Ivan Perez
80602918
Lab 3 Report

Introduction

The lab requires a figure of the binary tree to be made, an iterative version of search, Building a balanced binary tree from a sorted list in O(n) of time, create a sorted list from a binary search tree in O(n) time, and printing values at each depth of the tree.

Proposed solution design and implementation

The search method I had check the value of the node it was currently on. If it wasn't in the node, depending on the value, it would go to the left subtree if it was less than the nodes value, or the right subtree if it was greater than the Nodes value. When the value was found, it would return the address of the node, else, it would return none/null.

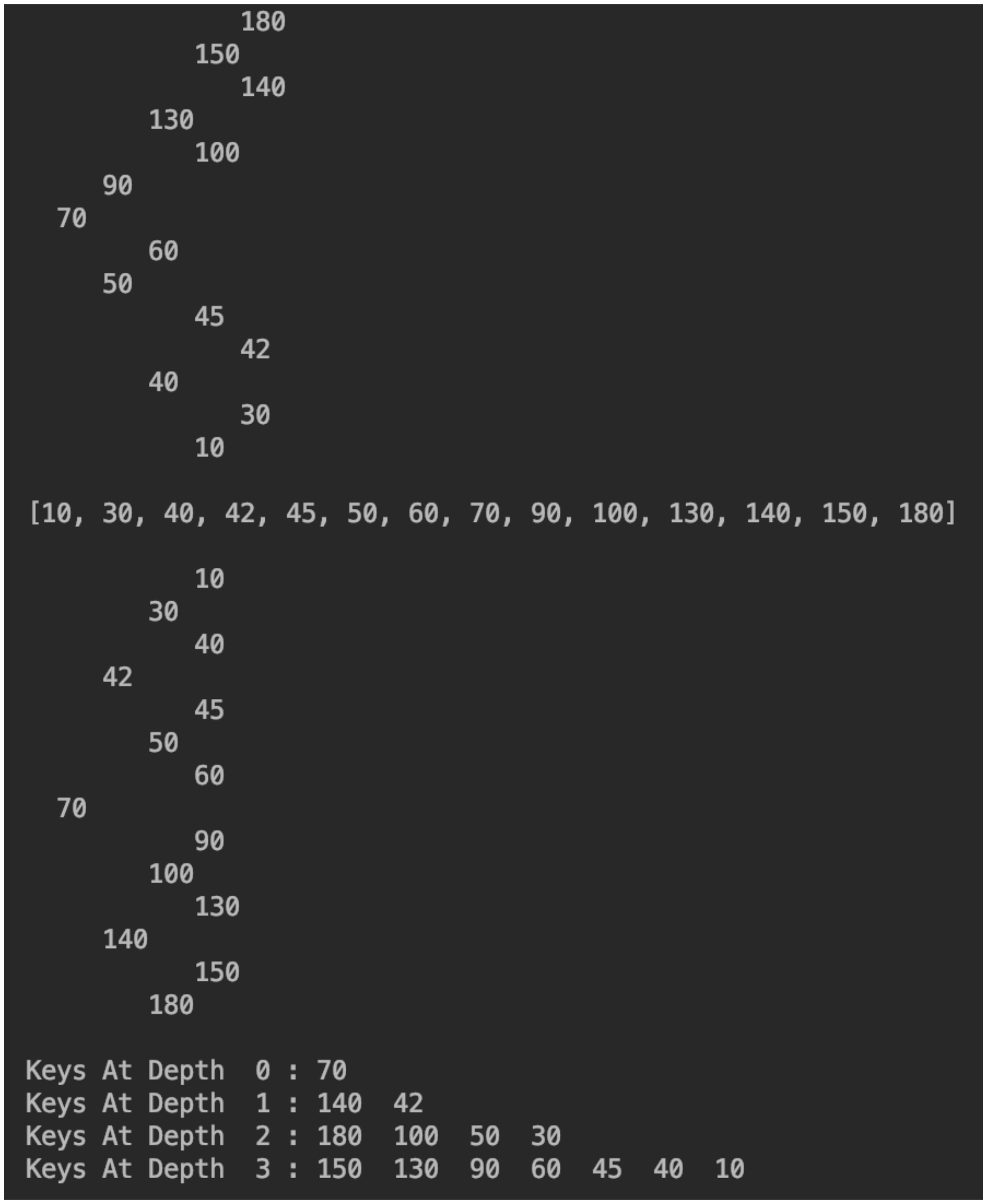
The balanced binary tree method was created by finding the median of the list first. From there, the values left of the list were created into a new list, the right to another list. The median of the list is turned into a binary tree node. The method is recursive so we send the left and right list separately until it reaches the last value which creates a node. After it is created into a node, it is returned and the binary search tree is constructed from its return statements.

The sorted list method was made recursively. The method follows the left subtree until it reaches a leaf. Once it reaches the leaf, the value is returned in a list, and the previous recursive call places the parent of the node's value in the list, then it calls for the right subtree to go through recursively. The pattern turns into parent's left child, parent, parent's right child.

The print values at every depth algorithm was divided into 2 methods. One method relies on recursion and it gets values at a specific level/ depth. The other method calls the first method and does the print structure to state what values there are in each level. The recursive method goes down each node until it reaches the nodes at the depth called. The variable n, the value that states which depth's values are needed, is subtracted by one for every depth traversed. Once n is 0, the value at those nodes are returned in a list. The second method has a while loop that calls the recursive statement to get the values of each depth. Then a print statement says what depth's values will be printed and then the values are printed from the list. The while loop ends when the recursive statement returns an empty list. The empty list shows there are no more values in the tree.

Experimental results

I first had the methods print the tree. Then I created a sorted list which it was then printed, I then balanced the tree and printed it. I then had the code print the values at each depth. The results, in that order, are displayed below. I used a pre-made list and made a Tree using the insert method by placing the values within the list.



```
              180
           150
              140
        130
           100
     90
  70
        60
     50
           45
              42
        40
              30
           10

[10, 30, 40, 42, 45, 50, 60, 70, 90, 100, 130, 140, 150, 180]

           10
        30
           40
     42
           45
        50
           60
  70
           90
        100
           130
     140
           150
        180

Keys At Depth  0 : 70
Keys At Depth  1 : 140  42
Keys At Depth  2 : 180  100  50  30
Keys At Depth  3 : 150  130  90  60  45  40  10
```

The runtime for each method is displayed below
Search: O(n)
BalancedTrees: O(n)
SortedList: O(n)

PrintDepths: O(n)

Conclusions

This lab was great for understanding binary search trees. The search method provided me an excellent way to traverse a tree for a value instead of going to each node until its found. I also learned the structure of a binary search tree so that I can construct a sorted list.

Appendix

```
################################################################
###############
#
# Course: CS2302 (MW – 1:30pm)
# Author: Ivan Perez
# Assignment: Lab 3 – Binary Search Trees
# Instructor: Olac Fuentes
# T.A.: Anindita Nath
# Last Modified: Mar 12, 2019
# Purpose: Contains a binary search tree object along with
methods that provide
# ways to utilize the binary search tree. Some of these object
methods
# are Node insertion, deletion, and find. The ones required for
this assignment are
# displaying a figure of the binary tree,an iterative search
method, balancing
# a tree, extract elements of a tree and place into a sorted
list, and print
# elements at a certain depth.
#
################################################################
###############

import random


# The code below is the code Mr. Fuentes supplied to us for the
Binary Tree object –
# 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


def Delete(T, del_item):
    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):
    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')

################################################################
###########
# The rest of the code is what was required by the assignment
and created by me


def Search(T, s):
    # iteratively searches for value and returns address
    if T is not None:
        temp = T
        while temp is not None:
            if temp.item == s:
                return temp  # returns address of node
            # iterates through correct subtree
            if s < temp.item:
                temp = temp.left
            else:
                temp = temp.right

        return None  # returns none if value not found


def BalancedTree(L):
    # Balances a Binary tree to make, at most, a height
difference of one
    if len(L) < 1:
        return None
    if len(L) < 2:
        return BST(L[0])
    half = len(L)//2  # gets median location
    # makes a left and right list from the median value
    r = L[:half]
    l = L[half+1:]
    T = BST(L[half])
    #  recursively call to make left and right subtrees
    T.left = BalancedTree(l)
    T.right = BalancedTree(r)
    return T


def SortedList(T):
    # Extracts values of a tree into a sorted list
    if T is None:  # base if Tree is empty
```

```
        return []
    if T.right is None and T.left is None:  # returns value when 
its a leaf
        return [T.item]
    # recursively calls for sorted list
    return SortedList(T.left) + [T.item] + SortedList(T.right)


def AtLevel(T, n):
    # Gets the elements at the depth n
    if T is None:
        return []
    if n == 0:  # Reaches the depth n
        return [T.item]
    else:  # recursively calls the left and right subtrees
        return AtLevel(T.left, n-1) + AtLevel(T.right, n-1)


def PrintDepths(T):
    # Prints the elements at all the depths of the tree
    l = [0]
    n = 0
    while len(l) > 0:
        l = AtLevel(T,n)  # Calls for elements at a certain 
level/depth
        if len(l) == 0:  # if there are no elements then tree 
search is compelete
            break

        # Prints elements
        print('Keys At Depth ', n, ': ', end='')
        for x in range(len(l)):
            print(l[x], ' ', end='')
        print()
        n += 1

# Code to test the methods above
# This code is to show that the methods work
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)

InOrderD(T, ' ')
print()
```

```
L = SortedList(T)
print(L)
print()
T = BalancedTree(L)
InOrderD(T, ' ')
print()
PrintDepths(T)
```

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.