Video Transcript:

Hi guys! Mosh here! Today, we're gonna
talk about the basics of data structures and algorithms which is one of the
topics that come up in coding interviews all the time. In fact, more and
more companies ask questions about data structures and algorithms to see if you
can think like a programmer in this video we're going to talk about the
basics of data structures on algorithms we'll be talking about Big O notation
arrays and linked lists after watching this video if you want to learn more I
encourage you to enroll in my ultimate data structures and algorithms course
the link is below this video now to watch this video you don't need any
prior knowledge of data structures on algorithms but you need to know the
basics of programming in this video I'll be using Java but if you don't know Java
that's perfectly fine you can code in your favorite programming language if
you enjoyed this tutorial please support me by liking and sharing it with others
also be sure to subscribe as I regularly upload near videos all right now let's
jump in and get started before we talk about data structures and
algorithms we need to talk about the Big O notation we use the Big O notation to
describe the performance of an algorithm a lot of people find Big O scary but
don't worry I'm gonna make it super simple for you so let's jump in and get
started so what is this Big O all about well let's start with the classic
definition on Wikipedia Big O notation is a mathematical notation that
describes the limiting behavior of a function when the argument tends towards
a particular value or infinity huh that's the reason why a lot of people
find Big O scary but as you will see in this section the underlying concepts are
actually not that hard we use Big O to describe the performance of an algorithm
and this helps us determine if a given algorithm is scalable or not which

basically means is this algorithm going to scale well as the input grows really large so just because your code executes quickly on your computer doesn't mean it's gonna perform well when you give it a large data set so that's why we use the big o notation to describe the performance of an algorithm now what does this have to do with data structures well as you will learn in this course certain operations can be more or less costly depending on what data structure we use for example accessing an array element by its index is super fast but arrays have a fixed length and if you want to constantly add or remove items from them they have to get resized and this will get costly as the size of our input grows very large so if that's what we need to do then we have to use another data structure called a linked list these data structures can grow or shrink very quickly but accessing a linked list element by its index is slow so that's why you need to learn about the Big O notation first before we can talk about various data structures also big companies like Google Microsoft and Amazon always ask you about Big O they want to know if you really understand how scalable an algorithm is and finally knowing Big L will make you a better developer or software engineer so over the next few videos we're going to look at code snippets and use the Big O notation to describe the performance of our algorithms you here's our first example this method takes an array of integers and prints the first item on the console it doesn't matter how big the array is we can have an array with 1 or 1 million items all you're doing here is printing the first item so this methyl has a single operation and takes a constant amount of time to run we don't care about the exact execution time in milliseconds because that can be different from one machine to another or even on the same machine all we care about is that this

method runs in constant time and we represented using the Big O of one this is the run time complexity of this method so in this example the size of our input doesn't matter this method will always run in constant time or Big O of 1 now what if we duplicate this line here we have two operations both these operations run in constant time so the runtime complexity of this method is Big O of 2 now when talking about the runtime complexity we don't really care about the number of operations we just want to know how much on an algorithm slows down as the input grows larger so in this example whether we have 1 or 1 million items our method runs in constant time so we can simplify this by writing down o of 1 meaning constant time let's look at another example next you here we have a slightly more complex example have a loop so we're iterating over all the items in this array and printing each item on the console this is where the size of the input matters if you have a single item in this array we're gonna have one print operation if you have a million items obviously we're gonna have a million print operations so the cost of this algorithm grows linearly and in direct correlation to the size of the input so we resent the runtime complexity of this method using the Big O of n where n represents the size of the input so as n grows the cost of this algorithm also grows linearly now it doesn't matter what kind of loop used to iterate over this array here we're using a traditional for loop we could also use a for each loop for example for int number in numbers we could simply print the number we're still iterating over all the items in this array we could also use a while loop or a do-while loop now what if we have a print statement before and after our loop what do you think is the runtime complexity of this method well you saw that this single operations running constant time so here we have the Big O of one our loop runs in Big O

of n and once again we have the Big O of 1 so the run time complexity of this method is o of 1 plus n plus 1 which we can simplify to or of 2 plus n however when using the Big O notation we drop this constants because they don't really matter here's the reason if our array has 1 million inputs adding two extra operations doesn't really have a significant impact on the cost of our algorithm the cost of our algorithm still increases linearly so we can simplify this by dropping this constant what matters is that the cost of this algorithm increases linearly and in direct proportion to the size of our input if you have five items in the input we're gonna have five operations if you have a million we're gonna have a million operations now what if you had two loops here so let me delete these print statements and duplicate this loop what do you think is the runtime complexity of this method it's gonna be big-oh of n plus n or Big O of 2n this is another example where we drop the constant because all we need here is an approximation of the cost of this algorithm relative to its input size so N or 2n still represents a linear growth now what if this method had two parameters an array of numbers and an array of names so first we iterate over the area of numbers and then we iterate over the array of names like this what do you think is the runtime complexity here well both these loops run in O of n but here's a tricky part what is n in this case we're not dealing with one input we have two inputs numbers and names so we need to distinguish between these two inputs we could use n for the size of the first input and M for the size of the second input so the runtime complexity of this method is going to be o of n plus M and once again we can simplify this to O of n because the runtime of this method increases linearly you in the last video you learn that simple loops running linear time or Olaf n but here we have nested loops this is the

algorithm that we use for printing all combinations of items in an array so what is the runtime complexity here well let's find out in our outer loop we're iterating over our input array so here we have o of n now in each iteration once again we're iterating over all the items in this array another example of oh of n so the runtime complexity of this method is o of n times N or N squared we say this algorithm runs in quadratic time as you can see in this diagram algorithms that run in O of N squared gets slower than algorithms that run in O of n of course this depends on the size of the input if you're dealing with an array of let's say 50 items we're not gonna see any differences but as our input grows larger and larger algorithms that run in O of N squared get slower and slower now what if you had another loop before or after this loop for example let's add a for loop and once again iterate over all the items in this array and print them on a console what is the runtime complexity of this method well here we have o of n so the runtime complexity of this method is gonna be o of n plus N squared now once again we can simplify this this square of this number is always greater than the number itself right so in this expression N squared always grows faster than n again use the Big O notation to understand how much the cost of an algorithm increases all we need is an approximation not as an exact value so here we can drop in and conclude that this method runs in O of N squared let's look at another example what if instead of this loop we had another nested loop here so for end of third in numbers there you go the run time complexity is now o of n cubed you can imagine this algorithm gets far slower than an algorithm with o of N squared you another growth rate we're gonna talk about is the logarithmic growth which we show with the Big O of log n here's the

logarithmic curve now compare this with a linear curve as you can see the linear curve grows at the same rate but the logarithmic curve slows down at some point so an algorithm that runs in logarithmic time is more efficient and more scalable than an algorithm that runs in linear or quadratic time let's see a real example of this let's say we have an array of sorted numbers from 1 to 10 and we want to find the number 10 one way to find the 10 is to iterate over this array using a for loop going forward until we find the 10 this is called a linear search because it runs in linear time in the worst case scenario if the number we're looking for is at the end of our array we have to inspect every cell in this array to find the target number the more items we have the longer this operation is gonna take so the run time of this algorithm increases linearly and in direct proportion with the size of our array right now we have another searching algorithm called binary search and this algorithm runs in logarithmic time it's much faster than the linear search assuming that our array is sorted we start off by looking at the middle item it's this item smaller or greater than the value we're looking for it's smaller so our target number in this case 10 must be in the right partition of this array right so we don't need to inspect any of the items in the left partition and with this we can narrow down or search by half now in the right partition again we look at the middle item is it smaller or greater than the target value it's smaller so again we ignore the items on the left and focus on the items on the right so in every step we're essentially narrowing down or search by half with this algorithm if we have 1 million items in our array we can find the target item with a maximum of 19 comparisons we don't have to inspect every item in our array this is logarithmic time in action we have lovers meet growth in algorithms where we reduce our work by half in every step

you're going to see this a lot in the second part of this series where we talk
about trees and graphs unfortunately I cannot show you an
example of this in code now because it's a bit too complex there are a few things
we have to talk about before you are ready to see that in code but trust me
you'll see that in the code in the future and it'll become super easy for
now all I want you to take away is that an algorithm with logarithmic time is
more scalable than one with linear time you the last growth rate we're going to talk
about in this section is the exponential growth which is the opposite of the
logarithmic growth so the logarithmic curve slows down as the input size grows
but exponential curve grows faster and faster obviously an algorithm that runs
in exponential time is not scalable at all it will become very slow very soon
again I cannot show you an example of this in code yet we'll have to look at
it in the future for now all you need to understand is that the exponential
growth is the opposite of the logarithmic growth and by the way these
growth rates we have talked about so far are not the only growth rates but these
are the ones that you see most of the time there are some variations of these
that we look at in the future for now just remember these five cares
you you have seen how we can use the Big O
notation to describe the runtime complexity of our algorithms in an ideal
world we want our algorithms to be super fast and scalable and take minimum
amount of memory but unfortunately that hardly if ever happens it's like asking
for a Ferrari for ten dollars it just doesn't happen most of the time we have
to do a trade-off between saving time and saving space there are times where
we have more space so we can use that to optimize an algorithm and make it faster
and more scalable but there are also times where we have limited space like
when we build an app for a small mobile device in this situations we have to

optimize for this space because scalability is not a big factor only one user is going to use our application at that moment not a million users so we need a way to talk about how much space an algorithm requires and that's where use the Big O notation again let's look at a few examples here we have this greet method that takes an array of strings and prints a high message for every name in this array now in this method we're declaring and loop variable and this is independent of the size of the input so whether our input array has 10 or 1 million items this method will only allocate some additional memory for this loop variable so it takes all a one space now what if we declare is string array like this we call it copying and initialize it like this so the length of this array is equal to the length of our input array so if our input array has a thousand items this array will also have a thousand items what is the space complexity of this method it's all of n the more items we have in our input array the more space our method is gonna take and this is in direct proportion to the size of our input array that's why we have oh of n here and by the way when we talk about space complexity we only look at the additional space that we should allocate relative to the size of the input we always have the input of size n so we don't count it we just analyze how much extra space we need to allocate for this algorithm so that's all about space complexity in this course we'll only focus on runtime complexity because that's a bit more tricky but from now on think about the space complexity of your algorithms especially in situations where you have limited space see if there are ways to preserve the memory hey guys Marcia I wanted to let you know that this video is actually part of my ultimate data structures and algorithms course the complete course is 13 hours long and I've divided it into three parts so you can take and complete each part easily if you're serious about

learning data structures and algorithms I highly encourage you to take this course and learn all the essential data structures and algorithms from scratch it's much easier and faster than jumping from one tutorial to another we'll be talking about various types of data structures such as linked lists stacks queues hash tables binary trees AVL trees heaps tries graphs and various types of sorting searching and string manipulation algorithms the course is packed with almost 100 interview questions these are the interview questions that get asked that companies like Google Microsoft and Amazon you can watch the course online or offline anytime anywhere as many times as you want and you would also get a certificate of completion and a 30-day money-back guarantee it's exactly like this tutorial it just has more content if you're interested click on the link below this video to enroll in the course thank you and have a great day in this section we're going to talk about our very first data structure and one that you're probably familiar with arrays arrays are built into most programming languages and we use them to store a list of items sequentially in a section first we're going to look at various strengths and weaknesses of our race then I'm gonna show you how to use arrays in Java and finally we're gonna build an array class from scratch this is a fantastic exercise for you to get your hands dirty in the code and get prepared for more complex data structures so do not skip this section even if you know arrays well so let's jump in and get started arrays are the simplest data structures and we use them to store a list of items like a list of strings numbers objects and literally anything these items get stored sequentially in memory for example if we allocate an array of five integers these integers get stored in memory like this let's say the address of the first item in memory is 100 as you probably know integers in Java take 4 bytes of memory so the second item

would be stored at the memory location 104 the third item would be stored at the memory location 108 and so on for this very reason looking up items in an array by their index is super fast we give our array an index and it will figure out where exactly in memory it should access now what do you think is the runtime complexity of this operation it's all of one because the calculation of the memory address is very simple it doesn't involve any loops or complex logic so if you need to store a list of items and access them by their index arrays are the optimal data structures for you now let's look at the limitations or weaknesses of arrays in Java and many other languages arrays are static which means when we allocate them we should specify their size and this size cannot change later on so we need to know ahead of time how many items we want to store in an array now what if we don't know we have to make a guess if our guess is too large will waste memory because we'll have cells that are never filled if our guess is too small our array gets filled quickly then to add another item we'll have to resize the array which means we should allocate a larger array and then copy all the items in the old array into the new array this operation can be costly can you guess the runtime complexity of this operation pause the video and think about it for a second here's the answer let's say our array has 5 items now we will add the sixth item you have to allocate a new array and copy all these five items into that new array so the runtime complexity of this operation is o of n which means the cost of copying these items into the new array increases linearly and in direct proportion to the size of the array now let's talk about removing an eye here we have a couple of different scenarios if you want to remove the last item that's pretty easy we can quickly look it up by its index and clear the memory so here we have ol1 which is our best-case scenario but when doing Big O

analysis we should think about the worst case scenario what is the worst-case scenario here this is when we want to remove an item from the beginning of the array we have to shift all the items on the right one step to the left to fill in the hole the more items we have the more this shifting operation is going to cost so for the worst-case scenario deletion is an O of n operation so because arrays have a fixed size in situations where we don't know ahead of time how many items we want to store in them or when we need to add or remove a lot of items from them they don't perform well in those cases we use linked lists which we're going to talk about later in the course now let's see a quick demo of arrays in Java you in this video we're gonna look at arrays in Java if you know arrays well feel free to escape this video so to declare an array we'll start with the type of the array let's say we want to declare an array of integers so we type int and then we add square brackets to indicate that this is an array and not just a regular integer next we give our variable a name like numbers and here we use the new operator to allocate memory for this array here we repeat the type of the array one more time but this time inside the square brackets we specify the size of this array let's say 3 now let's print this on the console we get this weird string what is this well this is a combination of the type of the array followed by an @ sign and then this value that is generated based on the address of this object in memory that is not useful you want to see the content of this array to do that we're gonna use the arrays class let me show you so here in sort of printing numbers you're gonna use the arrays class we type arrays as you can see in this class is declared in this package Java dot util so we press ENTER and IntelliJ imports this class on the top now we use the dot operator and call the two string method a password array here and this method will convert it to a string and

then we'll print that string on the console no let's run the program one more time there you go much much better so as we can see all items in a numeric array are initialized to zero now let me show you how to change the value of these items so after we declare our array let's say we want to set the first item we type numbers once again we use square brackets and here we specify an index the index of the first item is 0 so here we are referring or referencing the first item we set this to let's say 10 similarly we can set the second item to 20 and the third item to 30 now let's run the program one more time so here's the content of our array beautiful now if you know the items that you're gonna store in your array ahead of time there is a shorter and cleaner way to initialize your array instead of doing all this ceremony we can use curly braces to declare and initialize this array so here we type 10 20 and 30 and this will allocate an array of 3 items in memory and it will initialize each item according to the values we have passed here take a look we get the exact same result as before now this area objects have a field called length and this returns the size of the array let's print this on the console and see what we get so print numbers dot lengths there you go the size of this array is three and we cannot change it so if want to store four items here you have to create another array copy all the existing items and then add the fourth item this is the problem with arrays in Java so if you want to work with lists that grow or shrink automatically you'll have to use linked lists you're going to talk about them in the next section but before we get there I'm gonna give you a fantastic exercise we'll look at that next so as you learn arrays in Java or static which means they have a fixed size and this size cannot be changed but now we're going to do something really cool I've created this array class which is

like a dynamic array as we add new items to it it will automatically grow and as we remove items it will automatically shrink let me show you how that works so we create a new array object we call it numbers and then initialize it like this here we pass the initial size let's say 3 now this numbers object has a method for adding new items let's add 10 and then 20 and then 30 we also have a method for printing this array but technically this print method shouldn't be here because an array should not be concerned about printing its content on the console it shouldn't know anything about console it should only be concerned about storing these values displaying these values is a completely different concern and should not be implemented as part of this class but in this course we want to keep things simple that's why I've implemented the print method inside the array class now let's run this program and see what we get so 10 20 30 beautiful so the initial size of our array is 3 but we can easily add a new item and our array is going to automatically grow no problem we also have a method for removing items that is remove at which gets an index let's say we want to remove the last item what is the index of this item
well the index of the first item is 0 then 1 2 3 so let's remove the last item here's the result beautiful you also have one method for finding the index of an item let me show you so I'm gonna do a print statement here and call numbers dot index of this will return the index of the first occurrence of this item let's say 10 so because 10 is the first item in this array this method is gonna return 0 take a look 0 if we pass a number that doesn't exist in the array let's say 100 it's gonna return negative 1
okay now here's your exercise I want you to build an array class that works exactly like what you saw in this video this is a fantastic exercise for you to get your hands dirty in the code especially for working with data

structures and algorithms don't say all mush

this is too easy I already know how to do this trust me as we talk about more

complex data structures our code is gonna get more complex so I want you to

treat this as a warm-up exercise so pause the video and spent 20 minutes on

this exercise when you're done come back see my solution

you all right we're gonna solve this problem

step by step and this is the approach I want you to follow whenever you want to

solve a complex problem don't try to do too many things at once try to break

down that problem into smaller easier to understand easier to solve problems so

in this video we just want to focus on creating this array class and printing

its content on the console you're not going to worry about adding new items or

removing existing items we're gonna implement these features in the

following videos so let's add a new class here we're going to right click

this package and add a new Java class we call it array

now in this class first we need to add a constructor so we type public array here

we need a parameter to specify the initial size of the array so and length

now inside this class we're gonna store our items in a regular Java array so

we're going to declare a private field private int array called items now here

in the constructor we need to initialize this array based on the initial size so

we sell items to new interim a of length pretty straightforward now let's

implement the print method so public void print here we need to iterate over

all the items in this array and print them on the console pretty easy so 4

into I we set it to 0 as long as I is less than items that length we increment

I and in each iteration we simply print items of I so in each iteration we get

one item from the array and print it on the terminal now let's go back to our

main class we're going to create a new array object we call it numbers and set

it to a new array of 3 now let's print this object on a console so we get these three zeros but technically we shouldn't see anything because we haven't inserted any items in this array so let's go back to our array class

we need another field to keep track of the number of items in this array we cannot rely on items that length because this is the memory we are allocating initially we might allocate memory for 50 items like you might only insert two items in this array so every time we insert a new item we need to keep track of the number of items in this array how can we do that we can declare another private field private int let's call it count now back to a print method we're gonna replace items that length with count

so initially count is zero and this loop is not gonna get executed in the future every time we insert a new item in this array we're gonna increment count by one so now let's run this program one more time because our array is empty we don't see anything beautiful we have completed the first step so next let's implement the insert method alright now let's implement the insert method so public void insert we give it a parameter int item now what should be doing this method there are a couple of things we need to do if the array is full we need to resize it and also we need to add this new item the new item at the end of the current array let's not worry about the first step yet instead we're gonna do the second step which is easier so we want to add this new item at the end of this array how can we do this well we use the items field we use square brackets now we need to pass an index

what is this index this index should represent the last item in this array it's not gonna be items that length it's gonna be count so currently we don't have any items in this array so the index of the last item or the place where we should insert the new item is index 0 next time we add a new item the

index is gonna be 1 and then 2 and 3 so we said items of count 2 this new item
and then we increment count by 1 or we can simplify this code get rid of this
line and increment count over here so with this expression first we said items
of count 2 item and then count is going to be incremented by 1 let's test our
code up to this point so back to the main class we're gonna call the insert
method and pass a couple of numbers 10 on 20 nanus run the program there you go
beautiful so let's go back to the array class and implement this scenario
how can we tell if they are useful that's very easy we can write an
expression like this if items that length equals count now in this case
what should we do first we need to create a new array that is larger and
let's say twice the size then we need to copy all the existing items into this
new array and finally we're gonna set the items field to this new array
because currently the array that the items field is referencing is four so
let's implement each step first we need to create a new array this is pretty
easy we declare an int array let's call it new items and set it to a new entry
of count times two so this new array is twice the size of the old array now we
need to copy all the existing items here we're gonna use a for loop exactly like
the for loop we have here so we need to iterate over all the existing items and
reference them using their index so four and I we set it to zero as long as
I is less than count we incremented after each step now in each step we're
gonna set new items of I two items I buy that is pretty straightforward finally
we need to set the items field to this new array
so we set items to new items now let's test this so back to the main class I'm
gonna add a couple more items and run the program
look now we have a dynamic array that automatically grows as we add new items
to it so now that you understand how everything works I'm gonna go back to

the array class and get rid of these additional comments we don't need this
comments because our code is clean and straightforward we don't need to repeat
it we should only use comments for explaining wise and house not what the
code is doing that should be reflected in the code itself so delete delete and
delete next we're going to implement the delete operation
you alright now let's implement the remove
method so public void remove at we give it an index now what should we do here
first we want to validate the index and make sure it's within the right range
for example if someone passes negative 1 it doesn't make sense what doesn't mean
to remove the item at index negative 1 or let's say our array has 5 items as
you know the index of the last item in this case showed before what if someone
says remove the item at the index 5 or 6 or 7 okay it doesn't make sense so first
we want to validate the index second we want to shift the items to the
left to fill the hole we'll talk about what this means in a second let's
implement each of these scenarios one by one so first we're going to validate the
index this is pretty easy we can write an if statement like this if index is
less than zero or we use two vertical bars to indicate a logical or or index
is greater than or equal to count what does this mean well if count is four
that means the index of the last item is three so we cannot tell this array to
remove the item at index 4 or 5 and so on so that is why we have greater than
or equal to count here now what should we do in this case we don't want to
print a message on the console because this class might be used in an
application with a graphical user interface there we don't have a console
so instead we should throw an exception because this is a programming error if
someone passes an index that is out of range so by throwing an exception we
forced a program to crash and with this the programmer knows that they made a

mistake and they will solve this problem so we throw a new exception of type illegal argument exception that was the first step now let's work on the second part let's imagine we have an array like this

10 20 30 and 40 and then we want to remove the item at index 1 that is this 20 over here so in order to remove 20 we should copy 30 over here and then 40 over here so we're shifting each item one step to the left in other words the item at index one should be set to what we have at index 2 and what we have at index 2 should be set to what we have at index 3 how can we implement this this is very easy we need a for loop that starts from this index and it goes all the way until it reaches the end of this array so for int I we set this to index as long as I is less than count the increment I after each step now in each iteration we want to set the item at this index to the item to its right side that is pretty easy so we set items of I to items of I plus 1 ok so after we execute this for loop our array is going to look like this 30 is gonna be copied over here and then 40 is gonna be copied over here but we still have 4 items in this array we want to shrink this array so it looks like this how can we do that very easy after our loop with decrement count by 1 because count represents the total number of items currently in there a not the size of the array right so let's test our code back to the main class we added four items here before printing the array let's call remove add and remove the first item so 10 is gonna go away now we have 20 30 40 beautiful let's test it with a different index let's say index 1 now 20 is gone beautiful let's do another test and remove the last item so it pass index 3 40 is gone what if you pass in X 4 we got an exception of type illegal argument exception this is a programming mistake we don't want to print a message on the console we want to stop the execution of the program so now that we're done with the implementation of

the remove method let's get rid of these unnecessary comments and make our code clean next we're going to implement the search operation finally let's implement the search operation so public int because we want to return the index of the given item we call this method index of and give it a parameter item now what should we do here we want to loop over all the items in this array if we find the item you want to return the index otherwise we're gonna return negative one so once again we're gonna use a for loop that's pretty easy for I we set this to zero as long as I is less than count you're gonna increment it by one now we need to get the item at the given index and compare it with this item so we write an if statement if items of I equals this item then we want to return I as the index otherwise if you finish this loop and we're still here we didn't return from this method that means we couldn't find this item so we should return negative one let's test our new method back to the main class we added these four items 10 20 30 40 let's print numbers that index of 10 so that is 0 beautiful what about the index of a number that we don't have let's say 100 that is negative 1 so we're done with this implementation but before I remove the comment let me ask you a question what is the runtime complexity of this method pause the video and think about it for a second ok here's the answer we need to analyze the best case on the worst case scenario the best case scenario is where this item is the first item in this array so in that case the runtime complexity is o of 1 but the worst case scenario is where this item is at the end of the array so we have to loop over the entire array to find that item if our array has 1 million items that means we're gonna have 1 million comparison operations so in the worst case scenario the runtime complexity of this method is o of as I told you before when doing Big O analysis we always consider the worst case scenario

so the runtime complexity of this method is o of N

you so you learn how to build a dynamic

array from scratch and that was a great exercise

however Java has two implementations of dynamic arrays let me show you we have

two classes vector and array list both these classes are declared in the Java

that util package but they're slightly different the victor class would grow by

how to person of its size every time it gets full whereas the ArrayList will

only grow by 50% of its size also all the methods in the vector class are

synchronized this is an advanced topic and I'm gonna cover that in my upcoming

advanced Java course but basically when we say a method is synchronized that

means only a single thread can access that method in other words if you have a

multi-threaded application where multiple threads are gonna work with

this collection that you're not gonna be able to use the victor class you should

use the ArrayList class because the methods of the ArrayList are not

synchronized again I'm gonna cover that in detail in my upcoming advanced Java

course now let's have a quick tour of the ArrayList class so let's type array

list then this angle brackets you see here these represent a generic parameter

with this generic parameter we specify the type of each element in this array

list for example if you want to have an array list of integers we type integer

this integer class is a wrapper around the native or primitive int type so for

every primitive time that we have like int short white boolean whatever we have

a wrapper class for example we have short we have white we have boolean and

so on we can also have an ArrayList of strings or students assuming that we

have a student class in this demo I'm gonna create an array list of integers

so integer n term now we need to import the ArrayList class because it's

declared in a different package so we press Alt + Enter

there you go it's important on the top now let's create our ArrayList we call

this list and initialize it using the new operator like this new ArrayList and now we can call the add method we

can add a number here and duplicate this line a few times I have two more numbers

now we can print this list on the console so here is the content of our

array beautiful we can also remove items so remove we can remove a particular

object or remove an item at a given index for example we can remove the

first item and now ten is gone we only have 20 and 30 we can also find the

index of the first occurrence of an element so recall list that index of 20

because now after removing the first item 20 is going to be the first item

this method will return zero we also have last index of which will return the

index of the last occurrence of an item we also have contains which returns a

boolean value telling us if you have this item in our array or not and

finally we can use the size method to get the number of items in this array

and finally another useful method is the true array method this will convert this

list to a regular array object so there are times that you want to work with a

regular array object let's say you have a method that only accepts an array and

you cannot pass an ArrayList class there in that case we can easily convert your

ArrayList to a regular array linked lists are probably the most

commonly used data structures after arrays they solve many of the problems

with arrays and are also used in building more complex data structures so

in this section we're gonna look at linked lists we'll talk about how

they're structured in memory we'll look at the time complexity of various

operations on them and finally we're gonna build a linked list from scratch

again this is an incredible exercise for you to train your programming brain so

let's jump in and get started we use linked lists to store a list of objects

in sequence but unlike arrays linked lists can grow and shrink automatically

as you can see here a linked list consists of a group of nodes in sequence

each node holds two pieces of data one is a value and the other is the address

of the next node in the list so we say each node points to or references the

next node that's why we refer to these structures as linked lists because these

nodes are linked together we call the first node the head and the last node

the tail now let's look at the time complexity of various operations let's

say you want to find out if our list contains a given number we have to

traverse the list starting from the head all the way to the tail what is the

runtime complexity here it's o of n because the value that we are looking

for may be stored in the last node that is our worst case scenario right what

about looking up by index well unlike arrays where items are stored

sequentially the notes of a linked list can be all over the place in memory they

may not be exactly next to each other that's why each node needs to keep a

reference to the next node for this reason unlike arrays we cannot quickly

look up an item by its index we have to traverse the list until we find that

item in the worst case scenario that item can be at the end of the list so

once again here we have o of n what about insertions well it depends where

we want to insert an item if you want to insert a new item at the end we simply

need to create a new node and have the last node or the tail point to it we

should have a reference to the last node somewhere so we

have to traverse the list every time now we need to have the tail reference this

new node so inserting a new item at the end is an O of one operation

what about inserting at the beginning what do you think is the runtime

complexity here pause the video and think about it here's the answer it's an oil one

because again we should have a reference to the head or the first note so to

insert a new item at the beginning of the list we create a new note linked it
to the first note and then change the head to point to this new note again
this is very fast unlike a race we don't have to copy or shift items around we
simply update the links or references now what if you want to insert an item
somewhere in the middle let's say after the tenth note well first we have to
find that note that's an O of n operation and then we have to update the
links which is an O of one operation so inserting an item in the middle is an O
of n operation now let's talk about deletions I want you to pause the video
and think about three scenarios deleting an item from the beginning from the end
and from the middle draw on a piece of paper how the links should be updated
also calculate the runtime complexity for each scenario this is very important
make sure to do this little exercise because later on you're going to code
all of this if you don't understand these concepts you're not going to be
able to code them so pause the video do the exercise when you're done come back
continue watching all right here are the answers deleting
the first item is super fast we simply set the head to point to the second note
that's an O of one operation now we should also remove the link from the
previous head so it doesn't reference the second note anymore
why because if you don't do this Java's garbage collector thinks this objects
still used so it won't remove it from the memory that's why we should unlink
this object from the second object what about deleting the last item this
one is a bit tricky we can easily get the tail but we need to know the
previous node so we can have the tail point to that node how can we do that we
have to traverse the list from the head all the way to the tail as soon as we
get to the node before the last node we keep a reference to it as the previous
node then we'll unlink this node from the last node and find in half the tail

point to the previous node so the runtime complexity here is o MN because we have to traverse the list all the way to the end what about deleting from the middle again we have to traverse the list to find out the node as well as its previous node we should link the previous node to the node after this node and then remove this link so this object gets removed from memory by Java's garbage collector again here we have an O of n operation next we're gonna work with linked lists in Java in this video we're going to look at linked lists in Java so if we type linked list we can see this class is defined in Java that util package this angle brackets you see here these are generics that means we can store any kind of objects in this list you can store integers strings any type of objects so let's press Enter this class is imported on the top beautiful now let's say we want to store a bunch of integers in this linked list so we add angle brackets and type integer with capital I because here we're using the integer class that is defined in Java that Lang package not the built in primitive type so we should always reference a class here this integer class wraps a primitive integer okay or we could have a linked list of strings or if we don't specify anything here we can store any kind of objects in this list one node can hold an integer another node can hold a string so let's create a list once again we have to use the new operator to allocate memory for this object so a new linked list now we have a bunch of methods for adding new items we can add at the beginning or at the end let's add 10 at the end and then 20 and 30 now let's write a print line statement and print this list there you go it looks like we have an array but actually we're dealing with a list so don't let these square brackets fool you okay now we can also add an item at the beginning so we call list that at first let's add 5 here there you go now we have 5 10 20 or 30 we have similar

methods for removing items so we can call lists that remove last term of the
last item we also have remove which takes an index as well as remove first
for removing the first item another useful method is the contains method we
can use this to see if our list contains the number 10 so let's do a print line
statement and move this expression over here so our list certainly does include the
value 10 we have a similar method that is lists that index of which will return
the index of the first occurrence of this object so if you pass 10 here this
will return 0 because that is our first item there you go another useful method
is the size method so let's print that list that size this will return the
number of items in this list which is three beautiful and finally the last
useful method I want to cover is list the to array
there are times you want to work with an array so you can convert a linked list
to a regular array let's convert it to an array and store it here now we can
use the arrays class to convert this array to a string and then print it on
the console there you go so this is how linked lists work in Java you alright now just like the previous
section we're gonna build a linked list from scratch this is a great exercise
for you to practice all the materials in this course but before we get started I
want to give you a couple of hints to do this exercise you need two classes a
note class like this here we have a couple of fields an integer called value
and a node called Nick's so with this field we can keep a reference to the
next note we also need a linkless class with these two fields first and last we
could call them head and tail but to be consistent with the link list class in
java i decided to call this first and last I would recommend you to follow the
same names so as you will see my solution you don't get confused about
these names you can simply compare your code with mine and here are the metals I
want you to implement in this exercise at first at last

delete first till at last contains and index off these are the essential
methods that we need in a linked list so spend 30 to 40 minutes on this exercise
do not skip this it's super important because in the next section I'm going to
talk about stacks and queues and we're gonna implement them using a linked list
so linked list is one of those essential fundamental data structures that you
need to master all right enough talking so grab a coffee and get started
you all right let's start by implementing
the at last method so public void at last we give it an integer now what
should we do here the first step is to wrap this value of this integer inside a
node object so we create a node object like this
now as we can see we have repeated the name of the class twice and this is
unnecessary we can use the VAR keyword I let the Java compiler detect the type of
this variable so because we have new node on the right side the Java compiler
will know that this variable is a node object all right now we need to set node
that value to this item however this field is declared as
private and that's why we cannot access it from outside of this node class we
can come here and create a setter like public void set value which takes a
value and here we type this that value equals value but I want to show you a
better way I argue that this node class is part of the implementation of the
linked list we don't need to work with this node class directly so this should
not be declared as a public class here that can be accessed anywhere in our
program earlier when we worked with the linked list class in Java if we see a
node class we didn't we simply called various methods on the linked list and
the linked list took care of everything under the hood so this node class is
something that the linked list class shall have internally it's an
implementation detail so we can remove this setter and move this class inside

the linked list so we can add it here on the top

there you go now because this class is declared inside the link list we have

access to its private fields so we don't need a setter also we should change this

to private so nowhere in our program we can access the note class that's better

now another thing we need to improve here is this line with this

implementation we can have a note that doesn't have a value this doesn't make

sense whenever we create a note object it should always store a value so we can

create a custom constructor for the note class and pass this value there so here

in the note class we type public node in this constructor we add a value and we

set this dot value to value now we can get rid of line 18 and simply

pass the value here sorry I made them item so our code is shorter and our note

object will always be in a valid state you're not gonna have a note without a

value that doesn't make sense so we have a note what should we do next well that

depends on the state of the linkless if our linked list is empty we need to set

both the first and last note or head and tail to point to this new node otherwise

we need to append this node at the end of the list let me show you so here's

the first scenario we should check to see if the list is empty or not how can

we do that we write an if statement like this if first equals no that means we

don't have any nodes in this list because as soon as we add a node in this

list first should be initialized so if first is now we should set first to this

new node we should also set last to this new node or we can simplify these two

lines and initialize both these variables on the same line that is

better now we don't need these ugly curly braces so let's look at the other

scenario where our list has at least one node so else in this case we want to add

this node after the last node so we type last that next equals node we're linking

the last node to this new node finally we should update last to point to this

new node because now we have one new node in this list so we type last equals
node we're done with the implementation of this method so let's use our new list
and see if it's working properly so back to the main class I want to create a new
linked list or call it list now once again we can use the VAR
keyword to simplify this code here we're gonna call list then at last 10 and then
20 and then 30 now currently we don't have a method for printing this list and
I realize I forgot to tell you to implement this method but in this video
I don't want to spend time implementing the print method instead I'm gonna show
you a different technique so we add a breakpoint on this line by clicking on
this area look now it's red now we're gonna run this code using the debugger
so on the top look run debug main is the shortcut that's ctrl + D on Mac so here
we are all the previous lines are executed but this line is not so we can
click on this icon that is step over now all these lines are executed so let's
inspect our list object and see if it's structured properly so in this debug
window let me expand this good so here we have this list let's look at the
first node so the value is 10 now next this is referencing another
node what is this node here we have 20 and this is also referencing another
node in this node we have 30 but next here is set to null because this is the
last node in our list so far so good what about our last node this is
pointing to the same object where we have the value of 30 beautiful so we're
done with this step we'll implement another method next all right let's implement the ad first
method this is very similar to what we did in the previous video so public void
at first which takes an item now once again we need to wrap this item inside a
node object so far node equals new node of either now here we have two scenarios
if the list is empty we need to add the first node otherwise we need to prepend
this item to the list so we check if first is no then just like before we set

first and last to this new node otherwise we want this node to point to
our first node so your type node that next equals first and then we need to
set the first node to this new node so first
EKOS node we're done with the implementation of this method but I want
to show you a technique for making this code more readable and more maintainable
this is something that unfortunately most data structures books and courses
don't teach you most of the code samples I say in this book look disgusting
they look really ugly like old school the code we used to write in 1980s so
how can we improve this code and make it more readable well look at this logic
what is the point of this logic you're trying to see if this list is empty or
not so we can extract this into a private method and call it is empty let
me show you so here we create a private method it's private because this is
implementation detail we don't want this to be accessible outside of this class
so public boolean is empty now here is simply return head equals sorry
first equals no now we can improve this code by
replacing this logic with a call to this new method isn't the cleaner
let's also modify the add last method is empty beautiful next we're going to
implement the indexof method alright now let's implement the index of
methods of public int index of this item what should we do here
we need to traverse this list starting from the beginning all the way to the
end as soon as we find an item with this value we're gonna return the index of
that item but we don't have indexes here so how
are we gonna implement that well we can declare a variable index and initially
set it to 0 then as we're traversing this list we increment this index so we
need another variable let's say current we set this to the first node now we
need a while loop as long as current is not no which means we haven't reached

the end of the list we need to compare the value of the current node with this

item so if current that value equals this item we want to return the current

index silver return index otherwise you're gonna set current to the next

node so we set current to current dot next and at the same time we should also

increment index now if we reach the end of the list and we can't find a node

with this value we need to return negative 1 all right now let's test our

code so back to the main class we added a few items here now let's print list

that index of 10 so we get zero beautiful what about

index of 30 the last item I always look for this edge cases that is too perfect

and finally an item that doesn't exist in the list so negative one beautiful

next we're going to implement the contains method

you the contains method is pretty easy so

public boolean contains item now what should we do here once again we should

traverse the list starting from the beginning all the way to the end if you

find this item will return true otherwise we'll return false however we

already built this logic in our index of method so we can reuse this there is no

need to repeat this logic so we type return index of item does not equal to

negative one so if this expression if value is to true that means we have this

item in our list let's test this so back to the main class we're gonna call list

that contains for D obviously we don't have this item what about 10 we get true

beautiful that was very easy next we're gonna implement the delete first method right now let's work

on removing the

first item so public void remove first we don't need any parameters here this

one is a bit tricky imagine our list looks like this ten point into twenty

point into thirty now we want to remove the first item so we have this field

called first that is currently pointing to ten we should have this point to the

second node and this will bring our list forward it's gonna look like this right

however we still have this object this first note that is referencing the

second node so the garbage collector in Java will not be able to remove this

object from the memory to solve this problem we need to remove this link now

here's the tricky part if we remove this link we are not going

to be able to set first to point to the second node because the moment we remove

this link we lose track of the second point so to solve this problem we need

two different references first and second let me show you so I'm going to

bring this back let's write some code so first we declare a variable called

second we said this to first dot next so ii is pointing to twenty now that we

have this we can go and remove this link without worrying about losing track of

the second point because we have the second variable as a backup here so we

go and set first up next to know this will remove this link and finally we

need to update first and set it to point three second node so we set first to

second let's test this so back to the main class after adding these items

let's call list dot remove first just like before we're gonna run this program

using the debugger so ctrl + D okay what we have here in this list first is

pointing to the note that contains 20 and this is pointing to this other note

beautiful now what if our list is empty and we call the remove first method

let's see what happens we got an exception of type null pointer

exception this is a programming error we shouldn't let this happen so let's see

how the built in LinkedIn class in Java works and we'll implement the same

behavior in this class so temporarily I'm gonna create another linked list

this time we're using the class that is declared and the Java that util package

so we're gonna create a linked list of strings we call it X and instantiate it

like this now we call X EE move first let's see what happens so we got an exception but look at the

type of this exception no such element exception this is different from a

nullpointerexception this is a deliberate error handling so

we should not be able to remove an item from an empty list

so we're gonna go back to our linkless class and before this logic you want to

add an if statement like this if this list is empty look once again we're

reusing this beautiful method so if the list is empty we're gonna throw and you

know such element exception this is the proper way to implement this method

hey I just wanted to let you know that when I was reviewing my videos I noticed

I made a mistake here I didn't count for the scenario where our linked list has a

single item because this logic would work for a list that has at least two

items first and second so here we need to add an if statement and check to see

if first equals last that means we have a single item or a single note in this

list in this situation if you want to remove this item we should set both

these fields to no and then return because we don't wanna execute this

logic over here so see even I make mistakes so does everybody it doesn't

matter what matters is that we should always review our code we should test it

with different inputs and think of various edge cases

you alright now let's amend the remove last

method this is the trickiest part so pay close attention we're gonna declare in

your method public void remove last so let's imagine our list looks like this

10 pointing to 20 pointing to 30 and we have this last field that is pointing to

this node now to remove the last item we need to find the previous node this is

the tricky part so we need to traverse this list starting from the beginning

the moment we get here we need to keep a reference to this node so we can update

last and set it to point to the same node so let's implement this

step-by-step first we want to find the previous node we start by declaring a

variable called current and we set it to the first node now as long as current is not no we're gonna go forward first we check to see if current that next equals the last note if that's the case we need to break out of this loop otherwise we set current to point to the next node so at this point we have the previous note now if we're going forward I want to refactor this code and extract this logic into a separate private method because at a first glance it might not be clear what we're trying to do here so let's declare a private method that returns a node object we can call this get previous and give it a node object so whatever node we give it it will return the previous node so let's move this logic over here now instead of working with the last node you want to work with the node that is passed here so let's change that beautiful also instead of breaking out of this loop we can simply return the current node now what if we traverse the list all the way to the end but we couldn't find the node before this node we should return no now that we have all this logic in a single place we can go back to the remove last method and call get previous give it the last node and store the result in a variable called previous so in this case previous is gonna point to this node what should we do now well currently last is pointing to this node we should change last and make it point to previous so this will shrink our list like this however there's still this object that is pointing to this other object we should remove this link so the garbage collector in Java can also remove this last node from the memory so we said last two previous this will shrink our list and then to remove the link we said last that next to nan let's test our code after this point so back to the main class after adding these items let's remove the last item now let's start the debugger there you go so here's our list let's see what we

have here we have first that is referencing its node this is referencing
this other node and here we don't have anything after so we successfully
removed the last node beautiful let's also make sure that this last field is
referencing the same node beautiful so we're gonna stop the debugger now we
need to think of the edge cases what if the list is empty just like before we
should throw an exception so I'm going to remove these comments and check to
see if the list is empty we want to throw and you know such element
exception what if our list has only a single item
this logic is not gonna work because there is no node before the last node we
only have a single node so this logic is assuming that our list has at least two
nodes so we need another if statement if first equals last that means we have a
single node in this list in this situation
should set both these fields to know and then return so this other logic is not
executed so this is how we implement the remove last method
hey guys Marsha I wanted to let you know that this video is actually part of my
ultimate data structures and algorithms course the complete course is 13 hours
long and I've divided it into three parts so you can take and complete each
part easily if you're serious about learning data structures and algorithms
I highly encourage you to take this course and learn all the essential data
structures and algorithms from scratch it's much easier and faster than jumping
from one tutorial to another we'll be talking about various types of data
structures such as linked lists stacks queues hash tables binary trees AVL
trees heaps tryes graphs and various types of sorting searching and string
manipulation algorithms the course is packed with almost 100 interview
questions these are the interview questions that get asked that companies
like Google Microsoft and Amazon you can watch the course online or offline

anytime anywhere as many times as you want and you would also get a certificate of completion and a 30-day money-back guarantee it's exactly like this tutorial it just has more content if you're interested click on the link below this video to enroll in the course thank you and have a great day

Summary:

Understanding Data Structures and Algorithms

In this video, Mosh explains the basics of data structures and algorithms, which are essential topics in coding interviews. He mentions that more and more companies are asking questions about data structures and algorithms to see if a candidate can think like a programmer.

Big O Notation

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. In simpler terms, it helps us determine how an algorithm scales as the input grows larger. Mosh explains that knowing Big O will make you a better developer or software engineer.

Examples of Big O Notation

1. **Constant Time**: A method that takes a constant amount of time to run, regardless of the input size. Example: printing the first item in an array. Big O = 1.
2. **Linear Time**: A method that runs in direct proportion to the size of the input. Example: iterating over an array and printing each item. Big O = n.
3. **Quadratic Time**: A method that runs in N squared, where N is the size of the input. Example:

printing all combinations of items in an array. Big O = N squared.

4. **Logarithmic Time**: A method that runs in logarithmic growth, which is more efficient and scalable than linear or quadratic time. Example: finding an item in a sorted array.

Key Takeaways

* Big O notation helps us understand how an algorithm scales as the input grows larger.
* Knowing Big O will make you a better developer or software engineer.
* Common growth rates include constant, linear, quadratic, and logarithmic.
* Dropping constants in Big O notation is common, as they don't matter in terms of scalability.

Example Code Snippets:

* `print(numbers[0])` - Big O = 1 (constant time)
* `for (int i = 0; i < numbers.length; i++) { print(numbers[i]); }` - Big O = n (linear time)
* `for (int i = 0; i < numbers.length; i++) { for (int j = 0; j < numbers.length; j++) { print(numbers[i] + numbers[j]); } }` - Big O = N squared (quadratic time)
* `int index = binarySearch(numbers, target);` - Big O = log n (logarithmic time) The context is about algorithms and data structures, specifically focusing on arrays. The discussion involves understanding the time and space complexity of different operations, such as searching, inserting, and deleting items in an array.

Time Complexity:

* Linear search: This is a simple search algorithm that looks for an item in an array by checking each cell one by one. Its time complexity is O(n), meaning it takes longer to search for an item as the size

of the array increases.

* Binary search: This is a more efficient search algorithm that works by dividing the array in half and searching for the item in one of the two halves. Its time complexity is O(log n), making it much faster than linear search.

Exponential Growth:

* This is the opposite of logarithmic growth, where the time complexity increases rapidly as the size of the array grows.

Space Complexity:

* The amount of extra space an algorithm requires in addition to the input size is known as space complexity.
* For example, if an algorithm requires an extra array of the same size as the input array, its space complexity is O(n).

Arrays:

* Arrays are a basic data structure that stores items sequentially in memory.
* They are useful for storing a list of items and accessing them by their index.
* However, arrays have limitations, such as being static (their size cannot change after creation) and requiring a costly operation to resize them.

Operations on Arrays:

* Searching for an item in an array by its index is an O(1) operation, meaning it takes constant time regardless of the size of the array.
* Inserting a new item into an array requires resizing the array, which has a time complexity of O(n) in the worst case.
* Deleting an item from the beginning of an array requires shifting all the items to the left, which also has a time complexity of O(n) in the worst case.

Key Takeaways:

* Understanding the time and space complexity of different operations is crucial for designing efficient algorithms and data structures.
* Arrays are a basic data structure that can be useful for storing a list of items, but they have limitations that need to be considered.
* More efficient data structures, such as linked lists and binary search trees, can be used to improve the performance of algorithms in certain situations. **Context: Array Data Structure in Java**

In this scenario, we are learning about arrays in Java, a type of data structure that stores a collection of items of the same type in a single variable.

Declaring an Array in Java

To declare an array in Java, we use the following syntax:

```java
int[] numbers = new int[3];
```

This declares an array called `numbers` that can hold 3 integers.

Printing an Array in Java

To print the contents of an array, we can use the `Arrays.toString()` method:

```java
System.out.println(Arrays.toString(numbers));
```

This will print the contents of the array as a string, like this: `[0, 0, 0]`.

Initializing an Array in Java

We can also initialize an array with values directly:

```java
int[] numbers = {10, 20, 30};
```

This creates an array with 3 integers, initialized with the values 10, 20, and 30.

Array Class Exercise

The instructor provides an exercise to create an `Array` class that can dynamically grow and shrink

as items are added or removed. The class should have methods for inserting, removing, and printing items.

Array Class Implementation

We create a new class called `Array` with a constructor, fields for storing items and keeping track of the count, and methods for inserting, removing, and printing items.

Here's an example implementation:

```java
public class Array {
    private int[] items;
    private int count;

    public Array(int length) {
        items = new int[length];
        count = 0;
    }

    public void print() {
        for (int i = 0; i < count; i++) {
            System.out.println(items[i]);
        }
    }
```

```
    public void insert(int item) {
        if (items.length == count) {
            // resize the array
            int[] newItems = new int[items.length * 2];
            System.arraycopy(items, 0, newItems, 0, items.length);
            items = newItems;
        }
        items[count] = item;
        count++;
    }

    public void remove(int index) {
        // remove the item at the given index
        System.arraycopy(items, index + 1, items, index, count - index - 1);
        count--;
    }
}
```

This implementation includes a constructor that initializes the array with a given length, a `print()` method that prints the contents of the array, an `insert()` method that adds a new item to the end of the array and resizes the array if necessary, and a `remove()` method that removes an item at a given index. **Dynamic Array Implementation**

The context is about implementing a dynamic array from scratch in Java. A dynamic array is an array that can grow or shrink in size as items are added or removed.

Key Concepts

1. **Creating a new array**: When the array needs to grow, a new array is created with twice the size of the original array.
2. **Copying existing items**: The existing items are copied from the original array to the new array.
3. **Setting the items field**: The `items` field is set to the new array, which is now the current array.

Remove Method

The `remove` method is implemented to remove an item at a specified index. The steps are:

1. **Validate the index**: Check if the index is within the valid range. If not, throw an `IllegalArgumentException`.
2. **Shift items to the left**: Shift the items to the left to fill the hole created by removing the item.
3. **Decrement the count**: Decrement the count of items in the array.

Search Operation

The `indexOf` method is implemented to search for an item in the array. The steps are:

1. **Loop over the items**: Loop over the items in the array to find the item.
2. **Return the index**: If the item is found, return its index. Otherwise, return -1.

Runtime Complexity

The runtime complexity of the `indexOf` method is O(N), where N is the number of items in the array. This is because in the worst-case scenario, the method needs to loop over the entire array to find the item.

Java Built-in Classes

Java has two built-in classes for dynamic arrays: `Vector` and `ArrayList`. The differences between them are:

* `Vector` grows by 100% of its size every time it gets full.
* `ArrayList` grows by 50% of its size every time it gets full.
* `Vector` methods are synchronized, while `ArrayList` methods are not.

Example Code

```java
public class DynamicArray {
    private int[] items;
    private int count;

    public DynamicArray() {
        items = new int[4];
        count = 0;
    }

    public void add(int item) {
```

```
    if (count == items.length) {
        // Create a new array with twice the size
        int[] newItems = new int[items.length * 2];
        System.arraycopy(items, 0, newItems, 0, items.length);
        items = newItems;
    }
    items[count++] = item;
}

public void remove(int index) {
    if (index < 0 || index >= count) {
        throw new IllegalArgumentException();
    }
    for (int i = index; i < count - 1; i++) {
        items[i] = items[i + 1];
    }
    count--;
}

public int indexOf(int item) {
    for (int i = 0; i < count; i++) {
        if (items[i] == item) {
            return i;
        }
    }
    return -1;
```

```
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        DynamicArray array = new DynamicArray();
        array.add(10);
        array.add(20);
        array.add(30);
        array.add(40);

        System.out.println(array.indexOf(10)); // Output: 0
        System.out.println(array.indexOf(100)); // Output: -1

        array.remove(0);
        System.out.println(array.indexOf(20)); // Output: 0

        array.remove(1);
        System.out.println(array.indexOf(40)); // Output: -1
    }
}
```

```java
```

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(20);
        list.add(30);
        list.add(40);

        System.out.println(list.size()); // Output: 4
        System.out.println(list.contains(20)); // Output: true
        System.out.println(list.indexOf(20)); // Output: 1
        System.out.println(list.lastIndexOf(20)); // Output: 1
        list.remove(1);
        System.out.println(list.size()); // Output: 3
    }
}
```

``` The context is about linked lists, a type of data structure used to store a collection of objects in a sequence. Unlike arrays, linked lists can grow and shrink automatically as items are added or removed.

**Key Points:**

1. **Linked Lists:** A linked list consists of a group of nodes in sequence, each holding a value and a reference to the next node.
```

2. **Time Complexity:** Linked lists have a time complexity of O(n) for operations such as searching, inserting, and deleting items.
3. **Insertion and Deletion:** Inserting or deleting an item at the beginning or end of a linked list is an O(1) operation, while inserting or deleting an item in the middle is an O(n) operation.
4. **Java Implementation:** Java has a built-in `LinkedList` class that can store any type of object.
5. **Implementing a Linked List from Scratch:** Creating a linked list from scratch involves implementing two classes: `Node` and `LinkedList`.

Example Use Cases:

* Creating a linked list to store a collection of integers or strings
* Adding or removing items from the beginning or end of the list
* Searching for a specific item in the list
* Converting a linked list to an array

Key Methods:

* `addLast(int item)`: Adds an item to the end of the list
* `addFirst(int item)`: Adds an item to the beginning of the list
* `removeLast()`: Removes the last item from the list
* `removeFirst()`: Removes the first item from the list
* `contains(int item)`: Checks if the list contains a specific item
* `indexOf(int item)`: Returns the index of the first occurrence of a specific item
* `size()`: Returns the number of items in the list
* `toArray()`: Converts the linked list to an array **Linked List Implementation in Java**

In this context, a linked list is a data structure where each element is a separate object, known as a node. Each node contains a value and a reference (or link) to the next node in the list.

The instructor is implementing a linked list class in Java, providing comments on how to improve the code and make it more readable. The steps involved include:

1. **Improving the Node class**:
   - The `Node` class is moved inside the `LinkedList` class, making it a private implementation detail.
   - A custom constructor is added to the `Node` class to ensure that each node has a value.

2. **Implementing the `addLast` method**:
   - If the list is empty, the `first` and `last` nodes are set to the new node.
   - Otherwise, the new node is appended to the end of the list.

3. **Implementing the `addFirst` method**:
   - If the list is empty, the `first` and `last` nodes are set to the new node.
   - Otherwise, the new node is prepended to the list.

4. **Implementing the `indexOf` method**:
   - Traverse the list to find the index of the given item.
   - Return -1 if the item is not found.

5. **Implementing the `contains` method**:
   - Use the `indexOf` method to check if the item is in the list.

6. **Implementing the `removeFirst` method**:
   - Create a temporary reference to the second node (`second`).
   - Set the `next` field of the `first` node to `null`.
   - Update the `first` node to point to the second node.

Example Use Cases

```java
public static void main(String[] args) {
    LinkedList list = new LinkedList();

    list.addLast(10);
    list.addLast(20);
    list.addLast(30);

    System.out.println(list.indexOf(10));  // Output: 0
    System.out.println(list.indexOf(30));  // Output: 2
    System.out.println(list.indexOf(25));  // Output: -1

    System.out.println(list.contains(10));  // Output: true
    System.out.println(list.contains(30));  // Output: true
    System.out.println(list.contains(25));  // Output: false

    list.removeFirst();

    System.out.println(list.indexOf(20));  // Output: 0
```

}
```

This implementation provides basic operations for a linked list, including adding, removing, and searching for elements. **Context: Implementing Remove Method in Linked List**

The context is about implementing a remove method in a linked list data structure in Java. A linked list is a type of data structure where each element is a separate object, known as a node, that contains some data and a reference (or "link") to the next node in the list.

**Problem: Handling Exceptions**

The speaker is discussing how to handle exceptions when removing elements from a linked list. There are two types of exceptions that can occur:

1. **NullPointerException**: This occurs when trying to remove an element from an empty list.
2. **NoSuchElementException**: This occurs when trying to remove an element from a list that does not exist.

**Solution: Implementing Remove Method**

The speaker is implementing a remove method that handles these exceptions. The method has two cases:

1. **Removing the last element**: The speaker explains how to traverse the list to find the previous node and update the last node to point to the previous node.
```

2. **Removing the first element**: The speaker explains how to set the first node to null and update the last node to point to the second node.

Edge Cases

The speaker discusses two edge cases:

1. **List is empty**: The speaker explains that if the list is empty, the method should throw a NoSuchElementException.
2. **List has only one element**: The speaker explains that if the list has only one element, the method should set both the first and last nodes to null and return.

Course Promotion

At the end of the video, the speaker promotes their online course, "Ultimate Data Structures and Algorithms Course", which covers various data structures and algorithms, including linked lists, stacks, queues, hash tables, binary trees, and more.