Week 9 Module 17 Linked Lists Script

In this Module 1.2

In this module, we're going to learn about linked lists and what a linked list is and why we need linked lists. And then we're going to talk about templates for classes and functions, so that we can work with any data type. And then we're going to talk about how a linked list is designed and what we use linked lists for. And we're going to use linked list extensively for the rest of the semester here. In the later modules, you're going to see a lot of implementations of linked list; so we just want you to have a background for understanding what a linked list is and how you would use it and when you would work with it.

What is a Linked List 1.3

The linked list data structure is one of the fundamental data structures in computer science. It's basic idea is that it has two parts each: it's made up of nodes and each node has two parts. The first part is a data section which could be any data type, but that's the component, that's the element that we're going to store. If we were talking about an array this would be the individual item in an array but it's not so keep that in mind. So we've got this node that has a data section and it also has at least one pointer, if not multiple pointers to the next sections or to the rest of the list. And the idea is that the pointers are connected in a form of a chain, so that we have a pointer to the first node in the list and we call that the head pointer. We record the head pointer to the first node of the list and then the first node of the list's next pointer will point to the second node of the list, and the second node of the list's next pointer will point to the third node of the list, and so on and so forth and so on and so forth.

When we get to the end, we have what's called a null pointer; so we have the null which is an indication. So when the next pointers pointing at null that's an indication, that this is the last node on the list. So these nodes are tied together through one direction at least, where each node points to the next node. Now since this is only the most basic form of the linked list, in reality what's usually stored is both the next pointer and a previous pointer; so you may end up having two pointers in the in the linked list structure and of course the previous pointer just points at the node before this node, and of course the before the first node would be a null pointer. So it's sort of like a linked list in reverse: we've got the head pointer that points of the first node, the first node points at the second node, the second node points at the third node and so on and so forth. So this is the idea of how we can store a large amount of data in a linked list and we're keeping them in individual nodes; remember these are node pointers and they're pointing at individual nodes.

Linked List Visual 1.4

So what you can see here is a visual representation of the linked list. We have a head pointer that points at the first note and inside the node there is an object called a data section and the next pointer the next pointer points at the next node. So the pointers are all the same pointers; they're list node pointers. That's really what we're going to see later on in this and this module. So the objects that are being stored are whatever data type you'd like, we'll see how to do that later with templates, whatever data type you'd like. And we have this head pointer that points at the first node, the next pointer points the second node, the next pointer the second node points at the third node, and the next pointer of the third node points at null. So there's three nodes on this list. There's three data items on this list you can call them: one, two, and three, if they were integers, and you can store in them whatever integer you want. Now this list might also have a tail pointer to point at the last node and that's functionally dependent on what we decide to do with this linked list. So, we may end up needing a tail pointer and if

we do then we can have that as well to point to the last node in the list. And that's just for easy access if, for example, we want to expand on the list by adding a node then we can do that easily through the tail pointer or through the head pointer.

Why Do We Need Linked Lists 1.5

We've been working with the arrays up to this point, and they array data structure works great for how we'd like to store information. Unfortunately there are some limitations on arrays; if we were doing insertion into an array even if the insertion that we're doing is on to the end of the array, we have to make sure that there's space in the array available. If there isn't space available, we have to expand the array as we saw in the pointers and dynamic memory discussion. We have to expand the array and doing so takes big O of N. Regardless of that, if we want to do an insertion at the beginning of an array or if we wanted to an insertion in the middle of an array, we have to push back every element that's beyond that insertion point. So we have to go to the last element and move it back by one, we have to go to the second to last to move it to the last element and so on and so forth. That's a severe limitation on arrays in that we can't insert into the middle of them without a big O of N process.

But we don't have that problem with a linked list with a linked list. We can do an arbitrary insertion on to the beginning the end or anywhere in the middle in constant time we can do it in big O of one. And the reason for that is because these pointers keep track of where everything is going. And as long as we're suing that it's being stored in main memory, which it usually is, if we're using main memory that's ram we can access any node at a constant time. So we don't have a problem with being able to insert into the middle of the linked list.

Now unfortunately, the downside of the linked list is that in order to go through to find any node in the list we're going to have to use a linear searching algorithm so it's going to be big O of N. So we can't arbitrarily go to the third node of a linked list like we could with an array. With an array, if we want the third node of an array we simply go to that memory location and access the third element. But with a linked list to get to the third node you have to go through the second note, and to go through the second node have to go through the first node and so on and so forth. So we have these differences in access and insertion times and each one serves its own purpose. If we don't need arbitrary access to the middle of a linked list or the middle of the data set then a linked list would be a perfect way to store that. But if we do need arbitrary access to the middle of the dataset then a linked list is not a great idea. So it's really just playing the averages of what we expect to be doing with these things that we can make a decision on to whether we should use a linked list or an array. Array insertion is big O of N; linked list insertion is constant time. Array access is constant time, whereas linked list access is linear so you make the decision. Now linked lists can be reorganized; they can be merged. They can be broken apart into new lists. These operations only take constant time because we're just manipulating the pointers. linked lists don't require any overhead for storage. When we were working with vectors, we had to make sure to leave a little bit of extra room in case of expansion. With the linked list, we have no need for that so all we have of those pointers and they only take up four bytes for each node so hopefully it's not too significant a factor. But the idea here is that we can make a decision consciously about whether we choose an array or a linked list to get the job done.

Working with Templates 1.6

Before we get too deep into linked lists, I have to talk to you about something else that we need to go over first. C++ has this idea of templates and a lot of languages have these now; they might call them

generics, they might call them templates. But the idea is that we don't really know what data type we're going to be working with.

So in C++ we can template; we can use a template when we don't know what the data type is of one or more elements. Before each function or class that we're going to be working with, we have to put this indicator this is a template class T. And T is just my individual use, you can really use any name but I just used to see all the time so that's what I normally consider. And that's a data type that we don't know which data type it's actually going to be in the end. What C++ plus does is basically a find and replace through that function to replace T with the appropriate data type, whatever the data type is. So I've got here a function called my swap, which is going to be a templated function and it takes in two items and I don't know what data type the items are. It could be two integers. It could be two doubles. It could be two floats. It could be two strings. It could be two elephants. What it can't be is one int and one elephant, so we can't mix and match data type. T has to be T consistently whatever the data type is.

So here we're going to swap two items, meaning we're going to make the value that was in A into B. And we're going to take the value that was in B and store it in A, and the code for that is overly simplistic; it's just three copy operations. But the real takeaway here is that you can use any data type for this my swap function. And in C++ you would just call my swap and give it two integers or two elephants, it would be fine with that and you don't need to rewrite this function numerous times simply to change the data type. And that's a huge benefit when we get to working with larger classes, particularly with storage of items where we don't know what data type is going to be stored in really we don't care what data type is going to be store; our job is to simply store the data type. And I imagine this the way I tell my students to envision this is, imagine if you're running a storage company, like the big the big storage companies here in New York where you can rent out a room. The company doesn't care what you put into your room. I guess they might care if it's going to be dangerous but they don't care what you're going to put into that room; they simply care that you own the room. And that's exactly what we're doing here. We're saying we'd like to the ability to swap two things and don't worry about what those two things are.

Templated Classes 1.7

There's a little bit of extra work to be done when we template a class. So when we template the class, the name of the class actually changes to incorporate the template. So here I've got a simple description of a class called “sum valve.” Where we've templated it to store a simple data item and this is leading up to what a linked list node would look like and we'll get to that in a later slide. But basically the sum val class stores one data item, and it's got a function for get val and for set val, and we don't know what the T data type is and really we don't care what’s the T data type. And now inside main what we do is we’d create this as sum val int and that's where you've seen before, that's where a vector comes from. So you've done this before with vector and now you understand why; it's because vector is templated. So we have to tell it what data type we're working with inside the sum val object, so when we create it we say sum val int var name or sum val char var name or sum val whatever var name and you're giving it a data type to now store inside some val. Now the one thing to take notice of here, is that every function that we create has to be templated. So the set Val function, for example, has to be a templated function because first off it has a parameter of type T. But also because the class name is now changed to some value less than T greater than. So we can see that the actual name of the, of the class changes to incorporate the data type. And that's just an internal C++ representation but we do have to take note of that for any time we're going to work with a templated function inside of a templated class.

Designing a Linked List Node 1.8

First thing we have to do and we're really designed to classes here. But the first class we're going to design is a link list node class and of course can be templated because it's going to have to store the data item and we don't know what the data type of the data item is. So immediately we recognize that we have a templated class. So the nodes are templated. And what we're going to be storing here is the T data item and that is really the data that's going to be stored inside each node, so this is the data storage that we've got. The next thing that we've got is a pointer to the next node and if you pay attention to that, if you look carefully, you're going to see that this is an eldest node T pointer. So you're actually got a pointer to a node when we're creating the node. So yeah. This works because we can create a pointer to the class object that we don't have yet created, so we've created the LS Node class and in the process of creating the LS node class we need an Ellis node pointer. But that does work; C++ was allowed us to do that.

The next thing that we have is just a simple constructor and the constructor I just made to make life easier. It's really it's not necessary, but it's yet very useful in certain circumstances that you'll see. For example in the recursive copy operation that we do later on, we use the LS node constructor so that we can pass in either a data item or we can pass in a pointer to the next item and then of course it constructs those appropriately. So you have to look back to the object oriented lesson, if you need assistance on figuring that one out.

Then we also have a friend class, which is the LS class. The LS class should have access to everything inside the Ellis node. So we'll make the whole class a friend of this class so that all the functions inside the Ellis class do have access to all the private data here inside the illest Node class. So it basically makes this a lot more accessible and makes it a little bit easier. It's not necessary. We could do getters and setters but there's not really a great reason to do getters and setters especially because the Ellis class is going to need to manipulate those next pointers; so it can get very complex. So that's why I do the friend there. There's not much to this class; it's a very simple class. And in fact, in years past it was actually implemented as a struct with no functions because the only function that we really have is the constructor. So here we've got a very simple class to take care of the LS node and it does store data item so we're going to have to template it.

Designing a Linked List 1.9

Here's the basic format for the linked list; there's not much to it. It is a little bit busy here but we'll go through it step by step. The first thing that you are going to see is that there's a head pointer and really that's the only thing that this entire class needs to store; a pointer to the first node in the list. And of course it’s an LList node pointer, it looks exactly like the next pointer that we saw in the class, just a moment ago. But here we've got a pointer to the first node in the list. Now of course if that head pointer is null then that means there's nothing on the list, so that's going to be our indication for an empty list. And if you look down into the public section first line in the public section, you'll see that that's what the default constructor actually does; it just sets had pointer equal to head equal to null pointer.

We've got a copy constructor and assignment operator and destructor; that takes care of our big three operations. So those are our handle on the next three lines, because we are working with pointers, we are working with data storage that's on the heap, and we are going to have to make sure that that they had a storage is consistent. So again look back at your pointers and dynamic memory discussion to look at how the big three plays into a factor here.

The insertAtHead function is simply going to update the head pointer to point at the new node. So to do an insert and head all we have to do is set the head equal to a new node and make that new node's next pointer equal to the original head. So that means keeping track of what the original head pointer was maybe it was null, maybe it was a node. But if we're doing inserted head all we have to do is create a new node and insert it physically into the list.

removeFromHead is sort of the same thing which had to advance the head pointer to heads next. isEmpty is a simple function that just checks to see if the head pointer is null. Clear means we're going to have to eliminate these nodes and the easiest way to eliminate all the nodes is just constantly call remove from head in a loop until isEmpty. So it actually turns out to be while not is empty remove from head and that's it; that's the entire clear function. insertAtEnd we're going to see a little bit later on takes a little bit more effort and then insertAtPoint is almost the same sort of thing. And then we have a size function to tell us how many nodes there are actually on the list, which we're going to see in the next slide also.

So here is the basic format for the linked list if you boil it down, it's really just a head pointer and that's the entire linked list. And before we had object orientation back in C, linked lists were created with a struct for the node and then you would just simply keep a pointer to the beginning of the linked list and passed that pointer around. So we could do this all without a class, it's just having the class keeps it all encapsulated which is the point of a class together and makes it a little bit easier to work with. But there's not a lot of effort here and we get a lot of benefit.

Stopping at the End vs. Going off the End 1.10

One of the problems a lot of students have with linked lists is the idea of whether they should stop at the last node and do some more, or whether they need to go off into the null. And this often comes down to a lack of understanding with when to stop. So I'd like to go over that at this point. Both solutions are going to use a while loop, but the conditions inside those while loops are going to be slightly different.

Stopping at the end means that we want to run the while loop as long as the next pointer in the node that we're looking at is not equal to the null pointer. Now that requires that we keep a little bit of track of the current pointer. And it also requires that we're very careful that we make sure that we have at least one node, because if we start a temp pointer at null and we say temp pointer is next, then we've just de-ref null and the program crashed. So we've got to make sure that the linked list has at least one node and if we can do that then we can advance that temporary pointer until we reach the very end of the linked list, the very last node in the linked list.

However, if what we want to do for example is count the number of nodes on the list then we want to include that last node and we want to go until the temporary pointer equals the null pointer. Here's the biggest problem: we can't go backwards. Once we've reached know all we can't take a step back and move to the last node, so we've got to decide from the beginning whether we should go and stop at the last note or whether we should go off the end of the last node because we don't even need it. Here's a here's a size function so here's the function that size and what it does a starts a simple counter at zero sets the temporary pointer equal to head and runs that while loop until the temporary pointer gets to null pointer. It really is just advancing that temporary pointer; you see that line of code that says temp equals temps next, temp arrow next that line of code is going to advance that point or node by node and we're going to update the counter. When we're done temp is going to be equal to the null pointer and

we're done, we've counted all the nodes. So if you take a look at a couple of views of this, if the list is empty to begin with then temp equals no all temp equals had causes temp to be null, we don't do anything in the while loop and we will return zero which makes sense because list is empty. So that's the simple idea of how to do a size function but if we want to see how to do some work at the end like an insertion at the end assuming we don't have a tail pointer. Then the first thing to check of course is to see if the list is empty and if the list is empty, then an insert at the end is the same as the insert at the head. So let's just go ahead and use that code again so we can reuse that inserted head code. But assuming that there are some nodes on the list, we're going to have to go search for the last note on the list and that's what this… That's what this function is going to do here. The temporary pointer points at the new node that we're going to create. And now it's time to find out where to insert this new node so we have the end pointer and calling end pointer and we say while ends next is not equal to null pointer then we just advance that and pointer.

Of course this is a linear time problem, it's big O of N to find the last node in the list and then we can just update the next pointer of the last note on the list to point to the temp, to point to the new node. So what we've done is we've created a loop that finds us the last node and it's very different from the loop before it which counts the last node, or counts all the nodes including the last node. So we've got to keep in mind that those conditions are quite different in those while loops and to use the appropriate one at the appropriate time.

Recursion in Lists 1.11

Going back to discussion on recursion that we had a few modules ago; you're going to see a lot of recursion here in linked lists. It's often used in linked lists because if we look at a sub list, if you look at a list that doesn't necessarily start at the original head, you're going to see that it looks exactly like the larger list and in fact, even the empty list at the very end will look like a regular old list. So recursion is a real big popular topic here in linked lists because everything can be done using recursion or almost everything can be done using recursion.

So here what I'd like to demonstrate is the recursive copy function and what I did was I just simply have a pointer. So for example the assignment operator or the copy constructor might call this to make a copy of the right hand side list and it would be passed the right hand side head node pointer. And if you look at the only thing it does is check to see if the right hand side had nodes pointer is null. If it is then we return know it's an empty list we're done. And if not we can construct a new node based on the right hand sides data, of course, so that makes the actual copy of the data and then call recursive copy for the right hand sides next. So we're using the constructor here, that we created in the list node class to our benefit, so we can construct a list node based on the data and the copy of the right hand side next. So in fact if we look at this, if you look at the recursive algorithm, this copies the null pointer first we get the null pointer back and then it copies the last node and returns the pointer to the last node. So the recursive copy function which is copying the next to last node and on and on and on and on at the end. When we finish copying we finish copying the first node in the list and we return back the pointer, which is then stored as head and we've got the list copied. So ultimately this makes what would be a very complex or very time consuming algorithm, because remember even if we didn't insert an end that would be a linear time problem just to find the end. So we dealt them at least have a big O of N squared algorithm to find and search all the nodes which would be horrible but here we can do a recursive copy in linear time and be done with it.

What are Linked Lists Used For 1.12

You may ask yourself where we're going to use linked lists and it's a reasonable question, and anywhere that we need storage where we have constant time insertion with no overhead and we don't really need anything other than linear access to the individual nodes. That comes into play in a lot of situations in computer science so you'll run across this constantly. In a later module, we're going to talk about using linked lists to develop stacks and develop cues. We're also going to talk about other data structures which are either similar to linked lists or are based on linked lists.

So there's a lot of situations like that but one real world situation that you might actually have come across is the FAT thirty two file system. And this file system was in use in a lot of places for a long time; ever since from the basically the Windows ninety-five era on up through to, well it was still being used a little bit in Windows XP, but we stopped using it pretty much in Windows seven. You still use it today in a lot of situations where you have to transfer a file from a Mac machine to a PC and it needs to be editable on both of them. The only file system, at least right now, that's compatible for editing with both MacOS and with Windows is the FAT thirty two file system so we still use it. This file system stores a file, when we're storing a file what we do is we store a pointer to the first block on the hard drive where the file is stored and the first block on the hard drive stores a pointer to the second block on the hard drive. It's got quite a bit of data, but it also has a pointer to the second block. And then the second block stores a large amount of data, usually about four kilobytes, and a pointer to the third block and so on and so on and so on and so forth. The reason that this works is because very standardized format, we all understand it; it's a very old format so everybody knows how to work with it.

But there are some downsides to it and let me explain one of the biggest downsides that you might have actually experience. If you had a large video files stored on a FAT thirty two file system, this video file would be stored as pointers and the video file might be, let's say four gigabytes. But that means that it's broken down into four million blocks; each one of which stores a small portion of the video file. Now if you watch the video straight through you won't notice any problem, you'll retrieve the first block on your video player will display that and then the second block in your video player will display that and so on and so forth and so on and so forth. But if you're like me and you stop and start movies a whole bunch of times; you might stop the movie restart your machine come back to it a week later. And you drag the slider to the middle, well unfortunately we can't jump to the middle of that set of blocks. So we can't jump to the five hundred thousandth block out of the one million to get to the middle. What we actually have to do is access all five hundred thousand blocks and then we can get to the five hundred thousand and one block. So this is a downside of the FAT thirty two file system and you may have experience that if you have a video file stored on a FAT thirty two file system, drag the slider and wait for it to load all those five hundred thousand blocks and throw them away because all they're doing is accessing the pointers sequentially.

In this Module 1.13

We covered a lot of material in this module. We've looked at what is a linked list. We've looked at why do we need linked lists. We also talked about templates, and templates are something that are going to keep coming up time and time again throughout the rest of the semester. And in the data structures that we're going to talk about because we never know what they the type are storing inside of a data structure. We talked about the design of a linked list and you should definitely take a look at the code that we've posted in the CPP file for this module associated with this, so take a look at that because that is very useful resource. And then we talked about what linked lists are used for and get some real world example. So going forward you should understand linked lists and you should be able to make a

judgment call of whether you want to use an array or vector or a linked list because that's going to make a very big difference in terms of your performance of your program. And it's an important distinction that you should understand.