CS104: Data Structures & Object-Oriented Programming Summer 2020 – Practice Problems

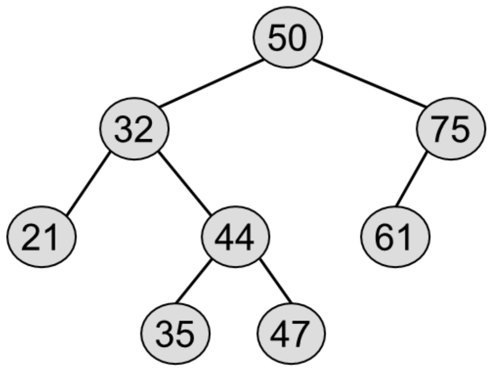
Your solutions should be as efficient as possible [i.e. not O(n^2) if it can reasonably be done in O(n)]. You will lose all or at least significant points, otherwise.

1. **AVL Trees** [9 pts].

Use the tree below for the initial state of an AVL tree before each of the following questions/tasks.

a. (3 pts) Give a key that could be **inserted** that would lead to an imbalance that requires a **double rotation (zig-zag)** case. Many values may exist, list only one: ____________.

b. (6 pts) Show the resulting AVL tree if a **remove(50)** operation is performed. Assume the remove algorithm swaps with the successor of a node, if need be. You **MUST** show what the tree looks like at intermediate steps. This will also help us assign partial credit if your final answer is wrong.



2. **[8 pts.] Hashing**

Assume a hash set of integer keys with initial table size, m=7, and a hash function, h(k) = (3*k) % m. The hash set uses double-hashing to resolve collisions with h2(k) = m - (k%m). Further assume that when the table reaches a loading factor of 0.5 and it will resize to a new size of 2*m+3 before a new insertion is applied (an insertion may end with a loading factor of >= 0.5 but the next insertion will then cause the resize). For the sequence below, fill out the tables below the contents of the table just before it resizes and after the last insertion. Upon resize, the keys present will be inserted in the new table in the order they appear in the old table.

Insert the following keys: 5, 22, 12, 15, 9, 6

Before Resizing (m=7) [Use only the rows necessary]

Key	h(k)	Probe sequence if necessary (Show sequence of array locations probed ending with final location where the key is placed)

After Resizing (m=17) – Show all keys rehashed in order + newly inserted keys

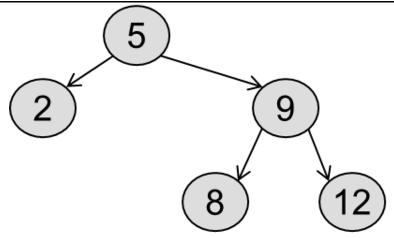
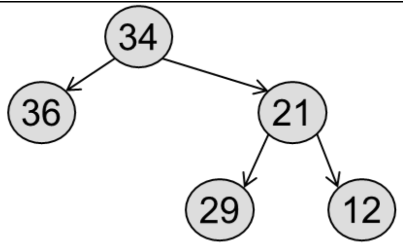
Key	h(k)	Probe sequence if necessary (Show sequence of array locations probed ending with final location where the key is placed)

Scratch work diagram:

0	1	2	3	4	5	6

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

3. **(12 pts.) Binary Search Trees**: Assume you are given a valid binary search tree (may not be balanced), where each node is an instance of the Node struct given below. Write a **recursive** function (or recursive helper function) that will:
    a. return the sum of all the nodes and
    b. update the value in each node with the **sum** of the all the values in the original tree *greater than and including* its own value. In the example below, 2's value becomes the sum of all values in the tree, 5's value becomes the sum of itself and all the values greater than it (i.e. 8, 19, 12), etc.

	
1 Possible Example of the Original Tree	**Expected resulting tree**. Return value should be **36**

```
// assume appropriate #includes
struct Node {
  int val;   Node* left;  Node* right;
};

// prototype helper functions here, if needed


int sumLargeToSmall(Node* root)
{
```