Lab 3: Binary Search Trees

Manuel Ruvalcaba

CS2302 1:30-2:50

1. Introduction

A binary search tree is a tree where each node can have a left and right node, with items less than the current node being on the left and items greater than the current node on the right. In this lab, we will implement various methods to display a binary search tree as a figure, search for an item, build a tree from a sorted list, extract elements from a tree into a sorted list, and print the items in specific depths.

2. Proposed Solution Design and Implementation

To solve the problem regarding the drawing of the BST as a figure, I had to implement several methods. Using the *matplotlib* library, I created four methods. These four methods consist of *DrawTree(), circle(), DrawCircles()*, and *DrawNumbers()*. The *DrawTree* method accepts a *tree* as a parameter and creates the tree structure of the BST by checking if the *right* or *left* of the current node exists. If it does, then it will draw a branch, otherwise it will do nothing. It does this recursively until the entire structure is complete. The *circle* method returns the *x* and *y* coordinates for the circles that are drawn in the *DrawCircles* method. The *DrawCirlces* method accepts a *tree* as a parameter and creates the circles that will represent the nodes of the tree. This is done in a similar fashion as the *DrawTree* method does by checking the *left* and *right*, then drawing the circles in their proper positions. The *DrawNumbers* method uses *ax.text* to create text boxes that a circular with a white face color. These text boxes are the same size as the nodes of the tree and sit in the same position as the nodes, where they belong. Lastly, I have a fifth method called *DrawBST* that calls the previously mentioned functions. If the tree is empty, it will do nothing. If the tree has one item, it will only call the *DrawNumbers*, and *DrawCircles* function. If the tree has more than one item, it will call all the functions previously mentioned.

Creating the method to search the tree iteratively was relatively simple. I needed a single function called *Search* that receives a *tree* and a *key* that will be searched for. While the current node *is not None* and the item has not been found, the current node's item will be compared to the key and if the key is less than, the current node will point to the *left* of the node. Otherwise, it will point to the *right* of the node. If the key is found or the current node is *None,* the *while* loop will no longer be executed and the reference to the current node will be returned. I also included a *SearchAndPrint* method that calls the *Search* function and prints the item or *None* if the item was not found.

When building a balanced tree from a sorted list in linear time, O(n), I needed a single method called *BuildTree* that accepts an empty tree and the sorted list. The base case checks if the list is empty, which simply returns. Otherwise, we get the item middle of the list and if the current node *is None*, we will create a new node with the element at the middle of the list. Then we make the left side of the tree equal to the recursive call that now receives the *left side* of the tree and the left half of the list, excluding the middle. Then we make the right side of the tree equal to the recursive call that now receives the *right side* of the tree and the right half of the list, excluding the middle. When it is finished, it will return the tree.

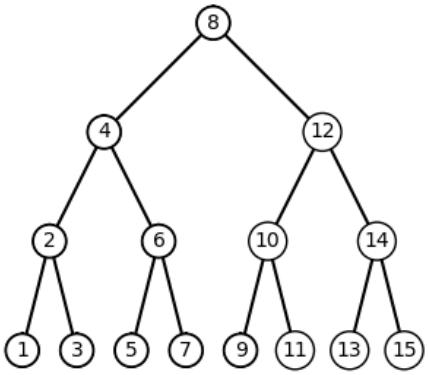
For extracting the elements of a tree into a sorted list, I used two methods called *Extract* and *Extractor*. The extractor method receives a *tree* and a *list*. If the current node *is not None*, then the method will recursively move to the left. Once it is at the smallest item, it will append that item. After that, it will append its parent, and then append the right child. After all the items are appended, it will return the list. The *Extract* method receives a *tree* and creates an empty list.

This method then returns what the *Extractor* method returns when the *tree* and empty list are the parameters. This method extracts the elements in linear time as it uses an *In Order* transversal, which is in linear time, O(n).

Lastly, for the printing of the items at different depths, I utilized three methods called *PrintAtDepths*, *PrintLevel*, and *TreeHeight*. The method *PrintAtDepths* receives a tree and calculates the height of the tree using the *TreeHeight* method. I then used the height in a *for* loop that goes for *i* in the range from zero to the height excluding the height. Inside the loop, the *PrintLevel* method is called with the tree and the current depth. The *PrintLevel* method checks if the current node *is None*. If it is, it simply returns. It then checks if it is at the correct depth. If it is then it prints the current item. Otherwise, it will recursively go to the left and the right until it reaches the correct depth.

3. Experimental Results

To test the methods, I decided to create a tree using my *BuildTree* method with a list containing the values [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]. The *BuildTree* method works in linear time and should build a balanced tree. I will then use the *DrawBST* method to display the binary tree as a figure. After that, I will use my *Search* method to search for both a key that is present and a key that is not present in the tree.



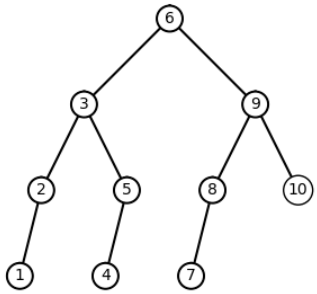
This is the tree as a figure. The *BuildTree* function was to build a tree that is balanced and the *DrawBST* method draws the tree the way that it was built. Therefore, both the *BuildTree* and the *DrawBST* work as intended. The tree took 140.63186699786456 milliseconds to be drawn.

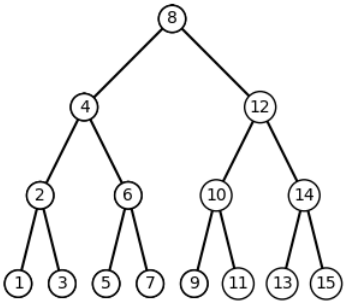
Now I will search for four integers in the tree, being the numbers 5, 10, 15, and 20 using my *Search* method. After searching for the numbers 5, 10, and 15, the references to the nodes will be returned in which the item can be printed. After searching for 20, the reference to *None* will be returned, showing that the key was not in the tree.

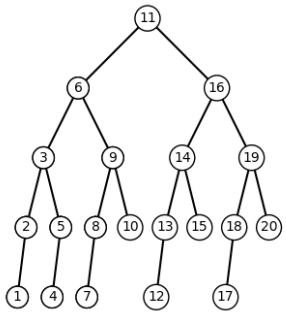
Key	Found?	RunTime in Milliseconds
5	Yes	0.197911002
10	Yes	0.198424998
15	Yes	0.181461997
20	No	0.193285006

To test the building of a balanced tree, I will create three trees from various sizes of sorted lists using my *BuildTree* method and include their drawings to show they are balanced.

Tree	Size	RunTime in Milliseconds
1	10	0.046779001
2	15	0.099727004
3	20	0.117204996



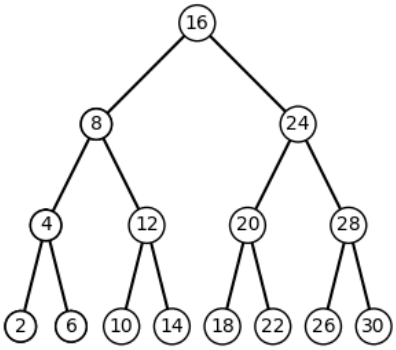




To test the extraction of elements from a tree into a sorted list, there will be three trees and I will extract the elements into a sorted list using the *Extract* method. I am also going to get the runtime and check to see if the results are as intended.

Tree	Sorted?	RunTime in Milliseconds
1	Yes	0.022103995
2	Yes	0.020048006
3	Yes	0.021077001

Finally, to test the printing of elements at different depths, I will have a tree built and I will show the drawn tree as well as the output given in the console.



Output given by the console:

Keys at Depth 0: 16

Keys at Depth 1: 8 24

Keys at Depth 2: 4 12 20 28

Keys at Depth 3: 2 6 10 14 18 22 26 30

Runtime of the method: 1.854199996159 milliseconds.

4. Conclusion

With this project, I have learned to build a tree from a sorted list, to extract items from a tree into a sorted list, to draw a BST as a figure, to search through a tree iteratively, and to print the items at different depths.

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.

5. Appendix

```
'''
Course: CS2302 MW 1:30-2:50
Author: Manuel A. Ruvalcaba
Assignment: Lab #3
Instructor: Dr. Olac Fuentes
TA: Anindita Nath, Maliheh Zargaran
Date of Last Modification: March 8, 2019
Purpose of the Program: The purpose of this program is to make figures to build
                        BST, to Search for an item iteratively, to extract the
                        elements from a BST, to build a BST from a list, and to
                        print the items by depth.
'''


class BST(object):
    # Constructor
    def __init__(self, item, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

import matplotlib.pyplot as plt
import numpy as np
import math
import timeit

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

def Delete(T,del_item):
    #Deletes an element
    if T is not None:
        if del_item < T.item:
            T.left = Delete(T.left,del_item)
        elif del_item > T.item:
            T.right = Delete(T.right,del_item)
        else:  # del_item == T.item
            if T.left is None and T.right is None: # T is a leaf, just remove it
                T = None
            elif T.left is None: # T has one child, replace it by existing child
                T = T.right
            elif T.right is None:
                T = T.left
            else: # T has two chldren. Replace T by its successor, delete successor
                m = Smallest(T.right)
                T.item = m.item
                T.right = Delete(T.right,m.item)
    return T

def InOrder(T):
    # Prints items in BST in ascending order
    if T is not None:
        InOrder(T.left)
        print(T.item,end = ' ')
        InOrder(T.right)

def InOrderD(T,space):
    # Prints items and structure of BST
    if T is not None:
        InOrderD(T.right,space+'   ')
```

```
        print(space,T.item)
        InOrderD(T.left,space+'    ')

def SmallestL(T):
    # Returns smallest item in BST. Returns None if T is None
    if T is None:
        return None
    while T.left is not None:
        T = T.left
    return T

def Smallest(T):
    # Returns smallest item in BST. Error if T is None
    if T.left is None:
        return T
    else:
        return Smallest(T.left)

def Largest(T):
    #Returns the largest element
    if T.right is None:
        return T
    else:
        return Largest(T.right)

def Find(T,k):
    # Returns the address of k in BST, or None if k is not in the tree
    if T is None or T.item == k:
        return T
    if T.item<k:
        return Find(T.right,k)
    return Find(T.left,k)
```

```
def FindAndPrint(T,k):
    f = Find(T,k)
    if f is not None:
        print(f.item,'found')
    else:
        print(k,'not found')

def TreeHeight(t):
    #Gets the height of the Tree
    if t is None:
        return 0
    else:
        lheight = TreeHeight(t.left)
        rheight = TreeHeight(t.right)
        if lheight > rheight:
            return lheight+1
        else:
            return rheight+1

def Search(T,k):
    #Searches for an item in a BST and returns a reference to the Node iteratively
    while T is not None and T.item != k:
        if T.item<k:
            T = T.right
        else:
            T = T.left
    return T

def SearchAndPrint(T,k):
    #Prints the result of the Search method
    f = Search(T,k)
    if f is not None:
        print(f.item,'found')
```

```
    else:
        print(k,'not found')


def Extract(T):
    #Creates an empty list and calls the Extractor Method
    a = []
    return Extractor(T,a)

def Extractor(T,a):
    #Extracts the elements from a BST into a sorted list. Running Time O(n)
    if T is not None:
        Extractor(T.left,a)
        a.append(T.item)
        Extractor(T.right,a)
    return a

def BuildTree(T,a):
    #Builds a tree from a sorted list. Running Time: O(n)
    if len(a) == 0:
        return
    middle = len(a)//2
    if T == None:
        T =  BST(a[middle])
    T.left = BuildTree(T.left,a[:middle])
    T.right = BuildTree(T.right,a[middle+1:])
    return T

def PrintAtDepths(T):
    #Prints the keys at the different levels
    h = TreeHeight(T)
    for i in range(0, h):
        print("Keys at Depth %d: " %i, end=' ')
```

```
        PrintLevel(T,i)
        print()

def PrintLevel(T,d):
    #Prints the keys at the different levels
    if T is None:
        return
    if d == 0:
        print(T.item, end = ' '),
    elif d > 0 :
        PrintLevel(T.left , d-1)
        PrintLevel(T.right , d-1)

def DrawBST(T, ax, p, r):
    #Draws a tree as a figure.
    if T is None:
        return
    z = 200
    if T.left is None and T.right is None:
        DrawCircles(T,ax,p,z)
        DrawNumbers(T,ax,p,z)
    else:
        q = np.array([[0,0],[0,0]])
        DrawTree(T,ax,q,z)
        DrawCircles(T,ax,p,z)
        DrawNumbers(T,ax,p,z)

def DrawTree(T,ax,p,z):
    #Draws the Tree for the BST
    if T.left is not None:
        q = np.zeros((2,2)) #array with points for the new tree
        dx = z/2 #difference in x between children nodes
        dy = 100 #difference in y between children and parent
```

```
        d = np.array([[-dx,-dy],[0,0]])
        q[:,0] = d[:,0]+p[0,0]
        q[:,1] = d[:,1]+p[0,1]
        ax.plot(q[:,0],q[:,1],color='k')
        DrawTree(T.left,ax,q,dx)
    if T.right is not None:
        q1 = np.zeros((2,2)) #array with points for the new tree
        dx1 = z/2 #difference in x between children nodes
        dy1 = 100 #difference in y between children and parent
        d1 = np.array([[dx1,-dy1],[0,0]])
        q1[:,0] = d1[:,0]+p[0,0]
        q1[:,1] = d1[:,1]+p[0,1]
        ax.plot(q1[:,0],q1[:,1],color='k')
        DrawTree(T.right,ax,q1,dx1)

def circle(center,rad):
    #makes the circle
    n = int(4*rad*math.pi)
    t = np.linspace(0,6.3,n)
    x = center[0]+rad*np.sin(t)
    y = center[1]+rad*np.cos(t)
    return x,y

def DrawCircles(T,ax,center,z):
    #Draws the circles for each node in the BST
    if T is not None:
        x,y = circle(center,15)
        ax.plot(x,y,color='k')
        dy = 100 #difference in y between children and parent
        dx = z/2 #difference in x between children and parent
        if T.left is not None:
            DrawCircles(T.left,ax,[center[0]-dx,center[1]-dy],dx)
        if T.right is not None:
```

```
            DrawCircles(T.right,ax,[center[0]+dx,center[1]-dy],dx)


def DrawNumbers(T,ax,center,z):
    #Draws the numbers inside the Nodes of the BST
    if T is not None:
        x = center[0]
        y = center[1]
        dy = 100 #difference in y between children and parent
        dx = z/2 #difference in x between children and parent
        ax.text(x, y, T.item, horizontalalignment='center',
verticalalignment='center', color="k", fontsize=10,
                bbox=dict(facecolor='white', edgecolor='black', boxstyle='circle'))
        if T.left is not None:
            DrawNumbers(T.left,ax,[center[0]-dx,center[1]-dy],dx)
        if T.right is not None:
            DrawNumbers(T.right,ax,[center[0]+dx,center[1]-dy],dx)


#This is where the code can be tested
Z = None
Y = [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30]
Z = BuildTree(Z,Y)
start1 = timeit.default_timer()
PrintAtDepths(Z)
stop1 = timeit.default_timer()
print('Time: ', (stop1 - start1)*1000)


plt.close("all")
fig, ax = plt.subplots()
DrawBST(Z, ax, [0,0], 100)
ax.set_aspect(1.0)
ax.axis('off')
plt.show()
fig.savefig('DrawBST.png')
```

```
start1 = timeit.default_timer()
SearchAndPrint(Z,20)
stop1 = timeit.default_timer()
print('Time: ', (stop1 - start1)*1000)
T = None
A = [70, 50, 90, 130, 150, 40, 10, 30, 100, 180, 45, 60, 140, 42]
for a in A:
    T = Insert(T,a)
SearchAndPrint(T,70)
SearchAndPrint(T,71)
print(Extract(T))
Z = None
Y = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 200, 400, 500]
Z = BuildTree(Z,Y)
PrintAtDepths(Z)
```