# CS 171: Introduction to Computer Science II
# Assignment #4: Playlist Application and Sorting

**[Due: on Gradescope ]**
**[Late submission: see syllabus]**

**Problem Description:** We are interested in developing a Playlist application that supports adding, deleting, and even sorting podcast episodes (alphabetically). To make navigating our Playlist flexible, we looked for a data structure that supports dynamic adding and removing of episodes, while also supporting easy and smooth traversing back and forth between episodes.

For these reasons, we will be using a **Doubly Linked List** to implement our Playlist, where each episode is represented as a node. Recall that each node in a doubly linked list has two links, one that refers to the *next* node and another that refers to the *previous* node. For our Playlist, this design choice means it will be easy to move forward or backward from any episode!

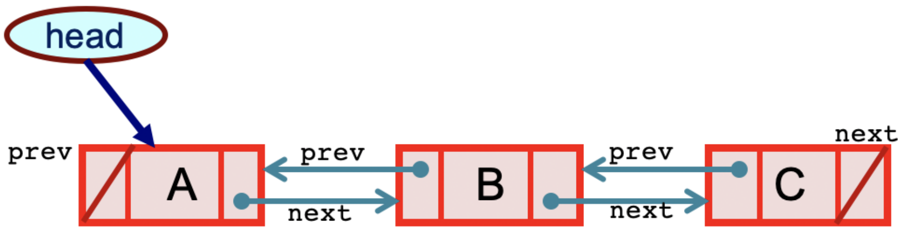
The starter code includes three Java classes:

(1) Class **Episode** represents an individual podcast episode. You can think of it as the equivalent of a 'Node' class in our lecture examples, but with more fields and methods. Each episode has a title, length (duration), a link to the next episode, and a link to the previous episode. Additionally, class Episode implements the Java interface **Comparable**, which enables comparing two episodes using the method `compareTo`. Please do not modify or add anything to this class.

(2) Class **Playlist** represents a doubly linked list of Episode objects. The playlist has two instance variables: the `head` (i.e. the first episode in the list), and a `size` integer (i.e. the total number of episodes so far). You will implement all the operations supported in this Playlist by filling the code for its member methods. Remember to always make sure that the Playlist's `head` and `size` variables are updated properly for each operation (e.g., the size is incremented by 1 if a new node is added, etc.). **The file Playlist.java is the only file you must submit to Gradescope.**

(3) Finally, class **ITunes** represents the application that will *test* the different features supported in your Playlist. You can modify the code in this class as you wish, to test and debug your Playlist implementation.

Note that the Playlist may contain no episodes (e.g. when it is first created), a single episode, or multiple episodes. The figure below shows how the nodes should be linked if there are multiple episodes in the list. Pay attention to how `next` and `prev` are wired. Also, note that de-referencing links in your code more than once is possible; so `episodeObject.prev.next` is valid (assuming `episodeObject` is an object of type Episode).



Example of a doubly linked list with multiple nodes.

## Part 1: Playlist Operations

Below is a summary of the methods in class Playlist relevant to Part 1 of this assignment. Some are already implemented (you can use them for testing and debugging), and others are up to you to implement:

- [**Implemented**] `String toString():` Our overriding of toString() prints the Playlist's episodes starting at the first (head) episode and ending at the last episode.
- [**Implemented**] `String toReverseString():` This method prints the Playlist's episodes *backwards*, starting at the last episode and ending at the first (head) episode. It utilizes the `prev` reference in each node to be able to move backwards.
- [**Implemented**] `int getSize()` returns the value of the size variable, and `boolean isEmpty()` returns true if the list is empty.
- [**13 points**] **TODO:** `void addFirst(String title, double duration):` Create a new Episode using the given title and duration parameters, then add this Episode properly at the beginning of the current Playlist.
- [**13 points**] **TODO:** `void addLast(String title, double duration):` Create a new Episode using the given title and duration parameters, then add this Episode properly at the end of the current Playlist.
- [**15 points**] **TODO:** `Episode deleteFirst():` This method should properly delete and return the first Episode in the playlist. If the list is empty, it should throw a `NoSuchElementException()`.
- [**16 points**] **TODO:** `Episode deleteLast():` This method should properly delete and return the last Episode in the playlist. If the list is empty, it should throw a `NoSuchElementException()`.
- [**16 points**] **TODO:** `Episode deleteEpisode(String title):` This method should properly delete and return the Episode with the given title. If the list is empty or the episode title is not found, it should throw a `NoSuchElementException()`.

## Part 2: Sorting the Playlist

What if the playlist user is interested in having all the episodes sorted alphabetically on their app? To provide this cool feature, you will implement a *sorting* algorithm for our playlist. But which algorithm should we use? For sorting linked lists, **mergesort** is often preferred since it can be implemented with $O(1)$ extra space, and since random-access algorithms like quicksort perform poorly on linked lists.

In the starter-code, we provided a partial implementation of mergesort for the Playlist, which is missing the `merge()` operation only. Your job is to provide the implementation of merge (without creating any auxiliary data structures!). Note that both splitting and merging nodes in a linked list is done by re-linking the nodes' prev and next references properly, without the need for creating an additional data structure to copy the elements back and forth (like we did for regular arrays in our lecture's mergesort examples).

Here is a summary of the sorting methods in class Playlist:

- [**Implemented**] `Episode getMiddleEpisode(Episode node):` Since mergesort requires repeated splits of the list into two halves, we provided this helper method that identifies the *middle* node in the list by defining two pointers to traverse the list: a `slow` pointer that hops one node at a time, and a `fast` pointer hops two nodes at a time. When `fast` reaches the end of the list, `slow` will be about half way. Elegant, eh?

- **[Implemented]** `Episode sort(Episode node):` Similar to our mergesort class example, this method splits the playlist recursively into two halves, and then calls the method `merge` that you will implement to merge the two sublists. The parameter `node` is a reference to the beginning of the list to be sorted.
- **[22 points] TODO:** `Episode merge(Episode a, Episode b):` This is the only method you need to implement. It is responsible for merging two sorted lists into one sorted list. You can approach this problem iteratively or recursively (your choice!). Your implementation must use the method `compareTo()`, available in class Episode, whenever you need to compare two Episode objects. Tip: There are several cases to deal with when merging two lists: either list could be empty, you may run out of elements in 'left' first, 'right' first, and so on.

## Getting Started and Debugging Tips

- Download the starter code on Canvas and understand it: **Episode.java**, **Playlist.java**, and **ITunes.java**.
- Start by solving the **add** and **delete** methods in class Playlist. Test your code **incrementally** and work on one method at a time. In class ITunes, you can test and debug each method by printing the list using both **toString()** and **toReverseString()**, to ensure that nodes are linked properly in both directions (via `next` and `prev` references).
- Then, move on to the **sorting** methods. Read the given methods very carefully and think about how the merge operation should be designed. Test your logic on a small concrete example first (on paper), then debug and test your implementation in class ITunes.
- Test your methods on different scenarios; e.g., an empty list, a list with one node, multiple nodes, deleting a node from different positions in the list, etc.
- The **only** file you will submit on Gradescope is **Playlist.java**.

## Grading

- If your program does not compile, you will get 0 points from the autograded portion.
- Playlist operations are properly implemented with correct logic and exception handling: 73 points
- Playlist mergesort correctly sorts the episodes in alphabetical order and is implemented efficiently (does not time-out on Gradescope): 22 points
- Code clarity, style, and comments: 5 points

## Honor Code

The assignment is governed by the College Honor Code and Departmental Policy. Remember, any code you submit must be your own; otherwise you risk being investigated by the Honor Council and facing the consequences of that. We do check for code plagiarism (across student submissions and against online resources). Please remember to have the following comment included at the top of the file.

```
/* THIS CODE WAS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING
CODE WRITTEN BY OTHER STUDENTS OR ONLINE RESOURCES.
_Student_Name_Here_   */
```

**Submission Checklist:** We've created this checklist to help you make sure you don't miss anything important. Note that completing all these items does not guarantee full points, but at least assures you are unlikely to get a zero.

- ☐ Did your file compile on the command line using JDK 11 or above (e.g. JDK 17)?
- ☐ Did your submission on Gradescope successfully compile and pass at least one autograder test case?
- ☐ Have you included the honor code on top of the file?
- ☐ Did you remove the `TODO` prefix from the methods you needed to implement?
- ☐ Did you give your variables meaningful names (i.e., no `foo` or `bar` variables)?
- ☐ Did you add meaningful comments to the code when appropriate?