# Linear Structures

*CSC 122: Data Structures*
*Due: ~~Friday 2/28 at 8:00 AM~~, via Moodle*
*Friday 3/6 at 8:00 AM, via Moodle*

**This assignment is more challenging conceptually than hw2, so please plan your time accordingly. I highly recommend you start early and work steadily at least every other day.**

---

In this assignment, we'll explore the relationship between an array-based and linked-list-based implementation of a "linear structure" ADT (for ints only). For this purposes of this assignment, we'll define the linear structure ADT as requiring:

- void addToStart(int value) - Adds the given value to the beginning of the structure.
- void addToIndex(int value, int index) - Adds the given value to the given position in the structure.
- int removeFromStart() – Removes the element at the beginning of the structure, and returns it.
- int removeFromIndex(int index) – Removes the element at the given position, and returns it.
- int getSize() – returns the size of the structure.
- int get(int index) – returns the element at the given index.
- void printAll() – prints each element in the structure in order.

We'll also define a constructor, and a special kind of method called a *destructor*. More details of the requirements of all of these methods are described further below.

The methods above form the *interface* for the linear structure ADT, as we're defining it here. We could imagine adding more to this, like checking if an element is in the list, sorting the elements, etc. But we'll stick with the above for now. So the above methods are the required interface, the externally-accessible operations, for any class that implements the linear structure ADT. There are various ways, though, to actually implement that behavior.

In this assignment you'll write two classes that implement the linear structure ADT in very different ways:

- IntArrayLS – an array-based implementation of the linear structure ADT, for ints. This is very similar to Java's ArrayList class.
- IntLinkedListLS – a linked-list-based implementation of the linear structure ADT, for ints.

You'll test your classes with a provided main function.

We need to explore a few new ideas in this assignment too, so please **treat this assignment description like a textbook chapter**. Please study it, wrestle with it, and let me know what questions you have. On a future exam, I could ask you about the concepts covered in this assignment. More importantly, the material includes very important concepts, including ideas that will come up in future courses and in job interviews.

## Get The Provided Code

I've given you some .h and .cpp files. In making your project, it's important that these be added to your project correctly. Please do the following right away:

- Make a new Visual Studio project.
- Make a new .cpp file by right-clicking on Source Files and choosing "Add→ New Item…" as usual. Call it main.cpp.
- In a simple text editor (like Notepad, but not Microsoft Word, and not Visual Studio), open the main.cpp I've provided. Copy all of that code in Notepad and paste it into your own main.cpp in Visual Studio.
- Do this for every .cpp and .h file I've provided. You'll need to give them precisely the same names I've given them, including capitalization.

If you don't do this correctly, then your project may not work correctly when you try to turn it in, or if you move it to another computer! Please let me know if you have any questions.

## Checking Your Work As You Go

I want to encourage you to check your work in small chunks as you go. Observe the provided main. As you work, you can comment out parts you're not ready to test, and get a couple methods working first. You can use the provided code in main.cpp to make your own smaller test cases as well.

## IntArrayLS

You are capable of writing the entire IntArrayLS class yourself, with a little guidance and sufficient time. But in order to focus our time, I've written some of it for you. Your task is to fill in a few gaps, marked by comments with the text "TO DO", in IntArrayLS.cpp. This is discussed further below.

### *Preprocessor Directives*

Start by observing IntArrayLS.h. You'll notice one new bit of syntax right away: the lines starting with #ifndef, #define, and #endif at the top and bottom of the file. This is common practice in every C++ header file. These are called *preprocessor directives*, and they're instructions to the compiler about how to compile the code.

#ifndef INTARRAYLS_H says "Hey compiler, if you've not already defined the symbol INTARRAYLS_H, then compile the code that follows."

#endif marks the end of the code under the #ifndef condition. So it's kind of like the closing } that matches an opening { for the #ifndef – although no such braces should actually be provided.

The #define line says "Hey compiler, define a symbol named INTARRAYLS_H. I don't care how it's defined, just consider it as something you know about, a symbol that exists."

Why do this? When the compiler compiles your code, it may end up looking at multiple files multiple times. For example, if both your main.cpp and your IntArrayLS.cpp file have a `#include "IntArrayLS.h"` line, then that header file will be "looked at" in the compilation of both .cpp files. This can sometimes cause compilation errors, and at the very least is unnecessary work for the compiler. So the preprocessor directives described above make sure that the compiler only looks at this code once.

How do the directives accomplish this? The first time IntArrayLS.h is examined in the current compilation, the symbol INTARRAYLS_H is not yet defined. Therefore, the code under the #ifndef is compiled (since #ifndef means "if not defined"). Then the #define will define the INTARRAYLS_H symbol, and the rest of the code is compiled.

If in the compilation process, the IntArrayLS.h file is examined a second time (due to multiple #include "IntArrayLS.h" lines), then this time #ifndef will check the symbol INTARRAYLS_H and find that it is already defined, therefore the code under the #ifndef is not compiled a second time.

Thus, the code is only compiled once.

The description above is actually simplified from the true compilation process, but this is plenty for us (and for most people, really).

Technically you could use any symbol name you want, by the way – it doesn't have to be INTARRAYLS_H. But it's traditional to name it according to the file name, in all upper-case, with a _ for the dot. That way, you won't have multiple header files trying to work with the same symbol, which would mess up the compilation process.

As you'll see further below, you'll want to use the same preprocessor directives in your IntLinkedListLS.h file, using the symbol INTLINKEDLISTLS_H.

### *IntArrayLS Destructor*

Take a look at the class declaration in IntArrayLS.h. Note the constructor, IntArrayLS, and a *destructor* called ~IntArrayLS. The destructor of a class must always be named ~*ClassName*. This is a method that does not exist in Java.

What's the purpose of the destructor? The destructor is code that must be executed when a class object is destroyed. One place where this happens is when a class object falls out of scope. For example, you might declare a local variable of type IntArrayLS in some function. When the function is about to finish, the destructor is called to "destroy" the object before the function finishes. In this case we don't have to call the destructor explicitly in any way – it gets called automatically.

Peek at the provided destructor definition in IntArrayLS.cpp. Note that it just says:
```
delete [] elements;
```
This special delete command with `[]` is used to delete a dynamically-allocated array. If you look in the IntArrayLS constructor and in expandArray, you can see that we're using `new` in both places to dynamically allocate the array space. When you dynamically allocate memory, you need to delete it when you're done. For dynamically-allocated fields, the destructor is the place to do it.

### *IntArrayLS.h – Other Details*

Look back in IntArrayLS.h. Note the other public methods. These correspond to the interface required for the linear structure ADT, as defined at the top of this assignment.

Also note the declaration of three private methods. These are private since they are not intended to be used outside of the class. They're helpful pieces of some of the public methods' tasks.

Finally, note three private fields declared for the class:

- elements - a pointer to an int. Actually this will serve as a pointer to the first element in a contiguous block of ints for the array. As you'll see, we can still use ordinary [] notation on elements. For example, elements[2] gives us the int at index 2 in the array, assuming that 2 is a valid index for the array.
- size – the number of elements in the array right now.
- capacity – the total amount of space we have for this array before we need to allocate a new bigger chunk of memory elsewhere.

If you're not sure why we need both capacity and size, please double check our discussion about arrays in class and let me know how I can help!

### *Finishing IntArrayLS.cpp*

Your task for this part, then, is to fill in the missing portions of IntArrayLS.cpp, as indicated with the "TO DO" comments. Before you attempt this, please do the following:

- Review our in-class discussion of how an array-based implementation of the linear structure ADT must shift elements around and potentially allocate a bigger chunk of memory.
- Study the very detailed comments provided in IntArrayLS.cpp.

If you really understand the concepts discussed in the above two bullet points, then this should go fairly smoothly. If, on the other hand, you don't understand the above two bullet points, then you risk wasting a lot of time writing confusing code that won't work at all. So please do that review/studying and let me know how I can help *before you start typing* any code of your own!

---

## IntLinkedListLS

In class we've written (or will soon write) many *functions* that work with linked lists. Part of your task for this problem is to convert those functions into methods in the IntLinkedListLS class. That will be pretty straightforward if you are confident in writing classes in C++ and understand those functions. There are a couple bits of code you'll need to write here, though, that are new problems; they're similar to what we've done in class, but not identical. Those will be no trickier than the in-class problems, but they are new.

The first thing to keep in mind is that the IntLinkedListLS class will implement the same methods as IntArrayLS. The difference is that IntArrayLS implements the methods using an array, while IntLinkedListLS implements the methods using a linked list. This means our two classes will have the same tradeoffs as we've discussed in our class time: IntArrayLS will have fast indexing but slow add/remove, while IntLinkedListLS will have slow indexing but (sometimes) fast add/remove.

Next, write an IntLinkedListLS.h file. Use the same kind of preprocessor directives as described for IntArrayLS.h. You'll only need two fields: the head of the linked list, and the size (an int). We'll keep track of the size explicitly, instead of calculating it with a loop every time we need it, so that we can get the size more efficiently.

You'll need the following public methods, just like with IntArrayLS:

- IntLinkedListLS() – the constructor. Just initialize the head to NULL, and size to 0. That's what an empty linked list is.
- ~IntLinkedListLS() – the destructor. You'll need to loop through the linked list and delete every node as you go.
- void addToStart(int value)
- void addToIndex(int value, int index)
- int removeFromStart()
- int removeFromIndex(int index)
- int getSize() – just a simple accessor to return the size field. This will require adjusting the size field in methods that affect the size of the structure. (That is, don't just calculate this with a loop in getSize().)
- int get(int index) – there's no fast indexing for a linked list, so you'll need to calculate this by looping through the structure.
- void printAll()

Most of these we've done in class (or will soon), though some of the names may be a little different. Note that none of these take a head pointer as an argument. This is because the head pointer is a field in the IntLinkedListLS class.

Once you've declared all this in IntLinkedListLS.h, define the methods in IntLinkedListLS.cpp.

Don't forget to update size in the methods having to do with adding and removing!

For more details of required behavior, please see the expected output in the next section.

---

## Expected Output

If you've changed the main a lot in your own testing, please save that code somewhere but then get the original version again for the official test. You should be able to run the provided main and get the output below (maybe after a page break or two).

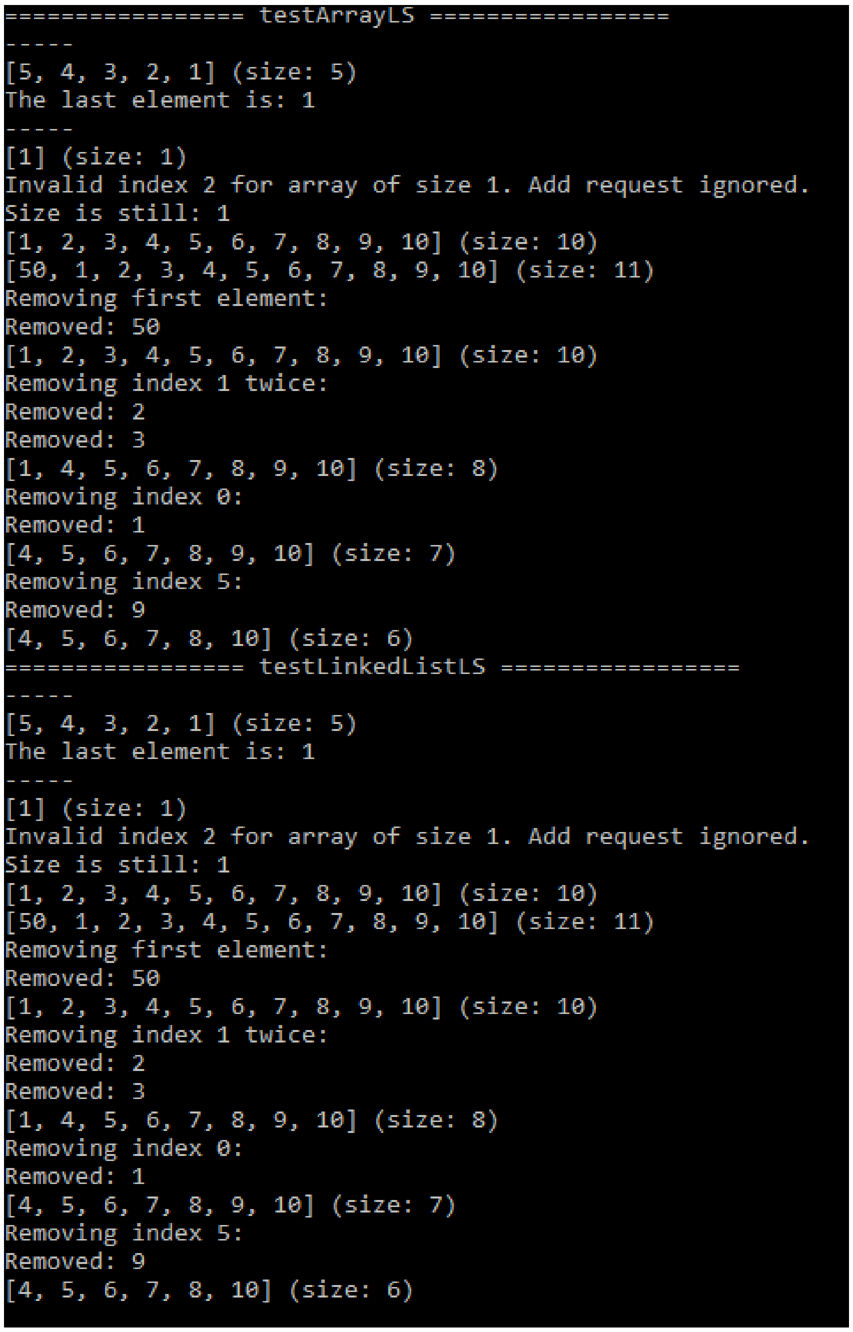
Note that the output is the same for both classes. This is as it should be – the two classes have the same external behavior, since they both implement the linear structure ADT as we've defined it here. But their implementations are completely different! Their efficiency is also different, as you know intuitively already, and will know more explicitly very soon.

Please pay careful attention to the output. Note the format for printAll, including the use of commas, with no comma after the last item. Also note that the size is printed with printAll. The remainder of the output is making sure that items are added and removed at the right places according to the method calls.

If your output does not match precisely, or you get any errors in your code, then those issues will need to be fixed in order to get full credit.

These two classes demonstrate the idea of *encapsulation*. When you hear that word, think of a “capsule”, holding things inside of it. The implementation details (e.g. for the array version, expanding an array sometimes, shifting items around sometimes) are hidden from the user – encapsulated in the class. All the user of the class needs to be aware of is the interface – what methods are available and what their external behaviors are. That is, the user only needs to know about the public section of the corresponding header file. The details are encapsulated, which frees the user from having to worry about those details.

To further this discussion, consider how you learned about ArrayList in CS1. You never had to think about how that class manages its capacity, expands as needed, shifts things around to maintain fast indexing. All those details were encapsulated, hidden away from you so that you could just use the class. That’s what we’re working on here. Encapsulation is one of the greatest advantages of object-oriented programming. The user of a class doesn’t have to worry about the implementation details, and can therefore focus on other important tasks.



```
================= testArrayLS =================
-----
[5, 4, 3, 2, 1] (size: 5)
The last element is: 1
-----
[1] (size: 1)
Invalid index 2 for array of size 1. Add request ignored.
Size is still: 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (size: 10)
[50, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (size: 11)
Removing first element:
Removed: 50
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (size: 10)
Removing index 1 twice:
Removed: 2
Removed: 3
[1, 4, 5, 6, 7, 8, 9, 10] (size: 8)
Removing index 0:
Removed: 1
[4, 5, 6, 7, 8, 9, 10] (size: 7)
Removing index 5:
Removed: 9
[4, 5, 6, 7, 8, 10] (size: 6)
================= testLinkedListLS =================
-----
[5, 4, 3, 2, 1] (size: 5)
The last element is: 1
-----
[1] (size: 1)
Invalid index 2 for array of size 1. Add request ignored.
Size is still: 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (size: 10)
[50, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (size: 11)
Removing first element:
Removed: 50
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (size: 10)
Removing index 1 twice:
Removed: 2
Removed: 3
[1, 4, 5, 6, 7, 8, 9, 10] (size: 8)
Removing index 0:
Removed: 1
[4, 5, 6, 7, 8, 9, 10] (size: 7)
Removing index 5:
Removed: 9
[4, 5, 6, 7, 8, 10] (size: 6)
```

---

## Where To Go From Here?

In class we'll study the idea of recursion, which can be very useful for working with linked lists and other data structures. We'll also study complexity analysis, which gives us a more precise measure of the efficiency of ADT implementations.

In terms of an ADT implementation in C++ specifically, there are more concepts we could consider that are not included in this assignment. In case you'd like to look into them on your own, I'll mention a few useful ideas below:

- C++ templates (similar to Java generics) would allow us to write our code to work for any type, not just ints. The data structures ideas are the same, it's just more syntax to work more flexibly with any type.
- To use templates would make it useful to do a deeper study of parameter passing, including when to use pass by reference to avoid unnecessary copying, and providing const correctness to make assurances to users.
- In order to ensure what are known as copy semantics for our class, we would need to correctly write the "big three": a copy constructor, operator=, and destructor.

These are great C++ topics, but in this course we'll instead move on to additional data structures topics.