Delete

September 24, 2023

1 Testing The Delete Node from CS 2302: Data Structures

We are examining a Python program designed to delete a node from a data structure. It's important to clarify that this program is not our creation; instead, it's a tool we are testing. The code can be found in book "CS 2302: Data Structures" authored by Roman Lysecky, Frank Vahid, and Evan Olds.

Our goal in this test is to assess the functionality and effectiveness of this program in removing nodes from a data structure. Deleting nodes is a fundamental operation in data structures, and this program allows us to explore and understand how it's implemented in Python.

1.1 Defining a binary tree

[ ]:
```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

#####################################################################

    def insert(self, node):

        # Check if the tree is empty
        if self.root is None:
            self.root = node
        else:
            current_node = self.root
            while current_node is not None:
                if node.key < current_node.key:
                    # If there is no left child, add the new
                    # node here; otherwise repeat from the
                    # left child.
                    if current_node.left is None:
```

```python
                        current_node.left = node
                        current_node = None
                    else:
                        current_node = current_node.left
                else:
                    # If there is no right child, add the new
                    # node here; otherwise repeat from the
                    # right child.
                    if current_node.right is None:
                        current_node.right = node
                        current_node = None
                    else:
                        current_node = current_node.right

###################################################################

    def search(self, desired_key):
        current_node = self.root
        while current_node is not None:
            # Return the node if the key matches.
            if current_node.key == desired_key:
                return current_node

            # Navigate to the left if the search key is
            # less than the node's key.
            elif desired_key < current_node.key:
                current_node = current_node.left

            # Navigate to the right if the search key is
            # greater than the node's key.
            else:
                current_node = current_node.right

        # The key was not found in the tree.
        return None

####################################################################

    def remove(self, key):
        parent = None
        current_node = self.root

        # Search for the node.
        while current_node is not None:

            # Check if current_node has a matching key.
            if current_node.key == key:
```

```python
                if current_node.left is None and current_node.right is None:  # Case 1
                    if parent is None: # Node is root
                        self.root = None
                    elif parent.left is current_node:
                        parent.left = None
                    else:
                        parent.right = None
                    return  # Node found and removed
                elif current_node.left is not None and current_node.right is None:  # Case 2
                    if parent is None: # Node is root
                        self.root = current_node.left
                    elif parent.left is current_node:
                        parent.left = current_node.left
                    else:
                        parent.right = current_node.left
                    return  # Node found and removed
                elif current_node.left is None and current_node.right is not None:  # Case 2
                    if parent is None: # Node is root
                        self.root = current_node.right
                    elif parent.left is current_node:
                        parent.left = current_node.right
                    else:
                        parent.right = current_node.right
                    return  # Node found and removed
                else:                                    # Case 3
                    # Find successor (leftmost child of right subtree)
                    successor = current_node.right
                    while successor.left is not None:
                        successor = successor.left
                    current_node.key = successor.key      # Copy successor to current node
                    parent = current_node
                    current_node = current_node.right     # Remove successor from right subtree
                    key = parent.key                      # Loop continues with new key
            elif current_node.key < key: # Search right
                parent = current_node
                current_node = current_node.right
            else:                        # Search left
                parent = current_node
                current_node = current_node.left
```

```
        return # Node not found


###############################################################################################

    def bfs(self):
        result = []
        if not self.root:
            return result

        queue = [self.root]

        while queue:
            current_node = queue.pop(0)
            result.append(current_node.key)

            if current_node.left:
                queue.append(current_node.left)
            if current_node.right:
                queue.append(current_node.right)

        return result
```

[ ]:

```
# Main program to test insert and search methods.
tree = BinarySearchTree()

Lista = [42, 17, 89, 5, 63, 30, 11, 56, 74, 28, 39, 8, 92]

for li in Lista:
    tree.insert(Node(li))
```