Module 18 Week 10 Stacks and Queues

Stacks and Queues – Intro 1.2

In this module we're going to talk about a couple of data structures the two data structures that we're going to talk about our stacks and queues. And these really popular data structures in computer science and we use them over and over and over again. So we're going to talk about first the stack we are going to explain what it is and how it works and what it's designed for and then I'm going to talk about some uses that we could apply the stack to and then we're going to talk about queues. We're going to talk about the same thing what it is and how it works and what it's designed for and then what we use queues for. So coming away from this you're going to have a good look at two very fundamental data structures in computer science and you should understand both how they are built and how they are used.

Stack 1.3

So our first data structure we're going to talk about is a stack and the way that you can best understand this is if you think about going to like a buffet at a large restaurant and at the beginning of the buffet there's a stack of plates. And the plates are nice and warm they're recessed into the table so the only plate that you can grab is the very top plate in the stack. So here we have this stack of plates it's recessed into the table and all you can grab at is the very top of the stack. And that's one of the purposes of a stack you can't access arbitrary information in the stack. So once you put things into the stack you can access only the last thing that's been put into the stack and then on top of that when it comes out of the stack you get the last thing that was put in so this is what's called a last in first out or first in last out depending on who you talk to. Which is abbreviated as either a FILO or a LIFO or a buffer. So the point here is that when we put things into the stack the last item that we inserted is the first one that's removed from the stack. Now the functions that we use on a stack are push and pop and top and those are the common functions that we use. Push to put something into the stack pop to remove that top element from the stack. We have a couple of supplemental functions that do other purposes of course like clear and is empty and size. You've probably seen these before they're pretty common clear throws away everything that's on the stack so if you just want to throw the whole stack away we can call clear. is empty tells us if there's anything left on the stack so it will return true if there is something at least and size will tell us how many things there are left on the stack. So we're going to work with these stacks and we're going to design how the stack works in the next slide.

Stack – How it Works 1.4

So the two fundamental functions again are push and pop and of course we'll have top and that just accesses the highest element on the stack. But the push function is going to add to the front of the stack and the pop simply removes from the front of the stack and you can do this for the back also it really doesn't matter. But the point is that the push and the pop are inserting and removing from the same place so that means that when we do the latest push the next pop is going to cause that item to be to come out immediately. So we don't have to or we are not looking at further information down in the stack now of course when we implement it will have to implement all those supplemental functions like clear and is empty and size. And we'll also have to implement the big three if we're talking about using an array. Now for this demonstration we're not going to use an array. We'll actually use the STL list class which already has the big three already built but I just did that so that it's a little bit easier for us to understand we don't have to worry about the complexity of building the big three.

Stack Storage 1.5

So we're going to have to store all the information from the stack so the user is going to give us a ton of information through push operations and we're going to have to put that information somewhere and the two data structures that we've worked with so far that we can use as fundamental storage are the list and the array. So we have to decide if we want to store the information that's given to us as a list in which case we have a linked list with each node pointing to the next node or as an array in which case we have sequential storage of a limited size where we can add on. So if we take a look at the comparison between the two methods that we could use and we look at each individual function. You'll see that the list for example the push operation is a constant time

operation big O of one. Because every time you add to the list all we're going to do is create a new node we are just going to put it on the front of the list and we're going to link it to the next node in the list. So a push operation is actually very simple and it doesn't take anything more than constant time we don't care if there's ten thousand of these elements already on the stack the next one that we're going to push isn't going to take any additional amount of time. As a comparison we have the array and if we do a push on an array sometimes it's constant time if we have enough space left on the array and sometimes it's going to be a linear time of the big O of N. And that's because we may have to expand that array if we ran out of space we may have to expand that array to create more space. So sometimes it's going to be constant time and sometimes it's going to be linear time which creates a little bit of inconsistency. If we need that consistency then that would negate the possibility of using an array.

As far as a pop it's always going to be a big O of one it is always going to be a constant time removal from the list we just throw away the top node. From the array we could just reduce the size by one the top element accessing the top element should always be constant time and that's really fundamental that's important we want that. From the list we would just access that top element through the head pointer from the array we would just access the zero element or if we wanted to size minus one element. For clear it's always going to be constant time for the array but it's going to be linear time for the linked list and then with is empty we always have just the size parameter we can always check to see if the size zero. And for the size well the array of course is going to be a constant time. How can we make the linked list be constant time? Well since we know how many push operations there is going to be we can simply use the size parameter we can create an integer and store that size parameter so that we always have it in a constant time access so we won't have to run through the list and actually calculate how many elements are on the list. We'll just keep an integer in the in the class that records how many elements are actually in the list at any given time. So if you take a look at these we can compare easily how the list works with how the array works and we have now the tools to understand the performance benefits. So we can choose between either the list or the array.

Stack Storage – Continued 1.6

So now we've seen how long each of the functions are going to take we can make a decision about whether we would use a list or an array to do the actual storage underlying stack. And it's pretty obvious at this point I think we would make the decision to do a linked list because we're seeing that we're going to do a lot of pop operations a lot more pushes a lot more pop operations than we would the clear operation. So it's relatively unlikely to do a clear as compared to a pop operation. Remember that the clear just completely dumps everything whereas the pop operation would remove one element. So to that end I think it makes considerably more sense to choose to use a linked list as the underlying storage mechanism for the stack rather than to use the array. And this is a great analysis that should be done for most of the data structures or for most of your programs that you're going to do make the decision as to whether or not you want to use A or B and play out the analysis. So that's just one of the points that I want to make is is it's not clear immediately whether we should use an array or list but once we flesh out what's actually going to happen on the on the stack you're going to see that the linked list actually plays out a lot better. So for this purposes we're going to use the STL list class we are not going to build it ourselves. So that's fine we know how to build a linked list we did that in a previous module so you don't have to worry about that. And the STL list includes all of the big three and includes all of the features of a linked list class so we can deal with just the stack portion. For our purposes the push function is going to call just simply push front on the STL list the pop function will call pop front. So we're going to push on to the front of the list and when it comes time to remove we're going to pop off the front to the list so we're popping the most recently pushed element. The other functions we're just going to map to the equivalent STL functions. So there shouldn't be very much code as far as we're concerned creating this.

Stack Code 1.7

So here's the code for the stack and you can see it fits easily in a couple lines it really doesn't take up a lot of space. We have as a private data member we have this data member called data which were storing a linked list of T objects. So we've included the STL list here and you can see that we're just using that underlying storage just as a mechanism to create the stack. The push function like we said it's just going to do a push front the pop function is

just going to do a pop front. So we can push items onto the front of the linked list and we can remove items from the from the linked list. Top just returns the top elements of the linked list. Remember the data dot begin is actually an iterator so we have to dereference it and that's the first element on the list. We can assume that there's something on the list if somebody asked for the top element. We can assume that there's something on the list already so we hope that this isn't going to deref null or cause any problems there. For is empty we're just telling C++ to return the size whether the size is equal to zero and for the size which return the size. When it comes time to clear we just simply call clear on the underlying data structure. So you can see that the code doesn't really have a lot of beef to it it's very relatively easy. All we're doing is inserting elements on to the front of the list and we're removing elements from the front of the list. We know that both of those functions are constant time function so insertions on to this stack are only going to take constant time. Of course accessing the top element of the stack is only and it's a constant time in fact if we take a look very carefully the only one that's not going to be constant time is that clear function which will be linear. But we aren't too concerned about that because we know that that's not going to happen terribly often. So the code is relatively easy to understand relatively easy to go through. Take your time taking a look at it and if we wanted to use this we would understand that we just have to create a stack and call push and pop as we see fit.

Stack – What is it Used For 1.8

So now that we've seen how to create the stack. How do we use the stack or what do we use the stack for? Well one of the most fundamental components of computer science of course as we see is the compiler and we're using the compiler extensively for all of our C++ work and there's a lot of other programming languages that use compilers. Compilers have a lot of places where they use the stack but one of the most fundamental places is pattern matching. So that when we have a piece of source code we want to make sure that for example all the opening parentheses correspond with all the closing parentheses and all the opening curly braces correspond with all the closing curly braces. And all the opening square brackets correspond with all the closing square brackets and it's not just a matter of counting how many openings we have and matching it to the equal number of closings. It has to be precise so that when we have an opening it matches with the closing both in number and in order so that's a really important factor. If we have an opening curly brace followed by an opening parentheses and then we have a closing curly brace before we have the closing parentheses we have an invalid syntax there because we haven't closed the parentheses before we close the curly brace. And the easiest way to deal with this is to create a stack and when we encounter an opening we can push on the opening symbol on to the stack. It's going to be stored there and remember that because this is the first in last out buffer what we're looking at on the top of the stack is the last thing that we inserted. So if we have an opening parentheses and then immediately a closing parentheses. We're going to see that the opening parentheses matches the closing parentheses. And it doesn't matter what came before it or what comes after it. We have to have that matching opening with that matching closing. So what we do is when we find an opening we push onto the stack and when we find a closing we compare it to the top of the stack and if it's a match then we can pop the top of the stack we can remove that opening element and say let's go to the next element and see that the next element matches. And the next element might be an opening or might be a closing. So the stack is used to do the pushes and the pops to make sure that the last thing that we see is the first thing that we see in the in the closing.

There's also a couple more examples of where we would use stacks for and that is in math operations it's very difficult to deal with infix operations using using a programming language like C++. So if we have even a simple task like two plus three times four we recognize from a math perspective that the three times four portion has to be done before the two plus three portion because that's order of operations. But the computer has a little difficult time because we're going to be scanning that from left to right. So there's a different type of notation that's called postfix notation and what we can do is take the operators the plus and the multiply and we can push them on the stack and there's a number of different rules about what you can push on top of something else you can't push a lower precedence operator on top of a higher precedence operator. And then we can order this so that two plus three times four actually turns out to be two three four times plus which makes it very easy for the next thing that we use stacks for and that's for converting a postfix notation into an actual value. We take the two the three and the four and we push those on to the stack and then when it comes time to do the multiplication we pop the last two elements off of the stack so we have the three times four so that makes it very easy to do order of operations

without having to worry about infix notation. We've converted it's postfix notation. Both of those examples we use stacks in the conversion we would use the stack to store operators and in the evaluation we'd use the stack to store operands would use to store numbers and when we get done we can push the number back onto the stack and when we're completely done with the evaluation the top number on the stack which should be the only number on the stack is the actual value. So we can see that in the examples in later on in the course but you'll see the infix to postfix conversion and you'll see postfix the value of valuation and it's done with stacks.

Stack – What is it Used For 1.9

So here's a piece of C++ code and I just wanted to show you how this works when we use a stack. So we're going to take the code we're going to parse it character by character. And what we're doing is we're saying if we see any parentheses or curly braces or square brackets we're going to push those onto the stack of their opening. And if they're closing we're going to compare them to the top of the stack make sure it matches. So we can see of course is this actually is an acceptable piece of code of course, we're going to take first the characters int main we're just throw them away because they're not relevant to our discussion. And we're going to take the parentheses we're going to push it onto the stack and when we see the close parentheses, you're going to discover that on the top of the stack there is an open parentheses and so that matches and the open parentheses would then be removed in the stack would be empty. We then encounter the opening curly brace and we'll push that onto the stack. Some more characters that we just completely throw away. And then we have an opening square bracket which will push on top of the opening curly brace. And then we have a closing square brackets so we throw away the openings square bracket; more characters that we throw away. There's an opening curly brace, and we have two opening curly braces on the stack some characters that we throw away, and then we have a closing curly brace. So we match it with the opening curly brace that's already on the stack and throw both of those away. We throw away all the other characters leading up to the opening square bracket, and the opening square brackets pushed on top of the opening curly brace that's already on the stack and then we have an opening parentheses, which gets pushed on top of the stack. And you'll seen at that point in the compiler there's an opening curly brace and opening square bracket and an opening parentheses. And then we have the closing parentheses so the opening parentheses comes off the stack. We have the closing square brackets so the opening square bracket comes off the stack; we throw some characters away. We have the closing curly brace and we throw away the opening curly brace. We're done with the input; the stack Is empty. This code is great. If anything doesn't match up, we can throw an error and we can actually tell the user tell the programmer roughly where the error is. For example, missing square bracket at line blah blah blah, and I'm sure you've seen that in your experience, this is the exact way that your compiler has generated that error.

Queue – What is it 1.11

Now that we're comfortable with the stack, let's take a look at our second data structure which is the queue. And the queue is our FIFO data structure: it's a first in, first out data structure, which means that whatever item is first in queued, is the first item that's de-queued. And this is exactly what you have at a line, what we call in the United States we call it a line, in Europe they might call it a queue which is exactly what we're demonstrating here. when you arrive at the bank if there's a line they have to get on you're going to enter into the line and you're going to wait until it's your turn. if we used a stack for something like that the first person to arrive in the morning might not be serviced until the very end of the day, but in the queue it's much more fair. The first item that comes into the queue is the first item that goes out of the queue; so they go out in order as opposed to in reverse order. the FIFO buffer or the queue is often use for storing information temporarily and then releasing it later so called a buffer. the functions that we have you might see them called N.Q. and D.Q. and top, but a lot of people refer to him just as push and pop and top the same as they do with the stack. So just be clear of what functions you're using and what data structure you're using, because it is possible you might have some overlap between the names push and pop. Of course, the supplemental functions that we're going to have are still the same as the stack; we're still going to have the clear, we're still going to have the is empty, and we're still going to have the size. and you can see that the elements that get inserted get inserted to the back of the queue and when they get removed they get removed from the front of the queue. So it's our simple FIFO data structure for just temporary storage of information.

Queue – How it Works 1.12

So if we do this, we're going to consider that we would either use a linked list again or an array and we'll talk about the performance differences between those in just a minute. But what we'll do is we'll add to one end of the data structure and we'll remove from the other end of the data structure. So, let's say enqueue is going to add to the end of the list or of the array, and dequeue is going to remove from the front of the array. Of course, we can do that backwards doesn't matter which way we choose, it's a personal preference thing. But for the purposes of demonstration we're going to say that enqueue adds the end and dequeue removes from the front. The supplemental functions, of course, we need to write those and we need to provide the big three. And again we'll use whatever STL structures are available either the vector of the list. So we don't have to write all these functions ourselves.

Queue Storage 1.13

So again we have the breakdown of whether we use a linked list or whether we use an array to make the decision on what we should use of the underlying data storage structure for our queue. so if we take a look at the list and we're breaking it down by function here. we take a look at the push function and of course the push function on the list; it's still going to be constant time. Now we can say it's going to be constant time because we're going to have pointers to both the head in the tail of the list and we're going to make sure that we can insert at either the head or the tail for us, for our purposes, we'll be inserting a tail and will be popping from the front but that's overly unimportant. the array, again, for the push operation sometimes it's going to be constant time and sometimes it's going to be linear. So the problem here is again if we run out of space in the underlying data structure of the array, We're going to have to expand the array and that takes a linear amount of time. so we have that trouble still.

Now pop on a list, we've said it's constant time and that's just removal of a node from the list, so we're just going to have a simple delete operation. but Pop on an array is really problematic, because we're going to be pushing out to the end the bottom of the array; it means we have to remove from the front from the top of the array. and if we remove the zero element from the arraign we're going to have to move all the elements upwards by one. So ultimately the pop operation is going to take a linear time and that right there is incredibly problematic. We'll come back to that in just a minute.

the top operation on either of them is going to take constant time, the clear operation to get on the linked list is going to take a linear time but the array is going to take constant time and is empty in size are all going to take constant time. So if we come back, circle back to thinking about that pop operation. You can tell that on a range of a very large size we can see that there's going to be a very big problem because removing that top element from the array Is going to cause us to have to shift everything up by one element inside the array. and that's a very very costly endeavor so it's going to take us a long time to do that. there are mechanisms for dealing with it, for instance we could record where the start of the array is instead of assuming that it's at the zero position but that runs the potential of wasting space in creating a more circular array. so there are mechanisms to dealing with this but ultimately what I think we're coming to terms with is that it's easier and more efficient to store this as a linked list.

Queue – Continued 1.14

So we've come to the conclusion, I think, that we are going to use a linked list again for this data structure. And the implementation here is going to be really similar to that of a stack; we're going to have just a couple of minor changes. Now, instead of inserting into and removing from the same point, either the front of the back, we're going to be removing or inserting from different points. So enqueue is going to insert on to the end of the dequeue is going to remove from the front of the queue. We really have to make sure that the underlying data structure of the linked list has both head and tail pointers, so that way we can do those constant time insertions. If we have something simple like a singularly linked list where we only have a head pointer, then insertion is going to require us to iterate or recursively go through the entire linked list so that we can find the end and insert on to the end

and that's too costly. It's far simpler to keep a tail pointer in addition to the head pointer so that we know where the end of the list is and then we can insert on to the end of the list, and we can remove from the front of the list.

So it's very possible to use a singularly linked list in this structure. We're going to just, again, use the STL list class to make it easy. But the STL list class is a doubly linked list and of course there's overhead in that. If we want to save some memory, we could create a singularly linked list where we have the head pointer pointing to the next element, in other words the oldest element in the list, and we can have the tail pointer pointing at the last element. When we do a removal, we just advance the head pointer to the next element remove that first element and when we do an insertion we just make the tail pointer point to the new node and then we can connect the previous last node to the new node. So we have options for doing the queue as a singularly linked list. But for our purposes we're just going to use the STL list class, so that it's easier and we can demonstrate it and it has all the big three already built into it so we're not really worried.

Queue Code 1.15

So here we have the code for the queue class. And again, you can see it looks very much similar to the stack class. We have the data storage as our private data members, are only private data member. It's a linked list of type T, so we have a templated and we have the enqueue and the dequeue functions, where we push on to the back of the linked list and we pop from the front of the linked list. Ehe top element just returns again, the iterator, the dereference of the iterator. And then the isEmpty, int size and the clear actually completely unchanged from the stack versus the queue. So the fundamental difference here is that the enqueue function is going to push on to the back, whereas the dequeue function is going to pop from the front. And by doing that we've changed the way that we're inserting and removing into this list and that means we go from having a stack to having a queue. First in, last out for the stack; first in, first out for the queue, and that makes all the difference in the world.

Queue – What is it Used For 1.16

So now that we understand how a queue is built, what do we use it for? The fundamental way that we use queues as storage buffer, so there's going to be a lot of situations where we're bringing information in. For example you've already worked with the IO stream and, whether we're talking about a file stream or whether we're talking about keyboard input, we are talking about bringing in a large amount of data at first, storing it temporarily, and then slowly feeding it into your user program. So in the case of reading a file, because it's an expensive endeavor to actually read data from the secondary storage device, what we're instead going to do is grab a large portion of the file, perhaps all of it, store it temporarily in a structure in main memory, and then as the program asks for it it's going to be fed in slowly. Now the best structure for that of course is a first in first out buffer, or a queue. So there's one situation is as there. We might use it in operating systems, later we're going to talk about memory management, and we're going to talk about removing the oldest page of main memory for example. And how do we know which one is the oldest? Well we keep a FIFO list, a FIFO queue, of all the pages that are in main memory and when we need to throw one away we throw away the front of the queue, so there's options for that. Really anywhere in computer science where we need in an ordered list; where we have first in first out property. That's where we're going to use a queue. So there's a lot of situations that you're going to see throughout your career in computer science that you're going to use a queue and they are a very popular structure.

Queue Image 1.17

So here we have just a simple sample of how we would use a queue. We have a couple of calls to the enqueuer function; we have values of twenty, fifty, and thirty. And when we insert twenty, or when we enqueue twenty, it's going to be the only thing that's on the list so it's pointers going to point to null. But when we insert fifty, the next pointer of the twenty node is going to point to fifty, meaning that twenty is still the head of the list or the head pointer still points to the node that stores the twenty. But the next pointer inside the twenty node is going to point to fifty, and so to the same happens when we have the thirty. When we start to dequeue these objects they come out in the exact same order as they went in, which is of course different from the way that the stack works. But that's because this is a first in, first out buffer, instead of the first in, last out buffer.

Stacks and Queues – Conclusion 1.18

So in this module, we took a look at the two fundamental data structures in C++ that we use stacks and cues. And we saw how they work with how they were implemented and what they're used for. Hopefully have a better understanding of how a stack works, how a queue works, and you also can produce one if you had to yourself. Now, fortunately, of course the STL took care of all this for us so if we needed a stack we could just pounding include stack and we get stack, and if we needed a queue we could just pound include queue and we end up with a queue. And all the features are there; all the functions are there. We have to push, the pop, the enqueue and the dequeue, and you can use those without having to build it yourself. But even if you had to build yourself, you realize now that it's only a couple of lines of code and it's not really going to take much to store it. So I hope you enjoyed that module and we'll see you for the next one.