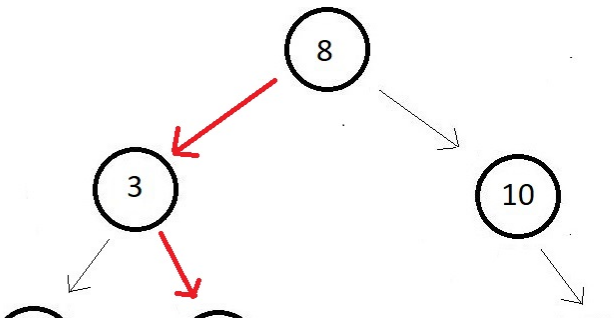
Show Course Contents

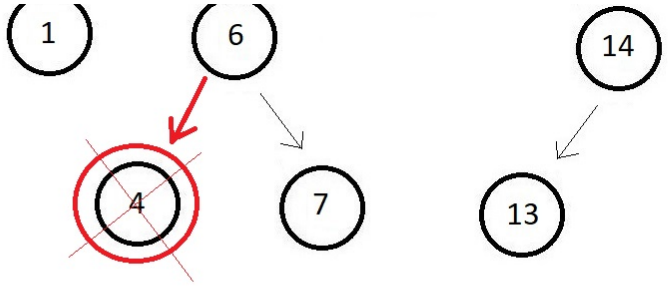
Overview Q&A Downloads Announcements

Deletion in a Binary Search Tree

In the last lecture, we saw insertion in a binary search tree. We learned how to program the insertion function in C. We drew similarities between the insertion and the search operation making it easy to digest. Today, we'll learn our third operation, the deletion operation in a binary search tree.

You might be wondering how big a deal is deleting some nodes in a binary search tree. Well, for cases like the one I have drawn below, where you'll be asked to remove say element 4 seems quite an easy job. You'll just search for the element and remove it.





But deleting in a binary search tree is no doubt a tough job like if you consider a case where the node, we'll have to remove might not be a leaf node or the node is the root node. But you should not be the one worrying here. I have some easy way out. We'll explore them in detail now.

So, whenever we talk about deleting a node from binary search tree, we have the following three cases in mind:

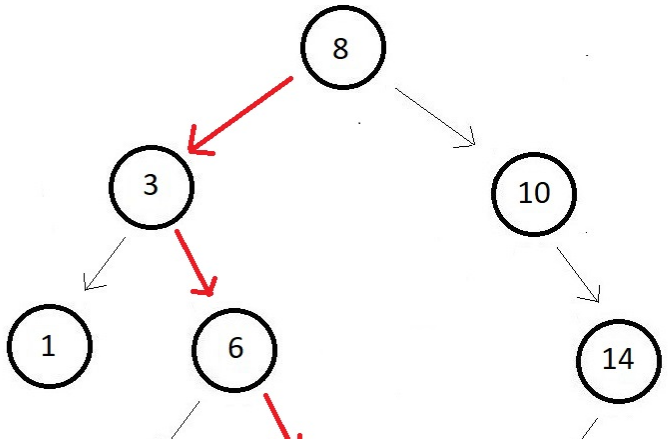
1. The node is a leaf node.
2. The node is a non-leaf node.
3. The node is the root node.

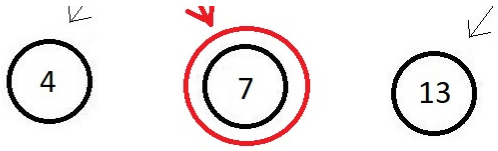
Let's deal with each of these in detail, starting with deleting the leaf node.

Deleting a leaf node:

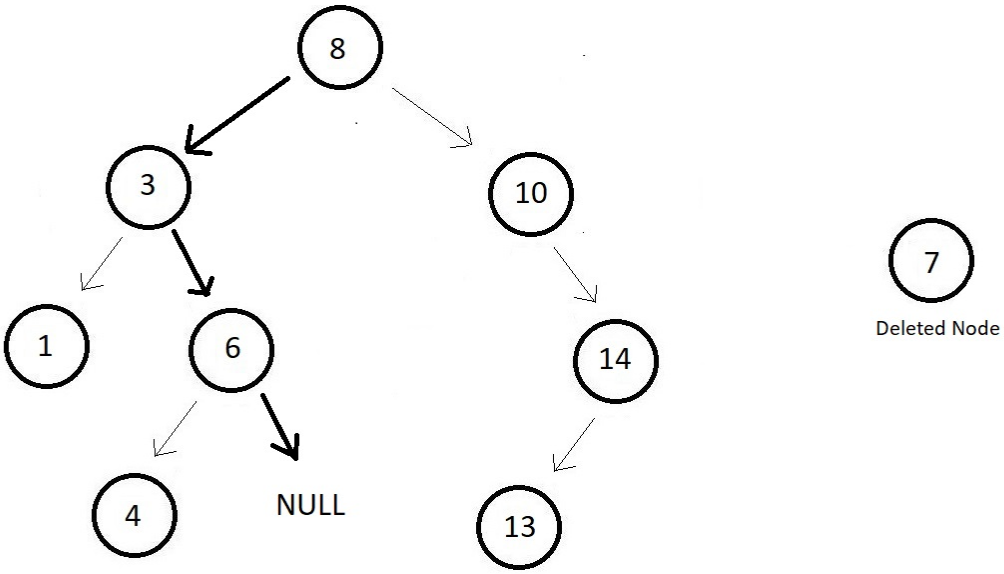
Deleting a leaf node is the simplest case in deletion in binary search trees where the only thing you have to do is to search the element in the tree, and remove it from the tree, and make its parent node point to NULL. To be more specific, follow the steps below to delete a leaf node along with the illustrations of how we delete a leaf node in the above tree:

1. Search the node.





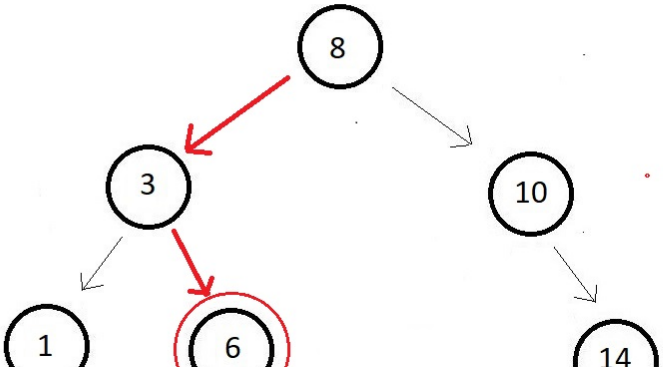
2. Delete the node.

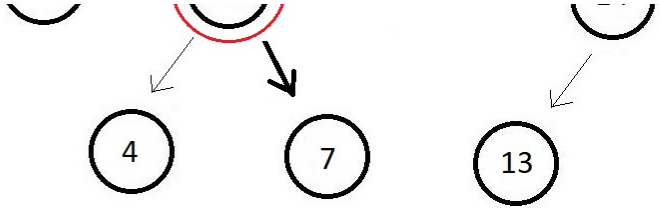


Deleting a non-leaf node:

Now suppose the node is not a leaf node, so you cannot just make its parent point to NULL, and get away with it. You have to even deal with the children of this node. Let's try deleting node 6 in the above binary search tree.

1. So, the first thing you would do is to search element 6.



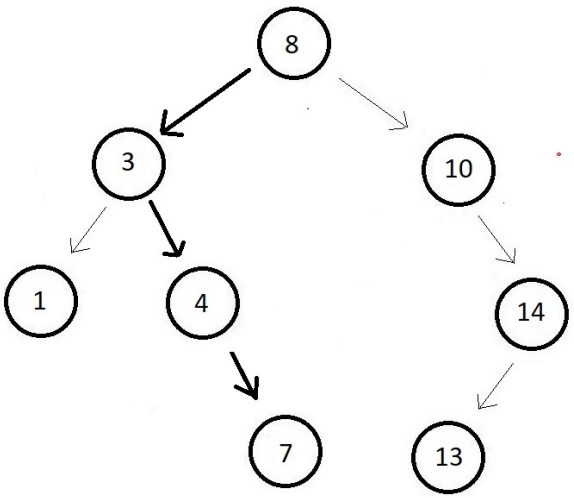


2. Now the dilemma is, which node will replace the position of node 6. Well, there is a simple answer to it. It says, when you delete a node that is not a leaf node, you replace its position with its **InOrder predecessor or Inorder successor.**

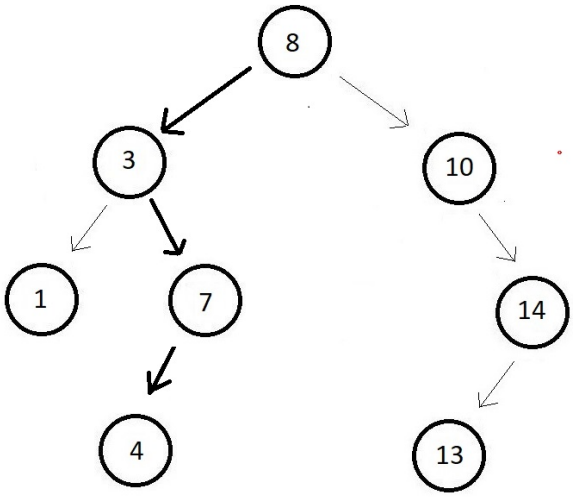
So, what does that mean? It means that if you write the InOrder traversal of the above tree, which I have already taught in my previous lectures, the nodes coming immediately before or after node 6, will be the one replacing it. So, if you write the InOrder traversal of the tree, you will get:

1→ 3→ 4 →**6** →7→ 8→ 10→ 13→ 14

So, the InOrder predecessor and the Inorder successor of node 6 are 4 and 7 respectively. Hence you can substitute node 6 with any of these nodes, and the tree will still be a valid binary search tree. Refer to how it looks below.



(i)



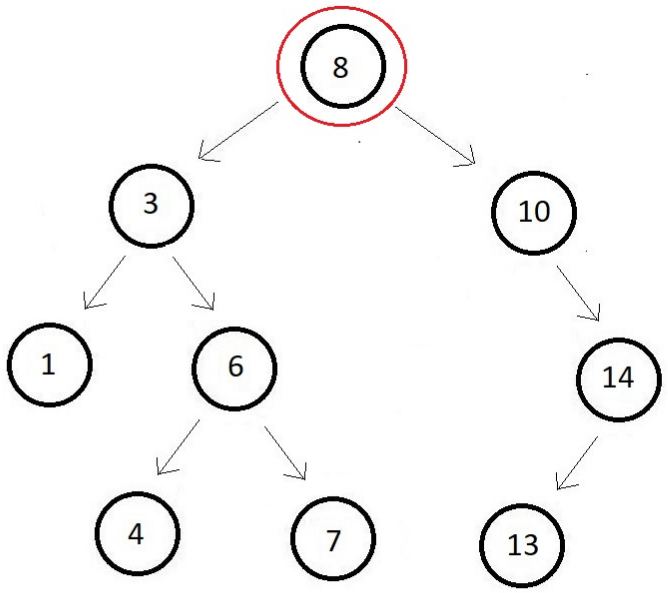
(ii)

So, both are still binary search trees. In the first case, we replaced node 6 with node 4. And the right subtree of node 4 is 7, which is still bigger than it. And in the second case, we replaced node 6 with node 7. And the left subtree of node 7 is 4,

which is still smaller than the node. Hence, a win-win for us.

Deleting the root node:

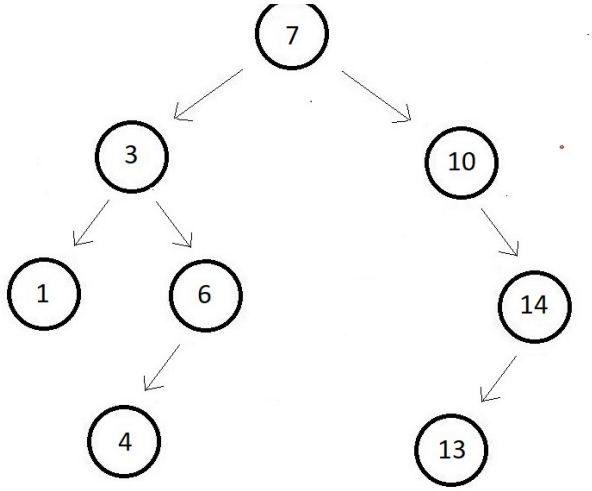
1. Now, if you carefully observe, the root node is still another non-leaf node. So, the basics to delete the root node remains the same as what we did for a general non-leaf node. But since the root node holds a big size of subtrees along with, we have put this as a separate case here.



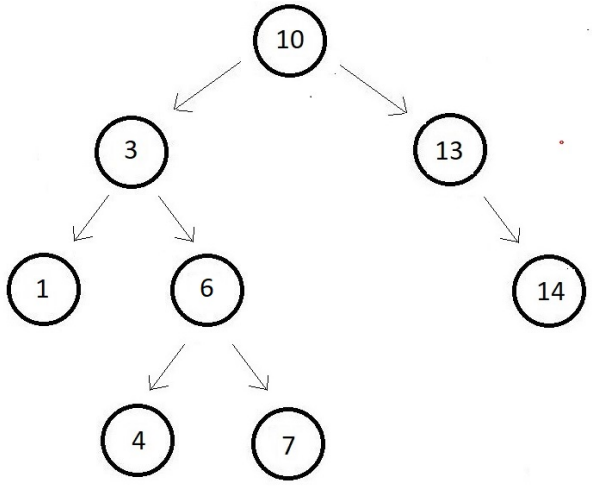
2. So, the first thing you do is write the InOrder traversal of the whole tree. And then replace the position of the root node with its **InOrder predecessor or Inorder successor.** So, here the traversal order is,

1→ 3→ 4 →6 →7→ **8**→ 10→ 13→ 14

So, the InOrder predecessor and the Inorder successor of the root node 8 are 7 and 10 respectively. Hence you can substitute node 8 with any of these nodes, but there is a catch here. So, if you substitute the root node here, with its InOrder predecessor 7, the tree will still be a binary search tree, but when you substitute the root node here, with its InOrder successor 10, there still becomes an empty position where node 10 used to be. So, we still placed the InOrder successor of 10, which was 13 on the position where 10 used to be. And then there are no empty nodes in between. This finalizes our deletion.



(i)



(ii)

So, there are a few steps:

1. First, search for the node to be deleted.
2. Search for the InOrder Predecessor and Successor of the node.
3. Keep doing that until the tree has no empty nodes.

And this case is not limited to the root nodes, rather any nodes falling in between a tree. Well, there could be a case where the node was not found in the tree, so, for that, we would revert the statement that the node could not be found.

So, that was all we had. We saw all three cases in detail. We applied all our previous knowledge of the traversal series to help ourselves here. Next, we will see the programming part of the deletion operation. Practice on your own BST. Try deleting nodes in between the trees.

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial where we'll learn the programming implementation of deletion operation in C. Till then keep coding.

Previous Next

CodeWithHarry