

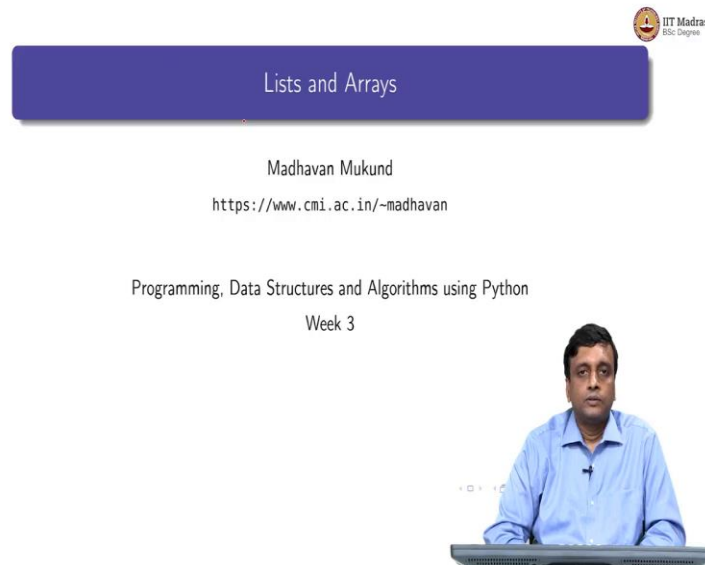


IIT Madras

ONLINE DEGREE

Programming, Data Structures and Algorithms using Python
Professor Madhavan Mukund
Difference between Lists and Arrays (Theory)

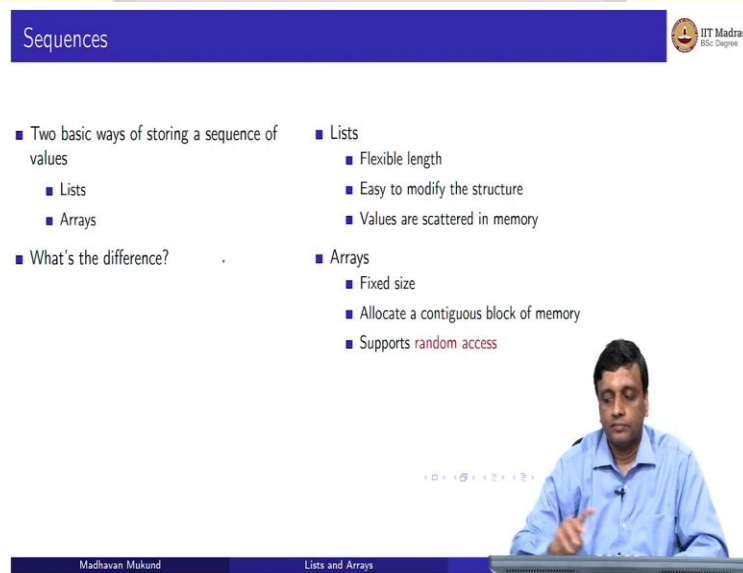
(Refer Slide Time: 00:09)



The slide features a purple header with the title "Lists and Arrays". Below the header, the text "Madhavan Mukund" and the URL "https://www.cmi.ac.in/~madhavan" are displayed. The course title "Programming, Data Structures and Algorithms using Python" and "Week 3" are also present. A small IIT Madras logo is in the top right corner. A video inset shows Professor Madhavan Mukund sitting at a desk with a laptop.

So, we have been studying sorting and searching and we have somewhat interchangeably been using words like lists and arrays. So, let us try and understand formally what is difference between a list and an array in the context of programming.

(Refer Slide Time: 00:24)



The slide has a purple header with the title "Sequences". It compares Lists and Arrays. A video inset shows Professor Madhavan Mukund sitting at a desk with a laptop.

Lists	Arrays
Flexible length	Fixed size
Easy to modify the structure	Allocate a contiguous block of memory
Values are scattered in memory	Supports random access

Two basic ways of storing a sequence of values:
■ Lists
■ Arrays

What's the difference?

So, we are interested in maintaining sequences of values. So, we have a first value or second value, and so on. And we would like to talk about the i th value and the j th value and maybe look it up or exchange it and so on. So, fundamentally, these two lists and arrays are two different ways of maintaining such a sequence. So, what is the difference?

So, list in the conventional understanding of the word is a flexible length sequence. So, it is a sequence that can grow and shrink. So, the idea with a list is that you can modify its structure. And this we see in a built in Python list, for example, because you can take a slice, for example, and replace it by a longer slice or a shorter slice. So, you can expand and contract a Python list, you can append something to the beginning, you can delete a value in the middle. So, the length can vary over time.

But because of this, it is hard to put this list in a fixed place in memory, because the place that you require for the list may grow and shrink over time. So, typically, a list is scattered in memory. So, we will come back to what this means. But it is, think of a list as being dispersed. So, you cannot point to a segment of memory and say the entire list is sitting here. It might be sitting in a lot of places.

An array, on the other hand, is typically something which is designed not to grow or shrink. So, in the conventional understanding of an array, it is a fixed size sequence. I need a sequence of 100 elements. Now, this makes a lot of sense when we are processing graphs, for example. In a graph, typically, we need something like the list of all the vertices or we need the adjacency matrix. And for these things, if we know the number of vertices, we know the length of the list and is never going to shrink or grow. We need to always have a list of n vertices. We always need an adjacency matrix of n by n .

So, if we have a fixed size, then we know in advance how much space we will need in memory. So, we can go ahead and block that space. So, this makes an array something which sits in a fixed space in memory and this will help us as we will see access the elements in an array faster than accessing them in a list.

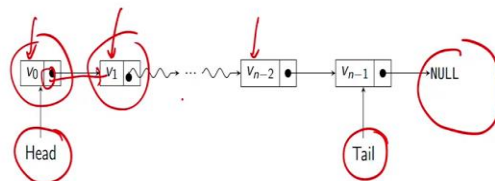
(Refer Slide Time: 02:30)

Lists



- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence

■ "Linked" list



Navigation icons: back, forward, search, etc.



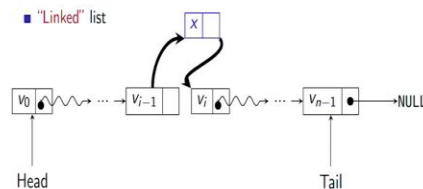
Madhavan Mukund

Lists and Arrays

Lists



- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
- "Linked" list
- Easy to modify
 - Inserting and deletion is easy via local "plumbing"
 - Flexible size



Navigation icons: back, forward, search, etc.



Madhavan Mukund

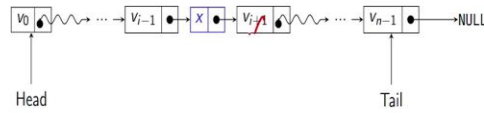
Lists and Arrays

सिद्धिर्भवति कर्मजा

Lists



- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - "Linked" list
- Easy to modify
 - Inserting and deletion is easy via local "plumbing"
- Flexible size



Navigation icons: back, forward, search, etc.

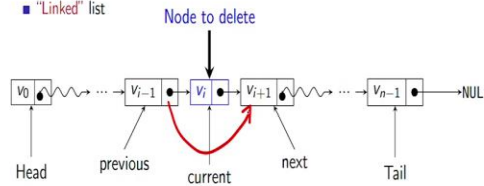
Madhavan Mukund

Lists and Arrays

Lists



- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - "Linked" list
- Easy to modify
 - Inserting and deletion is easy via local "plumbing"
- Flexible size



Navigation icons: back, forward, search, etc.

Madhavan Mukund

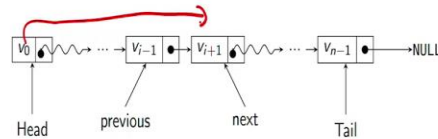
Lists and Arrays

सिद्धिर्भवति कर्मजा

Lists



- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - "Linked" list
- Easy to modify
 - Inserting and deletion is easy via local "plumbing"
- Flexible size



Madhavan Mukund

Lists and Arrays

Lists



- Typically a sequence of nodes
- Each node contains a value and points to the next node in the sequence
 - "Linked" list
- Easy to modify
 - Inserting and deletion is easy via local "plumbing"
- Flexible size
- Need to follow links to access $A[i]$
- Takes time $O(i)$



Madhavan Mukund

Lists and Arrays

So, let us look at this in a little more detail. So, a list in this flexible sense, is typically a sequence of node. So, each node has a value, and it will point to the next node in the sequence. So, you do not have to tell in advance where these nodes are, because you can just ask each node to tell us where the next node is.

So, it is like following directions to go someplace. You do not know the way all the way, but you know that everybody in the neighborhood knows the way. So, you as the first person saying I am going in that way, which road should I take. You go to the next guy and he would say which road should I take and you keep following these links.

So, the picture that we should have in mind is something like this. So, we have these nodes. So, each node has a value and each node has this information which tells us where the next node is. So, what we know typically is where the list begins. We could, if we want keep information about where the last node in the list is. This might be helpful, for instance, if we want to extend it by adding one more value there without going through the whole list. But in principle what we really need is this structure which is often called a linked list.

So, the linking basically refers to the fact that I have this. So, think of it like a train. If you have seen a train, a train consists of these independent wagons or bogies, and they are all connected to each other. So, this list is like a train. We can detach it, we can reattach it. And in principle, of course, in a physical train, there is some link is made out of some hard metal, so there cannot be so much distance. But if this link was made out of elastic and infinitely stretchable, you could have a train in which the different parts of it are all over the place.

So, the main advantage of having this linked structure is that it is very easy to modify a list. So, supposing I want to insert a new value x between the element i minus 1, V_i minus 1 and V_i . So, what I need to do is basically restructure this list. But it is, the structure is only logical, it is not a physical structure. Each of these notes are sitting somewhere in memory.

So, what I need to do is make this V_i minus 1 point to X and V_i X point to V_i . So, what I need to do is basically change from this link which says that V_i connects to V_i , V_i minus 1 connects to V_i , I need to say that V_i minus 1 connects to X and X connects to V_i . So, there is some local, what you can think of is plumbing. So, if you have some block in a pipe, then one way is to of course remove that piece of pipe and replace it or you can just bypass it by moving a pipe around it. So, you can do some local operation and not affect the whole plumbing network.

So, you are just taking some local neighborhood of the nodes that you want to update and you are making some changes in how the information flows or how the sequencing happens. So, this is how you insert. Similarly, so now at the end of this, I mean, I can think of it as I have put an X there.

Of course, none of these is actually in any physical sequence. These are all sitting somewhere abstractly in memory. So, this is only a logical sequence that I go from 0 to 1, 1 to 2, and so on.

And now after this insert, I go from i minus 1 to this new value X , and then I go from X to i . So, this should be i probably.

So, now, here is a symmetric situation. I want to delete a node in the middle of a list. So, here, what do I do? Well, it is simpler in a way. I need to just take this and bypass it. So, this is like, again, plumbing. You want to make sure that some piece of pipe which is blocked is bypassed, so you just go passed it. So, you do this. And then logically speaking, this V_i disappears. So, now I directly go from the previous node to the next node. That current node which I want to delete is gone.

Now, it is a separate matter in terms of the programming language, how that lost node is returned back to memory so the memory does not get used up with all this junk. That is not something we have to worry. Python does it automatically. So, Python is able to keep track of all these things, for instance, which get lost and take care of it so that they do not become permanently a drain on the memory that you are using.

So, in this way, if you have this linked structure, you can see that by doing some local operations, you are able to manipulate the list quite easily. But the problem with this is that you need to actually get to this position. So, you need to follow these links to access the i th thing. So, the, in order to put this here, I have to first get here. And the only way I can get here is to actually walk down this list from the head, because there is no way I know in advance where the i th element sits in memory. So, I need to spend time proportional to i , the position or the index I am looking for in order to reach the i th element.


So, this is the flip side of this flexibility. The positive side is that I can make these local changes easily where I am. But to reach the place where I need to make the change, I need to walk down always from the start, because there is no direct way to get there.

(Refer Slide Time: 07:25)

Arrays

- Fixed size, declared in advance
- Allocate a contiguous block of memory
 - n times the storage for a single value
- "Random" access
 - Compute offset to $A[i]$ from $A[0]$
 - Accessing $A[i]$ takes constant time, independent of i

Index	Value
$A[0]$	v_0
$A[1]$	v_1
\vdots	\vdots
$A[i]$	v_i
\vdots	\vdots
$A[n-1]$	v_{n-1}




Madhavan Mukund Lists and Arrays

Arrays

- Fixed size, declared in advance
- Allocate a contiguous block of memory
 - n times the storage for a single value
- "Random" access
 - Compute offset to $A[i]$ from $A[0]$
 - Accessing $A[i]$ takes constant time, independent of i
- Inserting and deleting elements is expensive
 - Expanding and contracting requires moving $O(n)$ elements in the worst case

Index	Value
$A[0]$	v_0
$A[1]$	v_1
\vdots	\vdots
$A[i]$	v_i
\vdots	\vdots
$A[n-1]$	v_{n-1}



Madhavan Mukund Lists and Arrays

The other option is to have an array. So, as we said an array will take a parameter n to start with and say that this is going to be always of size n . And usually, it is also assumed that in an array all the elements of the array are of the same type. So, in most programming languages this is also true of lists. A list is also a list of a fixed type. Now, Python has a very flexible thing which allows you to have lists in which some elements are integers, some are dictionary, some are this, some are that. So, in principle, a Python list is actually a list of what are called objects.

But in most programming languages is a fixed type. And a fixed type has a fixed size. If I want to store an integer, it will take so much size, if I want to store a float, it will take so much size.

So, if I know how much size a fixed value takes and if I need n of them, then I obviously need n times that space.

So, I can allocate when I start, if the programming, programmer asks for an array of size n of int, then I know how much space n ints take and I can say, okay, here is a space with n ints and I put it all in one position, in one place. So, from the starting point, I know that the first int is A_0 , the second int is A_1 and so on and this gives us what is called random access.

So, if I want to get to this entry and I know where the array starts, then I just have to calculate it from the way the array starts, I have to skip past i times the space for an integer and I know that that position is going to have this. It is like walking on tiles. If I know I have to walk 20 steps, then I can close my eyes and walk 20 steps and I know that I will be there. Except that I do not have to, in this case, physically walk. I do not have to go through 1, 2, 3, 4. It is not like a list. I do not have to say, oh, the next one is here and the next one is here, I do not.

I can directly say, okay, I want to bypass to the i th position. So, if I am here, this is where I need to be. So, I can actually, it just becomes an arithmetic calculation to calculate where it is. So, this is called random access.

So, technically, what it means is that whether I am accessing the beginning of the list, the middle of the list or the end of the list, it takes the same amount of time. It takes time which is independent. So, whether the list has 100 elements or 1 million elements, it is going to take the same amount of time to access it if it is an array, whereas in the previous example, in the flexible list, it took time proportional to where I want it to go.

So, if I wanted to work in the early part of the list, it was fast. If I wanted to work in the later part of the list, it would be slow, because I have to walk all the way there. Now, of course, this is not a situation where one is clearly better than the other. So, what we have lost in an array is the flexibility.

In a list we could insert and delete things, so we could grow and shrink the list without any problem, provided we knew what to do where we were. But in an array, if we have to grow this list, supposing I want to insert a value here, then everything below it has to shift by one position to make space. So, you know exactly what happens.

When you try to put a book into a crowded bookshelf, you have to push all the books. So, that is a lot of work. And in the worst case, if you are putting it right at the beginning of the bookshelf, then you have to move everything to the right. So, it could take order n work to make space to insert an element. And likewise, when you pull out a book from the bookshelf and you leave a hole in the shelf, you need to compress all the books. So, you need to push everything from the right to the left. So, again, it could take order n in time.

So, for our earlier thing that plumbing was only a two or three step operation. In the neighborhood of the element, I wanted to add or delete, I just had to move a few links. So, that was a constant overhead, only problem was to find out where I was. Here, even if I know where I am supposed to do it, it is going to take me in the worst-case time proportional to the number of elements which are to my right and that could be in the worst-case order n .

(Refer Slide Time: 11:04)

Operations

- Exchange $A[i]$ and $A[j]$
 - Constant time for arrays
 - $O(n)$ for lists
- Delete $A[i]$, insert v after $A[i]$
 - Constant time for lists if we are already at $A[i]$
 - $O(n)$ for arrays
- Need to keep implementation in mind when analyzing data structures
 - For instance, can we use binary search to insert in a sorted sequence?
 - Either search is slow, or insertion is slow, still $O(n)$

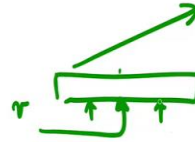
Handwritten notes: $(x,y) = (y,x)$, $(A[i], A[j]) = (A[j], A[i])$

Madhavan Mukund Lists and Arrays

Operations



- Exchange $A[i]$ and $A[j]$
 - Constant time for arrays
 - $O(n)$ for lists
- Delete $A[i]$, insert v after $A[i]$
 - Constant time for lists if we are already at $A[i]$
 - $O(n)$ for arrays
- Need to keep implementation in mind when analyzing data structures
 - For instance, can we use binary search to insert in a sorted sequence?
 - Either search is slow, or insertion is slow, still $O(n)$



$$(x, y) = (y, x)$$

$$(A[i], A[j])$$

$$= (A[j], A[i])$$

Navigation icons: back, forward, search, etc.



Madhavan Mukund

Lists and Arrays

So, to summarize, if I look at these two representations of sequences, the typical operations that we do when we are doing things like searching, sorting or manipulating arrays in general, so the one operation is to exchange two values. We do this all the time in sorting. We say okay take the say in selection sort, we find the maximum or the minimum, move it to the beginning or we might take two adjacent values which are out of order and swap them. In quicksort, we take the pivot value and we move it somewhere.

Now, in this we have to look up in general a value which is at some position i and some other position j which may be very far away. For an array, this is not a problem. If I know i and j , I can calculate instantly where A_i is, where A_j is, so it is exactly like swapping two values of a normal type. If I want to exchange the values of x and y or a and b is exactly the same cost as compared to swapping capital A_i capital A_j . But in a list, it is not the case. I need to actually walk down to A_i and I need to walk down to A_j . So, it will take me time proportional to the length of the list in the worst case in order to do the swap.

On the other hand, if I have, it does not happen in sorting and searching, because we are not manipulating the array. But if we have a need to actually make this sequence shrink and grow, then if I know where I am, if I am at a position and at this point, I want to make a change, then, for the list, the flexible list this is a constant time. I just have to do some rearrangement of these arrows which point from one node to the next. Whereas for an array in the worst case, if I need to grow and shrink, as we said, you have to shift a lot of value so it could be order n .

So, the reason to emphasize this is because the analysis that we do for our algorithms actually depends on what representation we are using, because we have been a little bit casual about this when we talked about searching and sorting, because we have always sort of assumed that we are working with arrays.

When we said A_i all these basic operations, when we said, for example, swapping A_i and A_j is a basic operation, it was a basic operation if I do something like x comma y is equal to y comma x , this is not a problem. But if I am saying A_i, A_j is equal to A_j, A_i . Now, the question is to whether this is the basic operation or not really depends on whether we are dealing with an array or a list. So, in an array, this is a basic operation. In a list, this need not be. If I do not know where I am, I have a problem.

So, as a concrete instance of this, let us look at this question of inserting in a sorted list. We use this as a part of our insertion sort. So, I have a sorted list and I want to take a value V and I want it. So, supposing this is an ascending order. So, then what we said is we will keep walking one by one and we will find a position and at that position, so till that position, for instance, we could keep swapping or whatever, and then we will insert it there. We typically did it from the right-hand side, it does not matter. But this is how we did it.

Now, you might ask, okay, why did we do all this, because it is a sorted sequence. So, why not just use what we already have talked about. So, we look here to say, should I insert it there. If it is the right place to insert, it is fine. Otherwise, I need to insert it here or here. So, I can use binary search to find out where to insert. And then having found it, I need to insert. Now, there are two separate operations here; one is binary search and one is inserting by making space.

If I assume it is an array, then binary search is good, because I can always go to the midpoint, I mean, the 0 to midpoint, that quarter point or the three quarter point in constant time. So, binary search works well in an array, does not work so well in a list, because each time I have to calculate the midpoint in the list I have to actually walk halfway down the list. So, this part of it is good. Binary search works with arrays, but then I have to insert.

Now, if I were in a list and I was told to insert here, it is very cheap. But if when a binary search finds it in an array fast and tells me to insert then it is expensive. So, I kind of gain on one and

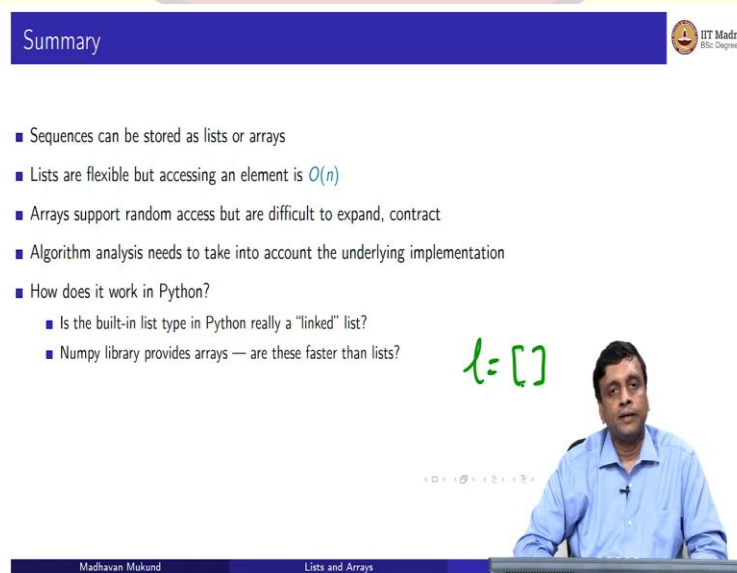
lose on the other. So, if I have an array, I will gain on the binary search and I will lose on the insert step.

And if I have a list representation, I will lose on the insert position finding step. Where to insert, I will take time to find it. But having founded the insertion will be just that plumbing. So, in some cases, the trade off is such that it does not matter. That is why we have been a little casual about insertion sort just saying that the insert is order n .

But it is not order n because of one or the other is because whichever you choose one of these two operations will push it to order n . So, the operation that pushes the order n is different. If it is an array, it is because of the insertion operation. And if it is a list, it is because of the finding the position operation. So, this is something to keep in mind. Whenever you are dealing with sequences, try to keep in mind whether you are dealing with lists or the arrays when you are doing the analysis especially.

And also, of course, when you need it for a particular requirement, just think about whether the list of sequence you are using is going to be frequently changing, is it dynamic, is it going to be growing and shrinking or is it something which is fixed once and for all. So, all these factors come into play when you choose one of these two as a representation for the sequence that you need in your program.

(Refer Slide Time: 16:14)



Summary

- Sequences can be stored as lists or arrays
- Lists are flexible but accessing an element is $O(n)$
- Arrays support random access but are difficult to expand, contract
- Algorithm analysis needs to take into account the underlying implementation
- How does it work in Python?
 - Is the built-in list type in Python really a "linked" list?
 - Numpy library provides arrays — are these faster than lists?

$l = []$

Madhavan Mukund Lists and Arrays

So, to summarize, sequences can be stored as lists or arrays. Lists are flexible, but accessing an element takes linear time. And arrays support random access, but expansion and contraction take linear time. So, when we do an analysis of an algorithm, we have to take this into account. Now, one of the things that we are doing in this course is we are using Python. So, the question is we have this data type called a list in Python.

So, is this data type called a list? When we write this, is this list a linked list in the sense that we described here with these nodes which are connected together or is it like an array and what do we do for arrays in Python? So, in Python, there is this library which you have possibly come across called numpy, which allows us to actually define these arrays with a fixed size. But now given what we know about Python lists, are these going to be faster than Python lists or slower than Python lists? So, these are some of the things that we will look at in a later lecture.

