

数据结构与算法

DATA STRUCTURE

第十二、三讲 树
胡浩栋

信息管理与工程学院
2018 - 2019 第一学期

课堂内容

- 树Tree

递归使用看场合

- 规模不大时候，利用call stack机制实现
- 比如把整数转换进制

```
void PrintDigit (int x, int base = 10)
{
    if (x >= base)
    {
        PrintDigit (x / base, base);
    }
    cout << x % base << " ";
}
```

- 树结构里的算法最好不使用递归（循环+栈结构）

树

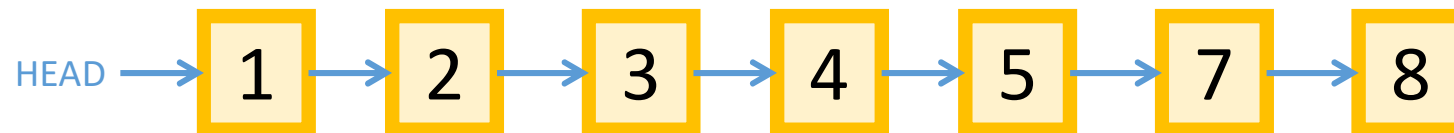
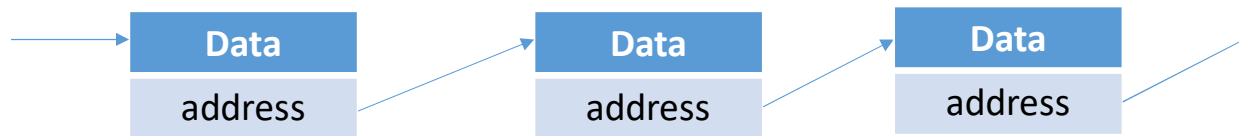
树结构

- 之前讲了array, linklist, stack, queue, 都是线性结构
- 树结构Tree是第一个非线性结构
- 数据结构中的重点

应用

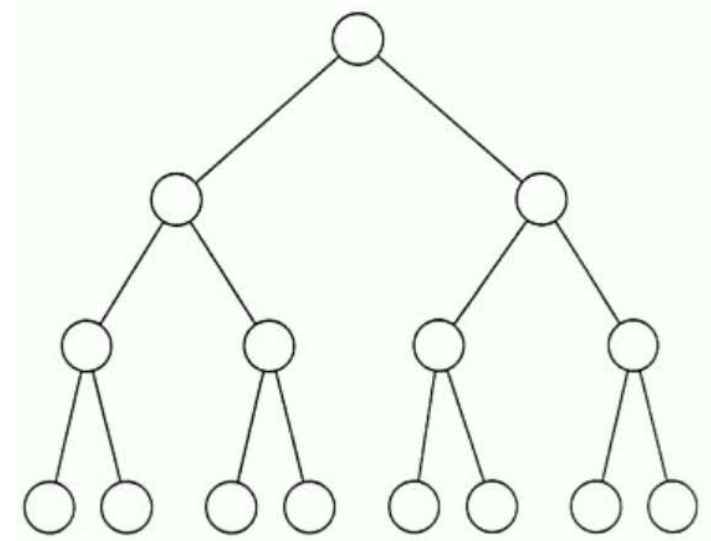
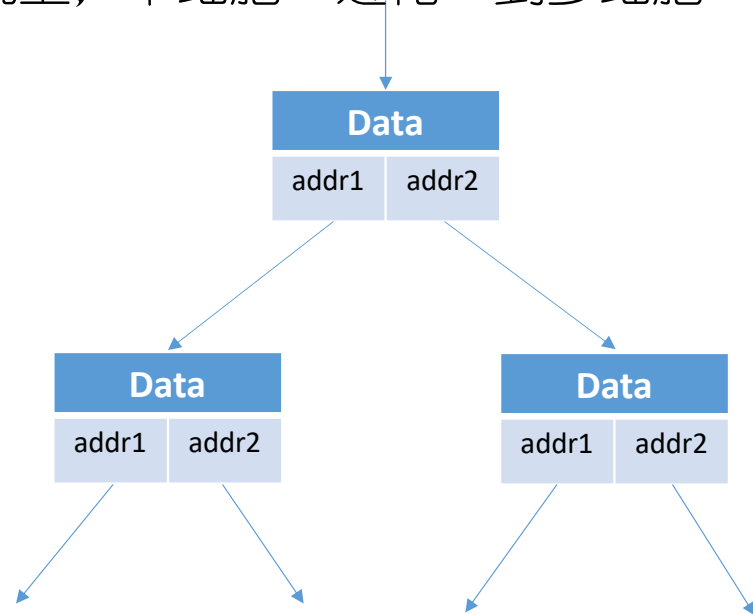
- 数据库用b-tree
- 文件系统
- 编译器
- 图算法
- 常用的数据结构map, dictionary

回顾链表



链表 → 树Tree

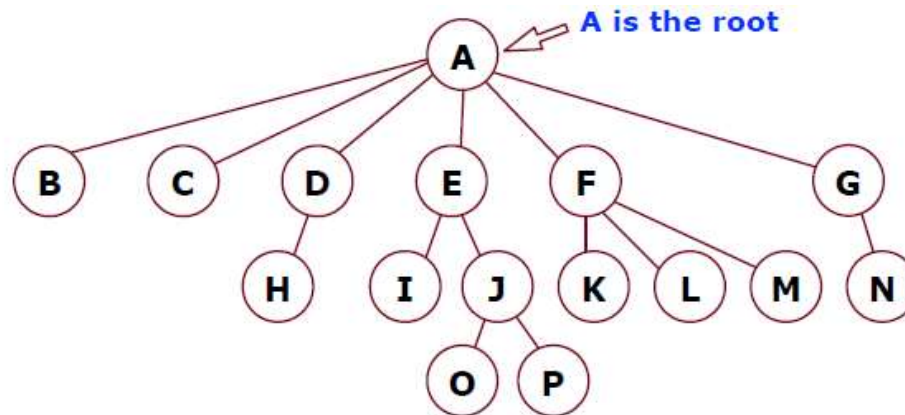
直观上，单细胞“进化”到多细胞



树的定义（递归）

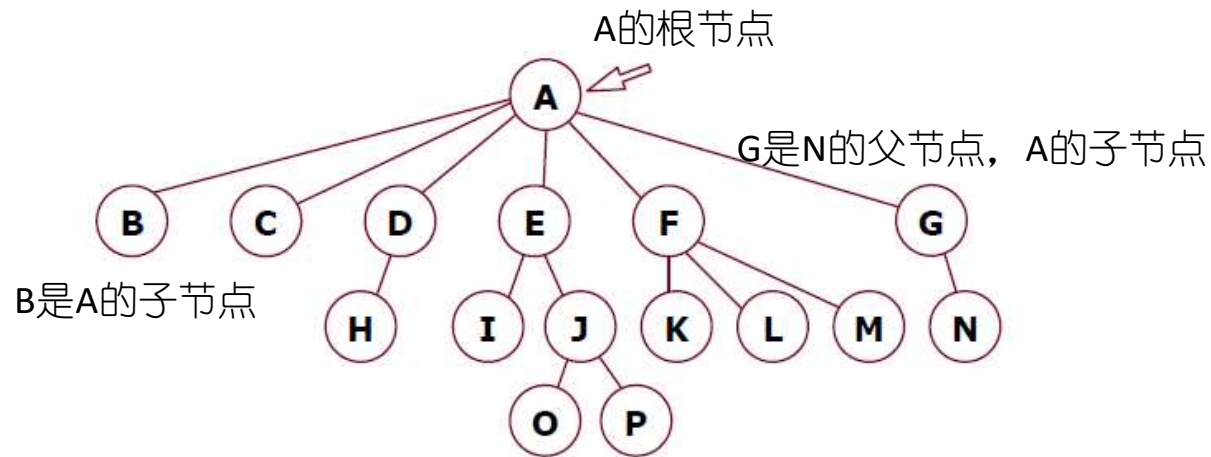
树就是一些节点nodes的集合

- 可以是空集
- 有一个根节点root，和0个或者多个子树 T_1, T_2, \dots, T_k 的根节点直接相连
- T_i 也是树结构，称为根节点root的子树subtree
- 如果有 N 个节点，那么有 $N - 1$ 条边



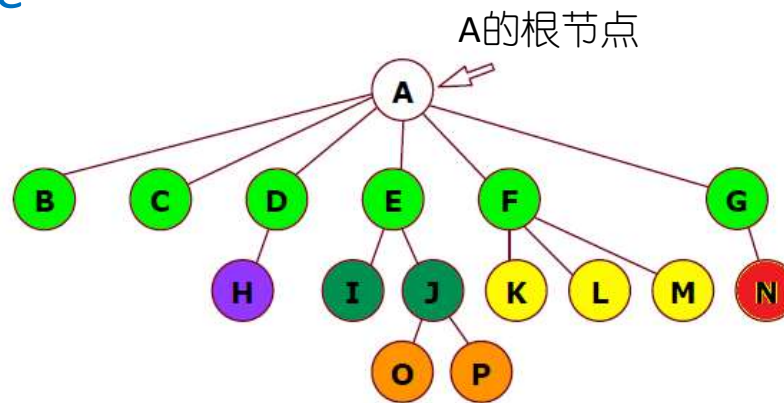
树的术语

- 子节点child node
- 父节点parent node



树的术语

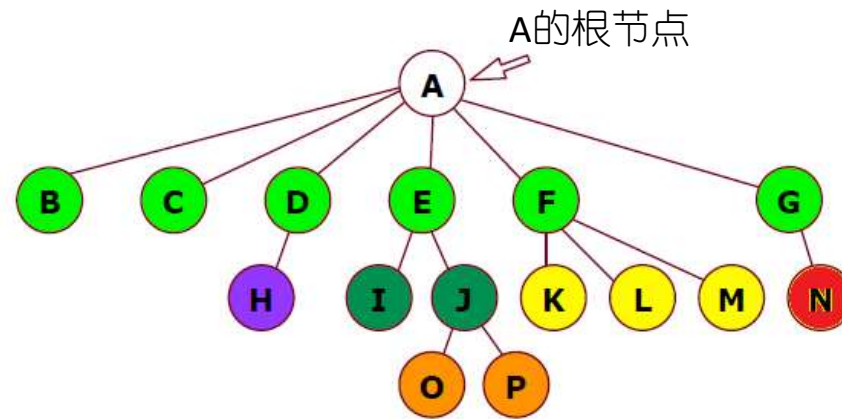
- 子节点child node
- 父节点parent node
- 兄弟节点sibling node



O和P是兄弟节点，
因为有共同的父节点J

树的术语

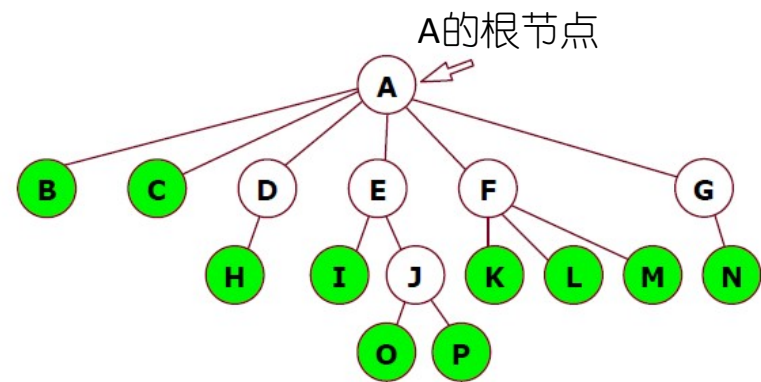
- 子节点child node
- 父节点parent node
- 兄弟节点sibling node
- 祖先节点ancestor
- 子孙节点descendant



O和P是A的子孙节点，
或者说A是O和P的祖先节点

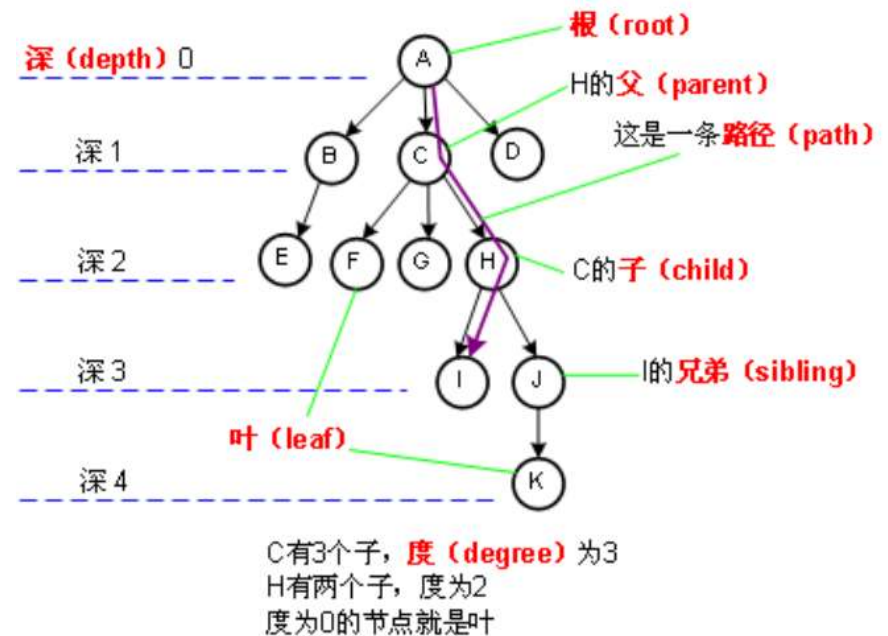
树的术语

- 叶节点 **leaf node** :
所有没有子节点的节点
- 度 **degree** 是指节点有几个子节点



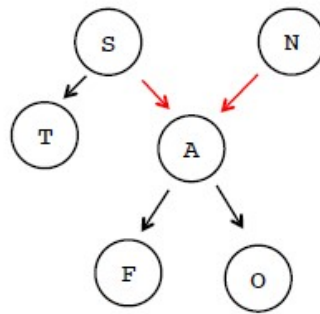
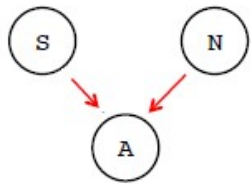
树的术语

- 定义路径path从某个父节点到子节点
- 路径的长度length就是边的数目
- 一个节点的深度depth就是从根节点到它自己的路径的长度
 - 根节点深度是0
 - J的深度是3
- 一个节点的高度height就是从它自己到子叶节点的最长路径的长度
 - H的高度是2
 - 叶节点的高度是0
- 树的高度height就是根节点的高度
 - 例子中树的高度是4

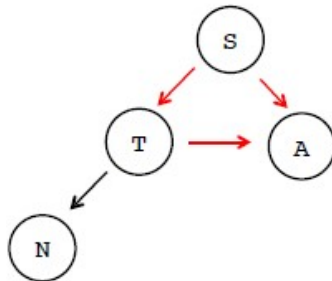
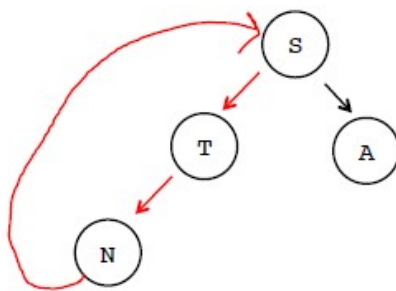


树的特点

- 每个节点只能有一个父节点



- 不能形成闭路



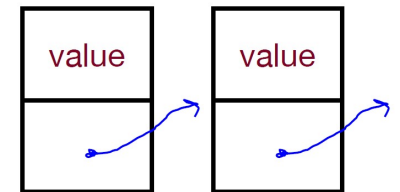
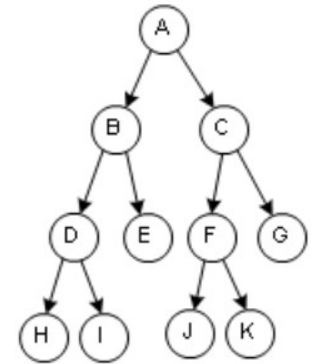
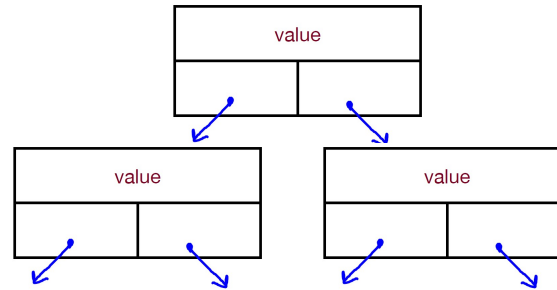
树的种类

- 无序树：树中任意节点的子结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：树中任意节点的子结点之间有顺序关系，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；
- 二叉查找树
- 堆
- 哈夫曼树
- 红黑树，AVL树，等变种

树的表达方式

- 类似于链表，只是链表只指向下一个元素
- 二叉树指向下两个元素

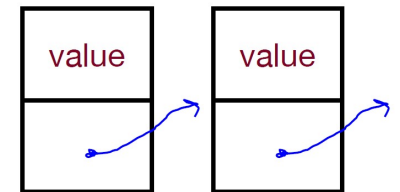
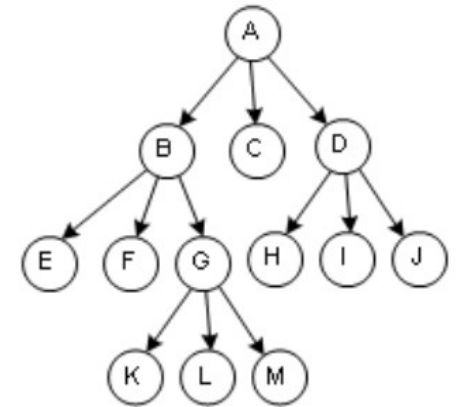
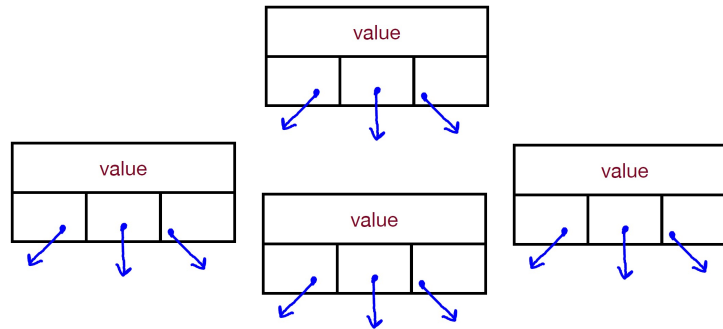
```
struct TreeNode
{
    int key;
    TreeNode *left;
    TreeNode *right;
};
```



树的表达方式

- 类似于链表，只是链表只指向下一个元素
- 三叉树指向下三个元素

```
struct TreeNode
{
    int key;
    TreeNode *left;
    TreeNode *middle;
    TreeNode *right;
};
```

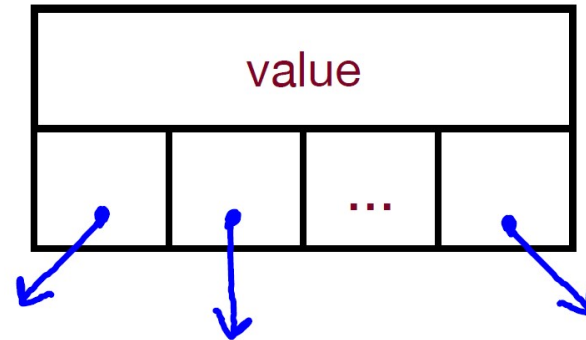


树的表达方式

- 类似于链表，只是链表只指向下一个元素
- 树指向下几个元素

```
struct TreeNode
{
    int key;
    vector<TreeNode*> children;
};

class Tree
{
private:
    TreeNode* root;
};
```



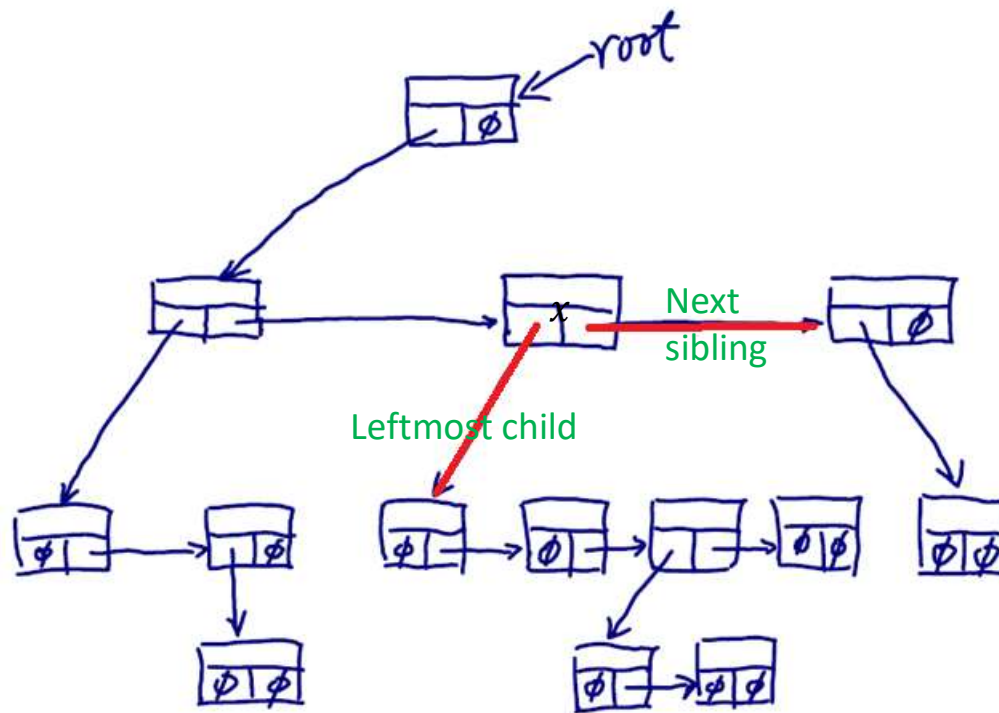
这里std::vector是动态数组。
如果用静态数组，缺点是一个我们不知道有多少子节点；
另一个是就算我们知道有上界，也会浪费很多空间

树的表达方式（二）

- 每个节点只用两个指针，却能表示一般的树（多个子节点）
 - 巧妙的用了链表来表示多个子节点
1. 节点 x 的一个指针指向 x 的最左边的子节点
 2. 节点 x 的另一个指针指向 x 的下一个兄弟节点

```
struct TreeNode
{
    int key;
    TreeNode * leftMost;
    TreeNode * nextSibling;
};
```

左子节点，右兄弟结点表达方式



父节点表示法

每个节点同样由两个部分构成，第一个部分仍然是数据，第二个部分也是指针，但指向父节点

由于树中每个结点最多只有一个父节点，因此每个节点只需要保存一个指针。

- 可根据指针是否为空判断一个结点是否为根节点。
- 找父节点容易，找子节点难。
- 这种方法一般是辅助手段，不是主要的表达树的方式。

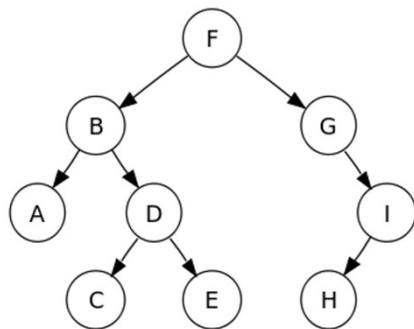
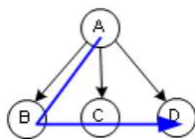
树遍历traversal的方法

树的遍历是按一定顺序访问树中每个结点，同时每个结点不要重复访问（数据）

- 前序遍历Pre-order：深度优先
- 后序遍历Post-order：深度优先
- 按层遍历Level-order：广度优先
- 中序遍历In-order：

前序遍历

- 先对当前节点操作
- 从左到右找子节点



前序遍历输出：F B A DCE GIH

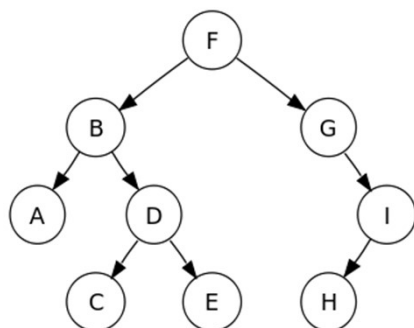
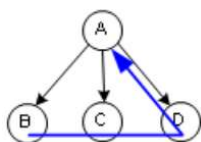
```
struct TreeNode
{
    int key;
    vector<TreeNode*> children;
};

void PreOrder(const TreeNode* root)
{
    if (root == nullptr)
    {
        return;
    }

    cout<< root->key << " ";
    for (int i = 0; i < root->children.size(); i++)
    {
        PreOrder(root->children[i]);
    }
}
```


后序遍历

- 从左到右找子节点
- 最后对当前节点操作



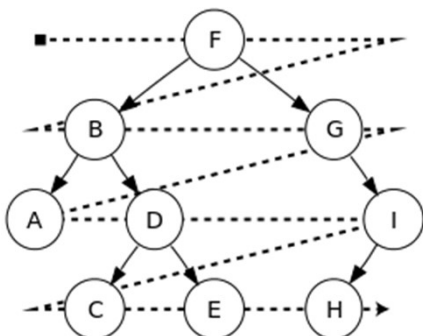
- 后序遍历输出: **A CED B HIG F**
- 回忆表达式语法树的后序遍历--
后缀表示法

```
struct TreeNode
{
    int key;
    vector<TreeNode*> children;
};
```

```
void PostOrder(const TreeNode* root)
{
    if (root == nullptr)
    {
        return;
    }
    for (int i = 0; i < root->children.size(); i++)
    {
        PostOrder(root->children[i]);
    }
    cout<< root->key << " ";
}
```

按层遍历

- 逐层，从左到右
- 广度优先
- 用队列实现，类似用queue实现stack



- 按层遍历输出：F BG ADI CEH
- 回忆杨辉三角形的输出遍历

```
struct TreeNode
{
    int key;
    vector<TreeNode*> children;
};

void LevelOrder(const TreeNode* root)
{
    queue<const TreeNode*> treeQueue;

    if (root != nullptr)
    {
        treeQueue.push(root);
    }

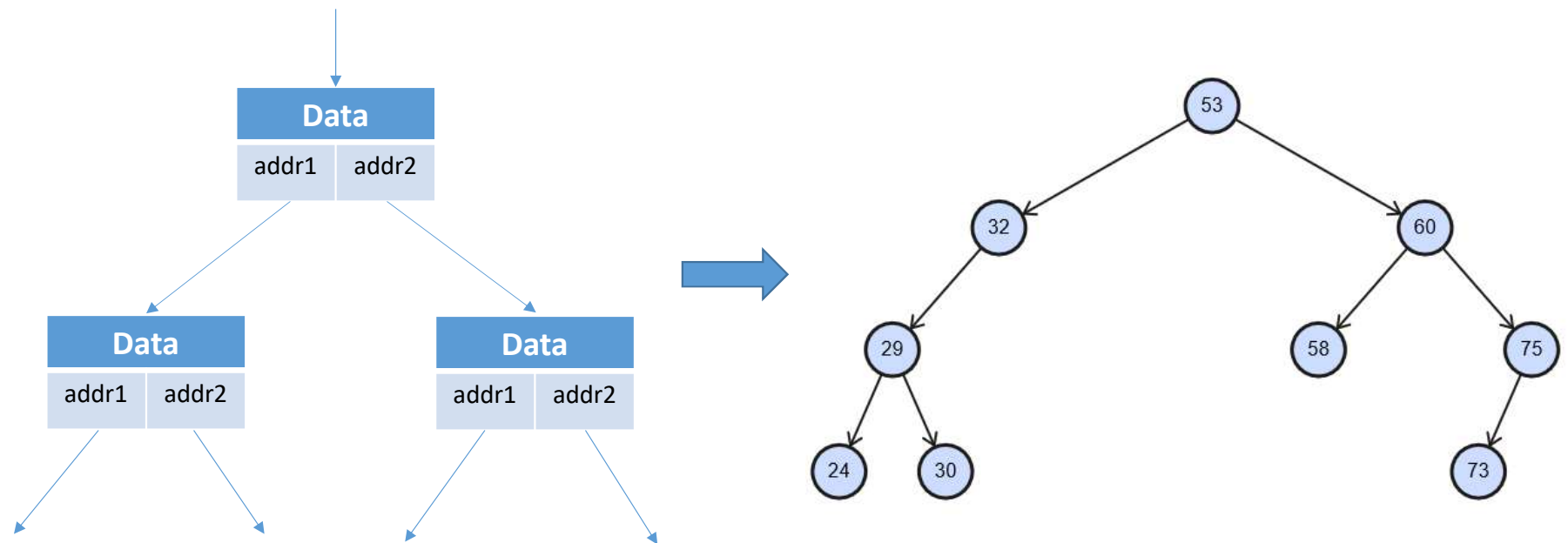
    while (!treeQueue.empty())
    {
        const TreeNode* node = treeQueue.front();
        treeQueue.pop();
        cout << node->key << " ";

        for (int i = 0; i < node->children.size(); i++)
        {
            treeQueue.push(node->children[i]);
        }
    }
}
```

二叉树

Binary Tree

每个节点最多两个子节点

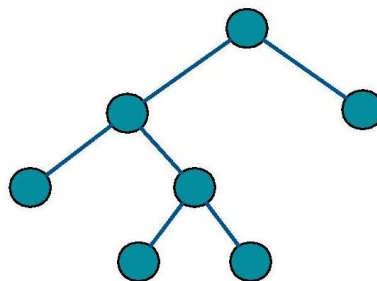


二叉树性质

- 性质 1：在一棵非空的二叉树的第 i 层上至多有 2^i 个节点
- 性质 2：一棵高度为 h 的二叉树至多有 $2^{h+1} - 1$ 个节点
- 性质 3：有 n 个节点的二叉树，最小高度为 $\log_2(n + 1) - 1$
- 性质 4：（二叉）树的边数 $e =$ 节点数 $n - 1$
- 性质 5：对于任何一棵非空的二叉树，如果其叶节点个数为 n_0 ，度为 2 的节点个数为 n_2 ，则 $n_0 = n_2 + 1$ 。
- 性质 6：有 L 个叶节点的二叉树，最小高度为 $\log_2 L$

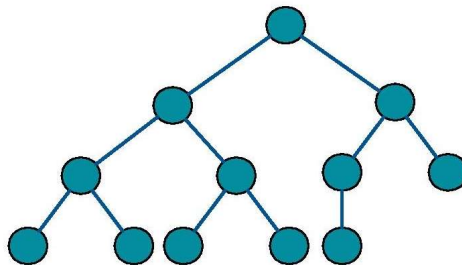
特殊二叉树

- 满full二叉树：每个节点要么有两个子节点，要么没有子节点
 - 性质：有 n 个节点的满二叉树， h 是树的高度，那么 $2^h + 1 \leq n \leq 2^{h+1} - 1$



- 完全complete二叉树

- 性质：有 n 个节点的完全二叉树，内部节点数是 $\lfloor n/2 \rfloor$
- 性质：有 n 个节点的完全二叉树，高度是 $\lfloor \log_2(n + 1) \rfloor - 1$



野生二叉树



二叉树遍历方法

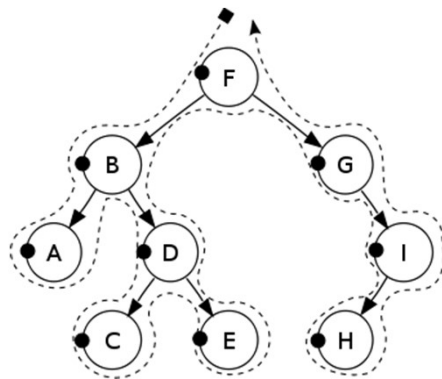
- 前序遍历Pre-order：深度优先
- 后序遍历Post-order：深度优先
- 中序遍历In-order：深度优先
- 按层遍历Level-order：广度优先

有其用途：

- 比如之前的计算表达式
- 比如图算法，道路搜索

前序遍历

- 先对当前节点操作
- 找左子节点
- 找右子节点



前序遍历输出：F B A DCE GIH

```
struct BNode
```

```
{
```

```
    int key;
```

```
    BNode * left;
```

```
    BNode * right;
```

```
};
```

```
void PreOrder(const BNode * root)
```

```
{
```

```
    if(root == nullptr)
```

```
    {
```

```
        return;
```

```
    }
```

```
    cout << root->key << " ";
```

```
    PreOrder(root->left);
```

```
    PreOrder(root->right);
```

```
}
```

前序遍历迭代解法

观察一个节点在第一次被访问到就能输出

1. 创建一个空stack，并把根节点压入栈
2. 如果栈不空，就一直循环：
 - 从栈里Pop一个元素node，然后打印
 - 把node的右子节点压入栈
 - 把node的左子节点压入栈

```
void PreOrder_Iterative(const BNode * root)
{
    stack<const BNode *> stk;
    if (root)
    {
        stk.push(root);
    }

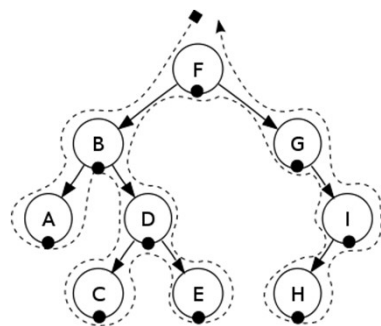
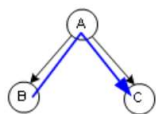
    while (!stk.empty())
    {
        const BNode * node = stk.top();
        stk.pop();

        cout << node->key << " ";

        if (node->right)
        {
            stk.push(node->right);
        }
        if (node->left)
        {
            stk.push(node->left);
        }
    }
}
```

中序遍历

- 找左子节点
- 再对当前节点操作
- 找右子节点



中序遍历输出: A B CDE F GHI

```
struct BNode
{
    int key;
    BNode * left;
    BNode * right;
};
```

```
void InOrder(const BNode * root)
{
    if(root == nullptr)
    {
        return;
    }
    InOrder(root->left);
    cout<< root->key << " ";
    InOrder(root->right);
}
```

中序遍历迭代解法

- 观察一个节点只有第二次被访问到才能输出
- 所以不能像preorder一次访问出栈
- 所以需要额外的指针跟踪当前访问节点
 1. 对于当前访问的节点，只要还有左子节点，它得压入栈保存，等第二次访问到
 2. 如果没有左子节点了，相当于从空左子节点返回了，栈顶节点可以输出
 3. 之后当前访问节点跟踪到右子节点后重复之前的步骤

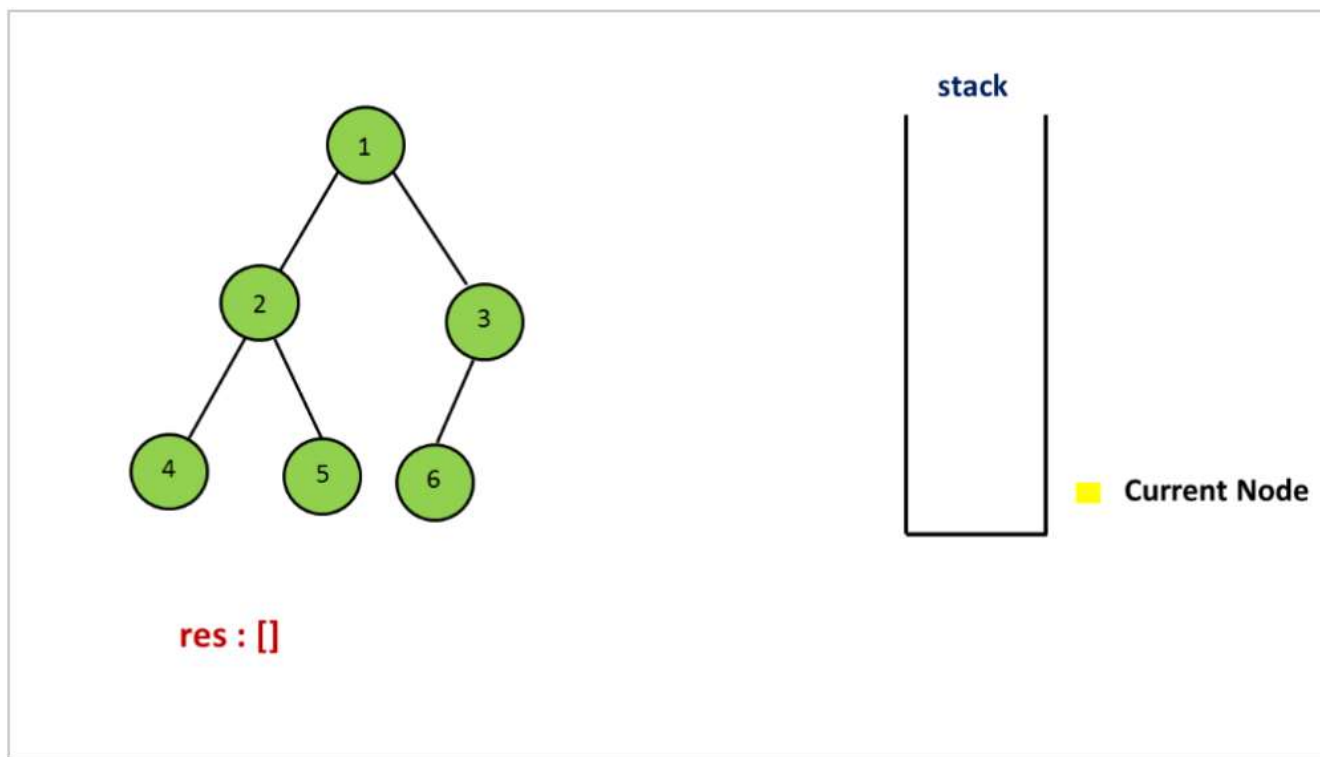
```
void InOrder_Iterative(const BNode * root)
{
    stack<const BNode *> stk;
    const BNode * curr = root;

    while (!stk.empty() || curr)
    {
        if (curr)
        {
            stk.push(curr);
            curr = curr->left;
            continue;
        }

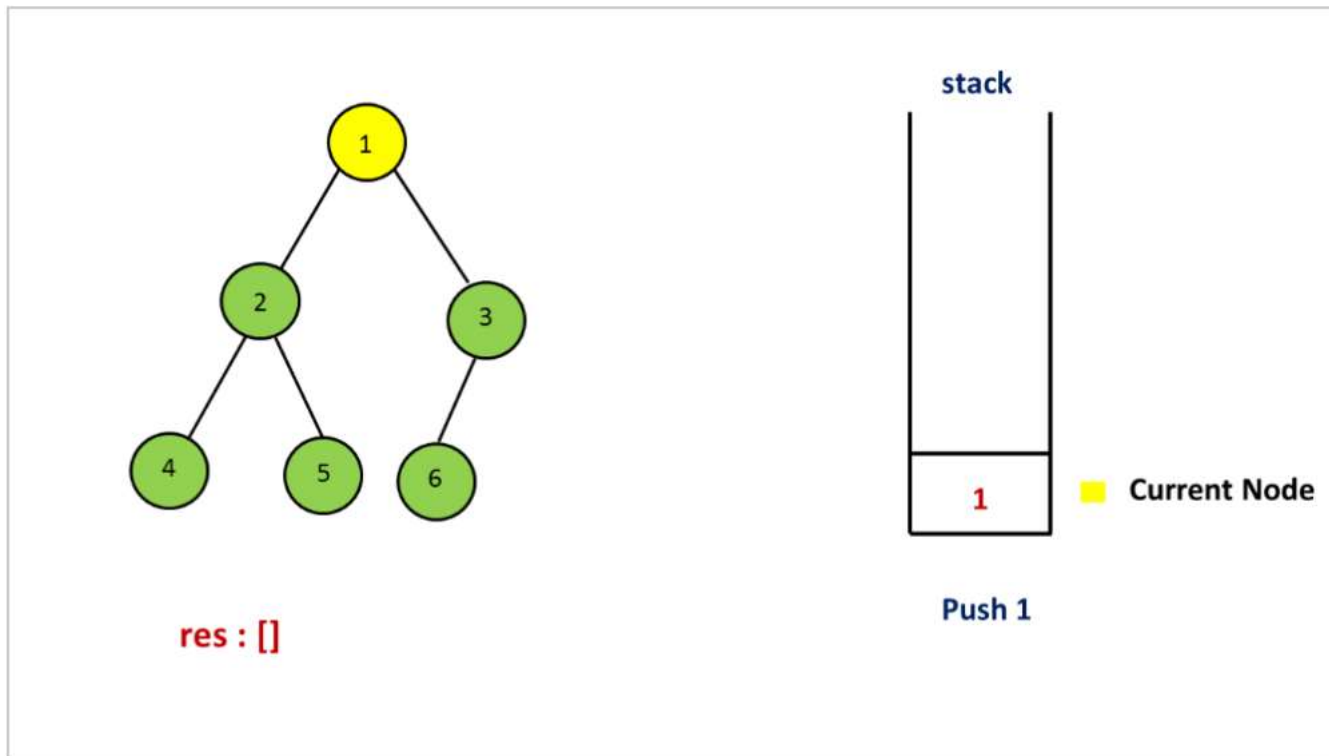
        curr = stk.top();
        stk.pop();

        cout << curr->key << " ";
        curr = curr->right;
    }
}
```

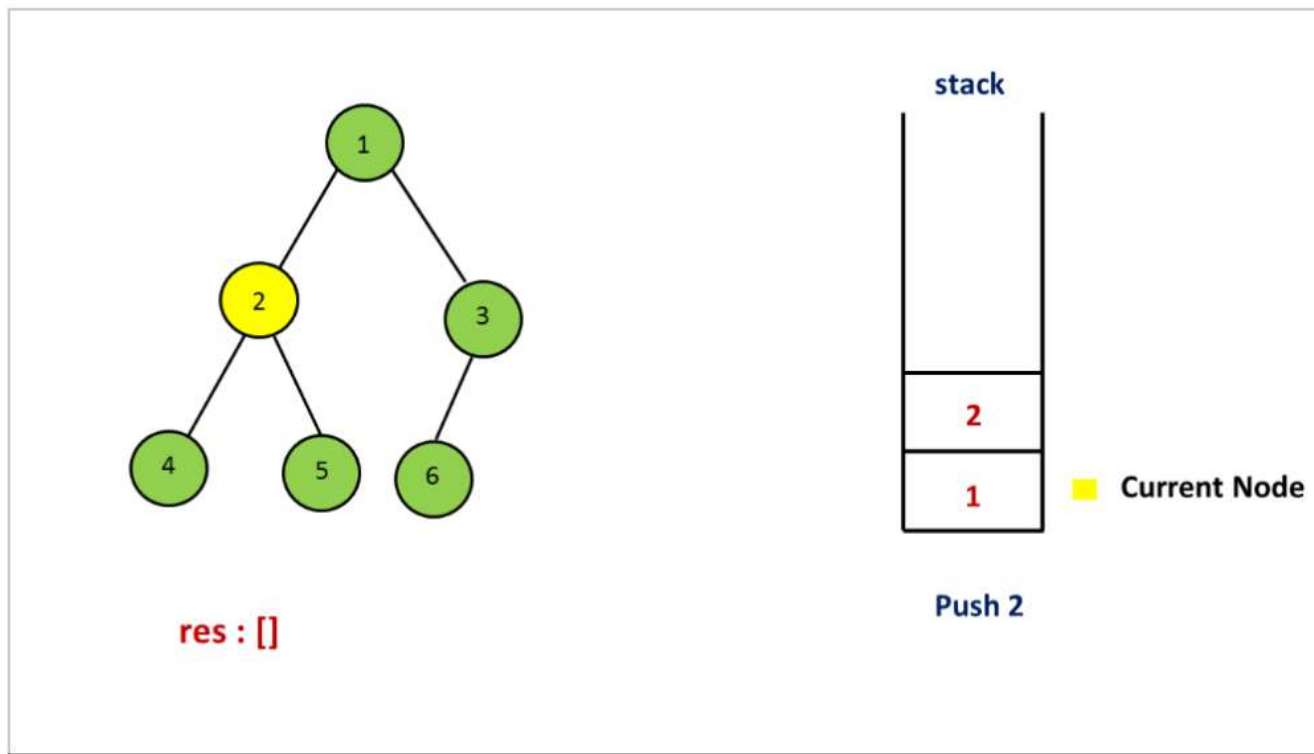
1.初始化栈



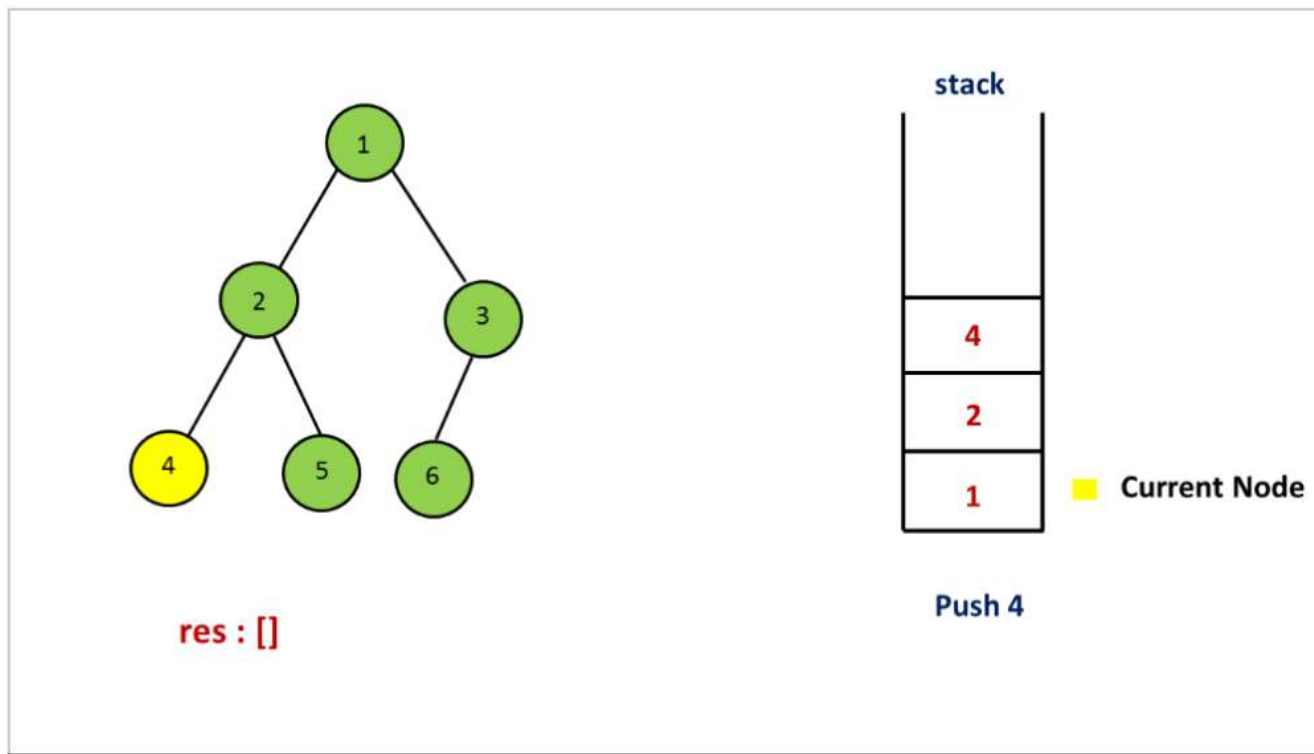
2. Curr1压入栈，并跟踪其左子节点



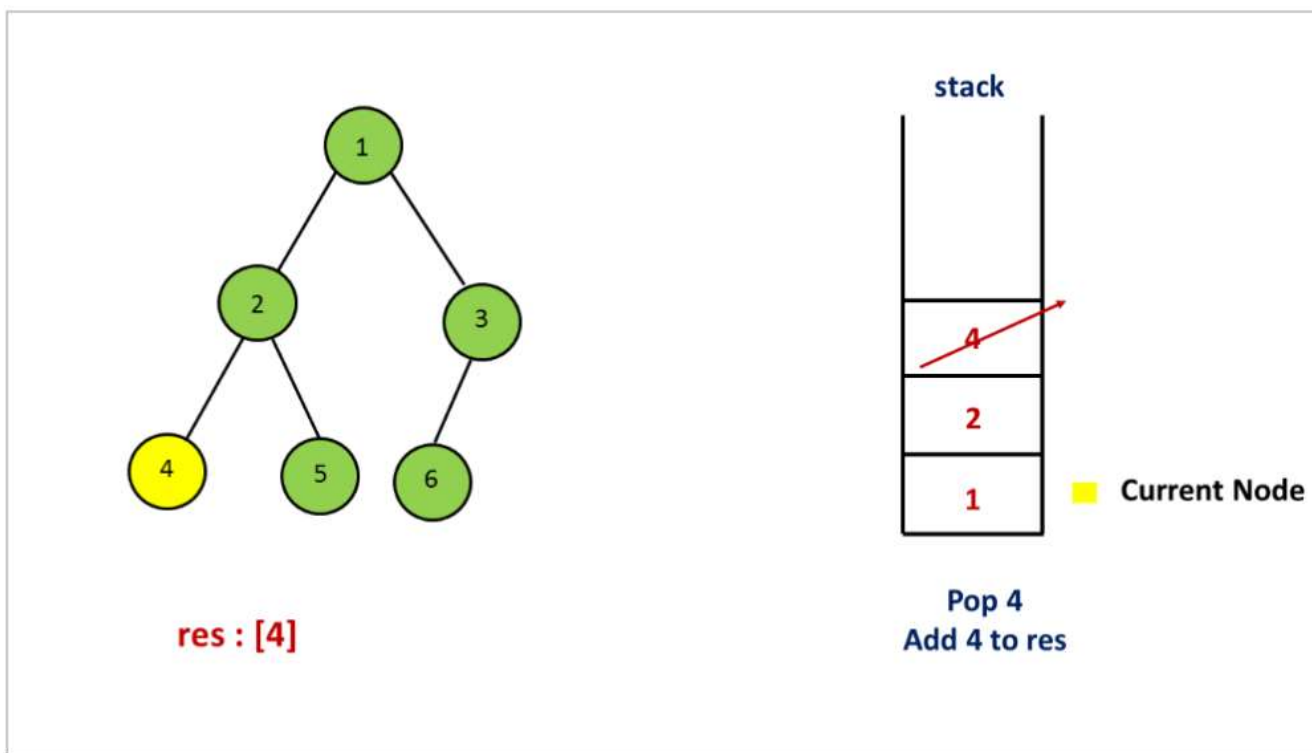
3. Curr2压入栈，并跟踪其左子节点



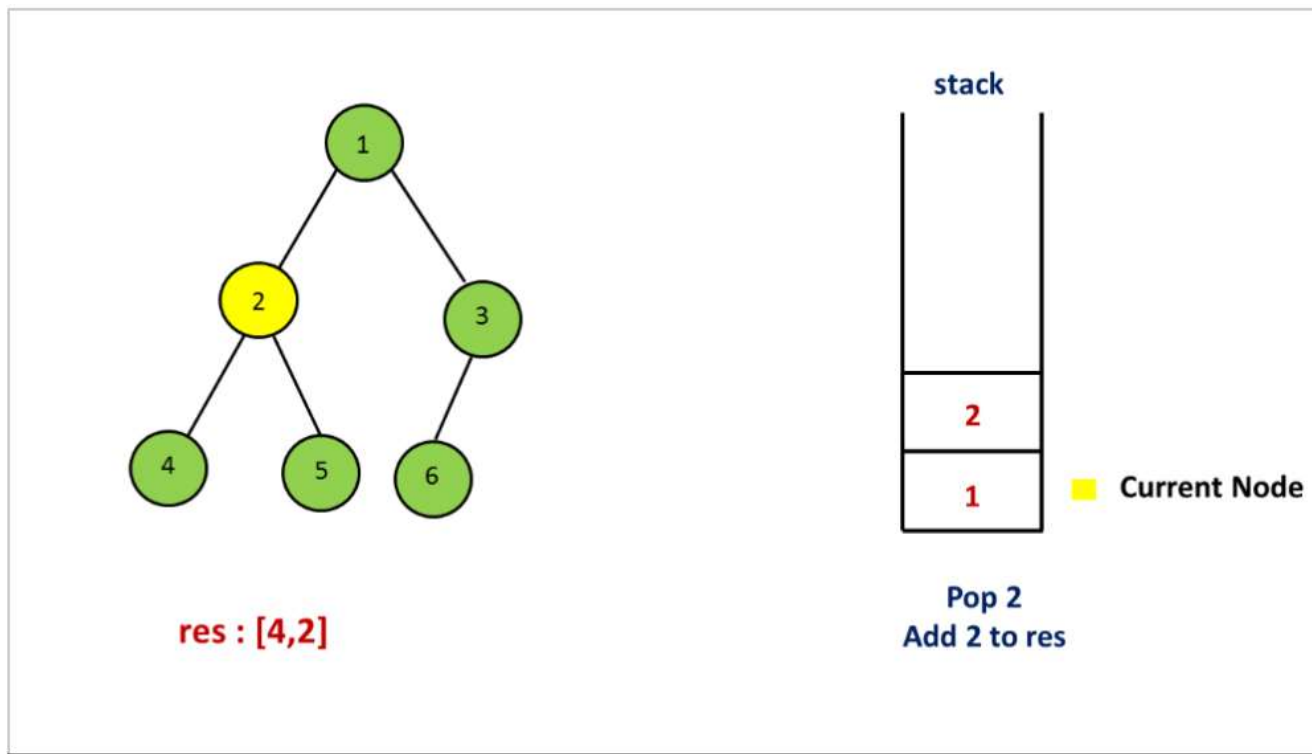
4. Curr4压入栈，并跟踪其左子节点



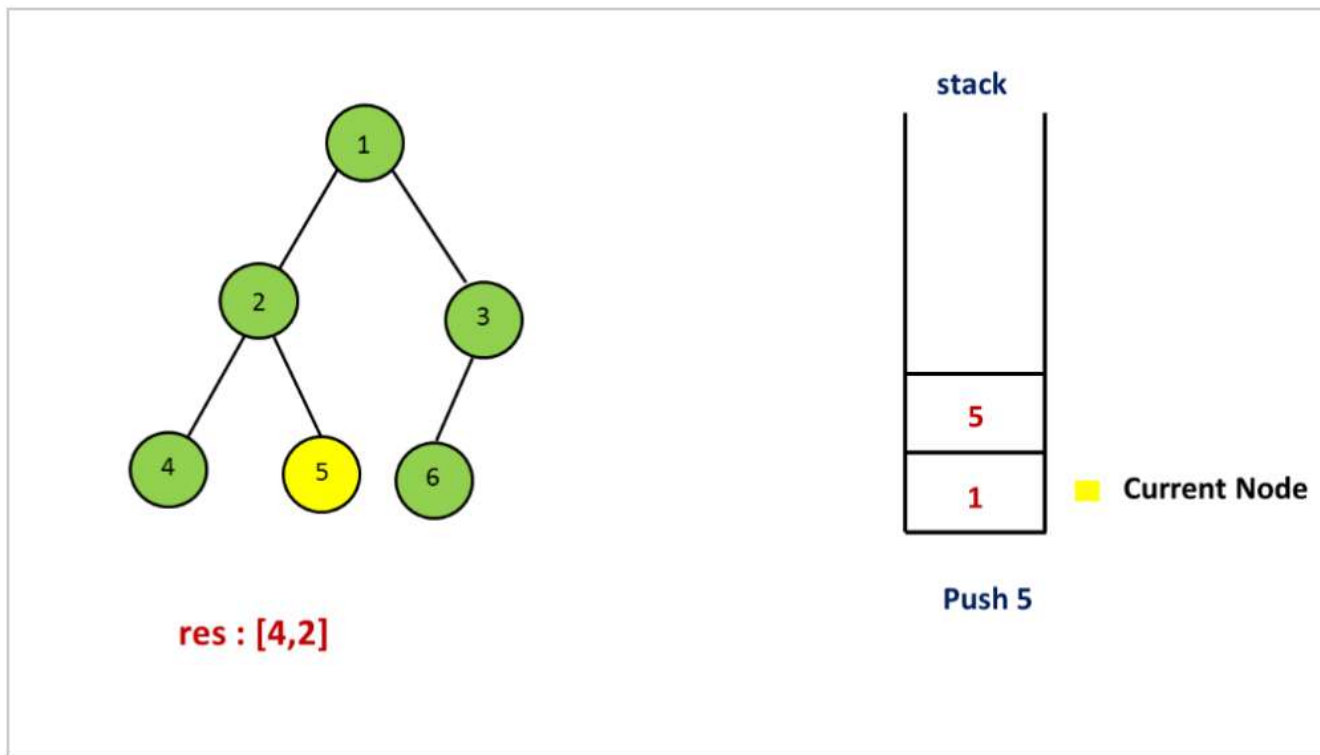
5. Curr 为空，只能第二次访问栈顶节点4，
所以出栈输出后，跟踪其右子节点



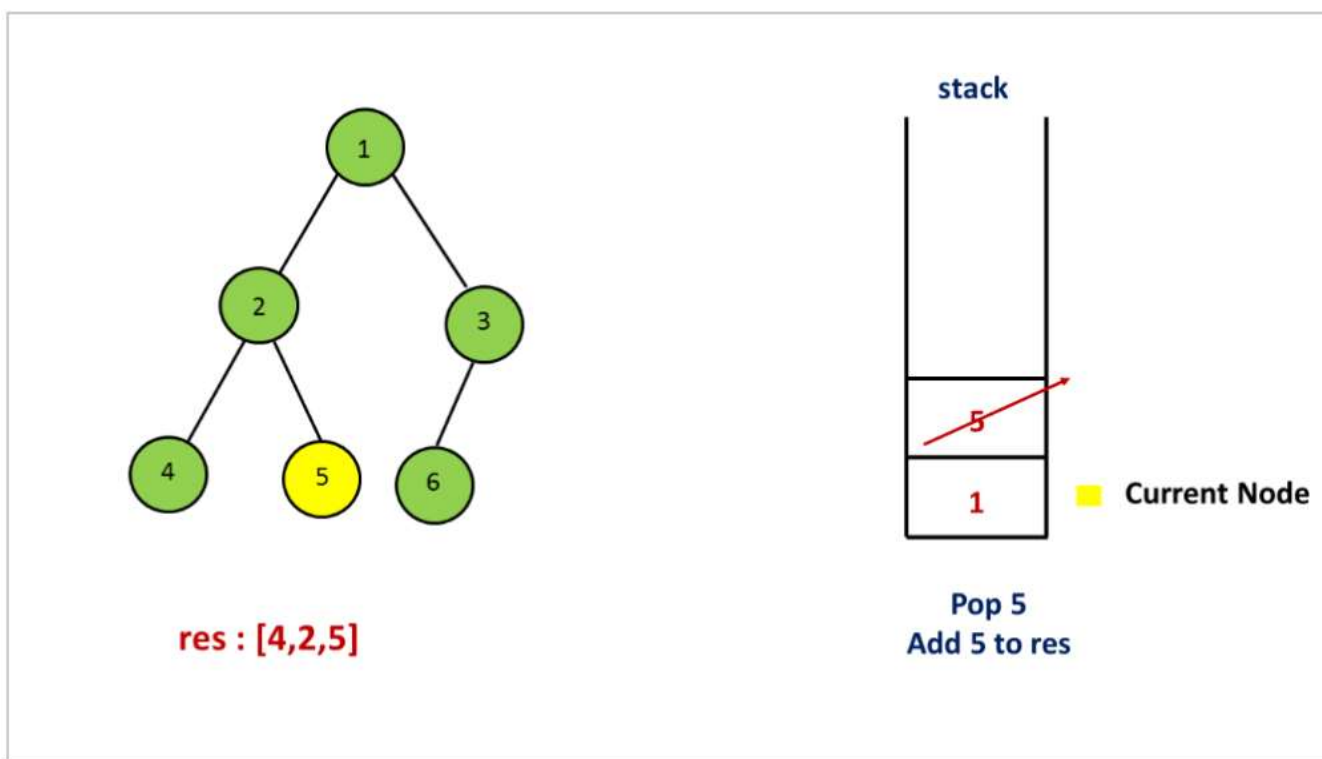
6. **Curr**是4的右子节点，还是空。只能第二次访问栈顶节点**2**。出栈输出**2**后，跟踪右子节点



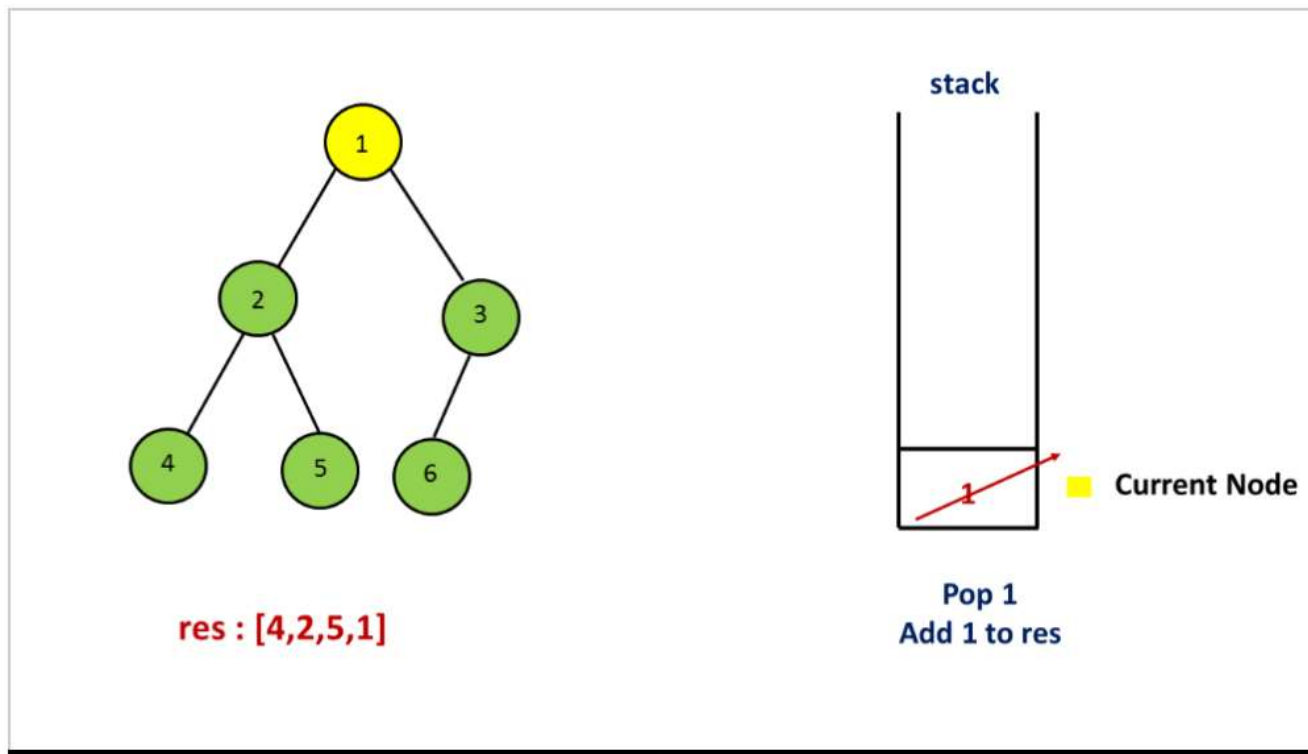
7. Curr5压入栈，并跟踪其左子节点



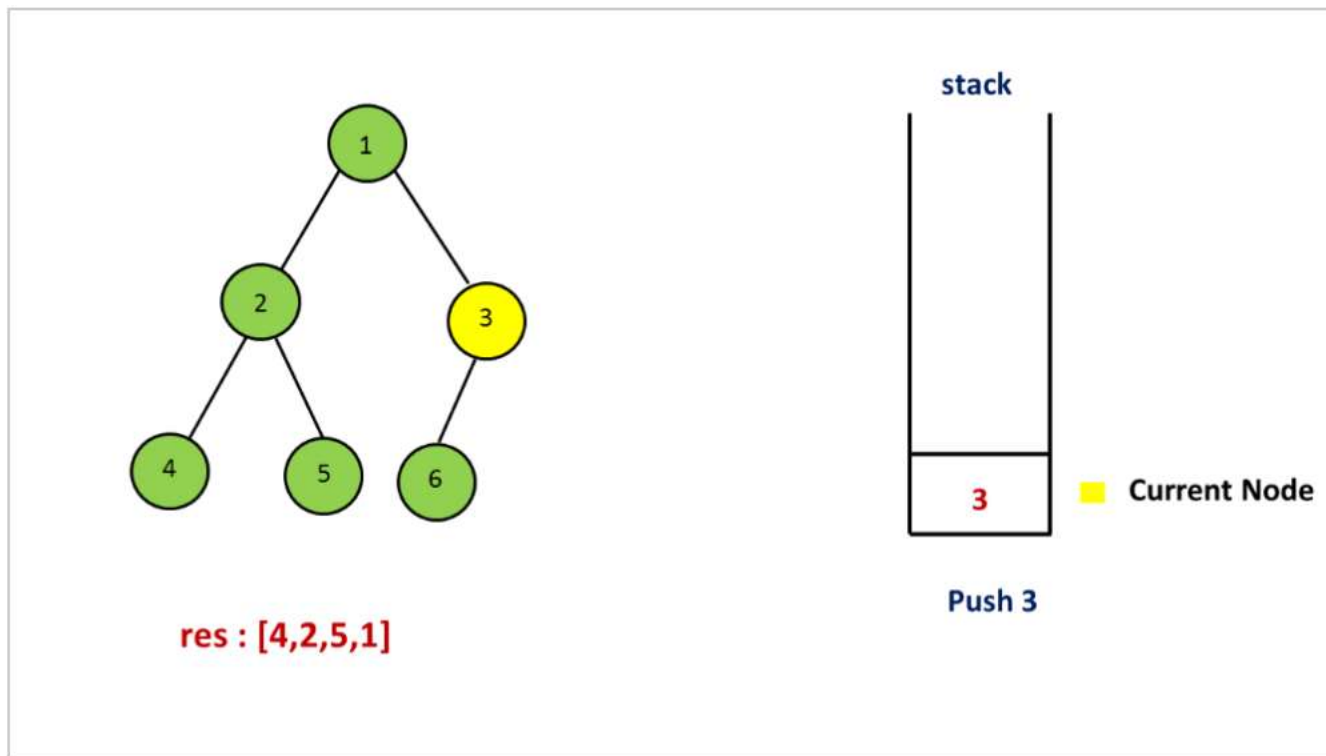
8. **Curr**为空，只能第二次访问栈顶节点**5**，
所以出栈输出后，跟踪其右子节点



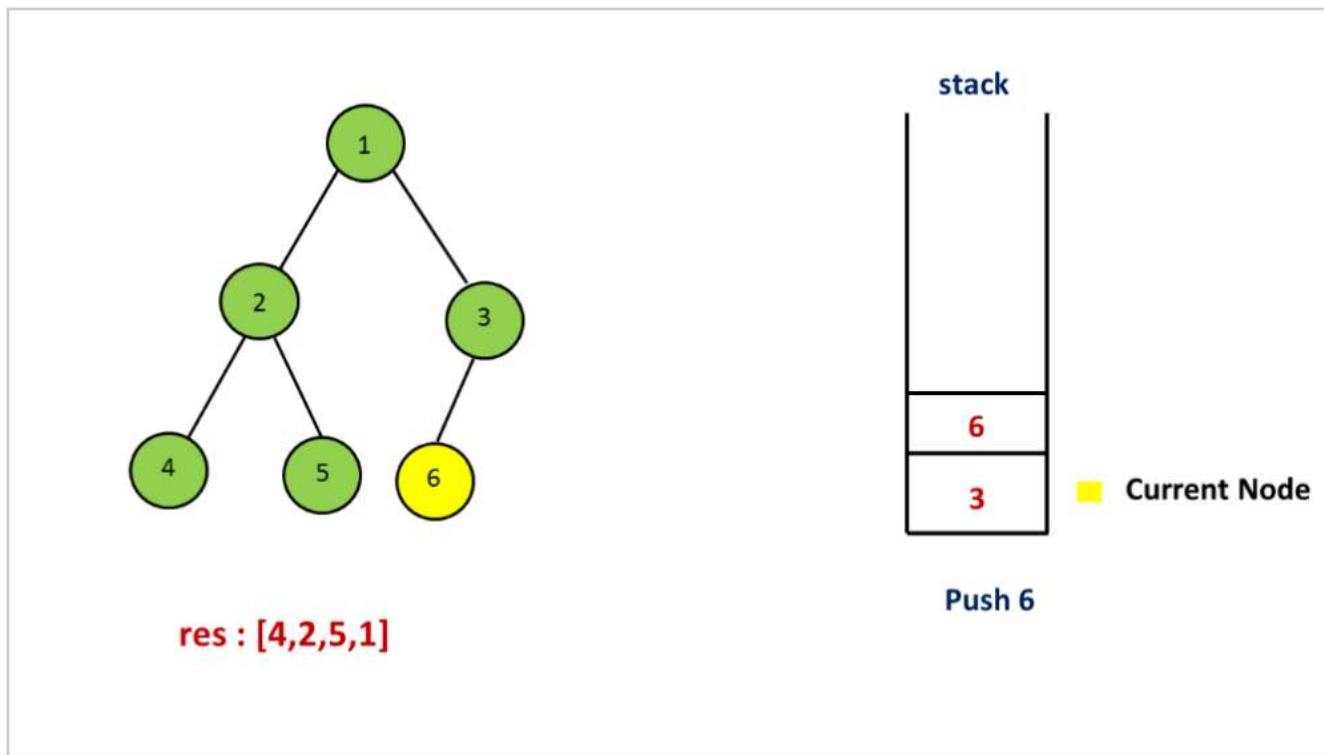
9. **Curr**是**5**的右子节点，还是空。只能第二次访问栈顶节点**1**。出栈输出**1**后，跟踪右子节点**3**



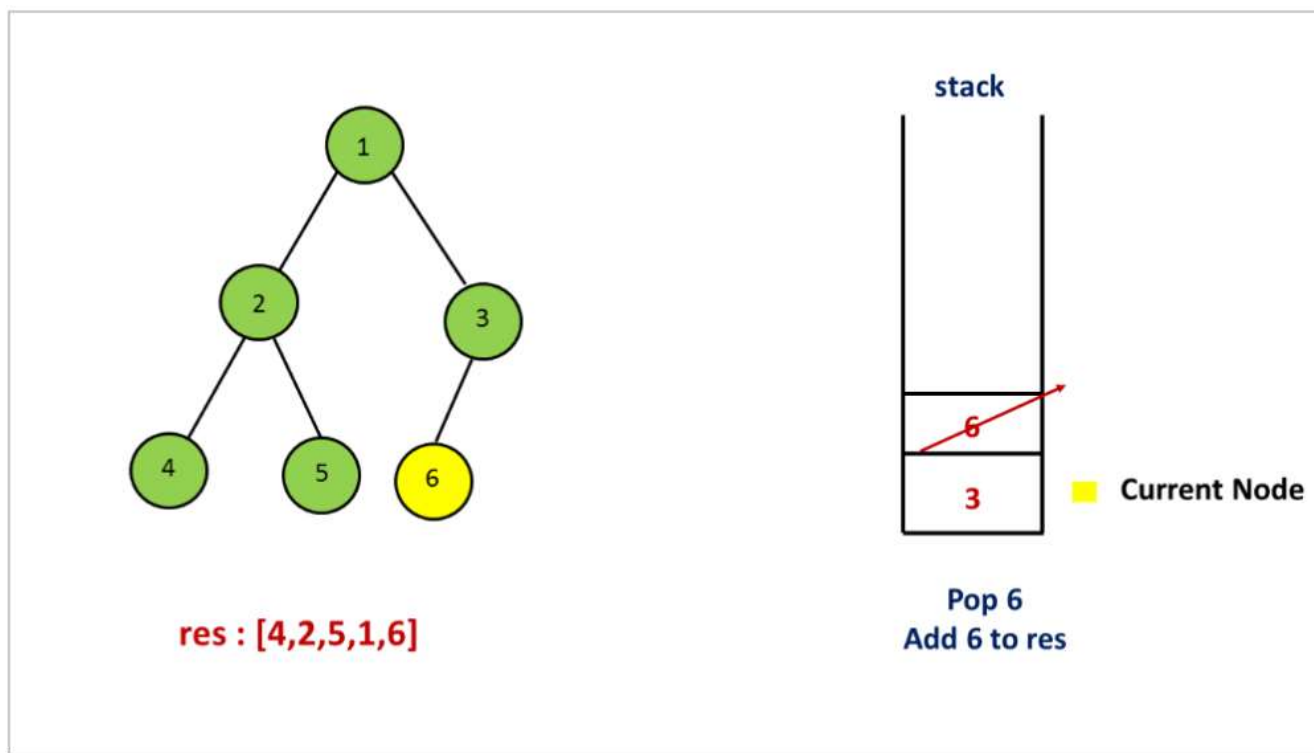
10.把curr3压入栈后，跟踪其左子节点



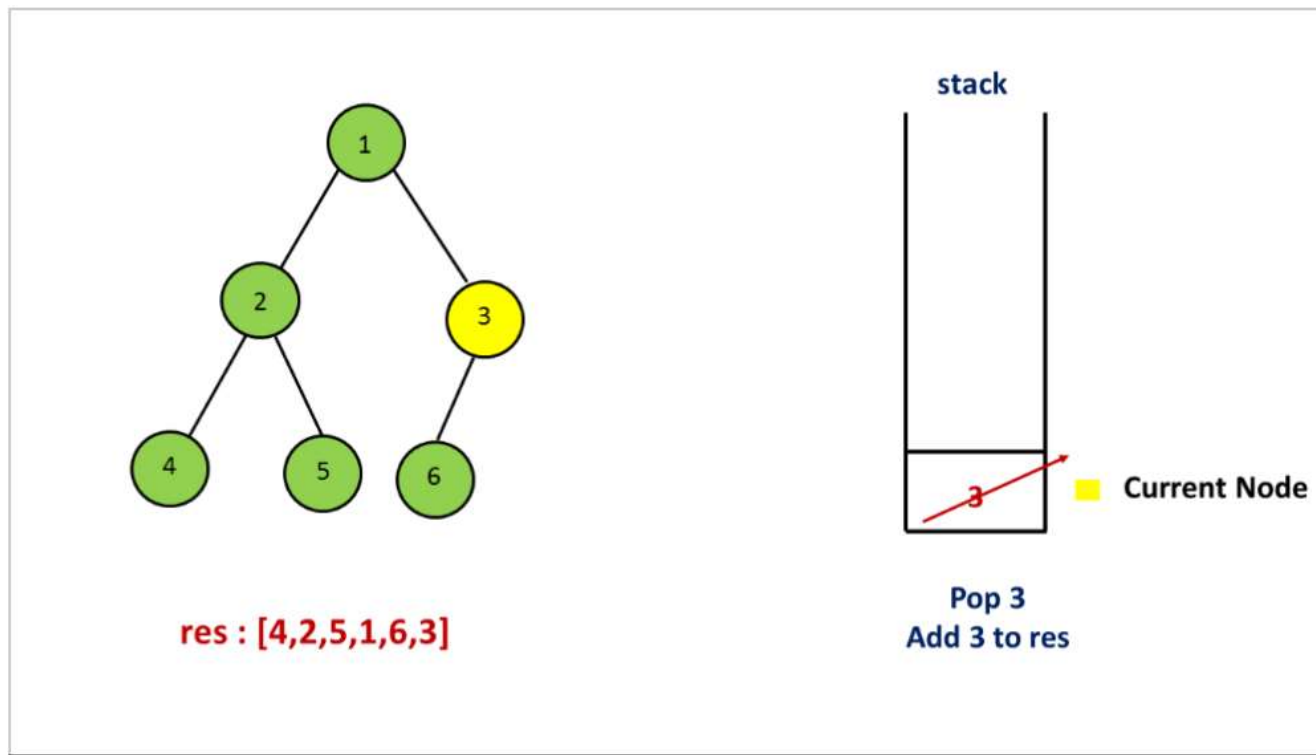
11. 把curr6压入栈后，跟踪其左子节点



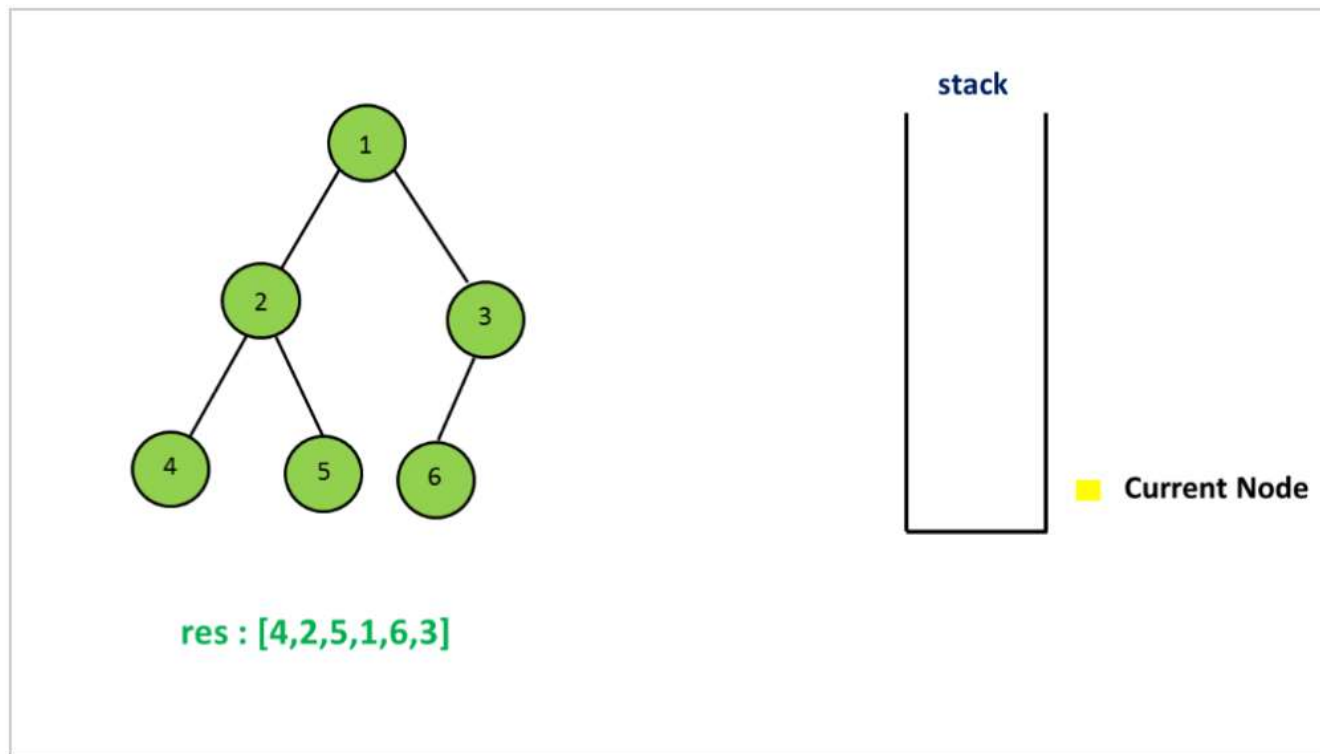
12. **Curr**为空，只能第二次访问栈顶节点6，
所以出栈输出后，跟踪其右子节点



13. **Curr**是6的右子节点，还是空。只能第二次访问栈顶节点**3**。出栈输出**3**后，跟踪右子节点

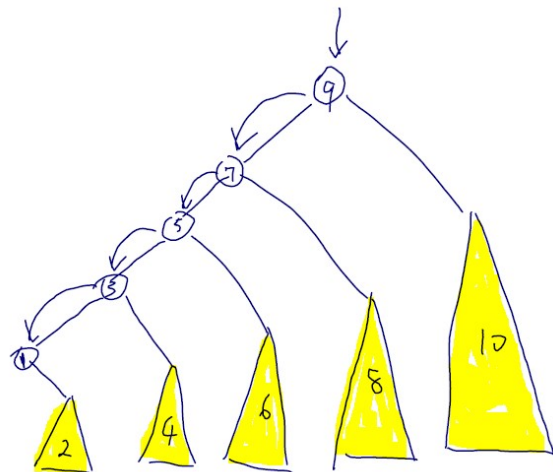


14. curr是3的右子节点，还是空。结束输出



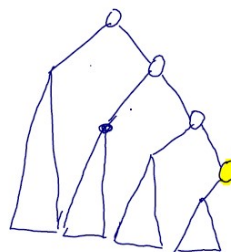
数学归纳法证明中序迭代算法

首先观察一下中序从左到右的元素
然后验证算法是按着这个顺序来的



观察按这个算法，

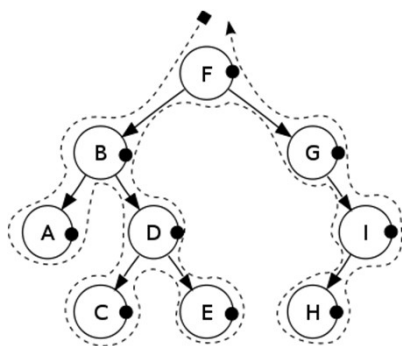
1. 算法结束的条件是栈空了，并且当前指针的右子节点是空
2. 算法能访问到树里的所有节点，即不会遗漏
3. 并且一个树里有且仅有一个元素满足退出条件，
4. **Rightmost**节点一定是最后访问的，满足退出条件，
5. 算法在进行子树2、4的中序输出时，不会破坏栈里之前压入元素



- 1.
- 2.
- 3.
- 4.
- 5.

后序遍历

- 找左子节点
- 找右子节点
- 最后对当前节点操作



- 后序遍历输出: A CED B HIG F

```
struct BNode
{
    int key;
    BNode * left;
    BNode * right;
};
```

```
void PostOrder(const BNode * root)
{
    if(root == nullptr)
    {
        return;
    }
    PostOrder(root->left);
    PostOrder(root->right);
    cout<< root->key << " ";
}
```

后序遍历迭代解法

观察一个节点只有第三次被访问到才能输出
使用两个栈，迭代过程大大简化

1. 创建一个空栈stk，并把根节点压入栈；
创建另一个空栈output
2. 如果栈stk不空，就一直循环：
 - 从栈stk里Pop一个元素top，并压入栈output
 - 把top的左子节点压入栈stk
 - 把top的右子节点压入栈stk
3. 从output栈顶往后输出结果

```
void PostOrder_Iterative(const BNode * root)
{
    if(root == nullptr)
    {
        return;
    }

    stack<const BNode *> output;
    stack<const BNode *> stk;
    stk.push(root);

    while (!stk.empty())
    {
        // Pop top element from stk;
        const BNode * top = stk.top();
        stk.pop();

        // Push top element to output
        output.push(top);

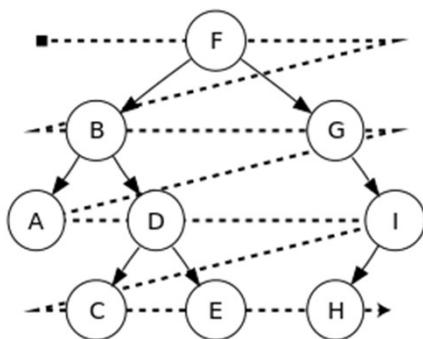
        // push left and right node to stk;
        if (top->left != nullptr)
        {
            stk.push(top->left);
        }

        if (top->right != nullptr)
        {
            stk.push(top->right);
        }
    }

    // print result in output stack
    while (!output.empty())
    {
        cout << output.top()->key << " ";
        output.pop();
    }
}
```

按层遍历

- 逐层，从左到右
- 用队列实现，类似用queue实现stack



- 按层遍历输出：F BG ADI CEH

```
struct BNode
{
    int key;
    BNode * left;
    BNode * right;
};

void LevelOrder(const BNode * root)
{
    queue<const BNode*> treeQueue;

    if (root != nullptr)
    {
        treeQueue.push(root);
    }

    while (!treeQueue.empty())
    {
        const BNode * node = treeQueue.front();
        treeQueue.pop();
        cout << node->key << " ";

        if (node->left != nullptr)
        {
            treeQueue.push(node->left);
        }
        if (node->right != nullptr)
        {
            treeQueue.push(node->right);
        }
    }
}
```

不同二叉树的个数

- $T(n + 1) = \sum_{i=0}^n T(i)T(n - i)$
- 这和Catalan number一样，即

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^n \frac{n+k}{k} \quad \text{for } n \geq 0.$$

- 参考： https://en.wikipedia.org/wiki/Catalan_number

线索二叉树

Threaded Binary Tree

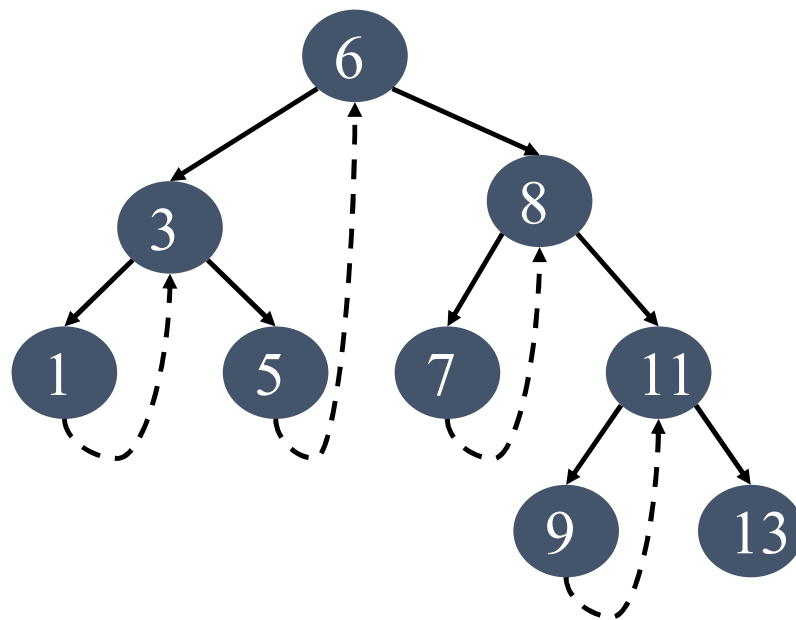
Motivation

- 如果我们不想递归，也不想用stack，有没有办法中序遍历？
- 一个是因为占空间，还有就是如果你在遍历树的时候，插入/删除发生了，就会出错
- Donald Kunth在1968年提出的问题
- Morris在1979年提出了一个方法：Threaded binary tree

思路

- 二叉树的叶节点的两个空指针浪费了空间
- 考虑利用他们来帮助inorder traversals
- 把那些节点，本来其右子节点是空的，现在改为指向中序遍历时的下一个节点
- 为了方便起见，我们在节点里设了一个bool，表明不是正常子节点

线索二叉树例子



Morris遍历算法，非递归，不用stack

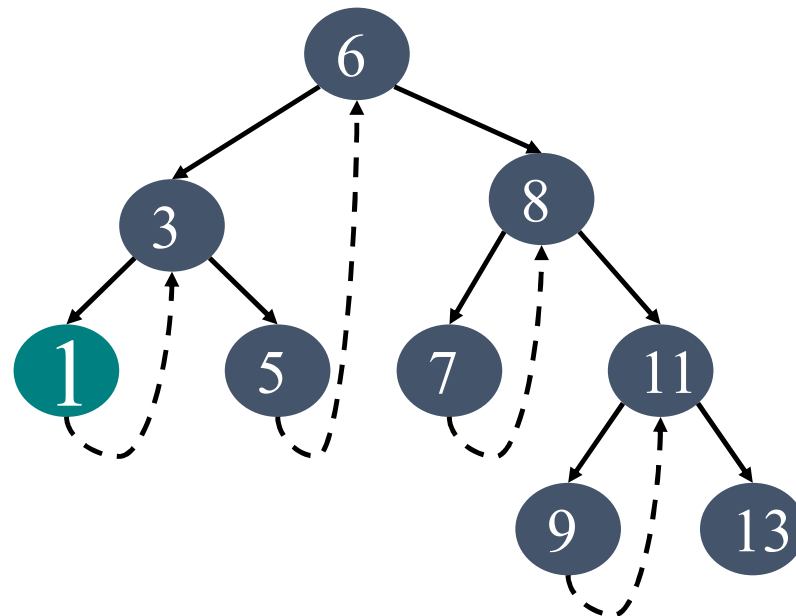
从根节点的最左边leftmost的元素开始，输出后，跟踪它的右子节点

1) 如果跟踪的链接是thread，输出当前结点后，继续跟踪到它的右子节点

2) 如果跟踪的链接不是thread，是正常的链接，那么跟踪到它的最左边leftmost的元素，输出后，对其右子结点的链接按循环1) 2) 处理

其实和之前中序遍历用stack思想一样

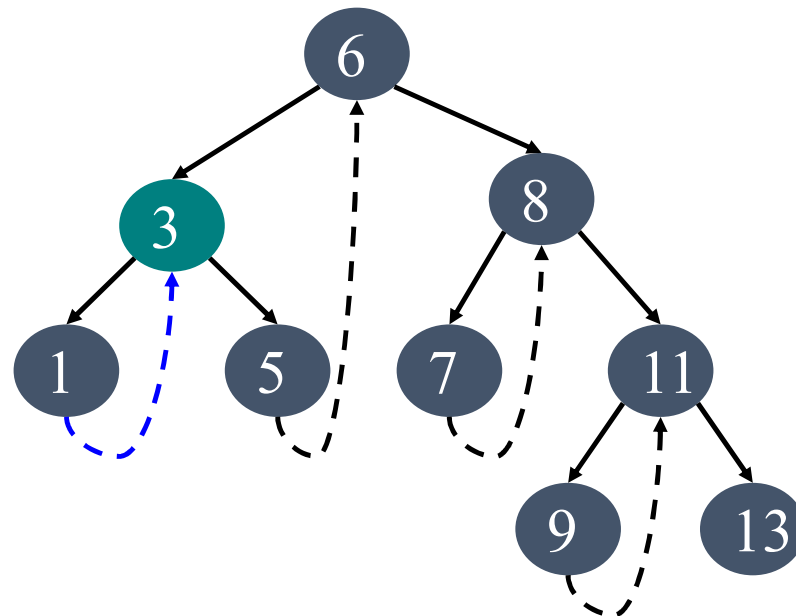
Threaded Tree Traversal



Output
1

找到根节点的leftmost节点，输出

Threaded Tree Traversal



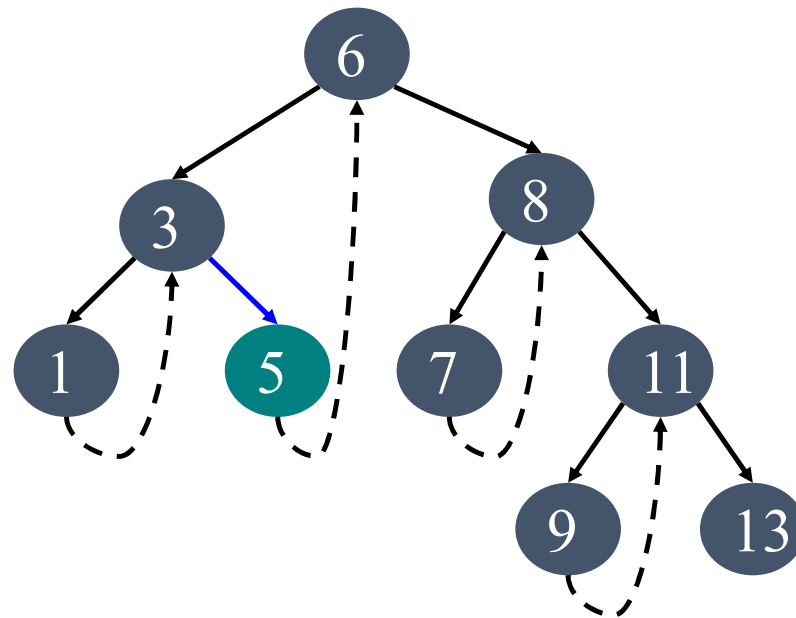
Output

1

3

1) 链接是thread，跟踪到3，输出后跟踪其右子节点

Threaded Tree Traversal

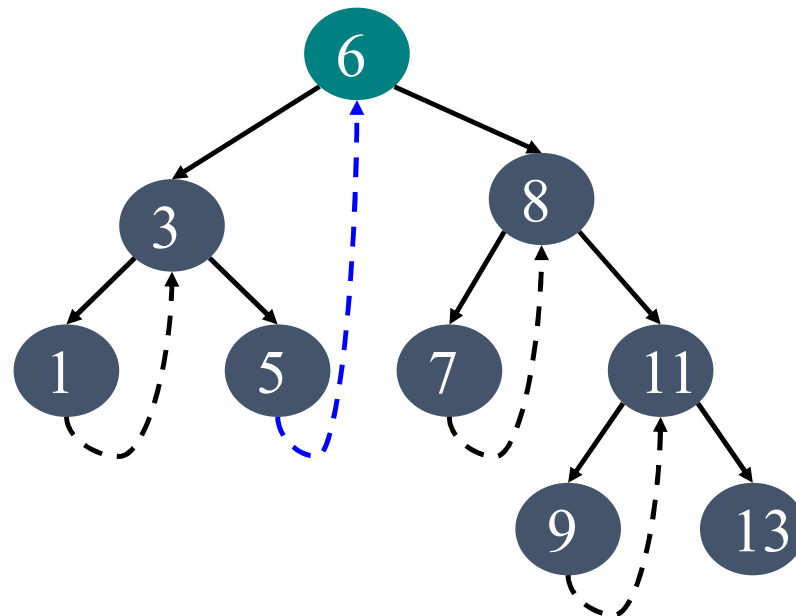


Output

1
3
5

2) 正常链接，跟踪到5的leftmost节点，就是自己。
然后跟踪其右子节点

Threaded Tree Traversal

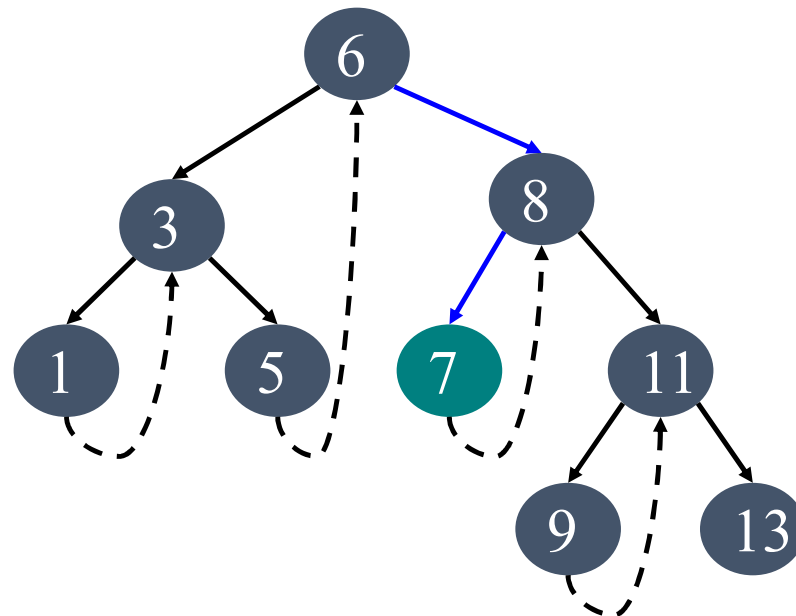


Output

1
3
5
6

1) 链接是thread，跟踪到6，输出后再跟踪其右子节点

Threaded Tree Traversal

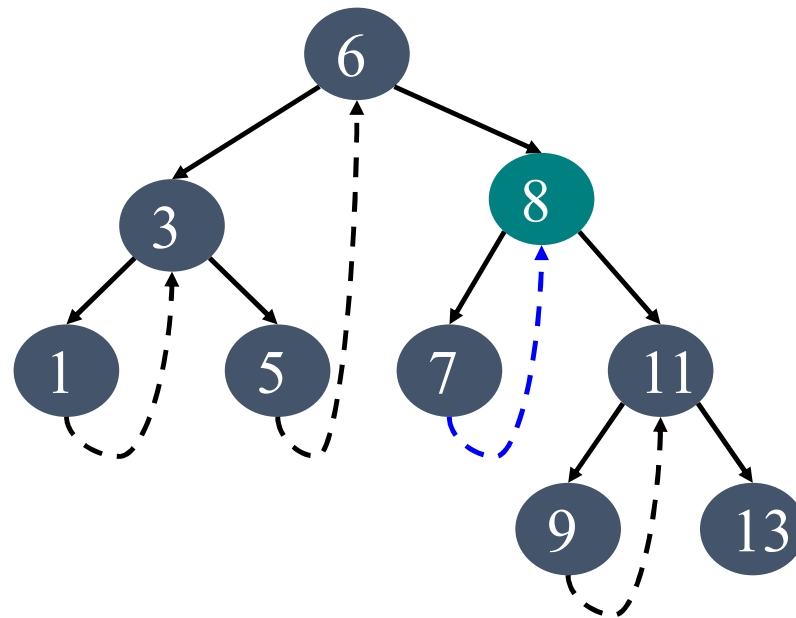


Output

1
3
5
6
7

2) 正常链接，跟踪到其leftmost节点7，输出后，跟踪其右子节点

Threaded Tree Traversal

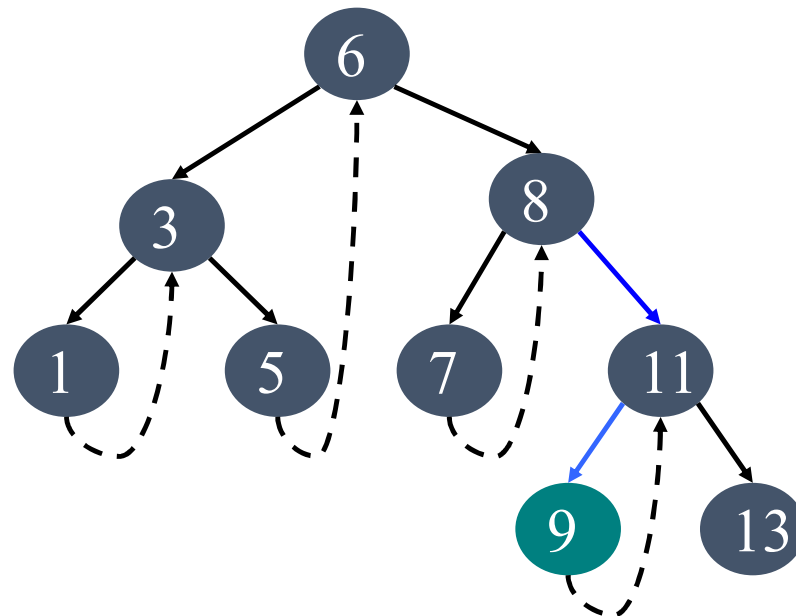


Output

1
3
5
6
7
8

1) 链接是thread，跟踪到8，输出后再跟踪其右子节点

Threaded Tree Traversal

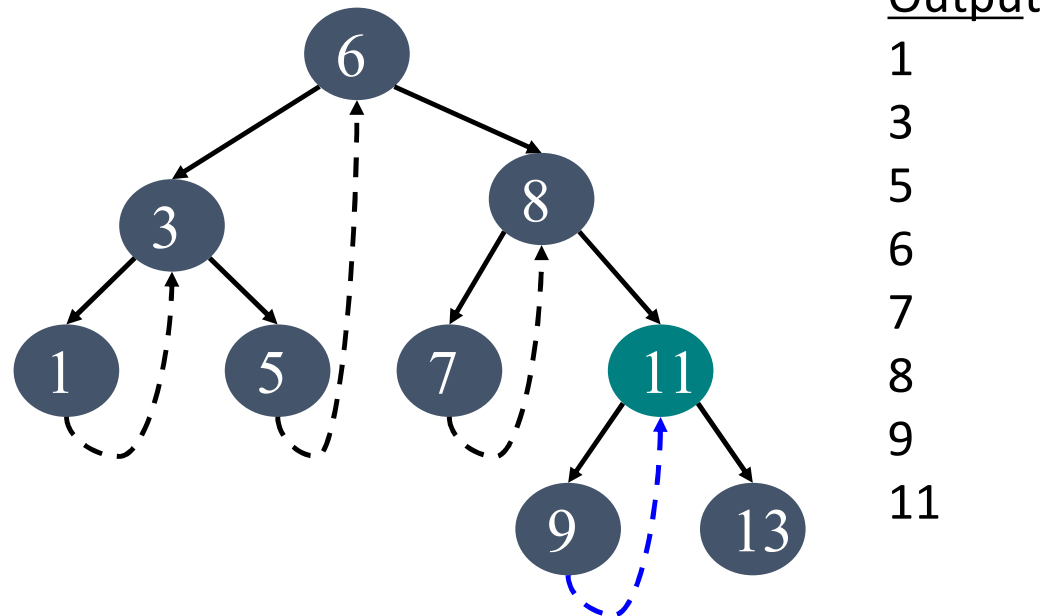


Output

1
3
5
6
7
8
9

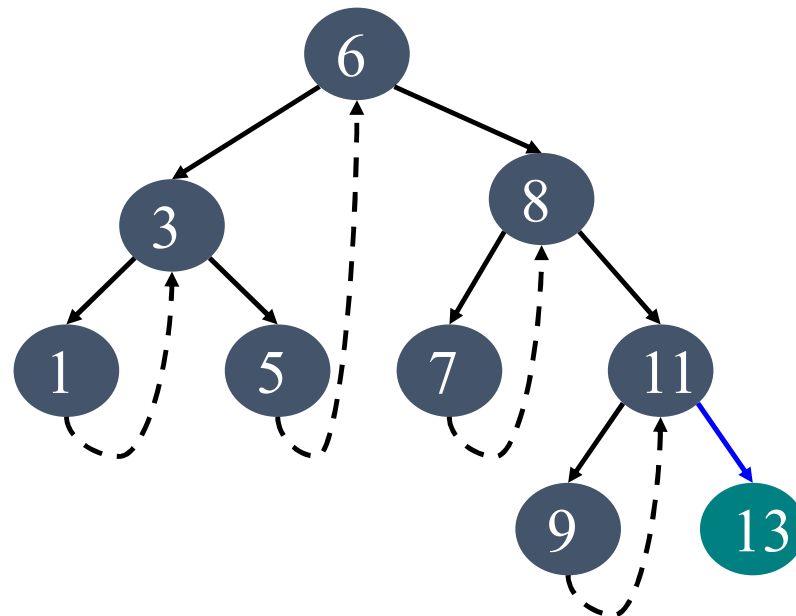
2) 正常链接，跟踪到其leftmost节点9，输出后，跟踪其右子节点

Threaded Tree Traversal



1) 链接是thread，跟踪到11，输出后再跟踪其右子节点

Threaded Tree Traversal

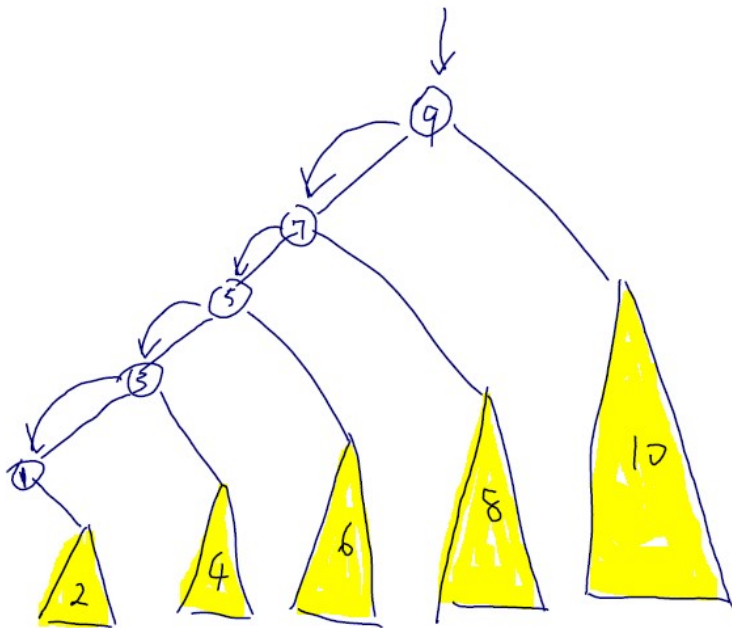


Output

1
3
5
6
7
8
9
11
13

2) 正常链接，跟踪到13的leftmost节点，就是自己。然后跟踪其右子节点，是空，表明结束。

中序遍历代码



```
// Utility function to find leftmost
// node in a tree rooted with root
Node* Leftmost(Node * root)
```

```
{
    while (root && root->left)
    {
        root = root->left;
    }
    return root;
}
```

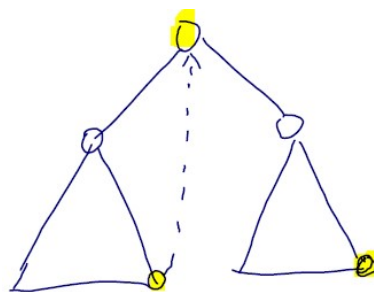
```
void Inorder(Node * root)
```

```
{
    Node * curr = Leftmost(root);
    while (curr)
    {
        cout << curr->key << " ";

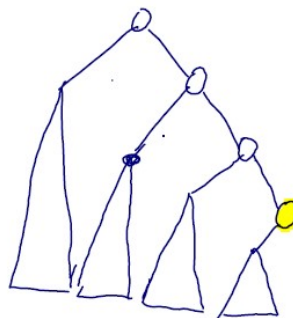
        // If this node is a thread node, then go to
        // inorder successor
        if (curr->rightThread)
        {
            curr = curr->right;
        }
        // Else go to the leftmost child in right subtree
        else
        {
            curr = Leftmost(curr->right);
        }
    }
}
```

把正常的二叉树线索化

1. 左子树里的rightmost节点返回后需要设置后继
2. 右子树里的rightmost节点同样需要返回



注意：只有左子树中的rightmost节点，它的后继不在这个子树里



```
// Converts tree with given root to threaded binary tree
// This function returns rightmost child of root
Node * ConvertToThreaded(Node *root)
{
    // Base cases : Tree is empty or has single node
    if (root == nullptr)
    {
        return nullptr;
    }

    // Find predecessor if it exists
    if (root->left)
    {
        // Find predecessor of root (Rightmost child in left subtree)
        Node * rightmost = ConvertToThreaded(root->left);

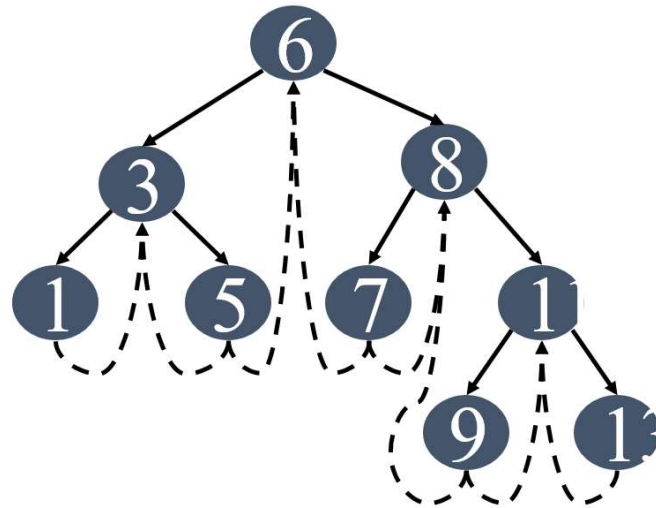
        // Link a thread from predecessor to root
        rightmost->right = root;
        rightmost->rightThread = true;
    }

    // Recursion for right subtree.
    if (root->right)
    {
        return ConvertToThreaded(root->right);
    }

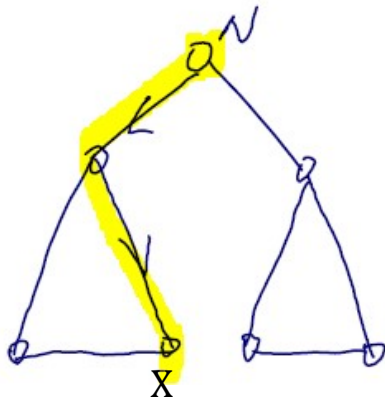
    // If current node is rightmost child
    return root;
}
```

线索二叉树改进

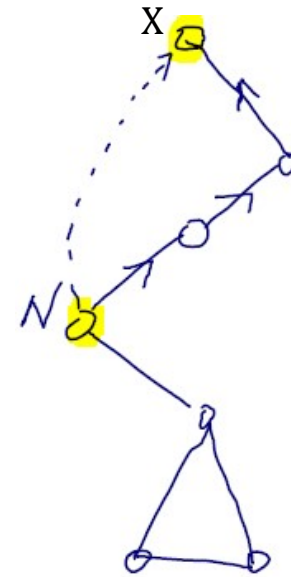
- 有些节点的左指针还是没利用上
- 考虑让它指向中序遍历时的直接前驱Predecessor



找节点N的前驱Predecessor

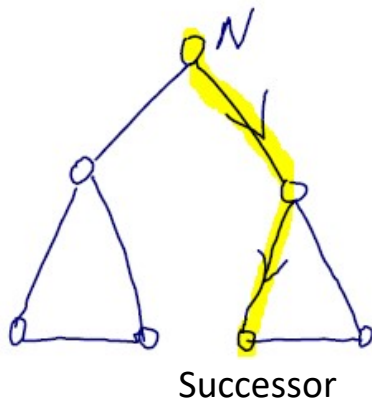


情况1：节点N的左子节点不为空，那么直接找左子树的rightmost节点

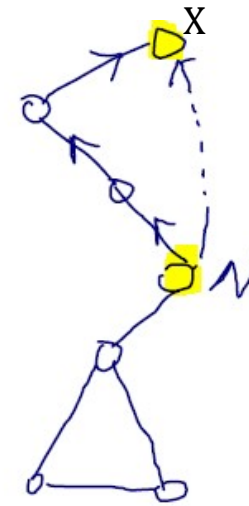


情况2：节点N的左子节点为空，那么找thread predecessor。
其实等同于从节点N开始找第一个父节点X，满足N在X的右子树上即可。

找节点N的后继Successor



情况1：节点 N 的右子节点不为空，那么直接找左子树的leftmost节点



情况2：节点 N 的右子节点为空，那么找thread successor。
其实等同于从节点 N 开始找第一个父节点 X ，满足 N 在 X 的左子树上即可。

Threaded binary search tree

- 线索化的二叉查找树
- 之后讲到二叉查找树后再回来看

Q&A

Thanks!