第5章 树与二叉树（上）

郑新、徐鹏飞、李健
北京师范大学人工智能学院
2025-2026学年 第一学期

考核要点

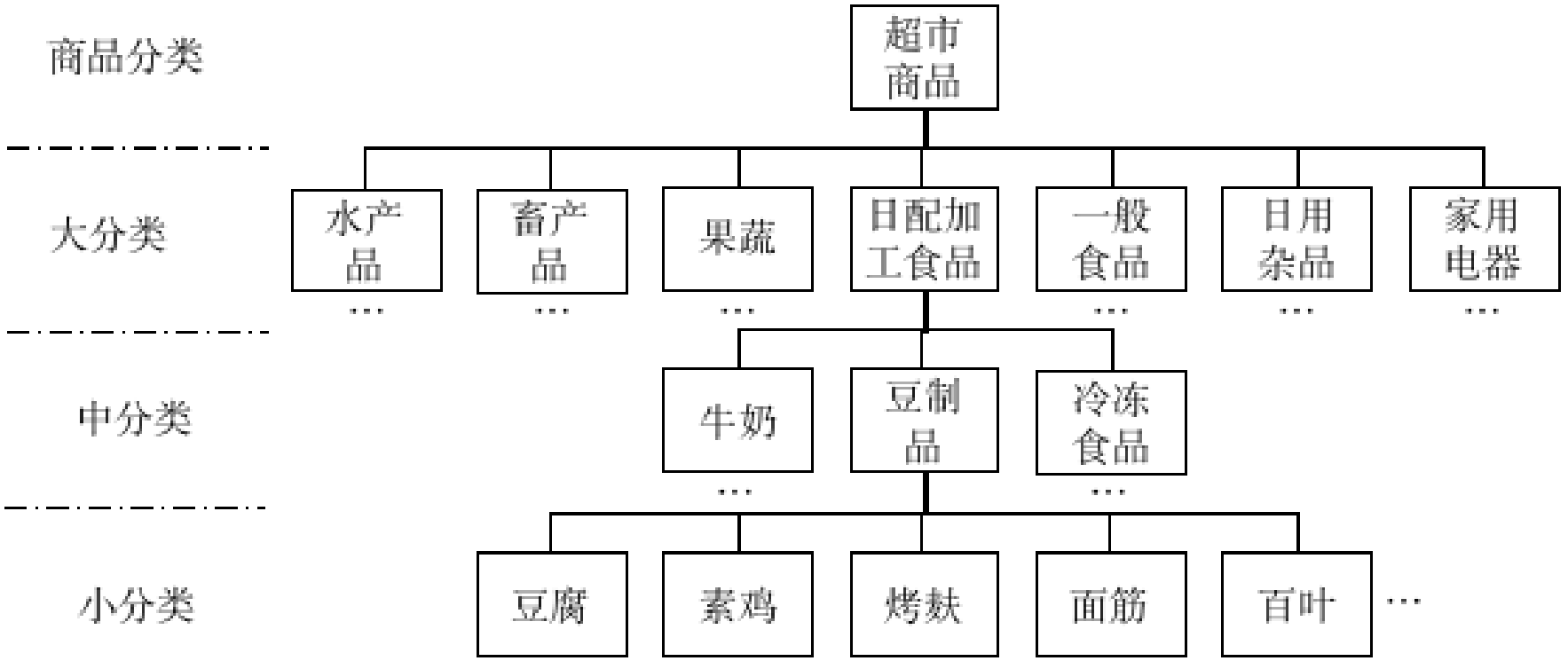
- **考核大纲**
  - 树的基本概念及存储结构
  - 二叉树的定义、 主要特征及其存储结构
  - 二叉树的遍历
- **复习要点**
  - 理解树与二叉树的基本概念（A）
  - 了解二叉树的顺序实现，熟练掌握二叉树的链接实现（A）
  - 熟练掌握完全二叉树的顺序实现（A）
  - 掌握二叉树的遍历（A）
  - 灵活运用二叉树解决计算机中的表达式处理问题（B）
  - 掌握重构（A）

提纲

- 5.1 问题引入：商品分类
- 5.2 树的定义与结构
- 5.3 二叉树
- 5.4 二叉树的遍历

5.1 问题引入：商品分类

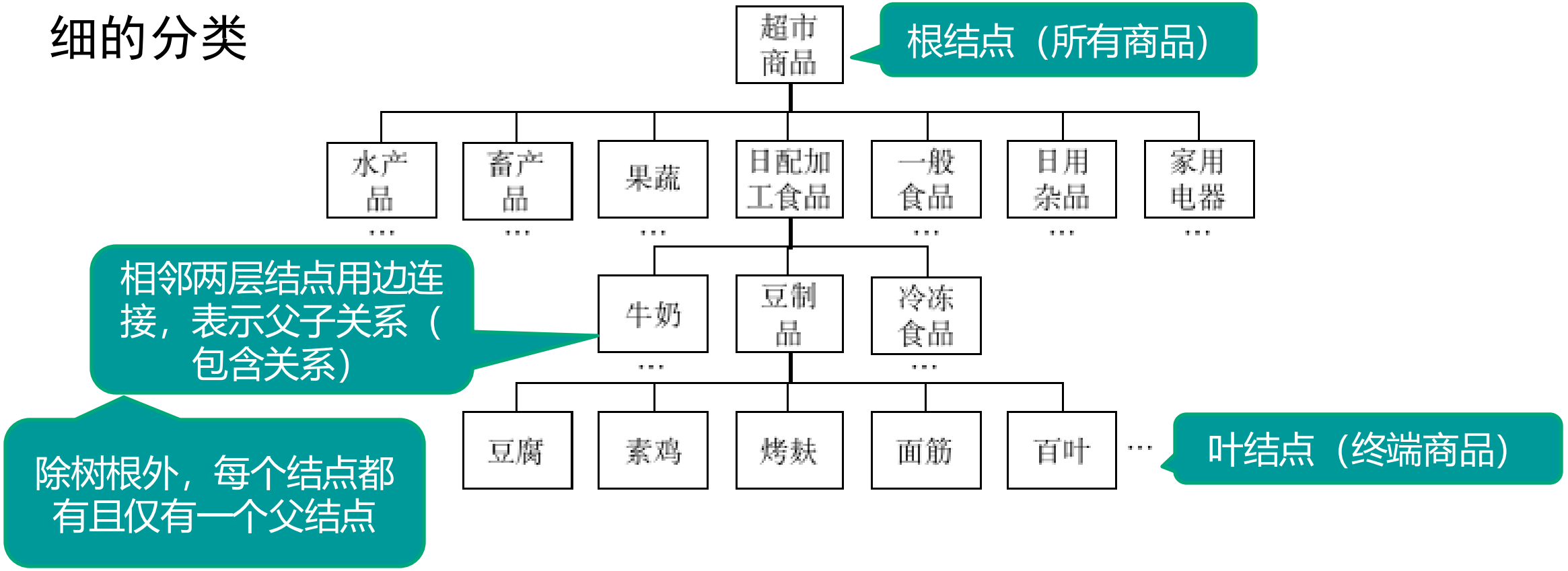
- **问题**：如何方便了解超市商品有哪些类别？每个类别可分为哪些更小的类别，以及属于该类别的终端商品有哪些？
- **关键**：如何展示商品不同类别之间的层次关系（包含关系）
- **商品分类树**：



5.1 问题引入：商品分类

- **商品分类树**

结构与倒立的树相似，展示对数以千万计的商品由上至下、由粗到细的分类



5.1 问题引入：商品分类

- **商品分类树**

结构与倒立的树相似，展示对数以千万计的商品由上至下、由粗到细的分类

- **问题**

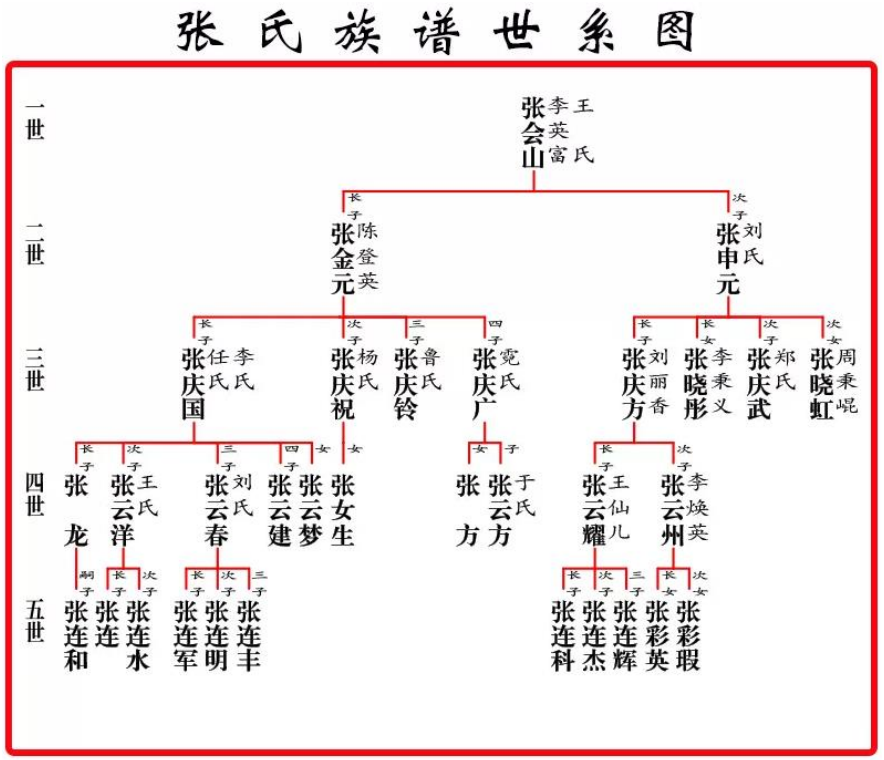
如何在计算机上表达商品分类树？能否用顺序表、链表等线性结构存储树？如何在计算机上查询每个分类包含哪些商品？如何判断两个分类之间是否有包含关系？

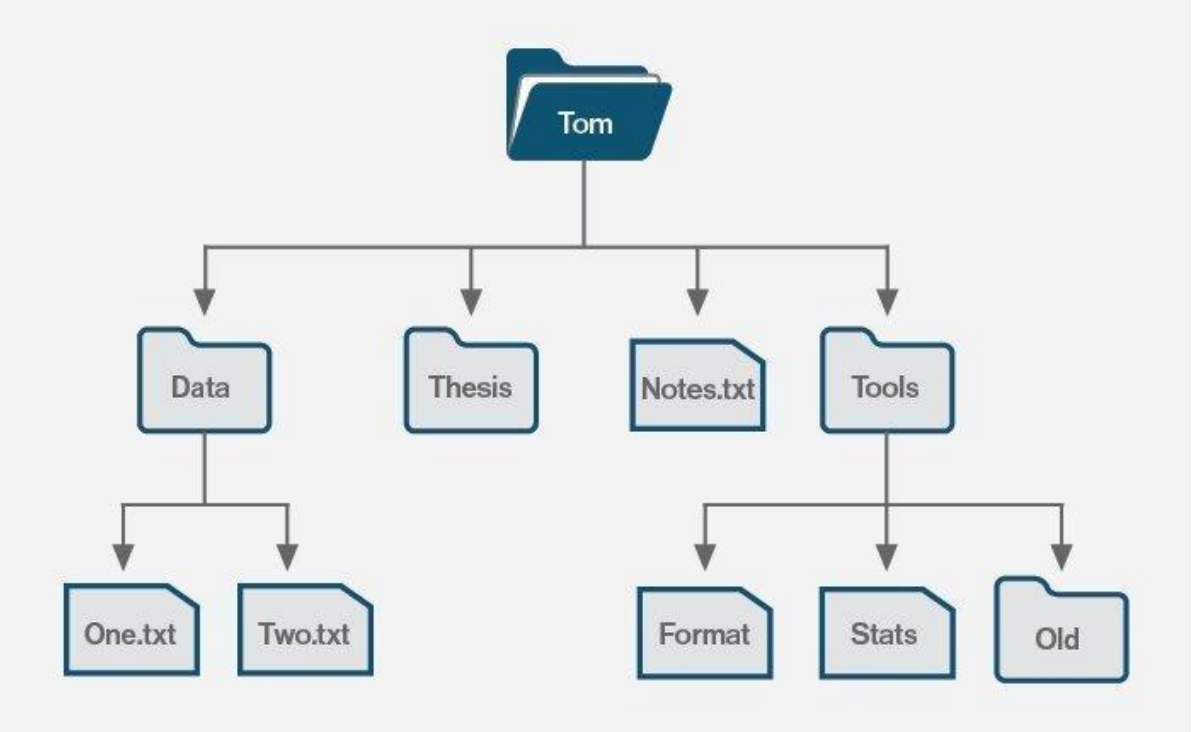
- **主要内容**

树与二叉树的定义、存储实现、遍历方式及其典型应用

5.1 问题引入：商品分类

- 其他举例？
  - 族谱、计算机文件系统、图书分类、行政区划、组织架构等





提纲

- 5.1 问题引入：商品分类
- 5.2 树的定义与结构
- 5.3 二叉树
- 5.4 二叉树的遍历

5.2 树的定义与结构

- **树的定义**：树是由结点组成的有限集合T，用|T|表示结点的数量。
  树具备以下性质：
  - （1） |T|=0，T是空树。
  - （2） |T|>0，非空树
    - T中有且仅有一个特殊结点$r \in T$，称为树根；
    - 其它结点T−{r}划分为m(≥0)个互不相交的子集$T_1, T_2, \ldots, T_m$，每个子集T_i $(i \in [1, m])$也是树，称为根结点r的子树；
    - 如果$|T_i|>0$，子树的根$r_i \in T_i$是r的子结点，r是r_i的父结点。

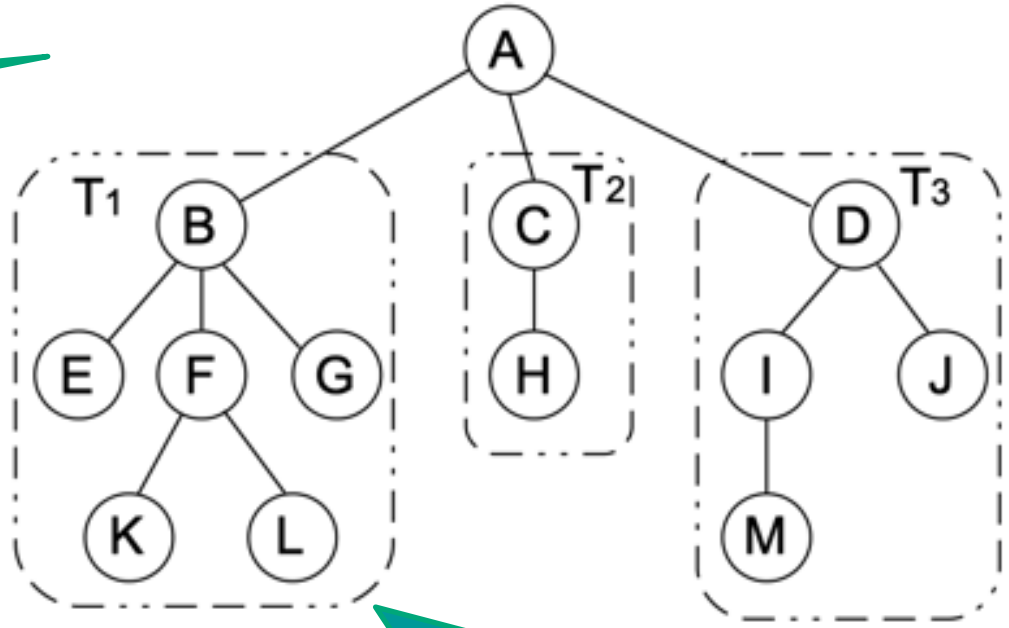
递归定义

- 父子关系属于二元关系 $<r, r_i>$
- 由于根结点没有父结点，而其它所有结点都有一个且仅一个父结点，所以由n个结点构成的树共包含n-1对父子关系
- 如果用边连接有父子关系的结点，则树中共有n-1条边

5.2 树的定义与结构

- 树的结构示例

树 T = {A, B, C, D, E, F, G, H, I, J, K, L, M}
- 树根：A
- 子树：T_1 = {B, E, F, G, K, L}，B为根
  T_2 = {C, H}, C为根
  T_3 = {D, I, J, M}, D为根
- 父子关系: <A, B>、<A, C>、<A, D>

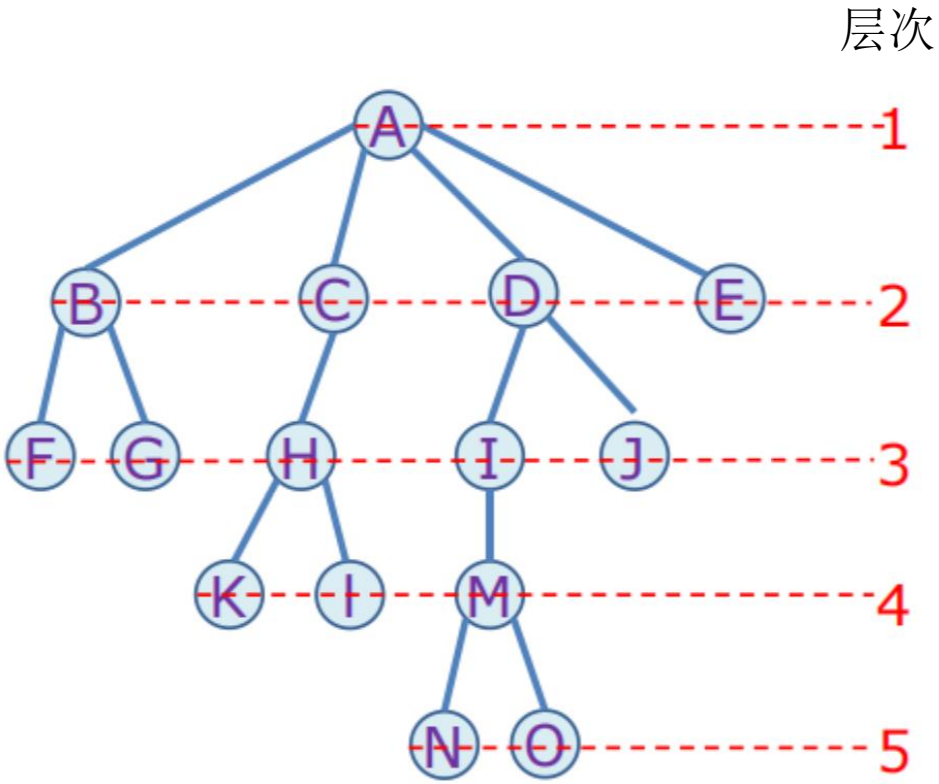


- 子树是不相交的
- 除了根结点外，每个结点有且仅有一个父结点
- 一颗包含N个结点的树，有N-1条边

T_1也是树（递归定义）
- 树根：B
- 子树：{E}, {F, K, L}, {G}
- 父子关系：<B, E>, <B, F>, <B, G>

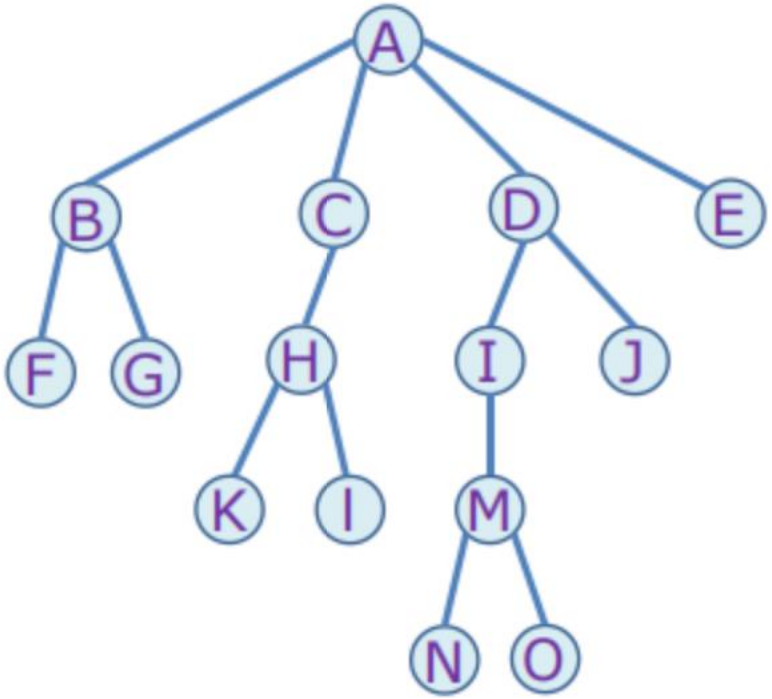
5.2 树的定义与结构

- 树的基本术语
  - 结点（Node）：树中的元素，结点是树的基本构成部分（A、B、C）
  - 结点的度（Degree）：结点非空子树个数（A的度为4，B的度为2）
  - 树的度：所有结点度的最大值称为树的度（右边树的度为4）；树中所有结点的度的和等于结点数-1，
  - 结点的层次（Level）：规定根结点在第1层，其他任意结点的层数是其父结点的层数加1（根节点A的层数为1，结点N的层数为5）
  - 树的深度/高度（Depth/Height）：树中所有结点中的最大层次是这棵树的深度（右边树的深度为5）



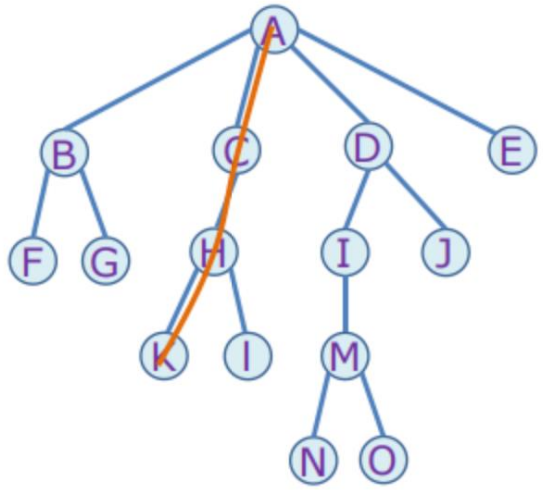
5.2 树的定义与结构

- 树的基本术语
  - 根结点（Root）：树中唯一没有入边（前驱）的结点（A）
  - 叶子结点（Leaf）：度数为0的结点（F、G、K、I、N、E...）
  - 森林（Forest）：m个不相交的树的集合（删除A后的四棵子树的集合）
  - 父结点（Parent）：又称为双亲。若某结点有子树，则该结点称为子树的父结点。（B是F、G的父结点，D是I、J的父结点）
  - 子结点（Child）：若A是B的父结点，则称B是A的子节点



5.2 树的定义与结构

- 树的基本术语
  - 兄弟结点（Sibling）：具有同一父结点的各结点彼此是兄弟结点
    （BCDE是兄弟、 FG是兄弟、 H没有兄弟、 IJ是兄弟）
  - 祖先结点（Ancestor）：沿树根到某一结点路径上的所有结点都是该节点的祖先结点（ACH都是K的祖先，ADIM都是O的祖先）
  - 子孙节点（Descendant）：某一结点的子树中的所有结点是该结点的子孙（树中所有的结点都是根节点A的子孙， HKI是C的子孙）
  - 路径（Path）：从某结点沿树中的边可到达另一个结点，则两个结点之间的边的集合称为路径，路径所包含边的个数称为路径的长度。
    （AK的路径为AC->CH->HK，其长度为3，AN的长度为4）



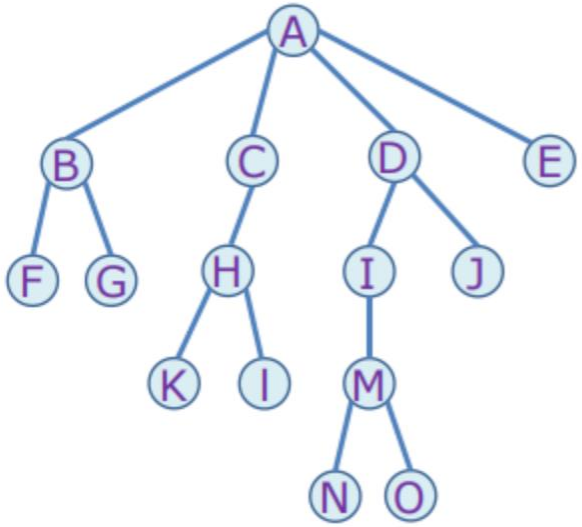
(1) 根结点是其它所有结点的祖先

(2) 如果结点u是结点v的祖先，则存在路径$<r_1, r_2, \ldots, r_n>$，满足$r_1=u$, $r_n=v$ 且 r_i 是 r_{i+1} 的父结点 $(0<i<n)$

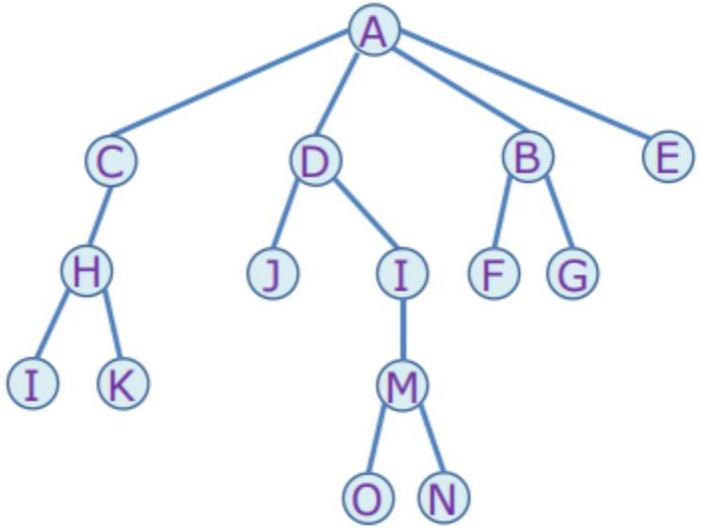
(3) 由于任何非根结点有且仅有一个父结点，从祖先结点u到子孙结点v的路径是唯一的

5.2 树的定义与结构

- 树的基本术语
  - 无序树：树中结点的各子树之间的顺序无关，可以任意交换位置。若T1=T2，则T1是无序树
  - 有序树：树中结点的各棵子树从左到右顺序相关，交换任意两棵子树的位置都将改变原来树的结构。若T1≠T2，则T1是有序树



树 T_1



树 T_2

5.2 树的定义与结构

- 树的基本操作
  - InitTree (tree): 初始化一个空树 tree
  - CreatTree (tree, definition): 按照 definition 构造一个树
  - IsEmpty (tree): 树 tree 为空返回 true，否则返回 false
  - Root (tree): 返回树 tree 的根结点
  - Get (tree, node): 返回树 tree 的结点 node 的值
  - Parent (tree, node): 返回树 tree 中结点 node 的父结点
  - GetChild (tree, node, k): 返回树 tree 中结点 node 的第 k 个子树
  - InsertChild (tree, node, k, subtree): 将树 subtree 插入到树 tree 中，使其成为结点 node 的第 k 个子树
  - Search (tree, x): 在树 tree 中查找值为 x 的结点，如果查找成功，返回结点，否则返回 NIL
  - Traverse (tree): 访问树 tree 中每个结点，且每个结点只访问一次

提纲

- 5.1 问题引入：商品分类
- 5.2 树的定义与结构
- 5.3 二叉树
- 5.4 二叉树的遍历

5.3.1 二叉树的定义

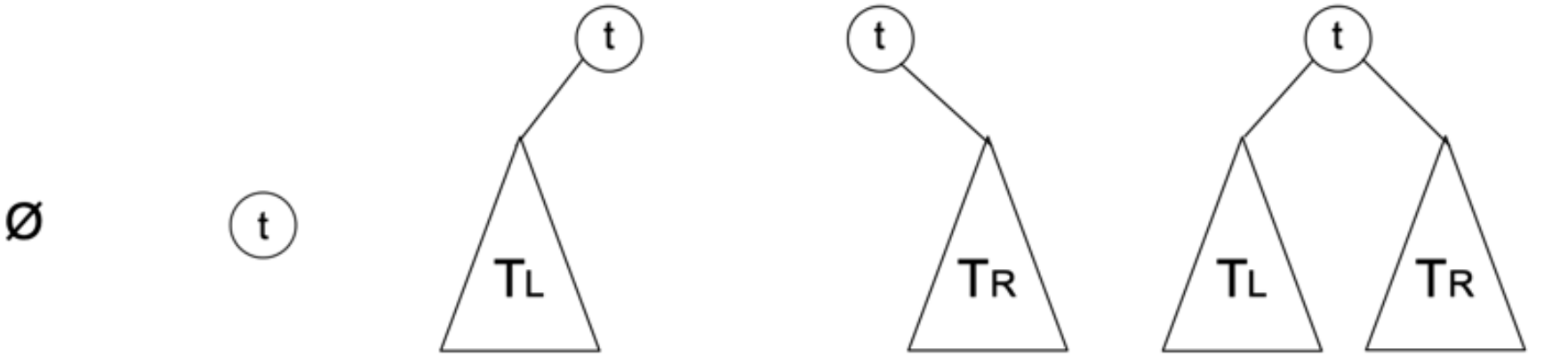
- 二叉树是由结点组成的有限集合T，用|T|表示结点的数量。树具备以下性质：
  - （1） |T|=0，T是空树。
  - （2） $|T|>0$，T中有且仅有一个特殊结点$r \in T$，称为二叉树的根结点；其它结点$T-\{r\}$划分为两个不相交的子集T_L和T_R。T_L是r的左子树，本身是一棵二叉树，如果$|T_L|>0$，T_L的根r_L是r的左子结点，r是r_L的父结点；T_R是r的右子树，也是一棵二叉树，如果$|T_R|>0$，其根r_R是r的右子结点，r是r_R的父结点。

二叉树是应用最广泛的树形结构

二叉树每个结点有且仅有两个子树，并且子树有左右区别（有序树）

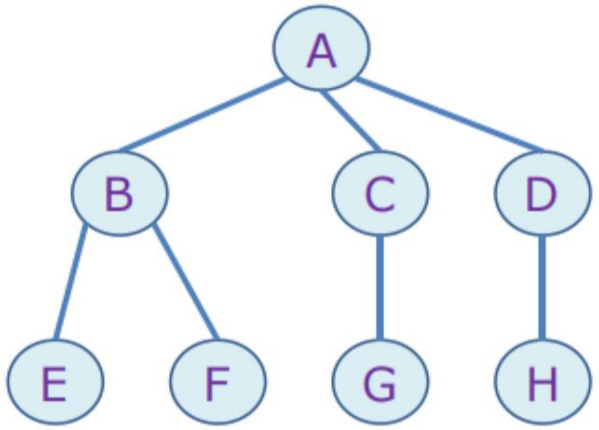
5.3.1 二叉树的定义

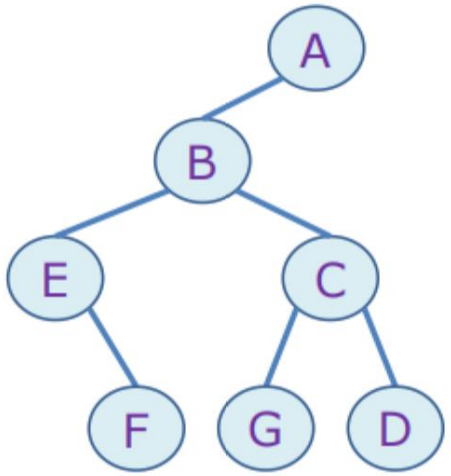
- 二叉树（ Binary Tree） 是结点的有限集合。
  - 该集合可以是空，也可以是由根结点和称为其左子树和右子树的两个不相交结点所构成的两个分支的树。



5.3.1 二叉树的定义

- 树与二叉树的区别
  - 树中结点的子树之间可以是无序的，而二叉树中结点的子树即使只有一棵子树，也要区分左、右子树。
  - 树中每个结点可以有 0 棵或多颗子树，而二叉树的每个结点最多只能有2棵子树。



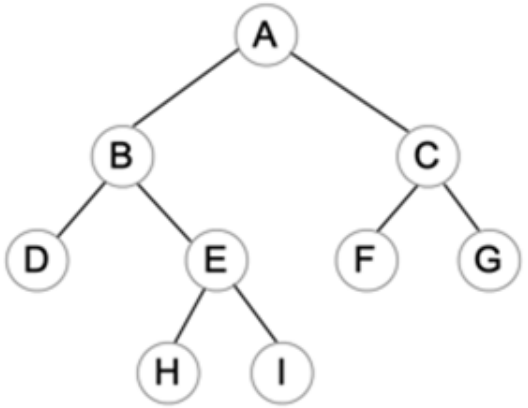


5.3.1 二叉树的定义

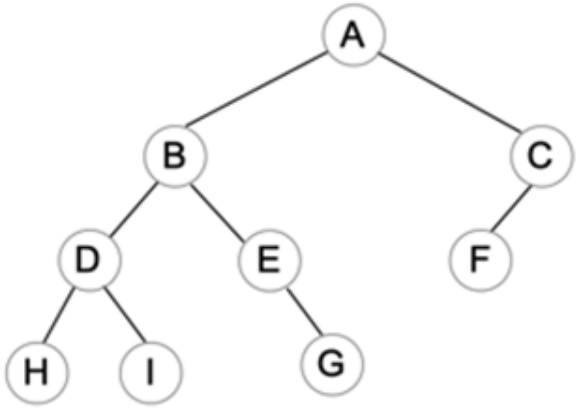
- 二叉树的基本操作
  - BinaryTreeNode (): 创建一个二叉树结点
  - CreatBinaryTree (value, left_tree, right_tree): 构造二叉树，根结点的数据为value，左子树和右子树分别是 left_tree 和 right_tree
  - IsLeaf (tree, node): 如果二叉树 tree 中结点 node 为叶结点，返回 true；否则返回 false
  - Height (tree): 返回二叉树 tree 的高度（深度）
  - PreOrder (tree): 前序遍历二叉树 tree
  - InOrder (tree): 中序遍历二叉树 tree
  - PostOrder (tree): 后序遍历二叉树 tree
  - LevelOrder (tree): 层序遍历二叉树 tree

5.3.2 满二叉树、完全二叉树、完美二叉树

- 满二叉树（Full binary tree）：由度为0的叶结点和度为2的中间结点构成的二叉树，树中没有度为1的结点



满二叉树



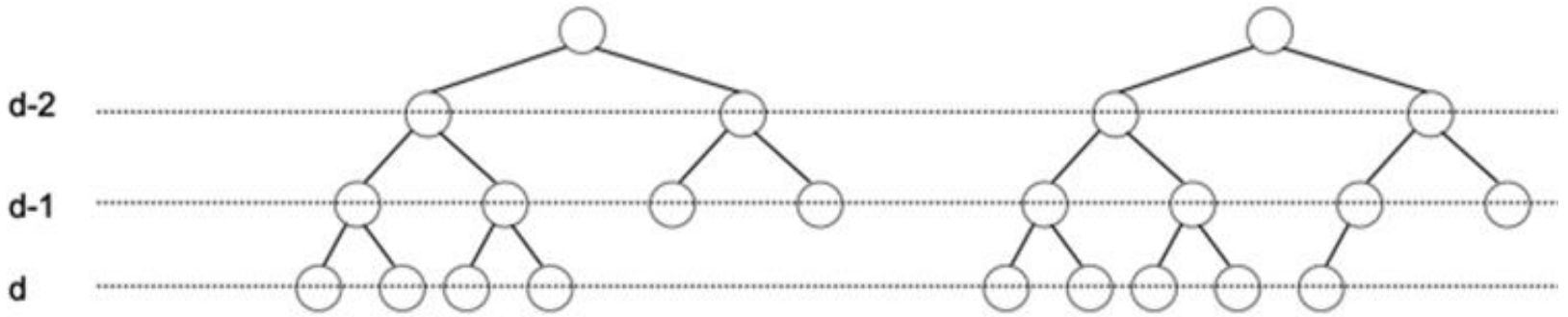
非满二叉树

形态结构特殊、应用广泛的二叉树

满二叉树实例：哈夫曼树、表达式树

5.3.2 满二叉树、完全二叉树、完美二叉树

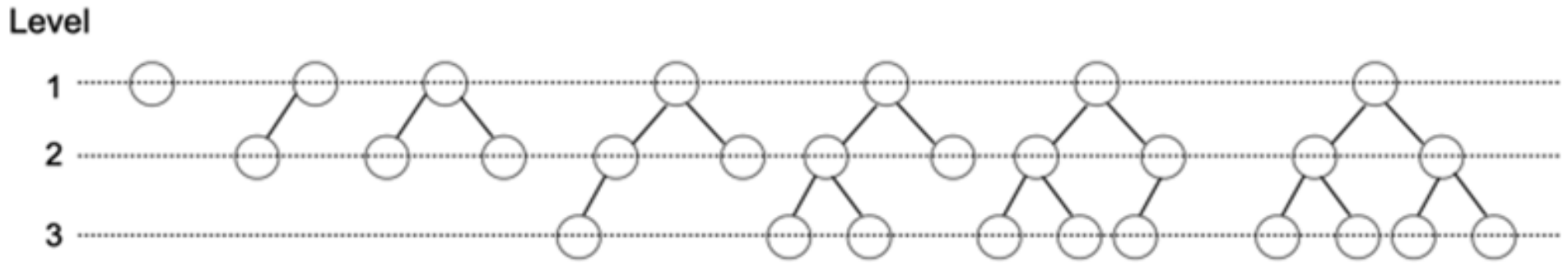
- 完全二叉树（Complete binary tree）
  - (1) 从第1层到第$d-2$层全是度为2的中间结点
  - (2) 第d层的结点都是叶结点，度为0
  - (3) 在第$d-1$层，各结点的度从左向右单调非递增排列，同时度为1的结点要么没有，要么只有一个且该结点的左子树非空



没有度为1的中间结点　　只有一个度为1的中间结点且左子树非空

5.3.2 满二叉树、完全二叉树、完美二叉树

- 完全二叉树

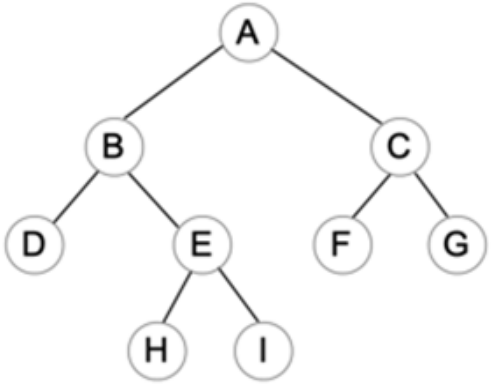


高度$1 \leq d \leq 3$的所有完全二叉树

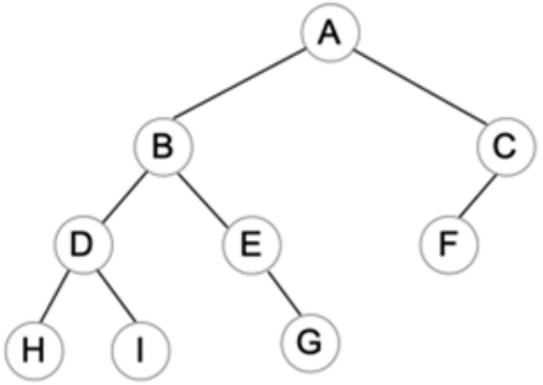
完全二叉树实例：堆

5.3.2 满二叉树、完全二叉树、完美二叉树

- 非完全二叉树示例



理由：度为0的结点D在度为2的结点E的左边，不满足条件(3)

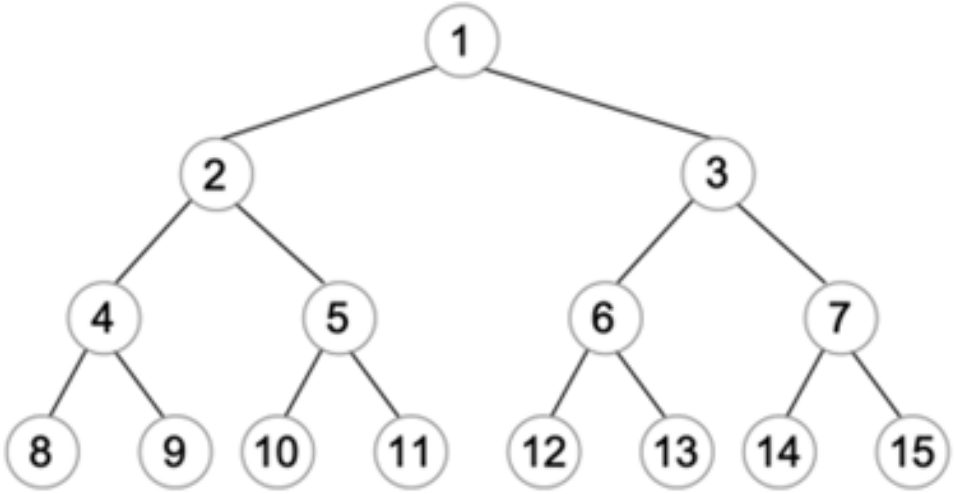


理由1： 结点C的度为1，不满足条件(1)

理由2：结点E的度为1，但左子树为空，不满足条件(3)

5.3.2 满二叉树、完全二叉树、完美二叉树

- 完美二叉树（Perfect binary tree）：对于高度（深度）为$d \geq 1$的完全二叉树，如果第d-1层所有结点的度都是2，则该树是一个完美二叉树

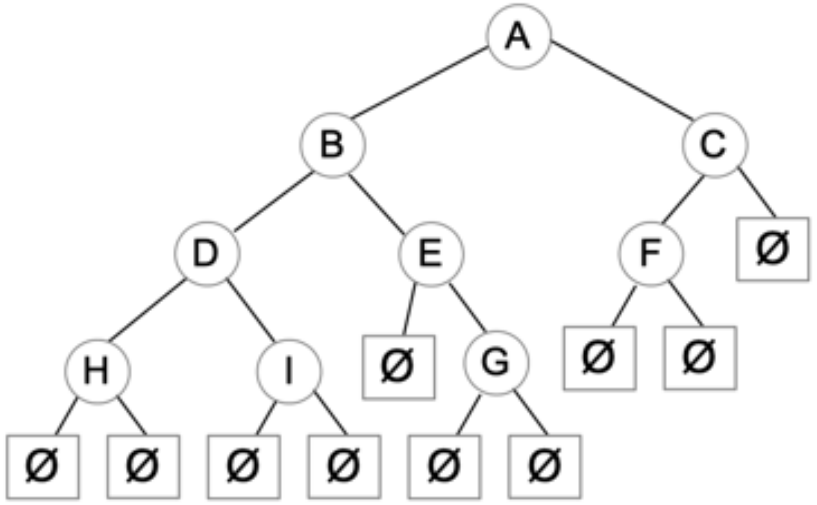


完美二叉树中的所有子树都是完美二叉树

去掉完全二叉树最下层的叶结点，由此生成的树是完美二叉树

5.3.2 满二叉树、完全二叉树、完美二叉树

- 扩充二叉树：在二叉树结点的空子树位置添加特殊的结点：空树叶，形成的二叉树称作扩充二叉树



二叉树中所有结点都被扩充成度为2的中间结点

扩充二叉树是满二叉树，并且所有的叶结点都是空树叶

5.3.3 二叉树基本性质

- **命题5-1：** 设非空二叉树中度为$i \in [0,2]$的结点数为n_i，则$n_0=n_2+1$。

  证明：
  (1) 除根节点以外，每个节点都有父节点，因此共有$n-1$子节点
  (2) 非空二叉树的结点总数 $n = n_0 + n_1 + n_2$
  (3) 由于每个非空节点有且仅有1个父节点，不同中间节点不重叠，因此子节点都总数为$n_1 + 2n_2$！ (思考)
  (4) 根据(1)和(2), 子结点数目：$n - 1 = n_0 + n_1 + n_2 - 1 = 2n_2 + n_1$
  综合上述分析，可得 $n_0 = n_2 + 1$，命题得证。

- **定理5-1：** （满二叉树定理）非空满二叉树中叶结点数等于中间结点数加1。

  证明：满二叉树没有度为1的中间结点，由命题5-1直接得证

5.3.3 二叉树基本性质

- **命题5-2：**二叉树的第i层最多有 2^{i-1} 个结点 $(i \geq 1)$ 。

证明： 第1层只有根结点，即2^0个结点。

假设第$i-1$层有2^{i-2} 个结点 $(i > 1)$。由于每个结点最多只有两个子结点，则第i层的结点数不会超过第$i-1$层结点数的2倍，即2^{i-1} 个结点。

- **命题5-3：**深度（高度）为$d(\geq 1)$的二叉树最多有 $2^d - 1$ 个结点。

证明：根据命题5-2，二叉树第$i \in [1, d]$层最多有 2^{i-1} 个结点，因此结点数的最大值等于：

$$\sum_{i=1}^{d} 2^{i-1} = 2^d - 1$$

5.3.3 二叉树基本性质

- **定理5-2**：深度（高度）为$d(\geq 1)$的二叉树是**完美二叉树**的充分必要条件是树中有 $2^d - 1$ 个结点。

充分条件

(1) 根据命题5-3，结点总数等于$2^d - 1$ 说明每层的结点数达到最大值

(2) 根据命题5-2，第$i+1$层的结点数是第 i 层的结点数的2倍$(1 \leq i < d)$

(3) 根据(2), 二叉树从第1层到第$d-1$层都是度为2的中间结点

因此，结点数达到最大值的二叉树是完美二叉树

必要条件

(1) 完美二叉树第1层的结点数为2^0

(2) 对所有$1 \leq i < d$，第i层的结点都是度为2的中间结点

(3) 第$i+1$层的结点数是第 i 层的结点数的2倍，即 2^i 个结点

因此，完美二叉树的结点总数：

$$\sum_{i=1}^{d} 2^{i-1} = 2^d - 1$$

5.3.3 二叉树基本性质

- 完全二叉树的分层编号

完全二叉树的结构特征

设完全二叉树的深度（高度）为$d(\geq 1)$

(1) 从第1层到第$d-1$层的结点构成完美二叉树

(2) 第$d-1$层结点的度从左向右非递增排列，并且度为1的结点最多1个且左子树非空

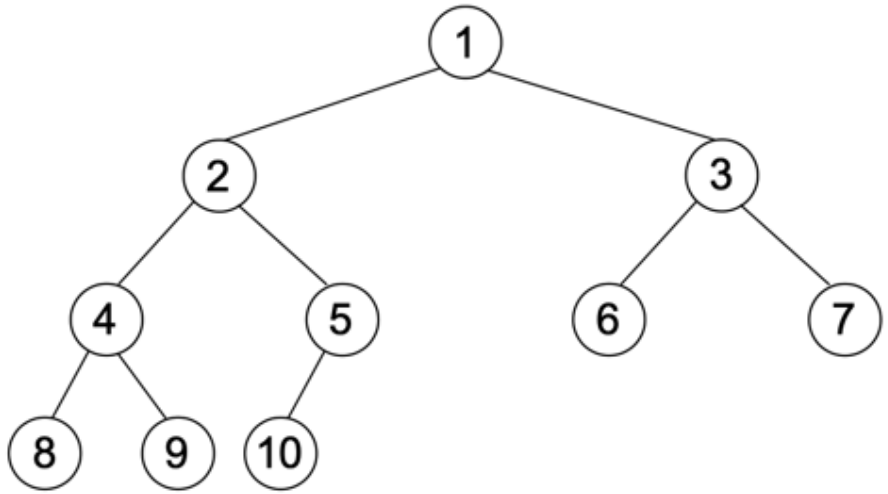
(3) 根据(2)，第d层的叶结点连续集中在最左边

分层编号

(1) 根编号为1

(2) 第$i \in [2, d]$层，左端结点设置为2^{i-1}

(3) 同一层结点，从左向右连续编号



实现对所有结点从上至下、从左向右连续编号！

5.3.3 二叉树基本性质

- **定理5-3**：完全二叉树有n个结点$(n \geq 1)$，按层次从左向右连续编号。树中任一结点k $(1 \leq k \leq n)$满足以下性质
  ① 如果$2k \leq n$，则结点k的左子结点是$2k$，否则没有左子结点；
  ② 如果$2k+1 \leq n$，则结点k的右子结点是$2k+1$，否则没有右子结点。
  ③ 如果$k>1$，则结点k的父结点是$\lfloor k/2 \rfloor$。

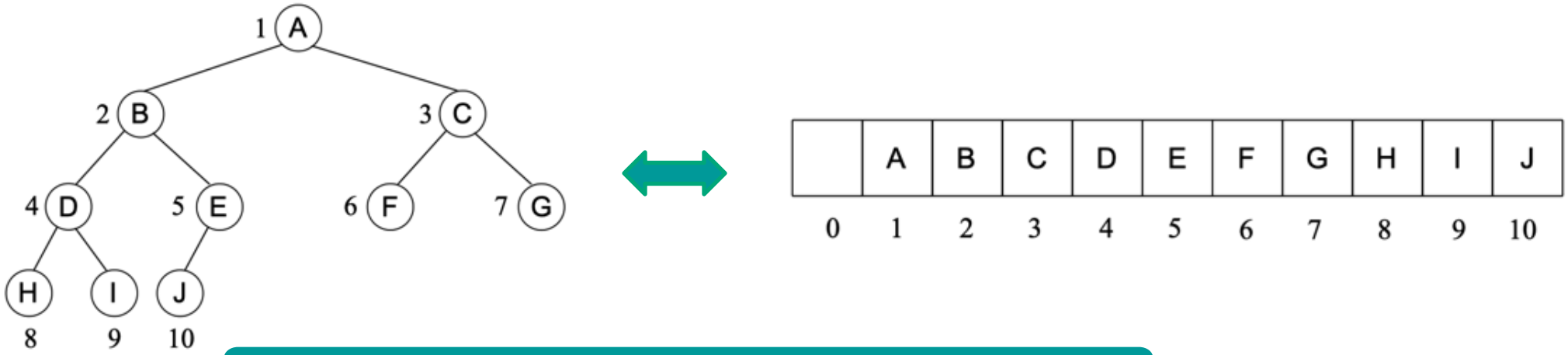
  证明：假设结点k是完全二叉树第i层从左到右第j个结点，$i \in [1, d-1], j \in \left[1, 2^{i-1}\right]$，即$k = 2^{i-1} + j - 1$。
  - $2k = 2^i + 2(j-1)$ 对应于第i+1层的第$2j-1$个结点
  - $2k+1$对应于第i+1层的第$2j$个结点。

- **命题5-4**：有n个结点$(n \geq 1)$的完全二叉树的深度$d = \lceil \log_2(n+1) \rceil$。

  证明：根据定理5-2和命题5-3，$2^{d-1} - 1 < n \leq 2^d - 1$

5.3.4 二叉树的顺序存储实现

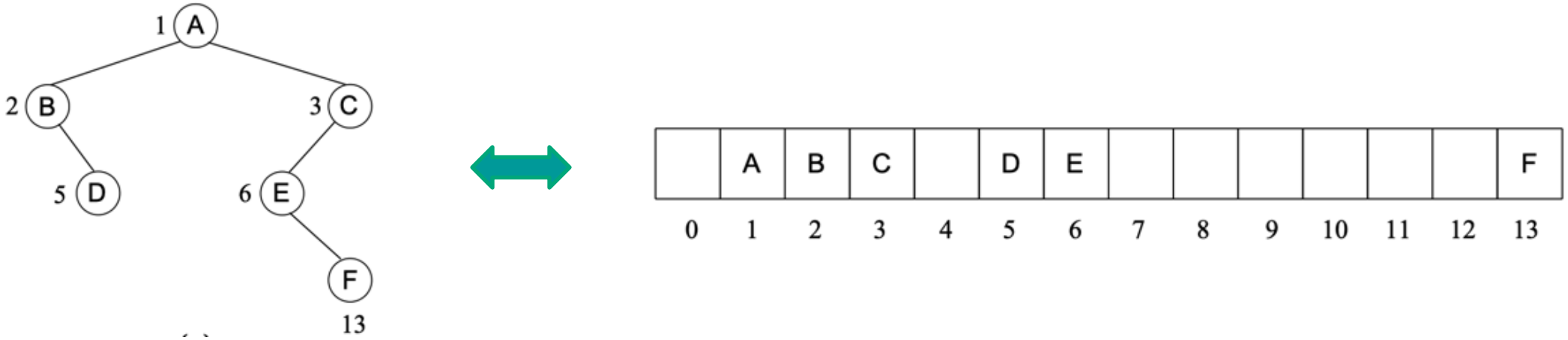
- **二叉树的主要存储方式：**顺序存储 + 链接存储
- **完全二叉树的顺序存储：**完全二叉树所有结点可以分层从左向右连续编号，可用一组地址连续的存储单元（**顺序表**）存储二叉树的各个结点



各结点的索引（编号）与顺序表位置——对应，
结点的数据存放在顺序表相应位置的单元中

5.3.4 二叉树的顺序存储实现

- **非完全二叉树的顺序存储**
  - **结点编号**：树根的索引为1；设结点的编号为k ($k \geq 1$)，如果其左子树非空，则左子结点的编号为$2k$；如果右子树非空，则右子结点为$2k+1$
  - **顺序存放**：用一组地址连续的存储单元存储二叉树的各个结点



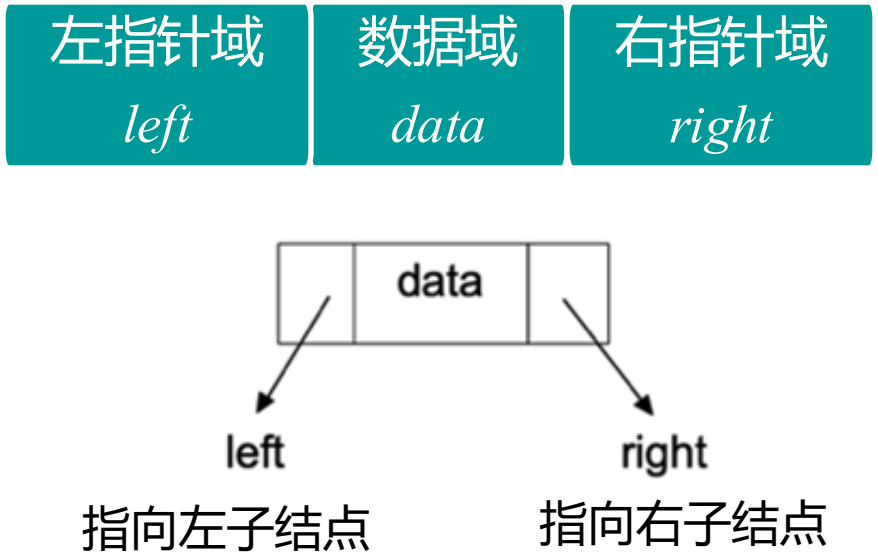
各结点的索引（编号）与顺序表位置——对应，结点的数据存放在顺序表相应位置的单元中

5.3.4 二叉树的顺序存储实现

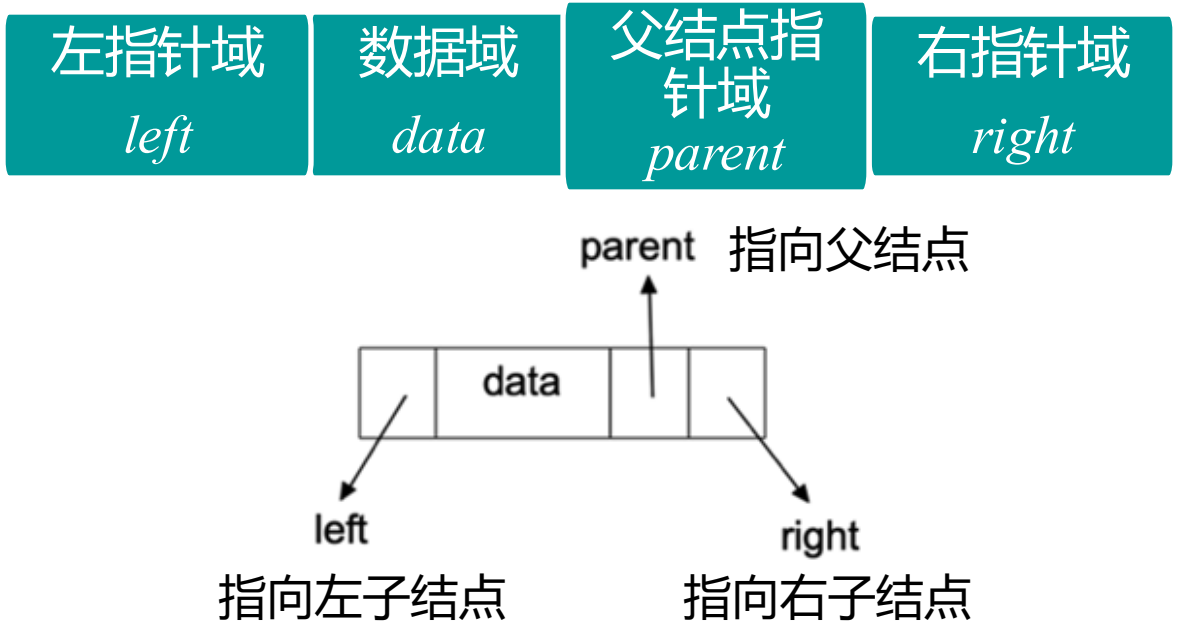
- 优点
  - 只需用顺序表存放结点数据，不需要保存结点间逻辑关系
  - 结点间逻辑关系可通过相对位置确定
  - 对于顺序存储结构第 k 个位置的结点 ($k \geq 1$)，其左右子结点分别存储在第$2k$和第$2k+1$个位置，父结点在位置$\lfloor k/2 \rfloor$。因此查找子结点和父结点只需O(1)的时间
  - 顺序结构是完全二叉树最简单、最节省空间的存储方式，n个结点只需O(n)的空间
- 缺点
  - 对一般的二叉树，可能造成空间浪费
  - 在最坏情况下，存放n个结点的二叉树可能需要长度为$O(2^n)$的顺序表
  - 可用其它序列化的方法降低顺序存储的空间复杂度，但无法通过在顺序表中的相对位置直接确定两个结点是否有父子关系，增加了查询结点间逻辑关系的时间复杂度

5.3.5 二叉树的链接存储实现

- **二叉树的主要存储方式：**顺序存储 + 链接存储
- **链接存储：**使用链表存放二叉树，比顺序存储能更有效地表达二叉树的非线性逻辑结构



二叉链表

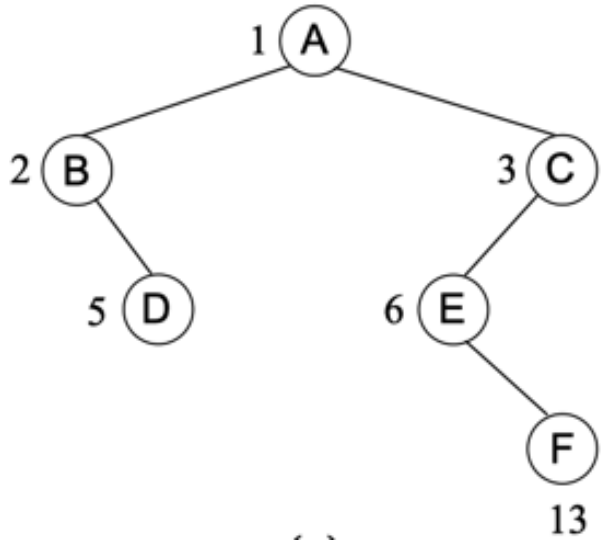


三叉链表

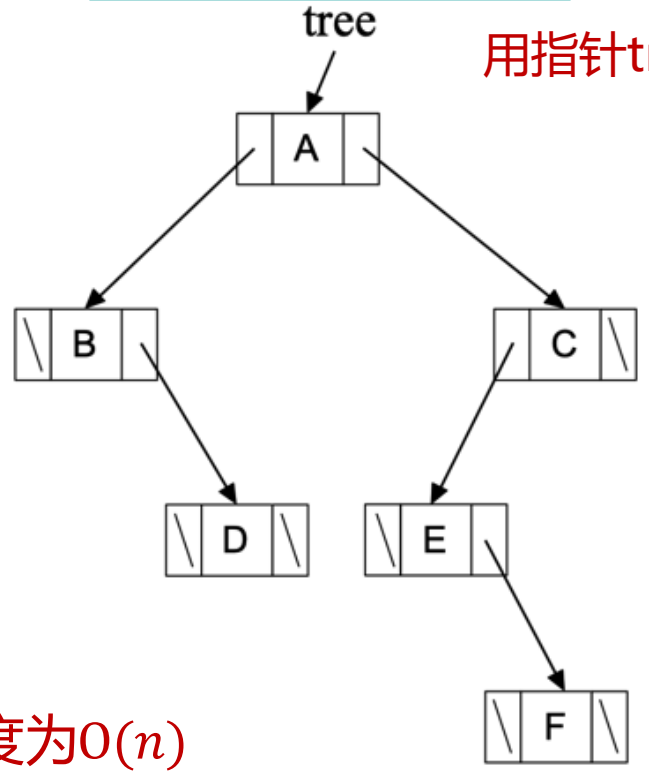
5.3.5 二叉树的链接存储实现

- **链接存储**：使用链表存放二叉树，比顺序存储能更有效地表达二叉树的非线性逻辑结构

逻辑结构



二叉链表存储结构



用指针tree表示二叉树

存放n个结点的二叉树只需n个二叉链表结点，空间复杂度为$O(n)$

5.3.5 二叉树的链接存储实现

- 二叉树的链接存储实现（C语言）

```
typedef int TElemSet;
typedef struct BinaryTreeNode *BinaryTree;
struct BinaryTreeNode {
        TElemSet data; /* 数据元素 */
        BinaryTree left; /* 左孩子指针 */
        BinaryTree right; /* 右孩子指针 */
};
```

5.3.5 二叉树的链接存储实现

- 构造二叉树算法

算法5-1：构造二叉树 CreateBinaryTree(*value*, *left_tree*, *right_tree*)

输入：结点数据*value*, 二叉树*left_tree*和*right_tree*

输出：以*value*为根结点数据、*left_tree*和*right_tree*为左右子树的二叉树

tree ← **new** BinaryTreeNode() //生成新的二叉链表结点

tree.data ← *value* //设置根结点的数据

tree.left ← *left_tree* //设置根的左子树

tree.right ← *right_tree* //设置根的右子树

return *tree* //返回构造的二叉树

5.3.5 二叉树的链接存储实现

```
BinaryTree CreateBinaryTree(TElemSet value, BinaryTree left_tree, BinaryTree right_tree)
{
        BinaryTree tree;
        tree = (BinaryTree)malloc(sizeof(struct BinaryTreeNode));
        tree->data = value;
        tree->left = left_tree;
        tree->right = right_tree;
        return tree;
}
```

提纲

- 5.1 问题引入：商品分类
- 5.2 树的定义与结构
- 5.3 二叉树
- 5.4 二叉树的遍历

5.4.1 二叉树遍历的基本概念

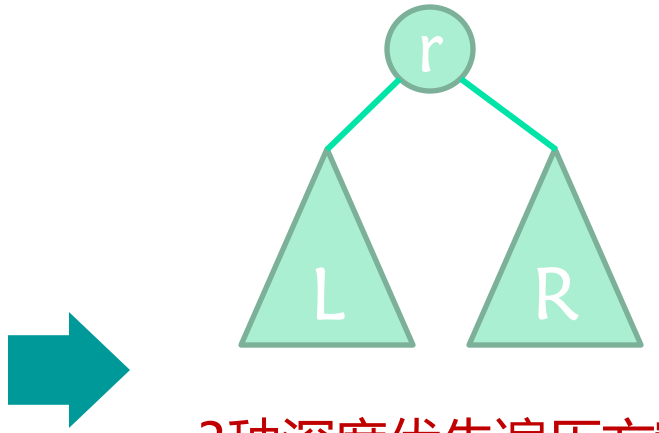
- **遍历的概念**：按预先设定顺序依次访问树中所有结点，并且每个结点仅访问一次
- **深度优先遍历**：按序独立处理二叉树的各个分支单元：根r、左子树L和右子树R

深度优先遍历

(1) **递归思想**：对左子树L和右子树R按相同的方案进行遍历

(2) **深度原则**：独立处理各分支，即在访问完一个分支的所有结点后，才能开始遍历下一个分支

(3) **限制子树的遍历顺序**：对子树的遍历始终按从左向右的顺序进行



3种深度优先遍历方案：

(1) rLR — 前序

(2) LrR — 中序

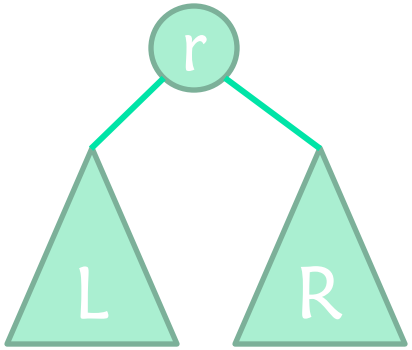
(3) LRr — 后序

- **广度优先遍历**：按层从上至下、从左向右依次访问树中所有结点，

也称作**层序遍历**

5.4.2 二叉树遍历的递归算法

- 深度优先遍历二叉树的三种方案



前序遍历方案

(1) 访问根结点r
(2) 前序遍历左子树L
(3) 前序遍历右子树R

中序遍历方案

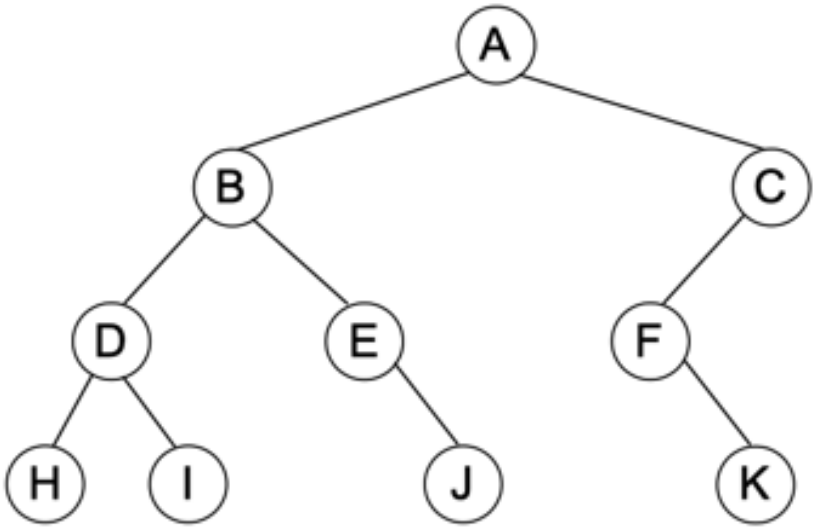
(1) 中序遍历左子树L
(2) 访问根结点r
(3) 中序遍历右子树R

后序遍历方案

(1) 后序遍历左子树L
(2) 后序遍历右子树R
(3) 访问根结点r

5.4.2 二叉树遍历的递归算法

- 遍历二叉树并按序输出结点数据



前序遍历：<A, B, D, H, I, E, J, C, F, K>

中序遍历：<H, D, I, B, E, J, A, F, K, C>

后序遍历：<H, I, D, J, E, B, K, F, C, A>

根结点位置：

- 前序遍历的最前
- 后序遍历的最后
- 中序遍历出现在左子树和右子树遍历结果之间

5.4.2 二叉树遍历的递归算法

- 前序遍历

算法5-2. 前序遍历二叉树 PreOrder(*tree*)
if tree ≠ NIL **then** //空树不做处理，直接返回
| Visit(*tree*) //访问根结点
| PreOrder(*tree.left*) //前序遍历左子树
| PreOrder(*tree.right*) //前序遍历右子树
end

```
void PreOrder(BinaryTree tree)
{
    if (tree != NULL) {
        Visit(tree);
        PreOrder(tree->left);
        PreOrder(tree->right);
    }
}
```

5.4.2 二叉树遍历的递归算法

- 中序遍历

算法5-3. 中序遍历二叉树 InOrder(*tree*)
if *tree* ≠ NIL **then**
| Inorder(*tree.left*) //中序遍历左子树
| Visit(*tree*) //访问根结点
| InOrder(*tree.right*) //中序遍历右子树
end

```
void InOrder(BinaryTree tree)
{
    if (tree != NULL) {
        InOrder(tree->left);
        Visit(tree);
        InOrder(tree->right);
    }
}
```

5.4.2 二叉树遍历的递归算法

- 后序遍历

算法5-4. 后序遍历二叉树 PostOrder(*tree*)
if *tree* ≠ NIL **then**
| Postorder(*tree.left*) //后序遍历左子树
| PostOrder(*tree.right*) //后序遍历右子树
| Visit(*tree*) //访问根结点
end

```
void PostOrder(BinaryTree tree)
{
    if (tree != NULL) {
        PostOrder(tree->left);
        PostOrder(tree->right);
        Visit(tree);
    }
}
```

5.4.2 二叉树遍历的递归算法

- 后序遍历递归算法的应用

算法5-5. 计算二叉树高度 Height(*tree*)
输入： 二叉树*tree*
输出： 二叉树的高度值

```
if tree = NIL then  //空树
|  return 0    //空树的高度为0
else
|  h_left ← Height(tree.left)          //遍历左子树，求子树的高度
|  h_right ← Height(tree.right)     //遍历右子树，求子树的高度
|  return Max(h_left, h_right) + 1  //树的高度等于左右子树高度
                                     // 的较大值加1
end
```

5.4.2 二叉树遍历的递归算法

- 后序遍历递归算法的应用

```
int Height(BinaryTree tree)
{
        int h_left, h_right;
        if (tree == NULL) { /* 空树 */
                return 0;
        }
        else {
                h_left = Height(tree->left); /* 遍历左子树，求子树的高度 */
                h_right = Height(tree->right); /* 遍历右子树，求子树的高度 */
                return Max(h_left, h_right) + 1; /* 树的高度等于左右子树高度的较大值加1 */
        }
}
```

5.4.2 二叉树遍历的递归算法

- **算术表达式**：由常数、运算符和括号组成，有前缀、中缀、后缀三种表示法
  - 中缀表示法：9 + (6 + 3 * 2) / (8 / (5 - 3))
  - 前缀表示法：+ 9 / + 6 * 3 2 / 8 - 5 3
  - 后缀表示法：9 6 3 2 * + 8 5 3 - / / +
- **表达式树**：表示算术表达式（非线性）逻辑结构的二叉树

中缀表达式的递归定义

(1) 单个常数是表达式

(2) 若exp1、exp2是表达式，则 (exp1) op (exp2) 也是表达式 （op表示运算符)

转换

表达式树

(1) 单个常数用单根树表示

(2) 若表达式包含多个常数及运算符，将表达式分解成(expr_left) op (expr_right)

(3) 用根结点表示运算符op，并用左右子树分别表示expr_left和expr_right（递归）

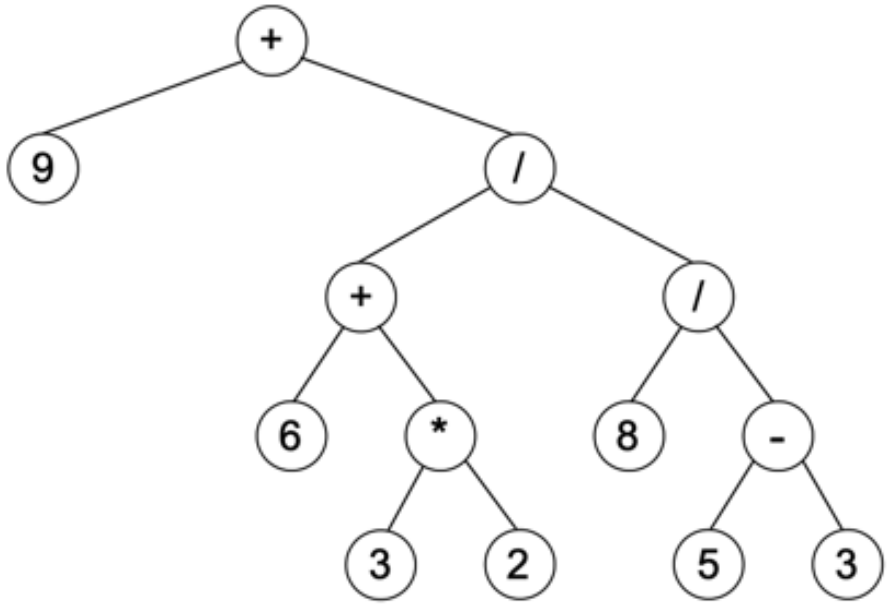
5.4.2 二叉树遍历的递归算法

- 表达式树

9 + (6 + 3 * 2) / (8 / (5 - 3))

中缀表达式

转换



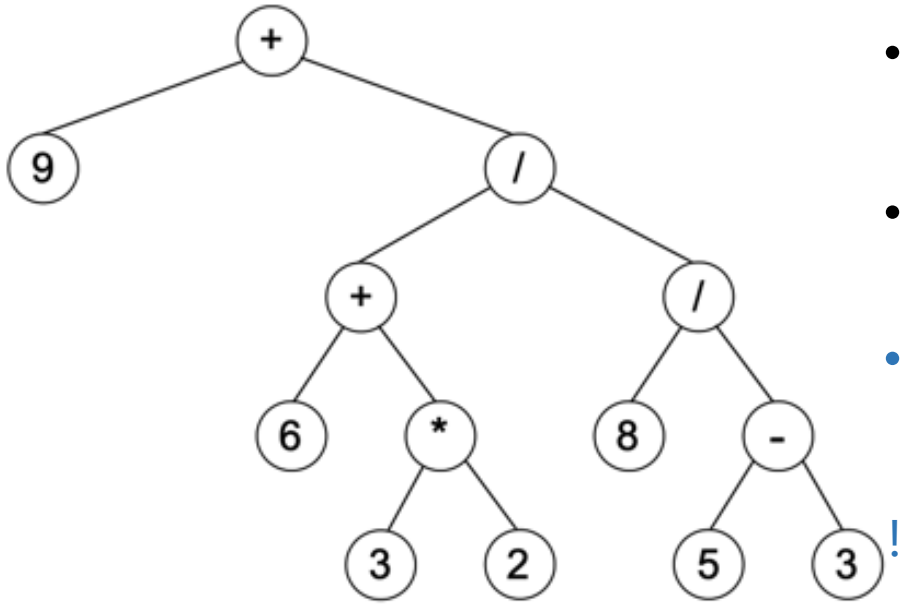
表达式树

- 叶结点都对应常数
- 每个中间结点都存放一个运算符
- 表达式树是满树
- 根结点是表达式中优先级最低的运算符

5.4.2 二叉树遍历的递归算法

- 表达式树的遍历

表达式：9 + (6 + 3 * 2) / (8 / (5 - 3))



- 前序遍历结果：+ 9 / + 6 * 3 2 / 8 - 5 3 = 前缀表示法
- 后序遍历结果：9 6 3 2 * + 8 5 3 - / / + = 后缀表示法
- 中序遍历结果：9 + 6 + 3 * 2 / 8 / 5 – 3 ≠ 中缀表示法

没有括号，不是正确的中缀表达式

需要对左右子树生成的表达式加上括号

5.4.2 二叉树遍历的递归算法

- 表达式树转换成中缀表达式

算法5-6. PrintInfixExpression(*tree*)
输入：非空二叉树*tree*
输出：中缀表达式（含冗余括号）

```
if tree.left ≠ NIL then     //左子树非空
| print（                   //输出左括号
| PrintInfixExpression(tree.left)        //中序遍历左子树并生成表达式
| print ）                  //输出右括号，与前面左括号一起将表达式加上括号
end
print  tree.data            //若结点是叶结点，输出常数；否则，输出运算符
if tree.right ≠ NIL then   //右子树非空
| print（                   //输出左括号
| PrintInfixExpression(tree.right)       //中序遍历右子树并生成表达式
| print ）                  //输出右括号，与前面左括号一起将表达式加上括号
end
```

5.4.2 二叉树遍历的递归算法

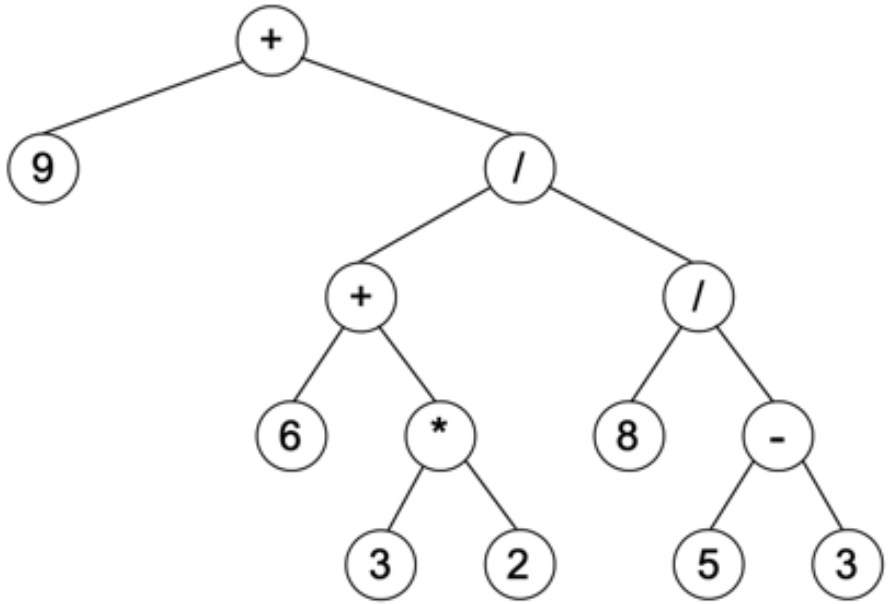
- 表达式树转换成中缀表达式

```
void PrintInfixExpression(ExprTree tree)
{
    if (tree->left != NULL) { /* 左子树非空 */
        printf("(");
        PrintInfixExpression(tree->left); /* 中序遍历左子树并生成表达式 */
        printf(")");
    }
    PrintData(tree); /* 若结点是叶结点，输出运算数；否则，输出运算符 */
    if (tree->right != NULL) { /* 右子树非空 */
        printf("(");
        PrintInfixExpression(tree->right); /* 中序遍历右子树并生成表达式 */
        printf(")");
    }
}
```

5.4.2 二叉树遍历的递归算法

- 表达式树转换成中缀表达式

表达式：9 + (6 + 3 * 2) / (8 / (5 - 3))



- 前序遍历结果：+ 9 / + 6 * 3 2 / 8 - 5 3　　前缀表示法
- 后序遍历结果：9 6 3 2 * + 8 5 3 - / / +　　后缀表示法
- 算法5-6：(9)+(((6)+((3)*(2)))/((8)/((5)-(3))))　中缀表示法

- 通过遍历将表达式树转换为算术表达式的时间复杂度是O(n), n表示结点数目

5.4.3 二叉树遍历的非递归算法

- **递归算法的问题：**
  - (1) 存在不支持递归算法的程序设计语言
  - (2) 递归算法在运行中，需要系统在内存栈中分配空间保存函数的参数、返回地址以及局部变量等，运行效率较低
  - (3)系统对每个进程分配的栈容量有限，如果二叉树的深度太大造成递归调用的层次太高，容易导致栈溢出

- **非递归算法实现的关键：**使用栈结构模拟函数调用中系统栈的工作原理

5.4.3 二叉树遍历的非递归算法

- 算法5-7：非递归前序遍历 PreOrder(*tree*)

```
InitStack(stack)    //初始化栈stack，用于存放结点
while tree ≠ NIL 或 IsEmpty(stack) = false do
| while tree ≠ NIL do        //当前结点不是空结点
| | Visit(tree)              //访问结点
| | Push(stack, tree)        //结点压入栈
| | tree ← tree.left         //沿左分支下移
| end
| if IsEmpty(stack) = false then   //如果栈不为空
| | tree ← Top(stack)
| | Pop(stack)               //弹出栈顶结点
| | tree ← tree.right        //移到栈顶结点的右子树
| end
end
DestroyStack(stack)
```

沿左分支下移，并将经过的结点压入栈

tree=NIL表示左子树为空，或者左子树遍历结束，则从栈中弹出结点，开始遍历结点的右子树

5.4.3 二叉树遍历的非递归算法

- 算法5-7：非递归前序遍历 PreOrder(*tree*)

```
void PreOrder(BinaryTree tree)
{
    Stack stack;
    stack = (Stack)malloc(sizeof(struct StackHeadNode));
    InitStack(stack); /* 初始化栈stack，用于存放结点 */
    while (tree != NULL || IsEmpty(stack)==false) {
        while (tree != NULL) { /* 当前结点不是空结点 */
            Visit(tree); /* 访问结点 */
            Push(stack, tree); /* 结点压入栈 */
            tree = tree->left; /* 沿左分支下移 */ }
        if (IsEmpty(stack)==false) { /* 如果栈不为空 */
            tree = Top(stack);
            Pop(stack); /* 弹出栈顶结点 */
            tree = tree->right; /* 移到右子树 */ }
    }
    DestroyStack(stack);
}
```

5.4.3 二叉树遍历的非递归算法

- 算法5-8：非递归中序遍历 InOrder(*tree*)

```
InitStack(stack)    //初始化栈stack，用于存放结点
while tree ≠ NIL 或 IsEmpty(stack) = false do
| while tree ≠ NIL do        //当前结点不是空结点
| | Push(stack, tree)   //结点压入栈
| | tree ← tree.left     //沿左分支下移
| end
| if IsEmpty(stack) = false then   //如果栈不为空
| | tree ← Top(stack)
| | Visit(tree)          //访问栈顶结点
| | Pop(stack)           //弹出栈顶结点
| | tree ← tree.right  //移到栈顶结点的右子树
| end
end
DestroyStack(stack)
```

沿左分支下移，并将经过的结点压入栈

tree=NIL表示左子树为空，或者左子树遍历结束，则从栈中弹出结点，开始遍历结点的右子树

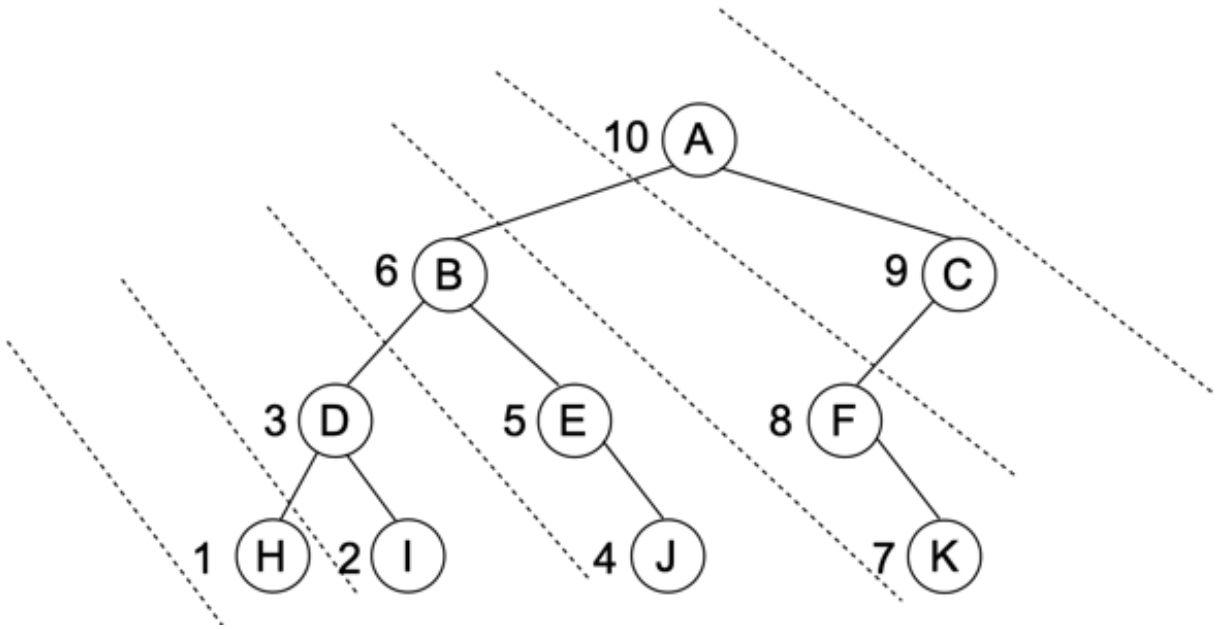
5.4.3 二叉树遍历的非递归算法

- 算法5-8：非递归中序遍历 InOrder(*tree*)

```
void InOrder(BinaryTree tree)
{
    Stack stack;
    stack = (Stack)malloc(sizeof(struct StackHeadNode));
    InitStack(stack); /* 初始化栈stack，用于存放结点 */
    while (tree != NULL || IsEmpty(stack)==false) {
        while (tree != NULL) { /* 当前结点不是空结点 */
            Push(stack, tree); /* 结点压入栈 */
            tree = tree->left; /* 沿左分支下移 */ }
        if (IsEmpty(stack)==false) { /* 如果栈不为空 */
            tree = Top(stack);
            Visit(tree); /* 访问结点 */
            Pop(stack); /* 弹出栈顶结点 */
            tree = tree->right; /* 移到右子树 */ }
    }
    DestroyStack(stack);
}
```

5.4.3 二叉树遍历的非递归算法

- **后序遍历算法的非递归化**



二叉树所有结点的后序遍历次序，用各结点左边的数字表示

后序遍历次序的结点序列可分为多个段，用虚线分开，每段中的结点具有以下性质：

(1) 段中各结点的访问次序是连续的，并且最先访问（次序最小）的结点没有右子结点

(2) 段中若有多个结点，则次序相邻的任意两个结点，次序小的结点是次序大的结点的右子结点

(3) 段中次序最大的结点如果不是根，则是其父结点的左子结点

5.4.3 二叉树遍历的非递归算法

- 算法5-9：非递归后序遍历 PostOrder(*tree*)

```
InitStack(stack)
while tree ≠ NIL 或 IsEmpty(stack) = false do
| while tree ≠ NIL do  //当前结点不是空结点
| | Push(stack, tree)  //结点压入栈
| | tree ← tree.left  //沿左分支下移
| end
|
| top ← Top(stack)     //stack非空，top指向栈顶结点
| pre_top ← NIL        //初始化 pre_top
|                      //如果栈顶结点的右子树为空，开始从栈弹出结点
```

沿左分支下移，并将经过的结点压入栈

开始时，*pre_top* = NIL

5.4.3 二叉树遍历的非递归算法

```
| while IsEmpty(stack) = false 且 top.right = pre_top do
| | Visit(top)          //访问当前栈顶结点
| | pre_top ← top       //栈顶结点传给pre_top
| | Pop(stack)          //弹出栈顶结点
| | if IsEmpty(stack) = false then
| | | top ← Top(stack)  //栈非空，top指向新的栈顶结点
| | else
| | | top ← NIL         //空栈，top赋值NIL，结束遍历
| | end
| end
| if top ≠ NIL then
| | tree ← top.right    //移到栈顶结点的右子树并开始遍历
| end
end
DestroyStack(stack)
```

从右子树为空的结点开始。依次弹出各段中的结点

若弹出的结点是新栈顶结点的右子结点，继续弹出

5.4.3 二叉树遍历的非递归算法

- 非递归后序遍历

```
void PostOrder(BinaryTree tree)
{
    Stack stack;
    BinaryTree top, pre_top;
    stack = (Stack)malloc(sizeof(struct StackHeadNode));
    InitStack(stack); /* 初始化栈stack，用于存放结点 */
    while (tree != NULL || IsEmpty(stack)==false) {
        while (tree != NULL) { /* 当前结点不是空结点 */
            Push(stack, tree); /* 结点压入栈 */
            tree = tree->left; /* 沿左分支下移 */
        }
        top = Top(stack); /* stack非空，top指向栈顶结点 */
        pre_top = NULL; /* 初始化pre_top */
```

5.4.3 二叉树遍历的非递归算法

- 非递归后序遍历

```
        while (IsEmpty(stack)==false && top->right==pre_top) {
          Visit(top); /* 访问当前栈顶结点 */
          pre_top = top; /* 栈顶结点传给pre_top */
          Pop(stack); /* 弹出栈顶结点 */
          if (IsEmpty(stack)==false) {
                top = Top(stack); /* 栈非空，top指向新的栈顶结点 */
            }
            else {
                top = NULL; /* 空栈，top赋值为空 */
            }
        }
        if (top != NULL) {
          tree = top->right; /* 移到栈顶结点的右子树并开始遍历 */
        }
    }
    DestroyStack(stack);
}
```

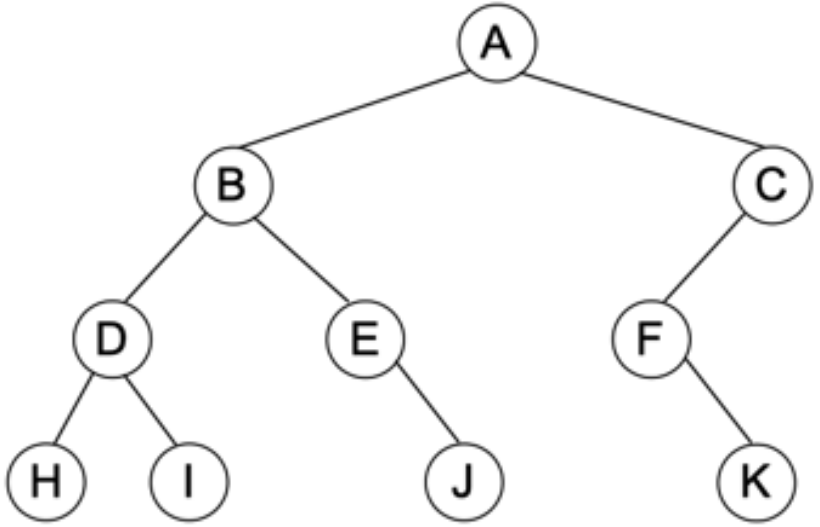
5.4.3 二叉树遍历的非递归算法

- 非递归算法复杂度分析

前序、中序和后序的非递归算法均使用一个存放结点的栈实现，并且在遍历过程中每个结点都有且仅有1次入栈和1次出栈的机会，因此算法的时间复杂度和空间复杂度都是O(n)，其中n表示二叉树的结点数

5.4.3 二叉树遍历的非递归算法

- **广度优先遍历（层序遍历）**：从根结点开始，从上至下按层访问每个结点，并且每层的结点按照从左向右的顺序进行处理



层序遍历：<A, B, C, D, E, F, H, I, J, K >

层序遍历方案通常设计成非递归的算法，可以用队列结构来实现

5.4.3 二叉树遍历的非递归算法

- 算法5-10：层序遍历二叉树 LevelOrder(*tree*)

```
InitQueue(queue)              //初始化队列queue，用于存放结点
EnQueue(queue, tree)          //树根结点入队
while IsEmpty(queue) = false do
| node_ptr ← GetFront(queue)  //取出队首结点
| DeQueue(queue)              //队首出队
| if node_ptr ≠ NIL then      //结点非空
| | Visit(node_ptr)           //访问结点
| | EnQueue(queue, node_ptr.left)   //左子结点入队
| | EnQueue(queue, node_ptr.right)  //右子结点入队
| end
end
DestroyQueue(queue)
```

- 复杂度O(n)

5.4.3 二叉树遍历的非递归算法

- 层序遍历

```
void LevelOrder(BinaryTree tree)
{
    Queue queue;
    BinaryTree node_ptr;
    queue = (Queue)malloc(sizeof(struct QueueHeadNode));
    InitQueue(queue); /* 初始化队列queue，用于存放结点 */
    EnQueue(queue, tree); /* 树根结点入队 */
    while (IsEmpty(queue)==false) {
        node_ptr = GetFront(queue); /* 取出队首结点 */
        DeQueue(queue);
        if (node_ptr != NULL) { /* 结点非空 */
            Visit(node_ptr); /* 访问结点 */
            EnQueue(queue, node_ptr->left); /* 左子结点入队 */
            EnQueue(queue, node_ptr->right);/* 右子结点入队 */
        }
    }
    DestroyQueue(queue);
}
```

5.4.4 二叉树的序列化与反序列化

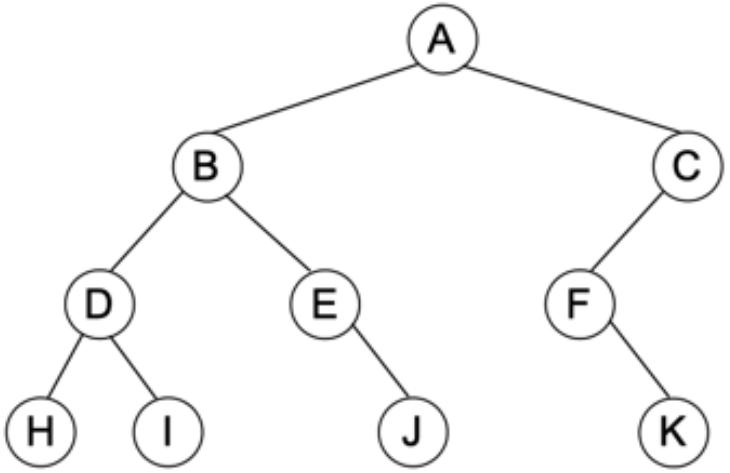
- **二叉树的序列化**：按某种遍历方案访问所有结点并依次输出结点数据，由此形成结点的线性序列

序列化的作用：将树的非线性结构转换成线性结构，便于使用线性表或字符串等存储

- **二叉树的反序列化**：根据线性序列重构原始的二叉树

- **问题**：
  - 完全二叉树的顺序存储是一种序列化方案，并且可以根据结点间的相对位置确定它们之间的逻辑关系，重构出二叉树
  - 但该方法对于一般的二叉树可能造成空间浪费，在最坏情况下，空间复杂度达到$O(2^n)$
  - 常用的前序遍历或层序遍历算法，产生的结点序列只包含了树结构的部分信息，通常无法重构二叉树

5.4.4 二叉树的序列化与反序列化



前序遍历：<A, B, D, H, I, E, J, C, F, K>

从序列中，最多只能确定A是根结点，其它信息，如左子树包含哪些结点等无法确定

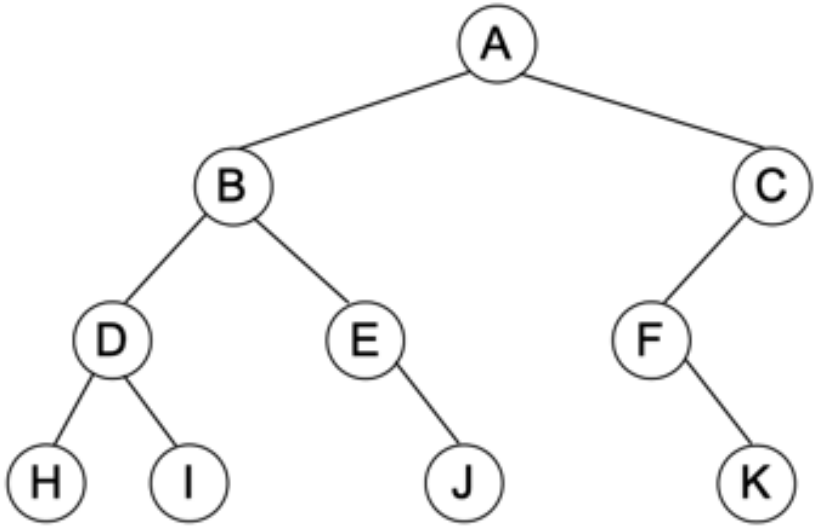
问题：如何使前序遍历的结果能够重构二叉树?

序列化方法

- 用特殊符号#表示空结点
- 当在前序遍历过程中遇到空结点或空子树时，不是直接返回，而是输出符号#，从而将空结点也标记在序列中

5.4.4 二叉树的序列化与反序列化

- **二叉树前序序列化：**前序遍历二叉树，如果结点非空，输出结点数据，否则输出#



通过在结点序列中插入空记号，记录二叉树的非线性结构

<A, B, D, H, #, #, I, #, #, E, #, J, #, #, C, F, #, K, #, #, #>

5.4.4 二叉树的序列化与反序列化

- 算法5-11：二叉树前序序列化 PreOrderSerialize(*tree*)

输入：二叉树*tree*
输出：二叉树的前序序列

```
if tree = NIL then  //空树
| print #    //输出特殊符号，代表空结点
else
| print tree.data   //输出结点数据
| PreOrderSerialize (tree.left)     //对左子树前序序列化
| PreOrderSerialize (tree.right)    //对右子树前序序列化
end
```

基于前序遍历算法

5.4.4 二叉树的序列化与反序列化

- 算法5-11：二叉树前序序列化 PreOrderSerialize(*tree*)

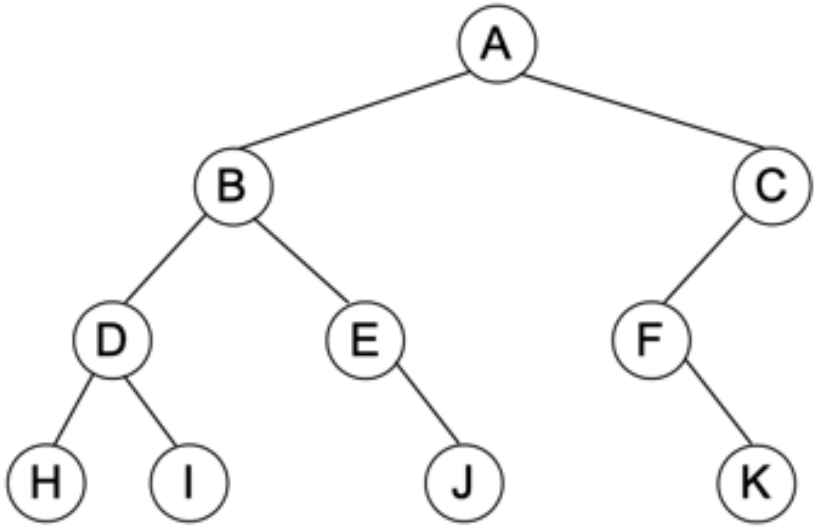
```
void PreOrderSerialize(BinaryTree tree)
{
    if (tree == NULL) { /* 空树 */
        printf("#\n"); /* 输出特殊符号，代表空结点 */
    }
    else {
        printf("%d\n", tree->data); /* 输出根结点数据 */
        PreOrderSerialize(tree->left); /* 对左子树前序序列化 */
        PreOrderSerialize(tree->right);/* 对右子树前序序列化 */
    }
}
```

5.4.4 二叉树的序列化与反序列化

- 前序序列的反序列化

从前序序列先端依次读取数据，执行下面的操作：

- 如果读取的数据是#，则返回NIL，表示空结点或空树
- 否则新建二叉树结点，把数据代入结点并递归地重构结点的左子树和右子树，然后返回结点。



<A, B, D, H, #, #, I, #, #, E, #, J, #, #, C, F, #, K, #, #, #>

前序序列

5.4.4 二叉树的序列化与反序列化

- **前序序列的反序列化**

输入：存放二叉树前序序列的线性表*preorder*，表中元素个数*n* (n>0)
输出：二叉树
全局变量：k，初始值为-1

```
k ← k + 1
tree ← NIL   //初始化一个空树
if k < n then  //k是线性表的有效序号
| data ← Get(preorder, k)       //读出线性表第k个元素
| if data ≠ # then      //非空记号
| | tree ← new BinaryTreeNode( )  //新建二叉树结点
| | tree.data ← data              //代入数据
| | tree.left ← PreOrderDeSerialize(preorder, n)  //重构左子树
| | tree.right ← PreOrderDeSerialize(preorder, n)  //重构右子树
| end
end
return tree  //返回新建的二叉树或空树
```

5.4.4 二叉树的序列化与反序列化

- 前序序列的反序列化

```
BinaryTree PreOrderDeSerialize(TElemSet preorder[], int n)
{
    BinaryTree tree;
    TElemSet data;
    k++;
    tree = NULL; /* 初始化一个空树 */
    if (k < n) { /* k是线性表的有效序号 */
        data = preorder[k]; /* 读出线性表第k个元素 */
        if (data != NIL) {
            tree = (BinaryTree)malloc(sizeof(struct BinaryTreeNode));/* 新建二叉树结点 */
            tree->data = data; /* 代入数据 */
            tree->left = PreOrderDeSerialize(preorder, n); /* 重构左子树 */
            tree->right = PreOrderDeSerialize(preorder, n); /* 重构右子树 */
        }
    }
    return tree; /* 返回新建的二叉树或空树 */
}
```

5.4.4 二叉树的序列化与反序列化

- 层序序列化

```
InitQueue(queue)              //初始化队列queue，用于存放结点
EnQueue(queue, tree)     //树根结点入队
while IsEmpty(queue) = false do
| node_ptr ← GetFront(queue)   //取出队首结点
| DeQueue(queue)              //队首出队
| if node_ptr = NIL then      //结点非空
| | print #                   // 打印空结点
| else
| | print(node_ptr.data)     //打印当前节点
| | EnQueue(queue, node_ptr.left)    //左子结点入队
| | EnQueue(queue, node_ptr.right)   //右子结点入队
| end
end
DestroyQueue(queue)
```

5.4.4 二叉树的序列化与反序列化

```
void LevelOrderSerialize(BinaryTree tree){
    Queue queue;
    BinaryTree node_ptr;
    queue = (Queue)malloc(sizeof(struct QueueHeadNode));
    InitQueue(queue);
    EnQueue(queue, tree);
    while (IsEmpty(queue)==false) {
        node_ptr = GetFront(queue);
        DeQueue(queue);
        if (node_ptr == NULL) { /* 空结点 */
            printf("#\n"); /* 输出特殊符号，代表空结点 */
        }
        else {
            printf("%d\n", node_ptr->data); /* 输出结点数据 */
            EnQueue(queue, node_ptr->left); /* 对左子树层序序列化*/
            EnQueue(queue, node_ptr->right);/* 对右子树层序序列化*/
        }
    }
    DestroyQueue(queue);
}
```

5.4.4 二叉树的序列化与反序列化

- 根据层序序列重构二叉树

```
BinaryTree LevelOrderDeSerialize(TElemSet levelorder[], int n)
{
    Queue queue;
    BinaryTree tree, node_ptr;
    TElemSet data;
    int k;
    if (n==1) { /* 序列中只有一个# */
        tree = NULL; /* 空树 */
    }
    else {
        queue = (Queue)malloc(sizeof(struct QueueHeadNode));
        InitQueue(queue);
        tree = (BinaryTree)malloc(sizeof(struct BinaryTreeNode)); /* 创建树根结点 */
        tree->data = levelorder[0]; /* 代入线性表第一个元素 */
        EnQueue(queue, tree);
        k = 1;
```

5.4.4 二叉树的序列化与反序列化

- 根据层序序列重构二叉树

```
while (k < n) { /* 从线性表第二个位置开始读取 */
    node_ptr = GetFront(queue); /* 队首结点出队 */
    DeQueue(queue);
    data = levelorder[k]; /* 线性表第k+1个元素 */
    if (data != NILL) {
        node_ptr->left = (BinaryTree)malloc(sizeof(struct BinaryTreeNode)); /* 生成左子结点 */
        node_ptr->left->data = data; /* 线性表第k+1个元素代入左子节点 */
        EnQueue(queue, node_ptr->left); /* 左子结点入队 */
    }
    else {
        node_ptr->left = NULL; /* 左子树设置为空树 */
    }
    k++;
    data = levelorder[k]; /* 线性表第k+2个元素 */
```

5.4.4 二叉树的序列化与反序列化

- 根据层序序列重构二叉树

```
            if (data != NILL) {
                node_ptr->right = (BinaryTree)malloc(sizeof(struct BinaryTreeNode)); /* 生成右子结点 */
                node_ptr->right->data = data; /* 线性表第k+2个元素代入右子节点 */
                EnQueue(queue, node_ptr->right); /* 右子结点入队 */
            }
            else {
                node_ptr->right = NULL; /* 右子树设置为空树 */
            }
            k++; /* k为下一个待处理的位置 */
        }
    }
    DestroyQueue(queue);
    return tree; /* 返回新建的二叉树 */
}
```

小结

- 二叉树是一种常用的树结构，它具有一些特别而且有用的性质
- 满二叉树、完全二叉树、完美二叉树是最常用的三种特殊形态二叉树
- 二叉树有顺序和链式两种存储表示。
  - 顺序存储将所有节点按照层次顺序存储到连续的存储单元，这种方法更适合完全二叉树;
  - 链式存储又称二叉链表，每个结点包含2个指针，分别指向左孩子和右孩子，这种方法适用于一般二叉树。
- 为解决二叉链表对父结点访问的限制，增加一个parent用于保存父节点的地址，这种方法称为三叉链。

小结

- 二叉树的遍历算法是其他运算的基础
- 通过遍历可以得到二叉树中结点访问的线性序列，实现非线性结构的线性化
- 深度优先遍历：根据结点访问次序的不同，常见遍历方法包括先序遍历、中序遍历、后序遍历
- 广度优先遍历：按从上到下的顺序，可以实现树的层次遍历
- 树遍历的时间复杂度都是 O(n)

树与二叉树高频必刷题（LeetCode）

分类	题号	题目	考察点（高频原因）	难度
二叉树基础概念	104	Maximum Depth of Binary Tree	递归/迭代实现深度计算，二叉树基础操作入门	Easy
二叉树基础概念	226	Invert Binary Tree	镜像翻转二叉树，逻辑直观且高频考察	Easy
完全二叉树特性	222	Count Complete Tree Nodes	利用完全二叉树索引特性优化计数，避免暴力遍历	Easy
二叉树遍历（核心）	94	Binary Tree Inorder Traversal	中序遍历（递归＋迭代），BST相关题基础	Easy
二叉树遍历（核心）	102	Binary Tree Level Order Traversal	层序遍历（BFS），树的广度优先遍历模板题	Medium
二叉树遍历（综合）	236	Lowest Common Ancestor of a Binary Tree	后序遍历应用，最近公共祖先（面试高频考点）	Medium
二叉树重构	105	Construct Binary Tree from Preorder and Inorder Traversal	前序＋中序重构树，重构类题的经典模板	Medium
二叉树与表达式处理	224	Basic Calculator	中缀表达式求值（关联表达式树），综合逻辑题	Hard