**Name: Patrick O’Brien**

For each of the below questions, write a short sentence or two to express (in your own words) your answer. Keep the answers short, but use complete, correct, English sentences.

If it helps to clarify the questions, feel free to mentally prefix all the questions with the phrase "According to the video…"

1. After you’ve watched all the videos, please answer this question:  
   Of all the videos that you watched, if you could pick one video to be re-recorded by the instructor outside of class which would you choose? Why?  
   (Keep in mind the recording outside of class will omit any pauses from the instructor answering student questions, have less hemming and hawing, etc, and generally be more concise)

|  |
| --- |
| Instead of already having the linked list implemented via a copy and paste, I think having people implement it step by step would be a better learning experience. Perhaps through an interactive tutorial on an existing coding website. |

**VIDEO: Nested Classes**

1. By default, where should you put your classes (even after watching this video)?

|  |
| --- |
| Put the classes outside of other classes. |

1. Why does it make sense to make the node class (within a linked list) into a nested class? (I.e., what rationale was provided in the video)?

|  |
| --- |
| It exists solely for the MyIntList class. Nobody else needs to make use of it or know about it.  So therefore it makes the most sense to put it inside the MyIntList class, creating a nested class. |

1. If IntListNode is marked as protected, which classes can access it? What if it's marked private?

|  |
| --- |
| Only things within the MyIntList class can access the data within the private IntListNode class.  If it is marked protected, then any subclasses connected to MyIntList class can also access it.  Nobody else outside of the class can create an IntListNode. |

1. When do you want to use a nested class?

|  |
| --- |
| Have a small helper class that you can attach methods to, but not allow anyone else to use it.  Only use when it’s a small helper class. |

1. Why is it ok to mark the data fields of the protected nested class as being public?

|  |
| --- |
| Because the node is just for the exclusive use of the list class, and it can’t be used for any other classes. There’s no reason not to mark the methods public.  You are saying that is only public within the list. You don’t have to create any getters and setters. |

1. Why does the LinkedList\_Verifier need your nested IntListNode to be protected, and not private?

|  |
| --- |
| Because it is a subclass, if it is private the subclass LinkedList\_Verifier will be unable to access it. |

1. In a nutshell, how does the LinkedList\_Verifier check to see if your linked list is correct?

|  |
| --- |
| It calls all of your add and remove routines. It has extra methods to check the array, signaling if certain things are correct or incorrect. |

**VIDEO: Linked Lists: Overview**

1. Up till now, what did you (typically) use in order to store a collection of things?

|  |
| --- |
| We would typically use an array. |

1. What is the major downside of using an array to store items?

|  |
| --- |
| If you run out of space, your only option is to create an entirely new array, copy everything other, and then destroy the old one. |

1. What are the (minimum) two fields that each node in the linked list must have?

|  |
| --- |
| Int data;  Node next; |

1. What value will the last node in the list use for it's **next** field?

|  |
| --- |
| Null is the next value, designating the end |

1. Which node will we keep track of? (Will we keep track of all of them, or just one?)

|  |
| --- |
| Only have to keep track of the very first one. If there is another item in the list, move forwards, finally check the end. |

1. If I wanted to add something to the middle of a linked list, how do I do that (answer this intuitively/pictorially, NOT using C# code)?

|  |
| --- |
| Splice it in the middle, and then add something.  With an array, you have to tell it to add something in the middle, and then move everything down one.  With the linked list, you can simply insert the new object between the two objects that create the middle. |

1. If I wanted to add something to the middle of an array, what would I have to do? Why is it easier to add something to the middle of a linked list?

|  |
| --- |
| With an array, you must move everything down 1, and then insert something into the now empty slot.  With a linked list, the objects are detached, but still reference one another. This way you can simply insert a new object between the two referencing objects, and change the new object so that it references the next item in the linked list. |

1. In addition to the "node" object/class that the video discusses, what other class/object does the video mention (towards the end)? What is the purpose of this second class?

|  |
| --- |
| The linked list object that points towards the first thing in the list. |

1. Are the "next" links for each node one-directional or bi-directional? If they're one-way, which way do they point?

|  |
| --- |
| The next links are one-directional single links going forwards.  Doubly linked list points both ways. |

**VIDEO: Linked Lists: AddAtFront**

1. Why can't we use the name **LinkedList** when creating our own linked list class? What is a good name to use instead?

|  |
| --- |
| LinkedList is a reserved system name type (framework class library), a good name is MyLinkedList |

1. What is the role/purpose of the MyLinkedList class?

|  |
| --- |
| Provide a nice clean interface that main/other class can use for managing the list. Provide general structure for managing the list. |

1. In this video, where are we going to add new items to the list?

|  |
| --- |
| In the main class  Mll.add( 17 ); |

1. Why is it important that the **front** reference start out with the value null?  
   Is it necessary to assign null to front ourselves? Why or why not?

|  |
| --- |
| The front value must start with the value null so that you can tell when the list is empty. It starts with null assigned as default, but it is good practice to show that the empty is equal to null. |

1. Should the Node class be nested or separate? If nested, which class should it be nested within?

|  |
| --- |
| The node class should be nested. The node class should be nested within the MyLinkedList class. |

1. What line of C# code will determine if a list is empty?

|  |
| --- |
| The front if it is equal to null. |

1. When the list is empty, how do you add a new node to the list?

|  |
| --- |
| To create a new node you state that:  If the value is null  Node l = new Node(newVal); Front = l;  Then l points towards a constructor that makes it so that l refers to the new object (the variable for front). |

1. The constructor for the Node class does not explicitly set the value of **next** – what value will next have?

|  |
| --- |
| If not explicitly set, the Node class will have the next value as the default value, which is null. It will be the end of the list if the next item in the list is not specified. |

1. How can you think about the computer's memory?  
   If I say "The node's address is 70,000" what does that mean (intuitively)?

|  |
| --- |
| When you create a new object, you are assigning the memory address of that object to front. Essentially giving the node address to front. |

1. C++ does allow you to actually get the memory address of objects – why does C#/Java/etc NOT allow us to get these memory addresses?

|  |
| --- |
| C#/Java/Etc is written to be easier to understand with object-oriented programming. Because it has garbage collection, assigning of memory addresses could break that. |

1. When adding a new node to the front of an existing list, what three steps do we have to do?  
   (List these intuitively/in English – NOT using C# code)

|  |
| --- |
| You do the following three steps to add a new node to the front of the existing list:   1. You create the new node and assign it the address of the new object that you create. The value is automatically assigned to null, pointing to the new object. 2. Take whatever front is, and then assign that value to the null object. The node points to the new list, but there’s nothing in the list that points towards the new node. 3. Front is assigned the value of the newly added node (nn). |

1. What is the C# source code for the three steps that you explained in the previous question?

|  |
| --- |
| Class MyLinkedList  {  Node front = null;  Public void addToFront (int newVal)  If(front == null)  {  Node l = new Node(newVal);  Front = l;  }  Else  {  Node nn = new Node(newVal);  nn.next = front;  Front = nn;  }  } |

1. What is the difference between **nn** and **nn.** ?

|  |
| --- |
| nn. is part of the actual object  the nn itself is just the memory address (80,000). |

1. What will happen if you accidentally reverse the order of the second two steps?

|  |
| --- |
| If you do it in the reverse order, you’ll create a new node that doesn’t point to anything. If the first thing you do is say that front is assigned nn, the entire rest of the list will be lost. |

1. In the Node class, is **next** an actual, embedded Node object? If not, then what is it?

|  |
| --- |
| Next is a reference to another node, it’s not actually a new node object.  It’s just a reference, not an actual embedded object. We’ve declared a reference, but we’re not using new to create a new thing. |

**VIDEO: Linked Lists: PrintAll**

1. What does the phrase "traversing the list" mean?

|  |
| --- |
| Traversing the list means that walking across every node in the list. Once everything is recorded you can search for specific things. |

1. What is the first step in printing all the nodes in the list?  
   (List this in English, not C# code)

|  |
| --- |
| Create a reference, not a new node, point it towards the first thing on the list, and print the data that is there. |

1. What steps will we repeatedly do, in order to print all the nodes in the list?  
   (List this in English, not C# code)

|  |
| --- |
| Create a reference, point it towards the first thing that is there, move the reference along the list until you reach null. |

1. What is the C# source code to print everything in the list?

|  |
| --- |
| Public void PrintAll()  {  Node cur = front;  While (cur != null)  {  Console.WriteLine(cur.data);  Cur = cur.next;  }  } |

1. Why doesn't PrintAll crash if the list is empty?

|  |
| --- |
| It would not crash because front would have the value null which is assigned to cur. If the list is empty it skips the while loop. |

1. Within the .Net platform library, what pattern is often used to store items in an array (instead of storing integers, like we typically do here)?

|  |
| --- |
| You would create a new object instead of using a single integer. Then you could add whatever you want to into the list. |

**VIDEO: Linked Lists: RemoveFromFront**

1. When you create a new MyLinkedList class, do you automatically create a Node class at the same time?  
   When do you get new Node objects?

|  |
| --- |
| No, in java it does that, in c# you do not create any subsidiary classes.  You get new node objects when you yourself create a node object. |

1. What two things will the RemoveFromFront method do?

|  |
| --- |
| It will remove a front node from the list  It will also return the thing when it is removed. |

1. Explain in English what should be done if the list is empty:

|  |
| --- |
| If it’s null you don’t need to do anything, you could return int32.MinValue.  You could also have it stop, and do something to indicate not to add that value to the list. |

1. If front is not null, what do we know about the list?

|  |
| --- |
| That it is not empty, that there is at least one node in the list. |

1. List out the C# source code that will remove the first item from the list & return it's data field:

|  |
| --- |
| Public int removeFromFront()  {  If (front == null)  Return Int32.MinValue;  Else  {  Int x = front.data;  Front=front.next;  Return x;  } |

1. If we call RemoveFromFront on a list with two items, what will the list look like after the method ends?

|  |
| --- |
| It moves the front object up by one, stores the value you want to return into x. |

1. What will happen to the nodes that no longer have any references referring to them?

|  |
| --- |
| The garbage collector will eventually notice that they are inaccessible and remove them. |

1. If we call RemoveFromFront on a list with only one item, what will the list look like after the method ends?

|  |
| --- |
| It will return the value null and will go straight to the return value. |

1. If we call RemoveFromFront on a list with no items, what will the list look like after the method ends?

|  |
| --- |
| It will look the same, since RemoveFromFront has a check to see if the list is empty. IF the list is empty, then it will not execute the else statement and simply return without the operation. |

**VIDEO: Linked Lists: Print At Location**

1. In terms of our PrintAtLocation method, what is the index of the first item in the list? What is the index of the second item?

|  |
| --- |
| Index of the first item in the list: 0  Index of the second item in the list: 1 |

1. What is the correct C# source code for the loop that walks to a particular node of the linked list?  
   Why will || (the logical OR) NOT work correctly in the loop?

|  |
| --- |
| While (cur != null && count < idx)  {  Cur = cur.next;  Count++;  }  It does not work because it will walk off the list, only when cur has the value null will it stop. && will work much better.  While the list has not yet run out, and we need to take a step forward. |

1. When the method goes directly to using a Console.WriteLine on cur.data, and the given index is larger than the list, what will happen (and more importantly, why does it happen)?  
   How do we fix the program so that it doesn't crash (list C# source code)?

|  |
| --- |
| When a method goes directly to using a Console.WriteLine on cur.data, and the given index is larger than the list, it will crash because the value is null. This happens because there are no objects outside the list (walked off the end of the list).  We have to say that if we run out of items, then skip that and move on, end the program. We say that there’s enough items in the list. Ex: if (cur != null) then print |

**VIDEO: Linked List Schema: Traversal**

Note that the document being discussed is actually listed in Lesson 05 (***Strategies for LL: Traversing A Linked List)***

1. When dividing up the work that needs to be done in order to walk through a linked list, what is the first step?

|  |
| --- |
| The first step is to assign cur to the first thing in the list. (Setup) |

1. Based on prior exercises (possibly in prior lessons), what will the "Iteration Logic " step need to do?

|  |
| --- |
| The iteration logic step needs to implement the checks:  While (cur != null) and the matching part, (cur = cur.next move it up one). |

1. What sort of work gets done in the "Teardown" step? Will you always need this step in order to traverse a linked list?

|  |
| --- |
| The teardown step is like:  After everything, print out the total. |

**VIDEO: Linked Lists: InsertAt**

1. When adding something to the middle of a list, how will we determine where in the list to put it? (And what is the other way to determine where to put the new value)

|  |
| --- |
| When adding something to the middle of the list, we determine where in the list to put it by: Deciding on a specific index to add it to  The other way to determine where to put the value is: Keep a sorted linked list and find a way to implement it by sorting through the linked list. |

1. If the index parameter is larger than the size of the list, what will our implantation do?

|  |
| --- |
| We should add it at the end, if the specified index is larger than the size of the list, it should throw an exception or crash since it is out of bounds of the list. |

1. What are the two basic cases that our code will need to handle?

|  |
| --- |
| 1. Update the very front of the list 2. Update something else in the list that comes after the list |

1. What are the two general values that the **front** reference might hold?

|  |
| --- |
| A null, or a location of the very first thing in the list. |

1. Under what circumstances to we want to add something to the front of the list?

|  |
| --- |
| 1. If the front is null 2. If the index is 0 |

1. Once you've decided to add something to the front of the list, what are the steps you need to go through in order to add a new node to the front of the list?

|  |
| --- |
| To add the thing to the front you have to:  If (front == null || index == 0)  {  Ln.Next = front;  Front = ln;  } |

1. Is the variable **ln** a reference, or the actual object?

|  |
| --- |
| The variable ln is a reference to the actual object. |

1. Why is the question "Add something to a linked list" a good (and surprisingly common) technical interview question?

|  |
| --- |
| Add something to a linked list is a surprisingly good technical interview question because:  It is 10-20 lines of code, and it’s extremely easy to not add everything to the list correctly. |

1. How do you use this new AddAt method (in, say, main)? Provide a C# code snippet that demonstrates this.

|  |
| --- |
| Main()  {  MyLinkedList ll = new …  ll.AddAt(5,0);  } |

1. Is it normal to sort an unsorted linked list? If not, what is normally done instead?

|  |
| --- |
| No, it’s not normal to sort an unordered list on the fly. Typically if you want it sorted, you keep it sorted as you go through it. With a linked list you have to go through every step, unlike an array. |

1. What is the C# code that provides basic pattern for walking down the list? How does it work? (For example, make sure that you could trace through this code)

|  |
| --- |
| Public void AddAt(int data, int index)  {  ListNode ln = new ListNode(data)  If (front == null || index == 0)  {  Ln.Next = front;  Front = ln;  Return;  }  ListNode cur = front;  // General pattern  While (cur != null)  {  Cur = cur.next;  } |

1. What is the C# code the provides both the basic list-walking logic AND stops walking at the point that will correctly leave cur pointing at the node PRIOR to desired location of the new node?  
   (Make sure that you use the version that will ensure that cur will not walk off the list)

|  |
| --- |
| ListNode cur = front;  Int counter = 0;  While (cur != null && counter < index)  {  Cur.next;  Counter++;  } |

1. How do we know that cur is not null when we start the list-walking loop?

|  |
| --- |
| The if statement earlier in the method when we check that if (front == null).  We then know that cur is not null when starting the list-walking. You have to make sure that it is not null. |

1. When we end the loop, where might **cur** be pointing to?

|  |
| --- |
| Cur might be pointing to null if you walked off the list. This would cause an error, so you have to stop it from walking off. |

1. How do we add something to the list, when **cur** ends up pointing to the last item in the list?

|  |
| --- |
| We say:  Cur.Next = ln;  // If we wanted to add 10 at the end of the list  Cur gets the memory address for 8500, goes to the next |

1. How do we add something to the list, when **cur** ends up pointing to an item somewhere in the middle of the list?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Adding 9 to a place somewhere in the middle of the list  Ln.next = cur.next;  Cur.next = ln;  Connect the new thing to the next thing in the list, then change the thing before it to the new thing in the list.   |  |  | | --- | --- | | data | 9 | | next | 8500 | |

1. What is the final, finished, complete C# code for the AddAt method?

|  |
| --- |
| ListNode front;  Public void AddAt(int data, int index)  {  ListNode ln = new ListNode(data);  If (front == null || index == 0)  {  Ln.Next = Front;  Front = ln;  Return;  }  ListNode cur = front;  Int counter = 0;  While (cur.next != null && counter < index -1)  {  Cur = cur.next;  Counter++;  }  Ln.next = cur.next;  Cur.next = ln; |

**VIDEO: Linked Lists: RemoveAt**

1. What is the first thing to check for? Why should the method return if this is true?

|  |
| --- |
| The first thing that you should check for is: Check to see if there is anything in the list (if it is equal to null or not)  The method should return if it is true because: If the list is empty, there is no reason to do anything extra. |

1. What is the C# code to remove the first item in the list?

|  |
| --- |
| The C# code to remove the first item in the list is:  If (index == 0)  {  Front = front.next;  Return;  } |

1. If **front.next** is null, how many items are in the list?

|  |
| --- |
| It means that there are 1 items in the list. You are saying remove that one and only item in the list… etc. |

1. What is the C# code to walk through the list, leaving cur correctly positioned to remove the target element?

|  |
| --- |
| The C# code to walk through the list:  While (cur.next != null && counter < index -1)  {  Cur = cur.next;  Counter++  } |

1. What are the two possible reasons why we've exited the list-walking loop?

|  |
| --- |
| We have run through the entire list  We have reached the target location in the list |

1. What is the C# code to **check that we should** remove the element after cur?

|  |
| --- |
| 1. If (counter == index -1 && cur.next != null ) |

1. What is the C# code to **actually remove** the element after cur?

|  |
| --- |
| 1. If (counter == index -1 && cur.next != null ) 2. { 3. Cur.next = cur.next.next; 4. } |