Bags of integers

Pierre Cagne

Abstract

This document is a companion for the class given for CS2440 at ASU on February 19, 2024 at 10AM. We explain how to construct an implementation of bags of integers, different from the one in the textbook, but without the users of the class to even notice.

1 Abstract Data Types

Although we will rely extensively on the example of *bags of integers* that you have already seen, today's class is really about the concept of *Abstract Data Types* (ADT for short). An ADT is a data type, much as every other types you have seen in Java (the type of integers, the type of strings, the type of booleans, etc.), but designed in a way that the users never have to know the nitty-gritty details of its implementation. Of course, like everything else in Java, ADTs are implemented by classes.

As an analogy, think of an ADT as a car: the implementation details of the ADT is the engine of a car, and the user of the ADT class is the driver. The driver need not know what is happening exactly inside the engine to be able to drive the car. They only need to know how to use the controls of the car: the steering wheel, the pedals, the gear shift, the dashboard, etc. Even if the driver is a mechanic and knows about the internal design of the engine, they cannot take advantage of this knowledge while driving. All that is accessible to the driver is the controls of the car, these are their only means of interaction with the engine while driving.

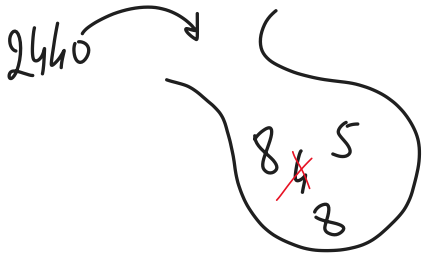
In a nuttshell, an object whose type is an ADT can be considered a "black box" with which the user can only interact through a carefully defined interface. This approach has two main benefits:

- the user cannot exploit the internal mechanics of the ADT to abuse their functionality (potentially leading to runtime errors);

- as long as the interface remains the same, the developer of the ADT can maintain and change the details of the implementation without needing the user to adapt their code to the changes.

2 Let's play a game

Recall that bags of integers model the naive notion of such bags: a (finite) collection of integers put together, without order, in which we can add elements, remove elements, search for a specific elements, etc. In particular there is no restriction on having multiple times the same number. For that reason, we call them sometimes *multisets of integers*. Note also that if a bag contains a finite number of element at any time, it is always possible to add new elements: a bag is never "full".



By playing the game I played in class, where you were in the shoes of the user and queried me about the (duffle) bag of integers I had, we determined that we want to offer these following controls to the user:

(i) Create a (empty) bag.

(ii) Give the size (total number of elements) of the bag.

(iii) Add an integer to the bag.

(iv) Remove an integer from the bag.

(v) Give the number of occurrences of a given element in the bag.

(vi) Print the content of the bag.

3 Implementation – interface design

In other words, we have the skeleton of our class! It suffices to implement a method for each of these control. For now, as we didn't decide on the actual implementation of bags yet, we can only produce the *signature* of each method. To do so, for each of the method, we reason as follows:

- As the user, what piece of information do I need to provide for this particular control? This gives the argument(s) of the method.
- As the user, what piece of information do I get back after the action of said control is done? This gives the return type of the method.

For example, the method to add an element needs, from the user, the information about the integer to be added: so the add method as an argument, of type int. However, the add method performs an action (adding the given integer to the bag), and produces no value for the user as a result. So the return type of the add method is void.

Now that we have the skeleton of our class, we can even write the code of our testing program. All of that before even thinking about how to implement bags per se!

4 Implementation – engine design

Now I can finally reveal what I was doing inside the duffle bag during our little game to organize internally. You might think you know because you already have seen an implementation of bags of integers, but I decided to trick you and do something completely different!

Inside my duffle bag, I had two arrays of the same length on top of each others:

data	8	4	5	10	6	42	55	⋯
occs	2	1	1	0	0	0	0	⋯

The cells in the top array represents elements of the bag only when the corresponding cell on the bottom is not 0, then this bottom cell is the number of occurrences of the top cell in the bag. Here in my example, the bag contains two 8s, a 4 and a 5, and that's it. A cell of the top array whose corresponding cell in the bottom is a 0 is considered junk, and its content is then completely irrelevant (and can be any integer).

To add an integer, say 2440, you proceed as follows:

- range over the top cells whose corresponding bottom cell is not 0,
- if you find 2440, just add one to the corresponding bottom cell,
- if you don't find it, find the first cell with 0 in the bottom array and put 2440 in the corresponding cell in the top array. And don't forget to update the bottom cell to 1!

data	8	4	5	2440	6	42	55	⋯
occs	2	1	1	1	0	0	0	⋯

But wait, what happen is there if we didn't find the element in the first pass and there is no more bottom cell with 0? In that case, we have to resize the array, in the pretty much the same way as you did for the IntArrayBag implementation.

To remove an element, say 4, it is even easier:

- range over the top cells until you find the element to be removed,
- when you find it, if the corresponding bottom cell is not 0, decrement it and return true,
- if you reach the end of the loop, it means the element wasn't in the bag to begin with and return false.

data	8	4	5	2440	6	42	55	⋯
occs	2	0	1	1	0	0	0	⋯

So if we want to add an new element again, say 42, we end up with:

data	8	42	5	2440	6	42	55	⋯
occs	2	1	1	1	0	0	0	⋯

To count occurrences of an element, say 5, simply range over the top cells whose corresponding cells is not 0, and return the value of the bottom cell when you find the element (and return 0 is the element is not found). Here, we find 5 in the top array with corresponding bottom cell containing 1, which is the number of 5s in the bag.

To return the size of the bag, it suffices to compute the sum of all the elements in the bottom array.

To print the content of the bag, the easiest is simply to range over the top cells and to print on the screen the number of the top cell as many times as the bottom cell indicates.

Now all it remains is to put that into code! You can find the actual implementation (and these notes) at: https://github.com/pierrecagne/CS2440-2024.

5 Conclusion

Between last session and today's, you have seen two implementation of *bags of integers*. So what is the correct one? Well they are both correct! As long as the user is given the interface we agreed upon, they are able to use either of the implementation in the exact same way. Both implementation are perfectly valid renderings of the same *abstraction*.

If you have to remember one thing from today, this is the following: when designing an ADT, first put on your users hat (be the driver) and design the interface before even thinking of the implementation details of the ADT; when you are satisfied with the interface, and only then, put on your developer's hat (be the engine mechanic) and implement the details of the ADT class.