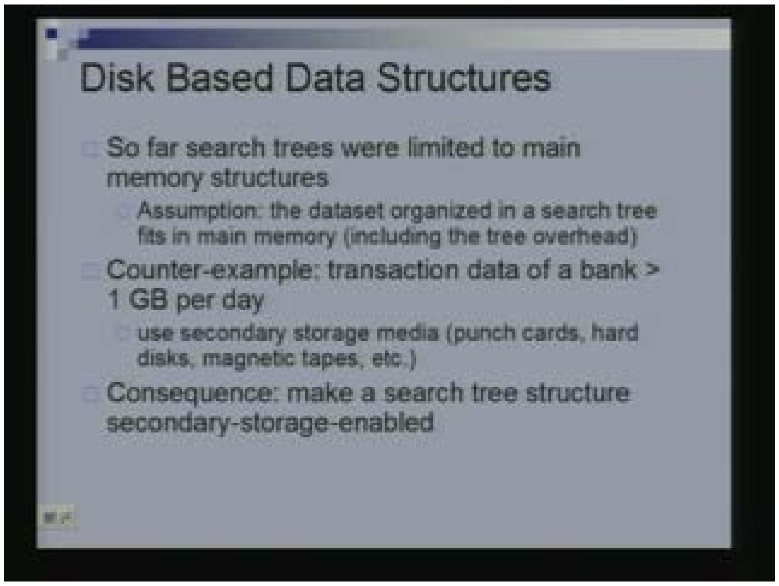
Data Structures and Algorithms

Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 16
Disk Based Data Structures

So today we are going to be talking about disk based data structures. In last class we looked at ab trees.

(Refer Slide Time: 01:17)



Disk Based Data Structures

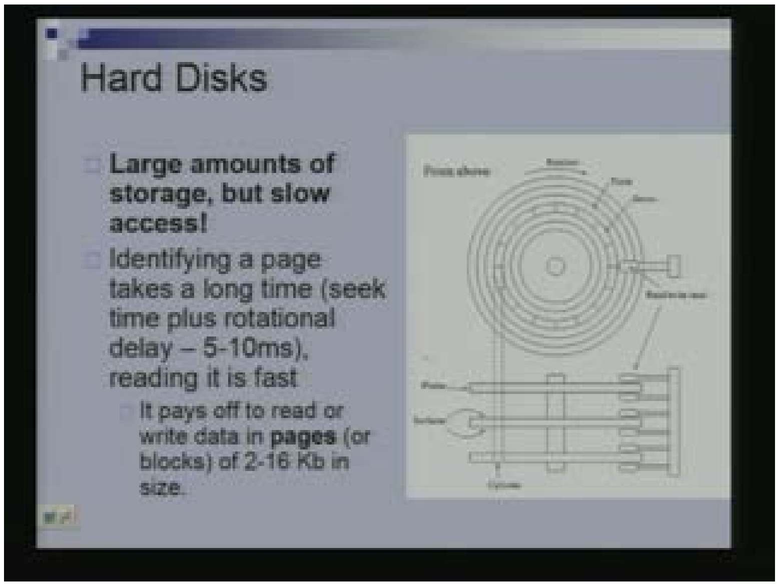
- So far search trees were limited to main memory structures
  - Assumption: the dataset organized in a search tree fits in main memory (including the tree overhead)
- Counter-example: transaction data of a bank > 1 GB per day
  - use secondary storage media (punch cards, hard disks, magnetic tapes, etc.)
- Consequence: make a search tree structure secondary-storage-enabled

So these were the extension of the 2-4 trees that we had seen. Today we are going to look at disk based data structure and in particular we are going to look at b trees and we will see that they are very similar, they are in fact ab trees for specific value of b. So now we are looking at is when we have a large amount of data over which to search. Till now all our search trees were limited to main memory, in the sense you build a binary search tree or build a 2-4 tree or a red black tree so we had this nodes which where objects in memory and you know the references corresponds to pointer addresses or memory addresses.

Now we are looking at the setting where we have the huge amount data. Let's say some kind of transaction data which could be bank share markets, you know setting we are large amount of data get generate and such huge amount of data is not stored in the main memory of computer. This is typically stored on disk and now you want to be able to search through this data or insert something in to this data or modify this data. How do you do that? So you just imagine the setting where you have let's say particular bank which has records of each of its customers. So they could be a million customers each of those records would be huge. The data should set in the each account should be huge because it would have all the transaction data associated with what has been history of transaction and so on. You can't expect all of that data to decide in the main memory of the computer so it would be kept on disk and now suppose type in particular account number you should be able to retrieve that particular account and the data associated with that account.

So question is how are going to do this? So we want to make a search tree which is some sense secondary storage in avl tree which will help you even when the data most of the data is stored on disk will still be able to search through it. Now the problem here is that the data is stored on disk that's fine but even the index that we are building. So what do you mean by index? The search tree, search that we have so huge that we cannot expect the entire search tree to fit in to the main memory. We will come to this in to the short while. So what's the problem in you have think on the disk verses when thinks are the main memory of the computer.

(Refer Slide Time 04:01)



Hard Disks

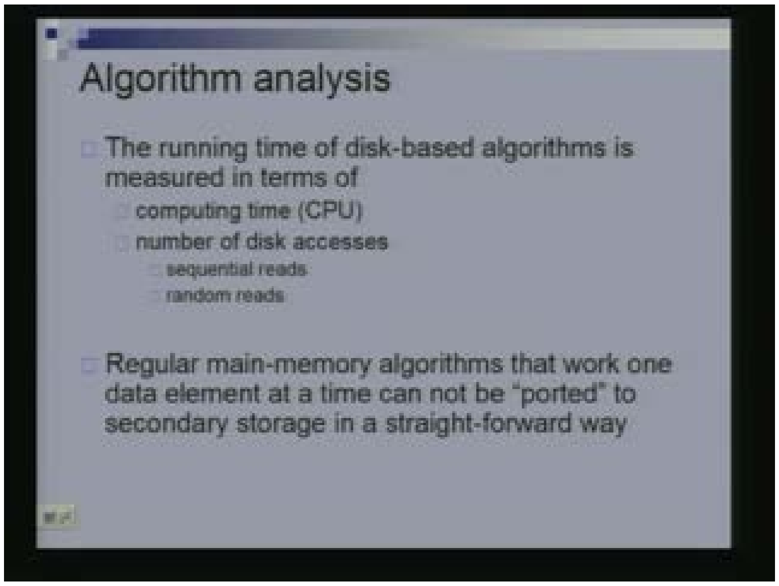
- Large amounts of storage, but slow access!
- Identifying a page takes a long time (seek time plus rotational delay – 5-10ms), reading it is fast
    - It pays off to read or write data in **pages** (or blocks) of 2-16 Kb in size.

One big problem is that disk access is very slow. How is disk accessed? You have let's say this is the disk, so you typically have bunch of disk in stack one and top of each other. Each one of them has its own read write head, head which will traverse which can move along this disk and get to a particular track. So these are the track on the disk the disk rotates in one particular direction and these read write head can decide that it wants to go on this track or this track or this track and so on. When you have to read a particular location so a track is further divided in to sectors. So when you have to read a particular sector, the disk rotates and this read write head gets to the appropriate tag and then starts reading the data from there.

So significant fraction of the time is spent in this read write head moving determining which track it has go to and moving and the rotation. So one of this is called seek latency, so this that time that is required for the head to get to the appropriate tag its called seek latency and the other one is called rotation latency. The time required for the disk to rotate so that the head is positioned at the right place, the right sector and once you have write at the right place then the entire sector is typically read. So one sector or let's say one let's call it as page so the data then is read in units in larger units. You don't read one byte of data from the disk. Why don't you read

one byte of data from the disk? Because you already spend so much in getting to that one byte then might it is well read whole lot of bytes. So you typically read or write in units called pages which are you know it could depend upon the particular computer system but it could be in this range 2 to 16 kilo bytes.
So you read this much amount so this could be one page setting here and you read this much amount of data and it will be moved in to the main memory. This is reading, when you writing similarly you would write back and entire page. What are the problems associated with this? We have to organize our search tree in such a manner that first it can sit suitably on this disk as I said or search structure itself is going to be so large that it cannot fit in to the main memory and that's what we are going to do in next slides. So when you have the disk based algorithm like this, the running time is going to be measured in terms of the time taken by the CPU and the number of disk access.

(Refer Slide Time 07:11)



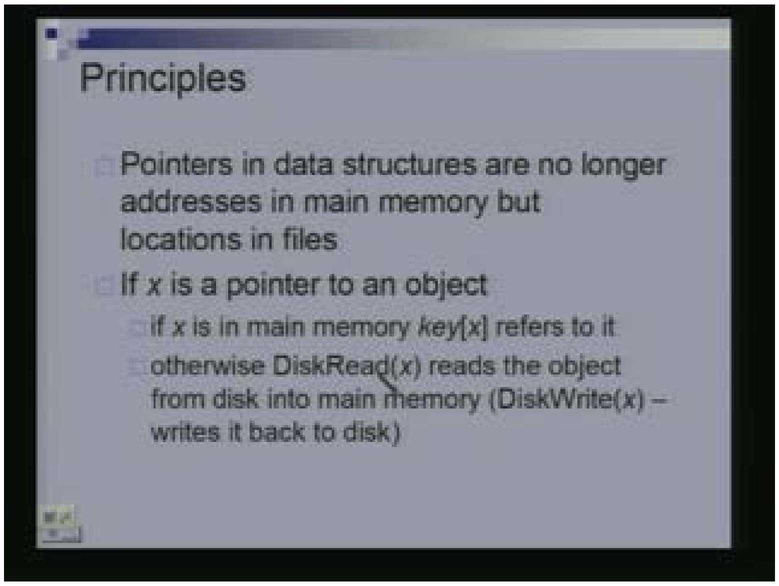
Algorithm analysis

- The running time of disk-based algorithms is measured in terms of
  - computing time (CPU)
  - number of disk accesses
    - sequential reads
    - random reads
- Regular main-memory algorithms that work one data element at a time can not be "ported" to secondary storage in a straight-forward way

Now this time, the time taken by the CPU is insignificant compared to the number of disk access in the time spend in each disk access. So what you would do in most of these algorithm is to organize the data in such a manner that there is a number of times you have to access the disk is very small and what I have said is you know till now we had seen the algorithm which are called main memory algorithm where the entire data sets in the main memory. So they cannot be easily ported to this module when part of the data sit on the disk, it cannot be ported in the straight forward way. What are the problems that are going to happen now that we have data setting on the disk? So one is as for as this pointer business is concern. Pointer is to same as references to objects. So till now we had reference to an object, we knew that correspondence to the address in the main memory. So you would go and access that location in the main memory.

Now if part of your data structure is sitting on the disk, so just imagine that your red black tree or whatever it was, part of it was in the main memory and part it was sitting on the disk. One of those pointer, one of those references is referring to a location on the disk now. So if it is referring to something in the main memory then you can go and get it at that particular memory location but if it is referring to something on the disk then you will have to do something lets use the operation disk read to read the particular data from the disk and disk write to write bind of the disk. We have to use such kind of operation to be able to access the data, I will come to what key means in a short while.

(Refer Slide Time 09:15)

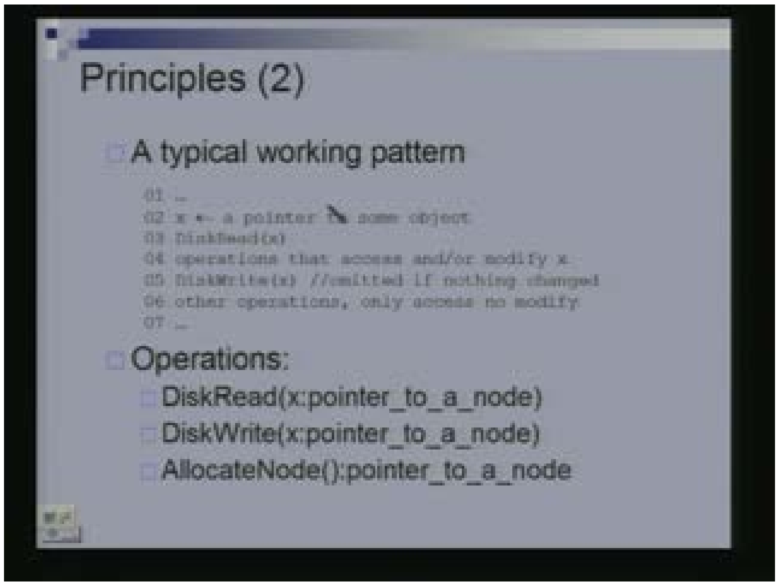


Principles

- Pointers in data structures are no longer addresses in main memory but locations in files
- If *x* is a pointer to an object
  - if *x* is in main memory *key*[*x*] refers to it
  - otherwise DiskRead(*x*) reads the object from disk into main memory (DiskWrite(*x*) – writes it back to disk)

So one big problem is that you know the pointer now have to be translated suitably. If the pointer is just pointing to a memory location then it is easy, you just get to the memory location and do your job
but if its not pointing to a memory location, if it is referring to something that sits in to the disk then we have to first fetch that block of data, that page of data from the disk in to the main memory and then you can access that particular object. So typical pattern would look like something.

(Refer Slide Time 10:01)



Principles (2)

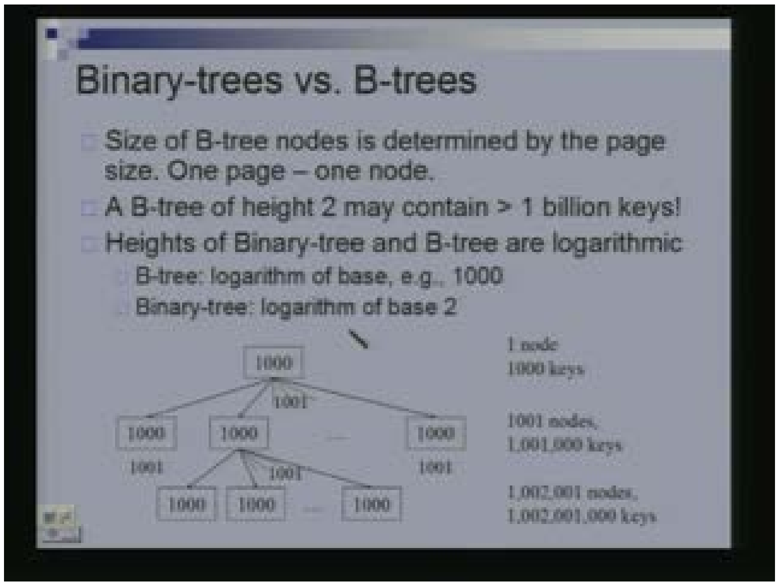
- A typical working pattern

```
…
x ← a pointer to some object
DiskRead(x)
operations that access and/or modify x
DiskWrite(x) //omitted if nothing changed
other operations, only access no modify
…
```

- Operations:
  - DiskRead(x:pointer_to_a_node)
  - DiskWrite(x:pointer_to_a_node)
  - AllocateNode():pointer_to_a_node

So x is the pointer to some object so you would read this object from the disk you will do some operation on this object and then you will eventually write it back and you might omit this step if you did not modify the object at all.
So we are going work with two operation today, disk read and disk write, reading a block from disk and writing block back to disk. So now let's come to what a, b tree is.

(Refer Slide Time 10:32)



Binary-trees vs. B-trees

- Size of B-tree nodes is determined by the page size. One page – one node.
- A B-tree of height 2 may contain > 1 billion keys!
- Heights of Binary-tree and B-tree are logarithmic
  - B-tree: logarithm of base, e.g., 1000
  - Binary-tree: logarithm of base 2

B tree is a same as your ab tree. Here I have drawn an example where in this ab tree each node has 1000 keys so you value of a here is at least thousand lets say is a 1000. So recall in an ab tree, each node has at least a children and at most b children. So here I have taken my value of a as 1001 so each node has at least 1001 children. So here I have just tried to illustrate why is this data structure now useful for disk based access. So you have this first node which has 1001 children right and since it has 1001 children how many keys would it have inside it?

1000 keys and just I have organize it very uniformly so that then this each of this each node has also 1000 children and then further more these leaves so its just two level trees these leaves also have 1000 keys in them. So how many keys are there in all in this? So that's a 1000 plus million plus a billion. So there will be so many keys in this entire two levels structure. So let's say I had a data base with so many records, so many different accounts so I could put those of this each of this accounts, records you know lets say the key just the account number so i could put all of those keys in to this kind of a structure. Now each node had contain 1000 keys. Now this entire structure cannot fit in to main memory, you can see this is already 10 to the 9 keys write which each of this keys there is associated with the pointer. Each of the key itself could be let's say 4 bytes of memory, so 4 bytes of memory let's say 4 bytes of pointers is about 8 bytes per key so that's 8 bytes times a billion that's 8 giga bytes.

So you need that kind of space just to keep this data structure in to a main memory and here we have not said anything about the data associated with this records. Each of those data themselves could be 100 mega bytes because my account has listed with all transaction that have been done on the that account lets say the last three years or four years or some such thing that's huge amount of data that's lets says stored on the disk we are not even bringing that in to the main memory. But what I am pointing out first is even if the actual data was stored on the disk just to be able to access the data, just using keys and pointers is so huge that you cannot have all of it in to the main memory.

Now why is this kind of structure useful? We are talking about this base, so what we are going to have now is that we are going to have this structure this called b tree. This structure itself would be kept on the disk, so each of these nodes would now be one page on that disk and it is just this very top node, the root node which will be kept in the main memory. Yes, now what happens if you have to search, what would you do? You will determine which of this 1000 keys between which two keys your particular key lies in.
Suppose it lies so and then go to the appropriate child node.

Suppose this is a child we have to go to, now what will you do? You will access this node from the disk and bring it in to memory. from here you will determine key which should be your next node suppose it is this node so you will access this node from disk and bring it in to main memory and from here you will be able to find your account number and then you will have to follow the account data, let's see. So how many disk access would you need? One to get this, one to get this and one to may be get the actual data associated the information associated with that particular record that you are trying to access.

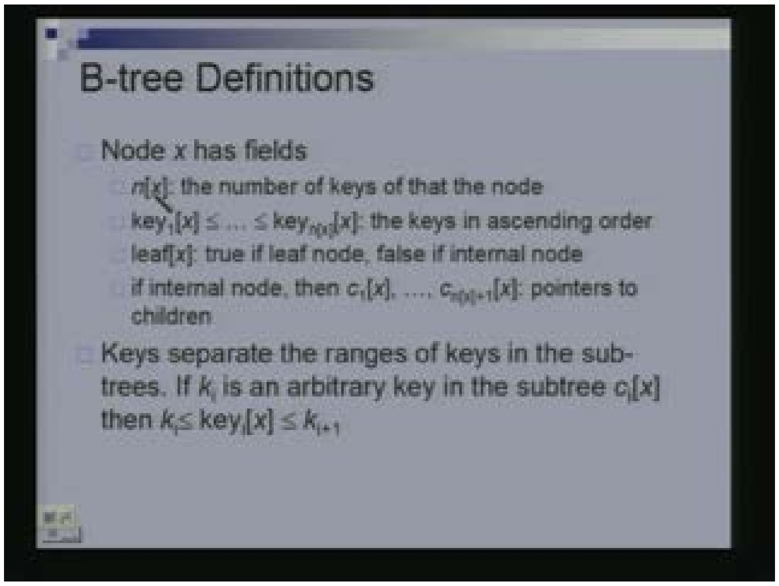
So how much disk access, how many disk access do you need? Height of the tree. So to reduce the height of the tree what do you want? So you want as little height as possible for the tree, yes so how can you reduce the height of the tree? Increase the value of a, increase your whatever the number of children each of this nodes have (Refer Slide Time 15:50) but what is limiting you from number of children each of this nodes and page size? Instead of 1000 why didn't I put in

10000 key here or 100000 keys in here? That would have been even better, that would reduce the height even further. If I had, that would reduce the height even further but i can even put too many because each of those keys is taking certain amount of space and i can only put as many keys as can fit in to one page and what is the page? Page is a unit of transfer between the disk and the main memory.

So that is define for the particular computer system. [Hindi conversation]. So depending upon if each of your keys comma pointer per taking let's say 8 bytes, your page size is 16 kilo bytes each of key comma pointer pair is taking 8 bytes that you can put at most 2000 keys in to one page and so that determine [Hindi conversation]. So this is just read writing what the node?

So x is now referring to an particular node so what does the node x have? n [x] is the number of keys in that node and then once again the first key is going to be less then the second key is going to be less than the, so key one of the node is less than the key two of the node and so and on.

(Refer Slide Time 17:08)

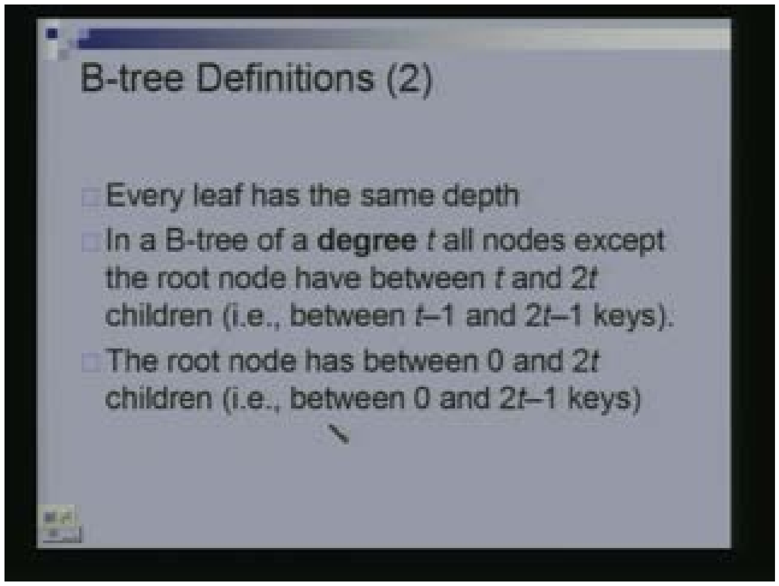


B-tree Definitions

- Node *x* has fields
  - *n*[*x*]: the number of keys of that the node
  - $\text{key}_1[x] \le \dots \le \text{key}_{n[x]}[x]$: the keys in ascending order
  - leaf[*x*]: true if leaf node, false if internal node
  - if internal node, then $c_1[x], \dots, c_{n[x]+1}[x]$: pointers to children
- Keys separate the ranges of keys in the sub-trees. If k_i is an arbitrary key in the subtree $c_i[x]$ then $k_i \le \text{key}_i[x] \le k_{i+1}$

So there are n of x keys and they will have lets say methods or particular bit in a node which specify whether the node is a leaf node or not. Whether it is has any further children, so leaf if this is true then that means leaf node other wise not and if its not a leaf node it's a internal node then we will have, if there are n of x keys then will have n of x +1 children of that node and this we all know we have seen this before if k sub i is in z key in the sub tree c sub i in the case of i is less than the ith key in the node and so on. So it's the same.

(Refer Slide Time 18:05)

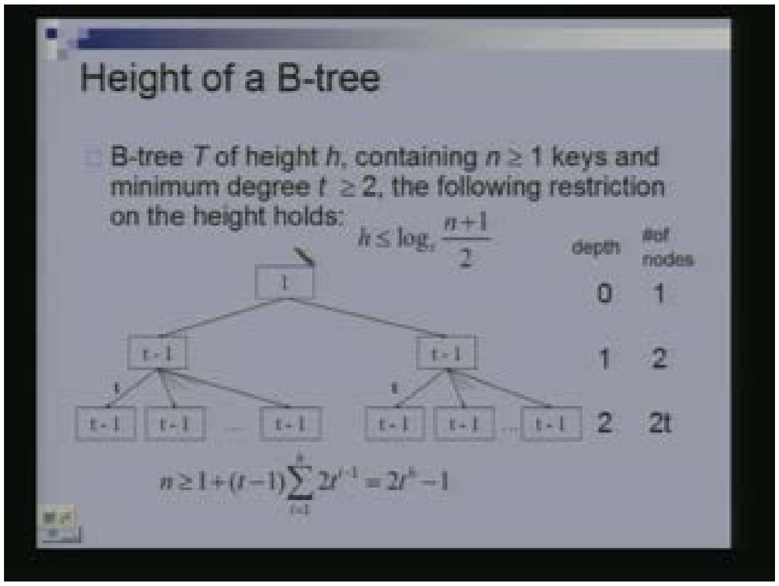


B-tree Definitions (2)

- Every leaf has the same depth
- In a B-tree of a **degree** *t* all nodes except the root node have between *t* and 2*t* children (i.e., between *t*–1 and 2*t*–1 keys).
- The root node has between 0 and 2*t* children (i.e., between 0 and 2*t*–1 keys)

So b tree is the essentially same structure as you ab tree. Now only difference is so for as ab tree we say we provided a lower bound on the value of b, we said b is should be at least 2a-1. What are we saying today we want that b should be? So if a node has degree t so t is the same as that we had talked off all nodes except the root had between t and 2 t. So this so we want that b, so today we want basically this is out bound for b so we were working with b equals two times a exactly. So b tree is essentially special kind of ab tree with ab with bb exactly two times a. So each node has between t and 2 t children and so it has at least, it may be between t -1 and 2 t minus one keys. So the other way in which b tree differs from a, ab tree is while ab tree is meant to be a data structure meant for an internal memory, b tree is a data structure meant for secondary storage. So you really have to choose large value of b so that height of the tree is smallest possible and the entire node, all the key is in the node can fit in to one page and the same as before the root node has between 0 and 2 t children. Actually i have said zero but you know, it need not be zero it could be 2 between 2 and 2t children, let’s make it 2 and 2 t.

(Refer Slide Time 20:08)



Height of a B-tree

- B-tree *T* of height *h*, containing $n \geq 1$ keys and minimum degree $t \geq 2$, the following restriction on the height holds: $h \leq \log_t \frac{n+1}{2}$

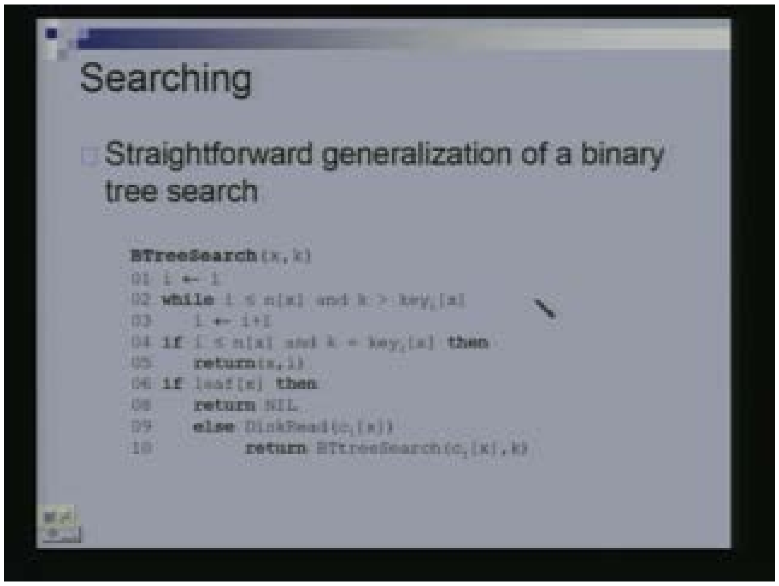
1
t - 1
t - 1
t
t
t - 1
t - 1
...
t - 1
t - 1
t - 1
...
t - 1

depth	#of nodes
0	1
1	2
2	2t

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 2t^h - 1$$

So the root node in this example has exactly two children and then this is t -1 keys and let's say t children. So this is the setting in which the height of the tree is as large as possible because each of the node has only t children. So if this is all just making sure that you remember things from the previous classes. There are one node here, two nodes here, 2 t nodes here and so on. So this is goes up to height h, you can compute you know this would be the total number of nodes that would had and since n is equal to this that will give you the height of the tree as this quantity. So I will just show you some pseudo code also for searching so that you know that these things are completely clear in your head.

(Refer Slide Time 20:53)



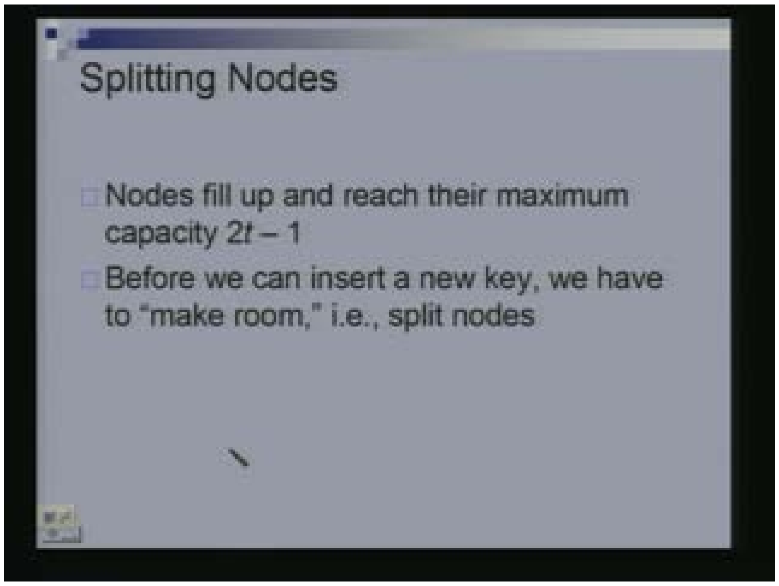
Searching

- Straightforward generalization of a binary tree search

```
BTreeSearch(x,k)
i ← 1
while i ≤ n[x] and k > key_i[x]
    i ← i+1
if i ≤ n[x] and k = key_i[x] then
    return(x,i)
if leaf[x] then
    return NIL
    else DiskRead(c_i[x])
         return BTtreeSearch(c_i[x],k)
```

Suppose I am searching for key k in a node x. So this will be a recursive search procedure. What am I going to do? I am going to first find out the key. What should I do? I first have to find out the key, the first key which is larger than k, the keys are arranging let's say increasing order in a node. So you have to find a first key which is larger the k so that's what I do here, I keep moving till I find a key which is larger than k then I come out. So if I found a key which is larger than k let's say that will give me the i^{th} key. So if I found exactly the key I was searching for then I return if the node that I am in was leaf node that I can't procedure any further then I return else either then the key is not there else I fetch the next node. I fetch the next node and i continue my search in the next node. What you have to do here? You were searching, this is let's say c_i (x), we have to know excess the particular node. So c_i (x) is just the reference to the child node, the appropriate child node, you just have to access that. I am going to skip this.

(Refer Slide Time 22:54)

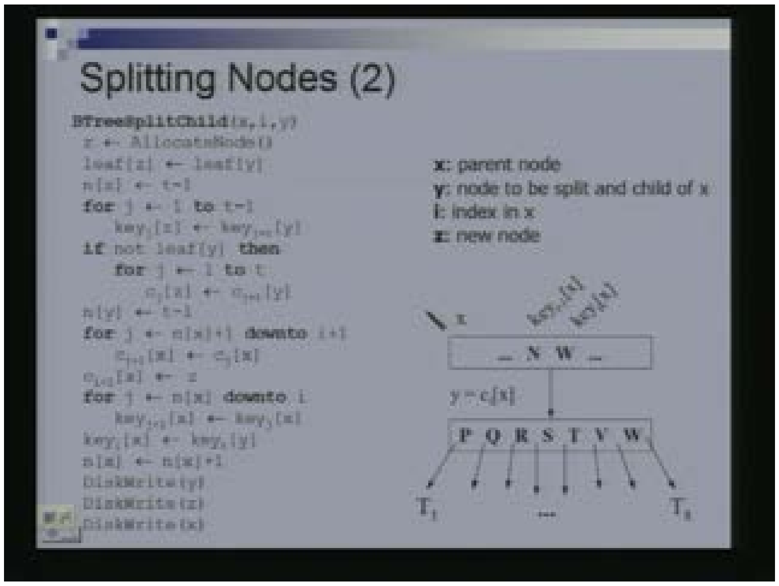


Splitting Nodes

- Nodes fill up and reach their maximum capacity $2t - 1$
- Before we can insert a new key, we have to "make room," i.e., split nodes

So splitting nodes the same idea is before. When you split nodes? When they are full so how many keys can sit in a node? We said 2t, 2t-1. It can have 2t children so it can have 2t -1 keys. So if it has 2 t - 1 keys and new key if we are trying to insert a new key then we split the node and what was the process of splitting, if this is the node that I am splitting let say t q value is 4. So this already has seven keys if I am trying to put one more here then I need to split it so s goes up and this gets split in to these. So I have put down the code here, you can have a look at this slides separately to understand this code. I am not going to spend too much time during it here.

(Refer Slide Time 23:38)



Splitting Nodes (2)

```
BTreeSplitChild(x,i,y)
  z ← AllocateNode()
  leaf[z] ← leaf[y]
  n[z] ← t-1
  for j ← 1 to t-1
      $key_j[z]$ ← $key_{j+t}[y]$
  if not leaf[y] then
      for j ← 1 to t
          $c_j[z]$ ← $c_{j+t}[y]$
  n[y] ← t-1
  for j ← n[x]+1 downto i+1
      $c_{j+1}[x]$ ← $c_j[x]$
  $c_{i+1}[x]$ ← z
  for j ← n[x] downto i
      $key_{j+1}[x]$ ← $key_j[x]$
  $key_i[x]$ ← $key_t[y]$
  n[x] ← n[x]+1
  DiskWrite(y)
  DiskWrite(z)
  DiskWrite(x)
```

x: parent node
y: node to be split and child of x
i: index in x
z: new node

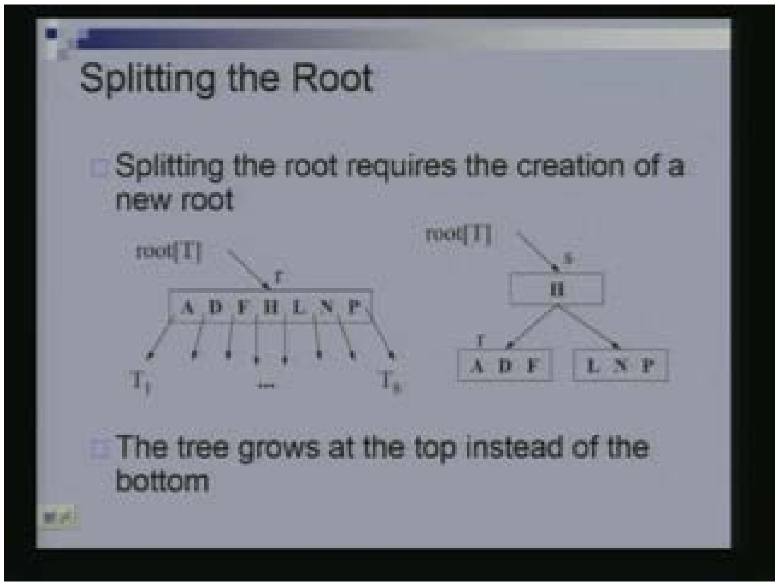
It’s very straight forward what you have to do, this is a procedure for splitting a node y whose parent is x and this is lets say the ith child of this parent. So i refers to that and what you do, you create a new node z and then first you copy the appropriate number of keys t - 1 keys from y to z that’s what being done here and then you also, if this is not a leaf node then you will copy the children also from y to z that’s being done here and then coping that. Here you are moving so this has to be promoted up here which means that these keys in here have to be moved one step right in the array that’s what is being done here and then this key that is to be moved up copied in to the appropriate place there and then we have these three describe operation because we have modified this node and this node and the new node we created.

So they all have to be return back to disk, so that is being done here. So the same thing is before the only catch today is that we are doing this disk operation also, you can look at this slides and understand the code. So how much time does it take for? So now today we are going to be measuring running time in terms of the number of disk access that we have to do. So in doing the split the number of disk access we have to do was three because we did three input outputs at the end. (Refer Slide Time 25:37) The CPU time which will now be proportional to the number of keys in that node because we have to moves certain keys and so on is going to be fairly small compared to this. So we will actually be counting time in terms of the number of disk access.

So today one variant of binary of b trees also works with,   so if you recall inserting when you did inserting in two four trees or in ab trees, we went down the tree, found out the place where we have to insert the element then try to put the element there and if that resulted in the node getting split, we split that node and that may have to insert the key in the parent node and that could lead to another split and so on and on. There is another way of doing this thing which is that we start from the top. So that is called the two parse operation where in first you come down and then you go up the tree. One possibility is to do the entire thing is only one pass, what is that mean? That is as you are starting from the top you check to see if the node that you are looking at already has its full quota of keys. What is the full quota? 2 t -1, if it’s already has 2 t - 1 keys then you are going to split that node right there and then and only then proceed down the tree, does everyone follow.

So we are going to start from the root and we are going to recursively travel all the way up to the leaf but before we descend to a lower level we make sure that the node contains strictly less than 2 t - 1 keys. If it has 2 t - 1 keys then we split the node right there. Let's understand this.

(Refer Slide Time 27:33)

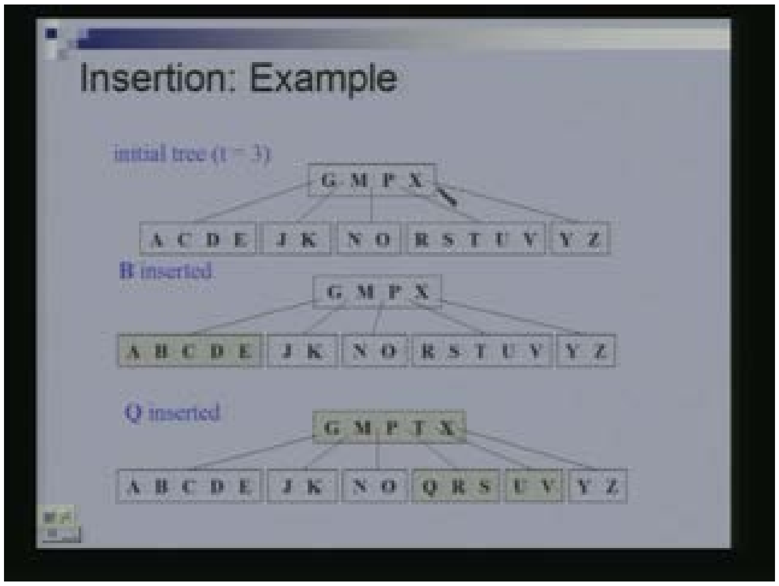


Splitting the Root

- Splitting the root requires the creation of a new root
- The tree grows at the top instead of the bottom

So I will come back to the slide later, let's say this was my root node. I am not yet inserted the key, this was the very first root node that I already counted this is already fill up I am working with t =4. So this has 2t-1, 7 keys in it. This is already full, so before descending down I see this is already full, I write at this step split this node in to a d f l n p and h moves up and now I will continue down the tree whatever key I am trying to insert I will now continue. So let me see, lets show you an example I will skip this slide and come back to this later. So let's take an example and then I will look at the code again. So suppose this is the tree, I am trying to t is the value of three I am trying to insert b so first I come here this node is not full, t is three so it node is full when it has five keys in it.

(Refer Slide Time 28:33)

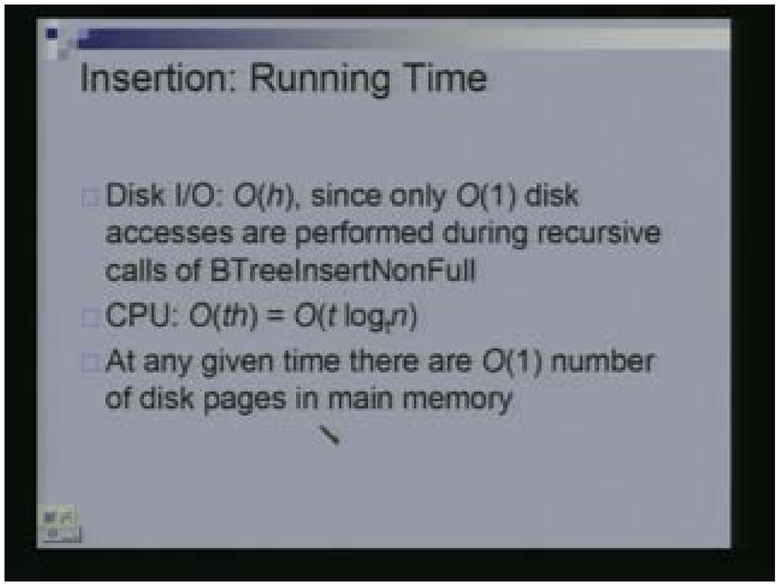


Insertion: Example

This is not yet full, so I can come down and b is inserted at this place. Suppose I am trying to insert q so once again I come to this node. This is not yet full, so I go down so actually this should b here I come here this is not yet full I come down here and then I put q here but this is full already so which means it will call this is split. So q r s would go on one side u v on the other and t gets promoted here. Note the very interesting thing why did this node not get split? What we had seen in the two four trees or ab trees what was happening was that when I inserted an element in to the parent, the parent could also gets split and that's not going to happen here anymore. Why? Because I came down from the parent only when the parent had room in it, if the parent had room then when I came down and I split this node and I put the one key in to the parent, the parent is now not going to get split. Let's with this happening again, so now when I am trying to insert the next key lets say l so now when I come to the root node, I will straight away split this because this already full, doesn't matter where l is going l is you know l m n o m l m n, it will come here this node is not full so but I am then I am trying to insert l I will right away split in to g m t x p and then come and insert l.

So now let's say when I try to insert f, I come here this is not full so I can continue this is not full so I can continue then I come here insert it here, split it and one guy gets promoted here. So the fact that this is not full lets me accommodate this addition key here without crossing ripple effect in the splitting process.

(Refer Slide Time 31:11)

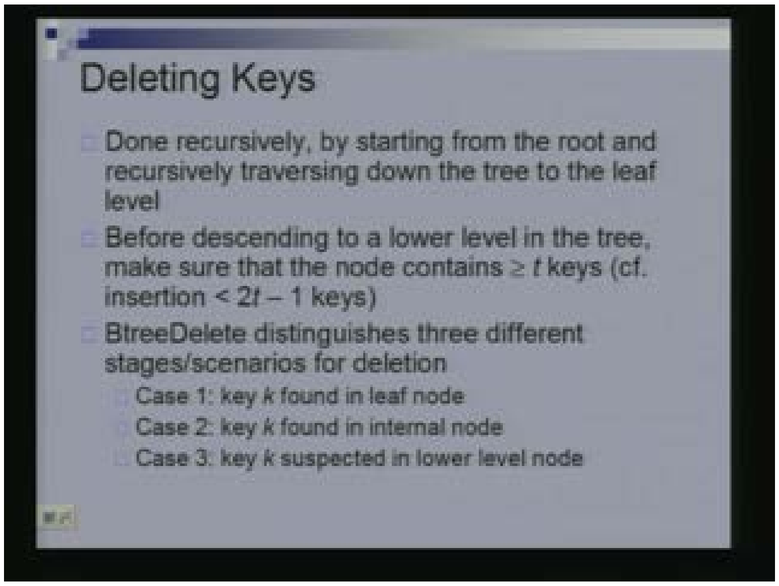


Insertion: Running Time

- Disk I/O: $O(h)$, since only $O(1)$ disk accesses are performed during recursive calls of BTreeInsertNonFull
- CPU: $O(th) = O(t \log_t n)$
- At any given time there are $O(1)$ number of disk pages in main memory

So in some this is like a one pass, you know in just one way down we have kind of access all the nodes. In just single pass we have access to all the nodes of the tree and whatever splitting then it will be done. So how many disk i was required? basically it just move down the tree once, we have to access every node and then every time you split the node, you have to right down that down back to the disk. Its parent node back to the disk and one new node that you create back to the disk. So for every time you split you might have to write down three nodes back to the disk. So you will have to read as many nodes as the height of the tree and how many nodes will you have to write back? You will have to write back at most three times the number of splits that many nodes you will have to write back to the disk. (Student Conversation-Refer Slide Time: 32:28). So towards the end I will discuss what are potential disadvantages of doing in this way.
If you keep splitting because you will have to now see what I am going to do when I have to delete a key. There I will try and let's see what I try and do that and then it will be clear. So when I am doing deletion, some going to actually skip the code I showed you, so you can look at those slides and understand them on your own.

(Refer Slide Time 32:50)



Deleting Keys

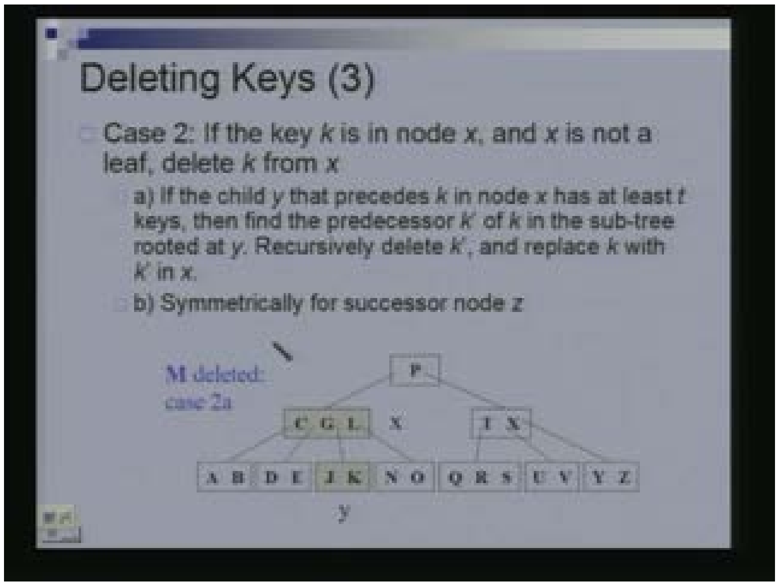
- Done recursively, by starting from the root and recursively traversing down the tree to the leaf level
- Before descending to a lower level in the tree, make sure that the node contains $\geq t$ keys (cf. insertion $< 2t - 1$ keys)
- BtreeDelete distinguishes three different stages/scenarios for deletion
  - Case 1: key *k* found in leaf node
  - Case 2: key *k* found in internal node
  - Case 3: key *k* suspected in lower level node

As far as deleting key is concern. Once again we recall the earlier procedure, we went down the tree we deleted the key, when we deleted the key that node could have less number of keys than it was supposed to have in which case we first try to borrow, if not successful merge. If we merge we have to remove one key from the parent that could cause the ripple effect, the cascading effect. So that we ended up doing all the way up to the top. Now we are again trying to do the single pass delete procedure which means that we just going to down from the top and do the deletes. So now what we are going to do is if we encounter a node which has already the minimum number of keys.

What is the minimum of keys in a node? t -1, if it just t -1 keys then we are going to make a effort to let it have more than t -1 keys strictly more, at least t. So what are we going to try the same thing as before? First we try to borrow from a sibling, if we are successful with that great. If not then that means the sibling also has t - 1 then we merge and when we merge we bring one from above and why can we now successfully bring one from above because the upper has strictly more than t – 1. So you can do the entire thing in a single pass in exactly this manner. So that's what we being said here before descending to lower level in the tree make sure that the node contains at least t keys.

In the case of insertion we require it contains strictly less than 2 t - 1 keys now we say it contain strictly more than t keys. Suppose this is the situation. So we are once again working with the t equals three, t equals three means each node has to have at least two keys. So this node it's okay, there is no problem with this node why because it has three keys so I can continue down. I can continue down I come to this node and f is deleted from here and there is no problem. But if I was trying to delete something from here then when I encounter this node which has only two key the minimum number then I will try to do something write then and there before proceeding down. So you know I have actually put down all the cases here where you know the key that we are trying to delete not in the leaf then you have to do all of the snaps so you seen all of these before so lets skip see some of this and this was the case when we have to merge. So sorry just once again let me make sure what right.

(Refer Slide Time 36:12)

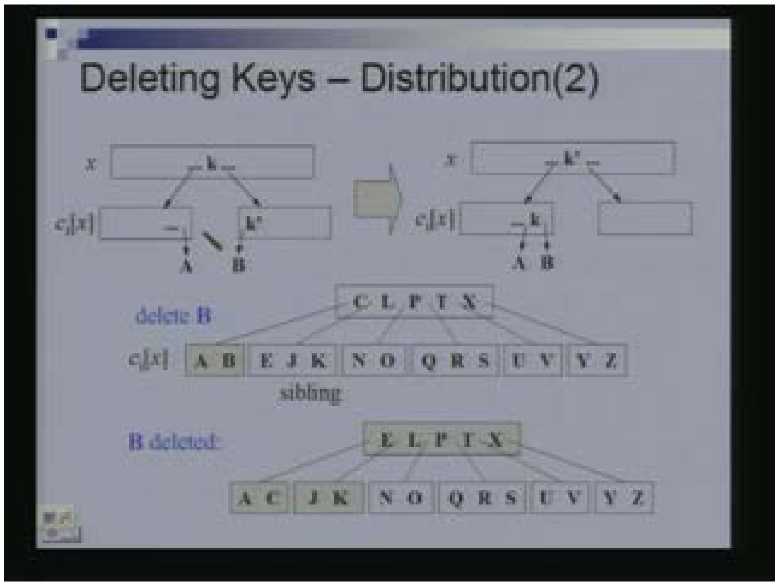


Deleting Keys (3)

- Case 2: If the key *k* is in node *x*, and *x* is not a leaf, delete *k* from *x*
  - a) If the child *y* that precedes *k* in node *x* has at least *t* keys, then find the predecessor *k'* of *k* in the sub-tree rooted at *y*. Recursively delete *k'*, and replace *k* with *k'* in *x*.
  - b) Symmetrically for successor node *z*

So if this is the setting if the particular node has only 2 - 1 keys then we have take action to ensure that it has at least t keys before we continue down and first thing is that we can borrow from the sibling and if not then we merge so this is the picture.

(Refer Slide Time 36:43)



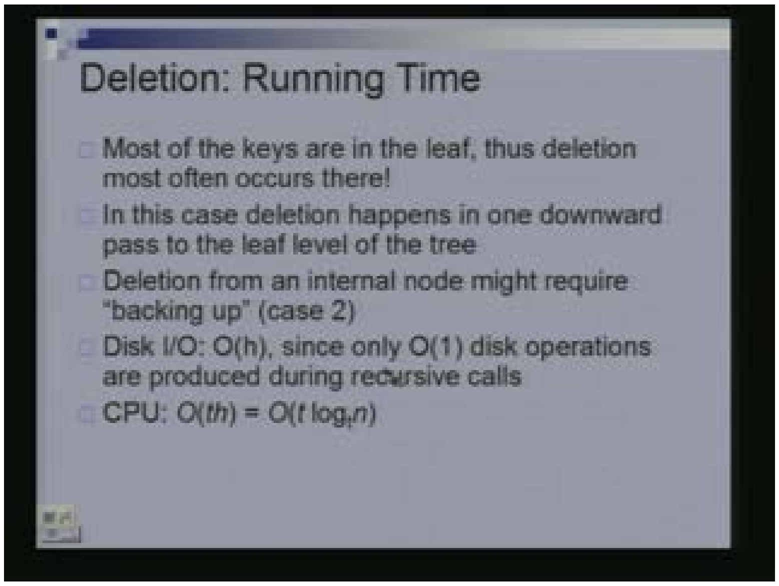
Deleting Keys – Distribution(2)

Let’s look at this one, we trying to delete b. We come here there is no problem here and when I am trying to delete here I have borrowed one from the sibling, you may have seen many example of this by now. So how do you borrow, one goes up and one comes down so this is the situation happening here and if we cannot borrow then you merge same as before you are deleting d.

So you come here C L when you are deleting d this is already has problem because it has minimum number of nodes so you going trying borrow one from the sibling but you are not going to be successful in borrowing one because that also has minimum so you merge and you get c l p t x bringing one down and then you go head and you delete d from here. So this is the situation that's going to happen. Then this also illustrates why this is the bads key, who can tell me why so this is answering the question that he had raised? Why did we do the same thing for the red black tree is and the ab trees? Student: so we could have delete either we will have… Staff: okay but [student conversation]. okay I am looking for one other answer which is perhaps who can [student conversation] good, split again see what's going to happen I just deleted and this is the picture I got, suppose I insert now what's going to happen I am going to go back this one, suppose I delete I am going to come back to this. Suppose I insert, this kind of going to go back and forth between this and I am spending lot of works in doing this because splitting the node lots of pointer moments and all of that.

So this key does not work in very well then this we have this kind of things happening. So insert and deletes are very highly interspersed, if you have a block of insert and block of delete happening then it's okay. Then you can still work with the scheme fine. Because if you had just deletes happening [Hindi conversation] so they would be no problem. This has large number of keys now so it can handle whole lots of delete without significant trouble. So if you had large sequence of delete then this is a fine strategy or if you had long sequence of insert but if you had these things alternating vary often then would be in some trouble. So once again what are the disk, I was going to look like?

(Refer Slide Time 39:52)



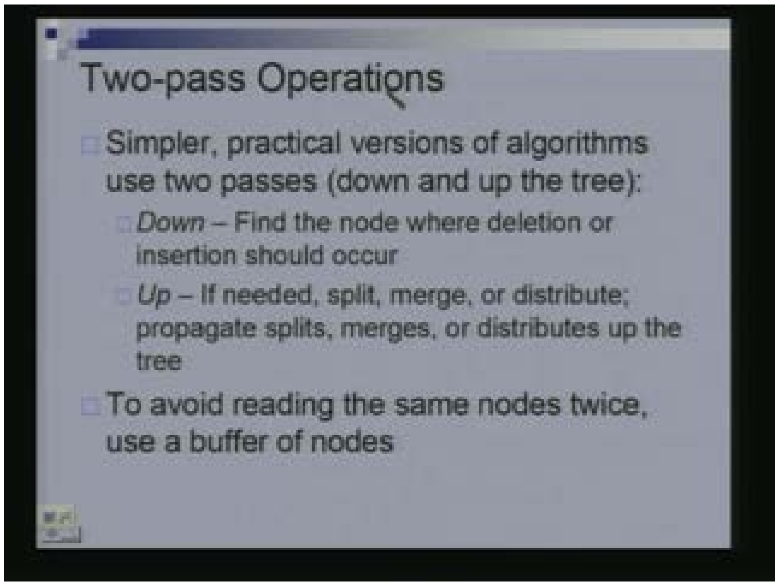
Deletion: Running Time

- Most of the keys are in the leaf, thus deletion most often occurs there!
- In this case deletion happens in one downward pass to the leaf level of the tree
- Deletion from an internal node might require "backing up" (case 2)
- Disk I/O: $O(h)$, since only $O(1)$ disk operations are produced during recursive calls
- CPU: $O(th) = O(t \log_t n)$

So we are going to read as many nodes as the height of the tree and each point when I am borrowing one from the sibling or merging one from the sibling and basically modify only my sibling node. So if I am doing any of those operation I might have to right back my sibling, my one node and the parent node, by the parent node because even when I borrow from the sibling one key goes up and one key comes down from the parent.

So its parent nodes also gets modified or even then merging again the parent gets modified, so every time I do some borrow or merge with the sibling I will have to right back three node and so that gives the number of disk right that we have to do a thing. So again this two pass operation we have actually seen before.

(Refer Slide Time 40:35)



Two-pass Operations

- Simpler, practical versions of algorithms use two passes (down and up the tree):
  - *Down* – Find the node where deletion or insertion should occur
  - *Up* – If needed, split, merge, or distribute; propagate splits, merges, or distributes up the tree
- To avoid reading the same nodes twice, use a buffer of nodes

So this was single pass we have doing things, the two pass we have already seen. So you first go down and then you go up as much as necessary so to say. But in the case of disk access you have to think carefully or you have to organize the things more carefully because if you first make one pass down and then you make a pass all the way back up then you spending twice as much time as you should have by just having made a single pass. So one thing you can do is that when you making first pass down you keeps all those blocks read in memory. How many blocks is that? How many pages is that just the height of the tree which is not too much. You keep them in memory. Why do we keep them in memory, because you might require them on the way back in second pass. So those are in optimization that you can do to try and reduce the type because as I said most of the time here spend in the disk access, you have to reduce the number of disk access as much as possible. So with that we are going to end this class today so this was mainly meant as a recap of a data structure that we saw today was just small extension of the ab tree but it is specifically for disk based accesses and this useful in that setting.