ProgrammingAbstractions-Lecture21

Instructor (Julie Zelenski):Hey, welcome to Friday. I've got a big voice today. We are going to pick up on the editor buffer case study where we left off on Wednesday's lecture. So talking about the link list implementation and kind of see what that gets us. And what we have left of today's lecture, we'll start talking about the map implementation. So we have two other classes to go, on which is map and set, and for those we're going to learn about the concept of binary search trees and hashing. And then that kind of completes the whole picture of all the pieces in the AT library. So we'll be in a good position from there to kind of pick up some advanced topics before we close off for the end of the quarter.

The reading is, we're finishing the last half of chapter nine and then starting on chapter 13, which is where the binary search tree stuff is covered. And then we will come back up and pick up chapter 11 after that. We will not do chapters 12 and 14. So if you are keeping track of things in the text, we will not cover those at all. I'm going to go to the Truman Café today after class, so if you have free time this afternoon and want to come and hang out and have a little snack and beverage with us. I would love to have you join me. We'll also do that next Friday, and I think that will be the last one I'll be able to do, because the very last of the quarter I'm going to be at a conference that day. So hopefully one of the last two days you can come and visit with us.

So let me ask a little bit about priority queue. How many people have started priority queue somewhere? Gotten some stuff done? Okay that's not a lot of you. What are the rest of you guys doing? How many people have finished at least one of the implementations? So people who have started. How many have finished both implementations? How many people are taking the weekend off? Hold your hand up. Any advice, from those of you who finished one or the other of the implementations, that you want to offer up to those who are a little bit farther back behind you? If you want them to struggle the way you did, you can just keep quiet. You can just smile. Did you do them in the order that they are listed? Did you link-list them in heap and that worked out okay? All good? Did anybody do them the other order just in case?

All right, that is your weekend excitement. Let us refresh about where we were at the end of Wednesday. As we had talked about the vector form of the buffer, so the editor buffer that's backing the word processing client, and the two stack version, where we're shuffling stuff, often on the before and after stacks to get things done, right? And when we were looking at the main six operations, we're looking at the four cursor movements and the two editing operations, right, that we had traded off. Where editing had been slow in the vector, to where editing was now fast, but then these long distance movements were slow, because of the shuffling. And then there was a little discussion at the very end about how this probably increases our space requirements, because we're now keeping two stacks that are each capable of holding the entire contents as opposed to one vector here with its slop.

So that was where we were at. We are still in kind of the pursuit of this Holy Grail of could we get it to where everything is fast. Is it always a matter of tradeoffs or could we just invest some more smarts and more cleverness and hard work and squeeze it down to where we could get everything to perform efficiently? So typically when computer scientists say they want something to perform efficiently, they're hoping for log in time or faster. Constant time is the best. Log in grows so slowly that in fact, logarithm time is effectively constant for all reasonable values of N up to millions and billions. And so typically anything that is linear or higher is often a weak spot that you are trying to look at, and certainly things that are quadratic or exponential are definite opportunities for improvement.

So let's look at this idea of the link list, so that both the vector and the stack – we're relying on this idea that continuous memory is the weak link that was causing some problems. In the vector, that shuffling to move things up and down, or in the case of the stack version, one side to the other was part of what we were working up against. And the link list promises to give us this opportunity to have these each individual character being managed separately, and flexibility might help resolve some of our problems here. SO what we're going to look at is the idea of implementing the buffer using a link list.

So if I have characters A, B, C, D, E, I could have a link list here where I'm pointing to the head cell, which is A, which points to B, and C, and D, and E, and down to the null. SO what I'm modeling it as in the private data section will be a little link list node with a character in there, a pointer to the next one. And then I'm going to keep two pointers, one to the front-most cell, and then that's going to model where the cursor is within the buffer. So rather than doing this like an index, an index was handy in the case of vector where we had direct access, right? Given the way the link list works, knowing that it's at the fifth cell, the fourth cell isn't really very helpful. It would be more helpful to have the pointer kind of directly in the midst of things to give us direct access to where the cursor is.

But let's think a little bit about that cursor, because there's an important decision to be made early about helping to facilitate the later work we have to do. I have the contents A, B, C, D, E, so the cursor is actually between two characters. If the cursor right now is situated after the A and before the B, what I'm modeling is A cursor B, C, D, then it seems like the obvious two choices for where the cursor might point is to A or to B. And I had said in the case of the vector that if it were index zero or index one that there wasn't a really strong preference for one over the other. There is going to be a really good reason to pick one for the link list over the other. Would you rather point to the A for where the cursor is or would you rather point to the B?

Student:[Inaudible]

Instructor (Julie Zelenski):Exactly, so the idea is that when the cursor position is there, the next edit actually goes in between the A and the B. So if I insert the character Z, it really does go right here. And if I am going to, if effect, wire that in and splice it in then list, if I'm pointing to B then I'm kind of hosed, because I'm already one past where I need to be able slice in. Because I need to know what was behind it to bring it in. So I'm pointing to the character behind the cursor gives you access to wiring this down to here and this into there, and then updating the cursor to point to the Z without having to do any of the difficult backward looking backward traversal.

So that strongly suggests what it is, it's going to point to A when it's between A and B; B when it's between B and C; and so on. There is another little point here, which is there are actually five letters in the cursor, but there are six cursor positions. It could be at the very beginning, in between all these, or at the very end. And the way we've chosen it right now, I can identify all of the five positions that have at least one character to the left, so being – pointing to the A means it's after A; pointing to the B it's after B; and so on, all the way down to the E. But I don't have a cursor position for representing when the cursor is at the very beginning of the buffer.

So one strategy I could use, right, is to say I'll just use the special cursor position of null. But I need one more pointer value to hold onto, and I could make a special case out of this because there's null. Once I've done that though, I've actually committed myself to this thing that's going to require all of my code to be managing a special case of, well when the cursor's null do something a little different than when the cursor is pointing to some cell. I'm going to show you a way that we can kind of avoid the need for making a special case out of it, by wasting a little memory and reorganizing our list to allow for there to be a sixth cell, and extra cell that we call the dummy cell.

So in this case that cell would go at the very front. It would have no character, just let the character field be unstatused. It's not really a character. What it is, is just a placeholder. And when I created the buffer to begin with, even when it's empty, it's always going to have that cell. I'm going to create that cell in advance, I'm going to have it sitting ready and waiting, and that cell never gets deleted or moved or changed. In fact, the list always, the head of this will always point to the same cell from the beginning of this object's lifetime to the end. So it will never change this at all, and the cells that follow are the ones that are going to be changing and growing and rearranging in response to editing commands.

What this is going to buy us is a couple of things. First off, it's going to make it so there is a sixth cursor position. So now when the cursor is at the very beginning, we can set the cursor to point to this dummy cell. Okay that's one thing that we solve by doing that. Another thing that we've solved is actually we have made all the other cells on the list, the ones that are being edited and manipulated. We've made them all totally symmetric. Previously there was always going to be a little bit of a special case about, well if you're the front of the list, we're seeing that sometimes we're doing inserts on the stack in queue link list.

That inserting later in the list involves kind of wiring in two pointers; one to command and one to go out. But there was a special case that oh, if you're the very first cell of the list, then your outpointer is the same, but your incoming pointer is resetting the head. That piece has actually gone away now, that now every cell in there always has a predecessor. It always has a previous, right? Something behind it so that actually they will always be pulling the pointer in and pulling the pointer out for every cell with character data that's being modified. And that means some of that extra little handing of oh, if you're the very first cell, have gone away. The very first cell is always this kind of dummy cell and doesn't require us to go out of our way to do anything special for it.

So this is going to solve a bunch of problems for us, and it's kind of a neat technique. This thing is used actually fairly commonly in situations just managing a link list just to buy yourself some symmetry in the remaining cells.

So let me look at some code with you, and I'm going to actually draw some pictures as I go. I'm not going to write this code in real time, because I think it's actually more important to see it acting on things than it is to see me typing out the characters. But the mechanism, right, in the case of the buffer here is it's going to have a head point, which points to the front most cell; and a cursor, which points to the character that proceeds the cursor. So if right now the contents of the buffer are the contents A, D, C. And let's say that right now the cursor is pointing to A. Actually, I need my dummy cell. Let me move everybody down one, so you just have dummy cell and my A and my B.

And so the cursor right now is at the very front of the buffer. This is kind of what it looks like. It has two characters, the A and the B. I'm going to trace through the process of inserting a new character into the buffer. It is that my local variable CP is going to point to a new cell allocated out in the heap. Let's say the character I'm going to insert is the Z, so I'll assign CH to the character, the slot right there. And then the next for this gets to – what the cursor is – kind of following the cursor. So the cursor is pointing to the cell before it, and the next field of that cell tells us what follows it; so if I trace my cursor's next field, it points to A, and so I'm going to go ahead and set up this new cell's next field to also point to A.

So now I've got two different ways to get to the A cell, tracing off the dummy cell or tracing off this new cell. And then I update the cursor's next field to point to CP. So I just copied right there that CP's next field was pointing to A. I copied it down here to kind of do the splice out. Now I'm going to do the splice in, causing it to no longer point to A, but instead to point to my new cell down here, CP. And then I update the cursor to point to this new cell. That just means that subsequent inserts, like the C is kind of behind the cursor after I've made this edit. That goes away when that one ends, and so now the new structure that I have here is head still points to the dummy cell that never changes. Dummy cell now points to Z, Z points to A, Z points to B, and then the cursor in this case is between the Z and the A, by virtue of the fact that the cursor is pointing to Z.

A subsequent one would kind of do the same thing. If I typed out ZY after I did this. We'd make a new cell over here. It would get what was currently coming out of the cursor's field. The next field, right, we would update cursor's next field to point to this new one. And then we would update cursor to point to this guy. And so now following the list, right, dummy cell, Z, Y, A, B, cursor pointing to the Y, which is the character directly to the left of the cursor. And so inserting in any position now, front cell, middle cell, end cell, doesn't require any special case handling. So you don't see any if of things.

And you notice that you'll never see an update to the head directly here. That the head was set up to point to the dummy cell in the constructor and never needs any adjusting. It's only the cells after it that requires the new arrangement of the pointers coming in and out.

When I'm ready to delete a cell, then the mechanism for delete in the buffer is to delete the thing that follows the cursor. So if I have Z, Y, A, B, this is the character that delete is supposed to delete, the one after the cursor. And so, the first thing it looks at is to make sure that we're not already at the end. In the case of the link list form of this, if the cursor was pointing to the last most cell, and then we looked at it's next field and saw that it was null, that would be a clue. Oh, it's pointing to the very last cell, there's nothing that follows it. So we have a quick test to make sure that there's something there to delete. There is something following the cursor, and in this case since the cursor is pointing to the Y, we're good.

It says look at its next field, it points to A. It says okay, call this thing old and point to that A. Now do a wire around it. So take the cursor's next field, and so this thing is where I'm targeting my overwrite, and copy out of the old its next. It's no longer pointing to the A, but instead it's pointing to the B. And deleting the old, which causes this piece of memory to be reclaimed, and now the contents of my buffer have been shifted over to where it goes Z, Y, and straight to D. The cursor in this case doesn't move. It actually deleted to the right, and so whatever was to the left of the cursor just stays in place. So again, no need to update anything special for the first or the last or any of those cells, but the symmetry of all the cells past the dummy kind of buys us some convenience of handling it.

Student: What does a dummy cell look like?

Instructor (Julie Zelenski):The dummy cell looks like just any other cell. It's actually a cell, T, and what you do is you just don't assign the Christian field, the character field, because it has nothing in it. So it actually looks just like all the others, it's just another node in the chain. But this one, you know you put there purposefully just as the placeholder. So it doesn't contain any character data. So if you're ready to print the contents of the buffer, you have to remember the dummy cell is there, and just skip over it. You tend to start your list at heads next, the first interesting cell to look at, then work your way down to the end. So it's just like all the others, but you just know that the character data here is useless, it's not important. You didn't even write anything in there.

Student: When you delete, you just go left?

Instructor (Julie Zelenski):It turns out the delete in this case goes forward. So you can imagine that in some other world it does, but all the ones we have done so far have always deleted forward. So the delete – for example, the stack case, grabbed from the after stack – it's done that way, because it happens to simplify some other things that you can imagine that delete and reverse. What would it take? Well, you'd have to back the cursor up. And we're going to find out pretty soon that would make it challenging for the link list. So partly we've picked a delete that made certain things work out a little better for us.

So both of these operations are O of 1; this is where the link list really shines, right, on this sort of stuff. Because if you have access to the point at which you want to make a modification, so if you are already kind of in the midst of the thing you're trying to rearrange, it's often to the link list's advantage, because it actually can do these this rewiring in the local context. And even if there's thousands of characters that follow the cursor. It doesn't matter. And there can be thousands of characters preceding the cursor for that matter, too. The idea that there's a much larger context that's working and doesn't impact its performance; it's really just doing this local little rearrangement of the pointers. And that's a real strength of the link list design is that flexibility that doesn't rely on everything living in contiguous memory.

So let's talk about the movement options for this guy. So right now I have the contents Z, Y with the cursor between that, and the B following that. So three characters worth, and then the initial two operations are pretty simple. The later two are going to be a little bit messier. So let's look at the easy ones first. Moving the cursor to the beginning, so that jump that pulls you back to the very beginning, very easy in a link list form. If I want to move the cursor to the beginning, all I need to do is take that cursor pointer, and reset it to the head, which is where the dummy cell is.

And so, by virtue of doing that, the character it points to is the one that's to the left, right of that, and to the left of that the dummy cell is one we're not counting. In fact, it's like nothingness that's there, and that means that the first character that follows the cursor is Z, which is the initial character of the buffer. So that reset, easy to do, we have easy access to the front of the list, and so no special crazy code required.

Moving the cursor forward, also an easy thing to do, link list are designed to make it easy to work your way from the front to the back. And so moving the cursor forward is advancing that cursor by one step. So in the array form of this, what was a cursor plus, plus, the comparable form of that in the link list is cursor equals cursor next. It has a little if test there, all of the versions actually have something that is comparable to this. If you're already at the end, then there's nothing to be done, so you don't advance the cursor off into space. You check to make sure that there is something for it to point to. So in this case, seeing if the cursor's next field is null, it's not, it points to Z. Then we go ahead and update the cursor to there, which has the effect of changing things to where the Z is the character to the left of the cursor, and then the YB follows it.

So moving this way, getting back to the beginning, and kind of walking our way down is kind of easy to do. Now we start to see where the link list causes us a little bit of grief, moving the cursor to the end. So now, starting at the beginning or the middle, or wherever I am, I want to advance my way to the end. I don't know where the end is. Link lists in general don't keep track of that, that in the simple form of the single link list, I'm going to have to work to find it. So I'm going to take where I am and walk my way down. And this one actually just makes use of an existing public number function to move cursor forward. It's like while the cursor next does not equal null, which is to say while we are not pointing at the very last cell of the link list, then keep going. So advance cursor equals cursor.

Some will say, "Is the cursor's next field null?" No, then set the cursor to be the cursor's next field. Is the cursor's next field null? No it's not, so advance the cursor to the next field. Is the cursor's next field null? Yes. So that would the last thing in the buffer, and so if we have hundreds or thousands or tens of characters, whichever it is, right, and depending on how close we already were, it will walk through everything from where the current editing position was to find the end of the buffer one by one, working it's way down.

Even more painful operation is the simple one of just moving backwards. If I'm pointing right now to that B, and I'd like to get back to that Y, the link list is oblivious about how you got there. It knows where to go from here, but that backing up is not supported in the simple form. In order to find out what preceded the V, our only option right now is to go back to the beginning, and find it. So starting this pointer CP at the head, and saying, do you point to; is your next field the cursor? And this one says, no. And then it says, well okay advance. Does your next field point to the cursor? No it's not. Go to this one, does your next field point to the same place as the cursor, yes. Wise next field, points to the same place the cursor does. That must mean that Y was the one that preceded the cursor in the link list.

So after having gone back to the beginning and walked all the way down, especially if I'm near the end, that's a long traversal. And when I get there, I can say "Oh yeah that's the one, back it up to there." Walking it all the way down; some good link list coding, good practice in there, but again a funny, funny set of trade offs, right? Does anybody have any questions about any part of the code?

Student: At the end could you just save the address of the pointer, and then the person who wants to go to the end could just access that?

Instructor (Julie Zelenski):That was a great idea. It's like, so what about improving this? Right now all we have is a pointer to the front, and each pointer has only a pointer to the next. It's like, maybe we need to add some more stuff, track some more things. What if we tracked the tail? Would that help us? If we tracked the tail, we could move to the end quickly. Would that help us moving backwards? No. It solves one of our problems though, so you're on to something. Let's look at what we've got, and then we'll start thinking about ways to fix it, because that's going to be one of the pieces we're going to think about.

So if I move to link list relative to what I have, again, it's like a shell game. The old ends moved. It used to be that editing was slow and we didn't like that. Then we made big movement fast, and now we have this funny thing were moving in certain ways is easy and moving other ways is bad. So moving down the document is good, moving backwards in the document not good. Moving back to the beginning is good, but moving to the end is bad. But then editing in all situations is good, another sort of quirky set of things. It just fells like it's like that game where the gophers pop there heads up, and you pound on them. And they just show up somewhere else. As soon as you get one thing fixed, it seems like something else had to give.

This, I think, would actually be my ideal editor, because it turns out I have exactly this working style, which is like, I'll be down at the end of the document, and I'll have to be generating new things. And I'm getting tired and I don't like writing, and what I keep wanting to do is I go back to the beginning. And I read that again, because I worked on that the most and it's nice. And so I go back to the beginning, and I'm like oh yeah, look at this. Look at how lovely this is, and then I never want to go back to the end where all the trouble is. So I just keep going back to the beginning and editing in the front, and then slowly get back to the end. And then I don't like it again, and then I go back to the beginning. But I never actually willingly go to the end to face my fears. So this is the editor I need.

Also has a bit of kicker in terms of space. So if the space for a character is 1 byte, which it typically is, and the space for a pointer is 4 bytes, then each of those link list cells is costing us 5 bytes, right? Which is sort of five times the amount of storage that the character itself would have taken alone, so in the vector it's fairly tight, a little bit more going up on that. But we've actually kind of blown it up by another factor of two even beyond the stack, which is not quite a good direction to be going.

So let's talk about what we can do to fix that. I'm not going to go through the process of this. I'm just going to kind of give you a thought exercise, so you can just kind of visualize it. What we have here, right, is an information problem. We only are keeping track of certain pathways through that link list. What if we actually increased our access to the list? We don't need to have the full access of a vector, being able to know the nth character by number is not actually as far as we need to go. But we do just need a little bit more coordination between a cell and its neighbors.

So the two things that would buy us out of a lot of the things that we're seeing here is if we added a tail pointer. So if I just add a tail pointer that points to the last cell, then I can move to the end quickly. You say cursor equals tail; the same that cursor equals head. So what was an O of N problem now is an O of 1. The backing up, not as easily solved by just having one thing that's going on; I also am going to need, on a per cell basis this ability to move backwards.

Well, it's just symmetry, right? If A knows where B is, and B knows where C is, and C knows where D is, what's to stop it from also just tracking the information going in the other way, so that D knows that it came from C, and C knows it came from B, and so on. And so having this completely symmetric parallel set of links that just run the other direction, then buys you the ability to move backwards with ease.

The other thing that this would give you is the tail pointer is almost not quite enough to give you the exact place of the end. It gives it to you, but you also have to be careful about what is it going to take to update it? And so inserting can easily update the tail pointer. It's the deleting that kind of would get you into trouble. If you were here and deleting, you would have to kind of keep track of making sure if you were deleting the thing that was the tail pointer, so you have to be a little about making sure you don't lose track of your tail. But once I have both of these in place, I suddenly have something that can move forward easily, O of 1; move to the front easily, O of 1; move to the tail easily, O of 1; move backwards O of 1. So I can make all of my operations O of 1.

Deleting, you're still going to have to delete the individual cells, but there's one place where it suffers, but that's a much more rare occurrence. And I can get all six; it moves fast, in every dimension; inserts and deletes all good. And there are two places we paid for it. One is certainly the memory. So a 4 byte pointer one direction, a 4 byte pointer another direction, plus the 1 byte for the character means we have a 9 byte per character storage usage.

And the other way, which is a little bit hard to represent with some number in a quantitative way, but actually it's also an important factor, which is your code got a lot more complicated. If you looked at the code for the vector of the stack version, it was a few lines here a few lines there. If you looked at the code for the link list singly we looked at, it's like you've got to wire up these pointers. Now imagine you've got to wire up twice as many pointers. Not only do you have the in and out going one direction, you'll have the in and out going the other direction, and just the opportunity to make errors with pointers has now gone up by a factor of two.

The idea that you could get a vector version of this up and running in no time, compared to several days, let's say, of getting the doubly link one working. It might be a factor to keep in mind. If you're kind of investing for the long haul, sure, work hard, but know that it wasn't for free. So you've got 89 percent overhead. That's a lot of overhead for small bits of data. More likely, and this is actually the way that most modern word processors do do this, is they don't just make one or the other; it's not just a array or a stack or a link list. They actually look at ways to combine the features of both to get a hybrid, where you can take some of the advantages of one pipe of data, but also try to avoid its worst weaknesses.

So the most common way they're going to do this is some kind of chunking strategy, where you have a segment of the buffer that kind of is moved aside and worked on in one little array. It might be that those arrays are ten or 20 or a sentence long, or paragraph long. It might be that they are actually dictated by when you change styles. All of the code that's in one font kind of gets moved into one chunk, and as soon as you change styles, it breaks into a new chunk that follows to kind of keep track of the formatting information. It depends on some of the other features that are present in the processor here, but it's likely that rather than having all 1,000 or all 10 million characters in one data structure, they're actually kind of separated to where you have a little bit of both.

So the most likely thing is that there is some kind of array strategy on a chunk basis. There's a linking between those chunks. And what this allows you do is to share that overhead cost. So in this case, if I had every sentence in a chunk by itself, and then if I had next and previous pointer, which pointed to the next sentence and the previous sentence, when I need to move the cursor forward or backwards within a chunk, it's just in changing this index. Oh, it's in index two, but now it's moving to three, or four, or five. And then when it gets to the end of that chunk, to the end of that sentence, and you move forward, well then it follows the next link forward.

Similarly moving backwards, right, so within a chunk doing array-like manipulations, and then as it crosses the boundary using a link list steps to get further down. So when it needs to move to the front or the back, those things are easily accessible using head and tail pointers. When I need to do an edit, it operates in this array-like kind of way when there's space within that chunk. So you start editing in a sentence, then it would kind of do some shifting on that array. But the array itself being pretty small means that although you are paying that cost of shuffling, you are doing it on a smaller chunk. It's not 1,000 characters that have to move, it's 20 or 40 that have to move within that sentence.

And that when you need to insert a new sentence or a new chunk in the middle, you are doing link list-like manipulations to build a new chunk and insert and splice, so you have that flexibility, where the many characters aren't all affected by it. So you have kind of local array editing and global link list editing that gives you kind of the best of both worlds. But again, the factor is cost showing up in code complexity. That what you are looking at is kind of space/time tradeoff. Well, I can throw space at this, and most people will say that in computer science, "Well I can throw space and this and get better times." So the double link list is a good example of something that throws space at a problem to get better time. There's a lot of overhead.

Moving to the chunk list says I want to get back some of that space. I'm not willing to invest 90 percent overhead to get this. Can I find a way to take a whole sentence worth of things and add a few pointers here? That means that there's only two 4 byte pointers per each sentence as opposed to each character, and that really way reduces the overhead. But then now, the problem is still back on your shoulders in terms of effort, now I have both an array, and a link list, and double links, and a lot of complicated stuff flying around to make it work. But it does work out.

It's kind of the process that designers go through when they're building these kinds of key structures. It like, what are the options, and do any of the basic things we know about work? What about building even fancier things that kind of take the best of both world and mix them together and have a little bit of the weaknesses of this a little of that, but none of the really awful worst cases present in the end result. So in this case you'd have a lot of things that were constant time where constant was defined by the chunk size. So O of 100, let's say, if that was the maximum chunk size to do these things, as opposed to O of N for the whole block net, so kind of a neat little thing to think through.

There was a time, just so you can feel gratified about how I've worked you very hard, but actually in the past I've been even harsher. We actually had them build the doubly linked chunked list editor buffer, and now we have you build the singly linked chunk list PQ, and it's actually still very complicated, as you'll find out, but not nearly quite as hairy as the full in both directions every way craziness. And you can kind of imagine in your mind. Any questions about edited buffer?

We’ve got two more data structures to implement, two more things that we've been using, and happy to have in our arsenal that we need to know how they work. Let's talk about map first, and in fact the strategy that I'm going to end up showing you for map is actually going to be the same one we use for set, then we'll come back and think of an even more clever way to do map, too. So we've got two things to talk about, which are binary search trees and hashing. We'll do binary search trees first, and then we'll get back to the hashing the second time around.

So map is, next to vector, probably the most useful thing you've got going on in your class library, all kinds of things that do look up based activities; looking up students by ID number, phone numbers, by name, any of these kinds of things where you need to have this key at associated satellite information. And you'd like to be able to do that retrieval and update quickly. The map is the one for you. So if you remember it's key value fears, it's all about key being the identifying mark that allows you to find the associated value. And then the value is actually just being stored and kind of retrieved without using it or examining it in any kind of interesting way. And what we're really interested in here is how to make this guy work really efficiently.

So efficiently being hopefully log in or better, if we could get that on both the update operations that add and remove things from the map, as well as the look up operation, then that would please probably all our clients immensely, if we could get both of those going well. So I'm going to build you on that. I probably have enough time to do that. That attacks it from using simple tools to start with, just to kind of get a base line for what we can do.

Vector, our old friend vector, can't get too much mileage out of vector, apparently, because you can always find a way to make it do something useful for you. It gives us the convenience of managing that thing with low overhead, direct access is good. We make a parastruct that has the key and the value together. We store it into a vector, and then we may or may not sort that vector. But if it's sorted, there's probably some good reasons. Think about what order to sort it in, what information to be using to sort that, and then we'll get value to add.

We're just going to go through doing it, rather than talk about it. We'll just go over and make a vector happen. Let's get over here in X-code. Let's take a look at my map dot H, and all I'm going to put in there is actually add and get value. And the other one contains key and remove." But these are these two operations that really matter to us, getting something in there, and getting something back out. And so I'm going to start by building this thing on vector. So let me go ahead and indicate that I plan to use vector as part of what I'm doing. So I defined a little structure that's the pair. I'm going to call that "pair" even, that's a nice word for it. So the pair of the key and the value that go together, and then what I will store here is a vector of those pairs. They'll give me a key to value; I'll stick them together in that little struct, stick them into my vector, and hopefully be able to retrieve them later.

So given that I'm depending only on vector, I don’t have any other information, I don't technically need to go out of my way to disallow then memory-wise copying, because vector does a deep copy. But I'm going to leave it in there, because I plan on going some place that is eventually going to need it. I might as well be safe from the get go. The vector will be set up and destroyed automatically as part of management of my data member. So I actually don't have any explicit allocation/deallocation, so if I look at my constructor and destructor they'll be totally empty. And then add and get value are where I'm going to actually get to do some work.

Before I implement them, I'm just going to think for a second about how this is going to play out. Let's say I'm doing a vector that's storing strings and their length. Let's say, it's just a map of string, and if I do M dot add car. I'm going to store a three with it, so on the other side of the wall, what we're going to do is make a little struct that has the word car and the number three with it, and we package it into our ray. And so then we get a second call to put something in like dog, also with a three, then we'll make a second one of these and stick it in our ray. And so on, so the idea is to have the vector really kind of manage where the storage is going. And then what we'll just do is package it up and stick it in there.

The one thing, though, that I have to think a little bit about, because it sounds like add should be pretty easy. It almost seems like what I need to do is something as simple as this. And I'll start to write the code, and then you can tell me why this isn't what I want to do. I say pair TP, and I say P dot key equals key P dot value equals val, and then I say entries dot add P. Almost, but not quite what I want, what have I forgotten to take care of? If it's not already in there; this has to do with just what's the interface for nap. What does nap say about it if you try to add for a key a different value.

So in the case of this one, it didn't quite make sense that I would add something, but let's say I accidentally put car in there with a link to the first time that I went to go fix it, right, that my subsequent to ask it store car with the proper value, three, doesn't create a totally new entry. It has to find the existing entry that already has car and overwrite it. So the behavior of mad was it could add when it was not previously existing. It also could be an overwrite of an existing value. So we do need to find if there is an entry that already has that key, and just replace its value rather than add a second entry with the same key. So at any given point, right, in this whole vector, there should be one entry for each unique key.

So I have to do a search. And let me write this code, and then I'm going to mention why I'm going to decompose it in a second. If I do entries dot size, I plus plus then if entries of I dot key equals key. Then let's break. Let me pull out I so I can get access to it when I'm done. So after this loop exits, if I is less than entries dot size, then I know that it found a match. In the case where it went through all of them, and it never hit the break. Then I will have gone all the way to where it is exactly equal to entries dot size; if it didn't then I just have a place to overwrite. And I can say entries of I dot val equals val. Otherwise I go through this process of adding a new one. I had to go hunt that thing down and replace it.

That code, when you look at it, you can say, "Gosh I feel like that code's going to be familiar." Because that idea of searching to find a match to the key probably is going to come in really handy when I need to do get value, but it has the same exact problem of oh, I've got to go find that match. So in fact, that kind of motivates the idea of why don't I just take this little piece of code and separate it out and do a helper that they can both use. I may not have planned for this from the beginning, but once I see it happening, I might as well fix it. So I can say find index for key that given a key will return to you the vector index at which that key is or a negative one. I don't know if I need to keep that one anymore. If it ever found it, it returns I otherwise it returned negative one. That can be our not found.

I can actually use that up here, and then I can say, if found does not equal negative one. Then we do that, and then that tells us that this little piece of code is going to come in handy right up here, get value if found does not equal negative one, then we can return that and what is the behavior of get value if it didn't find it? Does anybody remember? I ask it to get the value in this case for lollipop and it's not there, does it turn zero, what does it do? Untracked forget value, remember? This is an error.

There is not sort of a general case return value you can produce here that will really sort of sufficiently describe to someone who's made this call in error what happened, right? I can't return zero, because it happens to be that sometimes I'm putting in, and sometimes I'm putting strings or structs kinds of things. There's no general zero or negative one to use, so the behavior of get value is if you ask it to retrieve a value for a key, it ought to be there. So the corresponding implementation would likely also just go through the process of contains key, which we call find index for key and check. It could be used by the client if they need to know in advance that it's really there.

So let's see how I'm doing. Let's go back to my code here, and this being not my specialty, and just see that I can put something in and get it back out would be a good first test to see that things are going as they should. Oh, no it didn't like that. Oh yeah, find index for key. Find index for key was not populated clear in my dot H, so let me go move it there. I'll put it in the private section, because I don't want this to be something that is exposed outside of the implementation. The idea that there is an index and there is a vector is not really something that I would want to make part of my public interface. It's really just an internal detail of how I'm managing stuff.

Something about what I did, oh yeah, I used the right name for that. One more case let me just take a look at them both before I let the compiler tell me what is and what isn't right about them. Their both doing find index for key using linear search. If it comes back at not negative one pulling out the matching value and overriding it, and then in the case of adding or replacing the error – so the previously stored value for that is three, and if I go and I do one of those tests it's always good to know what happens if I ask it for something that doesn't exist. Making sure that the error handling that I put in there does what it was supposed to do, right? Are there any questions about the code that I wrote here?

The vector is always kind of a good starting place for these things, because actually it just tends to use very simple things, and it tends to lend itself to easy code. But because of vector constraints being at a contiguous back structure, right, there's likely to be some performance implication. And in particular, in the unsorted case here, both add and get value involve doing a linear search. In some cases it will find it quickly at the very beginning or even half way through. In other cases it won't find it at all and have to search through the whole thing. So on average, if you figure it's equally likely to be in any position or not there at all, it's at going to at least have to look at half of them or more. And so we can say they're both linear.

If we change it to be sorted, which is the first obvious improvement to make here, then get value gets a lot faster, because we can take advantage of binary search. I sort it by key, so ignoring the value, values are just satellite data. But stored in order of key, then all the A words, B words, C words, D words, what not, so walking it down the middle and seeing which half you look at, then looking at the quarter of there and the eight of there. It's going to very quickly narrow down on where it had to be if it was there or if it wasn't there at all. So we can implement that in logarithmic time, meaning that if you 1,000 entries, it takes ten comparisons to find it or agree that it's not there. If you double that it will take one more comparison. If you go to 2,000, negligibly faster, even numbers in the millions are still just a handful of comparisons to work it out, and say, to the twentieth for example is roughly a million; so 20 comparisons in or out, and so it didn't look at a million things a million times over.

However, that didn't improve add, why not? Why does keeping an assorted order not have the same boost for add as it did for get value? Can I do the same logarithmic search?

Student: You have to move all the elements –

Instructor (Julie Zelenski):Yeah, you can find the place fast, right, you can say returning the word apple it's like oh, okay. I can very quickly narrow in on where it needed to be. If it was there I could override it quickly, but in the case where it really needed to do an add, it's got to move them down. So the old add did all its work in the search and then just tacked it on the end if it didn't find it, or overrode the middle. But the new add also has to modify the array to put it in the right place, and putting it in the right place means shuffling to make space. So even though it found it quickly, it was unable to update that quickly, so it still ends up being linear in that case.

Now this actually not a terrible result, as it stands it's probably much more likely that you're going to load up the data structure once and then do a lot of searches on it, then infrequent additions. That's a pretty common usage pattern for a map. It's kind of like you build a dictionary, right, loading the dictionary once could be expensive, but then people get tons and tons of look ups of words later. But you don't change the definitions a lot later or add a lot of new words. So this might actually be quite tolerable in many situations that the building of it was expensive, but it gave you fast search access. But really what you want to do is say, can we get both of those? So driving to say what we do with the editor buffer, you say, "Well, what if I just want both of those operations to be logarithmic or better, what can I do?"

So I'll leave you with kind of just a little bit of a taste of where we're heading. Just to think a little bit about what the link list does and doesn't buy us, right, that the link list gave us flexibility at the editing, the local editing site, but it doesn't give us fast searching. Well maybe we can take this link list and try to add searching to the idea of this flexibility to kind of get the best of both worlds. So that's what we're going to start on Monday, building a tree. So if you have time today, I'd love to have you come to Truman, and otherwise have a good weekend, and get your PQ work in.

[End of Audio]

Duration: 48 minutes