Assignment 3: Forest for the Trees

Pre-Assignment Notes

The **late penalty** is -10% per day late. After two days, late submissions won't be accepted.

This is an **individual assignment**, meaning you're not supposed to share or copy your solution/code from other students or the internet. There's still the class's Piazza if you want to ask questions to the group, or you can ask during lecture, send an email to the class's help account, or attend an office hour on Discord.

Note you can use any code shared by the instructor or anything from class or the tutorial videos linked on the course website. You *can* get outside help from guides and other people, just try to make sure they're teaching you how to solve the assignment, not writing the code for you. You won't learn anything that way, and they can't write your exam or midterm for you. If in doubt about whether something is plagiarism or not, don't be afraid to reach out and ask!

1. Non-Linear and Non-Positional

Lately, we have begun exploring data structures that expand our understanding of how data can be organized. Unspoken assumptions in the more fundamental classes, like Stacks, Queues, Arrays, and Lists, have been overturned. This allows us to create new, more sophisticated structures and algorithms with different properties and advantages, but it also creates new challenges for recognizing where they apply and how to implement them.

One new direction we have explored is non-linear data structures, which do not keep their data in a linear sequence. The main example of this we have considered is the Tree, which replaces the idea of a node having links to next and previous nodes with a parent node instead having a number of children, each of which may have children of their own.

Then there are key-based data structures, rather than position-based ones. Instead of accessing a data entry based on its indexed position, as with an array, or its position relative to a known head or tail, as with a list, a key-based structure like a Priority Queue sorts entries according to their key. A key might be the same as the entry's data value, or it could be a variable that exists only to be that entry's key, or it could be derived from several things. Whatever it is, it must be consistent, and must allow for comparisons with other keys to establish an order for their attached entries – comparing Strings according to alphabetical order, for example.

From the outside, many programs that use these data structures may not appear very different than their linear or position-based alternatives. You can still add and retrieve elements of data, analyze their run-time efficiency, and build hierarchies of classes and functions that implement them. The inner workings of those implementations, however, look quite different, and that's what we're here to practice today!

2. Assignment Outline

Imagine you've just joined a company working on software to support forest management – real forest management, mind you, planting actual made-of-wood trees. The forestry service has hired your company to build a system to help track and assign the various tasks their workers need to complete each day.

You've been given three of the system's related data structures to complete. There will be other teammates at your company working on things like the interface, generating data, figuring out the client's requirements, etc, all of which will add additional constraints for each part of the system, so read each question carefully. Remember, the three parts must be stitched together into a single project by the end!

As a general matter, you are free to use any of the standard Java classes or interfaces that we have covered so far in answering these questions, as well as any that fall broadly within the same material that weren't explicitly covered (say you want to do something with an Iterator, for example). The single exception is the standard Java PriorityQueue class, which I am ruling out mainly to save you from discovering later that you'll have an easier time building your own version than trying to adapt the standard one.

Also, don't forget to include a reasonable amount of error and exception handling! I won't be listing every error for you (and obviously error-handling could be taken to some absurd extreme), but some amount of basic error handling will be factored into code quality going forward.

2.1 Branching Out

The first thing the forestry service wants is to build a collection of tasks that their forestry workers complete each day. The service interviewed the workers, each of whom described the tasks they completed on a given day, attaching an importance to the task from 1 to 10. Your teammates have looked at these daily task reports and decided to organize them into a rooted tree, where each path from the tree's root node (a task labelled "Open up in the morning" with a special importance of 11) follows a worker through their tasks, with the leaf node being the last task they completed that day. An example of how to visualize such a tree is given below:

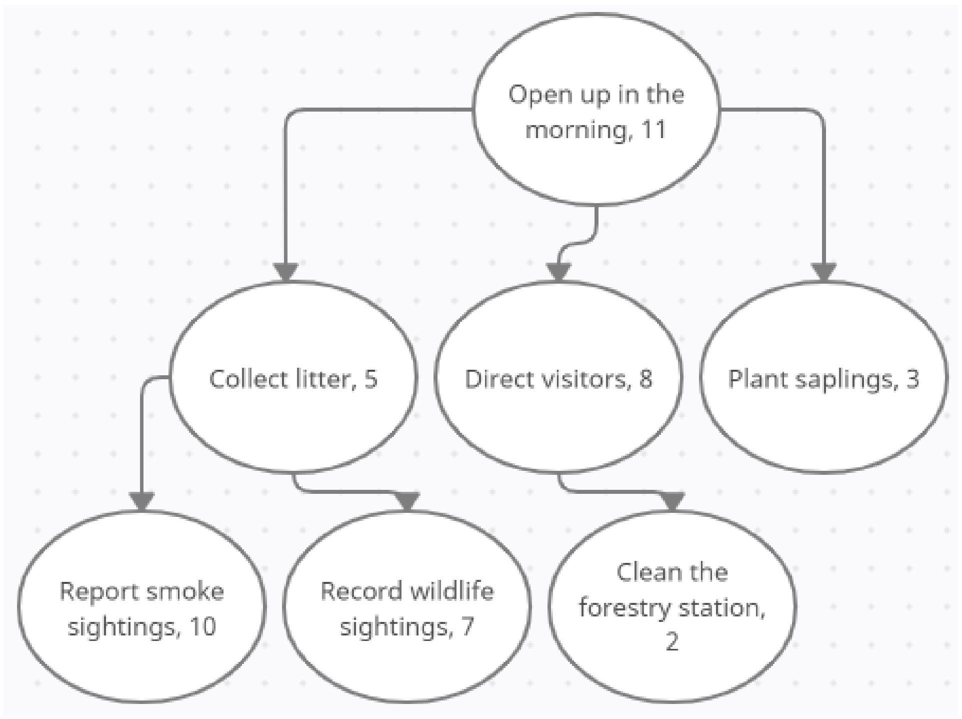
Open up in the morning, 11
Collect litter, 5
Direct visitors, 8
Plant saplings, 3
Report smoke sightings, 10
Record wildlife sightings, 7
Clean the forestry station, 2

Figure 1. An example tree made from the task reports of the forestry service workers

You must implement a Tree-based data structure that can store this collection. This Tree must be made up of nodes with (at least) a parent, children, a String describing the task, and an integer ranking the importance of that task.

Your task-collection Tree must include attaching and removing nodes, as well as the general supporting Tree methods like reporting the number of nodes and whether it's empty or not. There has also been a special request from someone else on the team for a function that prints to the terminal how many tasks of each importance level are currently in the Tree (for levels 1 to 10, only the opening task should have an 11). For obscure reasons, they've also asked to be sure that this function runs in constant time.

You don't have to worry about interface issues, like how exactly people will be finding the nodes they want removed or how to pick which existing node a new node should be attached to – these functions can simply receive a reference to a particular node in the tree as a given, and go from there. As for how data is read into the system or displayed, we'll leave that for 2.4, though you're free to come back later and add or modify functions to make that easier.

(*An important note about the tree structure and causality*: the way we've described this tree you're building presents each path from root to leaf as a set of related decisions – you can only get to “Report Smoke Sightings” in Figure 1, for example, by going to “Collect Litter” first. These relationships are important to preserve during this question, but they can be broken in the next two, meaning you will be able to do something like present the “Report Smoke Sightings” task first for having the highest importance without first presenting “Collect Litter”, they will not have a causal relationship after this point. I think this could be inferred from the scenario I'm laying out, I just want to make sure no one misses this point.)

2.2 Top Priority

Now that you can store a collection of tasks derived from task reports, the forestry service would like to build another structure which can organize and present these tasks to workers according to their importance. The idea is to produce an orderly queue of tasks that need to be done around the forest each day, and any worker can simply retrieve the next task from the queue and know it's the highest priority job yet to be completed.

Your team has requested that you build another data structure and supporting functions, this one built as a List-based Priority Queue. In particular, they'd like this system to be able to use one of the Trees you built in the previous question to construct the PQ – it should be able to take either your Tree class itself or the root node of your Tree and convert it into the underlying List.

Each entry-node in the Priority Queue should include the String task description as a value and integer importance ranking as a key. You should include functions which allow for adding a new entry as well as removing (or simply returning without removing) the next-highest-importance entry. You are also free to include any other supporting classes or functions you find you need, although these will also be assessed for their quality and performance.

As for the implementation details, your teammates have extracted some requirements that might help inform your design decisions. When it comes to efficiency, the management of the forestry service plan to review what tasks need doing and rebuild the queue each night, a process which can take as long as it needs to since it can be left to run overnight if need be. Forestry workers, however, want the queue to dispense their tasks as fast as possible, since there might be a lot of workers trying to access the system at once and the highest-priority tasks should be started ASAP!

2.3 Heaps of Trouble

While the priority-task system you built in 2.2 makes sense for managers coming up with tasks for the workers to complete the next day, some tasks may come up during the day itself. As such, the forestry service has requested a different priority queue for tasks that workers can add to and remove from themselves during the day, with slightly different requirements that lead to some major implementation changes.

The big change is that since the system must now balance workers both adding and removing tasks throughout the day, it's no longer okay to base the Priority Queue on an underlying data structure that favors one method over the other in terms of efficiency. As such, your teammates have asked you to create an alternative version of the Priority Queue from 2.2 based on a Heap.

This worker-driven PQ should have the same functions as the previous one. The Heap, however, will need to be implemented as well, including the Heap-sorting method that ensures the highest-priority entry remains at the root.

This Priority Queue must also include a constructor that takes in a task-collection Tree, as the previous one did, but now must convert it to a Heap rather than a List. Remember my note in 2.1 – while a Heap may also be a Tree, it will not need to preserve any causal relationships implied by the previous Tree structure. It need only be organized according to the importance keys.

2.4 Demo Day

Once you're done the three structures described above, write a Main method to serve as the test method that will not only fully demonstrate your code for your forestry service clients (and the TAs), but also offer some simple interactivity.

All input and output should be handled through the terminal, printing text straight to the screen and receiving typed text from the user. You should start by defining some tasks and ranking their importance (being sure to keep the user in the loop about what you're doing), then using them to build a task-collection Tree. The finished Tree should then print itself to the screen – within the reasonable limits of an all-text display format, of course (you may choose to simply print each node's contents and mention their parent, for example). Don't forget to also demonstrate the function that prints how many tasks there are at each importance level.

You should then use this Tree to instantiate both kinds of Priority Queue you developed. Both PQs should first print their contents in the order that they're stored (you may reuse whatever text-friendly tree-printing format you settled on for the task-collection Tree for the Heap, if you wish), and then print their ordered values as they remove entries until entirely emptying out.

At this point, the user should be prompted to add tasks of their own. They should then be able to type in new tasks and rate their importance, one after another. The user should also be able to signal (perhaps with an empty String) that they would instead like have the next highest-priority task printed out and removed, which both Priority Queues should then do.

Exactly how the user is prompted, how input is read in, and how the results are printed are left for you to define.

3. Deliverables

All assignment submissions will be done through CourSys at https://courses.cs.sfu.ca/. For this assignment, that should include:

1. A .zip archive of your code, organized into a single Java project. To ensure compatibility, it's

recommended to use IntelliJ while developing your code, then using the export function under File -> Export -> Project To .zip File.

Be sure you tidy up your code before submitting! Check the Java code style guide posted on the course website, and do your best to provide helpful comments.