Samuel Chong

CS 2302

Olac Fuentes

3/26/19

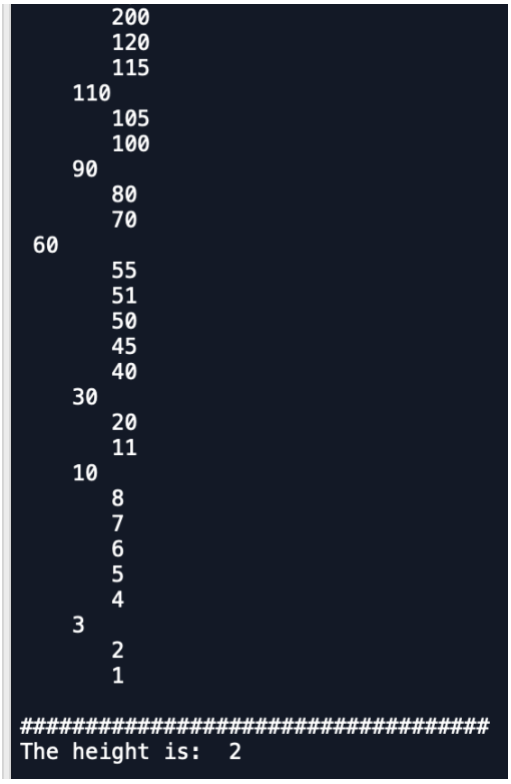
Lab4

Introduction:

Purpose of this lab was to learn how to use B-Trees. We computed the height, extract to a list, return the max, min values form the tree, return number of nodes per depth, print the nodes per depth, return the nodes that were full, and leafs also, and return where the key was stored in.

Height:

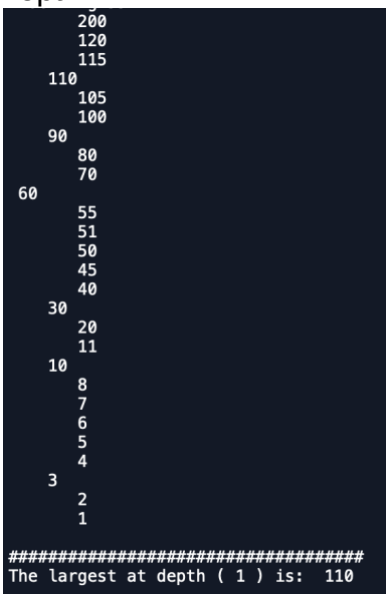
This method was provided by the professor and the way it works is that every time the recursion is made, you add one until reaching a leaf, which means you have reached the last level of the tree, so you return 0.



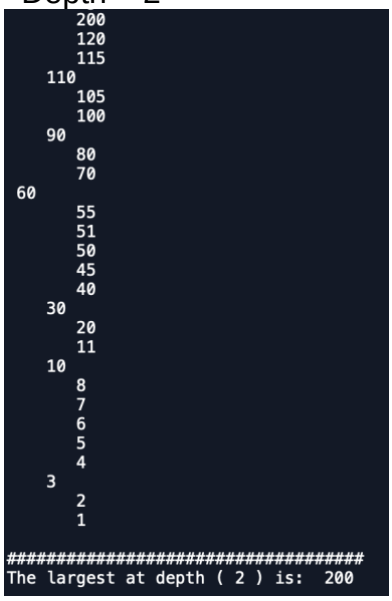
Largest:

If the depth was 0, then the method return the last position in the current item. It is a leaf or is out of bounce then return -1, else make the recursion with the last position of T. child and subtract 1 to depth until reaching the base case.

Depth = 1



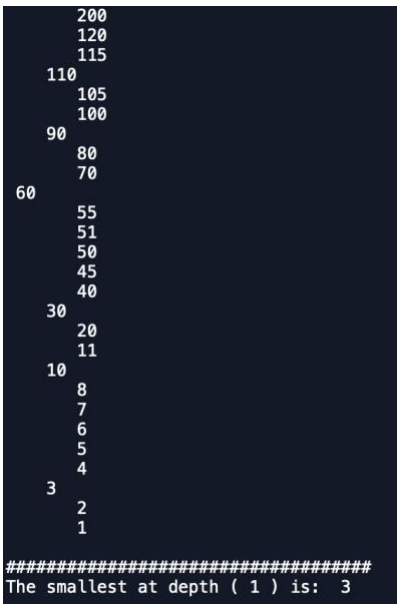
Depth = 2



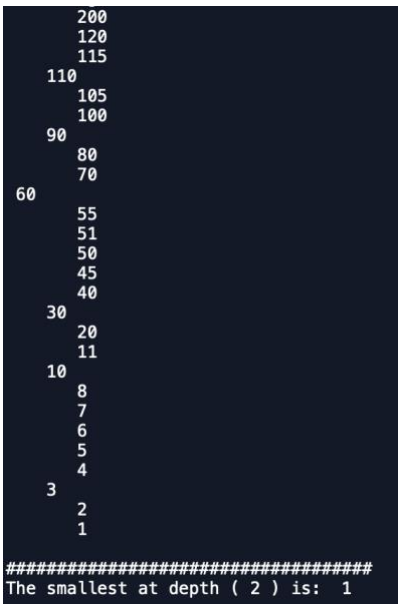
Smallest:

If the depth was 0, then the method return the first position in the current item. It is a leaf or is out of bounce then return -1, else make the recursion with the first position of T. child and subtract 1 to depth until reaching the base case.

Depth =1

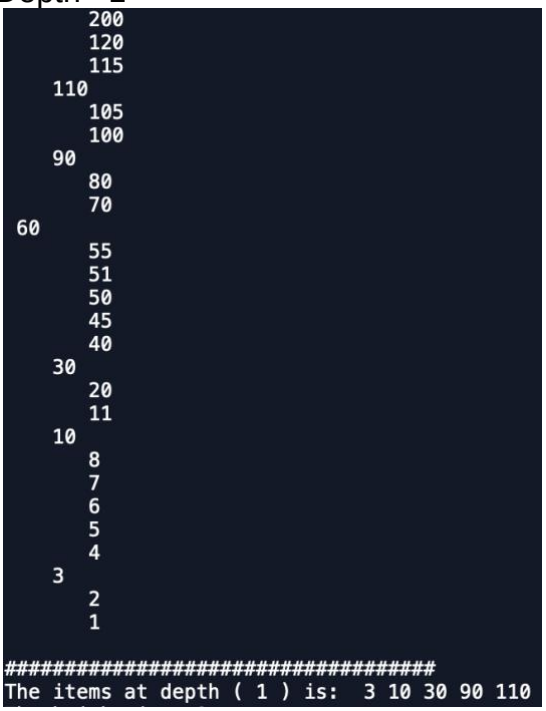


Depth=2

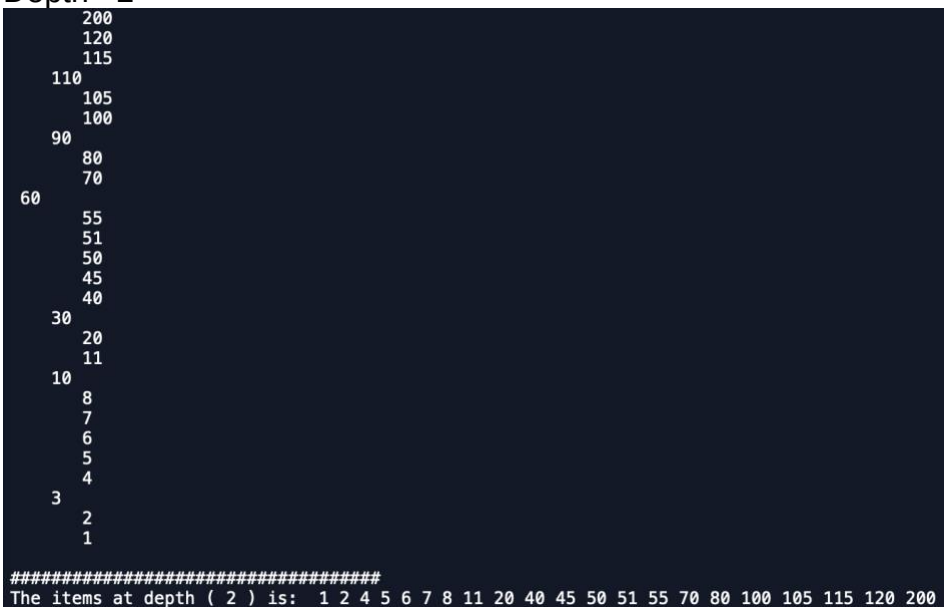


PrintAtDepth:
The method traverses the whole tree until we get to the depth we are looking for, if the base case is met then it prints every item in the current item using a for loop to traverse through the item. To traverse through the whole tree we use another for loop with the recursive call.

Depth =1

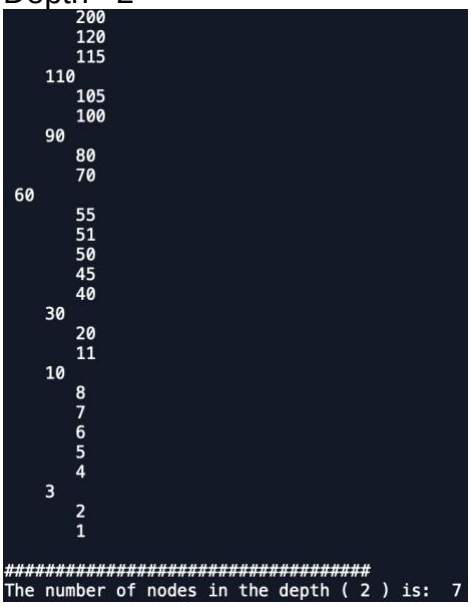


Depth =2

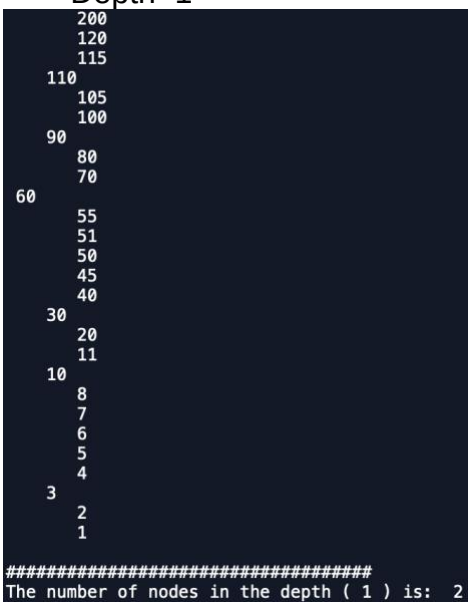


NodesAtDepth:
For this method we must traverse the whole tree as the previous method using a for loop and a recursive call. In order to keep track how many nodes are in the depth I used a variable and equaled it to the recursive call so the value won't be lost.

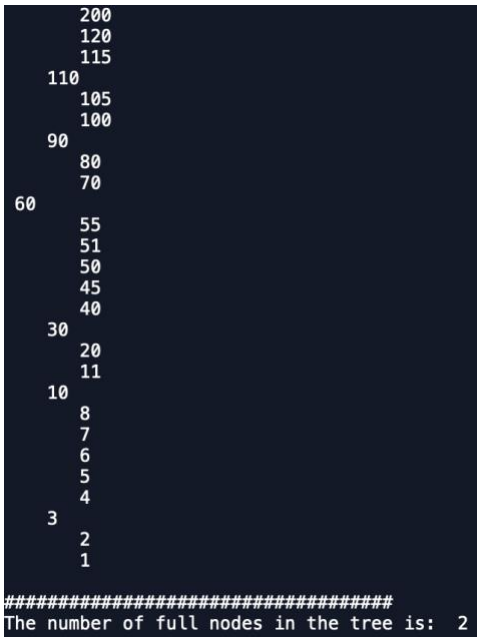
Depth =2



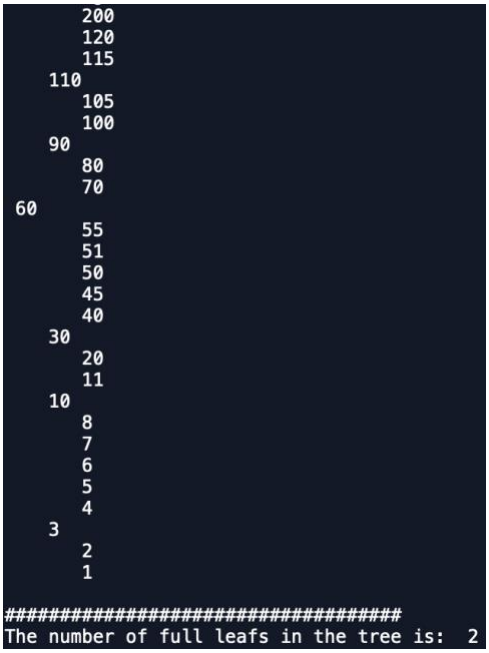
Depth=1



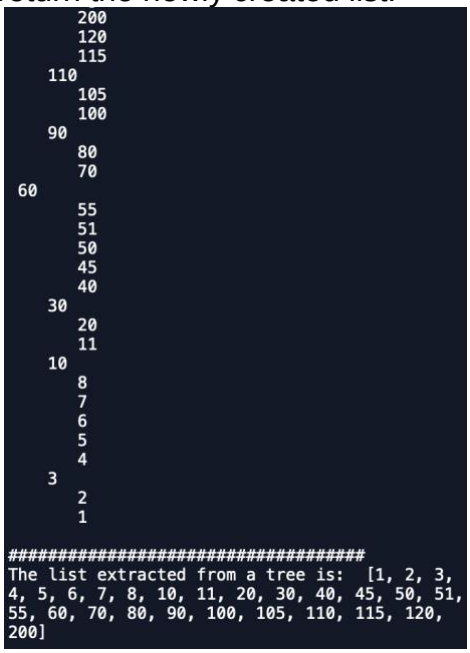
FullNodes:
This method traverses through the whole tree and it checks if the length of the current item is the same as max_items, if they are the same size then we add one to our counter.



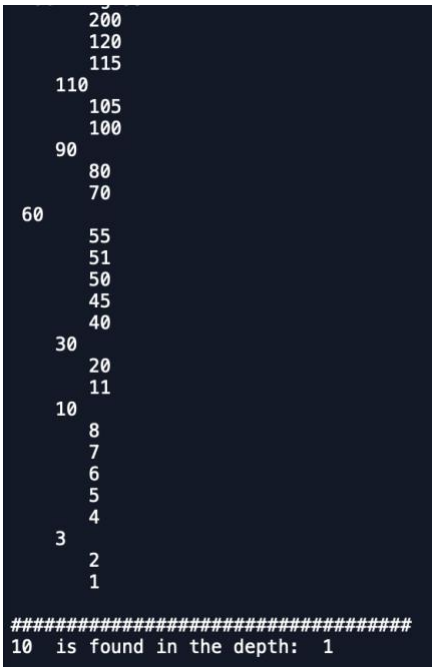
FullLeafs:
This method works the same as FullNodes but instead of only checking if the current item length is the same as the max_items, it must also be a leaf. It is a leaf and the length checks out, then add 1 to our counter and traverse to the next child.

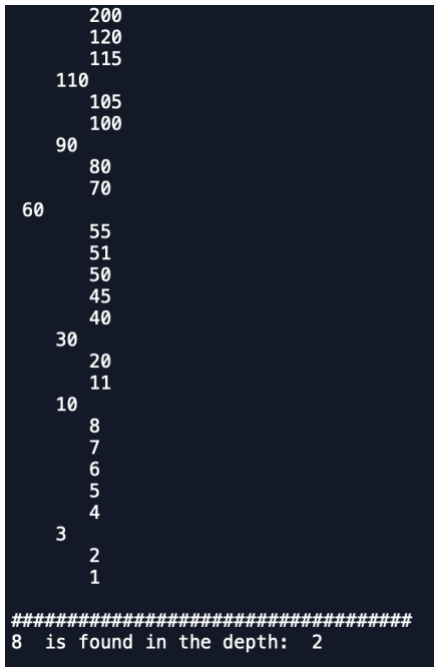


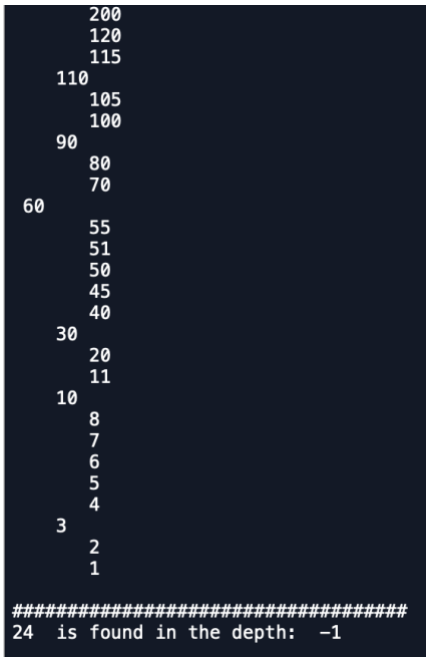
CreateList:
For this method we must traverse through the whole tree again using a for loop and the recursive call, we start with the 0 position (left most) because the smallest numbers are located there. If it is a tree then we append(add) the number that is in the current item, we do this by using a loop. After it has traversed through the whole tree, with must return the newly created list.



KeyAtDepth:
For this method we traverse the whole tree searching for the key, for every child we must check if the key is located there, if k was found then we return the depth it was located at. If we reach the leaf and k was not found then we return -1 as a depth because k was not in the tree.







Code:
import time

"""
@author: Samuel Chong
MW 10:30-11:50
Olac Fuentes
TAs: Anindita Nath and Maliheh Zargaran
3/25/2019
Purpose of this lab was to learn how to use B-Trees. We computed the height,
extract to a list, return the max, min values form the tree, return number of nodes per
depth,
print the nodes per depth, return the nodes that were full, and leafs also, and return
where the
key was stored in
"""

```
# Code to implement a B-tree
# Programmed by Olac Fuentes
# Last modified February 28, 2019


class BTree(object):
    # Constructor
    def __init__(self,item=[],child=[],isLeaf=True,max_items=5):
        self.item = item
        self.child = child
        self.isLeaf = isLeaf
        if max_items <=3: #max_items must be odd and greater or equal to 3
            max_items = 3
        if max_items%2 == 0: #max_items must be odd and greater or equal to 3
            max_items +=1
        self.max_items = max_items

def FindChild(T,k):
    # Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree
    for i in range(len(T.item)):
        if k < T.item[i]:
            return i
    return len(T.item)

def InsertInternal(T,i):
    # T cannot be Full
    if T.isLeaf:
        InsertLeaf(T,i)
    else:
        k = FindChild(T,i)
```

```
        if IsFull(T.child[k]):
            m, l, r = Split(T.child[k])
            T.item.insert(k,m)
            T.child[k] = l
            T.child.insert(k+1,r)
            k = FindChild(T,i)
        InsertInternal(T.child[k],i)

def Split(T):
    #print('Splitting')
    #PrintNode(T)
    mid = T.max_items//2
    if T.isLeaf:
        leftChild = BTree(T.item[:mid])
        rightChild = BTree(T.item[mid+1:])
    else:
        leftChild = BTree(T.item[:mid],T.child[:mid+1],T.isLeaf)
        rightChild = BTree(T.item[mid+1:],T.child[mid+1:],T.isLeaf)
    return T.item[mid], leftChild,  rightChild

def InsertLeaf(T,i):
    T.item.append(i)
    T.item.sort()

def IsFull(T):
    return len(T.item) >= T.max_items

def Insert(T,i):
    if not IsFull(T):
        InsertInternal(T,i)
    else:
        m, l, r = Split(T)
        T.item =[m]
        T.child = [l,r]
        T.isLeaf = False
        k = FindChild(T,i)
        InsertInternal(T.child[k],i)


def height(T):
    if T.isLeaf:
        return 0
    return 1 + height(T.child[0])


def Search(T,k):
```

```
    # Returns node where k is, or None if k is not in the tree
    if k in T.item:
        return T
    if T.isLeaf:
        return None
    return Search(T.child[FindChild(T,k)],k)

def Print(T):
    # Prints items in tree in ascending order
    if T.isLeaf:
        for t in T.item:
            print(t,end=' ')
    else:
        for i in range(len(T.item)):
            Print(T.child[i])
            print(T.item[i],end=' ')
        Print(T.child[len(T.item)])

def PrintD(T,space):
    # Prints items and structure of B-tree
    if T.isLeaf:
        for i in range(len(T.item)-1,-1,-1):
            print(space,T.item[i])
    else:
        PrintD(T.child[len(T.item)],space+'   ')
        for i in range(len(T.item)-1,-1,-1):
            print(space,T.item[i])
            PrintD(T.child[i],space+'   ')

def SearchAndPrint(T,k):
    node = Search(T,k)
    if node is None:
        print(k,'not found')
    else:
        print(k,'found',end=' ')
        print('node contents:',node.item)


def largestAtDepth(T,d):
    if d == 0:
        return T.item[-1] #return last element
    if T.isLeaf:
        return -1
    else:
        return largestAtDepth(T.child[-1], d-1) #traverse with last element
```

```
def smallestAtDepth(T,d):
    if d == 0:
        return T.item[0] #return the first position
    if T.isLeaf:
        return -1
    else:
        return smallestAtDepth(T.child[0], d-1) #traverse with the first positon



def printItemsInDepth(T,d):
    if d == 0: #if d = 0 print all the items in range of T.item
        for i in T.item:
            print(i, end=' ')
    else:
        for i in range(len(T.item)): #traverse the tree
            printItemsInDepth(T.child[i],d-1)
        printItemsInDepth(T.child[-1], d-1) #call the right part of the tree


def nodesAtDepth(T,d):
    if d == 0: #if depth 0 then return 1
        return 1
    else:
        count = 0
        for i in range(len(T.child)): #traverse the tree starting from position 0(left) and keep
moving
            count += nodesAtDepth(T.child[i],d-1)
        return count



def fullNodes(T):
    if T.isLeaf and len(T.item) == T.max_items:
        return 1 #isLeaf anf it reaches max_items return 1
    if len(T.item) == T.max_items:
        return 1 #return 1 if the size is the same as max_items
    else:
        counter = 0
        for i in range(len(T.child)): #traverse through the whole tree
            counter = counter + fullNodes(T.child[i])
        return counter


def fullLeafs(T):
    if T.isLeaf and len(T.item) == T.max_items:#isLeaf anf it reaches max_items return 1
```

```
        return 1
    else:
        counter = 0
        for i in range(len(T.child)): #traverse through each child of the tree
            counter = counter + fullLeafs(T.child[i])
        return counter

def createList(T,L):
    if T.isLeaf: #if it is a leaf then append all that are
        for i in T.item: #in size of T.item
            L.append(i)
    else:
        for i in range(len(T.item)):
            createList(T.child[i],L) #start appending from child 0 then keep iterating
            L.append(T.item[i])
        createList(T.child[-1],L) # append the last child
    return L


def keyAtDepth(T,k):
    if k in T.item: #if k is found
        return 0
    if T.isLeaf: #if it is a leaf and k was not found return -1
        if not k in T.item:
            return -1
    else:
        for i in range(len(T.child)): #traverse the whole tree from left to right
            d = keyAtDepth(T.child[i], k)
            if d != -1: #as long as the depth is not -1 then return the depth + 1
                return d + 1
        return -1




L = [30, 50, 10, 20, 60, 70, 100, 40, 90, 80, 110, 120, 1, 11 , 3, 4, 5,105, 115, 200, 2, 45,
6,
     7,8,51,55]
T = BTree()
for i in L:
    print('Inserting',i)
    Insert(T,i)
    PrintD(T,'')
    #Print(T)
    print('\n#########################################')
'''
```

```
SearchAndPrint(T,60)
SearchAndPrint(T,200)
SearchAndPrint(T,25)
SearchAndPrint(T,20)
'''
d = 1
newList = []



#height
start = time.time()
print("The height is: ", end=' ')
print(height(T))
end = time.time()
print()
print("Time for height is: ", end - start, "seconds")
print()
print()


#largestAtDepth
start = time.time()
print("The largest at depth (", d ,") is: ", end = ' ')
print(largestAtDepth(T,d))
end = time.time()
print()
print("Time for largestAtDepth is: ", end - start, "seconds")
print()
print()

#smallestAtDepth
start = time.time()
print("The smallest at depth (", d ,") is: ", end = ' ')
print(smallestAtDepth(T,d))
end = time.time()
print()
print("Time for smallest is: ", end - start, "seconds")
print()
print()

#printAtDepth
start = time.time()
print("The items at depth (", d ,") is: ", end = ' ')
printItemsInDepth(T,d)
end = time.time()
```

```
print()
print()
print("Time for printAtDepth is: ", end - start, "seconds")
print()

#nodesAtDepth
print()
start = time.time()
print("The number of nodes in the depth (",d,") is: ", end=' ')
print(nodesAtDepth(T,d))
end = time.time()
print()
print("Time for nodesAtDepth is: ", end - start, "seconds")
print()


#fullNodes
print()
start = time.time()
print("The number of full nodes in the tree is: ", end = ' ')
print(fullNodes(T))
end = time.time()
print()
print("Time for fullNodes is: ", end - start, "seconds")
print()

#fullLeafs
print()
start = time.time()
print("The number of full leafs in the tree is: ", end = ' ')
print(fullLeafs(T))
end = time.time()
print()
print("Time for fullLeafs is: ", end - start, "seconds")
print()

#createList

print()
start = time.time()
print("The list extracted from a tree is: ", end = ' ')
print(createList(T,newList))
end = time.time()
print()
print("Time for createList is: ", end - start, "seconds")
print()
```

```
#keyAtDepth
k = 24
print()
start = time.time()
print(k, " is found in the depth: ", end=' ')
print(keyAtDepth(T,k))
end = time.time()
print()
print("Time for keyAtDepth is: ", end - start, "seconds")
```

I, Samuel Chong, sign the academic honesty certification. This is my work and only my work. No external help was used for this lab. Also, this report was made by me and no collaboration was made.