Binary search trees sometimes called sorted binary trees are a type of container that stores data. This data structure can store data like numbers, names, and much more. The good part of it is that it allows us to lookup, add, or remove at time complexity of O(log n). Binary trees keep all their keys sorted in order to be able to use the principle of binary search. In this lab I will be creating 5 functions that work with binary trees. The first function is an iterative version of the search operation, next is to build a balanced tree given a sorted list as an input, next is extracting the elements in a binary tree into a sorted list, lastly is printing the elements in a binary search tree ordered by depth.

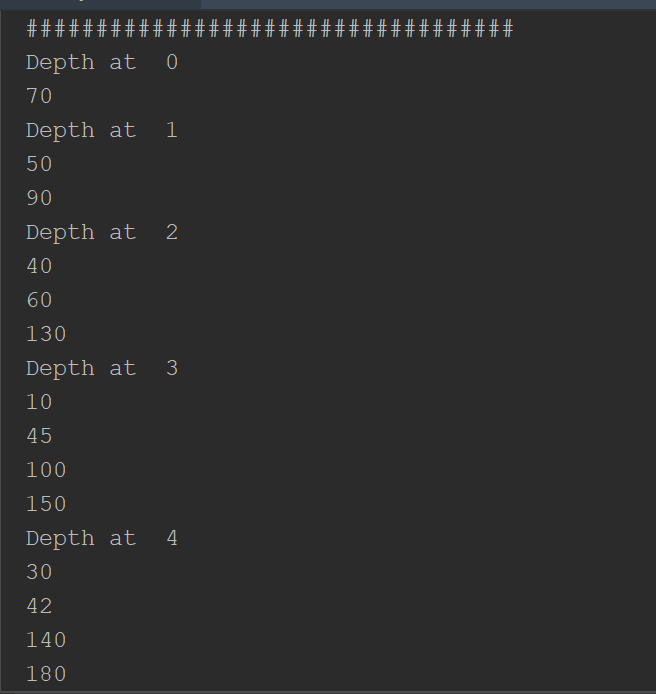
The first function that I worked on was to iteratively search for a key in a binary tree. For this we need to know how a binary search tree works. First a binary search tree is composed of objects called nodes. This objects store 3 variables. The first variable is the key or data that stores our information. The second variable is called left and this is a pointer to the next node that is less than the node you are in right now. Finally the third variable is called right and this is also a pointer to a node but this time it points to a node that is greater than the node you are in. Knowing this it was simple to create a iterative algorithm that looked for a key. In my program i created a function called Search() and it takes to parameters, ‘T’ that is the first node of the tree or root as most people call it and ‘t’ that is our target or the key that we are searching for. Inside the function i create a variable temp that stores my first variable. After using a while loop that checks if temp is not equal to none and that the data store in temp.item is not equal to our target then it will look in the tree for the item. It does this by either making temp equal to temp.left is temp.item is greater than our item or making temp equal to temp.right if temp.item is less than our target. If we find our target in the tree we return temp.item and if we dont we return None.

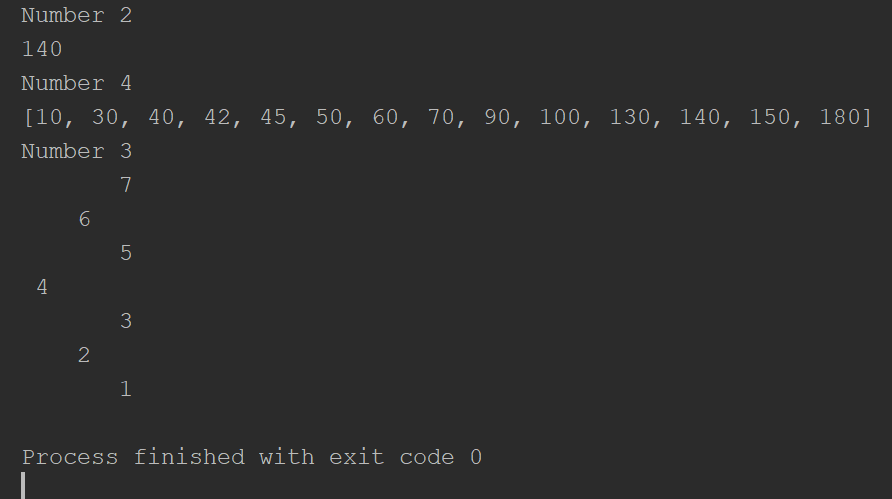
The second function is to build a tree using a sorted list as an input. The problem with building a binary search tree with a sorted list is that the time complexity of any method will then turn to the worse case that is O(n). Meaning that it would have to pass through each node to accomplish any function. To solve this problem I decided to take the sorted list and divided at the middle. What I mean by this is that I would find the middle element of the sorted list and add it as my first node the root. Then i would recursively called the left side of the list and the right side of the list and keep taking the middle node and adding it to my tree. To accomplish this i created a function called SortedListToBinaryTRee() that would take a sorted list as a parameter. After the list is passed if the list is empty then you return none. Otherwise if the list is not empty i find the middle index of the list and I create a node called root that stores my first item. After that I save the left side of the list in root.left by recursive call always inserting the middle element and doing the same with the right side of the list but saving it to root.right and then returning the first node called root.

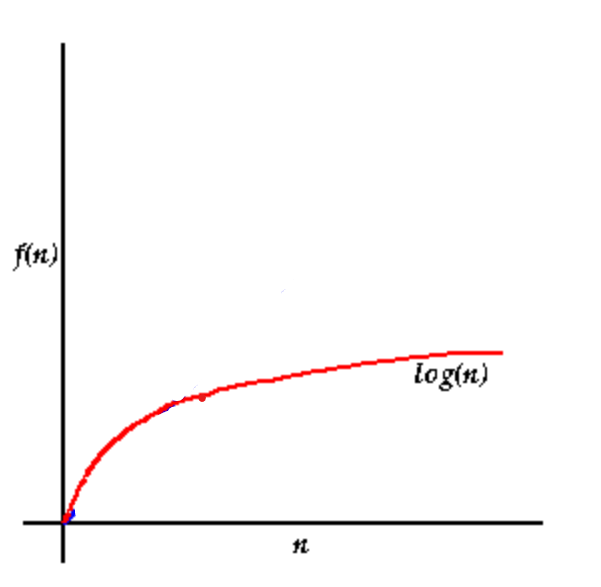
The third function was to extract the items in a binary tree and stored them in a list that should be sorted by the end. This was even more easy as I know how to traverse the binary tree in order. There is a function called InOrder() that will print the tree out in order from least to greatest. We do this by traversing to the left of the tree and then printing the nodes item and then traversing to right side of the tree and print thing the nodes item again. For this I created a function called BinaryTreeToSortedList() that takes two parameters ‘T’ that is a reference to my first node in my tree and ‘L’ a reference to an empty List. I then check if the tree is empty if so i just return the empty list back. If the tree is not empty the just like the function InOrder() i traverse the list to the left but now instead of printing the nodes item I append it to the empty list and then i do the same traversing to the right. At the end of the function I return the new sorted list back and save it to the previous empty list.

The last function is to print the elements of a tree by depth. The depth of a tree is referred to what level below the root or first node is the other node. For example the first node would be found at depth zero, but the nodes to the left and right of the first node would be found at depth one. For this i created a method that would print the elements at a certain depth. The function is called PrintDepth() that takes ‘T’ as my reference to my tree and ‘k’ that determine the depth that I want to print. Inside the function I check if my tree is empty if it is i just return else i recursively call my function by traversin to the left or to the right and subtraction one to my ‘k’. It will not print the nodes item until my depth value ‘k’ is equal to zero. The problem with this is that it will only print the depth level of a certain depth and not all the depth but to go around this problem i just created a for loop in my main method that would traverse for the range of depths that I want. If there are values at the depth of the index of the for loop then it would print the nodes items otherwise it would not print anything for that depth.

In conclusion I've become more accustomed to how binary trees work. I also learned how to print the values of a certain depth as before I had never even thought that it would be useful to implement this function. Also knowing how i can create a sorted list from a tree and a tree from a sorted tree will greatly help me in the future if I ever happen to encounter this problem. Overall this is one one of my favorite data structures as it is easy to use and can be executed with a time complexity of O(log n).







import time  
  
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')  
  
  
def PrintDepth(T,k):  
 if T is None:  
 return  
 if k == 0:  
 print(T.item)  
 else:  
 PrintDepth(T.left,k-1)  
 PrintDepth(T.right,k-1)  
  
  
def Search(T,t):  
 temp = T  
 while temp != None and t != temp.item:  
 if t < temp.item:  
 temp = temp.left  
 if t > temp.item:  
 temp = temp.right  
 if temp == None:  
 return None  
 return temp.item  
  
  
def BinaryTreeToSortedList(T,L):  
 if T is not None:  
 BinaryTreeToSortedList(T.left,L)  
 L.append(T.item)  
 BinaryTreeToSortedList(T.right,L)  
 return L  
  
  
def SortedListToBinaryTree(L):  
 if not L:  
 return None  
 mid = (len(L))//2  
 root = BST(L[mid])  
 root.left = (SortedListToBinaryTree(L[:mid]))  
 root.right = SortedListToBinaryTree(L[mid+1:])  
 return root  
  
  
# 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)  
  
  
print('Number 5')  
InOrderD(T,'')  
print('###################################')  
start = time.time()  
for x in range(5):  
 print('Depth at ', x)  
 PrintDepth(T,x)  
end = time.time()  
print('Time ',end-start)

Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone represents the work or ideas of another person as his/her own. Collusion involves collaborating with another person to commit an academically dishonest act. Professors are required to - and will - report academic dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

- Anthony Herrera