Problem Solving Practice

CIT 5940

January 24, 2025

DO NOT WRITE ANYTHING ON THIS PAGE!

In this activity, you will design and partially implement a `Leaderboard` class. A `Leaderboard` is an ordered and sorted listing of `Score` records. The `Score` class is implemented with the following instance variables:

```
public class Score implements Comparable<Score> {
    private int points;
    private String name;


    ...
}
```

A `Leaderboard` will store `Score` records in descending order where the order is determined by the points values of the records. A `Leaderboard` can be updated by adding a new `Score`, although there is no guarantee how the new `Score` compares to previously added ones.

Question 1. Understand the Problem: Choice of Data Structure

Arrays and Lists in Java are both ordered structures. They can also both be managed so that they remain sorted. Compare the **affordances** of Arrays vs. Lists and recommend which one of these two you would suggest using.

Solution:

Arrays are fixed in size and support random access/updating. It is not trivial to insert within an array nor to track the number of non-empty spaces. Lists grow and shrink based on the elements added/inserted/deleted, which is all handled for you. It does require additional planning to implement a maximum size for a List, but this is not a concern here, and anyway it's easier to do than implement a random access insert operation for Arrays, so I will suggest that a List is the suitable choice.

Question 2. Write Tests

The `Leaderboard` class will include a method `int addNewScore(Score s)` that takes in a new `Score` object, adds it to the proper position in the `Leaderboard`, and then returns an `int` representing the position of the new record, where 0 would be the position of the record with the highest point value, followed by 1 for the second-highest, and so on.

Write one test case for each of the following user stories that verifies the intended behavior. You can assume that the `Leaderboard` class includes a no-argument constructor for initializing an empty `Leaderboard` with a default maximum size of 100 and a method `int size()` that returns the number of records contained in the `Leaderboard`.

1. A new `Leaderboard` is created, which should be empty at first. Then, a first score is added. Test that the `Leaderboard` is empty after being constructed, and that it has the proper size after the first `Score` is added and that `addNewScore()` returns the correct value after the first addition.

2. A new Leaderboard is created, and then three scores are added to it in some order. A new score is then added which is neither the highest nor lowest score. Test that after this new score is added that the Leaderboard has the correct size and that the correct value is returned by addNewScore().

Solution:

Perfect Java syntax is not important. There are also several similar solutions that are acceptable.

```
@Test
public void testAddEmpty() {
    Leaderboard l = new Leaderboard();
    assertEquals(0, l.size());

    int actual = l.addNewScore(new Score(5, "Harry"));

    assertEquals(0, actual);
    assertEquals(1, l.size());
}

@Test
public void testAddThreeThenOne() {
    Leaderboard l = new Leaderboard();
    for (int i = 0; i < 3; i++) {
        l.addNewScore(new Score(5 * i, "Harry"));
    }
    int actual = l.addNewScore(new Score(7, "new"));
    assertEquals(4, l.size());
    assertEquals(1, actual);
}

```

Question 3. Analyze the Runtime: Adding a New Score

What is the runtime of this array-based implementation of addNewScore? Justify your answer. `System.arrayCopy()` is a handy built-in for copying n elements from an array in $O(n)$ time.

```java
    public int addNewScore(Score s) {
        // If scores is full, copy elements over to a bigger array before starting
        if (numEntries == scores.length) {
            Score[] newScores = new Score[scores.length + 10];
            System.arraycopy(scores, 0, newScores, 0, scores.length);
            scores = newScores;
        }
        // Find the index where the new score should live...
        int idx = 0;
        while (idx < numEntries && scores[idx].compareTo(s) < 0) {
            idx++;
        }
        // ...shift all elements at this position over to the right by one...
        System.arraycopy(scores, idx, scores, idx + 1, numEntries - idx);
        // ...and place the new Score.
        scores[idx] = s;
        return numEntries++;
    }
```

Solution:

The first conditional related to making room for new elements is linear ($O(n)$) in the number of elements already present in the leaderboard since all n existing elements must be copied one-by-one.

The while loop to find the index of insertion may take $O(n)$ time in the worst case because the search starts at the left (`idx = 0`) and increases up to `numEntries` $= n$. Even if you are lucky and insert at an early index, the number of elements copied with `System.arraycopy()` is as many as `numEntries - idx`, which is n when the index of insertion is 0. Therefore, the process of finding the insertion point and inserting there is always $O(n)$ (in fact, $\Theta(n)$).

The rest is constant time.