Department of Computing

COMP225 Assignment 1 Specification

Due date: 10.00am, 1 May, 2017
Assignment marks: 20% of overall unit marks.

Objective: *To understand that data structures have strengths and weaknesses, and to use the features of the data structure to get the best performance out of the algorithms using it. You will also learn how to use sorting for problem solving.*

Please note: This assignment specification aims to provide as complete a description of this assessment task as possible. However, as with any specification, there will always be things we should have said that we have left out and areas in which we could have done a better job of explanation. As a result, you are strongly encouraged to ask any questions of clarification you might have, either by raising them during a lecture or by posting them on the *iLearn* discussion forum devoted to this assignment.

Overview: Shuffling data
We are all used to the idea of sorting data in order to process it more effectively. But for some applications the opposite is required: to have data presented to us randomly jumbled. Music players like iTunes, iPods and iPhones have a "shuffle" setting for example: this could be called an anti-"boredom" application.

It turns out that randomly shuffling data is not an easy thing to get right on a computer even if using an accurate random number generator: in fact it's easy to get wrong! For more information on why that is so have a look at the resources here, attend the lectures, and read some old reviews of the iTunes shuffle facility here.

The original "Fisher-Yates" shuffle algorithm was popularised by Donald Knuth. Read more about it here. It runs in O(n) in the worst case, but uses direct addressing on an array. What if it turns out that the data you want to shuffle is not stored on an array? Could you still use Knuth's shuffling algorithm directly? (i.e. without copying the items to an array.) In this assignment you'll learn how to shuffle a linked list of items using sorting. This is rather slower (O(nlog(n)) rather than O(n) Why?), but conceptually it's much easier to understand and to get right. And it works on linked lists, which then allows flexible length which is what is required for implementing play lists in music players.

This Assignment
In this assignment you will implement sorting and shuffling on a linked list, adapting the basic algorithmic strategies we have already studied and implemented on arrays. You will use your algorithms to program a simple "*MusicPlayer*" class which includes the functionality to shuffle. You have been provided with the following incomplete program file, and you are asked to complete it in the way described below.

MusicPlayer.java This is the class which deals with implementing methods to build and control the music files on eg an MP3 player. It consists of an embellishment of the linked list class we studied in Week 1, except that the associated node class (now called "*musicNode"*) has three data fields, one for the track name, one for the number of times the track has been played, and an extra field which is to be used for shuffling a playlist. You can find a template for your programs in the Assignment program bundle.

Task 1 asks you to implement *sortTrack* using the *mergesort* strategy we have studied in lectures. The challenge here is to avoid using the additional scratch space needed in the array implementation. You are also asked to complete the **Task 1 Quiz** associated with this assignment. Please read carefully the special instructions for completing the Quiz.

I shall run a special session on this Assignment in the Week 6 lectures to discuss ideas for achieving this using linked lists.

In **Task 2** you are asked to implement the shuffling and recommendation methods as set out in the specifications below.

Task 1 (60 marks total: implementation plus Task 1 Quiz)

(50 marks) Your first task is to complete the methods *moveFirstNode* and *sortTrack* in the class *MusicPlayer*. You are also required to write a representative selection of methods which form the basis for "sanity checking" the correctness of *sortTrack*.

Make sure that your implementation of *moveFirstNode(MusicPlayer fromList, MusicPlayer toList)* has worst-case O(1) complexity, that your implementation of *sortTrack* uses the basic *mergeSort* strategy and has O(n log n) complexity and (unlike the implementation on arrays) uses only O(1) extra "scratch space". You can add any additional service methods you need in order to structure your implementation.

```
void moveFirstNode(MusicPlayer fromList, MusicPlayer toList)
        // Removes the top node of fromList and puts it onto (the top of) toList.
        // If fromList is empty, it does nothing.

void sortTrack()
        // Sorts the current list alphabetically according to its track field.

    int countItem(String item)
        // Returns the number of times that item occurs in the current list

    musicNode checkMembership(String _track)
        // If the given _track is present in the current list (i.e. the node whose "track" field
        // is equal to _track), returns the address of that node;
        // otherwise returns null.
```

Look in the JUnit tests to see examples of expected behaviour. Variations of these tests will be used in the final automarking.

To help you to understand its design and performance you are also asked to complete the Task 1 Quiz.

Task 1 Quiz (10 marks)

The quiz consists of multiple-choice questions to test your basic knowledge of linked lists and of how an implementation of sorting on linked lists compares with the corresponding implementation using arrays discussed in lectures. **T**he results will become available after the submission for the assignment has closed and all student submissions have been received. You have up to two attempts, and each attempt has a time limit. Please make sure therefore that you are happy with your answers before you submit.

Task 2 (40 marks total: implementation plus Task 2 Quiz)

The second task is to complete the various shuffle and recommendation functions.

The MusicPlayer is made up of a linked list of tracks. A track has a track name, a played field and a shuffleTag. Each time a track is played (by calling playTrack) the played field is increased by

one. The shuffleTag can have a random number assigned to it, and can therefore be used in the shuffling algorithms.

Task 2a (10 marks)

shuffle produces a random shuffle of all the tracks in the list; this means that when you run shuffle it should re-order the items in the list so that they are some random permutation of the original list. You will of course need to generate random numbers to enable you to complete this task. You may use the Java Random class which is part of the java.util.*; libraries. We saw an example of how that was used in the Widgets class which is part of the Week 2 program bundle. Recall that the method nextInt(N) returns a random number between 0 and N. Although it is not possible to implement the Fisher-Yates shuffle on a linked list, it is possible to implement shuffling using the random number generator and the sorting algorithm you implemented for Task 1. This is how you should implement shuffle.

Task 2b (10 marks)

smartShuffle also produces a random shuffle of all the tracks in the list, EXCEPT that no track appears in the shuffled list after any other track with higher played field. For example if there are 9 files, of which 2 have never been played, 3 have been played once and 4 have been played 4 times. Running *smartShuffle* on this playlist will mean that both the unplayed tracks will appear before the 3 single-time played tracks, and all of these will come before the remaining 4 tracks. However inspecting the two single played files, you'll see they are randomly shuffled; similarly the three singly-played files, and similarly the remaining 4 files.

Task 2c (10 marks)

recommended now tries to create a playlist that is based on browsing preferences. It uses a history of listening activity and creates a playlist that orders music it thinks you like. It does this by searching for the most frequently listened-to tracks in a list that logs tracks listened to or browsed for; it then rearranges the playlist so that tracks now appear only if they appear in the history list and ordered from most popular to least popular. (This is opposite to shuffling, and perhaps is just the “comfort” setting…)

Given a list of listening history (as an array of names of tracks listened to in order), *recommended*, computes which tracks appear both in the current playlist and the history list, and then removes all nodes that are not in the intersection. It then sorts the remaining nodes in playlist so that the most frequently listened-to track in the history list is first in the playlist, then the next most listened-to track is next, and so on. Notice that a track which appears 5 times in the history list means that it was listened to 5 times.

Note that the tracks in the history list and the playlist could be different. Any track that is in the playlist but not in the history list should be removed from the playlist.

For example, if the playlist contains tracks [a, c, d, y] (in that order) and the history list contains tracks {c, b, a, a, y, c, c} then the playlist should become:

[c, a, y]

Other examples can be found in the JUnit tests in the assignment sources.

Task 2 Quiz (10 marks)

The quiz consists of multiple-choice questions to test advanced knowledge of shuffling on linked lists. As for Task 1, you can submit your solutions only once, and the results will become available after the submission for the assignment has closed and all student submissions have been

received. You have two attempts, and each attempt has a time limit. Please make sure you are happy with your answers before you submit.

The class files
Download the Assignment 1 bundle from the iLearn Assignment 1 links.

This assignment is structured to allow you to decide how much effort you want to expend for the return in marks that you might hope for. You can choose which bits of the functionality of the full application you feel confident about implementing and then only write those parts for a proportion of the maximum possible marks. So you can decide upfront whether you are shooting for a pass or a high distinction and know exactly how much work will be required to obtain that mark.

Here is what is required to obtain marks in one of the performance bands for this assignment:

Pass: Completion of Task 1, implementations and Quiz.

.

Credit: the P level implementation + Task2a and 2b.

Distinction: the Cr level implementation + Task2c + Task 2 Quiz

One thing you must not do is to implement the mergesort algorithm so that it either (a) uses additional O(n) scratch space , or (b) so that it runs slower than in O(nlog n) time. This is because in either of those cases you will not have learned how to use the features of the data structure to get the best performance out of the algorithms using it!

What you must hand in

Please submit a single file called *MusicPlayer.java* containing the implementations of all of your programs.

Please make sure that you do not add a package name.
Please be sure to remove all syntax errors before you submit. Occasionally eclipse ignores these errors but the automarker does not.
Please make sure that the methods you implement are all public.
If you decide not to implement a method, please do not remove the method stub.
You must not change the definition of the *MusicPlayer* class nor the *musicNode*. You may only add additional methods, but you must not add additional data fields to the class definitions.

Your source code should be submitted via the Assignment 1 link on the COMP225 iLearn.

This assignment will be entirely marked by an automaker. This means that it is crucial that you submit files in the right format and with the right names. Unfortunately we are unable to mark individual assignments, but we will do a trial run of the automaker on **26th April** so that you can check before the deadline that you have the right format. If you would like your programs to be trialled please make sure they are submitted **before 26th April.** You will be able to resubmit as many times as you like before the deadline.

The links to the quizzes are also available from iLearn. Please note that you only have two attempts, and that each attempt has a time limit.