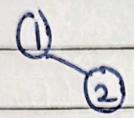
Algorithms HW4

Q1) Create AVL tree with 1 to 10 values

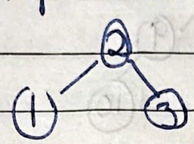
A1)

Insert 1

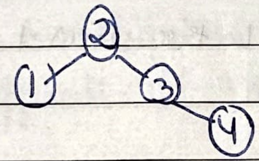
Insert 2



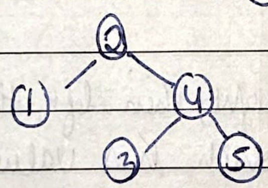
Insert 3 (Right-Right Rotation)



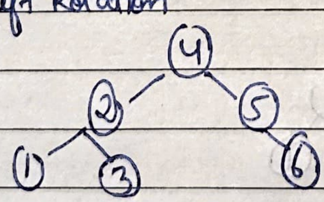
Insert 4



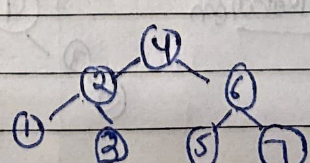
Insert 5 Left Rotation at 2



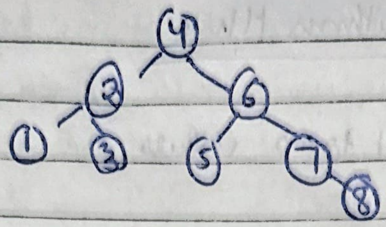
Insert 6 Left Rotation



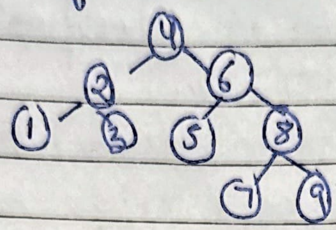
Insert 7 Left Rotation at 3



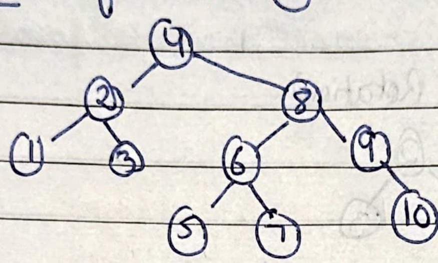
Insert 8



Insert 9 Left Rotation 5

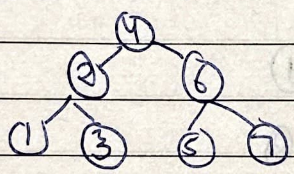


Insert 10 Left Rotation 6



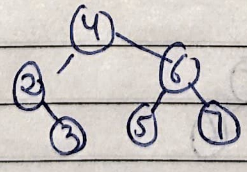
Q2 → Sketch AVL (1 to n)

A2 → Let's assume n=7, AVL tree has 1-7 values.

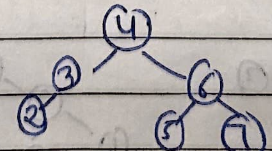


To acheive 3 rotations(right, then left, then left), we need to remove node which has value 1 (which is left of trees.

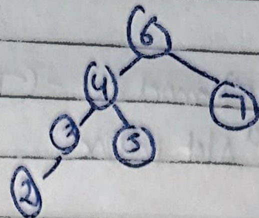
Remove 1



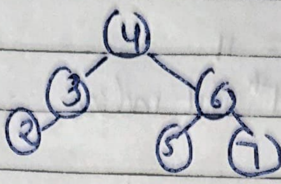
Right Rotation



Left Rotation



Left Rotation



Q3-> List all Coeffients

A3-> Hash function $h(x) = cx \bmod capacity$

Capacity = 24 ; Range = (0 to 23 inclusive)

For this hash function to return any index, 'c' must be coprime with 24 (as capacity is 24)

Common factors of 24 = 1, 2, 3, 4, 6, 8, 12, 24.

Not divisible by 2, 3 = 1, 5, 7, 11, 13, 17, 19, 23

If capacity = 0, then output would be 0.

So, coefficients = 1, 5, 7, 11, 13, 17, 19, 23

Q4-> Draw hash table with contents

A4-> Capacity = 20

Hash function $h(x) = 7x + 5 \bmod 20$. for values 1, 5, 11, 18, 3, 8

Insert 1

$h(1) = (7(1) + 5) \bmod 20 = 7 + 5 \bmod 20 = 12$

Insert 1 into 12th slot index

Insert 5

$h(5) = (7(5) + 5) \bmod 20 = 40 \bmod 20 = 0$

Insert 5 into 0th slot index

Insert 11

$h(11) = (7(11)+5) \bmod 20 = (77+5) \bmod 20 = 82 \bmod 20 = 2$

Insert 11 into 2^{nd} slot index

Insert 18

$h(18) = (7(18)+5) \bmod 20 = 11$

Insert 18 into 11^{th} slot index

Insert 3

$h(3) = (7(3)+5) \bmod 20 = 6$

Insert 3 into 6^{th} index

Insert 8

$h(8) = (7(8)+5) \bmod 20 = 1$

Insert 8 into 1^{st} index

So, hash table,

Index	Value
0	5
1	8
2	11
3	
4	
5	
6	3
7	
8	
9	
10	
11	18
12	1
13	

Index	Value
14	
15	
16	
17	
18	
19	

Q5→
A5→ To achieve efficient Set ADT, we can use hybrid data structure such as AVL tree which is mixture of balanced binary search-tree and hash table.

a) Data Storage in Memory:- We can use hash table to store values in different buckets with separate chaining for collision resolve. It will store reference to nodes in BST. We will use AVL tree to store elements so that in worst case complexity ($O \log(n)$), we can search elements efficiently.

b) Element Searching:- First, hash element to find corresponding bucket with time complexity of $O(1)$. If bucket is empty, element is not present. If it's not empty, we follow pointer to next node in balanced tree which complexity of ($O \log(n)$). for worst case.

c) Insert new elements:- First, find hash value of element to find its bucket which takes linear $O(1)$ time complexity. If bucket is empty, create new node in BST. If it already has bucket refrence, insert it there which takes $O(\log(n))$. maintaining its balance.
If rehashing requires, hash table grows and elements are redistributed to new nodes and remains balanced.