CMPSCI 187 / Spring 2015
Queues

Due on Friday, March 27, 2015, 8:30 a.m.

Marc Liberatore and John Ridgway

Morrill I N375
Section 01 @ 10:00
Section 02 @ 08:30

Contents

Overview

In this project, you'll exercise your understanding of queues and recursion. You'll implement a specification for an unbounded queue, using unit tests you develop yourself. You'll then use your queue implementation to list the entries in a directory tree. Finally, you'll implement the merge sorting algorithm using a recursive approach.

Learning Goals

- Demonstrate understanding of the implementation details of a generic unbounded queue.
- Show understanding of algorithm that uses a queue and a Java API class to walk a directory tree.
- Implement a sorting algorithm based upon queues and recursion.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then email cs187help@cs.umass.edu immediately.

Start this assignment as soon as possible. Do not wait until 5pm the night before the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

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.

You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment.

Policies

- For some assignments, it will useful for you to write additional class files. Any class file you write that is used by your solution MUST be in the provided `src` directory you export.
- The TAs and instructors are here to help you figure out errors, but we won't do so for you after you submit your solution. When you submit your solution, **be sure to remove all compilation errors from your project**. Any compilation errors in your project will cause the autograder to fail, and you will receive a zero for your submission.

Test Files

In the `test` directory, we provide several JUnit test cases that will help you keep on track while completing the assignment. We recommend you run the tests often and use them to help create a checklist of things to do next. But you should be aware that we deliberately don't provide you the full test suite we use when grading.

We recommend that you think about possible cases and add new `@Test` cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods taking integers as arguments handle positives, negatives, and zeroes (when those values are valid as input)?
- Does your code handle unusual cases, such as empty or maximally-sized data structures?

More complex tests will be assignment-specific. To build good test cases, think about ways to exercise methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. Note that we will not be looking at your test cases, they are just for your use.

Before submitting, make sure that your program compiles with and passes all of the original tests. If you have errors in these files, it means the structure of the files found in the `src` directory have been altered in a way that will cause your submission to lose some (or all) points.

Import Project into Eclipse

Begin by downloading the starter project and importing it into your workspace. It is very important that you **do not rename** this project as its name is used during the autograding process. If the project is renamed, your assignment will not be graded, and you will receive a zero.

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we may provide JUnit tests for classes that do not yet exist in your code. You can still run the other JUnit tests.

The project should normally contain the following root items:

`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 in the problem description and in the code we provide), 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. To help ensure that, we suggest that you set the support folder to be read-only. You can do this by right-clicking on it in the package explorer, choosing Properties from the menu, choosing Resource from the list on the left of the pop-up Properties window, unchecking the Permissions check-box for Owner-Write, and clicking the OK button. A dialog box will show with the title "Confirm recursive changes", and you should click on the "Yes" button.

`test` The test folder where all of the public unit tests are available.

`JUnit 4` A library that is used to run the test programs.

`JRE System Library` This is what allows Java to run; it is the location of the Java System Libraries.

If you are missing any of the above or if errors are present in the project (other than as specifically described below), seek help immediately so you can get started on the project right away.

Note: You may not use any of the classes from the Java Platform API that implement the Collection interface (or its sub-interfaces, etc.). The list of these classes is included in the API documentation, available at: http:

//docs.oracle.com/javase/7/docs/api/java/util/Collection.html. Doing so will be viewed as an attempt to cheat.

Notably, **do not submit an assignment where you import and use `java.util.Queue` in any file, or you will receive a zero.**

Problem 1

To begin, you will implement the `UnboundedQueueInterface` in `Queue.java`.

You are welcome to use code you've written for previous assignments, or code that we've provided. Make sure you place any such code under the `src/` directory, or your program will not compile when autograded.

What to do

For full credit, implement a queue by correctly implementing the six methods defined in `UnboundedQueueInterface` as well as two constructors, described below. You may implement the queue using either an array or a linked list, so long as it obeys the contract specified in `UnboundedQueueInterface`. In particular, several operations must be $O(1)$.

The `reversed()` method should return a copy of the `Queue` with the elements in reverse order, but must not change the current `Queue`. There are at least three valid approaches you can use, and you can use any of them. You could explicitly use a stack. You could use recursion. Or you could iterate through and directly manipulate the contents of the array or the `next` pointers in the linked list backing a `Queue`.

In addition to the usual no-argument constructor, your implementation of `Queue` must also include a *copy constructor*. A copy constructor takes a single argument, with a type of the class itself, and returns a *copy* of the object. In this case, we want you to create what is called a *shallow copy* of the `Queue`.

A shallow copy will contain a copy of the data structure (the array or the nodes of the linked list) used to hold the elements in the queue. The array (or nodes in the linked list) of the copy will not be identical (==) to the original. But the values stored in the array (or the `info` values inside the nodes in the linked list) will be identical. Put another way, the copy constructor should copy the references to the elements but not the elements themselves — the elements will be aliased between the original and the copy, but the data structure (the array or the nodes in the linked list) should not be aliased.

Why are there so few tests?

When you get a job (or take upper-level CMPSCI classes), it's unlikely that comprehensive and correct test suites will appear fully-formed from the ether (or on the hard disk of your computer). Instead, you'll need to develop your own tests, or suffer the consequences of ad hoc debugging. Since you've already implemented several ADTs similar to a queue, we're letting you write most of your own test suite for your `Queue` – we've provided test for the copy constructor, but they depend upon correct functionality of other methods we don't test. **You should write your own tests for the other methods of `UnboundedQueueInterface`.**

Problem 1 continued on next page...

You do not have to do so. We are not grading your test suite, only the correctness of your `Queue`'s implementation of `UnboundedQueueInterface`. But unless you are supremely and correctly confident in your abilities as a coder,[1] you will make mistakes. And isolating these mistakes is much faster with a repeatable test you can just click "Run" on, rather than using an ad hoc driver or the like.

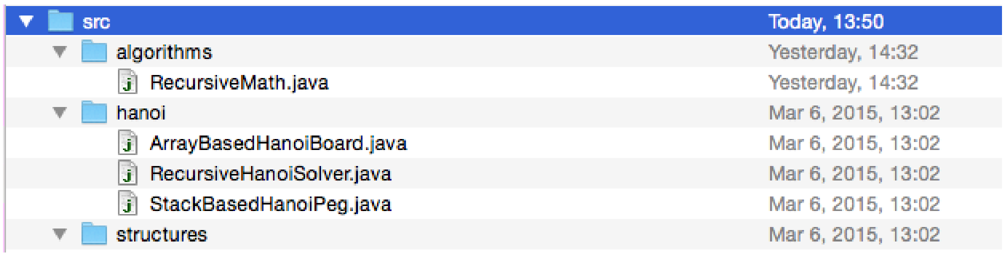
Presumably you've already been using the tests in the previous assignments to drive your coding — now take the next step and write those tests yourself. What should your tests do? We suggest you start by looking at the tests you've used previously for, say, the various stack implementations, and adapting them to a queue.

Problem 2

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*.

The filesystem in thirty seconds

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



```
▼ src                               Today, 13:50
    ▼ algorithms                    Yesterday, 14:32
          RecursiveMath.java        Yesterday, 14:32
    ▼ hanoi                         Mar 6, 2015, 13:02
          ArrayBasedHanoiBoard.java Mar 6, 2015, 13:02
          RecursiveHanoiSolver.java Mar 6, 2015, 13:02
          StackBasedHanoiPeg.java   Mar 6, 2015, 13:02
    ▼ structures                    Mar 6, 2015, 13:02
```

Given a particular element in the filesystem, we can talk about examining the tree that is *rooted at* (starts at) that node. In the screenshot above, we're examining the tree rooted at the `src` directory. It contains three elements, `algorithms`, `hanoi`, and `structures`. `algorithms` and `hanoi` each contain yet more items, while `structures` is empty.

What to do

For full credit, correctly 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

[1]Hint: You probably shouldn't be. Marc has been writing Java for over ten years and coding for over twenty, and still occasionally makes obvious-in-hindsight errors. He finds the least painful way to spot these errors, by far, is to write tests that check what should be "obviously" correct code. John wishes to one-up Marc by pointing out that he has been writing Java code for almost 20 years, and has been programming since 1971. He also has to admit to making stupid errors.

Problem 2 continued on next page. . .

node). Then it should 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. In essence, your code will perform a breadth-first search of a given tree within the filesystem.

Java represents nodes of this tree as objects of class `java.io.File`. You'll want to at least skim through the API documentation for `File`. Pay particular 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. (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 lexicographic order before enqueueing them. Enqueue lower-order nodes before higher-order nodes, as determined by the `compareTo()` method on the `File` objects. You may find the static method `sort()` in `java.util.Arrays` helpful for this task.

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

- `src`
- `src/algorithms`
- `src/hanoi`
- `src/structures`
- `src/algorithms/RecursiveMath.java`
- `src/hanoi/ArrayBasedHanoiBoard.java`
- `src/hanoi/RecursiveHanoiSolver.java`
- `src/hanoi/StackBasedHanoiPeg.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

Finally, examine `src/sorting/MergeSorter.java`. Here, you will use your `Queue` to implement the merge sort algorithm.

Problem 3 continued on next page. . .

What's going on in that type parameter?

`MergeSorter` has an unusual type parameter: `MergeSorter<T extends Comparable<T>>`. You can treat this as the usual generic type `<T>`; the extra bit (`extends Comparable<T>`) is indicating to the compiler that the generic type extends `Comparable<T>`. This means you can depend upon type `T` having a `compareTo()` method. Examine the Comparable API documentation for details.

Merge sort

The merge sort algorithm sorts an ordered data structure, such as an array or queue. In this problem, we'll consider its operation on queues. We'll consider a queue sorted when the elements are in ascending order, that is, when the front of the queue contains the smallest element, and the rear of the queue contains the largest.

At a high level, merge sort works by merging two smaller, sorted queues into a single sorted queue. The merge algorithm works as follows.

Suppose you have a `merge` method that takes two sorted input queues and one (initially empty) output queue. While both input queues are not empty, dequeue one element from the queue whose front element is smaller, and enqueue it into the output queue. Then recursively call this method. Eventually, one of the input queues will be empty. Finish by recursively dequeuing from the remaining input queue and enqueuing onto the output queue, until that input queue is also empty. When both input queues are empty, the output queue will be in sorted order.

Of course, you don't generally have the luxury of starting with a pair of sorted queues. Instead, you are given a single unsorted queue. To sort it using merge sort, you can use a *divide-and-conquer* approach.

First, *divide* the single large queue into a pair of smaller queues, each containing about half the elements of the single queue. Then recursively merge sort each smaller queue. Eventually, the recursive division will result in two single-element queues (or one single-element queue and one empty queue). By definition, these queues are sorted. In a queue with zero or one elements, there is no way for to be out-of-order.

Then, conquer! *Merge* the queues together pairwise until only one remains. Return that queue, as it is sorted.

What to do

`MergeSorter` contains four several methods that you must correctly complete using only recursion. Do not use iteration (`for`, `while`, or `do` loops), and do not use third-party methods for sorting (such as `Arrays.sort()`). If your code uses loops or third-party sorters, you will receive a zero for the assignment.

`mergeSort()` will be implemented recursively, dividing its input in half and recursively `mergeSort()`ing the halves, and then `merge()`ing the then-sorted halves.

`divide()` will be implemented recursively, moving elements from input queues to an output queue, then calling itself recursively.

`merge()` will not be recursive. Instead, it will call the recursive helper function `mergeHelper()`, which recursively merges two sorted queues into an single sorted queue (an *accumulator*) that `merge()` creates. `merge()` then

Problem 3 continued on next page. . .

returns this sorted queue.

`mergeHelper()` will be implemented recursively, moving the smallest element from the front of either input queue into the output queue (thereby accumulating progress), and recursing.

How can you determine the smaller of two elements `e1` and `e2` when merging? You can't 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`.

Finally, two hints: First, be sure you understand the base and recursive case of each of the recursive methods. Trial-and-error coding is unlikely to help you solve this problem. Second, you need not not declare any instance variables (attributes) in `MergeSorter`; there is no need to share state between these methods except through their argument lists and return values. Using instance variables may lead to hard-to-track-down errors. Good luck!

Export and Submit

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, click on the `queues-student` project in the package explorer. Then choose "File → Export" from the menu. In the window that appears, under "General" choose "Archive File". Then choose "Next" and enter a destination for the output file. Be sure that the project is named **`queues-student`**. Save the exported file with the `zip` extension (any name is fine). Log into Moodle and submit the exported zip file.