

Deletion in a Binary Search Tree (Implementation)

We will now write the implementation of the deletion function which covers all the cases that we discussed previously.

We'll cover the following



- Introduction
 - 1. Deleting an Empty Tree
- Searching for val
- Traversing
- When val Is Found
 - Deleting a Leaf Node
 - Deleting a Node with a Right Child Only
 - Deleting a Node with a Left Child Only
 - Deleting a Node with Two Children
- Putting It All Together
- Quick Quiz!

Introduction

Let's implement the delete function for BSTs. We'll build upon the code as we cater for each case.

Also, note that the `delete` function in the `BinarySearchTree` class is simply calling the `delete` function in the `Node` class where the core of our implementation will reside.

1. Deleting an Empty Tree

Let's start with a skeleton function definition and cater for the first case. If the root does not exist, we return `False` in the `BinarySearchTree` class.

main.py

BinarySearchTree.py

Node.py

```
1 from Node import Node
2 from BinarySearchTree import BinarySearchTree
3
4 import random
5
6
```

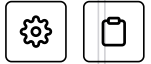


```
7 def display(node):
8     lines, _, _ = _display_aux(node)
9     for line in lines:
10         print(line)
11
12
13 def _display_aux(node):
14     """
15     Returns list of strings, width, height,
16     and horizontal coordinate of the root.
17     """
18     # None.
19     if node is None:
20         line = 'Empty tree!'
21         width = len(line)
22         height = 1
23         middle = width // 2
24         return [line], width, height, middle
25
26     # No child.
27     if node.rightChild is None and node.leftChild is None:
28         line = str(node.val)
29         width = len(line)
30         height = 1
31         middle = width // 2
32         return [line], width, height, middle
33
34     # Only left child.
35     if node.rightChild is None:
36         lines, n, p, x = _display_aux(node.leftChild)
37         s = str(node.val)
38         u = len(s)
39         first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
40         second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
41         shifted_lines = [line + u * ' ' for line in lines]
42         final_lines = [first_line, second_line] + shifted_lines
43         return final_lines, n + u, p + 2, n + u // 2
44
45     # Only right child.
46     if node.leftChild is None:
47         lines, n, p, x = _display_aux(node.rightChild)
48         s = str(node.val)
49         u = len(s)
50         # first_line = s + x * '_' + (n - x) * ' '
51         first_line = s + x * '_' + (n - x) * ' '
52         second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
53         shifted_lines = [u * ' ' + line for line in lines]
54         final_lines = [first_line, second_line] + shifted_lines
55         return final_lines, n + u, p + 2, u // 2
56
57     # Two children.
58     left, n, p, x = _display_aux(node.leftChild)
59     right, m, q, y = _display_aux(node.rightChild)
60     s = '%s' % node.val
61     u = len(s)
62     first_line = (x + 1) * ' ' + (n - x - 1) * \
63         '_' + s + y * '_' + (m - y) * ' '
64     second_line = x * ' ' + '/' + \
65         (n - x - 1 + u + y) * ' ' + '\\' + (m - y - 1) * ' '
66     if p < q:
67         left += [n * ' '] * (q - p)
68     elif q < p:
69         right += [m * ' '] * (p - q)
70     zipped_lines = zip(left, right)
71     lines = [first_line, second_line] + \
72         [a + u * ' ' + b for a, b in zipped_lines]
73     return lines, n + m + u, max(p, q) + 2, n + u // 2
74
```

```

75
76 BST = BinarySearchTree(50)
77 print("tree:")
78 display(BST.root)
79
80 BST.delete(50)
81 BST.delete(50) # Deleting in an empty tree
82 display(BST.root)
83

```



Searching for val

We'll now build up to the code for deleting in a BST. We've put it together in a runnable code playground at the end of the lesson!

Here's a snippet of the delete function. It now has some logic to **search for val**. Depending on the value of the node to be deleted, it will move on to the left or right subtree. If the value is not less than or greater than the value of the current node, that means it is equal to the current node which is what the else on **line 6** is for.

```

1 def delete(self, val):
2     if val < self.val: # val is in the left subtree
3         pass
4     elif val > self.val: # val is in the right subtree
5         pass
6     else: # val was found
7         pass
8

```



Traversing

We've now added some logic to traverse on to the relevant sub-trees. To search for `val` in the left sub-tree for instance, we simply recursively call `delete` on the left sub-tree (if the `leftChild` exists!). Otherwise, we've reached the end of our search and `val` was not found. The case for the right child is handled similarly.

```

1 def delete(self, val):
2     if val < self.val: # val is in the left subtree
3         if(self.leftChild):
4             self.leftChild = self.leftChild.delete(val)
5         else:
6             print(str(val) + " not found in the tree")
7             return None
8     elif val > self.val: # val is in the right subtree
9         if(self.rightChild):
10            self.rightChild = self.rightChild.delete(val)
11        else:
12            print(str(val) + " not found in the tree")
13            return None
14    else: # val was found
15        pass
16

```



When val Is Found

There can be a number of cases if the value to be deleted is found. Namely, the value to be deleted exists in a,

1. Leaf node
2. Node with a right child
3. Node with a left child
4. Node with 2 children

Let's write the code for each condition.

Deleting a Leaf Node

Once the node is found, we test to see if it is a leaf node, i.e., if both the left and right children of the node are `None`. We then delete the leaf node by making the leaf node's parent's left or right child equal to `None` by returning `None`.

```
1 def delete(self, val):
2     if val < self.val: # val is in the left subtree
3         if(self.leftChild):
4             self.leftChild = self.leftChild.delete(val)
5         else:
6             print(str(val) + " not found in the tree")
7             return None
8     elif val > self.val: # val is in the right subtree
9         if(self.rightChild):
10            self.rightChild = self.rightChild.delete(val)
11        else:
12            print(str(val) + " not found in the tree")
13            return None
14    else: # val was found
15        # deleting node with no children
16        if self.leftChild is None and self.rightChild is None:
17            self = None
18            return None
19
```



Deleting a Node with a Right Child Only

If the node has one right child only, we replace its node with its right child by returning it (remember the recursive calls set the parent equal to what will be returned by the function!)

```
1 def delete(self, val):
2     if val < self.val: # val is in the left subtree
3         if(self.leftChild):
4             self.leftChild = self.leftChild.delete(val)
5         else:
6             print(str(val) + " not found in the tree")
7             return None
8     elif val > self.val: # val is in the right subtree
```



```

9         if(self.rightChild):
10             self.rightChild = self.rightChild.delete(val)
11         else:
12             print(str(val) + " not found in the tree")
13             return None
14     else: # val was found
15         # deleting node with no children
16         if self.leftChild is None and self.rightChild is None:
17             self = None
18             return None
19         # deleting node with right child
20         elif self.leftChild is None:
21             tmp = self.rightChild
22             self = None
23             return tmp
24

```



Deleting a Node with a Left Child Only

This is handled similarly to deleting a node with a right child only.

```

1  def delete(self, val):
2      if val < self.val: # val is in the left subtree
3          if(self.leftChild):
4              self.leftChild = self.leftChild.delete(val)
5          else:
6              print(str(val) + " not found in the tree")
7              return None
8      elif val > self.val: # val is in the right subtree
9          if(self.rightChild):
10             self.rightChild = self.rightChild.delete(val)
11          else:
12             print(str(val) + " not found in the tree")
13             return None
14      else: # val was found
15          # deleting node with no children
16          if self.leftChild is None and self.rightChild is None:
17              self = None
18              return None
19          # deleting node with right child
20          elif self.leftChild is None:
21              tmp = self.rightChild
22              self = None
23              return tmp
24          # deleting node with left child
25          elif self.rightChild is None:
26              tmp = self.leftChild
27              self = None
28              return tmp
29

```



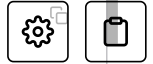
Deleting a Node with Two Children

If a node has two children and is to be deleted, it is replaced by its **inorder successor** i.e., the next node in order. To find the inorder successor, we traverse to the node with the smallest value (left-most) node in the right sub-tree of the node. The inorder successor is then deleted.

```

10         self.rightChild = self.rightChild.delete(val)
11     else:
12         print(str(val) + " not found in the tree")
13         return None
14     else: # val was found
15         # deleting node with no children
16         if self.leftChild is None and self.rightChild is None:
17             self = None
18             return None
19         # deleting node with right child
20         elif self.leftChild is None:
21             tmp = self.rightChild
22             self = None
23             return tmp
24         # deleting node with right child
25         elif self.leftChild is None:
26             tmp = self.rightChild
27             self = None
28             return tmp
29         # deleting a node with two children
30         else:
31             # first get the inorder successor
32             current = self.rightChild
33             # loop down to find the leftmost leaf
34             while(current.leftChild is not None):
35                 current = current.leftChild
36             self.val = current.val
37             self.rightChild = self.rightChild.delete(current.val)
38
39     return self
40

```



Putting It All Together

Here's the final source code. Try experimenting with it!

main.py

BinarySearchTree.py

Node.py

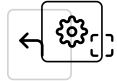
```

1  from Node import Node
2  from BinarySearchTree import BinarySearchTree
3  import random
4
5
6  def display(node):
7      lines, _, _, _ = _display_aux(node)
8      for line in lines:
9          print(line)
10
11
12  def _display_aux(node):
13      """
14      Returns list of strings, width, height,
15      and horizontal coordinate of the root.
16      """
17      # None.
18      if node is None:
19          line = 'Empty tree!'
20          width = len(line)
21          height = 1
22          middle = width // 2
23          return [line], width, height, middle
24

```



```
24
25     # No child.
26     if node.rightChild is None and node.leftChild is None:
27         line = str(node.val)
28         width = len(line)
29         height = 1
30         middle = width // 2
31         return [line], width, height, middle
32
33     # Only left child.
34     if node.rightChild is None:
35         lines, n, p, x = _display_aux(node.leftChild)
36         s = str(node.val)
37         u = len(s)
38         first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
39         second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
40         shifted_lines = [line + u * ' ' for line in lines]
41         final_lines = [first_line, second_line] + shifted_lines
42         return final_lines, n + u, p + 2, n + u // 2
43
44     # Only right child.
45     if node.leftChild is None:
46         lines, n, p, x = _display_aux(node.rightChild)
47         s = str(node.val)
48         u = len(s)
49     #     first_line = s + x * '_' + (n - x) * ' '
50     first_line = s + x * '_' + (n - x) * ' '
51     second_line = (u + x) * ' ' + '\\ ' + (n - x - 1) * ' '
52     shifted_lines = [u * ' ' + line for line in lines]
53     final_lines = [first_line, second_line] + shifted_lines
54     return final_lines, n + u, p + 2, u // 2
55
56     # Two children.
57     left, n, p, x = _display_aux(node.leftChild)
58     right, m, q, y = _display_aux(node.rightChild)
59     s = '%s' % node.val
60     u = len(s)
61     first_line = (x + 1) * ' ' + (n - x - 1) * \
62         '_' + s + y * '_' + (m - y) * ' '
63     second_line = x * ' ' + '/' + \
64         (n - x - 1 + u + y) * ' ' + '\\ ' + (m - y - 1) * ' '
65     if p < q:
66         left += [n * ' '] * (q - p)
67     elif q < p:
68         right += [m * ' '] * (p - q)
69     zipped_lines = zip(left, right)
70     lines = [first_line, second_line] + \
71         [a + u * ' ' + b for a, b in zipped_lines]
72     return lines, n + m + u, max(p, q) + 2, n + u // 2
73
74
75 BST = BinarySearchTree(6)
76 BST.insert(3)
77 BST.insert(2)
78 BST.insert(4)
79 BST.insert(-1)
80 BST.insert(1)
81 BST.insert(-2)
82 BST.insert(8)
83 BST.insert(7)
84
85 print("before deletion:")
86 display(BST.root)
87
88 BST.delete(10)
89 print("after deletion:")
90 display(BST.root)
91
```



Quick Quiz!

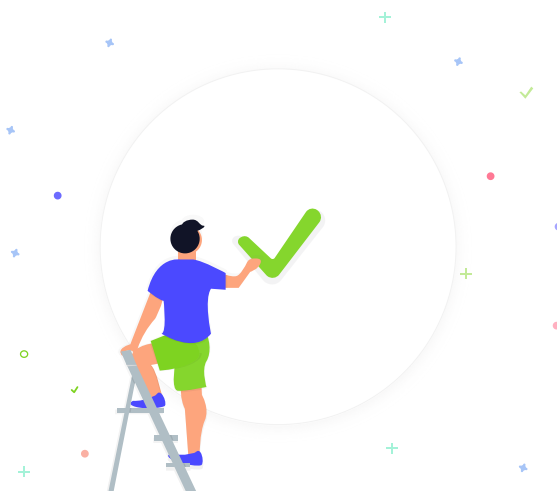
BST delete Quiz

Q  In the delete function, why are we only looking in the right-subtree for the smallest value in the node-with-two-children case?

Your Answer

- ☒ A) Because that node is one of the nodes that can replace the node to be deleted and still keep the BST properties
- ☐ B) Because iterating Right-Subtree takes more operations and thus more time complexity
- ☐ C) It's easier to implement a function to iterate left-subtree rather than right subtree

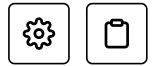
Great, you got it right!



Retake Quiz

So far, we have gone through basic topics on BSTs and then we studied and implemented BST Insertion and Deletion. In the next three lessons, we will cover some basic traversal strategies used in trees.

used in trees.



← Back

Next →

Deletion in a Binary Search Tree

Pre-Order Traversal

Mark as Completed



Report an
Issue



Ask a Question

(https://discuss.educative.io/tag/deletion-in-a-binary-search-tree-implementation__introduction-to-trees__data-structures-for-coding-interviews-in-python)

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.