ProgrammingAbstractions-Lecture13

Instructor (Julie Zelenski):Welcome to Friday. I keep telling you I'm gonna talk about linked list and then we don't ever get very far, but today really we are gonna talk about linked list and do some tracing and see some recursive work we can do with linked list and then I'll probably give you a little preview of the next topic, which is the introduction algorithm analysis in Big O toward the end. We won't get very far in that, but we will be talking about that all next week, which is Chapter 7 in its full glory. That will be pretty much everything we cover next week is coming out of Chapter 7, so it's all about algorithm and sorting and different algorithms. I will not be able to go – hang out with you today after class because I have an undergraduate counsel meeting. But it's a nice day so hopefully you'll have some more fun outdoor thing to do anyway. We will meet again next Friday. So put that on your calendar if you are the type who keeps a calendar. Anything administratively you want to ask? Stuff going on? How many people have started Boggle? Gotten right on it? Okay, there are two of you. Yay. All right. The other 50 of you, okay, okay. Now's a good time though. It is due a week from today so we gave you a little bit longer for Boggle because it's kind of recognition and, sort of, a bunch of big things that have to come together for that. But certainly not one of those things you wait for the last minute.

Just a reminder about another administrative things, which is the mid-term is coming up. So the mid-term actually is the Tuesday after Boggle comes in. The 19th, I believe, is the date of that and it's actually in the evening. If you are not available Tuesday to 7-9, that's actually when we're hoping to get almost all of you together, but if it really just doesn't work for you and there's no way you can accommodate it we will get you into a different time on Tuesday. You can send Jason an e-mail about that. I'll reiterate that next week just to remind, but just kind of right now if you want to keep your scheduling together if you can clear your evening of the 19th that actually is really good for us. Okay. So actually I'm gonna not even really work in the things. I'm gonna go back to my editing and talk about kind of where we were at at the end of Wednesday's lecture. Was we were doing a little bit of coding with the linked list itself. So having defined that recursive structure that has the information from one entry plus a pointer to the next and then so far the little pieces we have is something that we will print an individual entry, something that will create a new entry out in the heap, filling in with data that was read from the console typed in by the user, and then the last bit of code that we were looking at was this idea of building the list by getting a new entry.

So a pointer coming back from there that points to a new heap allocated structure that has the name and the address of someone and then attaching it to the front part of the list. So we talked about why that was the easiest thing to do and I'm gonna just re-draw this picture again because we're gonna need it as we go through and do stuff. Here's the main stack frame. It's got this pointer list, which is expected to point to an entry out in the heap. It starts pointing at null. We have this local variable down here, new one that is also a pointer to an entry and based on the result from get new entry, which will either return null to say there's no more entries or a new heap allocated thing if it gives us this new entry that says here's Jason and his phone number. That the lines there – the new ones

next equals list so new ones next field, right? Gets assigned the value of list, which affectively just copies a null on top of a null and then sets list to point to the same place that new 1 does. So on the first iteration, Jason is the new front most cell on the list, has a null terminator in the next field, which says there's no further cells. So we have a singleton list of just one cell. Now the subsequent iteration we call get new entry returns us a new pointer to another cell out here on the list. This one's gonna be Joel. He starts off as his own singleton list and then here's the attach again. New ones next field gets the value of list, so that changes Joel's next to point to where list does now. So I have two different ways to get to Jason's cell now.

Directly, it's the front most cell of the list still, but it's now following Joel. And then I update list to point to new one, so doing pointer assignment on this. Copying the pointers, making an alias, and now at the bottom of that loop, right? I now have list pointing to Joel, which points to Jason, which points to null. So my two entry list. Every subsequent cell that's created is prepended, right? Placed in the front most position of a list and then the remainder of the list trails behind it. So we should expect that if we put in the list A, B, C that if we were to go back and traverse it we'll discover it goes C, B, A. And so I was just about to show you that and then we ran out of time. So I'm gonna go ahead and finish that because I'm gonna write something that will print an entire linked list. It will take the list that we have and I'm gonna show you that. So the idea is to traverse a pointer down the list and print each one in turn. I'm gonna do that using a four loop. It may seem a little bit like an odd use of the four loop, but in fact what we're doing is really comparable to how you would do iteration down a link, an array, using kind of zero to n, you know, I++. We're doing the same thing, but instead of advancing an integer, which indexes into that, we're actually keeping track of a pointer, which moves down.

So the initial state of that pointer as we assign it to be the value of the list, so we alias to the first cell of the list. While that pointer points this invalid cell not to null, so as long as there are still cells to process then we'll print to cell. Then we'll advance the kind of equivalent of the I++. Here's the Kerr equals Kerr next. So advancing – so starting from Kerr equals Joel, right? To Kerr equals Jason and then Kerr equals null, which will terminate that. So as long as our linked list is properly terminated, right? We should print all the cells from front to back using this one loop. And so if I change this down here to be print list instead of just print to the front most entry and then I run a little test on this. So I enter Jake and his phone number, and I enter Carl and his phone number, and then I enter Ilia and his phone number, and I see that's all I have. Then they come back out in the opposite order, right? That Ilia, Carl, Jake because of the way I was placing them in the front of the list, right? Kind of effectively reversed, right? Them from the order they were inserted. Do you have a question?

Student: Do people ever make blank lists so that you can traverse backwards through them?

Instructor (Julie Zelenski):They certainly do. So the simplest form of the linked list, right? Has only these forward links, right? So each cell gets to the one that follows it in the list, but that is gonna create certain asymmetries in how you process that list. You're

always gonna start at the front and move your way to the back. Well, what happens if you're at a cell and you'd like to back up, right? It doesn't have the information in the single chain to do that. So you can build lists that either link in the other direction or link in both directions, right? To where you can, from a particular cell, find who proceeded it and who followed it. It just makes more pointers, right? And more complication. We'll see reasons why that actually becomes valuable. At this stage it turns out that it would just be complication that we wouldn't really be taking advantage of, but in a week or two we're gonna get to some place where that turns out to be a very, very useful addition to the list because it turns out that we really will need to be able to kind of move easily in both directions and right now we're not too worried about any direction other than the front ways. So this idea of the four loop, right? Is just one of the ways I could have done this. I could do this with a Y loop, right? Other sort of forms of iteration.

I could also capitalize on the recursive nature of the list, and this is partly why I've chosen to introduce this topic now, is because I really do want to stress that this idea that it's a recursive structure gives us a little bit of an insight into ways that operations on it might be able to take advantage of that recursive nature. So let me write this alternate print list, and I'll call it recursive print list, that is designed to use recursion to get the job done. So in terms of thinking about it recursively, you can think of, well, a linked list is the sequence of cells and iteration kind of thinks of the whole sequence at one time. Recursion tends to take the strategy of, well, I'm just gonna separate it into something small I can deal with right now and then some instance of the same problem, but in a slightly smaller simpler form. The easy way to do that with a linked list is imagine there's the front most cell. Which I could print by calling print entry, which prints a single entry. And then there is this recursive structure left behind, which is the remainder of the list and I can say, well, I can recursively print the list that follows. So print the front most cell, let recursion work its magic on what remains. Looks pretty good. Something really critical missing? Base case. Better have a base case here. And the simplest possible base case is list equals equals null. If list equals equals null then it won't have nothing to do, so I can just return or I can actually just do it like this. Say if list does not equal null, so it has some valid contents, something to look at, then we'll print the front most one and then let recursion take care of the rest. So I'll change my called print list to be a recursive print list. And I should see some [inaudible] results here. I say that Nathan is here, I say Jason is here, I Sara, Sara are you an H? I don't remember. Today it doesn't. Sara, Jason, Nathan coming out the other way. Okay. So I want to do something kind of funny with this recursive call. So it's not really in this case – both of these look equally simple to write.

We're gonna start to look at some things where this actually might buy us something. For example, lets imagine that what I wanted to do was print the list in the other order. So I don't have the list that links backwards. So if I really did want to print them with the last most element first, the way that they were added at the console, and the way I'm building the list, right? Is putting them in the other order. So, well, could I just take the list and print it back to front? I mean, it's simple enough to do that on a vector, if you had one, to work from the way back. Could I do it with the linked list? In the iterate formulation it's gonna actually get pretty messy because I'm gonna have to walk my way all the way

down to the end and then print the last one, and then I'm gonna have to walk my way down to the one that points to the last one and print it, and then I'm gonna have to walk my way down to the one that printed to the penultimate one and print it, but it does involve a lot of traversal and a lot of work. I can make the recursive print one do it with just one tiny change of the code. Take this line and put it there. So what I'm doing here is letting recursion do the work for me of saying, well, print everything that follows me in the list before you print this cell.

So if the list is A, B, C it says when you get to the A node it says okay, please print the things that follow me before you get back to me. Well, recursion being applied to the B, C list says, well, B says, well, hold onto B, print everything that follows me before you print B, when C gets there it says, well, print everything that follows me, which turns out to be nothing, which causes it to come back and say, well, okay, I'm done with the part that followed C, why don't we print C, and then print B, and then print A. So if I do this and print it through I put in A, B, C. Then they get printed out in the order I inserted them. They happen to be stored internally as C, B, A, but then I just do a change-a-roo was printing to print from the back of the list to the front of the list in the other order. But just the magic of recursion, right? A very simple little change, right? In terms of doing the work before the recursive call versus after gave me exactly what I wanted without any real craziness inserted in the code. We'll do a couple of other little things with you and I'm gonna do them recursively just for practice. If I wanted to count them, I wanted to know how many cells are in my list, then thinking about it recursively is a matter of saying, well, there's a cell in the front and then there's some count of the cells that follow it. If I add those together, that tells me how long this list is.

Having a base case there that says when I get to the completely empty list where the list is null then there are no more cells to count that returns my zero. So it will work its way all the way down to the bottom, find that trailing null that sentinels the end, and then kind of count on its way back out. Okay, there was one before that and then one before that and add those all up to tell me how many entries are in my address book. So I can do that here. I can say what is the count in my address book? And maybe I'll stop printing it because I'm – and so I can test it on a couple of things. Well, what if I put an empty cell in so I have a null? If I get one cell in there, right? Then I should have one and I can even do a couple more. A, B, C. Got three cells. I'll do another little one while I'm here. Is that when I'm done with the linked list, all those heap allocated cells, right? Are just out there clogging up my heap. So when I'm done with this list the appropriate thing to do is to deallocate it. I will note, actually, just to be clear though, that any memory that's allocated by main and kind of in the process of working the program that when you actually exit the program it automatically deallocated. So when you are completed with the program and you're exiting you can go around and be tidy and clean stuff up, but, in fact, there's not a lot of point to it.

The more important reason, right? To deallocation would be during the running of the program as you're playing games or monitoring things or doing the data. If you are not deallocating – if the program, especially if it's long running, will eventually have problems related to its kind of gargantuan memory size if it's not being careful about

releasing memory. When you're done deleting it manually or just letting the system take it down as part of the process there's not much advantage to. So if I'm gonna deallocate a list, if the list does not equal null there's something deallocate. I'm gonna do this wrong first and then we'll talk about why it's not what I want to do. That's good exercise remembering things. So I think, okay, well, what I need to do is delete the front most cell and then deallocate all those cells that follow it. So using recursion, right? To kind of divide the problem up. It's a matter of deleting the front cell and then saying, okay, whatever else needs to be done needs to be done to everything that remains. List dot next gives me a pointer to that smaller sub list to work on. As written, this does try to make the right delete calls, but it does something really dangerous in terms of use of free memory. Anybody want to help me out with that?

Student: After it deletes the first element it doesn't know where to find the rest of it.

Instructor (Julie Zelenski):That's exactly right. So this one said delete list. So if I think of it in terms of a picture here coming into the call list to some pointer to these nodes that are out here, A, B, and C, followed by null and I said delete list. So delete list says follow list to find that piece of memory out there in the heap and mark this cell as deleted, no longer in use, ready to be reclaimed. The very next line says deallocate list arrow next. And so that is actually using list, right? To point back into this piece of freed memory and try to read this number out of here. It may work in some situations. It may work long enough that you almost think it's actually correct and then some day cause some problem, right? Just in a different circumstances where this didn't succeed by accident. That I'm digging into that piece of freed memory and trying to access a field out of it, right? Which is not a reliable thing to do. C++ will let me do it, it doesn't complain about this. Ether it will compile time or run time in an obvious way, but it's just, sort of, a little bit of ticking time bomb to have that kind of code that's there. So there's two different ways I could fix this. One way would be to, before I do the delete is to just hold onto the pointer that I'm gonna need. So pull it out of the memory before we're done here. So I read it in there.

So the piece of memory that's behind it is still good. The delete actually just deleted that individual cell, so whatever is previously allocated with new, which was one entry structure, was what was deleted by that call. But what I was needing here was to keep track of something in that piece of memory before, right? I obliterated it, so that I can make a further use of it. The other alternative to how to fix this would be to just merely rearrange these two lines. Same kind of fix I did up above where go ahead and delete everything that follows so when I'm doing a list like this it would say, well, delete the D and C and only after all of that has been cleaned up come back and delete the one on the front. So it would actually delete from the rear forward. Work its way all the way down to the bottom, delete the last cell, then the next to last, and work its way out. Which would also be a completely correct way to solve this problem. Okay. Any questions on that one so far? Let me go back over here and see what I want to do with you next. So I talked about this, talked about these. Okay. Now, I'm gonna do something that's actually really pretty tricky that I want you guys to watch with me. All right. So this is the same code that we had for building the address book. The listhead, the new one, and so on.

And what I did was, I just did a little bit of code factoring. Where I took the steps that were designed to splice that new cell onto the front of the list and I looped them into their own function called prepend that takes two pointers. The pointer to the new entry and the pointer to the current first cell of the list and it's designed to wire them up by putting it in the front. So it looks like the code that was here, just moved up here, and then the variable names change because the parameters have slightly different names. The code as I'm showing it right here is buggy. It's almost, but not quite, what we want and I want to do this very carefully. Kind of trace through what's happening here, so you can watch with me what's happening in this process. So the variable is actually called listhead and this – I guess let me go ahead and do this. Okay. So let's imagine that we have gone through a couple of alliterations and we've got a good linked list kind of in place, so I have something to work with her. So let's imagine that listhead points to a cell A to B and then its got two cells, let's say, and then it's empty. Let's say I get a new one and this ones gonna be a J, let's say. Okay. So that's the state, let's say, that I'm coming into here and hit the prepend and I'm ready to take this J cell and put it on the front so that I have J, A, B on my list. Okay.

So let me do a little dotted line here that distinguishes my two stack rings. I'm gonna make this call to prepend and prepend has two variables, ENT and first, that were copied from the variables new one and listhead that came out of the build address book. Actually, it's not called main here. Let me call this build book. Okay. So I pass the value of new one as ENT. So what we get is a pointer to this same cell, a copy, and so this case, right? We copied the pointer, so whatever address was being held there is actually copied into here as this parameter. Similarly, listhead is copied into the second parameter, which is the pointer to the first cell. Okay. I want to do this in such a way that you can follow what's going on. So let's see if I can get this to work out. So it's pointing up there to first. Okay. So I've got two pointers to this cell A. One that came off the original listhead and one that came there from the copy in first. And then I've got two pointers to this J. One that came through the variable new one in the build an address book frame and one that was ENT in the prepend frame. Now, I'm gonna trace through the code of prepend. It says ent next field equals first. Okay. So ent's next field is right here. It gets the value of first. Well, first points to this cell. Okay.

So we go ahead and make the next field of J point to that cell. So that looks pretty good. That first line worked out just fine. It changed the cell J to point to what was previously the front of the list. So it looks like now it's the first cell that's followed by those. And then the next line I need to do is I need to update the listhead to point to J. So it says first equals ENT. So first gets the value of ENT. Well this just does pointer assignment. Sorry, I erased where it used to point to. And then I make it point to there, too. So at the bottom of prepend, if I were to use first as the pointer to the initial cell of the list it looks good. It points to J, which points to A, which points to B, which points to null. Everything looks fine here, right? At the end of prepend. But when I get back here to come back around on this loop; where is listhead pointing?

Student:

It’s pointing to A.

Instructor (Julie Zelenski):It’s pointing to A. Did anything happen to listhead in the process of this? No. That listhead points where it always pointed, which was whatever cell, right? Was previously pointed to. This attempt to pass it into prepend and have prepend update it didn’t stick. Prepend has two pointers that are copies of these pointers. So like other variables, ents, and strings and vectors and anything else we have, if we just pass the variable itself into a function call then that pass by value mechanism kicks in, which causes there to be two new variables. In this case, the ENT and the first variables, that are copies of the two variables listhead and new one that were present in build address book.

And the changes I make down here, if I really try to change ENT or change first, so I try to make ENT point somewhere new or make first point somewhere new, don’t stick. They change the copies, not the originals. This is tricky. This is very tricky because it’s entering that the first line was actually okay. That what the first line tried to do actually did have a persistent affect and the affect we wanted, which is it followed the ENT pointer to this shared location and changed its next field. So dereferencing that pointer and mucking around with the struct itself did have a persistent affect. It wasn’t another copy of the entry struck. There really is just this one entry struck that new one and ENT are both pointing to. So both of them, right? Are viewing the same piece of heap memory. Changes made out there at the structure, right? Are being perceived from both ends, but the pointers themselves – the fact that I had two different pointers that pointed to the same place, I changed one of my pointers, the one whose name is first, to point somewhere new. I didn’t change listhead by that action. That listhead and first were just two aliases of the same location, but they had no further relationship that is implied by that parameter passing there. How do you feel about that? Question?

Student:

And first and ent are gonna disappear now, right, afterwards?

Instructor (Julie Zelenski):No. First and ent would go away, but that just means their pointers would go. So when prepend goes away, right? This space gets deallocated, so it’s like this pointer goes away and all this space down here goes away. Then what we have is list still pointing to A and B and then new one, right? Is pointing to this J, which points to A, but, in fact, right? Is effectively gonna be orphaned by the next come around of the loop when we reassign new one. So that J didn’t get prepended. It got half of its attachment done. The outgoing attachment from J got done, but the incoming one didn’t stick.

This is pretty tricky to think about because it really requires really kind of carefully analyzing what the code is doing and not letting this idea of the pointer confuse you from what you all ready know about variables. If I pass – if you see a function call where I say binky of A and B, where A and B are integer values, if these are not by reference coming into binky then you know that when they come back A and B are what they were before.

If A was 10 and B was 20, they're still that. That the only way that binky can have some persistent affect on the values of A and B would be that they were being passed by reference.

If I change this piece of code, and it's just one tiny change I'm gonna make here, to add an ampersand on that first, then I'm gonna get what I wanted. So I want to draw my reference a little bit differently to remind myself about what it is. When I call prepend passing new one it got passed normally. New one is still just an ordinary pointer. So now I have two pointers where ENT and new one both point to that J. This one currently doesn't yet point up there. It's gonna in a minute, but I'll go ahead and put it back to its original state. Now, first is gonna be an alias for listhead, so that first is not going to be a copy of what listhead is. It's actually gonna be a reference back to here.

And a reference looks a lot like a pointer in the way that I draw it, but the difference is gonna be I'm gonna shade or, sort of, cross hatch this box to remind myself that first is attached a listhead and there's nothing you can do to change it. That first becomes a synonym for listhead. That in the context of the prepend function, any access to first is really just an access to listhead. So trying to read from it or write to it, you know, do you reference it, it all goes back to the original that this is just standing in for. That there actually is a relationship that's permanent from the time of that call that means that yeah, there really is now new variable first. First is actually just another name for an existing variable, this one of listhead.

So when I got through the ENT it gets the value – ENT's next gets the value of first. Well, first's value really is what listhead is. So that means it points up to A, so that part works as we were hoping before. Then when I make the change to first equals ENT, first is a synonym for listhead and that made listhead stop pointing to A and start pointing straight to J. And so this is still attached here. Let's do that. Let me see if I can just erase this part and make it a little clearer what's going on. So new one's pointing to J now, ENT is pointing to J, and listhead is pointing to J, and then this is still the reference pointing back to there. So now when prepend goes away, this extra pointer is removed, this reference is removed, all this space is reclaimed here in the stack, but listhead now really points to J, which points to A, which points to B as before.

So we wired in two pointers, right? One going out of the new cell into what was previously the first cell and then the first cell pointer now points to the new one rather than the original first cell. It was all a matter of this one little ampersand that makes the difference, right? Without that, right? Then we are only operating on a copy. We're changing our local copy of a pointer to point somewhere new; having no permanent affect on the actual state of it. So, actually, without that ampersand in there it turns out nothing ever gets added to the list. The listhead starts as null and stays null, but all these things will copy the value of listhead as a null, change it in the local context of the called function, but then listhead will stay null.

So, in fact, if I just ran this a bunch of times and I threw in a bunch of things without the ampersand I would find that all my cells just get discarded. I'd end up with an empty list

when I was done. All the orphaned out in the heap attached and not really recorded permanently. That's pretty tricky. Question?

Student: Why is the ampersand after this [inaudible]?

Instructor (Julie Zelenski):Because it is the – the ampersand applies to the – think of it more as applying to the name and it's saying this name is a reference to something of that type. So on the left-hand side is the type. What type of thing is it? Is it a string of anti-vector events or whatever? And then the ampersand can go between the type and the name to say and it is a reference to an existing variable of that type as opposed to a copy of a variable of that type. So without the ampersand it's always a copy. With the ampersand, you're saying I'm introducing first as a synonym for an existing entry star variable somewhere else and that in the context of this function access to this named parameter really is reaching out to change a variable stored elsewhere. It's quite, quite tricky to kind of get your head around what's going on here.

I'm gonna use this, actually, in running another piece of code, so I want to be sure that at this point everybody feels at least reasonably okay with what I just showed you. Question?

Student: What happens if we stop ampersand [inaudible]?

Instructor (Julie Zelenski):Jordan, you want to invert them or something like that?

Student: Yes.

Instructor (Julie Zelenski):It turns out that that won't work. Basically, it won't compile because in this case you can't take a pointer out to an entry reference. In fact, it just won't let you. So it doesn't really make sense is the truth. But think of it as like the – sometimes people actually will draw it with the ampersand attached even without the space onto the variable name. It's just a notation. A really good clue that this name, right? Is a reference to an existing variable and that its type is kind of over here. So type, name, and then name as a reference has that ampersand attached. Question?

Student: Is there a wrong side to passing both the pointers by reference?

Instructor (Julie Zelenski):Nope, not really. If I pass it by reference, right? Then it – nothing would change about what really happens, right? It would still leach into the thing and copy it. There is – for variables that are large, sometimes we pass them by reference just to avoid the overhead of a copy. Since a pointer is pretty small, that making a copy versus taking a reference it turns out doesn't really save you anything and actually it makes a little bit more work to access it because it has to kind of go one level out to get it. But effectively no.

In general though, I would say that passing things by reference, as a habit, is probably not one you want to assume for other kinds of variables. Just because that once you've passed

it by reference you've opened up the access to where if it changes it it does persistent and if you didn't intend for that that could be kind of a mystical thing to try to track down. So I think you should be quite deliberate about – I plan on changing this and I want to be sure I see that persistent effect. That I need that pass by reference there. Okay.

So let me go on to kind of solve a problem with the linked list that helps to motivate what a linked list is good for. Why would you use a linked list versus a vector or an array style access for something that was a collection? And so the built in array, which is what vector is built on, so they have the same characteristics in this respect. They store elements in contiguous memory. You allocate an array of ten elements. There's a block that is ten integers wide, let's say, where they were all sitting next to each other. In address 1,000 as one, 1,004 is the next, 1,008 and so on. What that gives you is fast direct access by index. You can ask for the third member, the seventh member, the 28th member, the 6 millionth member depending on how big it is, right?

By, sort of, directly computing where it will be in memory. You know where it starts and you know how big each element is. Then you can say here's where I can find the 10th element. That's an advantage, right? To the array vector style of arrangement. However, the fact that it's in this big contiguous block is actually very bulky. It means that if you were to want to put a new element in the front of an array or vector that has 1,000 elements you have to move over the 1,000 elements to make space. If it's starting to address 1,000 and you have them all laid out and you want to put a new one in the front, everybody's gotta get picked up and moved over a slot to make space. So imagine you're all sitting across a lecture hall and I decided I wanted to put a new person there, everybody gets up and moves one chair to the left, but there's not a way to just stick a new chair on one end of it using the array style of layout. Same for removing, right? So any kind of access to where you want to take somebody out of the array and close over the gap or insert in the middle and the beginning and whatnot requires everybody else moving around in memory, which gets expensive. Especially for large things, right? That's a lot of work. It also can't easily grow and shrink.

You allocate an array, your vector, to be a certain size underneath it. That's what vector's doing on your behalf. If you have ten elements and now you need to put in ten more what you need to do is go allocate a piece of memory that's twice as big, copy over the ten you have, and now have bigger space. If you need to shrink it down you'll have to make a smaller piece, copy it down. There's not a way to take a piece of memory and just kind of in place in memory where it is kind of extended or shrink it in the C++ language. So that the process of growing and shrinking, right? Requires this extra effort. So behind the scenes that's what vector is doing. When you keep adding to a vector, eventually it will fill to capacity. It will internally make more space and copy things over and you're paying that cost behind the scenes when you're doing a lot of additions and removals from a vector. The linked list, right? Uses this wiring them together using pointers, so it has a lot of flexibility in it that the array does not. Each element individually allocated. So that means you can pick and choose when and where you're ready to take some more space on and when you don't. So when you're ready to add a new person to the vector there's no worry about if there's a million elements there and now you don't have any

space you have to copy them. It's like you don't – you can leave the other million elements alone. You just take the heap, ask for one new element, and then splice it in. So the only allocation of the allocation concerns the individual element you're trying to do something with. You want to take one out of the middle; you just need to delete that one and close around the gap by wiring the pointers around it. So all the insert and remove actions, right? Are only a matter of wiring pointers. Wiring around something you're taking out, attaching one to the end or into the middle is splicing a pointer in, and is splicing the pointer out. So basically you typically have two pointers you need to reassign to do that kind of adjustment. If the element has ten members, it has a million members, right? Same amount of work to put a cell in there. No dependency on how big it is, right? Causing those operations to bog down.

The downside then is exactly the, sort of, opposite of where it came out in the array in the vector, which is you don't have direct access to the 10th element, to the 15th element, to the 260th element. If I want to see what that 260 of my linked list is I've got to start at the beginning and walk. I go next, next, next, next, next, next 260 times and then I will get to the 260th element. So I don't have that easy access to say right here at this spot because we don't know. They're all over the place in memory. Each linked list cell is allocated individually and the only way to get to them is to follow those links. Often that's not as bad a disadvantage as it sounds because typically, right? That you're doing things like walking down the array or the vector to look at each of the elements at the end you would also be walking down the linked list while you did it isn't really that bad. So in the case where you happen to be storing stuff in an index and you really want to reach back in there is where the linked list starts to be a bad call.

Student:

Can you take it on, like, from the list and find something?

Instructor (Julie Zelenski):Well, it can take – the thing is it – with a question with me like a long time. Like if I were looking through to see if there was an existence of a particular score in a vector of integers I have to look at them all from the beginning to the end. Now, the way I can access them is subzero, subone, subtwo, right? But if I'm walking down a linked list I'm doing the same sort of work. I have to look at each element but the way I get to them is by traversing these pointers. It might be that that's a little bit more expensive relative to the vector, but they're still gonna be about the same. If there's a million elements they're all gonna look at a million elements and whether it looked at them in contiguous memory or looked at them by jumping around it ends up being in the end kind of a very comparable amount of time spent doing those things.

What would be tricky is if I knew I wanted to get to the 100th element. Like I wasn't interested in looking at the other 99 that preceded it. I really just wanted to go straight to the thing at slot 100. The array gives me that access immediately. Just doing a little calculation it says it's here. Go to this place in memory. And the linked list would require this walking of 100 steps to get there. And so if there are situations, right? Where one of these is just clearly the better answer because you know you're gonna do that operation a

lot, right? You're gonna definitely prefer this. If you know you're gonna do a lot of inserting and rearranging within the list, the linked list may buy you the ease of flexibility of a kind of rewiring that does not require all this shuffling to move everything around. Question here?

Student: But if you were to know the size of the linked list?

Instructor (Julie Zelenski):Typically a linked list won't limit the size unless you tell it, but that's actually true about an array too, right? Which is you – the vector happens to it on your behalf, but underneath it, right? That the array is tracking how many elements are in there, so would a linked list. So you could do a manual count when you needed, which would be expensive, or you could just track along with that outer most pointer. Well, how many elements have I put in there? And then update it as you add and remove. So you'd likely want to cache that if it was important to you to know that. If a lot of your operations depended on knowing that size you'd probably want to keep it stored somewhere. Over here?

Student: Why does the vector use arrays then instead of linked list?

Instructor (Julie Zelenski):That is a great question. It's one that we actually will talk about kind of in about a week. It's like, well, why? So sometimes array – the vector is kind of taking the path of, well, it's just presenting you to the array in a slightly more convenient form. We will see situations where then the vector is just as bad as a choice as an array and then we'll see, well, why we might prefer to use a linked list and we will have – we will build linked lists instead basically. It has to make a choice and it turns out that for most usages the array is a pretty good match for a lot of ordinary tasks. The linked list happens to be a match for other tasks, right? And then we will build stuff out of that when that context comes up.

Student: Well, if you just, like, then you just, sort of, like in a vector class, like encapsulate all that and I – its final functionality would be the same.

Instructor (Julie Zelenski):You certainly could. So you could totally build a vector class that actually was backed by a linked list and we will see how to do that, right? And then it would have different characteristics in terms of what operations were efficient and which ones were inefficient relative to do an array backed one. And over here was also. Yeah?

Student: Well, I, just to – for using linked lists, like, so first in, first out or first in, last out seems to make really good sense, but are there – am I like missing –

Instructor (Julie Zelenski):You are way ahead of us, but, yeah, you're right on. The things like the stack and the Q, right? Seem like they're gonna be natural things that will work very, very well with a linked list. Things that required a lot of shuffling in the middle, right? It's a little unclear because you have to find the place in the middle, which is expensive, but then the insert is fast. The question is, well, which of these is actually gonna work out better? They both have some things they can do well and some things

they can't do well. So it will have to have know maybe a little bit more about the context to be sure which ones seem like the best solution and maybe in the end it's just a coin toss, right?

Some parts will be fast and some parts could be slow and anyway I can get the inverted form of that as an alternative that doesn't dominate in any clear sense. It's just like, well, they both have advantages and disadvantages. Question back here?

Student: Why was that infused to be based on a vector instead of a linked list?

Instructor (Julie Zelenski):So they – I think your question is kind of like, yeah, they likely are built on linked lists. We're gonna see how we can do it both ways, right? We'll see how we can do it on array, we'll see how we can do it on a linked list, and then we'll talk about what those tradeoffs look like. Well, which one ends up being the better call when we're there? So that will be about a week. So don't worry too much about it now, but when we get there we'll definitely see kind of both variants and then we'll talk about them in great detail. Okay.

So I'm gonna show you, just the last operation I want to show you on a linked list is doing an insert in sorted order and this is gonna require some pretty careful understanding of what's going on with the pointers and then we're also gonna see a recursive form of it that actually is a really clever and very elegant way to solve the same problem. So I'm gonna first do it iteratively just to kind of talk about. What's the mechanism for doing stuff iteratively on a linked list? It gets a little bit messy and so just be prepared for that. That I'm planning on taking my address book and building it up inserted order. So instead of doing a prepend where I just take each cell and put it on the front I'm actually assume that I'm gonna keep them all in order by name and whenever I get a new cell I want to put it in the right position relative to the other neighbors in the list that are all ready there.

So this is one of those places where it seems like the linked list is actually gonna do very well. I have to search to find the spot. Okay? Well, you're gonna have to search to find the spot in a vector, too. Once I find the spot it should be very easy to just wire it in without shuffling and rearranging anything around. It should be a pointer in and a pointer out. So I'm gonna start this insert sorted. It's gonna take the head of the list, right? As a bi-reference parameter because there are situations, right? Where we're gonna be changing where the list points to when the cells in the first one and then the new cell that's coming in is the second parameter. So we're gonna search to find the place where it goes.

So let's watch this go down it's loop. I do a four loop here that's like, well, Kerr starts this list while it's not null, advance by this, and if the cell that I'm trying to place, which in this case is the cell Rain, proceeds the one we're currently looking at, then this must be the place that goes in the list. So I look at Rain, I compare it to Cullaf and if Rain goes in front of Cullaf then this is where Rain should go in the list. Well, it doesn't actually proceed it, so I advance again. Then I look at Cullaf – Rain versus Lowery. Does Rain

proceed Lowery? It does not. So I just keep going. So I'm gonna break out of the loop as soon as I get to the place I compare Rain to Matt. Doesn't come out. I compare Rain to Thomas.

Now, this is the first time when new ones name came up as being less than Kerr's name. So that says, okay, well, Rain needs to go in front of Thomas. Okay. All is well and good almost. I have a pointer to Thomas. That's where my curve points to, right? And I want to put Rain kind of in the list right in front of Thomas. How do I get from Thomas to Matt? But it's not just enough to know Thomas, right? Because Thomas tells me like what's gonna follow Rain. So this is the cell where Rain's next is gonna point to where Thomas is because Rain is kind of gonna replace Thomas in the list and push Thomas one element down, but the situation is that I also need to know what proceeds Thomas. The cell that led into Thomas is the one whose next field needs to get reassigned to point to Rain.

So the way the loop is going, right? It has to go kind of one too far to know that it was there because, like, if I look at Matt I can say, well, does Rain go after Matt? Yes, it does. Right? But that's not good enough to know that it goes right after Matt. The only way I can tell if it goes right after Matt is to say, well, if the cell that comes up next, right? Is behind Rain is that it goes in between these two.

Student:[Inaudible] Thomas and then [inaudible] Thomas?

Instructor (Julie Zelenski):Yes. So at least I can do it by maybe kind of overwriting Thomas with Rain and then overwriting Rain with Thomas and kind of wiring them up. I'm gonna try to avoid that. I'm gonna actually say, well, let's just think about can we do it with leaving the information in the cells where it is? I don't know who else might point to Thomas, right? It would seem kind of weird if all of a sudden Thomas turned into Rain in some other way. That I don't know for sure who else is using this pointer. So lets assume that I can't do that variation of it, but I'd like to put Rain in between Matt and Thomas and the pointer I have is not quite good enough.

The cells as is, right? Have this asymmetry. They know what follows or not. What precedes them. So, in fact, what I'm gonna change here is I'm going to run two pointers down the list, kind of in parallel, where they're always one step apart. Then I'm gonna track what was the one I last looked at and what is the one I'm currently looking at and I call this dragging a previous pointer or trailing a previous pointer behind. So each time I'm about to advance Kerr equals Kerr next, the thing I'll do right before it is to assign previous to be what Kerr is now. So then it'll advance to the next one, test against null, and keep going. So at any given state, right? Previous is exactly one behind where Kerr is and there's one special case, right? Which is that at the very first time through the loop, right? Kerr is set to be the list. Well, what's the previous of the head of the list? There actually is no previous, so actually initialize previous to be null.

So the first time through, right? We've assigned previous to null and Kerr gets to go to the first one and then we say, well, is Cullaf – does Rain come before Cullaf? No. And so I'll advance both so move previous up to where Kerr is and then move Kerr up to the next

one. So now they're pointing at Cullaf and Lowery together and, again, I'm comparing Lowery to Rain. Does Rain go in front of Lowery? No. So then I move up previous and move up Kerr. Does Rain go in front of Matt? No. I advance them both, right? And then I have previous at the Matt entry, Kerr at the Thomas entry, and then Rain does go right in between those, right?

We know that it didn't precede Matt because we all ready made it through the loop. We know it does precede Thomas and that tells us it goes exactly in between them, right? That Matt should lead into Rain and Rain should lead into Thomas and that will have spliced it in the list at the right position relative to the other sorted entries. So I give myself just a little note here. Well, what are the possible values for previous? If the cell is gonna go anywhere in the list other than the very front, so as long as Rain follows at least one cell on the list, it's not the alphabetically first cell of all the ones I've seen, then previous will be some valid cell and potentially Kerr could be null, but there is a non-null prev in all of those cases.

There's also the possibility though that previous is null in the case where the new cell was the alphabetically first. So if that one actually began with an A, let's say it was Alena. Then the first test would be if Alena is less then Cullaf. It is. It would immediately break and so previous would be null, Kerr would point to the front most cell, and I've got a little bit of a special case there where inserting at the front of the list isn't quite the same thing as inserting further down. So let me look at the code that I added to this and I ran out of room to draw pictures, so we'll have to do a little bit on the board here.

That there are two pointers we need to assign for that new cell that's coming in. One is it's outgoing one and then another is it's incoming one. And so I have this list and, let's say, this new cell is going here. So this is like K, L, M, T, R. That the first line splices in the outgoing pointer in all cases. It says that the new ones next field is Kerr. So Kerr is the first cell, right? That we found that follows the cell we're trying to insert. It might be that Kerr is null, but that's totally fine too. In this case it happened to point – this is where Kerr was and this is where previous was. That it will make R point to T, so that is the outgoing pointer for the new cell and now there are two cases here.

The ordinary case is that there is some previous cell. In which case the previous cell's next field needs to get updated to point to the new one. So we use the pointer to this one. It's next field no longer points to T, but instead points to R. And so I inserted R in between M and T with that adjustment. In the case where the new cell, let's say, was the A, then Kerr would be here and previous would be null. I will set A as next field to the Kerr, which spliced the outgoing pointer of A to attach to the rest of the list. But here's the special case. There is no previous next field that's gonna point to A. A actually needs to get reassigned to be the front most cell of the list. So it looks like our prepend operation where I'm saying list equals new one and the list is some reference parameter to some list elsewhere that then gets permanently set to the new cell.

So I would call insert sorted from, like, the build address book function to attach it to the very front in this case. This is very common in linked list code, right? Is to have there be

a bit of a special case about either the first cell or the last cell or on the empty list, right? But that there is a large – all the cells in the interior kind of behave the same. They have a cell in front of them and a cell in back of them and that insert has certain properties, but there's a little bit of an oddness with that maybe that first cell is sometimes the very last cell that requires a little special case handling. In this case, right? Putting A in the front means somebody doesn't point into A. It's the listhead that points to A and so it's not somebody's next field that we're updating. It really is the listhead pointer itself. Questions about that one?

So I need to show you the recursive one. I'm not gonna be able to talk about it very much, but I'm gonna encourage you to take a look at it. Which is the same activity of inserting a cell into a list that's being kept in sorted order. Now, not using this iterative, not using this dragging of our previous pointer, but instead using a strong dense recursive powerful formulation here that basically has the idea that says, well, given a list that I'm inserting into, let's say it's the M through Z part of the list. I'm trying to insert R is I compare R to the front of the list and I say, well, does that become the new front of this list? That's what my base case is looking at.

Is the list empty or does this one go in front of it? If so, prepend it. Otherwise just insert it in the remainder of this list. So if it doesn't insert at the front of M then we pass the N or the O whatever is following it and have it kind of recursively insert into the remainder of the list. Eventually it will hit the space case when it either gets to the end of the list or when there is the cell we're trying to insert belongs in front of these remaining cells. In which case, we do a prepend right there and stop the recursion. A dense little piece of code. One I'll encourage you to kind of trace a little bit on your own to kind of get an idea of how it's working, but it kind of shows in this case that the recursion is really buying you something by making it, I think, a much simpler and more direct way to express what you want to do and then kind of the more mechanical means of doing it iteratively.

What we will talk about on Monday will be algorithms. We'll talk about algorithms, Chapter 7. Start talking about Big O and formal announcements algorithms and do a little bit of sorting. So I will see you on Monday and have a good weekend.

[End of Audio]

Duration: 52 minutes