Project 2: Determining your position in a "help queue"

- This assignment must be submitted by 11:45 PM, on Wednesday, November 8.
- Please submit a file called `project_2-`*uid*`.zip`, where *uid* is the unity id of the team leader. The zip archive should contain all `.java` files that implement your solution and *no other files*, with the exception of a README.txt file with compilation and execution instructions.
- It will count as 15% of your final grade.

Teamwork

As with the first project, work on this second project will be done in teams. You are required to work in teams of 3 or 4 members unless you have obtained special permission from the instructor. Preferably the teams created for the first project will also be those that work on the second project. Team composition may change from one project to the next and must change if the team for the first project did not have at least three members.

By 11:45 PM, November 1, you are required to register any *changes in team composition*. To do so, post a message with the names and unity id's of all team members and the team name of a new team on Piazza (you can reuse the name of an existing team if it's a matter of, for example, replacing a single member). If an existing team is disbanding, post a message to that effect. All posts of this kind should be posted to the `teams` folder. If your team is staying together, there is no need to register again.

The regrouping of teams should ideally be amicable – please try to make sure there is general agreement and that there are no hard feelings. Please talk to me, preferably as a group, if there were serious dysfunctions in your team during your work on Project 1. I would prefer that you stay together, but, if that's not desirable, I need to know why.

When working as a team on code and documentation, consider reviewing each other's code instead of merely assigning a different piece for each team member to create independently. I also strongly recommend pair programming – let me know if you're unfamiliar with this practice.

Academic Integrity

You ***are*** permitted to discuss your general approach, algorithms, and results with people who are not on your team – Piazza is an excellent forum for doing this. However, ***no*** actual code or documentation can be shared. This prohibition applies to code produced by another person in or for previous offerings of this course or another course. Any such sharing will be considered an academic integrity violation. You are welcome to use code in the textbook or another source, but using such code without citing your sources constitutes plagiarism. ***Cite your sources in comments.*** Do this for any classes/methods you didn't write entirely yourself (and don't modify the borrowed code – you need to practice good code reuse). You are also allowed to use classes and methods in the Java API if appropriate. In case of doubt about what is permitted, ask the instructor or check the Code of Student Conduct (see syllabus for the link)

Learning outcomes

In completing this project you will

1. demonstrate understanding of binary trees by implementing them in a nontrivial application;
2. be able to describe and implement the insert, retrieve, and delete operations and traversals of (not necessarily balanced) binary search trees;
3. (if you do extra credit) be able to describe and implement the insert, retrieve, and delete operations and traversals of a particular balanced search tree scheme (e.g., AVL or red-black).

Overview

As an administrator of the customer service division of a large internet service provider, you have set up a priority queue to handle the thousands of calls that come from customers every day, some reporting outages, others reporting internet performance degradation or poor voice quality on their lines. Every call is assigned a help ticket and a priority so that the technical support staff can handle calls efficiently. One service you would like to provide is to allow customers to query the system automatically to determine their position on the help queue, in other words, to determine how many other calls currently have higher priority than theirs.

Requirements for Implementing the Priority Queue.

I/O specs (come up with a way to simulate calls and requests for queue positions). Operations:

- insert a call with a given priority
- remove a call (using help ticket id)
- identify and remove highest priority call
- query about position in the queue (using help ticket id)

Actual input and output will be via the console (you can use redirection if you want to use input/output files; that is what we will do when grading). The input will be a sequence of lines that have one of the forms in the following table; it shows both the input format and what the corresponding output should look like. Your program should also ***echo the input*** immediately after each line to make it easier to decipher the output file. You may also find it useful to indent the non-echoed output.

input	**output**	**comments**
+ *priority*	`id =` *id*	adds a ticket with the given priority; id's are consecutive positive integers: the first + operation results in id 1, the second 2, etc.
− *id*	*priority,* `pos =` *position*	removes a ticket with the given id; priority (same as when the ticket was inserted) and current position in the queue; any valid integer can be used as a priority; note that there is a space between the − and the *id*
`*`	`id =` *id*, *priority*	removes the ticket with highest priority; the id and priority of the ticket with highest priority; as a side effect, the ticket is removed
`?` *id*	`pos =` *position*	current position in the queue of the ticket with the given id

So that position numbers are easy to distinguish from priorities, they will be preceded by `pos =`. *Assume that all the priorities are distinct.*[1] *Any attempt to insert a ticket with the same priority as one already in the queue should generate a warning. A similar warning should be generated if there is an attempt to remove a ticket that is not in the queue. Finally, there should be a warning if the queue is empty when a * command is issued. These warnings should read:*

`Warning:` `a ticket with priority` *p* `is already in the queue`
`Warning:` `there is no ticket with id =` *i* `in the queue`
`Warning:` `removal attempted when queue is empty`

The program should keep running until the input is exhausted, regardless of any anomalous conditions that may arise. Internal to your program, you should throw an exception as soon as the anomaly is discovered and catch it at the point where the warning message can be printed, i.e., in the part of the code that handles output or the main program (throw early, catch late). It is a good idea to have a single class, e.g., `Warning` that is used to handle all warning messages. This

[1] This is obviously not a realistic assumption but it greatly simplifies the implementation. Nodes in the search tree will have priorities as their keys and id's as their values. Actually, you could, in real life, add a small perturbation factor to each priority to guarantee uniqueness.

```
+ 10
+ 10
    id = 1
+ 100
+ 100
    id = 2
+ 50
+ 50
    id = 3
? 1
?  1
    pos = 3
- 2
- 2
    100, pos = 1
? 1
?  1
    pos = 2
? 3
?  3
    pos = 1
*
*
    id = 3, 50
? 1
?  1
    pos = 1
*
*
    id = 1, 10
```

Figure 1: An example that illustrates input and output of the program. Input commands are in bold face. Output is indented. Each command is echoed so that the output is easy to read.

will make it easy to make your messages match the ones given here and add new ones. The following additional warnings related to bad commands should be issued when appropriate. Following any of these warnings the program should skip the current command and read the next one.

`Warning: invalid command` *com*
`Warning: id` *id* `is not an integer`
`Warning: priority` *p* `is not an integer`

There are several other siutations that might call for warnings.

Get your program working first without worrying about the ***input*** *warnings. Correct handling of these warning situations will be only a small part of your grade. However, the queue warnings must be handled eventually.* The example in Fig. 1 illustrates how your program should behave on an example under normal circumstances. Eveything is shown as it would appear on the console except that output would not be indented in your program.

Implementation notes

The priority queue should be implemented as a binary search tree with priority – *not id* – as a key and an extra field for each node to keep track of the number of nodes in the subtree rooted at the node. The highest priority ticket will be the rightmost one in the binary-tree implementation of an ordered dictionary.

You will need to use a separate mapping scheme of some kind to associate a ticket id with a priority. Your choice must be efficient in the sense that all operations are $O(\lg n)$ in the worst case. The best choices are a TreeMap, a HashMap, or an ArrayList with binary search.

What makes this project interesting is the additional information you will need to maintain. To answer a position query you need to know how many have keys have higher priority than the one you are looking for. However, ***you are not allowed to solve this problem by traversing all the higher-priority keys***. That would take too much time. Your query must take time in $O(h)$, where h is the height of the tree.

One way to solve the problem is to store, for each node x in the tree, the number of descendants of x (including x itself). Recall the recursive algorithm for finding the successors of a given key in a binary search tree. This algorithm ran in time $O(h+m)$, where m is the number of successors. The m arose because the output was required to *list* all of the successor. All you need to do is count them. If you know $n(x)$ the number of nodes in the subtree $T(x)$ rooted at x, you can simply return $n(x)$ instead of listing all of the items in $T(x)$.

Suppose, in the example tree of Fig 2, you want to find out how many tickets have higher priority than the one with priority 15 (not inclusive). The recursive function that *lists* successors would do the following:

$$\begin{aligned}\textsc{successors}(a,15) &= \textsc{successors}(b,15) + a + \textsc{inOrder}(e)\\ &= \textsc{inOrder}(d,15) + a + \textsc{inOrder}(e)\\ &= [d, \quad a, \quad e,g,f]\end{aligned}$$

Since we have stored the *number* of nodes in each subtree, we can replace the INORDER calls with the appropriate numbers and simply add, ending up with:

1 (for $n(d)$) + 1 (for a) + 3 (for $n(e)$).

We don't list or count b because it has equal priority.

Whenever you do an insertion or removal you will need to update the number of descendants appropriately along the path where the insertion/removal took place. Because you don't know on your way down whether the insertion or removal will be successful, the modification will have to be done on the way back up the tree. Recursion is a nice way to do this.

Consider what should happen to the tree in Fig 2 when an item with priority 17 and id 8 is inserted.

$\Longrightarrow$ INSERT$((17[8]), a)$
  $\Longrightarrow$ INSERT$((17[8]), b)$
    $\Longrightarrow$ INSERT$((17[8]), d)$
      $\Longrightarrow$ INSERT$((17[8]), \textbf{\textit{null}})$

      $\Longleftarrow$ INSERT – create node $h = (17[8])$ with $n(h) = 1$
    $\Longleftarrow$ INSERT – $n(d) = n(d) + 1 = 2$
  $\Longleftarrow$ INSERT – $n(b) = n(b) + 1 = 4$
$\Longleftarrow$ INSERT – $n(a) = n(a) + 1 = 8$

A similar process takes place when a node is removed, except that now we decrement instead of increment counts.

Finally, any insertion or removal has to update the map that maps id's to priorities. Fig 3 illustrates the overall work flow of a single command. Each command is parsed. If it involves an id, the id is looked up in the map, and its corresponding priority is used to find the appropriate node in the

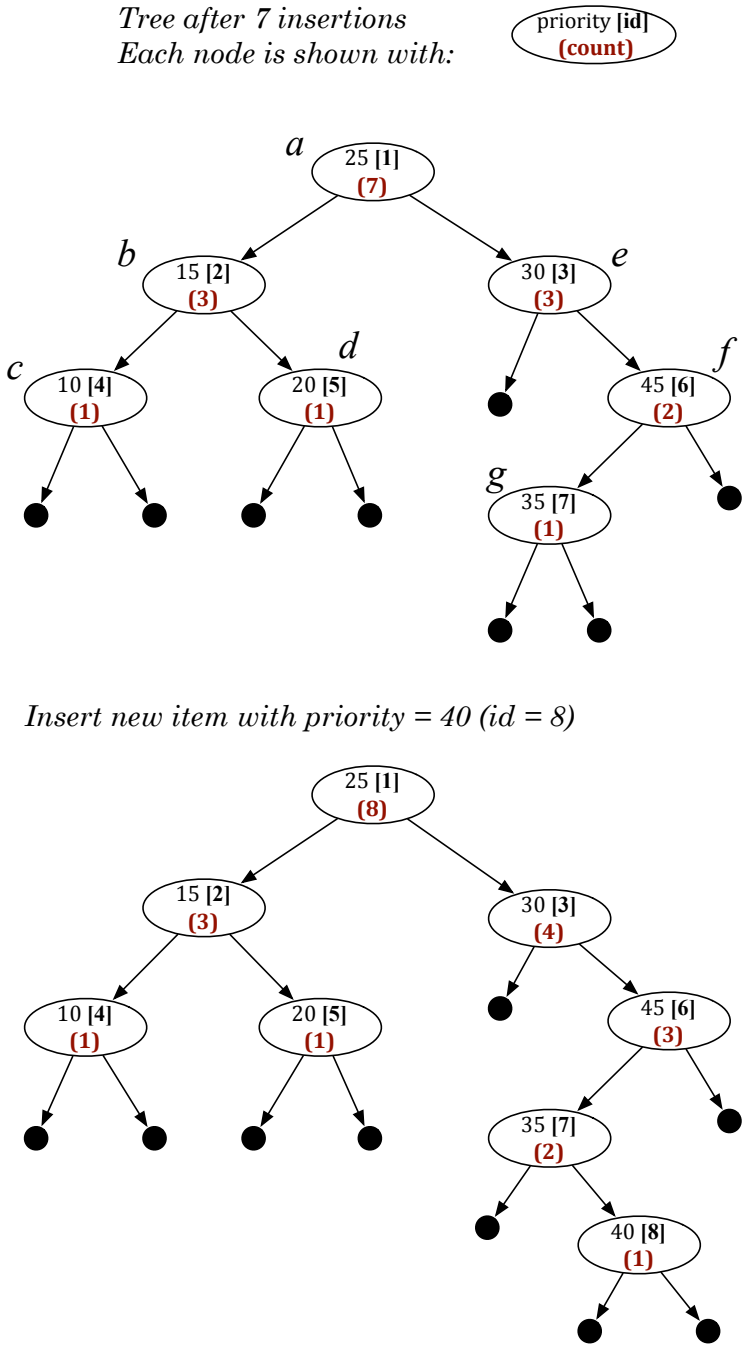

Figure 2: An insertion into a binary tree used to implement a priority queue that allows position queries.

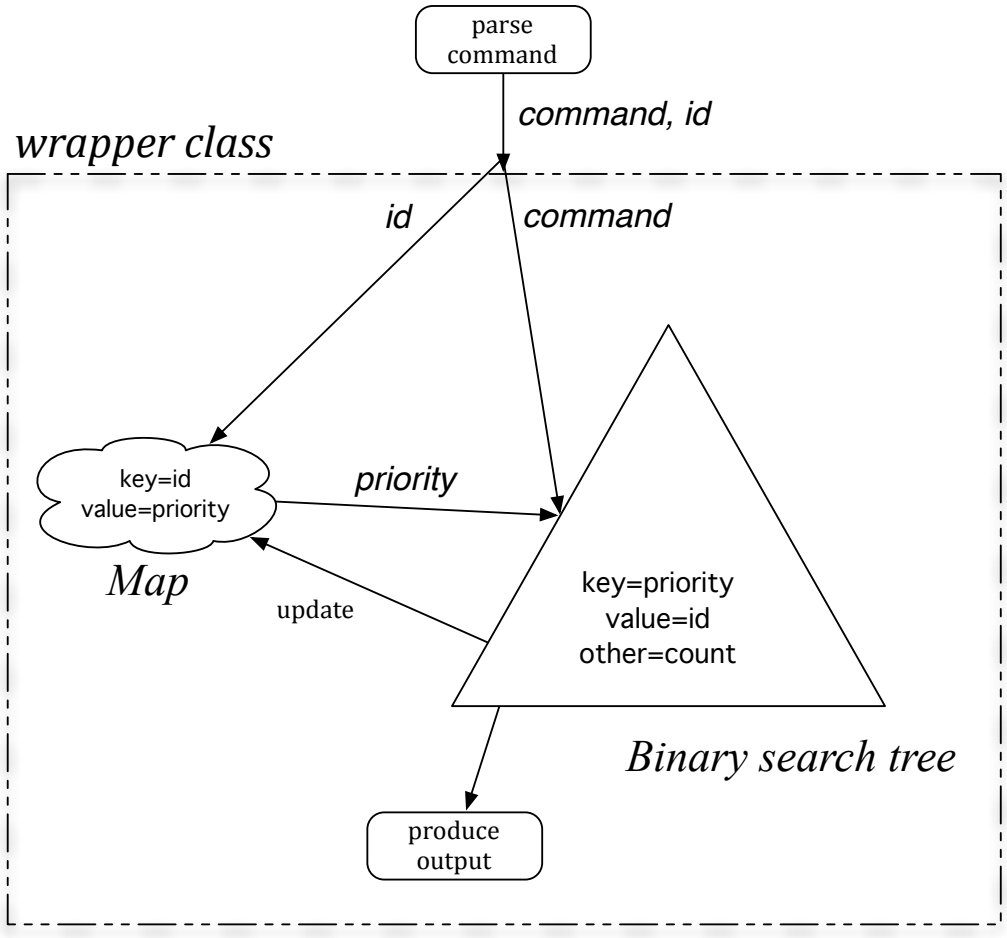

Figure 3: Control and data flow for an individual help queue command.

tree. The command determines the operation(s) to be performed on the tree. The results of these will be used to produce output and, if appropriate, modify $n(x)$ for the affected nodes. Also, if there is an insertion or deletion in the tree, the map needs to be updated. You should create a wrapper class that hides the workflow from the main program. The main program should only parse commands, call methods corresponding to each command — methods of the wrapper class — and catch exceptions, passing these to the `Warning` class.

Files provided

The archive `Project-2-Files.zip` contains the following.

- `HelpTickets.jar` – a jar file with a working solution (that uses an array instead of a binary tree); includes the command `p` for printing the queue; you can do the same in your solution.[2]
- `TreePrint.java` – a template for printing a binary tree in inorder; uses `IndentPrinter.java`, also included.
- files containing examples that do not produce any warnings:

[2]Your printout will, of course, differ from the printout of the array. When debugging, you will want to show the items sorted by priority and indented according to the tree structure – you could use `TreePrint.java` (possibly modified) to do this.

1. `example-in.txt` and `example-out.txt` – input and output for the example in Fig. 1.
2. `0100-example.txt` and `0100-example-out.txt` – a bigger example with the expected output.

- three other small examples that you can use as templates for further debugging:
  1. `no-removes.txt` – one that does no removes,
  2. `no-errors.txt` – one that has no errors,
  3. `one-error.txt` – one that has one error.

What to submit

You should submit a `zip` archive – *no `tar`, `jar` or `rar` archives, please* – containing several separate java source files that will allow us to do the following on a Linux command line (the `$` is the Linux command line prompt).

```
$ javac HelpTickets.java
$ java HelpTickets < test_input_1 > your_output_1
...
```

The first line above could be replaced with `$ javac *.java` — in either case, all of your source code needs to be in the same directory, the directory that opens when we unzip your archive. We will run your program on a VCL Linux platform or similar environment using Java version 8 — `openjdk version "1.8.0"` to be technically correct, but you should be safe if your program compiles and runs (on the command line) using Java 6 or more recent on any Unix-based platform, e.g., the terminal window of a Mac.

Obviously your main program will be in `HelpTickets.java`, but you are strongly encouraged to encapsulate and separately test whatever you can in other classes.

Extra credit

The assignment allows you to use an ordinary binary search tree. The problem becomes more complicated if you want to ensure that every operation is $O(\lg n)$ *in the worst case.* This requires using a balanced tree of some kind: red/black or AVL. You can earn as much as 20 points extra credit for a working implementation that uses a balanced tree. If you want to go for extra credit, you will submit additional files, including a file called `BalancedHelpTickets.java`, that we can compile and execute. It should look exactly like `HelpTickets.java`, except that the part that instantiates a binary search tree will create a balanced one. *Only one line should be different* when we do a diff between `HelpTickets.java` and `BalancedHelpTickets.java` – the remaining differences should be in the form of additional source files to implement the balanced tree.

Grading

category	**points**
Compiles and executes on simple examples involving three insertions and three position queries.	20
Executes correctly on the example given in Fig 1.	20
Executes correctly on boundary cases, including ones that should generate warnings.	20
Executes correctly and *efficiently* on large test cases specifically designed to generate balanced trees when ordinary binary search trees are used. These examples will be large enough to distinguish between implementations that are $O(h)$ and those that are $\Omega(n)$.	20
Is well organized, uses multiple well-designed other classes implementing abstract data types.	10
Is clearly documented, has useful javadoc comments.	10
Other team members give positive evaluations.	10
Total	**110**

100 points is a perfect score. The remaining 10 points are "built-in" extra credit.