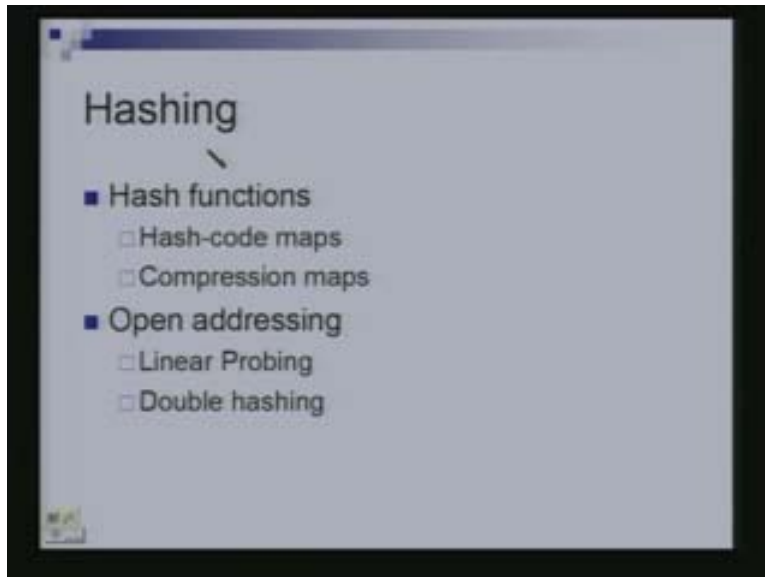


**Data Structures and Algorithms**  
**Dr. Naveen Garg**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**  
**Lecture– 5**  
**Hashing (contd.)**

Today we are going to continue our discussion on hashing. In the last class we saw about the hash table, the concept of hashing and also saw how to resolve collision in hashing using linked list. That method of collision is also called chaining.

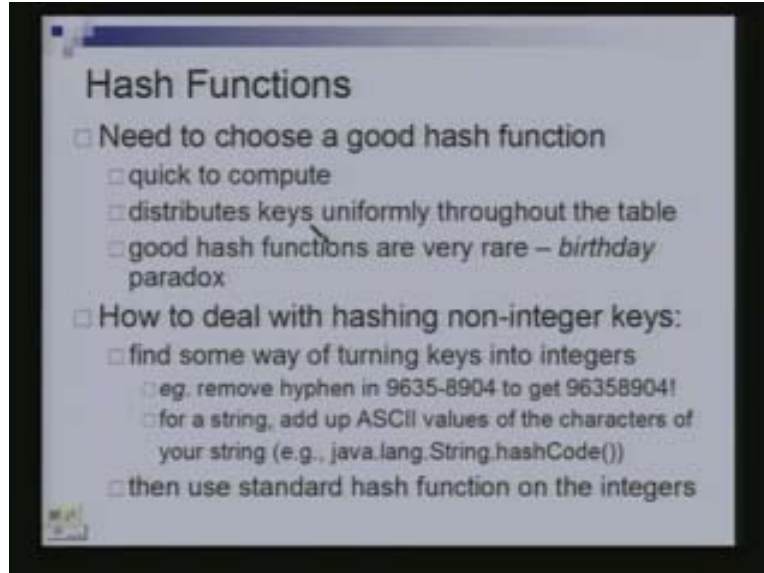
Today we are going to look at 2 other methods for collision resolution, linear probing and double hashing. We are also going to spend some more time discussing how the good hash function should look like.

(Refer Slide Time: 01:34)



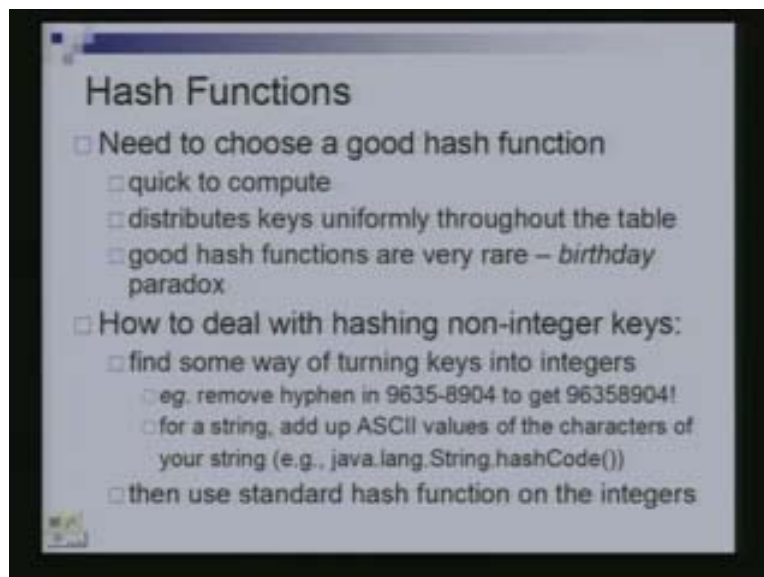
What is the good hash function? The function which can be computed quickly and as said in the previous class, it should distribute the keys uniformly over the hash table.

(Refer Slide Time: 01:53)



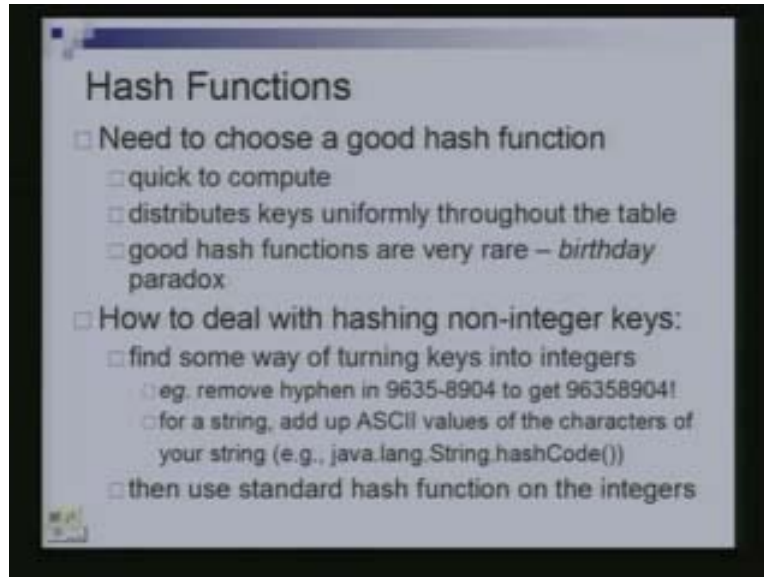
All the keys should not get mapped to the same location because then the performance of hashing would become as worse as that of a linked list. Good hash functions are very rare and there is a famous paradox called birthday paradox. There would be about 35 or more students sitting in the class. There is a very high probability and you can actually compute that probability in which 2 of you would have the same birthday. Although you would think that there are 365 days in the year and if each one of you were to have one of these days as a birthday then there is very small probability that 2 would have the same day. But that is not the case, even with just 35 people you would have fairly high probability that 2 people would have same birthdays.

(Refer Slide Time: 02:10)



The same kind of thing is happening here. Your days of the year corresponding to your slots in the hash table and even if I were to take a key and put it randomly in one of those slots there is very high probability that 2 keys would end up in the same slot that is birthday paradox.

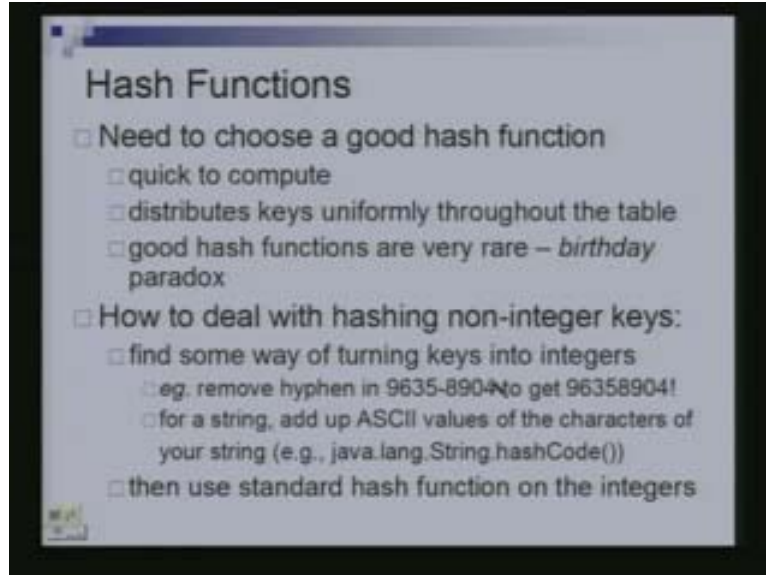
(Refer Slide Time: 02:50)



Collisions may take place in any kind of hash function you use. Then there is also a problem of how to deal with non-integer keys. In fact we saw an example in the last class where the keys were telephone numbers and we had returned the telephone numbers with hyphen.

How did we treat telephone number as an integer? We just dropped the hyphen in between and then we thought of it as an integer. You are going to see some more techniques of converting non-integers keys in to integer ones. The other example that I had taken in the last class was your entry number where again the key was a non-integer because it had C S Y or some other thing.

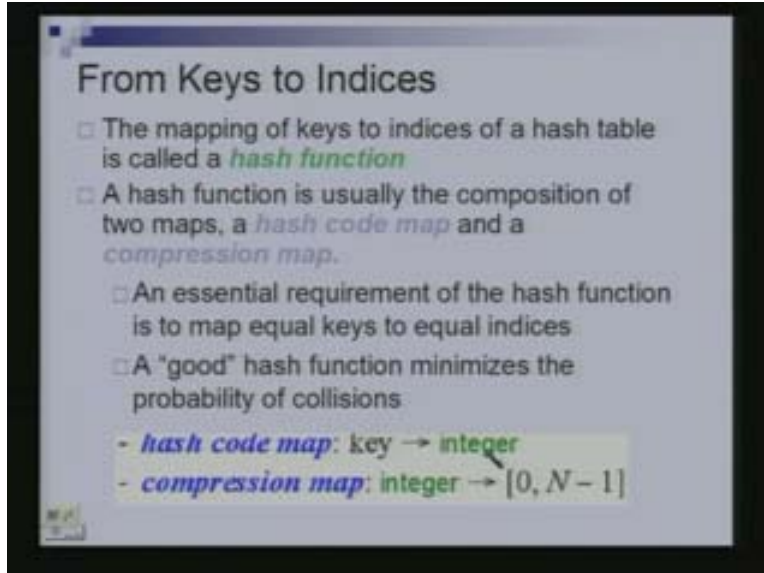
(Refer Slide Time: 03:22)



We have to convert in to integers and what we did in the last class was as I said, those keys are just going to take the last 2 digits as the hash function value. We are going to see some more techniques of converting non-integer keys into integer ones. Hash function can actually be thought of as being in 2 parts. There is a hash code map and there is a compression map and these 2 together make up a hash function.

A hash function is basically a mapping of keys to indices of a hash table. Your hash code map, maps the key to an integer. If your key is already an integer then there is no need for this but when your keys are not integer keys then you will have to 1<sup>st</sup> convert them in to integer keys. Key → integer, this integer could be from an arbitrary range but we need to bring it to the size of our hash table.

(Refer Slide Time: 04:17)



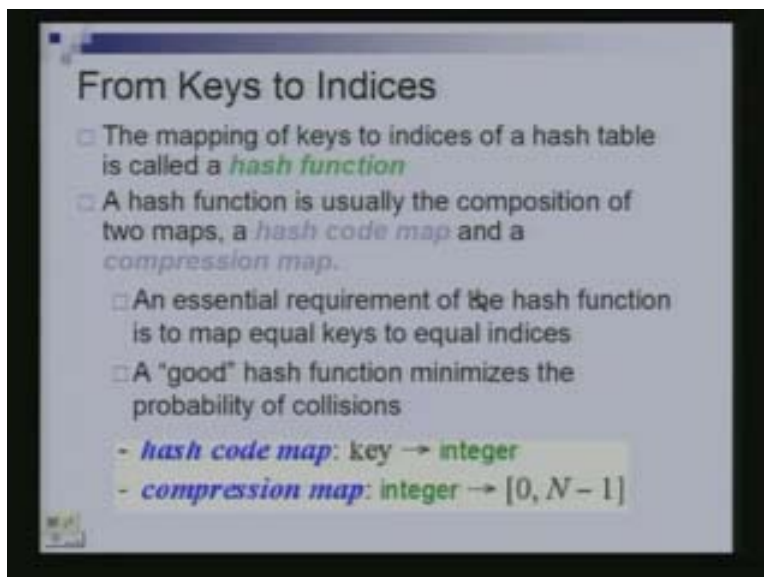
**From Keys to Indices**

- The mapping of keys to indices of a hash table is called a *hash function*
- A hash function is usually the composition of two maps, a *hash code map* and a *compression map*.
  - An essential requirement of the hash function is to map equal keys to equal indices
  - A "good" hash function minimizes the probability of collisions

- *hash code map*:  $\text{key} \rightarrow \text{integer}$   
- *compression map*:  $\text{integer} \rightarrow [0, N - 1]$

If  $n$  is my hash table then I need to bring this integer to the range of 0 through  $n-1$ , so that it can be mapped to an index of my table. That part we will call as compression map. We will see what kinds of functions are used for hash code map and compression map. Another important requirement of hash function is that if 1 key gets mapped to a certain index then the next time when I want to map a key it should get mapped to the same indexed location. It is not like, the next time it should get mapped to some other indexed location.

(Refer Slide Time: 05:03)



**From Keys to Indices**

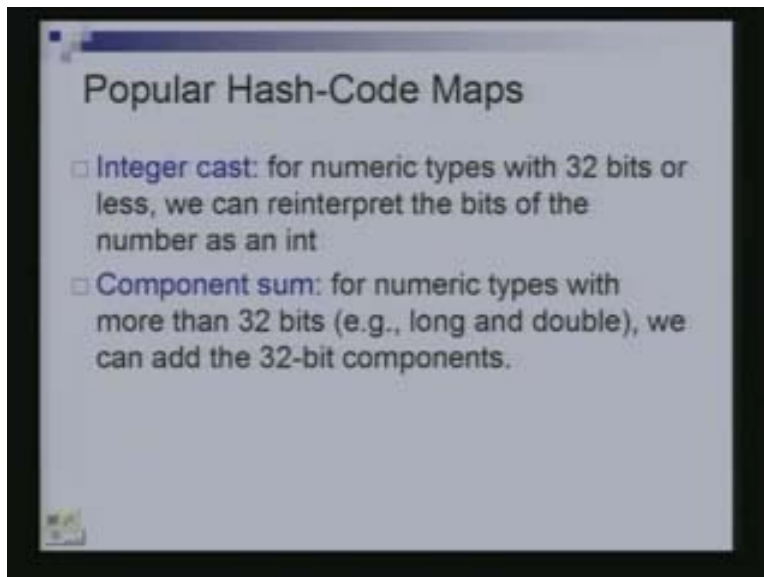
- The mapping of keys to indices of a hash table is called a *hash function*
- A hash function is usually the composition of two maps, a *hash code map* and a *compression map*.
  - An essential requirement of the hash function is to map equal keys to equal indices
  - A "good" hash function minimizes the probability of collisions

- *hash code map*:  $\text{key} \rightarrow \text{integer}$   
- *compression map*:  $\text{integer} \rightarrow [0, N - 1]$

In the last class we took an example of key which was 2004SA10110 and we mapped to location 10. I cannot have a hash function which sometimes maps to location 10 and sometimes maps to location 13. There could not be any kind of randomization happening there. Why is that because when I insert it, I may be mapped to location 10. When I try to retrieve or search for it then if it gets mapped to location 13, I would not know the location of the key. It should map equal keys to the same indices and of course and we should try to minimize the probability of collisions.

Let us look at the popular hash-code maps. The hash-code map is the part which converts your key to an integer. One thing is that we could just take anything as the bit pattern and interpret it as an integer. If you have a numeric type of 32 bits or less, we can reinterpret the bits of the number as an integer. Your key which has more than 32 bits in it which is a long or a double real number which takes more than 4 bytes, then you can take it in chunks of 32 bits and add them up.

(Refer Slide Time: 06:14)



Take the first 4 bytes and add the next 4 bytes to it and so on to get eventually some 32 bit and that could be an integer you are working with. Such a kind of tree could also be used to compute the hash code map of a string. Suppose I was using the key as your name. Given a particular name, let us say Ankur I want to convert it to an integer. One possibility would be take the ASCII code of A, N, K, U, R add them up and that I will interpret as an integer.

Why is this a bad strategy? Why would the number of collisions be high? Why would the sum of 2 different names be the same? Only if the order is different and that happens for many different words. It is not the case for the names, but many words in the English dictionary would be obtained from the same letters.

If you have 2 words such that the letters was same as g o d and d o g, then when you sum up the ASCII values they will be going to the same location only. We have to avoid such a kind of things. Even if the words were not the same but A was replaced by B and N was replaced by M even then we will end up with the same. These are all the reasons for why it is not a great strategy. Especially when you are trying to convert character strings in to an integer.

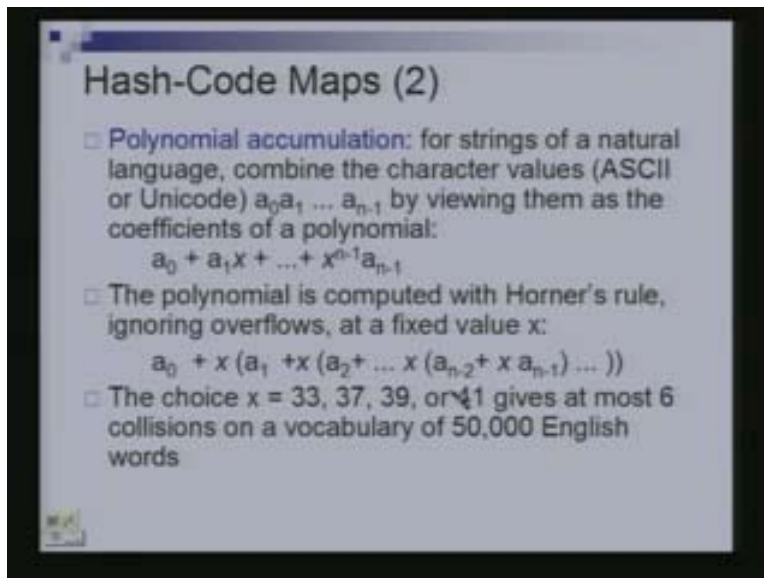
One technique used in such settings is called as polynomial accumulation. You have a certain string and  $a_0$  is the ASCII code for the 1<sup>st</sup> character of the string and  $a_1$  is the ASCII code for the 2<sup>nd</sup> character and so on. You are going to think of it as a polynomial whose coefficients are  $a_0, a_1$  up to  $a_{n-1}$ .

$$a_0 + a_1X + \dots + X^{n-1}a_{n-1} \quad a_0 + X(a_1 + X(a_2 + \dots X(a_{n-2} + Xa_{n-1}) \dots))$$

The above given expression is your polynomial and you are going to evaluate this polynomial at a certain value of x. The evaluated value is going to be the integer corresponding to this  $(a_0a_1\dots a_{n-1})$  string. That integer might be from a large range then we will use the compression map to map it to the table. But 1<sup>st</sup> we are looking at the hash code map were in we are trying to convert a string or a non-integer data in to an integer. We are looking at the setting where the string we have is this  $(a_0a_1\dots a_{n-1})$  and we are trying to convert it to an integer. Evaluate the below given polynomial at some integer value.

$$a_0 + X(a_1 + X(a_2 + \dots X(a_{n-2} + Xa_{n-1}) \dots))$$

(Refer Slide Time: 09:24)

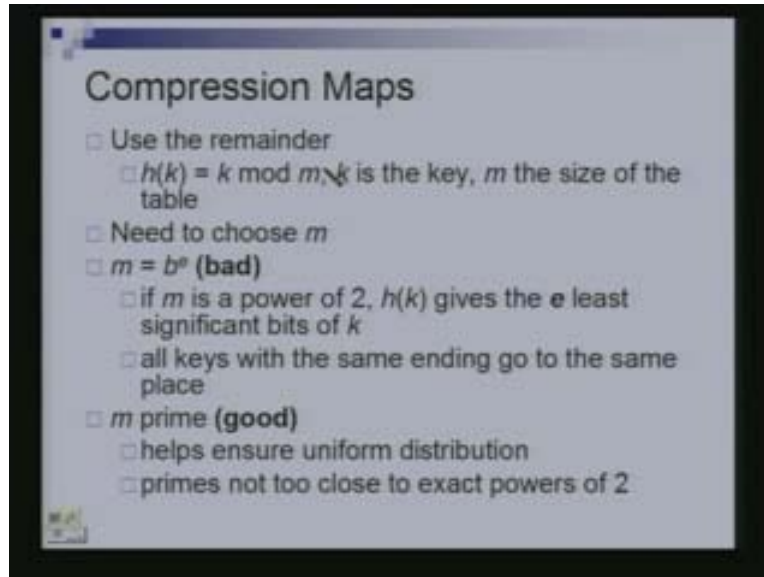


The value of x has been the experimental stuff, people have looked at and found that if you work with  $(x = 33, 37, 39 \text{ or } 41)$  these values and if you take an English dictionary with about 50, 000 words in it and use this technique to convert your words in to integer. Then you will not get too many collisions. At a particular time you will have at most 6



collisions. There is no theory behind it, this has been observed experimentally. This is an experimental study in favour of this kind of a hash code map.

(Refer Slide Time: 11:31)



Let us look at some compression map. Given an integer you have to map it to the small range of your table. One natural thing would be that  $k$  is your integer and your table is of size let us say little  $m$ . Just do  $k \bmod m$  and  $k \bmod m$  will give you some integer in the range 0 through  $m-1$  where  $k$  is the key and  $m$  is the size of the table.

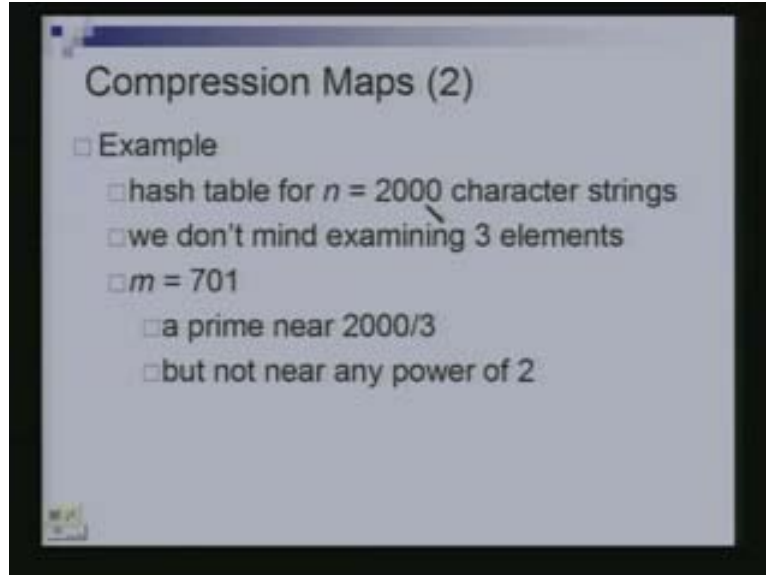
Suppose you were to choose your  $m$  and let us say your table is of size 1024,  $m$  is basically  $2^{10}$ . When I am taking some integer mod  $2^{10}$  then essentially that means I am taking the last 10 bits of that integer. Write the integer in its binary representation and then when I am taking mod 2 that means I am taking the last bit of the integer. If it is 0 then I get 0 always, if it is 1 I get 1. If I am taking mod 4, I am getting the last 2 bits. So if I am taking mod  $2^{10}$  then I am getting the last 10 bits.

All the integers which have the same last 10 bits would get mapped to the same location. This is bad because we are forgetting the other bits of the integer. We are just taking some small set of bits that is the last 10 bits based on the hash function. Hence one should not do such a thing.

In this case if you are using the simple compression map then you should not pick up the size of your hash table to be some power of 2. In fact it helps, if you take the size of the hash table to be a prime number.



(Refer Slide Time: 13:24)

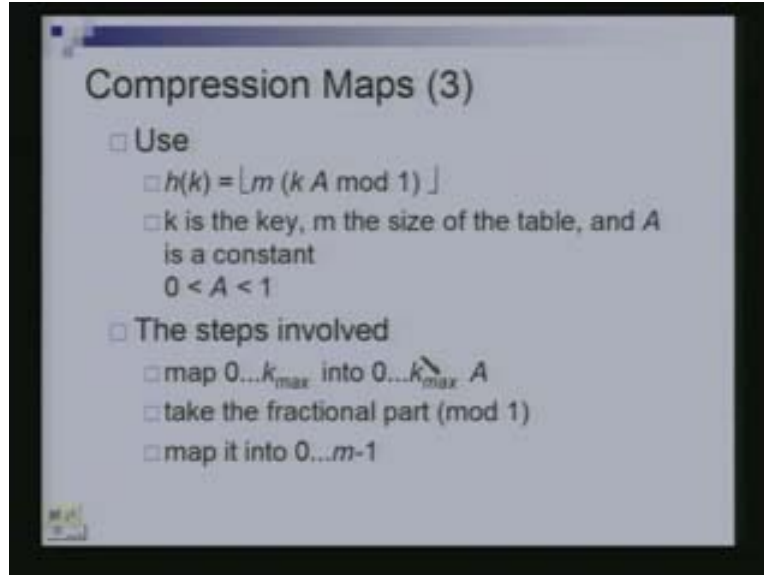


Let us look at an example. Suppose I had 2000 strings and I am trying to put it in hash table. I will try to pick the size of my hash table let us say at 701 which is the prime number. This will ensure that on an average I would see only 3 strings per location that is  $701 \times 3$  is roughly 2000. In my chaining, I would have 3 as the length of the linked list.

One important thing is that one should not pick up the size of the hash table close to a power of 2, because the same kind of effect will start happening when you have the size of the hash table to be exactly the power of 2. If you are going to use that kind of a compression map which is just  $\text{key} \bmod m$ , then keep in mind that  $m$  should not be a power of 2 or even close to a power of 2 and preferably it should be a prime number.

Things do not work when you see a lot of collisions happening. Lot of it depends upon the data and the keys you are trying to insert in to your hash table. These are generic principles which if you follow will improve the performance. There have been instances in which we did some experiment where it is better to take a number which is not necessarily a prime.

(Refer Slide Time: 15:35)



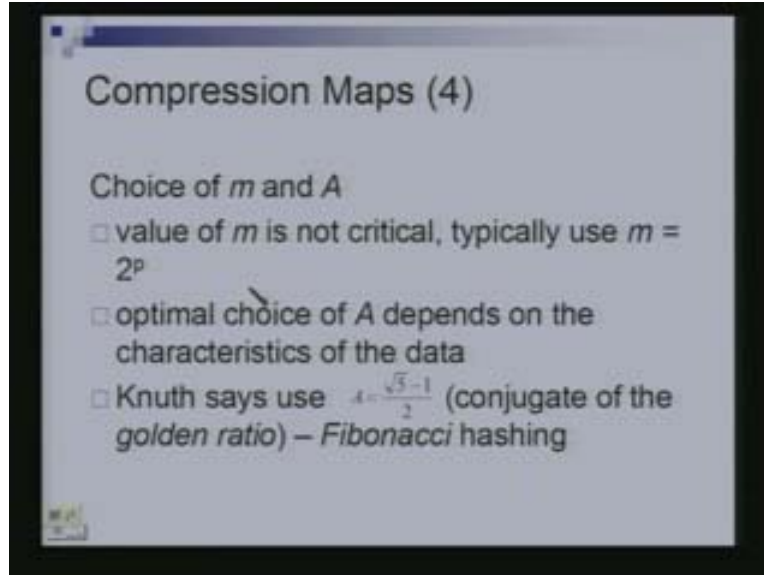
What are the other kinds of compression maps? There is other compression map you can use, essentially first I read out the 2<sup>nd</sup> part of the above slide. Suppose your keys are in the range of 0 through  $k_{\max}$ , recall now assuming that our keys are integers because we first used the hash code map to convert anything that was non-integral in to an integer. The keys are in the range 0 through  $k_{\max}$ , so first convert them from this range ( $0 \dots k_{\max}$ ) in to a range through  $k_{\max}$  times  $A$ .

Essentially we multiply each key with  $A$  where  $A$  is some number between 0 and 1. First we converted to this range ( $0 \dots k_{\max} A$ ). Now we take the fractional part of the each key that corresponds to  $kA \bmod 1$ . As a consequence we get a number between 0 and 1 because we took the fractional part. We have to map it in to the range 0 through  $m-1$  so I can just multiply that number I get between 0 and 1 by  $m$ . This number ( $kA \bmod 1$ ) was between 0 and 1 and when I multiply by  $m$  I get fractional number. That is why I took the floor function which means round down. Thus I rounded that number down to the nearest integer.

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

I will repeat it again. You first took a key and multiplied by  $A$  where  $A$  is some number between 0 and 1. Then from that you took the fractional part of that number which is again something between 0 and 1 and then you rounded it down. This is another popular compression map. You could have done something different, for instance I could just take this ( $0 \dots k_{\max} A$ ) and map it to ( $0 \dots m-1$ ) directly. Although it is not clear about how would you do it perhaps divide by  $m$  or some other thing. This is one of the popular ways of doing things.

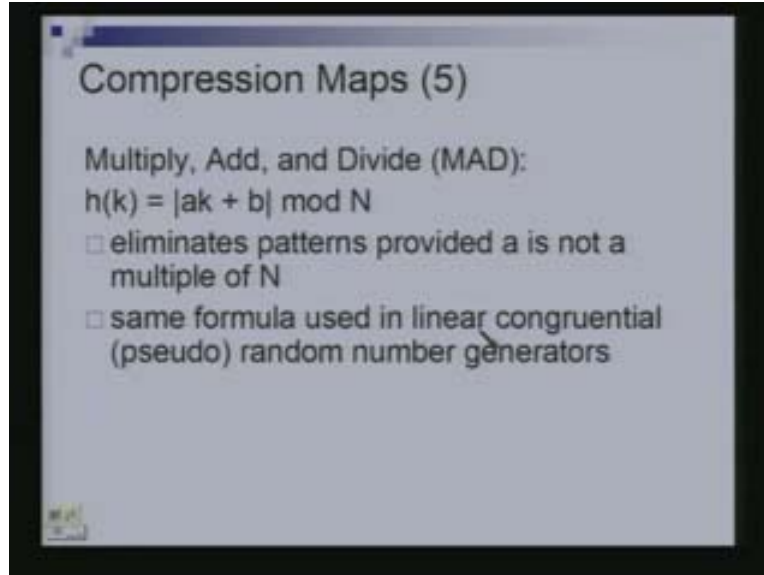
(Refer Slide Time: 17:41)



In the following case the choice of  $m$  is not critical. Even if  $m$  was the power of 2 now, the same kind of thing that happened before would not happen because we have done a lot of jugglery. We have taken that number, first we multiplied it by  $A$  which was a small fraction then we took the smaller fraction part and then plotted it to the range 0 through  $m$ . Here it is not critical that  $m$  not be a power of 2, we could use  $m$  as  $2^p$ . Some evidence if we use  $A$  as something like  $\frac{\sqrt{5}-1}{2}$  then it turns out to be good. If we use that value of  $A$  then it is called Fibonacci hashing.

Most of this is experimental without significant theory behind it. So if you might want to read more about hash function there is a nice book by Ronald Knuth on sorting and searching which covers hash functions in detail.

(Refer Slide Time: 18:51)

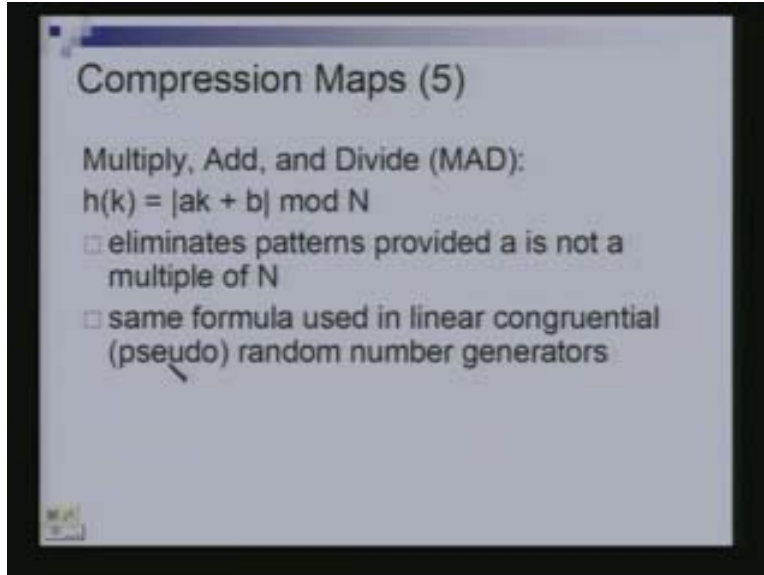


There is another technique for a compression map called the Multiply, Add, and Divide which says the following, take your key multiply it by a and add b. Thus a and b are 2 fixed numbers. Then compute modulo N where N is the size of your hash table, sometimes I use m and sometimes N. The first technique was just  $k \text{ mod } N$  but now we are doing something different. We are multiplying by a and adding b.

Here a should not be a multiple of N. If a were a multiple of N then  $a \text{ mod } N$  will be 0, so  $ak \text{ mod } N$  is also 0. For any key you will always get mapped to the same location b. In fact a and N should be co-prime if possible to avoid any kind of patterns happening. Such a technique is used in your random number generator also. You might have used the function random as a part of your programming. If you specify the range it gives your random number in that range.

How does it come up with a random number? Many of the random number generators are based on the technique called linear congruential generators. They start with a certain seed.

(Refer Slide Time: 20:51)

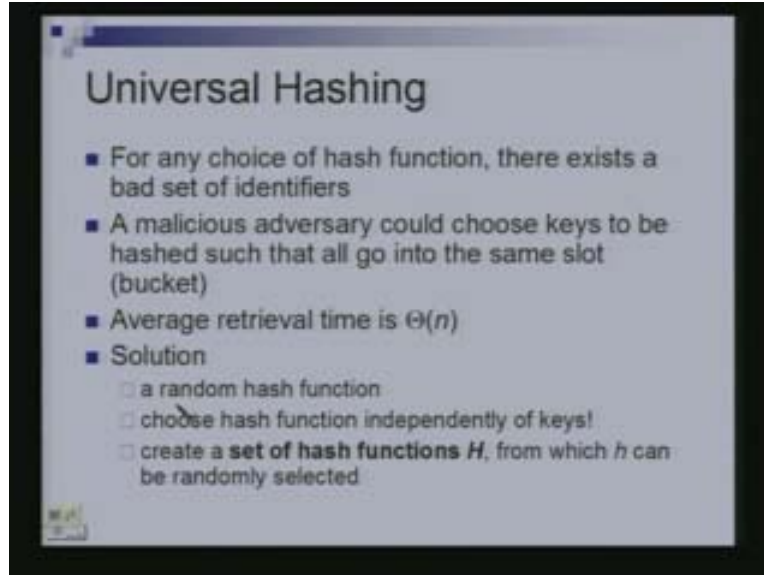


Seed is a starting value which could be user defined, you could provide what the seed is or it could be a random number generator which could just take the system time at that point or some other information and use that as a seed. That seed becomes the initial  $k$  value and then you compute this quantity ( $|ak + b| \text{ mod } N$ ) and the value you get becomes your random number.

$$h(k) = |ak + b| \text{ mod } N$$

The above function will give random number in the range 0 through  $n-1$ . Then for the next random number, you are going to use  $k$  which is the last value you return. We will use the last random number generated as a value of  $k$  and once again compute  $|ak + b| \text{ mod } N$ . You will use the value you get for the next time and so on. This is how you generate random number. Such numbers are actually called pseudo random number because they are not truly random. Once you know the seed you can actually figure out all the numbers that you get.

(Refer Slide Time: 21:39)



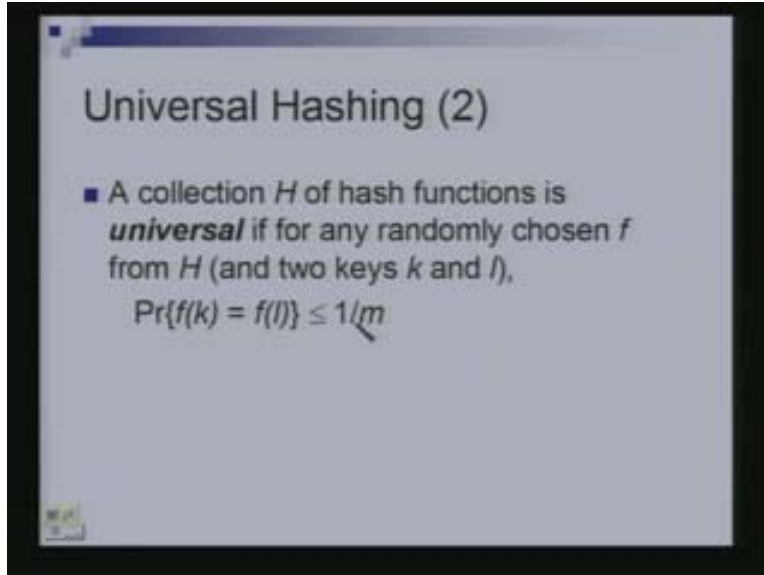
There is another technique called universal hashing which I am not going to go in much detail, I will just briefly tell you the idea. I pick up a hash function and tell you what the hash function is. You can always come up with set of keys such that all those keys using my hash function will get mapped to a very few locations.

I think of you as an adversary who is trying to make life difficult for me let us say, by picking key which all get mapped to a very few locations in the hash table so that I have to spend a lot of time doing insertion, deletion and searching.

One solution I can imply is that I do not even tell you the hash function which I am going to use. That means I am going to have a bunch of hash function let us say 15 different hash functions and before the process starts I am going to randomly pick 1 hash function out of these. Then with the keys that are given to me, I am going to use this hash function to put the keys in to the table. I have to use this same hash function for inserting all my keys, for doing the search, deletion and so on.

For one run of the hash table implementation I have to use the same hash function. I cannot change the hash function in the midway but the next time when I invoke this program, I could perhaps use a different hash function because that I have picked up randomly from my set of hash function. So even if you came up with the bad set of keys for one of my hash function, may be that is the hash function I did not pick up at all, when I was doing my implementation.

(Refer Slide Time: 23:43)



There are some results which say that you can pick up a collection of hash function and such a collection of hash functions is called universal, such that for any 2 keys the probability that they get mapped to the same location is no more than  $\frac{1}{m}$ .

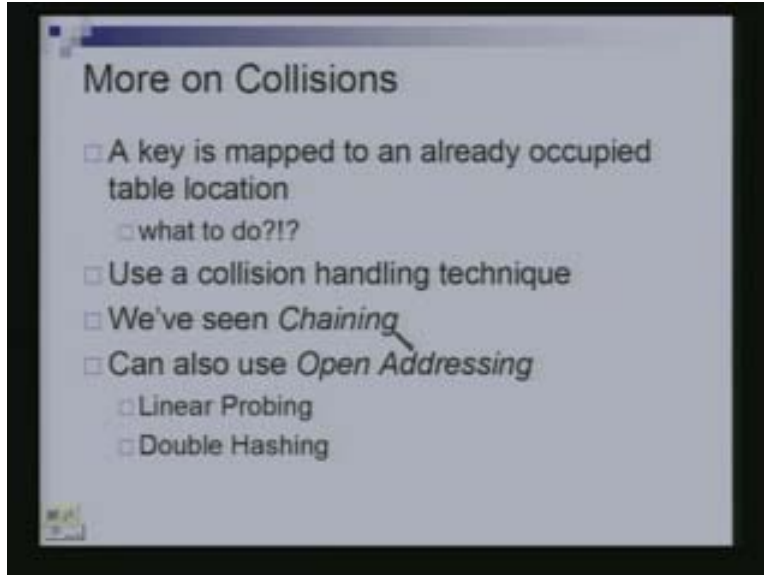
$$\Pr\{f(k) = f(l)\} \leq \frac{1}{m}$$

As I said, this is just a brief idea about the universal hashing and I am not going to see in detail. When you do your next course on algorithms in the 3<sup>rd</sup> year you will see more of universal hashing. So that is as far as the hash function is concerned. When you use hashing you will get collision, there is no way around it and one technique we saw in the last class was to resolve collisions what we call chaining.

If many keys go to the same location you just chain them up and put a linked list there. You can still do insert, search and delete by doing that operation in the linked list. You are going to see 2 other techniques today which fall under the general class of open addressing. One of these is called linear probing and the other is double hashing.

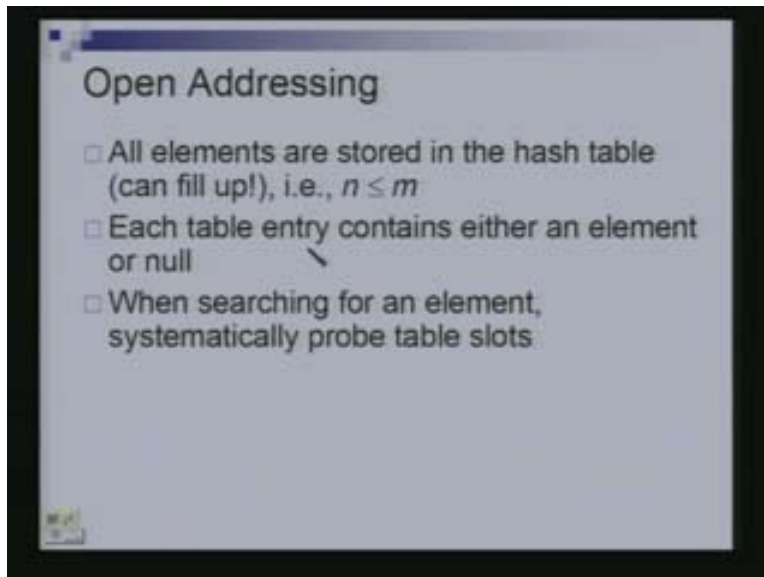


(Refer Slide Time: 24:31)



Open addressing differs from chaining in the following key fact. Recall in chaining none of these elements were actually stored in the table.

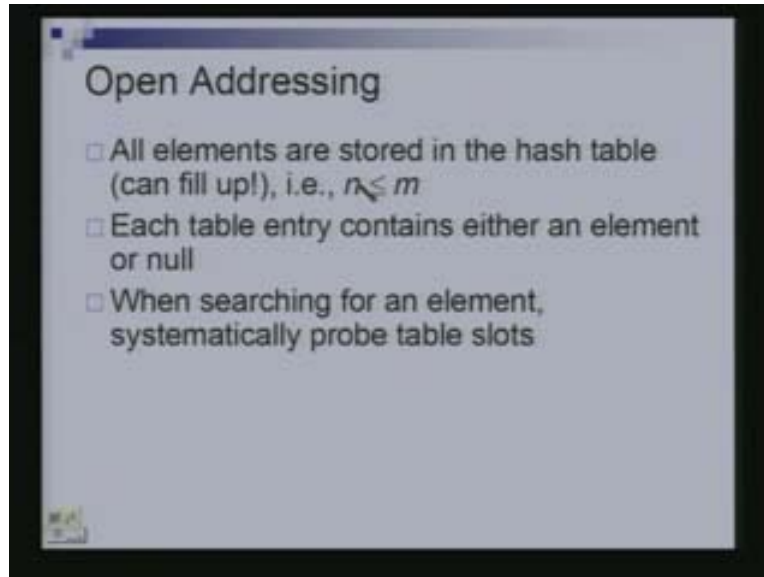
(Refer Slide Time: 25:06)



They were all stored outside the table, in the table all we had was a reference to the starting element of the linked list. The table was only storing the pointers or the references to the first element of the linked list. But now we are going to put all the elements in to the table itself. As I said hashing could map 2 elements to the same location in the table, we cannot put both of the elements to the same location. Still we want to put all the elements in the table, we will have to find some other locations for the

element. Clearly if all elements have to reside in that table, then the number of elements that we are trying to put  $n$  has to be less than the size of the table which is  $m$ .

(Refer Slide Time: 26:12)



I am going to work where  $m$  is the size of my table and  $n$  is the number of elements that I am trying to put. This was not a requirement for my chaining technique. I could have the number of elements as larger than the size of the table, because there the elements were not residing in the table. They were residing in the nodes which were a part of the linked list. Each entry of the table is now either going to contain an element or it is going to be null.

It is going to be null which means that does not have any element in it. When we are searching or inserting or deleting, we have to probe the elements of the table in a suitable manner.

We are going to think as if we are modifying the hash function a little bit. The  $U$  is the universe from which the keys are picked. Our hash function is mapping the keys, earlier this part  $\{0, 1, \dots, m-1\}$  was not there. We were mapping the keys ( $U$ ) to 0 through  $m-1$  and that would tell us where this key sets, for instance in the case of chaining. We are going to have a second parameter and when I am trying to insert the key that will be my first probe.

(Refer Slide Time: 26:59)

The slide is titled "Open Addressing (2)" and contains the following content:

- Modify hash function to take the probe number  $i$  as the second parameter

$$h: U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

- Hash function,  $h$ , determines the sequence of slots examined for a given key
- Probe sequence for a given key  $k$  given by  $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$  - a permutation of  $\langle 0, 1, \dots, m-1 \rangle$

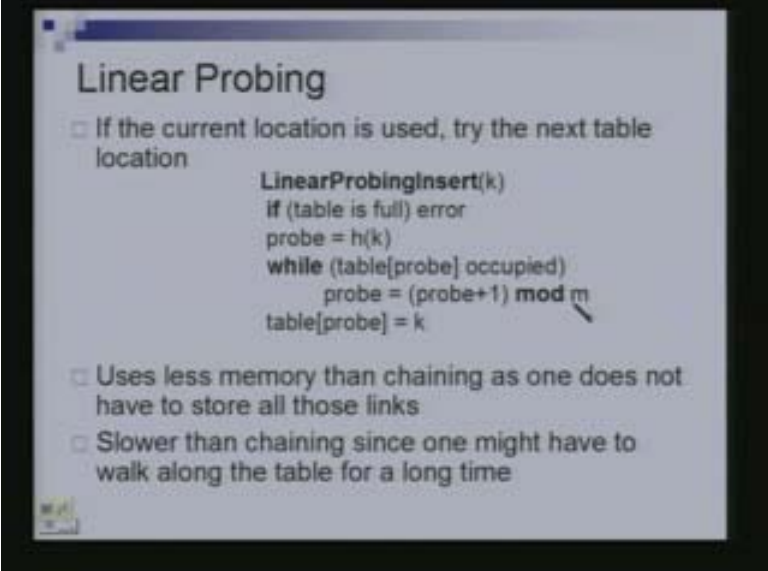
I will compute the value of the hash function for that key  $(k, 0)$  let us say for  $0^{\text{th}}$  probe and I obtained  $h(k, 0)$  as the value of my hash function. I look at the  $0^{\text{th}}$  location in the table, if that location is occupied then I have to look again. When I look up the next time I will have a value of 1 as the  $2^{\text{nd}}$  parameter.

The  $1^{\text{st}}$  parameter is still the key  $k$ . I am going to compute the value of the hash function for  $(k, 1)$  which gives some other location in the hash table and so on. I am going to different location in the hash table till I find an empty location, if the operation was one of insertion.

Depending upon the hash function we will have many different techniques. The hash function  $h$  is really determining sequence of slots which are examined for a certain key. The  $U$  was the range of the keys,  $U$  is the set which specifies the collections of keys that we have.

The number of elements we are trying to insert in to the hash table should be less than the size of the hash table. If I try to insert all the 100 students of this class to a hash table that I create then clearly the size of the hash table has to be more than 100. Because each of this student has to go to 1 location of the hash table.

(Refer Slide Time: 29:54)



**Linear Probing**

- If the current location is used, try the next table location

```
LinearProbingInsert(k)
if (table is full) error
probe = h(k)
while (table[probe] occupied)
    probe = (probe+1) mod m
table[probe] = k
```

- Uses less memory than chaining as one does not have to store all those links
- Slower than chaining since one might have to walk along the table for a long time

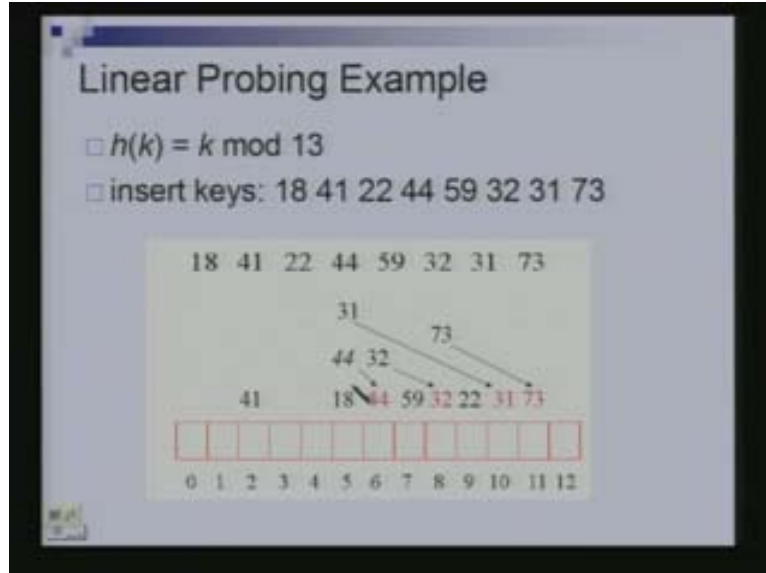
The first technique under open addressing is called linear probing. I have the key  $k$  which I am trying to insert. I have a hash function  $h$ , I compute  $h(k)$ . This  $probe = h(k)$  is the first place of the hash table that I am going to look at. If  $table[probe]$  is occupied then I just go to the next location. So  $probe$  is incremented by one and then once again I check if it is occupied.

If it is occupied then I increment again till I find an empty location and at that point I will put the element  $k$ . This is the guiding principles that if the current location is used, just go to the next location. The  $mod\ m$  is used to do rap around, if you reach the end of the table then you start at the beginning.

Your question is what happens when we retrieve the keys. We will come to that in a short while. When you are trying to insert, you compute the value of hash function and you go to a specific location as specified by the hash function for that key. If that location is occupied that is there is an element already sitting there, you go to the next location and if that is also occupied go to next location till you find the empty location.

One advantage it has over chaining is that it uses less memory. In chaining you have to keep track of references. Each of your nodes should have place for the element that it is storing. But it should also have the reference to the next node so that space is wasted. But this technique might end up slightly slower than chaining.

(Refer Slide Time: 32:19)



Let me show you an example. My hash function is  $k \bmod 13$ , a very simple hash function. My keys  $k$  are integers and I am trying to insert these keys in to the table. 13 is the size of my table and the location is from 0 to 12. The  $18 \bmod 13$  is 5, so 18 goes to location 5 because at that point the table was empty, so it can come there.  $41 \bmod 13$  is 2 so 41 goes to location 2,  $22 \bmod 13$  is 9 so 22 goes to location 9.

Till this there is no problem in inserting, as the table is empty.  $44 \bmod 13$  is 5, we want to put 44 in the 5<sup>th</sup> location. But this location is already occupied by 18, so 44 will have to search for the next location. As the 6<sup>th</sup> location is empty we put 44 there.  $59 \bmod 13$  is 7, we place 7 there as that location is empty.  $32 \bmod 13$  is 6, as 44 is sitting in 6 we go to the next location then 59 is sitting at that location, again we go to the next location and as that location is empty we put 32 there.  $31 \bmod 13$  is 5, so we should put it in 5<sup>th</sup> location but this location is occupied with 18 and the continuous locations are occupied by 44, 59, 32, and 22.

So we go to the next location which is empty and we put 31 in that location.  $73 \bmod 13$  is 8, as 8<sup>th</sup> location is already occupied we check for the next locations and we put 73 in the 11<sup>th</sup> location. All the elements are sitting in their respective position that is 41 at location 2, 18 at location 5, 44 at location 6 and so on. This also shows you one problem with this technique. The elements tend to aggregate, form clusters so you might have to go through many locations while searching for an element.

How would one search? The hash table is given in the slide below which is after inserting those elements. Suppose we are searching for key  $k$ , we are going to compute  $k \bmod 13$  because that was our hash function. Then this  $(k \bmod 13)$  is the first location we go to and after that if we do not find the element we do not say that the element was not in the table, rather we go to the next location.

(Refer Slide Time: 35:27)

The slide displays a hash table with 13 slots, indexed 0 to 12. The values are: 0: empty, 1: empty, 2: 41, 3: empty, 4: empty, 5: 18, 6: 44, 7: 59, 8: 32, 9: 22, 10: 31, 11: 73, 12: empty.

**Lookup in linear probing**

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

- To search for a key  $k$  we go to  $(k \bmod 13)$  and continue looking at successive locations till we find  $k$  or encounter an empty location.
- Successful search: To search for 31 we go to  $(31 \bmod 13) = 5$  and continue onto 6,7,8... till we find 31 at location 10
- Unsuccessful search: To search for 33 we go to  $(33 \bmod 13 = 7)$  and continue till we encounter an empty location (12)

If at the next location there is some element present then we go to the location following it and so on till we either find the element or we reach a an empty location. If we reach an empty location that means the element is not their in the table because if the element had been their in the table it would have been inserted at one of the locations that I have checked.

Let us see. Suppose I am searching for 31 so we go to  $31 \bmod 13$  which is 5. I come to the 5<sup>th</sup> location in which 31 is not there, so I go to the next location and search the element till I find it. I found the element in the 10<sup>th</sup> location. When I did not find it, I can not say that the element is not their in the table. It could be their, infact it is their.

Suppose I am searching for  $33 \bmod 13$  which is 7, I would start searching it from the 7<sup>th</sup> location. Till 11<sup>th</sup> location the element is not present and the 12<sup>th</sup> location is empty. This means 33 could not be their at all in this table. Because if 33 had been there in the table, then by this time it would have been definitely inserted in to the table till the 12<sup>th</sup> position because this is an empty location.

(Refer Slide Time: 36:32)

**Lookup in linear probing**

		41				18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12	

- To search for a key  $k$  we go to  $(k \bmod 13)$  and continue looking at successive locations till we find  $k$  or encounter an empty location.
- Successful search: To search for 31 we go to  $(31 \bmod 13) = 5$  and continue onto 6,7,8... till we find 31 at location 10
- Unsuccessful search: To search for 33 we go to  $(33 \bmod 13 = 7)$  and continue till we encounter an empty location (12)

That is an unsuccessful search, in an unsuccessful search the search terminates when you reach an empty location but a successful search will terminate when it finds the element. How do you delete? The following slide is my picture which is from the previous slide and I want to delete 32.

(Refer Slide Time: 38:15)

**Deletion in Linear Probing**

		41				18	44	59	22	31	73		
0	1	2	3	4	5	6	7	8	9	10	11	12	

- To delete key 32 we first search for 32.
- 32 is found in location 8. Suppose we set this location to null.

First I have to search for 32,  $32 \bmod 26$  is 6. I come to the 6<sup>th</sup> location, it is not there. Then I go to the next location, also it is not there. Then I find the element 32 in the 8<sup>th</sup> location. Suppose I removed it by setting this location to null. I removed 32 from that location.



Is this a good idea? No. Why this is not a good idea? Suppose now you search for 31. The  $31 \bmod 13$  is 5, so we come to the 5<sup>th</sup> location. But we did not find it there. Then we go to the next location for that element and we did not find it and at last we reached the empty location. Hence we will say that 31 is not their but still 31 is their. Why is a problem coming in?

Because when 31 was inserted that was the full location. That is why 31 was inserted later, but if you delete the element in the 8<sup>th</sup> location then you have a problem. Some how we have to do something different because we cannot just set this location to null or we cannot mark this location empty also. Look up will declare that 31 is not present, which is wrong. How do we delete? Instead of setting this 8<sup>th</sup> location to null we will place a tombstone, actually an x.

(Refer Slide Time: 39:49)

The slide shows a hash table with 13 slots (indices 0-12). The values are: 41 at index 2, 18 at index 5, 44 at index 6, 59 at index 7, X at index 8, 22 at index 9, 31 at index 10, and 73 at index 11. Below the table is a list of operations:

- Instead of setting location 8 to null place a tombstone (a marker) there.
- When lookup encounters a tombstone it ignores it and continues with next location.
- If Insert comes across a tombstone it puts the element at that location and removes the tombstone.
- Too many tombstones degrades lookup performance.
- Rehash if there are too many tombstones.

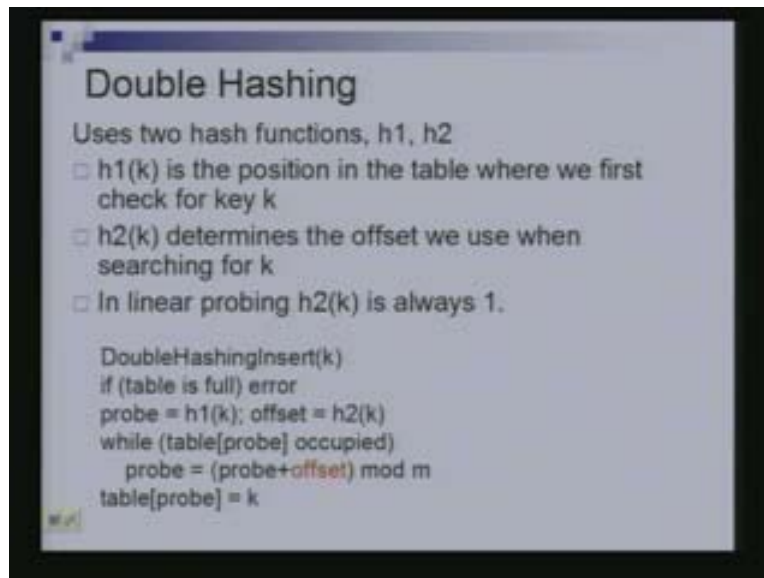
Tombstone is just a marker so you could set up a bit at that location which specifies that this location was occupied by some one. It is not always the case that this will be an empty location, at some point this was occupied by some one. How it will help us? When we are doing a look up and we encounter a tombstone, we do not declare that the search is ended and the element is not present but we continue. As before if I was searching for 31,  $31 \bmod 13$  is 5 so I would come to location 5 and go to the next location and at the 8<sup>th</sup> location I would see an x and not null which is a tombstone.

So I continue till I find either a null location or 31. I found 31 and declare 31 is their. When a look up encounters a tombstone it ignores and continues. When an insert encounters a tombstone what does it do? It will put the element at that position. We have to reclaim this space. What happens if there are too many tombstones? You do not have elements in the table, those are actually empty locations but in your search you still have to go beyond them. The performance of your search degrades.

If you have a lot of tombstones you should just rehash. Just remove all the elements and put them back again. The same kind of a technique you have to do when you grow the table. Now you are not growing the table, you have too many markers in the table so just do a rehash and that will create empty slots without the tombstones and your performance will increase again.

I will come to the other open addressing techniques. We looked at linear probing, we compute the hash function we look at that location and next location and so on. In double hashing we have 2 hash functions  $h_1$  and  $h_2$ .

(Refer Slide Time: 42:31)

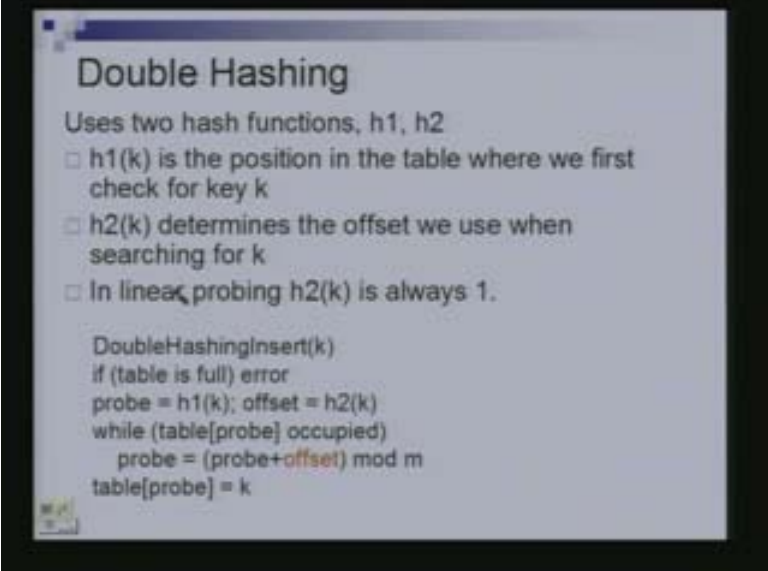


The value of  $h_1$  gives me the first position where I am going to look for the key  $k$ . Then  $h_2(k)$  will tell me the offset from the first position where I am going to look again for the key  $k$ .

Let us look at the piece of code given in the above slide. Probe is set to  $h_1(k)$ , so that is the first position I look at and offset is set to  $h_2(k)$ . First I will look at the locations specified by probe and the table, if it is occupied then the next location I will look at is  $probe + offset$ . Probe is set to  $probe + offset$  which means this is a next location I look at. If this is also occupied then the next location I will look is  $probe + offset + offset$  which means offset is determining key with how much distance I am going to advance.

Every time I do not see the element that I am searching for. For linear probing your offset is always 1. You were always going to the next location so that corresponds to an offset of 1. Instead of going to next location I jumped one location ahead that is I jumped 2 locations, then offset would have been 2 and so on. Offset in this case which is in the orange color in the slide below is determined by the hash function  $h_2(k)$ . This offset could be different for different keys.

(Refer Slide Time: 44:20)



**Double Hashing**

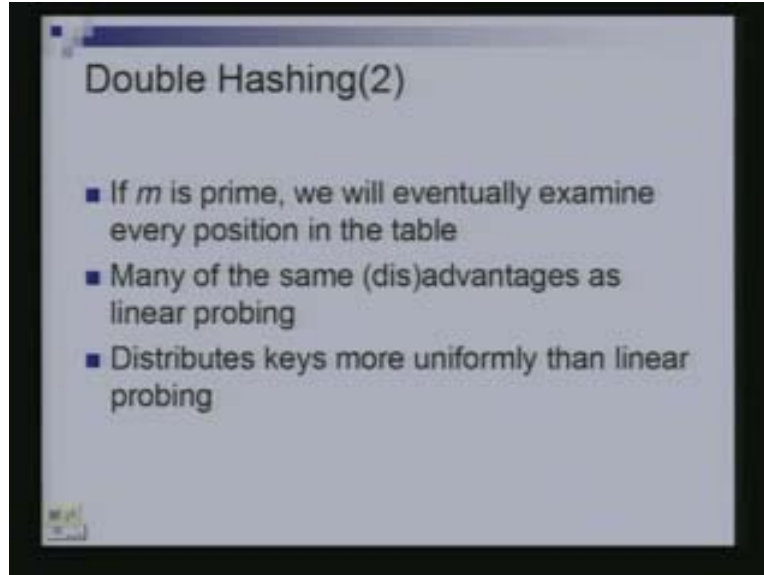
- Uses two hash functions,  $h_1$ ,  $h_2$
- $h_1(k)$  is the position in the table where we first check for key  $k$
- $h_2(k)$  determines the offset we use when searching for  $k$
- In linear probing  $h_2(k)$  is always 1.

```
DoubleHashingInsert(k)
if (table is full) error
probe = h1(k); offset = h2(k)
while (table[probe] occupied)
    probe = (probe+offset) mod m
table[probe] = k
```

We will look at an example of how double hashing works. If  $m$  is the prime then this technique will ensure that we look at all the locations of the table. In linear probing because the offset was one we would look at all the locations in the table. If there was an empty location you would always be able to insert the element.

We would not like the following to happen. There are empty locations in the table but you start from a certain location, since the offset is 3 you go 3 units ahead and you keep finding everything is full and then you come back to the starting location. Because you will not be able to insert the element at all. Maybe all of these elements that you looked at were full but the other locations in the table were empty.

(Refer Slide Time: 44:41)



Some how you do not cycle back. When you will cycle back? When your offset divides the size of the table. If the size of your table was a prime number then your offset would never divide it and this kind of a thing would never happen. In fact you would look at all the elements of the table. This is the small fact you can go back and prove that if  $m$  is prime then I have given you the rough arguments for this case, but you can also prove it more formally.

This has some of the same advantages and disadvantages as linear probing. One of it is it distributes keys more uniformly because you do not form clusters any more. These clusters were getting formed because you were just going one step at a time. If for some key you are going 7 steps ahead and for some other key you are going 13 steps ahead and for some other key you are going 2 steps ahead, then these clusters are not getting formed any more. That makes the performance better.

We will look at an example. I have 2 hash functions  $h_1$  and  $h_2$ . The  $h_1$  is the same as before,  $k \bmod 13$ . The element is also as same as before, we have a table of size 13. The  $h_2(k)$  is my 2<sup>nd</sup> hash function and is  $8 - (k \bmod 8)$ . It will always be a number between 1 and 8. It cannot be zero, because  $k \bmod 8$  lies between 0 and 7, so it is between 1 and 8.

(Refer Slide Time: 46:41)

The slide contains the following text:

**Double Hashing Example**

- $h1(k) = k \text{ mod } 13$
- $h2(k) = 8 - (k \text{ mod } 8)$
- insert keys: 18 41 22 44 59 32 31 73

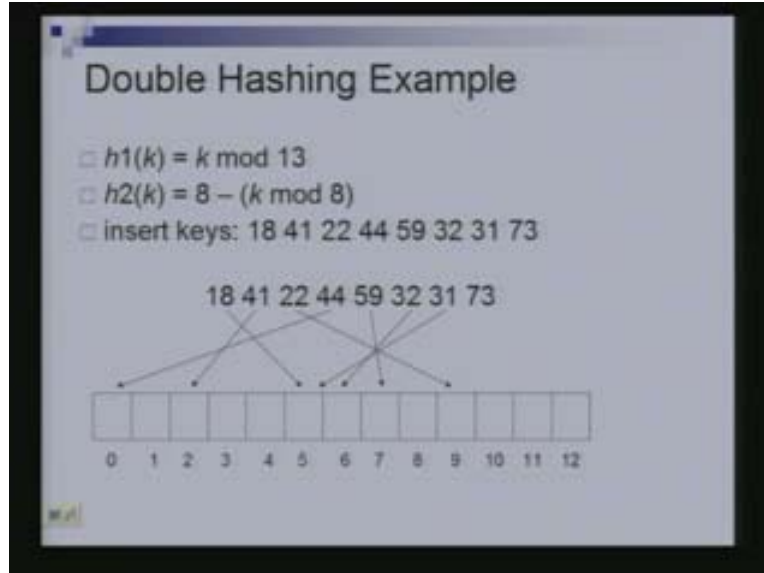
18 41 22 44 59 32 31 73

0	1	2	3	4	5	6	7	8	9	10	11	12

The zero does not make any sense, if it is zero then we are in trouble. If  $h2(k)$  is zero for some  $k$  then that means you are continuously looking at the same place and if that place were occupied then you cannot insert the element at all. Let us insert the first element 18,  $18 \text{ mod } 13$  is 5 so it will go to location 5. The  $41 \text{ mod } 13$  is 2 so it goes to location 2. The  $22 \text{ mod } 13$  is 9 so it goes to location 9. The  $44 \text{ mod } 13$  is 5 so it tries to go to location 5 but the location 5 is already occupied. We have to compute  $h2(44)$ .

What is  $h2(44)$ ?  $8 - (44 \text{ mod } 8)$ ,  $44 \text{ mod } 8$  is 4. So  $8-4$  is 4, I have to go 4 steps ahead. I will go to location 9 but that is also occupied, so I will go to location 0. That is empty so 44 will go to location 0. The  $59 \text{ mod } 13$  is 7 so 59 will go to location 7. The  $32 \text{ mod } 13$  is 6 so 32 will go to location 6. The  $31 \text{ mod } 13$  is 5 so we go to location 5 but that is occupied. I compute  $h2(31)$ ,  $31 \text{ mod } 8$  is 7 and  $8-7$  is 1. So 31 will check for the location 6 but 6 is also occupied. We have to go to 7, it is also occupied so go to 8 and this is not occupied, thus 31 go to location 8.

(Refer Slide Time: 47:39)



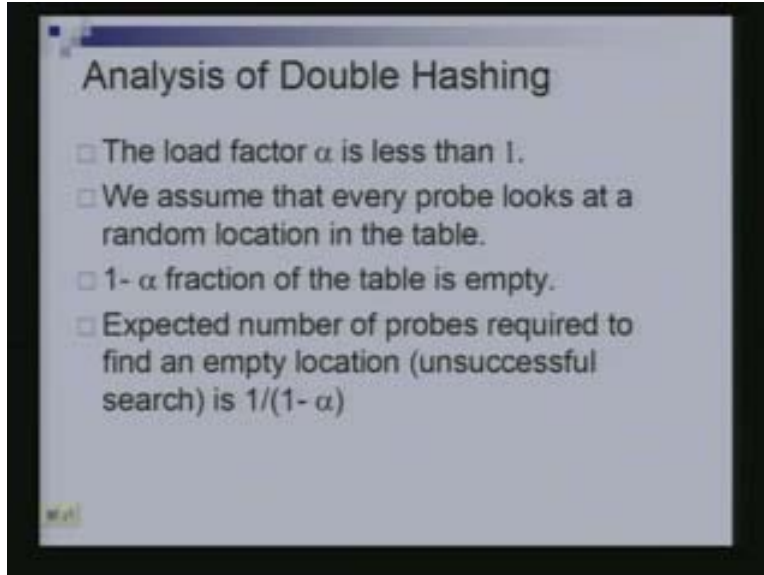
The  $73 \bmod 13$  is 8, so it will try to go to 8 that is occupied. We compute  $h2(73)$ ,  $73 \bmod 8$  is 1,  $h2(73)$  is 7 so we will go to  $8+7=15$ . The 15 is  $2 \bmod 13$ , we go to location 2 that is occupied so  $2+7$  is 9 where that is also occupied. The  $9+7$  is 16,  $16 \bmod 13$  is 3 so it goes to this location which is unoccupied. This is how the elements would be distributed in the table.

We will do some analysis of double hashing. Recall that I am going to assume that the load factor is less than one. What is the load factor? The number of elements divided by the size of the hash table  $\frac{n}{m}$  that is less than one. I need it to be less than one otherwise more than 1 does not make any sense. We are talking of a scheme where all the elements have to sit inside the hash table. We are also going to assume, this is similar to the assumptions that we made in the last class that every time I probe, I actually look at a random element in the hash table which is uniformly random.

The first time I probe I will take a random location in the hash table and try to put the element there. If it is occupied then once again I will pick a random location in the hash table and try to put it there. If that is also occupied once again I pick a random location in the hash table and try to put the element there.

Let us see how this performs, because we will only be able to analyze such a scheme. Because the other schemes are too dependent upon the hash function that we are using and we might not be able to analyze them. If  $\alpha$  is the load factor then that means  $1-\alpha$  fraction of the table is empty. If  $\alpha$  is half that means the number of elements divided by the size of the table is half. Which means only half the table is occupied and half the table is empty,  $1-\alpha$  fraction of the table is empty.

(Refer Slide Time: 50:27)

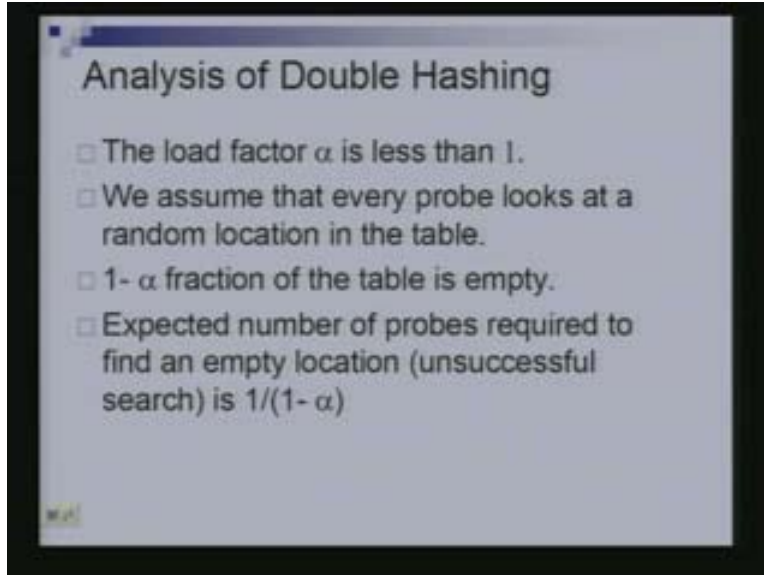


Suppose my search was an unsuccessful search. What does an unsuccessful search mean? That means the element is not in the table. When does an unsuccessful search stop? When I get an empty location. How many probes will be required before I get to an empty location?

The  $1 - \alpha$  fraction of the table is empty let say  $\frac{1}{10^{th}}$  of table is empty and 90% of the table is full. That is 10% is empty. The expected number of probes required before I hit  $\frac{1}{10^{th}}$  fraction of the table which is empty would be roughly 10. Because the first time with  $\frac{9}{10^{th}}$  probability, I will get to an occupied location and so on. So roughly after 10 trails I will hit an empty location because only  $\frac{1}{10^{th}}$  of the table is empty.



(Refer Slide Time: 51:39)



If  $1 - \alpha$  fraction of table is empty then roughly in an expected sense  $\frac{1}{1 - \alpha}$  probes are required before I hit an empty location and declare it to be an unsuccessful search. This is the expected numbers of probes required for an unsuccessful search.

Let us look at a successful search. I am going to talk about the average number of probes required for a successful search, not for one particular search but if I were to look at all the successful searches.

What are successful searches? Successful search are searches corresponding to the elements in the table. I have some number of elements in the table, let us say I search for the first element. Then how many probes are required? Suppose I search for the second element. How many probes are required and so on. Then I will take their average.

Let us try and compute this quantity. If you recall from the last class the average number of probes required for a successful search is the average number of probes required to insert those elements. Because when we are inserting those elements we are essentially doing the same thing. It is the same as the average number of probes required to insert all these elements and this is the quantity I am going to compute.

(Refer Slide Time: 53:11)

**Analysis (2)**

- Average no of probes for a successful search = average no of probes required to insert all the elements.
- To insert an element we need to find an empty location.

inserting	Avg no of probes	Total no of probes
First $m/2$	$\leq 2$	$m$
Next $m/4$	$\leq 4$	$m$
Next $m/8$	$\leq 8$	$m$

What is the average number of probes required to insert all the elements that I have in the table? When I am inserting an element I need to find an empty location again. Suppose I begin with an empty table and I am looking at the number of probes required to insert the first  $\frac{m}{2}$  elements. Size of the table is  $m$ , let us assume  $m$  is 100. I am talking of inserting the first 50 elements. Suppose I have already inserted 48, 49 elements and when I am trying to insert 50<sup>th</sup> element.

What is the expected number of probes that are required? The half of the table is empty, when I try once I may hit a full location. May be when I try again, in expectation I just need 2 probes to be able to insert this 50<sup>th</sup> element. For the other first 49 elements I might on an average even required less, but all I can say for sure that the average number of probes required for inserting these elements is  $\leq 2$ .

(Refer Slide Time: 53:51)

**Analysis (2)**

- Average no of probes for a successful search = average no of probes required to insert all the elements.
- To insert an element we need to find an empty location.

inserting	Avg no of probes	Total no of probes
First $m/2$	$\leq 2$	$m$
Next $m/4$	$\leq 4$	$m$
Next $m/8$	$\leq 8$	$m$

How many elements am I inserting? The  $\frac{m}{2}$  elements, on an average the total number of probes required is  $\leq m$  for these  $\frac{m}{2}$  elements. When I show you the rest, you will understand why I am doing this way. Suppose I have already inserted  $\frac{m}{2}$  elements in to my table and I am trying to insert the next  $\frac{m}{4}$  elements in to my table. When I am trying to insert the next  $\frac{m}{4}$  elements, just assume that I have already inserted  $\frac{m}{4} - 1$  and I am trying to insert this last element.

How much of the table is already full when I try to insert this last element? The  $\frac{3}{4}$  of table is already full. Only a  $\frac{1}{4}$  of the table is empty. So on an average I am going to require about 4 probes before I get to one of the empty location. I am searching for an empty location to put this element in. I need roughly 4 probes, infact I am just **praising** this as an upper bound and I need at most 4 probes to insert all of these  $\frac{m}{4}$  elements. The total number of probes required to insert these  $\frac{m}{4}$  elements is  $\frac{m}{4}$  times 4 which is no more than  $m$ .

(Refer Slide Time: 53:14)

**Analysis (2)**

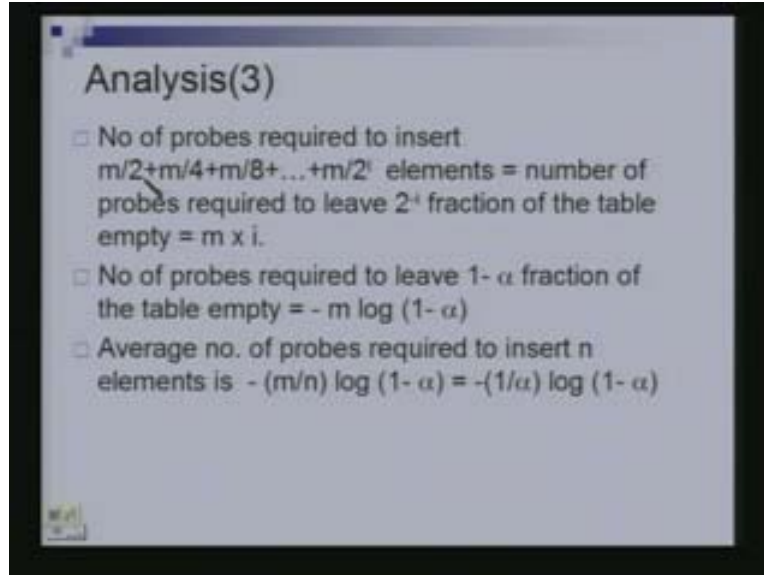
- Average no of probes for a successful search = average no of probes required to insert all the elements.
- To insert an element we need to find an empty location.

inserting	Avg no of probes	Total no of probes
First $m/2$	$\leq 2$	$m$
Next $m/4$	$\leq 4$	$m$
Next $m/8$	$\leq 8$	$m$

Similarly for these next  $\frac{n}{8}$  elements, when I am trying to insert the last of these  $\frac{n}{8}$  elements only  $\frac{1}{8^{th}}$  of the table is empty. On an average I require about 8 probes before I can get to one of those empty locations. For these  $\frac{n}{8}$  elements or for any one of them I would not have required more than 8 probes. I would have required between 4 and 8 probes for these  $\frac{n}{8}$  elements.

Because when I was inserting the first of these  $\frac{n}{8}$  elements only 3 quarters of the table was full. One quarter of it was empty, but I am just upper bounding it. I am just saying that no more than 8.

(Refer Slide Time: 57:19)



What is the total number of probes required? For  $\frac{m}{2}$  recall from previous slide I said  $m$ , for  $\frac{m}{4}$  this also I said  $m$ . What is the total required for these  $\frac{m}{2} + \frac{m}{4} + \frac{m}{8} + \dots + \frac{m}{2^i}$  elements? It is  $m$  times  $i$  ( $m \times i$ ). How many locations are empty in the table? What is the total number of elements in the table now?

After I inserted  $\frac{m}{2}$  elements what fraction of the table was empty? It is half. After I inserted  $\frac{m}{2} + \frac{m}{4}$  how much of the table was empty? It is  $\frac{1}{4}$ , so it is really this last number.

After I inserted  $\frac{m}{8}$  how much was empty? It is  $\frac{1}{8}$ . So after I inserted all of this that is

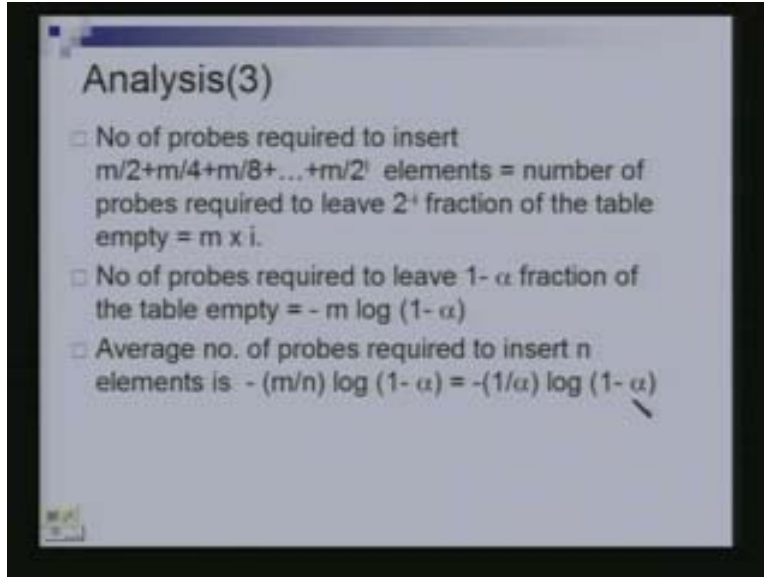
$\frac{m}{2^i}$  how much is empty?

It is  $\frac{1}{2^i}$  which is  $2^{-i}$  fraction that was empty. After I have inserted all of these fractions I

have only  $\frac{1}{2^i}$  fraction of the table empty and the total number of probes required to insert these elements is  $m$  times  $i$ . We have a load factor of  $\alpha$ , we already inserted enough elements so that the load factor is  $\alpha$ . When the load factor is  $\alpha$ ,  $1 - \alpha$  fraction of the table is empty. If I have  $1 - \alpha$  fraction of table empty, then how many probes are required? If I have  $2^{-i}$  fraction of the table empty then I require  $m \times i$  probe. What is  $i$ ?

The  $i$  is basically minus log of this ( $2^{-i}$ ) quantity. If I need to have  $1-\alpha$  fraction empty, so I just need  $-m \log(1-\alpha)$ . These are the numbers of probes required.

(Refer Slide Time: 58:17)



If I have  $2^{-i}$  fraction empty,  $2^{-i}$  is the number smaller than one. So to get to this point I require  $m \times i$  probes. So to get to a point where  $1-\alpha$  fraction was empty, I need  $-m \log(1-\alpha)$  this many probes. The above what we saw was the total number of probes required and the average was just divided by  $n$  that is  $-(\frac{1}{\alpha}) \log(1-\alpha)$ .

We will be able to capture it to a table. For an unsuccessful and successful probes, when we had chaining it was  $1+\alpha$ . For probing, for an unsuccessful search it was  $(\frac{1}{1-\alpha})$  and for a successful search what I just showed you is  $(\frac{1}{\alpha} \ln \frac{1}{1-\alpha})$ .

(Refer Slide Time: 01:00:14)

### Expected Number of Probes

- Load factor  $\alpha < 1$  for probing
- Analysis of probing uses *uniform hashing* assumption – any permutation is equally likely
  - What about linear probing and double hashing?

	unsuccessful	successful
chaining	$O(1 + \alpha)$	$O(1 + \alpha)$
probing	$O\left(\frac{1}{1 - \alpha}\right)$	$O\left(\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}\right)$

The last slide which shows how this performances of  $\alpha$  changes.

(Refer Slide Time: 01:00:38)

