Project 5: Queue

A Queue Abstract Data Type.

Overview

In this project, you'll exercise your understanding of queues and recursion. You'll implement an unbounded queue and use your queue implementation to list the entries in a directory tree. You'll also implement the merge sorting algorithm using queues. This is the first time that we will provide very few public tests and you have to develop the tests yourself to ensure your programs work correctly.

Learning Goals

- Implement a generic unbounded queue.
- Show understanding of algorithms that use a queue and a Java API class to walk through a directory tree.
- Implement a sorting algorithm based upon queues and recursion.
- Practice writing tests yourself to ensure the programs work correctly.

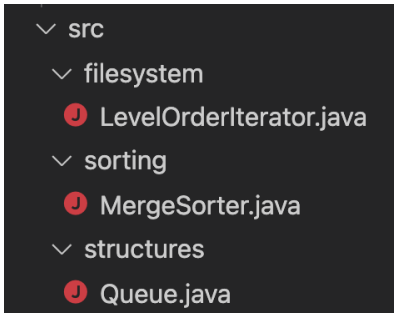
- Gain practice writing appropriate unit tests.

Project Specification:

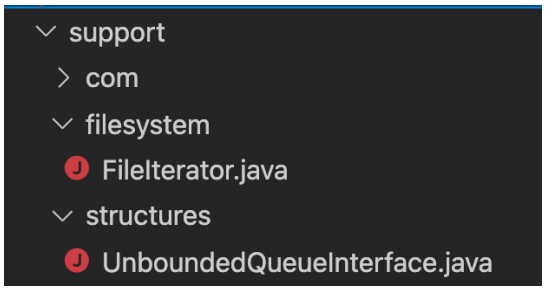
Project Code

The starter code contains the following files.

src: This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified), you can add new files, etc.

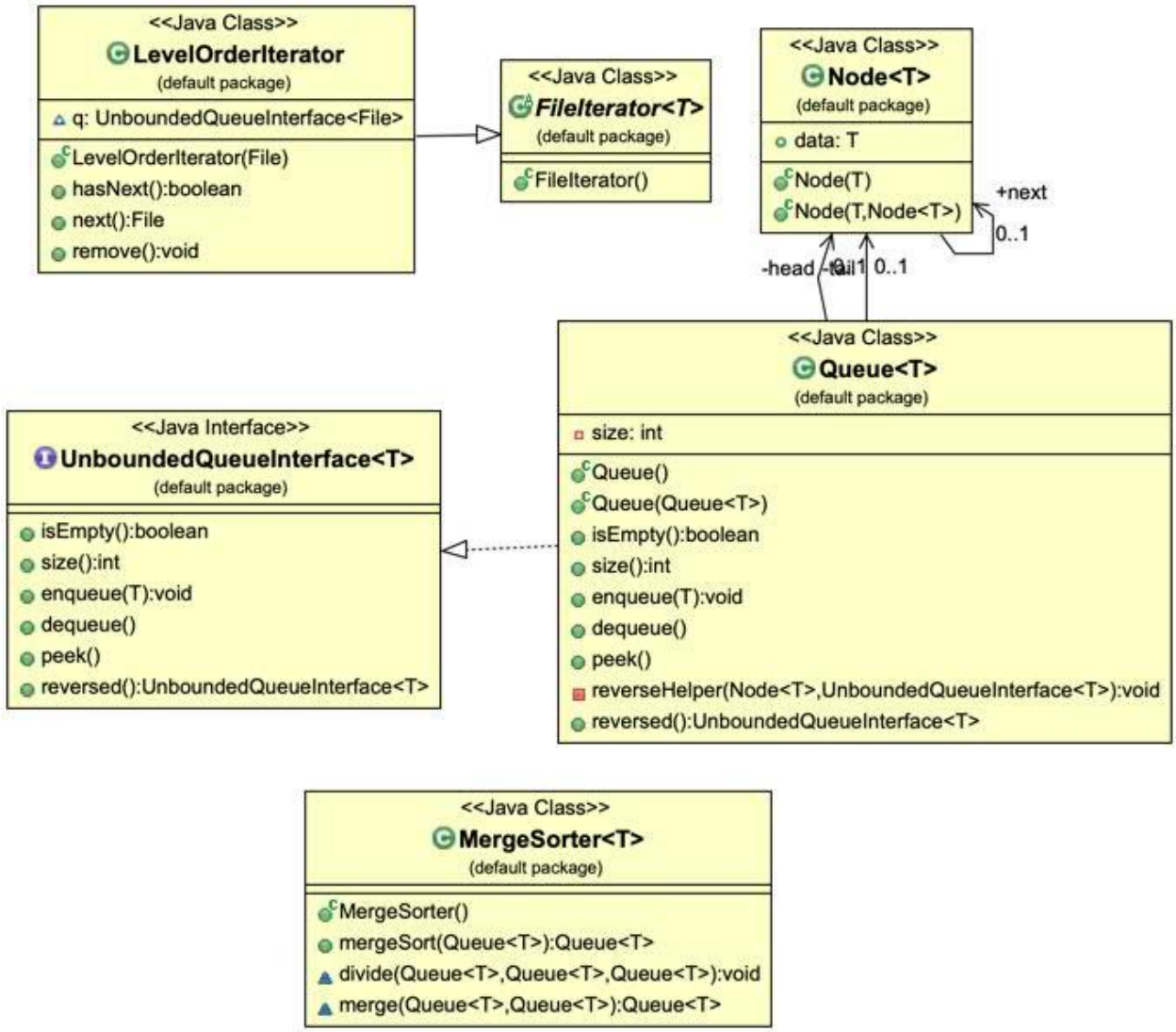


support: This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder.



test: Feel free to add your own tests here.

Project UML diagram



Development Notes

There are 3 specific problems for you to solve.

1. Implement a generic queue data structure.
2. Use the queue to iterate through files.
3. Use the queue to implement a merge sort algorithm.

Requirements. For all three problems in this project, you may NOT use any Java class that implements the Collection interface (e.g. ArrayList, Vector etc.). The list of these classes is included in the API documentation. Notably, do not submit code where you import or use any class from the java.util package.

Note that normally each Java file contains only one public class. However, you can define multiple non-public classes (or nested classes) in the same Java file, keeping in mind that such classes are not visible (public) to outside classes.

Problem 1: Implement a generic queue data structure

Complete the Queue (defined in `Queue.java`) by correctly implementing the six methods marked by *//TODO* as well as two constructors, described below.

You may implement the queue using either an array or a linked list, as long as it obeys the contract specified in `UnboundedQueueInterface`. Please carefully read the comments in this class to understand what each method does, what type of Exception each method is expected to throw, and importantly, several operations must be *O*(1).

If you use a linked list, a generic Node<T> structure is already defined at the end of `Queue.java`.

If you use an array, you must make it expandable so that its size is unbounded. The reversed() method should return a new copy of the Queue with the elements in reverse order, but must not modify the current Queue. Depending on how you store the queue elements, there are several implementation choices. You may choose an array to store the queue elements or if you use a linked list to store the elements, you can use recursion to print the reversed queue.

In addition to the usual no-argument constructor, your implementation of Queue must also include a *copy constructor*. See the In-class lecture notes for more information about the copy constructor.

Problem 2 Use the queue to iterate through files

Next, in **`src/filesystem/LevelOrderIterator.java`,** you will use your Queue to implement an Iterator over Files that will traverse (or *walk*) a filesystem tree in *level order (*explanation below*)*.

Filesystem explanation:

Your computer stores files and directories on disk. The filesystem is a data structure that organizes two types of nodes. Each node in the filesystem is either a file or a directory; and directories contain zero or more other nodes. If you drew out the content of a particular node, it would fan out in a tree-like structure. On OS X, this can be seen explicitly in the Finder as:

In the screenshot above, we're examining the tree rooted at the src directory. It contains three elements, filesystem, sorting, and structures. They happen to be all directories, but in practice they could be a mix of directories and files.

What to do.

implement `next()` and `hasNext()` in `LevelOrderIterator.java`. Your code should systematically iterate through the nodes in a filesystem tree, rooted at a given node. First, it should visit the root node. Then, it should visit each of that node's *children* (that is, each node directly connected to the root node). Then it should visit each of those node's children, and so on. This approach will visit the root node, then all of the nodes one level below it, then all of the nodes one level below those nodes. This is called a ***level-order traversal*** of a given tree within the filesystem.

Java represents filesystem nodes using class `java.io.File`. You'll want to skim through the API documentation for File. Pay attention to the constructor and the `exists()`, `isDirectory()`, and `listFiles()` methods.

To ensure you traverse the tree in level order, your code will need to visit the nodes in first-in-first-out order, adding children of the current node to a queue of nodes waiting to be visited in order. (Use the Queue you created in Problem 1; DO NOT use `java.util.Queue`.) When you obtain the children of a node, sort them in lexicography order before enqueueing them. The sorting can be done using Java's **`Arrays.sort`** method. The File class already defines a `compareTo` method, which `Arrays.sort` relies on.

A correct level-order traversal of the example tree above, rooted at src, is in the following order:

- `src`
- `src/filesystem`
- `src/sorting`
- `src/structures`
- `src/filesystem/LevelOrderIterator.java`
- `src/sorting/MergeSorter.java`
- `src/structures/Queue.java`

On OS X, the *path separator* is a forward slash (/); on Windows it is a backslash (\). Our tests will account for the difference automatically. Please do not attempt to write code that translates path separators based upon the host platform. Use the `listFiles()` method of File to obtain the contents of directories, and Java will correctly create the File objects representing the contents.

Do not attempt to define methods in `LevelOrderIterator` recursively. Doing so will result in a depth-first rather than breadth-first traversal of the filesystem (as would using a Stack rather than a Queue).

Problem 3 Use the queue to implement a merge sort algorithm

For this problem, you will complete the code in `src/sorting/MergeSorter.java`, where you will use your Queue to implement the merge sort algorithm.

Merge sort is a popular algorithm for sorting data elements. We'll consider a queue as sorted when the elements are in ascending order, that is, when the front of the queue contains the smallest element, the rear of the queue contains the largest, and every element in between also falls in order.

MergeSorter contains several methods that you must correctly implement. In particular, the `mergeSort` method is a recursive method. DO NOT use `Arrays.sort()`. Be sure you understand the base and recursive cases for the `mergeSort` method.

Hints:

- When comparing two generic type elements e1 and e2, you can't directly use e1 <= e2, since the elements are objects, not primitive types. Instead, use `e1.compareTo(e2)`. If its result is less than or equal to zero, e1 is less than or equal to e2.
- You need not declare any class variables (also called instance variables); there is no need to share state between these methods except through their argument lists and return values. Using class variables may lead to hard-to-track-down errors.

Tests

We have provided you with some public tests and some private tests in Gradescope. You will need to come up with some of your own. Try to think of all of the edge cases that could occur, and write tests to check for each of them. You will not be graded on your tests, but writing them and passing them is the only way that you can be reasonably sure that your code works.
Remember: Do not modify any existing instance variables, method signatures or any of the starter code provided. Do not import any code libraries.

Style Guide

Follow these style guidelines that will be manually graded in the project.

1. Remove all unnecessary lines of code.
2. Use proper indenting.
3. Use optimal blank lines and spaces for operators.
4. Use camelCase for variable/parameter and method names.
5. Use descriptive variable and method names.
6. Reuse existing methods to reduce duplicate code.

Export and Submit

Step 1: Export your project

Step 1: Export your project
Within VSCode click on the "*View* > *Command Palette*…" menu option. Then type into the

Command Palette: "*Archive Folder*" and hit enter. This will produce a Zip file of your project folder. You can then upload that zip file to the corresponding project assignment in Gradescope. You can add the Archive extension to VSCode if you don't have it.

Step 2: Submit the zip file to Gradescope
Log into Gradescope, select the assignment, and submit the zip file for grading.

The autograder will run successfully only if you submit a **correctly formatted zip file**. The autograder will also not run if your code **does not compile**, or if you i**mport libraries** that were not specifically allowed in the instructions.

Remember, you can re-submit the assignment to gradescope as many times as you want, until the deadline. If it turns out you missed something and your code doesn't pass 100% of the tests, you can keep working until it does.

Academic Honesty Policy Reminder

Copying partial or whole solutions, obtained from other students or elsewhere, is **academic dishonesty**. Do not share your code with your classmates, and do not use your classmates' code. Posting solutions to the project on public sites is not allowed. If you are confused about what constitutes academic dishonesty, you should re-read the course policies.