Department of Computer Science and Engineering
University of Puerto Rico
Mayagüez Campus

CIIC 4020 / ICOM 4035 – Data Structures
Spring 2019-2020
Project #2 – Frequency Distribution

Introduction

Frequency distribution consists of counting how many occurrences of each different object exist in a data set. There are many uses of frequency distributions; here are just a few examples:

- Finding the questions in an exam that were the most difficult for students.
- Creating bar charts or histograms to better visualize data.
- Making spelling suggestions in a word processor by combining edit distance with a list of most frequently-used words in a language.

In this programming project, you will implement *four* approaches to determine the frequency distribution of objects in a list. Additionally, you'll include a component to experiment with those four alternatives to measure the efficiency of each strategy in terms of execution time.

Detailed Specifications

The four strategies you will implement should work for any data set with objects of type `Comparable` and whose data type has overridden the **`equals`** method. Each one of the strategies to be implemented gets as input a data set (in the form of an `ArrayList`) and generates a final list of *entries* (like those used in a `Map`), which we will refer to as `results`, where each *key* will be a distinct object of the data set, and its corresponding *value* will be the frequency of that object in the data set. No entry in `results` should refer to an object that is not part of the data set.

Also, an experimentation component will be included (explained further below) with the goal to experiment with those four strategies and estimate their average execution times.

Four Strategies for Frequency Distribution

The four strategies are as follows:

1. **Sequential strategy** – Objects from the data set are explored one by one. For each object in the data set, *sequentially* (i.e. with a for-loop) verify if there is already an entry in `results` (initially empty) whose key is that same object (if both are equal). If so, we've previously found this object, so the value in that entry (the frequency of the object that is its key) must be increased by 1. If not, then a new entry is added to that list of entries with key equal to the object being considered and value set to 1. At the end, the list of entries (`results`) will contain one entry for each different object in the data set. Each entry will contain a particular object (the key) and its frequency (the value) in the data set.
2. **SortedList strategy** – This strategy is very similar to the previous one, but instead of directly storing the entries in `results`, you will first store them in a `SortedList`. In the sequential approach you need to search through all of `results` to determine that a new entry needs to be created. In this case, we can stop as soon as we find an object that is bigger (why?), without needing to reach the end of the `SortedList`. Once you've finished counting, copy the frequencies to `results`.
3. **Ordered strategy** – Sort the data set (it's an `ArrayList`, so you may use the **`sort`** method in the `ArrayList` class). Examine all objects in that sorted list, *taking advantage of the fact that they are in order*, to determine the frequency of each different object. For each different object in the data set, add an entry to the `results` list of entries to be returned (where an object is the key and its frequency is the value). **Note**: The strategy receives a *copy* of the original data set, so you are only sorting a copy of the data set; the original one remains intact.
4. **Map strategy** – Here you will use Java's `Hashtable` class, which is an implementation of the `Map` ADT (see documentation for `Hashtable` in Java documentation library). An object of that type, initially empty, will be used in this approach. The strategy processes each object from the data set as follows. The object itself is used as a key. If not found in the map, then a new entry is added to the map, where the object itself is used as the key of that entry and the value associated will be 1. If the object is found in the map (as the key of some entry… see **`containsKey`** and/or **`get`** method), then the associated entry is modified by increasing its current value by 1. At the end, when all of the objects in the original data set have been processed, the entries in the map are placed into the final list of entries (`results`). At that moment, there will be one entry for each different object in the original data set; and each such entry will contain, as its value, the frequency of that object, its key, in the data set.

The above mentioned entries are objects that your program manages. For that, you should use `Map.Entry` in Java, which is a nested interface of the `Map` interface (see provided code for examples). We will discuss this interface in more detail further below.

Experimentation Approach

The general framework of the experiment is discussed next. The goal is to estimate the time each strategy takes for different data set sizes. Data sets for the experiments consist of randomly generated integers. For each size n, the experiment is conducted several times: generate a new data set of that size consisting of randomly generated integers in the range 1..n/2, apply each strategy, measure the time each strategy take, compute the final average for each strategy, set that average as the time each strategy takes to solve the problem for a data set of that size. At the end, the system should produce, for each strategy, a table relating size and time from the results produced by the system.

The general procedure for the experimentation part is as follows:

1. For each value of *n* = 50, 100, 150, …, 1000
    A. Repeat the following 200 times:
        i. Generate a new list of *n* integers and store them in a list of type `ArrayList<Integer>`. That list cannot be altered by any of the strategies to count frequencies.
        ii. For each strategy do the following:
            a. Apply the described strategy to determine the frequency distribution of the different values in the generated list
            b. Measure the time it takes to accomplish the task
            c. Add the measured time to a total sum of times obtained for the current strategy, which is initially set to 0
    B. For each strategy do the following:
        i. Determine the average time (*t*) that the strategy took for the current value of *n* (divide the total sum of times by 200)
        ii. Store the values *n* and *t*, in the list of result entries corresponding to the particular strategy
2. Save the list of results for each strategy in a separate file.

Once those files are generated you must use a tool such as MS Excel, LibreOffice Calc or Google Sheets to analyze the results and compare the behavior of the four strategies tested by creating a graph (similar to what was done in the Sorting lab).

The above suggested values for *n*, its increment, and the number of trials for the same size, should be parameters that can be easily changed. This will allow you to easily use other values if needed. The ones given for the sizes to consider might be too small to achieve notable different times depending on how fast your computer is. If that is the case, you want to be able to easily make changes so that you may use larger sizes.

General Implementation Requirements and Ideas

In this section, we describe in more detail some requirements that you should comply with in your implementation. On the one hand, they will allow us to easily test your project, while on the other, it provides a schematic approach that is useful to manage the implementation process.

As part of the project, you should comply with the following:

1. Your project must include one Java package named testerClasses. In that package, for each strategy, there will be one tester class named *x*Tester, where *x* is the name of the strategy (Sequential, SortedList, Ordered, or Map). Each one of those tester classes include a main method, allowing the particular strategy to be executed from that class in the JVM. That class tests the particular strategy twice: first with data from a file of integer values named integerData.txt, and then with data from a file of strings named stringData.txt. These input files should exist in a local directory (inside your project's directory at the level of the src package if executed from Eclipse; or, if executed from the command prompt, at the level of the directory from where the command is issued)named inputData. You should carefully read about the File class in Java doc library to understand how file paths are managed independently of whether the system where the program is run is Unix, Windows, etc. Output for these testers should be displayed directly on the terminal window in the computer (or output window in Eclipse).
2. The project must include another package named experimentClasses. There, you will include a Java class named ExperimentalTrials, whose main method initiates the experimental process and finally generate the files with the experimental results for each one of the strategies discussed. Such files will be saved to a local directory to be named experimentalResults (at the same level of the inputData directory). The names of those files shall be:
   - `resultsSequential.txt`
   - `resultsSortedList.txt`
   - `resultsOrdered.txt`
   - `resultsMap.txt`

   Each one of these files will contain, for the corresponding strategy, and for each value *n*=50, 100, 150, …, 1000, one line containing the two numbers *n* and *t*, separated by at least one space character. For each such pair, *n* is the data set size and *t* is the particular average execution time (in milliseconds) that was determined by the experimentation process for that particular data set size.

Implementation Ideas

First, let's mention some Java classes and interfaces that you should be using in this project:

- **Interface Map.Entry<K, V>:** The idea of an object of this type is that it consists of a pair of values, one of generic type K and the other of generic type V. Any instance of such type pair establishes an association or relation between the two objects that are its components: the key and the value. Usually, the first value, usually referred to as the "key" of the entry (the one of type K), is a value that somehow differentiates the pair from other such pairs. The other value (the one of type V) is simply considered the "value" of the entry pair. Java provides an implementation for this: see class AbstractMap.SimpleEntry. You can use objects of this type to represent entries used in the different strategies being implemented.
- **Class Hashtable<K, V>:** This class implements a hash table, which maps keys (of type K) to values (of type V). You should see the Java documentation for details on all of the operations available, and you may use any that you see fit, but for this project you will probably only need the following: default constructor, **`entrySet`**, **`get`**, and **`put`**. Read the specifications of what they are supposed to do and the correct way to use them.

The following ADT (specified as an abstract class) corresponds to the datatype of a frequency counter object. For each of the described strategies there should be one subclass implementing the appropriate algorithm that such strategy uses to count the different objects in a data set given as an ArrayList<E>.

```
public abstract class FrequencyCounter<E extends Comparable<E>> {
    private String name;    // the name given to this strategy

    public FrequencyCounter(String name) {
        this.name = name;
    }

    /**
        Accesses the name of the particular strategy that the object
        instance corresponds to.
    **/
    public String getName() {
        return name;
    }

    /**
        Determines the frequency distribution of objects in a particular
        data set using a valid strategy. It's based on the algorithm defining
        the particular strategy to solve the frequency counting problem.
        @param dataSet the dataset of objects to be analized
        @return a list of entries, where each such entry is a pair:
                  key (of type E) is a reference to one instance of a
                  different object, and the value is the frequency of that
                  object in the list being analyzed
    **/
    public abstract ArrayList<Map.Entry<E, Integer>>
           computeFDList(ArrayList<E> dataSet);
}
```

Under this approach, you can then implement one class for each one of the strategies, with names: SequentialFD, SortedListFD, OrderedFD, and MapFD. For example, the following is the one (documentation is omitted for simplicity) implementing the *Sequential* strategy previously described.

```
public class SequentialFD<E extends Comparable<E>> extends FrequencyCounter<E>
{
   public SequentialFD() {
      super("Sequential");
   }
   @Override
   public ArrayList<Map.Entry<E, Integer>> computeFDList(ArrayList<E> dataSet) {
      ArrayList<Map.Entry<E, Integer>> results =
                        new ArrayList<Map.Entry<E, Integer>>();
      for (E e : dataSet) {
         boolean entryFound = false;
         for (int i=0; i<results.size() && !entryFound; i++) {
            Map.Entry<E, Integer> entry = results.get(i);

            if (entry.getKey().equals(e)) {
               entry.setValue(entry.getValue() + 1);
               entryFound = true;
            }
         }
         if (!entryFound) {
            //need to create a new entry for first instance found of object e
            Map.Entry<E,Integer> entry = new AbstractMap.SimpleEntry<E, Integer>(e,1);
            results.add(entry);
         }
      }
      return results;
   }  // end of method computeFDList
}
```

Other implementation ideas are:

1. To generate random numbers, we'll use the class Random in java.util.
2. To estimate execution time of code X, we will do the following:

```
long startTime = System.currentTimeMillis();
X
long estimatedTime = System.currentTimeMillis() - startTime;
```

Documentation & Comments

You must properly document your code using the Javadoc format. Include at the top of the class some comments explaining the nature and structure of your class, along with the corresponding comments and documentation for your methods. Different people will have different implementations, so it's important that your code is well documented and commented so that we may understand your intention. This is an important part of your information as a future professional in the computer science/engineering industry. You will lose points if you don't provide Javadoc-style comments and additional comments explaining your logic, or if we deem the amount and/or frequency of comments to be inadequate.

Submission

The final date to submit your program will be Saturd**ay, May 2, 2020** at 11:59pm. The name of your zip file must follow one of the two following formats: `P2_4020_nnnnnnnn_192.zip` (for those in CIIC4020) or `P2_4035_nnnnnnnn_192.zip` (for those in ICOM4035), where nnnnnnnn should be replaced with your student id number.

Academic Integrity

Do NOT share your code! You may discuss design/implementation strategies, but *if we find projects that are too similar for it to be a coincidence, all parties involved will receive a grade of 0*. Don't cheat yourself out of a learning experience; seek our help if you need it.

Final comments

The specifications of a project are the first, and arguably the most important, part of a software development project. Therefore, it's crucial that you read these specifications thoroughly so that you understand what is being asked of you. These are skills that you will need to succeed in your professional career, so it's imperative that you start applying and improving them now. If your program runs successfully, but does not adhere to the specifications, it is of no use. *Before you submit your project, review these specifications one last time and make sure you meet all of the requirements that have been imposed.*

If your code does not compile properly, your grade will be 0, NO EXCEPTIONS!