Hey? Good evening, everyone. Let me 1st begin by apologizing for not delivering this lecture in person. Well, I guess you never know when things are gonna happen. So sorry again. But let's get on with this. So last time we learned how to do the add 1st in the link list, and we also learned that the process was very easy, well, quite simple, and also computationally light. And it involves a lot of pointer rearrangements or or stitching, as I like to call it, but fairly intuitive, and the time complexity was big old one, because it didn't involve any kind of loops in here, and all the statements here are all simple statements because of comparisons and assignments and arithmetic operations. So now let's try to do the removing the 1st element or the 1st node of a linked list. and it's going to be the exact opposite of add first.st So the code you see on the left hand side is the sort of the main method of remove first, st and you can see that it has to return the element that we've removed, because that's the that's that's the the common trait of any kind of destructive update right? Every time you're removing something, or every every time you're overwriting something. Then the old value must be returned back to the caller. So here we are, 1st caching the 1st pointer, the head pointer of the lake list. So F is now pointing at the very 1st node, namely, the target node we want to remove. and if F is null, then that means the the whole link list is empty. Right? So the 1st pointer being null, means there's nothing to point at, and that clearly indicates that there isn't anything in this link list. Okay. Now, if you try to remove something from an empty linked list or empty any kind of empty container. Okay, then, that shouldn't work. Okay, that's kind of like an error case. So normally, we would try to return a null to indicate that there's been an error, right? That's very common thing to do. But here, instead of returning a null, we're throwing an exception. Okay? And that seems like a very costly thing to do, because I didn't explain earlier. But throwing an exception eats up a lot of computational cycles. And it's also very bad in terms of user experience, because it's basically breaking down the current flow of execution. Okay, so the user might just suddenly get thrown out in the middle of a competition, and your system might just go down right. So throwing, throwing an exception is something that you have to do very carefully. And yet, just because you try to remove something from an empty list, you're just throwing an exception. Right? So why do you think this would happen instead of just returning a null? And I'll let you guys think about it, and if you still can. Come up with an answer, then ask me about it tomorrow during the review session. But anyway, that's the official behavior of remove first, st and if F is not null, then we're going to call unlink 1st helper method by passing in F, the the 1st pointer. Okay. so unlink 1st is this guy here on the right hand side. First, st it's going to cache the item of the 1st node. So element now points at item, and also the next pointer of the 1st node. Okay? And then we're going to explicitly set those 2 components to null. You know, f dot item is null f dot. Next is null, and that step is not really necessary in terms of behavior. But we just want to help out the garbage collector so that these objects get reclaimed as soon as possible. and then we are going to set the 1st pointer to point at next, which is the right things to do, because if you are removing the 1st node, then the one coming after that must be the new 1st node. Right? So that's why we're having the 1st pointer point at the next node. and this last part might be a little bit confusing. But it's not. It's it's a very reasonable thing to do. So if next is null. Okay, so next is this guy. So f dot, next is null. That means we try to remove something from a list that contains only a single element. right? Because if the 1st next pointer is null, then there is no next, there's only the first.st So this list only contains a single item, and you just removed it. Okay? So now it should result in an empty list. And when it comes to linked list. if you want to indicate an empty list. Then you also have to set the last pointer to null. And this is what's happening inside of the if part. Now, the else part, which means that there are a couple of other elements left. So now, even after removing the 1st

element, you still have a non empty list. and in that case we're going to set next dot previous to null, because, remember, this is a doubly linked list, and the previous pointer of any node must point at the node that precedes it. Okay? But at this point next is the new head. So there shouldn't be anything before it. That's why we're setting the previous pointer to null. And that concludes the unlinked procedure. Don't forget to decrement the size by one, and make sure to return the element that you just removed. Okay? And that also gets removed from the remove 1st as well. So the overall time complexity of remove 1st is the same as add, first, st because well, it is making a constructor call to an exception constructor. But this itself is not related to the size of a linked list. Okay, so that's constant. With respect to the linked list. everything else will just be assignments, comparisons, and arithmetic operators. So they're all simple statements, and there's no loop involved. So this entire operation can be done in constant time. So we also know that remove 1st can be done very, very efficiently. Okay, so next up, we're going to look at, add last, which is an operation to add a new element at the end of the list. So this is the append operation that we learned from Arraylist. So just it's the same as the regular ad in array list. It's just that, you know. It has a special name at last. Okay. So the add last is, well, actually it. It has to go through the main method of add. Last, I believe I omitted that code here. But what really happens the most, the the method where the interesting thing happens is another helper method called link last. So this link last will try to remove the element or sorry the the the element pointed by E. Okay. all right. So the behavior of a link or sorry add last is almost the same as add first, st except that now, instead of having to work with the previous. Now you got to work with the next pointer, and also the last pointer, instead of the first.st So the the sort of the rough, animated version of that is to do this. Okay? So first, st you construct a new node containing the new data you want to insert, and then have the last pointer point at that newly constructed node. and at the same time the old lasts next must now point at the new node right? Because now this guy becomes the new last. So the previous, you know, you know, the old last must now precede this new last, and don't forget to have the previous pointer point at the old last like that. And that's how things are done. And this procedure is summarized in this code. Link last L variable L points at the last pointer, and we construct a new node by supplying the data. The previous node, which is the old last. and there won't be any next. No, because this is supposed to be the new last. So we're just passing in null and and then we're going to have the last pointer point at the new node here. So this is line 3, and if L is null, that means, you know the new last is null. Sorry, sir. The old last was null. Okay? And that means you're trying to add something into an empty list, in which case the newly created node should also be the 1st node. So that's why you're assigning first, st or having 1st point at the new node as well. Otherwise, if you've just appended it into an existing list, then make sure to connect this next pointer of the old last. so that it points at the new node and then increment the size. And we're done so. This operation is also very efficient, because, just like before, no loops, all simple statements. And this is a constant time operation. Right? So add first, st add last, remove first, st and. as we'll soon see, remove blast can all be done in constant time, and I'll just talk, and I'll talk about it right away. So remove last is not any different. Well, it's hardly different from remove first, st so it's a mirror opposite, like it's a symmetric version of remove. First, st if you understand, remove first, st then remove last is just a piece of cake. Okay. so let me just jump straight into unlink last helper method. And L now points at the last node. Okay, so we have the element that we're going to return@theendl.previous is the previous node of the old last. and we're going to garbage. Collect these items, and we'll have the last pointer point at the previous node, which is the new last node, because you're removing the last guy, the one before that should

become the new last guy. Now, if the previous is null, which means that you just removed the only existing item of the linked list. Then we should also set the 1st pointer to be null, because now the list is empty. otherwise we are going to set the next pointer of the previous to be null, because you just remove the old guy and the one before that will become the new node, the new last node. And so there shouldn't be any next for that. So we're nulling it out. Decrement the size and return the element. Okay? So like, I said before, this is all constant time, just like, remove 1st for the same reason. and if try to compare this against array lists. Corresponding versions. Right? And I'll get to that comparison chart at the end of the lecture. All right. Now we have the get 1st and get last, which means that you want to return the 1st item of the list. And the last item of the list. Okay. And this is really really easy, because we always maintain pointers to both 1st and last nodes. So get first.st We'll just simply get the 1st item. And if it's null, we're going to throw an exception again, why do we throw an exception instead of returning a null? So think about it. Otherwise we will just return the item of those 2 nodes. And the time complexities are obviously constant time for both cases. And now we're going to get to the more interesting part of the link list. So this is the typical insertion method. We have the same method in the array list. Right? We have the array list. Add that accepts an index and an object to insert same thing here. But the behavior is going to be the same, obviously. But the the implementation is going to be drastically different, because, unlike array list, we don't. We no longer have random access in the linked list class. Okay, we can still use an index to use to implement, get, or set, but that's only providing an illusion of random access. In reality, you actually have to travel from the very beginning all the way up to that index point costing linear time. So we don't have random access. So insertion is not going to be as well. It's not going to be the same as array lists. Add. but if you think about it, it's not gonna be any. Say, you know. more difficult than arraylists add, you know, it's just a matter of rearranging the pointers, as I always say. Okay. so consider this scenario. Suppose you have this existing link list. We have the first, st we have the last. And this is index, one right index, 0 index one. And let's suppose we want to insert a new data into index one. Okay, that means this guy should now become the new index one. Everything else will have to get shifted by one in terms of indexes. So what must happen first.st Well, we create a new node and have its previous pointer point at the node before and the next pointer point at the target index node, the current target index node. Right after that, old pointers have to be separate, or old connections or old links must be severed like that. Okay? And then also remember to establish the new pointers like that. Okay? And then we have the full insertion operation done. And of course you got to be careful when you're inserting into the the 1st or the last, because those are the boundary cases, and you might want to take extra care in terms of arranging the pointers, and and I'll talk about that in the code. But at least now you understand how this should behave in terms of assigning pointers. and let's take a look at the actual code all right on the left hand side. That's going to be our main entry point. That that's the the public. Add method. and if index equals size. which means that you want to insert at the very end of the link list. Okay, then, that is exactly the same as adding to the last of the index, or or last, of the link list. in which case we already have a very useful helper method, called linklast which we used for add last. So if we supply size as the index. and the behavior is just going to be exactly the same as add last. However. if that's not the case. And, by the way, at this point index is going to be non-negative, because the the usual index check is done here. Okay, so otherwise, we're going to call another helper method called Link before. And it's receiving 2 arguments. One is the argument or the element that you want to insert into. And the second one is the node at that very index. Okay. So you remember the node helper method from before right? This

is a 1 that travels from the beginning all the way to that index point and returns that node. Okay? So it's going to retrieve that node object at the given index. If you look at the link before signature, it's receiving E, the element and the successor Node, it's it's named successor, because the node at that index is about to become the successor of the node that we're about to insert. So bearing that in mind, let's take a look at the local variables. First, st we have the predecessor or pred. which is simply the previous, of the successor. Okay, so whichever successor that we have or the you know, the target. You know, Node at the index location, that guy's previous pointer is going to be the predecessor. So in this example case. Pred is now pointing at this node a okay? And then we're going to create a new node. So this node N with predecessor as its previous. So you know, this guy should become this guy's previous. That's obvious. We're supplying the data E and successor is going to be my, the new guys next, which makes sense because you want to insert this guy between these 2, A and B, okay. So at this point, the previous and next pointers of the new node have already been established correctly. Okay. all right. Next line successors previous, is about to become the new node or sorry successor. Previous is pointing at the new node. Okay? So, successor, meaning that this target location. this guy's previous should now point at me because I'm about to take its place. Okay, that makes sense. Now, one thing that we're checking here is if predecessor is null. Okay? So what is predecessor? Predecessor is the previous of the successors. So it, you know, in this scenario it's pointing at node. A. But what does it mean that the predecessor is null? Well, it simply means that the index was 0. Okay, you get it, because if it's if the index were 0, then we would have retrieved the the head node, the 1st node, and the 1st node does not have any previous pointer. Okay, so that's why the predecessor is going to be null. And that also means that we want to insert the new node into location in 0 index 0, and thereby making it the new first.st That is why we're going to have the 1st point at the new node in that case. Otherwise, if that is not the case. then we will simply let the predecessor's next point at the new node. So this next is now a valid object, a valid pointer. So this is guy. This is now pointing at the the new node. And okay, so that's what's happening in the else part. So that completes the stitching part. And now we've incremented the size by one, and we are done. Okay. Now, whenever I ask people to analyze the the time complexity of this add operation. There's a great deal of confusion. Okay. because I'm always asking about the add operation, this entire thing. But oftentimes people mistake that instruction to be applied to this particular method. Okay? So I'm I'm never going to ask you to to analyze the time complexity of the individual helper methods. Okay, I'm always going to ask you to analyze the time complexity of the entire operation. Okay, so if I ask you, what is the time complexity of this ad, then you should try to analyze this guy on the left hand side. Okay? And by the way, you still have to analyze the time complexity of this because this method is still being called inside of, add. okay. so that is also needed. But fortunately for us, this is very simple, right? All simple statements, just assignments and comparisons. So this is a constant time operation. What about Link last link last is the same as add last, and we all know that to take constant time. So this function call is constant time. This is comparison. Index check can also be done in constant time, regardless of the the link list size. so constant, constant, constant link before is also constant. So that's why a lot of people claim that insertion can also be done in constant time. But remember, we're also making another method call here. The behavior of the node method is to start at the beginning and iterate through the whole chain or linked structure. So this inevitably involves looping. Okay? And we all know that the node method takes linear time in the size of the linked list. So just because of that method only the whole operation is going to take linear time. So even though this stitching operation can be done very efficiently

in constant time we need to travel to that location to do that stitching. Okay. so that is the major bottleneck in causing this insertion method to become a linear time operation. So that was the animation. And now we're going to do the opposite. We're going to remove an element at a given index. And this is the main entry point. We have the index. And obviously we're going to check for the index. If we're given the wrong index, then some kind of exception is going to be thrown, and you should understand why that exception must be thrown as opposed to returning a null. But if the index is valid, then we're going to call another helper method called Unlink with providing the node at that given index. So we're going to follow the same logic as before. We're 1st going to locate the target node to remove. And then we're going to call the helper method unlink to perform the actual removal on that node. Okay, now, before we talk about the unlink operation, let me just go through the process of removing a node, which is quite simple. Okay? So you 1st sever these 2 next sorry, these 2 next pointers. Okay, so you sever these connections and then you sever those connections. Okay, we're not done yet, because we need to establish that kind of a skip connection. Okay? So once the new note or the target node is removed, then my previous is next shouldn't be my next and my next previous shouldn't be my previous. Okay, so this is quite a mouthful. But if you see this animation, that's exactly what it means. Okay. if I ask you to do this stitching manually, you know visually, then everyone could do it right. And that's exactly how things are done. So the code version of that is given in the unlink method. We 1st fetch the item of the target list. So this is going to be the one that gets returned at the end. We also get the next and the previous of the target node. And 1st thing we're going to check is if previous is null. Previous being null indicates that we wanted. We're trying to remove the head node. Okay? So that's why we're setting the 1st pointer point at the you know the current next? Okay? So if you're trying to remove the head node, that's fine, the the one after the the head note should become the new head. Okay? Otherwise, we're gonna set. The previous is next to become next. So it's this line here. So previous is next point at the you know current next. Okay, that's this one here. And we're gonna null out my target notes previous pointer for garbage collection. So that's this line. And the next thing we're gonna check is if next is null. So next, being null is the mirror case of previous being null, namely, you're trying to remove the last node. Okay? And and if that's the case, then it's pretty obvious where the new last is should be add, so the new last is now pointing at the previous. Okay makes sense. If that's not the case, then we're going to set the next previous to be previous. So next is previous pointing at my previous. So it's this line here. And we just achieved that. And then again, we're nulling out the next pointer of the target note for garbage collection. And then, finally, we ensure this to be done. You know the the item to be garbage collected by setting it to null, decorate the size and return element. Okay. So just like the insertion method, the one before this time. Complexity is also linear in the size of the linked list. Just because we have this node. Call the call to the node helper method. Okay, everything else. You can clearly see that this is these. This is a set of constant or simple statements. So this removal operation also needs to be done also requires linear time and the size of the list. All right. So the clear method. Now, if you don't really care about garbage collection, then this can be done very, very easily. Right? You can simply set the 1st and the last to be null, and that would be it. You can do it in constant time. But of course, if you, you know, just like array lists clear. If you want to facilitate garbage collection. Then you should actually go through the entire list following from start to end and zeroing out the pointers as you move on. But then, the way you do it is going to be a little different, because now, as you advance the pointers, you have to sever all the connections that you just looked at. Okay. So as you're moving through the nodes, you have to set the item to null, and the previous, and

next to null and go on to the next item, and so on. Okay, so I'll let you figure that out on your own. There are many ways to do this. so I strongly suggest that you try to implement this on your own, you know, at least on a pseudo code. So here's the chart that I was talking about comparing the complexity, the time complexity between the linked list and the ray list counterparts. Okay? So the 1st method we're going to look at is the insertion method, and we know for that for array list. This involves shifting a big chunk of the underlying array, so that costs linear time in the size of the list. and the link list will also require linear time, because we have to move to that location. Now, of course, once you've located that node. then removing or sorry, inserting a new node into that position. It can be done in constant time. but the sheer complexity of moving to that target location is going to cost linear time. Add, is the append operation. Okay? So if you're appending to array list, then that's going to be amortized. Constant time, fairly efficient most of the time. If you're appending to the link list that can always be be done in constant time. So this is the same as add last. if you want to remove something, something in the middle. right? Then link list will also require linear time. Same for array list. Okay, because you need to shift a chunk of the array to cover up the whole remove last can be done very efficiently in linked list in constant time. Same with array list, because at least the current scheme of the array list does not shrink the capacity. Okay, so removing the last can be done very easily. However, removing the 1st in an array list requires shifting the entire chunk to the left. so remove 1st an array list should take linear time. but in linked list that can be done in constant time. adding 1st into a link. List is also constant time. For the same reason remove 1st is constant time, but when it comes to ray list, adding 1st will always have to involve shifting the entire thing to the right. So that's going to involve copying n elements. And it's going to cost linear time. Get retrieving an an item at a certain index for a link list is gonna take linear time, because again, you have to travel to that location. whereas an array list, you have the benefit of random access. So you can just get that item right away. And this can be done in constant time, because we're operating on an array and arrays support a random access in constant time set for the same reason in linked list will take linear time. But and also constant time in array list. Okay? So it looks like we're making a lot of trade-offs right? Some of the methods are more important are, you know, more efficient in array list. And in some cases the link list implementation is more efficient. Okay, in particular, if you're operating on the front and the back, or the 1st and the last of the list. using an array linked list is much, much more efficient. and you might wonder when that's going to be useful. Why do we even care about the front and the back? Because we'll, you know, inserting into the middle is just as important. And we'll get to an example data structure next week after we come back from the the exam. Okay, so hold on to that. Try to think of a case where that could be useful. But Linkless does have its uses. Okay? And that is, that concludes the the list portion of the lecture. I also wanted to talk about. sorry. Talk about iterators. Okay, so let me bring it up real quick. It so iterators is supposed to have been the sort of a sec or a 3rd recitation. But you know I wasn't available here because of the other lecture that I had to take care of. But now that I have enough time, let me just quickly talk about it, and this will also be part of your midterm exam. Okay? So an iterator is not really tied to a linked list or an array list. In fact, it's not tied to a list based data structure at all. Okay, an iterator is a much more general mechanism that's applicable to any kind of data structure that you can think of. Okay. And as you can see from the name iterator. It iterates through the collection of your data structure. And this is an important concept which had been largely overlooked for the case of list-based data structures, because in the list based structures, we always had this notion of an index right? Even if linked list did not support random access. We still had indexes. So we were able to access that particular

position right? And there was a natural notion of ordering of the items. That's why they had indexes. But some data structures do not have indexes like a set data structure. For instance, it's simply a collection of stuff. Okay, the tree data structure that we're going to look at in a few weeks. That doesn't have any indexes either. Okay? And if we want to look inside of the data structure. how can we do it if we do not have any notion of indexes? Right? So that is, when the iterator comes into play. iterator allows us to look inside of the data structure in a sequential manner, even when this particular data structure does not have any concept of a sequence. this wasn't the case with list-based data structure, because we always we can easily access the items in in a pre-established order in in the index order. But even if the new data structure you have does not have any sequential information. You can still access them by using an iterator. Okay, so looking inside the data structure, that's what iterators are meant to do so in order to appreciate what the iterator does. We got to start at the Iterator interface. This is part of the Java's Standard library set, so it's always part of it. You can. You almost always have access to it. If you import the utils library package and forget about the remove part, for now we're not going to worry about that. But these 2 methods has next, and next, these are the most important ones. 2 most important ones has. Next returns a Boolean. and every time you call has next, it's going to check the current state of the iterator and return true. If there are items still left to be explored. Okay, so remember the iterator. You're going to use an iterator to enumerate all the items inside of any kind of data structure that you have right? And it's going to be done in a sequential manner. You're going to pick out one item at a time. Okay? And if you still have some unpicked item. then has next is going to return. True. so literally, there is the next item that you can retrieve. But if you've looked at all the items in your collection, then it's going to return false. And you know that you're done with the iteration. Okay? And this method called next is the the method that actually returns the next item in line. Okay. so the remove part, I said, you don't have to worry about it, for now you can just understand that to do the actual removal. But you know I'm not here to bother you with that kind of implementation. I'll I'm just trying to introduce you to the iterator interface and how to use it. Okay. so let's take a look at a particular sort of a version of an iterator. And it's called a list iterator. Okay? So a list iterator is also an interface that extends the usual iterator interface and the main difference between the iterator and list iterator is that it supports bidirectional iteration. Okay? So it has has. Next, it has next, that's common. But it also has has previous and previous. Okay? So those are the main differences, besides a couple of other utility methods. And if you try to use an iterator on a list based data structure, then this is the one that will most popularly used. But anyway, so let me 1st show you how to do an iteration. Okay, so let me quickly bring up some sorry. Bring up a test case. Sorry there must be something wrong here. Alright. Here's an example that I wanted to show you of how to use an array list. Sorry an iterator. Okay, so suppose you have a method that accepts an array list doesn't have to be an integer. But let's just call it an integer array list, and you want to look inside of it. You want to use a narrator. So now, obviously, you can use a for loop like that. and you can just do the usual printout like that right? That's sort of like standard procedure to kind of regurgitate all the contents of any kind of array list. But let's try to use it in in in an iterator context. And here. So you 1st declare in an instance of an iterator that is receiving the same kind of type as the array list type. So this is an integer, and you can call a dot iterator. Okay. now, a dot iterator is going to return an instance of some kind of an iterator that it has internally. And now that we have it, we can use it to look inside of the array list. Okay. so notice how I'm using has next. okay? So it has next. So while the iterator has next right? So it's gonna run as long as there's something to get out of. So as long as we have some unexplored items. Then, while loops going to run

over and over again. and next, we're going to look at how to get things out, and so you can just call simple. You simply call iterator dot next like that. And you can use it to print out stuff. And that's gonna print out the same sequence of numbers or data as it the 1st loop did. And I used an array list to demonstrate a point. But it doesn't have to be constrained to an array list. You can do any other data structure supported by implemented in Java. and you will still have access to this iterator mechanism. So let's see what happens inside of the iterator method. Okay? And you can see that when you call this, then it's gonna create an instance of a new class called Iter ITR. And return it. Okay, that's great. So what does this iter class look like? It's a private inner class. So it's a private inner class, which means that the outside cannot access this alright, but it is possible to create an instance of it and then return it to the outside that is allowed. And the reason one of the reason why it's allowed is that the the Itr class implements an iterator. and therefore it has has. Next, it also has next, okay. So I'll just talk about how as next, and next are implemented. Those are gonna follow some very simple rules. and they're all based on this notion of a cursor. So the the way this iterator maintains the sequence of items to look inside is by keeping track of this cursor. So the cursor is going to point at where this item, which next item should be returned. And cursor is basically an index. This is initialized as 0. And there's another and piece of information called Last Ret. And this is the index of the latest item that we, we return. And initially, it should be negative one, because we haven't returned anything initially. Okay. so what has next does is it simply compares cursor to size of the array list, and if it's the same as a size well, that means we just ran over the entire list. And the next item is, the is like is living in at index size. But remember, size is an invalid index. Okay, so we simply want to check if cursor is in the right range, and we just have to look at cursor not equal to size. And that's the very simple implementation of has. Next. later on, when we look at a tree. I'll show you an example of a more complex version of has next. Yeah, but it's fairly simple. Anyway. The next method the 1st call you don't have to worry about check for co modification. That's something unrelated to our class. It's a constant time method. Don't worry about it. I'm not going to ask you about it. But 1st thing to check is if cursor is greater than or equal to size. Okay? And if that's the case, we're going to throw an exception, because well, there's no way we can index into the outer range. So index being greater than or equal to size, means that it's an invalid index. So we're not going to check. We're not going to proceed any further. Okay? And again, why do we throw an exception instead of returning a null. Think about it. And the rest is fairly simple. Okay, element. Data is our usual element data. And we're we're again going to check. If I is greater than or equal to element data dot length. And if that is the case, then we're going to throw another exception. Okay, which is kind of weird. Because if you think about this. that's exactly the same kind of condition we checked up here. So why are we checking it again? Right? Looks like we're doing redundant stuff. But this is this is because we're we want to account for concurrent modification. Concurrent modification happens when there are multiple processes or multiple threads that tried to modify the same array list at the same time. Okay, so there's a possibility that something might have happened into the underlying array list at this point at at this, at the time we're executing this guy. Okay, so we just want to be extra careful. Oops. Sorry we just want to be extra careful that nothing else, nobody else has touched the array list in the meantime. Okay, so that's why you're doing the same check twice. But again, it's not something you have to to worry yourselves about, because the that is beyond the classes. Topic. Okay, but just understand this to be an extra cautious measure, so as not to cause an index out of bounds. Exception. Okay. but once that part is done, the rest is fairly simple. You simply increment the current index by one, assign it to the cursor. and then just retrieve the element data at that location

at location I and also don't forget to check or to set last return to I. And that concludes everything. So you can see how the next method will just continuously retrieve the next item in line over and over again until has next will finally return false, because at at some point a cursor is going to point at size, and and that's a signal to stop iterating. So this is the same code. Here I just skipped over. I I skipped remove part because that's a bit of a a bit of a hassle, but you know, if you must know, it just takes advantage of the remove method all right. And this is called. This is a list iter class, which is, you know, a subclass of iter. And remember, list, Iter is implementing list iterator which supports bidirectional iteration. Okay, and has previous and previous are almost the same, you know, not the same, but the mirror opposite of has next. And next. So I'm not gonna bother explaining this to you guys. But it's it's following the same principle. Basically. Okay. So again, sorry. I think that's the end of the slide. Let me just quickly show you how that can be done. So if you wanted to use an list, use a list. Iterator. Yeah, sure, you can do that. In fact, just do this a dot list iterator, and that's it. I mean, I didn't show you how to use. You know it, dot previous. But the is it previous? Hold on, I think. Oh, sorry I should have done this list. Iterator. Right? So it should be it dot previous. Okay, like that. If you wanted to look at the previous element. But, the way to use it is not any different from the regular iterator. Just make sure that you instantiate or or declare, and it as an instance of a list iterator. Okay. all right. So that's it for the lecture. I think that went fairly okay, much quicker than I thought. And so think about all the questions that I asked in the lecture. And please bring questions to tomorrow's or Wednesday's recitation, because that's going to be our review session for the midterm on Thursday. Okay, again, my sincerest apologies for missing the class today. I hope it doesn't affect you guys too much. but I'll try to make it up as much as possible, and also the missed office hours will be made up at some point. I'll I'll make an announcement. Okay. thank you all, and I'll see you on Wednesday.