Universidad Carlos III de Madrid

Algorithms and Data Structures (ADS)

Bachelor in Informatics Engineering
Computer Science Department

YEAR: 1º / SEMESTER: 2º

Final practice

Authors:
Juan Perea
Isabel Segura-Bedmar
Julian Moreno Schneider
Harith Aljumaily

Junio 2011

Introduction

As the purpose of this practice, you will have to develop an application to manage the computing rooms of the University.

The application will control how much time computers are taken, so that it can throw out students that have been connected for more than a given time when there are other students waiting for their turn.

Classes

The application will be formed by some object classes for the students, rooms and computers, the master and operative data structures to hold them, a management class, and a testing class.

Object classes

Student

The student class contains the field "`id`", that will be assigned in an automatic and consecutive way, as well as "`lastName`" and "`firstName`".

```
public class Student {
   int id;
   String lastName;
   String firstName;
   public String toString () {
      return id + "\t" + lastName + ", " + firstName;
   }
}
```

Room

The room class contains the field "`name`", with a format like "`1.1.B10`", as well as a flag indicating if the room is closed, and the list of computers in the room.

```
public class Room {
   String name;
   boolean closed;
   ComputerList computerList;
   public String toString () {
      return name;
   }
}
```

Computer

The computer class contains its complete identifier, formed by the room name and a consecutive number. If the computer is taken, it also holds the student that takes it, and the time when it was taken (the minutes passed).

```
public class Computer {
   String roomName;
   int number;
   Student student;          // null if not taken
   int minutesPassed;        // since it was taken
   public String toString () {
      return roomName + "-" + new DecimalFormat("00").format(number);
   }
}
```

Master data structures

The following data structures will be created to hold master data:

Students by name (`StudentTreeByName`).

Students will be stored in an unbalanced binary search tree, ordered by name (`lastName`, `firstName`).

Students by ID (`StudentTreeById`).

In parallel, students will also be stored in an AVL tree, ordered by their ID.

Computer rooms (`RoomList`).

Rooms will be stored in a list. Use of Java collections is allowed.

Computers in each room (`ComputerList`).

Computers in each room will be stored in a list. Use of Java collections is allowed. There will not be a structure to store all the computers in the campus.

Operative data structures

Student queue (`StudentQueue`)

For the correct operation of the application, a class that implements the student queue will also be created. There will be a student queue for the whole application.

Computer taking

There is no need to create any extra structure to deal with computer taking, all the information needed will be held in the computer object, as described before.

Management and testing

Manager

The single-instance Manager class will hold the data structures and control the rest of the classes.

It will also be responsible of time passing management, by simulating the minutes passed since the application was started..

```
public class Manager {
   private StudentTreeByName studentTreeByName = new StudentTreeByName();
   private StudentTreeById studentTreeById = new StudentTreeById();
   private RoomList roomList = new RoomList();
   private StudentQueue studentQueue = new StudentQueue();
   private int minutesPassed;
}
```

Please be aware that the data structures are private; this shouldn't change.

Tester

The static Tester class will test the functionalities by calling the methods in the other classes. It will be responsible of the population of the master data structures. It will also contain the `main` method.

Operations on data structures

QUESTIONS – Every method described in this section MUST include a comment stating its complexity (1, log N, N, N log N, N^2, etc.), with a brief justification and an indication about what N stands for.

The basic operations of each data structure will include:

Room list

This class can be based on `ArrayList`, and will add an extra method (remaining necessary functionalities should be available through `ArrayList`):

```
public class RoomList extends ArrayList<Room> {
   public Room getByName(String name) { ... }
}
```

Computer list

This class can be based on `ArrayList`, and will add no extra methods (methods in `ArrayList` should be enough):

```
public class ComputerList extends ArrayList<Computer> {
}
```

Student tree by name

This class will not be based on Java data structures. It will have the following methods:

```
public class StudentTreeByName {
      public void add(Student student) { ... }
      public Student getStudent(String lastName, String firstName) { ... }
      public ArrayList<Student> getStudentListByLastName(String lastName) { ... }
}
```

Student tree by ID

This class will not be based on Java data structures. It will have the following methods:

```
public class StudentTreeById {
      public void add(Student student) { ... }
      public Student getStudent(int id) { ... }
}
```

Remember this tree should be an AVL tree.

QUESTION – Why have we chosen to implement the tree by ID as an AVL tree, instead of the tree by name?

Student queue

This class will not be based on Java data structures. It will have the following methods:

```
public class StudentQueue {
   public Student peekFirstStudent() { ... }
   public Student dequeueFirstStudent() { ... }
   public void enqueueStudent(Student student) { ... }
   public Student removeStudentByNia(int nia) { ... }
}
```

QUESTION – Which of these methods is not in the standard queue abstract data type?

Primary management operations

Basic operations on internal data structures

The `Manager` object will have the following basic methods on its internal data structures, which mirror some of the basic functionalities described above:

```
public class Manager {
   public boolean canAddStudent(int id,String lastName,String firstName) { ... }
   public void addStudent(int id,String lastName,String firstName) { ... }
   public void addRoom(String roomName,int numComputers) { ... }
   public Student getStudentById(int id) { ... }
   public Student getStudentByName(String lastName,String firstName) { ... }
}
```

These methods should have a short number of lines of code, for example:

```
public class Manager {
   StudentTreeById studentTreeById = new StudentTreeById();
   public Student getStudentById(int id) {
      Student student = this.studentTreeByNia.searchById(id);
      return student;
   }
}
```

More operations on internal data structures

The `Manager` object will have some not so simple methods on its internal data structures:.

```
public class Manager {
   public List<Computer> getFreeComputerList() { ... }
   public List<Computer> getTakenComputerList() { ... }
   public Computer getComputerTakenForMostTime() { ... }
   public ArrayList<Student> getWorkingStudentList() { ... }
}
```

Computer taking and leaving

The `Manager` object will have simple operations that will be called when a student takes a free computer (normally after being in the queue) or frees a computer, either because he or she voluntarily leaves or forced by the `Manager` object because time passed:.

```
public class Manager {
   private void takeComputer(Computer computer, Student student) { ... }
   private Student freeComputer(Computer computer) { ... }
}
```

Please note these methods are kept private, they are for internal use. There will be another method to be called when a student leaves a computer.

Advanced management operations

These functions must write to the standard output.in order to follow how the system evolves. A sample output will be given at the end of this document.

Time management

In order to simulate passing of time, the manager object will have a member variable `minutesPassed` to control the number of minutes passed since the application was started, and this variable will be incremented in a simulated way by a method `advanceMinutesPassed`.

There will also be a method, `manageQueue`, that will be called to check if a student in the queue can be assigned either a free or a freeable computer:

```
public class Manager {
   int minutesPassed = 0;
   public void advanceMinutesPassed(int minutes) {
      while (minutes > 0) {
         ++this.minutesPassed;
         --minutes;
         manageQueue();
      }
   }
   private void manageQueue() {
      // while there are students in the queue
      //   if there are no free computers, look for the computer used for most time
      //   if the student has exceeded use time, free the computer
      //   if there is a free computer
      //     let the first student in the queue take it
      //   else
      //     return
   }
}
```

Please note `manageQueue` is kept private, it is for internal use, it must be called at the end of certain methods (like `advanceMinutesPassed`).

Student queue

Students can enter the queue, or they can leave the queue (or a computer if they are already working):

```
public class Manager {
   public Student putStudentInQueue(int id) { ... }
   public Student removeStudentFromQueueOrComputer(int id) { ... }
}
```

Please note that after a student enters the queue or leaves the system, `manageQueue` must be called in order to check if there is a free or freeable computer.

Room management

Finally, computer rooms, which are opened by default, can be closed and opened again.

```
public class Manager {
   public void openRoom(String roomName) { ... }
   public void closeRoom(String roomName) { ... }
}
```

When closing a room, the computers in it must be freed, and students who leave the room are put into the queue again.

Testing

Testing data structures

Each data structure should have at least one testing method. For example, to test the `StudentQueue`:

```
public class Tester {
   static void testStudentQueue1() {
      StudentQueue studentQueue = new StudentQueue();
      ...
      studentQueue.show();
   }
   public static void main(String[] args) {
      testStudentQueue1();
   }
}
```

Population of master data structures

Before testing starts, the master data structures need to be populated. This is a responsability of the `Tester` class:

```
public class Tester {
   static void populateRoomsAndComputers ( Manager manager ) { ... }
   static void populateStudents(Manager manager, int id1, int n) { ... }
}
```

And it will be invoked this way:

```
public class Tester {
   static void testManager1() {
      Manager manager = new Manager();
      populateStudents(manager, 10001, 100);
      populateRoomsAndComputers(manager);
      ...
   }
}
```

These functions will rely on the basic operations on data structures described above.

Rooms and computers in each room (`populateRoomsAndComputers`)

The room list will be populated with at least three rooms.

The computer list for each room will be populated with consecutive computer ids. Each room will have a different but small number of computers (so that they can be easily visualized and quickly taken).

Students (`populateStudents`)

There will be a method to populate the student trees with at least 100 student objects with consecutive IDs (for example, 1000 to 1099). This method will be based on two arrays with first names and last names, and a random number generator, like:

```
final static String[] firstNames = new String[] { "Ann", "Helen", "Elizabeth",
      "John", "Charles", "Michael", "Albert", "Joseph", "Sean" };
final static String[] lastNames = new String[] { "Robertson", "Smith", "White",
      "Owens", "Brown", "Edwards", "Connery", "Johnson", "Green" };
final static Random rnd = new Random();
```

so that full student names can be obtained this way:

```
String firstName = firstNames[rnd.nextInt(firstNames.length)];
String lastName = lastNames[rnd.nextInt(lastNames.length)];
```

The population algorithm should avoid duplicate names (before inserting a generated student, it will check it doesn't exist, and generate new names for the same NIA until they are not duplicated).

Testing the `Manager` class

The Manager class will have several testing methods covering all the functionalities described. This doesn't mean a testing method

For example:

```
public class Tester {
   static void testManager1() {
      Manager manager = new Manager();
      populateStudents(manager, 10001, 100);
      populateRoomsAndComputers(manager);
      //
      manager.putStudentInQueue(10001);
      manager.advanceMinutesPassed(3);
      manager.putStudentInQueue(10002);
      manager.advanceMinutesPassed(3);
      ...
   }
   public static void main(String[] args) {
      // testStudentQueue1();
      testManager1();
      // testManager2();
      // testManager3();
   }
}
```

When testing the `Manager` class, remember you can call only methods in the `Manager` class, all the internal data structures should be kept private.

Some clarifications

Java arrays will only be used for:

- Population of the student trees, as indicated in the example given previously.

Java data structures (`ArrayList`, `LinkedList`) will only be used for:

- Implementing room and computer lists.
- Methods returning collections.

Both student trees and the student queue will be implemented without using Java data structures or arrays, but developing linear (for lists) and non-linear (for trees) linked structures.

Sample output

This is a partial sample output for a testing method. The number at the beginning of the line represents the minutes passed.

```
  0: Student 10001-Johnson, John in queue
  0: Student 10001-Johnson, John is taking 1.1.A10-01
  3: Student 10002-Green, Ann in queue
  3: Student 10002-Green, Ann is taking 1.1.A10-02
  6: Student 10003-Connery, Ann in queue
  6: Student 10003-Connery, Ann is taking 1.1.A10-03
  9: Student 10004-Johnson, Charles in queue
  9: Student 10004-Johnson, Charles is taking 1.1.A12-01
 39: Student 10005-Lee, Sean in queue
 39: Student 10005-Lee, Sean is taking 1.1.A14-01
 49: Student 10006-Smith, Charles in queue
 49: Student 10006-Smith, Charles is taking 1.1.A14-02
 79: Student 10007-Connery, Charles in queue
 79: Freeing computer 1.1.A10-01; was taken by 10001-Johnson, John
 79: Enqueuing student: 10001-Johnson, John
 79: Student 10007-Connery, Charles is taking 1.1.A10-01
   …
169: Enqueuing student: 10008-Robertson, Michael
169: Student 10003-Connery, Ann is taking 1.1.A14-02
169: The student is not in the queue nor working: 10018
172: Student 10019-Edwards, Charles in queue
199: Freeing computer 1.1.A10-01; was taken by 10009-Johnson, Rachel
199: Enqueuing student: 10009-Johnson, Rachel
199: Student 10005-Lee, Sean is taking 1.1.A10-01
199: Freeing computer 1.1.A10-02; was taken by 10010-Robertson, Joseph
199: Enqueuing student: 10010-Robertson, Joseph
199: Student 10013-Lee, Charles is taking 1.1.A10-02
199: Freeing computer 1.1.A10-03; was taken by 10011-White, Sean
199: Enqueuing student: 10011-White, Sean
199: Student 10006-Smith, Charles is taking 1.1.A10-03
```