Lab 3. Analysis of Algorithms

Theme. In this lab, you will:

- use some common collection classes in the Java Collections Framework (JCF)
- observe the time taken to process various amounts of data using different JCF classes
- practise using Big O notation to describe the time efficiency of an algorithm

Key concepts: generic types, collections, Big-O notation

Required file(s): `lab3.zip`

1 Getting Started...

N.B.: Whenever you start up `eclipse`, make sure that you are in the appropriate workspace. You may check this using the drop-down menu option:

`File` ➡ `Switch Workspace`

This will pop up a dialogue window named `Workspace Launcher`. If the name of the workspace displayed in the text field is not your expected one, switch to the appropriate workspace by specifying the full location of your workspace folder.

1. In a web browser, access the archive `lab3.zip` from CS2310 on Blackboard. Extract its contents into your `eclipse` workspace for this module.

2. Start up `eclipse`.

3. Making use of the contents of your extracted archive, create a new Java project named **algorithm_analysis** in your `eclipse` workspace.

2 Algorithm Analysis

Project `algorithm_analysis` has two packages:

1. `dictionary`
2. `dictionary.exception`

Package `dictionary` contains the interface `MagicalBag`[1] and six realisations of that interface:

1. `MagicalBag1` - implemented using an array
2. `MagicalBag2` - implemented using a `java.util.ArrayList`
3. `MagicalBag3` - implemented using a `java.util.LinkedList`
4. `MagicalBag4` - implemented using a `java.util.HashSet`
5. `MagicalBag5` - implemented using a `java.util.LinkedHashSet`
6. `MagicalBag6` - implemented using a `java.util.TreeSet`

Each concrete implementation of the interface `MagicalBag` is expected to behave like an *unordered set*[2]. A user can add an element to the set. The user can also remove or retrieve (without removing) a random element from the set.

Class `WordPicker` in the `dictionary` package models a word selector. It can work as a standalone Java application. Given the name of a dictionary file and the type of bag to be used, a `WordPicker` performs three main operations (as defined in the `static` method `simulation`):

1. Create a specified type of `MagicalBag` object for keeping the dictionary data.
2. Pick 20 words at random and display them.
3. Remove twenty words from the `MagicalBag` and display them.

The time taken for executing each operation is recorded in a file for later display.

Classes `Tester5` and `Tester7` in the `dictionary` package provide a convenient way to compare the performance of the six different implementations of `MagicalBag`. Each class contains one static `main` method. You can run each class as a **Java application** without any program argument. It simply executes the main operations of a `WordPicker` in turn using different implementations of `MagicalBag` as well as various dictionary files that may be found in the folder `dictionary_files`. The results of each run are written to the file `output.txt`. `Tester5` and `Tester7` use IO features from Java 1.5 and Java 7, respectively.

Package `dictionary.exception` contains three `Exception` classes:

- `UnsupportedBagTypeException` — thrown when a specified type of bag is not supported by `WordPicker`;
- `FullBagException` — thrown when attempting to add an item to a full bag;
- `EmptyBagException` — thrown when attempting to pick or remove an item from an empty bag.

[1] A `MagicalBag` is "magical" because *duplicate* items in a magical bag will disappear in thin air! Hence, all items in a magical bag are always unique. `MagicalBag` is something of a misnomer here; `Set` would be better. However `Set` clashes with `java.util.Set` which is imported into several 'MagicalBag' classes.

[2] Elements in a set must be unique, i.e. no duplicate is allowed. The order of elements in the set is irrelevant and is hidden from the user.

Some of the required Java code is missing from the above interface and classes. Your task is to complete the missing implementation and to generate test results by running class `Tester`. You are also expected to inspect the methods `add`, `pick` and `remove` in each of the `MagicalBag` classes to see how they have been implemented so as to analyse their performance.

> **Hint**: The approximate locations where you are expected to add your Java code and relevant hints for accomplishing the tasks have been marked throughout the given Java programs. Look out for **block comments** that include a sequence of *four* exclamation marks:
>
> ```
> /* !!!!  ...    ...  */
> ```
>
> The location of code fragments that you are expected to inspect to have also been marked. Look out for **block comments** that include a sequence of *four* plus signs:
>
> ```
> /* ++++ ...    ...  */
> ```

Your Tasks

1. A `MagicalBag` object is expected to be used as a *generic* collection. It should also be possible to use an *enhanced for loop* to process the elements of a `MagicalBag`. Add the missing code to the *header* of interface `MagicalBag`.

2. Add the missing code to `MagicalBag` so as to remove the syntax errors.

3. Complete the implementation for the constructor of class `MagicalBag3`.

> **Hint**: `MagicalBag3` uses a `LinkedList` to store its contents.

4. Complete the implementation for removing a random element from a `MagicalBag3` object.

5. Complete the implementation for the constructor of class `MagicalBag4`.

> **Hint**: `MagicalBag4` uses a `HashSet` to model its contents. The efficiency of an `HashSet` object improves when it 'knows' the maximum amount data that it will need to hold during execution of the application. This will enable the `HashSet` object to be better-prepared for its task during its creation.

6. Complete the implementation for removing a random element from a `MagicalBag4` object.

> **Hint**: The required operation should be fairly similar to that for method `pick` in class `MagicalBag4`, except that in `pick`, the method simply returns the item found, whereas in `remove`, the method needs to also remove the found item from the bag.

7. Run `Tester`.

> **Hint**: You need to be patient. This operation will take some time...

8. While `Tester` is running, do the following tasks.

(a) In a web browser, display Java™ 2 Platform Standard Edition 8 API Specification, i.e.:

`https://docs.oracle.com/javase/8/docs/api/`

Briefly note the main difference in implementation between the following collections:

- array,
- `java.util.ArrayList`,
- `java.util.LinkedList`,
- `java.util.HashSet`,
- `java.util.LinkedHashSet`, and
- `java.util.TreeSet`.

(b) Go through the given classes and look for the block comments with

`/* ++++ ... ... */`

Note how methods `add`, `pick` and `remove` in each of classes `MagicalBag1`, `MagicalBag2`, `MagicalBag3`, `MagicalBag4`, `MagicalBag5`, `MagicalBag6` have been implemented.

9. When your `Tester` application has finished its execution, prepare three graphs[3]:

- The x-axis corresponds to the size of the given dictionaries, i.e. 10,000, 20,000, 50,000 and 75,000.
- The y-axis corresponds to the time taken to:

  (a) build each dictionary,

  (b) pick ten words from a bag, and

  (c) remove ten words from a bag.
- For each type of magical bags (modelled by `MagicalBag1`, `MagicalBag2`, `MagicalBag3`, `MagicalBag4`, `MagicalBag5`, `MagicalBag6`), plot the results as shown in the file `output.txt` as a line graph.

> **Hint**: The *x-axis* of your graph should show size of different dictionaries; whereas the *y-axis* should be the time taken for each type of bag to process one type of operation, e.g. the time taken to build the dictionary.

Based on the results shown in these graphs, answer the following questions:

(a) Which implementation of `MagicalBag` is the most time-efficient (for this particular application)?

(b) State the asymptotic time complexity of the three key operations in each implementation of `MagicalBag` using the *Big-O notation*.

[3] by hand or use a spreadsheet application such as Microsoft `Excel` or `Libreoffice`