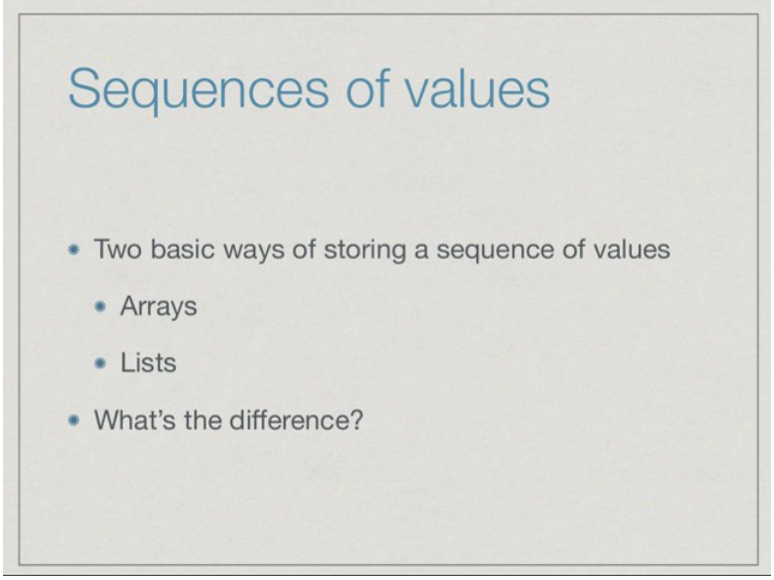
Programming, Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Chennai Mathematical Institute,Madras

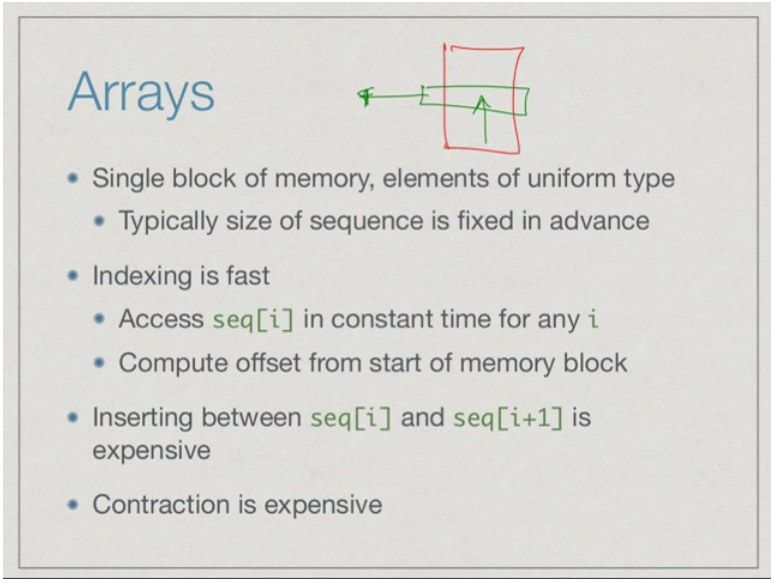
Week – 03
Lecture – 04
Arrays vs. Lists, Binary search

(Refer Slide Time: 00:02)



We have seen several situations where we want to store a Sequence of values. Now it turns out that in a program or in a programming language implementation, there are two basic ways in which we can store such a sequence. These are normally called Arrays and Lists. So, let us look at the difference between Arrays and Lists.

(Refer Slide Time: 00:22)



An array is usually a sequence which is stored as a single block in memory. So, you can imagine if you wish that your memory is arranged in certain way and then you have an array, so usually memories arranged in what are called Words. Word is one unit of what you can store or retrieve from memory, and an array will usually be one continuous block without any gaps.

And, in particular this would apply when an array has only a single type of value, so all the elements in the sequence are either integers or floats or something where the length of each element of the array is of a uniform size. We would also typically in an array no in advance how big this block is. So we might know that it has say 100 entry, so we have a sequence of size 100.

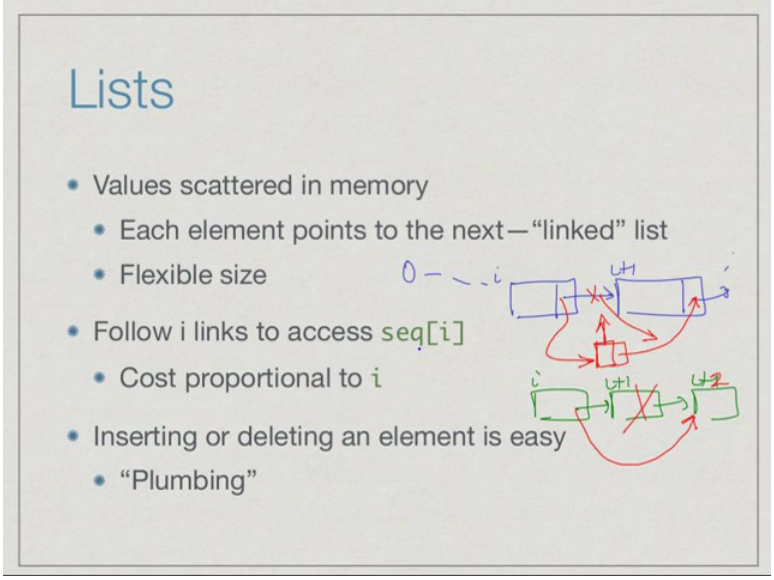
Now when this happens, what happens is that if you want to look at the jth element of a sequence or the ith element of a sequence, then what you want to think of is this block of memory starting with 1, 2, 3, up to i right and you want to get to the ith element quickly. But since everything is of a uniform size and you know where this starts, we know where the sequence starts you can just compute i times this size of one unit and quickly go and one shot to the location in the memory where the ith element is saved.

So, accessing the ith element of an array just requires arithmetic computation of the address by starting with the initial point of the array and then walking forward i units to the ith position. And this can be done in what we could call Constant time. By constant time what we mean is it does not really depend on i. It is no easier or no difficult to get the last element of an array as it is to get to the second element of an array, it is independent of i. It takes the fixed amount of time to get to sequence of y for any i.

Now, one consequence of this is inserting or contracting arrays is expensive, because now if i have an array with 0 to 99 and I want to add a new value here say at position i then first of all this array now becomes from 0 to 100 and now everything which is after i has to be shifted to accommodate space if we want to keep the same representation with the entire array is stored as a single block. So, when we have a single block of memory though it is efficient to get to any part of it quickly it is not very efficient to expand it because we have to then shift everything. The worst case for example, if this green block comes into 0th position then the entire array has to be shifted down by one position.

In the same way contraction is also expensive because we have to make a hole in some sense. If we remove this element out then we have a hole here and then we have to push everything up to block this hole, because – remember the array must have all elements contiguous that is without any gaps starting from the initial position.

(Refer Slide Time: 03:27)



The other way of storing a sequence, is to store it one element at a time and not bother about how these elements are with respect to each other in the memory. I can think of this memory as a large space and now I might have one element here, so this is my first element and then I will have a way of saying that from here the next element is somewhere else, this is what we call a Link. So very often in the implementation these are called linked list, so I may have the first element here. Now because of various reasons I might end up putting the second element here and so on.

You can imagine that if you have some say space in your cupboard and then you take out things and then you put things back but you put things back in the first place where you have an empty slot, then the sequence in which you put thinks back may not respect the sequence in which they appear finally in the shelf. So, here in the same way we do not have any physical assumption about how these elements are stored, we just have a logical link from the first element to the next element and so on.

The other part of this is that we do not have to worry about the overall length of the list because we know we started at the 0th position and we keep walking down. On the last position so say suppose the last position is in fact two then there would be some indication here saying that there is no next element, so two is the last element. A list can

have a flexible size and obviously because we are just pointing one element to another, we can also accommodate what we see in Python where each element of the list maybe of a different type and hence each value might have a different size in itself. It is not important unlike an array that all the values have exactly the same size because we want to compute how many values to skip to get to the ith element. Here, we are not skipping we are just walking down these links.

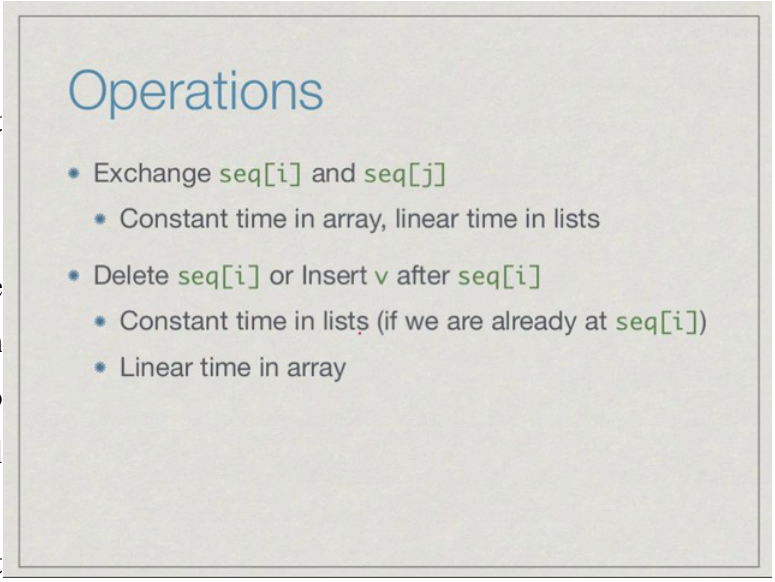
Since we have to follow these links the only way to find out where the ith element is is to start from the 0th element and then go to the first element then go to the second element and so on, because a priori we have no idea where the ith element is. So, after i steps we will reach the ith element. And if we have a larger value of i it takes longer to get there. So accessing the ith position in a sequence when the sequence is stored as a list takes time proportional to i, we cannot assume that we can reach any position in the list in constant time unlike in an array.

On the other hand it is relatively easy to either insert or delete an element in a list like this. Supposing, we have a list like this. Suppose, we start at 0th position and may come to the ith position and currently if we say that the ith position points to the i plus 1th position which point to the rest, and suppose we want to insert something here, then it is quite simple we just say that this is the new i plus 1th position. We create a new block in memory to store this value and then we will make this point here. So, it is like plumbing, we remove one pipe and we attach a pipe from the ith element to the new element and attach another pipe to the new element to what was beyond the ith element previously.

We just have to shift these three links around and this does not matter wherever we have to do it, any place in the list if we have, I have just have to make this local change in these links. And so this insertion becomes now a constant time operation if we already are at the position where we want to make the change. In the same way if we want to delete something that is also easy in fact it is even easier. So, I have say i pointing to i plus 1 pointing to i plus 2 and I want to remove this element, well then I just make this link directly point to the next one. Remember all these links are available to us we know this link we know this link, so we know where i plus second element is.

Similarly here, when we want to create a new element we get a link for it because we create it and we know what link to copy there because we already have it here. So we can copy it from the ith element to the new element. Therefore, in a list it is expensive to get to the ith element it takes time proportional to the position we are trying to get to, however, having got to a position inserting or deleting an element at that position is of constant time. Unlike in an array, where if we insert or delete at some position we have to shift a lot of values forwards or back words and that takes time.

(Refer Slide Time: 07:47)



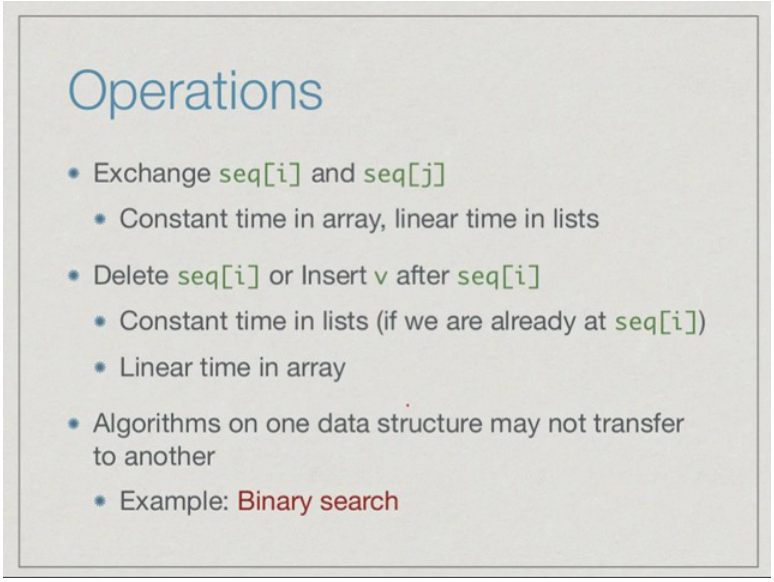
Let us look at typical Operations that we perform on sequences. So one typical operation, now if i just represent a sequence more abstractly as sequences we have been drawing it. Supposing, I want to exchange the values at i and j. This would take constant time in an array because we know that we can get the value at ith position, get the value at the jth position in constant time independent of i and j and then we exchange them it just involves coping this there and the other one back.

On the other hand in a list I have to first walk down to the ith position and then walk down to the jth position to get the two positions so I will have in a list I would have the sequence of links and then I would have another sequence of links. Then having now identified the block where the ith value is and the block where the jth values then i can of cause exchange them without actually changing the structure I just copy the values back and forth, but to find the ith and jth values it takes time proportional to i and j, so it takes

linear time.

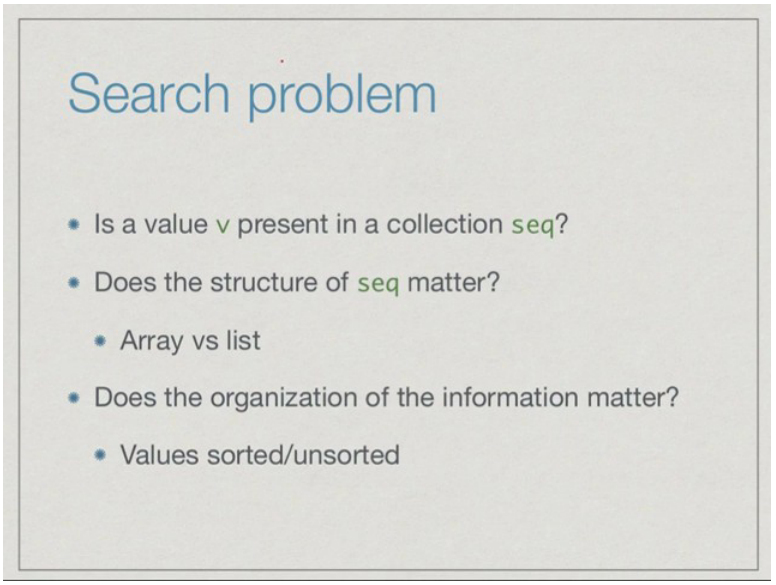
On the other hand as we have already seen, if you want to delete the value at position i or insert the value after position i this we can do efficiently in a list because we just have to shift some links around, whereas in an array we have to do some shifting of a large bunch of values before or after the thing and that requires us to take time proportional to i.

(Refer Slide Time: 09:12)



The consequence of these differences between the two representations of a sequence as an array and a list is that we have to be careful to think about how algorithms that we want to design for sequences apply depending on how the sequence is actually represented. An algorithm which works efficiently for a list may or may not work efficiently for an array and vice versa. To illustrate this, let us look at something which you are probably familiar with at least informally called Binary search.

(Refer Slide Time: 09:42)



The problem we are interested in is to find out whether a value v is present in a collection or we can even call it a sequence to be we more precise in a sequence which we call seq. So, we have a sequence of values we want to check whether a given value is there or not. For instance, we might be looking at the list of roll numbers of people who have been selected for a program you want to check whether our roll number is there or not.

There are two questions that we want to ask; one is is it important whether the sequence is maintained as an array or as a list and is it also important given that it is maintained as an array or a list whether or not there is some additional information we know for example, it is useful for array to be sorted in ascending order that is all the elements go in strictly one sequence from beginning to end, lowest to highest, or highest to lowest, or does it matter, does it not matter at all.