

89

```

1  AVLTreeUpdateHeight(node) {
2      leftHeight = -1
3      if (node -->left != null)
4          leftHeight = node -->left -->height
5      rightHeight = -1
6      if (node -->right != null)
7          rightHeight = node -->right -->height
8      node -->height = max(leftHeight, rightHeight) + 1
9  }
10
11      89      "left"      null
12  AVLTreeSetChild(parent, whichChild, child) {
13      if (whichChild != "left" && whichChild != "right")
14          return false
15
16      if (whichChild == "left")
17          parent -->left = child
18      else
19          parent -->right = child
20      if (child != null)
21          child -->parent = parent
22          89
23      AVLTreeUpdateHeight(parent)
24      return true
25  }
26
27      89      73      null
28  AVLTreeReplaceChild(parent, currentChild, newChild) {
29      if (parent -->left == currentChild)
30          return AVLTreeSetChild(parent, "left", newChild)
31      else if (parent -->right == currentChild)
32          return AVLTreeSetChild(parent, "right", newChild)
33      return false
34  }
35
36
37  AVLTreeGetBalance(node) {
38      leftHeight = -1
39      if (node -->left != null)
40          leftHeight = node -->left -->height
41      rightHeight = -1
42      if (node -->right != null)
43          rightHeight = node -->right -->height
44      return leftHeight - rightHeight
45  }
46

```

height = 0

height = 1

this sets 73 to null

```

1  AVLTreeRemoveNode(tree, node) {
2      if (node == null) {
3          return false
4      }
5
6      // Parent needed for rebalancing
7      parent = node-->parent
8
9      // Case 1: Internal node with 2 children
10     if (node-->left != null && node-->right != null) {
11         // Find successor
12         succNode = node-->right
13         while (succNode-->left != null) {
14             succNode = succNode-->left
15         }
16
17         // Copy the key from the successor node
18         node-->key = succNode-->key
19
20         // Recursively remove successor
21         AVLTreeRemoveNode(tree, succNode)
22
23         // Nothing left to do since the recursive call will have rebalanced
24         return true
25     }
26
27     // Case 2: Root node (with 1 or 0 children)
28     else if (node == tree-->root) {
29         if (node-->left != null) {
30             tree-->root = node-->left
31         }
32         else {
33             tree-->root = node-->right
34         }
35         if (tree-->root != null) {
36             tree-->root-->parent = null
37         }
38         return true
39     }
40
41     // Case 3: Internal with left child only
42     else if (node-->left != null) {
43         AVLTreeReplaceChild(parent, node, node-->left)
44     }
45
46     // Case 4: Internal with right child only OR leaf
47     else {
48         AVLTreeReplaceChild(parent, node, node-->right)
49     }
50
51     // node is gone. Anything that was below node that has persisted is already correctly
52     // balanced, but ancestors of node may need rebalancing.
53     node = parent
54     while (node != null) {
55         AVLTreeRebalance(tree, node)
56         node = node-->parent
57     }
58     return true
59 }

```

```

1  AVLTreeRebalance(tree, node) {
2      AVLTreeUpdateHeight(node)
3      if (AVLTreeGetBalance(node) == -2) {
4          if (AVLTreeGetBalance(node-->right) == 1) {
5              // Double rotation case.
6              AVLTreeRotateRight(tree, node-->right)
7          }
8          return AVLTreeRotateLeft(tree, node)
9      }
10     else if (AVLTreeGetBalance(node) == 2) {
11         if (AVLTreeGetBalance(node-->left) == -1) {
12             // Double rotation case.
13             AVLTreeRotateLeft(tree, node-->left)
14         }
15         return AVLTreeRotateRight(tree, node)
16     }
17     return node
18 }

```

70

```

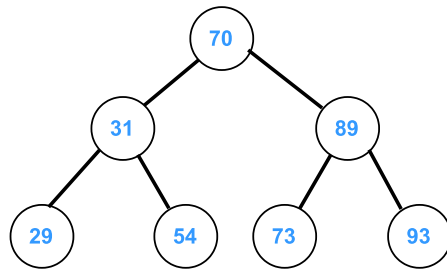
1  AVLTreeRemoveNode(tree, node) {
2      if (node == null) {
3          return false
4      }
5
6      // Parent needed for rebalancing
7      parent = node-->parent
8
9      // Case 1: Internal node with 2 children
10     if (node-->left != null && node-->right != null) {
11         // Find successor
12         succNode = node-->right
13         while (succNode-->left != null) {
14             succNode = succNode-->left
15         }
16
17         // Copy the key from the successor node
18         node-->key = succNode-->key
19
20         // Recursively remove successor
21         AVLTreeRemoveNode(tree, succNode)
22
23         // Nothing left to do since the recursive call will have rebalanced
24         return true
25     }
26
27     // Case 2: Root node (with 1 or 0 children)
28     else if (node == tree-->root) {
29         if (node-->left != null) {
30             tree-->root = node-->left
31         }
32         else {
33             tree-->root = node-->right
34         }
35         if (tree-->root != null) {
36             tree-->root-->parent = null
37         }
38         return true
39     }
40
41     // Case 3: Internal with left child only
42     else if (node-->left != null) {
43         AVLTreeReplaceChild(parent, node, node-->left)
44     }
45     // Case 4: Internal with right child only OR leaf
46     else {
47         AVLTreeReplaceChild(parent, node, node-->right)
48     }
49
50     // node is gone. Anything that was below node that has persisted is already correctly
51     // balanced, but ancestors of node may need rebalancing.
52     node = parent
53     while (node != null) {
54         AVLTreeRebalance(tree, node)
55         node = node-->parent
56     }
57     return true
58 }

```

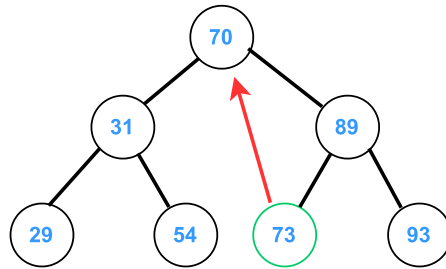
node : 70
node --> right : 89
node --> left : 31

succNode : 89
node --> right : 89
if succNode --> left != null then

Remove 70



Find 73



Find 73

