# 2、树

**程序员面试题精选**(01)－把二元查找树转变成排序的双向链表

　　题目：输入一棵二元查找树，将该二元查找树转换成一个排序的双向链表。要求不能创建任何新的结点，只调整指针的指向。

　　比如将二元查找树

10

/ \

6 14

/ \ /　 \

　4 8 12 　 16

转换成双向链表

4=6=8=10=12=14=16。

　　分析：本题是微软的面试题。很多与树相关的题目都是用递归的思路来解决，本题也不例外。下面我们用两种不同的递归思路来分析。

　　思路一：当我们到达某一结点准备调整以该结点为根结点的子树时，先调整其左子树将左子树转换成一个排好序的左子链表，再调整其右子树转换右子链表。最近链接左子链表的最右结点（左子树的最大结点）、当前结点和右子链表的最左结点（右子树的最小结点）。从树的根结点开始递归调整所有结点。

　　思路二：我们可以中序遍历整棵树。按照这个方式遍历树，比较小的结点先访问。如果我们每访问一个结点，假设之前访问过的结点已经调整成一个排序双向链表，我们再把调整当前结点的指针将其链接到链表的末尾。当所有结点都访问过之后，整棵树也就转换成一个排序双向链表了。

参考代码：

首先我们定义二元查找树结点的数据结构如下：

struct BSTreeNode // a node in the binary search tree

{

int m\_nValue; // value of node

BSTreeNode \*m\_pLeft; // left child of node

BSTreeNode \*m\_pRight; // right child of node

};

思路一对应的代码：

///////////////////////////////////////////////////////////////////////

// Covert a sub binary-search-tree into a sorted double-linked list

// Input: pNode - the head of the sub tree

// asRight - whether pNode is the right child of its parent

// Output: if asRight is true, return the least node in the sub-tree

// else return the greatest node in the sub-tree

///////////////////////////////////////////////////////////////////////

BSTreeNode\* ConvertNode(BSTreeNode\* pNode, bool asRight)

{

if(!pNode)

return NULL;

BSTreeNode \*pLeft = NULL;

BSTreeNode \*pRight = NULL;

// Convert the left sub-tree

if(pNode->m\_pLeft)

pLeft = ConvertNode(pNode->m\_pLeft, false);

// Connect the greatest node in the left sub-tree to the current node

if(pLeft)

{

pLeft->m\_pRight = pNode;

pNode->m\_pLeft = pLeft;

}

// Convert the right sub-tree

if(pNode->m\_pRight)

pRight = ConvertNode(pNode->m\_pRight, true);

// Connect the least node in the right sub-tree to the current node

if(pRight)

{

pNode->m\_pRight = pRight;

pRight->m\_pLeft = pNode;

}

BSTreeNode \*pTemp = pNode;

// If the current node is the right child of its parent,

// return the least node in the tree whose root is the current node

if(asRight)

{

while(pTemp->m\_pLeft)

pTemp = pTemp->m\_pLeft;

}

// If the current node is the left child of its parent,

// return the greatest node in the tree whose root is the current node

else

{

while(pTemp->m\_pRight)

pTemp = pTemp->m\_pRight;

}

return pTemp;

}

///////////////////////////////////////////////////////////////////////

// Covert a binary search tree into a sorted double-linked list

// Input: the head of tree

// Output: the head of sorted double-linked list

///////////////////////////////////////////////////////////////////////

BSTreeNode\* Convert(BSTreeNode\* pHeadOfTree)

{

// As we want to return the head of the sorted double-linked list,

// we set the second parameter to be true

return ConvertNode(pHeadOfTree, true);

}

思路二对应的代码：

///////////////////////////////////////////////////////////////////////

// Covert a sub binary-search-tree into a sorted double-linked list

// Input: pNode - the head of the sub tree

// pLastNodeInList - the tail of the double-linked list

///////////////////////////////////////////////////////////////////////

void ConvertNode(BSTreeNode\* pNode, BSTreeNode\*& pLastNodeInList)

{

if(pNode == NULL)

return;

BSTreeNode \*pCurrent = pNode;

// Convert the left sub-tree

if (pCurrent->m\_pLeft != NULL)

ConvertNode(pCurrent->m\_pLeft, pLastNodeInList);

// Put the current node into the double-linked list

pCurrent->m\_pLeft = pLastNodeInList;

if(pLastNodeInList != NULL)

pLastNodeInList->m\_pRight = pCurrent;

pLastNodeInList = pCurrent;

// Convert the right sub-tree

if (pCurrent->m\_pRight != NULL)

ConvertNode(pCurrent->m\_pRight, pLastNodeInList);

}

///////////////////////////////////////////////////////////////////////

// Covert a binary search tree into a sorted double-linked list

// Input: pHeadOfTree - the head of tree

// Output: the head of sorted double-linked list

///////////////////////////////////////////////////////////////////////

BSTreeNode\* Convert\_Solution1(BSTreeNode\* pHeadOfTree)

{

BSTreeNode \*pLastNodeInList = NULL;

ConvertNode(pHeadOfTree, pLastNodeInList);

// Get the head of the double-linked list

BSTreeNode \*pHeadOfList = pLastNodeInList;

while(pHeadOfList && pHeadOfList->m\_pLeft)

pHeadOfList = pHeadOfList->m\_pLeft;

return pHeadOfList;

}

**程序员面试题精选**(04)－在二元树中找出和为某一值的所有路径

题目：输入一个整数和一棵二元树。从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径。打印出和与输入整数相等的所有路径。

例如输入整数22和如下二元树

10

/ \

5 12

/ \

　4 7

则打印出两条路径：10, 12和10, 5, 7。

二元树结点的数据结构定义为：

struct BinaryTreeNode // a node in the binary tree

{

int m\_nValue; // value of node

BinaryTreeNode \*m\_pLeft; // left child of node

BinaryTreeNode \*m\_pRight; // right child of node

};

分析：这是百度的一道笔试题，考查对树这种基本数据结构以及递归函数的理解。

当访问到某一结点时，把该结点添加到路径上，并累加当前结点的值。如果当前结点为叶结点并且当前路径的和刚好等于输入的整数，则当前的路径符合要求，我们把它打印出来。如果当前结点不是叶结点，则继续访问它的子结点。当前结点访问结束后，递归函数将自动回到父结点。因此我们在函数退出之前要在路径上删除当前结点并减去当前结点的值，以确保返回父结点时路径刚好是根结点到父结点的路径。我们不难看出保存路径的数据结构实际上是一个栈结构，因为路径要与递归调用状态一致，而递归调用本质就是一个压栈和出栈的过程。

参考代码：

///////////////////////////////////////////////////////////////////////

// Find paths whose sum equal to expected sum

///////////////////////////////////////////////////////////////////////

void FindPath

(

BinaryTreeNode\* pTreeNode, // a node of binary tree

int expectedSum, // the expected sum

std::vector<int>& path, // a path from root to current node

int& currentSum // the sum of path

)

{

if(!pTreeNode)

return;

currentSum += pTreeNode->m\_nValue;

path.push\_back(pTreeNode->m\_nValue);

// if the node is a leaf, and the sum is same as pre-defined,

// the path is what we want. print the path

bool isLeaf = (!pTreeNode->m\_pLeft && !pTreeNode->m\_pRight);

if(currentSum == expectedSum && isLeaf)

{

std::vector<int>::iterator iter = path.begin();

for(; iter != path.end(); ++ iter)

std::cout << \*iter << '\t';

std::cout << std::endl;

}

// if the node is not a leaf, goto its children

if(pTreeNode->m\_pLeft)

FindPath(pTreeNode->m\_pLeft, expectedSum, path, currentSum);

if(pTreeNode->m\_pRight)

FindPath(pTreeNode->m\_pRight, expectedSum, path, currentSum);

// when we finish visiting a node and return to its parent node,

// we should delete this node from the path and

// minus the node's value from the current sum

currentSum -= pTreeNode->m\_nValue;

path.pop\_back();

}

**程序员面试题精选**(06)－判断整数序列是不是二元查找树的后序遍历结果

题目：输入一个整数数组，判断该数组是不是某二元查找树的后序遍历的结果。如果是返回true，否则返回false。 例如输入5、7、6、9、11、10、8，由于这一整数序列是如下树的后序遍历结果：

8

/ \

6 10

/ \ / \

5 7 9 11

因此返回true。

如果输入7、4、6、5，没有哪棵树的后序遍历的结果是这个序列，因此返回false。

分析：这是一道trilogy的笔试题，主要考查对二元查找树的理解。

在后续遍历得到的序列中，最后一个元素为树的根结点。从头开始扫描这个序列，比根结点小的元素都应该位于序列的左半部分；从第一个大于跟结点开始到跟结点前面的一个元素为止，所有元素都应该大于跟结点，因为这部分元素对应的是树的右子树。根据这样的划分，把序列划分为左右两部分，我们递归地确认序列的左、右两部分是不是都是二元查找树。

参考代码：

using namespace std;

///////////////////////////////////////////////////////////////////////

// Verify whether a squence of integers are the post order traversal

// of a binary search tree (BST)

// Input: squence - the squence of integers

// length - the length of squence

// Return: return ture if the squence is traversal result of a BST,

// otherwise, return false

///////////////////////////////////////////////////////////////////////

bool verifySquenceOfBST(int squence[], int length)

{

if(squence == NULL || length <= 0)

return false;

// root of a BST is at the end of post order traversal squence

int root = squence[length - 1];

// the nodes in left sub-tree are less than the root

int i = 0;

for(; i < length - 1; ++ i)

{

if(squence > root)

break;

}

// the nodes in the right sub-tree are greater than the root

int j = i;

for(; j < length - 1; ++ j)

{

if(squence[j] < root)

return false;

}

// verify whether the left sub-tree is a BST

bool left = true;

if(i > 0)

left = verifySquenceOfBST(squence, i);

// verify whether the right sub-tree is a BST

bool right = true;

if(i < length - 1)

right = verifySquenceOfBST(squence + i, length - i - 1);

return (left && right);

}

**程序员面试题精选**(11)－求二元查找树的镜像

题目：输入一颗二元查找树，将该树转换为它的镜像，即在转换后的二元查找树中，左子树的结点都大于右子树的结点。用递归和循环两种方法完成树的镜像转换。 例如输入：

8

/ \

6 10

/\ /\

5 7 9 11

输出：

8

/ \

10 6

/\ /\

11 9 7 5

定义二元查找树的结点为：

struct BSTreeNode // a node in the binary search tree (BST)

{

int m\_nValue; // value of node

BSTreeNode \*m\_pLeft; // left child of node

BSTreeNode \*m\_pRight; // right child of node

};

分析：尽管我们可能一下子不能理解镜像是什么意思，但上面的例子给我们的直观感觉，就是交换结点的左右子树。我们试着在遍历例子中的二元查找树的同时来交换每个结点的左右子树。遍历时首先访问头结点8，我们交换它的左右子树得到：

8

/ \

10 6

/\ /\

9 11 5 7

我们发现两个结点6和10的左右子树仍然是左结点的值小于右结点的值，我们再试着交换他们的左右子树，得到：

8

/ \

10 6

/\ /\

11 9 7 5

刚好就是要求的输出。

上面的分析印证了我们的直觉：在遍历二元查找树时每访问到一个结点，交换它的左右子树。这种思路用递归不难实现，将遍历二元查找树的代码稍作修改就可以了。参考代码如下：

///////////////////////////////////////////////////////////////////////

// Mirror a BST (swap the left right child of each node) recursively

// the head of BST in initial call

///////////////////////////////////////////////////////////////////////

void MirrorRecursively(BSTreeNode \*pNode)

{

if(!pNode)

return;

// swap the right and left child sub-tree

BSTreeNode \*pTemp = pNode->m\_pLeft;

pNode->m\_pLeft = pNode->m\_pRight;

pNode->m\_pRight = pTemp;

// mirror left child sub-tree if not null

if(pNode->m\_pLeft)

MirrorRecursively(pNode->m\_pLeft);

// mirror right child sub-tree if not null

if(pNode->m\_pRight)

MirrorRecursively(pNode->m\_pRight);

}

由于递归的本质是编译器生成了一个函数调用的栈，因此用循环来完成同样任务时最简单的办法就是用一个辅助栈来模拟递归。首先我们把树的头结点放入栈中。在循环中，只要栈不为空，弹出栈的栈顶结点，交换它的左右子树。如果它有左子树，把它的左子树压入栈中；如果它有右子树，把它的右子树压入栈中。这样在下次循环中就能交换它儿子结点的左右子树了。参考代码如下：

///////////////////////////////////////////////////////////////////////

// Mirror a BST (swap the left right child of each node) Iteratively

// Input: pTreeHead: the head of BST

///////////////////////////////////////////////////////////////////////

void MirrorIteratively(BSTreeNode \*pTreeHead)

{

if(!pTreeHead)

return;

std::stack<BSTreeNode \*> stackTreeNode;

stackTreeNode.push(pTreeHead);

while(stackTreeNode.size())

{

BSTreeNode \*pNode = stackTreeNode.top();

stackTreeNode.pop();

// swap the right and left child sub-tree

BSTreeNode \*pTemp = pNode->m\_pLeft;

pNode->m\_pLeft = pNode->m\_pRight;

pNode->m\_pRight = pTemp;

// push left child sub-tree into stack if not null

if(pNode->m\_pLeft)

stackTreeNode.push(pNode->m\_pLeft);

// push right child sub-tree into stack if not null

if(pNode->m\_pRight)

stackTreeNode.push(pNode->m\_pRight);

}

}

**程序员面试题精选**(12)－从上往下遍历二元树

题目：输入一颗二元树，从上往下按层打印树的每个结点，同一层中按照从左往右的顺序打印。 例如输入

8

/ \

6 10

/\ /\

5 7 9 11

输出8 6 10 5 7 9 11。

分析：这曾是微软的一道面试题。这道题实质上是要求遍历一棵二元树，只不过不是我们熟悉的前序、中序或者后序遍历。

我们从树的根结点开始分析。自然先应该打印根结点8，同时为了下次能够打印8的两个子结点，我们应该在遍历到8时把子结点6和10保存到一个数据容器中。现在数据容器中就有两个元素6 和10了。按照从左往右的要求，我们先取出6访问。打印6的同时要把6的两个子结点5和7放入数据容器中，此时数据容器中有三个元素10、5和7。接下来我们应该从数据容器中取出结点10访问了。注意10比5和7先放入容器，此时又比5和7先取出，就是我们通常说的先入先出。因此不难看出这个数据容器的类型应该是个队列。

既然已经确定数据容器是一个队列，现在的问题变成怎么实现队列了。实际上我们无需自己动手实现一个，因为STL已经为我们实现了一个很好的deque（两端都可以进出的队列），我们只需要拿过来用就可以了。

我们知道树是图的一种特殊退化形式。同时如果对图的深度优先遍历和广度优先遍历有比较深刻的理解，将不难看出这种遍历方式实际上是一种广度优先遍历。因此这道题的本质是在二元树上实现广度优先遍历。

参考代码：

#include <deque>

#include <iostream>

using namespace std;

struct BTreeNode // a node in the binary tree

{

int m\_nValue; // value of node

BTreeNode \*m\_pLeft; // left child of node

BTreeNode \*m\_pRight; // right child of node

};

///////////////////////////////////////////////////////////////////////

// Print a binary tree from top level to bottom level

// Input: pTreeRoot - the root of binary tree

///////////////////////////////////////////////////////////////////////

void PrintFromTopToBottom(BTreeNode \*pTreeRoot)

{

if(!pTreeRoot)

return;

// get a empty queue

deque<BTreeNode \*> dequeTreeNode;

// insert the root at the tail of queue

dequeTreeNode.push\_back(pTreeRoot);

while(dequeTreeNode.size())

{

// get a node from the head of queue

BTreeNode \*pNode = dequeTreeNode.front();

dequeTreeNode.pop\_front();

// print the node

cout << pNode->m\_nValue << ' ';

// print its left child sub-tree if it has

if(pNode->m\_pLeft)

dequeTreeNode.push\_back(pNode->m\_pLeft);

// print its right child sub-tree if it has

if(pNode->m\_pRight)

dequeTreeNode.push\_back(pNode->m\_pRight);

}

}

**程序员面试题精选**100题(27)-二元树的深度

题目：输入一棵二元树的根结点，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

例如：输入二元树：

10

/ \

6 14

/ / \

4 12 16

输出该树的深度3。

二元树的结点定义如下：

struct SBinaryTreeNode // a node of the binary tree

{

int m\_nValue; // value of node

SBinaryTreeNode \*m\_pLeft; // left child of node

SBinaryTreeNode \*m\_pRight; // right child of node

};

分析：这道题本质上还是考查二元树的遍历。

题目给出了一种树的深度的定义。当然，我们可以按照这种定义去得到树的所有路径，也就能得到最长路径以及它的长度。只是这种思路用来写程序有点麻烦。

我们还可以从另外一个角度来理解树的深度。如果一棵树只有一个结点，它的深度为1。如果根结点只有左子树而没有右子树，那么树的深度应该是其左子树的深度加1；同样如果根结点只有右子树而没有左子树，那么树的深度应该是其右子树的深度加1。如果既有右子树又有左子树呢？那该树的深度就是其左、右子树深度的较大值再加1。

上面的这个思路用递归的方法很容易实现，只需要对遍历的代码稍作修改即可。参考代码如下：

///////////////////////////////////////////////////////////////////////

// Get depth of a binary tree

// Input: pTreeNode - the head of a binary tree

// Output: the depth of a binary tree

///////////////////////////////////////////////////////////////////////

int TreeDepth(SBinaryTreeNode \*pTreeNode)

{

// the depth of a empty tree is 0

if(!pTreeNode)

return 0;

// the depth of left sub-tree

int nLeft = TreeDepth(pTreeNode->m\_pLeft);

// the depth of right sub-tree

int nRight = TreeDepth(pTreeNode->m\_pRight);

// depth is the binary tree

return (nLeft > nRight) ? (nLeft + 1) : (nRight + 1);

}

**程序员面试题精选100题(48)-二叉树两结点的最低共同父结点[数据结构]**

题目：二叉树的结点定义如下：

struct TreeNode

{

    int m\_nvalue;

    TreeNode\* m\_pLeft;

    TreeNode\* m\_pRight;

};

输入二叉树中的两个结点，输出这两个结点在数中最低的共同父结点。

分析：求数中两个结点的最低共同结点是面试中经常出现的一个问题。这个问题至少有两个变种。

第一变种是二叉树是一种特殊的二叉树：查找二叉树。也就是树是排序过的，位于左子树上的结点都比父结点小，而位于右子树的结点都比父结点大。我们只需要从根结点开始和两个结点进行比较。如果当前结点的值比两个结点都大，则最低的共同父结点一定在当前结点的左子树中。如果当前结点的值比两个结点都小，则最低的共同父结点一定在当前结点的右子树中。

第二个变种是树不一定是二叉树，每个结点都有一个指针指向它的父结点。于是我们可以从任何一个结点出发，得到一个到达树根结点的单向链表。因此这个问题转换为两个单向链表的第一个公共结点。我们在[本面试题系列的第35题](http://zhedahht.blog.163.com/blog/static/254111742008053169567/)讨论了这个问题。

现在我们回到这个问题本身。所谓共同的父结点，就是两个结点都出现在这个结点的子树中。因此我们可以定义一函数，来判断一个结点的子树中是不是包含了另外一个结点。这不是件很难的事，我们可以用递归的方法来实现：

/////////////////////////////////////////////////////////////////////////////////

// If the tree with head pHead has a node pNode, return true.

// Otherwise return false.

/////////////////////////////////////////////////////////////////////////////////

bool HasNode(TreeNode\* pHead, TreeNode\* pNode)

{

    if(pHead == pNode)

        return true;

    bool has = false;

    if(pHead->m\_pLeft != NULL)

        has = HasNode(pHead->m\_pLeft, pNode);

    if(!has && pHead->m\_pRight != NULL)

        has = HasNode(pHead->m\_pRight, pNode);

    return has;

}

我们可以从根结点开始，判断以当前结点为根的树中左右子树是不是包含我们要找的两个结点。如果两个结点都出现在它的左子树中，那最低的共同父结点也出现在它的左子树中。如果两个结点都出现在它的右子树中，那最低的共同父结点也出现在它的右子树中。如果两个结点一个出现在左子树中，一个出现在右子树中，那当前的结点就是最低的共同父结点。基于这个思路，我们可以写出如下代码：

/////////////////////////////////////////////////////////////////////////////////

// Find the last parent of pNode1 and pNode2 in a tree with head pHead

/////////////////////////////////////////////////////////////////////////////////

TreeNode\* LastCommonParent\_1(TreeNode\* pHead, TreeNode\* pNode1, TreeNode\* pNode2)

{

    if(pHead == NULL || pNode1 == NULL || pNode2 == NULL)

        return NULL;

    // check whether left child has pNode1 and pNode2

    bool leftHasNode1 = false;

    bool leftHasNode2 = false;

    if(pHead->m\_pLeft != NULL)

    {

        leftHasNode1 = HasNode(pHead->m\_pLeft, pNode1);

        leftHasNode2 = HasNode(pHead->m\_pLeft, pNode2);

    }

    if(leftHasNode1 && leftHasNode2)

    {

        if(pHead->m\_pLeft == pNode1 || pHead->m\_pLeft == pNode2)

            return pHead;

        return LastCommonParent\_1(pHead->m\_pLeft, pNode1, pNode2);

    }

    // check whether right child has pNode1 and pNode2

    bool rightHasNode1 = false;

    bool rightHasNode2 = false;

    if(pHead->m\_pRight != NULL)

    {

        if(!leftHasNode1)

            rightHasNode1 = HasNode(pHead->m\_pRight, pNode1);

        if(!leftHasNode2)

            rightHasNode2 = HasNode(pHead->m\_pRight, pNode2);

    }

    if(rightHasNode1 && rightHasNode2)

    {

        if(pHead->m\_pRight == pNode1 || pHead->m\_pRight == pNode2)

            return pHead;

        return LastCommonParent\_1(pHead->m\_pRight, pNode1, pNode2);

    }

    if((leftHasNode1 && rightHasNode2)

        || (leftHasNode2 && rightHasNode1))

        return pHead;

    return NULL;

}

接着我们来分析一下这个方法的效率。函数HasNode的本质就是遍历一棵树，其时间复杂度是O(n)（n是树中结点的数目）。由于我们根结点开始，要对每个结点调用函数HasNode。因此总的时间复杂度是O(n2)。

我们仔细分析上述代码，不难发现我们判断以一个结点为根的树是否含有某个结点时，需要遍历树的每个结点。接下来我们判断左子结点或者右结点为根的树中是否含有要找结点，仍然需要遍历。第二次遍历的操作其实在前面的第一次遍历都做过了。由于存在重复的遍历，本方法在时间效率上肯定不是最好的。

前面我们提过如果结点中有一个指向父结点的指针，我们可以把问题转化为求两个链表的共同结点。现在我们可以想办法得到这个链表。我们在[本面试题系列的第4题](http://zhedahht.blog.163.com/blog/static/254111742007228357325/)中分析过如何得到一条中根结点开始的路径。我们在这里稍作变化即可：

/////////////////////////////////////////////////////////////////////////////////

// Get the path form pHead and pNode in a tree with head pHead

/////////////////////////////////////////////////////////////////////////////////

bool GetNodePath(TreeNode\* pHead, TreeNode\* pNode, std::list<TreeNode\*>& path)

{

    if(pHead == pNode)

        return true;

    path.push\_back(pHead);

    bool found = false;

    if(pHead->m\_pLeft != NULL)

        found = GetNodePath(pHead->m\_pLeft, pNode, path);

    if(!found && pHead->m\_pRight)

        found = GetNodePath(pHead->m\_pRight, pNode, path);

    if(!found)

        path.pop\_back();

    return found;

}

由于这个路径是从跟结点开始的。最低的共同父结点就是路径中的最后一个共同结点：

/////////////////////////////////////////////////////////////////////////////////

// Get the last common Node in two lists: path1 and path2

/////////////////////////////////////////////////////////////////////////////////

TreeNode\* LastCommonNode

(

    const std::list<TreeNode\*>& path1,

    const std::list<TreeNode\*>& path2

)

{

    std::list<TreeNode\*>::const\_iterator iterator1 = path1.begin();

    std::list<TreeNode\*>::const\_iterator iterator2 = path2.begin();

    TreeNode\* pLast = NULL;

    while(iterator1 != path1.end() && iterator2 != path2.end())

    {

        if(\*iterator1 == \*iterator2)

            pLast = \*iterator1;

        iterator1++;

        iterator2++;

    }

    return pLast;

}

有了前面两个子函数之后，求两个结点的最低共同父结点就很容易了。我们先求出从根结点出发到两个结点的两条路径，再求出两条路径的最后一个共同结点。代码如下：

/////////////////////////////////////////////////////////////////////////////////

// Find the last parent of pNode1 and pNode2 in a tree with head pHead

/////////////////////////////////////////////////////////////////////////////////

TreeNode\* LastCommonParent\_2(TreeNode\* pHead, TreeNode\* pNode1, TreeNode\* pNode2)

{

    if(pHead == NULL || pNode1 == NULL || pNode2 == NULL)

        return NULL;

    std::list<TreeNode\*> path1;

    GetNodePath(pHead, pNode1, path1);

    std::list<TreeNode\*> path2;

    GetNodePath(pHead, pNode2, path2);

    return LastCommonNode(path1, path2);

}

这种思路的时间复杂度是O(n)，时间效率要比第一种方法好很多。但同时我们也要注意到，这种思路需要两个链表来保存路径，空间效率比不上第一个方法。

**程序员面试题精选100题(50)-树的子结构[数据结构]**

题目：二叉树的结点定义如下：

struct TreeNode

{

        int m\_nValue;

        TreeNode\* m\_pLeft;

        TreeNode\* m\_pRight;

};

输入两棵二叉树A和B，判断树B是不是A的子结构。

例如，下图中的两棵树A和B，由于A中有一部分子树的结构和B是一样的，因此B就是A的子结构。

                 1                                                   8  
               /    \                                               /    \  
              8    7                                             9    2  
            /    \  
           9    2  
                /  \  
               4  7

分析：这是2010年微软校园招聘时的一道题目。二叉树一直是微软面试题中经常出现的数据结构。对微软有兴趣的读者一定要重点关注二叉树。

                回到这个题目的本身。要查找树A中是否存在和树B结构一样的子树，我们可以分为两步：第一步在树A中找到和B的根结点的值一样的结点N，第二步再判断树A中以N为根结点的子树是不是包括和树B一样的结构。

                第一步在树A中查找与根结点的值一样的结点。这实际上就是树的遍历。对二叉树这种数据结构熟悉的读者自然知道我们可以用递归的方法去遍历，也可以用循环的方法去遍历。由于递归的代码实现比较简洁，面试时如果没有特别要求，我们通常都会采用递归的方式。下面是参考代码：

bool HasSubtree(TreeNode\* pTreeHead1, TreeNode\* pTreeHead2)

{

        if((pTreeHead1 == NULL && pTreeHead2 != NULL) ||

                (pTreeHead1 != NULL && pTreeHead2 == NULL))

                return false;

        if(pTreeHead1 == NULL && pTreeHead2 == NULL)

                return true;

        return HasSubtreeCore(pTreeHead1, pTreeHead2);

}

bool HasSubtreeCore(TreeNode\* pTreeHead1, TreeNode\* pTreeHead2)

{

        bool result = false;

        if(pTreeHead1->m\_nValue == pTreeHead2->m\_nValue)

        {

                result = DoesTree1HaveAllNodesOfTree2(pTreeHead1, pTreeHead2);

        }

        if(!result && pTreeHead1->m\_pLeft != NULL)

                result = HasSubtreeCore(pTreeHead1->m\_pLeft, pTreeHead2);

        if(!result && pTreeHead1->m\_pRight != NULL)

                result = HasSubtreeCore(pTreeHead1->m\_pRight, pTreeHead2);

        return result;

}

在上述代码中，我们递归调用hasSubtreeCore遍历二叉树A。如果发现某一结点的值和树B的头结点的值相同，则调用DoesTree1HaveAllNodeOfTree2，做第二步判断。

在面试的时候，我们一定要注意边界条件的检查，即检查空指针。当树A或树B为空的时候，定义相应的输出。如果没有检查并做相应的处理，程序非常容易崩溃，这是面试时非常忌讳的事情。由于没有必要在每一次递归中做边界检查（每一次递归都做检查，增加了不必要的时间开销），上述代码只在HasSubtree中作了边界检查后，在HasSubtreeCore中作递归遍历。

接下来考虑第二步，判断以树A中以N为根结点的子树是不是和树B具有相同的结构。同样，我们也可以用递归的思路来考虑：如果结点N的值和树B的根结点不相同，则以N为根结点的子树和树B肯定不具有相同的结点；如果他们的值相同，则递归地判断他们的各自的左右结点的值是不是相同。递归的终止条件是我们到达了树A或者树B的叶结点。参考代码如下：

bool DoesTree1HaveAllNodesOfTree2(TreeNode\* pTreeHead1, TreeNode\* pTreeHead2)

{

        if(pTreeHead2 == NULL)

                return true;

        if(pTreeHead1 == NULL)

                return false;

        if(pTreeHead1->m\_nValue != pTreeHead2->m\_nValue)

                return false;

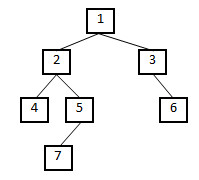
        return DoesTree1HaveAllNodesOfTree2(pTreeHead1->m\_pLeft, pTreeHead2->m\_pLeft) &&

                DoesTree1HaveAllNodesOfTree2(pTreeHead1->m\_pRight, pTreeHead2->m\_pRight);

}

**程序员面试题精选100题(60)-判断二叉树是不是平衡[数据结构]**

题目：**输入一棵二叉树的根结点，判断该树是不是平衡二叉树。如果某二叉树中任意结点的左右子树的深度相差不超过1，那么它就是一棵平衡二叉树。**例如下图中的二叉树就是一棵平衡二叉树：



在[本系列博客的第27题](http://zhedahht.blog.163.com/blog/static/25411174200732975328975/" \t "_blank)，我们曾介绍过如何求二叉树的深度。有了求二叉树的深度的经验之后再解决这个问题，我们很容易就能想到一个思路：在遍历树的每个结点的时候，调用函数TreeDepth得到它的左右子树的深度。如果每个结点的左右子树的深度相差都不超过1，按照定义它就是一棵平衡的二叉树。这种思路对应的代码如下：

bool IsBalanced(BinaryTreeNode\* pRoot)

{

    if(pRoot == NULL)

        return true;

    int left = TreeDepth(pRoot->m\_pLeft);

    int right = TreeDepth(pRoot->m\_pRight);

    int diff = left - right;

    if(diff > 1 || diff < -1)

        return false;

    return IsBalanced(pRoot->m\_pLeft) && IsBalanced(pRoot->m\_pRight);

}

上面的代码固然简洁，但我们也要注意到由于一个节点会被重复遍历多次，这种思路的时间效率不高。例如在函数IsBalance中输入上图中的二叉树，首先判断根结点（值为1的结点）的左右子树是不是平衡结点。此时我们将往函数TreeDepth输入左子树根结点（值为2的结点），需要遍历结点4、5、7。接下来判断以值为2的结点为根结点的子树是不是平衡树的时候，仍然会遍历结点4、5、7。毫无疑问，重复遍历同一个结点会影响性能。接下来我们寻找不需要重复遍历的算法。

如果我们用后序遍历的方式遍历二叉树的每一个结点，在遍历到一个结点之前我们已经遍历了它的左右子树。只要在遍历每个结点的时候记录它的深度（某一结点的深度等于它到叶节点的路径的长度），我们就可以一边遍历一边判断每个结点是不是平衡的。下面是这种思路的参考代码：

bool IsBalanced(BinaryTreeNode\* pRoot, int\* pDepth)

{

    if(pRoot == NULL)

    {

        \*pDepth = 0;

        return true;

    }

    int left, right;

    if(IsBalanced(pRoot->m\_pLeft, &left)

        && IsBalanced(pRoot->m\_pRight, &right))

    {

        int diff = left - right;

        if(diff <= 1 && diff >= -1)

        {

            \*pDepth = 1 + (left > right ? left : right);

            return true;

        }

    }

    return false;

}

我们只需要给上面的函数传入二叉树的根结点以及一个表示结点深度的整形变量就可以了：

bool IsBalanced(BinaryTreeNode\* pRoot)

{

    int depth = 0;

    return IsBalanced(pRoot, &depth);

}

在上面的代码中，我们用后序遍历的方式遍历整棵二叉树。在遍历某结点的左右子结点之后，我们可以根据它的左右子结点的深度判断它是不是平衡的，并得到当前结点的深度。当最后遍历到树的根结点的时候，也就判断了整棵二叉树是不是平衡二叉树了。