Algorithms and Data Structures

Generics, Functional Interfaces and Recursion for Supermarket Statistics

Assignment-2

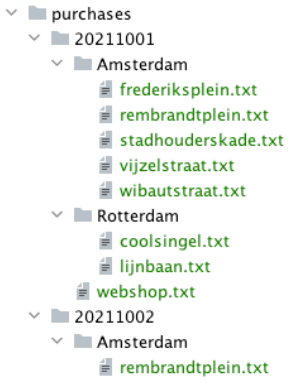
Version: October 1st, 2021

Introduction

In this assignment you will work on an application that calculates sales statistics from raw purchase information of a large Supermarket chain with branches across Europe.

The key questions are:

1. Which are the five products with ***lowest*** total volume of sales across all branches?
2. Which are the five products with ***largest revenue*** (= price * volume) from total sales?
3. What are the total volume and total revenue of all sales of all products?

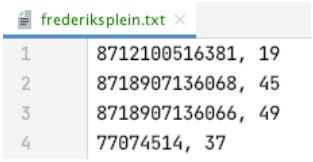


At the end of each day, each supermarket branch automatically uploads the daily raw purchase information to a central API. Headquarters collect these uploads into a vault of ordinary text files within a hierarchical folder structure. You will be given a small extract of this folder structure as shown in the picture at the left. Your solution shall run with direct use of this folder structure and shall be robust to deal with thousands of these files organized in many more hierarchical levels of the file vault.

Format of the input data.

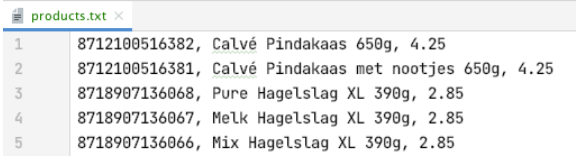
Each text file with raw purchase data contains the summary of sales of one branch of the supermarket chain at one given day. It contains a sequence of lines with each only two numbers separated by a comma:

1. The first number is the numeric barcode of a product that has been sold in the branch.
2. The second number is the amount of this product that has been sold on the given day.



There are no header data in the purchases file and no other information. The lines can be in any order.

Besides the vault with the raw purchase data, also a 'products.txt' file is provided that specifies descriptions and prices of all products available from the supermarket chain in any branch. Each line in this file contains three values separated by comma's:



The first value is the numeric barcode of the product.
The second value is a description of the product.
The third value is its price.

All products have a unique barcode number. The barcodes in the purchases files match the barcodes in the products file.

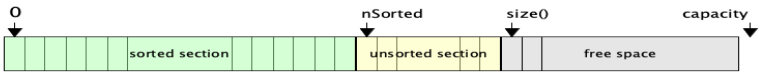
Solution approach.

Because we expect more requests like this from our client in the future, we want you to invest in a solid solution with reusable software that can also be used to meet future requirements. The basic design has been documented in below class diagram.

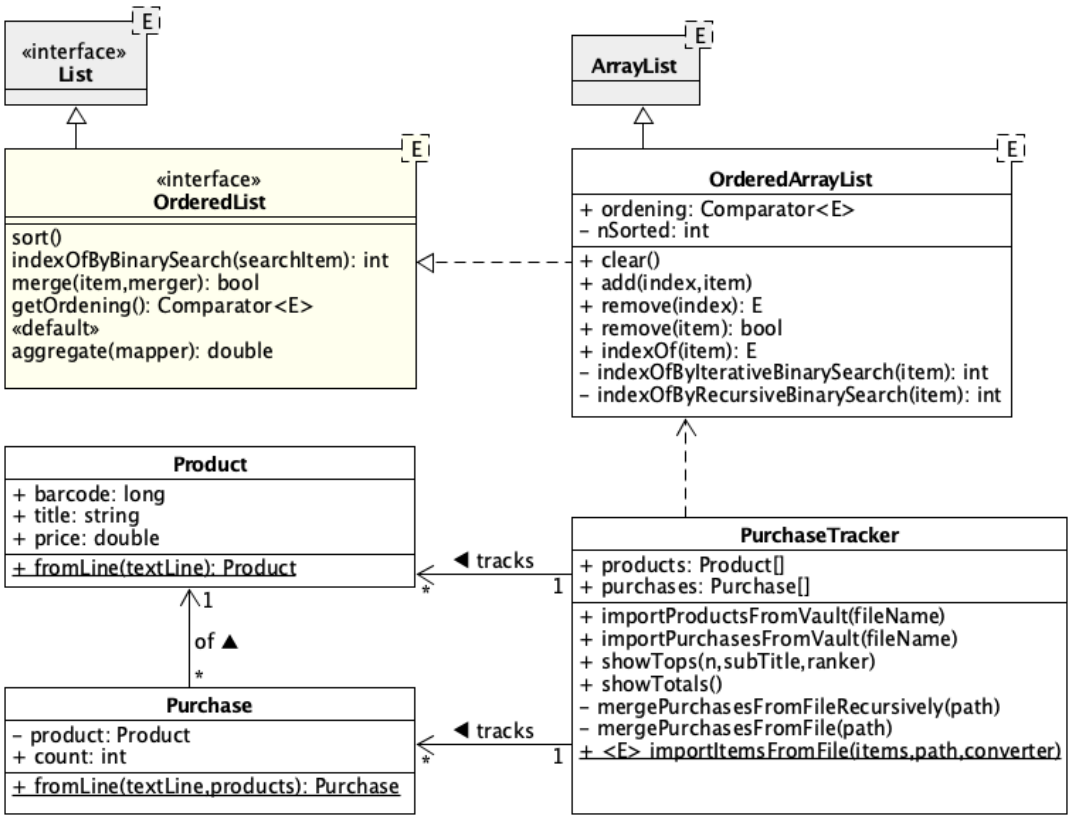
Product information will be captured in a Product class. Summarized purchase information will be captured in a Purchase class. The PurchaseTracker class implements the collections of all information and the logic to process the files in the vault and to produce the statistical results.

To support these computations, we believe it is a good design to implement a generic extension 'OrderedArrayList' of the ArrayList class. This 'OrderedArrayList' maintains an ordening on all its items, such that you can efficiently find an item by a binary search algorithm. It remembers the specific ordening being used by storing a reference to the Comparator that was passed to the latest sort. That way you can also use that Comparator in your binary search implementations.

You may think, what benefit will I have from the binary search if I must re-sort the arrayList each time when a new item is added. Hence, we introduce a representation invariant that also allows for adding, inserting and removing items without resorting:



Consider the first section of the arrayList to be sorted, and a (small) part at the end to be unsorted. The private instance variable nSorted tracks how many items are sorted. The indexOf method now can use binary search on all items 0 <= index < nSorted (the sorted section), and if the item is not found it shall try a linear search across all remaining items nSorted <= index < size() (the unsorted section). The add method can add a new item at the end without additional burden. If the unsorted part of the array has grown too large, the indexOf will be less efficient and your code should invoke a re-sort as appropriate. The sort methods repair the representation invariant by setting nSorted = size().



Assignment.

Unpack the starter project into your development environment and proceed with implementing the missing parts:

1. Complete the implementations of the Product and Purchase classes:
   a) you need to complete the toString method to obtain a proper text representation in output.
   b) you need to complete the static fromLine methods which you will need for converting text lines of the source files into object instances.
2. Complete the implementation of the generic OrderedArrayList class, which implements the OrderedList interface. (If you find it hard to directly code the generic implementations of your algorithms you may also copy the template into a ProductsList class implementing OrderedList<Product> and replace all E's by Product. Similarly, you may duplicate into a PurchasesList class replacing all E's by Purchase).
   (Correct implementations of the overridden remove methods are optional)
3. Implement two versions of the binary search: an iterative version and a recursive version
4. Complete the implementation of the default method aggregate in the OrderedList<E> interface.
5. Implement the missing parts of the PurchaseTracker
6. You may add more private and public methods to any of the classes and interfaces, but not change any signature of the specified public methods.
7. Run all provided unit tests and add unit tests for relevant code or situations that are not covered by the provided unit tests.
8. Run the main program and compare the results with below output.
9. Prepare your document with explanations of seven code snippets.
   Explain how the OrderedArrayList representation invariant is sustained by your overridden implementation of add(index,item) or one of the remove methods. (Choose the most relevant.)
   Explain correctness of one code snippet by involving a loop invariant.

Sample output of SupermarketStatisticsMain

```
Welcome to the HvA Supermarket Statistics processor

Imported 61 products from /products.txt.
Accumulated purchases of 61 products from files in /purchases.
5 purchases with worst sales volume:
1: 3214563456672/Robijn klein en krachtig kleur/30/243.30
2: 3214563456673/Robijn klein en krachtig donker/41/332.51
3: 770835504/Milner kaas stuk 1Kg/55/380.60
4: 3214563456671/Robijn klein en krachtig wit/73/592.03
5: 8712100516381/Calvé Pindakaas met nootjes 650g/81/344.25
5 purchases with best sales revenue:
1: 5000112544631/Coca Cola regular 1L/1189/1212.78
2: 8718907136068/Pure Hagelslag XL 390g/409/1165.65
3: 770835604/Milner kaas plakken 200g/348/1155.36
4: 770835505/Boeren kaas stuk 1Kg/129/1150.68
5: 4266423453203/Steenoven Pizza Hawaii/425/1083.75
Total volume of all purchases: 16730
Total revenue from all purchases: 38120,38
```

Grading.

At DLO you find the rubrics for the grading of this assignment. There are three grading categories: Solution (50%), Report (30%) and Code Quality (20%). Your Solution grade must be sufficient, before the grading of Report and Code Quality is taken into account. Similarly, your Report grade must be sufficient before the code quality grade is granted.

For a sufficient score, you may choose to use two separate (duplicated) implementations of ProductsList and PurchasesList in stead of a true generic implementation of OrderedArrayList<E> and you may fail the unit tests that verify the representation invariants of remove methods.