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

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

```
from Node import Node
from BinarySearchTree import BinarySearchTree

import random


```

```
def display(node):
    lines, _, _, _ = _display_aux(node)
    for line in lines:
        print(line)


def _display_aux(node):
    """
    Returns list of strings, width, height,
    and horizontal coordinate of the root.
    """
    # None.
    if node is None:
        line = 'Empty tree!'
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    # No child.
    if node.rightChild is None and node.leftChild is None:
        line = str(node.val)
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    # Only left child.
    if node.rightChild is None:
        lines, n, p, x = _display_aux(node.leftChild)
        s = str(node.val)
        u = len(s)
        first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
        second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
        shifted_lines = [line + u * ' ' for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, n + u // 2

    # Only right child.
    if node.leftChild is None:
        lines, n, p, x = _display_aux(node.rightChild)
        s = str(node.val)
        u = len(s)
#         first_line = s + x * '_' + (n - x) * ' '
        first_line = s + x * '_' + (n - x) * ' '
        second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
        shifted_lines = [u * ' ' + line for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, u // 2

    # Two children.
    left, n, p, x = _display_aux(node.leftChild)
    right, m, q, y = _display_aux(node.rightChild)
    s = '%s' % node.val
    u = len(s)
    first_line = (x + 1) * ' ' + (n - x - 1) * \
        '_' + s + y * '_' + (m - y) * ' '
    second_line = x * ' ' + '/' + \
        (n - x - 1 + u + y) * ' ' + '\\' + (m - y - 1) * ' '
    if p < q:
        left += [n * ' '] * (q - p)
    elif q < p:
        right += [m * ' '] * (p - q)
    zipped_lines = zip(left, right)
    lines = [first_line, second_line] + \
        [a + u * ' ' + b for a, b in zipped_lines]
    return lines, n + m + u, max(p, q) + 2, n + u // 2

```

```
BST = BinarySearchTree(50)
print("tree:")
display(BST.root)

BST.delete(50)
BST.delete(50)  # Deleting in an empty tree
display(BST.root)
```

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.

```
def delete(self, val):
    if val < self.val:  # val is in the left subtree
        if(self.leftChild):
            self.leftChild = self.leftChild.delete(val)
        else:
            print(str(val) + " not found in the tree")
            return None
    elif val > self.val:  # val is in the right subtree
        if(self.rightChild):
            self.rightChild = self.rightChild.delete(val)
        else:
            print(str(val) + " not found in the tree")
            return None
    else:  # val was found
        pass
```

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`.

```
def delete(self, val):
    if val < self.val:  # val is in the left subtree
        if(self.leftChild):
            self.leftChild = self.leftChild.delete(val)
        else:
            print(str(val) + " not found in the tree")
            return None
    elif val > self.val:  # val is in the right subtree
        if(self.rightChild):
            self.rightChild = self.rightChild.delete(val)
        else:
            print(str(val) + " not found in the tree")
            return None
    else:  # val was found
        # deleting node with no children
        if self.leftChild is None and self.rightChild is None:
            self = None
            return None

```

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!)

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

```
        if(self.rightChild):
            self.rightChild = self.rightChild.delete(val)
        else:
            print(str(val) + " not found in the tree")
            return None
    else:  # val was found
        # deleting node with no children
        if self.leftChild is None and self.rightChild is None:
            self = None
            return None
        # deleting node with right child
        elif self.leftChild is None:
            tmp = self.rightChild
            self = None
            return tmp

```

Deleting a Node with a Left Child Only

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

```
def delete(self, val):
    if val < self.val:  # val is in the left subtree
        if(self.leftChild):
            self.leftChild = self.leftChild.delete(val)
        else:
            print(str(val) + " not found in the tree")
            return None
    elif val > self.val:  # val is in the right subtree
        if(self.rightChild):
            self.rightChild = self.rightChild.delete(val)
        else:
            print(str(val) + " not found in the tree")
            return None
    else:  # val was found
        # deleting node with no children
        if self.leftChild is None and self.rightChild is None:
            self = None
            return None
        # deleting node with right child
        elif self.leftChild is None:
            tmp = self.rightChild
            self = None
            return tmp
        # deleting node with left child
        elif self.rightChild is None:
            tmp = self.leftChild
            self = None
            return tmp

```

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.

```
            self.rightChild = self.rightChild.delete(val)
```

```
            self.rightChild = self.rightChild.delete(val)
        else:
            print(str(val) + " not found in the tree")
            return None
    else:  # val was found
        # deleting node with no children
        if self.leftChild is None and self.rightChild is None:
            self = None
            return None
        # deleting node with right child
        elif self.leftChild is None:
            tmp = self.rightChild
            self = None
            return tmp
        # deleting node with right child
        elif self.leftChild is None:
            tmp = self.rightChild
            self = None
            return tmp
        # deleting a node with two children
        else:
            # first get the inorder successor
            current = self.rightChild
            # loop down to find the leftmost leaf
            while(current.leftChild is not None):
                current = current.leftChild
            self.val = current.val
            self.rightChild = self.rightChild.delete(current.val)

    return self

```

Putting It All Together

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

main.py

BinarySearchTree.py

Node.py

```
from Node import Node
from BinarySearchTree import BinarySearchTree
import random


def display(node):
    lines, _, _, _ = _display_aux(node)
    for line in lines:
        print(line)


def _display_aux(node):
    """
    Returns list of strings, width, height,
    and horizontal coordinate of the root.
    """
    # None.
    if node is None:
        line = 'Empty tree!'
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle
```

```

    # No child.
    if node.rightChild is None and node.leftChild is None:
        line = str(node.val)
        width = len(line)
        height = 1
        middle = width // 2
        return [line], width, height, middle

    # Only left child.
    if node.rightChild is None:
        lines, n, p, x = _display_aux(node.leftChild)
        s = str(node.val)
        u = len(s)
        first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
        second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
        shifted_lines = [line + u * ' ' for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, n + u // 2

    # Only right child.
    if node.leftChild is None:
        lines, n, p, x = _display_aux(node.rightChild)
        s = str(node.val)
        u = len(s)
#        first_line = s + x * '_' + (n - x) * ' '
        first_line = s + x * '_' + (n - x) * ' '
        second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
        shifted_lines = [u * ' ' + line for line in lines]
        final_lines = [first_line, second_line] + shifted_lines
        return final_lines, n + u, p + 2, u // 2

    # Two children.
    left, n, p, x = _display_aux(node.leftChild)
    right, m, q, y = _display_aux(node.rightChild)
    s = '%s' % node.val
    u = len(s)
    first_line = (x + 1) * ' ' + (n - x - 1) * \
        '_' + s + y * '_' + (m - y) * ' '
    second_line = x * ' ' + '/' + \
        (n - x - 1 + u + y) * ' ' + '\\' + (m - y - 1) * ' '
    if p < q:
        left += [n * ' '] * (q - p)
    elif q < p:
        right += [m * ' '] * (p - q)
    zipped_lines = zip(left, right)
    lines = [first_line, second_line] + \
        [a + u * ' ' + b for a, b in zipped_lines]
    return lines, n + m + u, max(p, q) + 2, n + u // 2


BST = BinarySearchTree(6)
BST.insert(3)
BST.insert(2)
BST.insert(4)
BST.insert(-1)
BST.insert(1)
BST.insert(-2)
BST.insert(8)
BST.insert(7)

print("before deletion:")
display(BST.root)

BST.delete(10)
print("after deletion:")
display(BST.root)

```

Quick Quiz!

BST delete Quiz

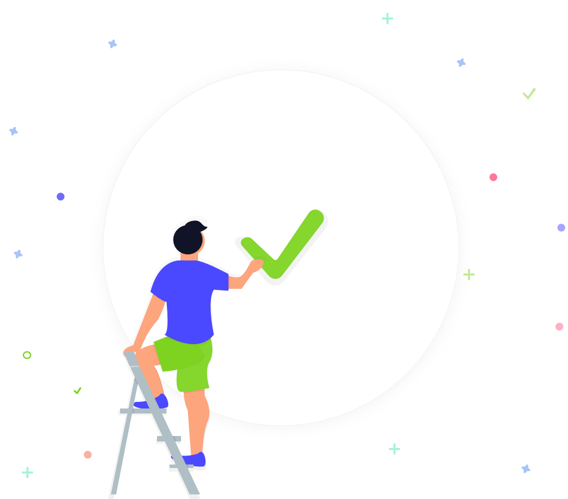
Q In the delete function, why are we only looking in the right-subtree for the smallest value in the node-with-two-children case?

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 sub-tree



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

Deletion in a Binary Search Tree

Next

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)