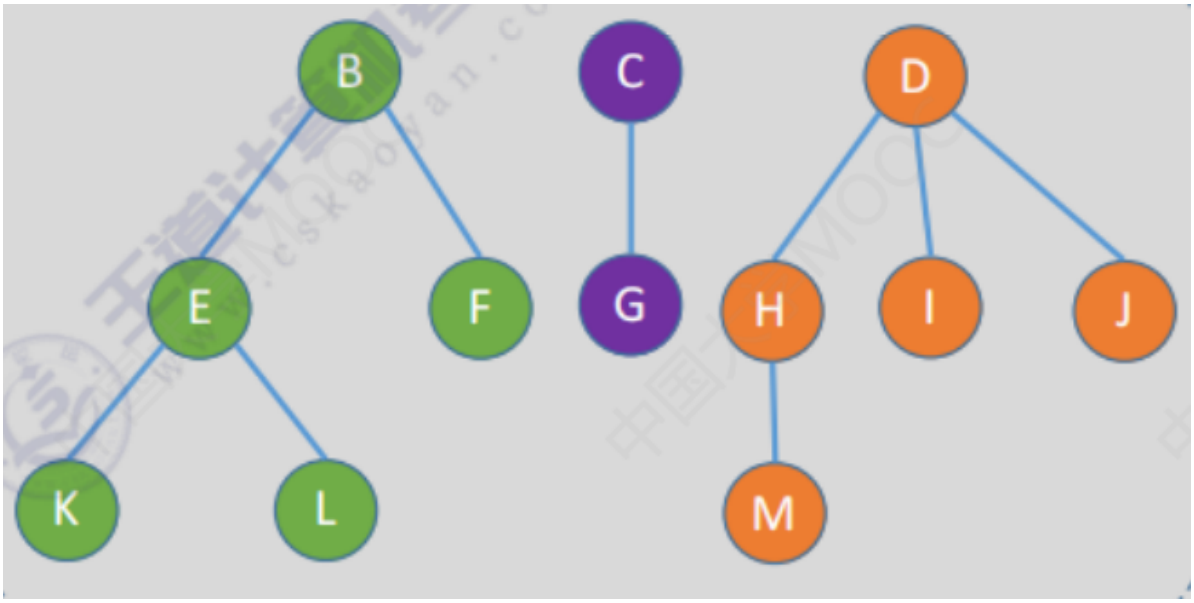
后根遍历：K E F B G C H I J D A

层序遍历：A B C D E F G H I J K

2、森林的遍历

- 先序遍历森林。若森林为非空，则按如下规则进行遍历:
  - 访问森林中第一棵树的根结点。
  - 先序遍历第一棵树中根结点的子树森林。
  - 先序遍历除去第一棵树之后剩余的树构成的森林。

> 效果等同于对各个树依次进行先根遍历，也等同于对对应二叉树进行先序遍历



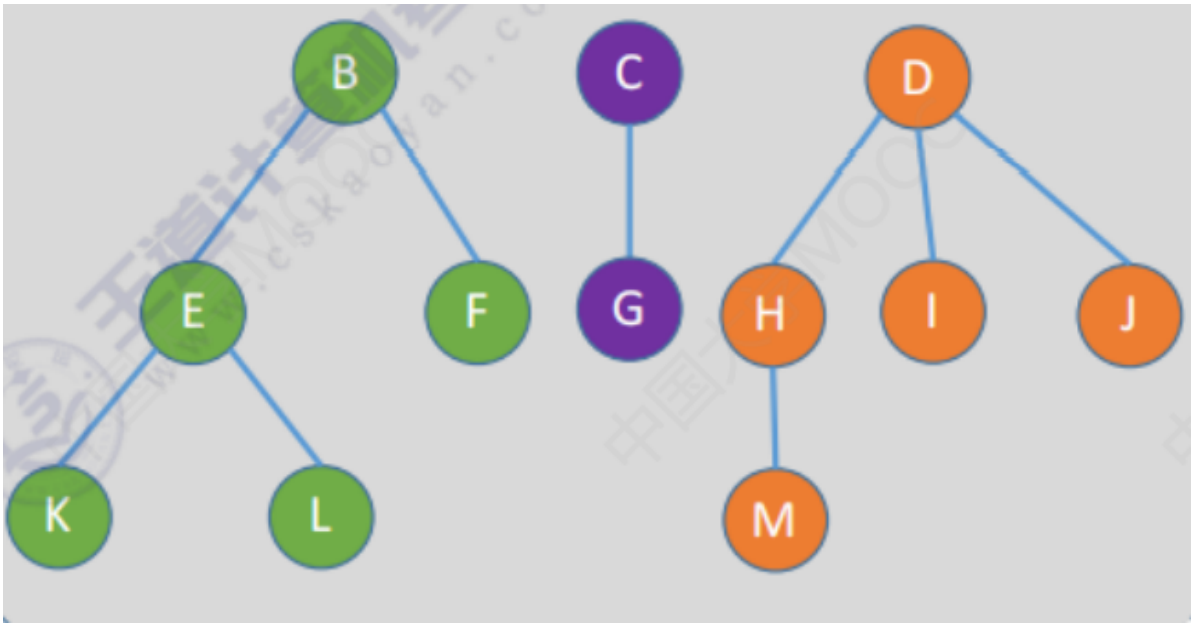
B C D

(B E F) (C G) (D H I J)

(B (E K L) F) (C G) (D (H M) I J)

- 中序遍历森林。森林为非空时，按如下规则进行遍历:
  - 中序遍历森林中第一棵树的根结点的子树森林。
  - 访问第一棵树的根结点.
  - 中序遍历除去第一棵树之后剩余的树构成的森林。

> 效果等同于依次对各个树进行后根遍历，也等同于对对应二叉树进行中序遍历




```
                B       C                D
(       E   F   B)  (G  C)  (       H   I  J  D)
((  K   L   E)  F   B)  (G  C)  ((  M   H)  I  J  D)
```

树和森林的遍历与二叉树遍历的关系

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

5.5 树和二叉树的应用

5.5.1 哈夫曼树和哈夫曼编码

1、哈夫曼树的定义

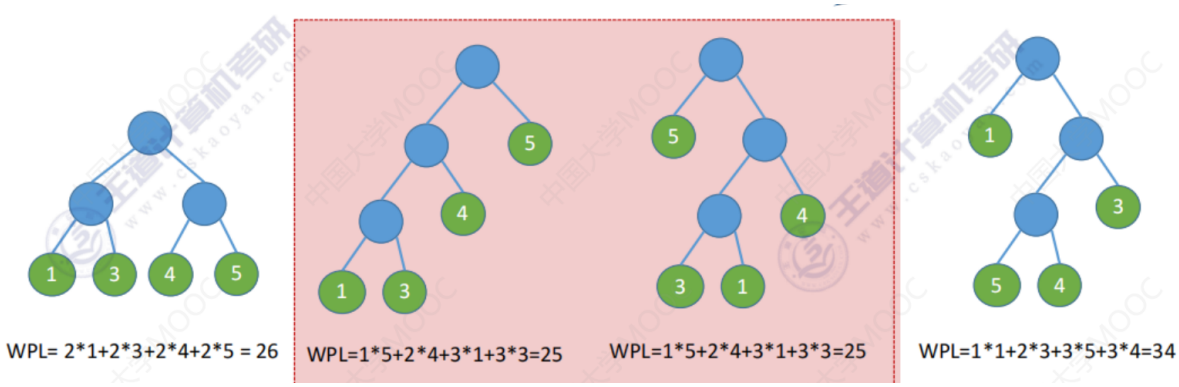
结点的权：有某种现实含义的数值（如:表示结点的重要性等)

结点的带权路径长度：从树的根到该结点的路径长度(经过的边数)与该结点上权值的乘积

树的带权路径长度：树中所有叶结点的带权路径长度之和(WPL, Weighted Path Length)

$$WPL=\sum_{i=1}^{n}w_il_i$$

在含有n个带权叶结点的二叉树中，其中**带权路径长度(WPL)最小的二叉树称为哈夫曼树**，也称**最优二叉树。**

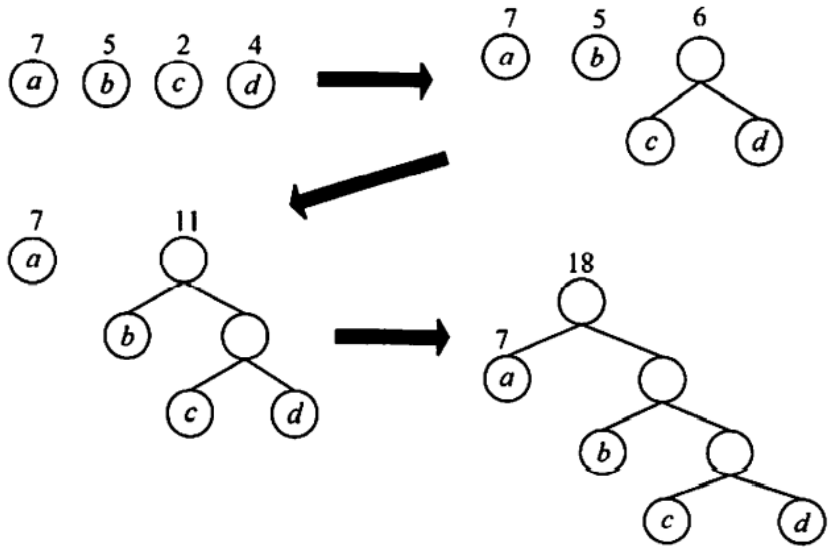


> **哈夫曼树不是唯一的！**

2、构造哈夫曼树

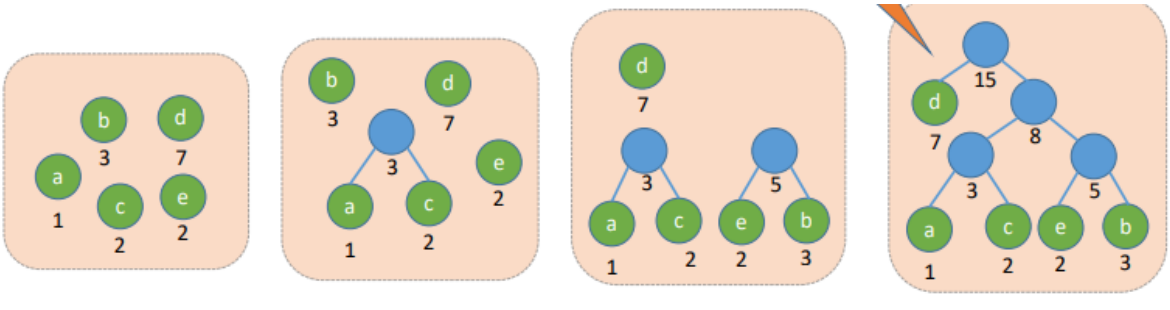
给定n个权值分别为wl, w2,..., wn的结点，构造哈夫曼树的算法描述如下:

1. 将这n个结点分别作为n棵仅含一个结点的二叉树，构成森林F。
2. 构造一个新结点，从F中选取两棵根结点权值最小的树作为新结点的左、右子树，并且将新结点的权值置为左、右子树上根结点的权值之和。
3. 从F中删除刚才选出的两棵树，同时将新得到的树加入F中。
4. 重复步骤2和3，直至F中只剩下一棵树为止。



构造哈夫曼树的注意事项:

1. 每个**初始结点最终都成为叶结点**，且**权值越小的结点到根结点的路径长度越大。**
2. 哈夫曼树的**结点总数为 2n−1。**
3. 哈夫曼树中**不存在度为 1 的结点。**
4. 哈夫曼树并不唯一，但 **WPL 必然相同且为最优。**



3、哈夫曼编码

将字符频次作为字符结点权值，构造哈夫曼树，即可得哈夫曼编码，可用于数据压缩

> 前缀编码：没有一个编码是另一个编码的前缀
>
> 固定长度编码：每个字符用相等长度的二进制位表示
>
> 可变长度编码：允许对不同字符用不等长的二进制位表示