AP/ITEC 2610 Fall 2020 Sections C/D

Assignment 3

Task Description

Please write a program in Java to sort and search the scores for the participants in a game. The public interface of **ScoreManager** is described below, along with a ready-to-use class **ScoreRecord**. Please follow the requirements in the description for each method to complete this class. Please also write a tester class to thoroughly test the four methods given in **ScoreManager.**

The **ScoreManager** class should contain the following public methods.

1. **`public ArrayList<ScoreRecord> readRecords(String pathName) throws IOException:`**
   - Read all the records from the **binary file** in the local file system as specified by the parameter `pathName` and return an `ArrayList` of `ScoreRecord`. The definition of `ScoreRecord` class is provided below.
   - We assume that each record in the file has a fixed length of 50 bytes, with name stored in the first 46 bytes and score stored in the following 4 bytes.
   - We assume that all the scores are non-negative integers. There could be an arbitrary number of records in the file.
   - A sample binary file containing 10 records is attached to this handout.

2. **`public ArrayList<ScoreRecord> mergeSortRecords(ArrayList<ScoreRecord> inputList):`**
   - Use the **merge sort** algorithm to sort all the records in `inputList` by score in descending order.
   - Use `System.out.println()` to output the intermediate results in each step during the merging process. For example, if at one step, two lists `[(Bill,78),(Tiffany,69),(Bob,65)]` and `[(John,90)]` are merged, the output should be
     ```
     Intermediate result: [(John,90),(Bill,78),(Tiffany,69),(Bob,65)].
     ```
     One line per result.

3. **`public void saveScores(String pathName, ArrayList<ScoreRecord> sortedList) throws IOException:`**
   - Use `RandomAccessFile` to save all the ordered records from the `sortedList` to the given `pathName` on the disk.

4. **`public ArrayList<ScoreRecord> scoreSearch(String pathName, int minScore, int maxScore) throws IOException:`**
   - Given a score range `[minScore,maxScore]` (both boundaries are inclusive) and the `pathName` of a file that stores records sorted by score

(as produced by `saveScores`), use **binary search** to find and return an `ArrayList` of all the records within this score range **without loading all records in the file into memory (i.e., do NOT attempt to read all records into an array/ArrayList and then perform the search there)**. For example, given the range `[69,80]` and the file which contains `[(John,90),(Bill, 78),(Tiffany,69),(Bob,65)]`, it should return `[(Bill,78),(Tiffany,69)]`. If no records fall in the score range, return an empty list. To accomplish this, you can adapt the binary search algorithm (for arrays) taught in class to the setting of random access files. To retrieve records in a range, you may use binary search to locate the record with the given `minScore` within the range (or if such a record does not exist, the first record with a score higher than `minScore` in the file) and then visit the subsequent records until the score is out of the given range or the end of the file is reached.

- Use `System.out.println()` to output each record visited during the search, in the format of

  ```
  Search visit: (Bill,78)
  ```

  One line per record.

Tips:
1. Use RandomAccessFile().seek(long pos) to do random access;
2. Use RandomAccessFile().length() to get the size of file (number of bytes).

ScoreRecord.java

```java
/**
* A score record with a participant name and score.
*/
public class ScoreRecord {
  private String name;
  private int score;

  /**
   * Constructs a score record.
   * @param name  the name of the participant
   * @param score  the score of the participant
   */
  public ScoreRecord(String name, int score) {
     this.name = name;
     this.score = score;
  }

  /**
   * Gets the name of the participant
   * @return  the name
   */
  public String getName() {
```

```
        return name;
    }

    /**
     * Gets the score of this record
     * @return  the score
     */
    public int getScore() {
        return score;
    }

    /**
     * Returns a string representation of (name, score) pair
     * @return  the string representation of this record in the format of (name, score)
     */
    public String toString() {
        return "(" + name + ", " + score + ")";
    }
}
```

What to submit

Submit ScoreManager.java only. Please use the exact file name given here and document it properly using JavaDoc. Nothing else (including ScoreRecord.java, the tester class, and the data files) should be submitted. Compile and test your code under the command line environment (e.g., by executing Terminal under macos or cmd under Windows).
DO NOT submit .class files! .java file ONLY.

Important Note

(1) Your assignment **will be given a zero** mark if only the compiled files (.class files) are submitted. Please make sure to submit the source file (.java file).

(2) Please make sure your code compiles under the command line (i.e., without an IDE). **Do not put any package statement at the beginning of your source file.** If you are using an IDE, this is especially important because some IDEs put your code under a particular package for your project. Any code that does not compile under the command line can only receive 20/100. Remarking requests like "...but the code works on my computer/in my IDE" will **not** be entertained.

Marking Scheme:

```
Style (variable naming, indentation, & Layout)   _____/10

JavaDoc Comments                                 _____/10
```

Code Compiles?	_____(yes/no)
Successful Execution of Test Cases	_____/80
Total	_____/100

According to this marking scheme the maximum mark you can get for code that does not compile is 20/100.