LAB 3 Adrian Monreal

This lab was assigned to show our skills while using Balanced BST, we had to implement 5 different functions the first one was to display a binary search tree, create a iterative version of the search function, build a binary search tree from a sorted list, extract the elements from a bst and put them into a sorted list, and print the elements in order by their depth.

How I implemented and understood the iterative search function was while the element was not none ask if the desired number or element is bigger or smaller than the current node if its smaller go to the left branch and if its larger go to the right branch.

How I understood how to create the BST from a sorted list was go to the center of the sorted list and create that as your root node since it is directly in the middle of all the numbers, then all the nodes to the left were the first half of the list and all the nodes to the right were the second half of the list. It is a recursive method where there are 2 recursive calls one is for the first half of the list and the second call is second half of the list.

For creating a sorted list from a BST I went on a guess I didn’t really understand it that well I was torn between how a native list could be called in a recursive call so how what I did I made 3 recursive calls the first is append with the left branch the next is the current node and the last is the right branch.

|  |
| --- |
| #lab 3 |
|  | #Adrian Monreal |
|  | #805870881 |
|  |  |
|  |  |
|  | import numpy as np |
|  | import matplotlib.pyplot as plt |
|  |  |
|  | 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 Smallest(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 SmallestRec(T): #recursive |
|  | # 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') |
|  |  |
|  | #1 |
|  | def drawNode(center, rad, numberInside): |
|  | 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) |
|  | ax.plot(x, y, color='k') |
|  |  |
|  | def drawTree() |
|  |  |
|  | #2 |
|  | def iterative\_search(T, key): |
|  | t = T |
|  | while t != None: |
|  | if t.item == key: |
|  | return t |
|  | elif t.item < key: |
|  | t = t.right |
|  | else: |
|  | t = t.left |
|  |  |
|  | #3 |
|  |  |
|  | def BSTFromList(SortedList): |
|  | mid = SortedList[len(SortedList)/2] |
|  | newRoot= BST(mid) |
|  | midIndex = len(SortedList) / 2 |
|  | newRoot.left = BSTFromList(SortedList[0:midIndex-1]) |
|  | newRoot.right = BSTFromList(SortedList[midIndex+1:-1]) |
|  |  |
|  | #4 |
|  | def ListFromBST(T): |
|  | if T is None: |
|  | return None |
|  | t=T |
|  | sortedList = [] |
|  | sortedList.append(ListFromBST(T.left)) |
|  | sortedList.append(t) |
|  | sortedList.append(ListFromBST(T.right)) |
|  | return sortedList |
|  |  |
|  | #5 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | 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) |