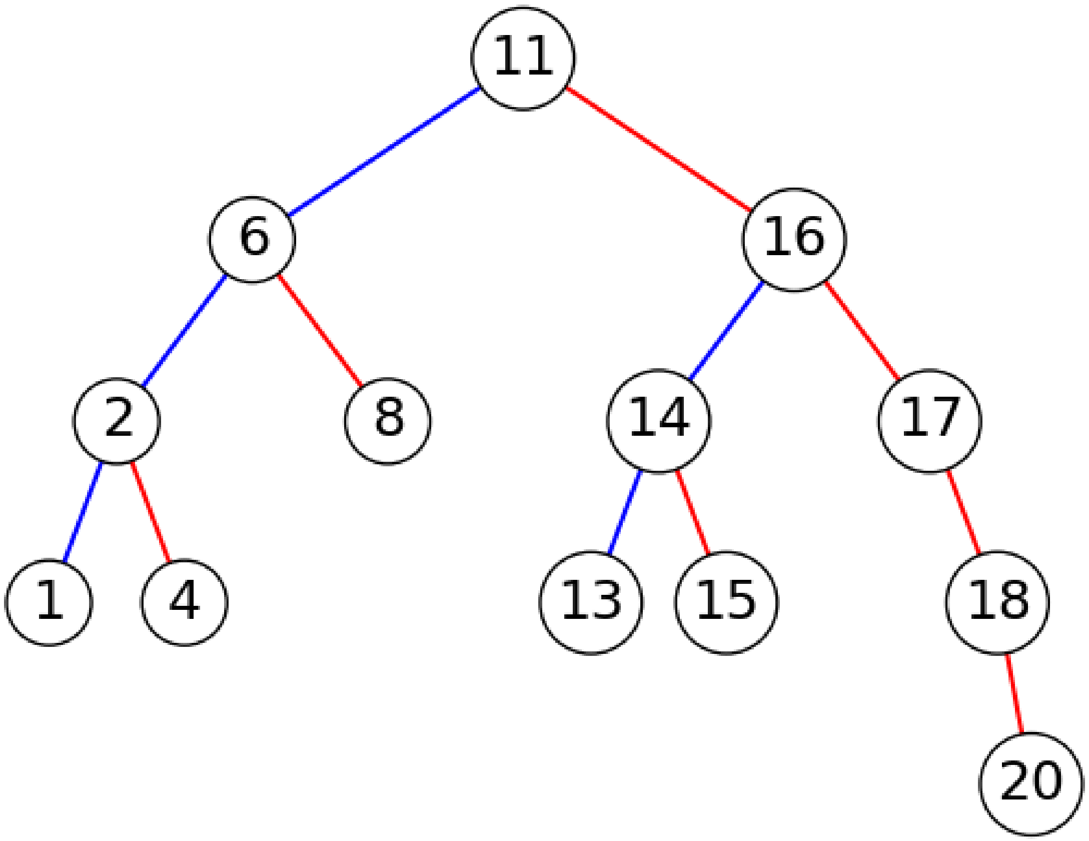
Lab IV

Binary Search Tree List



Course: CS 2302

Section: 12:00 p.m. – 1:20 p.m.

Author: Oswaldo Escobedo

Instructor: Dr. Fuentes

TA: Harshavardhini Bagavathyraj

Introduction

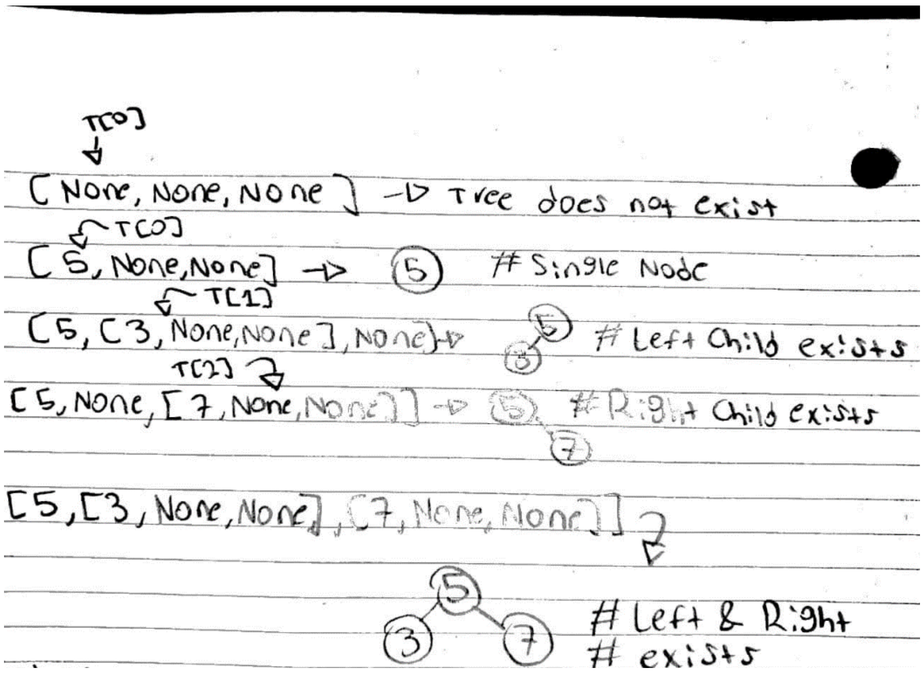
The objective of this laboratory is to add 9 methods to the bst_list class, which Dr. Fuentes provided us on blackboard. To create these methods we must trace and use the matplotlib python library. The 11 methods that we will add are the following:

1. *size(T)* – Calculates and returns how many data items are in the tree.
2. *minimum(T)* – Returns the smallest item in the tree.
3. *maximum(T)* – Returns the largest item in the tree.
4. *height(T)* – Returns the height of the tree (height of the deepest leaf).
5. *inTree(T,i)* – Returns a Boolean value (True/False) that tells us if *i* is in the tree or not.
6. *printByLevel(T)* – Prints the data items in level order traversal.
7. *tree2List(T)* – Returns a sorted list which contains all the items in the tree.
8. *leaves(T)* – Returns a list which contains all the leaves that are in the tree.
9. *itemsAtDepthD(T,d)* – Returns a list that contains all the items that are at depth *d* in the tree.
10. *depthOfK(T,k)* – Returns and calculates the depth of *k* in the tree, if not in the tree return -1.
11. *draw(T)* – Draws the tree.

Proposed Solution Design and Implementation

Observation:

Before starting to do the methods we must understand the implementation of a binary search tree in a list.


T[0]
[None, None, None] -▷ Tree does not exist
T[0]
[5, None, None] -▷ # Single Node
T[1]
[5, [3, None, None], None] -▷ # Left Child exists
T[2]
[5, None, [7, None, None]] -▷ # Right Child exists
[5, [3, None, None], [7, None, None]]
# Left & Right
# exists

Therefore,

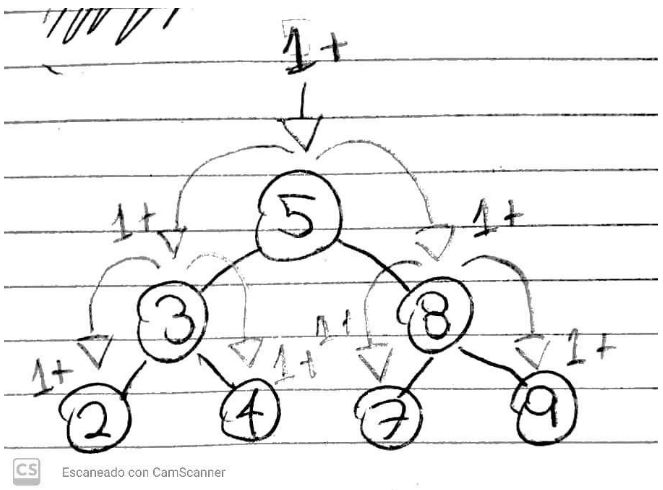
T[0] = represents the root.

T[1] = represents the left child.

T[2] = represents the right child.

The image explains the concept of the binary search tree in a list.

size(T)

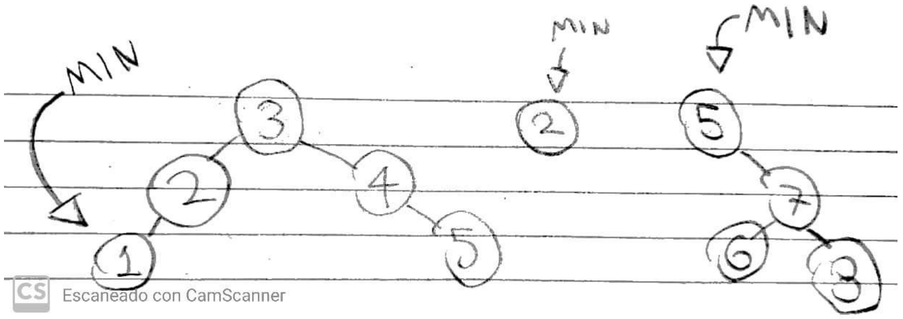


First, I started with the size method, which takes the bst list T as a parameter. Its purpose is to compute and return the total number of data items (nodes) in the tree.

To solve this, we must keep in mind that we will start counting the root, then we must traverse to the left and right child and make them our new roots.

Keeping that in mind, my approach was recursive, since it is much easier to implement the algorithm than in an iterative way, since in each step we will have twice the pointers, except for the root, which only has one pointer. Therefore, we will make our base case be when T is None. Then, we will add 1 + a recursive call that traverse the left subtree T[1] and + a recursive call that traverses the right subtree T[2].

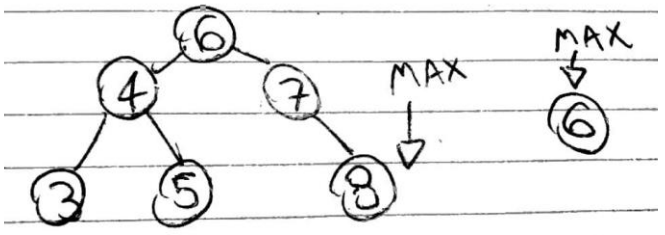
minimum(T)



The purpose of this method is to find and return the smallest value in the tree. To solve this, we must remember the definition of a binary search tree, which is that in the left sub-tree there are values less than the root and in the right sub-tree the values are greater than or equal to the root. Taking this into account, if we only traverse the left side always (without going to the right side) we will find the smallest value, since it is in the leftmost part of the tree.

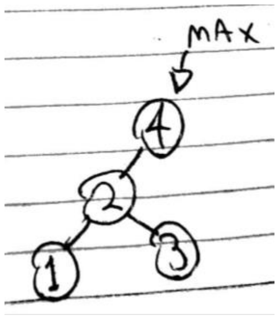
With that in mind, my approach was recursive (although the iterative approach is the same), I simply chose the recursive one because it looks cleaner, more readable, and shorter. Therefore, we will make our base case be when T [1] is None, meaning that there are no left children to visit. Afterwards, we will make a recursive call in which only focus on traversing to the left side always.

maximum(T)

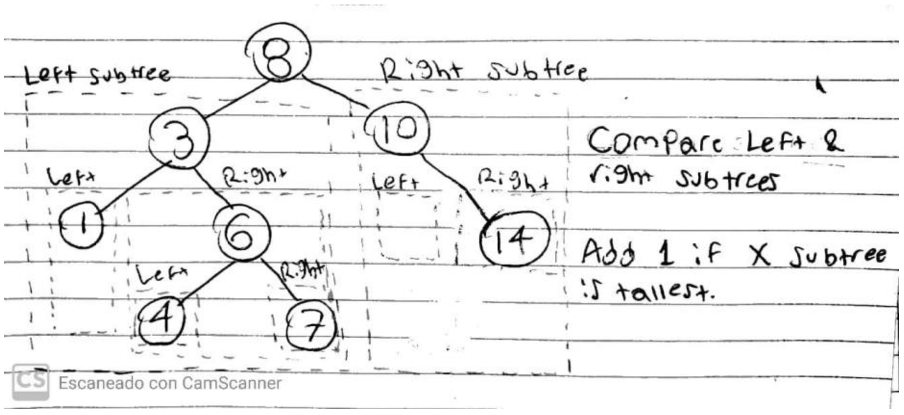


The purpose of this method is to find and return the largest value in the tree. To solve this, we must remember the definition of a binary search tree, which is that in the left sub-tree there are values less than the root and in the right sub-tree the values are greater than or equal to the root. Taking this into account, if we only traverse the right side always (without going to the left side) we will find the largest value, since it is in the rightmost part of the tree.

With that in mind, my approach was recursive (although the iterative approach is the same), I simply chose the recursive one because it looks cleaner, more readable, and shorter. Therefore, we will make our base case be when T[2] is None, meaning that there are no right children to visit. Afterwards, we will make a recursive call in which only focus on traversing to the right side always.



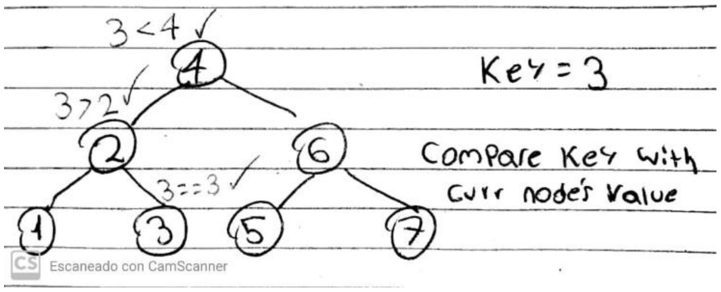
height(T)



The purpose of this method is to calculate and return the height of the tree. To solve this, we have to compare which subtree (left and right) has the highest height. If subtree X is taller than subtree Y, then add 1 and traverse the side where subtree X is located. To do this, we will do a recursive approximation not because it is cleaner but because we must make the problem become into smaller and smaller subproblems.

Taking that into account, our base case will be when T is nothing and we will return a value of -1 (because a tree that does not exist has no height). Then we will add 1 + max (recursive call to traverse to the left T[1], recursive call to traverse to the right T[2]). It should be noted that we will use the max method to find out which sub-tree has a greater height and traverse the one with a greater height.

inTree(T,i) *# Note: in the code I misspelled this method's name*



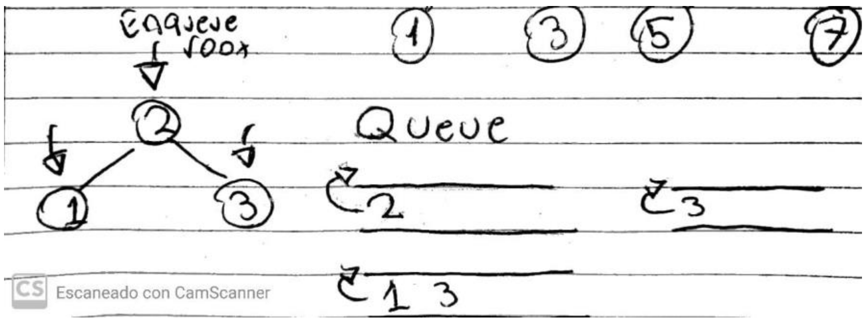
The purpose of this method is to find out whether *i* is in the tree or not. Therefore, if *i* is in the tree we will return True; otherwise, we will return False. To solve this, we have to compare the *i* with the current node we are in. In other words, if the *i* is smaller than the curr node, we will move to the left, otherwise we will move to the right side of the tree. By doing this we will locate in a very easy and fast way if *i* is in the tree or not. In short, we will use the main property of BST's.

With this in mind, our base case will be when T equals None, because if there is no tree then *i* is definitely not in the tree.

This time there was no problem if I went for a recursive or iterative approach, but I went for a recursive approach since I like how readable and short it looks. Therefore, the conditions to find *i* were the following:

- The first condition is when the root T[0] is *i* this will immediately return True.
- The second condition is when *i* is less than the root, this will call the method recursively to traverse the left subtree.
- The last condition will be when *i* is greater than or equal than the root, this will call the method recursively to traverse the right subtree.

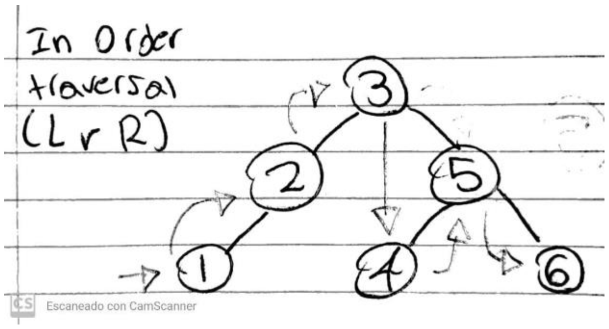
printByLevel(T)



The purpose of this method is to print the elements of the tree in level order. To solve this, we are going to use a queue, remember that first to enter is first to exit. That property will help us store the nodes in the left subtree first and then the right subtree.

With this in mind, we will use an iterative approach, as it is more efficient and shorter. In this approach, we will create a variable called Q, in which we will enqueue the root (this to keep a reference of the tree), then we will keep iterating until the queue is empty, then we will create t (which is a reference to the current root and it will also help us to enqueue the left and right nodes, if they exist), next, we will check if t has a left or right or both child's, if it has them, we will print t and then we will enqueue the left, the right or both child's, if it doesn't have them, we won't do anything.

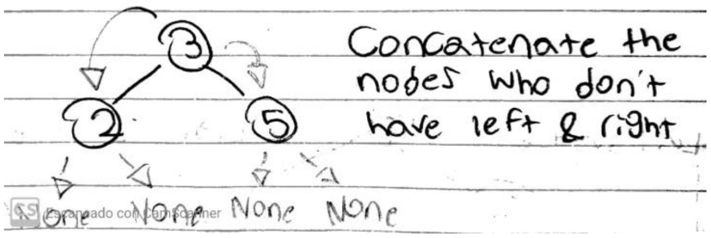
tree2List(T)



The purpose of this method is to create a sorted list containing all the data of the nodes that are in the tree. To solve this, we must take into account the three ways of traversing a tree, which are pre-order, post-order and in-order. The latter being the one we will use because the way the tree goes through is Left, Root, and then Right, this will help us start with the small value first, and therefore gradually increase until we reach the largest value.

With this in mind, we will use a recursive approach, since it is cleaner, shorter and more efficient than the iterative one (an iterative approach would be complicated and long). Therefore we will make our base case be when T is None and return an empty list, then we will return a recursive call that traverses the left subtree T[1] + a list concatenation of the current root's data T[0] + a recursive call that traverses the right subtree T[2].

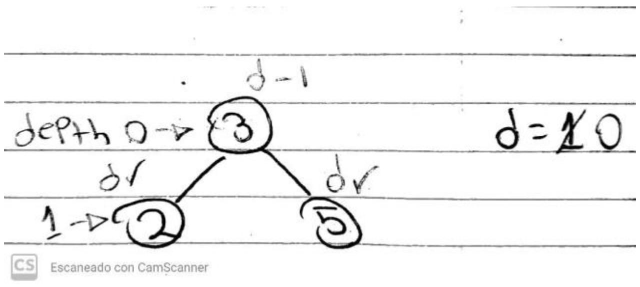
leaves(T)



The purpose of this method is to create a list which contains the leaves of the tree (node with no children). To solve this, we must traverse the entire tree and stop until the current root has no children.

With this in mind, we will use a recursive approach, as it is simpler, more efficient, and more readable than the iterative one. Therefore, we will make the edge case to be when T is None, meaning a tree does not exist, so we will return an empty list. Next, we will make our base case to be when T[1] and T[2] is None, meaning the current node is a leave because it has no left nor right child, so we will return a list that contains the data of the current root T[0]. Finally, the last condition will be when the current root has a left or right child, so we will make a recursive call that traverses the left subtree T[1] + a recursive call that traverses the right subtree T[2].

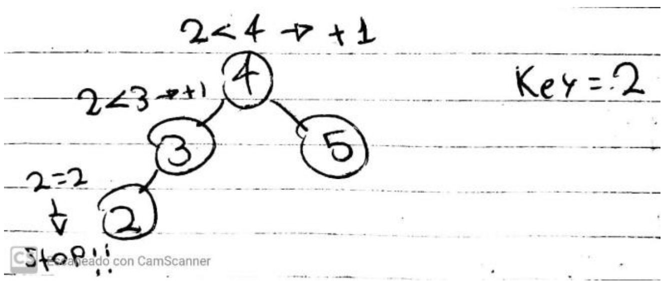
itemsAtDepthD(T,d)



The purpose of this method is to create a list containing all the elements stored in depth d. To solve this, we must take into account that d is an integer so, to reach the desired depth, we must subtract 1 each time we go down the tree. For example, if we have that d is 1 and we are at depth 0 (root), we must subtract one from d (d = 0), and then go down to depth 1 and thus add these elements to the list.

With this in mind, I will use a recursive approach since it is cleaner, shorter and more efficient than the iterative one. Therefore, we will make our base case be when T is nothing, since the tree does not exist and thus returning an empty list. If there is a tree, we will check if d is equal to 0 and we will return a list containing the current root's data T[0]. If d is not 0, then we will make two recursive calls, one in which we will traverse the left subtree and the other to traverse the right subtree, and within its parameters we will decrease d by one.

depthOfK(T,k)



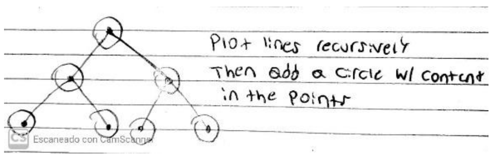
The purpose of this method is to calculate the depth of k. To solve this, we must use a search algorithm, similar to the one we used previously in the inTree (T, i) method, only this time instead to return a Boolean value, we will add 1 each time we go down in depth in the tree and if k is not in the tree we will return -1, meaning that k was not in the tree.

With this in mind, we will use an iterative approach, since I tried to implement the recursive but there was a bug and it didn't work. So then we will create a counter that stores the depth in which we are. Then, we will create a while loop that stops until T is None. Next, we will stop until k was found and we will return the value of the counter. The conditions inside the loop that we will use to traverse the tree are the following:

- If k < T[0], then traverse the left subtree and increment counter by one.
- Else, traverse the right subtree and increment counter by one.

Moreover, if we exit the while loop it means one thing: that k was not found, so we will return -1.

draw(T)



The purpose of this method is to draw the entire tree. To solve this, we will make use of the matplotlib library.

To make it more readable we will create two methods:

- draw(T) method, which has the function to check if a tree exists or not and defines the ax, x, y, change in x and the change in y.
- drawNode(ax,T,x0,y0,dx,dy) wrapper method, which has the following functions:
    - This method has the purpose of drawing the left and right branches of the tree and,
    - Draws a circle which contains the value of the current node.

With this in mind, we will use a recursive approach, due to its simplicity and how short its implementation is. Note: it should be noted that we will focus on talking only about the drawNode method, since the algorithm of drawing the tree is there.

First, this method has the condition to check if the current root has a left T[1], if it has a left then it will draw a left branch, then it will make a recursive call in which we will traverse the left subtree T[1] and we will subtract the value of x0 by dx (this to make the branch to go to the left), y0 will be subtracted dy (this to make the branch go down), we will divide dx by 2, and we will not change the value of dy.

Second, this method has the condition to check if the current root has a right T[2], if it has a right then it will draw a right branch, then it will make a recursive call in which we will traverse the right subtree T[2] and we will add x0 and dx (this to make the branch to go to the right), y0 will be subtracted dy (this to make the branch go down), we will divide dx by 2, and we will not change the value of dy.

Finally, we will use ax.text to create a circle text (representing a node), which will contain the value of the current root T[0].

Time Complexity:

While I was commenting I found that the time complexity of some methods is going to change depending if the tree is balanced or not. Therefore, I didn’t know what to put on my code so I decided to include here the time complexity of both structures.

	Unbalanced Tree	Balanced Tree
Size	O(n)	O(n)
Minimum	O(n)	Log(n)
Maximum	O(n)	Log(n)
Height	O(n)	Log(n)
inTree	O(n)	Log(n)
printByLevel	O(n)	O(n)
Tree2List	O(n)	O(n)
Leaves	O(n)	O(n)
itemsAtDepthD	O(n)	O(n)
depthOfK	O(n)	Log(n)
Draw	O(1)	O(1)
drawNode	O(n)	O(n)

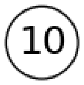
Experimental Results

Empty Tree

```
# Empty Tree
A = []
T = None

for a in A:
    print('Inserting',a)
    T = insert(T,a)
    print(T)

print(size(T)) # size -> 0
print(minimum(T)) # minimum -> Nonetype Error
print(maximum(T)) # maximum -> Nonetype Error
print(height(T)) # height -> -1
i = 1
print(inTree(T,i)) # inTree -> False
printByLevel(T) # print ->
print(tree2List(T)) # print -> []
print(leaves(T)) # leaves -> []
d = 0
print(itemsAtDepthD(T,d)) # depth -> []
k = 0
print(depthOfK(T,k))  # depth of k -> -1
draw(T) # Does not draw anything
```

```
0

TypeError: 'NoneType' object is not subscriptable

TypeError: 'NoneType' object is not subscriptable


-1
False
[]
[]
[]
hello
-1
```

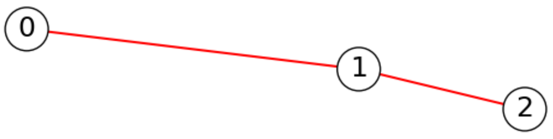
Single Node



```
# Single Node
A = [10]
T = None

for a in A:
    print('Inserting',a)
    T = insert(T,a)
    print(T)

print(size(T)) # size -> 1
print(minimum(T)) # minimum -> 10
print(maximum(T)) # maximum -> 10
print(height(T)) # height -> 0
i = 10
print(inTree(T,i)) # inTree -> True
i = 1
print(inTree(T,i)) # inTree -> False
printByLevel(T) # print -> 10
print()
print()
print(tree2List(T)) # print -> [10]
print(leaves(T)) # leaves -> [10]
d = 0
print(itemsAtDepthD(T,d)) # depth -> [10]
d = 1
print(itemsAtDepthD(T,d)) # depth -> []
k = 10
print(depthOfK(T,k))  # depth of k -> 0
k = 1
print(depthOfK(T,k))  # depth of k -> -1
draw(T)
```

```
Inserting 10
[10, None, None]
1
10
10
0
True
False
10

[10]
[10]
[10]
[]
0
-1
```

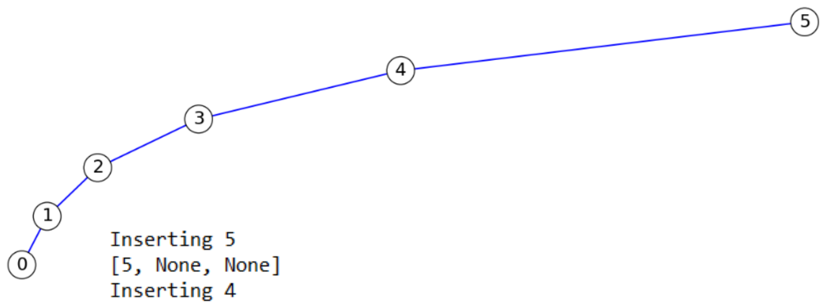
Unbalanced Tree 1



```
# Unbalanced Tree
A = [0,1,2]
T = None

for a in A:
    print('Inserting',a)
    T = insert(T,a)
    print(T)

print(size(T)) # size -> 3
print(minimum(T)) # minimum -> 0
print(maximum(T)) # maximum -> 2
print(height(T)) # height -> 2
i = 2
print(inTree(T,i)) # inTree -> True
i = 3
print(inTree(T,i)) # inTree -> False
printByLevel(T) # print -> 0 1 2
print()
print(tree2List(T)) # print -> [0, 1, 2]
print(leaves(T)) # leaves -> [2]
d = 1
print(itemsAtDepthD(T,d)) # depth -> [1]
d = 3
print(itemsAtDepthD(T,d)) # depth -> []
k = 2
print(depthOfK(T,k))  # depth of k -> 2
k = 1
print(depthOfK(T,k))  # depth of
draw(T)
```

```
Inserting 0
[0, None, None]
Inserting 1
[0, None, [1, None, None]]
Inserting 2
[0, None, [1, None, [2, None, None]]]
3
0
2
2
True
False
0 1 2
[0, 1, 2]
[2]
[1]
[]
2
1
```

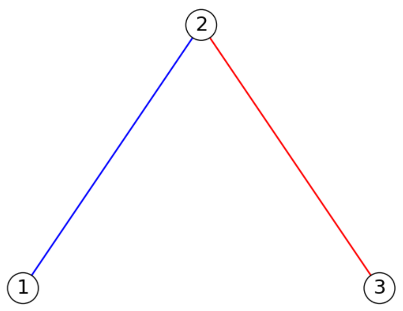
Unbalanced Tree 2



```
# Unbalanced Tree
A = [5,4,3,2,1,0]
T = None

for a in A:
    print('Inserting',a)
    T = insert(T,a)
    print(T)

print(size(T)) # size -> 6
print(minimum(T)) # minimum -> 0
print(maximum(T)) # maximum -> 5
print(height(T)) # height -> 5
i = 2
print(inTree(T,i)) # inTree -> True
i = -1
print(inTree(T,i)) # inTree -> False
printByLevel(T) # print -> 5 4 3 2 1 0
print()
print(tree2List(T)) # print -> [0, 1, 2, 3, 4, 5
print(leaves(T)) # leaves -> [0]
d = 2
print(itemsAtDepthD(T,d)) # depth -> [3]
d = 7
print(itemsAtDepthD(T,d)) # depth -> []
k = 2
print(depthOfK(T,k))  # depth of k -> 3
k = 7
print(depthOfK(T,k))  # depth of k -> -1
draw(T)
```

```
Inserting 5
[5, None, None]
Inserting 4
[5, [4, None, None], None]
Inserting 3
[5, [4, [3, None, None], None], None]
Inserting 2
[5, [4, [3, [2, None, None], None], None], None]
Inserting 1
[5, [4, [3, [2, [1, None, None], None], None], None], None]
Inserting 0
[5, [4, [3, [2, [1, [0, None, None], None], None], None], None], None]
6
0
5
5
True
False
5 4 3 2 1 0
[0, 1, 2, 3, 4, 5]
[0]
[3]
[]
3
-1
```

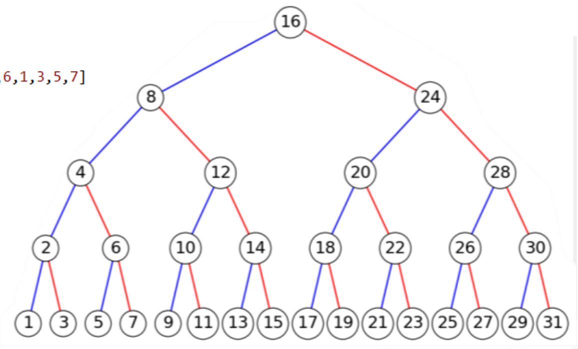
Balanced Tree 1



```
# Balanced Tree
A = [2,1,3]
T = None

for a in A:
    print('Inserting',a)
    T = insert(T,a)
    print(T)

print(size(T)) # size -> 3
print(minimum(T)) # minimum -> 1
print(maximum(T)) # maximum -> 3
print(height(T)) # height -> 1
i = 1
print(inTree(T,i)) # inTree -> True
i = 4
print(inTree(T,i)) # inTree -> False
printByLevel(T) # print -> 2 1 3
print()
print(tree2List(T)) # print -> [1, 2, 3]
print(leaves(T)) # leaves -> [1, 3]
d = 0
print(itemsAtDepthD(T,d)) # depth -> [2]
d = 1
print(itemsAtDepthD(T,d)) # depth -> [1, 3]
k = 3
print(depthOfK(T,k))  # depth of k -> 1
k = 5
print(depthOfK(T,k))  # depth of k -> -1
draw(T)
```

```
Inserting 2
[2, None, None]
Inserting 1
[2, [1, None, None], None]
Inserting 3
[2, [1, None, None], [3, None, None]]
3
1
3
1
True
False
2 1 3
[1, 2, 3]
[1, 3]
[2]
[1, 3]
1
-1
```

Balanced Tree 2



```
# Balanced Tree
A = [16,24,20,28,18,22,26,30,17,19,21,23,25,27,29,31,8,12,10,14,11,13,9,15,4,2,6,1,3,5,7]
T = None

for a in A:
    print('Inserting',a)
    T = insert(T,a)
    print(T)

print(size(T)) # size -> 31
print(minimum(T)) # minimum -> 1
print(maximum(T)) # maximum -> 31
print(height(T)) # height -> 4
i = 17
print(inTree(T,i)) # inTree -> True
i = 32
print(inTree(T,i)) # inTree -> False
printByLevel(T) # print -> 16 8 24 4 12 20 28 2 6 10 14 18 22 26 30 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
print()
print(tree2List(T)) # print -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
print(leaves(T)) # leaves -> [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
d = 2
print(itemsAtDepthD(T,d)) # depth -> [4, 12, 20, 28]
d = 3
print(itemsAtDepthD(T,d)) # depth -> [2, 6, 10, 14, 18, 22, 26, 30]
k = 7
print(depthOfK(T,k))  # depth of k -> 4
k = 10
print(depthOfK(T,k))  # depth of k -> 3
draw(T)
```

```
31
1
31
4
True
False
16 8 24 4 12 20 28 2 6 10 14 18 22 26 30 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
[4, 12, 20, 28]
[2, 6, 10, 14, 18, 22, 26, 30]
4
3
```

Conclusion

In conclusion, this laboratory helped me to practice and reinforce my knowledge and understanding of this data structure. In addition, it also helped me understand why this structure is very important in computer science and these reasons are as follows: with a binary search tree we can speed up the search for an element because in each iteration we ignore half of the tree and the same thing happens When we insert or remove a node, we can also add all the nodes we want, however, if the tree is not balanced it will not be efficient and the speed will be slower. Also, thanks to the implementation of the binary search tree in a list, I realized that we can implement any data structure in any way, the way we do it doesn't matter as long as it follows the same concept.

Appendix

```
#   Course: CS 2302
#   Assignment: Lab IV
#   Author: Oswaldo Escobedo
#   Instructor: Dr. Fuentes
#   TA: Harshavardhini Bagavathyraj
#   Date of Last Modification: 03/23/2020
#   Purpose of the Program: Implementation of binary search trees using lists
import matplotlib.pyplot as plt
import numpy as np


def insert(T,newItem): # Insert newItem to BST T
    if T == None:  # T is empty
        T = [newItem,None,None]
    else:
        if newItem< T[0]:
            T[1] = insert(T[1],newItem) # Insert newItem in left subtree
        else:
            T[2] = insert(T[2],newItem) # Insert newItem in right subtree
    return T

def inOrder(T):
    if T!=None:
        inOrder(T[1])
        print(T[0],end=' ')
        inOrder(T[2])

def size(T): # Returns the number of nodes in the tree, O(n)
    if T is None: # T is empty
        return 0
    return 1 + size(T[1]) + size(T[2]) # Traverses left and right sub-trees

def minimum(T): # Returns the smallest item in the tree, O(n)
    if T[1] is None: # There is no left subtree
        return T[0]
    return minimum(T[1]) # Traverses left subtree

def maximum(T): # Returns the largest item in the tree, O(n)
    if T[2] is None: # There is no right subtree
        return T[0]
    return maximum(T[2]) # Traverses right subtree

def height(T): # Returns the height of the tree, O(n)
```

```
    if T is None: # T is None
        return -1
    else: # Chooses the largest height from left and right subtree
        return 1 + max(height(T[1]),height(T[2]))

def isTree(T,i): # Searches if i is in the tree, O(n)
    if T is None: # T is None or i was not in the tree
        return False
    if T[0] == i: # Item was found
        return  True
    if T[0] > i: # Traverse left subtree
        return isTree(T[1],i)
    else: # Traverse right subtree
        return isTree(T[2],i)

def printByLevel(T): # Prints data items in level order traversal, O(n)
    Q = [T] # Enqueues root
    while len(Q) > 0: # While the queue is not empty
        t = Q.pop(0) # Dequeues front of queue
        if t!=None:
            print(t[0],end = ' ')
            Q.append(t[1]) # Enqueues left child
            Q.append(t[2]) # enqueues right child

def tree2List(T): # Returns a list of the tree's item in ascending order, O(n)
    if T is None: # T is None
        return []
    return tree2List(T[1]) + [T[0]] + tree2List(T[2]) # inOrder traversal

def leaves(T): # Returns a list containing the leaves of the tree, O(n)
    if T is None: # T is None
        return []
    if T[1] is None and T[2] is None: # Node is leave
        return [T[0]]
    return leaves(T[1]) + leaves(T[2]) # Node is not a leave

def itemsAtDepthD(T,d): # Returns a list containing the items at depth d, O(n)
    if T is None: # T is None
        return []
    else:
        if d == 0: # We are at the desired depth
            return [T[0]]
        return itemsAtDepthD(T[1],d-1) + itemsAtDepthD(T[2],d-1) # Traverses tree

def depthOfK(T,k): # Returns the depth of k in the tree, O(n)
    if T is None: # T is None
        return -1
    count = 0
    while T is not None:
        if k == T[0]: # k is in current node
            return count
        elif k < T[0]: # k is less than current node
            count += 1
            T = T[1]  # traverse left subtree
        else:
            count += 1
            T = T[2] # traverse right subtree
    return -1 # k was not in the tree

def draw(T): # Creates the ax, plt, and parameters of drawNode(), O(1)
    if T is not None:
        fig, ax = plt.subplots()
        drawNode(ax, T, 0, 0, 1000, 120)
        ax.axis('off')

def drawNode(ax,T,x0,y0,dx,dy): # Draws the tree, O(n)
    if T[1] is not None: # Left child exists
        ax.plot([x0-dx,x0],[y0-dy,y0],linewidth=1.5,color='b') # Draws left branch
        drawNode(ax, T[1], x0-dx, y0-dy, dx/2, dy) # Traverses left subtree
    if T[2] is not None: # Right child exists
        ax.plot([x0+dx,x0],[y0-dy,y0],linewidth=1.5,color='r') # Draws right branch
        drawNode(ax, T[2], x0+dx, y0-dy, dx/2, dy) # Traverses right subtree
    ax.text(x0,y0, str(T[0]), size=18,ha="center", va="center", # Draws a circle (node)
        bbox=dict(facecolor='w',boxstyle="circle"))  # containing the node's data



if __name__ == "__main__":

    plt.close('all')

    A = [11, 6, 16, 17, 2, 4, 18, 14, 8, 15, 1, 20, 13]
```

```
T = None

for a in A:
    print('Inserting',a)
    T = insert(T,a)
    print(T)

inOrder(T)
print()
print('Tree size: ', size(T))
print('Minimum: ', minimum(T))
print('Maximum: ', maximum(T))
print('Height: ', height(T))
i = 17
print('is',i,'? ',isTree(T,i))
print('print by level: ',end='')
printByLevel(T)
print()
print('tree to list: ',tree2List(T))
print('leaves: ',leaves(T))
d = 1
print('items at depth',d,'is:',itemsAtDepthD(T,d))
k = 3
print('Depth of',k,'is:',depthOfK(T,k))
draw(T)
```