COSC 102 Spring 2015

Sort Visualization Lab 12

The goals during the lab are

- to understand sort algorithms: read the description in the book and trace the execution using playing cards (**partner**)
- to understand the provided visualization Java code: you first modify `RandomWalk` to understand the paint method and the event queue process and (**individual**)
- to implement the mergesort and quicksort visualization: you start working on these sort visual classes (**partner**)

The visualizations are due next lab: each team will demo and explain their programs.

Sorting Investigation

Manual Trace of the Algorithms

In team of two study the distributed code. You are either tracing

- insertion sort and quicksort or
- selection sort and mergesort

for the following random sequence of integers $9, 6, 2, 7, 4, 8, 3$ sorting them in ascending order.

You have to act as the computer does: you don't see all the information at once. **Turn the cards faced down** to differentiate between the number currently examined and the rest of the numbers.

When you are done demonstrate to another team how the algorithms proceed on sorting the cards.

Run and Modify provided code

Once you have done it by hand, you can download the `.java` classes provided on our Moodle course. The code sorts an array of `Name` objects using the different algorithms.

- Run the `main` in `Driver.java`.
- Change `Driver.java` to sort an array of `Integer` instead.

Once you are done with this first part form a pair with one of the student of the other group as you will implement mergesort and quicksort in the visualizer.

Visualization Overview

Check this animation from Wikimedia commons that illustrates insertion sort visually.

In the animation, each bar represents a position in the input array, which is assumed to be an array of distinct integers from 1 to N. The bars go from left to right corresponding to positions in the input array, and the height of each bar corresponds to the value of the element at that position.

The animation starts with a random ordering of the elements and shows how insertion sort rearranges the elements in place.

In the second part of this lab, you will work with a partial Java program, built from scratch using the Swing graphics framework, to build such visualizations using mergesort and quicksort algorithms.

The Visualization Program

In the downloaded starter files for the lab five Java source code files are provided. `SortViz.java` contains the `main` method. Compile all the files and run the `SortViz` program, which should display a window.

Try the implemented animation example (not a sorting algorithm) by

1. selecting *Random Walk* from the *Algorithm menu*: some small squares are displayed in the window.
2. selecting *Resume* from the *Visualization menu*: the squares start moving around randomly.

Your code will follow the template of this random-walk example animation. To follow along, examine the code in `WalkViz.java` and `DisplayPanel.java` to understand how the various classes fit together:

1. When you choose an algorithm from the menu, a new instance of `DisplayPanel` (the class representing the contents of the window) is created. In `DisplayPanel.java` the method `createViz()` is called with a parameter telling the panel which type of animation to display.

2. A new object of that animation type is created. In this case, a new instance of `WalkViz` is created, because that is the class for the random-walk animation.

3. The constructor of `WalkViz` runs, which sets up the initial display. In this case, we create three objects representing colored squares at random positions.

4. The next thing that happens—non-interactively and silently—is that the `run` method of the animation class gets invoked: `run()` of `WalkViz` in this case. This class performs the algorithm being visualized, which, in this case, is just random movement for the objects. Each of those movements creates an *event* that gets added to a linked list—those movements aren't drawn directly.
   What is the instance name of that `Queue`?: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

5. There is a timer that runs once the user chooses *Resume* from the menu (it can be paused using *Pause* and its speed can also be changed from the menu). Each time the timer goes off, the `vizTimer()` method of the animation object gets called. Here, that function pulls an animation event off the head of the linked list and then changes the position of the corresponding walker object. Note that the current `DisplayPanel` width and height are provided in case of the window being resized.

6. The `vizPaint()` method of the animator class always draws the current state of the animation. Here, it calls the `draw` method of each `Walker` object, which just draws a rectangle at its position. Because the timer method updates the positions of the walkers, they will always be drawn at the correct position in the animation.

Choosing another algorithm from the menu will reset the animation pane with a new object.

To make sure you understand the provided code you first have to do the following three modifications to the `WalkViz` class.

1. Notice the animation terminates after a while. Modify the code so that the random walking animation is shorted.
2. Make each square bigger.
3. Change the drawing of the `Walker` object so that there is a bar underneath the square extending to the bottom of the window (leave a little gap to know that the 'bar' isn't extending beyond the visible part of the window).

Animating Sorts

Your job now is to produce two new animator objects, one each for mergesort and quicksort. You should use `WalkViz` as a template. Each class must extend the `AlgViz` abstract class and have the following methods:

- **`constructor`**: it initializes the sort and the animation. It should create a random permutation of integers and also create corresponding objects that can draw the visualization of that array. Note that you should account of the width and height of the window and the number of elements to be sorted, so that your animation display can show the entire data.
- **`run`**: this method actually sorts the array. As it does, it generates events that are added to a linked list. Each `Event` should correspond to a change in the display for the animation.
- **`vizTimer`**: this method repeatedly gets called to make the animation work, so it should take items from the events linked list and process them, changing the objects drawn in the visualization.
- **`vizPaint`**: this method actually draw the current state of the visualization. If you have multiple objects representing the animation, then this method is simple, as it is in the random-walk example: it just iterates through the objects and call their draw methods. Setup —-

There are comments in `DisplayPanel.java` that tell you where to add code to create the new objects you made in response to choosing *mergesort* or *quicksort* from the menu. Go ahead and replace those lines as instructed. If you then compile all the files and run `SortViz main`, you should be able to invoke your code by choosing those algorithms from the menu.

Customization

Your animation doesn't have to look exactly like the one above. You can use colors to indicate which items are being compared or which items were swapped. After you get the basics working, feel free to be creative!