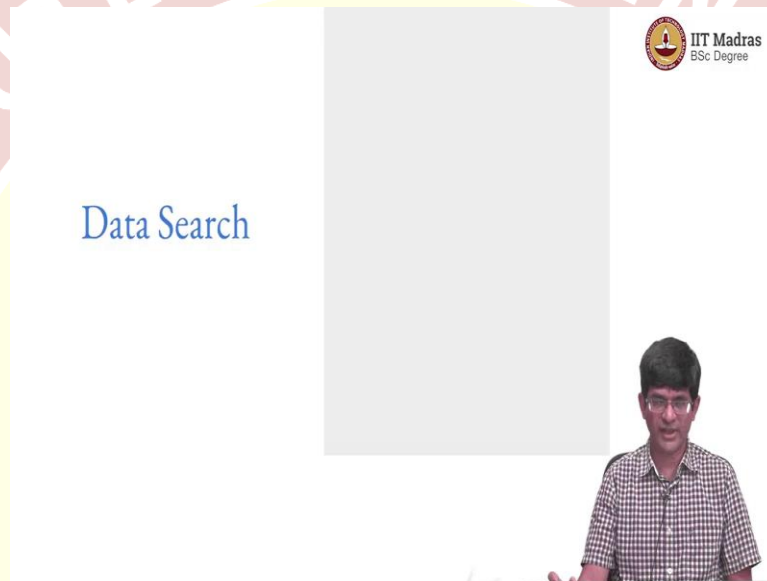


IIT Madras
ONLINE DEGREE

Modern Application Development - I
Professor Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Data Search

Hello, everyone, and welcome to this course on Modern Application Development.

(Refer Slide Time: 00:16)



So, before getting to the different types of databases, it is worth spending a little bit of time understanding how data search works.

(Refer Slide Time: 00:23)



$O()$ notation

- Used in study of algorithmic complexity: beyond scope of this course
- Rough approximation: "order of magnitude", "approximately" etc.
- Main concepts here:
 - $O(1)$ - constant time independent of input size - excellent!
 - $O(\log N)$ - logarithmic in input size - grows slowly with input - very good
 - $O(N)$ - linear in input size - often the baseline - would like to do better
 - $O(N^k)$ - polynomial (quadratic, cubic etc.) - not good as input size grows
 - $O(k^N)$ - exponential - VERY bad: won't work even for reasonably small inputs



And one of the things that I will be using over here is something called the big O notation($O()$). So, you might have come across this kind of notation, over here O and followed by parenthesis. And in computer science, in theoretical computer science, this is sometimes referred to as the big O notation, big $O(O())$ because there are also corresponding small o and a couple of other different variants of this. The big $O(O())$ is what is most commonly used. And it roughly means order of magnitude. So, what we are trying to convey when we use big O notation($O()$) is the order of complexity of the computation that you are trying to perform.

Now, without going into any details on big O notation($O()$) or how it is used in practice, you will need to do a proper course on algorithms to understand that. And hopefully, most of you have either already done it or we will be doing it at some point. Over here I am only going to touch upon some of the very basic concepts that we need for understanding some things over here. And the things to keep in mind are one of them is $O(1)$ essentially means constant time independent of the input size.

So, in other words, if I have, let us say, records of 1 million users stored somewhere in a database, but if I make a query about something, I do not know what, but if I make some kind of a query and I am able to get back a response in $O(1)$ time or constant time, that is excellent. What it means is that the time required to respond did not depend on the number of users or the number of entries in the database that is sort of the best-case scenario. You cannot really hope for

anything better than that. But it is also unrealistic, because anything which returns an answer independent of the number of entries that are there in the database cannot really be doing very useful work, because it is clearly not even looking at the data in some way.

Now, the next best, in some ways, is what is called $O(\log N)$. And when we say $O(\log N)$, what we are saying is that, if I have n records sitting inside my database, I am able to, let us say, retrieve one of them in $\log n$ number of steps or at least some constant times $\log n$ number of steps. Which means that as the database grows in size, it grows up by a factor of, let us say, 10 times.

The time taken to answer my query only goes up by a small increment. It does not go up 10 times in particular. So, if the data, the number of data entries goes up by a factor of 10, the time does not go up by a factor of 10. It goes up by something much less. This is also very good. I mean, this is pretty much the best that you can hope for in practical scenarios.

$O(N)$ is sort of the obvious thing. If I have a million entries, then how do I find one entry over there? I search through each one until I hit the entry that I am looking for. So, that is sort of the simple solution in most cases. In many instances, $O(N)$ is probably good enough, but not for things like database search.

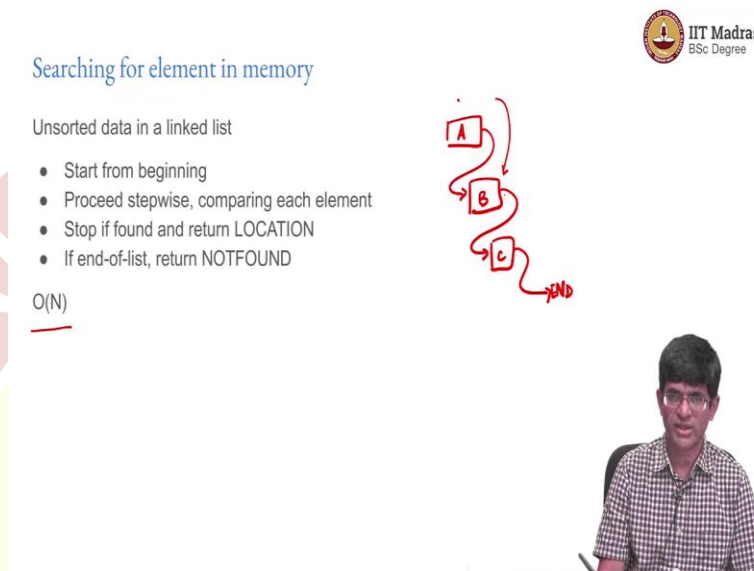
After that we have the big Os ($O()$) that are worse than $O(N)$. You can have $O(N^k)$, where if k is 2, you would call it $O(N^2)$, which is quadratic. If k is 3, you would have $O(N^3)$, cubic and so on. These are not particularly good. It means that the time required for performing an operation grows rapidly compared to the increase in size of the data itself. But even then is not as bad as $O(k^N)$, which is what we call exponential.

And exponential is very bad, meaning that even for relatively small values of N , it is not going to work very well. Imagine that k was equal to 2, what it means is that for every one unit increase in N that is for every record that you have added into the whatever your data store is, the time required for performing the operation is doubling. Just add one more entry and now you have to do double the amount of work. So, clearly, this is very bad.

There are problems where the algorithms, best algorithms known are even worse than exponential you can basically go to factorial and other functions. But we do not even consider

those. I mean, there is no point in really looking at those kinds of algorithms. So, clearly, we would like to see whether we can get to things which are $O(1)$ or $O(\log N)$ at worse $O(N)$.

(Refer Slide Time: 05:09)



The slide is titled "Searching for element in memory" and features the IIT Madras BSc Degree logo in the top right corner. It contains a list of steps for searching in an unsorted linked list and a diagram of such a list.

Searching for element in memory

Unsorted data in a linked list

- Start from beginning
- Proceed stepwise, comparing each element
- Stop if found and return LOCATION
- If end-of-list, return NOTFOUND

$O(N)$

The diagram illustrates a linked list with three nodes labeled A, B, and C. Node A points to node B, and node B points to node C. Node C points to a red "END" label, indicating the end of the list.

So, with that in mind, let us look at how we would actually search for an element in memory. And let us assume that somehow the data which I need has been stored in memory using some kind of a linked list. And what I mean by a linked list is there is some data out here, A, and that has a pointer to the next element B, that in turn points to the next element C, and so on until I hit the end of the link, end of the chain. So, now how do I search for B over here? I start from A, move on down the line, go to B, I hit the correct entry. I come back. If I want to search for C, I would need to go through A, then to B, then to C and stop.

Clearly, this algorithm is order of N in terms of the runtime, because in the worst case, I would need to go through all N elements in order to get to the final result. In the average, if I do not know which one I am searching for to start with, I can sort of expect that it is probably going to be somewhere in the middle and the time taken is going to be $N/2$, which is $O(N)$. Remember that $O(N)$ is only order of magnitude, so any such by 2 and such constants are ignored. It is still proportional to N . That is all that we care about.

So, in terms of database searching, not particularly good. Why, because typically you would have to search many times in a database. And as the size of the database grows, if each of those

searches is going to take time proportional to the number of elements, you might run into serious problems after some time.

(Refer Slide Time: 06:51)

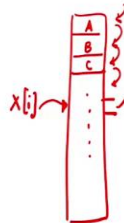
Searching for element in memory



Unsorted data in array

- Start from beginning
- Proceed stepwise, comparing each element
- Stop if found and return LOCATION
- If end-of-list, return NOTFOUND

$O(N)$



Now, what if this data was sitting in an array instead? So, it is an array in memory. What is the main difference? I know that the first element is stored here, the second element is stored here, the third element is stored here, and so on, which means that if I need to access, let us say, $X[i]$ I can directly jump to that location.

The problem is, I do not know what i is. I do not know which location to jump to. And so I still have to start from here, then go to the next element, then go to the next element until I finally hit the thing that I was searching for, which means that, because the data in the array was not sorted, I had to start from the beginning and go step by step, and therefore the time remains $O(N)$.

(Refer Slide Time: 07:42)

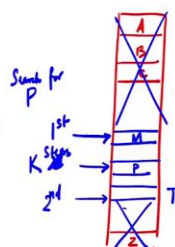


Searching for element in memory

Sorted data in array

- Start from beginning
- Proceed stepwise, comparing each element
- Stop if found and return LOCATION
- If end-of-list, return NOTFOUND

$O(N)$ but...



$$N \rightarrow \frac{N}{2} \rightarrow \frac{N}{4} \rightarrow \dots \rightarrow 1$$

$K = \lceil \log_2 N \rceil$

$A < B < C \dots < Z$
Comparison.



On the other hand, what if I had data in an array that I have guaranteed somehow is actually sorted? What I mean by sorted is basically that I guarantee that $A < B < C \dots < Z$, that the entries sitting inside this array satisfy some kind of a comparison property. And what do I do with this comparison property? I can basically say that, now I will start by looking over here. So, this is the first comparison. And let us say that I am searching for D or let us say I am searching for the letter P.

So, what happens over here is the first one that I hit this midway through, which let us say is M, I know that because $M < P$, therefore, it cannot be in the first half. I can basically rule out half the elements from the search. So, now I again go and look at the second half. So, basically, I am going to look at element from 14 to 26 if I assume that we have, then basically I am going to look at element number 20, which would probably be element T. And then after iterating some few times, I will finally end up at P, not N but let us say some K steps.

Now, what did we actually do over here? Since I had this comparison, the less than property that I could check, I could straightaway eliminate sort of large chunks of this memory. And basically say, okay, after the first step, this first half is eliminated, after the second step, the second quarter is eliminated, which means that at each stage I am going N to $N/2$ to $N/4$, at what point does it become equal to 1.

And if you think about it, basically what this is saying is that this K is basically going to be \log to base 2 of N , the ceiling of that. I mean, it has to be an integer number. So, the smallest integer that is greater than this $\log N$ to base 2. That is all that you really need in order to reduce the number of elements in your array to 1.

(Refer Slide Time: 10:23)

Searching for element in memory

IIT Madras
BSc Degree

Sorted data in array

- Look at middle element in array:
 - greater than target - search in lower half
 - lesser than target - search in upper half
- Switch focus to new array: half the size of original
 - Repeat

$O(\log N)$

Binary Search

So, what we have over here is, we can actually bring this entire thing down from $O(N)$ search to something which basically finishes in $O(\log N)$ and this is something called binary search and is very commonly used in various kinds of search optimizations. So, what we can see is that, if you are somehow able to store data in some form that is sorted and has some kind of comparison functions and easy ways to access locations, you can actually do searching within $O(\log N)$ steps, something to keep in mind when we start looking at actual databases.

(Refer Slide Time: 10:58)



Problems with arrays

- Size must be fixed ahead of time
- Adding new entries requires resizing - can try oversize, but eventually ...
- Maintaining sorted order $O(N)$:
 - find location to insert
 - move all further elements by 1 to create a gap
 - insert
- Deleting
 - find location, delete
 - move all entries down by 1 step



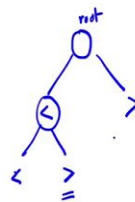
Now, arrays by themselves have certain problems. I mean, the biggest problem with it is you need to fix the size of the array ahead of time. Adding new entries is a problem. I need to resize the whole array. Deleting an entry is also a problem, because then I need to push everything back up. I need to keep the size of the array fixed. Otherwise, there is no point. I lose all the benefits of this sorting that I was talking about.

(Refer Slide Time: 11:24)



Alternatives

- Binary search tree
 - Maintaining sorted order is easier: growth of tree



Alternatives

- Binary search tree
 - Maintaining sorted order is easier: growth of tree
- Self-Balancing
 - BST can easily tilt to one side and grow downwards
 - Red-black, AVL, B-tree... more complex, but still reasonable
- Hash tables
 - Compute an index for an element: $O(1)$
 - Hope the index for each element is unique!
 - Difficult but doable in many cases



So, because of that, there are alternatives that have been proposed. And once again, you would have probably come across all this in the data structures and algorithms course. Binary search tree is a good way of solving this problem. It is a nice efficient way of maintaining a sorted order essentially, we have something called a root. And it has the property that everything over here is less than the root, everything over here is greater than the root. And similarly, I could have like multiple branches over here.

Once again, this would be less than, this would be greater than, but only greater than whatever element is, its immediate parent, not greater than the root itself, it cannot be. Just because it is on the left-hand side of the root guarantees that this element is less than the root. But because it is on the right-hand side of its immediate parent, it means it is greater than that parent. So, in this way, we build up some kind of a tree structure. And it turns out that this binary search that we talked about works brilliantly on this as well.

Now, there is a problem with binary trees. Once again, I am not getting into details, because it is out of scope of this course. The problem is so called balancing. Trees very quickly get unbalanced. Problem can be solved. There are so called red black trees, AVL trees and various other kinds of things. In particular, there is a whole set of structures called B trees, which not only allow you to build efficient binary trees, but are also friendly to storing data on storage mechanisms, such as disk.

All the other data structures that we talked about generally, assume that you have RAM, random access memory, and that that is where you are really storing the data. B trees, on the other hand, actually explicitly take into account the fact that you might be storing data on some other kind of storage mechanism. And what is the important part.

The disk means that you will only be able to retrieve or write data in chunks of some size, at least, that is when it becomes efficient. So, B trees are sort of built around that whole idea that they are supposed to be disk friendly, while at the same time having many of the benefits of efficient data storage and retrieval.

There is also something else called a hash table. And the hash table, once again, you will probably come across this in the data structures course. But the bottom line is that, if you have some kind of a magic function which can take whatever you are searching for and instantly compute an index for it, a number, and let us say that your storage mechanism was to say that whatever number I get out of that hash, I will use that as the index in memory where I am going to store this particular element.

If I can do that, and it is big if in some cases, you can actually do searching in unit time, constant time $O(1)$, because all you need to do is compute the hash of whatever you are searching for, go check that memory location, either it is there or it is not. So, in the cases where you can do it at least or you can do it efficiently, it turns out that nothing can really beat this, but it is not always applicable. There are certain specific conditions where it can be used.