**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 characters, 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 described 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](https://www.google.com/url?q=http://bytes.usc.edu/cs455/curr/labs/lab4/lab4.html%23static&sa=D&source=editors&ust=1675476041604796&usg=AOvVaw2HSlr-9-M1Y6IknDN5RIwW) 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](https://www.google.com/url?q=http://bytes.usc.edu/cs455/curr/assgts/style.html&sa=D&source=editors&ust=1675476041605760&usg=AOvVaw0pKZmHst7UxejcBthRM2aP) #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](https://bytes.usc.edu/cs455/curr/assgts/pa2/pa2.html#h.s2ejry2x3own) 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();

      . . .

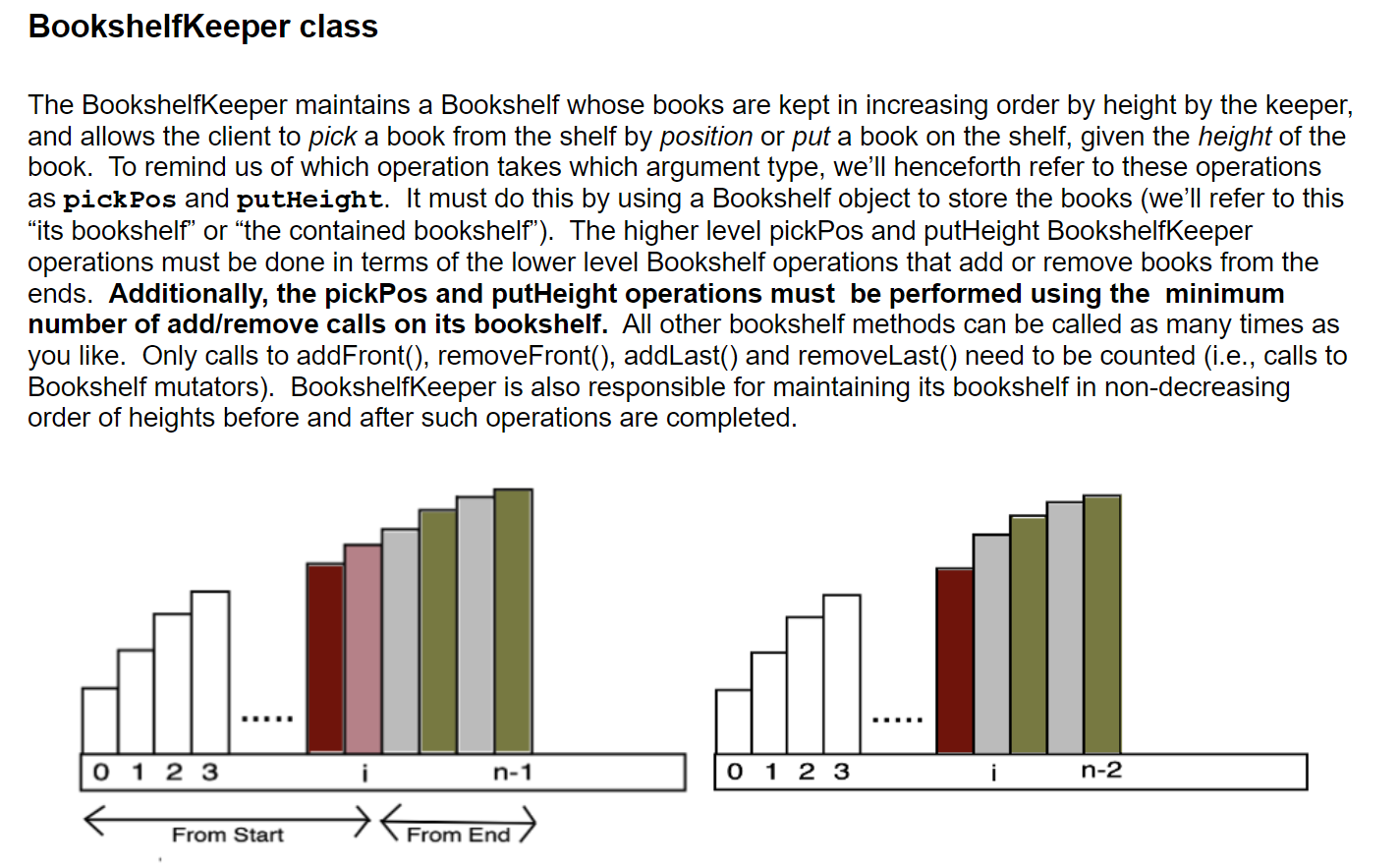
      return theString;

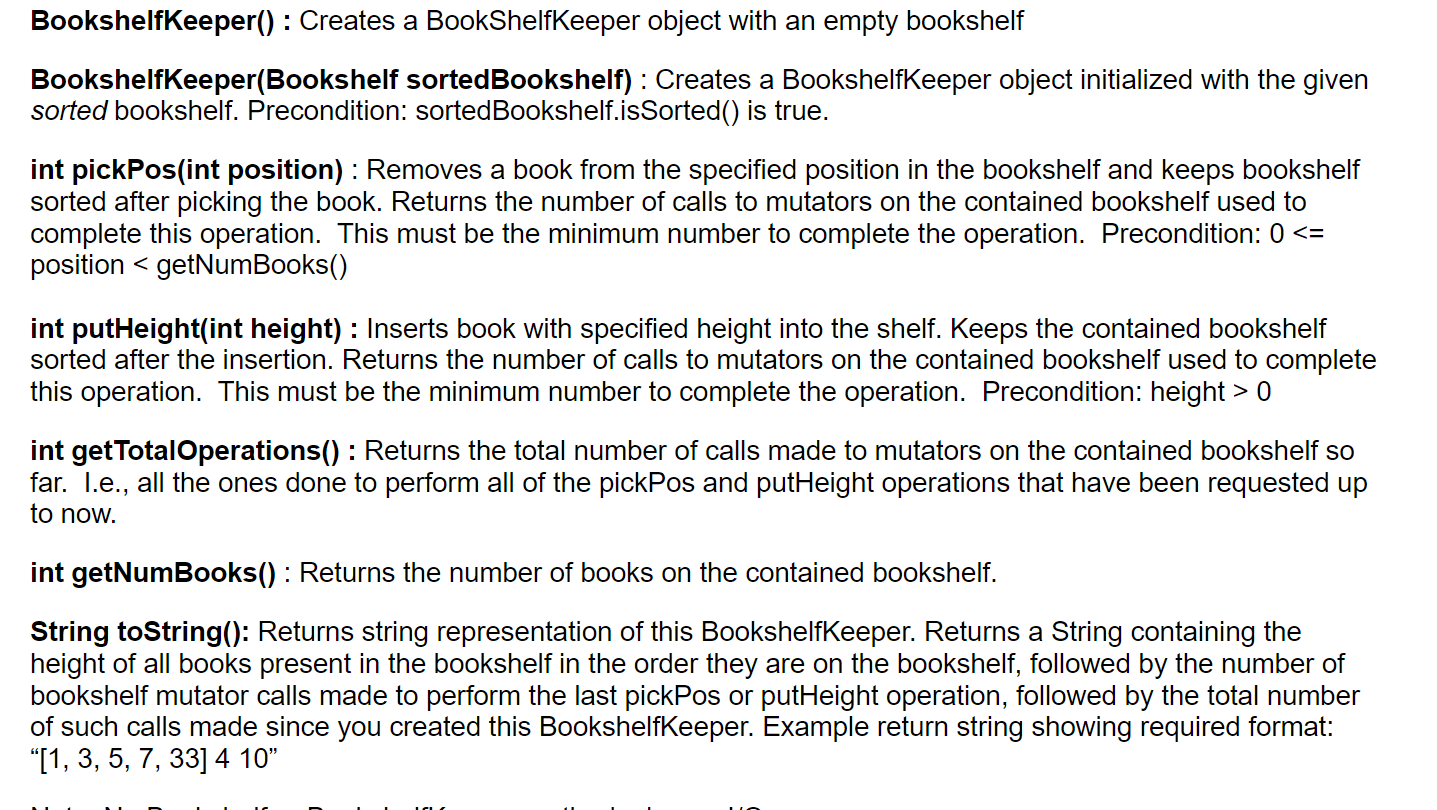
   }

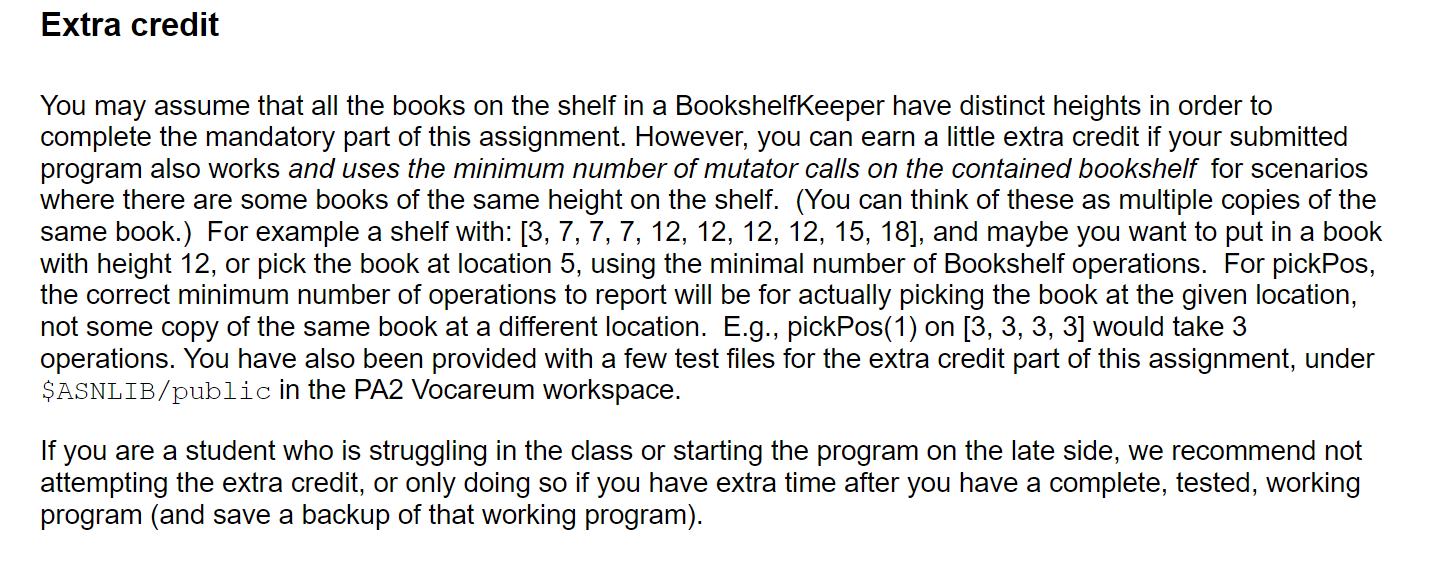
   . . .

}

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)







**============================================================================**

**可提交的====================================================================**

**public class BookshelfKeeperProg {**

**public static void main(String[] args) {**

**System.out.println("Please enter initial arrangement of books followed by newline: ");**

**Scanner in = new Scanner(System.in);**

**ArrayList<Integer> booksList = new ArrayList<>();**

**String line = in.nextLine();**

**Scanner lineScanner = new Scanner(line);**

**while (lineScanner.hasNextInt()) {**

**int data = lineScanner.nextInt();**

**booksList.add(data);**

**}**

**Bookshelf pileBooks = new Bookshelf(booksList);**

**BookshelfKeeper sortedBooks = new BookshelfKeeper(pileBooks);**

**boolean valIniBooks = readIniBooks(pileBooks);**

**if (!valIniBooks) {**

**System.out.println("Exit Program.");**

**return;**

**}**

**System.out.println(sortedBooks.toString());**

**System.out.println("Type pick <index> or put <height> followed by newline. Type end to exit.");**

**while (in.hasNextLine()) {**

**String currentInput = in.nextLine();**

**if ("end".equalsIgnoreCase(currentInput.trim())) {**

**System.out.println("Exit Program.");**

**return;**

**}**

**Scanner currentScanner = new Scanner(currentInput);**

**String currentData = currentScanner.findInLine(Pattern.compile("(-|\\+)?\\d+"));**

**int currentBook = Integer.valueOf(currentData);**

**boolean valAddBooks = readAddBooks(currentInput, currentBook, sortedBooks);**

**if (!valAddBooks) {**

**System.out.println("Exit Program.");**

**return;**

**}**

**if (currentInput.contains("put")) {**

**sortedBooks.putHeight(currentBook);**

**System.out.println(sortedBooks.toString());**

**}**

**if (currentInput.contains("pick")) {**

**sortedBooks.pickPos(currentBook);**

**System.out.println(sortedBooks.toString());**

**}**

**if (!readIniBooks(pileBooks)) {**

**System.out.println("Exit Program.");**

**return;**

**}**

**}**

**System.out.println(sortedBooks.toString());**

**System.out.println("Exit Program.");**

**}**

**private static boolean readIniBooks(Bookshelf pileBooksVal) {**

**if (Objects.isNull(pileBooksVal)) {**

**return false;**

**}**

**if (!pileBooksVal.isSorted()) {**

**System.out.println("ERROR: Heights must be specified in non-decreasing order.");**

**return false;**

**}**

**if (!pileBooksVal.isValidBookshelf()) {**

**System.out.println("ERROR: Height of a book must be positive.");**

**return false;**

**}**

**return true;**

**}**

**private static boolean readAddBooks(String addBooks, int dataVal, BookshelfKeeper bookshelfKeeper) {**

**if (Objects.isNull(bookshelfKeeper)) {**

**return false;**

**}**

**if (dataVal < 0) {**

**System.out.println("ERROR: Height of a book must be positive.");**

**return false;**

**}**

**if (addBooks.contains("remove")) {**

**System.out.println("ERROR: Invalid command. Valid commands are pick, put, or end.");**

**return false;**

**}**

**if ("pick".equalsIgnoreCase(addBooks) && dataVal > (bookshelfKeeper.getNumBooks() - 1)) {**

**System.out.println("Entered pick operation is invalid on this shelf.");**

**return false;**

**}**

**return true;**

**}**

**}**

**public class BookshelfKeeper {**

**/\*\***

**\* Representation invariant:**

**\* <p>**

**\* <put rep. invar. comment here>**

**\* -- maintaining its bookshelf in non-decreasing order of heights before and after operations are completed.**

**\*/**

**// <add instance variables here>**

**private Bookshelf sortedBooks;**

**private int numCurrent = 0;**

**private int numTotal = 0;**

**/\*\***

**\* Creates a BookShelfKeeper object with an empty bookshelf**

**\*/**

**public BookshelfKeeper() {**

**sortedBooks = new Bookshelf();**

**assert isValidBookshelfKeeper();**

**}**

**/\*\***

**\* Creates a BookshelfKeeper object initialized with the given sorted bookshelf.**

**\* Note: method does not make a defensive copy of the bookshelf.**

**\* <p>**

**\* PRE: sortedBookshelf.isSorted() is true.**

**\*/**

**public BookshelfKeeper(Bookshelf sortedBookshelf) {**

**sortedBooks = sortedBookshelf;**

**assert isValidBookshelfKeeper();**

**}**

**/\*\***

**\* Removes a book from the specified position in the bookshelf and keeps bookshelf sorted**

**\* after picking up the book.**

**\* <p>**

**\* 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.**

**\* <p>**

**\* PRE: 0 <= position < getNumBooks()**

**\*/**

**public int pickPos(int position) {**

**numCurrent = 0;**

**if (position < 0 || position > getNumBooks()) {**

**return numCurrent;**

**}**

**int bookSize = sortedBooks.size();**

**numCurrent = 2 \* Math.min(position, bookSize - 1 - position) + 1;**

**numTotal = numTotal + numCurrent;**

**if (position < bookSize - 1 - position) {**

**List<Integer> temp = new ArrayList<>();**

**for (int i = 0; i < position; i++) {**

**temp.add(sortedBooks.getHeight(i));**

**}**

**for (int i = 0; i <= position; i++) {**

**sortedBooks.removeFront();**

**}**

**int length = temp.size();**

**for (int j = length - 1; j >= 0; j--) {**

**sortedBooks.addFront(temp.get(j));**

**}**

**} else {**

**List<Integer> temp = new ArrayList<>();**

**for (int i = bookSize - 1; i > position; i--) {**

**temp.add(sortedBooks.getHeight(i));**

**sortedBooks.removeLast();**

**}**

**sortedBooks.removeLast();**

**int length = temp.size();**

**for (int j = length - 1; j >= 0; j--) {**

**sortedBooks.addLast(temp.get(j));**

**}**

**}**

**return numCurrent;**

**}**

**/\*\***

**\* Inserts book with specified height into the shelf. Keeps the contained bookshelf sorted**

**\* after the insertion.**

**\* <p>**

**\* 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.**

**\* <p>**

**\* PRE: height > 0**

**\*/**

**public int putHeight(int height) {**

**numCurrent = 0;**

**int position = findInsertPosition(sortedBooks, height);**

**int bookSize = sortedBooks.size();**

**numCurrent = 2 \* Math.min(position, bookSize - position) + 1;**

**numTotal = numTotal + numCurrent;**

**if (position < bookSize - position) {**

**List<Integer> temp = new ArrayList<>();**

**for (int i = 0; i < position; i++) {**

**temp.add(sortedBooks.getHeight(i));**

**}**

**for (int i = 0; i < position; i++) {**

**sortedBooks.removeFront();**

**}**

**sortedBooks.addFront(height);**

**int length = temp.size();**

**for (int j = length - 1; j >= 0; j--) {**

**sortedBooks.addFront(temp.get(j));**

**}**

**} else {**

**List<Integer> temp = new ArrayList<>();**

**for (int i = bookSize - 1; i >= position; i--) {**

**temp.add(sortedBooks.getHeight(i));**

**sortedBooks.removeLast();**

**}**

**sortedBooks.addLast(height);**

**int length = temp.size();**

**for (int j = length - 1; j >= 0; j--) {**

**sortedBooks.addLast(temp.get(j));**

**}**

**}**

**return numCurrent;**

**}**

**/\*\***

**\* 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 pick and put operations**

**\* that have been requested up to now.**

**\*/**

**public int getTotalOperations() {**

**return numTotal;**

**}**

**/\*\***

**\* Returns the number of books on the contained bookshelf.**

**\*/**

**public int getNumBooks() {**

**assert isValidBookshelfKeeper();**

**return sortedBooks.size();**

**}**

**/\*\***

**\* Returns string representation of this BookshelfKeeper. Returns a String containing 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 pick or put operation,**

**\* followed by the total number of such calls made since we created this BookshelfKeeper.**

**\* <p>**

**\* Example return string showing required format: “[1, 3, 5, 7, 33] 4 10”**

**\*/**

**public String toString() {**

**assert isValidBookshelfKeeper();**

**return sortedBooks.toString() + " " + numCurrent + " " + numTotal;**

**}**

**/\*\***

**\* Returns true iff the BookshelfKeeper data is in a valid state.**

**\* (See representation invariant comment for details.)**

**\*/**

**private boolean isValidBookshelfKeeper() {**

**return sortedBooks.isSorted();**

**}**

**// add any other private methods here**

**private int findInsertPosition(Bookshelf sortedBooks, int insertHeight) {**

**if (Objects.isNull(sortedBooks) || sortedBooks.size() == 0) {**

**return 0;**

**}**

**int bookSize = sortedBooks.size();**

**int low = 0;**

**int high = bookSize - 1;**

**int mid = 0;**

**while (low <= high) {**

**mid = (low + high) / 2;**

**if (sortedBooks.getHeight(mid) == insertHeight) {**

**return mid;**

**} else if (sortedBooks.getHeight(mid) > insertHeight) {**

**high = mid - 1;**

**} else {**

**low = mid + 1;**

**}**

**}**

**return low;**

**}**

**}**

**public class Bookshelf {**

**/\*\***

**\* Representation invariant:**

**\* <p>**

**\* <put rep. invar. comment here>**

**\* -- height>0 height of book is always positive.**

**\*/**

**// <add instance variables here>**

**private ArrayList<Integer> allBooks;**

**private int removedHeight = 0;**

**/\*\***

**\* Creates an empty Bookshelf object i.e. with no books**

**\*/**

**public Bookshelf() {**

**allBooks = new ArrayList<>();**

**assert isValidBookshelf();**

**}**

**/\*\***

**\* Creates a Bookshelf with the arrangement specified in pileOfBooks. Example**

**\* values: [20, 1, 9].**

**\* <p>**

**\* PRE: pileOfBooks contains an array list of 0 or more positive numbers**

**\* representing the height of each book.**

**\*/**

**public Bookshelf(ArrayList<Integer> pileOfBooks) {**

**allBooks = new ArrayList<>(pileOfBooks);**

**assert isValidBookshelf();**

**}**

**/\*\***

**\* Inserts book with specified height at the start of the Bookshelf, i.e., it**

**\* will end up at position 0.**

**\* <p>**

**\* PRE: height > 0 (height of book is always positive)**

**\*/**

**public void addFront(int height) {**

**assert height > 0;**

**allBooks.add(0, height);**

**assert isValidBookshelf();**

**}**

**/\*\***

**\* Inserts book with specified height at the end of the Bookshelf.**

**\* <p>**

**\* PRE: height > 0 (height of book is always positive)**

**\*/**

**public void addLast(int height) {**

**assert height > 0;**

**allBooks.add(height);**

**assert isValidBookshelf();**

**}**

**/\*\***

**\* Removes book at the start of the Bookshelf and returns the height of the**

**\* removed book.**

**\* <p>**

**\* PRE: this.size() > 0 i.e. can be called only on non-empty BookShelf**

**\*/**

**public int removeFront() {**

**assert allBooks.size() > 0;**

**removedHeight = allBooks.get(0);**

**allBooks.remove(0);**

**assert isValidBookshelf();**

**return removedHeight;**

**}**

**/\*\***

**\* Removes book at the end of the Bookshelf and returns the height of the**

**\* removed book.**

**\* <p>**

**\* PRE: this.size() > 0 i.e. can be called only on non-empty BookShelf**

**\*/**

**public int removeLast() {**

**assert allBooks.size() > 0;**

**int lastIndex = allBooks.size() - 1;**

**removedHeight = allBooks.get(lastIndex);**

**allBooks.remove(lastIndex);**

**assert isValidBookshelf();**

**return removedHeight;**

**}**

**/\*\***

**\* Gets the height of the book at the given position.**

**\***

**\* PRE: 0 <= position < this.size()**

**\*/**

**public int getHeight(int position) {**

**assert (position >= 0 && position < allBooks.size());**

**assert isValidBookshelf();**

**return allBooks.get(position);**

**}**

**/\*\***

**\* Returns number of books on the this Bookshelf.**

**\*/**

**public int size() {**

**assert isValidBookshelf();**

**return allBooks.size();**

**}**

**/\*\***

**\* 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]”**

**\*/**

**public String toString() {**

**assert isValidBookshelf();**

**return allBooks.toString();**

**}**

**/\*\***

**\* Returns true iff the books on this Bookshelf are in non-decreasing order.**

**\* (Note: this is an accessor; it does not change the bookshelf.)**

**\*/**

**public boolean isSorted() {**

**int numBooks = 0;**

**boolean sorted = false;**

**if (allBooks.size() == 0 || allBooks.size() == 1) {**

**return true;**

**}**

**for (int i = 0; i < allBooks.size() - 1; i++) {**

**if (allBooks.get(i + 1) > allBooks.get(i)) {**

**numBooks++;**

**if ((numBooks + 1) == allBooks.size()) {**

**sorted = true;**

**}**

**}**

**}**

**assert isValidBookshelf();**

**return sorted;**

**}**

**/\*\***

**\* Returns true iff the Bookshelf data is in a valid state.**

**\* (See representation invariant comment for more details.)**

**\*/**

**public boolean isValidBookshelf() {**

**boolean valid = false;**

**if (allBooks.size() == 0) {**

**valid = true;**

**} else {**

**int numBooks = 0;**

**for (int i = 0; i < allBooks.size(); i++) {**

**if (allBooks.get(i) > 0) {**

**numBooks++;**

**}**

**}**

**if (numBooks == allBooks.size()) {**

**valid = true;**

**}**

**}**

**return valid;**

**}**

**}**

**============================================================================**

**含一些注释的================================================================**

public class BookshelfKeeperProg {

public static void main(String[] args) {

System.out.println("Please enter initial arrangement of books followed by newline: ");

Scanner in = new Scanner(System.in);

// 1. 处理初始化的数据（你记得到时候翻译一下）

ArrayList<Integer> booksList = new ArrayList<>();

String line = in.nextLine();

Scanner lineScanner = new Scanner(line);

while (lineScanner.hasNextInt()) {

int data = lineScanner.nextInt();

booksList.add(data);

}

// 2. 初始化书架

Bookshelf pileBooks = new Bookshelf(booksList);

BookshelfKeeper sortedBooks = new BookshelfKeeper(pileBooks);

// 2.1 校验初始输入是不是对的

boolean valIniBooks = readIniBooks(pileBooks);

// while 主要用来循环的，判断真假的时候不用

// while (valIniBooks) {

//

// System.out.println(sortedBooks.toString());

// }

if (!valIniBooks) {

System.out.println("Exit Program.");

return;

}

System.out.println(sortedBooks.toString());

System.out.println("Type pick <index> or put <height> followed by newline. Type end to exit.");

while (in.hasNextLine()) {

String currentInput = in.nextLine();

if ("end".equalsIgnoreCase(currentInput.trim())) {

System.out.println("Exit Program.");

return;

}

// 这里每次输入的时候都需要校验，所以就在while里面做

Scanner currentScanner = new Scanner(currentInput);

String currentData = currentScanner.findInLine(Pattern.compile("(-|\\+)?\\d+"));

int currentBook = Integer.valueOf(currentData);

boolean valAddBooks = readAddBooks(currentInput, currentBook, sortedBooks);

if (!valAddBooks) {

System.out.println("Exit Program.");

return;

}

// 参数校验过后，就可以正常的执行 put 或者是 pick 了

if (currentInput.contains("put")) {

sortedBooks.putHeight(currentBook);

System.out.println(sortedBooks.toString());

}

if (currentInput.contains("pick")) {

sortedBooks.pickPos(currentBook);

System.out.println(sortedBooks.toString());

}

// 添加书籍之后，还需再检查合法不合法

if (!readIniBooks(pileBooks)) {

System.out.println("Exit Program.");

return;

}

}

System.out.println(sortedBooks.toString());

System.out.println("Exit Program.");

}

private static boolean readIniBooks(Bookshelf pileBooksVal) {

if (Objects.isNull(pileBooksVal)) {

return false;

}

//

// boolean isValid = true;

if (!pileBooksVal.isSorted()) {

System.out.println("ERROR: Heights must be specified in non-decreasing order.");

// isValid = false;

return false;

}

// todo 这里我看是类里面有现成的方法了，所以直接调用，不用再写了

// for (int i = 0; i < pileBooksVal.size(); i++) {

//

// if (pileBooksVal.getHeight(i) < 0) {

// System.out.println("ERROR: Height of a book must be positive.");

// isValid = false;

// }

// }

if (!pileBooksVal.isValidBookshelf()) {

System.out.println("ERROR: Height of a book must be positive.");

return false;

}

// return isValid;

return true;

}

private static boolean readAddBooks(String addBooks, int dataVal, BookshelfKeeper bookshelfKeeper) {

if (Objects.isNull(bookshelfKeeper)) {

return false;

}

// todo 这种写法没问题，但是就是一般尽量是快速结束，所以就类似这样写

// boolean validVal = true;

if (dataVal < 0) {

System.out.println("ERROR: Height of a book must be positive.");

// validVal = false;

return false;

}

if (addBooks.contains("remove")) {

System.out.println("ERROR: Invalid command. Valid commands are pick, put, or end.");

// validVal = false;

return false;

}

if ("pick".equalsIgnoreCase(addBooks) && dataVal > (bookshelfKeeper.getNumBooks() - 1)) {

System.out.println("Entered pick operation is invalid on this shelf.");

// validVal = false;

return false;

}

// return validVal;

return true;

}

}

public class BookshelfKeeper {

/\*\*

\* Representation invariant:

\* <p>

\* <put rep. invar. comment here>

\* -- maintaining its bookshelf in non-decreasing order of heights before and after operations are completed.

\*/

// <add instance variables here>

private Bookshelf sortedBooks;

// private int numPick = 0;

// private int numPut = 0;

// // 当前操作次数

private int numCurrent = 0;

// // 历史操作次数

private int numTotal = 0;

// private Bookshelf temp;

// private int numBooks = 0;

// private int midNum = 0;

/\*\*

\* Creates a BookShelfKeeper object with an empty bookshelf

\*/

public BookshelfKeeper() {

sortedBooks = new Bookshelf();

assert isValidBookshelfKeeper();

}

/\*\*

\* Creates a BookshelfKeeper object initialized with the given sorted bookshelf.

\* Note: method does not make a defensive copy of the bookshelf.

\* <p>

\* PRE: sortedBookshelf.isSorted() is true.

\*/

public BookshelfKeeper(Bookshelf sortedBookshelf) {

sortedBooks = sortedBookshelf;

// temp = new Bookshelf();

// numBooks = sortedBooks.size();

// midNum = numBooks / 2;

assert isValidBookshelfKeeper();

}

/\*\*

\* Removes a book from the specified position in the bookshelf and keeps bookshelf sorted

\* after picking up the book.

\* <p>

\* 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.

\* <p>

\* PRE: 0 <= position < getNumBooks()

\*/

public int pickPos(int position) {

numCurrent = 0;

if (position < 0 || position > getNumBooks()) {

return numCurrent;

}

int bookSize = sortedBooks.size();

numCurrent = 2 \* Math.min(position, bookSize - 1 - position) + 1;

numTotal = numTotal + numCurrent;

// 执行获取

if (position < bookSize - 1 - position) {

// 从前面取

// 先拿出来

List<Integer> temp = new ArrayList<>();

for (int i = 0; i < position; i++) {

temp.add(sortedBooks.getHeight(i));

}

// 不能再上一个循环中直接加，因为下标就变化

for (int i = 0; i <= position; i++) {

sortedBooks.removeFront();

}

// 把当前的拿出去

// sortedBooks.removeFront();

// 最后把拿出来的放进去

int length = temp.size();

for (int j = length - 1; j >= 0; j--) {

sortedBooks.addFront(temp.get(j));

}

} else {

// 从后面取

List<Integer> temp = new ArrayList<>();

for (int i = bookSize - 1; i > position; i--) {

temp.add(sortedBooks.getHeight(i));

sortedBooks.removeLast();

}

// 把当前的取走

sortedBooks.removeLast();

// 把取出来的放进去

int length = temp.size();

for (int j = length - 1; j >= 0; j--) {

sortedBooks.addLast(temp.get(j));

}

}

return numCurrent;

// numPick = 0;

// if (position < midNum) {

//

// for (int i = 0; i < position; i++) {

// // todo 这里应该是i 哇

// temp.addFront(sortedBooks.getHeight(i));

// sortedBooks.removeFront();

// numPick++;

// }

// sortedBooks.removeFront();

// numPick++;

// for (int j = 0; j < position; j++) {

// sortedBooks.addFront(temp.getHeight(j));

// numPick++;

// }

// } else {

//

// for (int i = 0; i < numBooks - position; i++) {

// // todo 这里也是 -i

// temp.addFront(sortedBooks.getHeight(sortedBooks.size() - j));

// sortedBooks.removeLast();

// numPick++;

// }

// sortedBooks.removeLast();

// numPick++;

// for (int j = 0; j < numBooks - position; j++) {

// sortedBooks.addLast(temp.getHeight(j));

// numPick++;

// }

// }

//

// assert isValidBookshelfKeeper();

//

// return numPick;

}

/\*\*

\* Inserts book with specified height into the shelf. Keeps the contained bookshelf sorted

\* after the insertion.

\* <p>

\* 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.

\* <p>

\* PRE: height > 0

\*/

public int putHeight(int height) {

/\*\*

\* 思路：

\* 1. 找到要插入的位置

\* 2. 计算离左右哪边短，取短的

\* 3. 通过2n + 1 的规律计算出当前操作的次数

\* 4. 将当前操作次数和历史操作次数累加

\* 5. 将数据插入

\*/

// 每次操作前将当前次数设置为0

numCurrent = 0;

int position = findInsertPosition(sortedBooks, height);

int bookSize = sortedBooks.size();

numCurrent = 2 \* Math.min(position, bookSize - position) + 1;

numTotal = numTotal + numCurrent;

// 执行插入

if (position < bookSize - position) {

// 从前插入

// 先拿出来

List<Integer> temp = new ArrayList<>();

for (int i = 0; i < position; i++) {

temp.add(sortedBooks.getHeight(i));

}

// 不能再上一个循环中直接加，因为下标就变化

for (int i = 0; i < position; i++) {

sortedBooks.removeFront();

}

// 把当前要插入的放入

sortedBooks.addFront(height);

// 最后把拿出来的放进去

int length = temp.size();

for (int j = length - 1; j >= 0; j--) {

sortedBooks.addFront(temp.get(j));

}

} else {

// 从后插入

List<Integer> temp = new ArrayList<>();

for (int i = bookSize - 1; i >= position; i--) {

temp.add(sortedBooks.getHeight(i));

sortedBooks.removeLast();

}

// 把当前的放入

sortedBooks.addLast(height);

// 把取出来的放进去

int length = temp.size();

for (int j = length - 1; j >= 0; j--) {

sortedBooks.addLast(temp.get(j));

}

}

return numCurrent;

// numPut = 0;

// int countHeight = 0; // count the number of height which is lower than a given height

//

// for (int i = 0; i < numBooks; i++) {

// if (sortedBooks.getHeight(i) < height) {

// countHeight++;

// }

// }

// if (countHeight < midNum) {

//

// for (int j = 0; j < countHeight; j++) {

// // todo 这里下标也没变，需要i

// temp.addFront(sortedBooks.getHeight(0));

// sortedBooks.removeFront();

// numPut++;

// }

// sortedBooks.addFront(height);

// numPut++;

// for (int k = 0; k < countHeight; k++) {

// sortedBooks.addFront(temp.getHeight(k));

// numPut++;

// }

// } else {

//

// for (int j = 0; j < countHeight; j++) {

// // todo 这里下标也需要变， -i

// temp.addFront(sortedBooks.getHeight(sortedBooks.size() - 1));

// sortedBooks.removeLast();

// numPut++;

// }

// sortedBooks.addLast(height);

// numPut++;

// for (int k = 0; k < countHeight; k++) {

// sortedBooks.addLast(temp.getHeight(k));

// numPut++;

// }

// }

//

// assert isValidBookshelfKeeper();

//

// return numPut;

}

/\*\*

\* 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 pick and put operations

\* that have been requested up to now.

\*/

public int getTotalOperations() {

// numTotal += (numPick + numPut);

//

// assert isValidBookshelfKeeper();

return numTotal;

}

/\*\*

\* Returns the number of books on the contained bookshelf.

\*/

public int getNumBooks() {

assert isValidBookshelfKeeper();

return sortedBooks.size();

}

/\*\*

\* Returns string representation of this BookshelfKeeper. Returns a String containing 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 pick or put operation,

\* followed by the total number of such calls made since we created this BookshelfKeeper.

\* <p>

\* Example return string showing required format: “[1, 3, 5, 7, 33] 4 10”

\*/

public String toString() {

// todo 这里因为用了下面的+ "" 会自动转换为String

// String numPickStr = String.valueOf(numPick);

// String numPutStr = String.valueOf(numPut);

// String numTotalStr = String.valueOf(numTotal);

// String Str0 = String.valueOf(0);

assert isValidBookshelfKeeper();

return sortedBooks.toString() + " " + numCurrent + " " + numTotal;

// todo 之前这样分开定义变量，统计put 和 pick 也行，只是当前用一个变量统计了

// if (numPick > 0) {

//

// assert isValidBookshelfKeeper();

//

// return sortedBooks.toString() + " " + numPick + " " + numTotal;

// }

// if (numPut > 0) {

//

// assert isValidBookshelfKeeper();

//

// return sortedBooks.toString() + " " + numPut + " " + numTotal;

// }

//

// return sortedBooks.toString() + " " + 0 + " " + numTotal;

}

/\*\*

\* Returns true iff the BookshelfKeeper data is in a valid state.

\* (See representation invariant comment for details.)

\*/

private boolean isValidBookshelfKeeper() {

return sortedBooks.isSorted();

}

// add any other private methods here

private int findInsertPosition(Bookshelf sortedBooks, int insertHeight) {

// todo 一般都会做一些输入校验

if (Objects.isNull(sortedBooks) || sortedBooks.size() == 0) {

return 0;

}

int bookSize = sortedBooks.size();

int low = 0;

int high = bookSize - 1;

int mid = 0;

while (low <= high) {

mid = (low + high) / 2;

if (sortedBooks.getHeight(mid) == insertHeight) {

return mid;

} else if (sortedBooks.getHeight(mid) > insertHeight) {

high = mid - 1;

} else {

low = mid + 1;

}

}

return low;

}

}

public class Bookshelf {

/\*\*

\* Representation invariant:

\* <p>

\* <put rep. invar. comment here>

\* -- height>0 height of book is always positive.

\*/

// <add instance variables here>

private ArrayList<Integer> allBooks;

private int removedHeight = 0;

/\*\*

\* Creates an empty Bookshelf object i.e. with no books

\*/

public Bookshelf() {

allBooks = new ArrayList<>();

assert isValidBookshelf();

}

/\*\*

\* Creates a Bookshelf with the arrangement specified in pileOfBooks. Example

\* values: [20, 1, 9].

\* <p>

\* PRE: pileOfBooks contains an array list of 0 or more positive numbers

\* representing the height of each book.

\*/

public Bookshelf(ArrayList<Integer> pileOfBooks) {

allBooks = new ArrayList<>(pileOfBooks);

assert isValidBookshelf();

}

/\*\*

\* Inserts book with specified height at the start of the Bookshelf, i.e., it

\* will end up at position 0.

\* <p>

\* PRE: height > 0 (height of book is always positive)

\*/

public void addFront(int height) {

assert height > 0;

allBooks.add(0, height);

assert isValidBookshelf();

}

/\*\*

\* Inserts book with specified height at the end of the Bookshelf.

\* <p>

\* PRE: height > 0 (height of book is always positive)

\*/

public void addLast(int height) {

assert height > 0;

allBooks.add(height);

assert isValidBookshelf();

}

/\*\*

\* Removes book at the start of the Bookshelf and returns the height of the

\* removed book.

\* <p>

\* PRE: this.size() > 0 i.e. can be called only on non-empty BookShelf

\*/

public int removeFront() {

assert allBooks.size() > 0;

removedHeight = allBooks.get(0);

allBooks.remove(0);

assert isValidBookshelf();

return removedHeight;

}

/\*\*

\* Removes book at the end of the Bookshelf and returns the height of the

\* removed book.

\* <p>

\* PRE: this.size() > 0 i.e. can be called only on non-empty BookShelf

\*/

public int removeLast() {

assert allBooks.size() > 0;

int lastIndex = allBooks.size() - 1;

removedHeight = allBooks.get(lastIndex);

allBooks.remove(lastIndex);

assert isValidBookshelf();

return removedHeight;

}

/\*\*

\* Gets the height of the book at the given position.

\*

\* PRE: 0 <= position < this.size()

\*/

public int getHeight(int position) {

assert (position >= 0 && position < allBooks.size());

assert isValidBookshelf();

return allBooks.get(position);

}

/\*\*

\* Returns number of books on the this Bookshelf.

\*/

public int size() {

assert isValidBookshelf();

return allBooks.size();

}

/\*\*

\* 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]”

\*/

public String toString() {

assert isValidBookshelf();

return allBooks.toString();

}

/\*\*

\* Returns true iff the books on this Bookshelf are in non-decreasing order.

\* (Note: this is an accessor; it does not change the bookshelf.)

\*/

public boolean isSorted() {

int numBooks = 0;

boolean sorted = false;

// todo 这里需要特殊处理一下输入长度为1的情况

if (allBooks.size() == 0 || allBooks.size() == 1) {

return true;

}

for (int i = 0; i < allBooks.size() - 1; i++) {

if (allBooks.get(i + 1) > allBooks.get(i)) {

numBooks++;

if ((numBooks + 1) == allBooks.size()) {

sorted = true;

}

}

}

assert isValidBookshelf();

return sorted;

}

/\*\*

\* Returns true iff the Bookshelf data is in a valid state.

\* (See representation invariant comment for more details.)

\*/

public boolean isValidBookshelf() {

boolean valid = false;

if (allBooks.size() == 0) {

valid = true;

} else {

int numBooks = 0;

for (int i = 0; i < allBooks.size(); i++) {

if (allBooks.get(i) > 0) {

numBooks++;

}

}

if (numBooks == allBooks.size()) {

valid = true;

}

}

return valid;

}

// todo 演示在当前这个类中进行测试

// public static void main(String[] args) {

//

// List<Integer> allBooks = Arrays.asList(new Integer[] {1, 32, 11, 13});

// System.out.println(isss(allBooks));

//

// }

//

// private static boolean isss(List<Integer> allBooks) {

// int numBooks = 0;

// boolean sorted = false;

//

// for (int i = 0; i < allBooks.size() - 1; i++) {

//

// if (allBooks.get(i + 1) > allBooks.get(i)) {

// numBooks++;

//

// if ((numBooks + 1) == allBooks.size()) {

// sorted = true;

// }

// }

// }

//

// return sorted;

// }

}

**==============================================================================**

**原来的========================================================================**

import java.util.Scanner;

import java.util.ArrayList;

public class BookshelfKeeperProg {

public static void main(String[] args) {

Scanner in = new Scanner(System.in);

ArrayList<Integer> booksList=new ArrayList<>();

String line=in.nextLine();

Scanner lineScanner = new Scanner(line);

while(lineScanner.hasNextInt()){

int data = lineScanner.nextInt();

booksList.add(data);

}

String addVals="";

int addData=0;

while (in.hasNextLine()){

String addLine=in.nextLine();

// Scanner addScanner = new Scanner(addLine);

addVals=in.next();

addData=in.nextInt();

}

Bookshelf pileBooks = new Bookshelf(booksList);

BookshelfKeeper sortedBooks=new BookshelfKeeper(pileBooks);

boolean valIniBooks=readIniBooks(pileBooks);

while(valIniBooks){

System.out.println();

System.out.println("Please enter initial arrangement of books followed by newline: ");

System.out.println(sortedBooks.toString());

}

System.out.println("Type pick <index> or put <height> followed by newline. Type end to exit.");

boolean valAddBooks=readAddBooks(addVals,addData,sortedBooks);

while(valAddBooks){

if(addVals.contains("put")){

sortedBooks.putHeight(addData);

}

if(addVals.contains("pick")){

sortedBooks.pickPos(addData);

}

if (addVals.contains("end")){

valIniBooks=false;

valAddBooks=false;

}

System.out.println(sortedBooks.toString());

}

System.out.println("Exit Program.");

}

private static boolean readIniBooks(Bookshelf pileBooksVal){

boolean isValid=true;

if(!pileBooksVal.isSorted()){

System.out.println("ERROR: Heights must be specified in non-decreasing order.");

isValid=false;

}

for(int i=0;i<pileBooksVal.size();i++){

if(pileBooksVal.getHeight(i)<0){

System.out.println("ERROR: Height of a book must be positive.");

isValid=false;

}

}

return isValid;

}

private static boolean readAddBooks(String addBooks,int dataVal,BookshelfKeeper allBooks){

boolean validVal=true;

if(dataVal<0){

System.out.println("ERROR: Height of a book must be positive.");

validVal=false;

}

if(dataVal>(allBooks.getNumBooks()-1)){

System.out.println("Entered pick operation is invalid on this shelf.");

validVal=false;

}

if(addBooks.contains("remove")){

System.out.println("ERROR: Invalid command. Valid commands are pick, put, or end.");

validVal=false;

}

return validVal;

}

}

这是里面调用得两个：第一个今天做过了 第二个还没测试 就是用现在做不出来得测试得

第一个：

import java.util.ArrayList;

public class Bookshelf {

/\*\*

Representation invariant:

<put rep. invar. comment here>

-- height>0 height of book is always positive.

\*/

// <add instance variables here>

private ArrayList<Integer> allBooks;

private int removedHeight=0;

/\*\*

\* Creates an empty Bookshelf object i.e. with no books

\*/

public Bookshelf() {

allBooks=new ArrayList<>();

assert isValidBookshelf();

}

/\*\*

\* Creates a Bookshelf with the arrangement specified in pileOfBooks. Example

\* values: [20, 1, 9].

\*

\* PRE: pileOfBooks contains an array list of 0 or more positive numbers

\* representing the height of each book.

\*/

public Bookshelf(ArrayList<Integer> pileOfBooks) {

allBooks=new ArrayList<>(pileOfBooks);

assert isValidBookshelf();

}

/\*\*

\* Inserts book with specified height at the start of the Bookshelf, i.e., it

\* will end up at position 0.

\*

\* PRE: height > 0 (height of book is always positive)

\*/

public void addFront(int height) {

assert height>0;

allBooks.add(0,height);

assert isValidBookshelf();

}

/\*\*

\* Inserts book with specified height at the end of the Bookshelf.

\*

\* PRE: height > 0 (height of book is always positive)

\*/

public void addLast(int height) {

assert height>0;

allBooks.add(height);

assert isValidBookshelf();

}

/\*\*

\* Removes book at the start of the Bookshelf and returns the height of the

\* removed book.

\*

\* PRE: this.size() > 0 i.e. can be called only on non-empty BookShelf

\*/

public int removeFront() {

assert allBooks.size()>0;

removedHeight=allBooks.get(0);

allBooks.remove(0);

assert isValidBookshelf();

return removedHeight;

}

/\*\*

\* Removes book at the end of the Bookshelf and returns the height of the

\* removed book.

\*

\* PRE: this.size() > 0 i.e. can be called only on non-empty BookShelf

\*/

public int removeLast() {

assert allBooks.size()>0;

int lastIndex=allBooks.size()-1;

removedHeight=allBooks.get(lastIndex);

allBooks.remove(lastIndex);

assert isValidBookshelf();

return removedHeight;

}

/\*

\* Gets the height of the book at the given position.

\*

\* PRE: 0 <= position < this.size()

\*/

public int getHeight(int position) {

assert (position>=0 && position<allBooks.size());

assert isValidBookshelf();

return allBooks.get(position);

}

/\*\*

\* Returns number of books on the this Bookshelf.

\*/

public int size() {

assert isValidBookshelf();

return allBooks.size();

}

/\*\*

\* 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]”

\*/

public String toString() {

assert isValidBookshelf();

return allBooks.toString();

}

/\*\*

\* Returns true iff the books on this Bookshelf are in non-decreasing order.

\* (Note: this is an accessor; it does not change the bookshelf.)

\*/

public boolean isSorted() {

int numBooks=0;

boolean sorted=false;

for(int i=0;i<allBooks.size()-1;i++){

if(allBooks.get(i+1)>allBooks.get(i)){

numBooks++;

if((numBooks+1) == allBooks.size()){

sorted=true;

}

}

}

assert isValidBookshelf();

return sorted;

}

/\*\*

\* Returns true iff the Bookshelf data is in a valid state.

\* (See representation invariant comment for more details.)

\*/

private boolean isValidBookshelf() {

boolean valid=false;

if(allBooks.size()==0){

valid=true;

}

else{

int numBooks=0;

for(int i=0;i<allBooks.size();i++){

if(allBooks.get(i)>0){

numBooks++;

}

}

if(numBooks == allBooks.size()){

valid=true;

}

}

return valid;

}

}

第二个：

public class BookshelfKeeper {

/\*\*

Representation invariant:

<put rep. invar. comment here>

-- maintaining its bookshelf in non-decreasing order of heights before and after operations are completed.

\*/

// <add instance variables here>

private Bookshelf sortedBooks;

private int numPick=0;

private int numPut=0;

private int numTotal=0;

private Bookshelf temp;

private int numBooks=0;

private int midNum=0;

/\*\*

\* Creates a BookShelfKeeper object with an empty bookshelf

\*/

public BookshelfKeeper() {

sortedBooks=new Bookshelf();

assert isValidBookshelfKeeper();

}

/\*\*

\* Creates a BookshelfKeeper object initialized with the given sorted bookshelf.

\* Note: method does not make a defensive copy of the bookshelf.

\*

\* PRE: sortedBookshelf.isSorted() is true.

\*/

public BookshelfKeeper(Bookshelf sortedBookshelf) {

sortedBooks=sortedBookshelf;

temp=new Bookshelf();

numBooks=sortedBooks.size();

midNum=numBooks/2;

assert isValidBookshelfKeeper();

}

/\*\*

\* Removes a book from the specified position in the bookshelf and keeps bookshelf sorted

\* after picking up 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.

\*

\* PRE: 0 <= position < getNumBooks()

\*/

public int pickPos(int position) {

numPick=0;

if(position<midNum){

for(int i=0;i<position;i++){

temp.addFront(sortedBooks.getHeight(0));

sortedBooks.removeFront();

numPick++;

}

sortedBooks.removeFront();

numPick++;

for(int j=0;j<position;j++){

sortedBooks.addFront(temp.getHeight(j));

numPick++;

}

}

else{

for(int i=0;i<numBooks-position;i++){

temp.addFront(sortedBooks.getHeight(sortedBooks.size()-1));

sortedBooks.removeLast();

numPick++;

}

sortedBooks.removeLast();

numPick++;

for(int j=0;j<numBooks-position;j++){

sortedBooks.addLast(temp.getHeight(j));

numPick++;

}

}

assert isValidBookshelfKeeper();

return numPick;

}

/\*\*

\* 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.

\*

\* PRE: height > 0

\*/

public int putHeight(int height) {

numPut=0;

int countHeight=0; // count the number of height which is lower than a given height

for(int i=0;i<numBooks;i++){

if(sortedBooks.getHeight(i)<height){

countHeight++;

}

}

if(countHeight<midNum){

for(int j=0;j<countHeight;j++){

temp.addFront(sortedBooks.getHeight(0));

sortedBooks.removeFront();

numPut++;

}

sortedBooks.addFront(height);

numPut++;

for(int k=0;k<countHeight;k++){

sortedBooks.addFront(temp.getHeight(k));

numPut++;

}

}

else{

for(int j=0;j<countHeight;j++){

temp.addFront(sortedBooks.getHeight(sortedBooks.size()-1));

sortedBooks.removeLast();

numPut++;

}

sortedBooks.addLast(height);

numPut++;

for(int k=0;k<countHeight;k++){

sortedBooks.addLast(temp.getHeight(k));

numPut++;

}

}

assert isValidBookshelfKeeper();

return numPut;

}

/\*\*

\* 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 pick and put operations

\* that have been requested up to now.

\*/

public int getTotalOperations() {

numTotal+=(numPick+numPut);

assert isValidBookshelfKeeper();

return numTotal;

}

/\*\*

\* Returns the number of books on the contained bookshelf.

\*/

public int getNumBooks() {

assert isValidBookshelfKeeper();

return sortedBooks.size();

}

/\*\*

\* Returns string representation of this BookshelfKeeper. Returns a String containing 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 pick or put operation,

\* followed by the total number of such calls made since we created this BookshelfKeeper.

\*

\* Example return string showing required format: “[1, 3, 5, 7, 33] 4 10”

\*

\*/

public String toString() {

String numPickStr=String.valueOf(numPick);

String numPutStr=String.valueOf(numPut);

String numTotalStr=String.valueOf(numTotal);

String Str0=String.valueOf(0);

if(numPick>0){

assert isValidBookshelfKeeper();

return sortedBooks.toString()+" "+numPickStr+" "+numTotalStr;

}

if(numPut>0){

assert isValidBookshelfKeeper();

return sortedBooks.toString()+" "+numPutStr+" "+numTotalStr;

}

return sortedBooks.toString()+" "+Str0+" "+numTotalStr;

}

/\*\*

\* Returns true iff the BookshelfKeeper data is in a valid state.

\* (See representation invariant comment for details.)

\*/

private boolean isValidBookshelfKeeper() {

return sortedBooks.isSorted();

}

// add any other private methods here

}