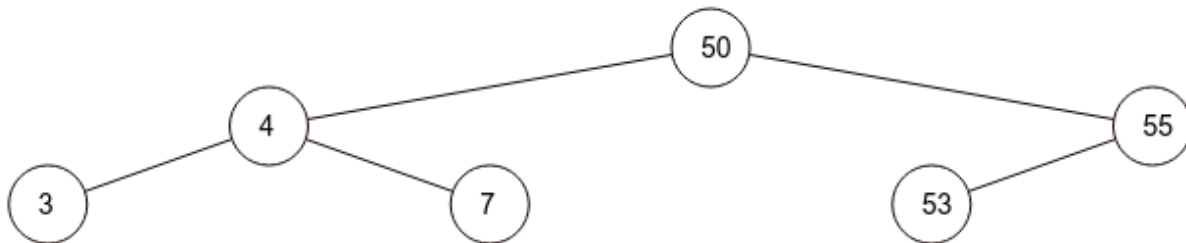
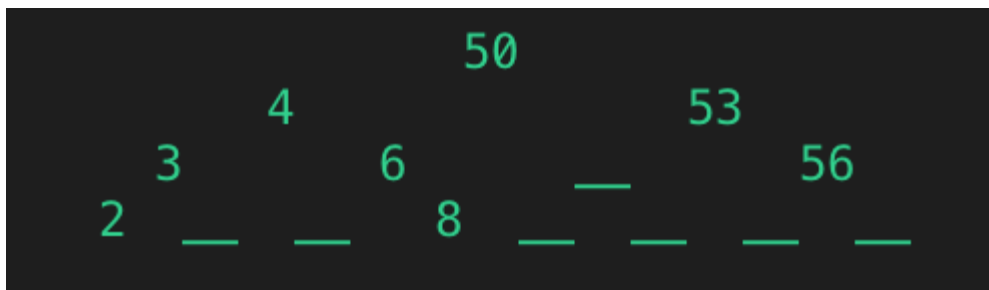

Vitamins (30 minutes)

1. Given the following Binary Search Tree, perform the following operations cumulatively:



Perform the following operations (10 minutes):

1. Insert 2
2. Delete 7
3. Insert 6
4. Insert 8
5. Delete 55
6. Insert 56



7. Pre Order Traversal 50 4 3 2 6 8 53 56 CLR
8. Post Order Traversal 2 3 8 6 4 56 53 50 LRC
9. In Order Traversal 2 3 4 6 8 50 53 56 LCR
10. Level Order Traversal 50 4 53 3 6 56 2 8

2. (10 minutes)

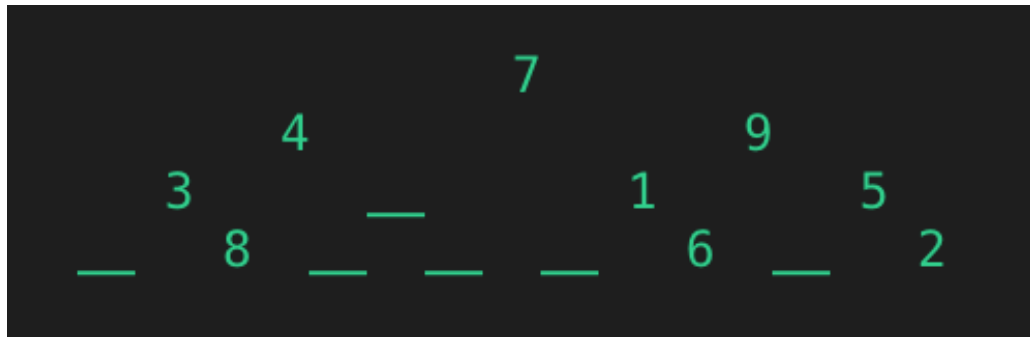
a. Given the following traversal of a **binary tree**. Restore the tree:

Preorder:

7 4 3 8 9 1 6 5 2

Inorder:

3 8 4 7 1 6 9 5 2



b. If you are given just the preorder, can this **binary tree** be unique? If not, give a counterexample (two different trees with this same preorder).

Preorder:

5 2 1 3 4 9 7 6 8

No, they can slanted to the left of right like so:

slanted to the right preorder

5
2
1
...

slanted to the left preorder

5
2
1
...

- c. Now, if you are given just the preorder of a **binary search tree**, will this tree be unique?

Preorder:

5 2 1 3 4 9 7 6 8

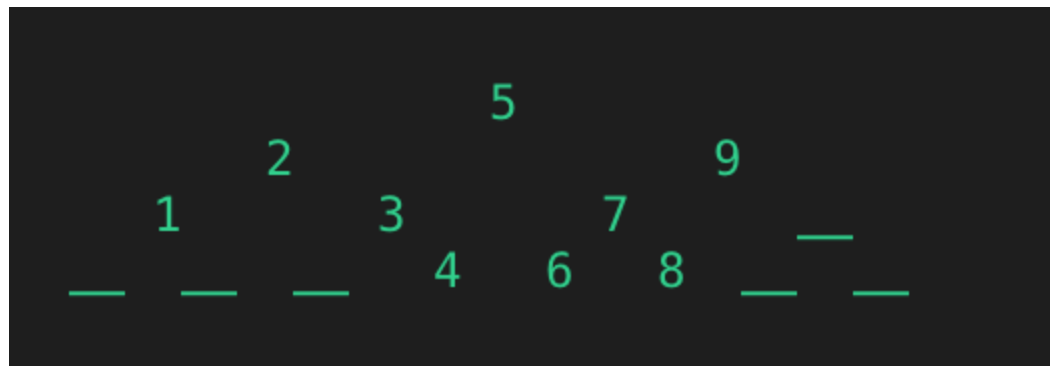
Inorder:

1 2 3 4 5 6 7 8 9 (implicitly given)

Yes, because it is a binary search tree. If you just have the preorder of a binary tree, the tree may not be unique because it can have any inorder traversal. With both the preorder and inorder traversals given, you can determine the binary tree.

In the case of the binary search tree, if you are given the preorder, it can only have one inorder traversal (bst property) so therefore, it has to be unique.

USE THE STEPS WRITTEN IN LAB 10 TO GUIDE THEM TO THE SOLUTION.



3. A student suggested the following implementation for checking whether a `LinkedBinaryTree` object is also a Binary Search Tree. (10 minutes)

```
def is_BST(root):
    if (root.left is None and root.right is None):
        return True

    elif root.left and root.right:
        check_left = root.left.data < root.data
        check_right = root.right.data > root.data
        return check_left and check_right and is_BST(root.left)
    and is_BST(root.right)

    elif root.left:
        check_left = root.left.data < root.data
        return check_left and is_BST(root.left)

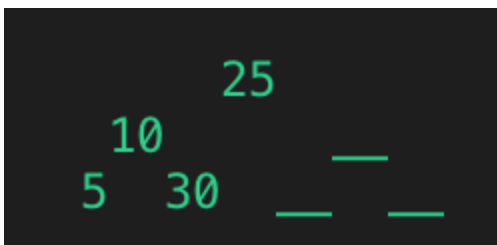
    elif root.right:
        check_right = root.right.data > root.data
        return check_right and is_BST(root.right)
```

Is there a problem with this function? If so, draw a simple binary tree that will cause this function to return an incorrect result.

ex) either draw a **BST** that makes `is_BST` return **False**

or a **non BST** that makes `is_BST` return **True**.

Yes, this code only checks the immediate left and right children for comparison. The code will return **True** instead of **False** for a binary tree like this:



Coding (See CS1134 Lab12 Solutions.py)

In this section, it is strongly recommended that you solve the problem on paper before writing code.

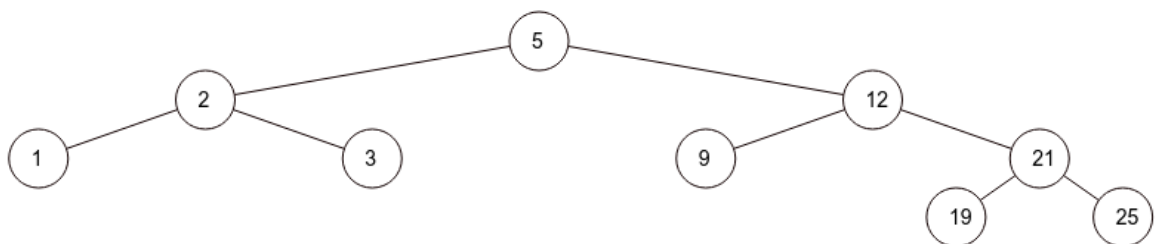
Download the **BinarySearchTreeMap.py** file under Resources/Lectures on NYU Classes

1. Given a non-empty BinarySearchTreeMap, implement a function that will return a tuple containing the minimum and maximum key in the tree. The function should be **iterative**. (10 minutes)

```
def min_max_BST(bst):  
    ''' Returns a tuple containing the min and max keys in the  
    binary search tree'''
```

What is the worst-case run time of your function?

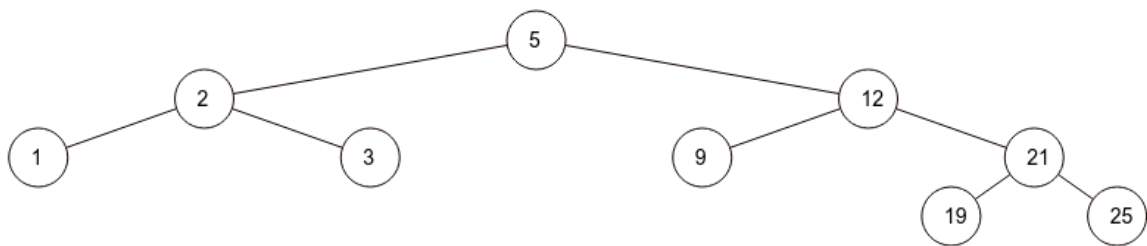
ex) min_and_max will return (1, 25) for this Binary Search Tree



2. Given a binary search tree and a number, n , write an **iterative** function that will find the greatest number in the binary search tree that is less than or equal to n . If there is no value less than N , then return -1. Your implementation should run in $O(h)$ worst case, where h is the height of the tree. There are no constraints on space. (15 minutes)

```
def glt_n(bst, n): #glt = greatest less than
''' Returns the greatest number in the binary search tree
less than or equal to n'''
```

ex)



Input : $N = 21$

Output : 21

Input: $N = 4$

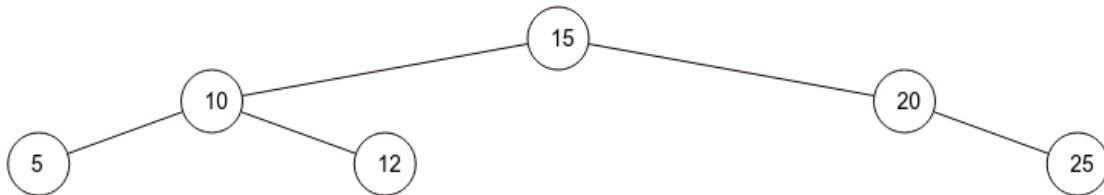
Output: 3

3. Given two Binary Search Trees consisting of unique positive elements, write a program to check whether the two BSTs contain the same set of elements or not. Although there are no time or space constraints, what is the worst-case run time and extra space complexity of your function? (30 minutes)

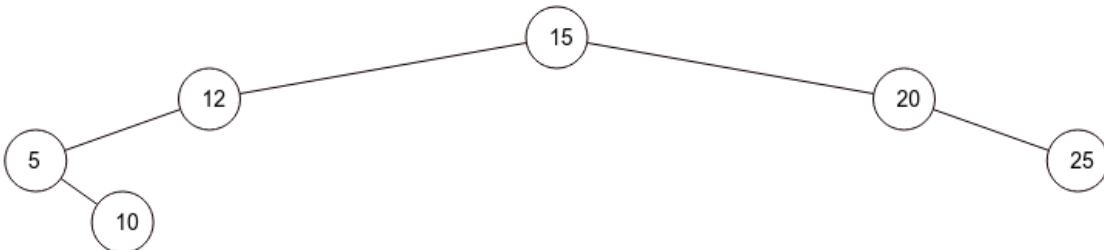
```
def compare_BST(bst1, bst2):  
    ''' Returns true if the two binary search trees contain the  
    same set of elements and false if not'''
```

Given the two trees, the program should return True.

BST 1



BST 2



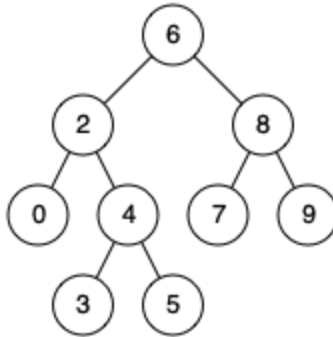
4. Let's fix the `is_BST` function that was incorrectly defined in the vitamins section of this lab. Given the root node of a `LinkedBinaryTree`, implement a function that will return `True` if the tree is a Binary Search Tree and `False` if not. (40 minutes)

The helper function should return a tuple triplet containing the min and max of the current subtree, and a bool value. (min, max, bool)

```
def is_BST(root):  
    return is_BST_helper(root)[2]  
  
def is_BST_helper(root):  
    ''' Returns a tuple (min, max, bool) '''
```


OPTIONAL

5. Given two nodes in a tree, implement a function to find their lowest common ancestor. The lowest common ancestor is the deepest node from which one can get to the two input nodes. Your implementation should run in **$O(h)$ worst case**.



- A. Implement such function for two nodes in a **BST** (Binary Search Tree)

```
def lca_BST(bst, node1, node2):
```

- B. Implement such function for two nodes in a **BT** (Binary Tree)

```
def lca_BT(bst, node1, node2):
```

Note: Don't worry about the run-time and extra space complexity. Simply come up with solutions, which you may improve upon if you're able to.

Hint: Optimal implementations for parts A and B **should not** follow the same approach. For part A, you should utilize the property of a Binary Search Tree.