Diego Quinones

CS 2302 - Data Structures

Professor Olac Fuente

Lab 3

March 10, 2019

```
# Code to implement a binary search tree
# Programmed by Diego Quinones
# Last modified March 10, 2019

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):
    # Returns largest item in BST. Error if T is None
    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 iterativeSearch(T, k):
    # Set current to root of binary tree
    if T is None:
        #return -1 if empty
        return -1
    while T is not None:
        #moves to each side depending if less or more
        if T.item>k:
            T=T.left
        elif T.item==k:
            return 1
        elif T.item<k:
            T=T.right
    #return -1 is not found
    return -1

def FindAndPrint(T,k):
    f = Find(T,k)
    if f is not None:
        print(f.item,'found')
    else:
        print(k,'not found')

def Extract(T,L):
    if T is not None:
```

```
        #moves to left child
        InOrder(T.left)
        #adds item to a BST
        L.append(T.item)
        #moves to the right child
        InOrder(T.right)
    return L

def PrintAtDepth(T):
  NewList=[T]
  count=0
  while NewList:
      #list that will store remaining values
      NewList2=[]
      #prints text and number of depth
      print('Keys at: ',count)
      for i in NewList:
          print(i.item)
          if i.left:
              #values being used get stored
              NewList2.append(i.left)
          if i.right:
              NewList2.append(i.right)
      count=count+1
      #NewList is reset with the remaining values
      NewList=NewList2

def BalancedTree(L):
    if not L:
        #if list is empty return none
        return None
    mid = round((len(L)) / 2)
    #divide by the middle
    T1 = BST(L[mid])
    #moves values to the left and right accordingly
    T1.left = BalancedTree(L[:mid])
```

```
    T1.right = BalancedTree(L[mid+1:])
    return T1

# 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)

InOrder(T)
print()
InOrderD(T,'')
print()

print('Iterative Search',end=' ')
findvalue=11
find=iterativeSearch(T,findvalue)
print()
if find==1:
    print(findvalue,'Value was found')
else:
    print(findvalue,'Value was not found')



print()
print('Extract',end=' ')
print()
L=[]
L=Extract(T,L)
for i in L:
    print(i, end=' ')
print()

print()
PrintAtDepth(T)
print()
```

```
print('Balanced Tree')
print()
L1=[3,5,7,20]
T1=BalancedTree(L1)
InOrder(T1)
print()
InOrderD(T1,' ')
```

Lab 3 consisted on using binary search trees in different types of problems. The first one consisted on displaying a set of values inside of a binary tree, these had to be represented in a visual way, based on how binary trees look. This code was the one I had the most trouble with because I didn't knew how to make the recursive tree plotting.

The second problem consisted on changing the search method to an iterative version, instead of its recursive form. This personally I thought was the easiest one, because searching uses a comparison that goes to one side based on int comparisons, it was extremely easy to solve.

Third problem asked to build a balanced binary search tree based on a sorted list, what I did to solve it was to assign two sides on the tree based on two sides of the sorted list, and by the end combined them to create the full tree.

The fourth problem had you appending the values in a search binary tree in a sorted list. My way of solving it was to use the printSort method as a base, but instead of printing in between recursive calls, I would append the value.

And lastly I had to display which values where at each depth on the tree. I created a counter which would be what displays on which depth we are, then I would store the values, and those would be printed. But when the loop is restarted it would use only the numbers that are left.

```
In [1]: runfile('/Users/diegoquinones/Desktop/bst.py',
wdir='/Users/diegoquinones/Desktop')
10 30 40 42 45 50 60 70 90 100 130 140 150 180
             180
          150
             140
       130
          100
    90
 70
       60
    50
          45
             42
       40
             30
          10

Iterative Search
11 Value was not found

Extract
10 30 40 42 45 50 60 90 100 130 140 150 180 70

Keys at:  0
70
Keys at:  1
50
90
Keys at:  2
40
60
130
Keys at:  3
10
45
100
150
Keys at:  4
30
42
```

```
42
140
180

Balanced Tree

3 5 7 20
     20
  7
     5
        3

In [2]:
```