C-二叉树的遍历（基本）

1.中序遍历

a.利用递归完成的中序遍历

递归的本质： 子问题与母问题的高度重复性和相似性
比如二叉树，访问头节点时接下来的访问顺序反复重复
同时，递归还具有隐形维护的功能。
由于子问题的分解，同时划分集合时递归序列的有序性，可以将这个有序的递归的数据集合视为一个特定的数据结构

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
/**
 * Note: The returned array must be malloced, assume caller calls free(). */
void inorder(struct TreeNode* root,int *res,int *returnSize)
{
    if(root==NULL)
    {
        return ;
    }
    inorder(root->left,res,returnSize);
    res[(*returnSize)++] = root->val;
    inorder(root->right,res,returnSize);
}
int* inorderTraversal(struct TreeNode* root, int* returnSize)
{
    int *res=malloc(sizeof(int)*501);
    *returnSize = 0;
    inorder(root,res,returnSize);
    return res;
}
```

二叉树的中序遍历隐形维护了一个栈，其中头节点先入，左子树和右子树分别入栈，然后再判断。

b.利用迭代完成的中序遍历

迭代的本质： 将函数的输出作为新的输入得到结果

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
/**
 * Note: The returned array must be malloced, assume caller
calls free().
 */
int* inorderTraversal(struct TreeNode* root, int* returnSize)
{

    *returnSize = 0;
    int top = 0;
    struct TreeNode *current_node =  root;
    int *res = malloc(sizeof(int)*501);
    struct TreeNode **task = malloc(sizeof(struct TreeNode*)*
501);
    while(current_node!=NULL||top>0)
    {
            while(current_node!=NULL)
        {
            task[top++] = current_node;
            current_node = current_node->left;
        }
        current_node = task[--top];
        res[(*returnSize)++] = current_node->val;
        current_node = current_node->right;
    }
    return res;
}
```

复杂度分析

时间复杂度：O(n)，其中 n 为二叉树节点的个数。二叉树的遍历中每个节点会被访问一次且只会被访问一次。

空间复杂度：O(n)。空间复杂度取决于递归的栈深度，而栈深度在二叉树为一条链的情况下会达到 O(n) 的级别。

Morris 中序遍历
采用类似于二叉链树的做法，减少了需要维护的栈空间，使空间复杂度带降到O(1)
思路：

思路与算法
Morris 遍历算法是另一种遍历二叉树的方法，它能将非递归的中序遍历空间复杂度降为 O(1)。
Morris 遍历算法整体步骤如下（假设当前遍历到的节点为 x）：
如果 x 无左孩子，先将 x 的值加入答案数组，再访问 x 的右孩子，即 x=x.right。
如果 x 有左孩子，则找到 x 左子树上最右的节点（即左子树中序遍历的最后一个节点，x 在中序遍历中的前驱节点），我们记为 predecessor。根据 predecessor 的右孩子是否为空，进行如下操作。
如果 predecessor 的右孩子为空，则将其右孩子指向 x，然后访问 x 的左孩子，即 x=x.left。
如果 predecessor 的右孩子不为空，则此时其右孩子指向 x，说明我们已经遍历完 x 的左子树，我们将 predecessor 的右孩子置空，将 x 的值加入答案数组，然后访问 x 的右孩子，即 x=x.right。
重复上述操作，直至访问完整棵树。

代码实现：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* inorderTraversal(struct TreeNode* root, int* returnSize)
{
    struct TreeNode * current_node = root;
    struct TreeNode * predecessor = NULL;
    int * res = malloc(sizeof(int)*501);
    *returnSize = 0;
    while(current_node!=NULL)
    {
        if(current_node->left!=NULL)
        {
            predecessor = current_node->left;
            while (predecessor->right!=NULL && predecessor->right!=current_node)
            {
                predecessor = predecessor->right;
            }
            if(predecessor->right==NULL)
            {
                predecessor->right = current_node;
                current_node = current_node->left;
            }else
            {
                predecessor->right = NULL;
                res[(*returnSize)++] = current_node->val;
                current_node = current_node->right;
            }
        }
        else
        {
            res[(*returnSize)++] = current_node->val;
            current_node = current_node->right;
        }
    }
    return res;

}
```