CS 455 Programming Assignment 2

Spring 2024 [Bono]
Due: Wed, Feb. 21, 11:59pm

Introduction

In this assignment you will get practice working with Java ArrayList, more practice implementing your own classes, and practice creating a procedural design in Java (in the BookshelfKeeperProg main class). Like you did in assignment 1, you will be implementing classes whose specification we have given you, in this case classes named Bookshelf, to represent books on a shelf that must be accessed in a specific way, and BookshelfKeeper, to represent a sorted Bookshelf. You will also be using tools to help develop correct code, such as assert statements, along with code to verify that your class implementation is consistent.

Note: this program is due after your midterm exam, but it's a fair amount bigger than the first assignment; we recommend getting started on it before the midterm. It only uses topics from before the midterm, so working on it now will also help you prepare for the exam. You will want to read the whole thing over before starting, but a suggested milestone is described in the section on Representation / Implementation.

Table of Contents

- How to use the provided test cases
- Converting a String into an ArrayList of numbers
- Structure of BookshelfkeeperProg
- Summary of requirements
- Extra credit
- README file / Submitting your program

Resources

- Horstmann, Section 7.7 Array Lists
- Horstmann, Special topic 8.1 Parameter passing
- Horstmann, Section 8.4 static methods
- Horstmann, Section 11.2 Text Input and Output
- Horstmann, Special topic 11.6, assert statements
- CS 455 Preconditions video (Week 4)
- CS 455 Representation invariants video (Week 4)
- Java ArrayList Documentation

The assignment files

The starter files we are providing for you on Vocareum are listed here. The files in **bold** below are ones you create and/or modify and submit. The files are:

- **Bookshelf.java** The interface for the Bookshelf class, that abstracts the idea of arranging books into a bookshelf so books don't fall down. The java file contains stub versions of the functions so it will compile. You will be completing the implementation of this class. You may not change the interface for this class, but you may add private instance variables and/or private methods to it. Described further here.

- **BookshelfKeeper.java** The interface for the BookshelfKeeper class, that enables clients to perform efficient *put* or *pick* operation on a bookshelf while maintaining it in a sorted state. The java file contains stub versions of the functions so it will compile. You will be completing the implementation of this class. You may not change the interface for this class, but you may add private instance variables and/or private methods to it. Described further here.

- **BookshelfKeeperProg.java** A terminal-based interactive program that allows the user to perform a series of pick and put operations on a BookshelfKeeper object; contains the main method. It can also be run in a batch mode by using input and output redirection. Described further here. You will create this file.

- **README** See section on Submitting your program for what to put in it. Before you start the assignment please read the following statement which you will be "signing" in the README:

> "I certify that the work submitted for this assignment does not violate USC's student conduct code. In particular, the work is my own, not a collaboration, and does not involve code created by other people or AI software, except for the resources explicitly mentioned in the CS 455 Course Syllabus. And I did not share my solution or parts of it with other students in the course."

If you choose to work on your assignment outside of the Vocareum environment, please see the detailed directions about this from assignment 1.

Note on running your program

You will be using assert statements in this program. To be able to use them, you need to *run* the program with asserts enabled (-ea flag). (You do not need to compile it any special way.) Here is an example:

```
java -ea BookshelfKeeperProg
```

You should run with this flag set every time.

Assert statements are another tool to help you write correct code. More about how you are using them here in the section on representation invariants.

NOTE: The tutorial about getting started with IntelliJ (onDocumentation page of course website) explains how to change your command-line arguments for running your program in that IDE. In Eclipse you use the Run Configurations settings (in the Run menu) to change what arguments are used when running a program. You can set both *program arguments* and *VM arguments* when running a program; `-ea` is a VM argument. (NOTE: VM stands for virtual machine, as in the Java Virtual Machine, which is what you are running when you do the "java" command.)

The assignment

You will be implementing a program to interact with a bookshelf-keeper, that maintains a shelf of books in increasing order by height, as shown in Figure 1. The program enables users to perform a series of put and pick operations via a BookshelfKeeper object that you'll implement. The *pick* operation refers to a case where you need to pick a book from a given position (index), and the *put* operation refers to putting a book of given height on the shelf, such that the bookshelf remains sorted after completing any operation. The only information you store about a book is its height. In this scenario there are further restrictions on how the operations will be done that we'll describe presently.

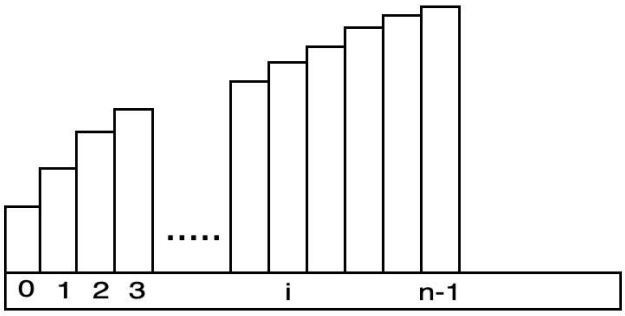

Fig 1 A typical sorted bookshelf

When doing pick or put operations, the BookshelfKeeper is only allowed to access books on the bookshelf from one of the two ends, so books don't fall over. Additionally, it must carry out the pick or put operation such that it minimizes the number of books moved this way. You'll see an example of this in the section on the BookshelfKeeper class.

The user input consists of the initial state of our bookshelf and a series of pick and put operations to perform; for each operation it will print out the contents of the updated bookshelf, the number of lower-level book-moving operations used, and the total number of lower-level operations used since the start of the program. There are more details about exactly what your input and output should look like with examples in the section on the BookshelfKeeperProg user interface.

This program will run in the console window and will not have a GUI. Here are two examples of how you might run the program in the Linux shell::

```
java -ea BookshelfKeeperProg
java -ea BookshelfKeeperProg < inputFile > outputFile
```

[Note: recall you are using the `-ea` argument for assertion-checking]

Some of the requirements for this assignment relate to testing, and style/design, as well as functionality. They are described in detail in the following sections of the document, and then summarized near the end of the document.

Overall class design

There are three classes in this program, Bookshelf, BookshelfKeeper, and BookshelfKeeperProg.

The Bookshelf class abstracts the idea of having multiple books on a bookshelf in *any* order, with the restricted access mentioned above: so books don't fall over, books on a Bookshelf can only

be inserted or removed at either of the ends, although you can look at any book on a bookshelf. More about this in the Bookshelf section.

A BookshelfKeeper contains a Bookshelf and has the pick and put operations: these must be implemented using the minimal number of Bookshelf mutator calls such that it keeps the underlying bookshelf sorted by the height of the books. More details about how these operations are performed are described below in the BookshelfKeeper section.

Finally, there is the BookshelfKeeperProg program which contains the main method and is responsible for reading user input and printing results. There are more details about exactly what your input and output should look like with examples in the section on the BookshelfKeeperProg.

We have used this approach to keep your implementation simple: once you implement the Bookshelf, you don't have to worry about how it carries out its operations as you are thinking about how to solve the BookshelfKeeper problem. In fact, you can implement and test your complete Bookshelf class before even figuring out how to write BookshelfKeeper. This is an example of separation of concerns: the only class concerned with how to implement a collection with limited book movements is Bookshelf; the only class concerned with how to minimize the number of these book movements, and maintain the order of books is BookshelfKeeper, and the only class to be concerned with the details of the user interface for the program is BookshelfKeeperProg (neither of the other classes do any I/O).

Separating these different levels of abstraction also makes it easier to modify the program later: one could substitute a different user interface (e.g., a GUI) and Bookshelf and BookshelfKeeper would not have to change at all. Or one could substitute a different implementation for a Bookshelf, and as long as its interface didn't change, the other two classes would not have to change at all. (For this assignment, however, you will not be making any alternate versions of the classes or user interface.)

Bookshelf class

What follows is the specification for the Bookshelf class. As mentioned previously, books can only be accessed in a specific way so they don't fall down: you can add or remove a book only when it's at one of the ends of the shelf. However, you can look at any book on a shelf by giving its current location (book locations start from 0). Books are identified only by their height. You can think of two books of the same height as just being two copies of the same book. You must implement the following methods so they work as described:

Bookshelf() : Creates an empty Bookshelf i.e. with no books.

Bookshelf(ArrayList<Integer> pileOfBooks) : Creates a Bookshelf with the arrangement specified in pileOfBooks. For example, when given [20, 5, 9], creates a bookshelf where 20 is the book at position 0, 5 is the book at position 1, and 9 is the book at position 2. Precondition: pileOfBooks contains an array list of 0 or more positive numbers representing the height of each book.

void addFront(int height) : Inserts book with specified height at the start of the Bookshelf, i.e., it will end up at position 0. Precondition: height > 0.

void addLast(int height) : Inserts book with specified height at the end of the Bookshelf. Precondition: height > 0

int removeFront() : Removes book at the start of the Bookshelf and returns the height of the removed book. Precondition: size() > 0

int removeLast() : Removes book at the end of the Bookshelf and returns the height of the removed book. Precondition: size() > 0

int getHeight(int position) : Gets the height of the book at the given position. Precondition: 0 <= position < size()

int size() : Returns the number of books on this Bookshelf.

boolean isSorted() : Returns true iff the books on this Bookshelf are in non-decreasing order. (Note: this is an accessor; it does not change the bookshelf.)

String toString(): Returns string representation of this Bookshelf. Returns a string with the height of all books on the bookshelf, in the order they are in on the bookshelf, using the format shown by example here:
"[7, 33, 5, 4, 3]"

You may have noticed that there is another method, **isValidBookshelf**, that's private, i.e., not part of the public interface. We will describe that later when we discuss representation invariants.

See the Representation/Implementation section for more hints on how to implement this class.

BookshelfKeeper class

The BookshelfKeeper maintains a Bookshelf whose books are kept in increasing order by height by the keeper, and allows the client to (1) *pick* a book from the shelf, given its *position* or (2) *put* a book on the shelf, given the *height* of the book. To remind us of which operation takes which argument type, we'll henceforth refer to these operations as **`pickPos`** and **`putHeight`**. It must do this by using a Bookshelf object to store the books (we'll refer to this "its bookshelf" or "the contained bookshelf"). The higher level pickPos and putHeight BookshelfKeeper operations must be done in terms of the lower level Bookshelf operations that add or remove books from the ends. **Additionally, the pickPos and putHeight operations must be performed using the minimum number of add/remove calls on its bookshelf.** All other bookshelf methods can be called as many times as you like. Only calls to addFront(), removeFront(), addLast() and removeLast() need to be counted (i.e., calls to Bookshelf mutators). BookshelfKeeper is also responsible for maintaining its bookshelf in non-decreasing order of heights before and after such operations are completed.

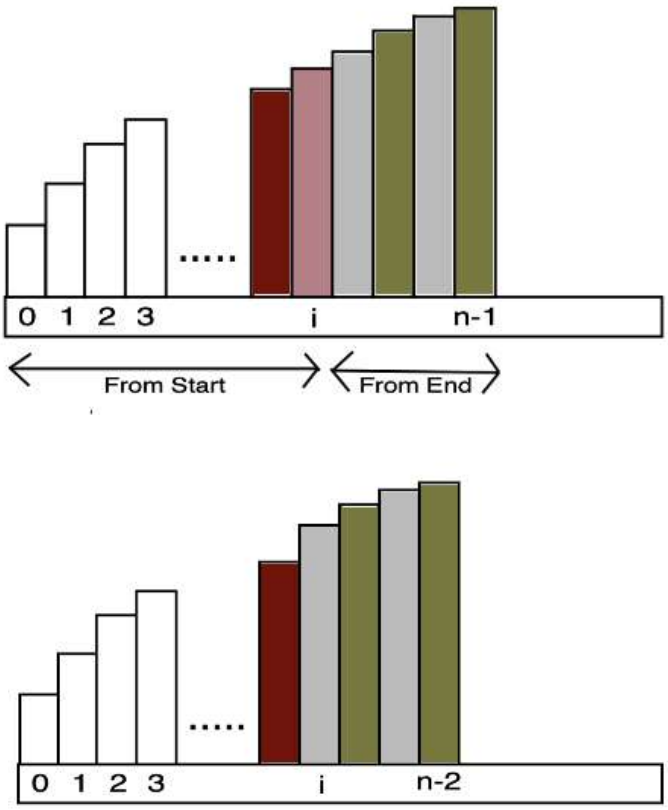

Fig. 2 Bookshelf before and after picking the book at position i

Let us take an example of a pickPos operation where you need to pick a book from position i as shown in figure 2. In order to perform this operation using Bookshelf methods you need to first remove books one at a time from one end until the book at position i is removed and then place the removed books (except the one you are picking) back on the shelf one at a time. All this needs to be done by making repeated calls to one of the four mutators of the Bookshelf object (addFront, removeFront, addLast and removeLast). The final arrangement achieved after removing the book formerly at position *i* is shown in figure 2. You may notice that the position of all books formerly to the right of i have a position one less than before.

As you may have realized by now, the pickPos operation can be performed either from the start of the bookshelf or from the end. To do it from the start, you would need to repeatedly call removeFront() to reach to book *i* as shown in figure 3(a). Similarly, from the end, you could repeatedly call removeLast() and the intermediate arrangement would look like figure 3(b). However, the BookshelfKeeper is required to choose the end which will result in the minimum number of total calls to any of the four Bookshelf methods mentioned above.

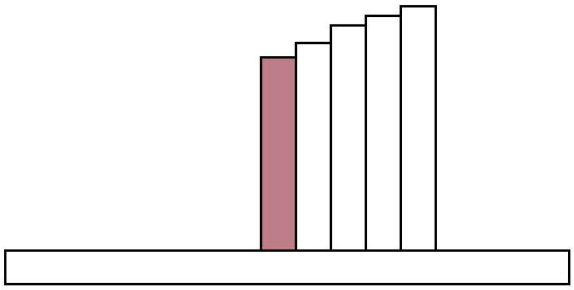

Fig. 3(a) picking from start of the shelf

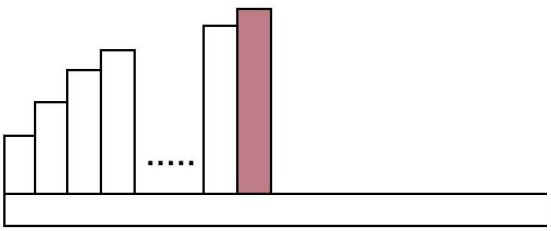

Fig. 3(b) picking from end of the shelf

Since you have now seen how to perform a pickPos operation, a putHeight operation can be performed similarly. A putHeight operation takes a book height as its argument, so the BookshelfKeeper first needs to identify the position where it needs to insert the new book, and then reach that position from the one of the two ends that results in a minimum number of calls to any of the four mutators on the contained bookshelf.

What follows below is the specification for the BookshelfKeeper class. You must implement the following methods so they work as described:

BookshelfKeeper() : Creates a BookShelfKeeper object with an empty bookshelf

BookshelfKeeper(Bookshelf sortedBookshelf) : Creates a BookshelfKeeper object initialized with the given *sorted* bookshelf. Precondition: sortedBookshelf.isSorted() is true.

int pickPos(int position) : Removes a book from the specified position in the bookshelf and keeps bookshelf sorted after picking the book. Returns the number of calls to mutators on the contained bookshelf used to complete this operation. This must be the minimum number to complete the operation. Precondition: 0 <= position < getNumBooks()

int putHeight(int height) : Inserts book with specified height into the shelf. Keeps the contained bookshelf sorted after the insertion. Returns the number of calls to mutators on the contained bookshelf used to complete this operation. This must be the minimum number to complete the operation. Precondition: height > 0

int getTotalOperations() : Returns the total number of calls made to mutators on the contained bookshelf so far. I.e., all the ones done to perform all of the pickPos and putHeight operations that have been requested since we created this BookshelfKeeper.

int getNumBooks() : Returns the number of books on the contained bookshelf.

String toString(): Returns a string representation of this BookshelfKeeper. Returns a String containing the height of all books present in the bookshelf in the order they are on the bookshelf, followed by the number of bookshelf mutator calls made to perform the last pickPos or putHeight operation, followed by the total number of such calls made since you created this BookshelfKeeper. Example return string showing required format:
“[1, 3, 5, 7, 33] 4 10”

Note: No Bookshelf or BookshelfKeeper methods do any I/O.

To get full credit on the assignment your BookshelfKeeper must work, including using the minimum number of operations on the contained bookshelf, only for situations with no duplicate books. I.e., you can assume when we test your program that all books in the initial BookshelfKeeper will have distinct heights, and any new books put in the BookshelfKeeper will have heights distinct from those already in the BookshelfKeeper. There is an optional extra credit portion for dealing with duplicate heights, described later.

We recommend you make liberal use of private helper methods in the BookshelfKeeper to simplify your public method implementations and possibly reuse code. Additionally, one of the style guidelines is to limit any method to at most 30 lines of code. More about that in the Summary of Requirements.

Representation/implementation

Bookshelf: The most straightforward way to represent data in a Bookshelf is as an ArrayList of the book heights, even though that’s not necessarily the most efficient representation. (For this assignment we are mostly concerned with efficiency in terms of number of calls to Bookshelf operations, not efficiency of how those operations are implemented.) For any advanced

students who decide to choose a different representation, you should document this in your README and explain why your choice is more efficient than the ArrayList representation.

Hints on developing the Bookshelf class: you can implement and unit-test your complete Bookshelf class before even figuring out how to write BookshelfKeeper; this would be an excellent milestone for this project (you will get credit for this milestone by doing it for Lab 5; see lab description for details.) Also, the Bookshelf code will be fairly short: peruse the Java ArrayList documentation for helpful methods.

Note: make a copy of the ArrayList you pass into the second BookShelf constructor (i.e. the ArrayList instance variable should be a different ArrayList object than the one the client passed in). With Java reference semantics, when you don't make a copy, it means both client code and Bookshelf code can change the object that is supposed to belong to the Bookshelf, which could result in unexpected behavior in your test program (Lab 5 milestone). There's an ArrayList copy constructor you can use for this. Here's an example of it's use:

```
ArrayList<Integer> foo = new ArrayList<>();
foo.add(32);
foo.add(15);
ArrayList<Integer> bar = new ArrayList<>(foo);     // call the copy
constructor
```

`bar` refers to a different object than `foo` does, but both array lists have the same value. This so-called *defensive copy* is something we'll be discussing in lecture soon.

As you learned earlier in the semester, in contrast, the following statement only copies object references:

```
ArrayList<Integer> blob = foo;       // blob and foo refer to the same
object
```

BookshelfKeeper: In order to implement the BookshelfKeeper class, you will need a Bookshelf object plus any other instance variables necessary to represent its state.

In our earlier example we didn't show where you would put all the books you temporarily take off the shelf to get to the correct position to do a pickPos or putHeight operation: recall you can only remove or add books from the ends of a bookshelf. Because of the way you'll need to use this extra "pile" of books, it makes sense to use a temporary Bookshelf object to store them. The temporary bookshelf would be a local variable, only used to help with one pickPos or putHeight operation. **Note: you do not count calls to methods on this temporary bookshelf in your totals for the main bookshelf.** Put another way, you can safely assume that calling methods on this auxiliary Bookshelf is free for you.

Representation invariants

Many of the development techniques we discuss in this class, for example, incremental development, the use of good variable names, and unit-testing, are to help you write correct code (and make it easier for you or someone else to enhance that code later). Another way to ensure correct code within a class is to make explicit any restrictions on what values are allowed to be in a private instance variable, and any restrictions on relationships between values in different instance variables in our object. Or put another way, making sure you know what must be true about our object representation when it is in a valid state. These are called representation invariants.

Representation invariants are statements that are true about the object as viewed by the implementor. Since for many classes, once a constructor has been called the other methods can

be called in any order, you need to ensure that none of the constructors or mutators can leave the object in an invalid state. It will be easier to do that if you know what those assumptions are. You can think of representation invariants as part of how one documents their instance variables.

There are two assignment requirements for your Bookshelf and BookshelfKeeper classes related to this issue listed here, and described in more detail right after that.

1. In a comment just above or below your private instance variable definitions for each of Bookshelf and BookshelfKeeper, list the representation invariants for that class.

2. Write the private boolean method isValidBookshelf() and isValidBookshelfKeeper(), and call them from other places in your program as described below.

1. The representation invariant comments

Write a list of all the conditions that the internals of an object of that class must satisfy. That is, conditions that are always true about the data in a valid object. For example in BookshelfKeeper class, one or more invariants would be related to the restriction that the contained Bookshelf is always in non-decreasing order.

2. isValidBookshelf() and isValidBookshelfKeeper() methods

These private methods will test the representation invariant for the internals of an object of that class. It will return true iff it is valid, i.e., the invariants are satisfied.

Call this function at the end of every public method, including the constructors, to make sure the method leaves the object in a valid state. This is one kind of sanity check: one part of a program double-checking that another part is doing the right thing (similar to printing expected results and actual results). Rather than putting this test in an if statement, you're going to put it in an assert statement. For example at the end of every public Bookshelf method (but before the return statement) you'll add the statement:

```
assert isValidBookshelf();
```

Assert statements are described in Special topic 11.6 of the text. Please make sure you are running your program with assertions enabled for every run of this program, since it's in a development stage. See earlier section for how to do this. You won't really know if they are getting checked unless you force one to fail.

The point of these assert statements is to notify you in no uncertain terms of possible bugs in your code. The program crashing will force you to fix those bugs. For example, in BookshelfKeeper if the books are not arranged in non-decreasing order in terms of their heights, then other methods such as pickPos or putHeight may not work as advertised.

User interface for BookshelfKeeperProg

BookshelfKeeperProg contains the main method that allows the user to perform a series of pickPos and putHeight operations on a bookshelf in an interactive mode with user commands called **`pick`** and **`put`**. It can also be run in a batch mode by using input and output redirection. Please take a look at the following examples for what your output must look like. User input is shown in **bold** in the examples below. The part in parentheses and italics on the right is an explanation of what's going on (not part of the input or output in the example).

Run 1: [Note: this sample input is available to you on Vocareum as the file `test1`, and the correct output as `test1.out`]

```
Please enter initial arrangement of books followed by newline:
1 5 8
[1, 5, 8] 0 0                          (no pick or put operations have been done yet)
Type pick <index> or put <height> followed by newline. Type end to
exit.
put 10
[1, 5, 8, 10] 1 1                      (fewer operations starting from the right end)
put 6
[1, 5, 6, 8, 10] 5 6          (either end results in the same number of bookshelf
operations)
put 4
[1, 4, 5, 6, 8, 10] 3 9        (puts it starting from the left end)
pick 3
[1, 4, 5, 8, 10] 5 14          (picks it starting from the right end)
end
Exiting Program.
```

Note that users are allowed to enter *no* numbers for the initial arrangement. E.g., if the call to in.nextLine() resulted in a String with no integers i.e. contains only whitespace character or is empty, it should be accepted as valid input by your program as demonstrated here in Run 2:

Run 2: [Note: this sample input is available to you on Vocareum as `test2`, and the correct output as `test2.out`]

```
Please enter initial arrangement of books followed by newline:

[] 0 0
Type pick <index> or put <height> followed by newline. Type end to
exit.
put 1
[1] 1 1
put 10
[1, 10] 1 2                            (puts it starting from the right end)
put 5
[1, 5, 10] 3 5                   (either end results in the same number of bookshelf
operations)
put 8
[1, 5, 8, 10] 3 8                (puts it starting from the right end)
```

```
pick 1
[1, 8, 10] 3 11                    (picks it starting from the left end)
end
Exiting Program.
```

Also, note that you're not forcing the user to enter *exactly* one space between the numbers entered for the initial configuration. E.g., if the call to in.nextLine() resulted in the following String, it should be accepted as valid input by your program:

```
"    1    9     11 13    17 19    "
```

Tabs are ok as whitespace too but we can't show them easily here. For input after the initial configuration, you also should accept whitespace on the input line. E.g., the following string should be considered valid:

```
"        put          14   "
```

You can assume that the line specifying an operation cannot be left blank.

Your output for a particular input must match what's shown above character-by-character, so we can automate our tests when we grade your program. The submit script will do a few of the automated tests (and give you a report on the results), thus we recommend you try your first submission early, so if it fails you would have time to still fix your code and resubmit before the final deadline. See the section on submitting for more details about this.

Error checking

This section illustrates the error-checking required for this program. When you encounter one of these errors, your program will print the error message and then immediately exit. Your program must ensure that:

- the list of numbers entered for initial arrangement of the bookshelf are all positive and in non-decreasing order.
- Once the initial configuration is read the only commands allowed are *pick*, *put*, and *end*.
- For put, the height given must be positive.
- For pick, the position given must be within the valid bounds for the bookshelf.

You do not have to check that the height or position are present and are integers (i.e., we will only test it on cases where the integer argument to *pick* or *put* is given and is an integer).

Please take a look at the following example for what your output must look like while handling errors. User input is shown in bold in the examples below.

Run 1:

```
Please enter initial arrangement of books followed by newline:
1 10 3 19
ERROR: Heights must be specified in non-decreasing order.
Exiting Program.
```

Run 2:

```
Please enter initial arrangement of books followed by newline:
-1 2 6 10
ERROR: Height of a book must be positive.
Exiting Program.
```

Run 3:

```
Please enter initial arrangement of books followed by newline:
1
[1] 0 0
Type pick <index> or put <height> followed by newline. Type end to
exit.
remove 0
ERROR: Invalid command. Valid commands are pick, put, or end.
Exiting Program.
```

Run 4:

```
Please enter initial arrangement of books followed by newline:
1
[1] 0 0
Type pick <index> or put <height> followed by newline. Type end to
exit.
pick 1
ERROR: Entered pick operation is invalid on this shelf.
Exiting Program.
```

Run 5:

```
Please enter initial arrangement of books followed by newline:
1
[1] 0 0
Type pick <index> or put <height> followed by newline. Type end to
exit.
put -1
ERROR: Height of a book must be positive.
Exiting Program.
```

Please note here that your error messages for cases with invalid input must match what's shown above character-by-character. All above examples and corresponding correct output are available on Vocareum in the files `err1` and `err1.out`, `err2` and `err2.out`, etc. to correspond to the above examples Run 1, Run 2, etc.

If a user enters data for a bookshelf that would result in multiple errors, your program is allowed to just print an error message about the first problem your program discovered, and then exit.

How to use the provided test cases:

As mentioned previously, we have provided sample input and output files for the valid and invalid input cases shown above. They are all available in a subdirectory of your home directory called

`test`. You should use input and output redirection to check that your results match what we have provided. You can use the following command pattern to generate an output file corresponding to a particular input file:

```
java -ea BookshelfKeeperProg < inputFile > yourOutputFile
```

Once your output file is generated, compare it with the provided output file using the Linux *diff* command. The statement below should produce no output if your program works correctly:

```
diff yourOutputFile providedOutputFile
```

For example for the test file provided with name `test1`, you will do something like the following

```
java -ea BookshelfKeeperProg < test/test1 > myout1.out
    diff myout1.out test/test1.out
```

to compare your output (`myout1.out`) with our reference output (`test1.out`).

Converting a String into an ArrayList of numbers

You will not be able to read in the initial sequence of books directly into an ArrayList of integers using repeated calls to nextInt, because you're using newline as a sentinel (i.e., a signal that that's the end of the input data), and nextInt skips over (as in, doesn't stop for) newlines. So you'll want to first read the input all at once using the Scanner nextLine method, and then convert them to an ArrayList of integers.

How does one do such a conversion? Section 11.2.5 of the textbook (called Scanning a String) shows one way of solving the problem of processing an indeterminate number of values all on one line. It takes advantage of the fact that the Scanner class can also be used to read from a String instead of the keyboard. Once you have your String of ints from the call to nextLine(), you create a *second* Scanner object initialized with this string to then break up the line into the parts you want, using the Scanner methods you are already familiar with.

Structure of BookshelfKeeperProg

The code for BookshelfKeeperProg is too long to be readable if you put it all into the main method. One could design and add another object to deal with the user interface, but instead here you'll use a procedural design in this main class to organize that code; we'll review procedural design here.

A good design principle (for procedural as well as object-oriented programming) is to keep each of your methods small, for easier program readability. In object-oriented programming, the class design sometimes naturally results in small methods, but sometimes you still need auxiliary private methods. The same principles apply for a procedural design. Since we haven't given you a predefined method decomposition for the BookshelfKeeperProg, you will have to create this decomposition yourself.

Java does not have stand-alone functions, unlike many other languages, so a procedural design in Java is just implemented as static methods in a Java main class that pass data around via explicit parameters. Static methods are discussed in Section 8.4 of the text, and this use of them was also discussed in a sidebar in Lab 4. You have seen a few examples of this in other test programs you have written, for example NumsTester.java of lab 4, and PartialNamesTester.java we developed in lecture. You have also seen some utility classes in Java that have only static methods: Math and Arrays (those two were not main classes, but rather, just classes to hold a bunch of static methods).

If you have learned about procedural design in other programming classes, you know that global variables are a no-no. This is another example of the principle of locality. Thus, in designing such a "main program" class, you don't create any class-level variables, because they become effectively global variables (see also Style Guideline #11). The "main class" does not represent any overall object. Instead you will create variables *local* to main that will get passed to (or returned from) its helper methods, as necessary.

Note: the Summary section below discusses a limit on method length as one of our style guidelines for this course.

A note about the System.in Scanner

This (and all Java programs that read from the console) should only have one Scanner object to read from System.in. If you make multiple such Scanner objects your program will not work with our test scripts. You will also have problems if you try to open and close multiple Scanners from System.in in your code. Once you create that one Scanner, you can pass it as a parameter to other methods to be able to use it in different places. Here is a little program with an example of this:

```
public class MyClass {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);  // create the Scanner
      . . .
      int dataVal = in.nextInt();  // using in directly in main
      . . .
      // readAndValidateString will also read some more input
      String moreData = readAndValidateString(in); // pass in as a param
here
      . . .
   }

   // prompts for a String from "in", reads it, and validates it.
   private static String readAndValidateString (Scanner in) {
      // don't create another Scanner for Sytem.in here; use the
parameter instead
      . . .
      String theString = in.next();
      . . .
      return theString;
   }
   . . .
}
```

As you may know by now, you can create Scanner objects that have different data sources. You may have multiple Scanner objects in your program overall: we're just saying you should only have one that was created with System.in as the source, i.e., using
`new Scanner(System.in)`

Summary of requirements

As in the first assignment, there are several requirements for this assignment related to design, testing, and development process strewn throughout this document. We'll summarize those and the functional requirements here:

- implement Bookshelf class according to its public interface (see Bookshelf class section). Make use of the representation for Bookshelf class described in this section.

- write representation invariant comments for Bookshelf and BookshelfKeeper classes.

- implement and use private Bookshelf and BookshelfKeeeper methods `isValidBookshelf` and `isValidBookshelfKeeper`, respectively, as described here.

- follow similar steps to implement BookshelfKeeper class according to its public interface (see BookshelfKeeper class section), including the previously described requirements about minimizing the number of bookshelf operations

- implement BookshelfKeeperProg with the user interface described in the section about the BookshelfKeeperProg.

- only use the expression `new Scanner(System.in)` at most once in your program (see previous section)

- do the error checking described in the Error Checking section.

- your code will also be evaluated on style, documentation, and design. We will deduct points for programs that do not follow the published style guidelines for this course (they are also linked from the Assignments page). (Note: For pa1 we only deducted points for problems related to some of the style guidelines.) One guideline we want you to be especially aware of is the limit of 30 lines of code at most allowable in a method. This is exclusive of whitespace, comment lines, and lines that just have a curly bracket by itself (i.e., you should not sacrifice code-readability to make your code fit into this limit).

For these "style" points we will also take into account the general quality of your design.

Extra credit

You may assume that all the books on the shelf in a BookshelfKeeper have distinct heights in order to complete the mandatory part of this assignment. However, you can earn a little extra credit if your submitted program also works *and uses the minimum number of mutator calls on the contained bookshelf* for scenarios where there are some books of the same height on the shelf. (You can think of these as multiple copies of the same book.) For example a shelf with: [3, 7, 7, 7, 12, 12, 12, 12, 15, 18], and maybe you want to put in a book with height 12, or pick the book at location 5, using the minimal number of Bookshelf operations. For pickPos, the correct minimum number of operations to report will be for actually picking the book at the given location, not some copy of the same book at a different location. E.g., pickPos(1) on [3, 3, 3, 3] would take 3 operations. You have also been provided with a few test files for the extra credit part of this assignment, under `$ASNLIB/public` in the PA2 Vocareum workspace.

If you are a student who is struggling in the class or starting the program on the late side, we recommend not attempting the extra credit, or only doing so if you have extra time after you have a complete, tested, working program (and save a backup of that working program).

README file / Submitting your program

You will be submitting Bookshelf.java, BookshelfKeeper.java, BookshelfKeeperProg.java and README. Make sure your name and login id appear at the start of each file. Reminder: if you developed your code locally on your laptop, before submitting you will need to upload your code to Vocareum, and compile and re-test it there, and then submit it there, possibly repeating if you fail the submit tests. There were more directions about this option in PA1. Do not wait until the last minute to do the testing on Vocareum.

Here's a review of what goes in the README: This is the place to document known bugs in your program. That means you should describe thoroughly any test cases that fail for the program you are submitting. (You should not include a history of the bugs you already fixed.) You also use the README to give the grader any other special information, such as if there is some special way to compile or run your program, or to let them know whether you would like to be evaluated for extra credit. You will also be signing the certification shown near the top of this document.

When you are ready to submit the assignment press the big "Submit" button in your PA2 Vocareum work area (don't wait until right before the deadline to do this for the first time). The submit script will check that you have the correct files in your work area and whether they compile. In addition, it will run your program on the test cases that we also provided you with the assignment, in particular to check if your output matches the correct output character-by-character, and to check if your program does the required error-checking. Make sure you read the submission report (more details about this were on lab 1). If you fail any of these tests (especially for non-error checking cases), you are unlikely to pass many of the other automated tests we will be doing when grading your code, so please go back and fix your code so it produces the correct output *before* your final submit.

Passing these submit checks is not necessary or sufficient to submit your code (the graders will get a copy of what you submitted either way). (It would be necessary but not sufficient for getting full credit.) However, if your code does not pass all the tests we would expect that you would include some explanation of that in your README. One situation where it might fail would be if you only completed a subset of the assignment (and in that case your README should document what subset you completed.)

You are allowed to submit as many times as you like, but we will only grade the last one submitted. If you are unsure of whether you submitted the right version, there's a way to view the contents of your last submission in Vocareum. See the item in the file list on the left called "Latest Submission".
