Programming and Data Structure

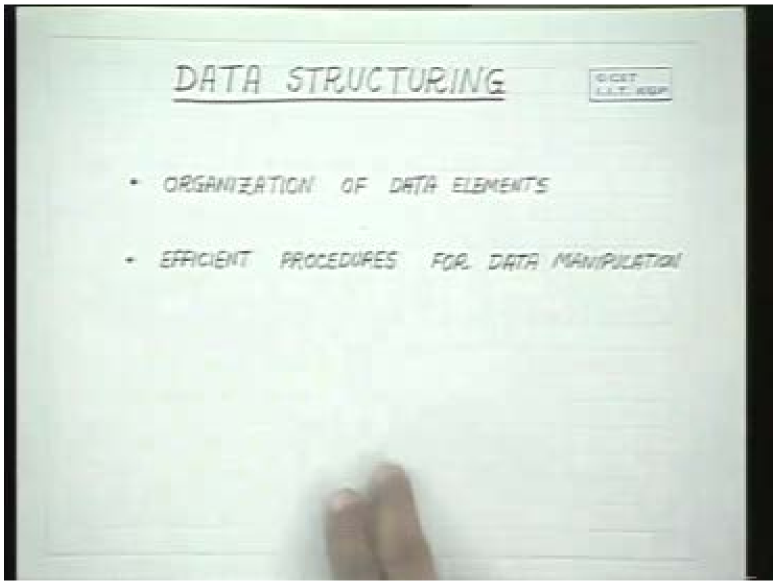
Dr.P.P.Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture 22

Search Trees

We have been discussing data structuring in the previous class and we found out that a large number of problems are pure data structuring problems, the most problems are mixture of algorithm design and data structuring. And we will continue our study of data structuring and to find out what are the efficient ways of solving a large number of problems. Data structuring has got two parts, one is the organization of the data elements that is what ever is the data how you are going to store the data, in what form and exactly in what structure and format. And secondly these data will be operated by a number of procedures. So the procedures are the functions which operate on the data must be efficient. Therefore data structuring inherently consists of two parts, one is the organization of the data elements and the other is efficient procedures for data manipulation. We saw a number of examples which were pure data structuring problems. For example we discussed complex numbers with operations add, subtract, read and write.

(Refer Slide Time 02:02)

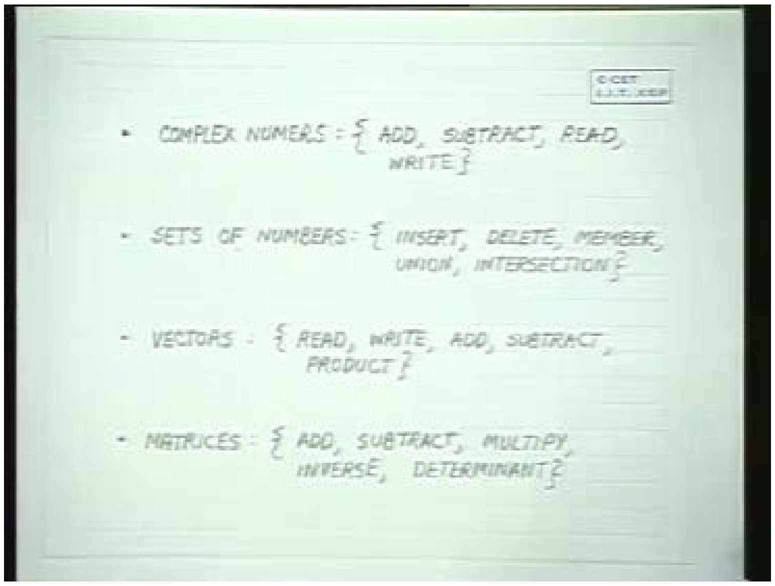


DATA STRUCTURING

- ORGANIZATION OF DATA ELEMENTS
- EFFICIENT PROCEDURES FOR DATA MANIPULATION

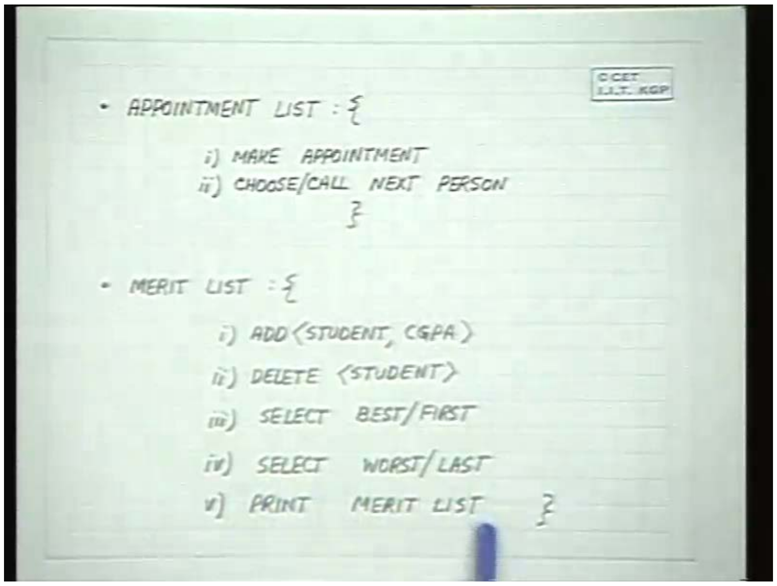
We discussed the problem of representing a set of numbers with the operations insert a number, delete a number, find out whether it is a member of a set, is a set of sets. So insert a number in a set x, delete a number from a set, find out whether a number is a element of a set, make a union of two sets, make an intersection of two sets. So this problem is a pure data structuring problem. Similarly on vectors you can have several operations, on matrices you can have operations.

(Refer Slide Time 02:48)



- COMPLEX NUMERS : { ADD, SUBTRACT, READ, WRITE }
- SETS OF NUMBERS : { INSERT, DELETE, MEMBER, UNION, INTERSECTION }
- VECTORS - { READ, WRITE, ADD, SUBTRACT, PRODUCT }
- MATRICES : { ADD, SUBTRACT, MULTIPY, INVERSE, DETERMINANT }

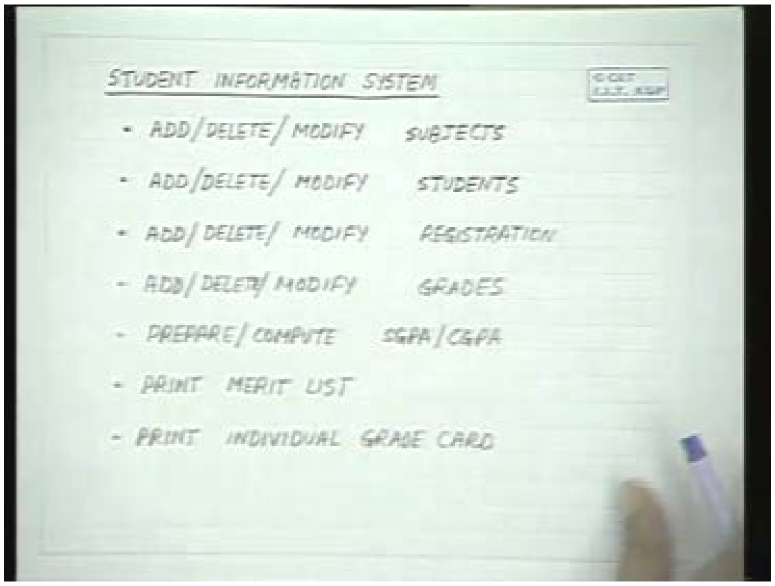
We also discussed some other problems like the problem of making an appointment list, a merit list of students with add a student, delete a student, select the first or the best, select the last or the worst then printing the merit list. And in general you can have a big problem like a student information system where subjects can be added, deleted or updated, student information can be added, deleted or updated, registration for students in particular subjects can be updated. The grades students obtained in the subjects is the register, sgpa cgpa calculation, grade card preparation, merit list preparation. So these problems inherently require storage of some data and some functions or procedures which manipulate this data. So these are pure data structuring problems which we are currently discussing.
(Refer Slide Time: 03:04)



- APPOINTMENT LIST : {
  - i) MAKE APPOINTMENT
  - ii) CHOOSE/CALL NEXT PERSON

  }
- MERIT LIST : {
  - i) ADD ⟨STUDENT, CGPA⟩
  - ii) DELETE ⟨STUDENT⟩
  - iii) SELECT BEST/FIRST
  - iv) SELECT WORST/LAST
  - v) PRINT MERIT LIST }

(Refer Slide Time 03:38)



STUDENT INFORMATION SYSTEM

- ADD/DELETE/MODIFY SUBJECTS
- ADD/DELETE/MODIFY STUDENTS
- ADD/DELETE/MODIFY REGISTRATION
- ADD/DELETE/MODIFY GRADES
- PREPARE/COMPUTE SGPA/CGPA
- PRINT MERIT LIST
- PRINT INDIVIDUAL GRADE CARD

So let us come to one problem which we saw in the previous class. We have seen some of them, though we have not seen all of them, we have not seen a problem like this but we saw how to make a simple problem, solve a simple problem like this using a linked list were to make an appointment, you would put it at the end of the list and to call the next person you would remove it from the beginning of the list then you would store this information in a linked list. We also saw how you will store complex numbers, how we would solve the problem of vectors and even discussed briefly this. We went into details with this and we saw that the complexity of each operation, the order of complexity of each operation has to be checked and we were interested in minimizing the worst case complexity.

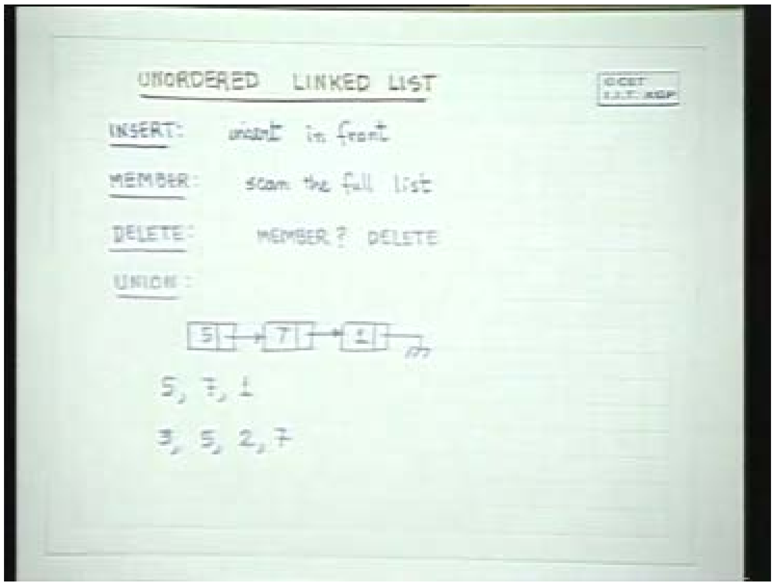
So we will just quickly review this problem of sets of numbers before we proceed on. So in sets of numbers the operations were insert an element into a set, delete an element from a set, whether an element is a member of a set, union of two sets, to form a third set, intersection of two sets, to form a third set. And the options that we saw, we discussed that we will have to implement it as a linked list. An array will not do because this is a dynamic structure where these elements due to insertion and deletion may vary from one to anything and we do not know the total size, therefore having a static structure was not possible. So we require a dynamic structure with mallocing of it and therefore we implement it as a linked list which conceptually looked like this.

At any point of time the node structure was a data field and a link field and the linked list at any point of time there is something like this say for example 5 3 9 1 and you had a pointer to this node structure which indicated what is the start of the list and may be we had a pointer to indicate what is the end of the list. So this was the conceptual linked list structure. I hope you all remember what we were doing here, the c declarations etc, how to malloc 1, all this we have already discussed. Now in this linked list data structure, we saw for this particular operation we had two options. Now we are giving them names, the options were actually on the basis of how we store the list.

The first operation was insert in the beginning of the list. The insert operation was implemented by inserting in the beginning of the list; the delete operation was implemented by scanning the list, searching where the element was and then deleting that element. Member were also scanning the list and finding out whether it existed, union of two was done by starting with two lists and in one list you hold the pointer to the element and in the other list you find out whether it exists or not. If it exists then you delete both of them and store it in the third list, otherwise you delete it only from the one list. And you continue this till you finish off both the lists, that is once you finish both list the other list can be just looked at the end. Should I repeat the whole thing again or is it okay? Okay. Student: Sir why do we disturb the two lists. No, we don't need to disturb the two lists or whether we need to, we will come to it.

Let's take an example, okay. So let us work out the simple unordered linked list. Is this visible, is this color visible? I will change the color. In the unordered linked list the insert operation was insert in front, that was simple.

(Refer Slide Time 10:10)


UNORDERED LINKED LIST

INSERT: insert in front

MEMBER: scan the full list

DELETE: MEMBER ? DELETE

UNION:

$$\begin{tikzcd} \boxed{5} \arrow[r] & \boxed{7} \arrow[r] & \boxed{1} \end{tikzcd}$$

5, 7, 1

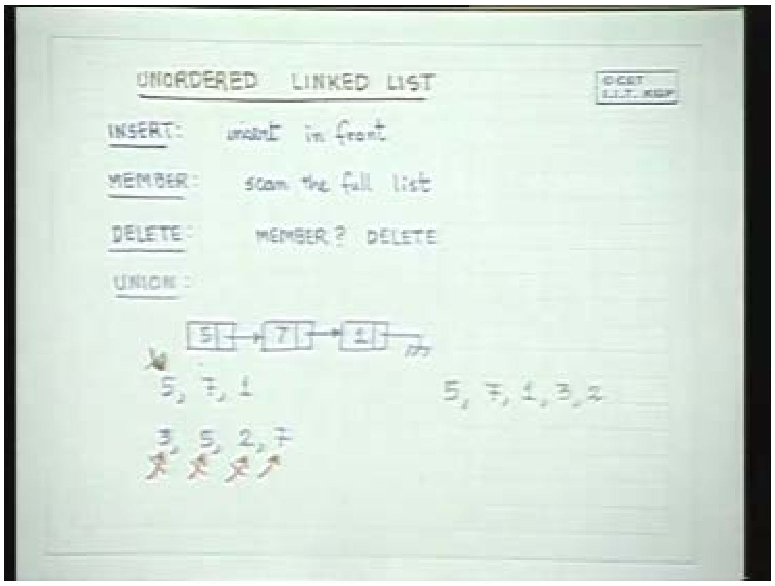
3, 5, 2, 7

The member operation: scan the full list, the delete operation was actually first do member and if it is present then delete. This part is okay, I don't need to go into details. Let's come to union. What was the union operation? We are giving two lists, so in our format the linked list structure of the two lists, any list is like this say 5 7 1 and they are just put in front because 1 came first then 7 then 5 so they are put like this. Now suppose we have 5, in one list one list 5 7 1 and in another list suppose we have 3 5 2 and 7. Now we are doing the union of these two.

So how do we do a union? We start with this list, so with this list the first element if you take and we have put, making the union in the third set. So we start here and put this element in the third set and we progress this pointer but before this progressing the pointer here, we must ensure we cannot just put this followed by this because they are not disjoint. Since they are not disjoint, we cannot take this one and just follow it up with this one. So now there are 2 or 3 things we can do. One is we can first put this whole list up here all right and then we start with this one whereas take this option, it will be easier. So we take this one and put all of them then we start with the second list, we see whether this is present here or not.

So we scan this, it is not present so we put it here then we come here, we scan this it is present here, so we don't put it here. We proceed, again we scan this, it is present here no, put it here. Again we scan change the pointer, we scan this it is present so we don't put it. So this is how we end up in doing the union.

(Refer Slide Time 12:05)



UNORDERED LINKED LIST

IIT KGP

INSERT: insert in front

MEMBER: scan the full list

DELETE: MEMBER? DELETE

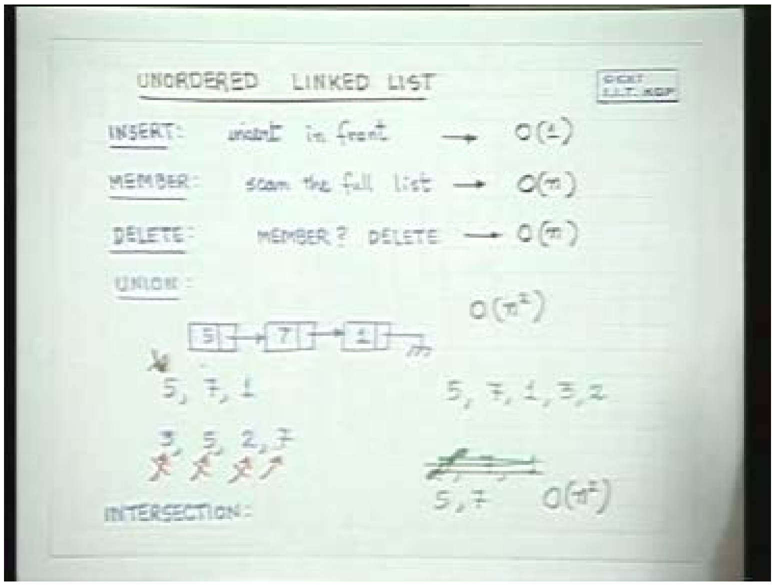
UNION:

5
7
1
5, 7, 1
5, 7, 1, 3, 2
3, 5, 2, 7

Can anybody tell me how you will do the intersection? How will we do the intersection? We take this list and put it here, take this one see if it is not there fine. Take this one, if it is there delete it sorry sorry take this out, see if it is already there. Okay let's do it in another way. We will start from this one and see if it is here, if it is not present here we don't put it. Take this one see if it is here, if it is here we put it. Take this one see if it is here, if it is not here we don't put it. Take this one, we put it. So now we have defined our algorithms where the storage of the data is how we are storing the data? As a node like this and conceptually we are storing it in as in a sequence as and how they come.

Now we have to analyze how efficient it is. This one takes constant term, so we say it is order one. This one is proportional to the size of the current list, so it is order n. This is also of order n. This you would, if you are intelligent you would put the larger value first and then start with the smaller one but on an average the smaller one can be half in the worst case. In the worst case the smaller one can be half the size and we would possibly have to scan all of them, so it would be n by 2 square.

(Refer Slide Time 14:41)



UNORDERED LINKED LIST

INSERT: insert in front $\rightarrow O(1)$

MEMBER: scan the full list $\rightarrow O(n)$

DELETE: MEMBER? DELETE $\rightarrow O(n)$

UNION: $O(n^2)$

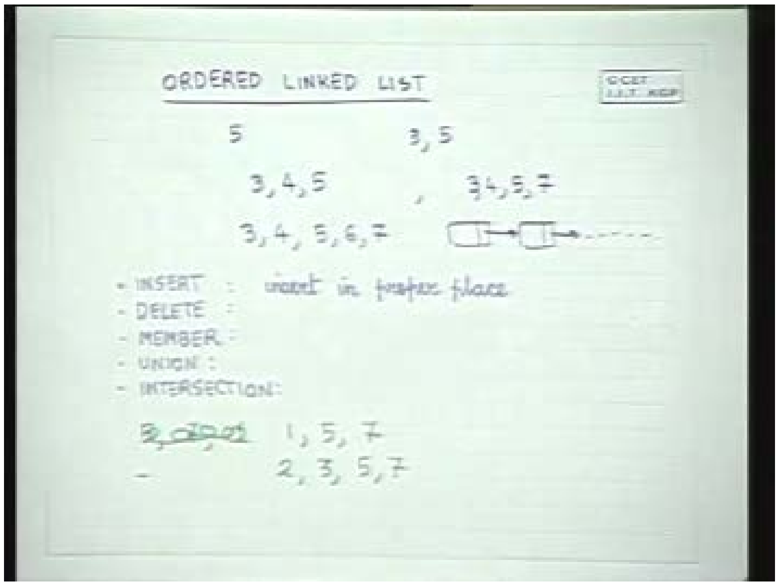
5
7
1
5, 7, 1
5, 7, 1, 3, 2
3, 5, 2, 7
5, 7
$O(n^2)$

INTERSECTION:

For each one we have to scan all the elements of the other one. It is n by 2 into n by 2, so this will come to order n square. Similarly this will come to order n square. If all of them occur with equal probability which let us assume is the case then order n square is the worst case complexity that we have obtained. What do we do in an ordered linked list? In an ordered linked list, we make certain modifications, we insert we always maintain the list in an ordered fashion. That is suppose first element entered is 5, the second element entered is 3 then this 3 will be put before 5.

If the next element entered is 4 then it will be put here. If the next element entered is 7 it will be made, if the next number is 6 it will become and this structure obviously as you can imagine is of this form. So insert now puts it in the proper position, all right. Insert puts it in a proper position, so insert in proper place. Delete, you scan and once you find it, you delete it, member is similar, union and intersection. Can you tell me how you would do union here? Let us take the same set of elements 5 7 1, so 5 7 1 would now be stored as 1 5 7 and 3 5 2 7 would now be stored as 2 3 5 7.

(Refer Slide Time 17:14)



ORDERED LINKED LIST

5 3, 5

3, 4, 5 , 3, 4, 5, 7

3, 4, 5, 6, 7

- INSERT : insert in proper place
- DELETE :
- MEMBER :
- UNION :
- INTERSECTION :

[illegible] 1, 5, 7

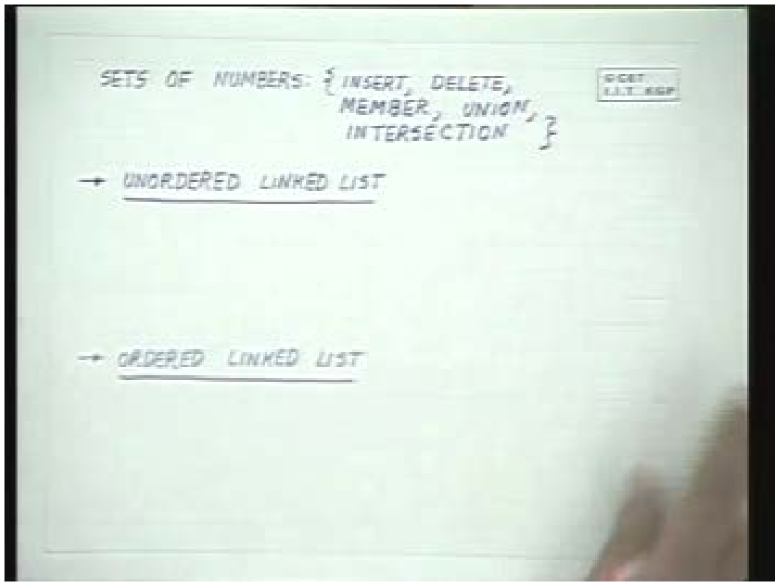
– 2, 3, 5, 7

To the user it is a set, to the user it is just a set with these operations. This is how we are implementing it. We are implementing it as an ordered list but to the user it is a set. So how would we do a union? If you recall the previous discussion, in the previous review do it like you did in merge routine in merge sort which start with these two because when we do union also, we would also have to implement it in an ordered fashion, the final result would be ordered. So we can modify our merge routine of merge sort and just implement it. The smaller of the two is first put in, 1 and this is pointer is shifted and these two, 2, this is shifted. These two, 3, this is shifted, both are equal put 1 and shift both right that's the only difference with merge sort. Both are same, put it here and shift both.

Now here we have reached the end of both the lists, we might not have reached the end of both the list, we have reached the end of one list and not the end of other, we will just part it up at the end just like we do in merge sort. So insertion takes how much time? Order n. Delete: order n. Member: order n. Union takes how much time? Order n_1 plus n_2, the total size is n, so this is also of order n. And how would you do intersection? In the same way, you would see 1 and 2 and smaller one without writing you would push. Here without writing you would push.

So let's just quickly revise how we will do this one. You start here and here, 1 and 2, say they are different so the smaller one is pushed out, nothing is done. They are unequal, the smaller one is pushed. They are unequal the smaller one is pushed, they are equal this is written and both are pushed. They are equal, this is written and both are pushed. And if you reach the end of one list, you are over; you don't need to continue with the rest. So this would also be order n. Now, see what we have done. Order one has been made order n that these two were order n, these two were n square have been reduced to n and if they are of equal probability, this is a better data structure in the worst case because you know this way you have equal probability but if you have a variation in probability and relationship that this occurs much much less than this. Then may be something else would have come out, you would have to do a deeper analysis. So in this data structure, for this problem among the two options may be we would choose this option. The question is are there other option or is this the best option. So we will just see whether there are other options and we will simplify the problem and we will remove this union and intersection and try to solve the problem in a simpler fashion.

(Refer Slide Time 21:20)



SETS OF NUMBERS: { INSERT, DELETE, MEMBER, UNION, INTERSECTION }

→ UNORDERED LINKED LIST

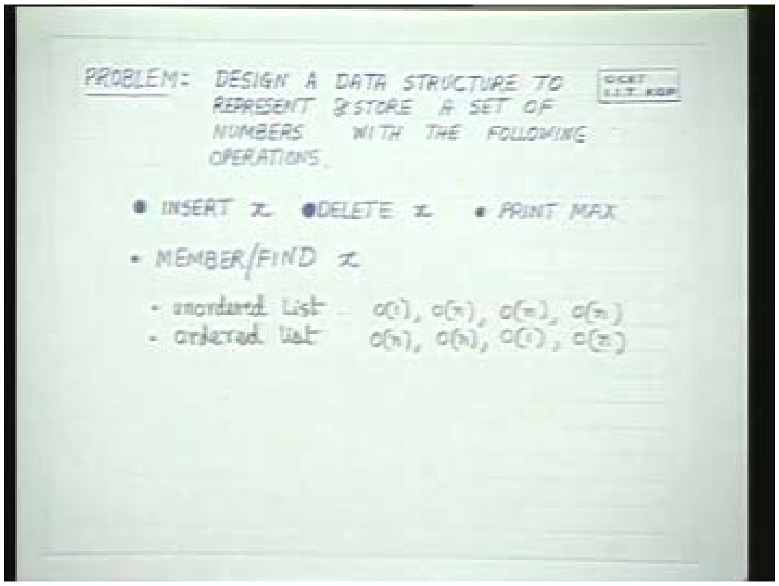
→ ORDERED LINKED LIST

C-DET
I.I.T. KGP

Then we will reintroduce these two and see whether we can get a better data structure. So the problem that we have in hand now is this. Design a data structure to represent and store a set of numbers with the following operations. Insert x, now in one set not in a set of sets because union and intersection are not there, we have simplified it to one set of numbers, inserting the z, delete from the z, print the largest in the set and member, whether x is a member of the set. So these three are same, print the largest, it does not affect our routine because in an unwanted list finding out the largest would have taken order n times but in an ordered list, finding out the largest will take constant time because the end pointer is there and you can immediately print the largest element.

So we have got now several options here, the first option is unordered list, the second is ordered list. In an unordered list insert is constant, order n, order n. So order one, order n order n and order n and in an ordered list, it is order n, order n, order one and order n. Competitive, both are competitive and you would have just argue between the probable times insert comes and print, max comes and they are both equivalent in nature.

(Refer Slide Time 23:14)



PROBLEM: DESIGN A DATA STRUCTURE TO REPRESENT & STORE A SET OF NUMBERS WITH THE FOLLOWING OPERATIONS

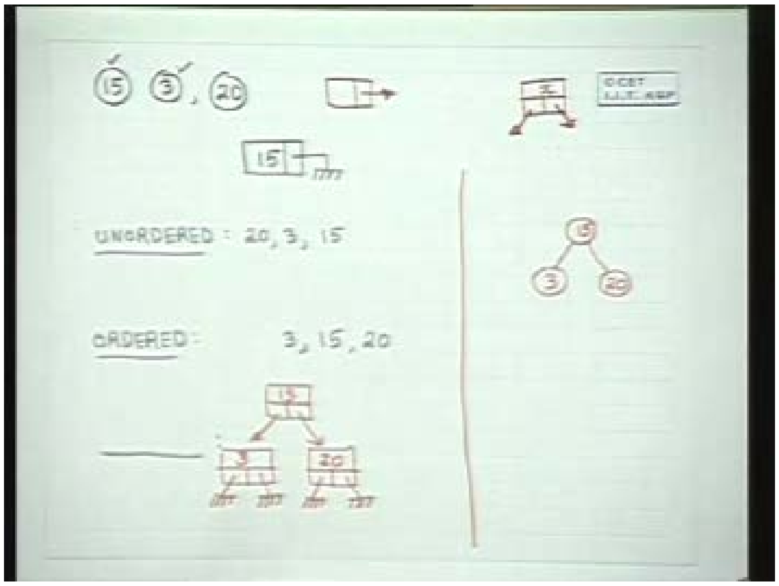
- INSERT x • DELETE x • PRINT MAX
- MEMBER/FIND x
  - unordered List $O(1)$, $O(n)$, $O(n)$, $O(n)$
  - ordered List $O(n)$, $O(n)$, $O(1)$, $O(n)$

Now let us see whether we can do something more interesting and I will try to show you by example how something more interesting can be done. Suppose the first number that comes to us is 15. In an unordered list, in an unordered list we would store 15 like this both unordered and ordered like this. So when the first number comes that's all we do. Now suppose another number comes 3, these are insert operations I am giving you sequence of insert operations. So in an unordered list, in an ordered list and in something else what we would do? In an ordered list, we would write 3 15, in an ordered list 15. This is one after another like this and in unordered, no, no, let us see we are putting the smaller earlier and in the unordered list we are just inserting in the front. So first 15 comes then 3 comes, so 15 is put here then 3 is put.

And I am trying to show you how you could do something else and in this something else we would put 15 here and this has got two pointers. This points to all elements which are less than it; this points to all elements which are greater than it. In the ordered list there is only one pointer pointing to all elements which are greater than it. In an unordered list this pointer had just got no meaning, it was just one after another but in an ordered list, this pointer had the meaning that the elements which are this side are all greater than the given element.

Now I put in two, elements suppose I have x here. All those which are less than it will go this side; all those which are greater than it will go this side. So I would put this one here, right. Now suppose another element comes up 20, then in an unordered list 20 would come here. In an ordered list 20 would come here and here 20 would come here that is in your implemented structure, it would be… is this visible, right.

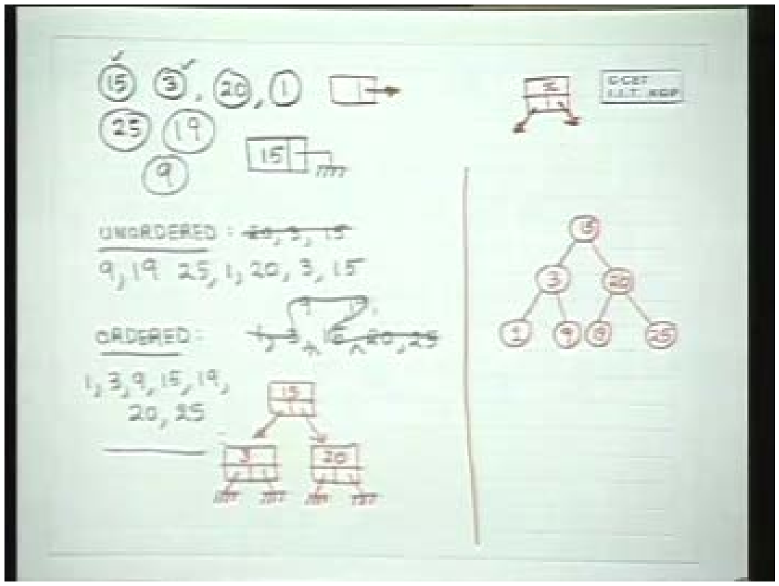
(Refer Slide Time 27:03)



Now suppose the member operation comes, here for three elements in the worst case for any member operation you would have to search for all the three, here also you would have to search all the three. But on an average here we would possibly terminate possibly earlier because a smaller element can come with a higher probability and all. Here all elements can be at the middle on an average but here some elements will be earlier, some elements here but here you would not have to make more than two comparisons. Here you would make at least 3 in the worst case. For any, suppose 15 could have a 3 here, if 20 came here you would have to make 3 here but here you would never make more than 2. How? You take, you read the first element. If the given element is greater you would only look this side, if it is smaller you would look only this side. So let's put in another element, let's put in 1. In the unordered list, it would now look 1 20 3 15. In the ordered list it would look like this and in this structure it would look like this. So, here 4 comparisons for finding.

See deletion also requires finding, so the complexity of member and deletion are identical. Here you would require 4 and 4 in the worst case but here 3 in the worst case, right. Let us put in another element, say 25, 25 would come here, 25 would come here and 25 would come here. How would you insert? You would first find where to insert. If you have to insert 25 in this, say 15, 25 is greater than 15 so you have to insert here. You come to 20, 25 is greater than 20 so you have to insert here. So there is no element here, this pointer is null, so you insert here we will come to details of this. But I am trying to just give you a conceptual structure. Algorithms for insertion, deletion will come but we are trying to see what the structure looks like. Now here we have to make 5 comparisons in the worst case but here no more than 3 states. Now suppose 19 comes then 19 is stored here, here 19 would be stored somewhere here, right.

(Refer Slide Time 31:09)

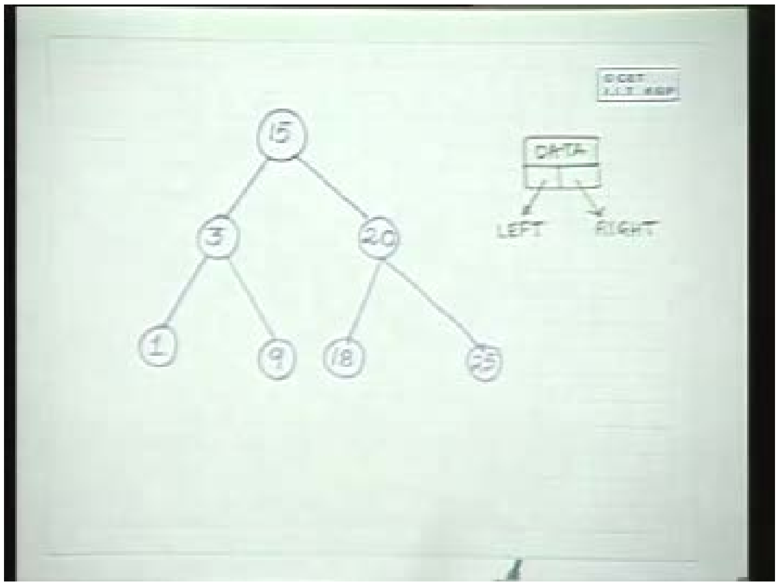


In the ordered list 1, 3, 15, 19, 20, 25 and here 19 would be stored here. Now 6, 6 still 3. Now suppose the next element is say 9, the 9 would be put somewhere here, so the ordered list now looks 1, 3, 9, 15, 19, 20, 25, right. This is the sequence here 9 would be… You can see I have picked up an example where this looks always better. I am trying to show a good case for this but we will come to bad cases also but let us see the good cases first. So this is how you would store it. So compared to an unordered list and an ordered list, this one has got two pointers. One, unordered list the pointer has no meaning it just says it is there as another element that's all.

In an ordered list all the elements, this points to all the elements which are greater and in this structure it points to… So it is a two linked list but it is not linking in a chain, these are les, so the one which has got say this pointer has got elements less, this full size will be less than 15. All elements here will be greater than 15, similarly all elements here will be less than 15, all elements here will be greater than 3 but less than 15. Since this looks something like this, this is called a tree and since it has got a key value with elements to the left and the right less than unequal and it helps to quickly search elements, it is called a search tree. And since it has got two pointers, it is called a binary search tree, all right.

So now we will investigate how good this structure is whether there is any strength of using this structure in our implementation of insert, delete, find, print max etc etc and we will see whether this structure is really good or whether this structure is not as good as this, in what situation is it worse, better. So we have got what is called a binary search tree, for example which we worked out will look something like this say I have just may be this was 19.

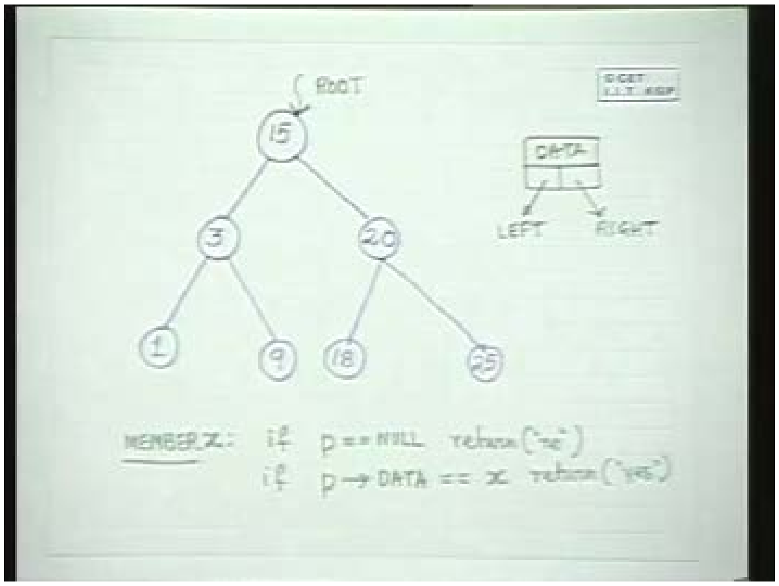
(Refer Slide Time 34:14)



All elements which are this side is less than equal to, all elements this side are greater than equal to. This is true for any node and the node structure diagrammatically is data and two link fields one is called the left and one is called the right, this points to the same node structure, this points to the same node structure. Now to find out an element, what do you do on this tree? What are the, how to find out an element? Given a tree, you start at the root; this is called the root of the tree. If you look it backwards, some people look at it like a tree but I don't know why it is called a tree I mean just because it has some branches possible. This is called the root of the tree and to work out the simplest member relation, my routine is like this. I check whether the element at the root is equal to the element I want to find out, all right.

First I check whether the pointer is null or not, if the pointer is null not available. So first thing you do is if pointer to that node is null, return no. Next is if p pointer data is equal to x, return yes. Can you tell me the rest of the two steps else if p pointer dot data is less than x then what do you do. If p pointer dot data is less than x, go to the right. That is the element which is less, which is more than this. So go to the right means what? What would you do, go to the right? You would call this on this node now, you can just recursively call it on this node, you can recursively call it on this p pointer dot right. You would move here, let us say will come to functions later on. You would move here and do the same thing. If it was greater that is p pointer dot data is greater than x then you would work out on p pointer dot left and continue and do the same thing.

(Refer Slide Time 35:52)

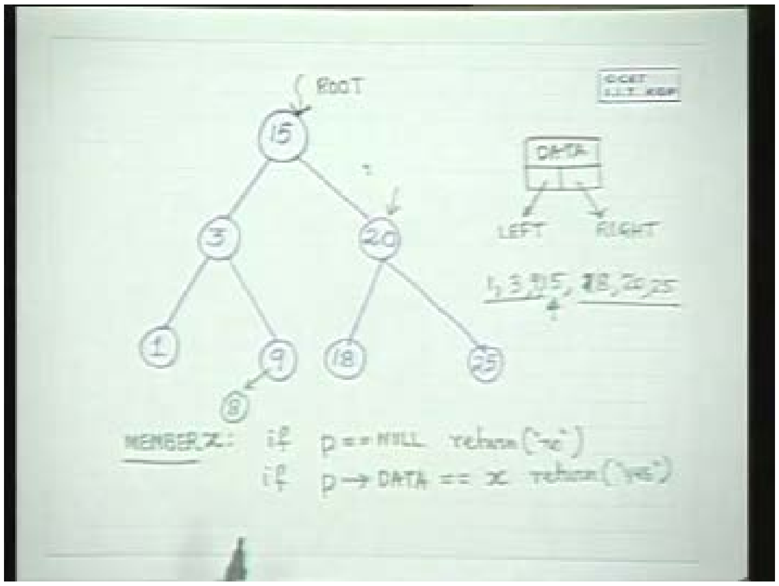


```
MEMBER x:  if  p == NULL  return("no")
           if  p→DATA == x  return("yes")
```

Now you can see one thing that here you would at least leave out some elements and in the best case you would leave out half the elements. That is half the elements this side would not be seen and only the other half you would work on, that is if you checked on this one, you would now be checking only on this side. Does this remind you of anything else? Suppose these were put on in an array 1 3 15 18 20 and 25, you would check in the middle, 1 3 sorry 9, you would check in the middle and if it is less you would look only this side otherwise you would look only this side. So it is, you are looking at it only less than side it is just like binary search.

How would you insert an element? You would do a find to see if it exists, while doing the find you would reach a point where it either exists then you would not do anything, you would not insert it because it's a set you wouldn't duplicate. And if you say suppose you have given 8 then you would come here, come here, come here and see it is not there because this pointer is null, you just simply insert it here because this is where you are supposed to find it and since it is not there, this is where you insert it. Are you getting the point? Delete is slightly more involved, we will come to delete later on.

(Refer Slide Time 39:18)


ROOT
15
3
20
1
9
18
25
8
DATA
LEFT
RIGHT
1, 3, 9, 15, 18, 20, 25
MEMBER x: if p == NULL return("no")
if p → DATA == x return("yes")

Maximum, which is the maximum element? Right, right, right, go on moving to the right till you reach the end that is automatically the maximum value. Now what is the complexity of finding out, what is the worst case complexity of say find because member is the most crucial operation? What is the worst case complexity of the member? Now here what is the in a binary search tree, in terms of the tree can you tell me what is the worst case of the member operation? n by 2, log n, tell me in terms of the tree n, tell me in terms of the tree, give me a more accurate calculation. It is the length of the longest path from the root to the node. The worst case complexity of find is the length of the longest path from root to leaf.

Now this length of the longest path will vary depending on the way data is entered. Suppose I give you 15 and then 12 and then 11 and then 10 and then 3 then it will just be like your ordered linked list and the length of the longest path will be n. So in the worst case it may be this, so all our operations insert is the length, proportional to the length of the longest path. We will see how delete will occur and it will be also be proportional to the length of the longest path. Find will be proportional to the length of the longest path etc etc etc. So, all of them will be proportional to the length of the longest path. On the other hand in the worst case it is order n. So now the next is how could we minimize the length of the longest path that would be, that is you would like to define a search tree in which you will be able to minimize the length of the longest path, so that if you could do that efficiently then we would get an algorithm which is better than ordered or unordered list. So we will see that in the next lecture.