二叉树遍历（前序、中序、后序、层次、深度优先、广度优先遍历）

二叉树是一种非常重要的**数据结构**，非常多其他数据结构都是基于二叉树的基础演变而来的。对于二叉树，有深度遍历和广度遍历，深度遍历有前序、中序以及后序三种遍历方法，广度遍历即我们寻常所说的层次遍历。由于树的定义本身就是递归定义，因此采用递归的方法去实现树的三种遍历不仅easy理解并且代码非常简洁，而对于广度遍历来说，须要其他数据结构的支撑。比方堆了。所以。对于一段代码来说，可读性有时候要比代码本身的效率要重要的多。

四种基本的遍历思想为：

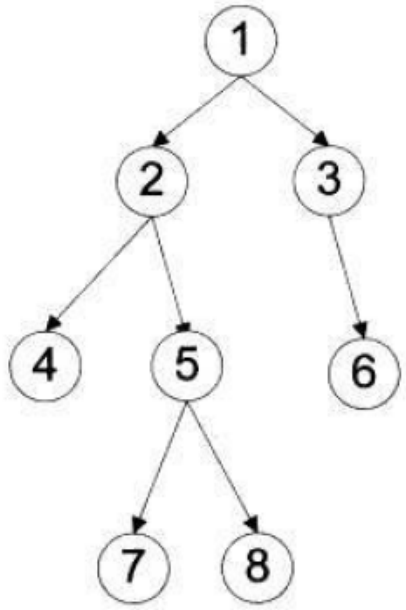
前序遍历：根结点 ---> 左子树 ---> 右子树

中序遍历：左子树---> 根结点 ---> 右子树

后序遍历：左子树 ---> 右子树 ---> 根结点

层次遍历：仅仅需按层次遍历就可以

比如。求以下二叉树的各种遍历



前序遍历：1 2 4 5 7 8 3 6

中序遍历：4 2 7 5 8 1 3 6

后序遍历：4 7 8 5 2 6 3 1

层次遍历：1 2 3 4 5 6 7 8

一、前序遍历

1）依据上文提到的遍历思路：根结点 ---> 左子树 ---> 右子树，非常easy写出递归版本号：

```java
public void preOrderTraverse1(TreeNode root) {
        if (root != null) {
            System.out.print(root.val+"  ");
            preOrderTraverse1(root.left);
            preOrderTraverse1(root.right);
        }
    }
```

2）如今讨论非递归的版本号：

依据前序遍历的顺序，优先訪问根结点。然后在訪问左子树和右子树。所以。对于随意结点node。第一部分即直接訪问之，之后在推断左子树是否为空，不为空时即反复上面的步骤，直到其为空。若为空。则须要訪问右子树。注意。在訪问过左孩子之后。须要反过来訪问其右孩子。所以，须要栈这样的数据结构的支持。对于随意一个结点node，详细过程例如以下：

a)訪问之，并把结点node入栈。当前结点置为左孩子；

b)推断结点node是否为空，若为空。则取出栈顶结点并出栈，将右孩子置为当前结点；否则反复a)步直到当前结点为空或者栈为空（能够发现栈中的结点就是为了訪问右孩子才存储的）

代码例如以下：

```java
public void preOrderTraverse2(TreeNode root) {
        LinkedList<TreeNode> stack = new LinkedList<>();
        TreeNode pNode = root;
        while (pNode != null || !stack.isEmpty()) {
            if (pNode != null) {
                System.out.print(pNode.val+"  ");
                stack.push(pNode);
                pNode = pNode.left;
            } else { //pNode == null && !stack.isEmpty()
                TreeNode node = stack.pop();
                pNode = node.right;
            }
        }
    }
```

二、中序遍历

1)依据上文提到的遍历思路：左子树 ---> 根结点 ---> 右子树，非常easy写出递归版本号：

```java
public void inOrderTraverse1(TreeNode root) {
        if (root != null) {
            inOrderTraverse1(root.left);
            System.out.print(root.val+"  ");
            inOrderTraverse1(root.right);
        }
```

```
    }
```

2）非递归实现，有了上面前序的解释，中序也就比较简单了。同样的道理。仅仅只是訪问的顺序移到出栈时。代码例如以下：

```
[java]
public void inOrderTraverse2(TreeNode root) {
        LinkedList<TreeNode> stack = new LinkedList<>();
        TreeNode pNode = root;
        while (pNode != null || !stack.isEmpty()) {
            if (pNode != null) {
                stack.push(pNode);
                pNode = pNode.left;
            } else { //pNode == null && !stack.isEmpty()
                TreeNode node = stack.pop();
                System.out.print(node.val+"  ");
                pNode = node.right;
            }
        }
    }
```

三、后序遍历

1）依据上文提到的遍历思路：左子树 ---> 右子树 ---> 根结点。非常easy写出递归版本号：

```
[java]
public void postOrderTraverse1(TreeNode root) {
        if (root != null) {
            postOrderTraverse1(root.left);
            postOrderTraverse1(root.right);
            System.out.print(root.val+"  ");
        }
    }
```

2）

后序遍历的非递归实现是三种遍历方式中最难的一种。由于在后序遍历中，要保证左孩子和右孩子都已被訪问而且左孩子在右孩子前訪问才干訪问根结点，这就为流程的控制带来了难题。以下介绍两种思路。

第一种思路：对于任一结点P，将其入栈，然后沿其左子树一直往下搜索。直到搜索到没有左孩子的结点，此时该结点出如今栈顶，可是此时不能将其出栈并訪问，因此其右孩子还为被訪问。

所以接下来依照同样的规则对其右子树进行同样的处理，当訪问完其右孩子时。该结点又出如今栈顶，此时能够将其出栈并訪问。这样就保证了正确的訪问顺序。能够看出，在这个过程中，每一个结点都两次出如今栈顶，仅仅有在第二次出如今栈顶时，才干訪问它。因此须要多设置一个变量标识该结点是否是第一次出如今栈顶。

```
void postOrder2(BinTree *root)     //非递归后序遍历
{
    stack<BTNode*> s;
    BinTree *p=root;
    BTNode *temp;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)                //沿左子树一直往下搜索。直至出现没有左子树的结点
        {
            BTNode *btn=(BTNode *)malloc(sizeof(BTNode));
            btn->btnode=p;
            btn->isFirst=true;
            s.push(btn);
            p=p->lchild;
        }
        if(!s.empty())
        {
            temp=s.top();
            s.pop();
            if(temp->isFirst==true)      //表示是第一次出如今栈顶
             {
                temp->isFirst=false;
                s.push(temp);
                p=temp->btnode->rchild;
            }
            else                         //第二次出如今栈顶
             {
                cout<<temp->btnode->data<<" ";
                p=NULL;
            }
        }
    }
}
```

另外一种思路：要保证根结点在左孩子和右孩子訪问之后才干訪问，因此对于任一结点P。先将其入栈。假设P不存在左孩子和右孩子。则能够直接訪问它；或者P存在左孩子或者右孩子。可是其左孩子和右孩子都已被訪问过了。则相同能够直接訪问该结点。若非上述两种情况。则将P的右孩子和左孩子依次入栈。这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被訪问。左孩子和右孩子都在根结点前面被訪问。

```
void postOrder3(BinTree *root)      //非递归后序遍历
{
    stack<BinTree*> s;
    BinTree *cur;                       //当前结点
    BinTree *pre=NULL;                  //前一次訪问的结点
    s.push(root);
    while(!s.empty())
    {
```

```
        cur=s.top();
        if((cur->lchild==NULL&&cur->rchild==NULL)||
           (pre!=NULL&&(pre==cur->lchild||pre==cur->rchild)))
        {
            cout<<cur->data<<" ";  //假设当前结点没有孩子结点或者孩子节点都已被訪问过
              s.pop();
            pre=cur;
        }
        else
        {
            if(cur->rchild!=NULL)
                s.push(cur->rchild);
            if(cur->lchild!=NULL)
                s.push(cur->lchild);
        }
    }
}
```

四、层次遍历

层次遍历的代码比較简单。仅仅须要一个队列就可以。先在队列中增加根结点。之后对于随意一个结点来说。在其出队列的时候，訪问之。同一时候假设左孩子和右孩子有不为空的。入队列。代码例如以下：

```
[java]
01. public void levelTraverse(TreeNode root) {
02.         if (root == null) {
03.             return;
04.         }
05.         LinkedList<TreeNode> queue = new LinkedList<>();
06.         queue.offer(root);
07.         while (!queue.isEmpty()) {
08.             TreeNode node = queue.poll();
09.             System.out.print(node.val+"  ");
10.             if (node.left != null) {
11.                 queue.offer(node.left);
12.             }
13.             if (node.right != null) {
14.                 queue.offer(node.right);
15.             }
16.         }
17.     }
```

五、深度优先遍历

事实上深度遍历就是上面的前序、中序和后序。可是为了保证与广度优先遍历相照顾，也写在这。代码也比較好理解，事实上就是前序遍历，代码例如以下：

```
[java]
```

```
01. public void depthOrderTraverse(TreeNode root) {
02.         if (root == null) {
03.             return;
04.         }
05.         LinkedList<TreeNode> stack = new LinkedList<>();
06.         stack.push(root);
07.         while (!stack.isEmpty()) {
08.             TreeNode node = stack.pop();
09.             System.out.print(node.val+"  ");
10.             if (node.right != null) {
11.                 stack.push(node.right);
12.             }
13.             if (node.left != null) {
14.                 stack.push(node.left);
15.             }
16.         }
17.     }
```