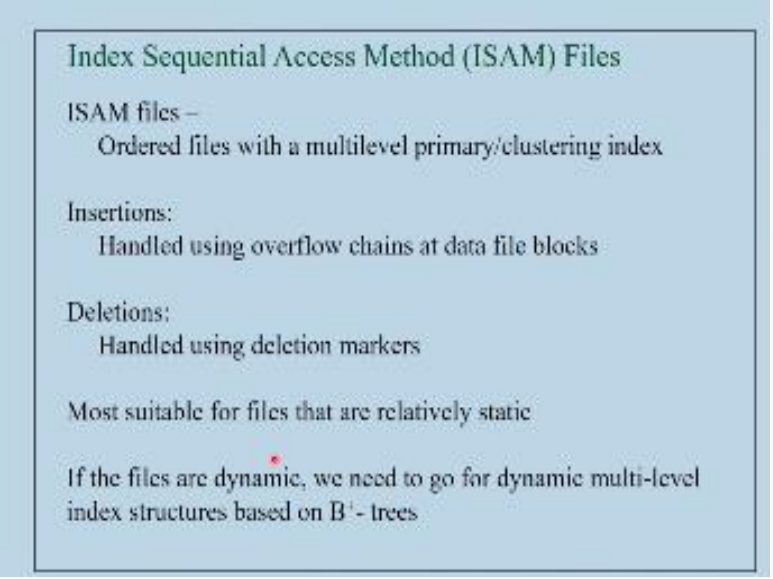
Database Systems
Prof. Dr. Sreenivasa Kumar
Computer Science and Engineering Department
Indian Institute of Technology – Madras

Lecture No. 31
B+ trees on Disks

Okay. So in the last lecture we have looked at various index structures but then like the primary index clustering index and the secondary index.

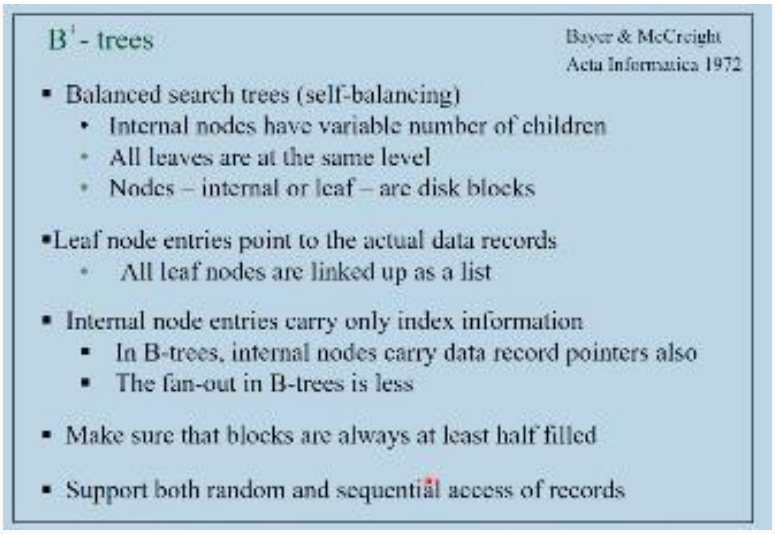
(Refer Slide Time: 00:40)



But we found that modifying these indices is costly. So, if we have a dynamic file then making use of these effectively making use of these indexes becomes a little difficult. So, the index sequential access method files that we mentioned last class last lecture are most suitable for files that are you know relatively static. Here and there a few insertions and deletions might occur, but we will be able to handle them using overflow chains or deletion markers etc.

What if the file is actually truly dynamic and mainly keeps on so of course we have also discussed how to manage that dynamic files using hashing. Okay.

(Refer Slide Time: 01:32)



Now in this lecture I will focus on how to use B+-trees in order to take care of a dynamic file and then give search access to the records in the file. So basically, we are also talking about B+-trees. Okay. So sometime after I explain this B+-trees I will later towards the end of the lecture probably we will talk about the simple B-trees. So, but we are focusing largely on B+-trees.

Now you have studied tree structures as a very helpful data structure for searching through a searching for an item in a given set of items. So, we have been familiar with binary searches, AVL trees for searching and then we you probably have studied red black trees or two three trees etc. So the basic principles of tree based searching is the same here in B+-trees also but one of the main things that you will find here is that what should be a node in the tree?

A node in the tree in the memory thing was having some item and then depending on whether it is binary or two tree etc. a few pointers right but now the new consideration comes into picture here in the sense that the access of data from the disk is always in terms of blocks. So, when you read an unit that is a block. So, it is about 4 kilobytes of data right.

So we will want to treat that as a node and then there is so much space in that node. There is so much space in that node so how do we effectively utilize that space. So basically, what we do here as you notice a little in a short while. The number of children that a node has will be very high. We call we also call it as a fan-out, call it as the fan-out. The fan-out will be very high running to something like a 100, 150 like that so a node will have 150 children.

Now if your fan out is high then you can see that the first level you have 150 children and the next level you have 150 times 150 nodes, the third level you have 150, 150 hundred times 150 cube. So, if you sum it up you can see that the structure can accommodate a huge number of nodes by just using two or three levels. Okay. So that is one main great advantage of these structures.

Now okay B+-trees are designed as balanced trees, so they are actually self balancing. We will show how exactly they are balanced, and these internal nodes will have some variable number of children. What I was talking about is the maximum that the degree of the fan-out is the maximum. Okay. So, we will see later on as to what are the other restrictions that are there. Now one other thing is that all the leaves are at the same level and the nodes are disk blocks.

Now in B+-trees it is the leaf node entries, entries that are there in the leaf node that actually point to the data records. So, the data records are actually in some other file in this block. Okay. So, we keep a record we keep a pointer to that data record in the B+-tree. Notice that we are talking of B+-trees more like another index structure. Okay. So, we will put the data records in a bunch of blocks and then keep the pointers to these data records in this tree structure. Okay.

Now all the data record pointers are going to be available only in leaf nodes. I will tell you what is the advantage of this decision little later after explaining this B+-trees. Okay. Then the internal loads what will the internal nodes do. Internal nodes will carry index information. Some information which will guide the search to go to the appropriate leaf level. So, unless you go to the leaf node you will not get the pointer to the data record.

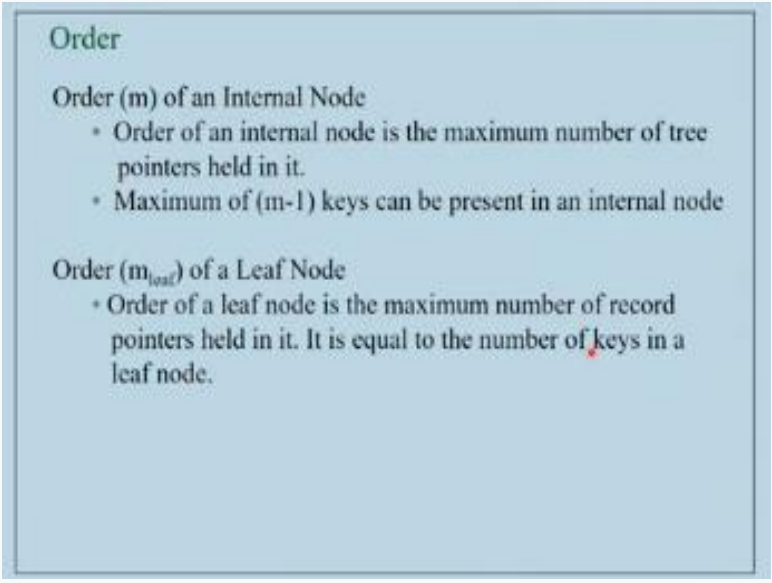
Okay and so the internal nodes will have suitable information so that the guide and the search will be guided to the appropriate leaf node. Okay. Now the difference actually in compared to B+-trees in B-trees is that the B-trees the internal nodes also carry data record pointers, we will come back to this point a little later. Now broadly what these structures do the B+-tree does is

to kind of make sure that all the nodes I kind of interchangeably use the word node and block because blocks are nodes.

So, make sure the blocks are all at least half filled. I will show you in more detail about this point. Now because all the leaf nodes are the ones that actually have the data record points. We have an additional feature in the tree saying that I can chain up the leaf nodes. The leaf nodes are at the last level right. So I will chain up these leaf nodes and then that will provide in some sense a sequential access to all the data records in ascending order of the search key tree, you take a binary search tree and then access the okay not binary all right yeah we will drop that. Right.

I will show you how leaf nodes are linked up little bit later. Okay. So, let us come to the details of the structure. All the leaf nodes remember that all the leaf nodes are going to be linked up as a list. Okay.

(Refer Slide Time: 09:26)



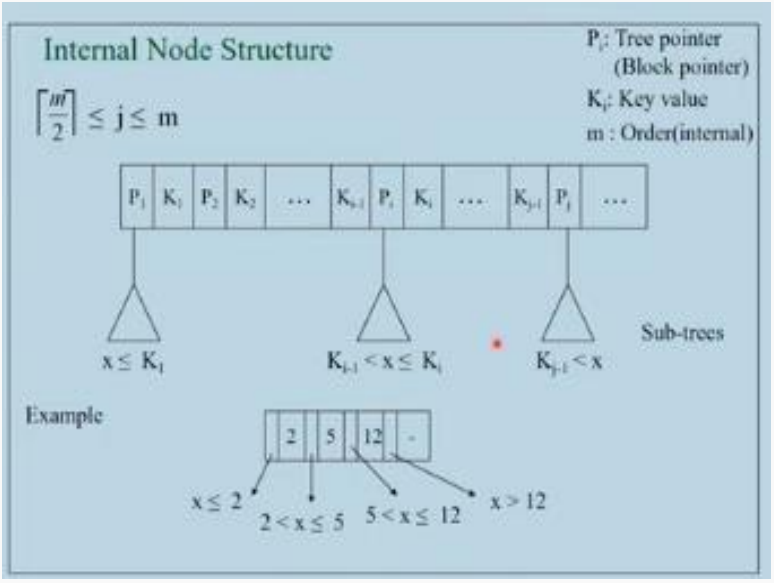
So now let us see there is a notion of order when the term order is used in order to indicate as to what is the maximum number of tree pointers that can be held in a in an intern in a node. So, for an internal node we define order m as the maximum number of tree pointers held in it. So, and if say for 4 pointers are held in it in order to guide the search you only need 3 keys. In a binary search tree you have 2 pointers to search between, to guide the search to either the left or the right you need 1 key.

If it is less than that key, you go left if it is greater than that you go right. So, 1 key and 2 pointers okay so if you want to expand this further now you keep 2 keys then you can handle 3 sub-trees. Okay. So, like that so you will have m is the maximum number of tree pointers held in it and m-1 will be the keys in the internal node. So that is m stands for the number of points. Now for the leaf nodes the order is different, so we write it as m leaf. M leaf of it leaf node is defined like this.

The order of leaf node is the maximum number of record pointers that are held in it. And it is also of course the record pointers and the you know key and the value of the search attribute that has okay the value of the search field and the corresponding record. What do you mean by the corresponding record? The record has exactly that value for that particular attribute like for example you are talking about role number right.

So that record has exactly this roll number. So, roll number and a pointer to the data record. So, for example if you are organizing this index on roll numbers. Okay now these are the orders.

(Refer Slide Time: 11:55)



So let us let me show you the, Raghav focus here. Let us look at the internal node structure. So here is the internal node structure. It is a wide stop here. The node will have huge number of keys and if let us focus on smaller example here. Suppose it has 3 keys then it will have 1, 2, 3,

4 pointers. Okay. So all those records whose value for such field is less than 2, x<2 will be found in this sub-tree.

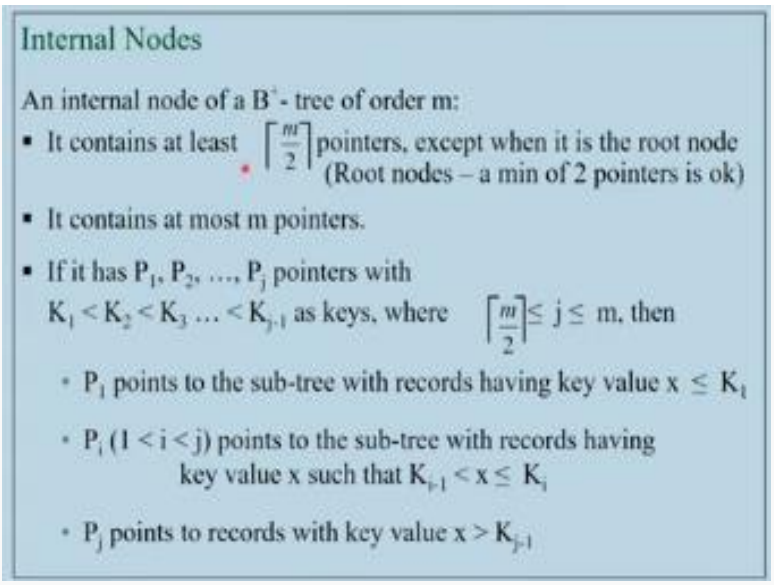
All those records whose value is greater than 2 but less than equal to 5 will be found in this sub-tree and so on. Now if you generalize this situation you have P1, K1, P2, K2 etc. So, these K1, K2, Ki-1, K1 these are all in sorted order. The keys are in sorted order. Okay and then each of these P1, P2 these are tree pointers, we have in fact block pointers because each you know each pointer is a block pointer.

So, this will point to a sub-tree in which all the you can find information about the records that have value of a search field x<=k1 which is a first key. So, between k1 and k2 you will find the in the sub-tree. So, Pi points to a sub-tree where you can find information about all the records in search field values are strictly greater than Ki-1 but less than equal to Ki. Okay. So that is how we will organize these sub-trees and now what is m, m is this maximum number of tree pointers that are there.

So, this j is upper bounded by m and interestingly it is also lower bounded by m by m by 3. So, it should be at least half full this is to ensure that you have balance in the energy. So, it is at least half full and at most n number of tree pointers will be there. Okay. So is this internal structure clear, their structure B, B+-tree, internal node. Now these keys are not giving you any pointer to the data record okay in which this particular value is present.

They are not giving anything like that. They are only telling you that if you are if the search field value is less than or equal to K1 go this side if it is greater than K2 go this side, that is all they said. So that is guiding the search the keys will guide the search.

(Refer Slide Time: 15:16)

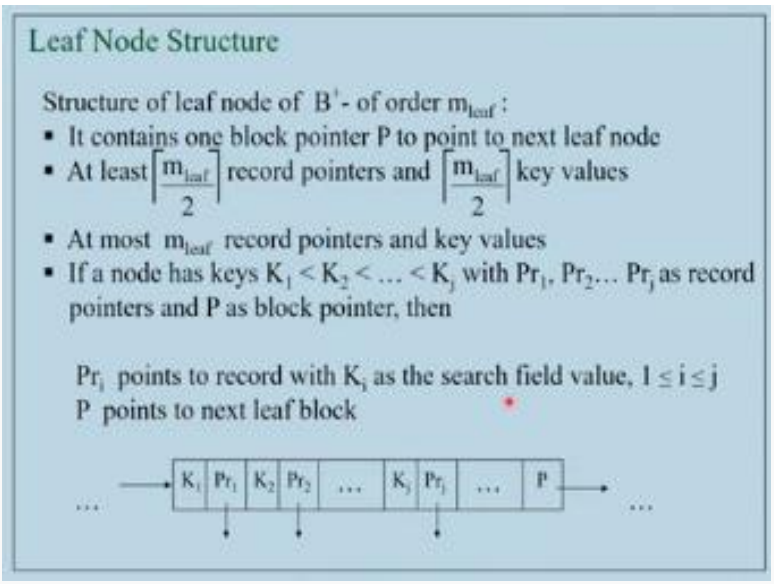


So here is all that information. So internal node of order m will have at least m by 2 number of pointers. We give an exception for the root node the root node is allowed to have just 2 pointers and it contains at most n pointers and these are the 3 pointers say if it has P1 through Pj 3 pointers with K1<K 2 strictly Kj- 1 as the keys then these are the this is how the sub-trees are pointed. So, Pj points to records with key value x greater than this particular value. Okay.

So, this is the scale. Let us proceed. So, this is very similar to the usual search trees, but the only issue here is that we have a huge number of keys. Why is it that we have huge number of keys because we can afford to have space there? When you access a when you access a unit from the disk you always get 4 kilobytes of data. So, if you have that much space and the keys are only some 20 bytes and things like that I can afford to organize a huge number of pointers.

That is the idea. So, let us now look at the leaf node.

(Refer Slide Time: 16:48)



Leaf nodes have a slightly different structure. Okay. Leaf nodes are the ones that they do not have to point to any tree nodes. They are the last. Right. You have reached it. So, the leaf will now have pairs of things. What are the pairs? The key value and the pointer to the actual record in the desk box that will contain that is the record whose value is equal to this key. Right. So K1, Pr1, K2, Pr2, Kj, Prj and the restrictions are like this. It contains. Okay.

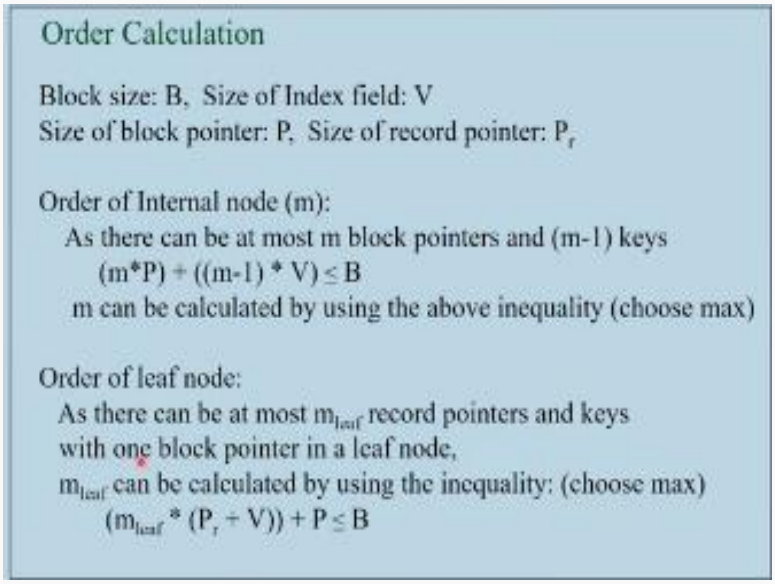
I told you that we are going to chain up these leaf nodes. So it is going to have 1 pointer P which will point to the next leaf node to the right. There are huge number of place leaf, so you start from the left most leaf node then you will have a pointer to the next, point to the next and then you can actually traverse through the leaf nodes starting from left to right. Okay. That is very interesting. Okay.

It contains at least m leaf by two record pointers and m at most m leaf by 2 and in the same number of key values. This is at least right and at most m leaf this is what we had defined as the maximum pointers it can have. So that is the order. So, at most m leaf record pointers and key values and of course the node the keys are ordered K1, K2, Kj and then these are the record pointers.

So, as I was indicating Pri points the record with Ki as it is search field value for all of these things. So, the leaf node structure is clear. The only interesting only aspect that you have to remember is that the key the structure the of key leaf nodes is different from the structure of

the internal nodes. Okay. So that makes def defining the node for the tree as a you know structure a little different. You have to define leafs separately and internal nodes separately and the programs that manage this. Okay.

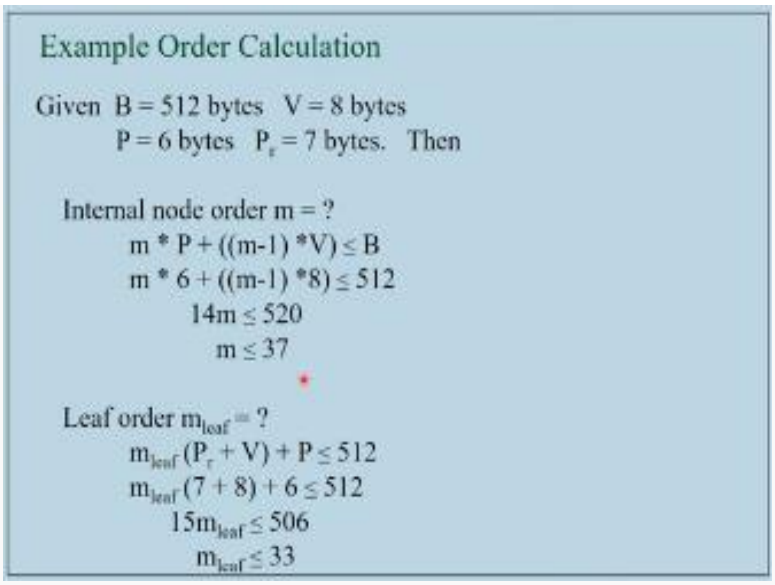
(Refer Slide Time: 19:34)



So now let us get some idea about what are these orders that we are talking about. Let us say block size B is given and the size of the index field is V then the size of the block pointers is P size of the record pointer is Pr. All these are there then the order of the internal node basically is governed by this inequality. So, m times the block pointer plus m-1 keys will be there tree pointers m-1 keys. So, each keys of size V so this sum should be less than or equal to the block size.

Basically this is the inequality that can be used to fix m. Obviously we will choose the maximum value that that satisfies this inequality. In a similar way for order of the leaf node since there are at most m leaf required pointers and keys and at one block pointer so you see m leaf times Pr + V and plus the block pointer. That size should be again less than equal to B. So this inequality that will govern the choice of m.

(Refer Slide Time: 21:00)

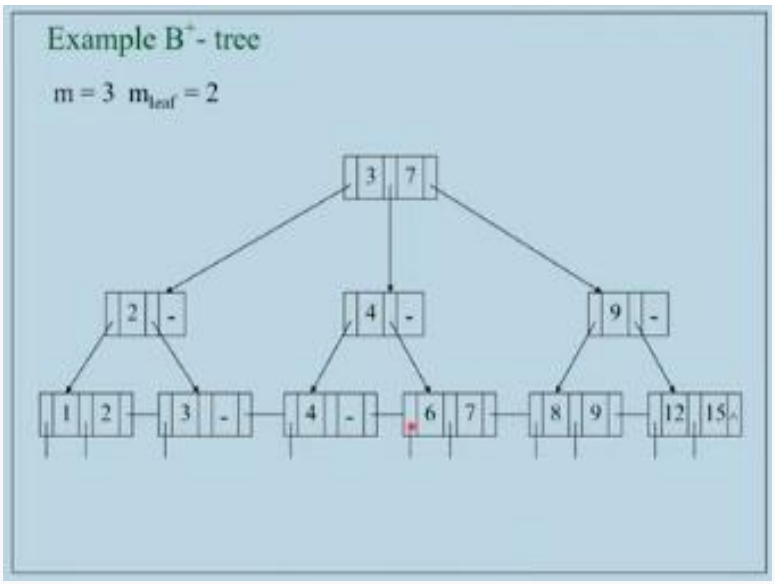


So for some typical numbers like if you take actually half a kilobyte either as a block size and the such key as 8 bytes and the other pointers etc. then you can see that you can later check these values m will be is about the max m is maximum 37 but a half a kilo byte is actually not a typical size for the block size. It will be something like 2 to 4 kilobytes so you can see that this will be actually pretty high.

So you can choose for some say 36 or 37 here and then go hard. So what you are saying here is that each internal node the moment you retrieve it you will get so many children. You can then search through and then find out the appropriate sub P to go down. The leaf node in a similar way you can substitute these values in this inequality, and you will find that m leaf is about 33. So, you can choose this 32 or 33 and then you can proceed to implement that.

Now this one will tell you the usually the B+-trees will have about 3 or 4 as the height because the fan-out is so high then you can handle reasonably big files. Now notice one thing here that nowhere we are using the size of the record itself. The size of the record is not being used at all. This the records are in the in a separate place on the disk. We are only looking at a pointer to them at the key value. No such activity. Okay.

(Refer Slide Time: 23:16)



So here is an example B+-tree structure. I have taken some ridiculously low numbers to illustrate the points bear with me for that because in practice this will be not possible. They will be huge. So, let us take m leaf as 2 that means each leaf will have exactly at most two record pointers and m is 3. That means each internal node will have maximum 3 children and a minimum of 2 children because m by 2 is 1.5 take C it should have minimum of 2 children maximum of 2 children. Okay.

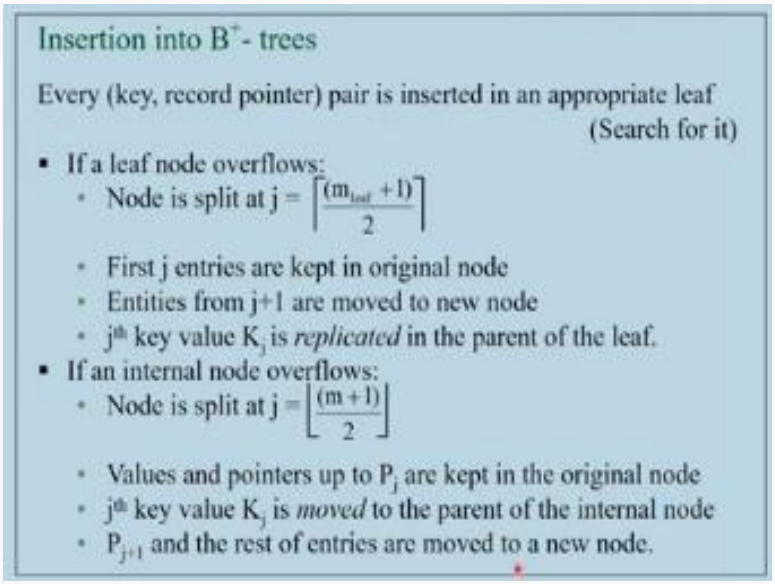
You can see that so the it is it should be obvious now by now that the search field values will get actually repeated inside in the internal node. All right and each of these such field values need not appear but some of them will appear here in internal nodes so that the search can be guided to the appropriate place. Now you can see that this in the leaf you can see that in the diagram here what I have done is to put the key here and the pointer here to it is right.

Okay and this is the pointer that I was talking about which links up the leaf nodes. So, if you start at the left most leaf node and then simply access these records using these record pointers then you will actually get all the data records in the increasing order of this particular field. In spite of whatever be the order in which they have been organized under this. Okay. So, this will this is what I was talking about as giving sequential access sorted access to the data records based on this particular record.

Now searching of course you will take the key and then go here and then check if it is less than equal to 3 then you go this side if it is greater than 3 you go this side etcetera the usual search mechanism you will make use of and then finally when you reach here you will get the record pointer if it exists. It is possible that say for example 5 is not existing. Okay. Is that clear? You have any questions.

Next we will actually discuss how to handle insertions into this structure and how the structure evolves as we insert and actually delete records. Okay.

(Refer Slide Time: 26:36)



So insertion into B+-trees. Now a key and record pointer is what we get. The key and the value of that search attribute. That is what a key. A key and a record pointer is what we get as input and you are supposed to insert that into B+-trees appropriately. Is that clear? Now obviously in order to insert this you have to first search for the appropriate place to insert it.

That means you have you take this key and then go to the appropriate leaf where it is supposed to be present and we will assume that this particular key is not present right now if it is present then there is no question of inserting right. So let us make an assumption that it is not present so you go to the appropriate leaf node by using the usual search mechanism and then insert it there. Okay.

Now various issues will come if there is no problem in the leaf in the sense that leaf is not full then we can simply insert the new key record new pair the key and record pointer into the into the leaf and then be done with that. What if the there are other so if the leaf node has some upper bound on the number of record pointers it can hold. Okay. So, suppose it overflows. It already has a huge number of record pointers and now you are inserting one more so there is a overflow.

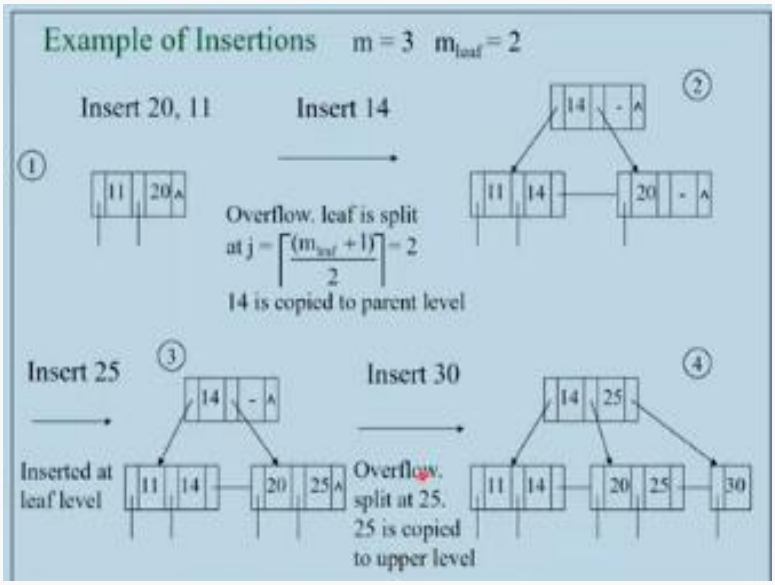
So when there is a overflow what we do is take that leaf simply kind of cut it into half. So node is split at j. So j is m leaf + 1 by 2. So simply to get it appropriate number. So you might question what is sacrosanct about this why cannot I take seal and things like that I mean flow and things like that but let us stick to one set of you know formulas for this. You are getting roughly a number in the middle okay so m leaf +1 by 2 either flow or seal will give you the middle number so we will take this C.

For uniformity sake let us all follow this one. So, node is split at j and the first j entries are kept in the original node okay and then all the entities from j+1 onwards are moved to a new leaf node. Now this new leaf node has to be accommodated into the structure. Okay. So jth key value the Kj is going to be replicated in the parent of that particular leaf node. So, you try to now insert this Kj key into the parent of the original node that you are splitting.

So, I will show you with an example now. Now we will come back to this case it is also possible that while you are trying to insert this keys into the internal nodes the internal node might also work. If there is enough space in the internal node then you are creating a new sub-tree in the internal node but if internal node has already sufficient number of sub-trees then the internal node will also have to be split. Okay.

So we roughly follow this same kind of a policy, for that the node is split at m+1 by 2 floor here in this case and the values and pointers up to that particular Pj are going to be kept in the original node and you can see that in this case the key Kj has to be actually moved not to be replicated it has to be moved to the parent of the internal node. So, I will illustrate this with examples now, we may come back to the slide itself ok.

(Refer Slide Time: 30:55)



Here is a series of these insertions. So, this is a bit of a complex slide. It has 4 pictures on it. So, follow it carefully. So, the pictures are numbered 1, 2, 3, 4. So, we start off with an empty structure which has just you know and then we try to insert 20 and 11 into that and remember that m is 3 and m leaf is 2. That means we can only have 2 record pointers. So, this is the so if there is just two records then you have just one leaf now.

You try to insert 14 into this so the appropriate place for this 11, 14 and 20 and there is no space here. So now I have to split it. So I will split it in the middle so middle is m leaf +1 by 2 which is 2. Ok. So, I keep 11, 14 here and 20 in the new leaf and then move this 14 which is this Kj and make a new node the parent of this particular leaf. It did not have parent. So now we are creating a parent and then insert this 14 in the in the parent now and then set the appropriate tree pointers.

So now you can see the 14 will start guiding this search. If they are less than equal to 14 you come here if it is greater than 14 you go there. All right. So leaf nodes if it is split then we take the middle key and then if it has a parent we will try to insert it into the parent if the node does not have a parent we will create a parent and insert this 14 into that and then set the tree pointers of it.

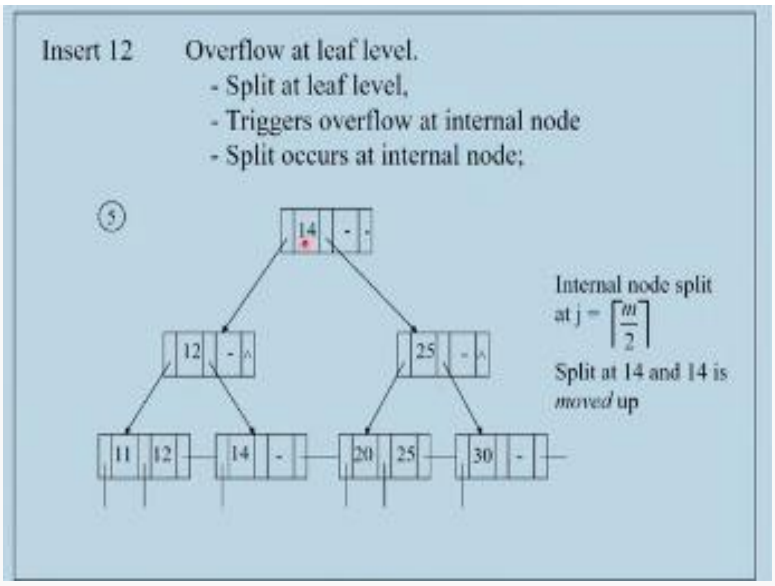
Now let us see in this structure node we will try to insert one more one more record. Let us try to insert 25. So you can see that you come to the root and then check that 40, 25>14 so you go to this place then fortunately there is space here because it can hold up to 2 right. So, you go and simply insert it there. So, you inserted it. So, no problem in this.

Now try to insert 30. So obviously in this case the search will guide to this leaf node and you are supposed to insert it here 20, 25, 30 but then this has to be split. So you split it so into 20, 25 and 30 create a new leaf node and they take this 25 and replicate it for the parent. The parent already exists so you will go and insert this 25 into the appropriate order and then set the new pointer to the okay pointer to the new leaf.

So when you take 25, less than or equal to 25 was supposed to be there here right so setting this 25 here you know fits our policy. Putting this new leaf sorry new key and putting it in the parent appropriately will keep our policy but the sub-tree will have keys which are less than or equal to this and that the new leaf has been created so this 25 and the pointer to the new leaf will be passed up.

So, when you insert 25 to its right you will insert this key at this point so that will take care of this new leaf. Okay. That is how insertion takes place. Of course, I am not giving you very detailed pseudo code for insertions, but I want to give you an idea about how insertions and deletions take place in the B+ trees. Okay. Now let us see to this structure try to remember this okay 14, 25 like this so now we try to insert 12.

(Refer Slide Time: 35:57)



Try to insert 12. So, 12 if you try to insert 12 it is less than 14. So it should come here so this should come as 11, 12, 14 right and so this cannot accommodate 3 so it has to be split and if you split then there will be 1, 2, 3, 4 keys 4 leaves and any internal node cannot handle 4 leaves, it can maximum handle 3 children. So the this node has to be split. Okay this node has to be split. So, this will be first split.

So 11, 12 and then a separate leaf with 14 will come into picture. Okay. So there are now 4 leaves 11, 12, 14, 20, 25 and 30. 4 leaves that comes in picture. Now this 12 you try to insert here. So, it comes as 12, 14, 25 it comes but then it cannot accommodate it. So this will be split, when you split this 12, 14 will go into one node and 25 will go into another node. Okay.

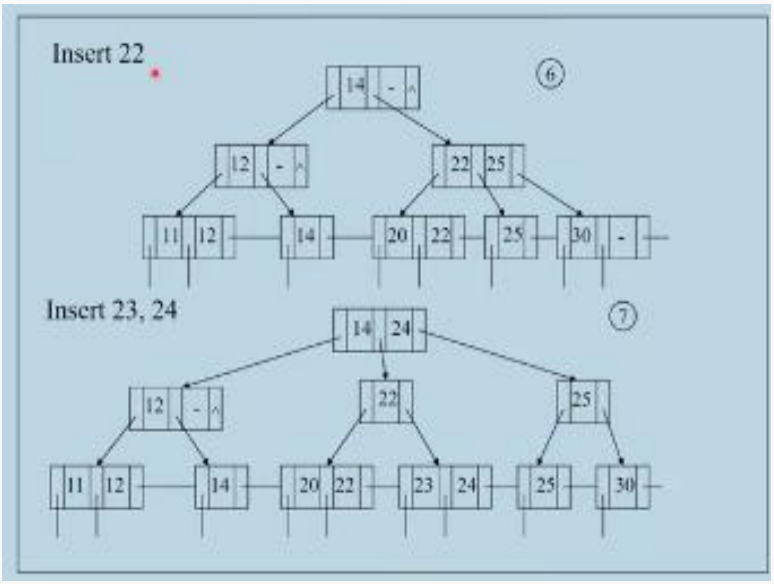
Now but in this case what exactly happens is that the middle key 14 is actually moved up the middle key since you could not accommodate 3 keys we will find the middle key and then move it up and since there is no parent for this the parent will be created and then 14 will be inserted here and the left will come here right will come. Okay. So, this is a new so the whole structure has gained height now.

We have a new root node, so they split at leaf level triggers overflow at the internal node and that split again that triggers a split at the internal node and when you split an internal load the middle key will be actually moved to a new level and then the pointers in this set of node level.

So remember this 14 has to be moved. There is no question of replicating because in the internal nodes there is no replication of keys.

So we will move it up and then set the pointers. Good. So, this is how the structure can gain height as it evolves.

(Refer Slide Time: 38:56)



Now try to insert 22 into this insert 22 into this. So sorry insert 22 into this it will 22 > 14 so you come here and then 22 < 25 so you come here you try to insert here and then the usual policy 20, 22, 25 so and then 2 leaves will be created and then 22 will be moved up so there is space here so 22 will come here. So, let us see the structure. So 20, 22 will come here 22 is replicated up and this new point the point will get adjusted.

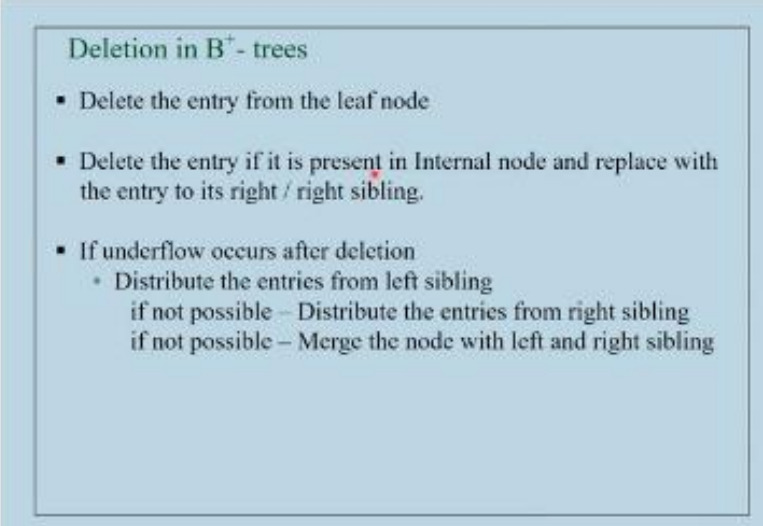
Now try to in this let us try to insert 23 and 24. 23 will come here, luckily there is space here so you can simply insert it here. When you now try to insert 24 again that also comes tries to come here so there will be again split here so that will you know cause actually a split of the internal node because a new 4 leaves have come into picture. So that will create a split here.

So you will now see that that split will not propagate further so 22, 25 are created the middle one 24 which we are trying to insert is actually moved up and then in appropriate it is put at an appropriate place so that this search can be guided to the appropriate sub-tree. Now you can

see that 24 is the maximum value in the sub-tree that is pointed to by this particular point. So that property is kept as it is okay.

So study these insertions carefully there is a you will find detailed pseudo codes for this insertion algorithms.

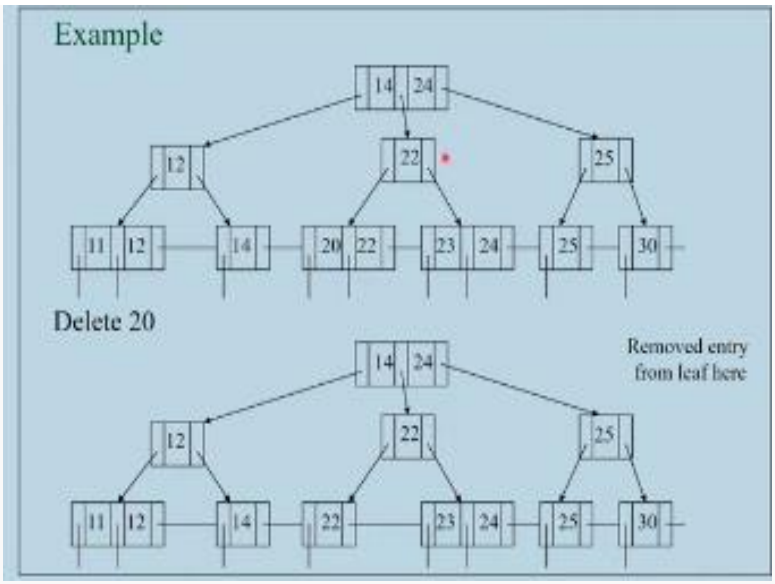
(Refer Slide Time: 41:03)



Now deletion you know has to be also handled. So, to delete the entry so supposing you get a delete request then obviously you should first locate it as to where it is in the leaf and then delete it from the leaf. Okay. Delete the entry if it is present in the internal node also and replace it with the entry to its right on the right side leaf we will see how exactly it is a so. So, underflows can occurs actually while you are deleting, in which case we actually you know redistribute the keys.

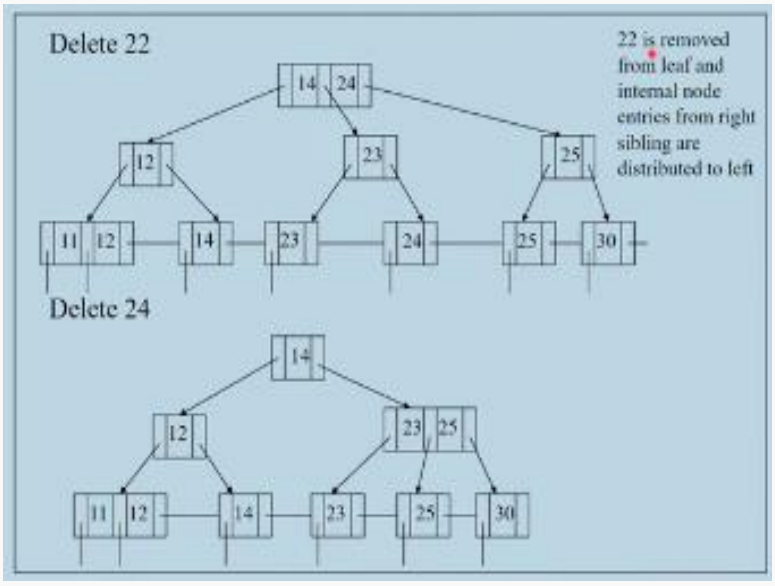
So we will try to borrow keys from the left sibling and if it is not possible borrow keys from the right sibling, it is still not possible we will merge the nodes but the details I am skipping but let me show you some examples.

(**Refer Slide Time: 42:00)**



So, let us say we have this kind of a B-tree I mean B+-tree. Let us various okay 20 is here so we are trying to delete 20. So, in this case we will go to the appropriate leaf and 20 and the record pointer will be deleted but fortunately this has another key, so no underflow occurs here. So, we can simply retain this leaf as it is. So, this structure does not change much. So, 22, 23, 24 that is the whole thing will be as it is.

(Refer Slide Time: 42:46)



Now let us take this and then try to delete 22 okay in this structure let us try to delete 22. If you delete 22 then this leaf itself goes. So, if this leaf goes then what is the point having this because if it loses one child so it cannot have less than that many number of children less than 2. So, something has to be done about this. So what we can do is to minimize the changes to

the tree what we can do is instead of you know removing this leave we can actually try and look at the left sibling I mean this is right sibling here right sibling and then borrow a key from there.

Suppose you borrow a key from there then I can retain this leaf as it is okay, and I do not have to do any much changes. Of course, since 22 has gone now so I should replace it by whatever I borrowed say I can borrow 23 from here and then replace this 22 by 23 then everything will be alright okay. So that is what we do when so try to minimize the changes to the tree so borrow a key from the right sibling and adjust it.

So, this is how we have done that. So, 22 is removed from the leaf internal node entries on the right sibling are distributed to the left. So, under this pointer this of course if you change this then you should change this also because this is they should will be maximum in the sub code. Now let us see if what if what happens if you try to delete 24 then there is no chance of these people helping each other.

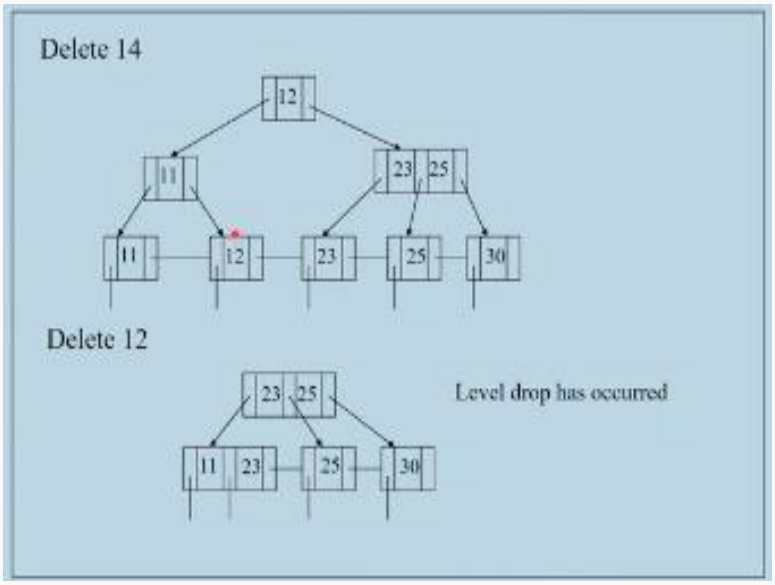
So in this structure let us delete 24 and then what exactly happens. So if you delete 24 then this fellow does not have it this leaf cannot borrow from 23 because that already has only one record. So basically, now this and this have to match. This internal node and that internal node. So basically, when you have deleted 24 then this has this particular internal node has only one leaf which is not allowed.

So, what these two people do these two people will collaborate and then we redistribute the sub-trees among them. So, if they cannot do they will simply merge so they will merge into one tree and then take care of the sub-trees. So, 23, 25, 30 are the 3 sub-trees to be taken here. So we will simply merge these 2 will merge these two and then not disturb the tree beyond that.

Of course, this 24 has to be deleted because 24 has been deleted. So now merge these two and then retain the pointer as it is then the search will be appropriately directed the greater than 14 will come here and 24 here is not in the picture. So, like this we can basically follow the policy

of borrowing from the siblings or cooperating with the siblings in either taking the sub-trees or merging the subtrees together.

(Refer Slide Time: 46:34)



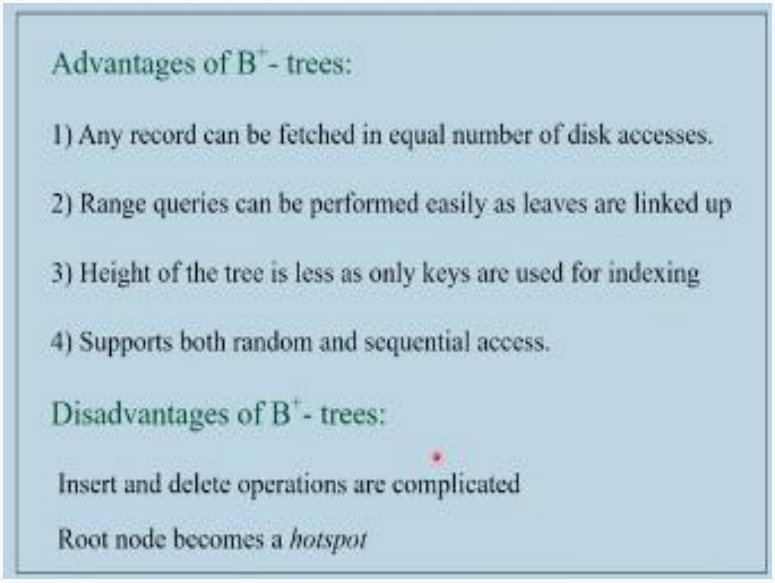
So, let us try to delete 14. So, in this the root has 14 let us try to delete 14. So, 14 is here so if I delete 14 then fortunately its sibling has 2 keys. So, I can borrow one key from here right if I borrow then this will change. If this changes this also has to change. So, this is how so you can change it appropriately keep the structure but then change it change the values. Now if you try to delete 12 in this structure then there is really no scope.

So actually, you can see that there will be level drop, the root will itself go until a new root will come into picture. So if we delete one now this fellow will have only one child which is not allowed okay and now you try to merge it. If you try to merge it then this fellow will have only one which is not allowed so then basically you will have to drop this parent and then adopt this as the new parent and then adjust the tree again.

So, this is how the B+-tree can actually lose height as it goes. Okay. So with this I have given you a an idea how the B+-trees will help us in organizing a dynamic file and they will they can be they can expand and shrink and then they will give two kinds of accesses on the search key it will give you a quick access what is the quick how quick you can get it is the height of the structure that many number of block axis you have to do and that is same for all case because all the leaves are at the same level. Okay.

So you will get quick access to a record and even if you traverse from the left to right on the on the leaf level then you will get access to all the records in the increasing order of that particular structural value. So, these are two things that you can do.

(Refer Slide Time: 48:58)



Only one issue with the B+-trees is that all the accesses have to go through the root. The root block of the structure is the most sought-after thing everybody has to start their search through the root. So, the root becomes a in some sense a most sought-after node block and kind of becomes a hotspot because we will later on see that if you want to change anything you have to lock things.

So if you lock it then nobody else will access it so there will be a lot of issues. So, it becomes a hotspot. So that is the only disadvantage. Let me just okay so we will stop here for today and I will see you tomorrow evening.