Introduction to Computers and Engineering Problem Solving

Spring 2005

Problem Set 9: Organizing containers for vessel loading

Due: 12 noon, Session 32

Problem summary

At container ports, trucks hauling containers on chassis line up at cranes to have their containers loaded onto an ocean-going vessel. There are C cranes and thus C lines of trucks. Each truck is hauling one container, which may be 10, 20 or 40 feet long. Not all cranes handle all container sizes.

Each container has a size (10, 20 or 40 feet), a weight (in tons, from 0 to 40), and a unique ID (string). It will also be in a line in a crane. We do not need to separately keep track of the truck; it is uniquely identified by the container it is hauling.

Containers arrive in time order; we give you a set of 25 arrivals. See the MIT server for the file. Each container chooses the shortest line among the cranes that can handle its size, and joins it at the end. After all 25 containers have arrived, the cranes service the containers in each line in order. At any point in the arrival or loading process, we need to know the following variables:

1. The list of containers in each line, in physical order
2. The list of containers by size, in descending order of weight
3. The number of containers in each line
4. The number of containers, by size
5. The total weight in all containers of a given size (10, 20 or 40 feet)

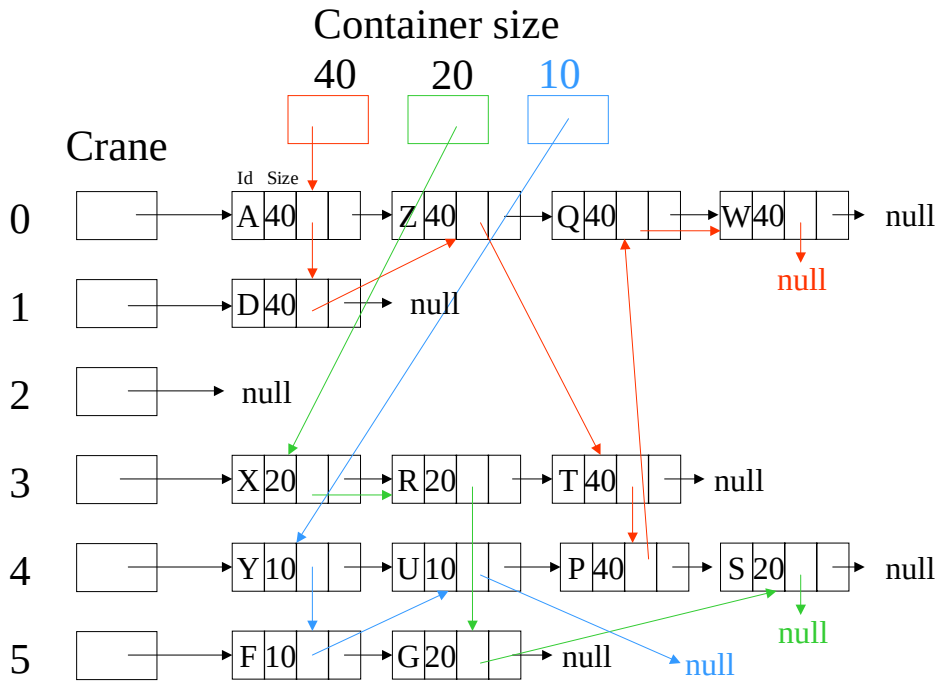
Your program must process the containers in the order in which they are added by the user (see below). The containers must be placed in a two-dimensional linked list. Each container must be in a list, by arrival order, in a line for a crane. And each container must also be in a list, by ascending weight order, in a "virtual line" by container size. That is, we want a list of all 10 foot containers, by weight, a list of all 20 foot containers, by weight, and a list of all 40 foot containers, by weight.

The methods your 2-D linked list class must support are:

1. Add a container to the line for a crane. It must go to the end of the line, and it must also be placed in the correct position in the 'virtual line' by weight for its container size. This method has two arguments, the Container and the crane number.
2. Return the first container in the line for a crane, by crane number. Do not remove the container from the line. This method takes an int argument and returns a Container.

3. Remove a container from an arbitrary position in the 2-D linked list. The argument to this method is the unique ID of the container. The return value is a boolean, indicating whether the container was found and removed.
4. Print out (System.out.println) the line of containers for any crane. Just print the container ID for simplicity. The crane is the input argument.
5. Print out the 'virtual' line of containers of any size. The size is the input argument.
6. Return the total weight in containers of a given size.

An example of container lines, and how they might conceptually be implemented in a 2-D linked list is given below:



(This figure is in color on the MIT server, which may be helpful. The container size lists are in order of decreasing weight, which happens to generally correspond with later arrivals in this example, but will not always be the case. We don't show the weights of each container.)

We give you ContainerTest.java, the class that contains the main() method.
This program has no user inputs at the start; the 25 containers are 'hard wired'. Your system will have 6 cranes; each can handle any container size.

The program has a loop to prompt the user for the action (method) she wishes to perform and any necessary arguments. The program exercises the six methods above:

1. When the user chooses to add a container, select the next Container in the array of 25 Containers. (You could easily prompt for the inputs to create each

Container, but it is much quicker to test your program with pre-built Containers so you don't have a lot of typing.)
2. When the user asks for the first container in a line, print its ID.
3. When the user removes a container, print whether it was removed.
4. Methods 4 and 5 above print out their results. For method 6 print out its return values.

Container data:

1. We code weight as an int, which will range from 0 to 40. If there are ties, you may order them arbitrarily.
2. The unique ID is coded as a one character String.
3. The container size is coded as an int: 10, 20 or 40.

Extra Credit

In addition to your solution above, you may implement a Swing GUI for this problem set and get up to 40 extra credit points. **You must first complete and submit the entire non-GUI solution as outlined above.** Do not attempt to develop your GUI until you have completed the normal assignment since it will be graded separately. You should understand that a GUI solution often requires changes to the rest of your code. Therefore we require you to develop your GUI solution in a separate directory (folder), copying all the files you need. When you submit your solution, first submit your original solution. Then upload your extra credit solution as a second .zip file. Both versions should contain all the files needed to compile and run your solution.

You can get extra credit on only one homework from problem sets 8 to 10. If you don't have time do to the extra credit this time, you still have the opportunity to do so in the future.

You are free to design the GUI as you wish; the following items are suggestions only. Design your GUI to essentially show the figure above, but without displaying the links. Provide buttons for a user to select the methods in your program; have text boxes or a dropdown available to select the unique container ID or crane/line number that are needed as arguments for some methods. Show the total number of containers in each crane line and "virtual line" by size as labels on the panel.

Turn In

1. Place a comment with your full name, MIT server username, section, TA name and assignment number at the beginning of all `.java` files in your solution.
2. Place all of the files in your solution in a single zip file.
    a. Do not turn in electronic or printed copies of compiled byte code (`.class` files) or backup source code (`.java~` files)
    b. Do not turn in printed copies of your solution.

3. Submit this single zip file.
4. Your solution is due at noon. Your uploaded files should have a timestamp of no later than noon on the due date.

Penalties

- 30% off if you turn in your problem set after Friday noon but before noon on the following Monday. You have one no-penalty late submission per term for a turn-in after Friday noon and before Monday noon.
- No credit if you turn in your problem set after noon on the following Monday.