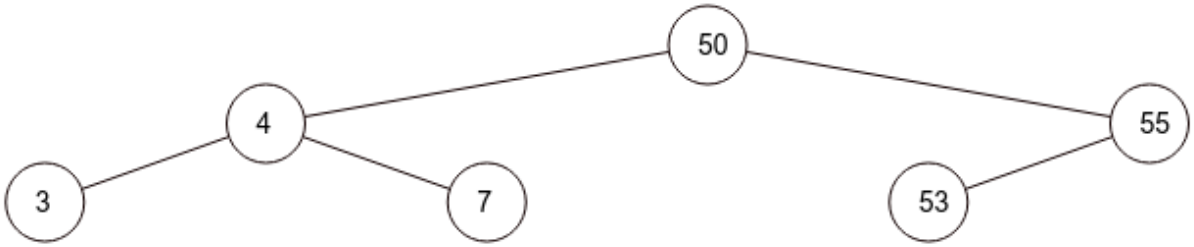
- This lab will cover Binary Trees, Binary Search Trees and Hash Tables
- It is assumed that you have reviewed chapter 8 & 11 of the textbook. You may want to refer to the text and your lecture notes during the lab as you solve the problems.
- When approaching the problems, think before you code. Doing so is good practice and can help you lay out possible solutions.
- Think of any possible test cases that can potentially cause your solution to fail!
- Your TAs are available to answer questions in the lab, and during office hours.

Part A
Binary Trees and Binary Search Trees
(2:00 PM - 4:10 PM)

Vitamins (30 minutes)

---

1. Given the following Binary Search Tree, 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
8. Post Order Traversal
9. In Order Traversal
10. Level Order Traversal

2. (10 minutes)

   a. If you are given just the preorder of a binary tree, 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

   b. Now, if you are given just the preorder of a **binary search tree**, will this tree be unique? If so, draw the binary search tree.

      Preorder:
      5 2 1 3 4 9 7 6 8

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**.

Coding (65 minutes)

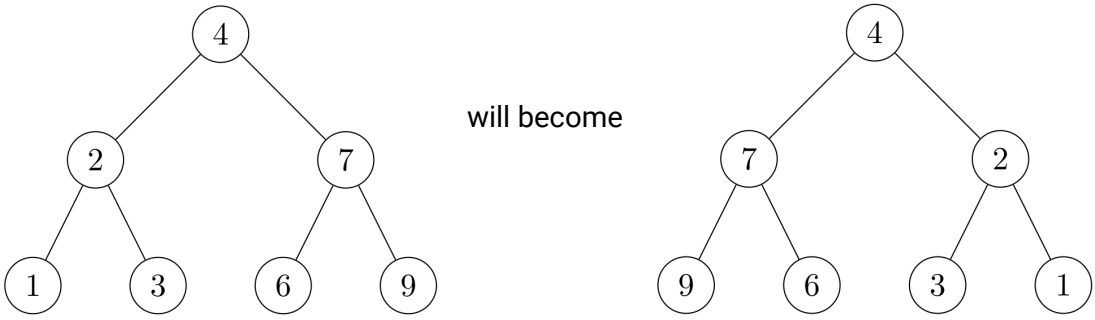
In this section, it is strongly recommended that you solve the problem on paper before writing code.

Download the **Lab6.py** file on Brightspace. This file contains two user-defined classes: `LinkedBinaryTree` and `BinarySearchTreeMap`.
Import the file like this: `from Lab6 import *`

1. Given the root node of a `LinkedBinaryTree`, root, write a **recursive** function that will invert the tree in-place (mutate the input tree). (15 minutes)

```
def invert_bt(root):
    ''' Inverts the binary tree using recursion '''
```

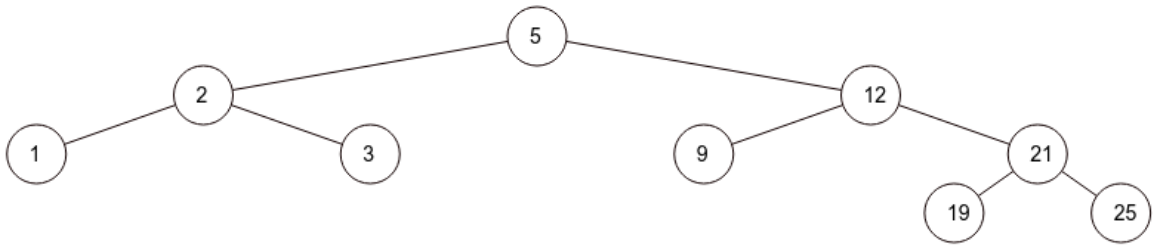
ex)



2. a) Given a non-empty `BinarySearchTreeMap` object, bst, write an **iterative** function that will return a tuple containing the minimum and maximum key of bst. Your implementation should run in O(h) worst case, where h is the height of bst. There are no constraints on space. (15 minutes)

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

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



b) Now, if you were to implement a similar function that returns the minimum and maximum values of a `LinkedBinaryTree` rather than a `BinarySearchTreeMap`, how would your algorithm change? What would be the worst-case runtime of that function? (5 minutes)

Optional

c) Implement the function described in part b. This function will be given root, the root node of a `LinkedBinaryTree`.

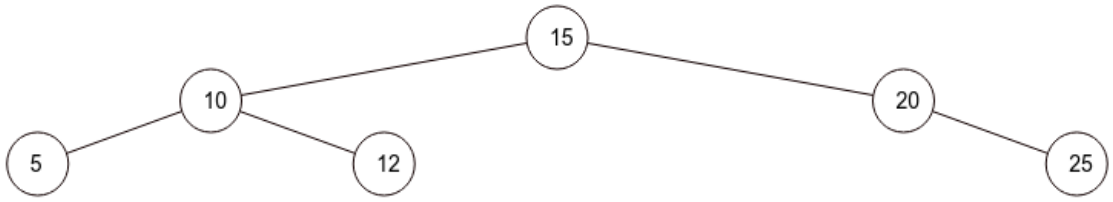
```
def min_max_BT(root):
''' Returns a tuple containing the min and max values in
the binary tree'''
```

3. Given two `BinarySearchTreeMap` objects, bst1 and bst2, consisting of unique positive elements, write a function that will check whether the two BSTs contain the same set of elements or not. (15 minutes)

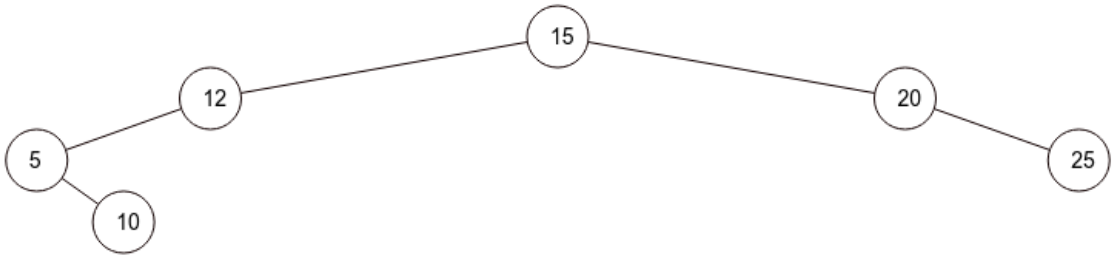
```
def compare_BST(bst1, bst2):
''' Returns true if the two binary search trees contain the
same set of elements and false if not'''
```

ex) Given the two BSTs, 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`, root, write a function that will return True if the tree is a BST and False if not. (15 minutes)

```
def is_BST(root):
''' Returns True if the tree is a BST and False if not'''
```

Hint: You should define a helper function that returns a tuple containing 3 elements, min, max, and bool.

Part B
Hash Tables
(4:20 PM - 5:30 PM)

Vitamins (20 minutes)

---

1. Given a hash table T with 25 slots that stores 2000 elements, the load factor α for T is _____. (3 minutes)

2. Provide the following **average-case** and **worst-case** runtime for the following methods. If the two runtimes are different, explain when the worst-case runtime happens and why. (10 minutes)

```
a.   def __len__(self):
```

```
b.   def is_empty(self):
```

```
c.   def __getitem__(self, key):
```

```
d.   def __setitem__(self, key, value):
```

```
e.   def __delitem__(self, key):
```

```
f.   def __contains__(self, key):
```

3. Analyze the average-case runtime of the following function, and give it an appropriate name (7 minutes):

```
def mystery(s1, s2):
    fMap = {} #fMap = frequency Map

    for char in s1:
        if char not in fMap:
            fMap[char] = 0

        fMap[char] += 1

    for char in s2:
        if char not in fMap:
            return False

        fMap[char] -= 1

    for key in fMap:
        if fMap[key] != 0:
            return False

    return True
```

What are the outputs of the following?

```
print(mystery("cheaters", "teachers"))

print(mystery("engineering", "gnireenigne"))

print(mystery("Python", "nohtyp"))
```

Coding (35 minutes)

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

1. Given a list of numbers, lst, write an **iterative** function that will return the number that appears most frequently in lst. Your implementation should run in O(n) average case. You can assume that the most frequent number in lst is unique. (10 minutes)

```
def most_frequent(lst):
      fmap = {}
      ...
```

```
Input: lst = [5,9,2,9,0,5,9,7]
Output: 9

Explanation: 9 appears the most in the array
```

2. Given a list of numbers, lst, write an **iterative** function that will return the first number that is not repeated in lst. Your implementation should run in O(n) average case. You may assume that there will be at least 1 number in lst that is not repeated. (10 minutes)

```
def first_unique(lst):
      fmap = {}
      ...
```

```
Input: lst = [5,9,2,9,0,5,9,7]
Output: 2

Explanation: 2 is the first non-duplicate number
```

3. Given a list of integers, lst, and a target value, target, write a function that will return a tuple of indices of two numbers that sum up to target, or (None, None) if there is no solution. Your implementation should run in O(n) average case. (15 minutes)

```
def two_sum(lst, target):
    seen = {}
    ...
```

```
Input: lst =  [-2, 11, 15, 21, 20, 7], target = 22
Output: (2,5)
```

```
Explanation: Indices 2 and 5 hold numbers 15 and 7, which both sum up to 22.
```

```
Input: lst =  [-2, 11, 15, 21, 20, 20], target = 22
Output: (None, None)
```