Sergio Gramer – 88521512
Lab #3: Binary Search Trees Report

In lab three we are working with binary search trees. We are given an initial code that will build a BST by insertion. We are to create a method that will create a BST in O(n) time without the use of the insert() method. We are supposed to create and iterative version of the search method. We are supposed to take a binary search tree and extract the items in a native python list in order. Finally, print the items in a BST by levels. For example: Level 0: Root, Level 1: Children of the root; so on and so forth.

My proposed solution includes first to create an iterative method for the search algorithm. I will send a BST and the item we will be searching for to the method. Then the method will compare the first item (root) in the tree with the item we are searching for. If the element that we are searching for is greater than that number, it will traverse the tree to the right. In the even that the number is smaller it will begin to traverse the tree to the left. If that number is found inside of the tree it will jump out and print a statement saying it was found. Otherwise once we have traversed the whole tree and find that it is not in there, by reaching a node that is "none" it will return and print "not found". The next thing that I will be tackling is created a BST in O(n) time. I initially began by sending the native python list into the method along with an empty tree set to "none". The first statement compared the tree to none and if it was a match it would insert the first element into that tree. I would also use a counter "i" that would keep track of where I was in the native python list. It was very complicated getting this to work. I could only get one half of the tree to be build although it was traversing the entire list. It was being called recursively though I gave that up. I ended up only sending the native python list into the method. Then begin building the list by splitting that list in half. Since I was taking in a sorted list, I would grab the middle element and put that as the root. Everything before would be to the left of said root and everything after would be the T.left. This was also built using recursion always making the problem smaller by splitting it in the middle. Everything to the before would be the left child and everything to the right would be the right children of said node. I would then work on printing the elements in a BST by level. At first I tried to do everything inside the method itself. Figuring out the height and printing everything inside of that method was too complicated. I would get errors with finding out the height. I then decided to make separate methods for doing things. I began by finding out the height depending on a tree that I sent. I did this by traversing to the left and to the right until I hit a node that was empty. I would keep a counter that kept track of each side separately. These two final values would then be compared and the largest would be sent back. In case they were equal we would just send the left arbitrarily. Once we had the height we could begin on traversing levels. For each level we would send that to a separate method that would print all of the elements inside of it. Go back and do the same for the rest of the levels.

We first do the Search method. Sample of the code:
NOTE: COMMENTS OMMITED FOR READABILITY

```
elif T.item > k :
        T = T.left
    else :
        T = T.right
  if T == None :
    print(k, 'Not found')
    return T
```

Results: We search for 3 items that are there.

```
In [12]: runfile('/Users/SergioGramer/Desktop/
Spring2019/CS2302/Lab 3 BST/lab3.py', wdir='/Users/
SergioGramer/Desktop/Spring2019/CS2302/Lab 3 BST')
1 2 3 4 5 7 8 9 10 12 15 18
       18
          15
    12
       10
          9
 8
       7
          5
    4
          3
       2
          1

10 found
4 found
8 found
```

We search for 1 item that is there and 1 that isn't:

```
In [11]: runfile('/Users/SergioGramer/Desktop/
Spring2019/CS2302/Lab 3 BST/lab3.py', wdir='/Users/
SergioGramer/Desktop/Spring2019/CS2302/Lab 3 BST')
1 2 3 4 5 7 8 9 10 12 15 18
       18
          15
    12
       10
          9
 8
       7
          5
    4
          3
       2
          1

10 found
50 Not found
```

We search for 3 items that are not there:

```
In [15]: runfile('/Users/SergioGramer/Desktop/
Spring2019/CS2302/Lab 3 BST/lab3.py', wdir='/Users/
SergioGramer/Desktop/Spring2019/CS2302/Lab 3 BST')
1 2 3 4 5 7 8 9 10 12 15 18
       18
          15
    12
       10
          9
 8
       7
          5
    4
          3
       2
          1

100 Not found
11 Not found
0 Not found
```

We build a tree in O(n) time with a sorted list: Sample code
NOTE: COMMENTS ARE OMMITTED FOR READABILITY

```
if len(A) == 1 :
    return BST(A[0])
else :
    mid = len(A)//2
    T = BST(A[mid])
    T.left = BuildList(A[:mid])
    T.right = BuildList(A[mid+ 1:])
    return T
```

We send an list in order. Print the list we sent. Print using the InOrder method from the BST and then print the BST.

```
V = None
T = None
A = [10, 4, 15, 2, 8, 1, 3, 5, 9, 7, 12, 18] #built with insert
S = [1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 15, 18] #built with Sorted python list
for a in A:
    V = Insert(V,a)
T = BuildList(S)
print('List sent: ', S)
InOrder(T)
print()
InOrderD(T,'')
print()
#Search(T, 100)
#Search(T, 11)
#Search(T, 0)
```

```
wdir='/Users/SergioGramer/Desktop/Spring2019/CS2302/La
List sent:  [1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 15, 18]
1 2 3 4 5 7 8 9 10 12 15 18
        18
           15
    12
        10
           9
 8
        7
           5
    4
           3
        2
           1
```

Finding the height of a tree: Level 0: Root
NOTE: COMMENTS ARE OMMITTED FOR READABILITY

```
if T is None :
    return -1
else :
    l = 1 + height(T.left)
    r = 1 + height(T.right)
    if l >= r :
        return l
    else :
        return r
```

```
In [19]: runfile('/Users/SergioGramer/Desktop
wdir='/Users/SergioGramer/Desktop/Spring2019/
1 2 3 4 5 7 8 9 10 12 15 18
        18
           15
    12
        10
           9
 8
        7
           5
    4
           3
        2
           1

Height of Tree:  3
```

Print nodes depending on depth:
NOTE: COMMENTS ARE OMMITTED FOR READABILITY

```
def Depth(T, h) :
    for j in range(h + 1):
        print('Keys at depth', j, ':')
        PrintLevels(T, j)
        print()

def PrintLevels(T, j) :
    if T is None :
        return
    if j == 0 :
        print (T.item)
    else :
        PrintLevels(T.left, j - 1)
        PrintLevels(T.right, j - 1)
```

Print all levels:

```
Height of Tree:  3
Keys at depth 0 :
8

Keys at depth 1 :
4
12

Keys at depth 2 :
2
7
10
18

Keys at depth 3 :
1
3
5
9
15
```

Print all but last:

```
Height of Tree:  3
Keys at depth 0 :
8

Keys at depth 1 :
4
12

Keys at depth 2 :
2
7
10
18
```

APPENDIX

Lab3.py (Lab specifies what was coded by Dr. Fuentes and what was coded by myself(Sergio Gramer))

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 28 16:38:43 2019

@author: SergioGramer
"""

# Code to implement a binary search tree
# Programmed by Olac Fuentes
# Last modified February 27, 2019
import numpy as np
import matplotlib.pyplot as plt
import math

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')

###### Sergio Gramer Code Begins #######


def circle(cx, cy, rad) :
    n = int(4*rad*math.pi)
    t = np.linspace(0,6.3,n)
    x = (cx+rad*np.sin(t))
    y = (cy+rad*np.cos(t))
    return x,y

def draw_circle(ax, cx, cy, radius) :
        x,y = circle(cx, cy, radius)
        ax.plot(x,y,color='k')

def circle_level(ax, n, cx, cy, radius) :
    if n > 0 :
        draw_circle(ax, cx, cy, radius)
        circle_level(ax, n-1, cx - orig_size//2, cy - 1000, radius)

    elif n == 0 :
        return

def draw_triangle(ax,p) :
        ax.plot(p[:,0],p[:,1],color='k') ## key operation to plot

def triangle_level(ax,n,p,w) :
    if n > 0 :
        draw_triangle(ax, p)
        q = (p * [w, 1])
        triangle_level(ax, n-1, q-[(orig_size*w)*w,orig_size],w) #will draw triangles on left hand side
        circle_level(ax, n-1, [(orig_size*w)*w], [orig_size], 100)
        triangle_level(ax, n-1, q+[orig_size - (orig_size * (w/2)), -orig_size], w) #will draw triangles on right hand
        circle_level(ax, n-1, [orig_size - (orig_size * (w/2))], [-orig_size], 100)
```

```
plt.close("all")
orig_size = 1000
p = np.array([[orig_size,0], [500,orig_size], [0,0]])
fig, ax = plt.subplots()
ax.set_aspect(1.0)
#ax.axis('off')
plt.show()
fig.savefig('triangle.png')

def Search(T, k) : #Searches a BST 'T' for item "k"
    while T is not None : #Ensures we don't traverse and empty BST
        if T.item == k :
            print(T.item, 'found') #Will print found once the item is found
            return T.item
        elif T.item > k : #Compares the number are looking for to the item we are
            T = T.left    #currently at to see if it is greater, goes to left
        else : #else goes to the right side
            T = T.right
    if T == None : #If we get to the end of the BST and haven't found it
        print(k, 'Not found') #print Not Found
        return T

def BuildList(A) : #Builds a list in O(n) time from a sorted native py list "A"
    if A is None or len(A) is 0 : #will ensure the list is not none or empty
        return None
    if len(A) == 1 : #If theres only one item in the list will return a tree of size 1
        return BST(A[0])
    else :
        mid = len(A)//2 #splits the list into 2 based on the middle item in the list
        T = BST(A[mid]) #Creates the first node in the BST with the middle value
        T.left = BuildList(A[:mid]) #Recursively makes the list shorter for the left and builds the left
        T.right = BuildList(A[mid+ 1:]) #recursively makes the list shorter for the right and builds the right
        return T

def height(T): #finds the height of a BST
    if T is None :
        return -1
    else :
        l = 1 + height(T.left) #calculates height of left side + 1 because we already passed root
        r = 1 + height(T.right) #calcules height of right side
    if l >= r : #if left side is greater or equal send left, no need to go further if equal
        return l
    else : #else right is greater send right
        return r

def Depth(T, h) : #method that prints keys at certain depth
    for j in range(h + 1): #send in calculated height
        print('Keys at depth', j, ':') #j will let us know what level we are on
        PrintLevels(T, j) #calls print levels
        print()

def PrintLevels(T, j) : #print levels will ptin out the items in that particular depth of a tree
    if T is None :
        return
    if j == 0 : # if j is 0 we are at root
        print (T.item)
    else :
        PrintLevels(T.left, j - 1) #recursively calls itself to print left child of node
        PrintLevels(T.right, j - 1) #recursively calls itself to print right child of node


# Code to test the functions above
i = 0
v = 0
V = None
T = None
A = [10, 4, 15, 2, 8, 1, 3, 5, 9, 7, 12, 18] #built with insert
S = [1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 15, 18] #built with Sorted python list
for a in A:
    V = Insert(V,a)
```

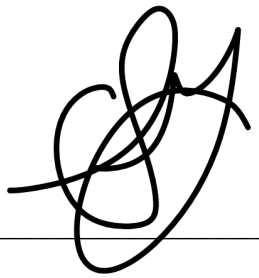
```
T = BuildList(S)
InOrder(T)
print()
InOrderD(T,'')
print()
#Search(T, 100)
#Search(T, 11)
#Search(T, 0)
h = height(T)
print('Height of Tree: ', h)
Depth(T, h-1)
```

ACADEMIC CERTIFICATION

I “Sergio Gramer” 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.

Signed: 03/13/2019

Sergio Gramer



_____