# Lab 3: Lists

COSC2436

Spring 2025

Due Sunday, April 6, 11:59 PM

## Introduction

This lab is all about lists. You are provided starter code with TODO’s to finish. For part 1, the implementation is provided, and you’re asked to create unit tests. For part 2, you’re asked to create an insertion sort for a LinkedList and to ensure that the provided unit tests pass. For part 3, you’re asked to create only the interface for a new feature in a fictitious scenario provided by AI. For part 4, you’re asked to refactor the linked chain implementation in part 1 to use smart pointers reusing the unit tests you created in part 1.

Make sure everything compiles and that all tests run at once from the main function.

## Part 1: Array and Linked Unit Tests

The array and linked implementations from the textbook have been provided for you. Write the unit tests for both the array and linked list. Use the axioms from the book to guide your choice of tests. An example of how to test for exceptions is provided in ***testArrayList***. Ensure that your tests cover all major functionality.

A blue screen with black and white text

Description automatically generated

## Part 2: Insertion Sort

The insertion sort is an O(n2) sorting algorithm that works like most people order a hand of cards. The dealer places a pile of cards in front of the player. The player picks up one card, then another and orders it with the first, then a third and orders it with the two. The player continues in this matter, inserting one card at a time in the proper order until the hand is sorted. This is the insertion sort. It’s not as efficient as the famous quick sort but it works well with small, linked lists and lists that are already partially sorted. It has other beneficial properties as well. See <https://en.wikipedia.org/wiki/Insertion_sort> for more info.

Using the templated function signature given in the starter code, write an insertion sort. Make sure it works with the given unit tests.

Which of our two data structures (arrays or links) are preferable to use with the insertion sort? Why? Are either more or less appropriate?

## Part 3: Software Engineering Theater

Write a playlist ADT from the following theater script depicting a conversation in a software shop amongst several coworkers discussing the design of their new music streaming service. A playlist ADT will be roughly like the list ADT, but with more features and looping functionality. Provide **only the ADT** and not the data structure implementation. Feel free to create supporting types like a play mode for use within your interface signatures.

### Characters:

* Manager (M): Oversees the project, ensuring it meets business needs.
* Developer (D): Responsible for creating the Playlist ADT in C++.
* User (U): Represents the end-user of the music streaming service.

*Scene: A meeting room. A whiteboard is set up, and everyone has laptops open.*

**M:**

Alright team let’s get down to business. We’re working on the music streaming service, and I want us to focus on the playlist functionality. The requirement is simple: create a system that can handle different modes like loop, random, and play-once for our users. How are we feeling about this so far?

**D:**

I think we have a good base. The playlist can be treated as an Abstract Data Type (ADT), allowing the player system to interact with it without worrying about the underlying mechanics. The playlist modes—loop, random, and play-once—will be managed internally. The player only needs to ask for the next song, and the playlist will decide what to play.

**U:**

That sounds good, but I have a few questions. Will I be able to add or remove songs from the playlist? What if I want to reorder the songs?

**D:**

Absolutely. The playlist should allow adding and removing songs dynamically. Reordering songs can also be an option, either by manually dragging them around or using prebuilt sorting options, like sorting by artist, album, or genre.

**M:**

Good point. But let’s focus on core features first. We need a clear structure for how these modes work. How should we handle switching between modes, and what happens if the playlist ends in each mode?

**D:**

In loop mode, once the last song finishes, the next song will be the first one in the list. For random mode, the songs will be shuffled, but each song should still be played exactly once before shuffling again. In play-once mode, the playlist just stops when it reaches the last song.

We could have methods like SetMode(Mode) where Mode is an enum—something like LOOP, RANDOM, and ONCE. The NextSong function would then depend on the mode that’s currently active.

**U:**

What about skipping or going back? Sometimes I want to skip a song or maybe rewind to the previous one.

**D:**

Good question. We should have methods like Skip and Previous. For Previous, in random mode, it might be a little tricky because we’d need to keep a history of the songs that have been played. But that’s doable.

**M:**

I like where this is going. Let’s also talk about persistence. What happens when a user closes the app? Will the playlist stay as it was?

**D:**

We could store the playlist state, including the current song, mode, and the history of songs played in random mode, to a file. When the app is reopened, we’ll load that data back in.

**U:**

Could I have multiple playlists? Like one for working out, one for relaxing?

**D:**

Definitely. We could have a PlaylistManager class to handle multiple playlists, allowing the user to create, name, and switch between them. Each playlist would be an instance of the Playlist class.

**M:**

Alright, I think we’re getting somewhere. One more thing: sometimes users will be using voice commands to interact with the playlist. Can we make sure it’s easy to skip, pause, or change the mode with simple commands?

**D:**

Sure, that’s more of an interface-level concern. We can create simple APIs like Skip, Pause, and SetMode that the voice command system could hook into.

**U:**

What if a song is unavailable due to licensing issues or poor network connectivity? Should the playlist skip over those automatically?

**D:**

That’s a great point. We can add error handling to the NextSong method. If a song isn’t available for some reason, it could automatically skip to the next available song, and perhaps notify the user about the issue.

**M:**

Alright, this is shaping up nicely. Let’s summarize the core functionality of the playlist ADT:

- Ability to add, remove, and reorder songs.

- Three playback modes: loop, random, and play-once.

- Methods for Skip, Previous, and SetMode.

- Persistence of playlist state between sessions.

- Support for multiple playlists via a manager.

- Error handling for unavailable songs.

- Easy integration with voice commands.

**D:**

I’ll start working on a rough draft of the ADT. It’ll include methods like NextSong, Previous, SetMode, AddSong, RemoveSong, and so on. I’ll also define the Mode enum and think about how to handle random shuffling and persistence.

**U:**

Looking forward to testing this out! One more thing—will it show what song is next? Like, if I’m in random mode, will it show me the next song that’s going to play?

**D:**

In random mode, the next song would be unpredictable by design, but in the other modes, we could have a method like PeekNextSong to give you a heads-up on what's coming.

**M:**

That should be optional though. Let’s keep random mode fully random unless explicitly asked for the next song.

**U:**

Got it! Sounds good.

**M:**

Alright team, we’ve got a solid plan. Let’s start with the implementation. Keep the user in mind when designing it—flexibility and simplicity are key.

## Part 4: Smart List

Convert the linked implementation LinkedList to use C++ shared pointers. Rename this new implementation ***SmartLinkedList***. Use the unit tests you created in part 1 to ensure your implementation is correct; simply copy them over and change the type to

***SmartLinkedList***. The new tests should run from a new ***testSmartLinkedList*** function. You will not need to call delete or new. And destruction and clear should be a lot simpler.