Gebze Technical University, Computer Engineering Department

CSE222 - Assignment 5

Task Scheduling Using Min Heap Data Structure

Due May 1st, 2025

In this assignment, you are expected to implement a min heap data structure and use it in an example use case, which is a task manager class that will shedule tasks issued by many users of different priorities, where a more important user has a higher priority compared to a less important user (paying less probably).

1 Introduction

In this assignment, you are expected to:

1. Implement a class that represents a min heap data structure.
2. Implement a task scheduler using the implemented min heap class.
3. Provide good documentation about your implementations.

Each of the above tasks is explained in its corresponding section, please read them carefuly.

2 Assignment Details

2.1 Min Heap Class

You are expected to implement a min heap class that will act like a priority queue. Do some research about binary heap and Min/Max heap data structures. Your new class should implement the following interface:

```
interface MyPriorityQueue <T extends Comparable<T>> {
     void add(T t);
     T poll();
     Boolean isEmpty();
}
```

The class will be generic, meaning that it should work with any type that satisfies the "extends Comparable<Integer>" requirement. "T extends Comparable<Integer>" means that the type T that will be stored in this class must implement the Comparable interface or any interface that extends it. This is required to compare elements in the queue.

2.2 The Task Manager

Our task manager is a software that receives a series of tesks and executes them according to their priorities. The priorities of tasks are determined by the users that issued them.

2.2.1 The Users

To define the users and their priorities, a configuration file is provided as input which includes the users and their priority values. Here is the format of the contents of the configuration file, with each line representing the priority of that user, and the total number of lines is the total number of users:

```
<Priority of the first user>
<Priority of the second user>
<Priority of the third user>
.
.
.
```

Example configuration file contents with 4 users:

```
20 // User 0 has priority 20
3 // User 1 has priority 3
10 // User 2 has priority 10
0 // User 3 has priority 0
```

Priorities start from 0 to the maximum number an integer can be. The smaller the priority value, the higher the user’s importance is in the system. In the example above, User 1 is more important than user 0, because its priority value 3 is smaller than the other’s value, which is 20. Also, user 2 is less important than user 1, becasue its priority value 10 is larger than 1.

Users should be defined at startup and remain as is throughtout execution. You need to store users in a data structure to associate each task with its respective user using the stored user objects. Creating a new user object for each task is not right, and will penalized. Here is the user class:

```
class MyUser {
 Integer id;
 Integer priority;

 public MyUser(Integer id, Integer priority) {
  this.id = id;
  this.priority = priority;
 }

 public Integer getID() {
  return this.id;
 }
```

```
  public Integer getPriority() {
   return this.priority;
  }
  }
```

2.2.2 The Tasks

After defining the users, the system will receive a series of tasks from the terminal, then execute them after getting the "execute" command. Each task has an ID, which is incremented each time a new task is issued. Tasks will be stored in a MyPriorityQueue instance, and will be queried back when needed.

A task is expected to be a simple class that implements the comparable interface, with the implementations of the methods left for you to do:

```
class MyTask implements Comparable<MyTask> {
 MyUser user;
 Integer id;

 public MyTask(MyUser user, Integer id);
 public String toString(); // Returns "Request <id> User <userID>"
 // And some method that you need to implement.
}
```

The input through the terminal is simply the userID to define a task. Since the system is virtual, we don't need more than that. Here is the input format:

```
<integer:userID> // Task 0
<integer:userID> // Task 1
<integer:userID> // Task 2
<integer:userID> // etc.
.
.
execute
```

Example input with users provided in the previous configuration example:

```
3 // Task 0 user 3
1 // Task 1 user 1
3 // Task 2 user 3
0 // Task 3 user 0
2 // Task 4 user 2
execute
```

Comparing tasks is an essential operation that will allow the system to prioritize tasks over others. A task has priority over another task if the user that issued it is more important. If the users are of equal importance, the task that was issued first should have higher priority. You need to consider this information while implementing MyTask's methods.

2.2.3 Operation

The task manager will continue to receive tasks until the "execute" command is received, where it executes all tasks and terminates. The task to be executed must be the one with highest priority in MyPriorityQueue, which is the one at the head of the heap. The execution of a task is printing the output of the toString method of the MyTask class to the terminal, one line for each task. Example output according to the tasks and users in the previous example:

```
Task 0 User 3
Task 2 User 3
Task 1 User 1
Task 4 User 2
Task 3 User 0
```

3 Testing

You need to test your code on your own before submitting. Make sure that tasks are executed in the right order. Possible test scenarios:

1. Multiple users with different priorities.
2. Same user with multiple tasks.
3. Both at the same time.

3.1 Input

Your code will be tested using a collection of test scenarios. The input is expected to be in the following format:

1. A configuration file. The path will be given at startup.
2. Requests will be issued through the terminal.

The path to the configuration file will be given as a command line argument. There should be no assumptions about where that file is, it must be read from the path the user provided.

3.2 Output

There are 2 types of outputs:

1. Error messages: Print them on System.err.
2. Normal messages: Execution of tasks. Print them on System.out.

Please do not print any unnecessary output, because it may make the evaluation of the results harder.

4 Documentation

You should provide comments explaining how your code works. Do not over explain, it will make reading your code harder. You should also generate the documentation using the javadoc command and provide it in your submission.

5 Rules

- Name of the submitted file should be in this format: <full_name>_PA5.zip.
- The archive type should be zip.
- Your submission should include the following:
    1. Your source code.
    2. A makefile to build and run your code.
    3. javadoc documentation.
- Specify where you used AI, if you did. (Not using it is for your benefit)
- Questions will be accepted until April 28, 2025.

6 Evaluation

6.1 Penalties

- Cheating. (-100)
- Code not compilable. (-100)
- Bad OOP design. (-50)
- Incompatible input/output format. (-10)
- No javadoc documentation included. (-10)
- Modifying the predefined classes and interfaces. (-20)

6.2 Rewards

- Including the complexity of all methods in javadoc. (+10)
- Your tests can be automated. (+10)
    - Your makefile must support the following commands:
        - make clean
        - make build
        - make run ARGS="<arguments>"
    - Your zip archive is flat. (makefile in the root directory)
    - Followed Input/Output rules.