原创 二叉树的常见算法

2018-08-05 16:28:21 幸福的起点_ 阅读数 16489 收藏 更多

1.二叉树的遍历算法

二叉树的遍历主要分为三种：**先序遍历，中序遍历和后序遍历。**还有一种就是按照层次遍历。
按照惯例，左孩子优先于右孩子，那么：

- **先序遍历**指的就是先访问本节点，再访问该节点的左孩子和右孩子；
- **中序遍历**指的就是：先访问左孩子，再访问本节点，最后访问右孩子；
- **后序遍历**指的就是：先访问左右孩子，最后访问本节点。
- **层次遍历**：按照树的每一层(高度)进行遍历。

树的节点的数据结构常声明为：

```
struct TreeNode {
    int val;
    TreeNode *left;     //左孩子节点
    TreeNode *right;    //右孩子节点
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
}
```

约定给出根节点，分别使用三种遍历方式得到二叉树的序列：得益于递归的简洁性，三种遍历方式的递归算法也是非常简洁和易懂的。

(1). 先序遍历

```
//递归版本
void preOrderTraversal(vector<int> &store, TreeNode *root) {
    if(!root) return;
    store.push_back(root->val);
    preOrderTraversal(store, root->left);   //左孩子优先
    preOrderTraversal(store, root->right);
}
```

先序遍历的理解：**沿着最左侧通路自顶而下访问各个节点，自底而上遍历对应的右子树。**迭代版本需要用到栈这种数据结构。

```
//递归版本
void preOrderTraversal(vector<int> &store, TreeNode *root) {
    stack<TreeNode *> S;
    S.push(root);
    while(!S.empty()) {
        TreeNode *curr_node = S.top();
        S.pop();
        if(curr_node) {
            store.push_back(curr_node->val);
            S.push(curr_node->right);   //左孩子优先，所以右孩子先入栈
            S.push(curr_node->left);
        }
    }
    return;
}
```

(2). 中序遍历

```
//递归版本
void inOrderTraversal(vector<int> &store, TreeNode *root) {
    if(!root) return;
```

```
    inOrderTraversal(store, root->left);
    store.push_back(root->val);
    inOrderTraversal(store, root->right);
    return;
}
```

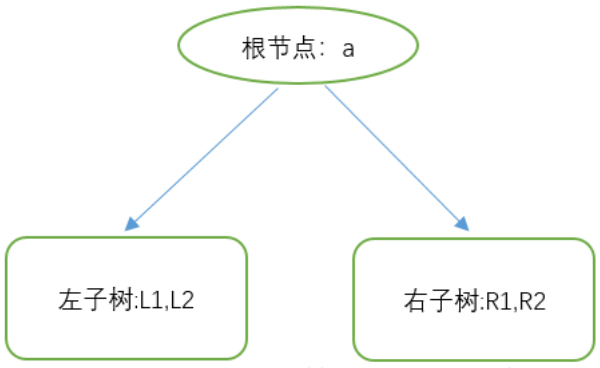
中序遍历的迭代版本需要借用一个数据结构：栈，使用一个栈来保存，根节点沿着左通路一直往下访问的节点。

```
void inorderTraversal(vector<int> &store, TreeNode* root) {
    strack<TreeNode *> S;
    while(root || !S.empty()) {
        while(root) {
            S.push(root);
            root = root->left;
        }
        TreeNode *curr_node = S.top();
        S.pop();
        store.push_back(curr_node->val);
        root = curr_node->right;
    }
    return;
}
```

(3). 后序遍历

```
//递归版本
void postOrderTraversal(vector<int> &store, TreeNode *root) {
    if(!root) return;
    postOrderTraversal(store, root->left);  //右孩子优先
    postOrderTraversal(store, root->right);
    store.push_back(root->val);
}
```

后序遍历的迭代版本和前序遍历类似：
可以证明，右孩子优先的先序遍历序列的逆序列就是左孩子优先的后序遍历序列。



假设根节点的值为a，L1和R1分别是左子树和右子树在右孩子优先的先序遍历序列，L2和R2分别是左子树和右子树在左孩子优先的后序遍历序列，所以序列 `"a,R1,L1"` 是序列 `"L2,R2,a"` 的逆序列即可。
从序列的组成来看，只需要证明 `R1` 是 `R2` 的逆序列且 `L1` 是 `L2` 的逆序列，显然这就将问题分解，平凡的情况下是显然成立的，因此可以归纳证明出这个

```
//迭代版本
void postOrderTraversal(vector<int> &store, TreeNode *root) {
    stack<TreeNode *> S;
    S.push(root);
    while(!S.empty()) {
        TreeNode *curr_node = S.top();
        S.pop();
        if(curr_node) {
            store.push_back(curr_node->val);
            S.push(curr_node->left);    //右孩子优先，所以左孩子先入栈
            S.push(curr_node->right);
```

```
        }
    }
    std::reverse(store.begin(), store.end());   //逆序列即为所求
    return;
}
```

(4). 层次遍历

二叉树的按照层次遍历，需要使用数据结构队列queue，每次出队列的元素，将其左右孩子入队列。

```
//迭代版本
void levelOrderTraversal(vector<int> &store, TreeNode *root) {
    queue<TreeNode *> Q;
    Q.push(root);
    while(!Q.empty()) {
        TreeNode *curr_node = Q.front();
        Q.pop();
        if(curr_node) {
            store.push(curr_node->val);
            Q.push(curr_node->left);
            Q.push(curr_node->right);
        }
    }
    return;
}
```

2. 二叉树的其它算法

(1). 二叉树的深度

递归版本非常简洁，也非常易懂；迭代版本则需要利用我们之前介绍的按照层次遍历，层数就是二叉树的深度。

```
//递归版本
int TreeDepth(TreeNode *pRoot) {
    return pRoot ? 1 + max(TreeDepth(pRoot->left),
                           TreeDepth(pRoot->right)) : 0;
}
//迭代版本
int TreeDepth2(TreeNode *pRoot) {
    queue<TreeNode *> Q;
    Q.push(pRoot);
    int depth = 0;
    while(!Q.empty()) {
        int len = Q.size();
        ++depth;
        while(len--) {
            TreeNode *curr_node = Q.front();
            Q.pop();
            if(curr_node) {
                Q.push(curr_node->left);
                Q.push(curr_node->right);
            }
        }
    }
    return depth - 1;   //将叶节点的空孩子节点也算作一层了，所以减1
}
```

(2). 二叉树的镜像

操作给定的二叉树，将其变换为源二叉树的镜像。使用递归，当节点存在至少一个孩子时，交换左右孩子，再递归处理。

```
//二叉树的镜像
void Mirror(TreeNode *pRoot) {
    if (pRoot && (pRoot->left || pRoot->right)) {
        std::swap(pRoot->left, pRoot->right);
        Mirror(pRoot->left);
        Mirror(pRoot->right);
    }
```

```
    return;
}
```

(3). 平衡二叉树

输入一棵二叉树，判断该二叉树是否是平衡二叉树。
平衡二叉树指的是：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树，见百度百科。**注意点：子树超过1且子树也是平衡树。**构造一个递归函数 `IsBalanced` 来判断这两个条件。

```
//平衡二叉树
bool IsBalanced(TreeNode *pRoot, int &pDepth) {
    if (!pRoot) {
        pDepth = 0;
        return true;
    }
    int left_depth, right_depth;    //记录左右子树的高度
    if (IsBalanced(pRoot->left, left_depth) &&
        IsBalanced(pRoot->right, right_depth)) {
        int diff = left_depth - right_depth;
        if (diff <= 1 && diff >= -1) {
            pDepth = 1 + (left_depth > right_depth ? left_depth : right_depth);
            return true;
        }
    }
    return false;
}
bool IsBalanced_Solution(TreeNode *pRoot) {
    int pDepth = 0;
    return IsBalanced(pRoot, pDepth);
}
```

(4). 对称的二叉树

请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。

```
//对称的二叉树
bool isSymmetrical(TreeNode *leftChild, TreeNode *rightChild) {
    if (!leftChild && !rightChild) {
        //左右子树同时为空
        return true;
    }
    else if (leftChild && rightChild) {
        //左右子树都不为空
        return leftChild->val == rightChild->val &&
               isSymmetrical(leftChild->left, rightChild->right) &&
               isSymmetrical(leftChild->right, rightChild->left);
    }
    else {
        return false;
    }
}
bool isSymmetrical(TreeNode *pRoot) {
    if (!pRoot) return true;
    return isSymmetrical(pRoot->left, pRoot->right);
}
```

(5). 把二叉树打印成多行

从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。
在求二叉树的深度的时候，迭代解法起始我们已经做了这个事情，只是没有按照多行输出，所以只需要记录每一行的 `val` 即可。

```
//把二叉树打印成多行
vector<vector<int>> Print(TreeNode *pRoot) {
    vector<vector<int>> store;
    queue<TreeNode *> Q;
    Q.push(pRoot);
    int index = 0;
    while (!Q.empty()) {
```

```
        int length = Q.size();
        store.push_back(vector<int>());
        while (length--) {
            TreeNode *curr_node = Q.front();
            Q.pop();
            if (curr_node) {
                store[index].push_back(curr_node->val);
                Q.push(curr_node->left);
                Q.push(curr_node->right);
            }
        }
        ++index;
    }
    store.pop_back();   //将叶节点的空孩子节点也算作一层了，所以pop_back()
    return store;
}
```

(6). 二叉树的下一个结点

给定一个二叉树和其中的一个结点，请找出**中序遍历**顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。节点的数据结构表示为：

```
struct TreeLinkNode {
    int val;
    struct TreeLinkNode *left;
    struct TreeLinkNode *right;
    struct TreeLinkNode *next;  //父节点
    TreeLinkNode(int x) :val(x), left(NULL), right(NULL), next(NULL) {}
};
```

若允许一定的**空间复杂度**，可以直接**中序遍历保存序列**之后直接查找。这种方法就不介绍了，下面介绍常数空间的算法：

- 如果该**节点有右孩子**，必然下一个节点就是该节点的右孩子的沿着左侧链下的最后一个左孩子；
- 该节点**没有右孩子，也没有父节点**，说明这个节点是最后一个节点；
- 该节点**没有右孩子，但是有父节点且是父节点的左孩子**，必然下一个节点就是该节点的父节点；
- 该节点**没有右孩子，且有父节点且是父节点的右孩子**，要么该节点是最后一个节点，要么沿着父链上升，之后第一个节点的是其父节点的左孩子，是下一个节点。

```
//二叉树的下一个结点
TreeLinkNode *GetNext(TreeLinkNode *pNode) {
    if (!pNode) return pNode;
    if (pNode->right) {     //有右孩子的情况
        pNode = pNode->right;
        while (pNode->left) {
            pNode = pNode->left;
        }
        return pNode;
    }
    else {      //没有右孩子的情况
        TreeLinkNode *parent = pNode->next;
        if (!parent) {
            return parent;
        }
        else {
            if (pNode == parent->left) {    //该节点是其父节点的左孩子
                return parent;
            }
            else {  //该节点是其父节点的右孩子，沿左侧链上升
                while (parent->next && parent == parent->next->right) {
                    parent = parent->next;
                }
                return parent->next;
            }
        }
    }
}
```

(7). 二叉搜索树与双向链表

输入一棵二叉搜索树，将该二叉搜索树转换成一个**排序的双向链表**。要求不能创建任何新的结点，只能调整树中结点指针的指向。
二叉搜索树(Binary Search Tree, BST)：它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不空，则左子树上所有结点的值小于它的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树。详见百度百科。
中序遍历一颗二叉搜索树，必然得到的是一个有序的序列。

```
void recurConvert(TreeNode *root, TreeNode *&pre) {
  if (!root) return;
  recurConvert(root->left, pre);
  root->left = pre;
  if (pre) pre->right = root;
  pre = root;
  recurConvert(root->right, pre);
}
TreeNode *Convert(TreeNode *pRootOfTree) {
  if (!pRootOfTree) return pRootOfTree;
  TreeNode *pre = 0;
  recurConvert(pRootOfTree, pre);
  TreeNode *res = pRootOfTree;
  while (res->left) res = res->left;
  return res;
}
```

(8). 二叉树中和为某一值的路径

输入一颗二叉树的跟节点和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。(注意: 在返回值的list中，数组长度大的数组靠前)

```
//二叉树中和为某一值的路径
void FindPath(vector<vector<int>> &vec_store,
              vector<int> store,
              TreeNode *root,
              int expNumber) {
    store.push_back(root->val);
    if (!root->left && !root->right) {
        if (expNumber == root->val) vec_store.push_back(store);
        return;
    }
    if (root->left)
        FindPath(vec_store, store, root->left, expNumber - root->val);
    if (root->right)
        FindPath(vec_store, store, root->right, expNumber - root->val);
    store.pop_back();       //回溯
}
vector<vector<int>> FindPath(TreeNode *root, int expectNumber) {
    vector<vector<int>> vec_store;
    vector<int> store;
    if(root) FindPath(vec_store, store, root, expectNumber);
    return vec_store;
}
```

(9). 按之字形顺序打印二叉树

请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。
思路，构建两个栈，依次保存奇数行和偶数行的节点，注意左右孩子的入栈顺序；例如：从第0行到第1行，第1行先左孩子，再右孩子入栈，这样出栈时就是从右往左的顺序；从第1行到第2行，先右孩子，再左孩子入栈，这样才能保证出栈的顺序是从左往右。

```
//按之字形顺序打印二叉树
vector<vector<int>> Print(TreeNode *pRoot) {
    if (!pRoot) return {};
    vector<vector<int>> result;
    stack<TreeNode *> odd, even;
    even.push(pRoot);       //从第零行开始
    while (!even.empty() || !odd.empty()) {
        vector<int> line;
        if (odd.empty()) {
            while (!even.empty()) {
```

```
                TreeNode *curr_node = even.top();
                even.pop();
                line.push_back(curr_node->val);
                if (curr_node->left) odd.push(curr_node->left);
                if (curr_node->right) odd.push(curr_node->right);   //注意，先左后右
            }
        }
        else {
            while (!odd.empty()) {
                TreeNode *curr_node = odd.top();
                odd.pop();
                line.push_back(curr_node->val);
                if (curr_node->right) even.push(curr_node->right);
                if (curr_node->left) even.push(curr_node->left);        //注意，先右后左
            }
        }
        result.push_back(line);
    }
    return result;
}
```

10. 二叉搜索树的第k个结点

给定一棵二叉搜索树，请找出其中的第k小的结点。例如： （5，3，7，2，4，6，8） 中，按结点数值大小顺序第三小结点的值为4。 如果按照中序遍历种思路是非常简单且容易实现的：

```
void inOrderTraversal(TreeNode *pRoot, vector<TreeNode *> &store) {
    if(!pRoot) return;
    inOrderTraversal(pRoot->left, store);
    store.push_back(pRoot);
    inOrderTraversal(pRoot->right, store);
}
TreeNode* KthNode(TreeNode *pRoot, int k) {
    vector<TreeNode *> store;
    inOrderTraversal(pRoot, store);
    return k > 0 && k <= store.size() ? store[k - 1] : nullptr;
}
```

11. 二叉搜索树的后序遍历序列

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。对于后序遍历，我们知道序列最后一个数字必定是二叉搜索树的根节点，由于保证序列的任意两个数字都互不相同，设序列中第一个大于根节点的位置 `[0,k)` 这一段子序列必然是**左子树后序遍历得到的子序列**， `[k+1, end)` 必然是**右子树后序遍历得到的子序列**，依此递归即可，另外，在 `[k+1, end)` 若有**点的值小于根节点**，说明不是二叉搜索树的后序遍历序列 。

```
//二叉搜索树的后序遍历序列
bool judgeBST(vector<int>::iterator first, vector<int>::iterator last) {
    if (last == first) return true;
    int root_val = *(last - 1);
    auto iter = first;
    while (iter < last - 1) {
        if (*iter > root_val) break;
        ++iter;
    }
    auto temp = iter;
    while (iter < last - 1) {
        if (*iter  <= root_val) return false;
        ++iter;
    }
    return judgeBST(first, temp) && judgeBST(temp, last - 1);
}
bool VerifySquenceOfBST(vector<int> sequence) {
    if (sequence.empty()) return false;
    return judgeBST(sequence.begin(), sequence.end());
}
```

11. 重建二叉树

输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入 {1,2,4,7,3,5,6,8} 和中序遍历序列 {4,7,2,1,5,3,8,6}，则重建二叉树并返回。

```
//重建二叉树
TreeNode *reConstructBinaryTree(vector<int>::iterator pre_first,
                                vector<int>::iterator pre_last,
                                    vector<int>::iterator vin_first,
                                    vector<int>::iterator vin_last) {
    if (vin_last - vin_first != pre_last - pre_first ||
            pre_last == pre_first ||
            vin_first == vin_last) {
        return nullptr;
    }
    TreeNode *curr_node = new TreeNode(*pre_first);
    if (pre_last == pre_first + 1) return curr_node;
    auto iter = vin_first;
    while (iter < vin_last) {
        if (*iter == *pre_first) break;
        iter++;
    }
    int len = iter - vin_first;
    curr_node->left = reConstructBinaryTree(pre_first + 1, pre_first + len + 1, vin_first, iter);
    curr_node->right = reConstructBinaryTree(pre_first + len + 1, pre_last, iter + 1, vin_last);
    return curr_node;
}
TreeNode *reConstructBinaryTree(vector<int> pre, vector<int> vin) {
  return reConstructBinaryTree(pre.begin(), pre.end(), vin.begin(), vin.end());
}
```

12. 树的子结构

输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）

```
//树的子结构
bool isSubtree(TreeNode *pRoot1, TreeNode *pRoot2) {
  //判断以pRoot2为根节点的树是否是以pRoot1为根节点的树的子树
    if (!pRoot2) return true;
    if (!pRoot1) return false;
    return pRoot1->val != pRoot2->val ? false :
           isSubtree(pRoot1->left, pRoot2->left) &&
           isSubtree(pRoot1->right, pRoot2->right);
}
bool HasSubtree(TreeNode *pRoot1, TreeNode *pRoot2) {
    if(!pRoot1 || !pRoot2) return false;
    return isSubtree(pRoot1, pRoot2) ||
           isSubtree(pRoot1->left, pRoot2) ||
           isSubtree(pRoot1->right, pRoot2);
}
```

相关的在线练习请参考牛客网剑指offer。

文章最后发布于: 2018-08-0

二叉树的各种遍历算法以及实例 阅读数 4万+

一、二叉树在计算机科学中，树是一种重要的非线性数据结构，直观地看，它是数据元素（在树中称为... 博文 来自： 大道至简，知...

想对作者说点什么

shawfy- 1个月前 #2楼

lz你12树的子结构代码的最后两句写错了，应该是HasSubtree(pRoot1->left, pRoot2) || HasSubtree(pRoot1->right, pRoot2);

fgdhio 5个月前 #1楼

可以