Robert Marc

80487972

CS 2302

MW 1:30-3:00

Lab 3 Report

Introduction

The purpose of this lab was to implement efficient ways of displaying, searching, building lists from binary search trees, and binary search trees from lists.

Proposed Solutions

The first problem had us plot a diagram of a binary tree with the appropriate number of branches and nodes. This was simple, as we had done something similar in lab 1. The method checks the current node for a left and right branch. If there is one, it draws a line to the next ‘node’. It then draws the current node and the item stored in that node. It then recursively passes through the tree. If it reaches a node without any branches, it only draws the node.

The second problem had us implement an iterative version of searching a binary search tree. It uses a current node place holder, starting with the root of the tree. It checks the current node’s item against the search key. If less, the current node pointer gets changed to the current node’s left branch. If more, the pointer gets changed to the current node’s right branch. It proceeds in this way until it either finds the search key or reaches a node that doesn’t have a left or right branch.

The third problem had us build a balanced binary search tree from a sorted list. For this, I found the middle of the list and started the binary tree from that point. Slicing the list into two smaller lists, one with elements less than the middle, and one with elements greater than the middle. It would then recursively pass on the ‘left’ and ‘right’ lists, building a sorted, balanced tree.

The fourth problem required us to create a sorted list from a binary search tree. This method works like the method from problem 3, but in reverse. It recursively advances to the bottom of the list, and begins appending elements to the list. It always checks to see if a node has a left and/or right branch, and recursively calls appropriately.

The final problem required us to print a sort list of elements present at each depth of a given binary search tree. I used a main method and helper method to achieve this. The helper method builds an array by traversing the binary search tree in order, while passing on the depth of the node. It uses the depth as the row index for storage of elements.

Experimental Results

Average Number of Comparisons for a given N across 100 tests

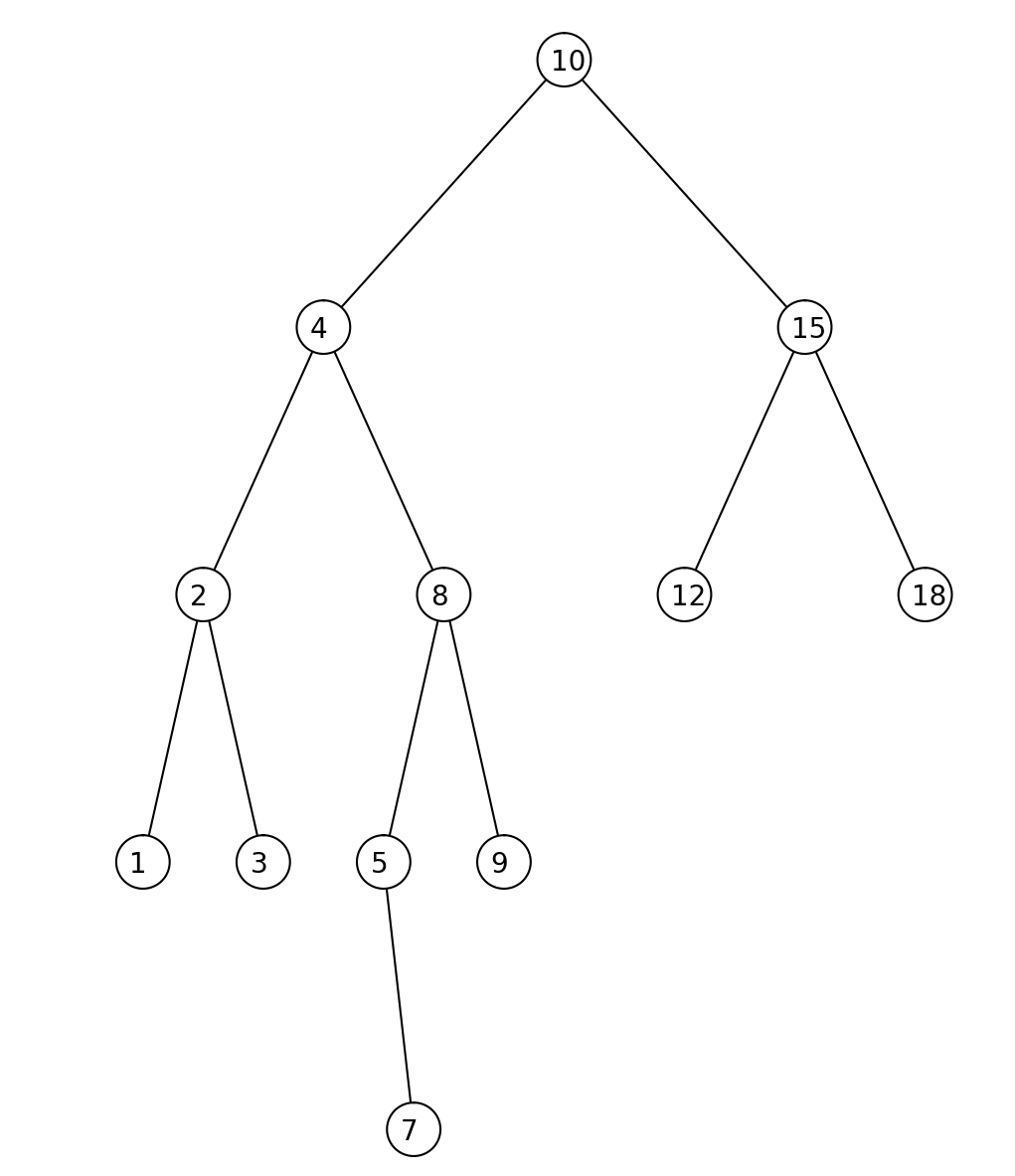
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N = | 8 | 16 | 32 | 64 | 128 | 256 |
| DrawTree | 8 | 16 | 32 | 64 | 128 | 256 |
| IterSearch | 3.58 | 5.12 | 5.93 | 7.5 | 8.95 | 10.73 |
| BuildFromList | 8 | 16 | 32 | 64 | 128 | 256 |
| BuildFromTree | 8 | 16 | 32 | 64 | 128 | 256 |
| KeysAtDepth | 16 | 32 | 64 | 128 | 256 | 512 |

DrawTree, BuildFromList, BuildFromTree all have Big-O = O(N)

KeysAtDepth has Big-O = O(2N)=O(N)

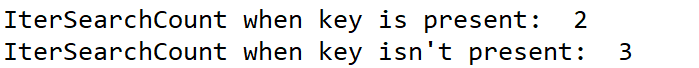
Outputs

Problem 1 (DrawTree)

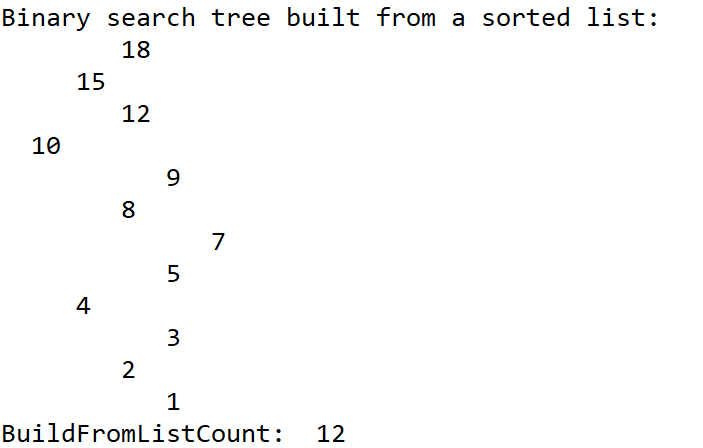


Problem 2 (IterSearch)

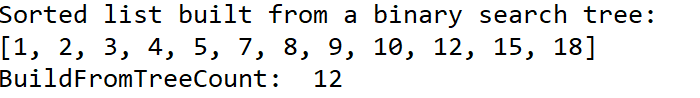
Search For Key In Tree and Search For Key NOT In Tree



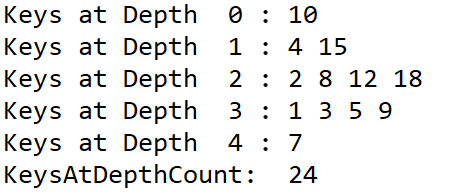
Problem 3(BuildFromList)



Problem 4(BuildFromTree)



Problem 5(KeysAtDepth)



Conclusions

This lab was pretty easy. I struggled with some parts of the implementation. For problem 1, I kept getting duplicate nodes, and couldn’t figure it out until I realized I was having the left and right branch nodes drawn alongside the branches themselves. For BuildFromTree, I was getting the elements added into the list properly, but had a whole bunch of ‘None’s between the elements. I realized this was a similar problem to the one I was having in problem 1. Other than that, I had no major issues completing this lab.

Academic Honesty Statement

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.”



Appendix

# Code to implement a binary search tree

# Programmed by Olac Fuentes

# Last modified February 27, 2019

"""

@Course: CS2302 MW 1:30-2:50 pm

@Author: Robert Marc, 80487972

@Assignment: Lab 3

@Instructor: Dr. Olac Fuentes

@TAs: Anindita Nath and Maliheh Zargaran

@Date of Last Modification: 3/8/19 @8:31 PM

@Purpose: To implement different methods using binary search trees

"""

import matplotlib.pyplot as plt

import numpy as np

import math

import random

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')

#Begin Student Code

def IterSearch(T,k):

"""

Iterative Search of a Binary Tree

searches tree T for key k

If found, returns the first note the key can be found in

If not, returns None

Begins with the head of the tree and compares k to the item in the current node

If k > current item, advances current node to the right

If K < current item, advances current node to the left

"""

global IterSearchCount

cur = T

while cur != None:

if k > cur.item:

IterSearchCount += 1

cur = cur.right

elif k < cur.item:

IterSearchCount += 1

cur = cur.left

else:

return cur

return None

def BuildFromList(L):

"""

Build a balanced binary search tree from a sorted list, L

Find the element in the middle of the list and adds that element to the tree

Split the remainder of the list into two, slicing out the middle element

Recursively call BuildFromList using the sliced lists to build the left and right branches of the tree

then returns the head of the Tree

"""

global BuildFromListCount

if len(L) != 0:

BuildFromListCount += 1

mid = len(L)//2

T = BST(L[mid])

L1 = L[:mid]

L2 = L[mid+1:]

T.left = BuildFromList(L1)

T.right = BuildFromList(L2)

return T

return

def BuildFromTree(T):

"""

Build a sorted list from a binary search tree, T

Starting at the head of the tree:

If the current node has a left and right branch:

Recursively calls BuildFromTree on the left branch while adding on the current elemnt to the middle

and recursively calls BuildFromTree on the right branch

If the current node does not have a left branch:

Recursively calls BuildFromTree on the right branch while adding the result to the current element

If the current node does not have a right branch:

Recursively calls BUildFromTree on the left branch while adding the middle element to the result

If the current node has no left or right branch

Returns the current element

This process builds the list by traversing the tree in order

"""

global BuildFromTreeCount

cur = T

if cur.left != None and cur.right != None:

BuildFromTreeCount += 1

return BuildFromTree(T.left) + [T.item] + BuildFromTree(T.right)

elif cur.left != None:

BuildFromTreeCount += 1

return BuildFromTree(T.left) + [T.item]

elif cur.right != None:

BuildFromTreeCount += 1

return [T.item] + BuildFromTree(T.right)

else:

BuildFromTreeCount += 1

return [T.item]

def KeysAtDepth(T):

"""

Main method for printing the keys at each depth

Uses a recursive helper method to build a 2x2 array with each key at each depth

Then uses two for loops to traverse the array while printing the keys

"""

k = []

KeysAtDepthHelper(T,0,k)

global KeysAtDepthCount

for i in range(len(k)):

print("Keys at Depth ",i,":",end=' ')

for j in range(len(k[i])):

KeysAtDepthCount += 1

print(k[i][j],end=' ')

print()

def KeysAtDepthHelper(T,d,k):

"""

Recursive helper method for building a 2x2 array with each key at each depth

Each row of the array represents the depth of the tree

Each column of the array represents the keys in that depth

Uses a similar method to building a sorted list from a tree

Instead of appending all elements to a 1-D list, it instead appends keys to the appropriate row for

the current depth

"""

global KeysAtDepthCount

if T != None:

KeysAtDepthCount += 1

if len(k)-1 < d:

k.append([])

KeysAtDepthHelper(T.left,d+1,k)

k[d].append(T.item)

KeysAtDepthHelper(T.right,d+1,k)

return

def Circle(center,rad):

"""

Simple method from lab 1 to calculate coordinates for plotting a 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 DrawNode(ax,center,radius,N):

"""

Method used to draw circles represnting nodes in a tree and the element in that node

"""

if N != None:

x,y = Circle(center,radius)

ax.plot(x,y,color='k',zorder=2)

ax.fill(x,y,color='w',zorder=2)

ax.text(center[0]-50,center[1]-40,str(N.item),fontsize=20)

def DrawTreeHelp(ax,T,c,w,l):

"""

Recursive helper method for drawing the lines between nodes of a tree

This method first recursively draws the appropriate number of lines between the nodes,

then draws the nodes at the intersections of these lines

"""

global DrawTreeCount

if T != None:

x1 = c[0]-(w\*.9),c[0]

x2 = c[0]+(w\*.9),c[0]

y = c[1]-l,c[1]

if T.left != None:

ax.plot(x1,y,color='k',zorder=1)

DrawTreeHelp(ax,T.left,[x1[0],y[0]],w/2,l)

if T.right != None:

ax.plot(x2,y,color='k',zorder=1)

DrawTreeHelp(ax,T.right,[x2[0],y[0]],w/2,l)

DrawTreeCount += 1

DrawNode(ax,c,100,T)

def DrawTree(T):

"""

Main method for drawing a tree diagram of a given tree, T

Sets up the subplots and original size of the plot as well as the starting coordinate of the tree

Then calls the recursive helper method to draw the tree

"""

global DrawTreeCount

if T != None:

plt.close("all")

orig\_size = 1000

c = [0,orig\_size]

fig, ax = plt.subplots()

DrawTreeHelp(ax,T,c,orig\_size,orig\_size)

ax.set\_aspect(1.0)

ax.axis('off')

plt.show()

fig.savefig('tree.png')

IterSearchCount = 0

BuildFromListCount = 0

BuildFromTreeCount = 0

KeysAtDepthCount = 0

DrawTreeCount = 0

#Binary search tree given to use in the lab documentation, used as a control for testing

T = BST(10)

Insert(T,4)

Insert(T,15)

Insert(T,2)

Insert(T,8)

Insert(T,12)

Insert(T,18)

Insert(T,1)

Insert(T,3)

Insert(T,5)

Insert(T,9)

Insert(T,7)

#Sorted list using the same elements as those in the tree from the lab documentation, used as a control for testing

SL = [10,4,15,2,8,12,18,1,3,5,9,7]

SL.sort()

DrawTree(T)

print("DrawTreeCount: ",DrawTreeCount)

n = IterSearch(T,12)

print(n.item)

print("IterSearchCount when key is present: ",IterSearchCount)

IterSearchCount = 0

n = IterSearch(T,17)

print("IterSearchCount when key isn't present: ",IterSearchCount)

TL = BuildFromList(SL)

print("Binary search tree built from a sorted list:")

InOrderD(T,' ')

print("BuildFromListCount: ",BuildFromListCount)

L = BuildFromTree(T)

print("Sorted list built from a binary search tree:")

print(L)

print("BuildFromTreeCount: ",BuildFromTreeCount)

KeysAtDepth(T)

print("KeysAtDepthCount: ",KeysAtDepthCount)

'''

#Testing Code

for k in range(0,100):

N = 128

T = BST(random.randint(0,(N\*N)))

for i in range(0,N-1):

Insert(T,random.randint(0,(N\*N)))

SL = []

for j in range(0,N):

SL.append(random.randint(0,(N\*N)))

SL.sort()

DrawTree(T)

IterSearch(T,random.randint(0,(N\*N)))

TL = BuildFromList(SL)

L = BuildFromTree(T)

KeysAtDepth(T)

DrawTreeCountAvg = DrawTreeCount/100

IterSearchCountAvg = IterSearchCount/100

BuildFromListCountAvg = BuildFromListCount/100

BuildFromTreeCountAvg = BuildFromTreeCount/100

KeysAtDepthCountAvg = KeysAtDepthCount/100

print("DrawTreeCountAvg = ",DrawTreeCountAvg)

print("IterSearchCountAvg = ",IterSearchCountAvg)

print("BuildFromListCountAvg = ",BuildFromListCountAvg)

print("BuildFromTreeCountAvg = ",BuildFromTreeCountAvg)

print("KeysAtDepthCountAvg = ",KeysAtDepthCountAvg)

'''