**Lab #3**

CS 2302

Ana Luisa Mata Sánchez

# Introduction

This lab consists of drawing a binary search tree using pyplot, creating an iterative version of the search function, building a balance binary search tree in O(n) time, extracting the elements of a binary search tree into a sorted list in O(n) time, and printing all elements in a binary search tree by their depth.

# Proposed solution design and implementation

**Module 1 – Drawing a binary search tree**

Using the circle method provided by Dr. Fuentes on Lab 1, I added the draw\_circle method and the draw\_bintrees method.

The draw\_circle method draws the circle using the calculations from the circle method. I made sure to add the zorder property with a higher number in order to display the circles on top of the binary tree. This method also fills the circle white to make it opaque.

The draw\_bintrees method draws the lines for the binary search tree. It first checks if the current node is not null because it will not draw a line if it is. Then it adds the text for that item using the center of the arch. After that, it checks the item to which kind of arch or line it will draw. If the item has both a right and left child, it draws an arch using the normal p array. If the item has only a right child or a left child, it draws a line to the right or left, using the right or left side of the p array to create a new array. After drawing, it calculates the change in x that is half of the distance between the middle of the arch and the right side of the arch. I then created two different arrays (one for the left child and the other for the right child) that add or subtract the distance to the new left and right points of the arch but keep the same point that was used as the right or left point in p as the new middle. After that it checks if there is a left child and if there is, it calls the method using the array for the left and it calls the method to draws a circle. After that it does the same for the right.

**Module 2 – Iterative version of the search operation**

The iterative method (iterSearch) first declares a Boolean variable to double check that the item was not found. After that it starts a while loop that keep going while the current node is not null and its item is also not null. Inside the while loop there are three if statements that check whether the current is the key, is less than the key, or is larger than the key. If the item is equal to the key it marks the Boolean variable as true, returns the node that contains the item, and breaks the loop. If the key is smaller or larger than the item, it iterates to the left or right of the tree, where the smaller and larger elements correspondingly are located. After the loop is done, it checks the Boolean variable to see if the item was found, if it was not which will be the case if it has reached the end, it will return null/none.

**Module 3 – Building a balanced tree from a sorted list**

The balanced tree method first checks if the length of the list is 0 and if it is the case it returns. A balanced tree from a sorted list will always use the middle element of a list as the root for the node. Knowing this, the method creates a new tree “T” and makes the root the middle element of the list. After inserting the root it divides the list in half without including the already inserted item. The next step, without using the insert method, calls the method with the left child and the right child each with a corresponding new list.

**Module 4 – Extracting the elements of a binary search tree into a sorted list**

This method starts by checking if the node is not null, if it is it just returns an empty list. If the tree is not null it traverses until it reaches the leftmost part of the tree, where the smallest method is found. After that it appends the parent of that node and then it appends the right element. After it finishes, it returns the new list.

**Module 5 – Printing the elements ordered by depth**

I split this task in two methods, one that finds the depth of the tree and another that is called with a for loop that goes from 0 to the depth and prints items by depth. The Depth method first checks if the tree is null, if it is the depth is 0. After that it keeps going either left or right (by checking that either the left or the right are not null) and adds one each time the method iterates. If both the right and left are none (for safety) it just returns 0.

# Experimental results

Graphic representation of T used in some of the following methods

180

150

140

130

100

50

45

42

30

10

|  |  |  |
| --- | --- | --- |
| **Method call** | **Output** | **Analytical running time** |
| print('############### Tree Figure ###############')  plt.close("all")  orig\_size = 1000  p = np.array([[-orig\_size,-orig\_size],[0,0],[orig\_size,-orig\_size]])  fig, ax = plt.subplots()  fig.set\_size\_inches(18, 18.6)  #draws the tree  draw\_bintrees(ax,p,orig\_size,T,orig\_size//6)  #draws the circle for the root  draw\_circle(ax, [0,0], orig\_size//6)  ax.set\_aspect(1.0)  ax.axis('off')  plt.show()  fig.savefig('BinTree.png') |  | O(n) |
| print('############### Search for an item ###############')  searchee = iterSearch(T,**10**)  if searchee != None:  print(searchee.item, "was found!")  else:  print("Item not found") | ############### Search for an item ###############  10 was found! | O(n) |
| print('############### Search for an item ###############')  searchee = iterSearch(T,**13**)  if searchee != None:  print(searchee.item, "was found!")  else:  print("Item not found") | ############### Search for an item ###############  Item not found | O(n) |
| print('###############Build Balance Tree###############')  **L = [1,2,3,4,5,6,7]**  Balanced = BuildBalanced(**L**)  InOrderD(Balanced," ") | 7  6  5  4  3  2  1 | O(n) |
| print('###############Build Balance Tree###############')  **L = [2,3,5,7,10,20,150,200,301]**  Balanced = BuildBalanced(**L**)  InOrderD(Balanced," ") | 301  200  150  20  10  7  5  3  2 | O(n) |
| print('###############Tree to List###############')  L = []  TreeInList = []  TreeInList = TreeToList(**T**,L)  print(TreeInList) | ###############Tree to List###############  [10, 30, 42, 45, 50, 100, 130, 140, 150, 180] | O(n) |
| print('###############Tree to List###############')  L = []  TreeInList = []  TreeInList = TreeToList(**Balanced**,L)  print(TreeInList) | ###############Tree to List###############  [2, 3, 5, 7, 10, 20, 150, 200, 301] | O(n) |
| print('###############Depth of Tree###############')  #get the depth of the tree  d = Depth(**T**)  #print elements by depth using a loop  for i in range(d+1):  print("Keys at depth ", i ,": ", sep = '', end="")  PrintByDepth(**T**,i)  print() | ###############Depth of Tree###############  Keys at depth 0: 100  Keys at depth 1: 50 130  Keys at depth 2: 42 150  Keys at depth 3: 10 45 140 180  Keys at depth 4: 30 | O(n) |
| print('###############Depth of Tree###############')  #get the depth of the tree  d = Depth(**Balanced**)  #print elements by depth using a loop  for i in range(d+1):  print("Keys at depth ", i ,": ", sep = '', end="")  PrintByDepth(**Balanced**,i)  print() | ###############Depth of Tree###############  Keys at depth 0: 10  Keys at depth 1: 5 200  Keys at depth 2: 3 7 150 301  Keys at depth 3: 2 20 | O(n) |

# Conclusion

This project helped me refresh the concept ot a binary search tree and how each method can be implemented in different ways.

# Appendix

|  |
| --- |
| # Author: Ana Luisa Mata Sanchez |
|  | # Course: CS2302 |
|  | # Assignment: Lab #3 |
|  | # Instructor: Olac Fuentes |
|  | # Description: Binary search tree operations |
|  | # T.A.: Anindita Nath |
|  | # Last modified: 03/08/2019 |
|  | # Purpose: Display a BST as a figure, search a BST, extract a sorted list into a BST, |
|  | # extract a BST into a sorted list and print all elements by depth. |
|  |  |
|  | import numpy as np |
|  | import matplotlib.pyplot as plt |
|  | import matplotlib.patches as patch |
|  | import math |
|  |  |
|  | # Code to implement a binary search tree |
|  | # Programmed by Olac Fuentes |
|  | # Last modified February 27, 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): |
|  | 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') |
|  |  |
|  | ################################# START OF NEW CODE ################################# |
|  |  |
|  | #Creates the circle |
|  | def circle(center,rad): |
|  | n = int(4\*rad\*math.pi) |
|  | t = np.linspace(0,6.3,n) |
|  | x = center[0]+rad\*np.sin(t) |
|  | y = center[1]+rad\*np.cos(t) |
|  | return x,y |
|  |  |
|  | #Draws the circle |
|  | def draw\_circle(ax,center,radius): |
|  | x,y = circle(center,radius) |
|  | #Add zorder to make sure that the circle is drawn after the tree |
|  | ax.plot(x,y, linewidth=2,color='k', zorder=1) |
|  | #Fill in the circle with white |
|  | ax.fill(x, y, color='w') |
|  |  |
|  | #Draws binary trees |
|  | def draw\_bintrees(ax,p,deltay,T,circlesize): |
|  | if T!=None: |
|  | #Adds the text for the current item, it places it centered on the center of the arch |
|  | plt.text(p[1][0],p[1][1], str(T.item), horizontalalignment='center', verticalalignment='center',fontsize=25) |
|  |  |
|  | #If there is both a right and left child, draw a full arch |
|  | if T.right != None and T.left != None: |
|  | plt.plot(p[:,0],p[:,1],linewidth=1,color='k', zorder=0) |
|  | #if there is only a right child, draw a right line |
|  | elif T.left == None and T.right!= None: |
|  | b = np.array([[p[2][0],p[2][1]],[p[1][0],p[1][1]],[p[2][0],p[2][1]]]) |
|  | plt.plot(b[:,0],b[:,1],linewidth=1,color='k', zorder=0) |
|  | #if there is only a left child, draw a left line |
|  | elif T.right == None and T.left!= None: |
|  | a = np.array([[p[0][0],p[0][1]],[p[1][0],p[1][1]],[p[0][0],p[0][1]]]) |
|  | plt.plot(a[:,0],a[:,1],linewidth=1,color='k', zorder=0) |
|  |  |
|  | #use distance to make the next arches |
|  | distance = math.hypot((p[0][0] - p[2][0]),(p[0][1] - p[2][1]))//4 |
|  |  |
|  | #new arches |
|  | r = np.array([[p[0][0]-distance,p[0][1]-(deltay)],[p[0][0],p[0][1]],[p[0][0]+distance,p[0][1]-(deltay)]]) |
|  | q = np.array([[p[2][0]-distance,p[2][1]-(deltay)],[p[2][0],p[2][1]],[p[2][0]+distance,p[2][1]-(deltay)]]) |
|  |  |
|  | #if there is a left child, draw the circle for that left child and call the method again |
|  | if T.left != None: |
|  | draw\_circle(ax, r[1], circlesize) |
|  | draw\_bintrees(ax,r,deltay,T.left,circlesize) |
|  |  |
|  | #if there is a right child, draw the circle for that right child and call the method again |
|  | if T.right != None: |
|  | draw\_circle(ax, q[1], circlesize) |
|  | draw\_bintrees(ax,q,deltay,T.right,circlesize) |
|  |  |
|  | #Iterative search |
|  | def iterSearch(L,key): |
|  | #boolean to keep track if we found the item |
|  | isFound = False |
|  |  |
|  | #iterate through the tree until you reach the end |
|  | while L!=None and L.item!=None: |
|  | #if you find the item stop and mark it as found |
|  | if L.item == key: |
|  | isFound = True |
|  | #return the item that you found |
|  | return L |
|  | break |
|  | #if you have not found it and it is larger than the one you are on, go left |
|  | elif key<L.item: |
|  | L = L.left |
|  | #if you have not found it and it is smaller than the one you are on, go right |
|  | elif key>L.item: |
|  | L = L.right |
|  |  |
|  | #if you reached the end and did not find the item, return None |
|  | if isFound == False: |
|  | return None |
|  |  |
|  | #Building a balanced binary search tree given a sorted list as input. |
|  | def BuildAVL(L): |
|  | if len(L) == 0: |
|  | return |
|  |  |
|  | #the root will always be the middle element |
|  | T = BST(L[len(L)//2]) |
|  |  |
|  | #split the list in two |
|  | leftlist = L[:len(L)//2] |
|  | rightlist = L[len(L)//2+1:] |
|  |  |
|  | #Call the method again while maintining the link to the root |
|  | T.right = BuildAVL(rightlist) |
|  | T.left = BuildAVL(leftlist) |
|  |  |
|  | #return the finished tree |
|  | return T |
|  |  |
|  | def TreeToList(T,L): |
|  | if T != None: |
|  |  |
|  | #add the left |
|  | if T.left != None: |
|  | TreeToList(T.left,L) |
|  | #then add the root, that is larger than the left but smaller than the right |
|  | L.append(T.item) |
|  | #add the right |
|  | if T.right != None: |
|  | TreeToList(T.right,L) |
|  |  |
|  | #return finished list |
|  | return L |
|  | else: |
|  | #if the tree is null return an empty list |
|  | return [] |
|  |  |
|  | #Calculates the depth of the tree |
|  | def Depth(T): |
|  | #a tree that is none has depth 0 |
|  | if T==None: |
|  | return 0 |
|  |  |
|  | #keeps going either left or right and returns the longest path |
|  | if T.left != None: |
|  | return 1 + Depth(T.left) |
|  | elif T.right != None: |
|  | return 1 + Depth(T.right) |
|  | else: |
|  | return 0 |
|  |  |
|  | #prints elements by depth |
|  | def PrintByDepth(T,d): |
|  | if T != None: |
|  | #when it has traveled all the way until the desired depth, print |
|  | if d == 0: |
|  | print(T.item, " ", end="") |
|  |  |
|  | #keep iterating thorugh depths |
|  | PrintByDepth(T.left,d-1) |
|  | PrintByDepth(T.right,d-1) |
|  |  |
|  | # 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(SmallestL(T).item) |
|  | print(Smallest(T).item) |
|  |  |
|  | FindAndPrint(T,40) |
|  | FindAndPrint(T,110) |
|  |  |
|  | n=60 |
|  | print('Delete',n,'Case 1, deleted node is a leaf') |
|  | T = Delete(T,n) #Case 1, deleted node is a leaf |
|  | InOrderD(T,'') |
|  | print('####################################') |
|  |  |
|  | n=90 |
|  | print('Delete',n,'Case 2, deleted node has one child') |
|  | T = Delete(T,n) #Case 2, deleted node has one child |
|  | InOrderD(T,'') |
|  | print('####################################') |
|  |  |
|  | n=70 |
|  | print('Delete',n,'Case 3, deleted node has two children') |
|  | T = Delete(T,n) #Case 3, deleted node has two children |
|  | InOrderD(T,'') |
|  |  |
|  | n=40 |
|  | print('Delete',n,'Case 3, deleted node has two children') |
|  | T = Delete(T,n) #Case 3, deleted node has two children |
|  | InOrderD(T,'') |
|  |  |
|  | ################################# START OF NEW METHOD CALLS ################################# |
|  | print('############### Tree Figure ###############') |
|  | plt.close("all") |
|  | orig\_size = 1000 |
|  | p = np.array([[-orig\_size,-orig\_size],[0,0],[orig\_size,-orig\_size]]) |
|  | fig, ax = plt.subplots() |
|  | fig.set\_size\_inches(18, 18.6) |
|  | #draws the tree |
|  | draw\_bintrees(ax,p,orig\_size,T,orig\_size//6) |
|  | #draws the circle for the root |
|  | draw\_circle(ax, [0,0], orig\_size//6) |
|  | ax.set\_aspect(1.0) |
|  | ax.axis('off') |
|  | plt.show() |
|  | fig.savefig('BinTree.png') |
|  | print() |
|  |  |
|  | print('############### Search for an item ###############') |
|  | searchee = iterSearch(T,10) |
|  | if searchee != None: |
|  | print(searchee.item, "was found!") |
|  | else: |
|  | print("Item not found") |
|  |  |
|  | print() |
|  | print('###############Build Balance Tree###############') |
|  | L = [1,2,3,4,5,6,7] |
|  | AVLTree = BuildAVL(L) |
|  | InOrderD(AVLTree," ") |
|  |  |
|  | print() |
|  | print('###############Tree to List###############') |
|  | L = [] |
|  | TreeInList = [] |
|  | TreeInList = TreeToList(T,L) |
|  | print(TreeInList) |
|  |  |
|  | print() |
|  | print('###############Depth of Tree###############') |
|  | #get the depth of the tree |
|  | d = Depth(T) |
|  | #print elements by depth using a loop |
|  | for i in range(d+1): |
|  | print("Keys at depth ", i ,": ", sep = '', end="") |
|  | PrintByDepth(T,i) |
|  | print() |