Data Structures and Algorithms

Assignment One

Due Monday 12 August 2019

Please ensure that relevant Java source files (excluding bytecode) are submitted in the correct assignment folder on AUT online by 11.59pm on the due date. Zip your project into a folder with your name and student id. Only classes from the Java standard library may be used for this assignment.

Questions:

1. The class *Element* represents a single entity that can be part of multiple elements forming a hotplate. Elements in a hotplate run in their own thread heating cooling when temperatures are applied. The constructor accepts the initial temperature and initializes any of its fields. The class also uses a static field *heatConstant*, which is shared by all *Element* instances, and ideally should be a value greater than 0.0 and less than 1.0. The class keeps track of which other elements it is horizontally or vertically connected to in a *List* called neighbours. A neighbouring *Element* can be added by calling the *addNeighbour* method. The *start* method should start running an Element instance in its own thread. While running the element compares the average temperatures of its neighbours with its own temperature and adjusts its current temperature using this formula before sleeping for a small period of time:

currentTemp += (averageTemps - currentTemp) x heatConstant.

Temperature can be applied to an element by calling the method *applyTemperature* using this formula:

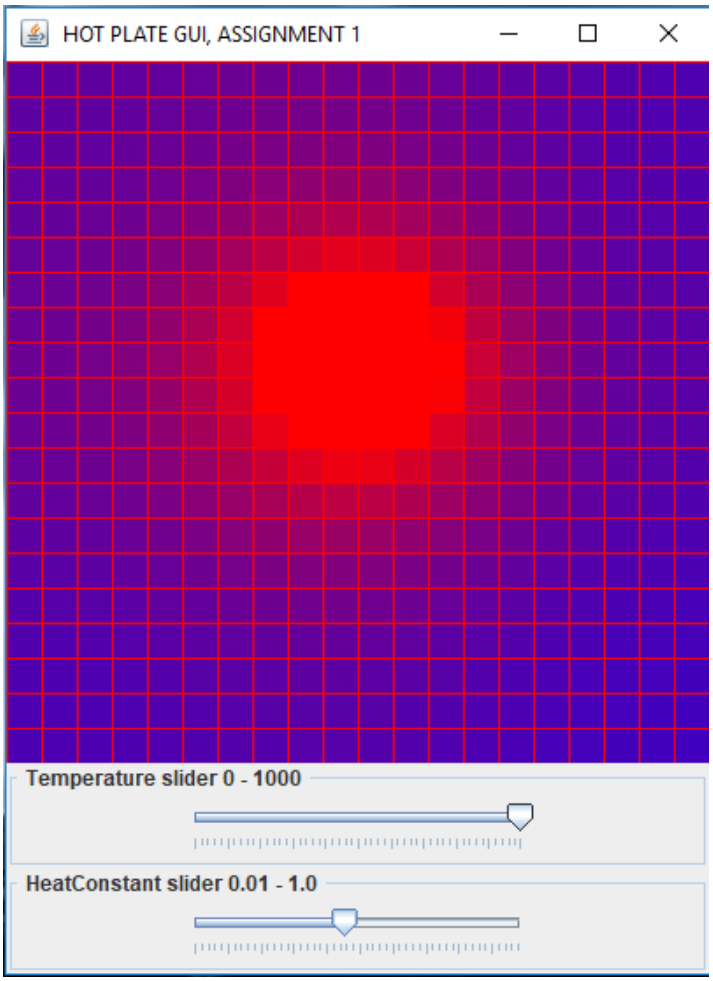
currentTemp += (appliedTemp - currentTemp) x heatConstant.

Because neighbour elements are running in their own threads and accessing each other's temperature, careful synchronization will be needed whenever the temperature field is accessed or changed. Use the UML diagram below as a guide.
(20 marks)

Element
- neighbours : List<Element> - currentTemp : double *+ heatConstant : double* - stopRequested : boolean
+Element(currentTemp : double) +start() : void +getTemperature() : double +requestStop() : void +run() : void +addNeighbour(element : Element) : void +applyTempurature(appliedTemp : double) : void

2. Create a *main* method or a test class that creates two *Element* objects that are neighbours of each other, the first element temperature set at 300, the 2nd at 0 and use an appropriate heat constant. Start both threads and periodically print out both element temperatures so that they eventually should reach approximately the same temperature as each other (within some small epsilon value) at which point the program should exit.
(5 marks)

3. Prepare a GUI called *HotplateGUI* that holds a central panel which draws a rectangular grid representing each of *Element* objects as a 2-dimensional array. Each element has up to four neighbours to its left, right, up and down (care must be taken for Element objects on edges whom may not have a neighbour in a certain direction). The application should listen for mouse events. When the mouse is dragged over a drawn element its *applyTemperature* method should be called, passing an applied temperature to it. This applied temperature should be able to vary and can be set with a *JSlider* temperature gauge. The GUI should also use a *Timer* that constantly repaints the panel with all the *Element* objects in it. Each element should fluctuate from the colours Blue (cold) and Red(hot) on the hotplate as their corresponding temperatures increase or decrease.
(10 marks)



4. Extend the *LinkedSet* class creating a subclass called *LinkedRRSet<E extends Comparable<E>>.* It overrides the *add* method of *LinkedSet* to put elements in a *natural order* (ie using the elements *compareTo* method) with no duplicates. It also has methods *retain* and *remove* for dealing with ranges of elements. Each method accepts two parameters of type *E*, *start* and *end*, which specify a range of elements to *retain* or *remove* from the set (determined by the compareTo method of the elements). If either parameter is **null** the range is considered *open-ended*. The *retain* method should retain just those elements of the set that are between *start* (inclusive) and *end* (exclusive), and **return** the elements that have been removed as a *set*. The *remove* method does the opposite, retaining the elements that are outside the specified range, and returning the elements removed as a *set*. The asymptotic complexity of both retain and remove methods can be done in O(n) time. Throw suitable exceptions if start and/or end are not in the set. Include a driver *main* method to test the class with many variations. (15 marks)

Examples:
set = {1,2,3,4,5,6,7}
retain(2,6)
set = {2,3,4,5} returned set = {1,6,7}

set = {1,2,3,4,5,6,7}
remove(2,6)
set = {1,6,7} returned set = {2,3,4,5}

set = {1,2,3,4,5,6,7}
remove(4,5)
set = {1,2,3,5,6} returned set = {4}

set = {1,2,3,4,5,6,7}
retain(6,7)
set = {6} returned set = {1,2,3,4,5,7}

set = {1,2,3,4,5,6,7}
retain(null,4)
set = {1,2,3} returned set = {4,5,6,7}

set = {1,2,3,4,5,6,7}
retain(4,null)
set = {4,5,6,7} returned set = {1,2,3}

set = {1,2,3,4,5,6,7}
retain(4,null)
set = {4,5,6,7} returned set = {1,2,3}

set = {1,2,3,4,5,6,7}
retain(null,null)
set = {1,2,3,4,5,6,7} returned set = { }