**Course Name:** 2302 **Author:** Olugbenga Iyiola **ID:** 80638542 **Instructor:** Olac Fuentes **TA:** Nath Anindita/ Malileh Zargaran **LAB #3 Report**

**Introduction**

The purpose of this lab is to implement basic binary search tree operations, including insertion, deletion, search and display. It also includes building a balanced binary search tree given a sorted list as input in O(n) time, extracting the elements in a binary search tree into a sorted list in O(n) time and printing the elements in a binary tree ordered by depth.

A binary search tree (BST), also known as an ordered binary tree, is a node-based data structure in which each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree.

**Proposed Solution Design and Implementation**

The binary search tree is implemented with singly linked list and the matplotlib is used to plot a graph for its display. The various operations are implemented in O(n) time as specified and the pseudocode for the various implementations are given below;

**Display the Binary Search Tree as a Figure**

If T is None

Return

If T.left is None and T.right is None

Plot text for BST root

If T.left is not None

Get coordinate for next left node

Plot text and line for BST to join root to left node

Make Recursive using left node as parameter

If T.right is not None

Get coordinate for next right node

Plot text and line for BST to join root to right node

Make Recursive using right node as parameter

**Iterative Version of the Search Operation**

If T is None or T.item == Search Item

Return T

While T is not None

If T.item is greater than Search Item

T = T.left

Else If T.item is lesser than Search Item

T = T.right

Else return T.item i.e current item == search item

**Building a Balanced Binary Search Tree Given a Sorted List as Input**

If T is None or length of T is zero or list start index is greater than

list last index

Get middle index of list

Create node of middle index item

Make recursive on left side of BST

Make recursive on right side of BST

**Extracting the Elements in a Binary Search Tree into a Sorted List**

If T is None

Return list index

List index = Recursive call with T.left

Fill array with T.item

Increment index

List index = Recursive call with T.right

Return list index

**Printing the Elements in a Binary Tree Ordered by Depth**

If T is None

Return 0

If level is Zero Print item for root

Print item for when:

T.left and T.right is not None

T.left is None and T.right is not None

T.left is not None and T.right is None

Next level for left tree = Recursive with T.left

Next level for right tree = Recursive with T.right

Return level

**Experimental Result**

System Specification: HP Windows 10, 1.60GHZ Intel® Celeron® , 4.GB RAM, 64-bit operating system

The results of the various test cases for each of the algorithms are shown below:

**Iterative Version of the Search Operation**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in nanoseconds** |
| **14** | **222727** |
| **64** | **142084** |
| **140** | **143364** |
| **280** | **138884** |

Running time is O(n)

**Building a Balanced Binary Search Tree Given a Sorted List as Input**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in nanoseconds** |
| **14** | **123485236** |
| **64** | **454179131** |
| **140** | **1123215147** |
| **280** | **2153144348** |

Recurrence Equation: T(n) = 2T(n/2) + 1 which, using the master theorem, gives us O(n).

**Extracting the Elements in a Binary Search Tree into a Sorted List**

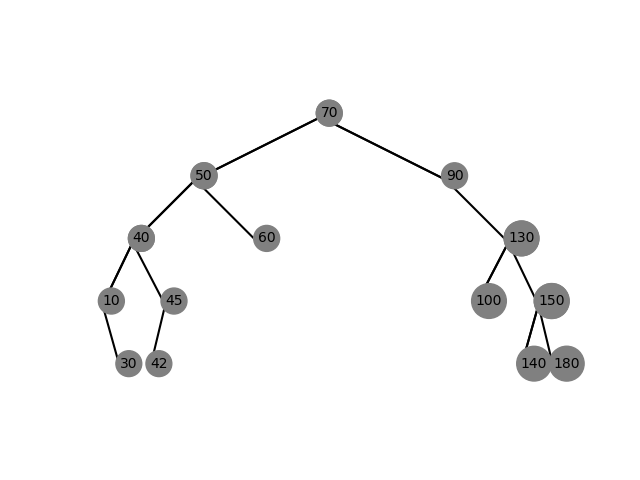
|  |  |
| --- | --- |
| **N(Input)** | **Runtime in Nanoseconds** |
| **14** | **1228836** |
| **64** | **2977368** |
| **140** | **29504869** |
| **280** | **11579860** |

Recurrence Equation: T(n) = 2T(n/2)+ 1 which, using the master theorem, gives us O(n).

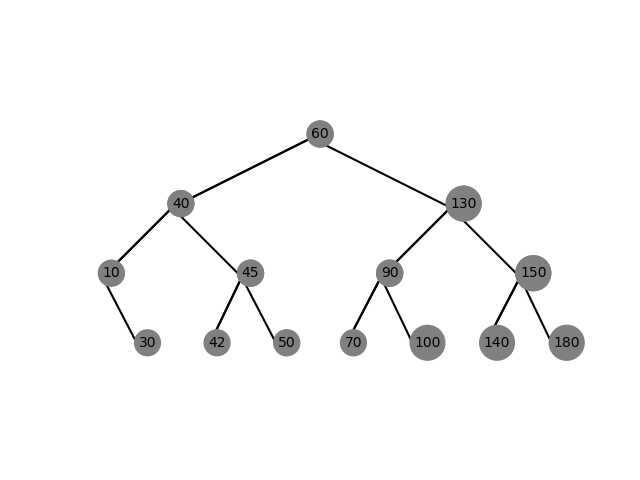
**Printing the Elements in a Binary Tree Ordered by Depth**

|  |  |
| --- | --- |
| **N(Input)** | **Runtime in Nanoseconds** |
| **14** | **2076861** |
| **64** | **1808053** |
| **140** | **1645489** |
| **280** | **1558446** |

Recurrence Equation: T(n) = 2T(n/2)+ 1 which, using the master theorem, gives us O(n).

**s**

**Figure of Binary Search Tree**



**Figure of Balanced Binary Search Tree**

**CONCLUSION**

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient; they are also easy to code. Having a sorted array is useful for many tasks because it enables binary search to be used, to efficiently locate elements. The problem with a sorted array is that elements can't be inserted and removed efficiently, however with a self-balancing binary search tree (BST), all of the above run in logarithmic time.

**Appendix**

***Programmed by Olac Fuentes***

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')

# Code to test the functions above

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)

InOrder(T)

print()

InOrderD(T,'')

print()

print(SmallestL(T).item)

print(Smallest(T).item)

FindAndPrint(T,40)

FindAndPrint(T,110)

n=60

print('Delete',n,'Case 1, deleted node is a leaf')

T = Delete(T,n) #Case 1, deleted node is a leaf

InOrderD(T,'')

print('####################################')

n=90

print('Delete',n,'Case 2, deleted node has one child')

T = Delete(T,n) #Case 2, deleted node has one child

InOrderD(T,'')

print('####################################')

n=70

print('Delete',n,'Case 3, deleted node has two children')

T = Delete(T,n) #Case 3, deleted node has two children

InOrderD(T,'')

n=40

print('Delete',n,'Case 3, deleted node has two children')

T = Delete(T,n) #Case 3, deleted node has two children

InOrderD(T,'')

**Wikipedia**

[**https://en.wikipedia.org/wiki/Sorting\_algorithm#Comparison\_of\_algorithms**](https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms)

**Academic Dishonesty**

This work was done by me without any act or practice of academic dishonesty

**SIGNATURE**

**OLUGBENGA IYIOLA(OT)**

**……………………………………………….**