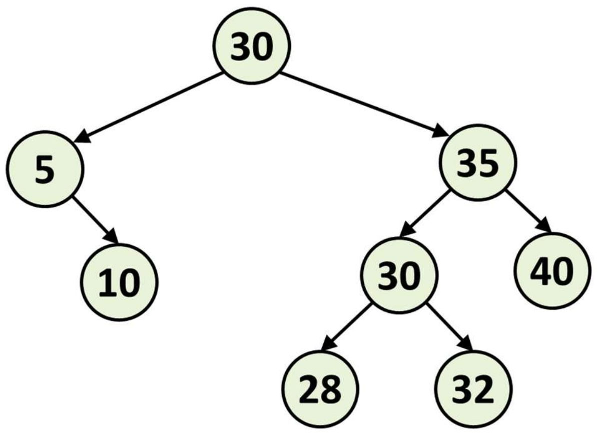
National University of Singapore
School of Computing
IT5003: Data Structure and Algorithm
Semester I, 2019/2020

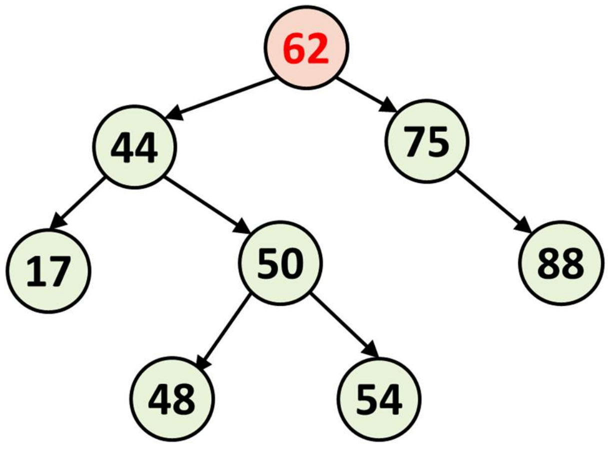
Tutorial x Lab 6
AVL and Hashing

General idea: Tutorial type questions will be listed first in this document. Some tutorial question will then be implemented in the lab portion (later part of this document). Lab exercise cases to be submitted on Coursemology will be marked [SUBMISSION] at the front.

1. **[AVL Tree Insertion]** Show the rotation(s) performed when "33" is inserted into the following AVL Tree:

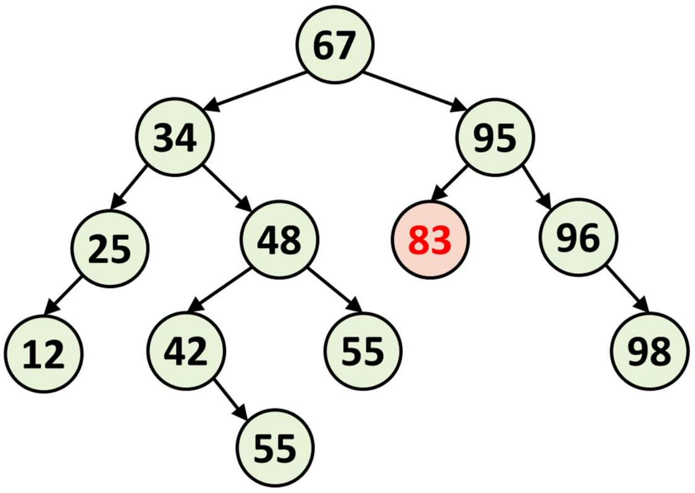


2. **[AVL Tree Deletion]** Show the AVL Tree resulting from the removal of the entry with key 62 from the following AVL tree.



3. **[AVL Tree Deletion – General]**

   a. Given the following AVL tree, if node 83 is deleted, how many rotations are needed to obtain an AVL tree again? Draw the tree after each rotation.



   b. In general, how many sub-trees will need to be re-balanced in the worst case when an item is deleted from an AVL tree? Give an example to support your argument.

4. **[Hash Collision Resolution]** For the following parts, you are given a hash table of size *m = 11* and the hash function ***h(key) = key % m***. Perform the sequence of operations specified with the hash collision method given in the question. Each part **start from an empty hash table**.

   a. Insert keys { 10, 22, 31, 4, 15, 28, 17, 88, 59 } in sequence. Then delete key 4, followed by find key 37. Use **separate chaining** to solve the collision problem.

   b. *Insert(17), Insert(37), Insert(59), Insert(70), Find(60), Delete(59), Find(70), Insert(16)*. Use ***linear probing*** to resolve collision.

   c. *Insert(20), Insert(82), Insert(28), Insert(93), Find(51), Delete(82), Find(93), Insert(24), Insert(68)* . Use ***quadratic probing*** to resolve collision.

   d. *Insert(32), Insert(49), Insert(65), Insert(26), Find(37), Delete(26), Insert(98).* Use ***double hashing*** to resolve collision. The secondary hash function is **h_2(key) = 7 – (key % 7).**

5. **[Hash Function Analysis]**

   a. Consider the hashing function h(key) = (key^2) % 11, where the table size is 11 and the keys are integers from 0 to 10. Is this a ***perfect* hashing function**?

   b. Consider the hashing function h(key) = (key * 7) % 11, where the table size is 11 and the keys are integers from 0 to 10. Is this a ***perfect* hashing function**?

   c. Consider the hashing function h(key) = key % 100, where the table size is 100 and the keys are even integers from 0 to 1000000. Is this a ***uniform* hashing function**?

   d. Consider the hashing function h(key) = (key * 7) % 49, where the table size is 49 and the keys are integers from 0 to 1000000. Is this a ***uniform* hashing function**?

   e. Consider the hashing function h(key) = floor(√key) % 100, where the table size is 100 and the keys are integers from 1 to 10000. Is this a ***uniform* hashing function**?

~~~ Lab Questions ~~~

1. [**AVL Tree Insertion - Submission**] Complete the _balance() method so that the AVL tree insertion is performed correctly. Note that all the rotation mechanisms have been implemented for you, i.e. you just need to use them properly. In addition, the prettyPrint() method is also given, so that it is easier for you to visualize the tree.

2. [**AVL Tree Deletion**] This is a one-line problem ☺. Implement the deletion function for AVL Tree. Take ideas from (1) above to implement the deletion.

3. [**Hash Table – Opening Addressing Simulation**] To reduce your coding time, we choose to "simulate" one aspect of the hash table instead of implementing a complete hash table. Write the following functions that returns the **probe sequence** as a Python list. For simplicity, we use the primary hash function h(key) = key % m, where m is the table size. You can assume **m is > 3**.

   a. Linear Probing: Given table size ***m*** and key ***k***, return the probe sequence that goes from [h(k), h(k)+1, …., m-1, 0, …. h(k)-1]. This function returns ***m*** probes.

```
def linearProbe( m, k ):
""" Return the linear probe sequence as a Python List."""
```

Example:

`linearProbe(11, 35)` → **[2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 1]**

`linearProbe(7, 35)` → **[0, 1, 2, 3, 4, 5, 6]**

b. Quadratic Probing: Given table size ***m*** and key ***k***, return the probe sequence that goes from [h(k), h(k)+1^2, h(k)+2^2, h(k)+3^2, …]. This function returns $\left\lfloor \frac{m}{2} \right\rfloor$ probes.

```
def quadraticProbe( m, k ):
""" Return the quadratic probe sequence as a Python
List."""
```

Example:

quadraticProbe(11, 35) ➔ **[2, 3, 6, 0, 7]**
quadraticProbe (7, 35) ➔ **[0, 1, 4]**

c. Double Hashing [**Submission]**: Given table size ***m***, key ***k*** and value ***p***, return the probe sequence that goes from [h(k), h(k)+ h_2(k), h(k)+ 2*h_2(k), h(k)+ 3*h_2(k), …], where h2(k) = (k % p) + 1. This function returns ***m*** probes.

```
def doubleHashing( m, k, p ):
""" Return the double hashing probe sequence as a Python
List."""
```

Example:

doubleHashing(11, 35, 4) ➔ **[2, 6, 10, 3, 7, 0, 4, 8, 1, 5, 9]**
doubleHashing(7, 35, 4) ➔ **[0, 4, 1, 5, 2, 6, 3]**

You can use the probe sequence functions to understand / verify some of the interesting theorem of hashing. e.g. you can see that the double hashing probes all distinct slots in "random" fashion if the ***m*** and ***p*** are chosen properly. Try out different value to check your understanding.

We will evaluate both correctness and programming style of your submitted code:

a. [Correctness 70%]: The code works according to the functional requirement, e.g. output value, output format, side effect, efficiency etc.

b. [Programming Style 30%]: Reuse own code whenever appropriate, modularity, good variable / method naming convention, comments (only if appropriate)