CS310 Data Structures
San Diego State University
Summer 2018

Program One

Inheritance and Complexity (50 points)

Overview

In short, this assignment requires students to work with a partner to develop and test three priority queues composed with: An unsorted array, a sorted array, and a heap-based priority queue. In each of these implementations students shall use the existing `java.util` data structure as the backing storage. Additionally, students must design the driver application demonstrating its correct timing and performance.

Students shall implement multiple data structures built from the same interface file. The project explores both *interfaces* and *inheritance.* Additionally students shall use *composition* during implementation, so upon its conclusion, students will possess a richer understanding of these classical object-oriented concepts. After students implement the interfaces in multiple concrete classes, they shall develop and perform integration timing tests on them to verify they meet expectations.

Requirements

This project utilizes several Java files organized into multiple packages, so the location of the Java source files remains critical. Files located in an incorrect location or with erroneous packaging information fail to compile properly and produce a substantial point penalty on the overall program.

Part One: Setup Java (Maven) Project

First, students must download the instructor provided interfaces and setup a Maven-based Java project using their chosen development environment. Because this assignment uses complicated packaging, the directory structure on the file system in the source folder remains critical.

Maven Settings:

Archetype	org.apache.maven.archetypes:maven-archetype-quickstart
GroupId	edu.sdsu.cs
ArtifactId	program1

Upon the correct creation of the Maven project, make certain to enable Auto-Import in the IDEA development environment. Maven downloads any necessary files in the background, so the computer typically requires an Internet connection for this initial process. The `pom.xml` file generated during project creation should not require student modification.

Instructor Provided Files:

File	Package	Description
IQueue.java	edu.sdsu.cs.datastructures	Interface file outlining the core functionality for a basic Queue ADT.
IPriorityQueue.java	edu.sdsu.cs.datastructures	Interface extending the basic Queue interface with additional Priority functionality.

Students must copy these files into the project and place them into the appropriate directory. The interface files *must* remain in their specified package. Many IDEs conveniently fix packaging errors either behind the scenes or with a simple click of a button – pay attention to the file layout!

Student Generated Files:

File	Package	Description
App.java	edu.sdsu.cs	Driver program demonstrating the timing for each of the developed data structures.
HeapPriQueue.java	edu.sdsu.cs.datastructures	A concrete class implementing the IPriorityQueue interface using a java.util.PriorityQueue
ArrayPriQueue.java	edu.sdsu.cs.datastructures	A concrete class implementing the IPriorityQueue interface using a java.util.ArrayList and binary search
UnorderedPriQueue.java	edu.sdsu.cs.datastructures	Concrete class using an unordered java.util.ArrayList and sequential search

The development environment usually generates App.java during the project's initial setup, so students will not need to create this file from scratch, but they will need to modify it to include the functionality required by this prompt.

After copying the interface files into the project, students must create the concrete classes for the required data structures. Right clicking on the `edu.sdsu.cs.datastructures` directory in the IDEA project directory structure will bring up a context menu allowing the student so select 'New' and the 'Java Class.' This leads to a dialog box which allows the student to populate it with the required data.

Conveniently, classes constructed in this fashion possess the correct datastructures packaging automatically.

Part Two: Data Structures

Using the instructor supplied `IPriorityQueue` interface, students shall produce the Java class files necessary for additional data structures. Each of these data structures shall implement the same interface. Therefore, they demonstrate *type-polymorphism*. Whenever an application requires an `IPriorityQueue`, any of the data structures developed for this assignment will fill the role.

The required concrete classes either use a different backing data structure or a different internal algorithm. Each impacts performance as the driver will illustrate. Students must provide:

1. In the first data structure, students shall use *composition* with the standard priority queue (which is heap-based) provided by Java. That is, the `HeapPriQueue` HAS-A `java.util.PriorityQueue` inside. The method calls to the class simply redirect calls to this data structure. The overwhelming majority of the methods implemented for this structure will consist of simple one-line statements redirecting calling traffic.

2. In the second data structure, students shall again use *composition*, but they shall use a standard `java.util.ArrayList` as the backing structure. Many of the method calls in this implementation will also simply redirect to the internal `ArrayList`. The `ArrayList`, however, does not perform any internal ordering, so students must supply the proper priority queue logic themselves. This data structure ***must*** use the *binary search* algorithm to establish the proper insertion point at the time the item is added to the queue.

3. The final data structure also uses a `java.util.ArrayList`, but instead of storing the elements in order on insertion, it simply throws them to the end of the list. The removal operation, however, requires the data structure to scan through its entire contents with *sequential search* to find the item with the lowest priority. Adding to this data structure should be fast, but removing will require extra time.

Each data structure must, per the interface documentation, provide two distinct constructors. The first accepts no parameters and replaces the default constructor. By default, this method configures the data structure to use the default, natural ordering found within objects. Rather than creating an internal `Comparator` object, Students should use `Comparators.naturalOrder()` to fulfill this task.

Software developed for this program must adhere to a consistent naming convention, and submitted code shall be formatted.

None of the classes developed for this assignment shall include public functions or methods not specified in an interface file.

Students should use private, or protected, internal methods to help clean the code.

Part Three: Driver (Integration Test)

To help students better understand how an implementation impacts performance, and to verify the implemented data structures meet expectations, they must develop an application program illustrating the complexity growth for two key operations between the different queues. Students must develop timing tests of sufficient size to clearly illustrate the method's actual behavior.

Recall, one cannot estimate an operation's complexity through a single empirical measurement. Instead, one must vary the scale of the input size. Furthermore, small scaled tests remain susceptible to noise in the measurement as other processes on the host machine interfere with the timing. Consequently, students should begin these tests with data sets in the tens, if not hundreds, of thousands.

One approach developers might take could look like:

1. Establish the initial test size for N
2. Record the time with `System.nanotime()`
3. Perform the operation under test on the data structure N times
4. Check the current `System.nanotime()` and establish the test's duration
5. Record the result (e.g., an output file, the screen)
6. Reset the data structure with a `clear()` command
7. Perform steps 2-6 again for input sizes 2N and 4N

Additionally, the order of the inputs will change how the machine moves through the data structure's software. For example, binary search performs differently on inputs (e.g., duplicate, increasing order, decreasing order, random), so the array based priority queue might behave significantly differently if it simply appends to the end of the existing array instead of shuffling things around randomly. The student's software shall establish the worst case input sequence for each operation on each data structure.

Methods to time: `offer(), poll()`

Analysis of each implementation should reveal its expected computational complexity, and the timing tests should verify this behavior.

Delivery

Students must perform an electronic submission of the Java source code produced for this project through the supplied class accounts on the `Edoras` university server. This process requires students create the necessary folder structure on the remote server, so they will likely use secure shell and secure copy to accomplish this task.

The grading script will attempt to collect the completed work by recursively copying the following directory: `~/submission/program1`

Students should simply recursively copy their project's source folder -- complete with its directory structure -- to this folder for proper credit.

Students shall not submit a copy of the target directory or any of the `.class` files generated during the build cycle.

Grading

General	No Java files in the *default* package All student created files include a comment indicating both partner's names Clean formatting No extra public methods or fields Uses correct Java naming conventions for class, fields, and methods. Project compiles without error All data structures provide the required number of constructors
App.java	In package `edu.sdsu.cs` Performs correct timing tests of sufficient size (e.g., N=2000, 2N, 4N) Indicates complete when finished Clearly displays results of timing in useful format on screen Clearly writes results of timing tests to output file: `output.txt`
ArrayPriQueue.java	In package `edu.sdsu.cs.datastructures` Correctly implements the binary search algorithm Demonstrates correct timing for poll() and offer() Demonstrates correct priority queue behavior
UnorderedPriQueue.java	In package `edu.sdsu.cs.datastructures` Correctly implements the sequential search algorithm Demonstrates correct timing for poll() and offer() Demonstrates correct priority queue behavior
HeapPriQueue.java	In package `edu.sdsu.cs.datastructures` Demonstrates correct priority queue behavior