COMP-10205 – Data Structure and Algorithms

Assignment#4
Array Lists and Linked Lists

Submission Requirements

Complete the following assignment and submit electronically in the assignments folder on MyCanvas as an IntelliJ Project – Zip the entire folder (Assignment4.zip). This assignment must be completed individually. You may not use ChatGPT or any other AI coding assistant.

Background / Problem Summary

The goal of this assignment is to complete the code and compare the timing for three classes `ArrayList`, `SortedLinkedList` and `SortedArray`. You will also test a primitive array of strings. Each data structure must keep its list sorted after each `add` operation. This will look different for each list:

- `SortedLinkedList` and `SortedArray` will handle this problem internally during the `add` method.
- The `ArrayList` will `add` a single element to the list and then `sort` after each `add` operation, in order to maintain a list of sorted order.
- The primitive `array` will also be sorted with the supplied `iSort`() method.

You have been provided some starter code for the main method in `A4Main.java`, and a data file called bnames.txt.

Part A – Implementing the code

You will need to add the following functionality:

- `SortedLinkedList` and `SortedArray` `add()` method
  - adds the element in sorted order to the list
  - For example when you add the following words in this order [Bob, Carol, Aaron, Alex, Zaphod], the list when printed using the `toString` method will appear as [Aaron, Alex, Bob, Carol, Zaphod].
  - Each element will be added to the list in its proper sorted location, so that the list is always in a sorted state. It is **NOT** acceptable to add the item to the end of the list and then sort it.
- `SortedLinkedList` and `SortedArray` `get()` method
  - Gets the element by its index from the data structure. The behaviour should be similar from the users perspective to `ArrayList.get(index)`
- `SortedLinkedList` remove – removes the item from the list

The classes provided are generic classes. They will work with any `Object` that is `Comparable`.

The code for testing your methods has been provided in `A4Main.java`.

The supplied method `ckSumSorted()`, originally presented in Assignment 2, has been updated to work with `Strings`. However, it only works with simple arrays. The supplied test code copies data from each sorted data structure into an array of strings so the checksums can be validated.

Once you have the code working for the Strings, add to `runPartA` method a second linked list that holds any other comparable object. You may use <`Integer`>, <`Double`> or <`Long`> for example. Show that you can add and remove items from this second linked list of a different data type.

Part B – Timing the code

In the `runPartB` method you will read all of the names from bnames.txt into a `ArrayList` (do not use `SortedLinkedList` or `SortedArray`). Code is provided at the top of Part B. The data file in bnames.txt is not sorted. When chkSumSorted is called it should return -1.

Code is provided that copies the first ArrayList into a 2nd ArrayList and sorts the data after each insertion to ensure the list is always sorted. The sample code also verifies the checksum at the end. This will be our library base case that we are trying to beat. The checksum from this case should tell us whether our new classes are working correctly or not in the other cases.

A version of insertion sort is supplied that works with primitive arrays. Fix the `iSort()` method so it takes two parameters and only sorts the first n elements. Create a simple array large enough to store the entire `ArrayList`, copy items in one at a time, and call `iSort()` after each copy so that the list stays in sorted order. Ensure that the collection of output messages for this test is similar to the messages for the supplied test, and that the checksum is correct.

Write similar performance tests for `SortedLinkedList` and `SortedArray`, follow the example output style used in the previous tests. These methods have built in sorting capabilities, so you don't need to call a sort method after each update. The messaging should drop the detail that a full sort is being done after each insert. Both methods should be doing a single special insert with time complexity O(n).

Once the main is completed you should have timing and output for 4 different methods:

- `ArrayList` using built in sort
- Primitive `array` using `iSort`
- `SortedLinkedList`
- `SortedArray`

Add a Discussion of performance in a comment to the top of the file that contains your main method discussing the results obtained. Answer each of the following questions:

1) Summarize your timing results for the 4 different methods in a simple and concise table at the top of the assignment.
2) Explain any performance difference between the `SortedLinkedList<T>` and the `SortedArray<T>` classes when adding items. Explain why they don't run at the same speed. Is this the result of a different algorithm? Or something else?
3) When would you choose to use a `SortedLinkedList` over an `SortedArray` based on the results of this assignment? Is speed the only factor?
4) `ArrayList` uses the built-in sorting method. Look up the documentation for it. This method has a very good time complexity. You should notice that sorting after adding each item is slower than using the `SortedLinkedList`. The `SortedLinkedList` class is essentially the same as `insertion sort` when a collection of items are added (inserted) into the correct spot. Why is it that `SortedLinkedList` is able to do better than the built-in sort which uses a much faster method?
5) Our method `iSort`() should be orders of magnitude slower than the built-in sorting method. Is it? If not why not? `SortedLinkedList` uses insertion sort. Is it faster? Why?

Marking Scheme

Program structure – JavaDoc, follows best programming practices - 15%
Completion of `LinkedList` Requirements – 25%
Completion of the `SortedArray` Requirements – 10%
Reading from file and print timing results to screen – 15%
Performance comparison write-up as a comment at the top of the code. – 35%