1) Let's consider the probability of no collision. That would be ${}^{n}P_{k}$ (no. of arrangements of k elements without collision) divided by n^{k} (total number of arrangements).

So,

probability of a collision =

$= 1 - \text{probability of no collision}$

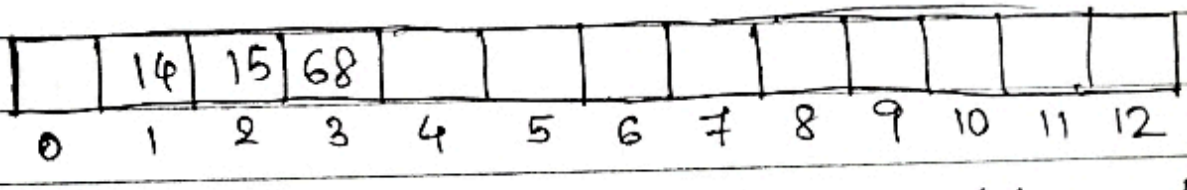
$= 1 - \left(\frac{{}^{n}P_{k}}{n^{k}}\right)$

2) Linear Hashing Execution:

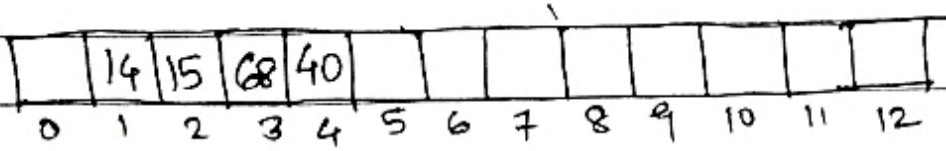
Let's convert the elements that we need to insert in the first step into mod13 form:

1, 2, 3, 1, 5, 1, 2, 4, 4, 8, 2.

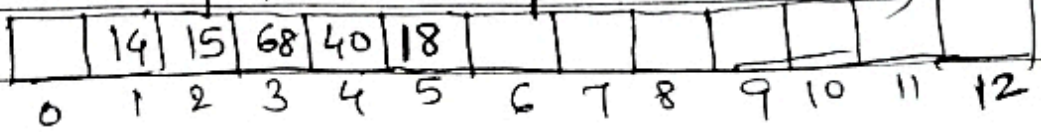
→ Insertion of first 3 elements is simple. The elements 14, 15, 68 will go to indices 1, 2 & 3 respectively.



Now, the next element 40 should go to index 1 but since that place is occupied, we insert it in the next empty place (index 4)



Inserting 18 is simple (index 5).



66 → should go to index 1 but that's occupied So it goes to the next empty index 6.

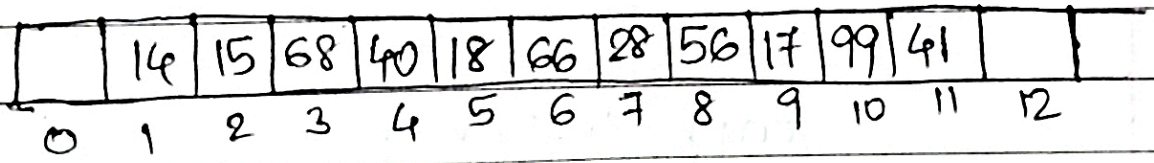
28 → index 2 is occupied so it goes to index 7.

56 → index 4 is occupied. so it goes to next empty index 8.

17 → index 4 is occupied. So it goes to index 9.

99 → index 8 is occupied. So next empty index is 10.

41 → ~~Position~~ index 2 is occupied. So this goes to index 11.



Step 2: Delete 40, 56, 17, 41

→ We check index 1 for deleting 40. Since we can't find it there, we start parsing the array from index 1 and find 40 at index 4. Set index 4 to $.

→ We check index 4 for deleting 56 but don't find it there. We parse the array from index 4 onwards till we find an empty index in search of 56. If we find an empty index, that means 56 isn't present in the array & deleting 56 command is invalid. If we find 56, we set the value at that index to '$'.

Note that the index having value '$' isn't considered an empty index.

We similarly perform deletion for other values as well and this is the resulting array:

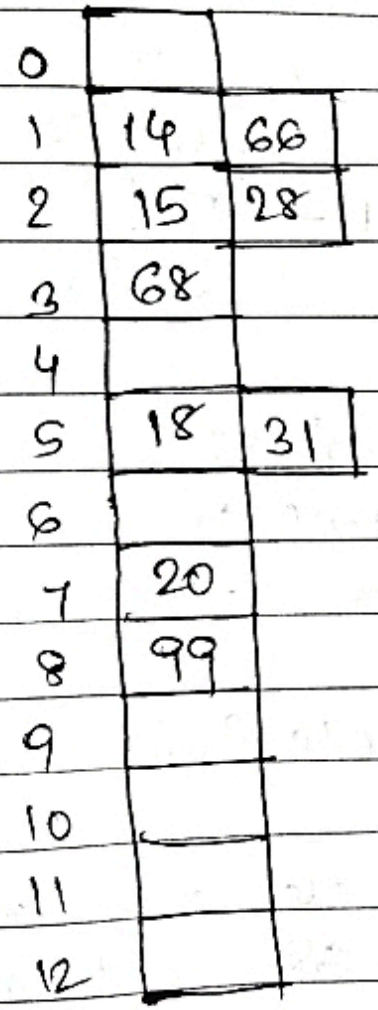
	14	15	68	$	18	66	28	$	$	99	$			$
0	1	2	3	4	5	6	7	8	9	10	11	12		

Step 3: Insert 20, 31

→ 20 should go to index 7 but that is occupied. So, we find the next empty index or index with value '$' and insert 20 there. Similarly we insert 31.

	14	15	68	$	18	66	28	20	31	99	$	
0	1	2	3	4	5	6	7	8	9	10	11	12

Chaining Execution is pretty simple. ~~The results~~ After all the inserts & deletes the resulting data structure will look like this:



3) We make 2 hash maps for this question, with each of them using chaining.

→ In the first one, we use name as the key, and store nodes containing name and address at appropriate linked lists.

→ In the second hash map, we used address as the key, and store nodes containing name and address at appropriate linked lists.

→ In nodes of the first hash map, we store pointers to the corresponding nodes of the second hash map, and vice versa (so every node has a pointer to its corresponding node in the other map.).

- Insertion :-
Create two nodes, insert them in the two hash maps. Both insertions to the two hash maps take O(1) on an average. Hence, insertion takes overall O(1) average time.

- Inspection & counting

If we are given a name to search, the search for it in the first hash map.
If we are given an address the search in the second hash map.

- **Deletion :-**

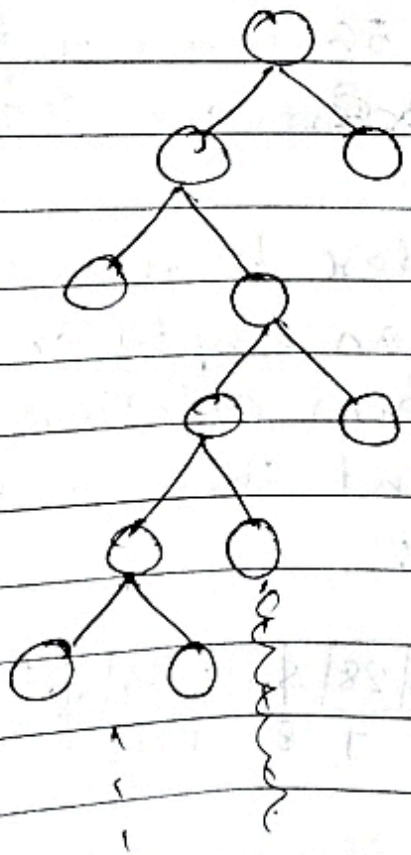
Search in the appropriate hash map according to the input (name or address), and then delete that node along with the node in the other hash map that it is pointing to. Since searching is O(1) on an average in a hash map, & node deletion in a ~~Sorted~~ linked list takes constant time, this would also take overall O(1) on an average.

4) For maximum height, the root node will have 2 children but after that at every level either both the nodes will have 0 children or only one of them has 2 children.

→ For such a binary tree the total no. of nodes can never be even. So the maximum height would be (considering the root node at height 1)

$$\left(\frac{n+1}{2}\right).$$

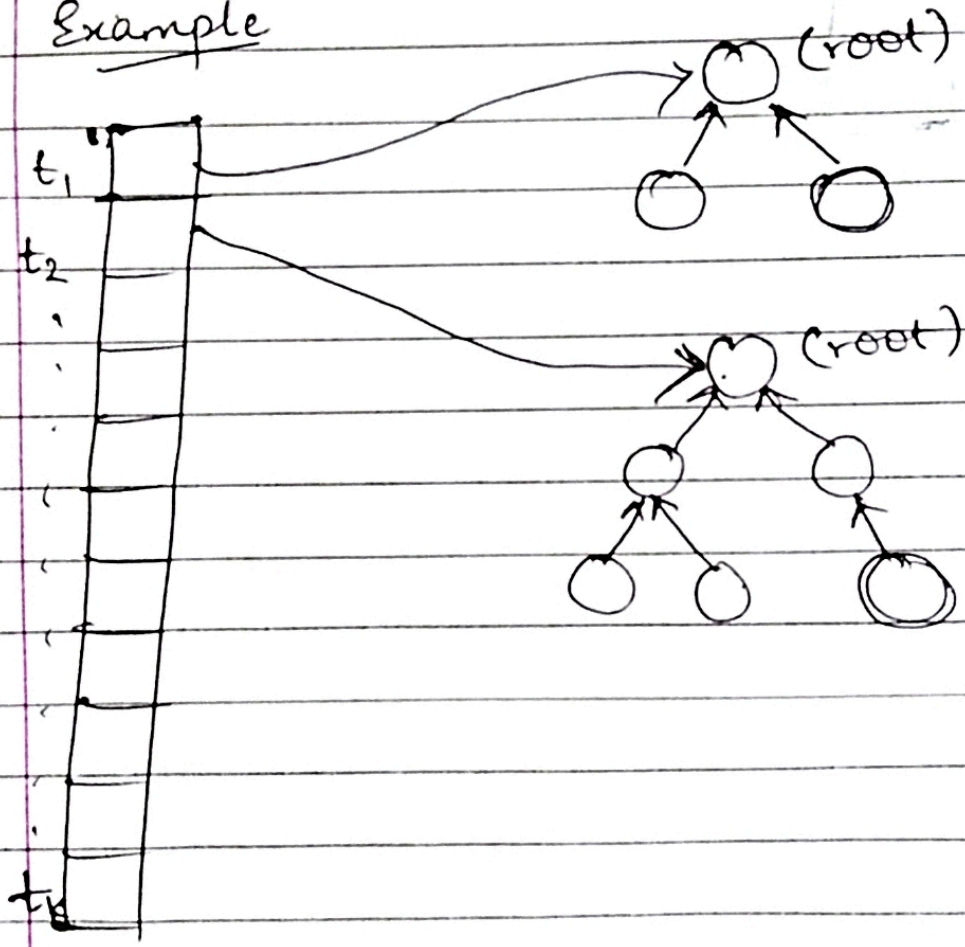
Tree would look something like this



5) Let's have an array of pointers for all the teams such that each element in that array points to the root of the tree of that team.

→ Given a tree structure of the team, store the structure with reversed edges.
(for optimisation), i.e., for each node of the tree store a pointer to it's parent node.

Example

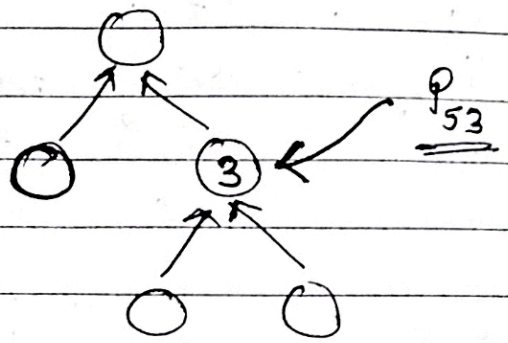


→ Now, store an array of list for all the employees such that each element in the array is a list of pair(i, pointer) such that i is the team number and pointer points to that employee in the tree structure of team i.

e_1	$(1, P_{11})$	$(6, P_{61})$
e_2	$(3, P_{32})$	
e_3	$(1, P_{13})$	$(5, P_{53})$ $(4, P_{43})$
:		
e_n		

P_{53} points to employee 3 in team 5.

Suppose team 5 is



→ This array will have lists with max size 3 as each employee is in atmost 3 teams.

→ For storing this array of all the employees, we'll have to parse all the trees once.

→ Now, we can easily check if employee A has the same boss B in 2 different ways.

→ Go to all the pointers pointing to employee A in the array of employees & check if B is an ancestor of A atleast in 2 different ways.

(Reversing the edges optimises this lookup).

There can be maximum 3 pointers to A. Hence, this look up takes $O(h)$ time. where h is max height out of all the given tree structures.

⇒ For the second part of the question, 2 teams can meet at the same time if there are no common employees who are part of both the teams.

⇒ Finding the minimum no. of distinct hours required for the whole company to meet becomes similar to graph coloring problem (which you might have studied in Discrete Structures). Graph coloring problem is an NP hard problem & hence we'll try to find an approximation to this no. here.

→ Create an undirected graph s.t. the vertices of the graph represents the team and an edge between any two vertices represents that there are common employees between those 2 teams.

→ Such a graph can be created by simply parsing the array of employees once.

Example :- if A is in team 1, 3 & 5. There vertices 1, 3 & 5 are fully connected with each other.

For finding the no. of distinct hours, we can ~~color~~ follow the basic greedy coloring algorithm:

(i) Color the first vertex with first color.

(ii) Do the following for remaining V-1 vertices:

⇒ Consider the currently picked vertex & color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all the previously used colors appear on vertices adjacent to v, assign a new color to it.

We can revisit this problem once graphs are covered.