OneStepOffer 第十讲

Tree

课前回顾

上节课我们讲了 dfs 深度优先 三种顺序的遍历：前序，中序，后序

前序：[1 2 4 5 3 6]
中序：[4 2 5 1 3 6]
后序：[4 5 2 6 31]

dfs 练习题

Construct Binary Tree from Preorder and Inorder Traversal (Medium)

题目描述
给定一个二叉树的前序遍历和中序遍历结果，尝试复原这个树。已知树里不存在重复值的节点。

输入是两个一维数组，分别表示树的前序遍历和中序遍历结果;输出是一个二叉树。

Input: preorder = [4,9,20,15,7], inorder = [9,4,15,20,7]
Output:

```
    4
   /\
  9  20
     /\
    15  7
```

题目分析

1. 前序遍历的第一个节点是 4，意味着 4 是根节点。
2. 我们在中序遍历结果里找到 4 这个节点
3. 根据中序遍历的性质可以得出，4 在中序遍历数组位置的左子数组为左子树，节点数为 1，对应的是前序排列数组里 4 之后的 1 个数字(9)
4. 4 在中序 遍历数组位置的右子数组为右子树，节点数为 3 (20 这个数)，对应的是前序排列数组里最后的 3 个数字

...

C++ 解答

```
// 主函数
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    if (preorder.empty()) return nullptr;
    unordered_map<int, int> hash;
    for (int i = 0; i < preorder.size(); ++i) hash[inorder[i]] = i;
     return buildTreeHelper(hash, preorder, 0, preorder.size() - 1, 0);
}

// 辅函数
TreeNode* buildTreeHelper(unordered_map<int, int> & hash, vector<int>& preorder, int s0, int e0, int s1) {
     if (s0 > e0) return nullptr;
     int mid = preorder[s1], index = hash[mid], leftLen = index - s0 - 1;
     TreeNode* node = new TreeNode(mid);
     node->left = buildTreeHelper(hash, preorder, s0, index - 1, s1 + 1);
     node->right = buildTreeHelper(hash, preorder, index + 1, e0, s1 + 2 + leftLen);
     return node;
}
```

Java 解法

```
// 主函数
TreeNode buildTree (int[] preorder, int[] inorder) {
    if (preorder.length == 0) return null;
    HashMap<Integer, Integer> hash = new HashMap<Integer, Integer>();
    for (int i = 0; i < preorder.length; i++) {
        hash.put(inorder[i], i);
    }
    return buildTreeHelper(hash, preorder, 0, preorder.length - 1, 0);
}
// 辅函数
TreeNode buildTreeHelper (HashMap<Integer, Integer> hash, int[] preorder, int s0, int e0, int s1) {
    if (s0 > e0) return null;
    int mid = preorder[s1];
    int index = hash.get(mid);
    int leftLength = index - s0 - 1;
    TreeNode node = new TreeNode(mid);
    node.left = buildTreeHelper(hash, preorder, s0, index-1, s1+1);
    node.right = buildTreeHelper(hash, preorder, index+1, e0, s1+2);
    return node;
}
```

时间/空间复杂度分析

时间复杂度：0(n) -- 遍历pre-order + 递归 buildTreeHelper

空间复杂度：O(n) -- hashmap

二叉树查找

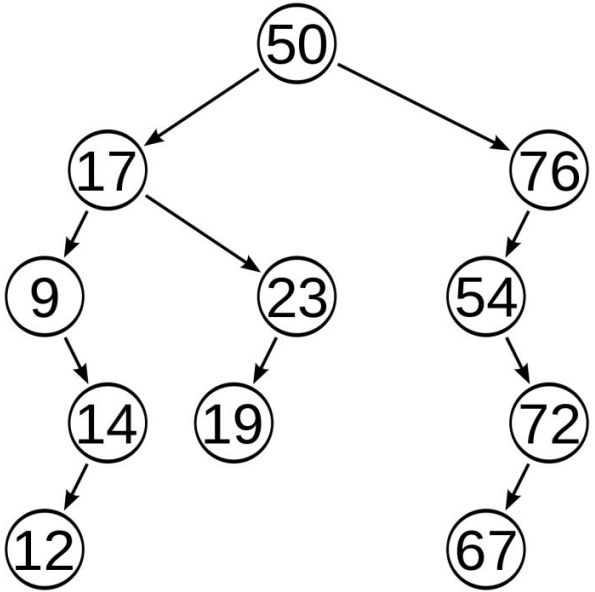
二叉查找树(Binary Search Tree, BST)是一种特殊的二叉树

对于每个父节点，其左子节点 的值小于等于父结点的值，其右子节点的值大于等于父结点的值。

因此对于一个二叉查找树，我们可以在 $O(n \log n)$ 的时间内查找一个值是否存在:

从根节点开始，若当前节点的值大于查找值 则向左下走，若当前节点的值小于查找值则向右下走。同时因为二叉查找树是有序的，对其中序 遍历的结果即为排好序的数组。

二叉树查找 例子



BST Node

```
public class BST_Node {
    int data;
    BST_Node left;
    BST_Node right;
}
```

BST Insertion Function (插入)

```
BST_Node insert(BST_Node t, int x) {
    if (t == null) {
        t = new BST_Node();
        t.data = x;
        t.left = t.right = null;
    } else if (x < t.data) {
        t.left = insert(t.left, x);
    } else if (x > t.data) {
        t.right = insert(t.right, x);
    }
    return t;
}
```

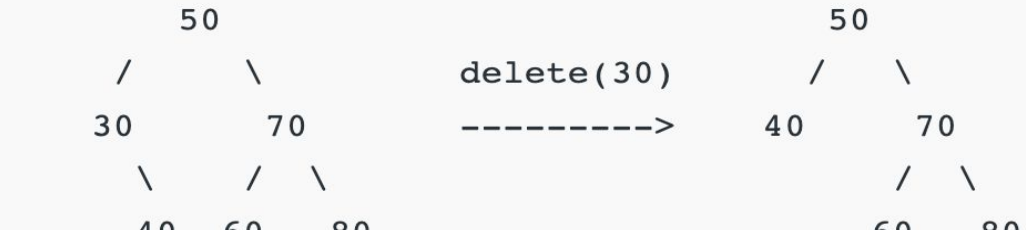
BST find, findMax, findMin

```
BST_Node find (BST_Node t, int val) {
    if (t == null) return null;
    if (val < t.data) return find(t.left, val);
    if (val> t.data) return find(t.right, val);
    return t;
}

BST_Node findMin (BST_Node t) {
    if (t == null || t.left == null) return t;
    return findMin(t.left);
}

BST_Node findMax (BST_Node t) {
    if (t == null || t.left == null) return t;
    return findMax(t.left);
}
```

BST Remove

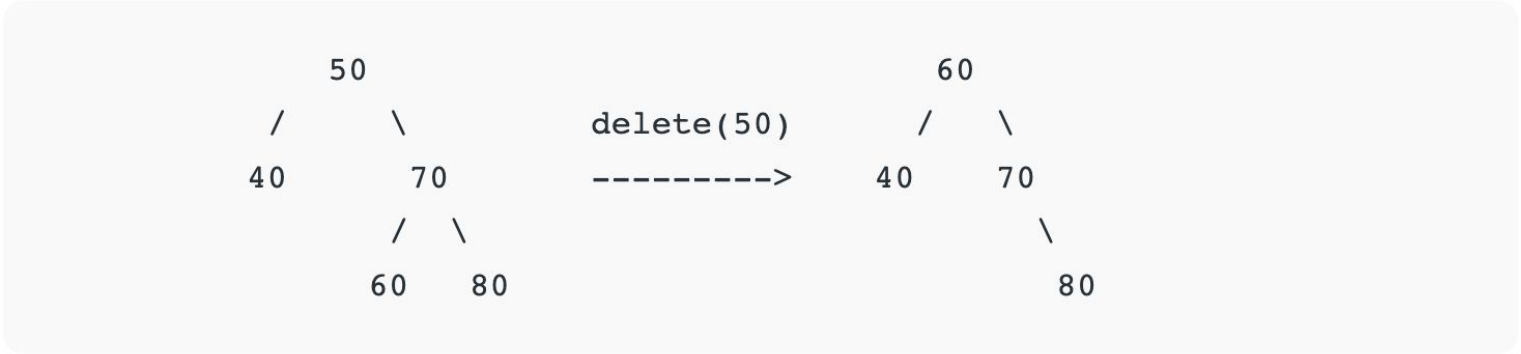
1) ***Node to be deleted is leaf:*** Simply remove from the tree.

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child



BST Remove

3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



BST Remove Implementation

```
BST_Node remove(BST_Node t, int x) {
      BST_Node temp;
      if (t == null) return null;
      else if (x < t.data) t.left = remove(t.left, x);
      else if (x > t.data) t.right = remove(t.right, x);
      else if (t.left !=null && t.right!=null) {
         temp = findMin(t.right);
         t.data = temp.data;
         t.right = remove(t.right, t.data);
      } else {
         temp = t;
         if (t.left == null) t = t.right;
         else  t = t.left;
      }
      return t;
}
```

BST find, insert, remove C++实现

```
template <class T>
class BST {
   struct Node {
      T data;
      Node* left;
      Node* right;
   };
   Node* root;
   Node* makeEmpty(Node* t) {
     If (t == NULL) return NULL;
     makeEmpty(t->left);
    makeEmpty(t->right);
    delete t;
    return NULL;
   }

   Node* insert(Node* t, T x) {
     if (t == NULL) {
         t = new Node;
         t->data = x;
         t->left = t->right = NULL;
     } else if (x < t->data) {
         t->left = insert(t->left, x);
     } else if (x > t->data) {
         t->right = insert(t->right, x);
     }
     return t;
   }
```

```
Node* find(Node* t, T x) {
   if (t == NULL) return NULL;
   if (x < t->data) return find(t->left, x);
   if (x > t->data) return find(t->right, x);
   return t;
}

Node* findMin(Node* t) {
   if (t == NULL || t->left == NULL) return t;
   return findMin(t->left);
}

Node* findMax(Node* t) {
    if (t == NULL || t->right == NULL) return t;
    return findMax(t->right);
}
```

```
Node* remove(Node* t, T x) {
    Node* temp;
    if (t == NULL) return NULL;
    else if (x < t->data) t->left = remove(t->left, x);
    else if (x > t->data) t->right = remove(t->right, x);
    else if (t->left && t->right) {
         temp = findMin(t->right);
         t->data = temp->data;
         t->right = remove(t->right, t->data);
     } else {
          temp = t;
          if (t->left == NULL) t = t->right;
          else if (t->right == NULL) t = t->left;
          delete temp;
     }
     return t;
}
```

BST 和其他数据结构区别

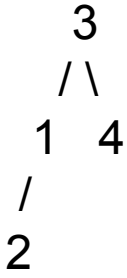
Operation	BST	Array	Sorted Array	Tree
Insertion	O(h)	O(N)	O(N)	O(N)
find	O(h)	O(N)	O(logN)	O(N)
findMin/Max	O(h)	O(N)	O(1)	O(N)
Remove	0(h)	O(N)	O(N)	O(N)

99. Recovery Binary Search Tree

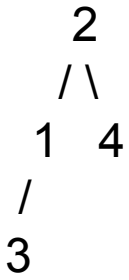
给定一个二叉查找树，已知有两个节点被不小心交换了，试复原此树。

输入是一个被误交换两个节点的二叉查找树，输出是改正后的二叉查找树。

Input:



Output:



input 2 和 3 被调换

题目分析

我们可以使用中序遍历这个二叉查找树，同时设置一个 prev 指针，记录当前节点中序遍历时的前节点。

如果当前节点大于 prev 节点的值，说明需要调整次序。

如果遍历整个序列过程中只出现了一次次序错误，说明就是这两个相邻节点需要被交换；

如果出现了两次次序错误，那就需要交换这两个节点。

C++ 实现

```
// 主函数
void recoverTree(TreeNode* root) {
    TreeNode *mistake1 = nullptr, *mistake2 = nullptr, *prev = nullptr;
    inorder(root, mistake1, mistake2, prev);
    if (mistake1 && mistake2) {
        int temp = mistake1->val;
        mistake1->val = mistake2->val;
        mistake2->val = temp;
    }
}
```

```
// 辅函数
void inorder(TreeNode* root, TreeNode*& mistake1, TreeNode*&
mistake2, TreeNode*& prev) {
    if (!root) return;
    if (root->left) inorder(root->left, mistake1, mistake2, prev);
    if (prev && root->val < prev->val) {
        if (!mistake1) {
            mistake1 = prev;
            mistake2 = root;
        } else {
            mistake2 = root;
        }
    }
    prev = root;
    if (root->right) inorder(root->right, mistake1, mistake2, prev);
}
```

Java 实现

```
TreeNode x = null, y = null, pred = null;
public void swap(TreeNode a, TreeNode b) {
    int tmp = a.val;
    a.val = b.val;
    b.val = tmp;
}

public void findTwoSwapped(TreeNode root) {
    if (root == null) return;
    findTwoSwapped(root.left);
    if (pred != null && root.val < pred.val) {
        y = root;
        if (x == null) x = pred;
        else return;
    }
    pred = root;
    findTwoSwapped(root.right);
}
```

```
public void recoverTree(TreeNode root) {
    findTwoSwapped(root);
    swap(x, y);
}
```

时间空间复杂度分析

时间复杂度:O(n) 需要inorder遍历所有的节点

空间复杂度:O(1)