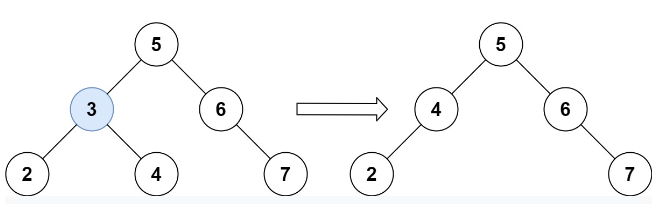
<https://leetcode.com/problems/delete-node-in-a-bst/>

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return *the* ***root node reference*** *(possibly updated) of the BST*.

Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

**Example 1:**



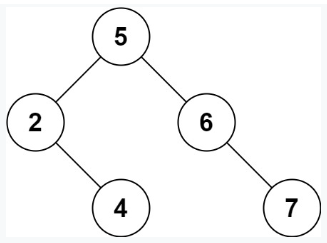
Input: root = [5,3,6,2,4,null,7], key = 3

Output: [5,4,6,2,null,null,7]

Explanation: Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the above BST.

Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.



**Example 2:**

Input: root = [5,3,6,2,4,null,7], key = 0

Output: [5,3,6,2,4,null,7]

Explanation: The tree does not contain a node with value = 0.

**Example 3:**

Input: root = [], key = 0

Output: []

**Constraints:**

* The number of nodes in the tree is in the range [0, 104].
* -105 <= Node.val <= 105
* Each node has a **unique** value.
* root is a valid binary search tree.
* -105 <= key <= 105

**Follow up:** Could you solve it with time complexity O(height of tree)?

**Attempt 1: 2022-10-25**

**Solution 1:  Recursive traversal without removing actual node (60min)**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public TreeNode deleteNode(TreeNode root, int key) {

if(root == null) {

return null;

}

// Recursively find the node that has the same value as the key,

// while setting the left/right nodes equal to the returned subtree

if(root.val > key) {

root.left = deleteNode(root.left, key);

} else if(root.val < key) {

root.right = deleteNode(root.right, key);

} else {

// Once the node is found, have to handle the below 4 cases

// 1.node doesn't have left or right - return null

// 2.node only has left subtree- return the left subtree

// 3.node only has right subtree- return the right subtree

// 4.node has both left and right - find the minimum value in the right subtree,

// set that value to the currently found node, then recursively delete the

// minimum value in the right subtree

if(root.left == null) {

return root.right;

} else if(root.right == null) {

return root.left;

} else {

TreeNode minNode = findMin(root.right);

root.val = minNode.val;

root.right = deleteNode(root.right, minNode.val);

}

}

return root;

}

// Style 1

//private TreeNode findMin(TreeNode node) {

// while(node.left != null) {

// node = node.left;

// }

// return node;

//}

// Style 2

private TreeNode findMin(TreeNode node) {

if(node.left == null) {

return node;

}

return findMin(node.left);

}

}

Time Complexity: O(h) ~ O(n) -> best case ~ worst case

Space Complexity: O(h) ~ O(n) -> best case ~ worst case

**Refer to**

<https://leetcode.com/problems/delete-node-in-a-bst/discuss/93296/Recursive-Easy-to-Understand-Java-Solution>

Steps:

1. Recursively find the node that has the same value as the key, while setting the left/right nodes equal to the returned subtree
2. Once the node is found, have to handle the below 4 cases

* node doesn't have left or right - return null
* node only has left subtree- return the left subtree
* node only has right subtree- return the right subtree
* node has both left and right - find the minimum value in the right subtree, set that value to the currently found node, then recursively delete the minimum value in the right subtree

public TreeNode deleteNode(TreeNode root, int key) {

if(root == null){

return null;

}

if(key < root.val){

root.left = deleteNode(root.left, key);

}else if(key > root.val){

root.right = deleteNode(root.right, key);

}else{

if(root.left == null){

return root.right;

}else if(root.right == null){

return root.left;

}

TreeNode minNode = findMin(root.right);

root.val = minNode.val;

root.right = deleteNode(root.right, root.val);

}

return root;

}

private TreeNode findMin(TreeNode node){

while(node.left != null){

node = node.left;

}

return node;

}

**Refer to**

<https://leetcode.com/problems/delete-node-in-a-bst/discuss/1591176/C%2B%2B-Simple-Solution-w-Images-and-Detailed-Explanation-or-Iterative-and-Recursive-Approach>

Firstly, we need to search for the key valued node in the BST. If it doesn't exist, we can directly return the BST as it is.

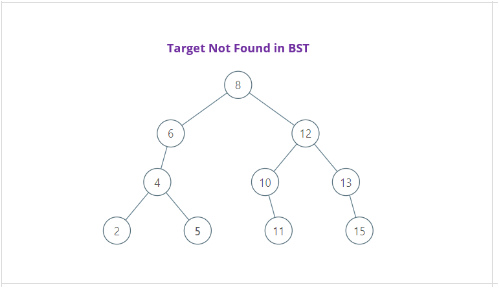
If the node is found in BST, we need to delete it. But depending on where the node occurs, we need to employ different ways to delete it. So, it's helpful to think about the various cases that may occur and see how we can deal with them separately. Let's see the different cases possible (the node to be deleted will be referred as target node **T** below) -

The above logic can also be implemented recursively. For me, I found the iterative version more intuitive which is why I mentioned it first above. But for anyone looking for recursive implementation of it, here goes

**Case 1:**

T not found in BST after the search.

We can directly return root of BST as it is



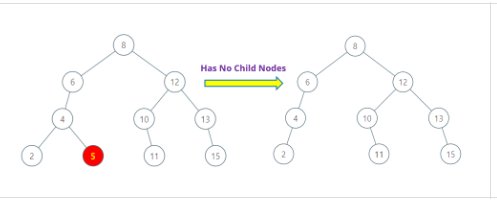
**Case 2:**

T found in the BST.

It has no children which means it is a leaf node.

So, we can delete it just by updating T's parent pointer to null and then deleting T

In example, we simply updated right pointer of 5's parent node-4 as null



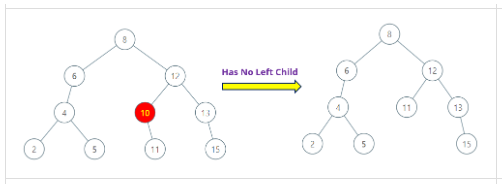
**Case 3:**

T found in BST and only has the right child/subtree

Cant directly delete T as it has its right child that must remain in the BST

we **update child pointer of parent of T to the right child of T**

In example, we updated left child pointer of 12 to point at 11 which is right child of the node to be deleted-10



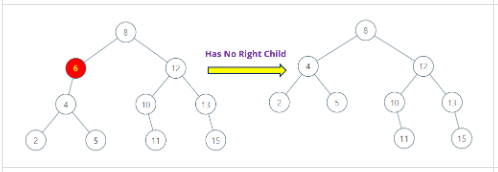
**Case 4:**

T found in BST and only has the left child/subtree. This is similar to previous case

Cant directly delete T bcoz we need to take care of its left child

We **update child pointer of parent of T to the left child of T**

In example, we updated left child pointer of 8 to point at 4 which is left child of node to be deleted-6



**Case 5:**

T found in BST and has both left and right child/subtree

In this case, to maintain BST property, we have two choices -

1. Replace T by largest node in left subtree
2. Replace T by smallest node in right subtree

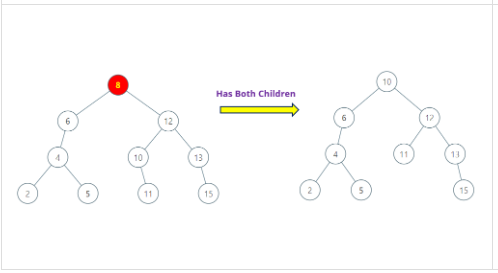
Let's choose 2nd option.

We find smallest node in right subtree by **going right once and then as left as possible**

It is then replaced with node to be deleted. But again, we need to take care of its right child

For this, we **update child pointer of parent of the smallest node to the right child of smallest node**

In image example, we update left child pointer of 12 point to 11 which is child node of smallest node-10



**Test:**

**1.Delete 8 (a value exist in tree, will delete node 8 and re-construct the tree)**

**2.Delete 16 (a value NOT exist in tree, will not delete anything just return raw tree by returning the root)**

public class Test {

public static void main(String[] args) {

/\*\*

8 10

/ \ / \

6 12 Remove 8 6 12

/ / \ ====> / / \

4 10 13 4 11 13

/ \ \ \ / \ \

2 5 11 15 2 5 15

\*/

Test b = new Test();

TreeNode eight = b.new TreeNode(8);

TreeNode six = b.new TreeNode(6);

TreeNode twelve = b.new TreeNode(12);

TreeNode four = b.new TreeNode(4);

TreeNode ten = b.new TreeNode(10);

TreeNode thirteen = b.new TreeNode(13);

TreeNode two = b.new TreeNode(2);

TreeNode five = b.new TreeNode(5);

TreeNode eleven = b.new TreeNode(11);

TreeNode fiftheen = b.new TreeNode(15);

eight.left = six;

eight.right = twelve;

six.left = four;

twelve.left = ten;

twelve.right = thirteen;

four.left = two;

four.right = five;

ten.right = eleven;

thirteen.right = fiftheen;

TreeNode result = b.deleteNode(eight, 8);

// TreeNode result = b.deleteNode(eight, 16);

System.out.println(result);

}

private class TreeNode {

public int val;

public TreeNode left, right;

public TreeNode(int val) {

this.val = val;

this.left = this.right = null;

}

}

public TreeNode deleteNode(TreeNode root, int key) {

if (root == null) {

return null;

}

if (key < root.val) {

root.left = deleteNode(root.left, key);

} else if (key > root.val) {

root.right = deleteNode(root.right, key);

} else {

if (root.left == null) {

return root.right;

} else if (root.right == null) {

return root.left;

}

TreeNode minNode = findMin(root.right);

root.val = minNode.val;

root.right = deleteNode(root.right, root.val);

}

return root;

}

private TreeNode findMin(TreeNode node) {

while (node.left != null) {

node = node.left;

}

return node;

}

}

**Solution 2:  Recursive traversal with removing actual node (360min, too long to figure out how the connection between node can rebuild)**

**Wrong Solution: Missing handling if 'par' == null**

**Test case:**

[5,3,6,2,4,null,7]

3

5 5

/ \ / \

3 6 Remove 3 4 6

/ \ \ ====> / \

2 4 7 2 7

java.lang.NullPointerException: Cannot assign field "left" because "<local3>" is null

at line 67, Solution.deleteNode

at line 24, Solution.deleteNode

at line 54, \_\_DriverSolution\_\_.\_\_helper\_\_

at line 87, \_\_Driver\_\_.main

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public TreeNode deleteNode(TreeNode root, int key) {

if(root == null) {

return null;

}

// Recursively find the node that has the same value as the key,

// while setting the left/right nodes equal to the returned subtree

if(root.val > key) {

root.left = deleteNode(root.left, key);

} else if(root.val < key) {

root.right = deleteNode(root.right, key);

} else {

if(root.left == null) {

return root.right;

} else if(root.right == null) {

return root.left;

} else {

/\*\*

E.g Remove 8

8 10

/ \ / \

6 12 Remove 8 6 12

/ / \ ====> / / \

4 10 13 4 11 13

/ \ \ \ / \ \

2 5 11 15 2 5 15

\*/

// Once the node(=8) is found, we are going to find smallest node in its

// right subtree(=10), strategy is going right once then as left as possible,

// after finding the leftmost node(=10), that would be the new root, since

// current root(node 8, the one's value equal to 'key') will be physically

// removed and its position will be replaced by this leftmost node(=10).

// The new root's left & right connection has to rebuild. Remove its parent

// connection(=12), disconnect its left child connection(null) then reconnect

// to current root's left child(=6), disconnect its right child connection(=11)

// then reconnect to current root's right child(=12)

// To remove new root's parent connection record it first

TreeNode par = null;

// Going right once then as left as possible to get new root

// e.g 'par'(=12) and 'newRoot'(=10)

TreeNode newRoot = root.right;

while(newRoot.left != null) {

par = newRoot;

newRoot = newRoot.left;

}

// First remove 'newRoot' from original tree, then build reconnection by inheriting

// original 'root' left and right child connection

// e.g 'newRoot'(=10) will be removed by connecting 'par'(=12)

// left child to 'newRoot' right child(=11)

par.left = newRoot.right;

// e.g 'newRoot'(=10) will connect original 'root'(=8) left child(=6) and right child(=12)

newRoot.left = root.left;

newRoot.right = root.right;

return newRoot;

}

}

return root;

}

}

**Fix by adding handling of if(par == null) case, when 'newRoot' has no left child, 'newRoot.left' will be null, and while loop logic will not hit and 'par' won't assign value, hence 'par.left = newRoot.right' will throw out NULL point exception**

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode() {}

\* TreeNode(int val) { this.val = val; }

\* TreeNode(int val, TreeNode left, TreeNode right) {

\* this.val = val;

\* this.left = left;

\* this.right = right;

\* }

\* }

\*/

class Solution {

public TreeNode deleteNode(TreeNode root, int key) {

if(root == null) {

return null;

}

// Recursively find the node that has the same value as the key,

// while setting the left/right nodes equal to the returned subtree

if(root.val > key) {

root.left = deleteNode(root.left, key);

} else if(root.val < key) {

root.right = deleteNode(root.right, key);

} else {

if(root.left == null) {

return root.right;

} else if(root.right == null) {

return root.left;

} else {

/\*\*

E.g Remove 8

8 10

/ \ / \

6 12 Remove 8 6 12

/ / \ ====> / / \

4 10 13 4 11 13

/ \ \ \ / \ \

2 5 11 15 2 5 15

\*/

// Once the node(=8) is found, we are going to find smallest node in its

// right subtree(=10), strategy is going right once then as left as possible,

// after finding the leftmost node(=10), that would be the new root, since

// current root(node 8, the one's value equal to 'key') will be physically

// removed and its position will be replaced by this leftmost node(=10).

// The new root's left & right connection has to rebuild. Remove its parent

// connection(=12), disconnect its left child connection(null) then reconnect

// to current root's left child(=6), disconnect its right child connection(=11)

// then reconnect to current root's right child(=12)

// To remove new root's parent connection record it first

TreeNode par = null;

// Going right once then as left as possible to get new root

// e.g 'par'(=12) and 'newRoot'(=10)

TreeNode newRoot = root.right;

while(newRoot.left != null) {

par = newRoot;

newRoot = newRoot.left;

}

/\*\*

E.g Remove 3

5 5

/ \ / \

3 6 Remove 3 4 6

/ \ \ ====> / \

2 4 7 2 7

\*/

// Fix by adding handling of if(par == null) case, when 'newRoot' has no left child,

// 'newRoot.left' will be null, and while loop logic will not hit and 'par' won't

// assign value, hence 'par.left = newRoot.right' will throw out NULL point exception

if(par == null) {

newRoot.left = root.left;

return newRoot;

}

// First remove 'newRoot' from original tree, then build reconnection by inheriting

// original 'root' left and right child connection

// e.g 'newRoot'(=10) will be removed by connecting 'par'(=12)

// left child to 'newRoot' right child(=11)

par.left = newRoot.right;

// e.g 'newRoot'(=10) will connect original 'root'(=8) left child(=6) and right child(=12)

newRoot.left = root.left;

newRoot.right = root.right;

return newRoot;

}

}

return root;

}

}

Time Complexity: O(h) ~ O(n) -> best case ~ worst case

Space Complexity: O(h) ~ O(n) -> best case ~ worst case

**Test:**

public class Test {

public static void main(String[] args) {

/\*\*

5 5

/ \ / \

3 6 Remove 3 4 6

/ \ \ ====> / \

2 4 7 2 7

\*/

Test b = new Test();

TreeNode six = b.new TreeNode(6);

TreeNode four = b.new TreeNode(4);

TreeNode two = b.new TreeNode(2);

TreeNode five = b.new TreeNode(5);

TreeNode three = b.new TreeNode(3);

TreeNode seven = b.new TreeNode(7);

five.left = three;

five.right = six;

six.right = seven;

three.left = two;

three.right = four;

TreeNode result = b.deleteNode(five, 3);

System.out.println(result);

}

private class TreeNode {

public int val;

public TreeNode left, right;

public TreeNode(int val) {

this.val = val;

this.left = this.right = null;

}

}

/\*\*

5 5

/ \ / \

3 6 Remove 3 4 6

/ \ \ ====> / \

2 4 7 2 7

\*/

public TreeNode deleteNode(TreeNode root, int key) {

if (root == null) {

return null;

}

// Recursively find the node that has the same value as the key,

// while setting the left/right nodes equal to the returned subtree

if (root.val > key) {

root.left = deleteNode(root.left, key);

} else if (root.val < key) {

root.right = deleteNode(root.right, key);

} else {

TreeNode par = null;

TreeNode newRoot = root.right;

while (newRoot.left != null) {

par = newRoot;

newRoot = newRoot.left;

}

//if(par == null) {

// newRoot.left = root.left;

// return newRoot;

//}

par.left = newRoot.right;

newRoot.left = root.left;

newRoot.right = root.right;

return newRoot;

}

return root;

}

}

**Refer to**

<https://leetcode.com/problems/delete-node-in-a-bst/discuss/93296/Recursive-Easy-to-Understand-Java-Solution/97771>

I did not come up with the smart idea of editing node value instead of reconnecting node, so here is my code for node reconnecting way. It shares similar time performance as the recursive way.

public TreeNode deleteNode(TreeNode root, int key) {

if(root == null) return root;

if(root.val < key) root.right = deleteNode(root.right, key);

else if(root.val > key) root.left = deleteNode(root.left, key);

else{

if(root.left == null) return root.right;

else if(root.right == null) return root.left;

else{

TreeNode newRoot = root.right, par = null;

while(newRoot.left != null){

par = newRoot;

newRoot = newRoot.left;

}

if(par == null){

newRoot.left = root.left;

return newRoot;

}

par.left = newRoot.right;

newRoot.left = root.left;

newRoot.right = root.right;

return newRoot;

}

}

return root;

}