Module 6: Critical Thinking

Binary Search Tree

Nolan Byrnes

CSC506 – Design and Analysis of Algorithms

Colorado State University – Global Campus

Professor Isaac Gang

October 22nd, 2022

Module 6: Critical Thinking

I developed a program that consists of a class for a binary search tree, a node class, a main function that creates a balanced binary search tree and then inserts and removes a node. This paper explains the classes and their methods, as well as explains the execution results.

**Node Class**

Binary Search Trees consist of Nodes, which are defined as a “basic data structure which contain data and one or more links to other nodes” (CodeAcademy, n.d., para. 1). In a binary search tree, each node has the data the node represents, as well as a maximum of two children, which are the left and right child. The Node class can be viewed in figure 1.

Figure 1.

Node Class

Text

Description automatically generated

Note. The Node class was developed using Python 3

**Binary Search Tree Class**

The Binary Search tree class takes in an array and builds the binary search tree based on the list that is provided to it. Since Binary Search Trees do not allow duplicate values, we remove the duplicates by converting the array into a set, and back to a list by using [\*set(arr)]. the \_\_init\_\_() method then uses the function build\_tree() to build the binary search tree, and returns the root node, which is assigned to the root\_node property of the class.

The \_\_str\_\_() method returns the string representation of the Binary Search Tree Class, and inside of this method, I used a pretty\_tree() function to help print out the binary search tree in a way that we could easily see it’s structure. This pretty\_tree() function that I used was acquired from the course textbook called In Design and Analysis of Algorithms by Lysecky. This pretty\_tree() function can be observed in figure 2 below.

Figure 2.

Binary Search Tree class

Text

Description automatically generated

Note. The pretty\_tree() function helps to print out the Binary Tree in a way that we can see how each node is connected to one another. This function was obtained by the course textbook In Design and Analysis of Algorithms by Lysecky .

The Binary search tree class also includes the functions build\_tree(), insert(), and delete(). The structure of the binary search tree class can be observed in figure 3.

Figure 3.

Binary Search Tree class

Text

Description automatically generated

Note. The Binary\_search\_tree class was developed in Python 3.

**Build tree Method**

The Binary search tree class builds a balanced binary tree of Node objects, and returns the root node with the build\_tree() method. Programiz defines a balanced binary tree as one “in which the height of the left and right subtree of any node differ by not more than 1” (Programiz, n.d., para. 1). The build\_tree() method works by sorting the list, identifying the middle index of the list and creating a node from the value of the middle index, and assigning the values to the left of that middle index to the left\_arr, and the values to the right of the middle index to the right\_arr. To assign each of the created nodes left and right children, it recursively calls the build\_tree function.

The recursive function stops when the list can no longer be split, and will return None, which would represent the left or right child of the node at the bottom of the tree that does not have a left or right child. The build tree method can be viewed in figure 4.

Figure 4.

Build\_tree() function

Text

Description automatically generated

Note. The build\_tree() function belongs to the Binary\_search\_tree Class.

After the recursive function completes, it returns the level 1 node, or root node, and assigns it to the binary search tree’s root node. When passing the array [1, 7, 4, 23, 8, 9, 4, 3, 5, 7, 9, 67, 6345, 324] to the Binary Search Tree, we get the results as shown in figure 5 below.

Figure 5.

Binary Tree Initialization results

Text

Description automatically generated with low confidence

Note. The results represent the Binary search tree after the class object has been initialized, due to the build\_tree() method being called in the constructor.

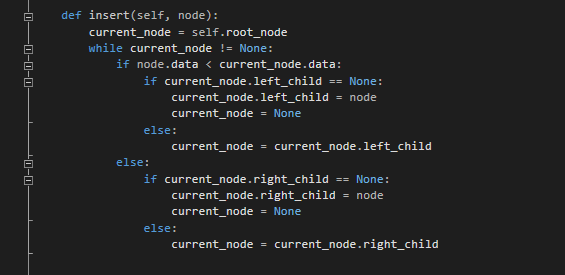
**Insert Method**

After the Binary Search Tree is created, we can then insert additional nodes into the tree by using the insert() method. This method takes a Node as input, and places it into the appropriate spot in the binary search tree by traversing down from the root node. In each iteration of the loop, it checks if the value of the node being inserted is less than the current node. If so, it will check if the left node is available, and if it is, it will insert the node into that position. If the left node is not available, it will assign the left child as the current node and continues with the next iteration.

The inverse applies when the node being inserted is greater than the current node, as it will assign the node to be inserted to the right child if available, and if not, assign the right child as the current node, and continue with the next iteration. The implemented insert() function is shown in figure 6 below.

Figure 6.

Insert method of the Binary\_search\_tree Class



Note. This method belongs to the Binary\_search\_tree class and will insert a node into a developed tree.

After inserting a node with the value of 123 to the created tree shown in figure 4, we can see that the new node has been placed as the left child of the node with the value of 324 in figure 7 below.

Figure 7.

Binary Tree after inserting a node

A picture containing map

Description automatically generated

Note. The results represent the binary search tree after inserting the 123 node.

**Delete Method**

With the binary search tree, we can remove a node from the tree by using the delete() function shown in figure 8 below.

Figure 8.

Delete method of the Binary\_search\_tree Class

Text

Description automatically generated

Note. The Delete method of the Binary\_search\_tree class will remove an existing Node from the binary tree if the node currently exists.

To delete a specific node from the binary search tree, the delete function first must locate the node to be removed. It does so by traversing down the tree starting from the root node. If the node to be removed is less than the current node, it traverses to the left child, and if greater than the current node, it traverses to the right child. This is done until the node to be removed has been identified. Once the node to be removed has been found, there are four different situations that could occur at this point. We would be either be removing a leaf node, a node with only a left child, a node with only a right child, or a node with a left and right child. Removing a leaf node is the easiest situation, as all that needs to occur is that the node’s parent needs to assign the reference to the node being removed to None. If we are removing a node that has a left or right child, we will assign the nodes parent to the child of the node to be removed. In cases where the node to be removed has both a left and right child, the method will identify the successor to take the place of the node being removed by finding the lowest node of the current nodes right subtree. When we are removing the 67 node shown in figure 5, the left-most node of the right subtree below the 67 node is 123, so the 67 node is replaced by the 123 node as shown in figure 9 below.

Figure 9.

Binary Tree after using the delete method

Diagram

Description automatically generated with low confidence

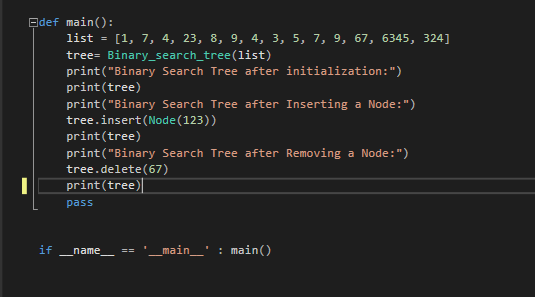
Note. The results represent the binary search tree after removing the node with the value of 67.

**Main**

In my Main() function, I assigned a list, and passed into the Binary\_search\_tree class. After the tree was created, I then inserted a Node with a value of 123, and then removed the node with the value of 67, printing the resulting tree after creating the tree, inserting the node, and removing the node. You can observe the main() function and the final output results in figures 10 and 11 below.

Figure 10.

Main function of executing program



Note. This main function is used to create the binary search tree, insert, and remove a node, while printing the tree in between each step.

Figure 11.

Full execution results of the main() function



Note. The results represent binary search tree after initializing, inserting a node, and removing a node.

**Conclusion**

A balanced binary search tree is one where the height of any node’s subtree will only differ no more than 1. Keeping a binary search tree balanced helps to keep a faster time complexity when performing operations on the binary search tree. If the binary search tree becomes imbalanced, it could end up operating more like a linked list, and we would need to search through every node to find a particular node. For both the insert and delete functions of the Binary\_search\_tree class, the methods traverse down the tree to find the appropriate spot to insert a node, or locate a node to be removed, so by creating a balanced binary tree, the methods that are used can run more efficiently.

**REFERENCES**

Codecademy. (n.d.). Nodes. <https://www.codecademy.com/learn/cspath-cs-102/modules/nodes/cheatsheet>

Programiz. (n.d.). Balanced binary tree. <https://www.programiz.com/dsa/balanced-binary-tree>

Lysecky, R., & Vahid, F. (2019a). 6.11.1: Working with a binary search tree. In Design and Analysis of Algorithms. zyBook.