

## 一、二叉树

结构体定义：

```
struct SBinaryTreeNode
{
    int m_Val;
    SBinaryTreeNode* m_pLeftChild;
    SBinaryTreeNode* m_pRightChild;

    SBinaryTreeNode() : m_Val(0), m_pLeftChild(nullptr),
        m_pRightChild(nullptr) {}
};
```

### 1. 二叉树中节点个数

```
//*****
//FUNCTION:: 二叉树节点个数
unsigned int getTreeNodeNum(SBinaryTreeNode* vRootNode)
{
    if (vRootNode == nullptr) return 0;

    unsigned int uiLeftNum = getTreeNodeNum(vRootNode->m_pLeftChild);
    unsigned int uiRightNum = getTreeNodeNum(vRootNode->m_pRightChild);

    return uiLeftNum + uiRightNum + 1;
}
```

### 2. 二叉树深度

```
//*****
//FUNCTION:: 二叉树深度
unsigned int getTreeDepth(SBinaryTreeNode* vRootNode)
{
    if (vRootNode == nullptr) return 0;

    unsigned int uiLeftDepth = getTreeDepth(vRootNode->m_pLeftChild);
    unsigned int uiRightDepth = getTreeDepth(vRootNode->m_pRightChild);

    return uiLeftDepth > uiRightDepth ? (uiLeftDepth + 1) : (uiRightDepth + 1);
}
```

### 3. 二叉树前序遍历, 中序遍历, 后续遍历

```
//*****
//FUNCTION:: 前序遍历
void PreorderTraverse(SBinaryTreeNode* vRootNode)
{
    if (vRootNode) return;

    std::cout << vRootNode->m_Val << std::endl;

    PreorderTraverse(vRootNode->m_pLeftChild);
    PreorderTraverse(vRootNode->m_pRightChild);
}
```

```

//*****
//FUNCTION:: 中序遍历
void InorderTraverse(SBinaryTreeNode* vRootNode)
{
    if (vRootNode) return;

    InorderTraverse(vRootNode->m_pLeftChild);
    std::cout << vRootNode->m_Val << std::endl;
    InorderTraverse(vRootNode->m_pRightChild);
}

```

```

//*****
//FUNCTION:: 后序遍历
void PostorderTraverse(SBinaryTreeNode* vRootNode)
{
    if (vRootNode) return;

    PostorderTraverse(vRootNode->m_pLeftChild);
    PostorderTraverse(vRootNode->m_pRightChild);
    std::cout << vRootNode->m_Val << std::endl;
}

```

#### 4. 分层遍历二叉树 (从左到右, 从上到下)

```

//*****
//FUNCTION:: 分层遍历二叉树(从上到下, 从左到右)
void LevelTraverse(SBinaryTreeNode* vRootNode)
{
    if (vRootNode) return;

    std::queue<SBinaryTreeNode*> BinTreeNodeQueue;
    BinTreeNodeQueue.push(vRootNode);

    while (!BinTreeNodeQueue.empty())
    {
        SBinaryTreeNode* pFrontNode = BinTreeNodeQueue.front();
        std::cout << pFrontNode->m_Val << std::endl;
        BinTreeNodeQueue.pop();

        if (pFrontNode->m_pLeftChild)
            BinTreeNodeQueue.push(pFrontNode->m_pLeftChild);
        if (pFrontNode->m_pRightChild)
            BinTreeNodeQueue.push(pFrontNode->m_pRightChild);
    }
}

```

#### 5. 二叉查找树转有序的双向链表

```

//*****
//FUNCTION:: 二叉查找树变为有序的双向链表
SBinaryTreeNode* convert(SBinaryTreeNode* vRootNode)
{
    if (!vRootNode) return;

    SBinaryTreeNode* pLastNodeInLast = nullptr;
    convertNode(vRootNode, &pLastNodeInLast);

    SBinaryTreeNode* pHeadOfList = pLastNodeInLast;
}

```

```

        while (pHeadOfList != nullptr && pHeadOfList->m_pLeftChild != nullptr)
        {
            pHeadOfList = pHeadOfList->m_pLeftChild;
        }

        return pHeadOfList;
    }

void convertNode(SBinaryTreeNode* vNode, SBinaryTreeNode** vLastNodeInLast)
{
    if (!vNode) return;

    SBinaryTreeNode* pCurrentNode = vNode;
    if (pCurrentNode->m_pLeftChild)
        convertNode(pCurrentNode->m_pLeftChild, vLastNodeInLast);

    pCurrentNode->m_pLeftChild = *vLastNodeInLast;
    if (*vLastNodeInLast != nullptr)
        (*vLastNodeInLast)->m_pRightChild = pCurrentNode;

    *vLastNodeInLast = pCurrentNode;
    if (pCurrentNode->m_pRightChild)
        convertNode(pCurrentNode->m_pRightChild, vLastNodeInLast);
}

```

#### 6. 二叉树第 K 层节点个数

```

//*****
//FUNCTION:: 二叉树第K层节点个数
unsigned int getNodeNumKthLevel(SBinaryTreeNode* vRootNode, int vK)
{
    if (vRootNode == nullptr || vK <= 0)
        return 0;
    if (vK == 1) return 1;

    unsigned int uiLeftNum = getNodeNumKthLevel(vRootNode->m_pLeftChild, vK -
1);
    unsigned int uiRightNum = getNodeNumKthLevel(vRootNode->m_pRightChild, vK -
1);

    return uiLeftNum + uiRightNum;
}

```

#### 7. 二叉树叶子节点个数

```

//*****
//FUNCTION:: 二叉树中叶子节点个数
unsigned int getLeafNodeNum(SBinaryTreeNode* vRootNode)
{
    if (!vRootNode) return 0;
    if (vRootNode->m_pLeftChild == nullptr && vRootNode->m_pRightChild ==
nullptr)
        return 1;

    unsigned int uiLeftNum = getLeafNodeNum(vRootNode->m_pLeftChild);
    unsigned int uiRightNum = getLeafNodeNum(vRootNode->m_pRightChild);

    return uiLeftNum + uiRightNum;
}

```

## 8. 判断两棵树是否相同

```

//*****
//FUNCTION::判断两棵树是否结构体相同
bool isSameTree(SBinaryTreeNode* vRootNode1, SBinaryTreeNode* vRootNode2)
{
    if (!vRootNode1 && !vRootNode2)
        return true;
    else if (!vRootNode1 || !vRootNode2)
        return false;

    bool bResultLeft = isSameTree(vRootNode1->m_pLeftChild,
vRootNode2->m_pLeftChild);
    bool bResultRight = isSameTree(vRootNode1->m_pRightChild,
vRootNode2->m_pRightChild);

    return bResultLeft & bResultRight;
}

```

## 9. 判断树是否是平衡二叉树

```

//*****
//FUNCTION:: 判断树是否是平衡二叉树
bool isAVLTree(SBinaryTreeNode* vRootNode, int& voHeight)
{
    if (!vRootNode)
    {
        voHeight = 0;
        return true;
    }

    int iLeftHeight = 0;
    bool bResultLeft = isAVLTree(vRootNode->m_pLeftChild, iLeftHeight);
    int iRightHeight = 0;
    bool bResultRight = isAVLTree(vRootNode->m_pRightChild, iRightHeight);

    if (bResultLeft && bResultRight && std::abs(iLeftHeight-iRightHeight) <= 1)
    {
        voHeight = std::max(iLeftHeight, iRightHeight) + 1;
        return true;
    }
    else
    {
        voHeight = std::max(iLeftHeight, iRightHeight) + 1;
        return false;
    }
}

```

## 10. 二叉树镜像

```

//*****
//FUNCTION:: 二叉树镜像
SBinaryTreeNode* mirror(SBinaryTreeNode* vRootNode)
{
    if (vRootNode == nullptr) return nullptr;

    SBinaryTreeNode* pLeftChild = mirror(vRootNode->m_pLeftChild);
    SBinaryTreeNode* pRightChild = mirror(vRootNode->m_pRightChild);
}

```

```

        vRootNode->m_pLeftChild = pRightChild;
        vRootNode->m_pRightChild = pLeftChild;

        return vRootNode;
}

```

## 11. 树中两个节点最低公共父节点

- (1) 如果两个节点分别在根节点的左子树和右子树，则返回根节点
- (2) 如果两个节点都在左子树，则递归处理左子树；如果两个节点都在右子树，则递归处理右子树

```

//*****
//FUNCTION::
bool findNode(SBinaryTreeNode* vRootNode, SBinaryTreeNode* vNode)
{
    if (vRootNode == nullptr || vNode == nullptr)
        return false;
    if (vRootNode == vNode)
        return true;

    bool bFound = findNode(vRootNode->m_pLeftChild, vNode);
    if (!bFound)
    {
        bFound = findNode(vRootNode->m_pRightChild, vNode);
    }

    return bFound;
}

//*****
//FUNCTION:: 二叉树两个节点的最低公共祖先节点
SBinaryTreeNode* getLastCommonParent_1(SBinaryTreeNode* vRootNode,
SBinaryTreeNode* vNode1, SBinaryTreeNode* vNode2)
{
    if (findNode(vRootNode->m_pLeftChild, vNode1))
    {
        if (findNode(vRootNode->m_pRightChild, vNode2))
            return vRootNode;
        else
            getLastCommonParent_1(vRootNode->m_pLeftChild, vNode1,
vNode2);
    }
    else
    {
        if (findNode(vRootNode->m_pLeftChild, vNode2))
            return vRootNode;
        else
            return getLastCommonParent_1(vRootNode->m_pRightChild, vNode1,
vNode2);
    }
}

```

## 2) . 非递归解法

```

//*****
//FUNCTION::
bool getNodePath(SBinaryTreeNode* vRootNode, SBinaryTreeNode* vNode,
std::list<SBinaryTreeNode*>& voPath)
{
    if (vRootNode == vNode)
    {
        voPath.push_back(vNode);
        return true;
    }
    if (vRootNode == nullptr)
        return false;

    bool bFound = getNodePath(vRootNode->m_pLeftChild, vNode, voPath);
    if (!bFound)
    {
        bFound = getNodePath(vRootNode->m_pRightChild, vNode, voPath);
    }

    if (!bFound)
    {
        voPath.pop_back();
    }

    return bFound;
}

//*****
//FUNCTION:: 二叉树两个节点最低公共祖先节点 (非递归解法)
SBinaryTreeNode* getLastCommonParent_2(SBinaryTreeNode* vRootNode,
SBinaryTreeNode* vNode1, SBinaryTreeNode* vNode2)
{
    if (!vRootNode || !vNode1 || !vNode2)
        return nullptr;

    std::list<SBinaryTreeNode*> Path1;
    bool bResult1 = getNodePath(vRootNode, vNode1, Path1);
    std::list<SBinaryTreeNode*> Path2;
    bool bResult2 = getNodePath(vRootNode, vNode2, Path2);

    if (!bResult1 || !bResult2)
        return nullptr;

    SBinaryTreeNode* pLastParent = nullptr;
    for (auto Itr1=Path1.begin(), Itr2=Path2.begin(); (Itr1!=Path1.end()) &&
Itr2 != Path2.end(); Itr1++, Itr2++)
    {
        if (*Itr1 == *Itr2)
        {
            pLastParent = *Itr1;
        }
        else
            break;
    }

    return pLastParent;
}

```

## 12. 二叉树最大节点问题

递归解法：

(1) 如果二叉树为空，返回 0，同时记录左子树和右子树的深度，都为 0

(2) 如果二叉树不为空，最大距离要么是左子树中的最大距离，要么是右子树中的最大距离，要么是左子树节点中到根节点的最大距离+右子树节点中到根节点的最大距离，同时记录左子树和右子树节点中到根节点的最大距离。

```

//*****
//FUNCTION::二叉树节点最大距离
int getMaxDistance(SBinaryTreeNode* vRootNode, int& voMaxLeft, int& voMaxRight)
{
    // voMaxLeft, 左子树中的节点距离根节点的最远距离
    // voMaxRight, 右子树中的节点距离根节点的最远距离

    if (vRootNode == nullptr)
    {
        voMaxLeft = voMaxRight = 0;
        return 0;
    }

    int maxLL, maxLR, maxRL, maxRR;
    int maxDistLeft = 0, maxDistRight = 0;
    if (vRootNode->m_pLeftChild)
    {
        maxDistLeft = getMaxDistance(vRootNode->m_pLeftChild, maxLL, maxLR);
        voMaxLeft = std::max(maxLL, maxLR) + 1;
    }
    else
    {
        maxDistLeft = 0;
        voMaxLeft = 0;
    }

    if (vRootNode->m_pRightChild)
    {
        maxDistRight = getMaxDistance(vRootNode->m_pRightChild, maxRL,
maxRR);
        voMaxRight = std::max(maxRL, maxRR) + 1;
    }
    else
    {
        maxDistRight = 0;
        voMaxRight = 0;
    }

    return std::max(std::max(maxDistLeft, maxDistRight), voMaxLeft +
voMaxRight);
}

```

## 13. 前序和中序遍历重建二叉树

- (1) 如果前序遍历为空或中序遍历为空或节点个数小于等于 0，返回 NULL。
- (2) 创建根节点。前序遍历的第一个数据就是根节点的数据，在中序遍历中找到根节点的位置，可分别得知左子树和右子树的前序和中序遍历序列，重建左右子树。

```
//*****
//FUNCTION::前序和中序遍历重建二叉树
SBinaryTreeNode* rebuildTree(int* vPreOrder, int* vInOrder, int vNodeNum)
{
    if (vPreOrder == nullptr || vInOrder == nullptr || vNodeNum <= 0)
        return nullptr;
    SBinaryTreeNode* pRootNode = new SBinaryTreeNode;
    pRootNode->m_Val = vPreOrder[0];
    pRootNode->m_pLeftChild = nullptr;
    pRootNode->m_pRightChild = nullptr;

    int rootPosInOrder = -1;
    for (int i=0; i<vNodeNum; i++)
    {
        if (vInOrder[i] == pRootNode->m_Val)
        {
            rootPosInOrder = i;
            break;
        }
    }

    if (rootPosInOrder == -1)
    {
        std::cout << "Invalid input" << std::endl;
        return nullptr;
    }

    //重建左子树
    int NodeNumLeft = rootPosInOrder;
    int* pPreOrderLeft = vPreOrder + 1;
    int* pInOrderLeft = vInOrder;
    pRootNode->m_pLeftChild = rebuildTree(pPreOrderLeft, pInOrderLeft,
NodeNumLeft);

    //重建右子树
    int NodeNumRight = vNodeNum - NodeNumLeft + 1;
    int* pPreOrderRight = vPreOrder + 1 + NodeNumLeft;
    int* pInOrderRight = vInOrder + NodeNumLeft + 1;
    pRootNode->m_pRightChild = rebuildTree(pPreOrderRight, pInOrderRight,
NodeNumRight);

    return pRootNode;
}
```

#### 14. 判断二叉树是否是完全二叉树

若设二叉树的深度为  $h$ ，除第  $h$  层外，其它各层 ( $1 \sim h-1$ ) 的结点数都达到最大个数，第  $h$  层所有的结点都连续集中在最左边，这就是完全二叉树。



```

//*****
//FUNCTION:: 判断二叉树是否是完全二叉树
bool isCompleteBinaryTree(SBinaryTreeNode* vRootNode)
{
    if (vRootNode == nullptr)
        return false;

    std::queue<SBinaryTreeNode*> NodeQueue;
    NodeQueue.push(vRootNode);

    bool bMustHaveNoChild = false;
    bool bResult = true;
    while (!NodeQueue.empty())
    {
        SBinaryTreeNode* pNode = NodeQueue.front();
        NodeQueue.pop();
        if (bMustHaveNoChild) // 已经出现了有空子树的节点了, 后面出现的必须为叶节点 (左右子树都为空)
        {
            if (pNode->m_pLeftChild || pNode->m_pRightChild)
            {
                bResult = false;
                break;
            }
        }
        else
        {
            if (pNode->m_pLeftChild && pNode->m_pRightChild)
            {
                NodeQueue.push(pNode->m_pLeftChild);
                NodeQueue.push(pNode->m_pRightChild);
            }
            else if (pNode->m_pLeftChild && pNode->m_pRightChild ==
nullptr)
            {
                bMustHaveNoChild = true;
                NodeQueue.push(pNode->m_pLeftChild);
            }
            else if (pNode->m_pLeftChild == nullptr &&
pNode->m_pRightChild)
            {
                bResult = false;
                break;
            }
            else
            {
                bMustHaveNoChild = true;
            }
        }
    }

    return bResult;
}

```