# CICS 210 Data Structures

Programming Assignment 3: List, Iterator, and Comparator

## Introduction

This assignment is **divided into three distinct parts**, each of which emphasizes key concepts from the material we have covered thus far. The first part requires the implementation of the List data type, exploring variations of the implementation discussed in class. The second part extends this by focusing on the development of iterators for both the ArrayList and LinkedList implementations of the List data type. In the third part, you will delve into the creation of Comparators for a data type that represents web pages and URLs. It is essential to approach this assignment sequentially, particularly ensuring that Part 1 and Part 2 are completed in order, as each builds on the foundations laid by the preceding tasks.

## Downloading and importing the starter code

To begin this assignment, you will follow a familiar procedure to set up your development environment. Please adhere to the detailed instructions below to ensure you have access to all necessary resources.

### Initial Setup

1. **Download Starter Code**:
    - Navigate to the assignment page where you will find the archive file containing the starter code for this assignment. Click on the link to download the file to your local machine.
2. **Decompress the Archive**:
    - Once downloaded, locate the file in your downloads directory. The file will typically be in a compressed format (.zip). Use your operating system's decompression utility to extract the contents of the file. Ensure that all files are extracted without error to avoid any missing components.
3. **Importing into Visual Studio Code (VSCode)**:
    - Open Visual Studio Code, the recommended Integrated Development Environment (IDE) for this course.
    - In VSCode, navigate to the `File` menu and select `Open Folder...`.
    - Browse to the directory where you decompressed the starter code. Select the folder labeled `list-iterator-comparator-student` (this folder contains the `support`, `src`, and `test` directories as described previously).

- Click `Open` to import the entire project into VSCode. The IDE should now display all the files and directories within the project in its explorer pane, allowing you to view and edit the code as required.

### Verifying Project Structure

Upon successful import, your VSCode workspace should reflect the following structure, which is critical for the organization and testing of your implementations:

- **support/**: Contains the interface and other support files. These are provided to guide your implementation and should not be modified.
- **src/**: This is where you will implement the required functionalities of the `ArrayList` and `LinkedList` as specified in the `List` interface.
- **test/**: Contains unit tests designed to validate your implementations. Regularly running these tests will help ensure your code meets the assignment specifications.

### Getting Started with Development

With the project successfully set up in VSCode, you are now ready to begin problem solving and coding. Remember to frequently save your changes and execute tests to verify the behavior of your code. Regular testing and revision will be key to successfully completing this and future assignment.

Should you encounter any issues during the setup or have any questions regarding the project structure, please do not hesitate to reach out for assistance during office hours or on the course discussion forum. Our goal is to ensure that you have a smooth start to this assignment and are fully equipped to tackle the challenges it presents.

# Part 1 List Data Type and Implementations

## Overview

During our course lectures and associated readings, we have explored the foundational concepts of references and arrays to create more complex data structures. Notably, we have initiated partial implementations of the List Abstract Data Type using two distinct approaches: an array-based implementation and a node-based implementation known as a "linked list."

In the first part of this assignment, you are tasked with advancing these initial models. This will involve adding new methods or modifying existing ones in each of the List implementations.

#### Supporting Resources and Testing

- **Unit Tests**: To aid in the verification of your implementations, we have provided a comprehensive set of unit tests. These tests are designed to automate the validation process, ensuring your methods perform as expected under various conditions.

- **Interactive Testing**: For those who prefer a more hands-on approach to testing, consider writing a Java class with a main method. This setup will allow for interactive testing, providing you with immediate feedback on the functionality of your methods.
- **Debugger Utilization**: We recognize the importance of thorough debugging and have therefore disabled the timeout code in the test files to facilitate the use of debugging tools without interruption. However, if you encounter situations where your code does not respond during testing, we recommend uncommenting the following two lines at the top of each test file to help identify issues:

  ```
  @Rule
  public Timeout globalTimeout = Timeout.seconds(seconds:10); // 10 seconds
  ```

  These settings will help manage any infinite loops or other issues that cause the tests to hang, ensuring a smoother testing experience.

### Moving Forward

This part of the assignment is crucial for deepening your understanding of data structures through practical application. As you work through these tasks, remember to utilize the resources and support available to you, and do not hesitate to seek assistance if needed. This is an excellent opportunity to refine your coding skills and gain confidence in implementing and manipulating complex data structures.

## Goals

- Practice writing instance methods.
- Practice implementing generic abstractions.
- Practice with Array and Linked List implementations of the List abstract data type.
- Test code using unit tests.

## Complete ArrayList and LinkedList Implementations

As part of your academic development in data structures, this assignment will require you to delve deeply into the mechanics of the List data type, utilizing Java. Below you will find detailed instructions on the scope and requirements of the task.

### Overview of Assignment Structure

- **Directory Structure:**
  - `support/`: This directory includes immutable files such as the `List` interface which outlines the required implementations. Do not modify these files.
  - `src/`: This directory contains the `ArrayList` and `LinkedList` classes where you will implement the functionalities specified in the `List` interface.

- `test/`: Contains tests that validate the implementations in the `src/` directory.

### Guidance

Throughout these classes, comments labeled **TASK** have been provided to guide you through the implementation process. Be sure to read the comments carefully, as they outline the specific functionality each method should implement.

### Implementation Instructions

1. **Starting Point**: Begin with the `ArrayList` implementation, as it is generally simpler and more straightforward to debug. Once you have successfully implemented and tested the `ArrayList`, proceed to the `LinkedList`.
2. **Essential Methods**:
    - Initially, focus on implementing the `.equals()` method for both `ArrayList` and `LinkedList`. This method is crucial as it is a prerequisite for most of the provided tests. The method should include:
        - Checks for `null` values and type validation, which have been partially implemented for you.
        - Logic to ensure that each element in the current list is semantically equivalent to the corresponding element in another list. If they do not match, the method should return `false`.
3. **Method Implementations**:
    - You are encouraged to reference code discussed in class or the textbook. However, the educational value of this assignment lies in engaging with the material through individual thought and application, rather than mere replication of existing code.
    - This hands-on experience is vital as you will be expected to demonstrate proficiency in these implementations in assessments such as quizzes and exams.
4. **Restrictions**:
    - You must not use or import the `java.util.ArrayList` or `java.util.LinkedList` classes. The objective is to construct these structures from scratch to solidify your understanding of their internal workings.
    - Efficient execution of `.equals()` can be done using an iterator, which is covered in Part 2 of this assignment. However, we are restricting you from using iterators in this part of the assignment because we want you to get more practice with using and manipulating references. If we find that you do use iterators in this part of the assignment, even if you pass the Gradescope tests, when we manually review your submission we will mark it as 0.

### Specific Requirements for Data Structures

- **ArrayList**:
    - Implement dynamic resizing of the backing array to ensure that it grows as needed. The resizing strategy should reflect an optimal, amortized constant time complexity, ideally by doubling the capacity of the array each time it reaches capacity.
- **LinkedList**:
    - Maintain a `tail` reference that points to the last element of the list. This requires careful updates to the `tail` reference during additions to an empty list and removals from the list.
    - Enhance the efficiency of the single-argument `add()` method to operate in constant time by utilizing the `tail` reference to append directly to the end of the list, instead of traversing from the head.

### Code Readability

- Ensure that your code is readable and well-formatted, adhering to standard Java formatting guidelines. Consider using tools like Visual Studio Code's formatting commands to maintain consistent indentation and style.
- Choose descriptive and meaningful variable names that reflect the purpose of the variables.
- Avoid overly complex control flow in your code to maintain clarity and maintainability.

### Manually Graded Components

In addition to automated tests, this assignment includes manually graded components focusing on the implementation details mentioned above, particularly the dynamic behavior of the `ArrayList` and the operational efficiency of the `LinkedList`.

This assignment not only tests your ability to implement fundamental data structures but also assesses your ability to write clean, efficient, and readable code. Let's delve deeper into the grading criteria with more specific guidance and examples to ensure you have a clear understanding of what's expected in your implementations for the `ArrayList` and `LinkedList`.

### Grading Criteria - Detailed Expectations

1. **Dynamic Array Growth in ArrayList**:
    - **What to Avoid**: Do not create a fixed-size array intended to pass all tests without resizing.
    - **What to Do**: Implement logic to dynamically increase the size of the array when it reaches capacity. This should be done using a method that checks if the array is full and then resizes it appropriately.
2. **Optimal Growth Behavior in ArrayList**:

- **Efficiency Requirement**: The array should resize in a way that the time cost of resizing is spread out over a series of operations, hence achieving amortized constant time.
- **Implementation Tip**: Double the size of the array each time it fills up, as this minimizes the number of resizings required over the lifetime of the array.

3. **Tail Reference Management in LinkedList**:
    - **Adding to an Empty List**: Ensure that both `head` and `tail` point to the newly added node.
    - **Removing the Last Element**: Update `tail` to point to the second last element, which may involve traversing the list if a previous node reference is not maintained.
    - **General Management**: Always update the `tail` pointer whenever the last element changes.
4. **Efficient Append Method in LinkedList**:
    - **Constant Time Append**: Use the `tail` pointer to add elements at the end of the list directly without needing to traverse the entire list.
5. **No Iterator Usage**:
    - We will be inspecting your code to ensure that you do not use iterators in the methods we ask you to implement in this part of the assignment. A 0 will result in their usage here.
6. **Code Readability for Both Implementations**:
    - **Formatting**: Use IDE tools to automatically format your code. For instance, in Visual Studio Code, you can use the `shift + alt + F` (or `shift + option + F` on a Mac) shortcut to format the document.
    - **Variable Naming**: Choose names that reflect the purpose of the variable. Avoid single-letter names except for common idioms (like `i` for loop indices).
    - **Control Flow**: Keep your logic straightforward. Break complex methods into smaller, more manageable sub-methods.

**Good Practices Example**:

```
// Bad practice: unclear variable names and complex logic
int d = 0; // What is 'd'?
for (int i = 0; i < n; i++) {
    if (a[i] == 0) d++;
}

// Better practice: clear variable names and simple logic
int zeroCount = 0;
for (int index = 0; index < arrayLength; index++) {
    if (array[index] == 0) zeroCount++;
}
```

By adhering to these detailed guidelines, you'll not only meet the grading criteria but also enhance the robustness and readability of your code, setting a strong foundation for future software development projects.

# Part 2 Iterator Implementation

## Overview

In both class discussions and the assigned readings, we have explored two fundamental concepts in Java that are closely tied to Abstract Data Types (ADTs) and collections. This part of the assignment focuses on the Iterator ADT.

1. **Iterator**: An interface that allows for sequential access to elements within a collection, enabling iteration one element at a time. Iterators also provide support for Java's enhanced `for` loop, facilitating streamlined traversal of collections.

In this part of the assignment, you will gain practical experience implementing the Iterator interface, deepening your understanding of their role in Java's collection framework.

## Implementing Iterators for ArrayList and LinkedList

In this part of the assignment, you will focus on implementing iterator methods for the `ArrayList` and `LinkedList` classes. Both of these classes implement the `List` interface, which in turn extends the `Iterable<E>` interface. Due to this, both the `ArrayList` and `LinkedList` classes are required to provide an implementation of the `iterator()` method, which returns an `Iterator<E>` for traversing the elements of the collection.

You will also be responsible for completing the implementations of the `ArrayListIterator` and `LinkedListIterator` classes, both of which define an `Iterator<E>` that facilitates sequential access to the elements in their respective collections.

### Key Concepts:

1. **ArrayList and LinkedList Classes**: These classes implement the `List` interface and therefore must provide an `iterator()` method to return an `Iterator<E>`. This iterator enables you to iterate over the collection using the enhanced `for` loop or manually by calling `hasNext()` and `next()`.
2. **ArrayListIterator and LinkedListIterator Classes**: These classes handle the actual iteration over the elements of the `ArrayList` and `LinkedList`, respectively. You will implement the core iterator methods (`hasNext()`, `next()`, and any other required methods) in both of these classes.

### Tasks:

In the provided code, comments have been placed to indicate the specific tasks that you need to complete. These comments will guide you in implementing the required functionality for the iterator methods in both the `ArrayList` and `LinkedList` classes, as well as in the corresponding `ArrayListIterator` and `LinkedListIterator` classes.

You are expected to ensure that the iterators behave correctly and efficiently, enabling seamless traversal of both `ArrayList` and `LinkedList` collections.

# Part 3 Implementing Comparators

## Overview

In this section of the assignment, you will focus on the other fundamental concept in Java that is closely tied to ADTs and collections.

1. **Comparator**: An interface used to define a partial order on objects by determining, for two given instances of the same type, which object should be considered to "come first."

You will be working with three classes: `CasedURLComparator`, `LargestPageComparator`, and `WebPageRecord`. Each class is designed to handle comparisons between `WebPageRecord` objects using different criteria.

The goal of this part is to practice using the `Comparator` and `Comparable` interfaces in Java, which are essential for ordering and comparing objects in a structured way. As with earlier parts of the assignment, comments indicating specific **Tasks** have been added to the code to guide you in completing the necessary methods.

### Instructions

1. **CasedURLComparator**: This class implements the `Comparator` interface and compares two `WebPageRecord` objects based on their URLs. The comparison can be either case-sensitive or case-insensitive, depending on the `ignoreCase` flag passed to the constructor.
    - **Task**: Implement the `compare()` method to handle case-sensitive or case-insensitive comparisons of the URLs.
    - **Hint**: Consider using `String` methods like `toLowerCase()` if `ignoreCase` is `true`.
2. **LargestPageComparator**: This class also implements the `Comparator` interface but compares `WebPageRecord` objects using the following priority order:
    - First, compare based on the `length` of the web pages, where larger pages come first.
    - If the `length` values are equal, compare the length of their `firstLine` strings.
    - If both the page length and first line lengths are equal, compare the URLs lexicographically.

- **Task**: Implement the `compare()` method according to this multi-step priority comparison.
- **Hint**: Use `Integer.compare()` to compare the lengths, and remember to handle ties by comparing URLs as a last resort.

3. **WebPageRecord**: This class implements the `Comparable` interface and defines the natural order of `WebPageRecord` objects. The order is primarily based on the lexicographic ordering of their URLs, but if two `WebPageRecord` objects have the same URL, the `lastRetrieved` field is used as a tiebreaker.
    - **Task**: Implement the `compareTo()` method to compare `WebPageRecord` objects based on the lexicographic order of their URLs and, if necessary, their `lastRetrieved` timestamps.
    - **Hint**: Use the `String`'s `compareTo()` method for comparing URLs and the `Instant`'s `compareTo()` method for comparing timestamps.

**Guidance**

Throughout these classes, comments labeled **TASK** have been provided to guide you through the implementation process. Be sure to read the comments carefully, as they outline the specific functionality each method should implement.

- **Comparator**: In Java, the `Comparator` interface allows you to define custom sorting orders for objects. You will use this interface in `CasedURLComparator` and `LargestPageComparator` to compare `WebPageRecord` objects.
- **Comparable**: The `Comparable` interface is used when objects have a natural order. In the `WebPageRecord` class, you will implement the `compareTo()` method to define how `WebPageRecord` objects should be ordered based on their URLs and retrieval times.

By completing these tasks, you will strengthen your understanding of how to implement and use comparators to sort and order objects in Java. Test your implementations thoroughly using the provided unit tests to ensure correctness.

Here's a revised, clearer, and more professional version of the instructions for submitting the assignment:

# Submitting the Assignment

Once you have completed all necessary changes to your code, you will need to export your entire Java project as an archive file for submission. Follow the same process outlined in previous assignments to generate a `.zip` file of your project. Upload this `.zip` file to Gradescope.

If you would like to expedite the upload process, you may compress only the `src/` directory. This is the only part required by the autograder to evaluate your submission.

Please note that you can resubmit your assignment as many times as needed until the deadline. If your code does not pass all of the tests on your initial submission, you are encouraged to continue making

improvements and resubmit. Final manual grading will not begin until after the late submission deadline has passed.

**Reminder:** There are 3 parts to this assignment. Make sure you complete each part of the assignment before your final submission.