Abstract Data Structures

HashSets

Scenario

We will use the same scenario as the Hash Table unit. This time we will implement it in Java with a HashSet.

A school is running a coding competition. A system to register student competitors is required. It is important that the students register only once and fast access, preferably O(1) is required when inserting, removing or searching for a student.

Decomposition

The problem was decomposed into 3 different subsystems:

- StudentManager class
  - initialise 2 sets, Computer Science students and Competition students
  - Register students to the competition set
  - Remove students from the competition set
  - Make the sets available to external classes
- Registration GUI class
  - Create a gui allowing competition students to be added/removed
  - Handle Add/Remove button clicks
- Driver class
  - Construct the Registration Gui

This System Overview guided the development of the GUI prototype.

Watch the Helper Video on the unit page for further information on how the program was developed.

Task 1

In your IDE, create a new project and set up the `Main` class, `StudentManager` class and `RegistrationFrame` class.

Run the project from `main()`. Test the app by adding the following student ids:

stu1111	stu0000	stu2000	stu1111

The last ID should be rejected as it **already exists** in the set. No duplicates are allowed.

Task 2

The `StudentManager` class processes two `HashSet` objects. Why are these not `Set` objects? Try changing this line of code:

```java
computerScienceStudents = new Set<>();
```

What error do you get? Do some research and figure out the nature of this error. Make notes on it here:

Task 3

Under the hood, when we initialise a HashSet, 16 *buckets* are created. We might expect the size() of the set to be 16. Is this correct?

In the `registerCompetitionStudent` method of the `StudentManager` class, add the following line of code after we `add` an id to the `competitionStudents` set:

```java
System.out.println(competitionStudents.size());
```

Run the program and add some student ids, including some duplicates. We can conclude that the **size()** of a HashSet does not refer to the number of buckets. What does it refer to? Also, what can we conclude about the **add()** method of the HashSet class? What HashSet behaviour does the **add()** method guarantee?

Task 4

A HashSet monitors the **load factor**:

Load Factor = Number of Entries/Capacity

When the **load factor** exceeds the **resize load factor** (0.75 by default), the hash table is considered too full and *performance may begin to degrade due to increased collisions*. To maintain efficient searching and insertion, the HashSet creates a larger hash table and performs **rehashing**. During rehashing, every existing element is processed again, new bucket indices are calculated based on the larger table size, and the elements are redistributed into new buckets in the resized table.

Research if it is possible to determine the number of buckets that a Hash Set's table has in Java. Is it possible to find out the resize load factor? Make any notes here.

Task 5

The HashSet class's **contains()** method is very similar to the ArrayList class's contains method.

```java
if(!computerScienceStudents.contains(id){
	//do something
}
```

It simply returns true or false if the parameter is found in the set. No doubt, it does not perform a linear search. It uses the hash function to determine the bucket in the hash table to see if the bucket contains the search item.

Task 5

The `remove()` method of the `HashSet` class is also very interesting. Unlike an `ArrayList`, it does not need to perform a linear search through every element. Instead, it uses the item's hash code to calculate the bucket index where the item should be located. It then locates that bucket, applying the collision resolution strategy used by the hash table implementation if necessary, and removes the item if it is found. The method returns true if the item was successfully removed and false if the item was not present in the set.

SuperSets and Subsets

In this scenario, we can satisfy ourselves that the `competitionStudents` set is always a subset of the `computerScienceStudents` set. However, in the real world of computing, where **admin** staff may have privileges beyond normal users, mistakes may be made.

For example, there could be a scenario where a student ID is entered into the competition set that is not in the computer science set.

The `HashSet` class conveniently allows us to check if a set is a subset of another using a method called `containsAll`.

Let's simulate this scenario.

In the `StudentManager` class, create a new property `admin`:

```
Java
private boolean admin;
```

Then, in the `StudentManager` constructor method set `admin` to `true` or `false` (you choose).

```
Java
admin = true;
```

Now let's update the logic of the `registerCompetitionStudent` method. The logic is:

```
if the id is not a computer science id and admin is false
      return "Registration invalid"
else
      add the id to the set //because either id is valid or admin is true
      return "Registration valid"
```

```java
if(!computerScienceStudents.contains(id) && !admin){
	return "Registration invalid!";
}else{
	competitionStudents.add(id);
	return "Registration valid";
}
```

Finally, every time we click the Register button, the last thing we will do is check if the sets are still valid and output a warning message if they are not.

"The sets are valid if `competitionStudents` is a subset of `computerScienceStudents`"

In the `RegistrationFrame` class, design a function called `checkSets`:

```java
public void checkSets(){
	Set<String> compsci = manager.getComputerScienceStudents();
	Set<String> competition = manager.getCompetitionStudents();

	if(compsci.containsAll(competition)){
		System.out.println("The sets are valid!");
	}else{
		System.out.println("Invalid sets!");
	}
}
```

And call the function somewhere near the bottom of the ActionPerformed method so that we can check the sets after any button click, eg after the refreshLists call:

```java
refreshLists();
checkSets();
```

So everytime we click either button, we will also be checking the validity of the superset/subset relationship of the 2 sets, which the containsAll method of the HashSet class allows us to do. Play with the setting of the admin variable and try to register and remove valid and invalid ids. Look out for the warning messages in the console.

Review

A `HashSet` is an Abstract Data Type because it defines what operations can be performed on a set of elements (such as add, remove, and contains), but does not specify how those operations are implemented internally[1].

Its `add` method conveniently will not allow duplicate entries into the underlying Hash Table.

The `containsAll` method will help to determine if one set is a subset of another.

**interesting note*

We cannot use a regular for loop to iterate through a Hash Table because it does not use classic indexes. For example, we cannot use `get(i)` like we can with ArrayLists.

However if you look in the `RegistrationFrame` class, you can see we can still use an enhanced for loop to visit each bucket in the table and retrieve its contents eg:

```java
for(String id : manager.getCompetitionStudents()){
```

[1] Think about it - do you have any idea how the HashSet class actually adds, removes or inserts data into a Hash Table? We understand the underlying theory of Hashing Functions and Collision Resolution Strategies, but the actual implementation is unknown to (most of) us.

```
        System.out.println(id);
}
```