**Author:**  *Nguyen, Abram*

**Assignment:** *Lab 3 Report*

**Course:** *CS 2302 - Data Structures*

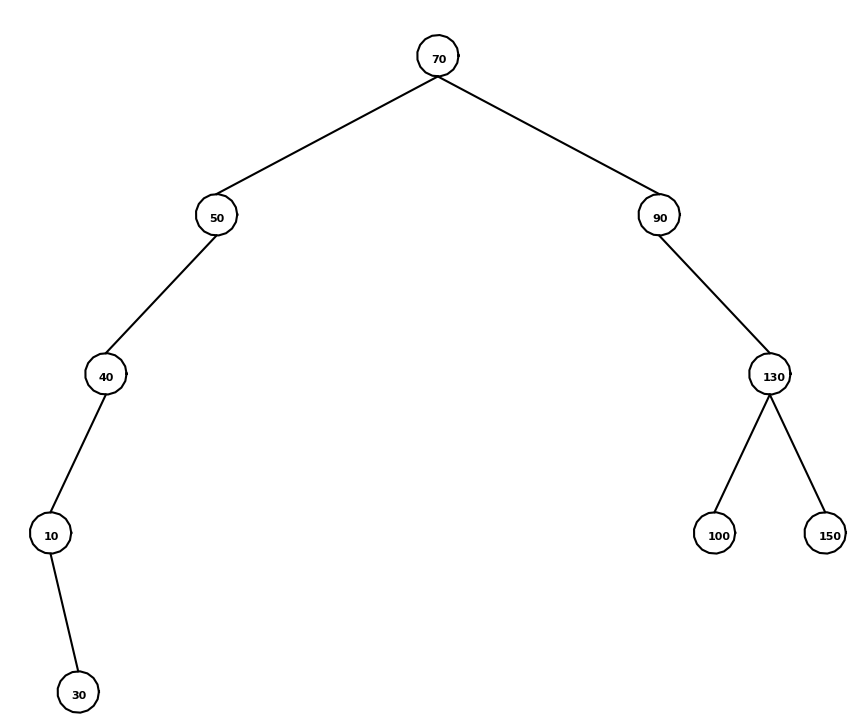
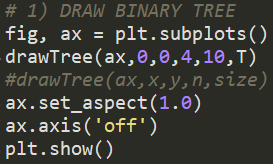
**Instructor:**  *Fuentes, Olac*

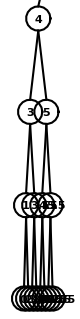
**T.A.:**  *Nath, Anindita*

Introduction:

In this lab, I’m demonstrating the use of binary search trees. I’ve implemented binary search trees using nodes and a more complex version of linked lists. I use binary search trees to organize a list of integers, manipulate it, and represent the tree using python’s matplotlib plotting library.

Proposed solution design and implementation:

1. To draw the binary tree, I manipulated previously used pieces of code from the first lab. I used a method to draw a circle and another to draw the branches of the binary tree and its values. My method of drawing the binary tree included drawing circles(node) as well as at most two lines(branches) beneath a node. Within each circle, I added text that was that node’s data value. I made sure that the program wouldn’t draw branches or nodes that were null. The program came out looking too complex, I don’t think I made it as efficient at it could have been. I used the variables ‘n’ and ‘size’ to stretch and shrink the drawings horizontally and vertically, respectively.
   1. My first method didn’t come out the way I really wanted it to. I had trouble with the proportions and size of the drawing. This includes the circles and the numbers as text. The numbers in the tree’s nodes seem smaller than they actually are in this picture, but I’m including this graphic to show that the lines weren’t positioned how I planned them to be and the text inside the circles aren’t completely centered.
2. For my iterative search method, without recursion, I used a while loop to iterate through the tree. There were three cases for each node I had to consider, which would tell me which node was going to be the next node to traverse to.
   1. If k, a given value, is equal to the value of the current node, return that node.
   2. If k is less than the value of the current node, traverse to that node’s left child.
   3. If k is more than the value of the current node, traverse to that node’s right child.
3. In a binary search tree, for this assignment, a node’s children both have a difference in depth of at most 1. If either side of a tree must contain about the same number of elements, the root of the tree must be the median of a list of numbers. In each iteration of my method to create the balanced search tree, I’ll take the middle element of a sorted list and make that the root, then do the same for the left side of the tree and the right side of the tree.
4. To extract the binary search tree into a sorted list, I used the python native lists, which made the method easier to write. I traversed the entire tree using inorder traversal and added every node’s value to a list. The algorithm was simple, I just needed the time to make the code execute as it should. Essentially, I read the binary tree from left to right, which is how a binary search tree should be read to be ordered.
5. Printing all items at a certain depth, recursion is important for this method. I need to remember where I should stop iterating as well as where I last left off with each call. I traverse down the tree while k is still more than 0. K is essentially a counter to let me know what level (depth) of the tree to go to before returning and printing an item.

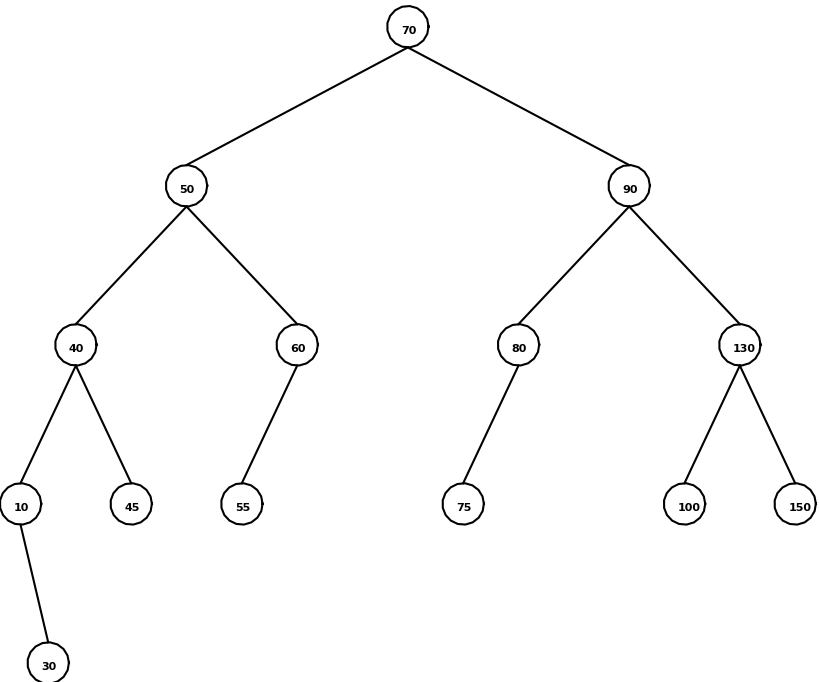


Experimental results:

1. I noticed how the nodes start to get too close to one another as the height of the tree increases in my drawing. This is because of the size of each circle. I can fix it by changing a parameter in my method, but I wanted to avoid having that variable ‘n’ altogether. I’ve shown the list I used and just a portion of the drawing I made from this list to show the result of the cluttered nodes. This is the main flaw of my binary search tree drawings. I could also edit the size of my circles, but the text would end up bleeding out the side of the circles anyway.

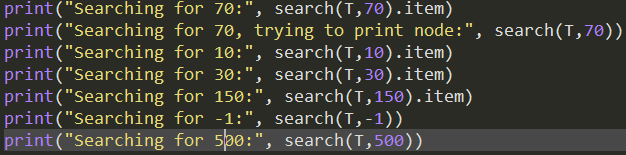


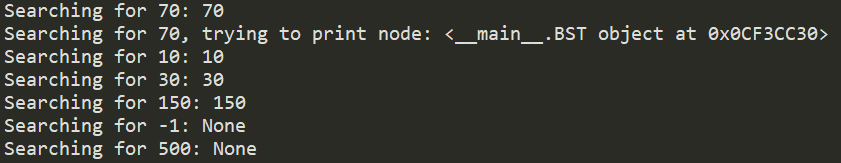
1. The iterative search method I wrote gives an error depending on how I call the method. The method will return a node as instructed, but can give an error if I print the returned node the wrong way. For example, if I search for an item and the method returns ‘None’, I can’t call the method: *search(T, -100).item*, I have to call it just: *search(T, -100)*. A node of ‘None’ has on item to call upon. I have to be careful when searching for any number this way. Also, if I just try to print a node without it’s item, it gives what I assume is the node object’s address.

I’ve included the tree’s list of numbers that I’m experimenting with as well as the method calls and outputs. 

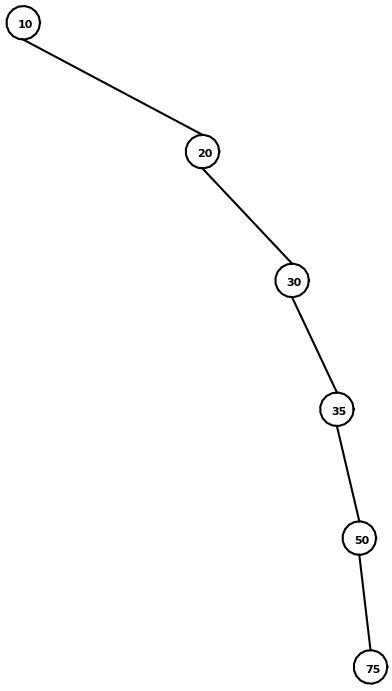


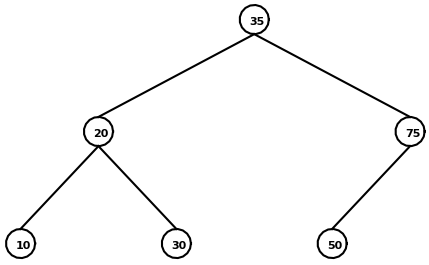
I tested different values, some that I know existed in the tree and some that I know didn’t exist in the tree. Notice that I try printing a node in my second test case. It just prints what seems to be the node’s address. I would fix that issue by adding “.item” to the call, which is shown in the first test case. When I knew that the search method would return an empty node, I didn’t include “.item” in the method call because it would cause an error. In my test cases, it just prints “None”.



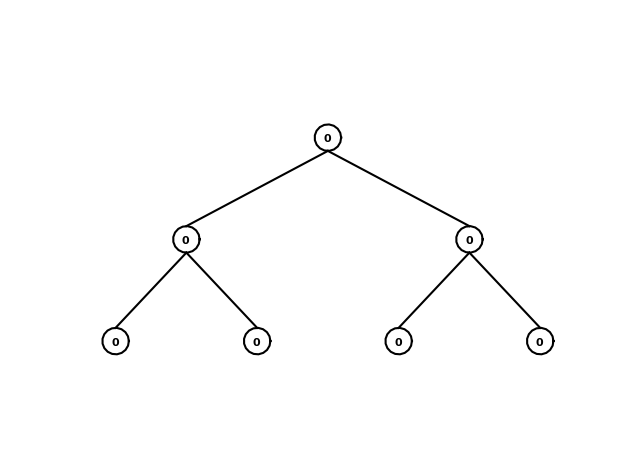


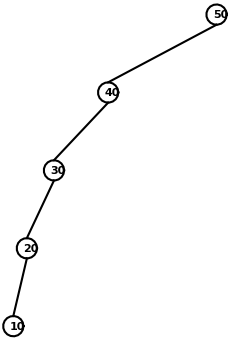
1. After my method to create a balanced tree from a sorted list, I made sure it was balanced by drawing the tree using my previously written drawing method. This is the best way to make sure my method of balancing a tree is valid.
   1. In my first test case, I used the same list I used for this method in my original lab assignment, but I think it works well as a test case for a balancing method. I’ll show the list I used, what the tree would look like unbalanced (using the insert function) as well as the balanced tree to showcase the results of my method.



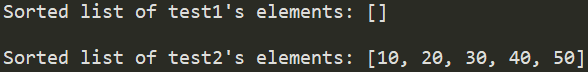


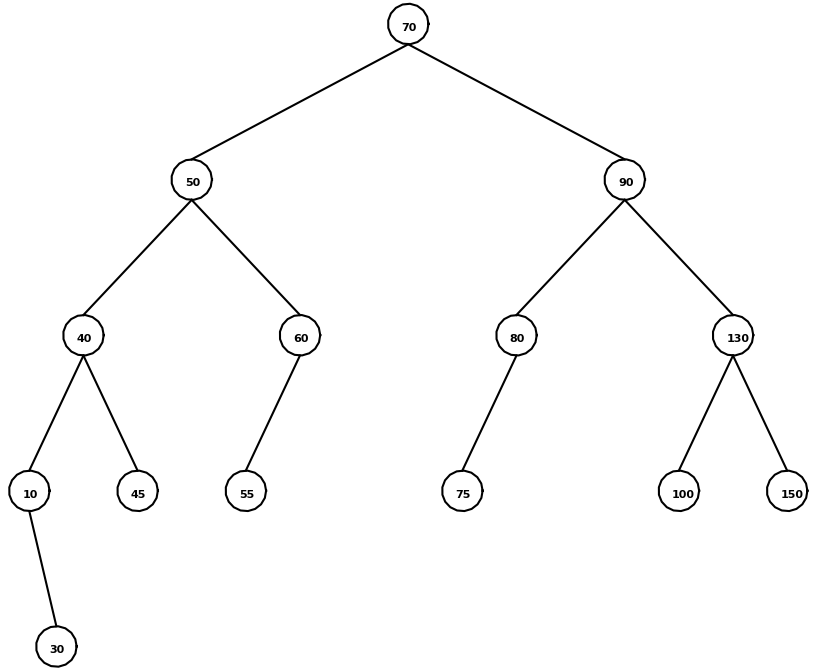
* 1. For my second test case, I wanted to see what would happen if I used a list of the same number. The tree came out as expected.



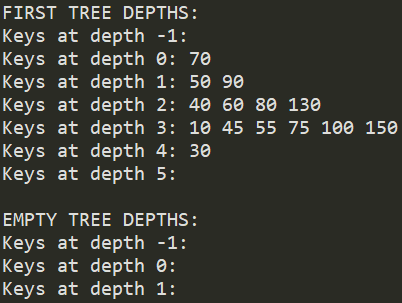
1. For my fourth experiment, I tried extracting a couple different trees containing different sets of numbers in different orders.
   1. I tested the case of having an empty tree
   2. I tested the case of having a completely unbalanced tree that does not contain any right children

The results of my cases are shown. I included a drawing of the tree I used in test case 2. The first tree, of course, was empty.





1. For this experiment, I’ve decided to use 2 different trees, each tested with multiple values of ‘d’, depth. The method should print nothing at all if there are no elements at a given depth.
   1. The first tree I tested was this one, shown at the right ⇒
   2. The second was an empty tree.



Conclusion:

I learned more about binary trees by manipulating and playing with them in this lab. I recognize the use of extracting a list from a search tree as well as even printing every item at a given depth. I was disappointed that I couldn’t quite get to the point that I wanted to in drawing my trees, but this will definitely be a useful learning experience for the future. It really did help to see my binary search trees as a figure that I could view on my computer screen. I think that now I have a stronger grasp on binary trees and binary search trees as a concept. I feel fairly more confident in my skills as a programmer to utilize binary search trees.

Appendix:

|  |  |
| --- | --- |
| """ |  |
|  | Author: Nguyen, Abram |
|  | Assignment: Lab 3 |
|  | Course: CS 2302 - Data Structures |
|  | Instructor: Fuentes, Olac |
|  | T.A.: Nath, Anindita |
|  | Last modified: March 7, 2019 |
|  | Purpose of program: The purpose of this program is to demonstrate the use and |
|  | versatility of binary search trees as well as this |
|  | data structure's method of organization of information. |
|  | """ |
|  | import matplotlib.pyplot as plt |
|  | import numpy as np |
|  | import math |
|  |  |
|  | # Code to implement a binary search tree |
|  | # Programmed by Olac Fuentes |
|  | # Last modified February 27, 2019 |
|  |  |
|  | class BST(object): |
|  | # Constructor |
|  | def \_\_init\_\_(self, item, left=None, right=None): |
|  | self.item = item |
|  | self.left = left |
|  | self.right = right |
|  |  |
|  | def Insert(T,newItem): |
|  | if T == None: |
|  | T = BST(newItem) |
|  | elif T.item > newItem: |
|  | T.left = Insert(T.left,newItem) |
|  | else: |
|  | T.right = Insert(T.right,newItem) |
|  | return T |
|  |  |
|  |  |
|  | """ |
|  | ############################################################################### |
|  | # Programmed by Abram Nguyen ################################################## |
|  | # Last modified March 7, 2019 ################################################# |
|  | ############################################################################### |
|  | """ |
|  |  |
|  | ############################################################################### |
|  | # Display the binary tree as a figure ######################################### |
|  | ############################################################################### |
|  | def drawTree(ax,x,y,n,size,T): |
|  | if T != None: |
|  | drawCircle(ax,[x,y],1.5) #draw a circle |
|  | ax.text(x-.5, y-.5, T.item, size=8, weight='bold') #include node's value |
|  | #if T has a left child: |
|  | if T.left != None: |
|  | #plot the left branch and draw the rest of the tree |
|  | ax.plot([x,x-(2\*\*n)],[y-1.5,y-size], color='k') |
|  | drawTree(ax,x-(2\*\*n),y-size-1.5,n-1,size,T.left) |
|  | if T.right != None: |
|  | #plot the right branch and draw the rest of the tree |
|  | ax.plot([x,x+(2\*\*n)],[y-1.5,y-size], color='k') |
|  | drawTree(ax,x+(2\*\*n),y-size-1.5,n-1,size,T.right) |
|  |  |
|  | # Draws a circle of radius 'r' at center point [x,y] |
|  | def drawCircle(ax,center,r): |
|  | x,y = circle(center,r) |
|  | ax.plot(x,y,color='k') |
|  |  |
|  | # Creates a circle given a center and radius |
|  | def circle(center,radius): |
|  | n = int(4\*radius\*math.pi) |
|  | t = np.linspace(0,6.3,n) |
|  | y = center[1]+radius\*np.sin(t) |
|  | x = center[0]+radius\*np.cos(t) |
|  | return x,y |
|  |  |
|  | ############################################################################### |
|  | # Iterative version of the search operation. ################################## |
|  | ############################################################################### |
|  | def search(T,k): |
|  | while T != None: |
|  | if k == T.item: |
|  | return T |
|  | #if k is still smaller, traverse left |
|  | elif k < T.item: |
|  | T = T.left |
|  | #if k is still larger, traverse right |
|  | elif k > T.item: |
|  | T = T.right |
|  | return T |
|  |  |
|  | ############################################################################### |
|  | # Building a balanced binary search tree given a sorted list as input. ######## |
|  | ### Note: this should not use the insert operation, ########################### |
|  | ### the tree must be built directly from the list in O(n) time. ############### |
|  | ############################################################################### |
|  | def balance(L): |
|  | if L: |
|  | # median(middle) of list will be the top node to keep the tree balanced |
|  | med = len(L) // 2 |
|  | T = BST(L[med]) |
|  | #insert right sub-tree |
|  | T.right = balance(L[med+1:]) |
|  | #insert left sub-tree |
|  | T.left = balance(L[:med]) |
|  | return T |
|  |  |
|  | ############################################################################### |
|  | # Extracting the elements in a binary search tree into a sorted list. ######### |
|  | ### As above, this should be done in O(n) time. ############################### |
|  | ############################################################################### |
|  | def extract(T, L): |
|  | if T != None: |
|  | #traverse through every node in the tree |
|  | extract(T.left, L) |
|  | L += [T.item] #append every item to L, a native python list |
|  | extract(T.right, L) |
|  |  |
|  | ############################################################################### |
|  | # Printing the elements in a binary tree ordered by depth. #################### |
|  | ### The root has depth 0, the root’s children have depth one, ################# |
|  | ### and so on. ################################################################ |
|  | ############################################################################### |
|  | def depthPrint(T, k): |
|  | if T != None: |
|  | if k == 0: |
|  | #print the current node(s) data |
|  | print(T.item, end=" ") |
|  | else: |
|  | #traverse to every node at level k |
|  | depthPrint(T.left, k-1) |
|  | depthPrint(T.right, k-1) |
|  |  |
|  | """ |
|  | # Code to test the functions above ############################################ |
|  | """ |
|  | T = None |
|  | #A = [10, 9, 8, 7, 4, 5, 3, 1, 3.5, 4.5, 5.5, 2, 3.6, 4.75, 5.25, 5.75, 3.25, 4.25, 0] |
|  | A = [70, 50, 90, 130, 150, 40, 10, 30, 100, 60, 80, 45, 55, 75] |
|  | for a in A: |
|  | T = Insert(T,a) |
|  |  |
|  | print("ANALYZING TREE 'T'...") |
|  | print("======================\n") |
|  |  |
|  | # 1) DRAW BINARY TREE |
|  | fig, ax = plt.subplots() |
|  | drawTree(ax,0,0,4,10,T) |
|  | #drawTree(ax,x,y,n,size) |
|  | ax.set\_aspect(1.0) |
|  | ax.axis('off') |
|  | plt.show() |
|  |  |
|  |  |
|  | # 2) ITERATIVE SEARCH |
|  | print("Searching for 70:", search(T,70).item) |
|  | print("Searching for 70, trying to print node:", search(T,70)) |
|  | print("Searching for 10:", search(T,10).item) |
|  | print("Searching for 30:", search(T,30).item) |
|  | print("Searching for 150:", search(T,150).item) |
|  | print("Searching for -1:", search(T,-1)) |
|  | print("Searching for 500:", search(T,500)) |
|  | print() |
|  |  |
|  | # 3) BALANCE A TREE |
|  | B = [10, 20, 30, 35, 50, 75] |
|  | C = balance(B) |
|  | fig, ax = plt.subplots() |
|  | drawTree(ax,0,0,4,10,C) |
|  | ax.set\_aspect(1.0) |
|  | ax.axis('off') |
|  | plt.show() |
|  |  |
|  | D = [0,0,0,0,0,0,0] |
|  | E = balance(D) |
|  | fig, ax = plt.subplots() |
|  | drawTree(ax,0,0,4,10,E) |
|  | ax.set\_aspect(1.0) |
|  | ax.axis('off') |
|  | plt.show() |
|  |  |
|  |  |
|  | # 4) EXTRACT TREE INTO LIST |
|  | Z = None |
|  | test1 = [] |
|  | extract(Z, test1) |
|  |  |
|  | fig, ax = plt.subplots() |
|  | drawTree(ax,0,0,4,10,Z) |
|  | #drawTree(ax,x,y,n,size) |
|  | ax.set\_aspect(1.0) |
|  | ax.axis('off') |
|  | plt.show() |
|  | print("Sorted list of test1's elements:", test1) |
|  | print() |
|  |  |
|  |  |
|  | Y = None |
|  | y = [50, 40, 30, 20, 10] |
|  | for j in y: |
|  | Y = Insert(Y,j) |
|  | test2 = [] |
|  | extract(Y, test2) |
|  |  |
|  | fig, ax = plt.subplots() |
|  | drawTree(ax,0,0,4,10,Y) |
|  | #drawTree(ax,x,y,n,size) |
|  | ax.set\_aspect(1.0) |
|  | ax.axis('off') |
|  | plt.show() |
|  | print("Sorted list of test2's elements:", test2) |
|  | print() |
|  |  |
|  |  |
|  | # 5) PRINT ELEMENTS AT GIVEN DEPTH |
|  | print("FIRST TREE DEPTHS:") |
|  | print("Keys at depth -1:", end=" ") |
|  | depthPrint(T,-1) |
|  | print() |
|  | print("Keys at depth 0:", end=" ") |
|  | depthPrint(T,0) |
|  | print() |
|  | print("Keys at depth 1:", end=" ") |
|  | depthPrint(T,1) |
|  | print() |
|  | print("Keys at depth 2:", end=" ") |
|  | depthPrint(T,2) |
|  | print() |
|  | print("Keys at depth 3:", end=" ") |
|  | depthPrint(T,3) |
|  | print() |
|  | print("Keys at depth 4:", end=" ") |
|  | depthPrint(T,4) |
|  | print() |
|  | print("Keys at depth 5:", end=" ") |
|  | depthPrint(T,5) |
|  | print("\n") |
|  |  |
|  | print("EMPTY TREE DEPTHS:") |
|  | Empty = None |
|  | print("Keys at depth -1:", end=" ") |
|  | depthPrint(Empty,-1) |
|  | print() |
|  | print("Keys at depth 0:", end=" ") |
|  | depthPrint(Empty,0) |
|  | print() |
|  | print("Keys at depth 1:", end=" ") |
|  | depthPrint(Empty,1) |
|  | print() |

I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

* Abram Nguyen