ProgrammingAbstractions-Lecture21

Instructor (Julie Zelenski):Hey there! Oh, we're back. We're back and now we're getting down and dirty. We'll do more pointers, and link list, and implementations. It's actually going to be like our life for the next week and half. There's just even more implementation, trying to think about how those things work on the backside. So I will do another alternate implementation of the stack today. So we did one based on vector and we're going to do one based on a link list. And then I'm going to do a cube based on link list, as well, to kind of just mix and match a little bit. And then, we're still at the end of the case study, which the material is covered in Chapter 9. I probably won't get thorough all of that today. What we don't finish today we'll be talking about on Friday, and then we'll go on to start talking about map implementation and look at binary search trees. How many people started [inaudible]? How many of you got somewhere feeling good? Not just yet. One thing I will tell you is that there are two implementations you're working on and I listed them in the order of the one that the material we've seen the most, right. We've talked about link list more than we have talked about things that are tree and heat based. So I listed that one first. But, in fact, actually, in terms of difficulty of coding, I think, the second one is little bit more attractable than the first.

So there's no reason you can't do them in either order if it feels good to you to start on the second one before you come back to the first one. Or just trade off between working on them if one of them is giving you fits just look at the other one to clear your head, so just a thought about how to approach it. Okay. So I am going to do some coding. I'm going to come back to my slides in a little bit here. But to pick up where we left off last time, was we had written a vector based implementation of the stack attraction and so looking at the main operations here for a stack or push and pop array that its mechanism here is just to add to the end of the vector that we've got. So I'm letting the vector handle all the growing, and resizing, and memory, and manipulations that are needed for that. And then, when asked to pop one, we just take the one that was last most added, which will be at the far end of the array. And so given the way vector behaves, knowing that we know it's a continuous array based backing behind it, then we're getting this O of one access to that last element to either add a new one down there or to retrieve one and remove it from the end so that push and pop operations will both being in constant time no matter how many elements are in the stack, 10, 15, 6 million, the little bit of work we're doing is affecting only the far end of the array and not causing the rest of the configured storage to be jostled whatsoever so pretty easy to get up and running.

So in general, once you have one abstraction you can start leveraging it into building other attractions and the vector manages a lot of the things that any array based implementation would need. So rather than using a raw array, there's very little reason to kind of, you know, ditch vector and go the straight raw array in this case because all it basically does it open up opportunities for error and kind of management and the kind of efficiency you can gain back by removing vectors intermediate stuff is really not very profound so not worth, probably, taking on. What we do is consider the only link limitation. We had talked about a link list for a vector and we kind of discarded it as likely not to lead anywhere interesting. I actually, am going to fully pursue it for this

stack because it actually turns out that a link list is a good fit for what the behavior that a stack needs in terms of the manipulations and flexibility. So if I have a stack of enterers where I push 10, 20, and 30. I'm going to do that same diagram we did for the vector one as I did for the stack is one possibility is that I put them in the list in this order. Another possibility is that I put them in, in this order. We'll call this Strategy A; we'll call this Strategy B. They seem, you know, fairly symmetric, right. If they're going to be a strong reason that leaves us to prefer Strategy A over B, or are they just equally easy to get the things done that we need to do? Anybody want to make an argument for me. Help me out.

Student: Last in, first out.

Instructor (Julie Zelenski):Last in, first out.

Student:[Inaudible].

Instructor (Julie Zelenski):Be careful here. So 10, 20, 30 means the way the stack works is like this, 10, 20, 30 and so it's at the top that we're interested in, right?

Student: Um hm.

Instructor (Julie Zelenski):That the top is where activity is taking place.

Student: Oh.

Instructor (Julie Zelenski):So we want to be adding things and losing from the top. So do we want to put the top at the end of our list or at the front of our list? Which is the easier to access in the link list design?

Student:[Inaudible].

Instructor (Julie Zelenski):The front, right? So in the array format we prefer this, putting the top of the vector on the far end because then there is the only jostling of that. But needing access to the top here in the front is actually going to be the better way to get the link list strategy you work in.

Let me actually go through both the push and the pop to convince you why this is true. So if I have access in the 10, 20, 30 case I could actually keep a separate additional pointer, one to the front and one to the back. So it could actually become maintaining at all times as pointing to the back. If I did that, right, then subsequent push up to a 40, it's pretty easy to tack on a 40. So it turns out it would be that the adding it's the push operation that actually isn't really a problem on this Strategy A. We can still kind of easily, and over time, if we have that tail pointer just tack one on to the end.

As for the pop operation, if I have to pop that 30, I need to adjust that tail pointer and move it back a cell. And that's where we start to get into trouble. If I have a pointer to the

tail cell there is no mechanism in the singly linked list to back up. If I say it's time to pop 30, then I would actually need to delete the 30, get the value out, but then I need to update this tail pointer to point to 20, and 30 doesn't know where 20 is, 30's oblivious about these things. And so the only way to find 20 in singly linked list would be go back to the very beginning, walk your way down, and find the one that pointed to the cell you just took out of the list and that's going to be an operation we want to avoid. In the Strategy B case, right, adding a cell, this means putting a 40 in allocating a new cell and then attaching it, splicing it in, right between wiring in two pointers, and we can do these links in constant time, and then popping one is taking that front row cell off. Easy to get to, easy to update, and splice out so no traversal, no extra pointers needed. The idea that there are, you know, 100 or 1,000 elements following the one that's on top is irrelevant. It doesn't have any impact, whatsoever, on the running time to access the front, or stuff another front, or taking it on or off the front. So it is Strategy B that gives us the best setup when it's in the back of the linked list.

So let me go ahead and do it. I think it's kind of good to see. Oh, like, what is it like to just write code and make stuff work. And so I'm kind of fond of this and you can tell me. I make a cell C it has an L type value and it has a soft T next right there. And then I have a pointer to the front linked list cell. So in addition, you know, it might that in order to make something like size operate efficiently, I might also want to catch the number of cells that update, or added or renew. I can also decide to just leave it out for now. Maybe I'll actually even change this to be the easier function, right, which is empty form. That way I don't even have to go down that road for now. Go back over here to this side and I've got to make everything now consistent with what's going on there. Okay. Let's start off with our constructor, which is a good place to make sure you get into a valid state. In this case, we're going to use the empty list so the head pointer points and tells us we have no cells whatsoever in the list. So we don't pre allocate anything, we don't get ready. What's kind of nice with this linked list is to allocate on demand each individual cell that's asked to be added to the list. The delete operation actually does show we need to do some work. I'm actually not going to implement it but I'm going to mention here that I would need to delete the entire list, which would involve, either iterates or recursion my way down to kind of do all the work. And then I changed this from size to iterate. I'd like to know if my list is empty. So I can just test for head is equal, equal to null. If it's null we have no list.

Let me work on push before I work on pop. I'll turn them around. The push operation is, make myself a new cell. All right, let's go through the steps for that. New cell equals new cell T. [Inaudible] the size to hold a value and an X link. I'm going to set the new styles val to be the perimeter that came in. And now, I'm going to splice this guy onto the front of my list. So the outgoing pointer I'm doing first to the new cell is allocated and leads to what was previously the front row cell. And then it is now updated to point to this one. We've got pointer wired, one value assigned, and we've got a new cell on the front. The inverse of that, taking something off my list, I'm going to need to kind of detach that front row cell, delete its memory, get its value, things like that. So the error checking is still the same at the very beginning. Like, if I've got an empty staff I want to be sure I don't just kind of do reference no and get into some bad situations. So I report that error

back using error, which will halt the program there. Otherwise, right, the element on top is the one that's in the value field of the head cell. So knowing that head's null is a safety reference to do there. And then I'm going to go ahead and get a hold of what that thing is and I'm going update the head to point to the cell after it. So splicing it out and then deleting the old cell. Just take a look at these. Once we start doing pointers, right, there's just opportunities for error. I feel kind of good about what I did here but I am going to just do a little bit of tracing to kind of confirm for myself that the most likely situations that get you in trouble with link list is you'll handle the mainstream case but somehow forget one of the edge cases. Such as, what if the list was totally empty or just had a single cell? Is there some special case handling that needs to be dealt with? So if I think about it in terms of the push case, you know, I might say, well, if there's an existing set of cells, so we're using Strategy B here, let me erase A so I know what I'm looking at, that its head is currently pointing to a set of cells, right, that my allocation of the new cell out here, right, assigning its value, assigning its next field to be where head points two, and then sending head to this new guy, it looks to be good. If I also do that tracing on the – what if the list that we were looking at was just null to begin with. So there were no cells in there. This is the very first cell. It isn't going to have any kind of trouble stumbling over this case. So I would allocate that new cell 40, assign its value. Set the next field to be what head is, so that just basically sets the trail coming out of 40 to be null and then updates the head there. So we've produced a single linked list, right, with one cell. It looks like we're doing okay on the push behaviors.

Now, the pop behavior, in this sort of case may be relevant to think about, if it's totally empty. All right, we come in and we've got an empty cell, then the empty test should cause it to error and get down to the rest of the code. Let's say that B points to some sequence of things, 30, 20, 10, that taking the head value off and kind of writing it down somewhere, saying, okay, 30 was the top value. The old cell is currently the head so we're keeping a temporary on this thing. And then head equals head error next, which splices out the cell around so we no longer have a pointer into this 30. The only place we can get back to it is with this temporary variable we assigned right here of old and then we delete that old to reclaim that storage and then our list now is the values 20 and 10, which followed it later in the list. To, also, do a little check what if it was the very last cell in the list that we're trying to pop? Does that have any special handling that we have overlooked if we just have a ten here with a null after it? I'll go through the process of writing down the ten. Assigning old to where head is, and assigning head-to-head error next, where head error of next cell is null but pointing to that, and then we set our list to null, and then we delete this cell. And so after we're done, right, we have the empty list again, the head pointer back to null. So it seems like we're doing pretty good job. Have a bunch of cells. Have one cell. No cells. Different things kind of seem to be going through here doing the right thing. Let's do it. A little bit of code on this side.

Student:[Inaudible].

Instructor (Julie Zelenski):Okay. Pop empty cell. I've got that. Oh, I think I may have proved, oh, it deliberately makes the error, in fact. That it was designed to test on that.

[Inaudible] up to the end. So let's see if we get back a three, two, one out of this guy. Okay. So we manage to do okay even on that.

So looking at this piece of code, all right, the two operations we're most interested in, push and pop, right, are each oh of one, right. The number of elements out there, right, aren't changing anything about its performance. It does a little bit of imagination and a little bit of pointer arrangement up here to the front to the end and so it has a very nice tight allocation strategy, too, which is appealing relative to what the vector-based strategy was doing. It's like, now, it's doing things in advance on our behalf like extending our capacity so that every now and then we had some little glitch right when you were doing that big resize operation that happened infrequently, you won't have that same concern with this one because it does exactly what it needs at any given time, which is allocate a single cell, deletes a single cell, and also keeping the allocation very tidy in that way. So they'll both end of being oh of one. And both of these are commonly used viable strategies for how a stack is implemented. Right. Depending on the, you know, just the inclination of the program or they may have decided one way was the way to go versus another that you will see both used in common practice because there really isn't one of them that's obviously better or obviously worse, right. Now, this uses a little bit more memory per cell, but then you have excess capacity and vector has excess capacity but no overhead per cell. There's a few other things to kind of think about trading off. You say, well, the code, itself, is a little harder to write if you're using only [inaudible]. That means it's a little bit more error prone and more likely, you'll make a mistake and have to do debug your way through it and the way with the vector it was pretty easy to get without tearing out your hair.

Now, if we can do stack and queue that fast – I mean, stack that fast, then you figure queue must be not so crazy either. Let me draw you a picture of queue. Okay. I've got my queue and I can use the End queue operation on it to put in a 10, to put in a 20, to put in a 30. And I'm going to try and get back out what was front most. Okay. So queue is FIFO, first in first out, and waiting in line, whoever's been in the queue longest is the first one to come out, very fair handling of things. And so if were to think about this in terms of let's go straight across the linked list. We just got our head around the link list with the stack. Let's see if there's a link list for the queue provide the same kind of easy access to the things we need to do the work. So it seems like the two strategies that we looked at for stack are probably the same two to look at for this one, right, which we go in the front ways orders. I mean the front of the queue accessible in there and reversing our way down to the tail or the other way around. Right. I mean, the tail of the queue up here in the front leading the way to the head of the queue. So this is head – I'll call it front of queue, and this is back, this is the back and that's the front. Okay. So that will be GA that will be GB. But first off, ask yourself, where does the queue do its work? Does it do it at the front or does it do it at the back? It does it at both. But when you're end queuing, right, you're manipulating the back of the queue, adding something to the tail end of the line. When you are de queuing, you're adding something to the front of the queue.

So like unlike the stack where both the actions were happening in the same place, and that kind of unified your thinking about where to allow for that, you know, easy access to

the top, the one thing the stack cared about, but the bottom of the stack, right, was, you know, buried and it didn't matter. The queue actually needs accesses to both. So that sort of sounds like that both these strategies, right, have the potential to be trouble for us. Because it's like in the link list form, right, you know the idea that access to the head is easy, access to the tail that is a little bit more tricky. Now, I said we could add a tail pointer. And maybe that's actually going to be part of the answer to this, right, but as it is, with no head pointers in here, that adding an item to A would involve traversing that queue to find the end to put a new one on. Similarly, if indeed, the inverse operation is going to be our trouble, which is de queuing, we have to kind of walk its way down to find the last element in the link list ordering, which is [inaudible] queue. So what say we add a tail pointer in both cases? So we have a head that points to here and a tail that points to there, it's like that will make our problem a little bit easier to deal with. So let me look at B first. That adding to the back of the queue is easy. We saw how we did that with the stack, right, so that the End queue operation doesn't need that tail pointer and it already kind of wasn't a problem for us. Now, the D queue operation would mean, okay, using our tail pointer to know where the last close value is tells us it's ten.

But we have the same problem I had pointed out with the stack, which is, we can get to this value, we can return this value, you know, we can delete this cell, but what we need to do, also, in the operation, update our tail pointer so a subsequent D queue can do its work efficiently. And it's that backing up of the tail pointer that is the sticky point that given that ten doesn't know who points into it, the single link list if very asymmetric that way. But you only know what's follows you, not what proceeds, but backing up this pointer is not easy. It involves the reversal of going back to the beginning, walking your way down to find somebody who points to the last [inaudible]. So it seems like this tail pointer didn't buy us a lot. It helped a little bit, you know, to get some of the operation up and running, but in the end, keeping and maintaining that tail pointer sort of came back to bite us. In the case of the Strategy A, the tail pointer gives us immediate access to the back, which we're going to need for the end queue. If I go to end queue of 40, in this case, then what I'm going to need to do is make a new cell, attach it off of the current cell, and then update the tail, right, to point to that cell while that update's moving forward down the list, right. If you're on the third cell and you add a fourth cell, you need to move the tail from the third to the fourth. Moving that direction's easy. It's the backing up where we got into trouble. So in fact, this suggests that Strategy A is the one we can make both these operations manipulate in constant time in a way that this one got us into trouble. I think we can do it. Let's switch it over. I've got an empty queue here. Also, it is empty in size. This is the same problem with size, which is either you have to go count them or you have to cache the size. I will build the same exactly structure, in fact.

Now, I'll type next. It's going to have both a head and a tail pointer so we can keep track of the two ends we've got going. I think I'm missing my [inaudible]. Slip over. Okay. And then I will make the same comment here about, yeah, you need to delete all cells. So I set the head and the tail to null. I'm now in my empty state that tells me I've got nothing in the queue, nothing at all, so far. And my Y is empty. Oh. We'll check that return equals, equals null, just like it did. In fact, it looks a lot like the stack, actually. And here I'm going to show you something kind of funny. If I go take my stack CCPs pop. So if

you remember what pop did over here, is it took the front most cell off the list. And in the case of the queue that was the popping operation. And it turns out the D queue operation is exactly the same. If we're empty, right, then we want to raise there. Otherwise, we take the head's value. We move around the head, we delete the old memory. So it's not actually the top element. I should, actually, be a little bit more careful about my copying and pasting here. I could really call that front. It's the front-most element on the top. But it's like the exact same mechanics work to take a cell off the front in that way. Well, that's sort of nice. And it's like, yeah, well, since I have some code around I want to go try it. My stack, also, kind of does some things useful in push that I might be able to use. It's not exactly the same because it's adding it to the front, but it is setting up a new cell. So I'm going to make a new cell and I set its value. Then, it's attaching looks a little different. Let's take a look. We omitted a cell, copied the value in, now the goal is giving our pointer to the tail, we want to attach onto the tail. So we know that it's always going to have a next of null, the new cell. So no matter what, it will be the new last cell and it defiantly needs that value that tells us we're at the end of the list. And then we have to attach it to our tail. I'm going to write this code first and it's going to be wrong. So just go along with me in this case. If the tail's next is the new cell. So if I'm wiring in the pointer from the tail onto the cell we have there, and then we want to update the tail to point to the cell. So almost, but not quite, everything I need to do. Someone want to point out to me what it is I have failed to consider in this situation.

Student:[Inaudible].

Instructor (Julie Zelenski):It's the what?

Student:[Inaudible].

Instructor (Julie Zelenski):Yeah.

Student:– trying to make null point to something [inaudible].

Instructor (Julie Zelenski):Exactly. So if my queue is totally empty. So in these cases like – one thing you can often think about whenever you see yourself with this error, and you're dereferencing something, you have to considered, well, is there a possibility that that value is null. And if so I'd better do something to protect against it. So in the case of the new cell here, I'm setting aside and not clearing that cell. We know it's valid, [inaudible], we've got good memory. That part's good but the access to the tail, and that's assuming there is a tail, that there is at least one cell already in the queue that the tail is pointing to.

When that's not true, this is going to blow up. It's going to try to dereference it. So what we can do here is we can just check for that. We can say if the queue is empty then what we want to do is say that head equals tail equals new cell. Otherwise, we've got some existing tail and we just need to attach to it.

Now, this is what I meant about that often there are little bit of special cases for certain different inputs. In particular, the most common ones are an empty list or a single link list being distinguished from a list that has two or more elements. So sometimes it's all about – a single link list has problems and so does the empty list. In this case, it's exactly the empty list. Like a single cell is actually fine, you know, one, ten, 6,000 all work equally well. It's that empty case that we have to work on setting our head and tail to that.

So with this plan we go back to my use of this. And stop talking about my stack and change into my queue, end queuing instead of pushing. [Inaudible]. Oh, 591 errors. Well, that's really – 591, what is that? [Inaudible]. Oh, yeah, this is the year I was going to talk about but I failed to do.

So this is the kind of thing you'll get from failing to protect your [inaudible] from being revisited. And so I'll be – if I haven't deft this funny little symbol I just made up then I want to find it in the end deft. So if it came back around and it saw this same header file, it was supposed to say I already seen that symbol and I won't do it. But, in fact, I had accidentally named one of them with a capital H and one with a lowercase H. It says if this symbol's not been identified then define this other symbol and keep going. And so the next time it saw the header files it said if that this lowercase H hasn't been defined, well it hadn't so it made it again and then it kept doing that until it got really totally twisted up about it. So I will make them match, with any luck, without the caps lock on. Three times is a charm. And then one, two, three pop out the other side of queue, as I was hoping. Okay. So you know what I'm going to do, actually? I'm actually not going to go through the alternate implementation of queue. I'm just saying like given that it works so well for stack, right, to have this kind of single link list, you know, efficient allocation, it's not surprising that queue, like, given to us. And what this is, like I said earlier, I said, well, you know you can do things with vector, you can do things with stack, anything you do with stack you can do with vector if you were just being careful. Right. The pushing, and popping, and the queue D queue, are operations that you could express in other terms of vector adds and removes and inserts that things.

But one of the nice things about providing a structure like a stack or queue is it says these are the only things you're allowed to add at this end, remove from that end, it gives you options as a implementer that don't make sense for the general purpose case. That the vector as a link list had some compromises that probably weren't worth making. But in the case of stack and queue it actually came out very cleanly and very nicely, like this tidy little access to one end, or the other end, or both, right, was easily managed with a link list and then it didn't have any of the constraints of the continuous memory to fight with. And so often having a structure that actually offers less gives you some different options as an implementer because you know you don't have to support access into the middle in an efficient way because the stack and queue don't let you. They don't let you riffle through the middle of the stack or the queue to do things and that gives you some freedom as an implementer, which is neat to take advantage of. Yes, sir.

Student:[Inaudible] in Java that, you know, we supposed to use an Array whenever we could because Array uses more memory –

Instructor (Julie Zelenski):Um hm.

Student: Does the queue and stack use more memory than an Array, for example?

Instructor (Julie Zelenski):So typically they'll be about the same, because of exactly this one thing, which is it allocates tightly so it asks for one cell per entry that's in there, right. And that cell has this overhead in the form of this extra pointer. Right. So in effect it means that everything got a little bit bigger because everything's carrying a little overhead. Okay. So there's a little bit more memory.

The thing the Array is doing is it tends to be allocating extra slots that aren't being used. So it's not usually tightly allocated either. So in any given situation it could be as much as twice as big because it had that extra space. So kind of in the tradeoff they tend to be in the same general range. This one has about twice the space, I think it's about four bites it has a four-byte pointer. The thing's much bigger and it turns out actually this four bite pointer might be a smaller percentage of the overhead and will be a larger amount of the capacity in the excess case.

So for a lot of things they're going to be about the same range. And then there's a few isolated situations where, yeah, if you have very big things being stored having a lot of excess capacity is likely to be a more expensive part of the vector strategy than the link list. But the link list does have this built in overhead for every element. And so it's not the case where you would say right off the bat, well, definitely an array over link list because that the allocation space. It's like, hey, you have to think about the whole picture. That is going to be the whole next lectures. Well, you know, it's about tradeoffs. It's not that one strategy's always going to be the clearly better one. If it is, then we don't need to think about anything else. There are times when you know that one is just great and there's no reason to think about anything else. And then there's other times when there are reasons to think about different ways to solve the same problem. So let me propose a case study that has some meat. You may think along the way, but you're going to really have to pretend that this is going to be relevant at first because it's going to seem like you've been transferred back to 1970, which was a very bad year, long before you born.

I want you to think about the idea of a text setter. So Microsoft Word, or you know your email send window, or BB edit, or X code. All these things, right, have some backing of what usually is called a buffer, a buffer, sort of a funny word, right, but a buffer of characters behind it. Okay. And that buffer is used as the kind of document storage for all the characters that you're editing and moving around. As you kind of bop around and move around there's something often called the cursor where you type characters and characters get moved as you insert. You can select things, and delete them, and cut, copy, and paste, and all that sort of stuff. So what does that data structure look like? What is the kind of thing that wants to back a buffer? What kinds of things are important in that context to be able to support and what operations needs to be optimized for that tool to field [inaudible]? So what we're going to look at is a real simplification kind of taking that problem, kind of distilling it down to its essences and looking at just six operations that are kind of at the core of any text editor about moving the cursor forward and

backwards with these commands. Jumping to the front and the back of the entire buffer and then inserting and deleting in relative to the cursor within that buffer. And we're going to do this with an extremely old school interface that's actually command base. It's like VI comes back from the dead for those of you who have ever heard of such things as VI and E Max. And so we're going to look at this because there's actually a lot of ways you can approach that, that take advantage of some of the things that we've already built like the vector, and stacking queue, and the link list. And sort of think about what kind of options play out well.

What I'm going to show you first off is just what the tool looks like so you can really feel schooled in the old way of doing things. So I insert the characters A, B, C, D, E, F, G into the editor. And then I can use the command B to backup. I can use the command J to jump to the beginning, E to jump to the end, and so F and B, B moving me backwards, F moving me forwards, and then once I'm at a place, let's say I jump to the end, I can insert Z, Z, Y, X here at the beginning and it'll slide things over, and then I can delete characters shuffling things down. So that's what we've got. Imagine now, it's like building Microsoft Word on top of this. There's a lot of stuff that goes, you know, between here and there. but it does give you an idea that the kind of core data requirements every word processor needs some tool like this some abstraction underneath it to manage the character data. Okay. Where do we start? Where do we start? Anyone want to propose an implementation. What's the easiest thing to do?

Student:[Inaudible].

Instructor (Julie Zelenski):In a way. Something like that. You know, and if you think Array, then you should immediately think after it, no, I don't want to deal with Array, I want a vector. I'd like a vector. Vector please. Right. So I'm going to show you what the interface looks like, here, that we have a buffer in these six operations. We're going to think about what the private part looks like, and we say, what about vector. Vector handles big, you know, chunky things indexed by slots. That might be a good idea. So if I had the buffer contents A, B, C, D, E, right, and if I could slam those into a vector, and then I would need one other little bit of information here, which is where is the cursor in any given point because the cursor is actually where the uncertain lead operations take place relative to the cursor. The buffer's also going to manage that, knowing where the insertion point is and then deleting it and inserting relative to that. So I say, okay. I need some index. The cursor actually is between two character positions. This is like slot zero, one, and two. And the cursor is between, actually, one and two. And there's just a minor detail to kind of have worked out before we start trying to write this code, which is do we want the cursor to hold the index on the character that's to the left of it or to the right of it. It's totally symmetric, right.

If I have these five characters, it could be that my cursor then goes from zero to four, it could be that it goes from zero to five, or it could be it goes from negative one to four, depending on which way I'm doing it. I'm going to happen to use the one that does zero to five where it's actually recorded as the character that's after it. Okay. So that's just kind of the staring point. I'm going to write some code and make it happen. So I'm going

to remove this. Reference it, that's okay. And then I'm going to add the things I want to replace it with. So I put in the editor code and to put in the buffer code. Was buffer already in here? I think buffer's already here. Okay. Let's take a look. I get my buffer. So right now, let's take a look what's in buffered. I think I have the starting set information. So it has move cursor over, you know move the cursor around, it has the delete, and it has – let me make it have some variables that I want. I've got my cursor. I've got my vector of characters.

Right now, it has a lot of copying, I think, just in anticipation of going places where copying is going to be required. Right now, because it's using vector and vector has deep copying behavior, I'm actually okay if I let windows copy and go through. But I'm actually going to not do that right away. All right. So then let's look at the implementation side of this and I have an implementation in here I want to get rid of. Sorry about this but I left the old one in here. That will do all the things that will need to get done. Okay. So let's deal with at the beginning. Starting off, right, we will set the cursor to zero. So that's the vector gets started, initialize empty, and nothing else I need to do with the vector. A character [inaudible] cursor position in that, and then moving the cursor position forward, it's mostly just doing this, right, cursor plus, plus, right. It's just an easy index to move it down. But I do need to make sure that the cursor isn't already at the end. So if the cursor is less than the size of the vector then I will let it advance. But it never gets beyond the size itself. So like all of these are going to have same problems. Like, if I'm already at the end and somebody asks me to advance, I don't want to suddenly get my cursor in some lax stated that sort of back that.

Similarly, on this one, if the cursor is greater than zero, all right, then we can back it up and move the cursor around. Moving the cursor at the start is very easy. Moving the cursor to the end is, also, very easy, right. We have a convention about what it is. And so then inserting a character is, also, pretty easy because all the hard work is done by the vector. When I say insert this new character at the cursor position, right, then this character advance the cursor to be kind of past it, and then deleting the character, also, pretty easy. Let me show you, before I go finishing the cursor, I just want to show you this little diagram of what's happening while I'm doing stuff. So this is kind of a visualization of the things that are happening. So this is showing the vector and its length. And then the cursor position as we've done things. Inserted six characters, the cursor's currently sitting at the end, and for the length of the cursor, actually, the same value right now. If I issue a back command, right, that deducts the cursor by one, it shows another one backs up by one, and then eventually the cursor gets to the position zero and subsequent backs don't do anything to it, it just kind of bounces off the edge. And moving forward, things are easier to deal with. It will advance until the gets to the length, the size of that text, and it won't go any further. And that jumping to the front in the end are just a matter of updating that cursor position from zero to the size and whatnot.

The operation where things get a little more bogged down is on this insert where I need to insert a position. If I insert the X, Y, Z into the middle there and you can see it kind of chopping those characters down, making that space to insert those things in the middle. Similarly, doing delete operations, if I jump to the beginning I start doing deletes here. It

has to copy all those characters over and deduct that length, right, so that the vectors do that work for us on remove that. That means that if the vector's very large, which is not a typical in a word processor situation, you could have you PhD theses which has hundreds of pages, if you go back to the beginning and you start deleting characters you'd hate to think it was just taking this massive shuffle time to get the work done. So let's see my – I'm going to bring in the display that – whoops, I don't like something here. [Inaudible]. Now, what do we have? Oh, look at this. Well, well, we will continue on. This inside – okay. Hello. [Inaudible]. I have no idea. I'm not going to try to [inaudible]. Okay. It's inside. That's got me totally upset. Okay. We still have our old code somewhere. Okay. Why do I still have the wrong – what is – okay. Oh, well. Today's not my lucky day. I'm not going to worry about that too much. I'm just going to show [inaudible]. Okay.

So seeing what's actually happening, right, it's like just mostly kind of moving that stuff back and forth, and then having the vector do the big work, right, of inserting and deleting those characters, and doing all the copying to get the thing done. We'll come back over here and kind of see what good it does and what things it's not so hot at, right, is that it is easy to move the cursor wherever you want. You know, that moving it a long distance or a short distance, moving it is a little bit or to the beginning to the end, equally easy as by just resetting some variable. It's the enter and delete, right where this one actually really suffers, right, based on how many characters, right, follow that cursor, right you could potentially be moving the entire contents of the buffer. Adding at the end is actually going to be relatively fast. So if you have to type in order, like you just type you theses out and never go back to edit anything, then it would be fine, adding those characters at the end. But at any point, if you move the cursor back into the mid of the body, somewhere in the front, somewhere in the middle, right, a lot of characters get shuffled as you change, makes edits in the middle there. And so what this seems that it actually has good movement but bad editing as a buffer strategy. That's probably, you know, given that we have these six operations we'll all interested in, this is probably not the mix that seems to make the most sense for something that's called an editor, right. It is the editing operations are the ones that is weakest on, it has its most debilitating performance, it wouldn't make that good of an editor, right, if that was going to be your behavior.

It has one advantage that we're going see, actually, we're going to have to kind of make compromises on it, which it actually uses very little extra space, though, but it's using the vector, which potentially might have excess capacity up to twice that. So maybe it's about two bites per char in the very worse case. But it actually has no per character overhead that's already imbedded in the data structure from this side. Now, I'm just going to go totally somewhere else. Okay. So instead of thinking of it as vector, thinking of it as continuous block, is to kind of realize the editing operations, right, that comes back to the idea, like if I were working at the very end of my document that I can edit there efficiently. It kind of inspires me to think of, how do I think I can make it so the insertion point is somehow in the efficient place. Like the edits that are happening on the insertion point, can I somehow make it to where access to the insertion point is easy? That rather than bearing that in the middle of my vector, if I can expose that to make it accessible easily in the way the code manipulates the internal of the buffer.

And so the idea here is to break up the text into two pieces. Things that are to the left of the cursor, things that are to the right, and I'm calling this the before and the after. And then organize those so they're actually averted from each other to where all the characters that lead up to the cursor are set up to where B is very close, assessable at the top of, in this case, a stack, and that the things that are farther away are buried further down in the stack in that inaccessible region. And similarly over here, but inverted, that C is the top of this stack and D and E, and things are further down, they are buried away from me, figuring the things I really need access to are right next to the cursor that I'm willing to kind of move those other things father away to gain that access. And so what this buffer does is it uses two stacks, a before stack, an after stack, and that the operations for moving the data around become transferring things from the before to the after stack. Where's the cursor? How does the cursor represent it? Do I need some more data here? Some integer some –

It's really kind of odd to think about but there is no explicit cursor, right, being stored here but the cursor is, in this case, right, like modeled by what's on the before stack or all the things to the left of the cursor. What's on the after stack is following the cursor. So in fact, the cursor is represented as the empty space between these two where you have, you know, how many ever characters wrong before is actually the index of where the cursor is. So kind of a wacky way of thinking about it, but one that actually have some pretty neat properties in terms of making it run efficiently. So let me show you the diagram of it happening. So insert A, B, C, D, E, F, G. And so, the operation of inserting a new character into this form of the buffer is pushing something on the before stack. So if I want to move the cursor, let's say I want to move it back one, then I pop the top off of the before push it on the after. I keep doing that and it transfers the G to the H, you know, the E and so on, as I back up. Moving forward does that same operation in the inverse. If I want to move forward I take something off the top of the after and push it on before. So it's kind of shuffling it from one side to the other. Given that these push and pop operations are constant on both implementations of the stack we saw, then that means that our cursor movement is also of one so I can do this. And if I do deletes it's actually just popping from the after stack and throwing it away. So also easy to do edits because I'm talking about adding things to the before and deleting things from after, both of which can be done very efficiently.

The one operation that this guy has trouble with is big cursor movement. If I want to go all the way to the beginning or all the way to the end, then everything's got to go. No matter how many things I have to empty out the entire after stack to position the cursor at the end, or empty out the entire before stack to get it all the way over here. So I could of sort of talk you though this code, and actually, I won't type it just because I actually want to talk about its analysis more than I want to see its code. Each of these actually becomes a one liner. Insert is push on the before stack. Delete is pop from the after stack. The cursor movement is popping from one and pushing on the other depending on the direction and then that same code just for the Y loop, like wild or something on the one stack, just keep pushing and popping from one empty to the other. So very little code to write, right, depending a lot on the stacks abstractions coming through for us and then making this ultra fit that we've managed to get editing suddenly, like, a lot more efficient

but the cost of sacrificing one of our other operations. Let's take a look at what that look like, right.

Suddenly I can do all this insert and delete at the insertion point with very little trouble, and I suddenly made this other operation, oddly enough, slow. All a sudden it's like if you go back to the beginning of your document you're in the middle of editing, you're at the bottom. You say, oh, I want to go back to the beginning and you can imagine how that would feel if you were typing it would actually be the act of going up and clicking up in the top right by where you made the insertion and all a sudden you'd see the white cursor and you'd be like, why is that taking so long. How could that possibly be, you know, that it's doing this kind of rearrangement, but yet it made editing even on a million character document fast. But that movement long distances, you know, jumping a couple of pages or front to back, suddenly, having to do this big transfer behind the scenes, so different, right. So now I had six operations, I had four fast, two slow, now I have four fast, two slow. It still might be that, actually, this is an interesting improvement, though, because we have decided those are operations that we're willing to tolerate, being slower, especially, if it means some operation that we're more interested in, if performance being faster. And that likely seems like it's moving in the right direction because editing, right, being the whole purpose, right, behind what we're trying to do is a text setter, that seems like that might be a good call.

What I'm going to start looking at and it's going to be something I want to talk about next time, is what can a link list do for us? Both of those are fighting against that continuous thing, the copying and the shuffling, and the inserting is there something about that flexibility of the rewiring that can kind of get us out of this some operations having to be slow because other operations are fast. Question.

Student:[Inaudible].

Instructor (Julie Zelenski):Yeah.

Student: Why is space used for stacks when those are half?

Instructor (Julie Zelenski):Yeah. So in this case, right, depending on how you're modeling your stack, if it is with vectors, right, then you're likely to have the before and the after stack, both have capacity for everything, even when they're not used, right. And so it's very likely that either right give 100 characters they're either on one stack or the other, but the other one might be kind of harboring the 100 spots for them when they come back. Or it could just be that you have the link list, which are allocating and de allocating but has the overhead of that. So it's likely that no matter which implementation the stack you have, you probably have about twice the storage that you need because of the two sides and the overhead that's kind of coming and going there. So we will see the link list and we'll talk that guy through on Friday. I'll see you then.

[End of Audio]

Duration: 51 minutes