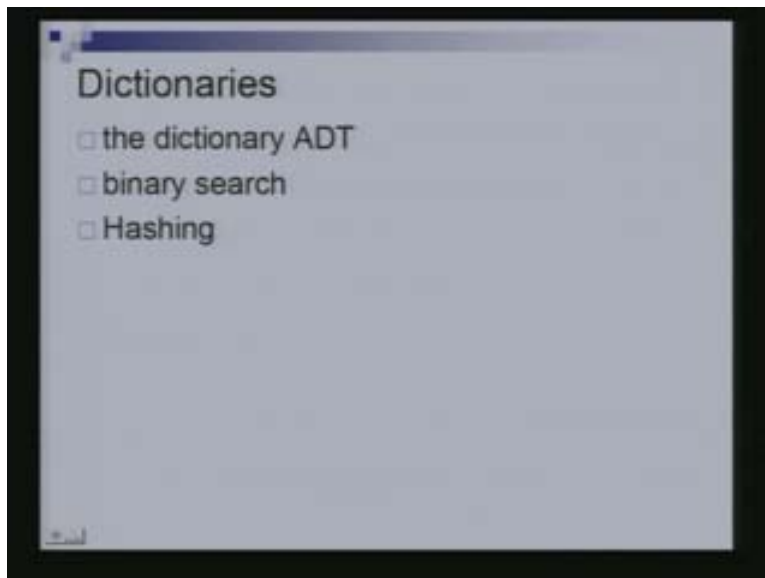


Data Structures and Algorithms
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture – 4
Dictionaries

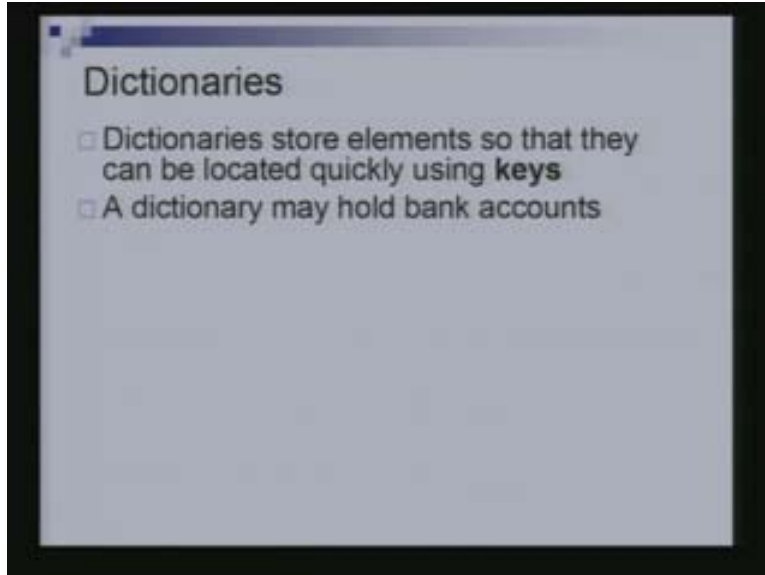
We are going to look at the dictionary abstract data type. We are also going to see how binary search is done, what are all the analysis for binary search and then we will go on to hashing, hash table, how hashing is done, see the collision resolution techniques and then in the next class we will follow up with more hashing techniques.

(Refer Slide Time: 1:17)



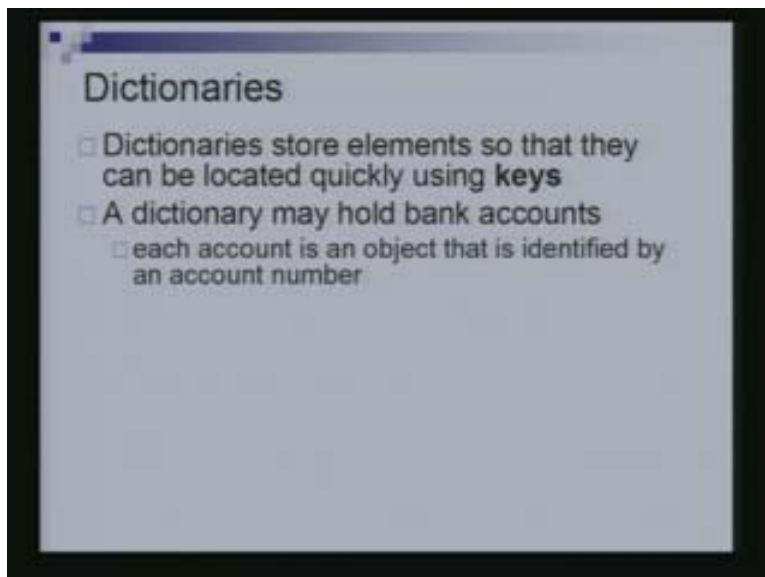
A dictionary is an abstract data type that stores elements which can be located very quickly. One example that we can have for a dictionary is to store bank accounts.

(Refer Slide Time: 1:37)



What is the notion of the key, when you store bank accounts? The notion of a key is your account number, bank account has lots of information associated with it, but you are going to access the bank account or data associated with that account by using the account numbers. Thus the account number becomes the key.

(Refer Slide Time: 1:53)

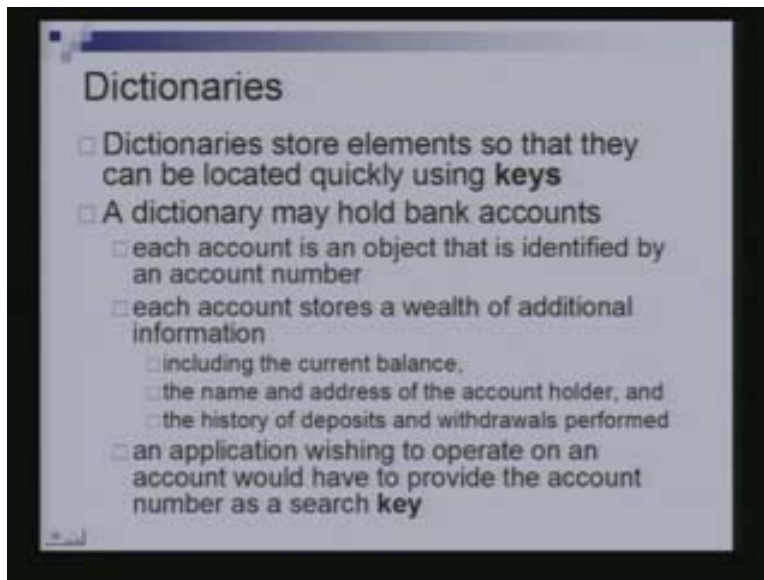


As I said account stores wealth of information, it could be your current balance, it could be the name and address of the account holder and it could be the list of transactions done in the last few days and so on.

When you have to access any of the above information you need the notion of the key. A dictionary is something that stores the elements. When we talk of an element we will mean all of the above additional information which is also given in the slide below. When we talk of key, the key would be the account number that helps us to access the particular information.

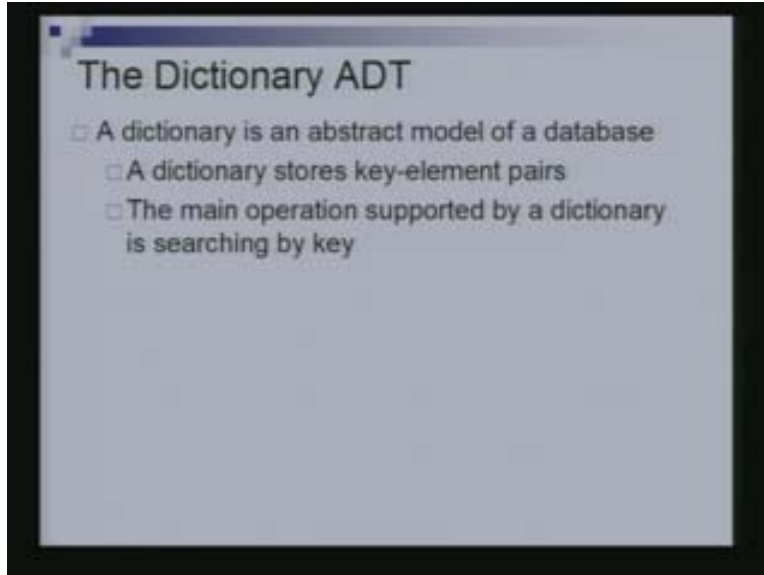
Any application that wishes to do any kind of operation on an account, will have to provide the account number as key. The process cannot be continued without a key.

(Refer Slide Time: 2:14)



The dictionary is basically an abstract model of a database. It is going to store the key-element pairs. The key could be an account number or it need not be an account number in all the case. Suppose if I have the student records then what would be the most natural notion of a key. Your entry number which is not an integer, you may also have characters and so on. Anything uniquely identifies particular student or particular account becomes the key.

(Refer Slide Time: 3:03)

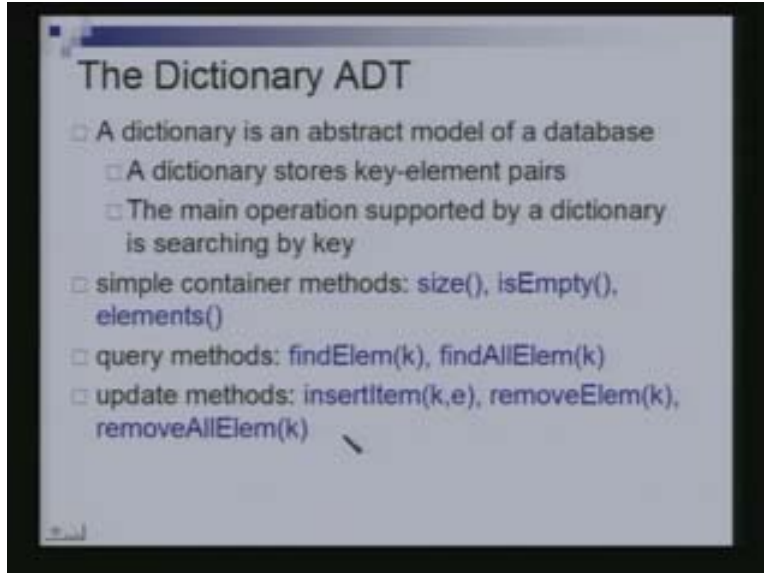


One of the main operations that is supported by a dictionary is searching by key. What are the kinds of method that we have in a dictionary abstract data type? The standard container methods that we have seen for queues and stacks and so on. One is `size ()` which tell us how many elements are there in the dictionary, `isEmpty ()` tells whether the dictionary is empty or not and `elements ()` which returns all the elements in the dictionary.

Then we will have query methods. Given a particular key find the element corresponding to this key (`findElem (k)`). In settings you might have the same key associated with many different elements and one could have such a kind of settings. We will see example of this kind later. Then given a particular key, you want to return all elements who have that key.

You could have update methods. Given I might want to insert an element `e` which has a key `k`, to insert that in to my dictionary, to remove an element with a certain key and to remove all elements which have a key `k`.

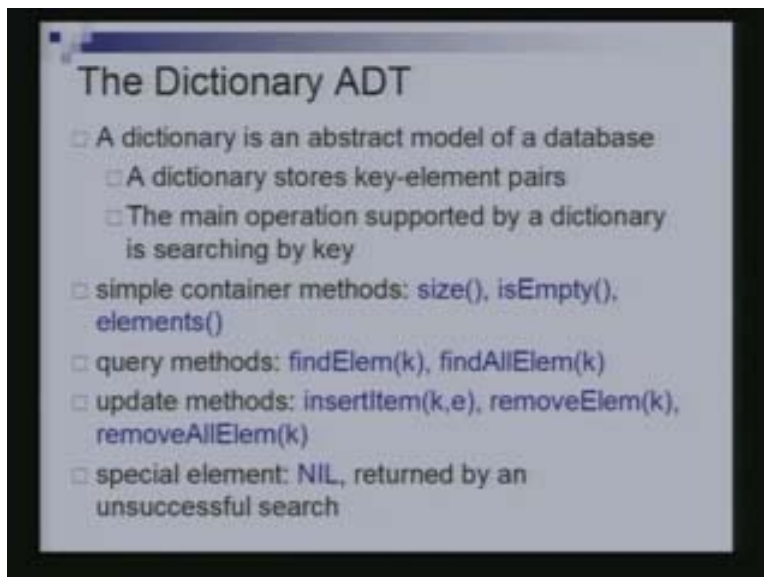
(Refer Slide Time: 3:33)



Just you would think it as a standard thing, but now you should remember the notion of the key and that is crucial for help in searching. Without the key it will be very difficult for us to search through the database.

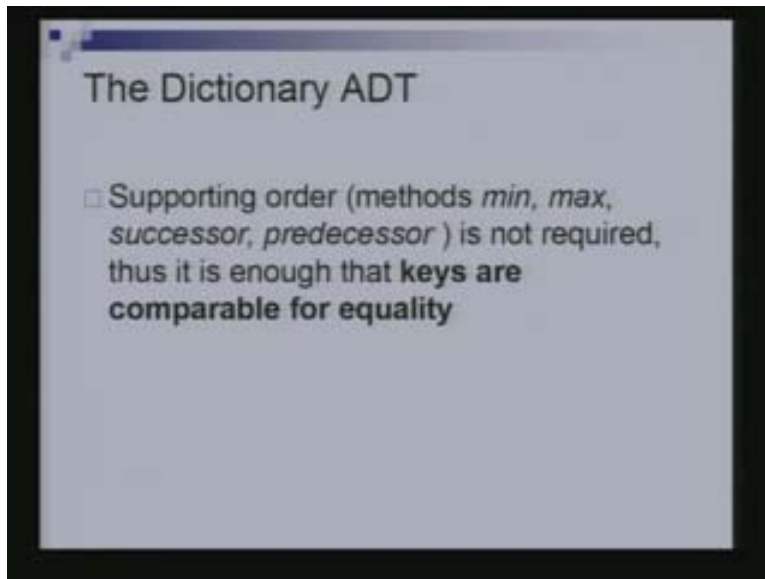
We will have a special element NIL, which will be returned by an unsuccessful search. It means that if I am searching for a particular key, there is no such element with that key in my dictionary and then my procedure will just return a null. This is some special element.

(Refer Slide Time: 5:02)



One thing that we will keep in mind is that we only require comparison of keys for equality. Given two particular keys, we are not going to say one key is less than the other and one key is more than the other. This is not really required, because all you doing is searching for a certain key. All that is required is, given two keys you want to say whether they are the same or not.

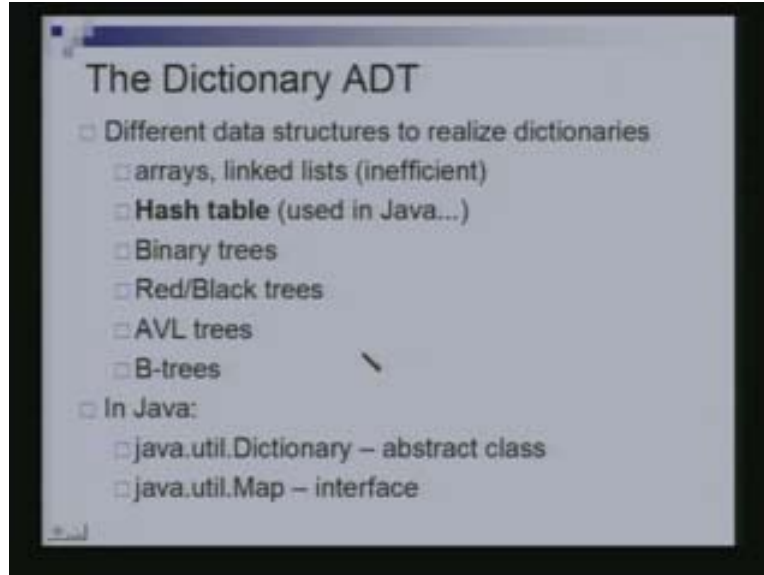
(Refer Slide Time: 5:31)



For instance again for student record, I could have name as your key in that case you know there is really no notion of taking two names and saying one name is smaller or larger than the other. Only operation we require is comparing keys for equality. We do not need any order on our keys.

Dictionary is the abstract data type then in the next few lectures we will see how this particular abstract data type can be implemented. We are actually going to see many different ways of implementing this abstract data type. You already have seen some ways of implementing this abstract data type.

(Refer Slide Time: 6:36)



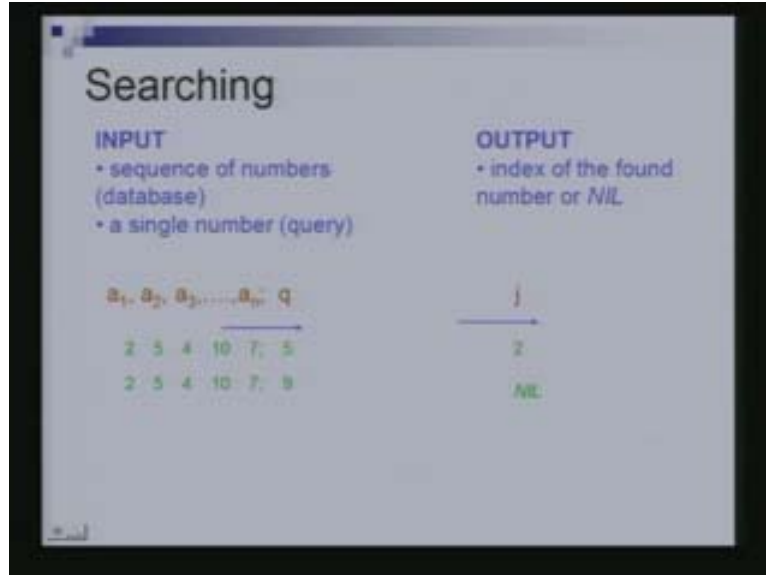
For instance you could use an array or a linked list to implement a dictionary. Suppose if I had student records, how would you use a linked list to implement this dictionary? Every node would have the key which is the entry number and all the data associated with that particular entry number. It is a very inefficient but it is the way of doing things. There is no notion of a predecessor or successor.

If we were using a linked list how would we organize it? That is up to you, you could connect them completely arbitrary manner. The nodes could be in completely arbitrary manner or you could might want to organize them some way but you could just throw them arbitrary in to the linked list. That is the implementation of the linked list and it is not very efficient.

Today we are going to see the hash table and in later class we are going to look at binary tree, red or black trees, avl trees, B-trees. These are all the mechanisms of data structures to implement this dictionary abstract data type which is a very important data type. We are going to be spending quite some time on this particular data type in these lectures.

In java you have an abstract class called java.util.dictionary which lays out the specification also an interface called java.util.map. Before you get more into dictionary you have to understand about the abstract data type. Let us look at the problem of searching. This is the small aside but we will see why I am making this aside and how it will end up to the subsequent discussions.

(Refer Slide Time: 8:23)



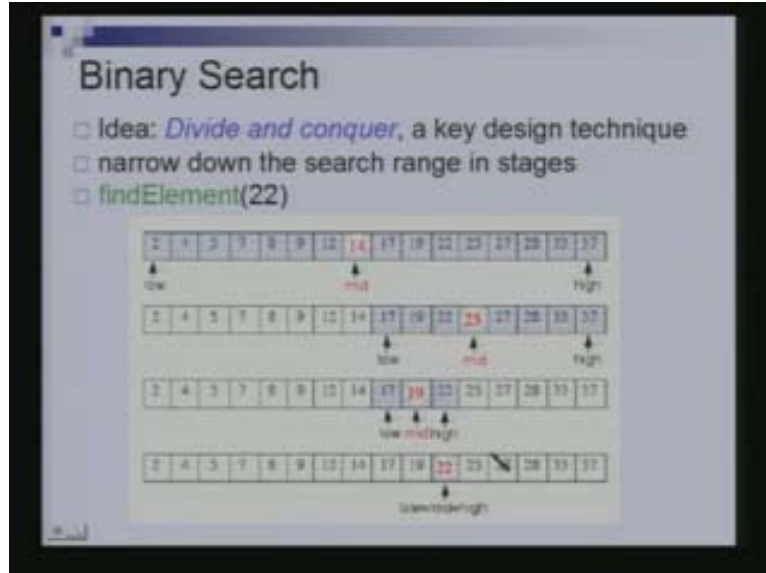
The problem of searching is if you are given a sequence of numbers let's say these are your keys in the database. I give you a single number which is my query. For example I could have 2, 4, 5, 10, 7 are the keys in my databases and I query 5. What do you have to return?

You have to return the position of the key 5 in the database where it is sitting. Index of the found number or nil. So 5 is sitting at position 2 and you return 2, while 9 does not appear anywhere and hence return nil. This is the problem and we call it as searching.

Let us see how we can do that. You are going to see a technique called binary search. I think all of you have seen this before but again we are going to recall the technique and do an analysis of it.

The key idea behind your binary search is divide and conquer. This is a design technique that we are going to see in future classes also and applied to different problems. In divide and search we really narrow down the range of elements in which we are searching for the query that is the key. Let me take you through an example.

(Refer Slide Time: 10:14)



Suppose the elements given in the slide are sitting in an array and these elements have to be in a sorted order, increasing or decreasing for binary search to work. Suppose I am searching for an element 22.

What do I do in binary search? I will go and look at the middle element, in this case it is the element 14. I will compare 14 and 22. As 22 is larger than 14 that means if that 22 lies in this databases it would lie to the right of 14, because these set of elements are in increasing order. This means that the element 22 has to lie between 14 and 37. We use these 2 variables low and high to indicate the part of the array in which we have to search for our element. Initially we have to search for the element in this entire array but after looking at 14 we have figured out that we should search for the element only after 14 in the 2nd part of the array.

Once again we have to repeat the same thing that is go to the middle element, compare 22 with this middle element. As 22 is less than the middle element which means the 22 has to be to the left of 25. This means we are searching for 22 between 17 and 22 which is given in the 3rd part of the diagram in the above slide.

Once again we go to the middle element, we compare the middle element 19 with 22. 22 is larger and if 22 is their, then it has to be in the particular location and it is their. Hence we can say that 22 is at the location which is given in the slide and return this information. You all have seen binary search before, this is nothing new perhaps.

I am going to write down a recursive procedure to do binary search. The procedure given below in the slide is just a pseudo code and all of you can read and understand this quickly.

(Refer Slide Time: 12:15)

A recursive procedure

```
Algorithm BinarySearch(A, k, low, high)
if low > high then return Nil
else mid ← (low+high) / 2
  if k = A[mid] then return mid
  elseif k < A[mid] then
    return BinarySearch(A, k, low, mid-1)
  else return BinarySearch(A, k, mid+1, high)
```

The slide includes three diagrams illustrating the binary search process on an array of 27 elements: [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 29, 31, 33].

- Diagram 1:** Initial state with low=0, mid=13 (value 27), and high=26. The element 14 is being searched.
- Diagram 2:** Since 14 < 27, the search range is updated to low=0, mid=6 (value 9), and high=12.
- Diagram 3:** Since 14 > 9, the search range is updated to low=7, mid=11 (value 22), and high=12.

Once again we had the notion of low and high. Low was the lower end of the range and high was the higher end of the range which we have to search. In the procedure call A is the array and k is the element or the key which you are searching for, low is for lower end and high is for higher end. If low is more than high then basically that means you are invoking something wrong. You just return a null that is the key is not there. Else you go to the middle element which is obtained by taking the average of low and high and check if the middle element is the key you are looking for. If it is the key you are looking for then just return the position where you found the key, so you will return mid.

If the key you are looking for is less than the middle element then you know that you have to search in the left part of the array. The left part of the array has a starting location low and the ending location mid-1. You are going to search in that, and this (return BinarySearch (A, k, low, mid-1)) is your recursive call to that procedure.

If it is not the case you come to the else and in this case what you have to do is to search in the right part of the array which means the mid+1 to high. Else return BinarySearch (A, k, mid+1, high) This is how you can do binary search for small pieces of code and this is recursive.

You can also write an iterative procedure for this, there are exactly equivalent so that you know how to go from a recursive procedure to an iterative procedure. I have just written down an iterative procedure also.

What is happening in our iterative procedure? The low to begin with 1 and high to begin with n. In the slide below blue color is the element 1 through n, let say in an array and we are doing the same thing roughly except now we are putting it in a loop and updating high and low every time.

After the first step low becomes mid+1, because the element was larger than the mid element $A[\text{mid}] > k$, low becomes mid + 1. When the element is smaller than the mid element then high becomes mid-1 and we just go through this loop till we either find the key. In that case we just return the location where we found the key or low becomes larger than high in which case you would come out of this do-while loop and return NIL.

(Refer Slide Time: 13:54)

An iterative procedure

INPUT: $A[1..n]$ – a sorted (non-decreasing) array of integers, key – an integer.
OUTPUT: an index j such that $A[j] = k$.
 NIL, if $\forall j (1 \leq j \leq n): A[j] \neq k$

```

low ← 1
high ← n
do
  mid ← (low+high)/2
  if A[mid] = k then return mid
  else if A[mid] > k then high ← mid-1
  else low ← mid+1
while low <= high
return NIL
  
```

The diagram shows an array of 10 elements. In the first step, the middle element (index 5) is compared to the key. Since it is greater, the search range is updated to the first 4 elements. In the second step, the middle element (index 2) is compared. Since it is less than the key, the search range is updated to the last 3 elements (indices 3-5). This process continues until the key is found.

I have just shown you how to write a binary search in two different ways. You can write it in a recursive procedure or in an iterative procedure to do the same thing. How much time does binary search take? $\log_2 n$. You all know that but why does it takes $\log_2 n$ time. Exactly the size of the problem is halved at every step. Range of the items that we have to search in is halved after each comparison. If essentially the range of the elements in which I am searching for my key is n then after the first comparison it goes down to $\frac{n}{2}$.

After the second comparison the range goes down to $\frac{n}{4}$ and so on. After $\log n$ comparison the range would go down to 1. When the range goes down to 1, either it is that element or it is not that element and you can stop. You will roughly require $\log_2 n$ comparisons. If you have an array implementation that is your elements sitting in an array, then you can go to which ever location you desire in constant time.

(Refer Slide Time: 15:59)

Running Time of Binary Search

- The range of candidate items to be searched is *halved after each comparison*

comparison	search range
0	n
1	$n/2$
2	$n/4$
...	...
2^i	$n/2^i$
$\log_2 n$	1

- In the array-based implementation, access by rank takes $O(1)$ time, thus binary search runs in $O(\log n)$ time

Then each of these comparisons can be done in one unit of time that is in constant time and you can do the entire process in only $O(\log n)$ time. When you do not write any base for $\log n$, we will understand that it is base 2.

Suppose the numbers in that array were not in sorted order. Till now we assumed that the numbers were in increasing order. You can do binary search even if the numbers were in decreasing order but if they were in sorted order you can do binary search.

(Refer Slide Time: 16:52)

Searching in an unsorted array

INPUT: $A[1..n]$ – an array of integers, q – an integer.
OUTPUT: an index j such that $A[j] = q$. NIL, if $\forall j (1 \leq j \leq n): A[j] \neq q$

```
j ← 1
while j ≤ n and A[j] ≠ q
do j++
if j ≤ n then return j
else return NIL
```

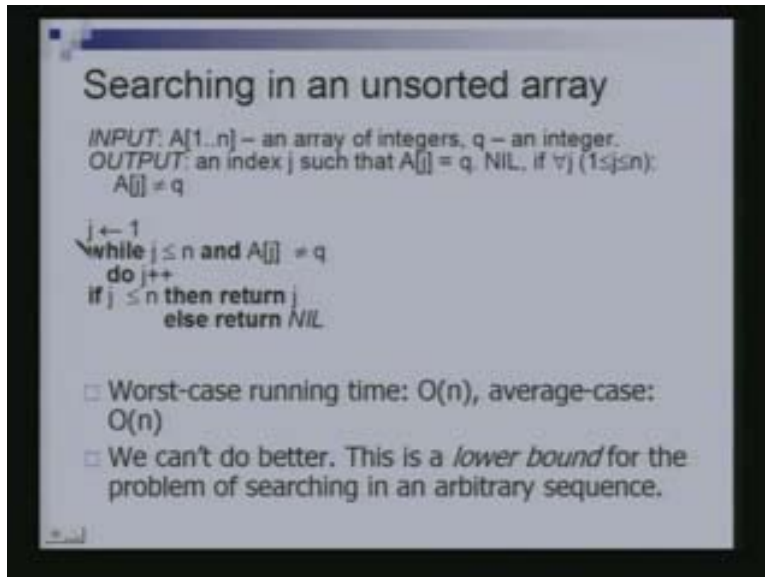
- Worst-case running time: $O(n)$, average-case: $O(n)$
- We can't do better. This is a *lower bound* for the problem of searching in an arbitrary sequence.

If they were not in sorted order and still you are searching for a key k . There is nothing else you can do, but to go through the entire array one element after the other and compare your key against that. That is the only thing you can really do.

The worst case time, then becomes order n . If you are lucky you might get the element at the beginning of your search. On an average you will get it, it may be 1st time you got it at the very first position, some other element you searched and got at the 3rd position. Hence in an average you are going to spend some order n time.

This is really the best you can do, if the array is sorted. You can see there is a huge difference coming up already. By sorting, if you were to sort the element to begin with, then you can search much more quickly. Given in the slide is a small pseudo code which is just saying that you run through the array one element at a time and compare your key against the element in the array.

(Refer Slide Time: 18:16)



Searching in an unsorted array

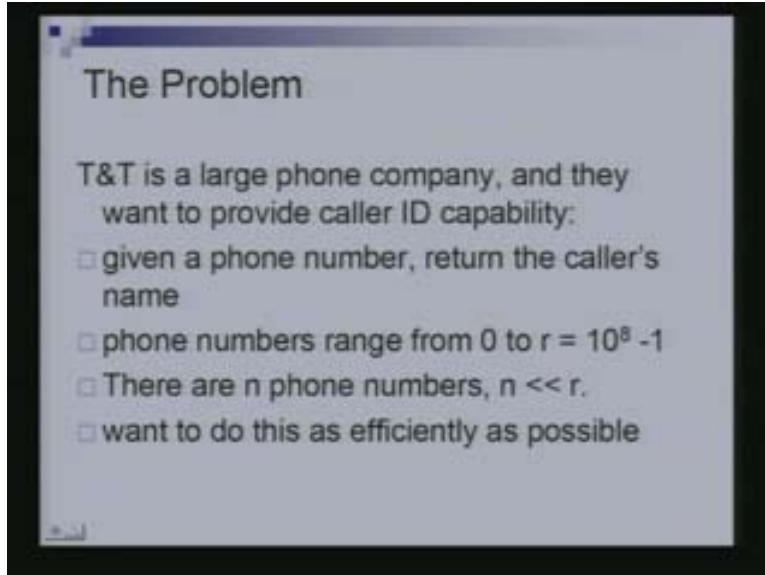
INPUT: $A[1..n]$ – an array of integers, q – an integer.
OUTPUT: an index j such that $A[j] = q$. NIL, if $\forall j (1 \leq j \leq n): A[j] \neq q$

```
j ← 1
while j ≤ n and A[j] ≠ q
do j++
if j ≤ n then return j
else return NIL
```

- Worst-case running time: $O(n)$, average-case: $O(n)$
- We can't do better. This is a *lower bound* for the problem of searching in an arbitrary sequence.

That ends my aside on searching and I am going to go back to my dictionary problem. I am going to look at the setting where you are asked to implement a caller id facility for a large phone company. Given a particular phone number when a call comes in, based on the phone number you can figure out the name of the person who is making the call.

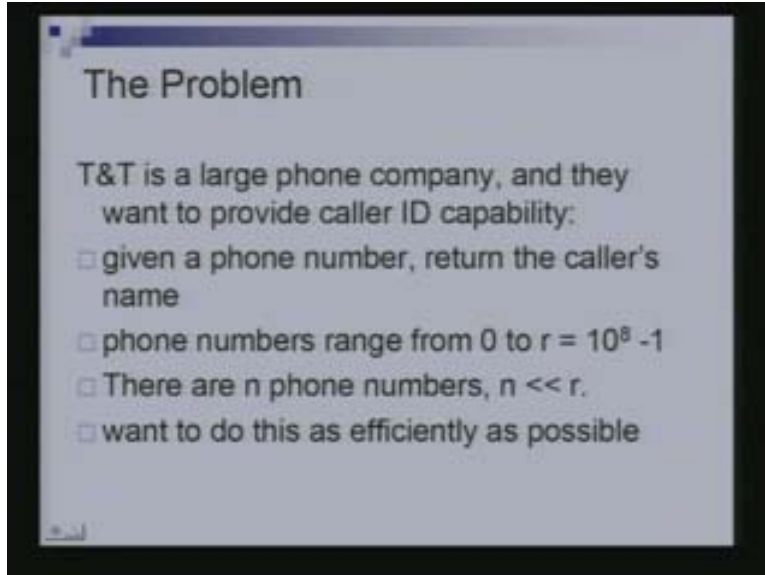
(Refer Slide Time: 18:30)



That is what the company wants to do. Given the phone number you want to return the caller's name. Let us assume that our phone numbers are all 8 digit numbers as in the case in Delhi. The range of phone numbers would be $10^8 - 1$ that is of 100 million phone numbers. The number of different phone numbers is much less than this. The range is 100 million, but may be Delhi has only about a million phone numbers. That is because not all numbers are present.

There are n phone numbers and n is much smaller than r . r is the range which is 100 million because the phone numbers are 8 digit numbers and n is the actual different number of elements in your database. You want to do this as efficiently as possible.

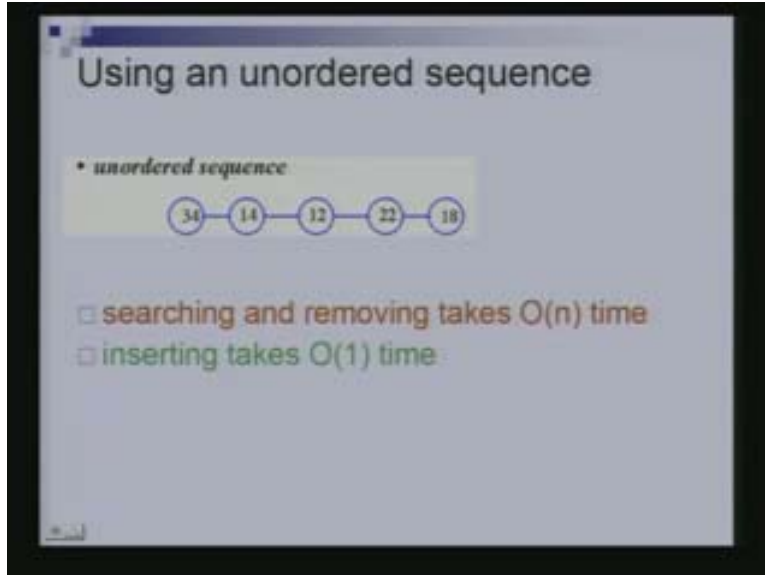
(Refer Slide Time: 19:14)



You can use an unordered sequence to do this. It means suppose the numbers given in the slide below were the phone numbers. I have not put down the 8 digit numbers but let say these were the phone numbers and you could just put them in a list in any arbitrary order.

How much time does searching take? Order n , because you cannot do anything else but to traverse through this list. In the worst case you might have to go through the entire list. Hence it will take order n time to search for a particular phone number. Given a particular phone number, if you have to return the name of the person, you will have to take roughly order n time to do that.

(Refer Slide Time: 21:44)



The list in the above slide is an unordered list that is there is no particular order. How does one remove an element? Suppose a particular person decides to give up his connection. You have to remove that particular data record from this list. First you will search for the record then you will remove it. Searching itself takes order n time then removing also takes at least that much time. Once you found where the element is then you can do some small modification to do the entire thing in order n time.

Why does inserting take only constant time? Because it is an unordered list, you do not really care for where you are putting the elements. You might put at the very first location. Thus inserting takes very less time. It is not clear whether this particular implementation is good for this application.

But there are certain applications in which, this way of doing things is fallible and one such application is where you have to maintain log files. When you have to maintain some kinds of log file for instance any kind of transactions that are happening in the database you try to maintain log files.

(Refer Slide Time: 22:45)

Using an unordered sequence

- *unordered sequence*

34 — 14 — 12 — 22 — 18

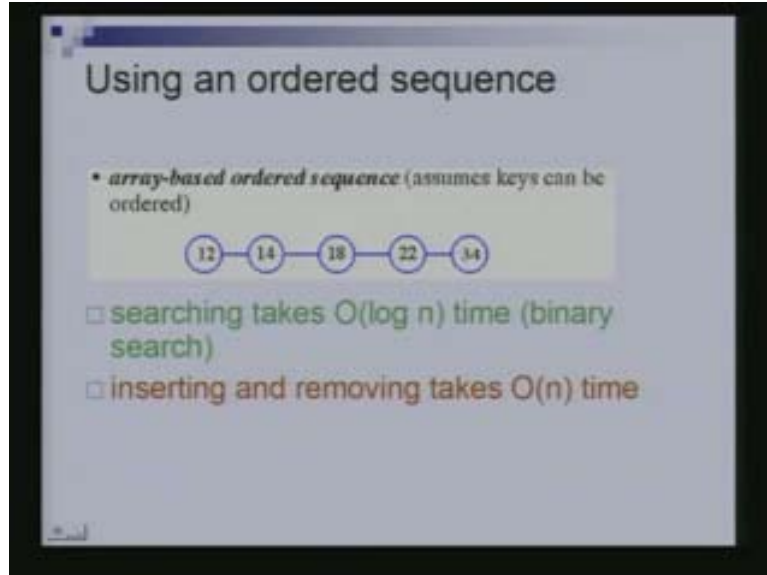
- searching and removing takes $O(n)$ time
- inserting takes $O(1)$ time
- applications to log files (frequent insertions, rare searches and removals)

If there is any problem you can figure out, you can revert the transaction whatever was done or for instance in your system administration you would keep track of all the various activities that were happening in your system and maintain the log of them.

For log files, it is very rarely that you need to do search or removals but you need to add data frequently to your file. Every time when some transaction happens you need to add. Insertions are very frequent but searches and deletion are much rare. In that case this implementation is good because insertion takes only constant time.

Really you have to see between these three operations (search, remove and insert) for which is the operation being performed more frequently, to decide what type of data structure used to implement the dictionary data type.

(Refer Slide Time: 23:24)



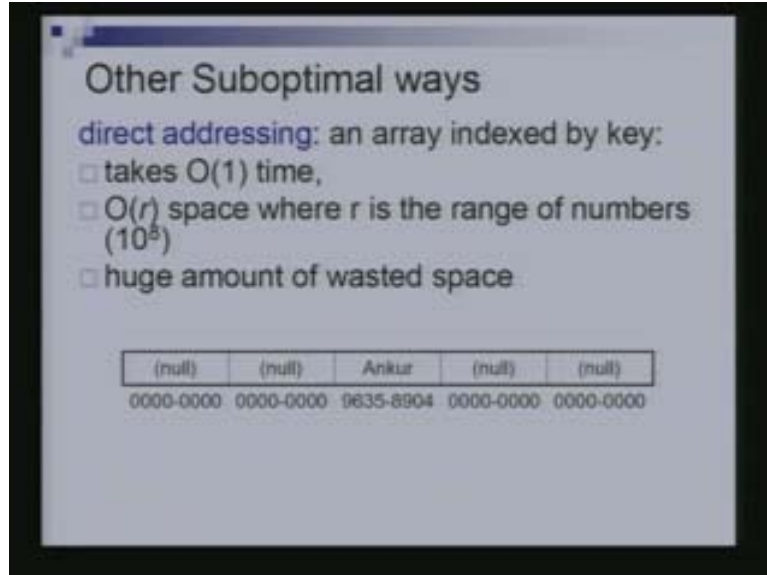
You can use ordered sequence also in that case let us say the elements were put in an increasing order of the key. Searching takes $\log n$ time. $\log n$ provided, you had some kind of direct access mechanism into the thing which is used in array or some other thing which let you go to whichever element you wanted to go.

Searching takes only $O(\log n)$ time, inserting and removing will take order n time because if all of them were put in an array. Then if I have to maintain the sorted order and to insert the element in a particular location, I have to shift everything to the right of the element. Insertion will take order n time in the worst case and similarly deletion you may have to move it back. We have seen examples in previous class.

Why does insertion take order n time? First we have to search for where the element has to be inserted and once we know the position then this is an array in which all the elements were put in an array. We have to create space there by moving everything to one step right. In the worst case we might have to move order n elements to the right.

What is order $n + \log n$? It is order n , you have to recall your big-oh notation, order $n + \log n$ is order n . This would make sense, when you have to do a lot of searching in your dictionary but not many insertions and deletions from the dictionary. There is one other way which will be useful for our subsequent discussion and that is as follows. Let us say I take an array of size 10^8 which is a huge array.

(Refer Slide Time: 25:28)



Ankur had a phone number of “9635-8904 “, I go to location “9635-8904 “ in the array and put Ankur name there and the additional information associated with that in that array. At that very position which corresponds to Ankur phone number. All operations insert, search and delete will take only constant time.

Why because I just have to insert a caller id capability. What does that mean? Given a phone number, I want to know who is that person. Given the phone number I just go straight in to that location of the array and retrieve the name. Given a phone number I want to create a new phone connection, so I take the phone number and go to the particular location, put the name of the person who got the connection.

Similarly if I want to delete a particular phone connection, I just go to the particular location and I remove the element. All the operations take only constant time. What is the bad thing with this implementation? You are wasting a lot of space. It is not that you cannot do all of these operations very quickly. You can but here space is turning out to be an issue.

We are going to use what is called hashing. We are going to use a hash table which will tell us do the things quickly, I have said $O(1)$ excepted time. It will not take too much space either and let us see the idea.

What was the problem with the previous technique? In the previous technique we had 100 million phone numbers, so we had to create an array of size 100 million. But let us say there were only 1 million users, most of the array was getting wasted. There was nothing in their.

(Refer Slide Time: 27:18)

Another Solution

- Can do better, with a **Hash table** -- $O(1)$ expected time, $O(n+m)$ space, where m is table size
- Like an array, but come up with a function to map the large range into one which we can manage
 - e.g., take the original key, modulo the (relatively small) size of the array, and use that as an index
 - Insert (9635-8904, Ankur) into a hashed array with, say, five slots: $96358904 \bmod 5 = 4$

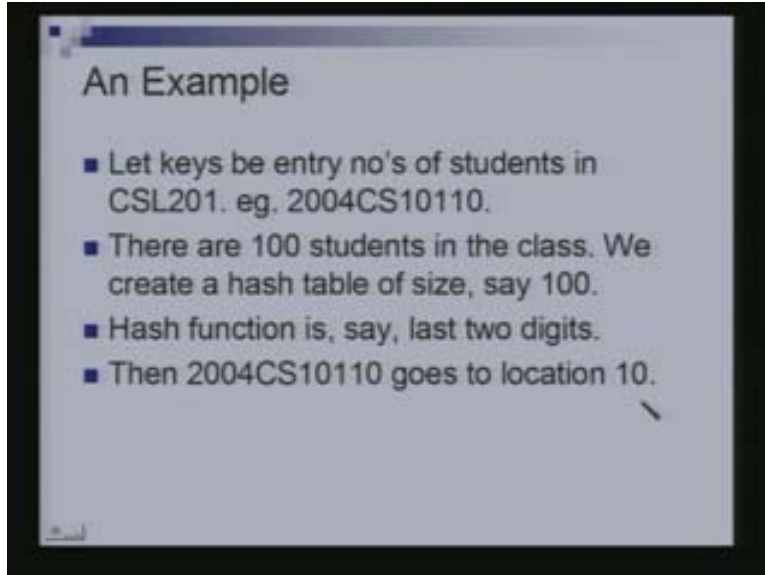
(null)	(null)	(null)	(null)	Ankur
0	1	2	3	4

Suppose I could create a smaller array with only 1 million locations in it and mapped those 1 million users to locations in that array. That is what we are going to do.

Let us say in a hypothetical setting they were only small number of users. I was only trying to keep the phone numbers of 5 of my friends but still I wanted to do something fancy. I create an array of 5 elements and I take Ankur phone number and I compute this value $96358904 \bmod 5$, which is the size of my array. I get a number between 0 and 4. In this particular case I would get 4. Depending upon what I get, I put Ankur at location 4 in my array. I am not using too much space and may get away with constant time for insertion and delete. Let us see whether we can or we cannot do that.

Let me take another example just to make sure that you understand about the idea. Let us say keys are not phone numbers but entry numbers of students in this class. Your entry numbers looks something like this “2004 or 3 or 2”, then you have 2 characters and then you have another 5 digits at the end. There are about 100 students in this class, the range of this numbers is huge. This is infact, I do not even know the range because they can take different set of values. But I would like to create a table of size of about 100, because there only 100 people in the class, why should I spend more space than that.

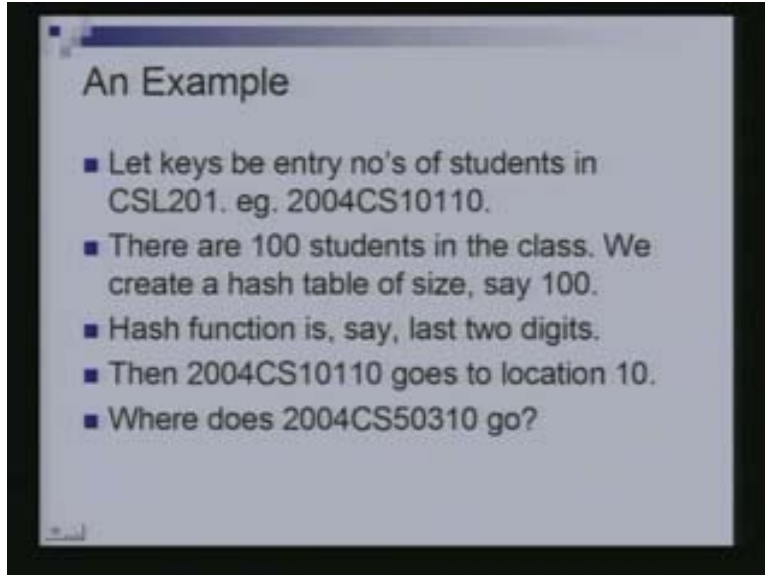
(Refer Slide Time: 29:37)



Let me pick up a hash function. This function which I was using in the previous example mod 5 is also called as a hash function. I am going to pick up a hash function which does the following. It takes the last 2 digits of your entry number. In this case "2004CS10110" would get mapped to location 10. It just picks up the last 2 digits of the entry number. That is what I am going to do.

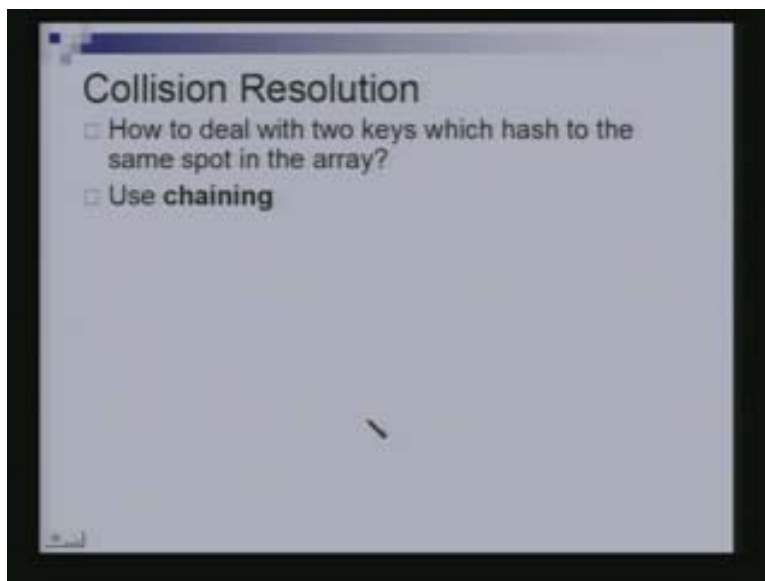
I am going to take each one of your entry numbers, going to look at that last 2 digits and I just have a table of size 100. I am just going to put you in that location, depending upon the last 2 digits. What if I had a clash? Suppose I had another person with this entry number 2004CS50310. I do not know if they are in this class, there could be another person also. The problem is these 2 are going to the same location number 10. We will come to this problem but if this problem did not arise then did you see that we are in very good shape. Then you can do insert, delete and search all in constant time because it is very much like the array implementation that I showed you. Except that it does not waste all the space.

(Refer Slide Time: 31:12)



Let us see how we can address the problem of clash. If 2 elements are getting mapped to the same location in our hash table, this is called the collision. We have to find a way to address this issue. How do we deal with the 2 keys which mapped to the same spot in our hash table? We use the concept called chaining. There are many ways of addressing this issue and today in our class we are going to look at the very first, simple technique called chaining.

(Refer Slide Time: 32:07)

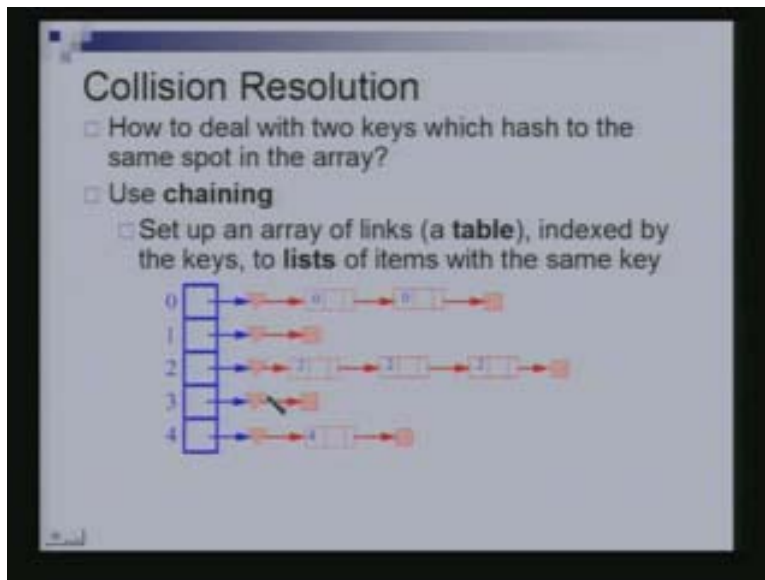


The blue color thing in the slide below is my hash table. I am not going to put the elements in this hash table but I am going to have a linked list starting at each of the

locations. I am going to put the elements in the linked list. Suppose my hash table had only 5 locations in it. May be I was just using the hash function which was taking the key and computing modulo 5 or some other thing.

If 2 or more keys were mapped to the 2nd location, I will just keep adding them to the linked list. As you can see from the picture given in the slide below, it was the case 3 keys were getting mapped to location 2 and 1 key was getting mapped to location 4. There were no keys getting mapped to locations 1 and 3. This does address the problem of collision but what is the other problem does it create.

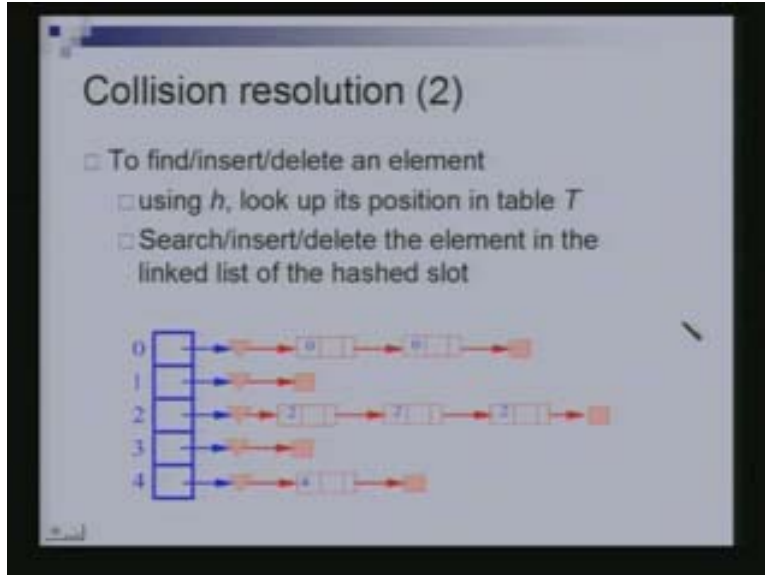
(Refer Slide Time: 32:34)



While we have resolved the collision problem and we are not able to do things in constant time any more. In the worst case all the keys get mapped to 1 location in this hash table. If all of them get mapped to the same location in the hash table then your data structure reduces to a linked list data structure which we know has the worst case time of order n for search and delete. Still insert has constant time.

Whether each of the nodes in the linked list will contain both the identity in the phone number, caller id example? Each node will have both the phone number and all the data associated with that person who sits there.

(Refer Slide Time: 34:07)

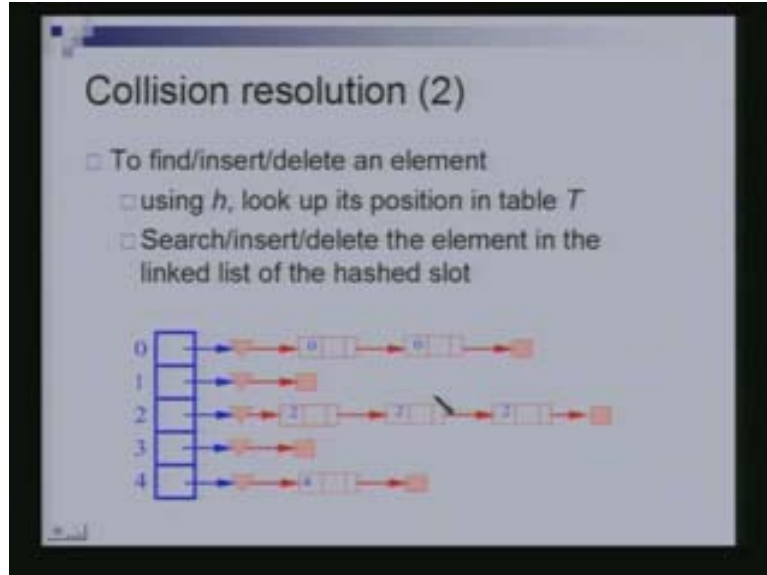


This is quick recap about how we are going to insert and delete of an element. For all of these three operations you have to do essentially the same thing. You have to use your hash function h , to determine where that key is in this table. We have seen 2 examples of hash functions. In one case we said we will just take the key modulo 5, in the other case we said we just take the last 2 digits of the key. But they could be many different kinds of hash function and in the next class we are going to see what are the different kinds of typically used hash functions.

The last 2 digits can also be regarded as modulo 100. The reason I did not write modulo 100 because that was not an integer at all. It was your entry number and it had some characters in it. You are going to use your hash function to find the position of the key in the table. Then if you are going to search or insert or delete, do that in the linked list associated with that position.

There are options that you might want to maintain the list which is in 2^{nd} position in a sorted order, you might want to keep it in unordered. If you want to maintain in a sorted order then insertion is going to take more than a constant amount of time. If you want to keep it in unordered then insertion is going to take only constant amount of time because you can just insert at the very beginning or at very end of the linked list.

(Refer Slide Time: 35:46)



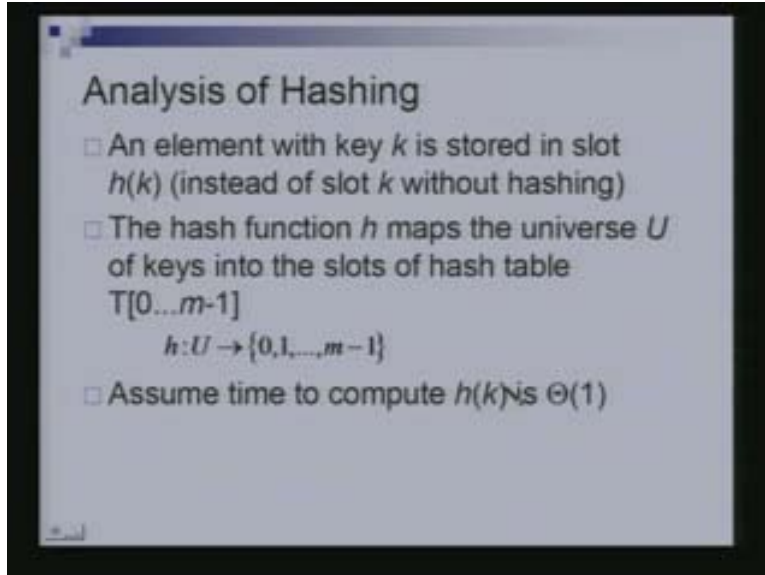
If you want to insert an element at the very end of linked list then you do not need to traverse the entire list to reach the end. You can always have another pointer which always points to the end of the linked list and use that to update. Although there is no reason why you want to insert at the end.

Suppose if you want to insert at the end, you could also do that in constant time always. By maintaining one pointer we will have two pointers from the 2nd location, one going to the front of this linked list and one were going to the tail of the linked list. Use that pointer to add an element at the end of the linked list.

You can do ordering if you want to keep it ordered. We are not saying that in the hash table you have to keep anything ordered. If you want to keep it ordered you can do whichever if there is a notion of order on your keys then you can use that notion to order the elements.

An element with key k is stored in the slot $h(k)$ in which h is the hash function and $h(k)$ is the value of hash function. The hash function is mapping the universe of all keys, let us say U to slots of the hash table. If the hash table was of size m , so it is a function which is mapping from U the universe to 0 through $m-1$. We are going to assume for the rest of the discussion that the time to compute the hash function for given key k is a constant time. Because quite often we just have to do simple arithmetic operations to compute the value of the hash function. We are going to assume that the time taken to compute the hash function is independent of the number of elements in the table.

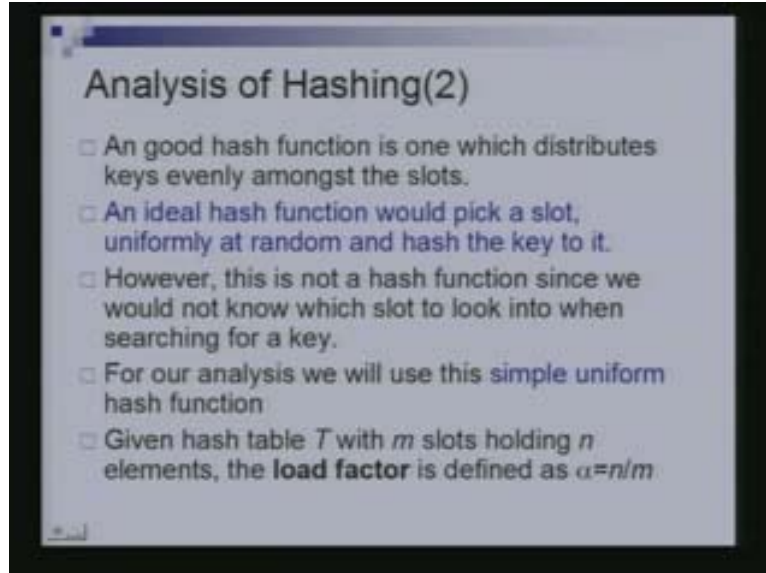
(Refer Slide Time: 37:36)



As far as the choice of hash functions are concerned, we are going to see in the next class what are good choices of hash function. Lot of research has done in this and then we will see what are the kinds of hash function that people typically use. I just gave you 2 simple examples of hash function so as to motivate the concept.

What is the good hash function? A good hash function is one which tries to distribute the keys uniformly over the table. It should not map all the keys to location 1 or location 2 or any such thing, because then there would be too many collisions, your data structure would start looking like a single linked list and that is not what you want to have. You want to have a hash function which distributes things uniformly over the table. Why uniformly, so that each of the list is small.

(Refer Slide Time: 38:53)



An ideal hash function would do something like the following. It would take an element and let us say I have a table of 100 locations. It will pick at random one of those 100 locations then throw the element there. This kind of it shows that every location would have roughly the same number of elements. But this is not a hash function what I just said, you can not have a hash function which takes a key and puts it at a random location.

Why this is not a hash function? Because when I am searching for the element, where am I going to go and look. I do not know what random location it had picked at that point. While this is an ideal hash function, it is not really a hash function.

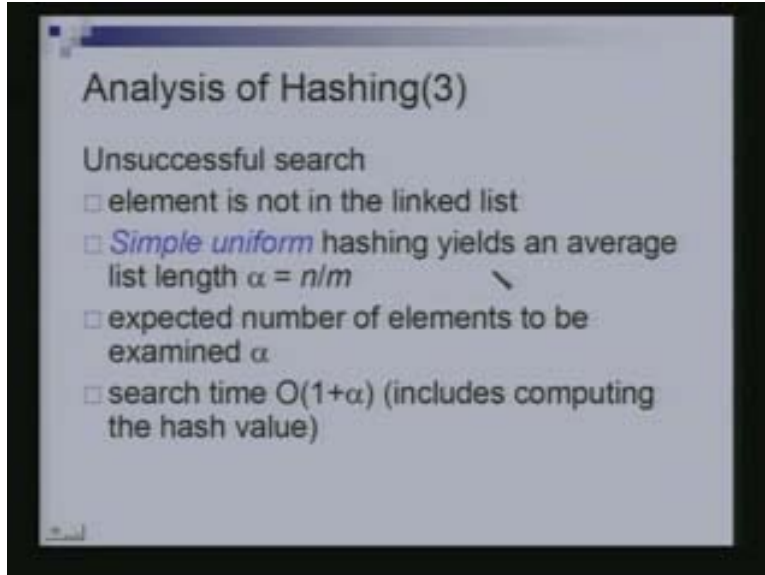
But for our analysis we are going to assume such a hash function. The hash function is just essentially does the following that it takes the element and throws it randomly, uniformly with same probability in one of those locations of the table. We will call this as simple uniform hash function and we are going to use this for analysis.

We will use another term called the load factor of the table which is just a number of elements in the table divided by the number of slots, the size of the table and we will call this load factor alpha.

$$\alpha = \frac{n}{m}$$

What is going to happen when we are trying to search and our search is unsuccessful? It means that I took the element I computed the value of the hash function and I went to the particular slot in the hash table. I went through the entire linked list and did not find the element at all.

(Refer Slide Time: 40:52)

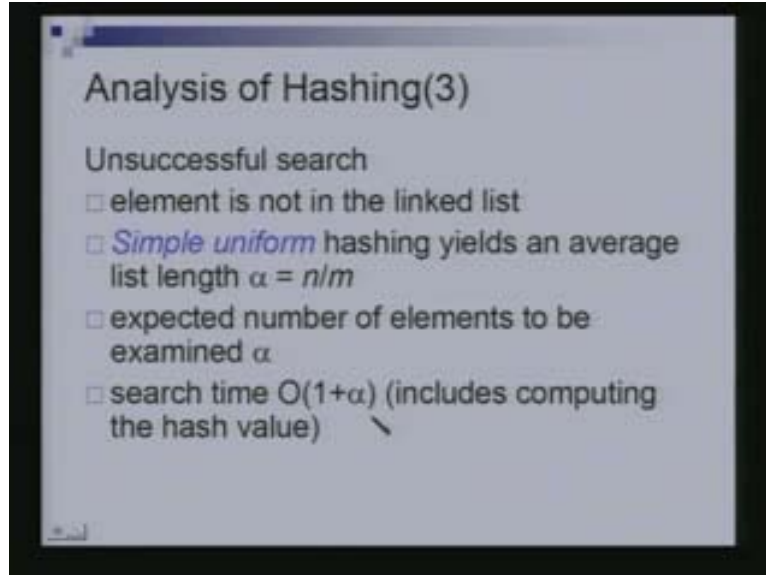


How much time do you spend? I spend time propositional to the size of the linked list that I have to go through. Because I said computing the hash function takes constant time, you did not take any time to go to the right linked list but once you went to the right linked list you still have to step through the entire linked list, follow pointer by pointer till you reach the end. The time is propositional to the size of the linked list.

What is the average size of your linked list? If there are n elements that are thrown in my table and m is the number of slots and if I had this simple hash function which was essentially distributing the things uniformly then on an average you would expect that each linked list is of size $\frac{n}{m}$ which was the load factor of the table. The expected number of elements that need to be examined is α and the total search time where I am using this 1, 2 denote the time taken to compute the hash function is roughly $1+\alpha$.

This tells you that if your α is let us say only a half then the expected search time would be roughly a constant. $O(1+\alpha)$ is expected under the ideal hash functions. You can always create a bad hash function for which the time taken will be order n . $O(1+\alpha)$ represents the time that is spent in computing the hash function.

(Refer Slide Time: 42:57)



We would not want a hash function for which every thing is getting mapped to one location. Because that is a linked list, why would you want to do something like that. This again brings back to the same question, the efficiency of this data structure relies critically on the hash function we choose. We will see what are the good hash functions in the next class.

Designing hash function is much more of an art than science. You have to really look at the data to design a good hash function. I am going to show you in the next class some principle behind the hash function that is what kind of hash functions one should use. But there is no theorem which says that this is the best hash function and you should always use this.

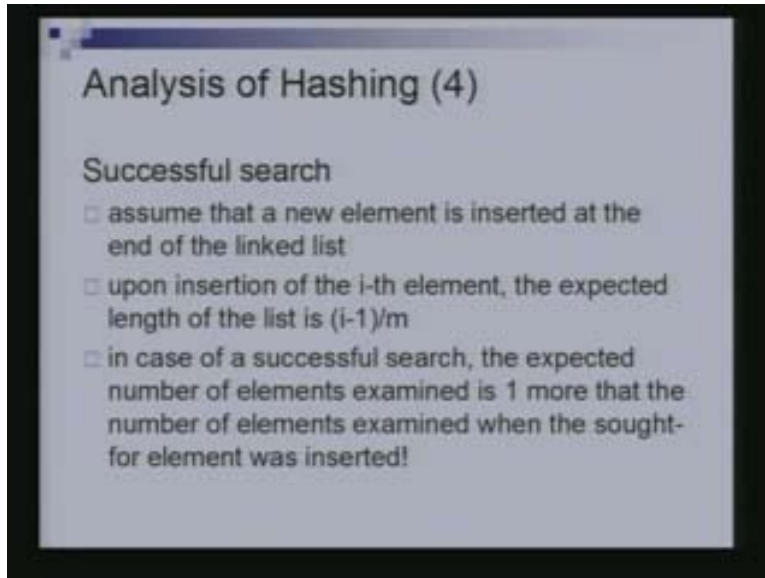
What happens when we make a successful search? That was for unsuccessful search but when I make a successful search, it means once again I took the key, computed the value of the hash function, went in to the appropriate linked list and then I am walking through the linked list. But I do not have to reach till the end of the linked list, at some point in the middle itself I may be able to find my element.

How many elements do I have to traverse in this process? The position at which it was found but how do I know that. What is the expected time I would take overall? Expected time I mean, the average time I would take to search all those n elements that are there in the database.

You can have many different ways of arguing it but let us do it in the following way. Suppose I was searching for the i^{th} element which was inserted in to my database. The element or the key that I am looking for was inserted in to the hash table when there were only 9 elements. This was the 10th element, if it was the 10th element that was inserted

then the expected length of the list in which it was inserted was 9^m . That is what we argued just now.

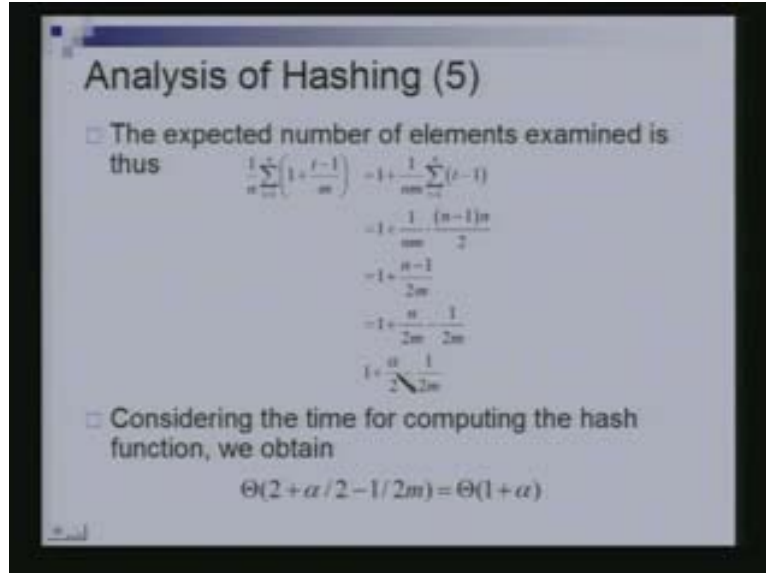
(Refer Slide Time: 45:17)



Exactly m is the number of slots in the hash table. In the case of successful search expected number of elements examined is 1 more than the number of elements examined when that particular element was inserted. When the 10th element was inserted $i=10$, I went through the linked list in which the element was inserted and appended at the very end.

I had to compare that element with all the various elements. When I insert the element basically it is same as that the number of comparisons I have done is, 1 more than the number of comparisons I would have done in an unsuccessful search. We have to go through the entire linked list when we are inserting because to make sure that the element is not already there then we might insert at the end. We could also insert at the beginning but it is the same thing.

(Refer Slide Time: 47:15)



One could have an analysis which looks like the following. This is not critical you can all prove it in different manners. I am looking at the element 1 through n, there were n elements in my database. When the i^{th} element was inserted, then the expected length of the linked list at the end of which it was inserted is roughly $\frac{i-1}{m}$ and the 1 is for our hash function computation.

This is $(1 + \frac{i-1}{m})$ roughly the expected time required to insert at the i^{th} element. We are just summing this quantity up over all the n elements and taking the average. If you just go through this math, you will get something like the following and many of you could have figured this out on your own.

$$1 + \frac{\alpha}{2} - \frac{1}{2m}$$

The average time would be roughly the expected length of the list divided by 2. Whenever we are doing average time computations, when I said take a linked list and what is the average time to search for an element. You said I might have to go till the end of the linked list or find the element right at the beginning, on an average I will take half the length of the linked list.

(Refer Slide Time: 48:27)

Analysis of Hashing (5)

- The expected number of elements examined is thus

$$1 + \sum_{i=1}^n \left(1 + \frac{i-1}{m}\right) = 1 + \frac{1}{m} \sum_{i=1}^n (i-1)$$

$$= 1 + \frac{1}{m} \cdot \frac{(n-1)n}{2}$$

$$= 1 + \frac{n-1}{2m}$$

$$= 1 + \frac{n}{2m} - \frac{1}{2m}$$

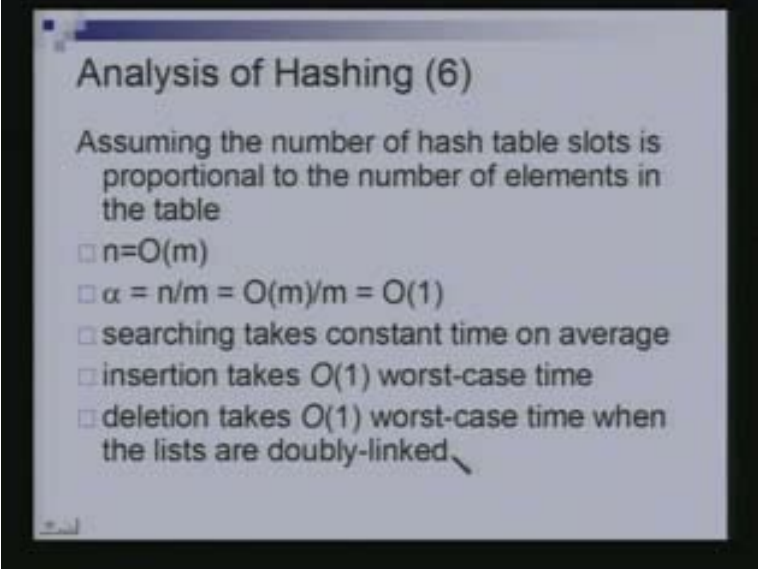
$$= 1 + \frac{\alpha}{2} - \frac{1}{2m}$$
- Considering the time for computing the hash function, we obtain

$$\Theta\left(2 + \frac{\alpha}{2} - \frac{1}{2m}\right) = \Theta(1 + \alpha)$$

You are seeing a similar kind of behavior in this $\left(1 + \frac{\alpha}{2} - \frac{1}{2m}\right)$, in which $\frac{1}{2m}$ is the very low order term which you can just ignore. What you are seeing is something like $1 + \frac{\alpha}{2}$, we do not really have to go through this math but you can also follow it, this $1 + \frac{\alpha}{2}$ is more intuitive. One could just say that the average time for successful search would be more like $1 + \frac{\alpha}{2}$. Again it is $O(1 + \alpha)$ where 2 is not important. Both for successful and unsuccessful search we are taking a similar kind of time.

What should α be that is what should be a good choice of α ? Thus the α is the load factor of the table that is the number of elements in the table divided by the number of slots in the table.

(Refer Slide Time: 48:52)



The slide is titled "Analysis of Hashing (6)". It contains the following text and list items:

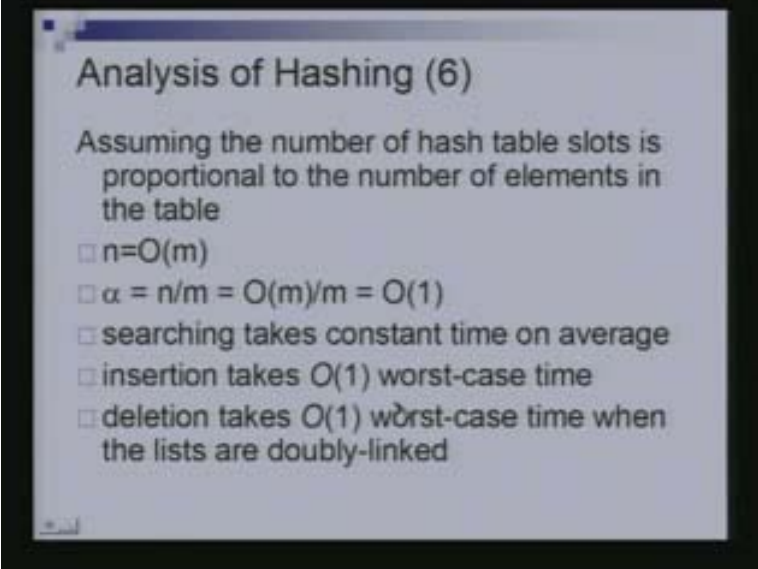
Assuming the number of hash table slots is proportional to the number of elements in the table

- $n = O(m)$
- $\alpha = n/m = O(m)/m = O(1)$
- searching takes constant time on average
- insertion takes $O(1)$ worst-case time
- deletion takes $O(1)$ worst-case time when the lists are doubly-linked

If I pick the size of hash table to be the number of elements that I am going to be inserting in the hash table, then α would be roughly a constant $O(1)$. All your searching, insert and delete would take constant amount of time. In the expected sense I mean when you have an ideal hash function which you can not really have.

What if we did not know how many elements we have to insert then what should we do? With what size our hash table should start? We used a concept of growable stack, so the same idea is used in many of this data structures. You start with some thing small and if the number of elements you inserting becomes so large that the sizes of linked list become very large, then it perhaps time to move the entire set of element in to a larger hash table.

(Refer Slide Time: 49:15)



The slide is titled "Analysis of Hashing (6)" and contains the following text:

Assuming the number of hash table slots is proportional to the number of elements in the table

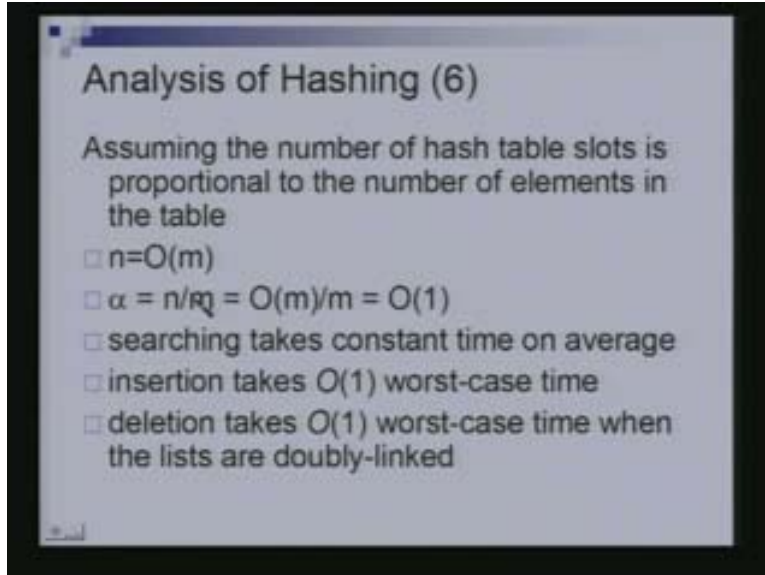
- $n = O(m)$
- $\alpha = n/m = O(m)/m = O(1)$
- searching takes constant time on average
- insertion takes $O(1)$ worst-case time
- deletion takes $O(1)$ worst-case time when the lists are doubly-linked

Either you have to compute a new hash function or we will see how to modify these things. How you can do small modification to the hash function so that you can put it in the larger hash table. One should design your hash function keeping in mind that you might have to go from a smaller table to a larger table and even to a larger table and so on.

What will happen to the space when the number of hash table slots was proportional to the number of elements? It depends upon the big-oh. Let us say we pick the number of hash table slots to be equal to the number of elements. There is no problem, suppose n was a 1000 and m was also a 1000, this hash table can accommodate any number of elements. It not just that it can accommodate only 1000 elements. Why because in the linked list you can attach any number of elements that come.

It is just that performance of the hash table would deteriorate if you had 10,000 elements coming because then each linked list would be of size 10 roughly, may be more or may be less. On an average the linked list length would be 10 in that case it make sense to move to a larger hash table.

(Refer Slide Time: 51:25)



Analysis of Hashing (6)

Assuming the number of hash table slots is proportional to the number of elements in the table

- $n = O(m)$
- $\alpha = n/m = O(m)/m = O(1)$
- searching takes constant time on average
- insertion takes $O(1)$ worst-case time
- deletion takes $O(1)$ worst-case time when the lists are doubly-linked

If you know that the number of elements is only 1000 and you create a hash table of size 10,000 then there is wastage of space. You should always start with a small hash table and if need be grow it, rather than starting with a very large hash table and having wastage of space.

Today we saw binary search which many of you have seen before, we also saw a little bit of hashing and we saw the dictionary abstract data type. In the next class we are going to continue with hashing c concepts of good hash function and see other ways of resolving collision.