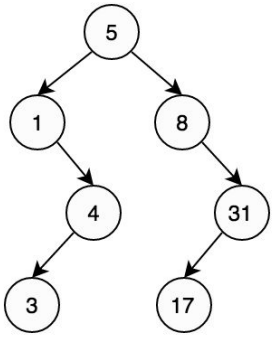
- This lab will cover Binary Search Trees.
- You may want to refer to the text and your lecture notes during lab as you solve the problems.
- When approaching the problems, think before you code. It is good practice and generally helpful to lay out possible solutions for yourself.
- You should write test code to try out your solutions.
- You must stay for the duration of the lab. If you finish early, you can help other students to complete the lab. If you don't finish by the end of the lab, it is recommended that you complete the lab on your own time.
- Your TAs are available to answer your questions in lab, during office hours, and on Piazza.

Vitamins (maximum 30 minutes)

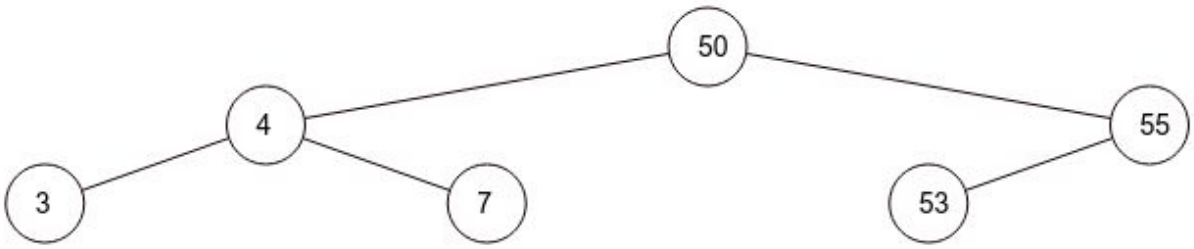
---

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



   i. Insert 2
   ii. Insert 7
   iii. Delete 8
   iv. Insert 8
   v. Delete 1

2. Given the following Binary Search Tree:



Perform the following operations:

1. Insert 2
2. Delete 7
3. Insert 6
4. Insert 8
5. Delete 55
6. Insert 56
7. Pre Order Traversal
8. Post Order Traversal
9. In Order Traversal

3. Insert the keys 5, 3, 9, 6, 2, 8 in that order into an initially empty binary search tree. After each insert, evaluate whether the resulting tree violates the properties of an AVL tree? If so, at which node does an imbalance occur?

Coding

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

1. The goal of this exercise is to plot a graph of tree heights vs tree sizes. To do this, you will generate several lists with random numbers, ranging between 1 and *n*, and insert each list into a binary search tree. Do this for at least ten *n*s ranging between 8 and 32768. (For every *n* you will create a **new** binary search tree.) For each binary search tree you create and populate, measure its height using the `height()` method from class. Export the results to a *csv* file and plot the graph. You may use the following code:

```
import csv

def create_csv(sizes_lst, heights_lst):
    file = open('heights.csv','w')
    if len(sizes_lst) != len(heights_lst):
        raise Exception("lists given must be of equal sizes")
    for i in range(len(sizes_lst)):
        file.write(str(sizes_lst[i]) + "," + str(heights_lst[i]) +
'\n')
    file.close()
```

Note the the *i*th element in `sizes_lst` must correspond to the *i*th element in `heights_lst`.

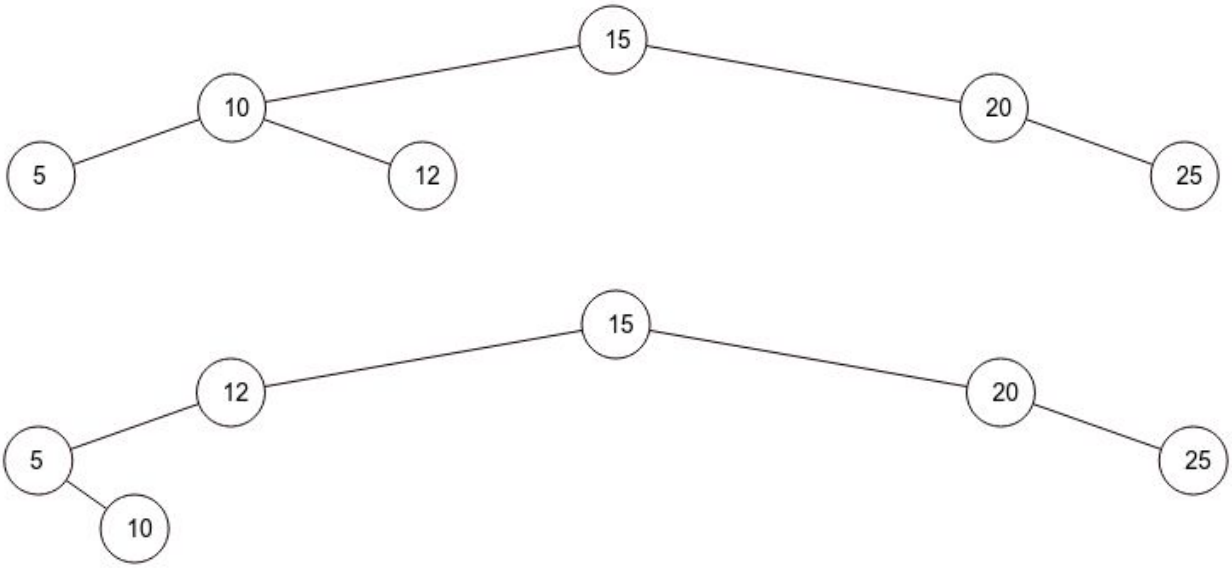
- What function best describes the resulting graph?
- What function would best describe the resulting graph if the values inserted were mostly increasing? decreasing?

2. Give an implementation for the recursive helper function `is_bst_helper` which is called by the function `is_bst` to determine whether a binary tree is a binary search tree. Your implementation should have **linear time** complexity.

```
def is_bst(binary_tree):
    return is_bst_helper(binary_tree.root)[0]
def is_bst_helper(curr_root):
    #your code here
```

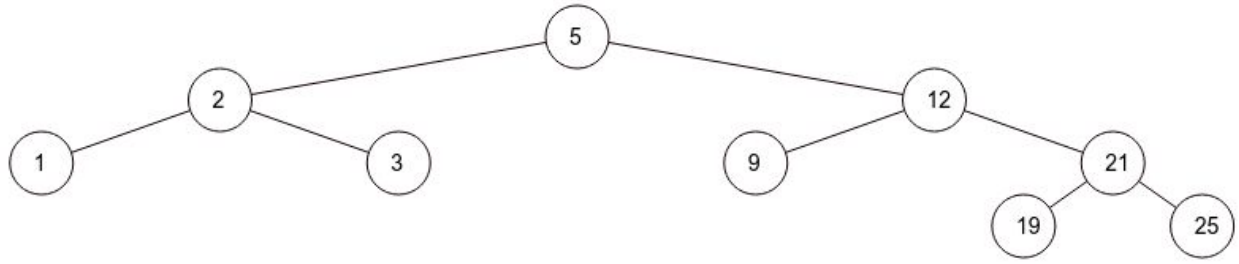
Hint: `is_bst_helper` should return a tuple which gives the max and min of the subtree and a boolean value.

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. Attempt to write the program in θ(n) time complexity. While there are no constraints on space, try to analyse your solution's space complexity.



Given the two trees, the program should return True.

Extra: Given a binary search tree and a number N. Write a program to 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. Give an implementation that runs in O(h) worst case, where h is the height of the tree. There are no constraints on space.



Examples: For the above given binary search tree

Input : N = 24

Output :result = 21

Input : N = 4

Output : result = 3