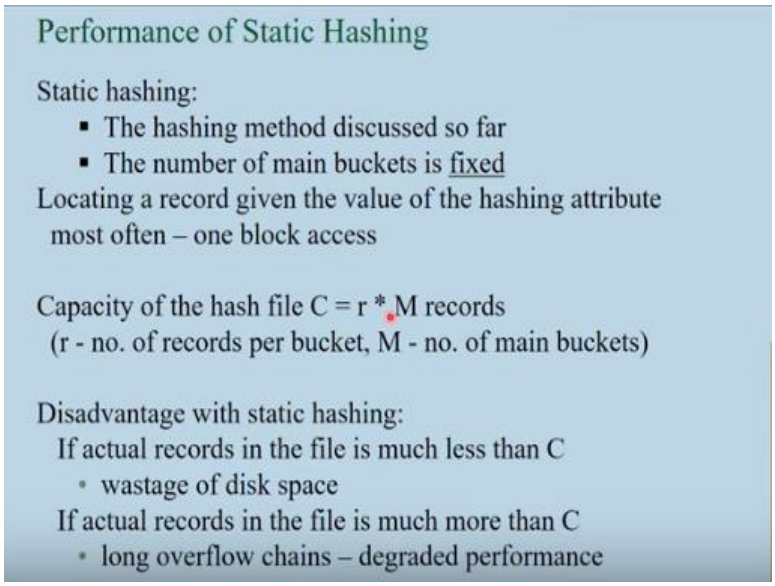
Introduction to Database Systems
Prof. Sreenivasa Kumar
CS & E Department
Indian Institute of Technology-Madras

Lecture - 27
Dynamic File Organization Using Hashing

So in the last class I have been talking about file organizations, right. So a primary file organization, the kind of policy or the logical method we use in order to organize records in a file. So we started looking at various file organizations.

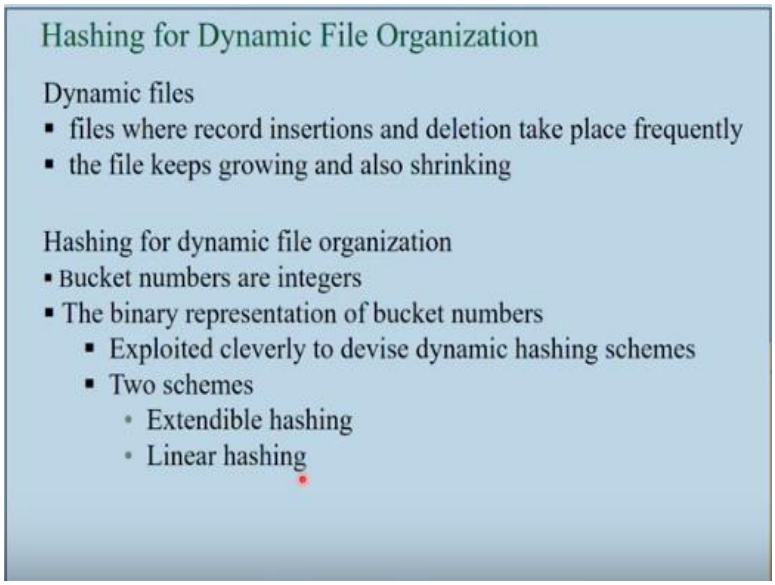
(Refer Slide Time: 00:39)



Using hashing, one can you know adopt hashing as a means of organizing the file. So we looked at static hashing in the last class. So the static hashing method the disadvantages is that when the actual records in the file is much less than its capacity okay, then there will be wastage of disk space, but if the actual records is actually much more than its capacity, then they there is a tendency for overflow chains to form and so the performance will degrade.

So for those, so in this class, we will concentrate on files, which are likely to you know grow and shrink as time progresses. So this kind of dynamic files we have we are not able to fix an exactly exact size for these files. These files are such that they will their size will vary over a period of time. And so we want to still organize them using hashing. So how do we proceed? So that is what we will discuss in this class.

(Refer Slide Time: 01:57)

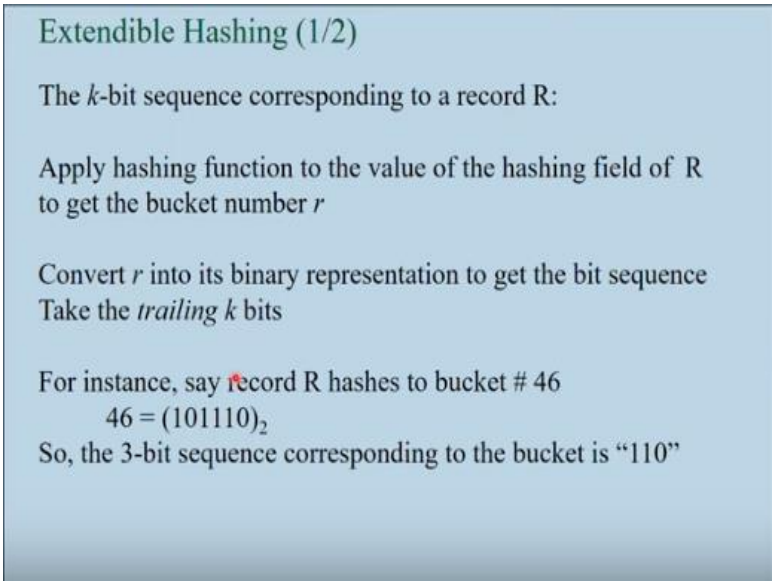


So hashing for dynamic file organization. So these dynamic files are where the record insertions and deletions take place very frequently and so the file keeps growing and shrinking. Now what we will do is to discuss two specific kind of techniques called extendable hashing and linear hashing. These are very interesting schemes using which we can you can basically employ hashing for dynamic file organization.

Both of them basically make use of the fact that the bucket numbers are integers. Bucket numbers after you apply the hash function to the hashing field, you get a bucket number. And that bucket number is some is an integer so we can convert that integer into its binary representation. So we get a bunch of sequence of bits and we cleverly make use of these bits and organize a dynamic hashing scheme.

So we will see the details. So both extendable hashing and linear hashing make use of a binary representations of the bucket numbers after you apply the hash function. Okay, so let us get into the details.

(Refer Slide Time: 03:18)

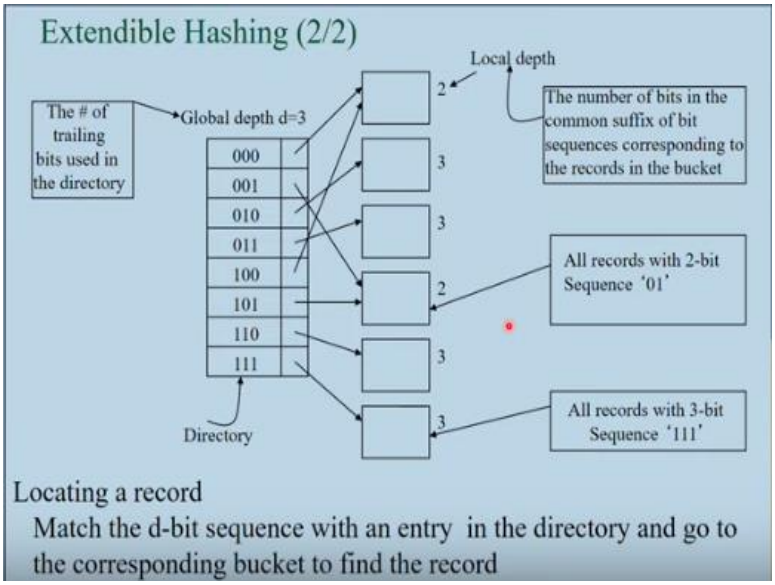


So before I proceed with extendable hashing, let me set up what I call as a k-bit sequence corresponding to a record R okay. So you take a record R apply the hash function to the hashing field of that particular record, you get a bucket number. Basically, convert that bucket number, let us say r is the bucket number convert r into its binary representation, you get a bit sequence. So some number of bits will be there, okay.

So some n number of bits will be there let us say. But then take the k trailing bits. The last bits, take the k trailing bits and then call that as the k-bit sequence for that particular record, in fact for the bucket actually called k-bit sequence for that particular bucket. Now we will see how exactly we will make use of these k-bit sequences. And this k is actually will be varying over the usage of this particular scheme.

We will notice that. For instance if you say you get a bucket number 46 you convert that into binary you get the so a 3-bit sequence corresponding to this bucket would be 110. A 4-bit sequence of course will be of course 1110 but we will go with so depending on the requirement we will be using the a d-bit sequence corresponding to this bucket. But the important thing to remember here is that we are actually taking in this class we are taking trailing bits, trailing k bits.

(Refer Slide Time: 05:02)



Okay, here is how extendable hashing works. There is a very interesting picture here. Focus on this. The scheme makes use of a directory, it makes use of a directory. So this is the directory, okay. And this directory has certain depth called the global depth. The global depth is basically what is the number of trailing bits that we are currently making use of.

So let us say d equals 3, then all the 3-bit sequences are present in the directory, all the 3-bit sequences are present in the directory. And then what the directory structure is like this. So it has a 3-bit sequence because their global depth is 3 right now. It has a 3-bit sequence and it has a point, disk pointer to a block, to a bucket actually, to a bucket. A disk pointer to a bucket, okay.

Now the idea is that to use this file system this to use this you get the 3-bit sequence corresponding to a record that you are searching for right. So and then you look up here. So this is a all 3-bit sequences are here. So you will get the 3-bit sequence. And then you go to that particular bucket. In that bucket, your record will be present, okay. So that is how it is organized.

Now these buckets, these buckets have another an important parameter associated with them called the local depth, okay. The local depth is actually the number of bits in the common suffix of the bit sequences corresponding to all the records that come to this particular bucket, okay. So for example for the local depth of this is 2 actually.

What it means here is that so 000 records that have the k-bit sequence 000 and records that have k-bit sequence 100, both of them map to this particular bucket.

And we can see that the common suffix for both of them is this the 00. Okay, so that the length of the common sequence is the local depth for this particular thing. Now basically what we are doing here is that, if the number of records that map to the bucket corresponding to 000 and 100 is not sufficient to have independent two different buckets.

We are actually putting them together in one bucket, together in one bucket. Now as the time progresses, more records will come into the picture. And it is possible that a particular bucket with some local depth, which is less than the global depth might actually have overflow. If it overflows what we do is to basically split this particular bucket.

Split this bucket 2 bucket, and redistribute the records in that particular bucket into two different buckets and then make use of the next trailing bit, okay. So for example here, let us say there is a insertion and a new record comes into this particular bucket. What we can now do is to split this bucket and make use of the, so the local depth was 2. So we are only using 2 bits so far.

And then you know kind of mapping all those records into this. Now we can actually make use of the next trailing bit and then adjust these records that come into this into two buckets, and then actually make this the second pointer to the second bucket, they split the image of the bucket, okay. So I will show you examples in which where we actually do the insertion from an empty file etc. and that points become clear.

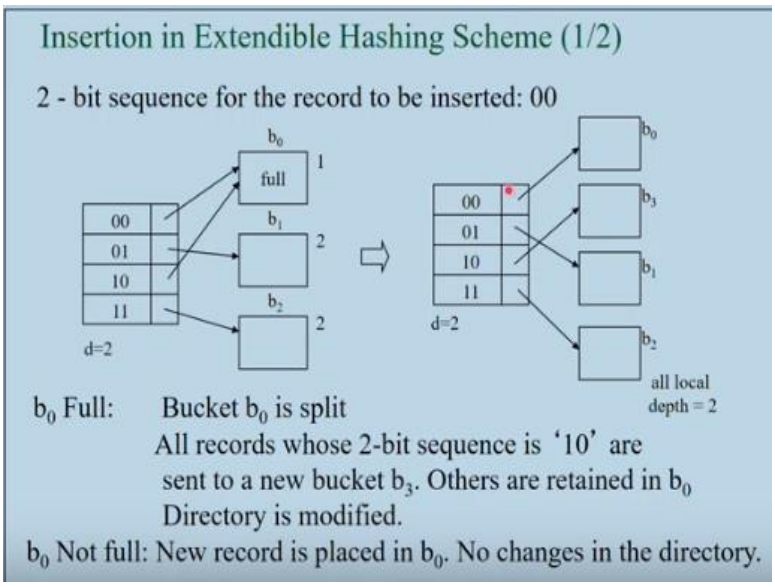
Okay, so what happens in this case, in this situation is that after some time, after some time, it is possible that all of these buckets that had local depth one less than the global depth might actually you know get split, okay. So when such a thing happens, and then you again get one more request for insertion, then we do not have an option to split this current buckets.

At that time what we will do is this is a clever technique. At that time what we will do is simply double the directory structure itself. We will increase the global depth by one, increase the global depth by one and double the directory structure and then adjust the whole structure again so that you know we can make use of this scheme, okay. So that is how so with the same number of buckets say for example, after some time, instead of 6 we might actually have 8 buckets, okay.

When all of them this one is split, this one is also split, then we will get 8 buckets, right. So and then each of these things will point exactly one bucket. Each of these entries will point to one bucket exactly. So at the time, suppose an insertion comes into the picture and we are supposed to split this there is no one more bit to use, make use of in this situation.

What we will do is to in order to create that additional bit, what we will do is to double the directory structure, double the directory structure and remap all these pointers to the first part of the directory, okay. So we will see exactly how we do that in a example situation now. I will illustrate insertions.

(Refer Slide Time: 11:29)



Let me take a toys kind of situation here, where we had using 2-bits sequence for the records. And so we have the global depth of just two here, okay. Global depth is two. And let us say we have two of these things mapping to this bucket and this is pointing to this bucket and here is another thing. So this one both of these two things have

local depth equal to the global depth whereas this one have local depth equal to one less than the global depth which is equal, right.

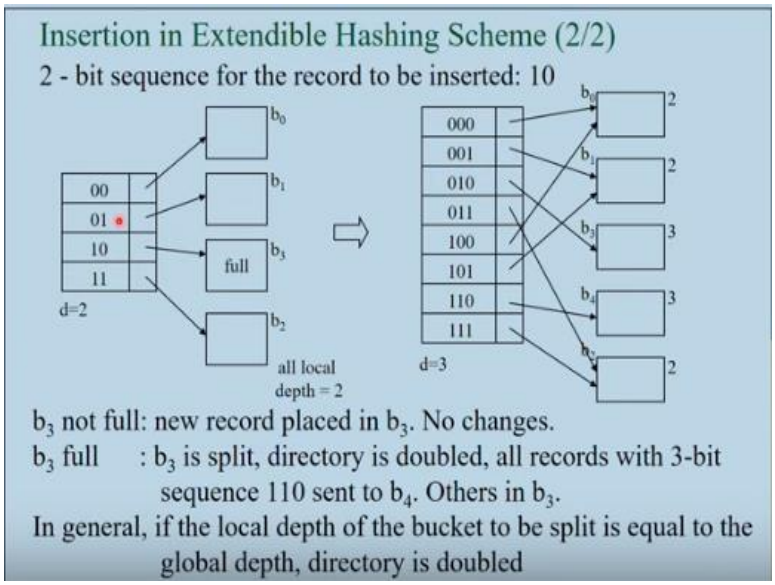
So now let us say the 2-bit sequence for the record to be inserted is actually 00, okay. So you look up here in this directory and then you figure out that it has to go into here. And so there will be two cases either this is full or there is some space for record here. If there is a space for the record, then there is no issue you can actually insert that record into that bucket itself. Suppose this is full. Suppose this is full.

Then what we will do is we will split that bucket. Bucket b 0 is going to be split. So all the records whose 2-bit sequence is 10 okay are sent to a new bucket b 3, which will create b 3, okay. And others are retained in b 0 and the directory is appropriately modified, okay. So that is how it will come. Now okay. So actually in the previous case 00 and 10 are both pointing to this place, okay.

So now if you consider the set of all records here, they either have the 2-bit sequence 00 or this 2-bit sequence 10, okay. So when you split it, you retain all those records that have 00 as the sequence in b 0 and send all those records that have the 2-bit sequence 10 to the new bucket. That is what we have done here, the new bucket okay. So basically now we had made use of the yeah, the full bit sequence.

Earlier we essentially were making use of just one bit here equal to the local depth. Now we are making use of the 2-bit sequence, okay. So now the directory structure is like this. All buckets have local depth 2, okay. So the directory is modified. Now let us see that I will now replicate this picture into the next slide.

(Refer Slide Time: 14:27)



And then illustrate as to how one more bit sequence comes into the picture what exact one more record comes into the picture and then what will exactly happen? So this is the picture that we had in the last slide. So all the 2-bit sequence all the all these directory entries were pointing to a full bucket I mean a normal kind of each of them were pointing to one bucket.

And now let us assume that 2-bit sequence for the record to be inserted is 10, okay. And let us also assume that this particular bucket is full. This bucket is full. So there is no space for, each of these buckets is a sequence of consecutive slots where records can be kept. So it is either a block or a you know a small set of consecutive blocks, okay? So there is no space here.

And so now if you tried to split this, we cannot make use of the current directory at all, because we cannot accommodate one more bit because it is a 2-bit, it is based on 2-bits. So what we will now do is to double the directory structure and bring in 3-bit sequences into the picture. Bring in 3-bit sequences into the picture. Now you can see that so there are there will be two 3-bit sequences.

Then we will have 00 as the trailing sequence, right. 100 and 000 okay. So if you focus on that 000 and 1000. They will both now point to the same bucket b 0. That is how we will map this to the new directory. So in the new directory, this b naught will continue to be there as it is, but only thing is now it has become a bucket with one less

you know depth. Here it was equal to the global depth now it will be depth 2, the global depth is become 3, okay.

So that means two of these entries will actually point to that, what are those two entries? What were the ones that were pointing earlier, okay. So all those records with trailing bit 00 were coming here. Now they may actually have either 000 or 100. In either case, they will come here. That is how we will adjust the directory links. In a similar way here for b 1, all those records with 01 were coming here.

Now we will see that 001 101 will come here. So 001 101 both of them will come here, okay. So that is how we will, when we double the directory structure, when we double the directory structure, we will appropriately adjust this pointers, and then see the number of buckets actually has just grown by one only. So because b 3 is full, we split that and now create b4, b 3 and b 4 we created.

So b 0, b 1, b 2 were existing; b 3 had to be split. So now the record in b 3 are split between b 3 and b 4, b 3 b 4. Now b 3 records that were coming here had 10 as the 2-bit sequence. Now they will have 010 as the 3-bit sequence. And the other one, the 110 will now go to the new bucket, will go to the new bucket. So when you split the records here, when we split the records, all of them had 10 as the 2-bit sequence.

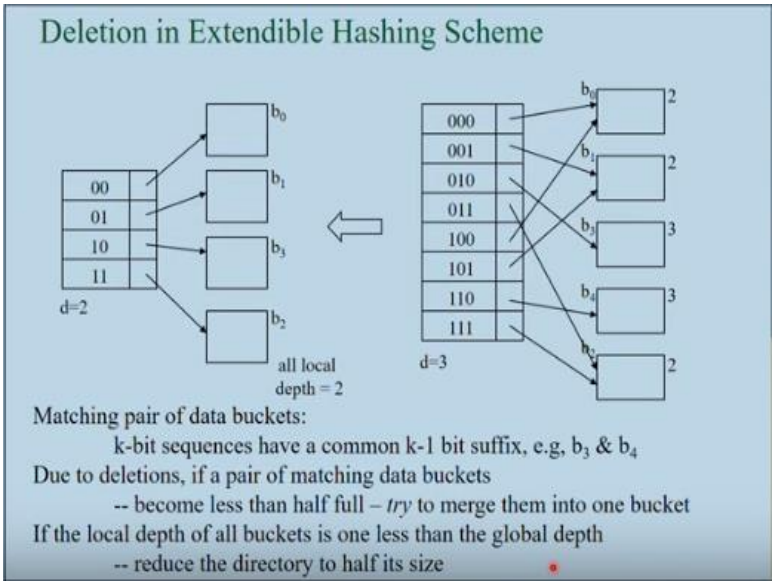
So if you now consider the 3-bit sequence corresponding to them, they will either have 110 or 010, okay. So all the 110 things you will put into a new bucket. All the 010 things you put it into old bucket, okay. So that is how we will adjust the directory. So you can see that wherever it is actually necessary to create more space, we have done that. So at this place b 3 new records are coming into picture.

So we have created an additional bucket. And then in order to take care of coming to this place, we have doubled the directory structure. And then we retained all the other pointers as they are in some sense as they are in the new directory structure. So that is how the file can expand. And in fact, file can also shrink when new records when records are getting deleted.

When records are getting deleted, it is possible that in some of these buckets, slots will become empty. The slots will keep becoming empty. So when slots become empty okay so yeah, so this point basically you know explains this b 3 being split.

So in general we can actually, before we consider deletion, in general if the local depth of the bucket to be split is equal to the global depth, local depth of the bucket that has to be split is equal to the global depth then the directory structure has to be doubled, okay. The directory has to be doubled.

(Refer Slide Time: 20:35)



Now deletion, let us look at deletion. So deletion and extendable hashing. For example, let us look at this. The current scenario that was there. So this one is the let us say some more some records have got lost I mean have been deleted, okay. Now you can observe that there are what are called matching pair of data buckets here, okay? The k-bit sequences have a common k - 1 bit suffix. Then those are the matching buckets, right?

For example, b 3, b 4 are the matching buckets in this case, okay. So now what happens is that in case the, in case due to deletions if a pair of matching data buckets both become less than half full, both become less than half full, then you can actually combine the records and put them into one bucket because the both buckets have become less than half full.

So if you combine, you will still be able to you will be able to accommodate the records into one normal one bucket, right. They were in two buckets and both of them are less than half full. So we can now put them in one bucket. So the what you do is you take the matching buckets merge them. Merge them and reduce its local depth, make use of one bit less now and then adjust the pointers appropriate. So this pointer and this pointer will now point to the same bucket actually.

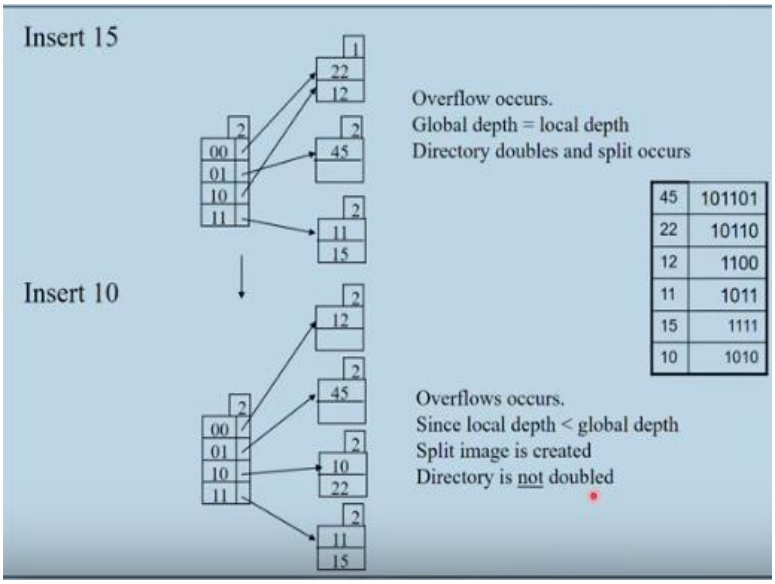
So that is how when you merge you can adjust the directory. So you can take care of the growth of the file as well as the shrinkage in the file using this nice directory structure. And in general of course there is a assumption here that the hashing function hashes the records uniformly across the bucket numbers. And you can also see that what is the maximum that directory size you can have.

That will be equal to the number of bits that are available, number of bits that are available for the, the bucket numbers. Okay. Say if the bucket numbers are 0 to 64, then you know that there are you know so many bits 5 bits, right? So that is the maximum you could go. Now one other interesting thing can happen. So as you are deleting as you are deleting the local depths will keep coming down actually.

Okay, so if the local depth of all the buckets is actually one less than the global depth, then we actually can reduce the directory to half of its structure and then adjust the pointers again. Okay, just like we have doubled the directory when there is requirement, we can also reduce the, you know cut the directory by half and then adjust the pointers again using this nice kind of scheme, okay.

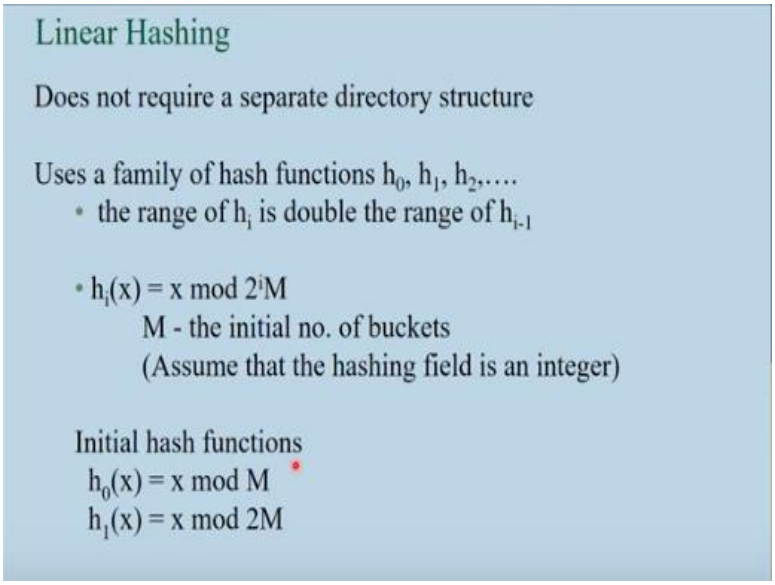
So that kind of explains the very nice you know technique called the extendable hashing.

(Refer Slide Time: 24:11)



I have a bit more detailed example where I have taken a small you know bucket capacity and then gave the insertions and then have shown you how exactly the whole thing goes. I will not go through this example because it is available in the slides. You please go through them. Even the bit sequences are kept here. So you can follow it easily.

(Refer Slide Time: 24:37)



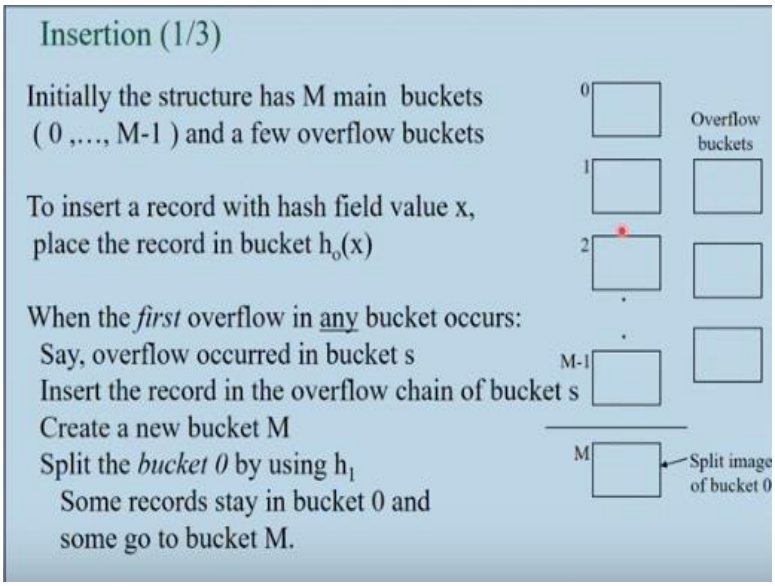
So let me go to the other interesting technique, which is called the linear hashing. This is very interesting, equally interesting technique called linear hashing. This does not even require a separate directory structure. But it can manage the dynamic file very nicely. What it does is instead of just using one hash function, it actually makes use of a family of hash functions, okay. So h 0, h 1, h 2 etc., it makes use of so it assumes the existence of a family of hash functions, which are like this like the range of h i is

double the range of h i – 1 okay. The range is the number of values, the range of the values that it gives, the bucket numbers. And the range of h i is double the range of h i – 1, okay.

So I will show you what is the such a family here. For example if h i of x is x mod 2 power i M then this property will be holding. So h naught of x is mod M, x mod M where M is the initial number of buckets that we have, okay. Then h 1 of x will be x mod 2M okay. So the range of h 0 is 0 to M – 1. The range of h 1 is 0 to 2M – 1. It will double, okay.

So we will make you we will assume the existence of such a family of hash functions, but at any point of time, we will actually use a pair of hash functions, h i, h i + 1 like that. We will use a pair of hash functions. So let us assume that the initial hash functions are h naught of x is x mod M, h 1 of x is x mod 2M for some M which is the actually the number of initial buckets, okay.

(Refer Slide Time: 26:51)



Here is how the the structure looks like. Here is the M 0, 1, 2, 3, M – 1. These are the main data buckets. And a few overflow buckets are assumed to be existing, okay. Now the insertion to insert a record say with hash field value x, we will compute h naught of x, so we will get a bucket number, go there and insert it here. That is all, simple. In case there is no space there, go and insert the record into the overflow chain, okay.

That is so simple. Now whenever the first overflow in any of these buckets in the data bucket 0 through M – 1 occurs, okay. Say overflow occurred in some bucket s, insert the record into the overflow chain of this bucket s, okay. What we do is create a new bucket M, create a new bucket M and split the bucket 0 records, bucket 0 records using the hash function h 1, okay. Split the bucket 0 by using h 1.

So some of the records will stay in bucket 0 and some will go to M actually. So if you take a bucket M, so all these buckets were having, when you take a mod M, they were coming to 0. Now if you take mod 2M, mod 2M, they will actually go to either 0 or M, okay. If the we will we can show that, little later I will show that. So when you take a h 1, the h 1 is x mod 2M, x mod 2M. So you take the records that are here, okay.

Now rehash them with the h 1, rehash them. Some of them will have 0 value as a hash value. Some of them will have actually M. So now we create a new bucket M. This is going to be called as a split image of bucket 0, okay. Now notice one thing here that we are not actually touching the bucket in which the overflow has actually occurred. Somewhere in these buckets, a overflow has occurred.

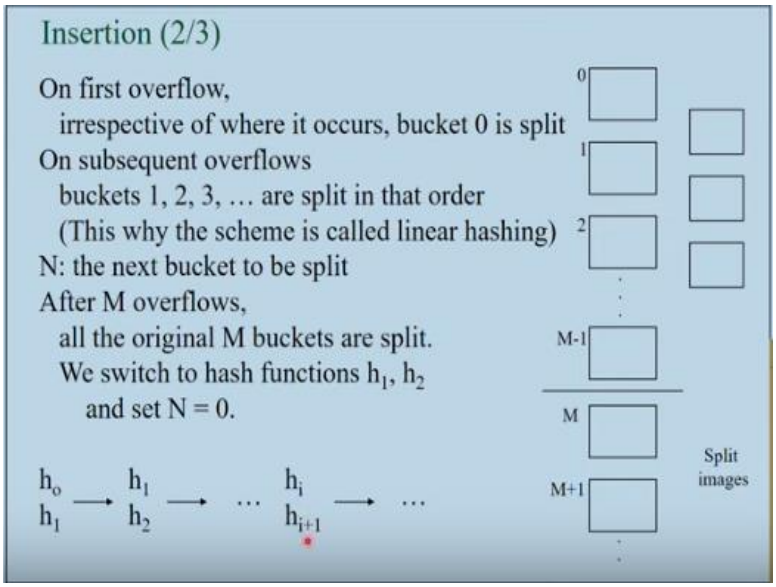
So that overflow kind of starts triggering these expansion of the structure. So we do not actually touch the bucket where the workflow has occurred. We will use a overflow chain there. But we start splitting the buckets from 0 onwards. The moment the first overflow occurs we will start splitting 0. And then next one more overflow occurs we will split 1, and then one more overflow 2.

So we actually split the buckets in a linear order. That is why it is called linear hashing. It is called linear hashing because we spread the buckets in linear order. So whenever we split, we kind of create the mirror image of this particular bucket like this. So M is the mirror image for 0. M + 1 is the mirror image for 1, M + 2 will be the mirror image for 2 etc. And then whenever we create a mirror split image, split image, we basically take the records in that bucket, okay.

Use the next hash function h 1 redistribute the records from this into this bucket and its split image, okay. So this is a very nice idea. I do not we do not even have a

directory structure here. We do not even have a directory structure. All that you have to remember that all you have to actually now keep track of is how many of the original buckets have already been split. That number you should know. That number you should know, okay. So let me show you the actual okay.

(Refer Slide Time: 31:35)



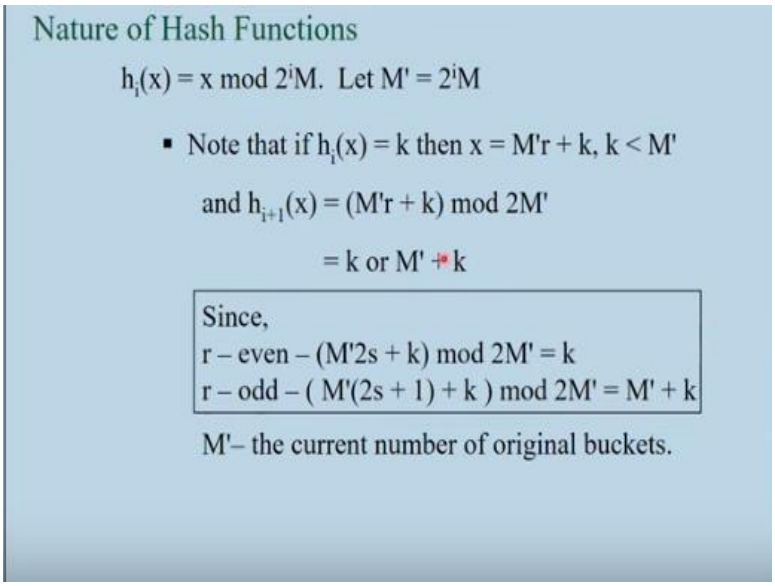
On subsequent overflows, bucket 1, 2, 3 are going to be split in that order and we keep creating the split images. So N will keep track of one number, which is the next bucket to be split. Next bucket to be split. So after M number of overflows 0, 1 through M right. So there are M data buckets to start with. So if M overflows actually occur, then all the M original buckets would have been split into M additional buckets.

So now we have 0 through 2M – 1 so many 2M number of buckets actually. We now have 2M number of buckets. Then what we will do actually here at this point is to switch over from h 0, h 1 to h 1, h 2. We will switch over to the next pair of hash functions, okay. So h 1, we have been using h 0 to start off, and then for redistributing the records from the bucket to its split image we were using h 1, okay.

Now if all the original data buckets have been split, then obviously we have to now use h 1 to locate the record. And for splitting, we will use h 2, okay. And we will actually of course, this number N which keeps track of the next bucket to be split we have to set it to 0 okay. So this is how the linear this one will work. So we make use

of h 0, h1 to start off then h1, h2 and so on at any point of time we will be using h 1, h i + 1 pair of hash functions in order to handle this here.

(Refer Slide Time: 33:35)



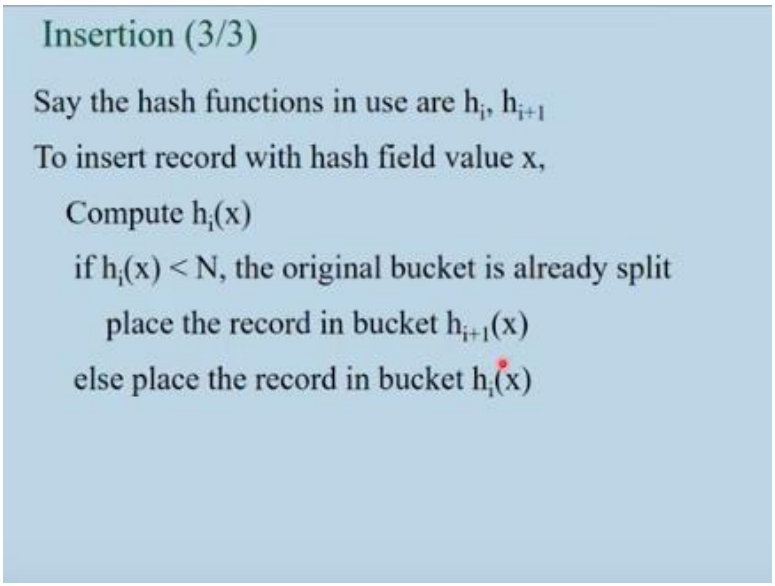
So here is a here is a proof that if you have h i of x as x mod 2 power i M then you know if you apply h i + 1 to the same number to apply h i + 1 x you will actually get either k or M prime plus k, okay. Let us so let us put this 2 power i M as M prime, okay. Now let us apply the so if h i of x equals k, let us say if h i of x equals k then you basically can write x as some M prime times r + k right using the usual mode, where k is strictly less than M prime.

Now consider applying this next hash function h i + 1 to the same x. So h i + 1 x is this M prime r + k mod 2M, 2M prime okay. The nature of h i + 1 is that it is x mod double the original number, double the current original buckets. So if M prime is here then this will become 2M prime, okay. So now consider that. That will actually be either k or M prime + k, how? Then you can show that here.

So you can if you focus on this r there are two cases r is even or r is odd. So if r is even then it will be M prime 2s + k mod 2M prime in which case you can see that this is divisible by 2M prime and so it will become k. If r is odd then M prime times 2s + 1 + k mod 2M prime right in which case you can see that this one will give you M prime and so this will be M prime + k, okay.

So we can we have guaranteed that if you use the h i + 1 next hash function on the bucket number you either get the k that you used to get when you apply the old function or M prime + k, okay. So this is a crucial requirement for us. Because this M prime + k, you know is our split image for bucket k, okay.

(Refer Slide Time: 36:30)

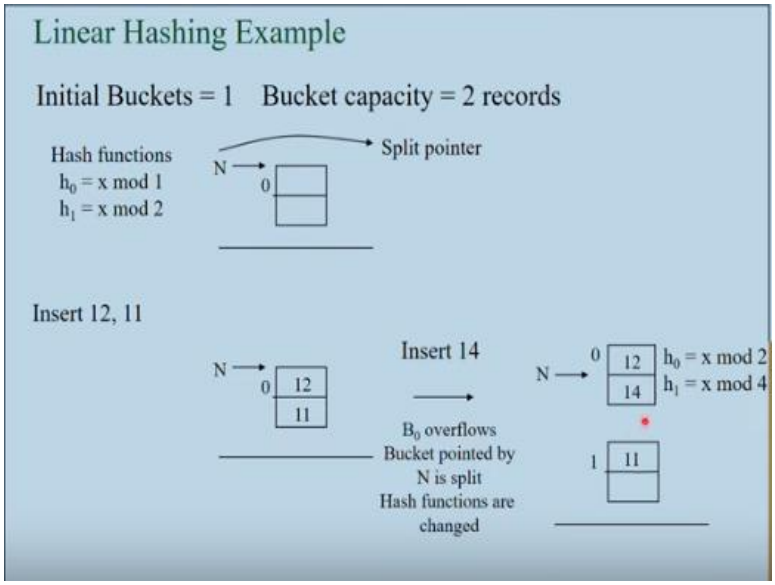


Okay, so here is the actual insertion algorithm. Say the hash functions we are currently in user are h i and h i + 1. So to insert the record with hash field value x, we will first compute h i of x. We compute h i of x. If h i of x is less than N, what is the N? N is the next bucket to be split. We always maintain this number. If h i of x is less than n, then the original data bucket is already split, already split.

So what we need to now do is to place the record in the bucket that is given by h i + 1 x, apply h i + 1 x with the same bucket number to the field value and then you will get a bucket number. So this will be either the original k or its split image M prime plus k and then place the record there. In case it is not the case that the h i of x is actually greater than n, then you know that the origin data bucket is actually not yet split.

And so you can actually place it in that bucket itself as it is directly. While placing the record, especially in original data buckets, there might be a overflow problem in which case you go to the overflow chain and then insert the record there, okay. So this is how linear hashing works.

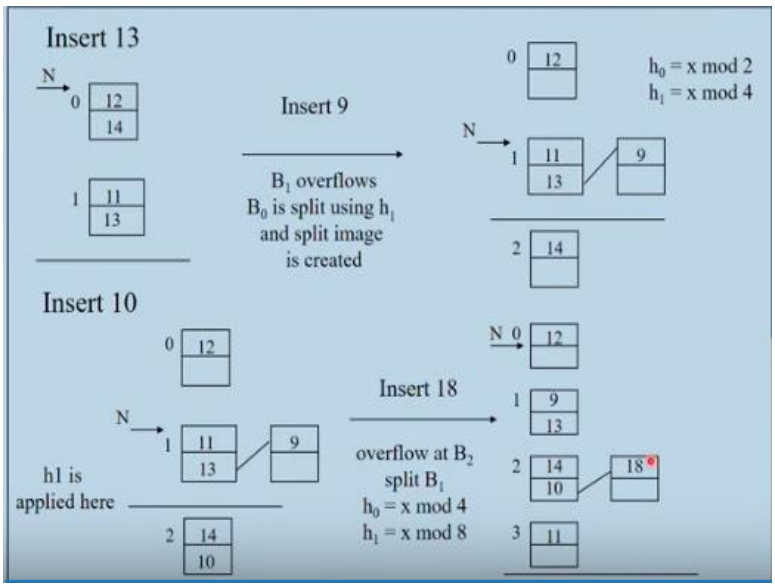
(Refer Slide Time: 38:08)



Again I have a detailed example where you know linear hashing has been applied with some specific bucket numbers bucket capacity 2 two records per bucket like that and then I have shown how exactly so I have taken h naught of x is x mod 2, x mod 4 and then illustrated as to how these okay the initial hash functions are x mod 1 and x mod 2 and then after the insertions insert 12 and 11.

So they both so since it is mod 1, so they simply go there and then insert 14. So this place has no more space. So it has to be split and so now you have to use h 1 f and n has to be split which is like this and then so and then now you use h mod x and h mod 4. So because h mod 4 this one gives 1 when you consider mod 4 it will go to the new bucket, okay.

(Refer Slide Time: 39:48)



So you can follow this example and then understand this linear fashion. So basically both these techniques are making use, cleverly making use of the binary representation of the buckets, binary representation of the bucket numbers okay. And then and then cleverly ensuring that the structure expands or shrinks. So in the case of linear hashing for example, when shrink occurs, you can actually keep discarding the split images okay.

So if so then when all the you can merge the split image with the original data bucket if both of them become less than half like that. So when they the number of buckets decreases then you can actually switch back to the old pair of hash functions, okay. So that is how the linear hashing works. So I think we will stop here for today.

I will take I will discuss index structures for other, you know file structures in the next class. But please go through these examples and also work out your own examples in order to get a good grip of this how both extendible hashing and linear hashing work.