Ordered linked list

Overview

You're asked to implement a generic data structure called ordered linked list and use it in a simple application. The required methods are listed in the given interfaces. You need to implement them according to the descriptions below.

Logistics

Submission is the same as assignments (github + Canvas). Late submissions will not be graded. Make sure your submission code compiles. Those that don't compile will be heavily penalized!

Don't modify the given interface files. You can modify anything in the implementation classes but the class names.

In Part 1, you are **not** allowed to use any built-in stuff such as ArrayList, LinkedList, Collections.sort(), etc. In Part 2, you can use anything from the Java libraries. If you use built-in stuff to solve Part 2 instead of the data structure from Part 1, you’ll still get full credit.

No requirement on style, comment, or unit test. You can add any helper public/private methods.

Problem statements

Part 1: Data Structure

The ordered linked list is different from ordinary linked list in that 1) the elements in the linked list are ordered in **nondecreasing** order; 2) it has a capacity/length limit. For example, for an ordered linked list of capacity 5, Case 1 is valid, Case 2 and 3 are invalid.

a. 2 -> 5 -> 10 -> 24
b. 2 -> 5 -> 10 -> 24 -> 27 -> 29 (capacity limit exceeded)
c. 2 -> 1 -> 10 -> 24 (out of order)

Note that the examples are using integer, but your implementation needs to be generic!

Problem 1: Constructors (5pts)

One constructor of OrderedListImpl takes an integer argument that specifies the capacity limit and creates an empty linked list. The constructor should throw *IllegalArgumentException* if the specified cap is less than 0. Apart from the required constructor (needed for grading tests), you can also add whatever overloaded constructors you want.

Problem 2: toString (10pts)

You’re required to override *toString():*
toString() should convert the list to a string in the following format:
[e1 e2 e3 ? ? ?] (brackets are needed, elements separated by one space)
The question marks represent the unused capacity. For example, an ordered linked list with capacity 7 and elements 3->89->1289 will be printed as:
[3 89 1289 ? ? ? ?]

Problem 3: add (10pts)

The add() method adds an element to the list. After adding the new element, make sure the list is still ordered.

When add() tries to add an element while the list has already reached its capacity limit, there will be two cases:

1. If the element to be added is larger than the current minimum element in the list, the new element will be added and the minimum element will be removed to make sure the capacity is not exceeded.
2. If the element to be added is smaller than or equal to the current minimum element in the list, the new element will not be added to the list. In other words, nothing will happen.

Problem 4: getMax (5pts)

The getMax() will return the maximum element in the list. It returns null if the list is empty.

Problem 5: merge (10pts)

The *merge(OrderedList other)* method merges two ordered linked lists. After merging, a new ordered linked list will be returned and the two original ones should remain unchanged. The new ordered list should have a capacity that equals the sum of the capacities of the two original lists. The new list will have all the elements from the two original lists, linked in nondecreasing order. If `other` has a different implementation type than `this`, an *IllegalArgumentException* should be thrown (hint: instanceof is okay to use).

Problem 6: resize (10pts)

The resize(int newCapacity) method changes the capacity of the list to `newCapacity`. If the number of elements in the list is larger than `newCapacity`, it will discard the smallest elements to make the size of the list `newCapacity`.

For example:
Before resize: 8 -> 12 -> 14 -> 17
Call resize(3);
After resize: 12 -> 14 -> 17

Part 2: Application

Now that we have the data structure ready, let’s use it in a simple application.

We want to build a game score history system for each game player. It keeps the top-N game scores of that player for each month. Then at the end of the year, it merges the per-month top-N scores and produces the player’s top-M game scores of the year. You’re asked to implement the given interface.

Try to take advantage of the ordered list you built, it should make the implementation of this application very simple. If you don’t have the ordered list implemented, you’re allowed to use any built-in data structure and algorithms.

Problem 1: Constructor (5ps)

The constructor takes two integers, one defines the value of N in top-N, the other defines the value of M in top-M. For example, TopScoreTrackerImpl(2, 4) will create a tracker that keeps the top-2 game scores for each month, and produces the top-4 game scores of the year.

Problem 2: addRun (5pts)

The addRun(int m, Run r) method tries to add Run \`r\` to the top-N score list of month \`m\`. If there are currently less than N runs in the top-N list, \`r\` will be added unconditionally. Otherwise, \`r\` only gets added when the score of \`r\` is larger than the existing lowest score in the top-N list.

Problem 3: annualReport (10pts)

The annualReport() method merges all the 12 per-month top-N lists and produces a sorted (non-decreasing order) list of top-M gameplay scores of the year. Note that it returns a String.
The String should be formatted as [(run_id, run_score) (run_id, run_score) ….], separated by space. It should have exactly M elements.
For example: [(3, 578) (1, 874) (12, 1233) (10, 1555)] for TopScoreTrackerImpl(2, 4).