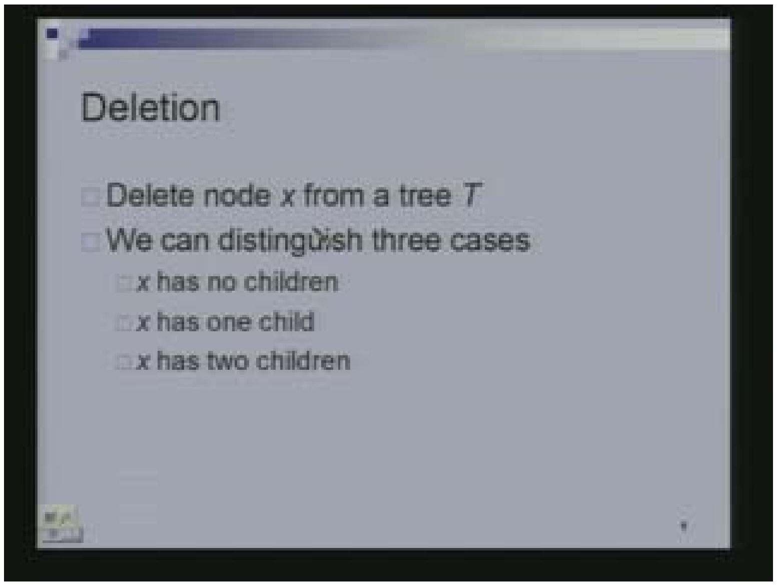
Data Structures and Algorithms

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

Lecture – 9
Deletion

Today we are going to start with deletion. In the last class we saw how to do insertion, search, computing successor, computing predecessor, computing minimum and maximum in a binary search tree. Today the only 1 operation that is left is the deletion and so we are going to see how to do deletion in a binary search tree. Then we are going to address the question that I had raised in the last class which was that if I were to insert some n elements into a binary search tree, suppose I were to randomly permute my elements and insert them. Then what can we say about the time that the insertion would take. Today we will see all of that in detail.

(Refer Slide Time: 2:21)



Deletion

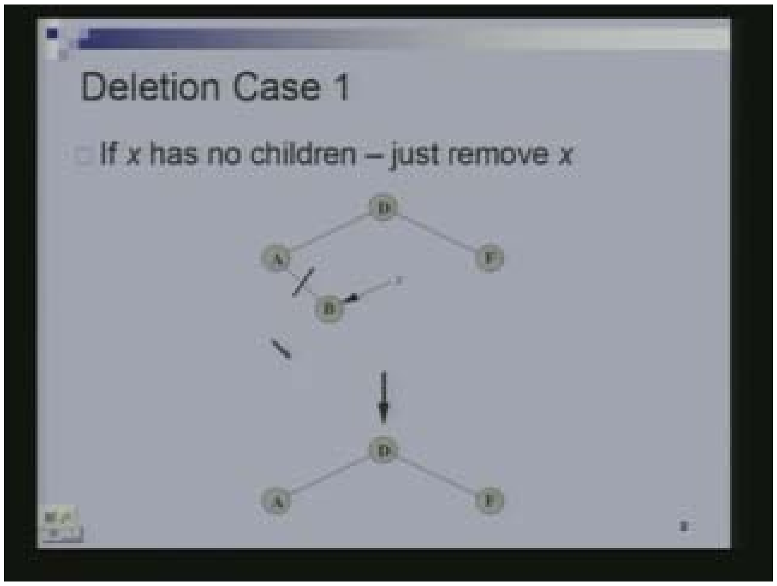
- Delete node *x* from a tree *T*
- We can distinguish three cases
  - *x* has no children
  - *x* has one child
  - *x* has two children

We are given a node x in the tree and you have to delete the node. We will distinguish 3 cases. X has no children which will be the easiest case for deletion, x is the leaf in that case. If x has only 1 child even then the case will be very easy and we will see all of that and when x has 2 children it is slightly trickier but still fairly straight forward.

When x has no children then deletion is trivial. Why is that because the point B is the node x, it does not have any children so it is a leaf. I need to delete the node x. I should just cut this length which means we will have to change either the left child or the right child of this parent.

In this case node B was the right child of the parent A, so we have to set the right child of A to null. If B were the left child then we would have set the left child to null. The 2nd diagram in the below slide is the tree that would be left after the deletion. If the node that you are trying to delete is the leaf node then the problem is trivial.

(Refer Slide Time: 3:09)

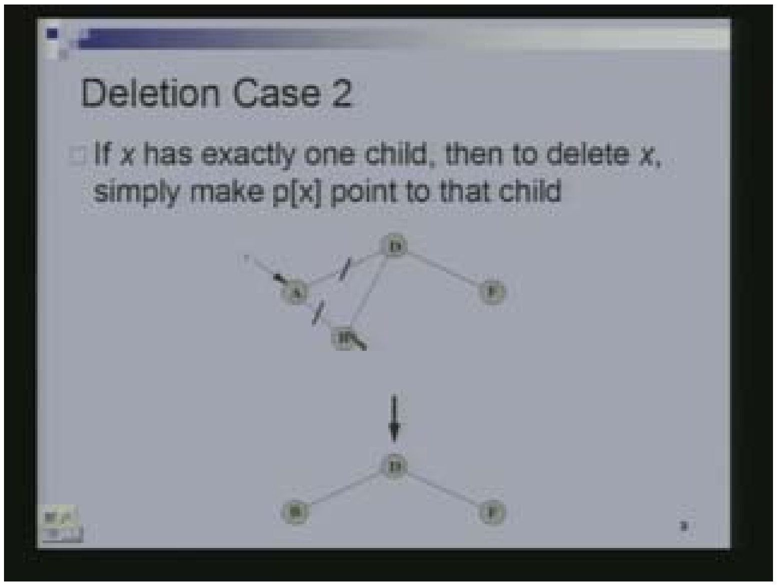


Deletion Case 1

- If x has no children – just remove x

What happens if the node that you are trying to delete has only one child? From the slide given below, you are trying to delete A, it has only 1 child which is B. A does not have a left child. The operation is like that you can think the part ABD as a linked list. It is just like a linked list and you are trying to delete a node from the linked list. You take the previous node which will be D and you will make its left child point to B.

(Refer Slide Time: 4:54)



Deletion Case 2

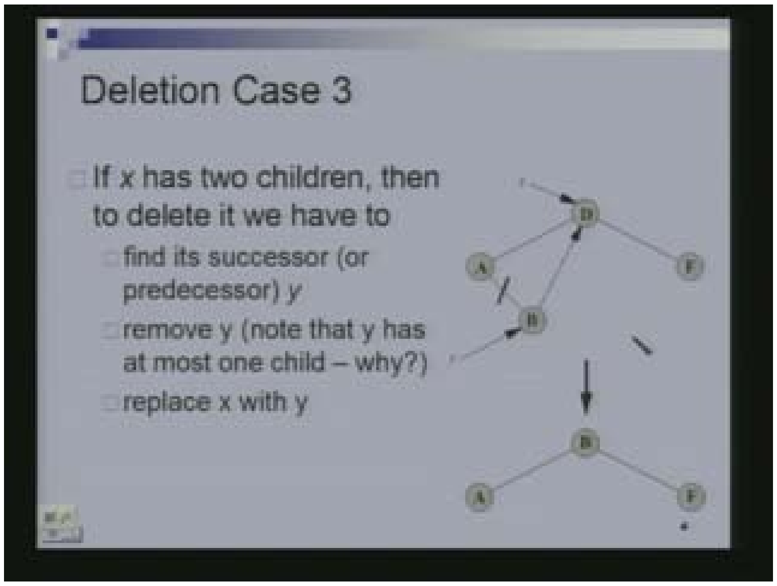
- If x has exactly one child, then to delete x, simply make p[x] point to that child

Effectively you are removing the 2 links which is coming from A and you are establishing the link directly from D to B. Are you convinced that this would maintain the search property. Why should B be on the left of D and not on the right of D? Because B is less than D, and B is in the left sub tree of D. We would make B as the left child and everything that is below B that is all the descendants of B are also less than D. Because they are all in the left sub tree of D. We will just continue with them as there is no change. The only change that happens is that the 2 links from A go away and you set up 1 link from B to D. So very little has to be done in this case. In the above slide 2nd diagram will be the new setting or this is what will happen after the deletion. You have D, B and F but A goes away.

The 3rd case is when the node to be deleted has 2 children. X is the reference to the node which is to be deleted. The node to be deleted is the one containing D and it has 2 children. Since it has 2 children let us do the following, it has a left sub tree and a right sub tree. We find the predecessor of D. We have seen how to do successor but we also said that you can equally find the predecessor.

How do you find the predecessor? The predecessor of D would be the largest element in the left subtree. We just have to come left and keep going right, since B does not have right child so B is essentially the predecessor of D in this example.

(Refer Slide Time: 07:40)



Deletion Case 3

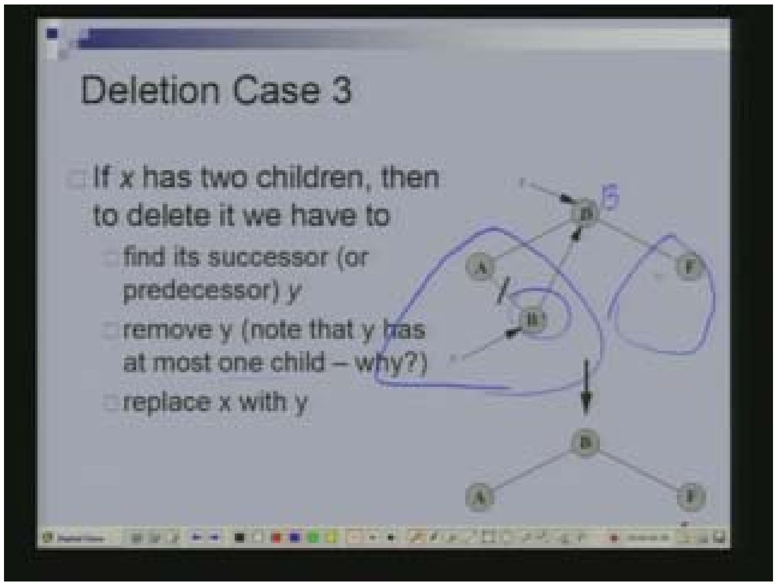
- If *x* has two children, then to delete it we have to
  - find its successor (or predecessor) *y*
  - remove y (note that y has at most one child – why?)
  - replace x with y

How many children does B have? B can have 1, it can have a left child nothing is preventing us from B having a left child. I could have a situation in which let us say T, you could have something like T. But B does not have a right child, there is nothing in the right side of B. If B had a right child then it would not have been the predecessor of D, because we would have gone down further.

B has only 1 child or no child, the node T need not be there. That is what is being said here B or y is the same thing. The y is the reference and B is the content sometimes I am calling it B, sometimes y but this has atmost 1 child. We are going to delete B and essentially we are going to move B to D. That is we are not establishing such a link from B to D, but we are going to replace D with B and delete the node B.

Why would the search property not get violated by moving the content of B to D? Since B was the successor of D, everything in this left subtree was less than D and B is the largest element in this left subtree. Everything in this left subtree is also less than B, if I move B to D there is no problem as far as the search property is concerned. Everything in the right subtree is more then D and so it is also more than B. By moving B to D again the search property is not violated.

(Refer Slide Time: 08:38)



Deletion Case 3

- If *x* has two children, then to delete it we have to
  - find its successor (or predecessor) *y*
  - remove y (note that y has at most one child – why?)
  - replace x with y

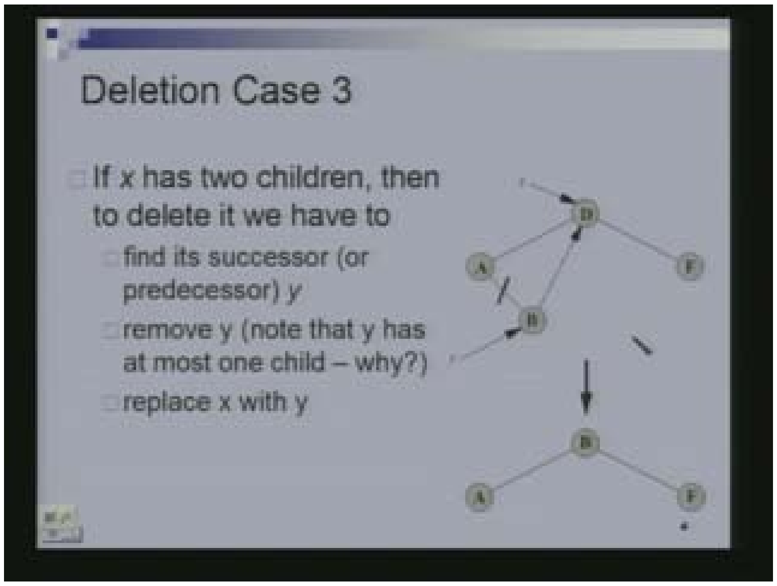
I can move B to D and can I delete B easily. Because B has either no child or 1 child. We have already seen how to delete a node which has no child or only 1 child. If B had a left subtree and suppose I decide to move B to D and there is nothing left so we will delete the node B. When we delete the node B, what happens we will make the right child of A point to that node. The node B gets deleted and the link corresponding to that goes away and we create 1 link and it goes to the point where it is directed and B moves up.

That is the operation and now we have covered all the 3 cases. The 1st case was when there were no children, 2nd case was 1 child and 3rd case was 2 children. We said in the 3rd case we will have to move the content of B to D and then delete the node B. We worked with the predecessor but we could also have worked with the successor, since D has 2 children we can also find the successor of this node by going once right and then keep going left. The same kind of a thing can be done even there, we could have replaced D with its successor instead of its predecessor. But in this example I have shown the predecessor.

What do you think is the running time of the delete operation? Suppose you do not even have to search for the node. I tell you this is the node and I give you the reference to the node that you want to delete. How much time does it takes to delete? In the 1st case when the node is a leaf, how much time it would take? Order one because it just needs to go the parent. There is a pointer in the node, reference to the parent node I just go to the parent and I just update the link that is the right or the left child of the parent. I am giving the reference of the particular node which has to be deleted. In every node we also have a parent, left child and a right child. We can always go back to the parent and update the content to show the deletion operation.

In the 2nd case if we are deleting the node A, then once again we need go to the parent and modify its left child, again a constant time operation. In which case do we take more time? In the case 3, because here we have to find a predecessor or a successor. Predecessor can take time as large as the height of the tree because we will have to go once left and then keep going right. We can take in the worst case time proportional to the height of the tree. Case 3 takes more time and the time taken in the worst case is the order of the height of the tree.

(Refer Slide Time: 13:15)

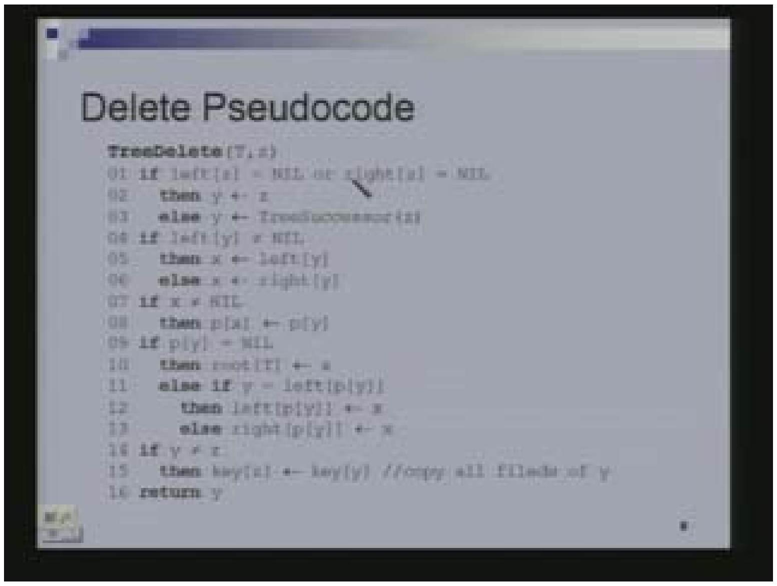


Deletion Case 3

- If x has two children, then to delete it we have to
  - find its successor (or predecessor) y
  - remove y (note that y has at most one child – why?)
  - replace x with y

The pseudo-code for deletion is given in the below slide, so that you will understand roughly what is happening and we do not have to look in detail. If the left or the right is nil then we know that either it is a leaf node or it is a node with only 1 child. Then in that case, z was the node which I am trying to delete. So y $\leftarrow$ z and we look at the TreeSuccessor (z). I am not completely sure that what I have is right, so let us just skip this. You would have understood the delete operation. I might have made some mistakes somewhere in the pseudocode. Predecessor or successor is the same thing, we can also work with the successor that is what I said. For the successor we have to search in the right tree. I do not think that it is very critical but let us skip this thing for now. You understood the delete operation and the pseudo code you can all write yourself.

(Refer Slide Time: 13:51)



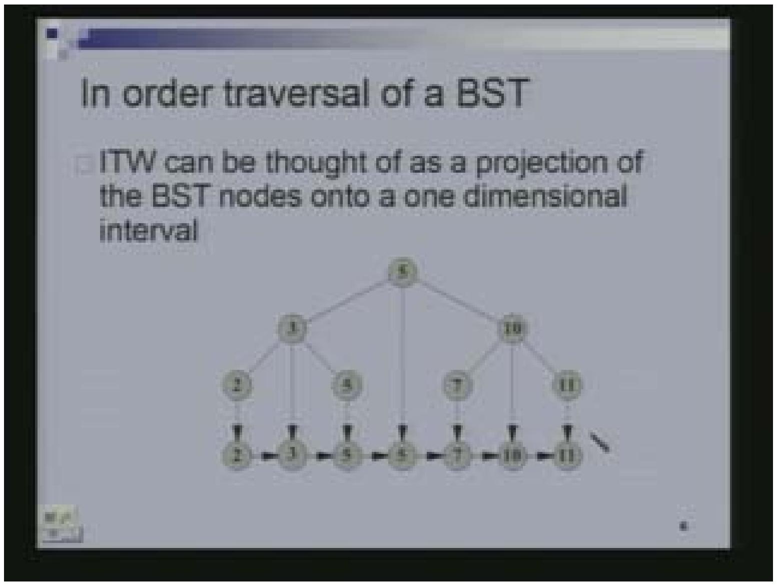
Delete Pseudocode

```
TreeDelete(T,z)
if left[z] = NIL or right[z] = NIL
  then y ← z
  else y ← TreeSuccessor(z)
if left[y] ≠ NIL
  then x ← left[y]
  else x ← right[y]
if x ≠ NIL
  then p[x] ← p[y]
if p[y] = NIL
  then root[T] ← x
  else if y = left[p[y]]
    then left[p[y]] ← x
    else right[p[y]] ← x
if y ≠ z
  then key[z] ← key[y] //copy all fileds of y
return y
```

Suppose I give you a binary search tree and I do an in-order traversal of this binary search tree. What does in-order do? First the left then the 5th node then the right. First we will be printing out all the keys on the left. Suppose all I do when I visit a node is print out the key. That is what my traversal procedure is, first I will print out the keys which are on the left in some order. Then I will print the key which is in the center and then I will print out the keys on the right. This means that I will print out the root key 5 after I have print out the left and before I print out the right. All the keys on the left are less than or equal to 5 and all of the keys on the right are greater than or equal to 5, which means that 5 really comes at the right place in the ordering of these keys.

Because everything which is less than 5 comes before 5 and everything that is more than 5 comes after 5. Hence 5 come really at the right place and the same argument can be set for every key not just the root key. When will you print 3 out? After I have printed everything to the left which means everything that is smaller than 3 will come before 3 and then I will print out things on the right.

(Refer Slide Time: 16:54)



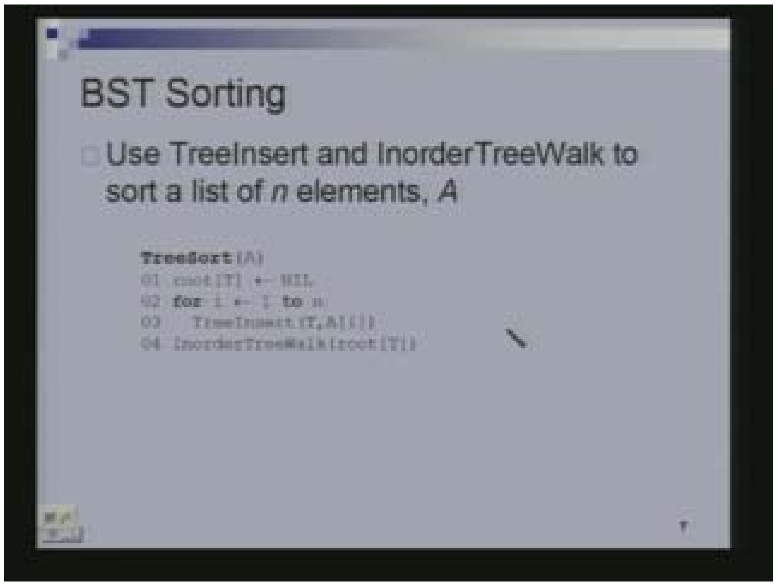
In order traversal of a BST

- ITW can be thought of as a projection of the BST nodes onto a one dimensional interval

In in-order traversal, if you were to just to look at this and see in what order you would print out. First you will come to 3 and then go left, then come to 2 and print this one. Then you will print 3, 5 then you will come right and then go left and so on. This is the order in which you will print the keys. The property of the in-order traversal of a binary search tree is that it prints out the keys in increasing order.

This can be a good method of sorting a bunch of keys. I will call this the binary search tree base sorting procedure. What is the method? You have a bunch of keys which you want to sort. You first insert all the keys into a binary search tree. That is what you are doing in the slide given below. You are taking all the keys and inserting them into a binary search tree and then just do in-order traversal of this tree. You will get all the keys in an increasing order that is you have sorted a set of keys.

(Refer Slide Time: 19:31)



BST Sorting

- Use TreeInsert and InorderTreeWalk to sort a list of *n* elements, *A*

```
TreeSort(A)
root[T] ← NIL
for i ← 1 to n
  TreeInsert(T,A[i])
InorderTreeWalk(root[T])
```

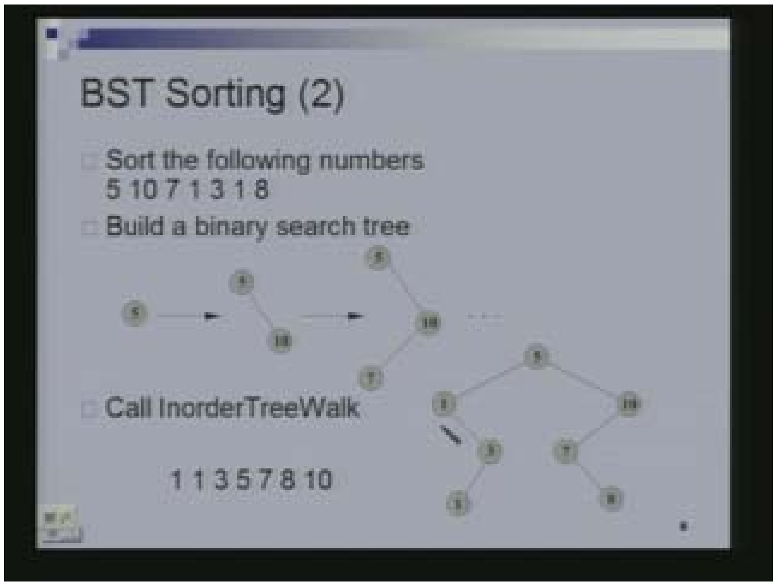
How much time does this procedure take? We have to insert all the keys and then we have to do an in-order tree work. Let us first look at how much time does the in-order tree work take. How much time does it take to traverse the nodes of a tree in in-order? It is order n. Why it is order n? We have to print all the node but I might take much more time. I need atleast order n time may be I need more time. We will look at this question later.

How much time does it take to do an in-order tree work? Actually it just takes linear time that is it takes order n time. We will not do it in todays class, we will see how to argue that it just takes order n times. How much does the part which is given below takes, when I am inserting all the elements into a tree?

```
for I ← 1 to n
  TreeInsert (T, A[i])
```

It is n log n. But why it is n log n?

(Refer Slide Time: 20:14)



BST Sorting (2)

- Sort the following numbers
  5 10 7 1 3 1 8
- Build a binary search tree
- Call InorderTreeWalk

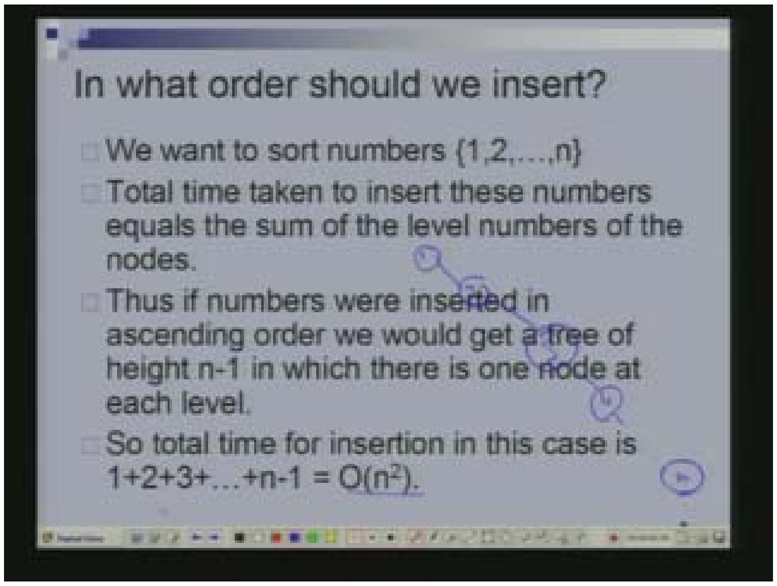
  1 1 3 5 7 8 10

This is my BST sorting procedure. I want to sort the keys given in the above slide, I insert them into a binary search tree one after the other. First 5 then 10, 7 then when I insert 1 it will go as the left child of 5 then when I insert 3 it will go as the right child of 1. When I insert the other 1 it will go as the left child of 3. It depends upon how you are doing it, it could also have come as the left child of the other 1 and it is the same thing. And 8 would come as the right child of 7 and the final diagram in the above slide is the tree you would get. If you have to do any in-order tree work on this tree you would get exactly the sorted sequence.

How much time does it takes to insert all the n elements? It depends upon the sequence. If I were to sort let us say given a set of numbers 1 through n and I just want to sort them. The total time taken to insert this numbers is equal to the sum of the levels at which the nodes will come up because when I insert a particular number I come down the tree and I insert it at a particular place. The number of levels I traverse or the number of comparisons I do before I insert it is exactly equal to its level. Either level or level +1 or level -1 but you can think of it as the level for now.

If I had inserted the numbers in sorted order, first I inserted 1 then I inserted 2, 3 and so on. What is the kind of a tree I would get? I would have 1 followed by 2 followed by 3 and so you remember that kind of a picture. I would get a tree some thing like the one which is given in the slide in blue color from 1 to n. What is the sum of levels? It is from 0+1+2+3… (n-1) and O (n^2) is the series which sums to n^2. That is not good, the time taken for insertion could be as bad as n^2.

(Refer Slide Time: 22:15)



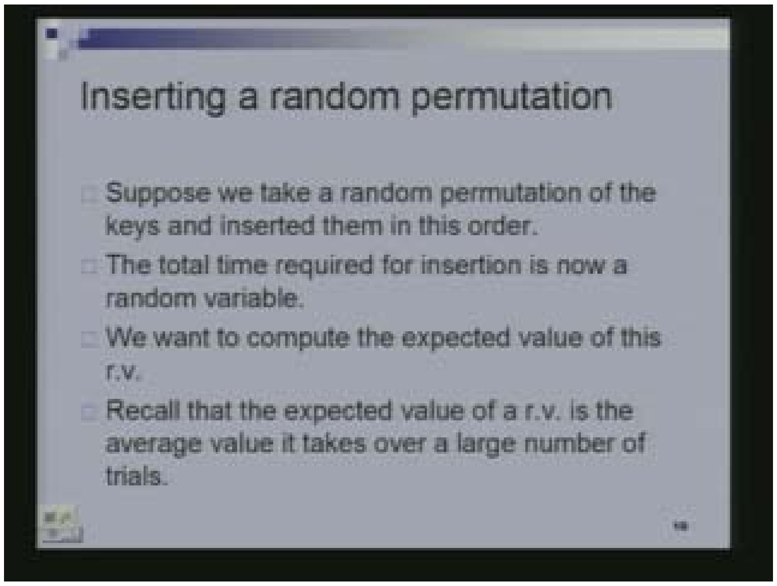
In what order should we insert?

- We want to sort numbers {1,2,…,n}
- Total time taken to insert these numbers equals the sum of the level numbers of the nodes.
- Thus if numbers were inserted in ascending order we would get a tree of height n-1 in which there is one node at each level.
- So total time for insertion in this case is 1+2+3+…+n-1 = O(n^2).

Is this clear to everyone and this bring backs the question that we had asked in the last class. I take a random permutation of the elements. There are n factorial different permutations I take one of them at random. I insert the elements in this particular order. I want to compute the time it takes to insert. The time required for insertion is a random variable.

We saw an example where the time required for insertion could be as bad as n^2. That is what I showed you in the previous slide. In the best case it could be quite small and we will the best case. It is really a random variable it depends upon the sequence and the permutation in which we are inserting the elements. In fact we had discussed the best case in the previous class, we said the first element inserted should be $\frac{n}{2}$ then it should be $\frac{n}{4}$ and then $\frac{3n}{4}$ and so on.

(Refer Slide Time: 24:52)



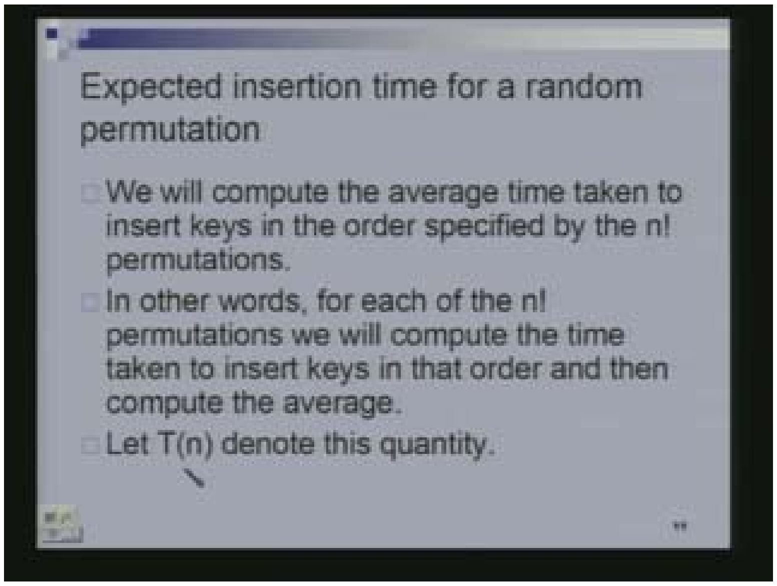
Inserting a random permutation

- Suppose we take a random permutation of the keys and inserted them in this order.
- The total time required for insertion is now a random variable.
- We want to compute the expected value of this r.v.
- Recall that the expected value of a r.v. is the average value it takes over a large number of trials.

All the way we had not computed the total time for insertion. The total time for inserting these n elements is really a random variable and for random variables we compute what is the expectation of the random variable. Recall what is the expectation of a random variable? If a random variable takes many different values then the expectation of the random variable is obtained by taking the average if the probability of each value is the same. Otherwise you essentially do many trials which means that you compute the value of the random variable then you do the experiments once again and so on. Look at the value of the random variable that you get and take the average over all the trials. That is how you compute the expectation of the random variable.

In our case it is easier because each of the permutation is equally likely. We are saying we will just pick one of the permutations uniformly at random. Each of the permutation is equally likely so we are going to see how much time it takes to insert the elements in the order given by a certain permutations. How do we compute the average? We will take a permutation, insert the elements in that order and compute the time it takes to do that insertion. Then take another permutation compute the time it takes to insert the elements in that order, take the 3rd permutation compute the time it takes to insert the elements in that order and so on.

(Refer Slide Time: 26:18)

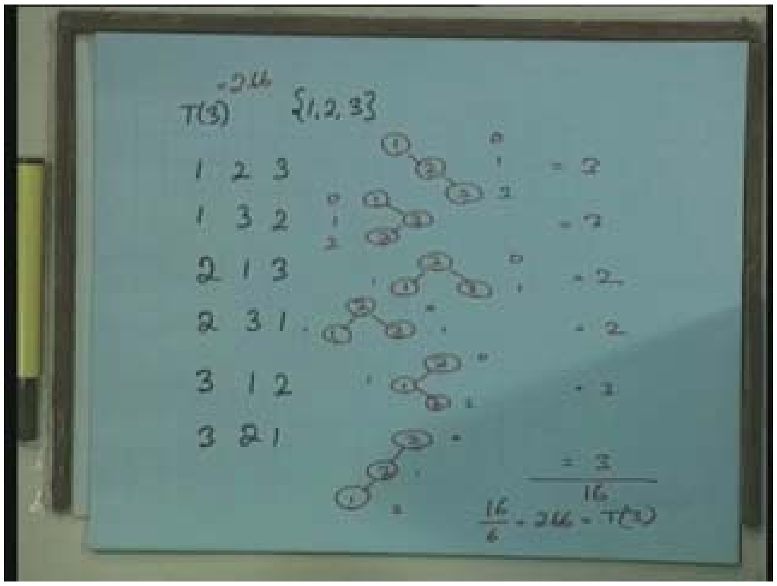


Expected insertion time for a random permutation

- We will compute the average time taken to insert keys in the order specified by the n! permutations.
- In other words, for each of the n! permutations we will compute the time taken to insert keys in that order and then compute the average.
- Let T(n) denote this quantity.

There are n! different permutations, while compute the time it takes to insert the elements in the order specified by these n! different permutations. You will get the total time this huge quantity and will then divide by n! to compute the average and that could be the expected time it takes. Just to summarize, for each of this n! permutations we will compute the time taken to insert the keys in the order specified by that permutation and then we will compute the average. The average is computed over the n! permutations. We will denote this quantity by T (n), whatever is this value will be T (n). Let us see how to compute T (3).

(Refer Slide Time: 28:50)

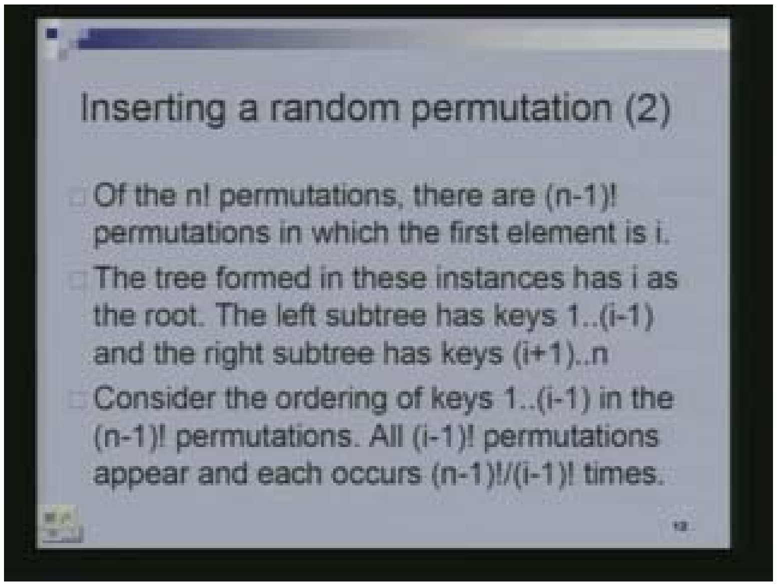


T (3) is the tree obtained or is the expected height of a tree on 3 nodes. What are the various possible trees on 3 nodes that we get? What are the various permutations of elements 1, 2, 3? We will have 6 different permutations as given in the slide. When we have this particular permutation what is the tree we would get? If we were to insert the elements in this order (1 2 3) and you will get a tree whose root is 1, 2 would come to the right and 3 would come after that. When the elements are inserted in this particular order (1 3 2), we would get a tree 1, 3 to the right and 2 to the left of 3.

If the elements are inserted in this order (2 1 3), we will get a tree which is 2, 1 to the left and 3 to the right. When elements are inserted in this order (2 3 1), we get a tree which is 2, 3 to the right and 1to the left. When elements are inserted in this order (3 1 2), we get a tree which is 3, 1 to the left and 2 to the right of 1 but to the left of 3 and in this order (3 2 1), we get 3, 2 to the left and 1 to the left.

What is the total time taken for insertion? We recall what we said, it is just the sum of the levels of the nodes. The levels of the nodes are 0, 1, 2 and so the sum is 3. In the next one the levels of the nodes are again 0, 1, 2 so the sum is 3. For the next one it is 0 and 1 in which both 1 and 2 are at level 1, so the sum 2. For the next one also the sum is 2. For the 5th and 6th one the sum is 0, 1, 2 that is 3. Total of 16 and so the average is $\frac{16}{6}$ which is 2.66 which is T (3). Hence we computed T (3) as 2.66. That is what being said here, for each of the n! permutations we are going to compute the time taken to insert the keys which is what we did for T (3). Then compute the average that gives us the value of T (3). But we cannot do the computations in this manner, we have to do it cleverly because we have to compute what T (n) is. Now am going to fix a particular element i and I am going to look at all permutations in which the element i is the very first element in the permutation.

(Refer Slide Time: 29:10)



Inserting a random permutation (2)

- Of the n! permutations, there are (n-1)! permutations in which the first element is i.
- The tree formed in these instances has i as the root. The left subtree has keys 1..(i-1) and the right subtree has keys (i+1)..n
- Consider the ordering of keys 1..(i-1) in the (n-1)! permutations. All (i-1)! permutations appear and each occurs (n-1)!/(i-1)! times.

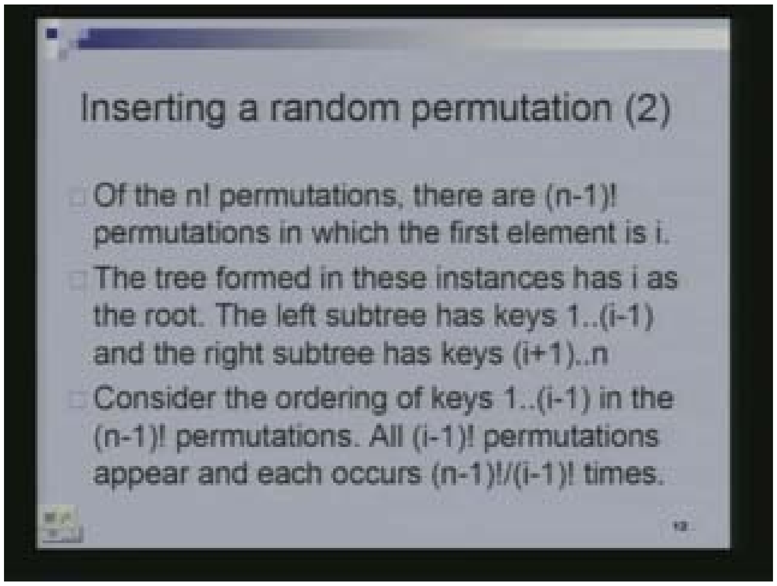
12

There are (n-1)! permutations in which i is the first element. In these (n-1)! permutations the tree that we obtain will have i as the root because it is the very first element and then the keys one through (i-1) will be in the left subtree of the root and the keys i+1 through n would be in the right subtree of the root.

Let us look at one of this (n-1)! permutations and restrict our attention to the keys 1 through (i-1). They will be coming at many different places and at some places in this permutation.

Let me look at another one of these (n-)! permutations. These keys 1 through (i-1) would be coming at some other place. If I look at all of these (n-1)! permutations they will induce permutations of keys 1 through (i-1) elements also. Every permutation of 1 through (i-1) would be there.

(Refer Slide Time: 31:35)



Inserting a random permutation (2)

- Of the n! permutations, there are (n-1)! permutations in which the first element is i.
- The tree formed in these instances has i as the root. The left subtree has keys 1..(i-1) and the right subtree has keys (i+1)..n
- Consider the ordering of keys 1..(i-1) in the (n-1)! permutations. All (i-1)! permutations appear and each occurs (n-1)!/(i-1)! times.

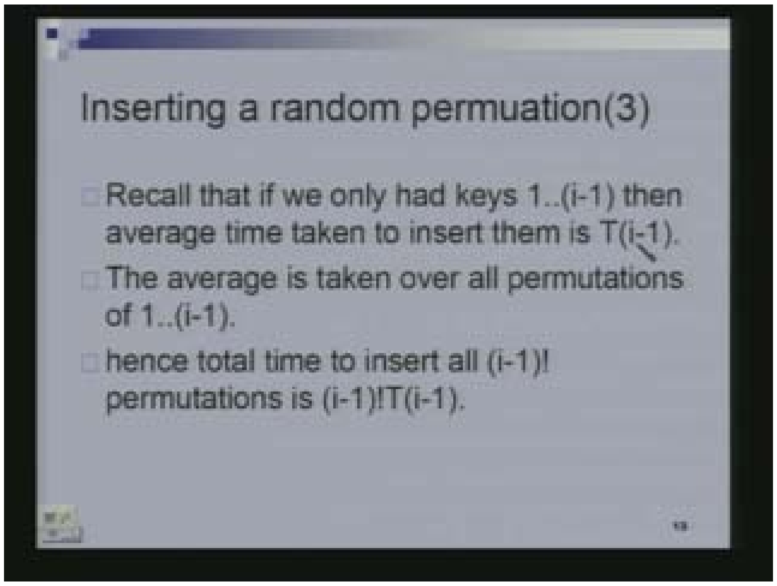
12

In particular each permutation of 1 through (i-1) would occur how many times. There are (n-1)! permutations in all and there are (i-1)! permutations of the elements 1 through (i-1) and there is no reason why one of these permutation will occur more often than the other permutation. Each of them is occurring equally likely. How many times is each one of them occurring?

It is $\frac{(n-1)!}{(i-1)!}$ times. You can also think of it in a slightly different manner. You have these (i-1) elements somewhere and the other (n-i) elements are getting inserted at different places.

How many different permutations are there of those (n-i) elements that are getting inserted? You can compute that and it will be exactly this $\frac{(n-1)!}{(i-1)!}$ quantity. This is a simpler argument, there are (n-1)! permutations in all and there are (i-1)! permutations of these elements 1 through (i-1) and there is no reason why one of these permutation should be more likely than the other one. They will all occur the same number of times. It is symmetric of law, each permutation therefore of these elements is occurring exactly $\frac{(n-1)!}{(i-1)!}$ times.

(Refer Slide Time: 33:12)



Inserting a random permuation(3)

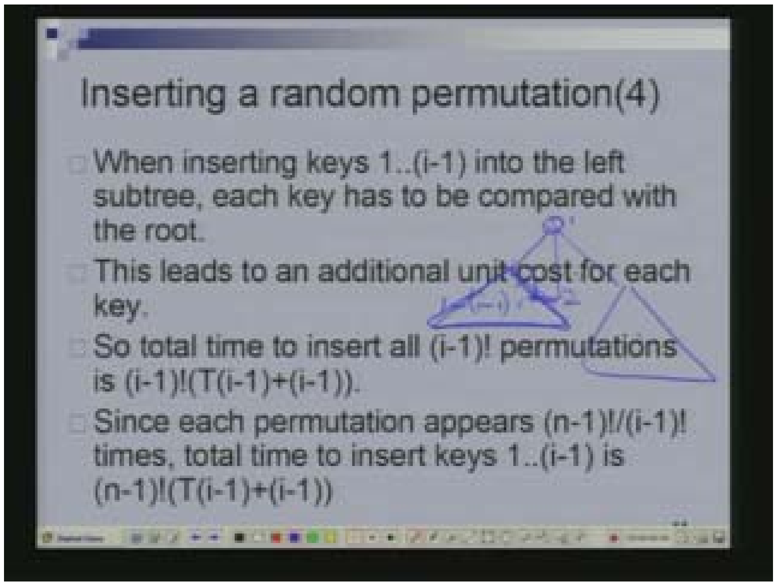
- Recall that if we only had keys 1..(i-1) then average time taken to insert them is T(i-1).
- The average is taken over all permutations of 1..(i-1).
- hence total time to insert all (i-1)! permutations is (i-1)!T(i-1).

Suppose I had only (i-1) keys, 1 through (i-1). Then the average time taken to insert these keys is T (i-1), this is what we defined T (n). This average is the average taken over all the permutations of 1 through (i-1). The time taken to insert all these (i-1)! permutations are just (i-1)! T (i-1). The average time is T (i-1) and the average is taken by computing the total time to insert all the (i-1)! permutations divided by (i-1)! The total time to insert all the (i-1)! permutations are (i-1)! T (i-1). For instance in the example that we did earlier for T (3), the total time taken to insert all the permutations was 16 which we had computed, that was the average times 6.

When we are inserting the keys 1 through (i-1) into the left sub tree, we go back to the previous setting. Recall we are considering only permutations in which the first element is i. The keys 1 through (i-1) when we are inserting they will all be compared first against i and then we will go and put them in the left subtree.

Each key has to be compared with the root which is i. It is like as if I am creating a left subtree of (i-1) elements but I can not just count that as the average call. I am also counting one more because for each of those elements in the left subtree their level is actually one more than, if it were just the left subtree.

(Refer Slide Time: 34:60)



Inserting a random permutation(4)

- When inserting keys 1..(i-1) into the left subtree, each key has to be compared with the root.
- This leads to an additional unit cost for each key.
- So total time to insert all (i-1)! permutations is (i-1)!(T(i-1)+(i-1)).
- Since each permutation appears (n-1)!/(i-1)! times, total time to insert keys 1..(i-1) is (n-1)!(T(i-1)+(i-1))

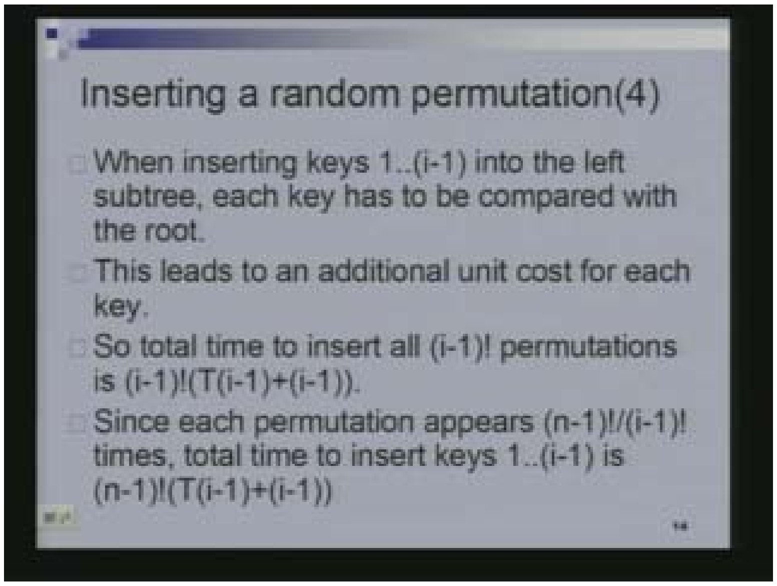
Let me clarify what I am saying. In the above slide the small round in blue color is the i and the keys 1 through (i-1) are coming inside the triangle. If that triangle in blue color were my tree then I know that the average time to create this tree is T (i-1) but I am creating a tree of over n nodes. The keys 1 through (i-1) are coming in the first triangle and the other keys are coming in the next triangle.

How much time I am spending in creating this part (darker triangle)? It is the average for 1 through (i-1) plus one more for each one of these, because first I compare with i and then come down because the level of each of these is actually one more in tree i than it is in the tree (i-1).Then it has level 2 in the (i-1) tree and level 3 in the original tree. What is the total time to insert all the (i-1)! Permutations?

Recall T (i-1) is the average time to insert (i-1) keys. In the tree that I am creating am taking one unit extra for every node that I am inserting, because first I am comparing it against the node. (i-1)! (T (i-1) + (i-1)) is the time I am taking to insert T (i-1) keys and that is the average time I am taking to insert the (i-1) keys in my tree.

What is the total time? I have my (i-1)! different permutations so the total time is this product (i-1)! (T (i-1) + (i-1)). Each of the permutations appears so many times that is $\frac{(n-1)!}{(i-1)!}$ times. What is the total time I spend in inserting keys 1 through (i-1)? The total time to insert all the (i-1)! permutations is (i-1)! (T (i-1) + (i-1)). But each of the permutations is appearing $\frac{(n-1)!}{(i-1)!}$ times. So it is just $\frac{(n-1)!}{(i-1)!}$ times (i-1)! (T (i-1) + (i-1)) which is (n-1)! (T (i-1) + (i-1)). This (n-1)! (T (i-1) + (i-1)) is the total time I take to insert the keys 1 through (i-1) into my tree where this sum is taken over all the (n-1)! permutations and this is not actually the total time but this is the total time for permutations in which the first element is i.

(Refer Slide Time: 36:59)



Inserting a random permutation(4)

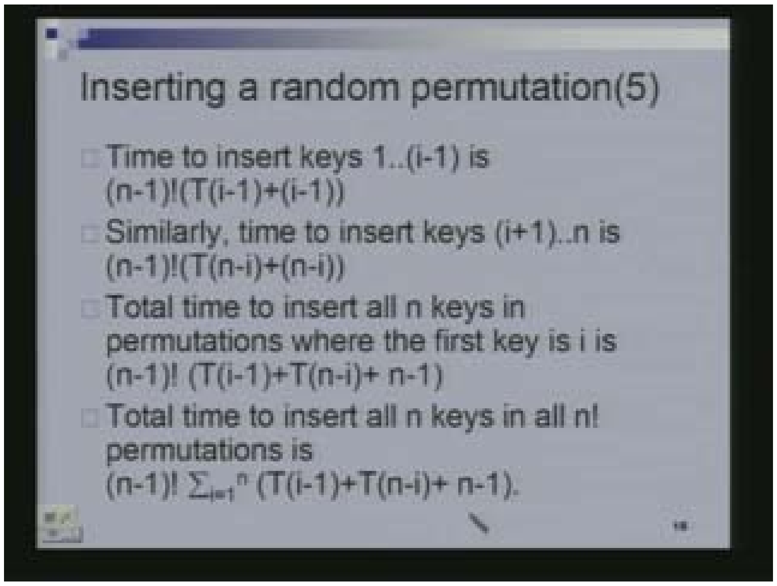
- When inserting keys 1..(i-1) into the left subtree, each key has to be compared with the root.
- This leads to an additional unit cost for each key.
- So total time to insert all (i-1)! permutations is (i-1)!(T(i-1)+(i-1)).
- Since each permutation appears (n-1)!/(i-1)! times, total time to insert keys 1..(i-1) is (n-1)!(T(i-1)+(i-1))

14

The time to insert keys 1 through (i-1) is (n-1)! (T (i-1) + (i-1)). Similarly the time to insert keys (i+1) through n will be similar to the one before. Wherever there is (i-1) I will replace it by (n-i), because that is the number of keys I am inserting. That would give me this expression (n-1)! (T (n-i) + (n-i)) and this is to insert the keys in the right subtree.

The total time to insert all the n keys is just the sum of the above 2 quantities. (n-1)! (T (i-1) + T (n-i) + n-1) is the total time over all the permutations in which the first key is i. What is the total time to insert all the n keys in all the n! different permutations? It is just the sum of this quantity (n-1)! (T (i-1) + T (n-i) + n-1) as i goes from 1 through n because the first key can be 1 or 2 or 3. (n-1)! $\sum_{i=1} n(T(i-1)+T(n-i)+n-1)$

(Refer Slide Time: 40:22)



Inserting a random permutation(5)

- Time to insert keys 1..(i-1) is (n-1)!(T(i-1)+(i-1))
- Similarly, time to insert keys (i+1)..n is (n-1)!(T(n-i)+(n-i))
- Total time to insert all n keys in permutations where the first key is i is (n-1)! (T(i-1)+T(n-i)+ n-1)
- Total time to insert all n keys in all n! permutations is (n-1)! $\sum_{i=1}^{n}$ (T(i-1)+T(n-i)+ n-1).

16

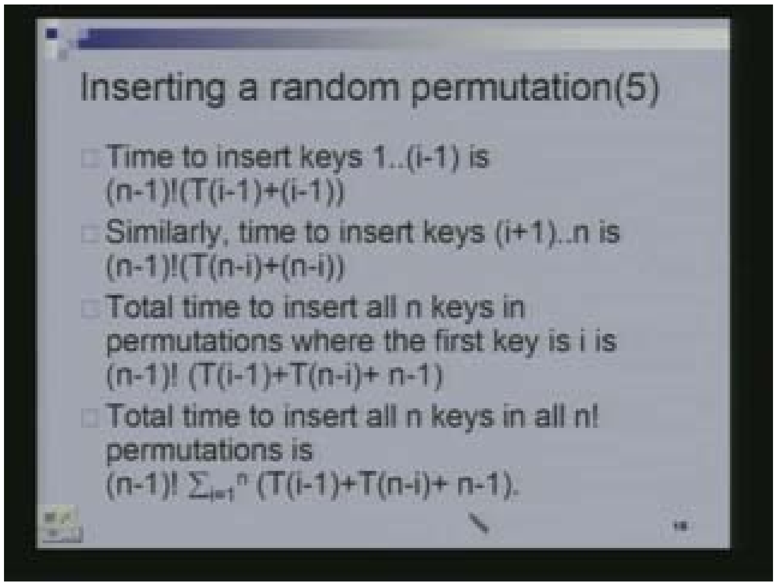
All I am doing is trying to figure out the total time taken to insert all the n elements not in one permutations but over all n! permutations put together. I am trying to do it in a way so that am able to analyze it. There are (i-1)! different permutations of (i-1) elements. In the sequence of 1through n those (i-1) elements are present at different places.

All possible permutations of (i-1)! different permutations of those (i-1) elements are there but each of those permutations itself is occurring (n-1)! over (i-1)! times. We have to keep that in mind. We are going to work with the expression which is given below.

$$\text{(n-1)!} \sum_{i=1} n(T(i-1)+T(n-i)+n-1)$$

The total time to insert all the n keys is above thing, this is exactly the 16 which we had computed for T (3).

(Refer Slide Time: 40:22)



Inserting a random permutation(5)

- Time to insert keys 1..(i-1) is (n-1)!(T(i-1)+(i-1))
- Similarly, time to insert keys (i+1)..n is (n-1)!(T(n-i)+(n-i))
- Total time to insert all n keys in permutations where the first key is i is (n-1)! (T(i-1)+T(n-i)+ n-1)
- Total time to insert all n keys in all n! permutations is (n-1)! $\sum_{i=1}^{n}$ (T(i-1)+T(n-i)+ n-1).

What is the average time to insert n keys? It is basically, this is (n-1)!$\sum_{i=1} n(T(i-1)+T(n-i)+n-1)$ the total time, so this divided by n! would be the average. Below given is the one which I would get. Just to make sure this divided by n! will give me $\frac{1}{n}$ and this expression I have written where i go from 1 through n.

$$\frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i)+n-1)$$

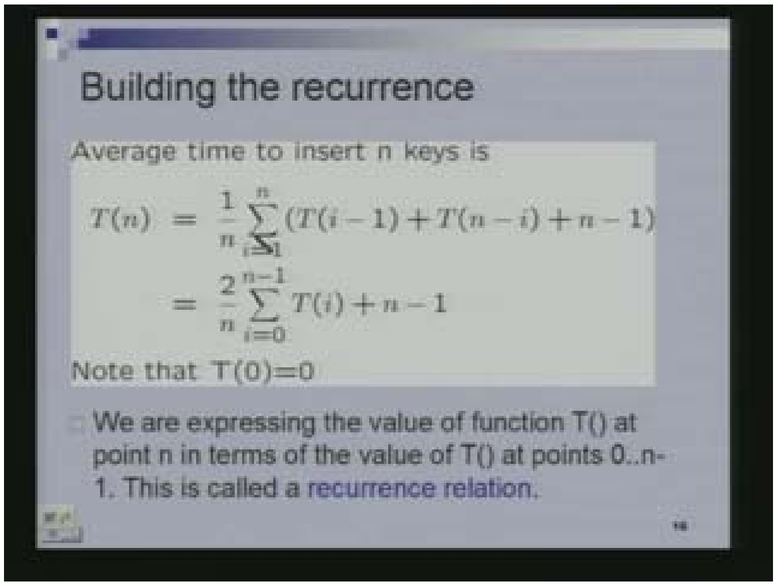
Note that each of these terms T (i-1) and T (n-1) will appear twice. If I look at T (3), it will appear once for i equals 4 and it will appear once for i equals (n-3). Each of the term will appear twice, so that this part of the sum $T(i-1)+T(n-i)$ is 2 times T (i) where I going from 0 through n-1.

This part of (n-1) have moved out, so it is n-1 where I go from 0 to n divided by n, so it becomes just n-1. This n-1 is not multiplied by $\frac{2}{n}$ and this $\frac{2}{n}$ only multiplies the following part.

Why do i go from 0 through n-1?

Because you will never generate T (n) neither from T (i-1) nor from T (n-i). You will generate T (0), when i equals 1 you will have T (0) and when i equals n you will again have T (0).

(Refer Slide Time: 42:16)



Building the recurrence

Average time to insert n keys is

$$T(n) = \frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i)+n-1)$$
$$= \frac{2}{n}\sum_{i=0}^{n-1}T(i)+n-1$$

Note that T(0)=0

- We are expressing the value of function T() at point n in terms of the value of T() at points 0..n-1. This is called a recurrence relation.

What is T (0)? T (0) is zero, in fact T (1) is also zero. T (2) is not zero but T (1) is zero because it is just a single element. We say it is at level 0 so we just do not count it any time for inserting it. Such a thing is called recurrence relation.

The function T is a function of n, we are expressing the value of this function at point n in terms of its values at previous points. I can use the below given relation to compute T (0), T (1), T (2) and so on.

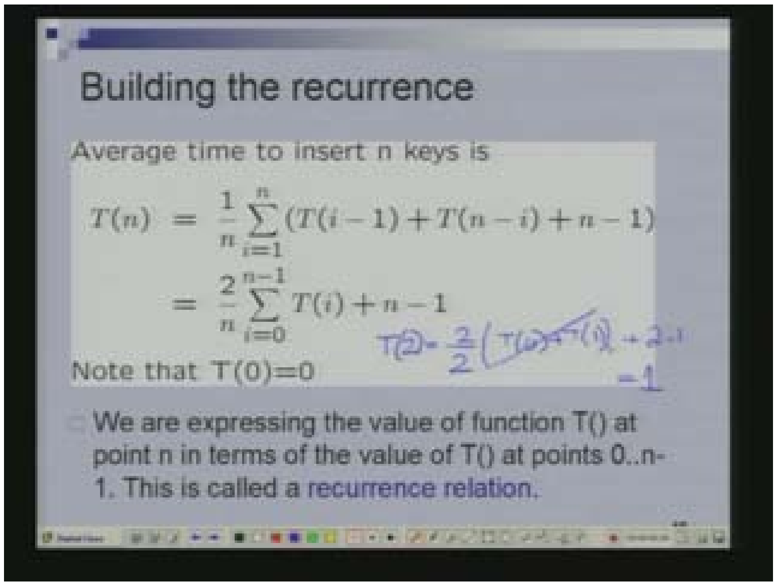
$$\frac{2}{n}\sum_{i=0}^{n-1}T(i)+n-1$$

T (0) is zero. What is T (1)?

$$\frac{2}{n}\sum_{i=0}^{n-1}T(i)+n-1$$

When we substitute n=1 in the above equation, we get T (1) as zero which we argued it before. So T (0) and T (1) both are zero. I can use that to compute T (2). What will I get the value of T (2)? If I were to just use this expression I would get 1 which is explained in the slide given below. Thus T (2) turns out to be 1. If you are not convinced so far lets also compute T (3). T (3) will be $\frac{2}{3}$ times T (0) which is zero so let me ignore that. T (1) which is also zero so let me ignore that and T (2) which is non-zero so let me leave that around plus n equals 3-1.

(Refer Slide Time: 43:43)



Building the recurrence

Average time to insert n keys is

$$T(n) = \frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i)+n-1)$$
$$= \frac{2}{n}\sum_{i=0}^{n-1}T(i)+n-1$$

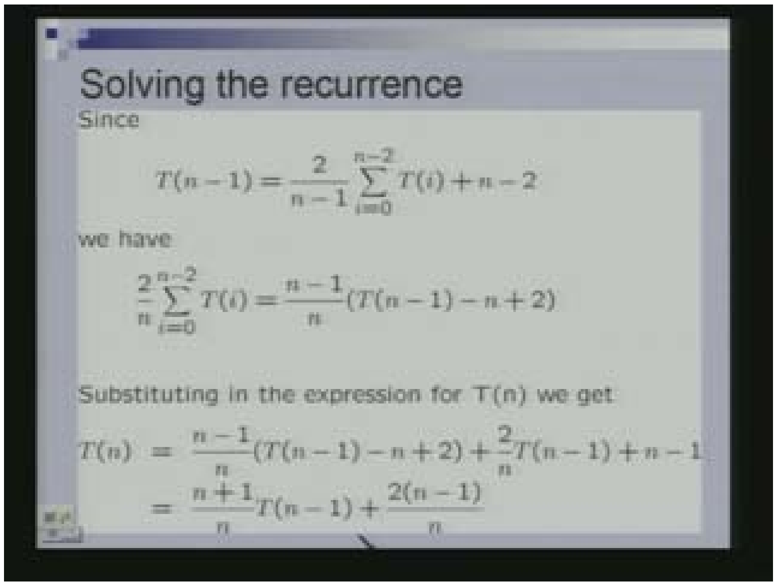
$T(2) = \frac{2}{2}(T(0)+T(1)) + 2-1$
$= 1$

Note that T(0)=0

- We are expressing the value of function T() at point n in terms of the value of T() at points 0..n-1. This is called a recurrence relation.

T (2) was 1 so this becomes $\frac{2}{3}$ + 2 which is 2.66 that is exactly what we computed. You can use this recurrence to compute any T (I). We have done something fairly sophisticated. We have obtained some kind of a relation which tells us any T (n) provided we know the previous values. But we will do a little bit more, we will try to figure out how to solve this recurrence relation also. That is what we are going to do now.

(Refer Slide Time: 45:20)



Solving the recurrence

Since

$$T(n-1) = \frac{2}{n-1}\sum_{i=0}^{n-2}T(i)+n-2$$

we have

$$\frac{2}{n}\sum_{i=0}^{n-2}T(i) = \frac{n-1}{n}(T(n-1)-n+2)$$

Substituting in the expression for T(n) we get

$$T(n) = \frac{n-1}{n}(T(n-1)-n+2)+\frac{2}{n}T(n-1)+n-1$$
$$= \frac{n+1}{n}T(n-1)+\frac{2(n-1)}{n}$$

The T (n) is $\frac{2}{n}$ times the sum of the previous T plus n-1. I can write T (n-1) is $\frac{2}{n-1}$ times the sum of the previous T + (n-2). I just replace the n by n-1 and so I can just rearrange this quantity $\frac{2}{n-1}\sum_{i=0}^{n-2} T(i)$ or $\frac{2}{n}\sum_{i=0}^{n-2} T(i)$ times this sum equals , I have moved this n-2 on the other side so I get T (n-1) – n+2 and I have scaled it by $\frac{n-1}{n}$ because I am trying to compute $\frac{2}{n}$ instead of $\frac{2}{n-1}$.

$$\frac{2}{n}\sum_{i=0}^{n-2} T(i) = \frac{n-1}{n}(\text{T (n-1)-n+2})$$

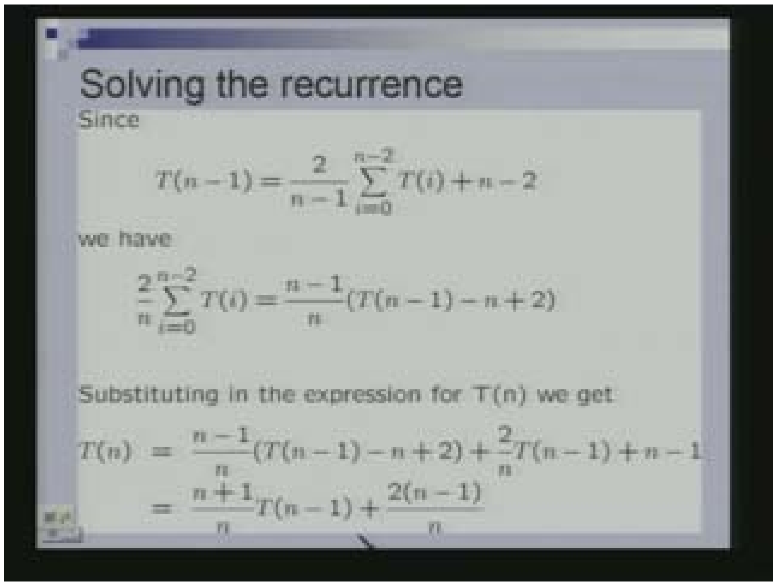
If the above equation were $\frac{2}{n-1}$ times, this would just have been this part (T (n-1)-n+2)), if you cancel out the n the $\frac{2}{n-1}$ is exactly this part (T (n-1)-n+2)). I just want to compute this quantity $\frac{2}{n}\sum_{i=0}^{n-2} T(i)$, because now am going to substitute this in the previous expression that is $\frac{2}{n}\sum_{i=0}^{n-1} T(i) + n - 1$.

I have computed 0 through n-2 as this quantity $\frac{n-1}{n}(T(n-1) - n + 2)$, so am going to substitute that. Thus T (n) then becomes this sum $\frac{n-1}{n}(T(n-1) - n + 2)$ + $\frac{2}{n}$ T (n-1) +n-1. Then I just did some rearrangement, that is $\frac{n-1}{n}$ and $\frac{2}{n}$ that makes it $\frac{n+1}{n}$. The $\frac{n-1}{n}$ cancels out and so I am just left with 2 times $\frac{n-1}{n}$. So we will work with the below equation.

$$\frac{n+1}{n}T(n-1) + \frac{2(n-1)}{n}$$

All I have done is some rearrangement to simplify things. I have written T (n) in terms of T (n-1). T (n) was in terms of all the previous T. This is the simplification I have achieved with this point.

(Refer Slide Time: 47:15)



Solving the recurrence

Since

$$T(n-1) = \frac{2}{n-1}\sum_{i=0}^{n-2} T(i) + n - 2$$

we have

$$\frac{2}{n}\sum_{i=0}^{n-2} T(i) = \frac{n-1}{n}(T(n-1) - n + 2)$$

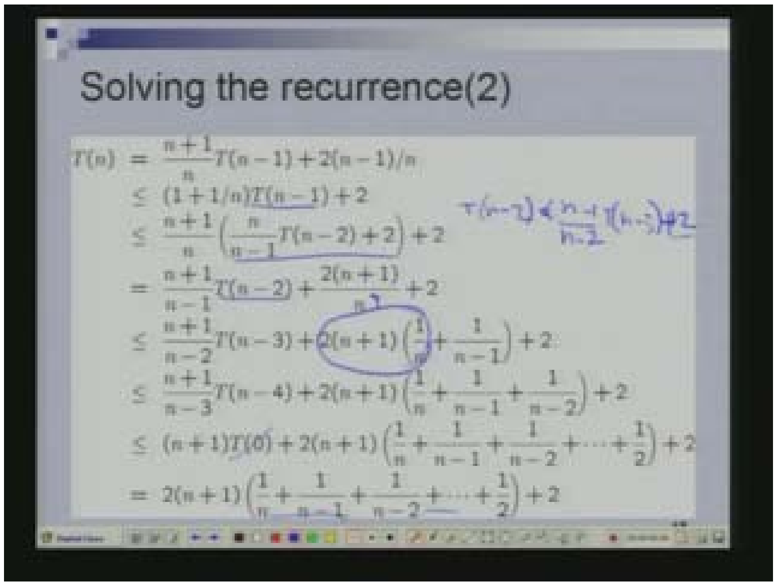
Substituting in the expression for T(n) we get

$$\begin{aligned} T(n) &= \frac{n-1}{n}(T(n-1) - n + 2) + \frac{2}{n}T(n-1) + n - 1 \\ &= \frac{n+1}{n}T(n-1) + \frac{2(n-1)}{n} \end{aligned}$$

Let us see. It is a bit of a dense slide but we will run through this very quickly. This is what we have $\frac{n+1}{n}T(n-1)+\frac{2(n-1)}{n}$. I will just replace that is $\frac{2(n-1)}{n}$ is less than 2 because n-1 is less than n. That is why the less than or equal to, actually it should be strictly less because $\frac{n-1}{n}$ is strictly less than 1.I just simplify this a bit and now I am going to do the following.

I am going to replace T (n-1) by using this recurrence relation that is by using this expression $\leq \frac{1+1}{n}T(n-1)+2$. The $\frac{1+1}{n}$ which is $\frac{n}{n-1}$ times T (n-2) +2, where the other +2 is coming from the before expression. All I have done is replacing T (n-1) with $(\frac{n}{n-1}T(n-2)+2)$.

(Refer Slide Time: 49:25)



Solving the recurrence(2)

$$\begin{aligned} T(n) &= \frac{n+1}{n}T(n-1)+2(n-1)/n \\ &\le (1+1/n)\underline{T(n-1)}+2 \\ &\le \frac{n+1}{n}\underline{\left(\frac{n}{n-1}T(n-2)+2\right)}+2 \\ &= \frac{n+1}{n-1}\underline{T(n-2)}+\frac{2(n+1)}{n}+2 \\ &\le \frac{n+1}{n-2}T(n-3)+2(n+1)\left(\frac{1}{n}+\frac{1}{n-1}\right)+2 \\ &\le \frac{n+1}{n-3}T(n-4)+2(n+1)\left(\frac{1}{n}+\frac{1}{n-1}+\frac{1}{n-2}\right)+2 \\ &\le (n+1)T(0)+2(n+1)\left(\frac{1}{n}+\frac{1}{n-1}+\frac{1}{n-2}+\cdots+\frac{1}{2}\right)+2 \\ &= 2(n+1)\left(\frac{1}{n}+\frac{1}{n-1}+\frac{1}{n-2}+\cdots+\frac{1}{2}\right)+2 \end{aligned}$$

$T(n-2) \le \frac{n-1}{n-2}T(n-3)+2$

Let me just simplify it. The n and n cancels in that expression so I get $\frac{n+1}{n-1}T(n-2)$ and 2 is the same then $\frac{2(n+1)}{n}$.

$$\frac{n+1}{n-1}T(n-2)+\frac{2(n+1)}{n}+2$$

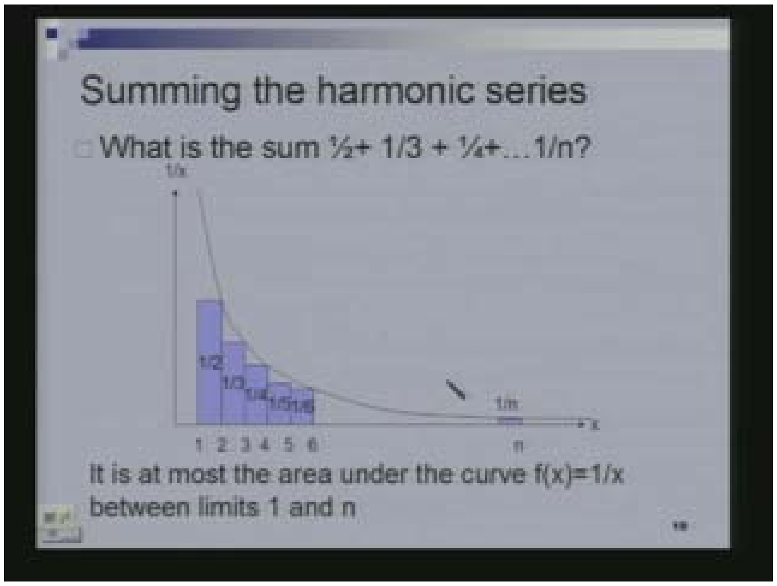
This is just the simplification and it is exactly equal. Once again I am going to do the same. I am replacing T (n-2) with the value of that.

What is T (n-2)? It would be T (n-2)≤ $\frac{n-1}{n-2}T(n-3)+2$, that is what I am going to replace it for T (n-2). When I do that (n-1) gets cancelled and I get $\frac{n+1}{n-2}$T (n-3). This $\frac{n+1}{n-1}$ multiplies with this 2 and gives $\frac{2(n+1)}{n-1}$. This part $\frac{2(n+1)}{n}$ is the same as $2(n+1)\frac{1}{n}$. When I replace it I get this $\frac{n+1}{n-3}T(n-4)$ and this quantity gets decreasing with every step and the sum inside keeps increasing with every step. Next time I will add $\frac{1}{n-2}$, the next time I will add $\frac{1}{n-3}$ and so on. Eventually it goes down and when T (n-4) goes down to (n-n), I get a zero. And in the denominator I would get (n-n-1) so it is 1 and I just get (n+1). Inside I will get all the terms going up to $\frac{1}{2}$+2. Since T (0) is 0 that part goes away and I get exactly the equation which is given below.

$$2(n+1)(\frac{1}{n}+\frac{1}{n-1}+\frac{1}{n-2}+...+\frac{1}{2})+2$$

So I get T (n) is less than or equal to the above sum. This is $(\frac{1}{n}+\frac{1}{n-1}+\frac{1}{n-2}+...+\frac{1}{2})$ the sum of a harmonic series and it is going up all the way up to $\frac{1}{n}$.

(Refer Slide Time: 52:47)



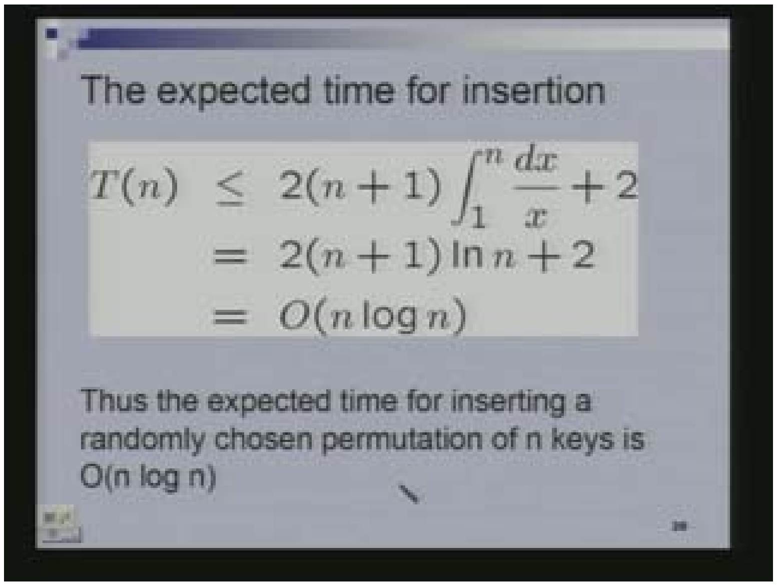
Summing the harmonic series

- What is the sum ½+ 1/3 + ¼+…1/n?

It is at most the area under the curve f(x)=1/x between limits 1 and n

We are summing a harmonic series, we have to sum $\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+...\frac{1}{n}$. One trick that we employ very often, you take this graph and this is a graph of a function $\frac{1}{x}$. These are the points 1, 2, 3, 4 and let me draw line at 2.
What is the height of this rectangle?
It is $\frac{1}{2}$, the width of this rectangle is 1 and its area is $\frac{1}{2}$. I have drawn a line at 3, so its area is $\frac{1}{3}$. I am getting the terms so next one is $\frac{1}{4}$, $\frac{1}{5}$ and the last one at n, I will draw it in this manner.

(Refer Slide Time: 53:27)



The expected time for insertion

$$\begin{aligned} T(n) &\le 2(n+1)\int_1^n \frac{dx}{x} + 2 \\ &= 2(n+1)\ln n + 2 \\ &= O(n \log n) \end{aligned}$$

Thus the expected time for inserting a randomly chosen permutation of n keys is O(n log n)

This sum is less than or equal to the area under the curve f (x) = $\frac{1}{x}$ and it is atmost the area under the curve in between the limits 1 through n. Recall T (n), it is $2(n+1)(\frac{1}{n}+\frac{1}{n-1}+\frac{1}{n-2}+...+\frac{1}{2})+2$ that is harmonic sum +2. I am going to just put that 2 (n+1) and that harmonic sum was the integral of $\frac{1}{x}$ from the limits 1 through n +2. This integral part is just log (x). It will be just log of n and 2 (n+1) ln n +2 is exactly the sum which is O (n log n).

T (n) $\le$ O (n log n)

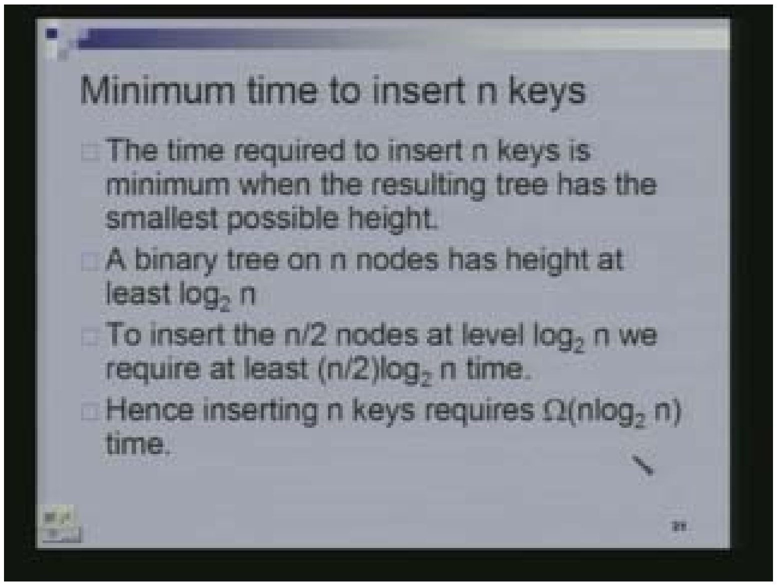
It is the expected time for inserting the n keys. We have done a very sophisticated computation.

We looked at all the n! different permutations possible and also we computed the total time taken to insert all the n! different permutations and we took the average. We got n log n. Recall we saw the worst case was n^2 and there is a particular permutation which puts the elements in increasing order for that the total time for insertion is n^2. But the average time is n log n.

The best time is also n log n and the best time would happen when the tree was very shallow. If you have a tree on n nodes and it has a height atleast log n, log n-1 or roughly $\log_2 n$. If you take a complete binary tree of height log n the last level has $\frac{n}{2}$ keys that is in the leaf there are $\frac{n}{2}$ leaves. How much time you will take to insert each one of those leaves?

Log n, because you will have to come down the entire tree. So to insert each one of those $\frac{n}{2}$ keys you are taking $\log_2 n$ time. Just to insert the leaves you take $(\frac{n}{2})$ $\log_2 n$ time which is Ω $(n\log_2 n)$ time a function which is growing faster than $n\log_2 n$. It is some constant time but it is $n\log_2 n$.

(Refer Slide Time: 55:55)



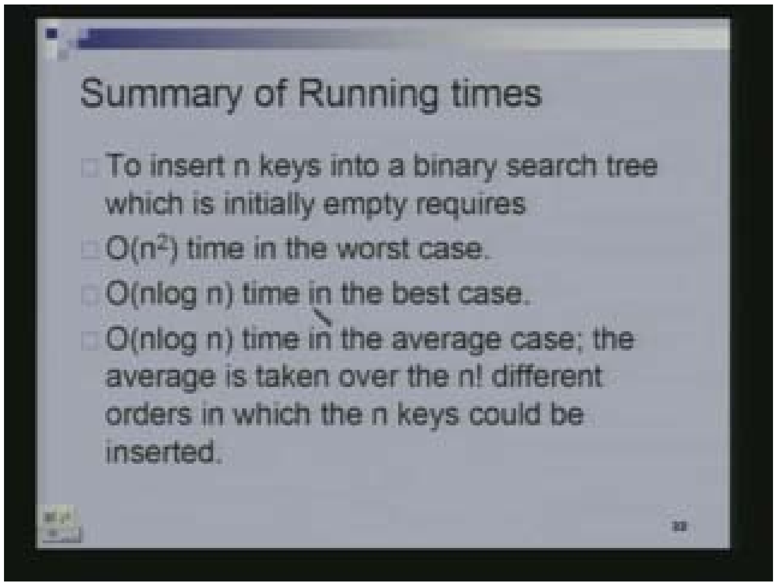
Minimum time to insert n keys

- The time required to insert n keys is minimum when the resulting tree has the smallest possible height.
- A binary tree on n nodes has height at least $\log_2$ n
- To insert the n/2 nodes at level $\log_2$ n we require at least (n/2)$\log_2$ n time.
- Hence inserting n keys requires $\Omega(n\log_2 n)$ time.

In the best case you are taking n log n time, in the worst case you are taking n^2 time but the average case is also n log n. That is a very interesting part. If I would ask you the average case, you would just take the average of the above 2 cases and report it. Why it is turning out to be O (n logn)? What does it convey? Most of the trees are fairly shallow and do not have too much depth. Most of the trees have depth of only about log n or some constant times n log n. So that the average is still only n log n.

You have a bunch of numbers, the numbers here are the total time for insertion under the various permutations. The smallest of those numbers is n log n, the largest of those numbers is n^2 but the average is also n logn or some constant times n log n. That means most of the numbers are closer to the minimum than to the maximum.

(Refer Slide Time: 56:20)



Summary of Running times

- To insert n keys into a binary search tree which is initially empty requires
- O(n^2) time in the worst case.
- O(nlog n) time in the best case.
- O(nlog n) time in the average case; the average is taken over the n! different orders in which the n keys could be inserted.

That is all I was going to discuss in the class today. What we discussed today was the deletion procedure and then we also discussed this particular analysis which says that if you were to take a random permutation of n elements and insert them, then the average time or the expected time for insertion is n log n. In the worst case there could be a permutation for which the time taken is n^2. We have seen examples of those, if the elements are in increasing order or in decreasing order then we take n^2 time. The best possible permutations are also going to take at least n log n times or some constant times n log n. But the average case is also n log n.