

# LEETCODE: 450

---

By

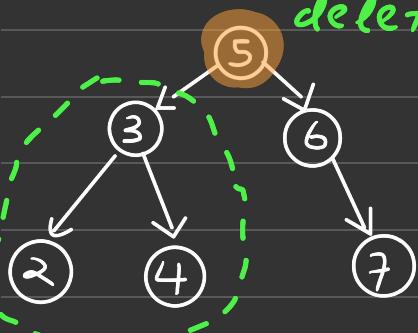
Priyansh



# PROBLEM:

We are provided with a BST And a key value that is present in any of node of the BST and we have to delete that node maintaining definition of BST...

Example: Lets begin with the testcase of deleting a leaf node!!!



target = 4

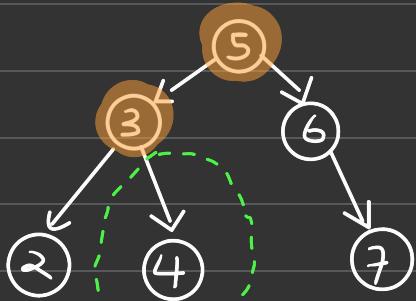
$4 < 5$  move left And rearrange  
left Subtree:-

using recursion

We would try to rearrange left subtree!!!

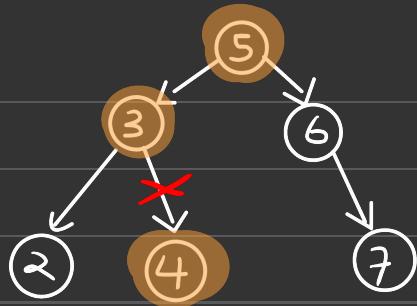
target = 4

$4 > 3$  move right And rearrange  
Right Subtree:-



we would try rearranging

Right Subtree node of current root = 3

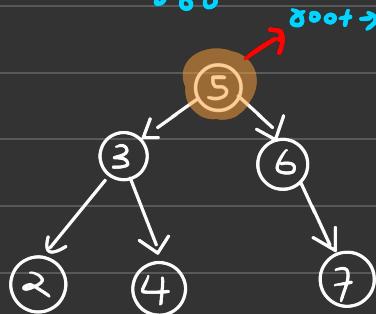


$\text{target} = 4$

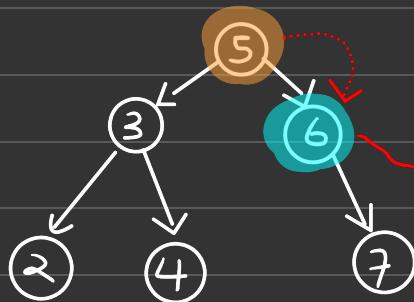
$$4 == 4$$

Now perform deletion  
Since its a leafnode  
we would return  $\text{NULL}$ .

Case 2) Target Node is having a single child!!!



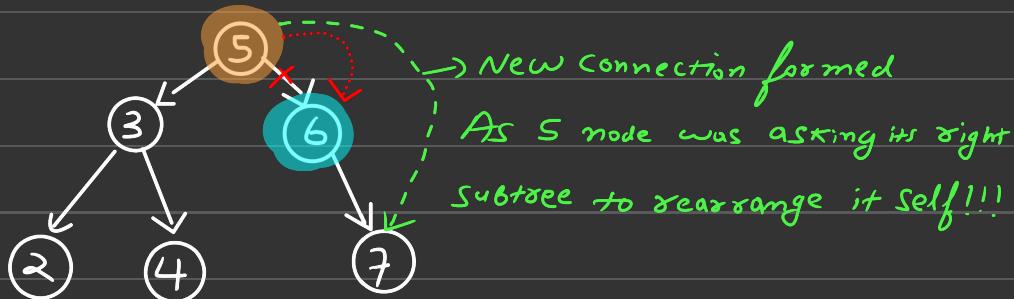
$\text{root} \rightarrow \text{Val} \neq \text{target}$   $5 < 6 \rightarrow 5$  would actually ask  
 $\text{target} == 6$  to rearrange its  
right subtree  
which could possibly  
have node with  
 $\text{Val} == 6$



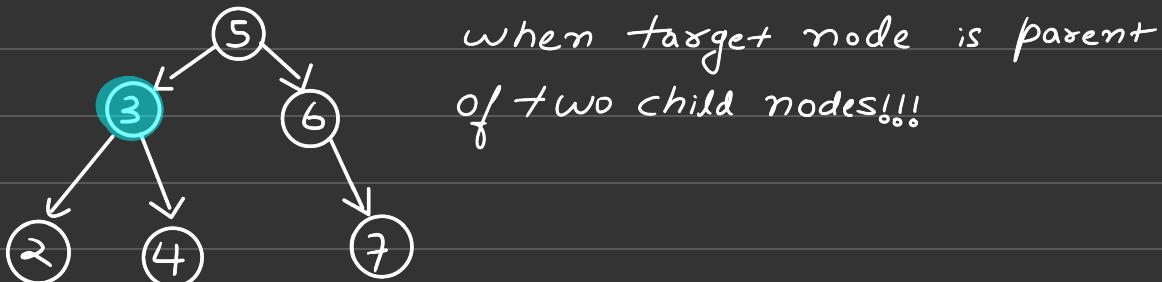
6 will say my Val is equal to  
 $\text{target} \rightarrow \text{Val}$

$08 \rightarrow \text{root} \rightarrow \text{Val} == \text{target} \rightarrow \text{Val}$

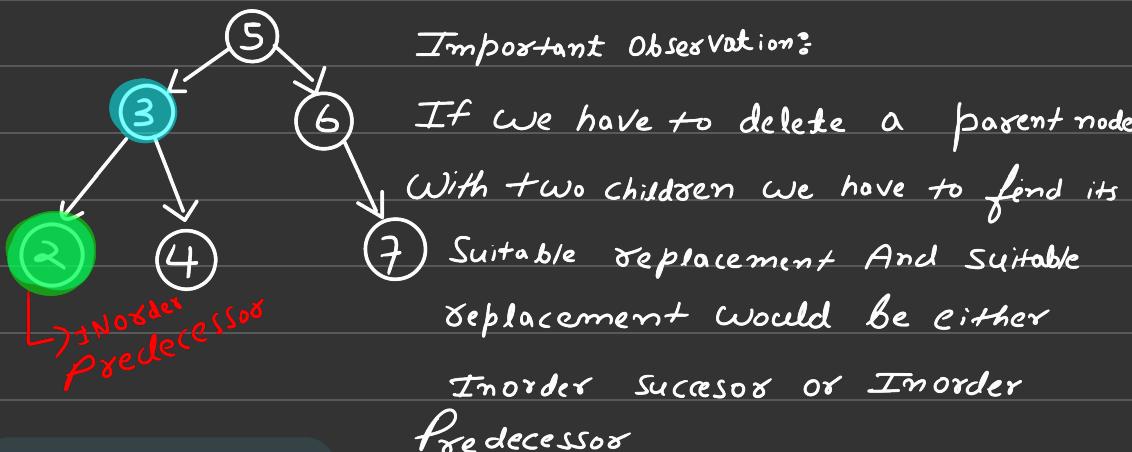
Now it would return its child  
Node that is not  $\text{NULL}$  to its  
calling function (It would be an  
updated connection)



## MOST IMPORTANT CASE:



LETS PROCEED with deletion of node 3...



\* We would use in-order predecessor

So to find replacement of the node to be deleted we would firstly find out its in-order predecessor by a function:-

```
1 class Solution {
2 public:
3     TreeNode* in_predecessor(TreeNode* root){
4         TreeNode* pred = root->left;
5         while(pred->right){
6             pred = pred->right;
7         }
8         return pred;
9     }
}
```

}  
function to  
find In order  
predeccesor

Now we would update current root's value to the value of its in-order predecessor now we would call the function delete\_node to delete that in-order predecessor Node... (Method works Always As in-order predecessor node would Always be parent of Single child or will be a leaf Node)

# Code Implementation:

```
TreeNode* deleteNode(TreeNode* root, int key) {  
    if(root == NULL) return NULL;  
    if(root->val == key){  
        if(root->left == NULL && root->right == NULL) return NULL;  
        if(root->left == NULL || root->right == NULL){  
            if(root->left != NULL) return root->left;  
            else return root->right;  
        }  
        if(root->left != NULL && root->right != NULL){  
            TreeNode* pred = in_predecessor(root);  
            root->val = pred->val;  
            root->left = deleteNode(root->left,pred->val);  
        }  
    }  
    else if(root->val > key){  
        //move left  
        root->left = deleteNode(root->left,key);  
    }  
    else{  
        //move right  
        root->right = deleteNode(root->right,key);  
    }  
    return root;  
}  
};|
```

```
TreeNode* deleteNode(TreeNode* root, int key) {
```

if(root == NULL) return NULL; → Base

if(root->val == key){ → When root->val == key

if(root->left == NULL && root->right == NULL) return NULL;

if(root->left == NULL || root->right == NULL){

if(root->left != NULL) return root->left;

else return root->right;

}

if(root->left != NULL && root->right != NULL){

TreeNode\* pred = in\_predecessor(root);

root->val = pred->val;

root->left = deleteNode(root->left,pred->val);

}

else if(root->val > key){

//move left

root->left = deleteNode(root->left,key);

} { Go to root->left, rearrange and

else { Attach updated node }

//move right

root->right = deleteNode(root->right,key);

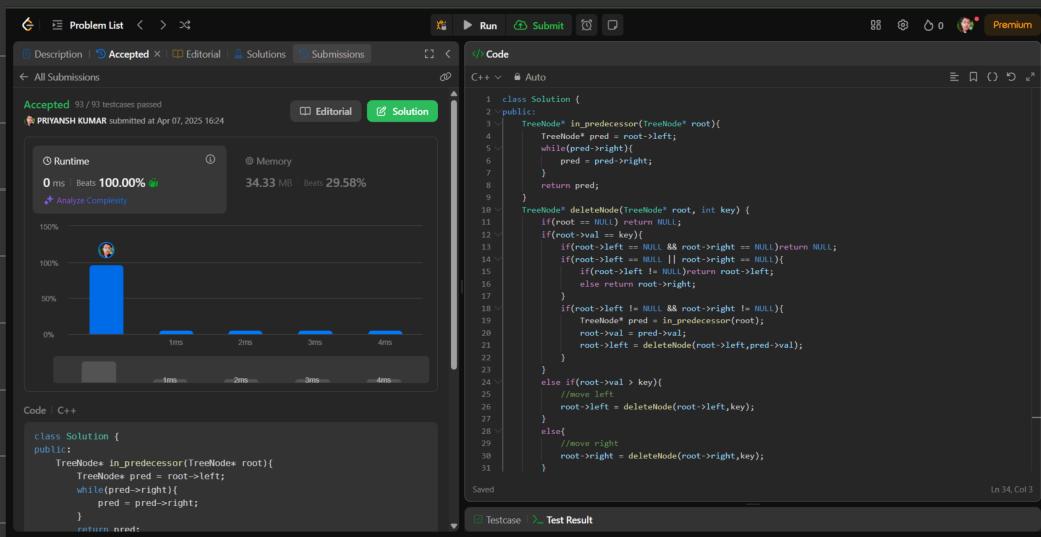
} { Go to root->right rearrange and

else { Attach updated Node }

Made with Goodnotes

Finally return root;

# LEETCODE: 450



The screenshot shows a LeetCode problem page for problem 450. At the top, there's a navigation bar with links for Description, Accepted, Editorial, Solutions, Submissions, Run, Submit, and Premium. Below the navigation is a summary bar indicating the code is Accepted, 93 / 93 testcases passed, submitted by PRIYANSH KUMAR at Apr 07, 2025 16:24. It also shows Runtime (0 ms, 100.00%) and Memory (34.33 MB, 29.50%). A chart titled "Runtime" compares execution times across various users, with the current user's bar reaching nearly 100%. Below the chart is a "Code" section in C++ with syntax highlighting. The code implements two functions: `in_predecessor` and `deleteNode`. The `in_predecessor` function finds the predecessor of a given node. The `deleteNode` function deletes a node from a binary search tree. The code is annotated with comments explaining its logic, such as handling cases where the root is NULL or the node has one child. The bottom right corner of the code editor shows "In 34, Col 3".

```
1 class Solution {
2 public:
3     TreeNode* in_predecessor(TreeNode* root){
4         TreeNode* pred = root->left;
5         while(pred->right){
6             pred = pred->right;
7         }
8         return pred;
9     }
10    TreeNode* deleteNode(TreeNode* root, int key) {
11        if(root == NULL) return NULL;
12        if(root->val == key){
13            if((root->left == NULL && root->right == NULL))return NULL;
14            if((root->left == NULL || root->right == NULL)){
15                if((root->left != NULL))return root->left;
16                else return root->right;
17            }
18            if((root->left != NULL && root->right != NULL)){
19                TreeNode* pred = in_predecessor(root);
20                root->val = pred->val;
21                root->left = deleteNode(root->left,pred->val);
22            }
23        }
24        else if((root->val > key)){
25            //move left
26            root->left = deleteNode(root->left,key);
27        }
28        else{
29            //move right
30            root->right = deleteNode(root->right,key);
31        }
32    }
33 };
34 }
```