6 第三讲 树（上）

笔记本:	浙江大学《数据结构》		
创建时间:	2025/3/26 21:45	**更新时间:**	2025/4/8 20:51
作者:	panhengye@163.com		
URL:	vscode-file://vscode-app/c:/Users/Lenovo/AppData/Local/Programs/cursor/reso...		

学习过程中请特别关注：

- 完全二叉树如何在数组中实现完美存储
- 如何利用堆栈和队列实现二叉树的非递归遍历
- 已知二叉树的先序和中序遍历结果如何还原二叉树

我关于前、中、后序遍历方式的思考

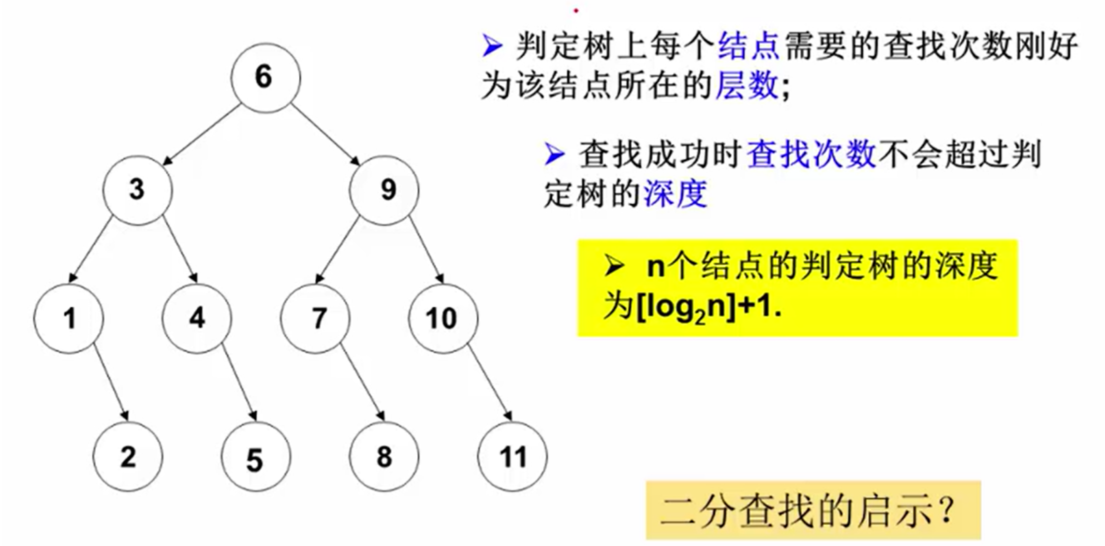
我的学习目标

- 能够熟练地用代码（无AI辅助）实现
    - 递归（[什么是递归（recursion）]）方法的前/中/后序遍历
    - 层序遍历
- 理解并掌握辅助数据结构的使用
    - 队列：用于层序遍历
    - 栈：理解递归过程
- 画图练习
    -
      ```text
      例如：画出以下树的前/中/后序遍历过程
            A
          /   \
         B     C
        / \   /
       D   E F
      ```

引入：客观世界中许多事物存在层次关系

- 分层次组织在管理上具有更高的效率
- 数据三大基本操作：查找、插入、删除
    - 静态查找：e.g. 查字典
    - 动态查找：e.g. 餐厅排队（员工根据空出来的座位叫号）

> 由于Python的动态特性和内置的边界检查，哨兵模式的性能优势可能不如在C/C++中明显。某些情况下，使用Python的内置函数（如index()）或列表推导式可能是更好的选择



平均查找长度（Average Search Length） ASL

- 衡量查找算法效率的一个重要指标
- 在不同的查找情况下，我们通常会区分：成功和失败两种情况

树的定义

树（Tree）：n个（n ≥ 0 ）个结点构成的有限集合

- 树中有一个称为“根”（root）的特殊结点，用r表示
- 其余结点可以分为m(m > 0)个互不相交的有限集，其中每个集合本身又是一棵树，称为原来树的“子树”(SubTree)

通过辨识树和非树得到的启发

- 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
- 一棵N个结点的树有N-1条边

```
树是保证结点联通的最小的连接方式
```

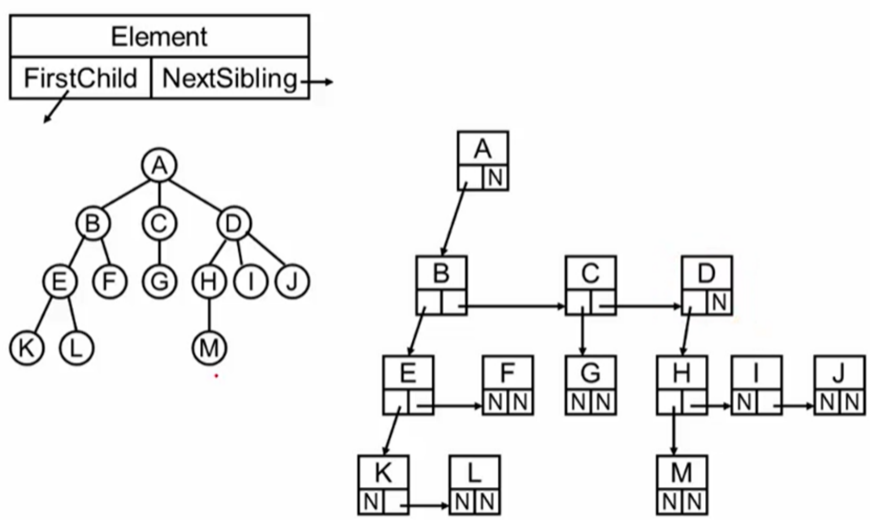
树的一些术语

- 结点的度（degree）：结点的子树个数
  - 解释度的由来
    - 数学中最早用于表示多项式中最高次项的指数；后来在图论中表示与某个顶点相连的边的数量；后来扩展到树中表示某个结点的分支数量
    - 可以理解为结点向外延伸的能力，“度”在中文语境中本来就有数量的含义，比如速度
    - 这种命名反映了数学中常见的概念迁移现象，即用已有的术语来描述新的但性质相似的概念
- 树的度：树的所有结点中最大的度数
  - 就像种树要预留空间，这里也需要考虑树冠展开的最大情况
- 叶节点（Leaf）:度为0的结点

    - 之所以度为0的结点被称为（leaf），它就相当于一棵树的末梢，再也没有长出枝丫的可能了
- 父节点（parent）
- 子节点（child）
- 兄弟结点（sibling）：具有同一父节点的各节点彼此是兄弟结点
- 路径和路径长度
    - 路径所包含边的个数为路径的长度
    - 路径为一个结点序列，其中n_i是n_{i+1}的父节点（从上往下）
- 祖先结点（Ancestor）:沿着树根到某一个结点路径上的所有结点都是这个结点的祖先
- 子孙结点（Descendant）：与祖先结点相对
- 结点的层次（level）：
    - 根结点在1层
    - 其他任意结点的层数是父结点层数+1
- 树的深度（Depth）：树中所有结点中的最大层次
- 树的高度（Height） -- 补充知识点
    - 节点的高度
        - 从该节点到其最远叶子节点的最长路径上的边的数量
        - 叶子节点的高度为0
    - 树的高度：根节点的高度就是整棵树的高度，也就是从根节点到最远叶子节点的最长路径上的边的数量
    - 有时也会用"深度"(Depth)这个概念,但深度是从上往下数(根为0),而高度是从下往上数(叶子为0)
    - 空树的高度定义为-1

树的表示

- 儿子-兄弟表示法
    - FirstChild
    - NextSibling



    -

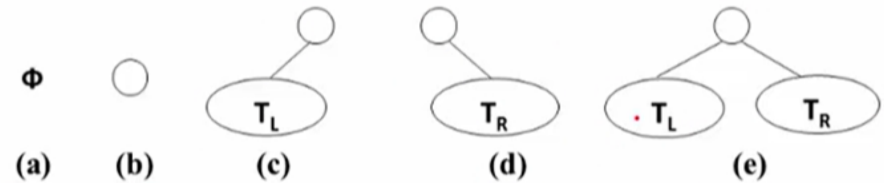
二叉树的定义

一个有穷的结点集合：

- 这个集合可以为空
- 若不为空，则它是由根结点和称为其左子树T_L和右子树T_R的两个不相交的二叉树组成

二叉树是度为2树的一种特殊情况，具有左右之分

二叉树具体五种基本形态



特殊的二叉树

- 斜二叉树（Skewed binary Tree）
- 完美/满二叉树（Perfect Binary Tree）
- 完全二叉树（Complete Binary Tree）

> 完全二叉树相对于完美二叉树，在最后一层允许有空缺：
>
> ①除最后一层外，其他层必须填满节点
> ②最后一层允许有空缺，但节点必须从左到右连续排列，不能跳跃。即如果最后一层某个位置是空的，那么它右边所有的位置都必须是空的

二叉树的几个重要性质

- 一个二叉树第i层的最大结点数为：$2^{i-1}, i \geq 1$
- 深度为K的二叉树有最大结点总数：2^{k-1}， $k \geq 1$
- 对任何非空二叉树，叶结点的总数n_0与度为2的非叶结点个数为n_2，则$n_0 = n_2 + 1$

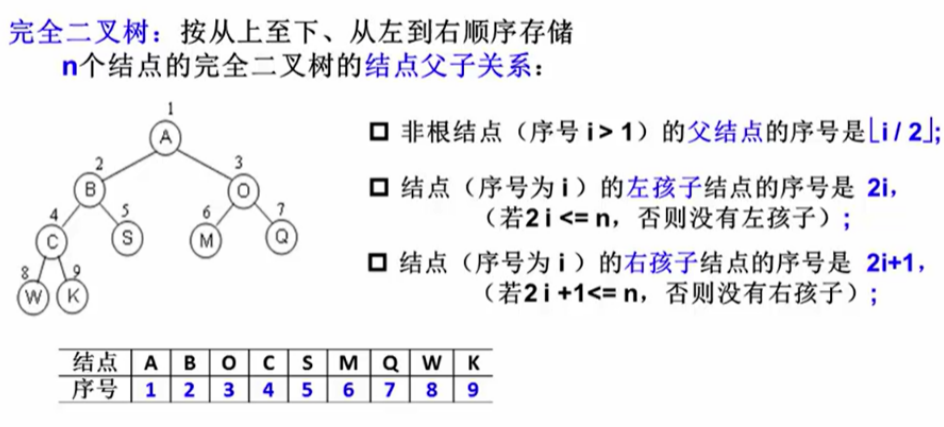
二叉树的抽象数据类型定义

类型名称：二叉树
数据对象集：一个有穷的结点集合。
若不为空，则由根结点和其左、右二叉子树组成。

操作集：BT ∈ BinTree, Item ∈ ElementType，重要操作有：
1、Boolean IsEmpty(BinTree BT)：判别BT是否为空；
2、void Traversal(BinTree BT)：遍历，按某顺序访问每个结点；
3、BinTree CreatBinTree( )：创建一个二叉树。

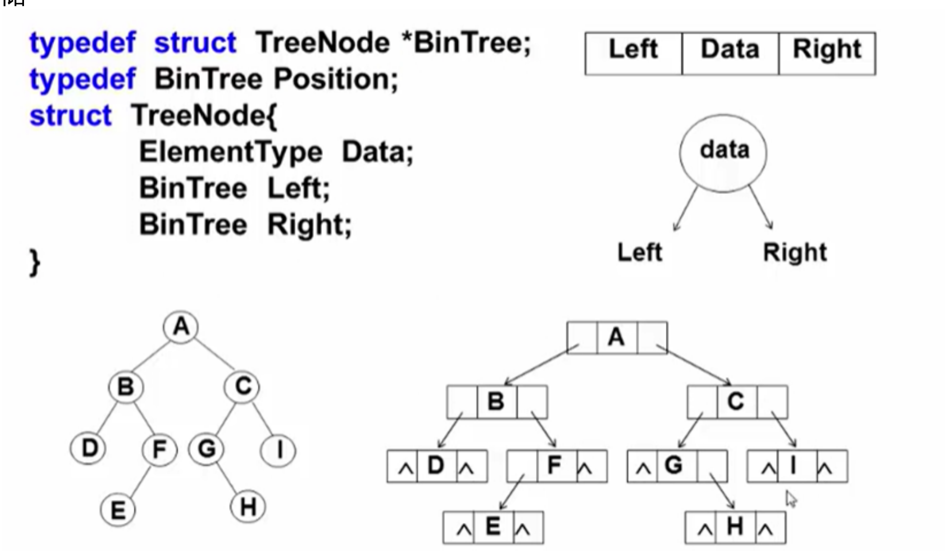
二叉树的存储结构

1. 顺序存储结构

1. 

2. 缺点：一般二叉树的情况下，会造成空间的浪费

2. 链表存储

1. 

二叉树的遍历

思考：为什么说“在研究二叉树这种数据结构的时候，它的常见用法中遍历是最重要的”？
[为什么遍历对于二叉树而言非常重要]

1. 先序遍历
   1. 访问根结点
   2. 先序遍历其左子树
   3. 先序遍历其右子树

```
void PreOrderTraversal( BinTree BT )
{
    if( BT ) {
        printf("%d", BT->Data);
        PreOrderTraversal( BT->Left );
        PreOrderTraversal( BT->Right );
    }
}
```

4.

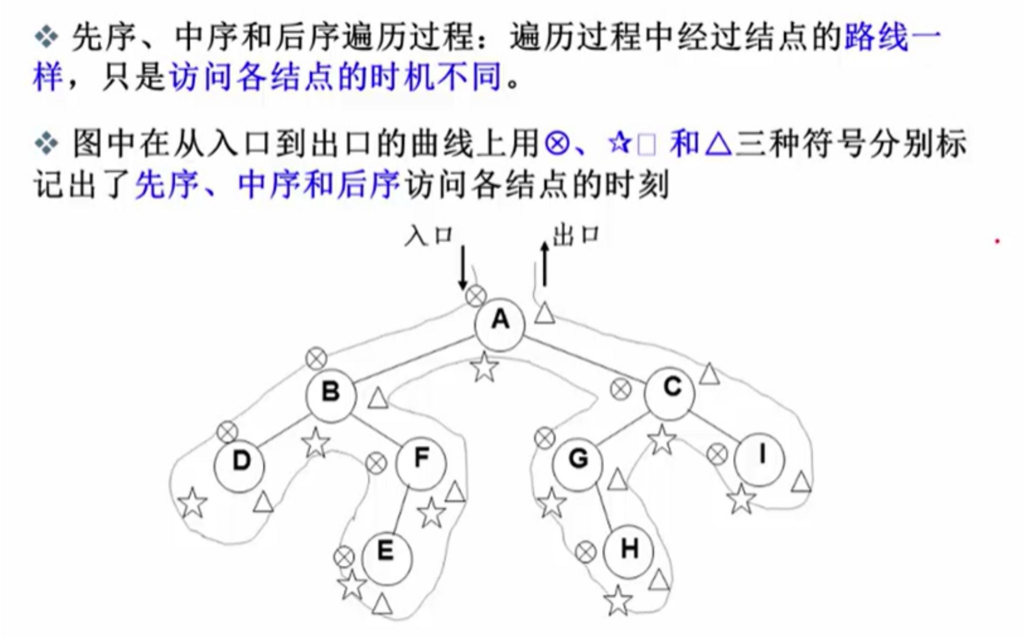
2. 中序遍历
    1. 中序遍历其左子树
    2. 访问根结点
    3. 中序遍历其右子树

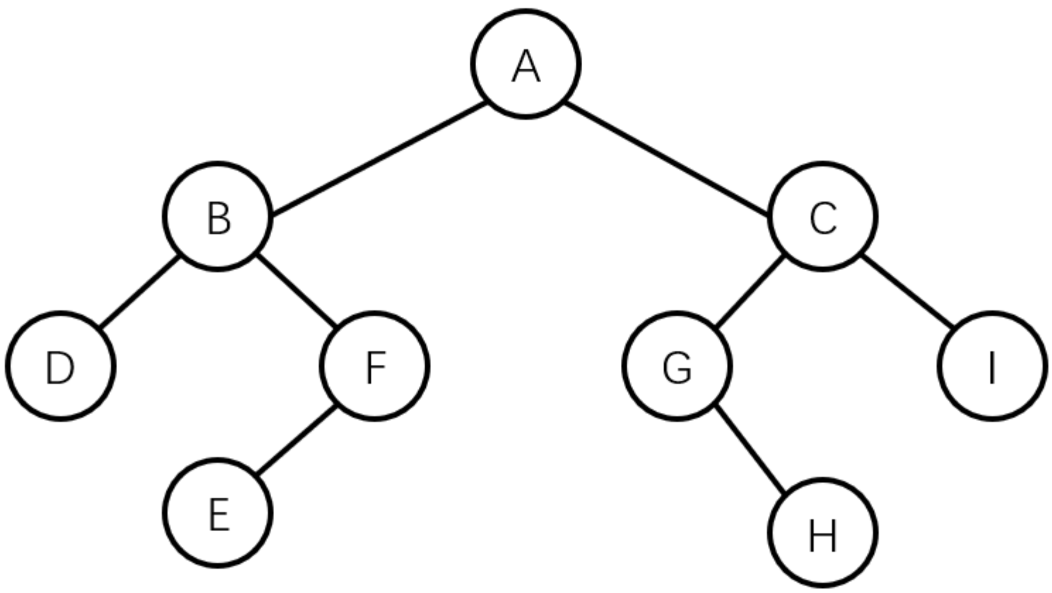
```
void InOrderTraversal( BinTree BT )
{
    if( BT ) {
        InOrderTraversal( BT->Left );
        printf("%d", BT->Data);
        InOrderTraversal( BT->Right );
    }
}
```


4.

3. 后序遍历
    1. 后序遍历其左子树
    2. 后序遍历其右子树
    3. 访问根结点
    4.

```
void PostOrderTraversal( BinTree BT )
{
    if( BT ) {
        PostOrderTraversal( BT->Left );
        PostOrderTraversal( BT->Right);
        printf("%d", BT->Data);
    }
}
```

4. 



思考：对于上图这样的树，其四种遍历方式出来的顺序是什么？

- 先序：A B D F E C G H I
- 中序：D B E F A G H C I
- 后序：D E F B H G I C A
- 层序：A B C D F G I E H

中序非递归遍历

算法（Python版）

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
```

```
        self.right = None


def inorder_traversal(root):
    # 初始化一个空栈
    stack = []
    current = root

    # 当current不为空或栈不为空时继续循环
    while current or stack:
        # 一直向左遍历，将所有左侧节点压入栈中
        while current:
            stack.append(current)
            current = current.left

        # 从栈中弹出节点并访问
        current = stack.pop()
        print(f"{current.data:5d}", end="")  # 保持与C代码相同的输出格式

        # 转向右子树
        current = current.right
```

关键点解析

```
    # 当current不为空或栈不为空时继续循环
    while current or stack:
```

- `current` 不为空：
  - 表示当前还有节点需要访问
  - 这种情况下，我们需要继续向左遍历，把所有左侧节点都压入栈中
- `stack` 不为空：
  - 表示栈中还有未处理完的节点
  - 即使 `current` 为空，只要栈不为空，就说明还有节点需要访问

缺少任何一个条件都可能导致遍历不完整或过早终止

层序遍历

> 二叉树遍历的实质是将二维结构变成一个一维的线性序列的过程

```
from collections import deque


class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def level_order_traversal(root):
    # 如果是空树则直接返回
    if not root:
        return

    # 创建并初始化队列
    queue = deque([root])

    # 当队列不为空时继续循环
    while queue:
        # 从队列中取出一个节点
        node = queue.popleft()
```

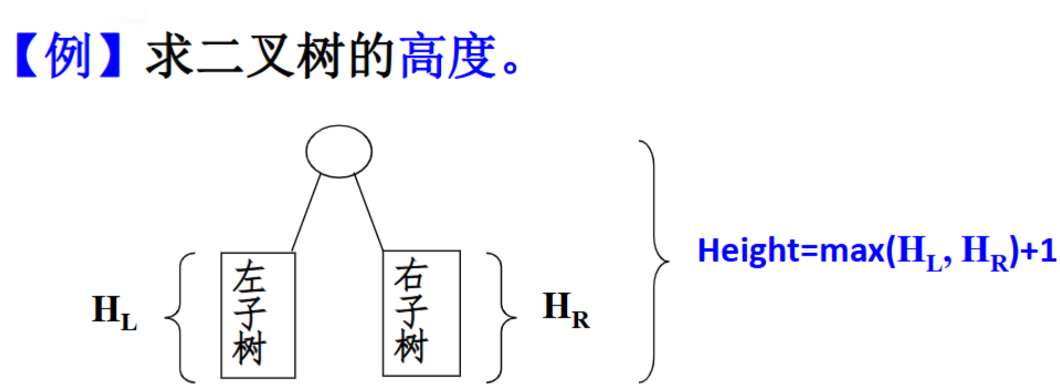
```
        # 访问（打印）该节点
        print(f"{node.data}")

        # 如果左子节点存在，将其加入队列
        if node.left:
            queue.append(node.left)

        # 如果右子节点存在，将其加入队列
        if node.right:
            queue.append(node.right)
```

注：该代码使用了Python中内置的[双端队列]类

循环部分做三件事：

- 从队列里抛出一个元素
- 打印刚刚抛出的元素
- 将该元素的左右儿子放进队列

遍历应用例子



利用后序遍历的程序框架实现