Maria F. Corona Ortega
ID: 80560734
E-Mail: mfcoronaortega@miners.utep.edu
CS 2302 | MW 1:30-2:50 am
Instructor: Dr. Olac Fuaentes
TA: Anindita Nath

CS 2302 Lab 3 Report

Introduction

In this lab we explore implementations and their variations regarding binary search trees. We have been provided with the program *bst.py* which includes basic functions for a binary search tree. We are to then expand this program and include the inquired operations described on this lab.

Proposed Solution

- **Task 1 (Displaying the tree as a figure):** My approach to this problem would be similar to that of a print in tree structure but instead of printing the item in the node, it plots a circle with such item that is in the node.
- **Task 2 (Iterative version of search operation):** The original search operation is recursive. What we use is a while loop that searches both right and left side of Btree since BTrees are in order it checks either or side depending on the size of *k* which is the item being searched. This does not search both.
- **Task 3 (Building a balanced tree):** Given a sorted list and without implementing the insert operation we need to build a balanced tree BTree. When building a balanced tree its going to really come down to the way items are inserted. In order to insert the items in a way to keep them balanced we will insert only the median of each left and right side of list. We also try and keep this implementation recursively in order to keep the running time at O(n).
- **Task 4 (Extracting elements into a sorted list):** This method similarly to "Task1" will have a similar implementation to the given a given method. This one is *InOrderT.* This method print elements in ascending order. Similar to the implementation we need, we will instead of print in order, append to a given new list. I want to implement this recursively so therefore I will also have an empty list in the parameter to append to.
- **Task 5 (Print Items ordered by depth):** My approach to this is to first print at depth 0. After this print both children then call left and right and print those children then so on.

Methods

This will not include methods given by file located on class website and will only address the modifications done to original source code.

- ***PlotTree:*** The initial implementation was iterative and that is the one that worked best however was only ably to print one side or the other of BTree. Second

implementation is recursive similar to print. Within method we initiate parameters for library *pyplot* . Through each iteration we alter the item being printed within circle and change the radius coordinates to look like a BTree.

- ***Search:*** This method takes two parameters *T* (the BTree itself) and *k* (the item we are searching for. Using a while loop, we first check if *k* is greater than or less than the current item. this will determine which direction the iteration will take. If *k* is greater than *T.item* it will be located to the right of *T.item* if it is less it will be located to the left. Therefore the iteration will proceed to the right or left child depending on the result of these if statements. It will then compare the current item the same way and proceed in needed direction. If neither one of these statements are true it could be either one, *k* has been found or two, we have reached the end of the list and *k* was not found to which it will then return *T* or *None* respectively.
- ***Balanced:*** This method receives a single parameter this being a sorted list of native python type. Since the method is recursive this first checks if L is empty, if so it does return None. First thing we do is initialize a variable *m* which contains an index, in the first call it is the location of the middle element in the list. We then create a new element if *BST* type with the contents of *m.* Recursively we call the list to the left and right of the first element (being the middle element) and we keep splitting the list in half and creating the BST object. in the end we return the newly created BTree *T* this will be balanced due
- ***IntoList:*** This method receives a BST, *T* and an empty list *L*. We first check if *T* is none. We start on the left since this list does need to be sorted in ascending order. once we reach the smallest element and append until we reach the right hand of the list. In the end we return sorted list *L.*
- ***PrintByDepth:*** This method need to print every element by depth. The implementation of this method was unsuccessful.

Experimental Results

Search

Will generate a random list into BTree from number 0-2*length of list the *Search* method will then run with a number *k* that is included within the list at insert halfway or *A[len(A)//2]*.

N items in list	K variable included	K Variable not included
10	3.53700015693903e-05	2.524800947867334e-05
15	2.8946989914402366e-05	2.4863984435796738e-05
25	3.1041010515764356e-05	0.00021691899746656418
50	7.470199489034712e-05	3.3402000553905964e-05

Balanced
Will generate a sorted list into BTree at different sizes the *Balanced* method will then run and build a balanced tree.

N items in list	Running time
10	2.6029010768979788e-05
15	2.1513988031074405e-05
25	3.0516006518155336e-05
50	0.00014661799650639296

IntoList
Will generate a random list into BTree from number 0-2*length of list the *IntoList* method will then run and generate a sorted list.

N items in list	Running Time
10	1.732501550577581e-05
15	1.0297982953488827e-05
25	1.476399484090507e-05
50	3.952599945478141e-05

Conclusion

In conclusion we learned that running times on these algorithms are important since some of these methods had a relatively decent execution time even with a higher number of items. We learned efficient implementation of BTree functions, how to debug and how to test with multiple scenarios.

Appendix

```
#Course: CS 2302 Data Structures | Spring 2019
#Author: Maria Fernanda Corona Ortega
#Assignment: Lab 3
#Instructor: Olac Fuentes
#Purpose of Code: The purpose of this code is to explore basic Btrees
and
#operations regarding basic BTree functions
#through plotting and recursion
#Last Modification: 4/3/2019 8:40pm

import matplotlib.pyplot as plt
import timeit
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')

##########################################################################
################################################

def PlotTree(T):#FIXME
    fig, ax = plt.subplots()
    ax = fig.add_subplot(111)
    x = 0
    y = 0
```

```
    if T is not None:
        InOrderD(T.right,space+'   ')
        circle = plt.Circle((x, y), radius=5)
        ax.add_patch(circle)
        items = ax.annotate(T.item, xy=(x, y), fontsize=20,
ha="center")
        x = x + 10
        y = y - 10
        InOrderD(T.left,space+'   ')

    ax.axis('off')
#    ax.set_aspect('equal')
    ax.set_aspect(1.0)

    ax.autoscale_view()

    plt.show()

def Search(T,k):#Done Iterative version of the search operation.
    # Returns the address of k in BST, or None if k is not in the tree
    while T is not None:
        if T.item < k:
            T = T.right
        if T.item > k:
            T = T.left
        else:
         return T

def Balanced(L): #Done Building a balanced binary search tree
    #height on left and right sides of tree cannot differ by more than
1
    if not L:
        return None
    m = len(L)//2
    T = BST(L[m])
    T.left = Balanced(L[:m])
    T.right = Balanced(L[m+1:])

    return T

def IntoList(T,L):#Done Extracting the elements in a binary search tree
into a sorted list.
    if T is not None:
        IntoList(T.left, L)
        L.append(T.item)
        IntoList(T.right, L)
    return L

def PrintByDepth(T): #FIXME
    if T  is not None:
        return



########################################################################
########

T = None
```

```
A= []

##############################TESTING SEARCH
METHOD#############################

#size = 100
#
#for i in range(50):
#    A.append(random.randint(0, size))
#
#for a in A:
#    T = Insert(T,a)
#
#InOrder(T)
#print()
#
#InOrderD(T,'')
#print()
#
#start = timeit.default_timer()
#
#print(Search(T,A[len(A)//2]))
#
#stop = timeit.default_timer()
#
#print('Time 1: ', stop - start)

#
#start = timeit.default_timer()
#
#print(Search(T,size+1))
#
#stop = timeit.default_timer()
#
#print('Time 2: ', stop - start)

##############################TESTING BALANCED
METHOD#############################

#for i in range(50):
#    A.append(i)
#
#start = timeit.default_timer()
#
#T = Balanced(A)
#
#stop = timeit.default_timer()
#
#print('Time: ', stop - start)
#
#InOrder(T)
#print()
#
#InOrderD(T,'')
#print()
```

```
############################TESTING INTOLIST
METHOD############################

NL=[]
size = 100

for i in range(50):
    A.append(random.randint(0, size))

for a in A:
    T = Insert(T,a)

InOrder(T)
print()

InOrderD(T,'')
print()

start = timeit.default_timer()

NL = IntoList(T, NL)

stop = timeit.default_timer()

print('Time: ', stop - start)

print(NL[:])

##############################OTHER
METHODS##################################

#PrintByDepth(R)
#PlotTree(T)
```

Academic Honesty Certification

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.

Maria F. Corona Ortega

09/09/2017