CS 2302 Data Structures

Lab Report #3

Due: October 21, 2020
Professor: Olac Fuentes
TA: Anindita Nath
Student: Manuel Gutierrez Jr

Introduction

For this lab, we were asked to work with python List, Binary Trees, time complexity and code methods that use this data structure to print out specific elements. For this lab, it is key to remember the basics of how List references work and how you can use them to implement different data structures such as Binary Trees, It is important to note how to translate implementation from every data structure. List are required for every problem in the lab. The main objective of this lab is to get a deeper understanding of how references work and how different Data Structures such as Trees have different time complexity and IMplementation but can be interrelated to each other.

Proposed Solution Design and Implementation

Note : Anytime I use the words right node and left node i will be referring to T[1] and T[2] respectively as this assignment is easier if you see the list as trees rather than list

Operation #1:

The problem size was solved by traversing every possible Node in the list seen as the list was divided into T[0] which represented the head T[1] represented the left node and T[2] represented the right node by calling them recursively I could traverse all the possible nodes within th elist and returning an integer value.

Operation #2:

The problem Minimum was solved by traversing only the left side of the Binary Tree and finding the leftmost Node. This was achieved by passing the left node or in this case T[1] every single time till T[1] was None then we would return the head of that list

Operation #3:

The problem Maximum was very similar the problem minimum and all I did differently was traverse the right side of the list (T[2]) till i was at the right most node or in this case when T[2] was None

Operation #4:

The problem Height was solved by traversing every possible height and returning the biggest one I did this with two recursive calls that would assign the the left side of the tree with a number and the right side of the tree with a number at the end it would compare the size and it would return the largest of the two numbers

Operation #5:
The problem inTree was fairly logical as we can solve this in O(log n) if the tree is balanced or O(n) if the tree is not balanced but we accomplis this by traversing the list based on the value if the value is greater than the head that means it will be ing the right portion of the tree if the value is less than the root it will be in the left portion of the tree this logic is repeated till the value is found or the last node is reached

Operation #6:
Printing by level is achieved by appending the T[1] and T[2] by doing so the stack can pop the left node and the right node at every level and simply pop them and print them.THis is done with two append functions

Operation #7:
The problem leaves was exactly like printing by levels but an additional conditional needed to be made I used the condition that if both of the left and right portion were None then it was a leaft and I would append it to the list after that i woul duse two recursive calls that would concatanate the list every time so only values wold show at the very end

Operation #8:
The problem itemsAtDepthK was just like the problem 7 and 6 all that was needed was an extra conditional that checked if we had reached the desired depth after that we used two recursive calls to get every node at that depth

Operation #9:
The problem depthOfK was just like find wbut with an additional conditional adn a integer return statement so we did the same as find and compare the value of k to see if it was bigger or smaller than the node after that we traversed the tree accordingly and tried to find the value K and the return the desired value with a counter to see how many times we had to traverse a node this can be done in O(log n) if the tree is balanced.

Operation #10:
The problem draw was implemented by manipulating the given code in the bst file by changing the aspect of the left node and right node in the bst to match the left and right nodes of a list it could easily be implemented.

Operation #11:
The problem Tree_toList_stacks was rather difficult and I needed to look at previously written code for reference but i figured i could traverse to the left most node and append to the list atht

way. I would do a sorted traversal by booking for the left most node and working my way up to the right most node

Operation #12:
The problem is full was very similar to leaves so i woul duse the same conditional and checked if the children contained leaves that were full and if they all did i ould return true but if it had a missing element i would return false

Operation #13:
The problem is perfect is very similar to is full but i just used the code we had written for one of the exercises and manipulated it to fit my needs by creating helper methods height and number of nodes i could create an equation that could determine if the list was perfect

Operation #14:
The problem delete was by far the hardest problem to solve and this took careful segmentation of the method delete that was given in the bst file using that and print statements to debut i used never examples to test the method and come up with all the required modification for the delete method to sun on the list

Experimental Results

Note : Anytime I use the words right node and left node i will be referring to T[1] and T[2] respectively as this assignment is easier if you see the list as trees rather than list

Operation #1:

```
print("question 1")
print(size(T)) [11, 6, 7, 16, 17, 2, 4, 18, 14, 8, 15, 1, 20, 13]
print(size(emptyTree)) []
print(size(unbalancedTree)) [1,2,3,4,5,6]
```

Time complexity is O(n) as all nodes must be traversed to determine the size of the list
I used multiple testing scenarios to depict if it truly needed to traverse every node to determine the list is empty the size of it will be zero as depicted by my implantation

Operation #2,#3:

```
print("question 2")
print(minimum(T))
print(minimum(unbalancedTree))
```

I used a balanced tree and and unbalanced tree here to depict the difference in time complexity for the balanced tree the complexity will be O(log n) as every time it traverses a node it will cost it half every as it splits the left node and the right node every time when searching fo the minimum. This is also true for maximum

Operation #4:

```
print("question 4")
print(height(T))
print(height(emptyTree))
print(minimum(unbalancedTree))
```

I sued a variety of test cases here the basic ones where having a empty tree and a balanced tree because i wanted to check the case when it was empty it should return negative one i took this into consideration and made the assumption that there will be one and added that as an addition as in the end it will subtract one from the total the time complexity for this is O(n) in both cases or O(1) if the tree is empty

Operation #5:

```
print(inTree(T, 1))
print(inTree(T, 4))
print(inTree(T, 5))
print(inTree(emptyTree, -1))
print(inTree(unbalancedTree, 5))
```

This is similar to finding the maximum and minimum and is the same process with an additional conditional. In this case i choose a balanced tree and and unbalanced tree to depict the time complexity difference. For the balanced tree it will take O(log n) for the unbalanced tree it will take O(n)

Operation #6:

```
print("question 6")
printByLevel(T)
printByLevel(emptyTree)
printByLevel(unbalancedTree)
```

Print by level will take O(n) no matter if it is balanced or not as it needs to print out every element based on the level.

Operation #7:

```
print("question 7")
print(tree_to_list_stack(T))
print(tree_to_list_stack(testTree))
print(tree_to_list_stack(emptyTree))
print(tree_to_list_stack(unbalancedTree))
```

Tree to List has a time complexity of O(n) because it used stacks instead of recursion needing only a while loop to traverse the entire list in an ordered function

Operation #8,#9,#10:

```
print("question 8")
print(leaves(T))
print(leaves(testTree))
print(leaves(emptyTree))
print(leaves(unbalancedTree))

print("question 9")
print(itemsAtDepthD(T, 0))
print(itemsAtDepthD(T, 1))
print(itemsAtDepthD(T, 2))
print(itemsAtDepthD(T, 3))
print(itemsAtDepthD(T, 4))
print(itemsAtDepthD(unbalancedTree, 3))

print("question 10")
print(depthOfK(T, 10))
print(depthOfK(T, 11))
print(depthOfK(T, 20))
print(depthOfK(emptyTree, 0))
print(depthOfK(unbalancedTree, 3))
```

All of the following are very similar and provide the same complexity they will either be O(log n) if they are balanced or O(n) if they are not balanced simply because when searching a binary tree it divided the searchers by two every single search

Operation #11:

```
print("question 11")
draw(T)
draw(testTree)
draw(emptyTree)
draw(unbalancedTree)
```

Draw has a complexity of O(n) simply because all nodes need to be printed and two recursive calls are made to draw them

Operation #12:

```
print("question 12")
print(is_full(T))
print(is_full(unbalancedTree))

print("question 13")
print(is_perfect(T))
print(is_perfect(unbalancedTree))
```

Twelve and Thirteen are very similar and have the sma complexity you must traverse every node to find out if they are full and you must calculator th eamount of nodes and the height for it to be considered perfect

Operation #14:

```
print("question 14")
print(find_node_and_parent(testTree,11,9))# working fine
print(delete(testTree, 9))
print(delete(testTreeDelete, 8))
print(delete(T, 6))
print(delete(emptyTree, 0))
print(delete(unbalancedTree, 3))
```

The deleted method was especially troublesome and I had to use multiple cases to test out every individual part the complexity of is the same as insert and will take O(log n) if the tree is balanced or O(n) if the tree is not balanced

Conclusion

This lab helped me have a better understanding of howlists behave and how references are used in dealing with meeting one data structure to another. Overall I thought this lab was actually pretty easy but it was a lot of work and a lot of smaller problems that when segmented could be achieved with a couple lines of code.I learned the importance of different implementations based on the data structure you are working on. It helped me have a deeper understanding of how lists can make up various data structures including but not limited to BST. Over all this was a fun lab that helped understand trees and list better

```
# Implementation of binary search trees using lists
import matplotlib.pyplot as plt
import numpy as np

def insert(T,newItem): # Insert newItem to BST T
   # Running time is O(log n) for a balanced tree
   if T == None:  # T is empty
       T = [newItem,None,None]
   else:
       if newItem< T[0]:
           T[1] = insert(T[1],newItem) # Insert newItem in left subtree
       else:
           T[2] = insert(T[2],newItem) # Insert newItem in right subtree
   return T

def inOrder(T):
   # Prints keys in the tree in ascending order
   # Running time is O(n) for a balanced or unbalanced tree
   if T!=None:
       inOrder(T[1])
       print(T[0],end=' ')
       inOrder(T[2])

#O(n)
```

```
def height(T): # MAY be used by is_perfect(t)
   if T == None:
       return -1
   return max(height(T[1]),height(T[2]))+1

#O(n)
def count_nodes(T): # MAY be used by is_perfect(t)
   count = 0
   Stack = [T]
   while len(Stack)>0:
       T = Stack.pop(0)
       count = count + 1
       if T!=None:
           Stack.append(T[1])
           Stack.append(T[2])
   return count

#O(n)
def size(T):
   if(T == None):
       return 0
   return 1 + size(T[1]) + size(T[2])

#O(n)
def minimum(T):
   if(T[1] == None):
       return T[0]
   return minimum(T[1])

#O(n)
def maximum(T):
   if(T[2] == None):
       return T[0]
   return maximum(T[2])

#O(n)
def height(T):
   if T == None:
       return -1
```

```
    heightOfLeft = 1 + height(T[1])
    heightOfRight = 1 + height(T[2])

    if heightOfLeft > heightOfRight:
        return heightOfLeft
    return heightOfRight

#O(log n) in a balanced tree
def inTree(T,i):
    if(T == None):
        return False
    if(i == T[0]):
        return True
    if(i > T[0]):
        return inTree(T[2], i)
    return inTree(T[1], i)

#O(n)
def printByLevel(T):
    Stack = [T]
    while len(Stack)>0:
        T = Stack.pop(0)
        if T!=None:
            print(T[0],end = ' ')
            Stack.append(T[1])
            Stack.append(T[2])
    print()

#O(n)
def leaves(T):
    L = []
    if T != None:
        if T[1] == None and T[2] == None:
            L.append(T[0])
            return L
        else:
            L = L + leaves(T[1])
            L = L + leaves(T[2])
    return L
```

```
#O(n)
def itemsAtDepthD(T,d):
    L = []
    if T==None:
        return []
    if d == 0:
        L.append(T[0])
    return L + itemsAtDepthD(T[1],d-1) + itemsAtDepthD(T[2],d-1)

#O(log n) in  a balanced tree
def depthOfK(T,k):
    if T==None:
        return -1
    if T[0]==k:
        return 0
    child = T[1]
    if T[0]<k:
        child = T[2]
    d = depthOfK(child,k)
    if d>=0:
        d+=1
    return d

# draw akes O(1) as its just a wrapperfunction but draw1 takes O(n) and over all the
draw functions take O(n)
def draw(T):
    fig, ax = plt.subplots()
    if T != None:
        draw1(T,ax, 0, 0, 1000, 120)
    ax.axis('off')
    plt.show()

def draw1(T, ax, x0, y0, delta_x, delta_y):
    delta_x = max([20,delta_x])
    if T[1] is not None:
        ax.plot([x0-delta_x,x0],[y0-delta_y,y0],linewidth=1,color='k')
        draw1(T[1],ax, x0-delta_x, y0-delta_y, delta_x/2, delta_y)
    if T[2] is not None:
```

```
        ax.plot([x0+delta_x,x0],[y0-delta_y,y0],linewidth=1,color='k')
        draw1(T[2],ax, x0+delta_x, y0-delta_y, delta_x/2, delta_y)
    ax.text(x0,y0, str(T[0]), size=14,ha="center", va="center",
        bbox=dict(facecolor='w',boxstyle="circle"))
#O(n)
def tree_to_list_stack(T):
    current = T
    stack = []
    L = []
    while True:
        if current != None:
            stack.append(current)
            current = current[1]
        elif(stack):
            current = stack.pop()
            L.append(current[0])
            current = current[2]
        else:
            break
    return L

#O(n)
def is_full(T):
    if T == None:
        return True
    if T[1] == None and T[2] is None:
        return True
    if T[1] != None and T[2] != None:
        return (is_full(T[1]) and (is_full(T[2])))
    return False

# height takes O(n) times adn count of nodes also takes O(n) times over all it will
take O(n) times
# even though the function only has if stamenets an assignments
def is_perfect(T):
    h = height(T)
    n = count_nodes(T)
    if h == -1 or n ==0:
        return True
```

```
    if h >= 0:
        equation = (2**(h+2))-1
    if n == equation:
        return True
    return False

# O(log n) in a balanced tree
# for an unballeanced tree worst case scenario will be O(n) best case O(1) if the tree is empty
def delete(T,key):

    '''
    #cheatingall this does is make the node nll but never deletes it
    if T==None:
        return T
    if T[0]==k:
        T[0] = None
        return T
    child = T[1]
    if T[0]<k:
        child = T[2]
    delete(child,k)
    '''

    # start of fuentes code
    if T == None:
        print('Trying to delete from empty tree')
        return -1
    node_to_delete, parent = find_node_and_parent(T,None,key) # Returns reference to node to delete and its parent
    if node_to_delete == None:
        print('Trying to delete key that is no in the tree')
        return -1
    num_children = int(node_to_delete[1]!=None) + int(node_to_delete[2]!=None) # Returns reference to node to delete and its parent
    if num_children==0: # key is in a leaf node
        if parent==None: # Deleting root, which is the only node in the tree
            T[0] = None
        elif parent[1] == node_to_delete:
```

```
            parent[1] = None
        else:
            parent[2] = None
        # at any point here you can return the tree

    elif num_children==1: # key is in a node that has one child
        child = node_to_delete[1]
        if child==None:
            child = node_to_delete[2]
        if parent==None: # Deleting root
            T[0] = child
        elif parent[1] == node_to_delete:
            parent[1] = child
        else:
            parent[2] = child

    # this part needs to be solved
    else:  # key is in a node that has two children
        t =  node_to_delete[2]
        while t[1]!=None:     # Find key's successor
            t=t[1]
        successor = t[0]
        delete(T,successor)  # Delete key's successor
        node_to_delete[0] = successor   # Copy successor to node that contains key
(thus deleting key)
    return T #at the end return the tree with the deleted node

#O(log n) if the tree is balanced
#O(n) in a worst case or O(1) in the best of cases
def find_node_and_parent(T,parent,key):
    # Same as find, but it also returns the parent of the node that contains key
    # if key is in the tree
    if T[0] == key:
        return T, parent
    if T[0] > key:
        child = T[1]
    else:
        child = T[2]
    if child == None:
```

```
        return None, T
    else:
        return find_node_and_parent(child,T,key) # in order to progress T becomes the
parent


if __name__ == "__main__":
    #list used for testing
    A =[11, 6, 7, 16, 17, 2, 4, 18, 14, 8, 15, 1, 20, 13]
    B = [11, 9, 12]
    C = [11, 9, 12, 8]
    D = [1,2,3,4,5,6]

    #trees used for testing
    T = None
    testTree = None
    testTreeDelete = None
    emptyTree = None
    unbalancedTree = None

    # to make the list it takes
    for a in A:
        print('Inserting',a)
        T = insert(T,a)
        print(T)

    for b in B:
        print('Inserting',b)
        testTree = insert(testTree,b)
        print(testTree)

    for c in C:
        print('Inserting',c)
        testTreeDelete = insert(testTreeDelete,c)
        print(testTreeDelete)

    for d in D:
        print('Inserting',d)
        unbalancedTree = insert(unbalancedTree,d)
```

```
    print(unbalancedTree)



inOrder(T)

print("\n")
print("question 1")
print(size(T))
print(size(emptyTree))
print(size(unbalancedTree))

print("question 2")
print(minimum(T))
print(minimum(unbalancedTree))

print("question 3")
print(maximum(T))
print(maximum(unbalancedTree))

print("question 4")
print(height(T))
print(height(emptyTree))
print(minimum(unbalancedTree))
 print("question 5")
print(inTree(T, 1))
print(inTree(T, 4))
print(inTree(T, 5))
print(inTree(emptyTree, -1))
print(inTree(unbalancedTree, 5))

print("question 6")
printByLevel(T)
printByLevel(emptyTree)
printByLevel(unbalancedTree)

print("question 7")
print(tree_to_list_stack(T))
print(tree_to_list_stack(testTree))
```

```
print(tree_to_list_stack(emptyTree))
print(tree_to_list_stack(unbalancedTree))

print("question 8")
print(leaves(T))
print(leaves(testTree))
print(leaves(emptyTree))
print(leaves(unbalancedTree))

print("question 9")
print(itemsAtDepthD(T, 0))
print(itemsAtDepthD(T, 1))
print(itemsAtDepthD(T, 2))
print(itemsAtDepthD(T, 3))
print(itemsAtDepthD(T, 4))
print(itemsAtDepthD(unbalancedTree, 3))

print("question 10")
print(depthOfK(T, 10))
print(depthOfK(T, 11))
print(depthOfK(T, 20))
print(depthOfK(emptyTree, 0))
print(depthOfK(unbalancedTree, 3))

print("question 11")
draw(T)
draw(testTree)
draw(emptyTree)
draw(unbalancedTree)

print("question 12")
print(is_full(T))
print(is_full(unbalancedTree))

print("question 13")
print(is_perfect(T))
print(is_perfect(unbalancedTree))

print("question 14")
```

```
print(find_node_and_parent(testTree,11,9))# working fine
print(delete(testTree, 9))
print(delete(testTreeDelete, 8))
print(delete(T, 6))
print(delete(emptyTree, 0))
print(delete(unbalancedTree, 3))
```

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