Hash Table Data Structure

We have an array of unsorted elements and you are searching for an element. For that we have to go through the entire array looking for that element. If we are lucky, then we will find it in the beginning of an array or else in any random position. If the element you are searching for does not exist you have to search every index of the array from the beginning till the end. Therefore the time complexity of searching is O(n). What about inserting and deleting an item from the array? In order to do these operations we first need to search for that particular item and then add or delete. As mentioned earlier searching takes O(n) and insertion and deletion takes O(1). Therefore the whole process of insertion and deletion takes O(n) + O(1), which is O(n).

What if we you use a sorted array? Then you can apply binary search for searching an element. Binary search takes O(lgn), faster than n. What about insertion and deletion? Now as the array is sorted you can't just insert anywhere. You have to maintain the order. Therefore after an insertion you have shift elements to the right. In the same way for deletion you have to shift the elements to the left. Therefore in both insertion and deletion you have to search first and then carry out the desired operation. Searching takes O(lgn) and insert/delete takes O(n) due to shifting. The total time required is O(lgn) + O(n), which is O(n).

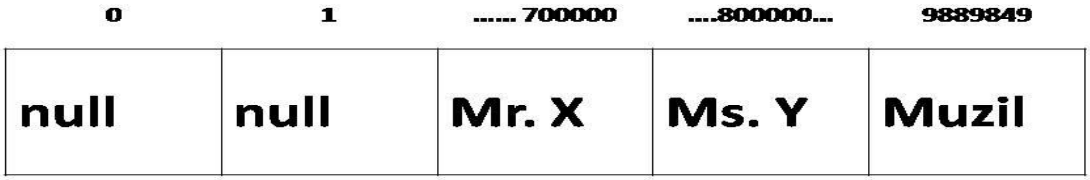
We have discussed about arrays. What about linked list? For an unsorted list insertion at head requires O(1). Insertion at the end, if we keep a tail reference requires O(1) else O(n). Searching takes O(n) and deletion takes O(n). If the list is sorted all of the operations take O(n).

So far I have discussed about performance, what about space ? Suppose we need to store the username against phone numbers. That is, if a user enters a phone number he can see the corresponding user name. There are 2 ways of doing that.

1) keep 2 arrays numbers[] and names[] which will contain the number and the corresponding name respectively. When a user enters a number, that number will be searched from the numbers array. If found, take the index of the array where the number was found and then retrieve the name using that index from the names array. The following picture will give you a clear idea. The first array is the numbers array and the second one is the names array. The user searched by the number 7100090. It was found in the index 2 and its corresponding name was found in the same index of the names array.

We needed 2 arrays of size n.

2) The second technique would be using one array with indices as the phone number. Look at the following diagram.



Can you imagine how big the array should be and how many spaces will be wasted !!!

Hash table: In order to do the operations discussed above (search, insert and delete**) fairly fast** and using a **moderate** space, a new but very simple data structure has been developed called the hash table. Hash table is nothing but a normal array but used in a different way. In the previous example which dealt with phone numbers can be solved using one smaller array using hash table. This is how it works.

Let the name of the array be names of size 10 and let the phone number be 9889849. We are going to generate a **key** out of this phone number, which will be the **index** of the array. Let the key generated be **5**. [let us not worry about how we got 5. I will discuss it later]. In the names array, we will insert the corresponding name of the phone number. **names[key] = "Muzil".**

See. Hash table solved the problem using **one small array** and the time of insertion is O(1), because we did not have to do any shifting. If we want to delete the name of the same number holder it will also take O(1). Why ?? This is because we do not have to search the whole array. We know the key, which is 5, and we

can go directly to that location and nullify it. **names[key] = null**. In the same way search also takes O(1). How ?? Evaluate this piece of code.

```
if (names[key]==null){
        return false;
}else{
        return true;
}
```

Therefore it can be said that the hash table technique is just using a regular array with **an addition thing**, that is, the key must be generated.

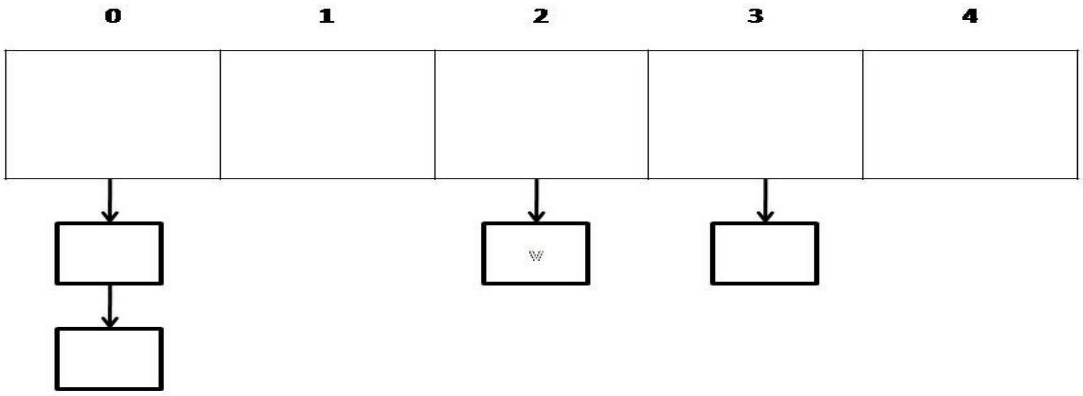
Hashing: Let us now discuss how the key was generated. The process of generating the key is known as hashing or hash function [a method that generates the key]. The key can be generated in many ways. The general convention is:
for numbers, we add them up and mod by the array length. 9889849 = 9+8+8+9+8+4+9 = 55
(55 mod names.length) = 5;
for strings, sum up the ascii values of the characters and mod by the array length.
ABC = 65+66+67 = 198
(198 mod 10) = 8;

Please note there is no particular rule for a hash function. Different organizations can have their own hash functions depending on their needs.
Hash functions are chosen very wisely. Why? Because hash table also got issues.

Issues: We have seen the number 9889849 has been inserted in the position 5. Suppose there is another number 9898948 with the name Khan and we need to insert it. Same way let us use the hash function used before to deduce the position. 9+8+9+8+9+4+8 = 55. 55 mod names.length = 5. We find the index to be 5 and that position is already occupied. What to do? Should we replace the old one? Of course not. There are ways to solve this **collision**.

Solution to collisions: There are 2 collisions. 1) forward chaining 2) linear probing

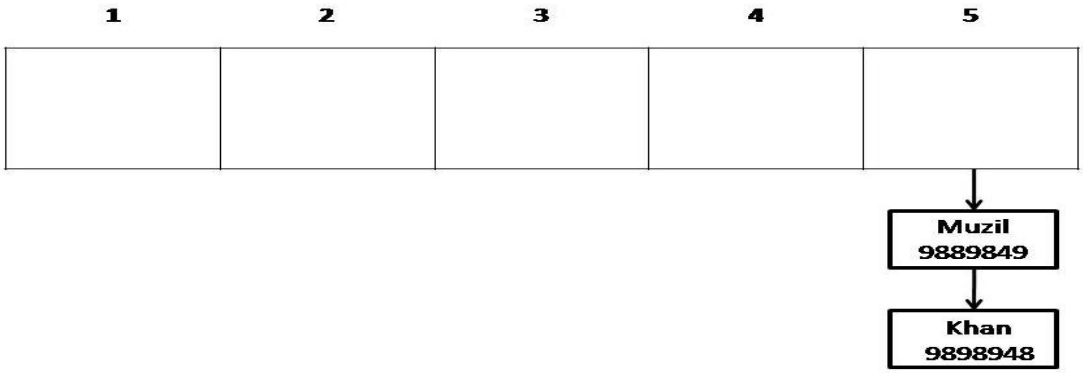
1) Forward Chaining: In this method the hash table (the array) is modified to an array of linked list. That is, each position of the array will have a linked list.



The nodes of the linked list will have the content. Look at the diagram and notice at position 0. There are 2 nodes which means one index of the array has 2 items (collision). This is one way of handling collision. There is a collision at index number 0 but the items are stored in a linked list. Now if we need to add more item in index 0, we can easily do that without any data loss.

Now let us analyze the time complexity of our new hash table. As linked list has been added, we know that the search and delete both has a O(n) for linked list. Insert is O(1).

Let us go back to our previous phone example. Let the hash table be initially empty. The first input arrived is 9889849 -> Muzil. Using the hash function we have deduced the index and that is 5. First we are creating a node with the name Muzil and the number and then adding the node in the array inside position 5. The second input is 9898948 -> Khan. The hash function gives 5 again. In position 5 the new node with name Khan and number is added with the existing node. We can add in head or at the end of the linked list keeping a tail reference. Inserting a new node at the beginning or at the end takes O(1).



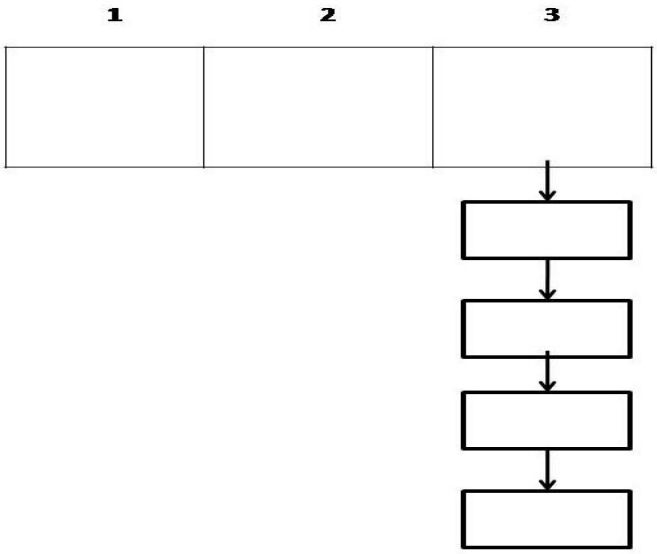
This is what the hash table looks like after insertion. We have 2 items in the table and therefore n = 2. Now let's see how searching is done. A user wants to retrieve the name of the number 9898948. Therefore using the same hash function the index is calculated which is 5. Now in index 5, the linked the contains 2 nodes.

Search all the nodes until the number matches. Then return the corresponding name.

```
search (int num){
        int k = hash (num);
        Node h = names[k];
        while (h!=null){
                if (h.number == numnber){
                        return h.name;
                }
                h=h.next;
        }
}
```

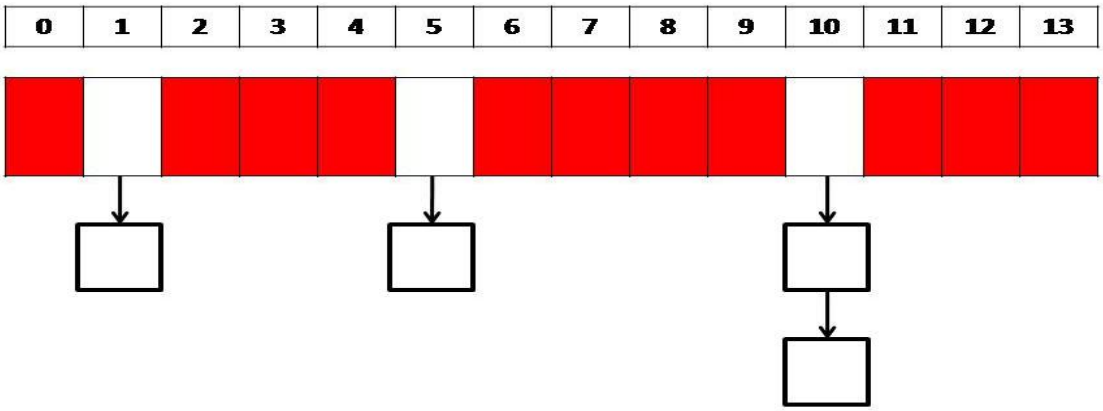
How many times will the loop run ? n times and hence the search is O(n). Same reason and technique for delete. Therefore delete is also O(n).

Now the question is how big should be the array ? If the array is too small compared to the number of input, then there will be too many collisions, hence forward chaining will look like the picture below.



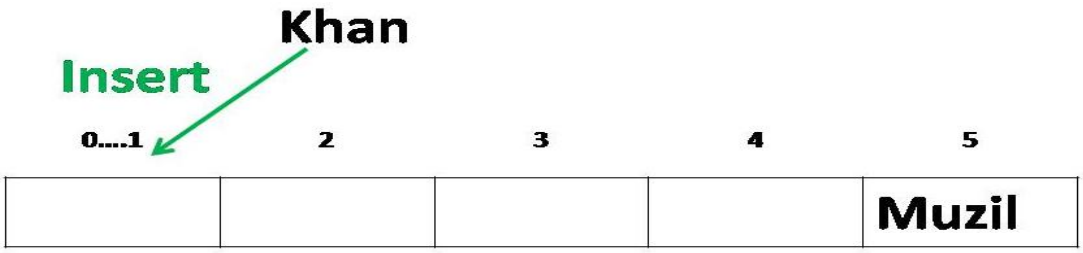
The chain will be too long. There if the input size, n = 1000, the time complexity of the operations will be 1000.

What if the array is too big ? Too many spaces will be wasted. The diagram below shows the unused space marked in red.



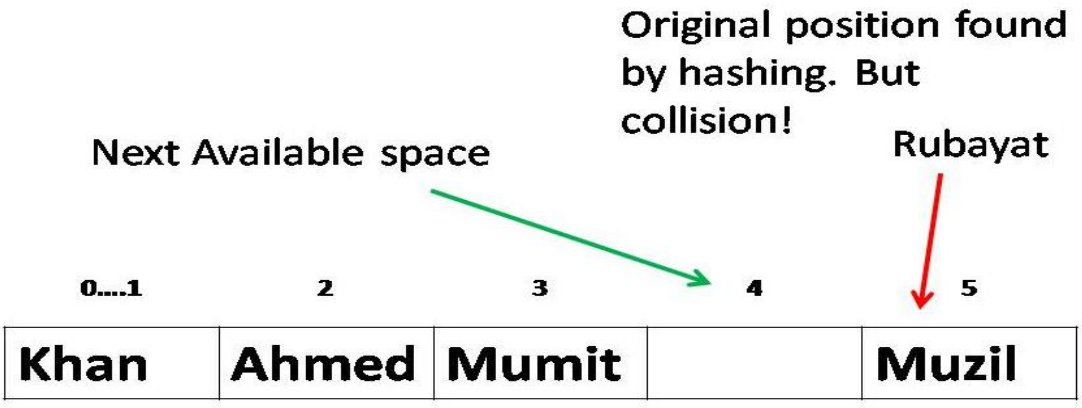
2) **Linear Probing:** This method avoids the burden of linked list. It will use a normal circular array to handle collision. Let us go back to our same old phone example 9889849 -> Muzil and 9898948 -> Khan and we are using the same array. Suppose we have inserted the first input in the position 5 and while inserting the second input we have encountered a collision. What linear probing does is, it looks for the next available space and inserts the item. Look the diagram below for a better understanding.

Next available position after 5 ?????
As it is a circular array,
position = (position+1) % array.length;
= (5+1)%6
= 0



What if we have a new item, Mumit whose key is 3[found using the hash function] ? Then we will first check if that position is empty or not. If yes insert or else find the next empty space and insert.

This is how insertion is done in linear probing. Let us analyze its complexity. If the position is empty then insertion takes O(1) [best case]. What if the situation is such that the next available space is the last space of the array ?



In such situation we have to run a loop to the end of the array. If the size of the array is n then this would take n iterations to find the empty space in the worst

case. Therefore we can say that in the worst case insertion takes O(n). The pseudocode of insertion should be something like this:

```
insert (element, key , array){
        if (array[key]=empty){
                array[key]=element;
        }else{
                i = key;
                while (array[i] not empty){
                        i++;
                }
                array[i]=element;
        }
}
```

Now what if the array is filled up and there is a new element to insert ? In that case I need a new array but copying elements from the old to the new one will not help in fact it will give you a wrong answer. What you have to do is for each element in the old array, re hash each element, find the new position and then insert into the new array.

What about the other operations delete and search ? I believe you are smart enough to figure how to carry them out and compute the time complexity.

Summary: The array of the hash table and the hash function must be chosen in a way such that the number of collisions is minimum. There is no way to avoid collisions but choosing the hash table and the function smartly can reduce collisions significantly. Both the collision avoidance techniques, forward chaining and linear probing, has advantages and drawbacks.