Laboratory #3

Understanding Concept of Binary Search Trees

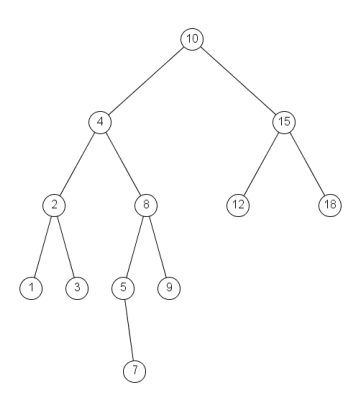
Ivan Gastelum

Professor Olac Fuentes

CS 2302 Data Structures

**INTRODUCTION:**

In this third laboratory of Data Structures, I learned to implement data in a Binary Search Tree (BST). What is a BST? A Binary Search Tree, as it states from its name, is a data structure that starts with a root, this root is normally an object with information inside and have up to two children that are commonly named left and right. A BST should be sorted in order to commonly apply many functions to it, such as Inserting, Deleting, Searching, FindLargest, FindSmallest, etc. Below you will find a representation of a BST, which will be replicated and shown later on.



First task is set to replicate the figure above using the plot functionalities in python by importing matplotlib.pyplot as plt. In this case, a BST should be created with a given list input, and then manipulating the items of each node to show them at the exact location of the tree.

Second task asks to create an iterative version of the recursive method Search(T,k) with k being the key or item that you are looking for in the tree. In other words, a search method should be created using only loops, and not recursion.

In the third task, a balanced binary search tree should be built out of a sorted list. A balanced search tree refers to a tree that automatically keeps its height (maximal number of levels below the root) in the face of arbitrary insertions and deletions. This method should perform the objective in O(n) time.

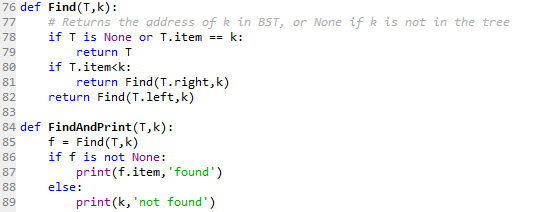
Fourth task is the opposite of the third task. Converting a sorted list from a BST. Again, to be done in O(n) time as well.

Fifth task should give the depth level of each item of the tree.

**PROPOSED SOLUTION DESIGN AND IMPLEMENTATION:**

Solution 1: Before thinking of implementation a graphical representation of any BST, it was necessary to understand how to plot any tree with as many roots or children. Once, the concept is well defined, implementing the numbers is extremely easy. Only by using the concept implemented in Laboratory #1 exercise 3, it only needed to print the items at the same location when the nodes break into two for right child and left child.

Solution 2: Below you can see a recursive method that returns the address of a given value k.



What was done in order to create an iterative version of this code is to think of a loop way to perform the same functionality as the recursive calls. In this case, if we look at the base case, this same condition could be implemented in a while loop, and have a while T is not None, then perform the code. Such code inside the while loop should have three if conditions, one checking if the item of the tree equals the key, if true, then break while loop. However, if False, then we should keep traversing the Tree and keep comparing the items with the key until there is a True equality. If have traversed the whole tree, this means the key was not in the Tree, and we have reached T to be None, so as a result we will return None. See Appendix to see code.

Solution 3: To create a balanced binary search tree out of a sorted list is not as difficult as it sounds. Having a sorted list makes everything much simpler, meaning that we won’t have to use the Insert method to correctly place each node in its correct location. The list should recursively twice by splitting into halves, always obtaining the middle item and insert it into left and right positions.

This algorithm gives us the following running time function:

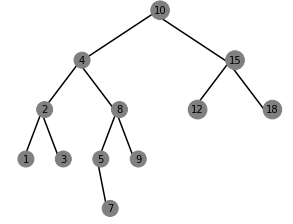
See code in Appendix.

Solution 4: The solution was straight forward. The idea is to start appending number into an empty list from the most left to the most right, this ways you get a sorted list. So, recursively, we start traversing from root to the items to the left until we hit None. Then we return A and append the smallest number, and we continue on the left side by recursively traversing left, if any then appending the items. Such algorithm does not have O(n) time.

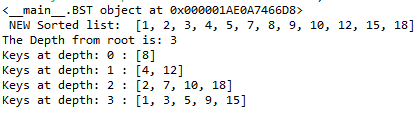
Solution 5: For the last task, before printing values corresponding to each depth level, other method needed to be created. First approach, creating a main method to have all other sub methods, this main method I called “Print\_By Depth”. Inside this main method, there is a for loop in the range of the deepest level of the tree, hence another method needs to be done in order to obtain the depth.

Getdepth method was created recursively to keep adding by 1 until we hit the base case. Sequentially, another method was created to create different lists by appending the values corresponding to the depth level. See appendix for code.

**EXPERIMENTAL RESULTS:**



**From 2 – 5 this is the output:**



First line shows the direct address of the key ‘9’, it is basically saying that the number was found the BST, otherwise it would’ve printed ‘None’.

Second line show a new Sorted List ‘E’ when the method FromTree\_ToList is call with E as the list, and T as the BST.

Third line print the depth of the Tree.

Next lines print the corresponding items of the tree with respect to the depth level they are in. Note that these values are the obtained from calling the method FromList\_ToTree in order to create the Tree without using the Inserting method.

**CONCLUSIONS:**

In conclusion, Binary Search Trees share the same concept of a Linked List, with the only difference of having two attributes instead of only one. The ‘next’ attribute is replaced with ‘left’ and ‘right’ pointers. Furthermore, all functionalities share a similar Base Case to Linked Lists, where T in the case of Trees should point to Null when they start returning the call.

**APPENDIX:**

