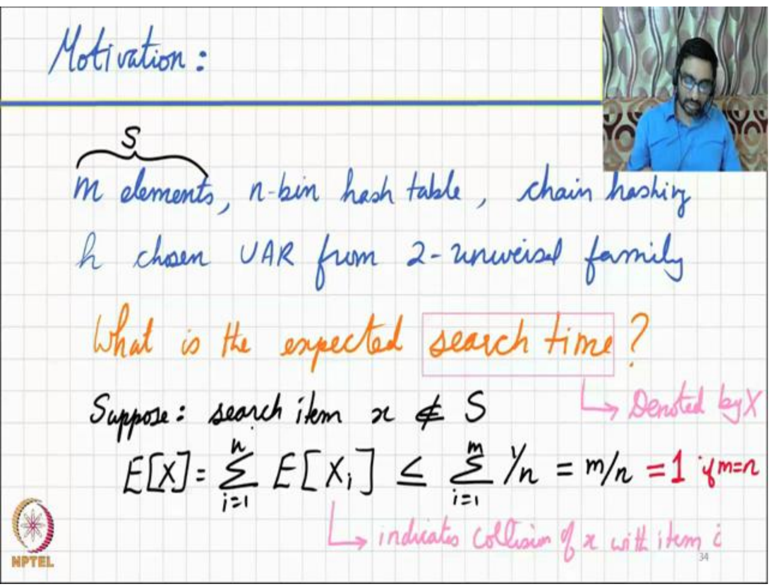
Algorithms for Big Data
Prof. John Ebenezer Augustine
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 30
Perfect Hashing

In this segment, we are going to talk about Perfect Hashing, Using 2-universal families of hash functions.

(Refer Slide Time: 00:22)

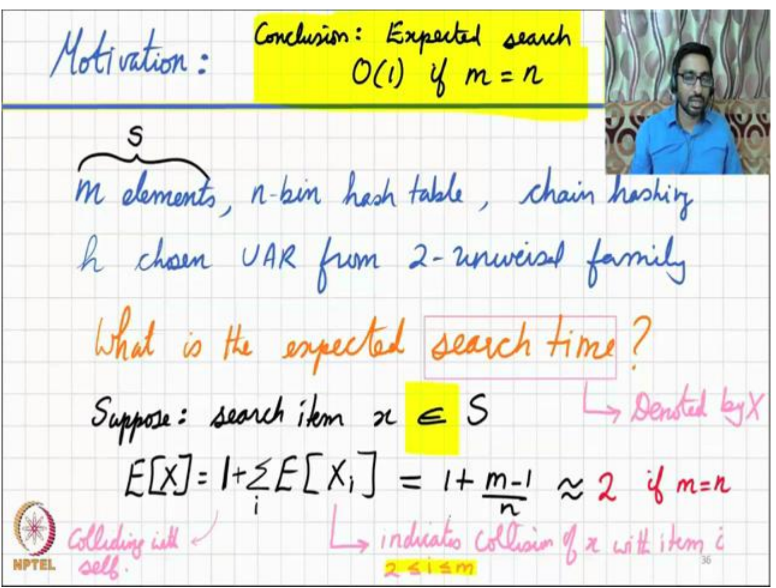


Let us begin some motivation. Let us say we have m elements and n bin hash table. Let us say we are performing chain hashing and let us assume the hash function h is chosen uniformly at random from at two universal family fashions. The question is what is the expected search time? Let us assume that. Let us start with the case that X is not in the dictionary. So, S is the set of items in the dictionary and let assume what else even happens when X is not in the dictionary.

The search item, let us denote the search time by the random variable X. E of X is going to be the summation over 1 to n, E i is an indicator random variable that denotes the collision of X, where the search time X with the item i in the dictionary and just to be clear, the summation must go from i equal to 1 to M and this summation is at most summation i equal to 1 to m 1 over n because we have assumed that we have drawn from the random from a 2-universal family of hash functions. That is at most m over n and

when we limit ourselves to the case where m equals n, then the expected search time is only 1.

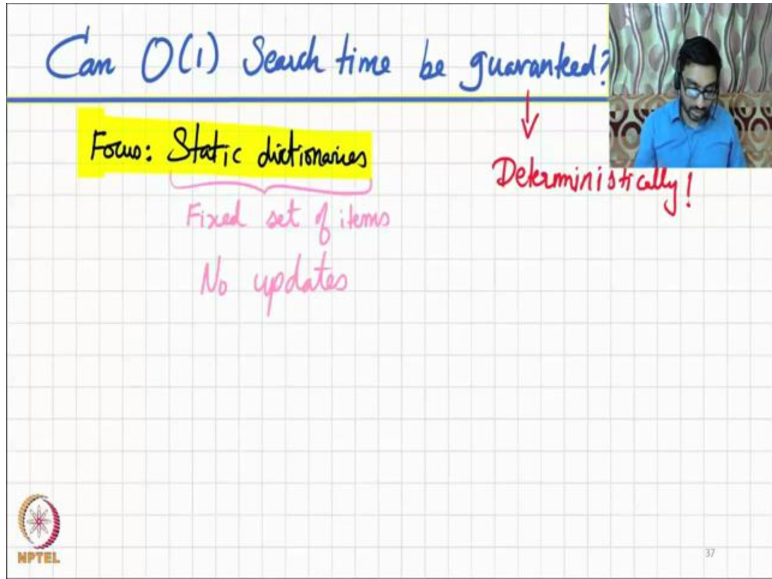
(Refer Slide Time: 02:07)



Now, what about the case where the search item X is not an s, where E can similarly bound the expectation on X, but now we need to be very careful because X is in the dictionary. We cannot sum, I mean we cannot use, indicate a random variable for the item X in the dictionary because X will collide with itself with probability 1.

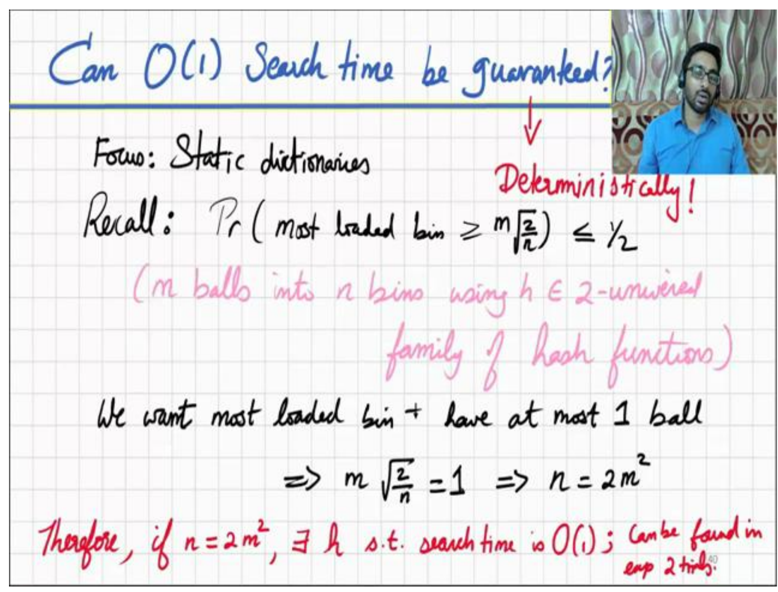
So, we are going to isolate that by a separate 1 and for the remaining cases, we are going to use X i, but this really does not affect us much because when we apply the same way of evaluating e of X i s, we get 1 plus m minus n over m which is approximately 2 when m equal to n and important conclusion we can make out of this exercise is that the expected search time is only O of 1 constant when m equal to n and the idea of perfect hashing is motivated from this context, but one of be a little bit more at our expectation instead of requiring an expected search time of O of 1.

(Refer Slide Time: 03:47)



We are interested in a deterministic search time of O of 1 can achieve this requirement and we are going to live it our focus to static dictionaries in which we are going to store a fixed set of items. This is going to be no updates to these items and for this fixed set, we will not be able to query whether an item x in the universe is present in the dictionary or not.

(Refer Slide Time: 04:19)



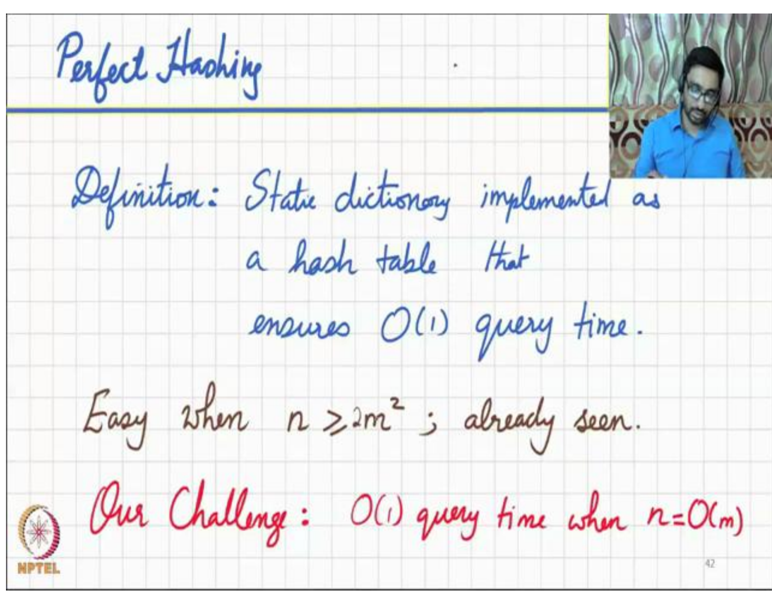
The important thing is that we are not even going to require complete randomness. We are going to assume that the hash function we have available shows uniformly at random

from a two universal family of hash functions. Before we see let us recall the balls and bins result that we have. We have m balls and n bins and these balls are hash into bins using the hash function that shows uniformly random from a family of hash functions. Then, we know that the probability that the most loaded bin is more than m times square root of 2 over n is at most a half.

So, if we want constant query time, we have to ensure that this m times 2 square root of 2 over n is 1 or lesser and this will mean that n has to be at least two times m square and when we set n equal to 2 m square, the probability with which we get a collision with which we get a search time of 1 is at most a half.

Suppose, we pick a hash function h from uniformly at random from this family of hash function and we hash each item in the dictionary to location using this hash function with probability at least half where going to get no collisions at all which means that if we tried to construct such static dictionary with probability half, we will succeed in getting a perfect hashing where in the search time is at most O of n and this is simple. But the trial if we want to insist on success, we simply repeat and we get success. So, with expected two times we will be able to get a static dictionary in which n items are mapped into 2 m square root locations and the query item for any item is at most 1.

(Refer Slide Time: 07:03)

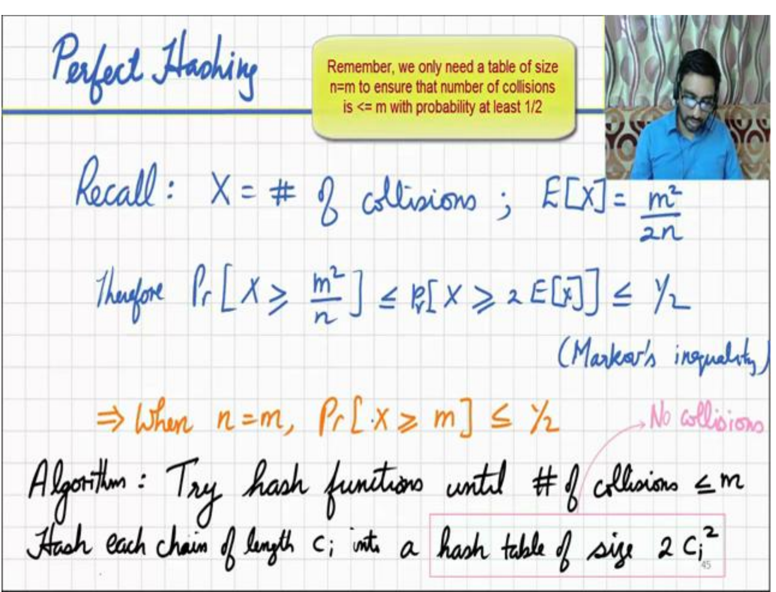


So, all of this discussion motivates what we call perfect hashing. So, we want to store a static dictionary and we want to have that implemented as a hash table and the key

requirement is that our query time in the words case must be O of 1 and as we have just seen if we are allowed to use a lot of ha, i mean much larger tables. So, if we are allowed to use a table of length at least 2 m square, then this problem is already solved and that is easy. You just repeatedly attempt constructing the static dictionary using the straight forward hashing technique until we get this requirement.

Our question now is a little bit more challenging. How do we construct such a hash table using only little space that is proportional to the number of items and still want query time to be constant not with high probability, but deterministically.

(Refer Slide Time: 08:15)

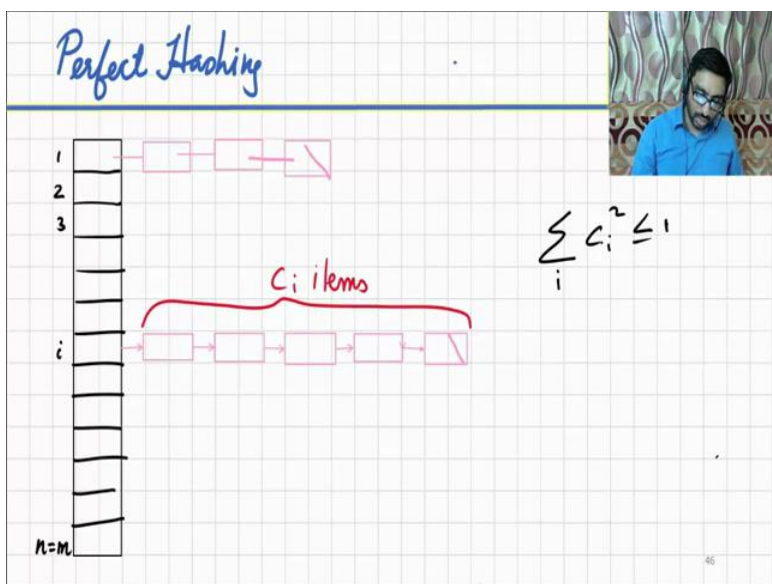


Let us remind ourselves in the previous segment that the number of collision exceeds m square by n with probability at most a half and when n equal to m, the number of collisions will exceed m with probability at most a half. So, here is an idea for how to construct this perfect hashing. We try hash functions until the number of collisions is at most m and of course, we can by the probabilistic method, we can keep repeating our choice of h until we find the hash function h that satisfies the requirement, but this is not sufficient for us because now there are collisions. So, some of the collisions can happen on the same bin which means that the query time need not be O of 1 anymore, but here is what we do.

Whenever an item collide with another item, that means that particular hash location will have multiple items in it, so let say that we create a chain of length c i at that location i.

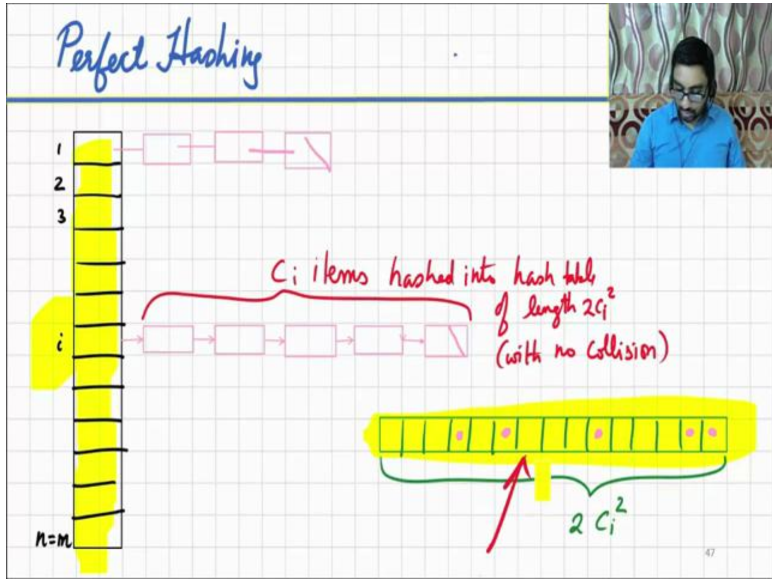
What do we do is, we take those c i items and hash them using a different hash function into a hash table of size two c times c i square and we choose the second hash function for the c i items in such a way that there are no collisions in the second hash table, but keep in mind the second hash table has to be constructed for each location where there is multiple items that are hash into the location.

(Refer Slide Time: 10:25)



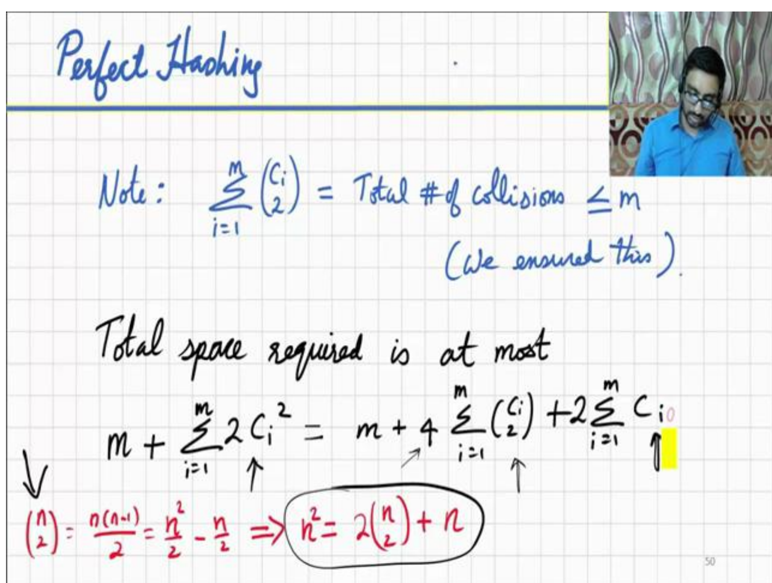
So, here is a picture. So, what we have is a hash table. Also, we use a hash function in h such that the number of collisions is kept to with m. We know that is possible and it can be done using the probabilistic method. We basically repeat until we find such a hash function. So, total number of collisions is at most m square. Summation over all i c i square is at most m.

(Refer Slide Time: 11:00)



Let us focus on one particular hash location i at which some c i items have been hashed into. So, change there is c i items. What we do is, we take the chain of items and hash it into another hash table of length 2 c i square and we ensure that this second hash table has no collisions what so ever.

(Refer Slide Time: 11:35)



As I just mentioned a little while ago if you count a total number of collisions basically summation i equal to 1 to m c, I choose 2 that is basically total number of collisions and that at most m. We ensure this. Now, Let us try to notice first of all that the search time is

O of 1. Let us say we have an item you want to search, we hash the item and look at its location within the hash table and if it hashes into a location with just one item in it or no items in it, then we are done.

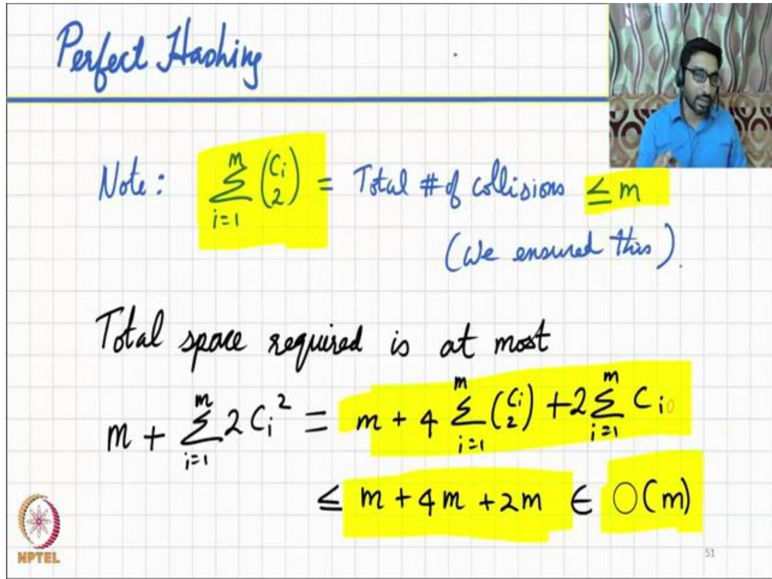
We know that either the item is present or not present, but if we hash into a location that has collisions, so there is a chain at that location and then instead of going down the chain, we simply use in that location we insert. In fact, we do not even store the chain. We only need to store the function used to hash whether use to hash the items on the chain into another table, right. So, we take that hash function and again hash the same item and that will give us a location inside the second table and that could be empty or contain the item that we are looking for.

So, the query time is clear is of one question is what about the space required. Now, the complication comes from the fact that in addition to the original hash table at each location where multiple items that where hashed into, we took those items and we created a separate table for each one of them and now many such tables could be created.

So, the question is to ensure that the total space that we have used to create all of those tables is at most o of m. So, Let us look at that. So, the total space required is that at most m plus i equal to 1 to m and twice c i square because each time there was a collision at those items where there were items that where hashed into, we created separate hash tables of 2 c i square. So, that is the total space and that is equal to m plus this 2 c i square. We want to bring it to a form that we can analyze easily.

Recall that n choose 2 can be written as n times n minus 1 by 2 of course, and that is equal to n square by 2 minus n by 2. So, this means that n square can be written as two times n choose 2 plus m which is what we are going to do here. So, this c i square will going to write it as 2 c i choose 2 and that is why we have 4 here plus two times the summation c i.

(Refer Slide Time: 15:12)



Recall that we know that the summation over all i c i choose 2 is at most m. So, this means that we can bind this entire quantity over here by m plus 4 m plus 2 m which is at most o of m.

With that we conclude our segment on perfect hashing. So, we have seen so far is that we can store a static dictionary using a hash table in such a manner that we can search for each item in O of time and more importantly we do not need the simple uniform hashing assumption. For this instead all we need is that the hash functions we use are drawn uniformly at random from a family of hash from a two universal family of hash functions.