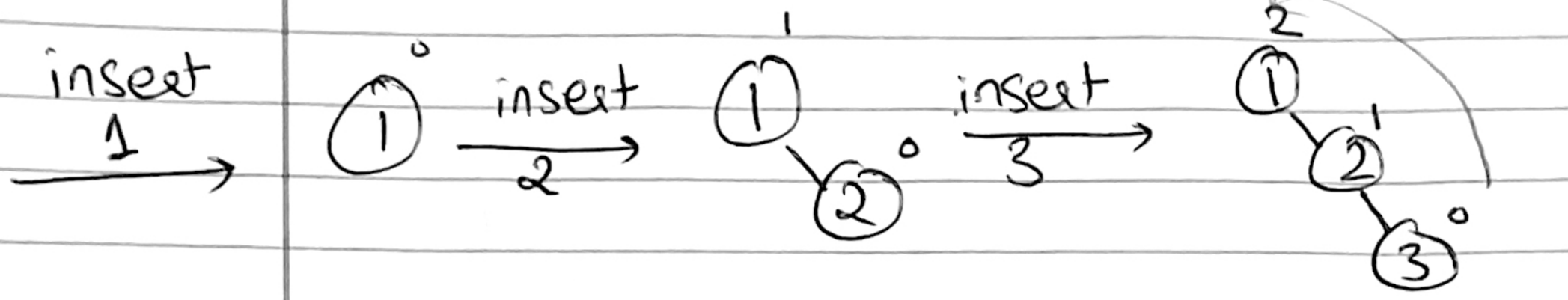
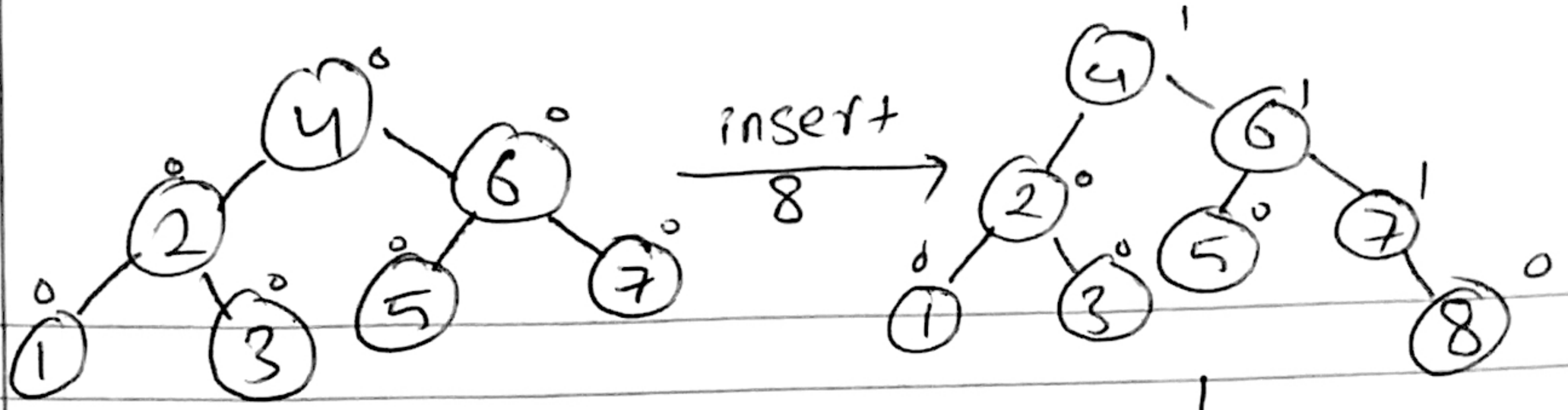
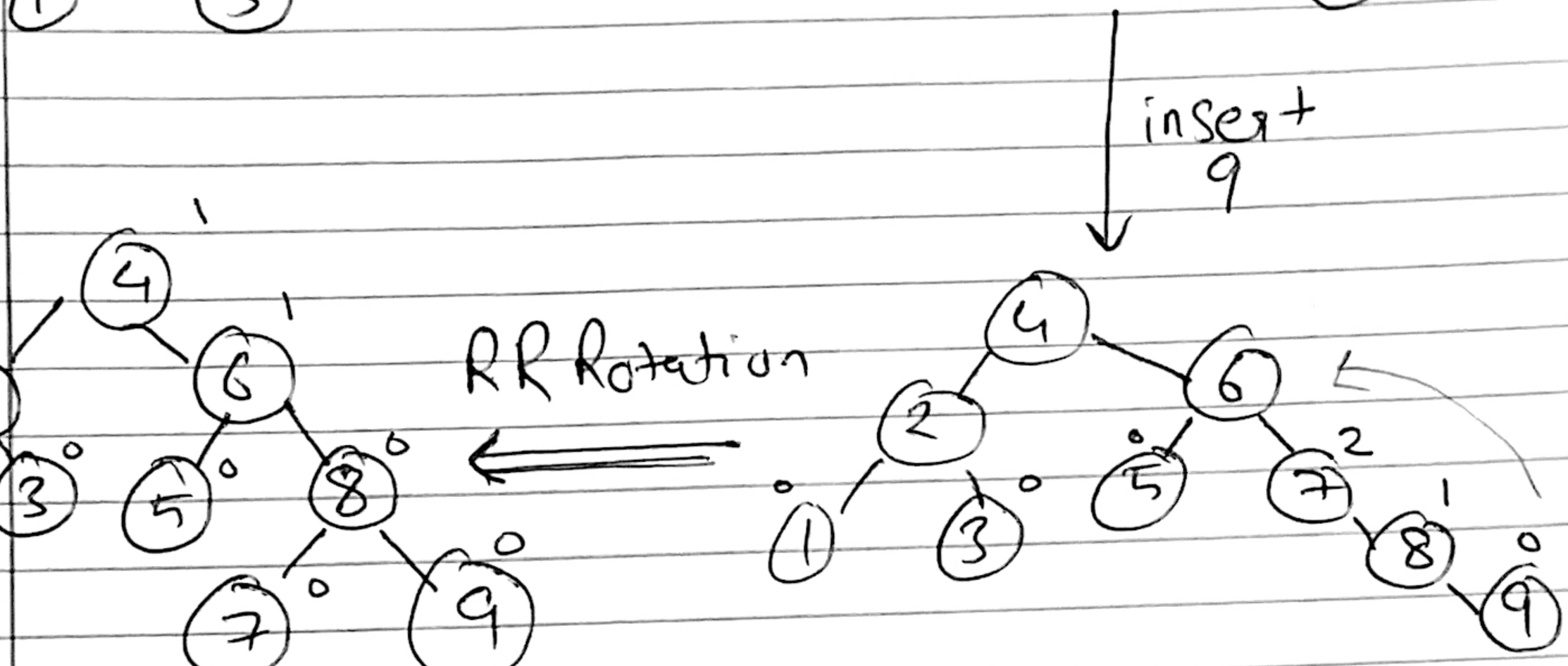
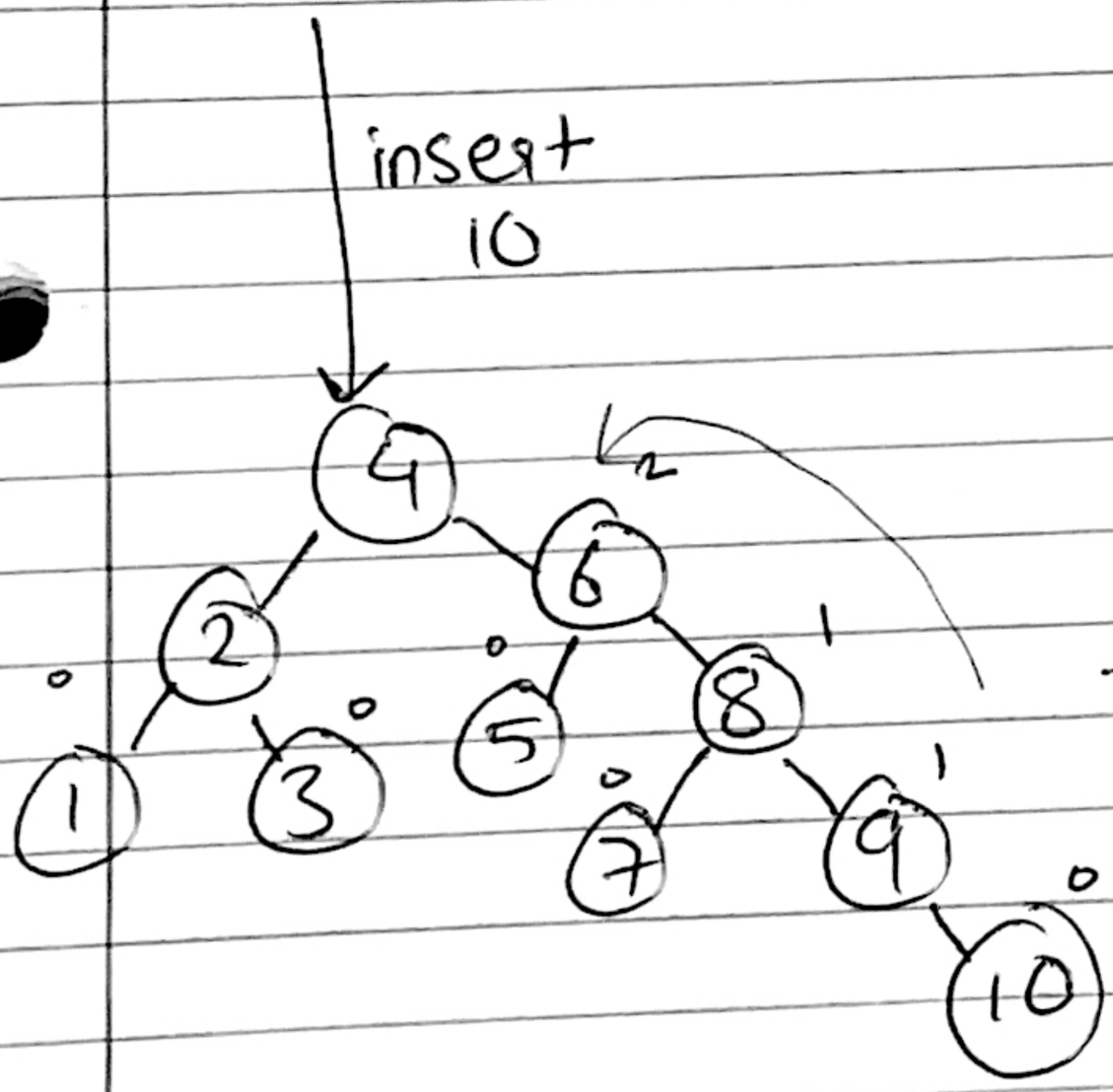
(1) KEYS: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

AVL TREE



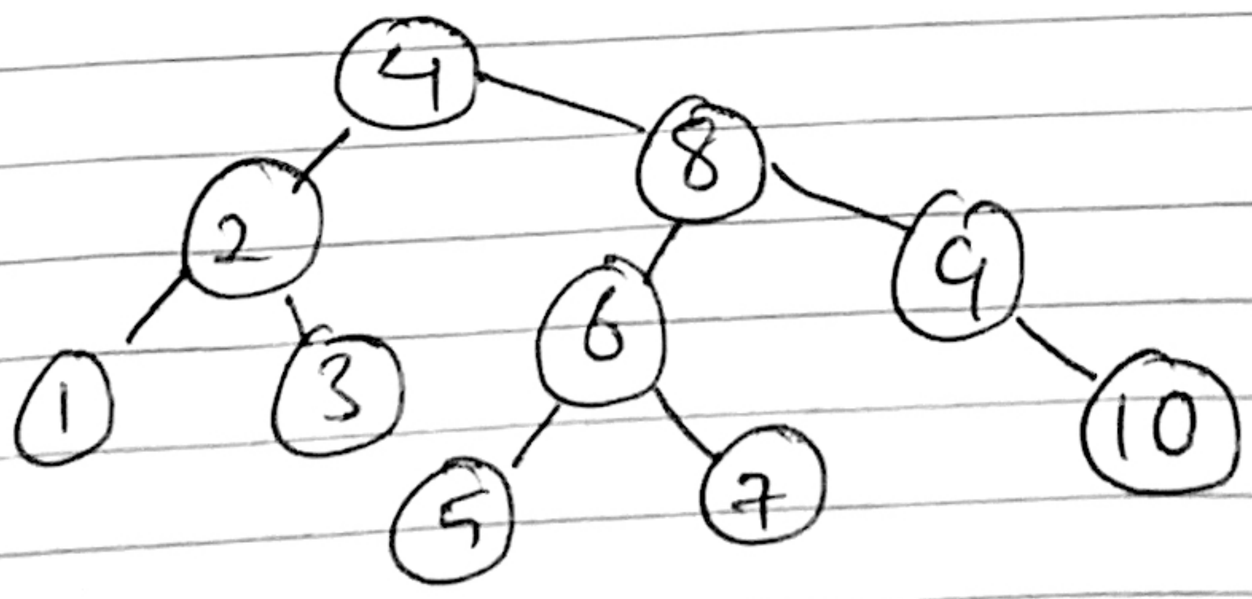




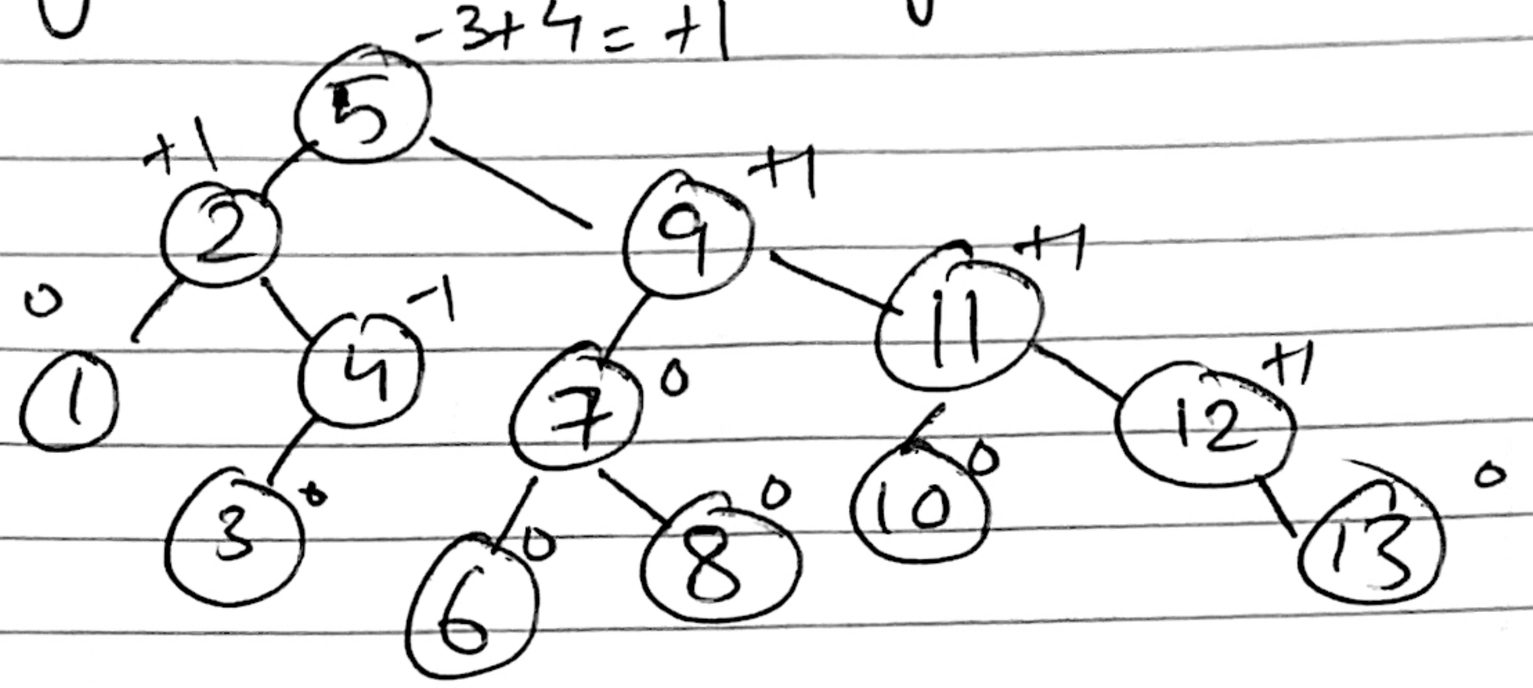


∴ BALANCED AVL TREE

→



(2) Taking an AVL TREE of 1-13



REMOVING ONLY 1 NODE WHICH RESULTS

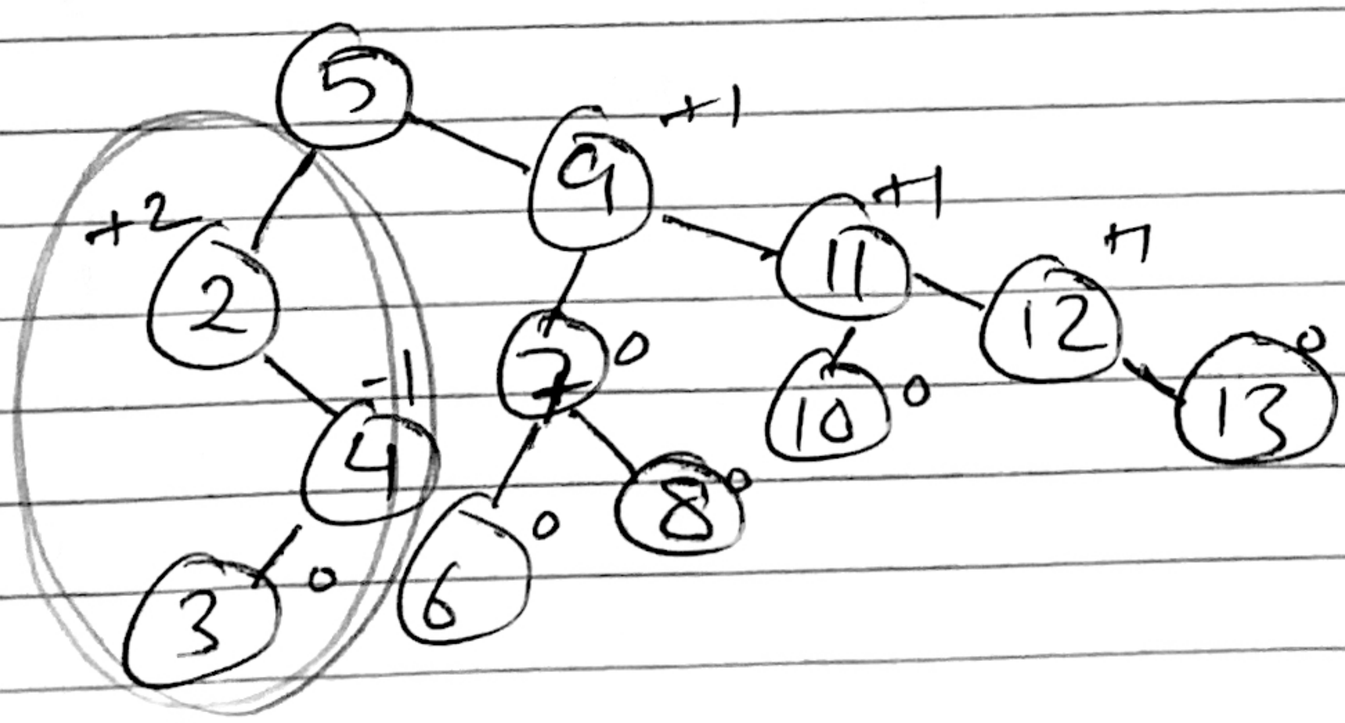
IN → RIGHT, LEFT & LEFT ROTATIONS

∴ A RL Rotation & then RR Rotation

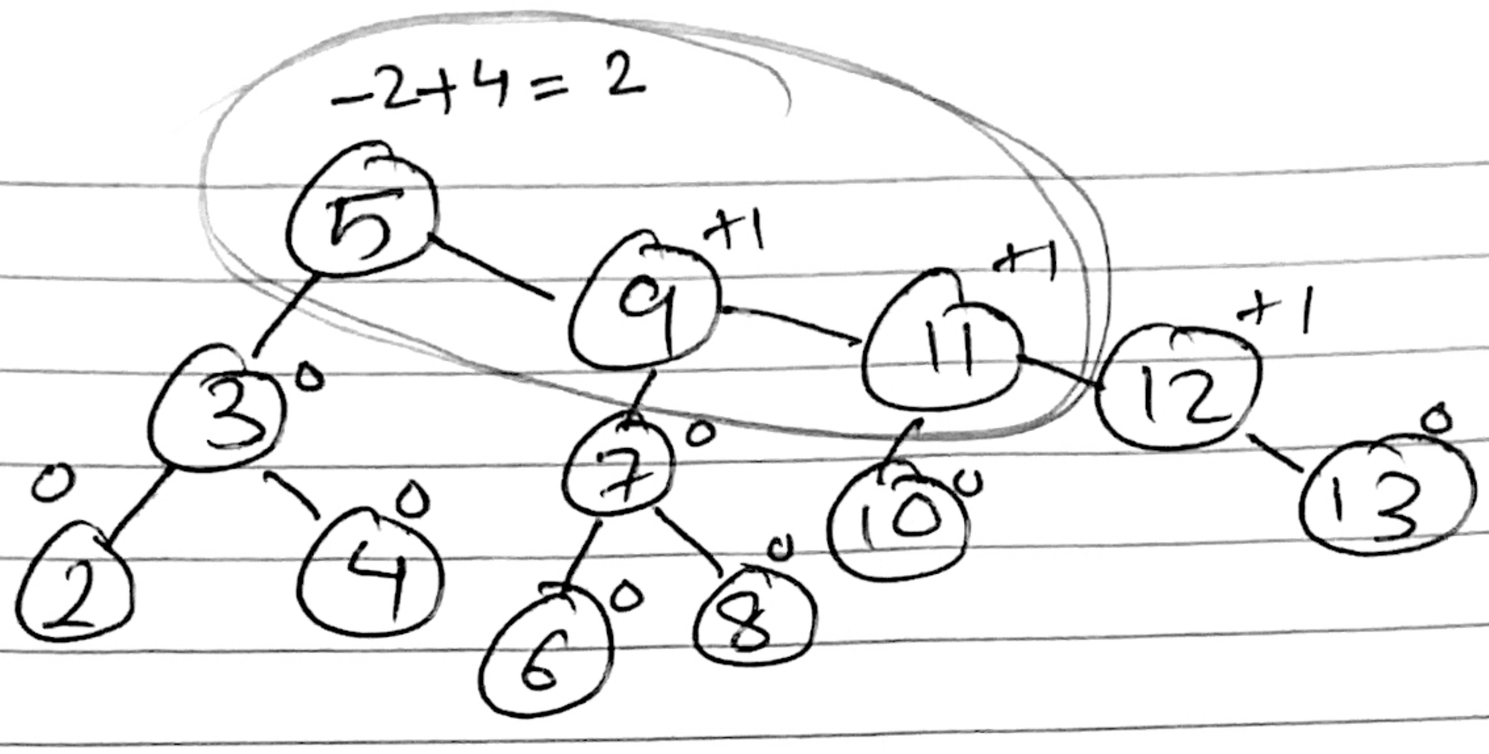
OR

A LL Rotation then RR Rotation (TWICE)

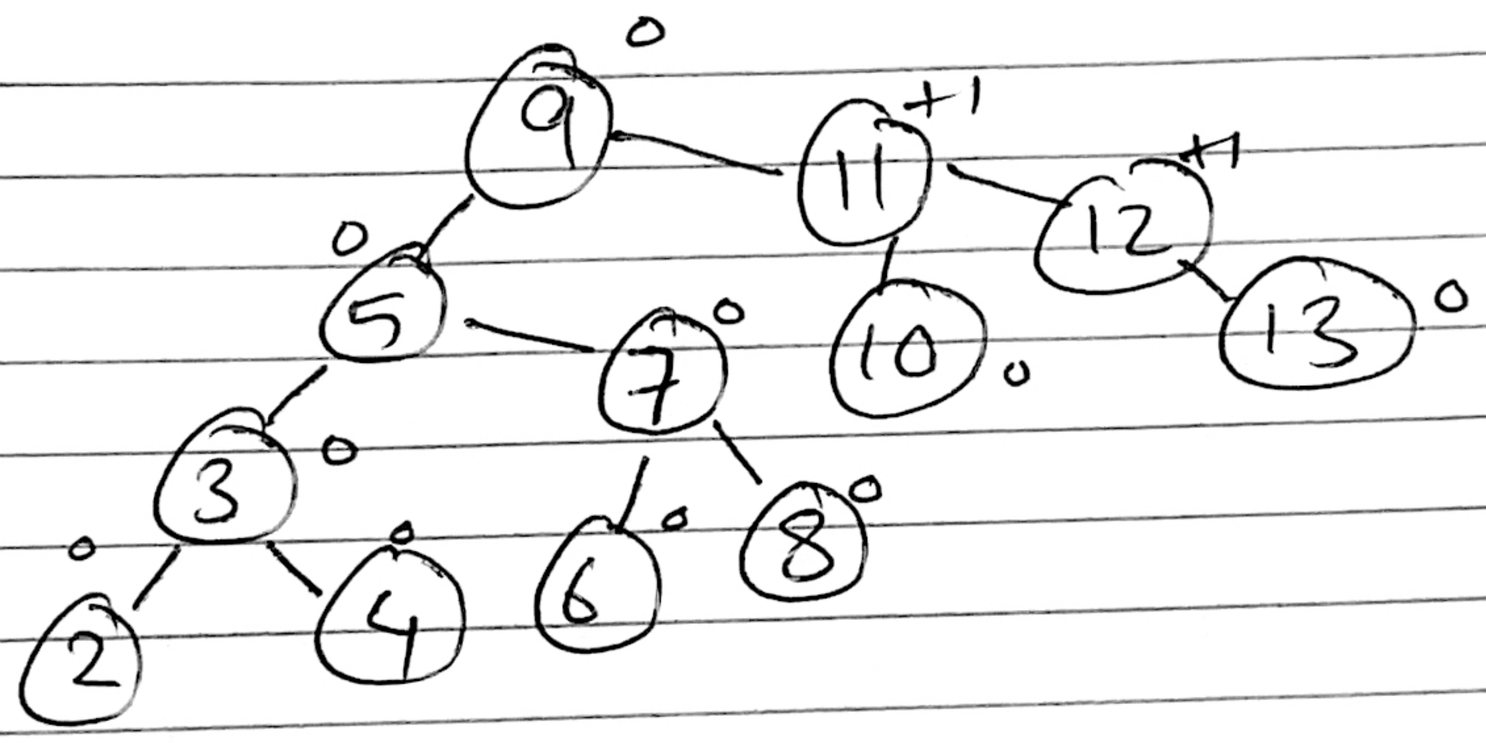
⇒ In the above tree if we remove 1



⇒ This Results in a RL Rotation.



∴ A RR Rotation at Nodes: 5, 9, 11



Hence, Removing Node 1 will result in exact 3 Rotations as shown above.

NODE 1 is REMOVED

Answer 3;

The hash function is h(x)=c x mod 24, and we need to find all values of c such that h(x) can return any index in the range [0, 23].

To ensure that all indices are possible, c must be **coprime** with the capacity (24). The prime factors of 24 are 2 and 3, so c must not be divisible by 2 or 3.

Example:

Let's take an example where c=6 and the capacity is 24. The GCD(6, 24) = 6.

Now consider the hash function h(x)=(6 × x)mod 24

Let's calculate h(x) for a few values of x:

- h(0)=(6×0)mod 24=0
- h(0)=(6×1)mod 24=6
- h(0)=(6×2)mod 24=12
- h(0)=(6×3)mod 24=18
- h(0)=(6×4)mod 24=0

This pattern of 0, 6, 12 , 18 will keep on repeating. As a result it will never return indices like 1, 2, 3, 4, 5, 7, 8, etc.

Coefficients c such that GCD(c, 24) = 1:

The numbers in the range 0 to 23 that are **coprime** with 24 are:

c=1,5,7,11,13,17,19,23

Hence, the list of numbers is **1,5,7,11,13,17,19,23.**

Answer 4:

HASH FUNCTION: h(x) = 7x+5 mod 20

CAPACITY = 20

KEYS: 1, 5, 11, 18, 3, 8

EMPTY HASH TABLE

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

INDEX

<u>FINDING INDEX OF EACH KEY</u>

KEY: 1

h(x) = 7(1)+5 mod 20 = 12 mod 20 = 12

UPDATING HASH TABLE

												1							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

KEY: 5

h(x) = 7(5)+5 mod 20 = 40 mod 20 = 0

UPDATING HASH TABLE

5												1							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

KEY: 11

h(x) = 7(11)+5 mod 20 = 82 mod 20 = 2

UPDATING HASH TABLE

5		11										1							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

KEY: 18

h(x) = 7(18)+5 mod 20 = 131 mod 20 = 11

UPDATING HASH TABLE

5		11									18	1							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

KEY: 3

h(x) = 7(3)+5 mod 20 = 26 mod 20 = 6

UPDATING HASH TABLE

5		11				3					18	1							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

KEY: 8

h(x) = 7(8)+5 mod 20 = 61 mod 20 = 1

UPDATING HASH TABLE

5	8	11				3					18	1							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

FINAL HASH TABLE

5	**8**	**11**				**3**					**18**	**1**							
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**

INDEX

ANSWER 5

Set ADT Combining Hash Table and Balanced Tree

To combine the advantages of a **hash table** (for fast lookups) and a **balanced binary search tree** (for sorted data), we can design a hybrid structure:

1. **Memory Storage**: Store data in a hash table where each entry (bucket) contains a balanced binary search tree (eg. AVL tree). Each bucket corresponds to a hashed index.
2. **Search Function**:
    - **Hash the key to find the appropriate bucket in the hash table in O(1) expected time.**
    - **In the bucket, search the AVL tree for the key in O(log n) worst-case time complexiity.**
3. **Insertion**:
    - **Hash the key to find the correct bucket in O(1) expected time.**
    - **Insert the element into the AVL tree stored in the bucket in O(log n) worst case time.**
    - **If the AVL tree becomes unbalanced after the insertion, perform rotations to restore balance (ensuring O(log n) insertion time).**

This structure benefits from the average O(1) complexity of hash tables for lookups, but also uses AVL trees to handle collisions, allowing efficient searches even when multiple elements are hashed to the same bucket.