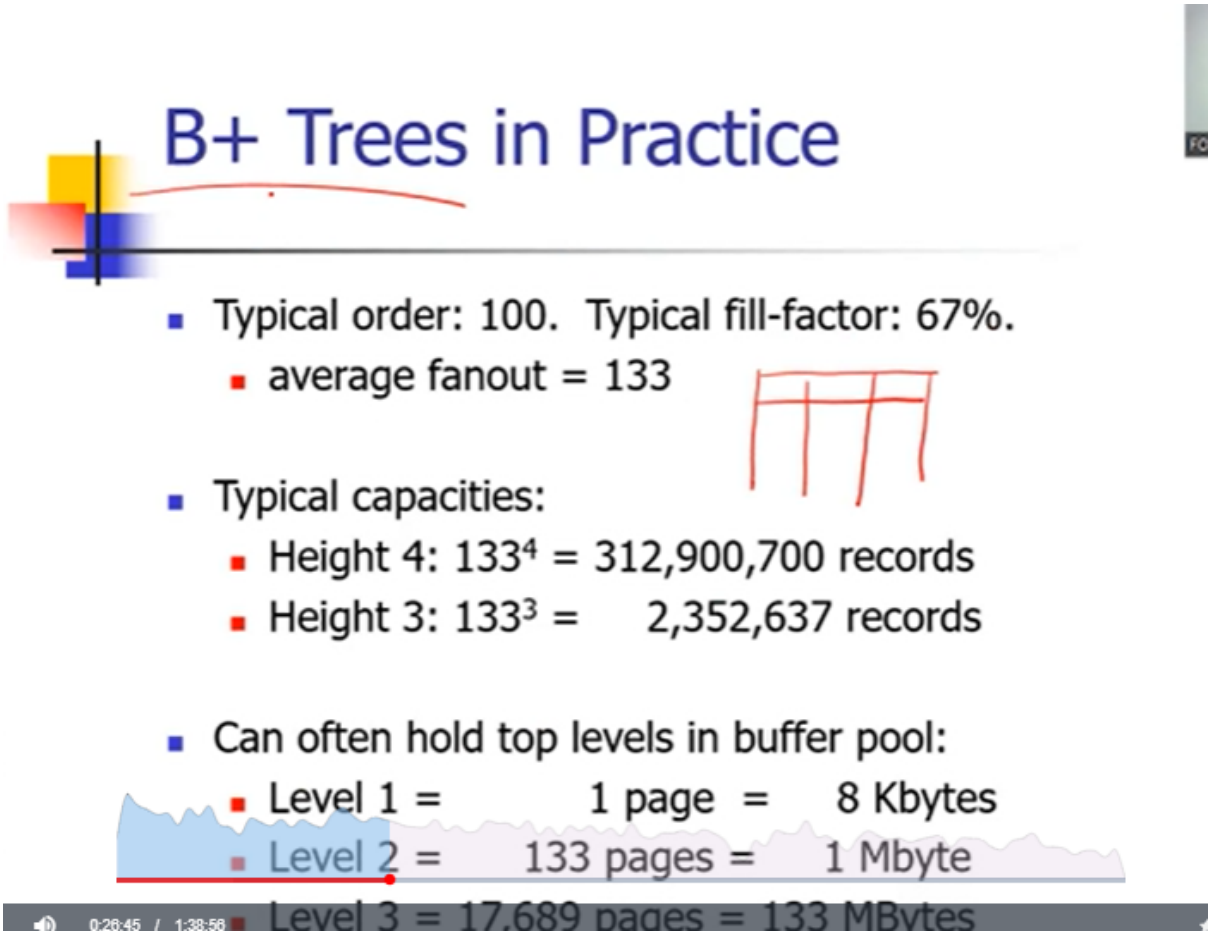
B+ Tree indexes - Good for range selections.

B+ tree is tree. So it has edges and nodes. One node is one page (One page = one data block from hard disk point of view).
50% of values in a node is called an **order.** A typical order = 100. Which means tin here are 200 pointers to a node. Except for the root node, other nodes need to have a minimum of 50% pointers (an order).

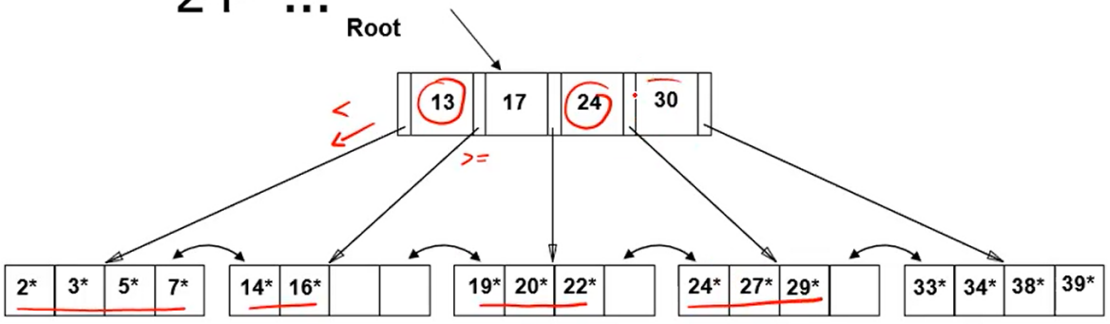


If you have a 4 level height of B+ tree, that means, you can easily find data location from the B+ tree instead of searching in a table of 133^4 = 312900700 records.

Searching values in B+ tree

Example B+ Tree...

- Search for 5*, 15*, all data entries >= 24* ...



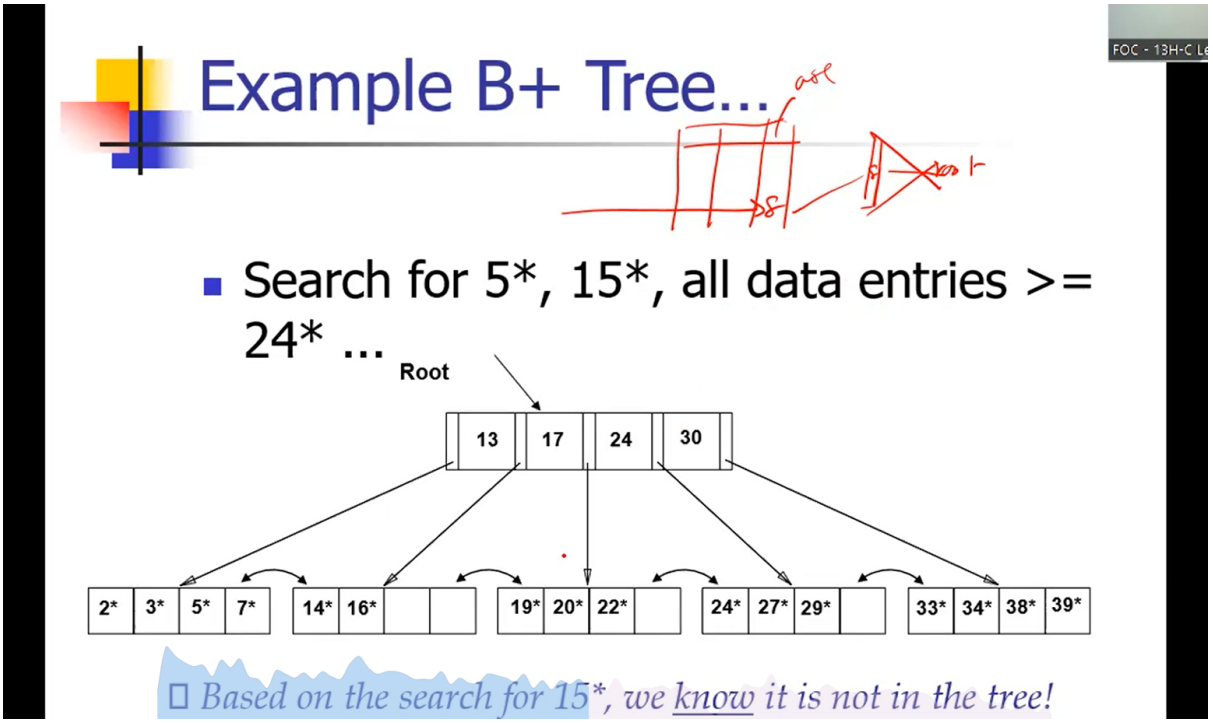
Based on the search for 15*, we know it is not in the tree!

In this B+ tree order is 2. So we have 4 pointers in a node.

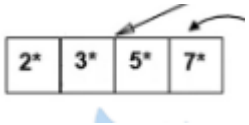
1. In the above B+ tree if you want to find 16, you start from parent node. By looking at it you know it's in between 13 and 17 sections. So you go to that child node and retrieve 16th recordId from the leaf level.
2. If you want to find 24, you start from the parent node, you can 24 is there, but 24th record id is saved in the leaf level. So you go to that leaf node and take the recordId.
3. Lets say you are going to do a range selection from 27 to 38, in that case first you start from the leaf node which 27 is available and take 27, 29. But you can see 33, 34, 38 are in the next leaf node. In here you can directly go into next leaf node because they are doubly linked. No need to go to parent level and go there.

- Actually you can just say only the recordIds are available in leaf nodes. What available on leaf nodes are depends on the alternatives. If this is alternative 1, the whole record is available in leaf node, if this is alterative 2 or 3, only the recordId is available.

Adding values to B+ tree



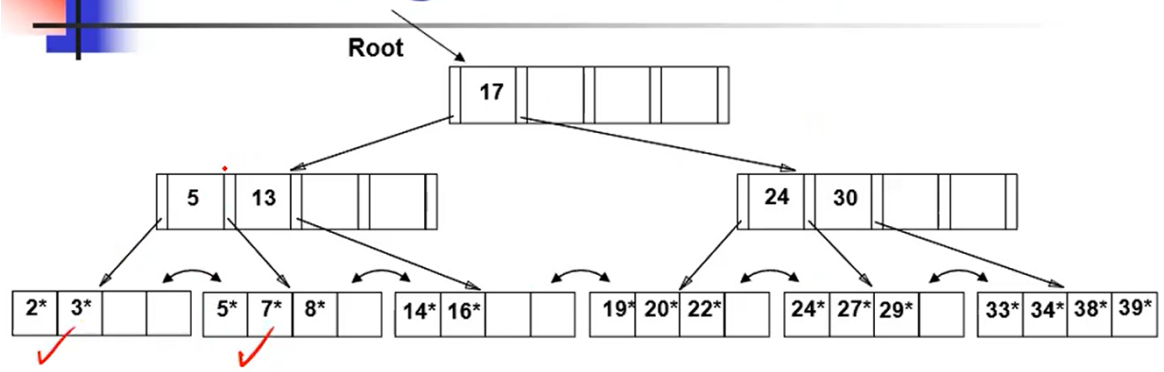
Here let's say you want to add 8 to this B+ tree. You know 8 should go here.



But this child node is full. If it's not full you can just add 8. Since this is full you are going to split this node into 2 child nodes. And divide the values to them. Now you have 2, 3, 5, 7, 8 values. First you are going to sort this, then take the middle value and *copy up.* Now you can have 2, 3 in one child node and 5, 7, 8 in the other child node. So 5 is going to be copy up to parent node. But parent node is also full. Here as well you can split the parent. Now you have 5, 13, 17, 24, 30 in parent level. First sort it. Then you can *push up* the middle value which is 17. And divide other values with 2 paret nodes. So first node will have 5, 13. Other node will have 24, 30. (In *push up* you dont duplicate the values in child node. In *copy up* you duplicate. The reason is, at the end of the day your record id will be saved in leaf nodes, not in parent nodes. So no need to *copy up* parent nodes values when spliting,)

The resulting b+ tree

Example B+ Tree After Inserting 8*

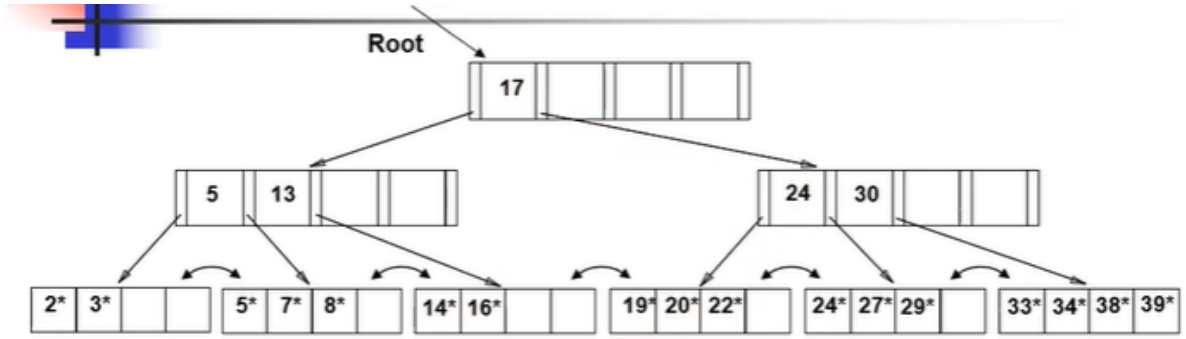


- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Try adding these values to B+ tree
10,15,11,17,20,14,19,9,12

Answer

Deleting from B+ tree

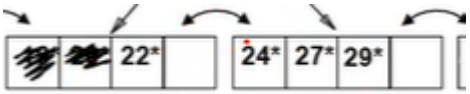


1. Delete 19 and 20.

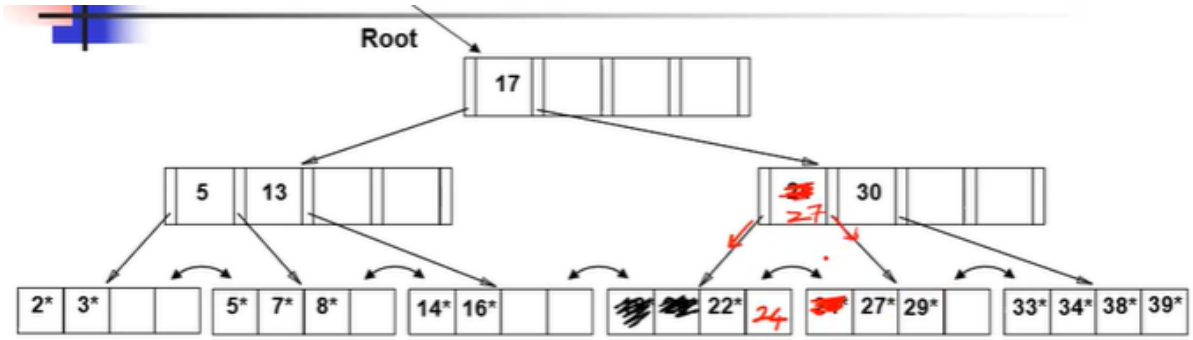
We can just remove 19. No problem since still we have 50% pointers in that node after 19 is removed.

But we can't just remove 20. because then 50% of pointers are not in that node.

Since these nodes are doubly linked lists we can ask nabouhr nodes for pointers like, we can take 24 from sibling node.



Resulting B+ tree.



Hashing

1. **Static Hashing**

You know in here you store values by making the values go through a function and according to what function returns. Lets assume that function is => value%10.

Here, if you want to save a lot of values which have 1 remainder (value%10 = 1), that node is going to be large. Hard to retrieve values. Also if you delete some value from here, that space won't be released in static hashing.

Static Hashing... (contd.)

Problems...

- Insertion can create long overflow chains can develop and degrade performance.
- Deletion may waste space

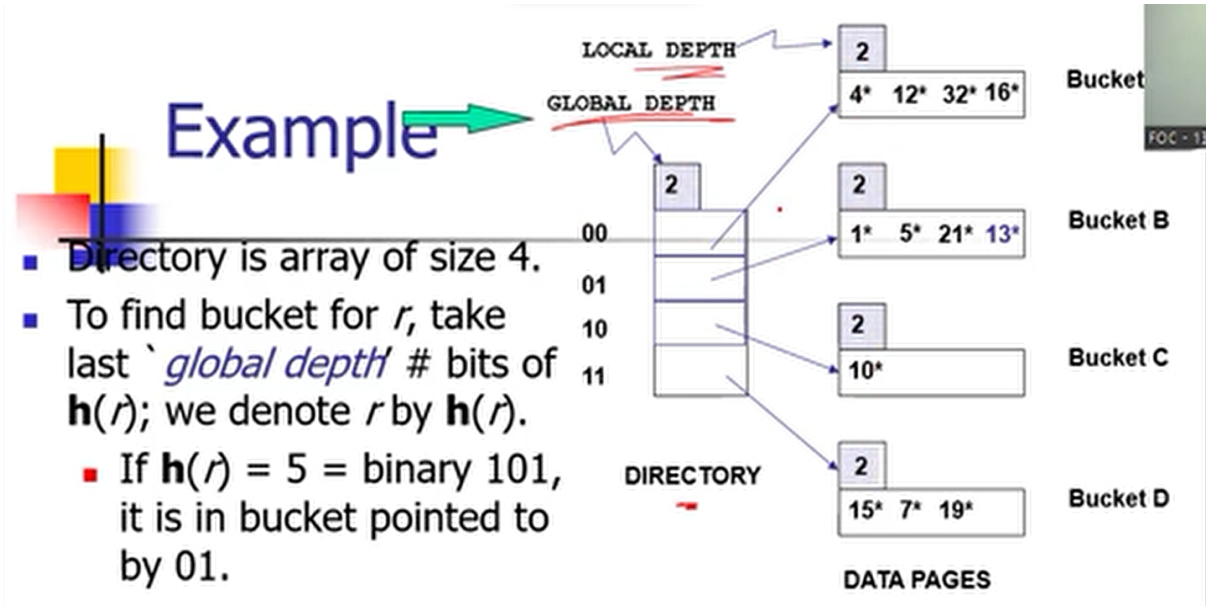
To overcome these problems in static hashing, dynamic hashing came.

2. **Dynamic hashing**

There are 2 types of Dynamic hashing techniques.

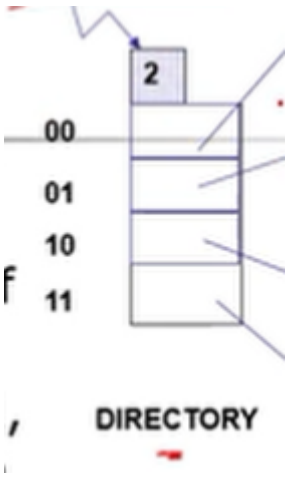
1. Extensible hashing
2. Linear hashing.

We focus on extensible hashing here.

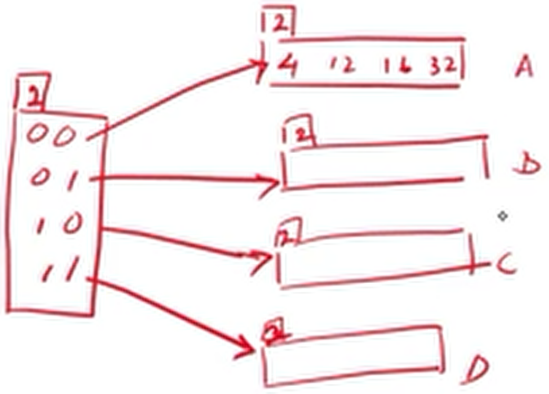


Global depth - no of bits you took to create the directory. (in the above example global depth is 2)

In here you have a directory. In the above example, the directory's array size is 4. If the size is 4, we know (2^2 = 4, take 2's power. Local depth is 2) with binary we can have 00, 01, 10, 11 possibilities.

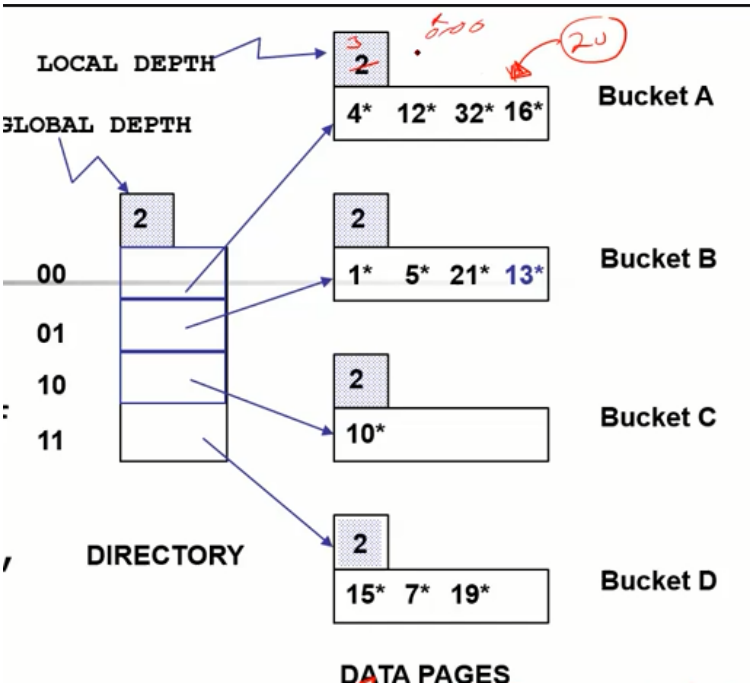


Each element in the directory is a bucket. In the above example we have A, B, C, D buckets. We store values by taking its last global depth bits.

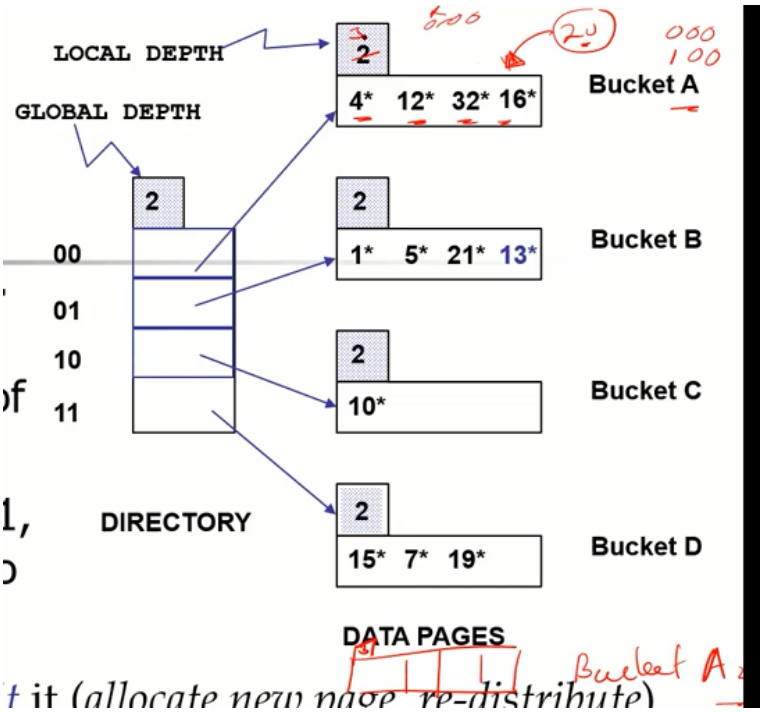


Lets say we want to find 4. 4 in binary is 100. Since local depth is 2, take the last 2 bits of the binary value. It's 00. So 4 is available in 00 bucket.

Inserting a value

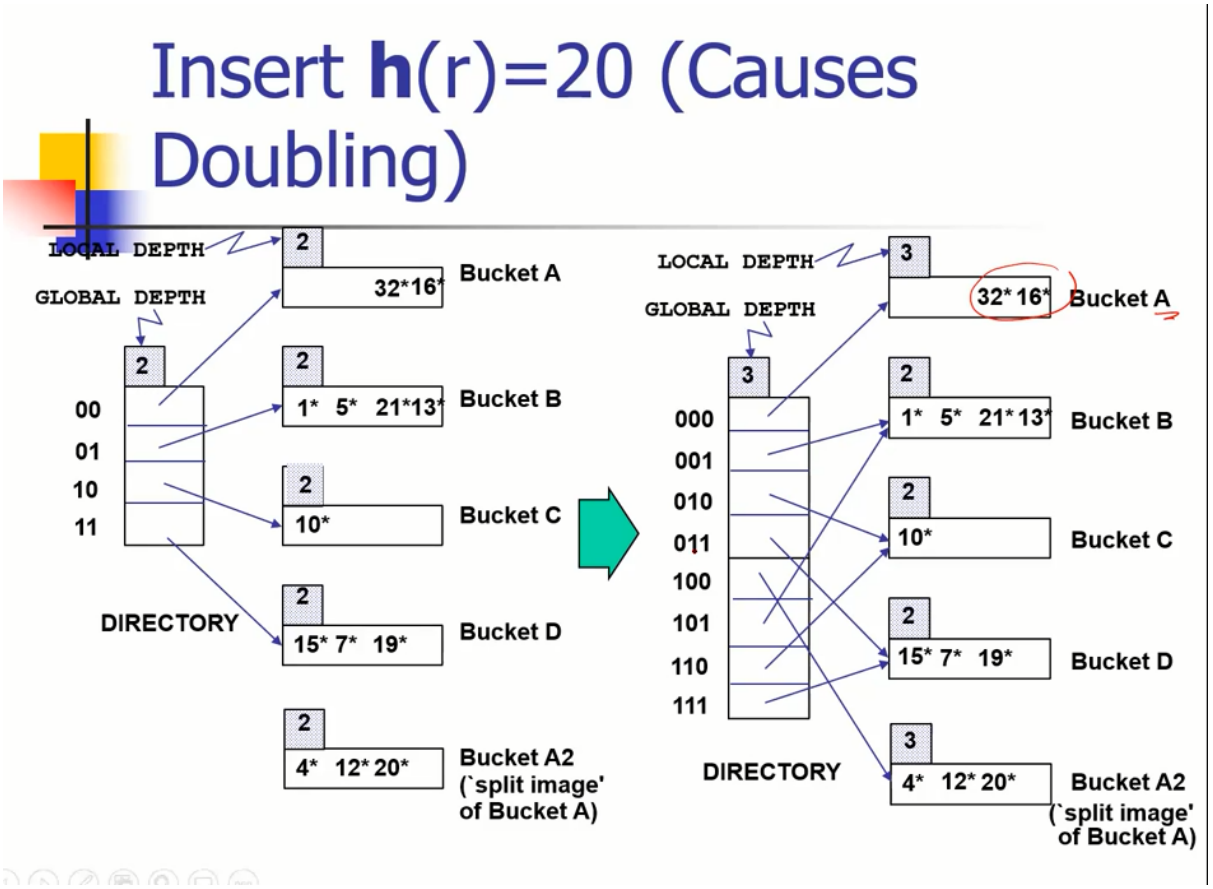


Here if we want to insert 20. (00 is the last 2 bits) we can save it in Bucket A. But Bucket A is full here. In that case we can split the bucket (allocate new page, distribute it) instead of overflowing the bucket like in static hashing.



When we allocate a new page for bucket A2 we have to increase the local depth by 1. Local depth mean, number of bits we want to consider to take values into the bucket. So now we have to consider 000 and 100 last bits as well.we can have 000 into Bucket A and 100 Bucket A2.

But the global depth should always be equal to local depth or greater than local depth. In that case we have to increase global depth as well.



Notice how 001 and 101 elements in the directory are pointing to B buckets. Reason is, local depth of bucket B is 2. So even though it is 001 or 101 bucket B only cares about the last bits which is 01.