Design and Analysis of Algorithms, Chennai Mathematical Institute

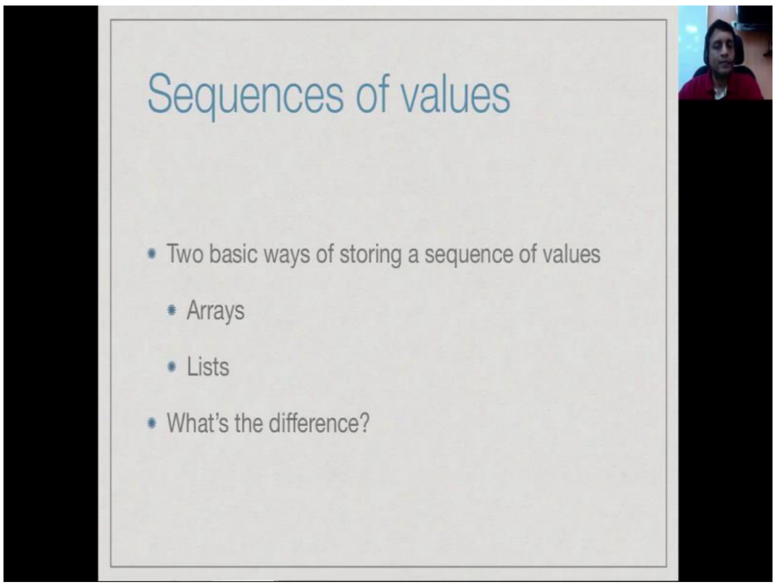
Prof. Madhavan Mukund

Department of Computer Science and Engineering,

Week – 02
Module – 01
Lecture - 09
Arrays and lists

So, in this unit we are going to look at searching and sorting for a value in a list of values.

(Refer Slide Time: 00:06)

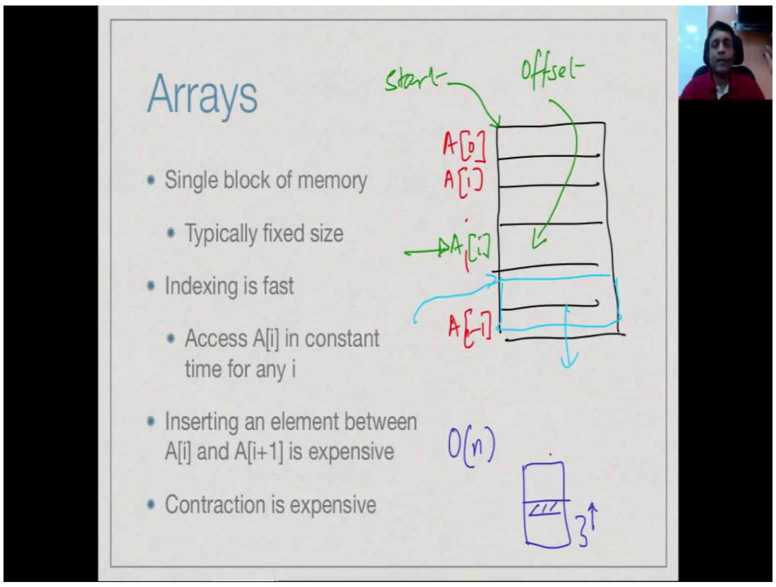


Sequences of values

- Two basic ways of storing a sequence of values
  - Arrays
  - Lists
- What's the difference?

So, let us review how list of values are actually stored in a computer. So, there are two basic ways of storing a sequence as you know arrays and lists. Now, functionally they may looks similar, but from a complexity theory point of view, from an efficiency point of view, how the data is organized makes the difference. So, let us just review some basics ideas about arrays and lists.

(Refer Slide Time: 00:30)



Arrays

- Single block of memory
  - Typically fixed size
- Indexing is fast
  - Access A[i] in constant time for any i
- Inserting an element between A[i] and A[i+1] is expensive
- Contraction is expensive

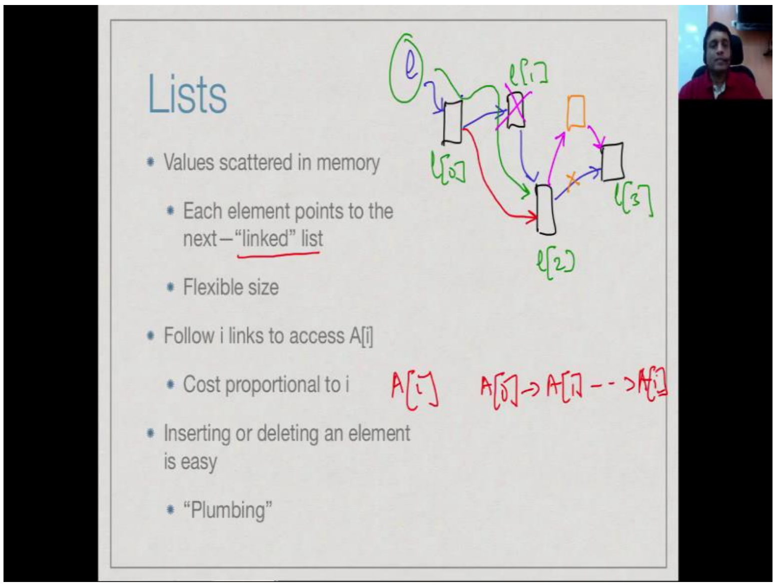
So, an array to begin with is a single block of memory. So, if you have an array, should think of it as something which has consecutive elements storing the values in the array. So, if you have an array A of size n, then A 0 is immediately followed by A 1 and so, on until A n minus 1. So, the crucial fact now is that if I know where the array starts, then if I want to get to some value A i, then I just have to multiply by i, the size of each unit of array to find out exactly where A i is.

So, this amounts to saying that we can access any value A i, any position i in the array in constant time regardless of whether i is the beginning or at the end because we just have to compute the offset. So, we have the starting position and then given i, we can compute directly in one shot just doing arithmetic with respect to the starting address, the position of A i.

Now, on the other hand, if I want to insert an element between A i and A i plus 1, this becomes a big complicated that because, supposing now want to push a value here; that means, I have to take these values and move them down,; that means, each of these value has to be shifted by 1. Therefore, the time taken to insert an element depends on the position, but in the worst case I might have to shift all the elements by 1. So, this could take time proportional to the size of the arrays. So, this could be a order n operations.

The same way contraction can require me to take an array and then if I want to remove this element, then I have to shift all of these elements up by 1. So, if I want to expand or contract an array is an expensive operation, but I can access any given element in an array in a fixed cost time which is independent of the position. And I can just treat it as like any other access, like accessing a variable x or y or anything else simple in my program.

(Refer Slide Time: 02:24)



Lists

- Values scattered in memory
  - Each element points to the next—"linked" list
  - Flexible size
- Follow i links to access A[i]
  - Cost proportional to i
- Inserting or deleting an element is easy
  - "Plumbing"

A list on the other hand is generally a flexible structure and as elements are added, they gets scattered around the memory with no fixed relationship to each other. So, typically a list will have values in different parts of the memory and the idea is that each value will points to the next. So, supposing this is the beginning of the list, then this will point to the next element, this will point to the third element, this will point to the fourth element and so, on.

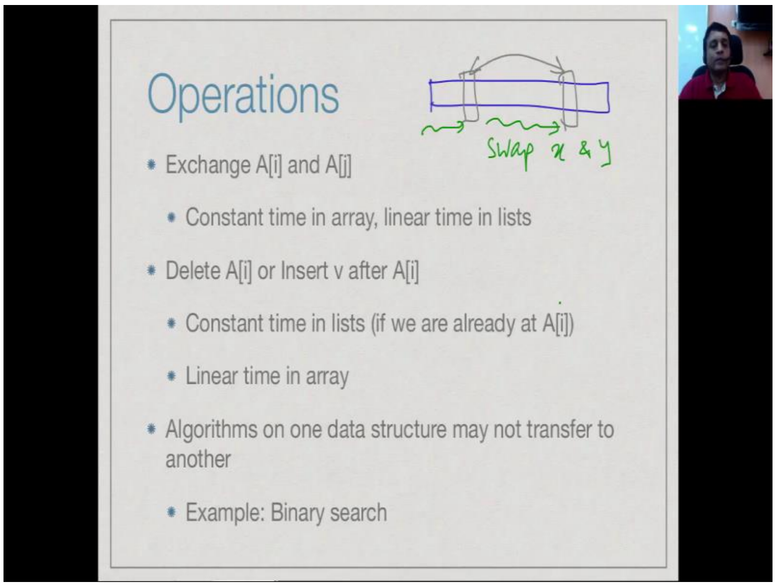
So, because of this linking structure usually in most introductory data structures courses, lists are often referred to as link list. So, link list is just a concrete way of implementing such a flexible step structure. So, now what this means is that, if I want to find where say l 2 is so, let us assume that this is l 0, l 1 and l 2. Then, I have no idea in general, where l 2 is located, but I know where l 0 located, because my list name and my program l will point to l 0.

So, I will have to follow these arrows until I reach l 2. So, in general if I want to go l A i, I have to start at A 0, then follow my link to A 1 and so, on. So, this will take me i steps,. So, therefore, the further down in a list I need to go, the longer it takes me,. So, accessing A i is proportional to i. So, in general if I have to access the i, the limit of l is it is the linear dimension. On the other hand, inserting and deleting is comparatively easy, if I know where I am, supposing I want to insert something between l 2 and l 3, then what I would do is I would first construct a new node here and then I will do what I would call plumbing.

So, I will remove this link and I would instead, insert a link from l 2 to the new node and from the new node to l 3 and I know which links to add, because this is a node I have just created. So, I know its location and l 2 points to l 3. So, I can transfer that information to the new node so, that these links can be established.

And a similar thing happens when I want to delete a node, if I want to delete this node for example, then what I have to do is, I just take this link and I bypass it and go directly. So, by just shuffling these links around, in a local sense in a constant amount of time, we can insert or delete at any point in a list. But, finding the position requires me to start at the beginning and go the end, this takes linear time.

(Refer Slide Time: 04:47)



Operations

- Exchange A[i] and A[j]
  - Constant time in array, linear time in lists
- Delete A[i] or Insert v after A[i]
  - Constant time in lists (if we are already at A[i])
  - Linear time in array
- Algorithms on one data structure may not transfer to another
  - Example: Binary search

So, this distinction has an impact on what we can do. Suppose, we want to take a sequence of values and we want to take a value at position i and position j and I want to exchange it. So, if I know the positions i and j, in an array I can easily get to A i and A j at constant time and exchange them. So, this is exactly like swapping x and y and no difference in swapping x and y, I am swapping A i and A j.

But, if it is a list, then I have to walk down to find this, then I have to walk down to find that,. So, I have find these two things and then I have to exchange. So, it will take me a linear time just to locate the positions to exchange. On the other hand, as we have already seen, if you wanted to delete an element from a middle of a list or we want to insert an element into a list, this takes constant time, provided we are already at A i.

So, if we reach A i and we find at this position you want to do something we can do that operation, insertion or deletion in constant time. But, at that position, if you want to insert a value or contract the array, then we have to shift a whole number of elements forwards or backwards and this will take linear time. So, sometimes this impacts what we can do, an algorithms are work well on one data structure will not work in the other, though they are abstractly representing the same thing, a sequence of values from one to n or 0 to n minus 1. As an example of this, we will see soon binary search. Binary search works on arrays, but does not work on list, because it requires us to repeatedly probe the array at some index i in an efficient way.