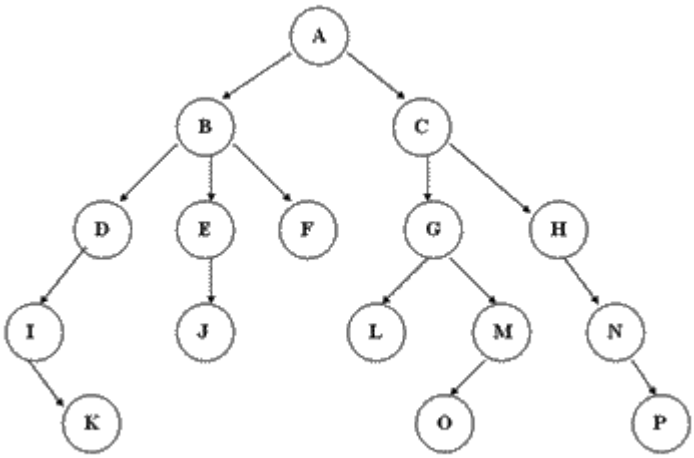
树

树（tree）是一种抽象数据类型（ADT）或是实现这种抽象数据类型的数据结构，用来模拟具有树状结构性质的数据集合。它是由n（n>0）个有限节点组成一个具有层次关系的集合。

- 每个节点都只有有限个子节点或无子节点；
- 没有父节点的节点称为根节点；
- 每一个非根节点有且只有一个父节点；
- 除了根节点外，每个子节点可以分为多个不相交的子树；
- 树里面没有环路(cycle)



相关概念

1. **节点的度**：一个节点含有的子树的个数称为该节点的度；
2. **树的度**：一棵树中，最大的节点度称为树的度；
3. **叶节点**：度为零的节点；
4. 节点的**层次**：从根开始定义起，根为第1层，根的子节点为第2层，以此类推；
5. **深度**：对于任意节点n,n的深度为从根到n的唯一路径长，根的深度为0；
6. **高度**：对于任意节点n,n的高度为从n到一片树叶的最长路径长，所有树叶的高度为0；

树的分类

- 无序树：树中任意节点的子节点之间没有顺序关系，这种树称为无序树，也称为自由树；
- 有序树：树中任意节点的子节点之间有顺序关系，这种树称为有序树；
    - 二叉树：每个节点最多含有两个子树的树称为二叉树；
        - 完全二叉树：对于一颗二叉树，假设其深度为d（d>1）。除了第d层外，其它各层的节点数目均已达最大值，且第d层所有节点从左向右连续地紧密排列，这样的二叉树被称为完全二叉树；
            - 满二叉树：所有叶节点都在最底层的完全二叉树；

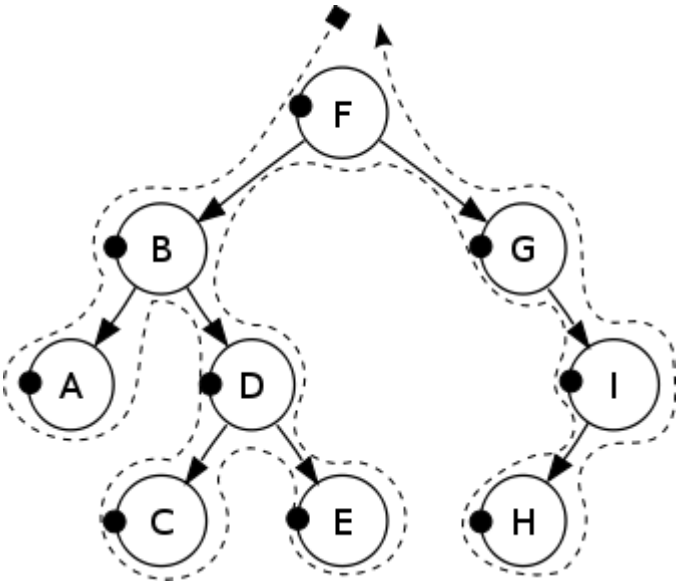
        - 平衡二叉树（AVL树）：当且仅当任何节点的两棵子树的高度差不大于1的二叉树；
        - 排序二叉树(二叉查找树，Binary Search Tree)：也称二叉搜索树、有序二叉树；
    - 霍夫曼树：带权路径最短的二叉树称为哈夫曼树或最优二叉树；
    - B树：一种对读写操作进行优化的自平衡的二叉查找树，能够保持数据有序，拥有多于两个子树。

树的遍历

深度优先

前序遍历 Pre-Order Traversal

指先访问根，然后访问子树的遍历方式



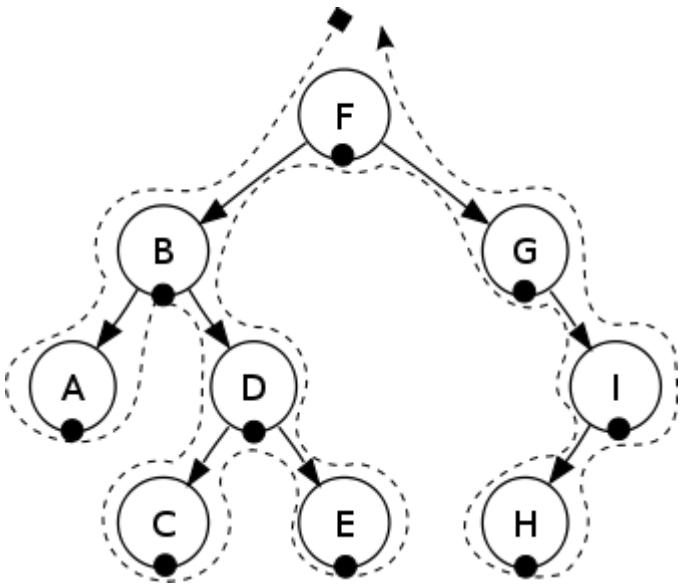
深度优先遍历 - 前序遍历：F, B, A, D, C, E, G, I, H.

```
// 递归版
void pre_order_traversal(Tree_node *root)
{
    // Do Something with root
    if (root->lchild != NULL)
        pre_order_traversal(root->lchild);
    if (root->rchild != NULL)
        pre_order_traversal(root->rchild);
}

// 迭代版
void pre_order_traversal(Tree_node *root)
{
    stack<Tree_node *> st;
    while (root || !st.empty()) {
```

```
        while (root) {
            // Do Something with root
            st.push(root);
            root = root->lchild;
        }
        if (!st.empty()) {
            root = st.top();
            st.pop();
            root = root->rchild;
        }
    }
}
```

中序遍历 In-Order Traversal

指先访问左（右）子树，然后访问根，最后访问右（左）子树的遍历方式



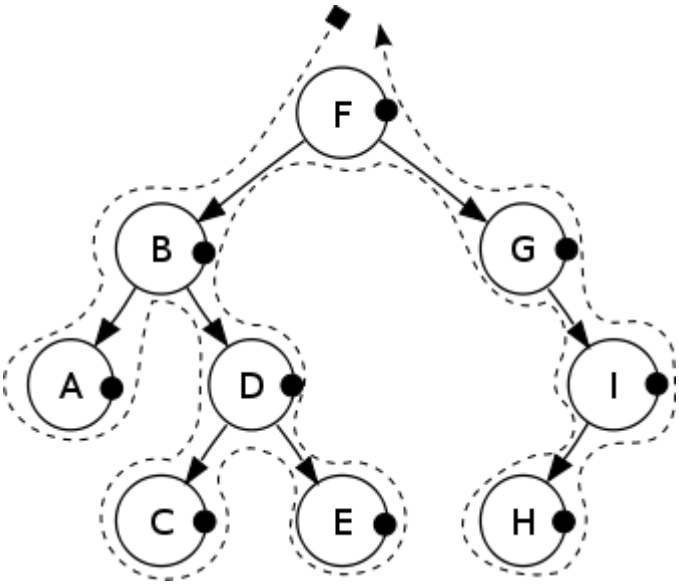
深度优先遍历 - 中序遍历：A, B, C, D, E, F, G, H, I.

```
// 递归版
void in_order_traversal(Tree_node *root)
{
    if (root->lchild != NULL)
        in_order_traversal(root->lchild);
    // Do Something with root
    if (root->rchild != NULL)
        in_order_traversal(root->rchild);
}

// 迭代版
void in_order_traversal(Tree_node *root)
{
    stack<Tree_node *> st;
    while (root || !st.empty()) {
        while (root) {
```

```
            st.push(root);
            root = root->lchild;
        }
        if (!st.empty()) {
            root = st.top();
            st.pop();
            // Do Something with root
            root = root->rchild;
        }
    }
}
```

后序遍历 Post-Order Traversal

指先访问子树，然后访问根的遍历方式



深度优先搜索 - 后序遍历：A, C, E, D, B, H, I, G, F.

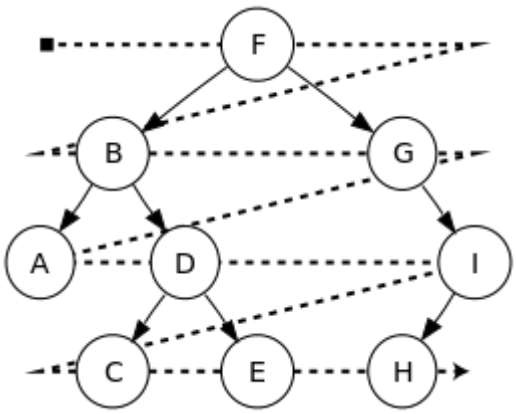
```
// 递归版
void post_order_traversal(Tree_node *root)
{
    if (root->lchild != NULL)
        post_order_traversal(root->lchild);
    if (root->rchild != NULL)
        post_order_traversal(root->rchild);
    // Do Something with root
}

void post_order_traversal(Tree_node *root)
{
    stack<Tree_node *> st;
    Tree_node *last_visit = nullptr;
    while (root || !st.empty()) {
        while (root) {
            st.push(root);
```

```
            root = root->lchild;
        }
        root = st.top();
        if (!root->rchild || last_visit == root->rchild) {
            // Do Something with root
            last_visit = root;
            st.pop();
            root = nullptr;
        } else {
            root = root->rchild;
        }
    }
}
```

广度优先

层次遍历

二叉树的广度优先遍历又称按层次遍历，是先访问离根节点最近的节点。算法借助队列实现。



广度优先遍历 - 层次遍历：F, B, G, A, D, I, C, E, H.

```
void layer_traversal(Tree_node *root)
{
    queue<Tree_node *> qu;
    qu.push(root);
    while (!qu.empty()) {
        root = qu.front();
        qu.pop();
        if (root) {
            // Do Something with root
            qu.push(root->lchild);
            qu.push(root->rchild);
        }
    }
}
```