

二叉树遍历

核心：

代码实现：

前序遍历（递归方式）：

前序遍历（非递归方式）：

中序遍历（递归方式）：

中序遍历（非递归方式）：

后序遍历（递归方式）：

后续递归（非递归方式）：

对于二叉树遍历常用方式：**前序遍历**，**中序遍历**，**后序遍历**。

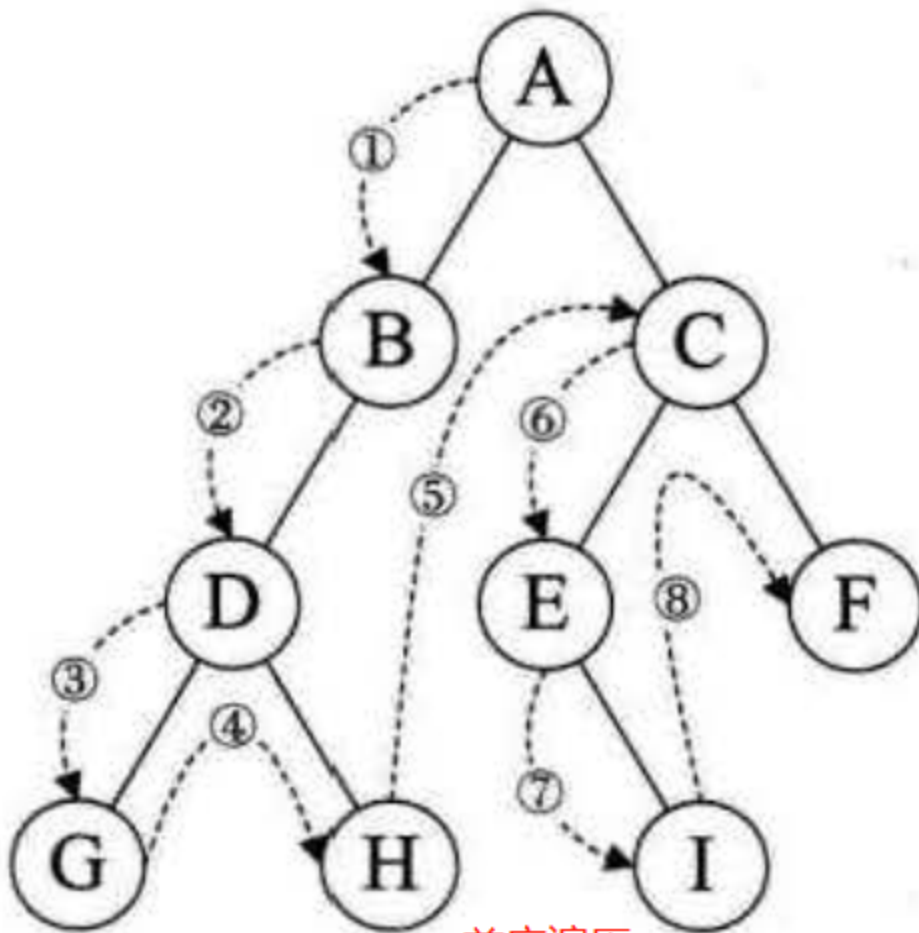
核心：

前序遍历：先遍历根结点，然后左子树，再右子树

中序遍历：先遍历左子树，然后根结点，再右子树

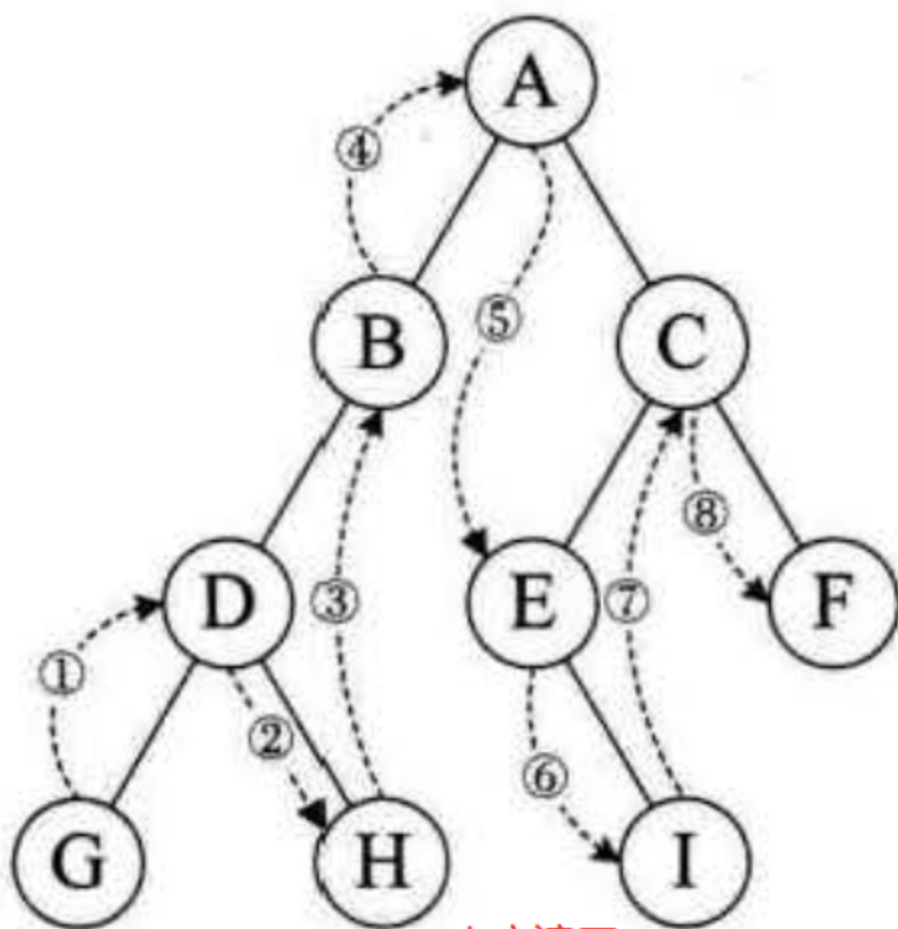
后续遍历：先遍历左子树，然后右子树，再根结点

前序过程如下：



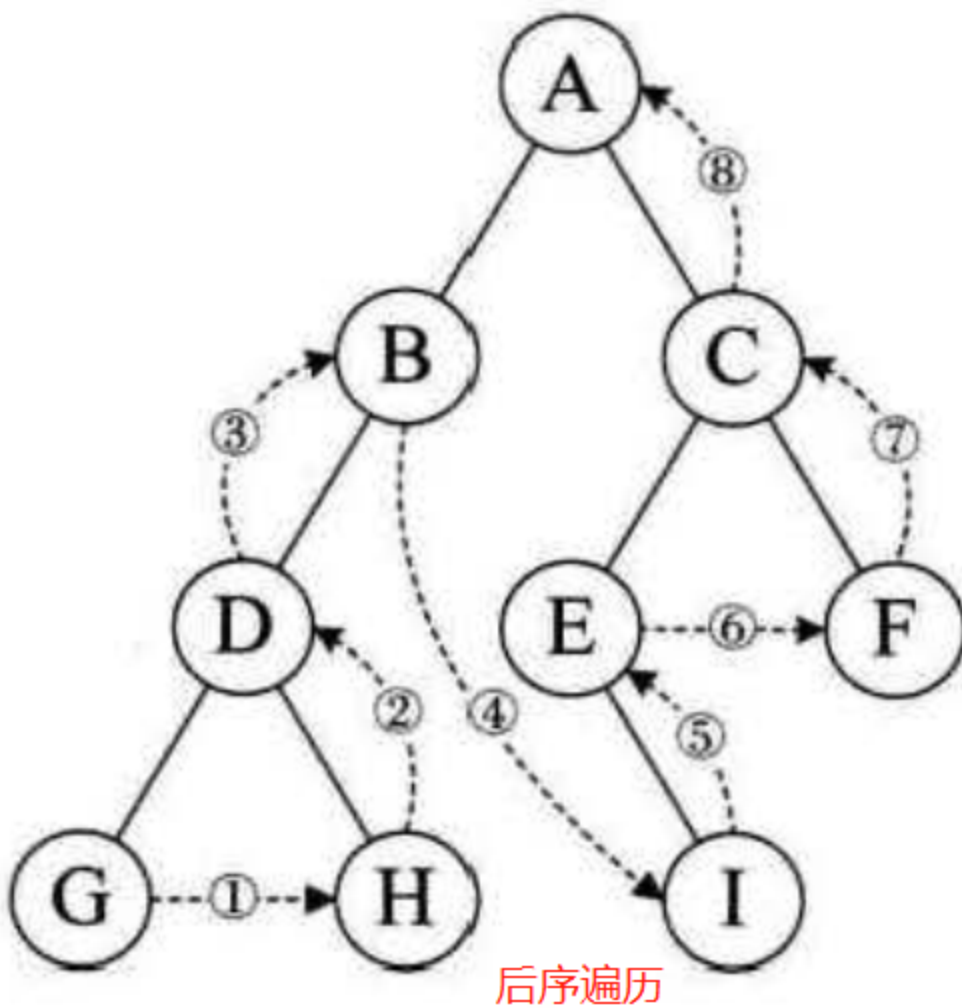
前序遍历

中序过程如下：



中序遍历

后序过程如下：



代码实现：

我们规定有一颗这样的二叉树：

```
1 class TreeNode {
2     constructor(value, left, right) {
3         this.value = value;
4         this.left = left;
5         this.right = right;
6     }
7 }
8
9 let node7 = new TreeNode(7);
10 let node6 = new TreeNode(6);
11 let node5 = new TreeNode(5);
```

```

12 let node4 = new TreeNode(4);
13 let node3 = new TreeNode(3, node6, node7);
14 let node2 = new TreeNode(2, node4, node5);
15 let node1 = new TreeNode(1, node2, node3);
16 // node1 就是我们的根节点。
17 /**
18  *    1
19  *   2  3
20  *  4 5 6 7
21  *
22  */

```

前序遍历（递归方式）：

```

1 var preorderTraversal = function(root) {
2     let arr = [];
3     let traverse = (root) => {
4         // 先处理根节点
5         if(root == null) return;
6         arr.push(root.value);
7         // 递归处理左节点
8         traverse(root.left);
9         // 递归处理右节点
10        traverse(root.right);
11    }
12    traverse(root);
13    return arr;
14 };

```

递归方式很好理解，根左右即可。接下来看看非递归算法，这里面就需要利用到栈结构的知识了。

前序遍历（非递归方式）：

```

1 var preorderTraversal1 = function(root) {
2     if(root == null) return [];
3     let stack = [], res = [];

```

```

4     stack.push(root);
5     while(stack.length) {
6         let node = stack.pop();
7         res.push(node.value);
8         // 左孩子后进先出，进行先左后右的深度优先遍历
9         if(node.right) stack.push(node.right);
10        if(node.left) stack.push(node.left);
11    }
12    return res;
13 };

```

接下来看看中序遍历。

中序遍历（递归方式）：

```

1 let inorderTraversal = function(root) {
2     let arr = [];
3     let traverse = (root) => {
4         if(root == null) return;
5         traverse(root.left);
6         arr.push(root.value);
7         traverse(root.right);
8     }
9     traverse(root);
10    return arr;
11 };

```

中序遍历（非递归方式）：

```

1 let inorderTraversal = function(root) {
2     if(root == null) return [];
3     let stack = [], res = [];
4     let p = root;
5     while(stack.length || p) {
6         while(p) {
7             stack.push(p);

```

```

8         p = p.left;
9     }
10    let node = stack.pop();
11    res.push(node.value);
12    p = node.right;
13 }
14 return res;
15 };

```

接下来看看后序遍历。

后序遍历（递归方式）：

```

1
2 var postorderTraversal = function(root) {
3     let arr = [];
4     let traverse = (root) => {
5         if(root == null) return;
6         traverse(root.left);
7         traverse(root.right);
8         arr.push(root.value);
9     }
10    traverse(root);
11    return arr
12 };

```

后续递归（非递归方式）：

```

1 var postorderTraversal = function(root) {
2     if(root == null) return [];
3     let stack = [], res = [];
4     let visited = new Set();
5     let p = root;
6     while(stack.length || p) {
7         while(p) {
8             stack.push(p);

```

```
9         p = p.left;
10     }
11     let node = stack[stack.length - 1];
12     // 如果右孩子存在，而且右孩子未被访问
13     if(node.right && !visited.has(node.right)) {
14         p = node.right;
15         visited.add(node.right);
16     } else {
17         res.push(node.value);
18         stack.pop();
19     }
20 }
21 return res;
22 };
```