CS1020 Sit-in Lab 02 A - Automated Storage Facility

Semester 2 AY2013/2014

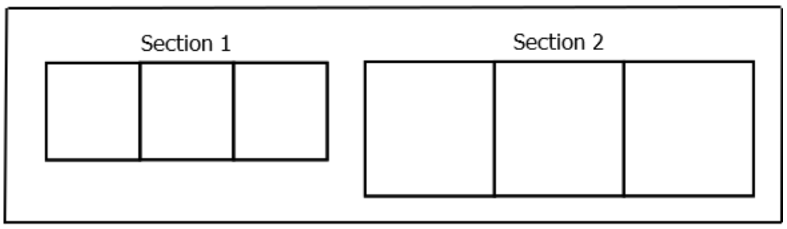
Background

A company wants to build an automated storage facility which can run without any human intervention. The facility is partitioned into M sections, where each section is basically a shelf-like contraption with movable parts partitioned into N columns. Items are stored in the columns of the contraption. The innovative part is the ability of the contraption to automatically rearrange the items stored in it. Before actually building the storage facility, the company wants to run a simulation of the items being stored and retrieved from the facility. You have been hired to write the simulation software.

Use Java ArrayList to help you in this task. **Not using ArrayList will result in marks being halved. Using any other type of arrays instead of ArrayList will also result in marks being halved!**

Problem Description

As mentioned, each of the M sections is partitioned into N columns. All the columns in a given section are of the same size (expressed as a positive integer) and can contain items $\leq$ that size. Different sections have columns of different sizes. An example of a facility with 2 sections each comprising of 3 columns is shown below:



An item that is stored in the facility has 2 attributes. First is a unique positive integer identifier. Second is the size of the item expressed as a positive integer.

When an item comes in to the facility, it will be stored according to the following rules:

1. Find the section with the smallest column size the item can fit into and has an available column to fit the item in.
2. If such a section is found, move all items to the right by one column, and place the item in the leftmost column.
3. If no such section is found, return the item.

An example of an item being successfully stored is given below.

```
There are 3 sections of 3 columns each in the facility.
The column sizes of the sections are 20,30,40 respectively.
The current storage configuration is as follows.

$ - empty column
number - identifier of item in column

section 1 (column size 20)
1 $ $

section 2 (column size 30)
12 16 21

section 3 (column size 40)
23 90 $

An item with identifier 99 of size 25 is to be stored.
Following the rules of storage it is stored as shown below.

section 1 (column size 20) <-- cannot be used as column size < item size
1 $ $

section 2 (column size 30) <-- cannot be used as no empty column
12 16 21

section 3 (column size 40)
99 23 90 <-- stored in leftmost column of this section.
```

In order for an item to be retrieved, its identifier must be provided. If the given item identifier is not found among any of the items stored in the facility, the retrieval request is rejected. Otherwise, the item is retrieved according to the following rules:

1. Locate item and remove it from its section.
2. All items to the right of the removed item are automatically shifted left by one column so that no gaps are present.

An example of an item being successfully retrieved is given below.

```
Item with identifier 17 is to be retrieved.

Current storage configuration

section 1
1  2  $

section 2
13 17 35

Storage configuration after item is retrieved

section 1
1  2  $

section 2
13 35 $  <- item found and retrieved. All items right of 17 are moved 1 column left.
```

Input

The 1st line in the input is N (N $\geq$ 1) which gives the number of columns for each section. *Use descriptive variable name instead of N in your program. This applies for the other descriptors here.* The 2nd line is M (M $\geq$ 1) the number of sections. The next M lines are the sizes of the columns in each section from **smallest to largest**. The next M lines after that represent an item put into the leftmost column of each section and is in the format below:

<item identifier> <item size>

You can assume the items used to "initialize" each section will always fit into that section. The lines following that until the end of the input represents the two possible operations (store, retrieve) to be executed. Their formats are as follows:

- Store operation: S <item identifier> <item size>
- Retrieve operation: R <item identifier>

An example input is given below:

```
3       <-- number of columns in each section
2       <-- number of sections
10      <-- size of section 1
20      <-- size of section 2
3 5     <-- store item with identifier = 3, size = 5 in leftmost column of section 1
4 15    <-- store item with identifier = 4, size = 15 in leftmost column of section 2
S 12 3  <-- store operation, item identifier = 12, item size = 3
S 20 7
S 1 2
R 20    <-- retrieve operation, item identifier = 20
R 2     <-- retrieve operation which will fail as there is no item 2
S 30 15
```

Output

Print out configuration of each section from section with smallest column size to section with largest column size. For each section, print on a separate line the item identifier of the items stored from left to right of the section (separated by a comma, WITHOUT any space). If there are no items in the section, print a single @. Example output for the given input above is shown below:

```
12,3
30,1,4
```

Skeleton Program

Your program is to be named **AutomatedStorage.java** (do not change this name). A skeleton program is provided, but you can change it any way or add new class(es) and method(s) as you deem fit. **Write all your code in AutomatedStorage.java, do not create new files!** The skeleton program is as follows:

```
import java.util.*;

/* Service class representing an item */
class Item {
        private int id;
        private int size;

        public Item(int iId, int iSize) {id = iId; size = iSize;}
        public int getId() {return id;}
        public int getSize() {return size;}

        // Fill in the code
}

/* Service class representing a section in the automatic storage facility */
class Section {
        private ArrayList<Item> shelf;
        private int colSize;
        static private int nCol;

        // Fill in the code
}

/* Client class to simulate an automatic storage facility */
public class AutomatedStorage {
        // Fill in the code

        public static void main(String[] args) {
                Scanner sc = new Scanner(System.in);
        }
}
```

Testing your program

The following **sample** input and output files are in your plab account:

```
AutomatedStorageTC1.in, AutomatedStorageTC2.in, AutomatedStorageTC3.in
AutomatedStorageTC1.out, AutomatedStorageTC2.out, AutomatedStorageTC3.out
```

AutomatedStorageTC1.in, AutomatedStorageTC2.in and AutomatedStorageTC3.in are the input test cases, while AutomatedStorageTC1.out, AutomatedStorageTC2.out and AutomatedStorageTC3.out are the expected output for the respective test cases.

After you have compiled your program, to test it with say AutomatedStorageTC1.in, you type:

```
java AutomatedStorage < AutomatedStorageTC1.in
```

Grading Scheme

1. Program correctness = 70 marks, Design = 20 marks, Programming = 10 marks
2. No marks awarded if the program does not compile.
3. Marks will be deducted if **student particulars and program description** are not filled up in the top portion of the source code.
4. There are 10 test cases. Each test case is worth 7 marks.

5. Marks will be halved if ArrayList is not used or any other types of arrays is used.
6. Things to look out for under design
   (a) correct usage of programming constructs (eg use correct type for variables)
   (b) Not overly complicated logic
   (c) No redundant logic
7. Things to look out for under programming style
   (a) Meaningful comments (including pre- and post-condition description if necessary)
   (b) Proper indentation
   (c) Meaningful identifiers