A6 - Shipping Game

Contents

1. Heaps

As a prologue to Shipping Game, you have to implement a Heap. Heaps are extremely useful data structures that play a vital role in many graph algorithms, including Dijkstra's Algorithm.

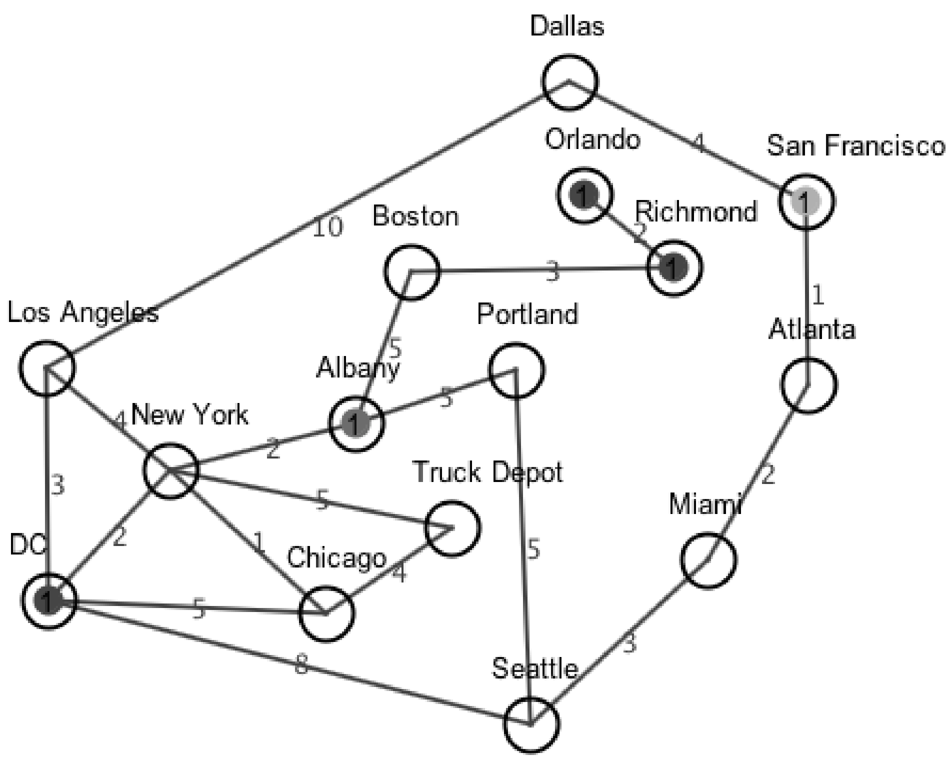
Write a class that implements the MinHeap interface and sticks to the given time bounds. To test your code, run the HeapTester main method with your heap implementation classname as an argument. If all of the tests pass, then you're ready to go!

Due to the nature of the project, writing the Heap is a very small part of the project from a conceptual sense. Try to have the heap finished **one week** after starting the project, so that you have enough time to work on the rest of the project.

TODO: Expand on this section.

2. Game Overview

In Shipping Game (less descriptive name pending) you play the part of a shipping company engineer who makes sure that deliveries get where they're going in a timely fashion. In an instance of Shipping Game, you are presented with an undirected, weighted graph that represents the world. The nodes in this graph are cities where parcels are picked up from or dropped off to, and the edges are highways that connect the cities. Parcels are distributed around the map, each with a starting location (a city in the map) and a delivery destination (a different city in the map).



Cities in the map are labeled with their name, highways with their length. The smaller filled-in circles that appear in some cities are parcels that need to be picked up and delivered elsewhere on the map.

In order to accomplish this task, you have a fleet of trucks that you are able to control. At the beginning of the game you have the option to give your trucks instructions. Additionally, whenever a truck reaches an important point, such as reaching its destination after traveling a highway, it will let you know so you can give it additional instruction. All Trucks begin the game on the Truck Depot city (there will be exactly one truck depot in every map) and must return there after delivering all parcels for the game to end.

3. Installation and Running

Installation

The code for this project is provided as a .jar file, without source-attachment. Various data files are provided as text files in folders.

1. Create a new Java project in Eclipse, name it A6.

2. Download ShippingGame.jar and all of the data folders, drag them into your A6 folder. Don't change the names of any of the folders or the code won't be able to find the data.

3. Right click A6 –> Build Path –> Add External Archives –> Select ShippingGame.jar and click Open.

4. Right click A6 -> Build Path –> Configure Build Path... –> Expand ShippingGame.jar by clicking on the triangle to the left of it –> Select Javadoc Location –> Click Edit... –> Select "Javadoc URL" –> Click Browse... –> Select the "doc" folder (don't expand it) and click Open, Click Ok, Click Ok.

5. Create a package in the src folder (right click src –> new... -> package) called **student**. All code you write for this project must be in the student package.

6. Create a Class in the student package (called whatever you like) that extends **game.Manager**. Notably, you can't call it Manager. This is your manager class, and is your main submission.

Running

1. Select A6 -> Run As... –> Run Configurations... –> New Launch Configuration (blank page with star in top left)

2. Click Search... next to Main class –> Select game.main

3. Switch to the Arguments tab –> Enter the name of your manager class that you created in the student class in the top text box. No .java necessary, just the name of the class.

4. Click run.

5. From here on out, the settings are saved, so you can simply run the project by clicking the green run button as usual.

Arguments

You can provide additional arguments to the runner to have it do different things. All argument combinations have to start with the name of your manager. With just the manager classname provided, the game will launch in GUI mode. There are no other applicable arguments for this mode. Any other combination of arguments will launch the game in GameRunner mode instead, where the game will automatically run your manager on a sequence of maps.

To run GameRunner on a series of seeds <seed1> <seed2> <seed3> such as $15, 12345, 907230587$, use each seed as an argument, separated by a space. (replace MyManager with the name of your Manager):

MyManager 15 12345 907230587

To run GameRunner on a series of n random seeds, use the $-r$ flag, followed by the number of random seeds to run (replace MyManager with the name of your Manager). For example, to run your manager on 10 random seeds:

MyManager -r 10

The GameRunner mode includes the GUI to show the progression of the run games. It also prints output of the games to the console. If you would like to just see the output and not show the GUI (headless), add the $-h$ tag. Thus, either:

MyManager -h 15 12345 907230587

or

MyManager -h -r 10

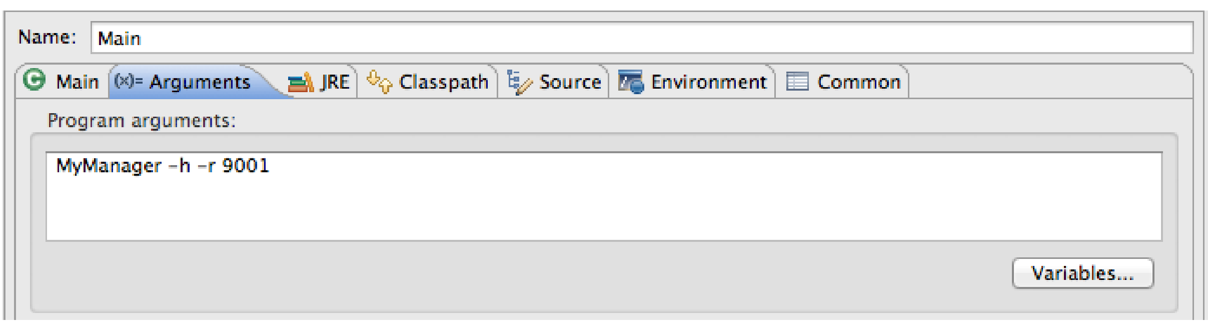
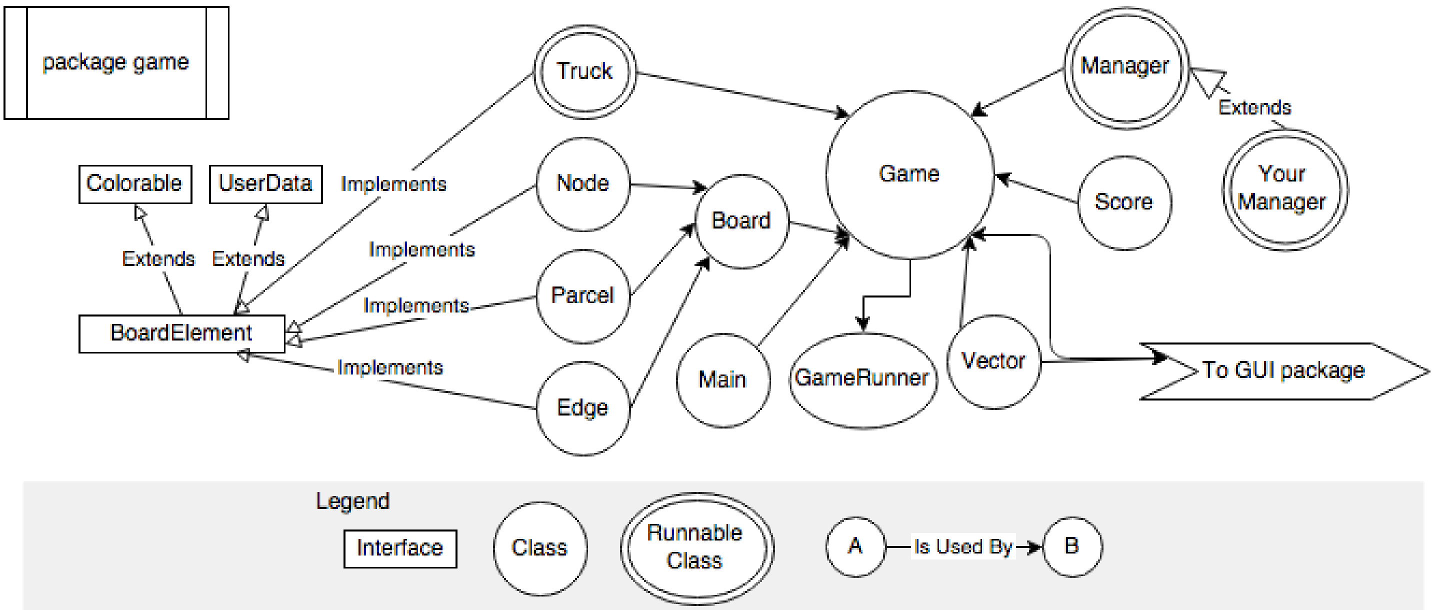


Figure 1: *Running MyManager in headless mode on 9001 random seeds.*

4. Class Explanation

Assuming your installation worked correctly, you should be able to see the javadoc specifications for all of the public methods available to classes in the game package. If not, consult section 2 of this guide or ask on Piazza/at office hours.
The game structure follows the below diagram.



Each interface and class in the figure is briefly explained below. For a more thorough explanation or the method list of a certain interface or class, see the javadoc spec for that interface or class.

Colorable

The colorable interface is implemented in classes that have a "color". Pertaining to ShippingGame, BoardElement extends Colorable so that all elements that are drawn on the map are guaranteed to have a color.

UserData

The UserData interface is implemented in classes that allow the user (Your Manager) to store data in them. You don't have to use this feature of classes that implement UserData, but it may prove useful in some graph algorithms.

BoardElement

In addition to being a concatenation of the requirements of the Colorable and UserData interfaces, the BoardElement interface specifies required behaviors for displaying an object on the GUI. Implementing classes are required to be able to provide various GUI-sensitive information, such as how to draw it, what it's name on the GUI is, and if there are any trucks currently "on it".

Truck

Instances of the Truck class are the game pieces of ShippingGame. Trucks take instructions from Managers, are able to travel the map and pick and drop off a single parcel at a time. Trucks are runnable - each instance of truck is set to run in its own thread.

Node

Instances of the Node class make up the Board. Each Node in the map represents a city that is connected to other cities by highways (Edges). Nodes store parcels until a truck picks them up. One special Node is designated as the TruckDepot, where all Trucks begin the game and must return to before the game can end.

Edge

Instances of the Edge class connect the Nodes in the Board. Each Edge represents a highway that connects two cities (Nodes). Edges in ShippingGame are undirected (bidirectional) and weighted (have length). The weight of the edge tells how long it takes for a truck to travel a given Edge - higher weight takes longer to travel.

Parcel

Instances of the Parcel class are the scoring pieces of ShippingGame. Parcels start at a certain Node on the Board, and award points when they are successfully delivered to their destination Node. The game can only end once all of the Parcels have been delivered.

Board

The Instance of the Board for each game stores the Nodes, Edges, and Parcels associated with the game. It has the scoring constants related to various actions taken during the game, and convenience methods such as getting a random Node or random Edge.

Score

The Instance of Score for each game encapsulates the score for the game. In addition to preventing the user (Your Manager) from altering the score, it has convenience methods for checking for a valid color and calculating the cost of a given Truck Speed.

Vector

The Vector class is very similar to the Point class built into java, except that it allows for doubles in all of its calculations rather than restricting to integers. It is used internally for calculation and is provided to you for convenience.

Main

The Main class begins execution of the game, and stores static convenience methods for use all over the project.

Manager

The Manager class is abstract - it is up to you to extend it. The non-abstract behavior of the Manager class gives convenience methods for getting the map, trucks, and parcels associated with the game. It also gives the Notification enum, which specifies the different reasons a Truck would notify the Manager of a change.

Game

An instance of the Game class represents the game as a whole. The game class is the unifying factor that ties all of the other pieces together, along with communicating with the GUI to make sure the visual version of the game is up to date. It includes convenience methods for querying the current state of the game.

Game Runner

An instance of the Game Runner class allows the automatic running of many games. This is useful to automatically test your manager on may different board, and can be accessed via arguments given to main.

Your Manager

An instance of Your Manager fills in the missing behavior of the Manager class to determine how the game runs. See the next section for more on what you have to do in your extension of the Manager class.

5. Your Tasks

In order to complete the assignment, your primary task is write a class extending the abstract Manager class explained in the previous section. In order to do this, you will have to override and implement the two abstract methods declared in the Manager class: **run()** and **truckNotification(Truck t, Notification message)**. These two methods determine the behavior of the trucks in the game. On their own, trucks don't do anything; it's up to you to instruct them.

Figure 2: *The average intelligence of your shipping boys*

run() is called by the game as soon as the game begins, and allows you to do initial computation and give your trucks their initial set of instructions. Additionally, the body of run() will run in a separate thread from all of the trucks, so you can continue to do computation after the trucks have begun their travel. Your implementation can either loop forever and continually add information to the trucks or execute a single time for initial instructions and then rely on the truckNotification method for further interaction.

truckNotification(Truck t, Notification message) is called by trucks whenever they do something of note. For a full list of the reasons why a truck would call this method, see Manager.Notification in the previous section and in the javadoc. This method is called by the truck in its own thread in order to ask for more instructions. For example, upon arrival at a new node in the graph, the truck may send a notification that there is at least one parcel at the current node. Perhaps you want that truck to pick up that parcel before continuing on its route.

6. Concurrency

Because every Truck you are interacting with exists in its own thread, Shipping Game is inherently multithread. Your manager has to be able to handle multiple trucks calling truckNotification concurrently, without crashing or giving incorrect instructions. That said, depending on the complexity of your solution, you may not need to worry about concurrency at all. For example, consider the following pseudo-code approach to the shipping game problem.

Algorithm 1 Basic Preprocessing

```
for Parcel p do
    Choose arbitrary truck t to deliver p. Store p in a data structure in t's user data.
end for
```

Algorithm 2 Basic Truck Notification (t)

```
if Preprocessing not done then
    Return
end if
if Undelivered Parcel in game then
    if t holding parcel then
        Route t to its parcel's destination, then drop off the parcel
    else
        find the next parcel assigned to t, route to that parcel.
    end if
else
    Route t home
end if
```

This algorithm makes use of every truck, but because there are no data structures that are accessed by multiple trucks and all internal data structures are guaranteed to be thread-safe, there is no need for worry over currency. Assuming no errors, a correctly implemented version of this algorithm would probably net at minimum a good grade, likely higher.

If, however, you're shooting for a perfect grade, you may have to do some amount of work to ensure that your complex network of information sharing between trucks remains thread safe. Concurrent modifications of most data structures cause incorrect states, usually leading to null pointers or array index OOB errors. Failure to write thread safe code might lead to unpredictable and catastrophic errors.

In this vein, you have a few different options to keep code thread-safe. The more powerful the tool, however, the greater the possibility of creating new and more difficult problems. Ideally, come up with a strategy, than pick the least-

Figure 3: *Don't rely on Spongebob to keep your code thread safe.*

comprehensive tool that gives you just enough capability to prevent potential thread safety problems.

Approaches to Concurrency

- **wait() and notify()** - Power: medium. Possibility for Error: very high. Java's built-in concurrency system. Threads are able to cause themselves to wait, using any object as a key, and to "wake up" other threads waiting on a given object. Working out a system of wait() and notify() calls that correctly limits thread movement is surprisingly difficult. At the end of the day, there are higher level systems that are able to accomplish the same tasks in a much more safe way. Don't use these unless you really want to get your hands dirty.

- **Synchronized Collections** - Power: low. Possibility for error: none. The collection class (API) provides the ability to create a synchronized collection (or synchronized set, list, map, etc...). These collections are inherently thread-safe for basic operations, including adding elements, removing elements, getting elements, and checking the size of the collection. Notably, you can't iterate over a synchronized collection without a synchronized block (see next section), so complex operations that require iterating aren't safe. If you can get away with just using Synchronized Collections, you should do so and won't have any problems.

- **Synchronized Blocks** - Power: medium. Possibility for Error: low.
  A synchronized block limits one thread to executing its body at a time. Other threads wait on before the statement until the executing thread finishes the block. Moreover, each synchronized block is called on some object in the form:

  ```
  synchronized(obj){
  //...Do things that only one thread should be doing.
  }
  ```

  And only one thread can access any synchronized block that is associated with that block at a time. This means that if there are two synchronized blocks with the same object:

  ```
  synchronized(obj){
  processA();
  }
  synchronized(obj){ //Note - synchronized on same obj
  processB();
  }
  ```

  processA() and processB() could not possibly be executed concurrently, because a thread's presence in one of the two blocks prevents other threads from being in either of the two blocks.
  The simplest way to use synchronized blocks on Synchronized collections to allow iteration (See API). That way only one complex (iterative) process can occur on the collection at a time, and nothing can go wrong. **This combination is probably the best solution for this project, and should be considered first**.

- **Locks and Locking** - Power: High. Possibility for Error: High
  Locks (and Semaphores and other equivalent classes) take the two pieces of a synchronized block - preventing other threads from accessing code, and letting other threads know that the going is clear - and separate them into two individual lines of code. Only one thread can "own" a mutex at a time (semaphores are generalized to allow N threads), so when a thread acquires a Lock l, all other threads that attempt to call l.lock() are forced to wait until the thread that owns the lock calls l.unlock(), at which point an arbitrarily chosen thread that is waiting will call l.lock() and proceed. This gives you the ultimate control over which threads execute what where and when, but is very prone to issues such as deadlock in which two threads are each waiting to acquire a lock owned by the other. These are overkill for all but the most complex solutions, and most likely not necessary for this project.