9 Heapsort; StressTests

Practice Problems: The Java Collections Framework

Practice problems help you get started, if some of the lab and lecture material is not clear. You are not required to do these problems, but make sure you understand how you would solve them. Solving them on paper is a great preparation for the exams.

Read through the documentation for Java Collections Framework library. Find how you can run the sorting algorithms defined there. Write a simple program that will test these algorithms and measure their timing in a manner similar to the last lab.

Pair Programming Assignment

9.1 StressTests - Timing Tests

Your job is now to be an algorithm detective. The program we give you allows you to run any of the six different sorting algorithms on data sets of five different sizes using three different `Comparator`s to define the ordering of the data. When you run the program, the time that each of these algorithms took to complete the task is shown in the console.

To run the program you need to do the following:

- Create a new *Java Project* in Eclipse (e.g. *SortingTests*).
- Go to *Preferences* and choose to add a library then choose *Add External jars* to add the file **sorting.jar** to the project. Also add the library **jpt.jar** and, of course, **tester.jar**.
- Create a new *Package* in Eclipse and give it a name `student`. Import into this package the two files: **SortingHeapSort.java** and **Heapsort.java**.
- Download (or find on your computer) the file **citydb.txt** and save a copy in the Eclipse directory that has the **src** and **bin** directories for the *SortingTests* project.

- Go to the *Run* menu, choose *Run Configurations*, select to make a new configuration. Name it *SortingTests* then click on the button to *Select main*. One of the choices should be `sorting.Interactions`. Choose that one. You can now run the program. It will come up with a GUI with several buttons.

- To set up the timing tests you need to go through three steps:

  1. You need to read in the data for the 29470 cities from the file **citydb.txt**. The button *FileInput* opens a file chooser dialog. Select the **citydb.txt** file.
  2. Now hit the *TimerInput* button. It lets you select which algorithms to test, which `Comparator`s to use, and what size data should be used in the tests.

     Start with just a few small tests, to see how the program behaves, before you decide to run all tests.

     The last choice if *heapsort*. The two files in the `student` package provide only hooks to the stress test program — the method `heapsort` in the class `Heapsort` just returns the original unsorted `ArrayList`.
  3. Now you can run the actual tests by hitting the **RunTests** button.

     You can repeat the last two steps as many times as you want to.

Exploration:

Run the program a few times with small data sizes, to get familiar with what it can do. Then run experiments and try to answer the following questions:

A. Which algorithms run mostly in quadratic time, i.e. $O(n^2)$?

B. Which algorithms run mostly in $O(n.log_n)$ time?

C. Which algorithms use the functional style, using `Cons` lists?

D. Which algorithm is the *selection sort*?

E. Why is there a difference when the algorithms use a different `Comparator`?

F. Copy the results into a spreadsheet. You may save the result portion in a text editor with a .csv suffix and open it in *Excel* (or some other spreadsheet of your choice). You can now study the data and represent the results as charts. Do so for at least three algorithms, where there is one of each — a quadratic algorithm and a *linear-logarithmic* algorithm.

   Produce a report with a paragraph that explains what you learned, using the *Excel* charts to illustrate this.

   Your report should have a professional look – typed, computer generated charts, reasonably organized. It does not have to be more than 2 pages long, one page is OK.

G. Add your implementation of the *heapsort* to the sorting algorithms that will be measured (see **Section 9.2**). The skeleton for this is already in place. Run additional tests to evaluate the performance of the *heapsort*. Add to your report a sentence or two describing your findings.

9.2 Priority Queue: Heap and Heapsort

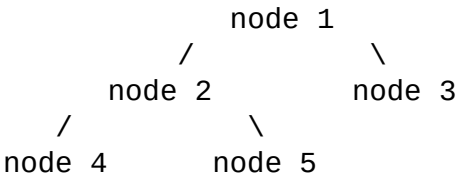
Your goal now is to design a *priority queue* using the heap algorithm. It will then be used to implement the *heapsort* algorithm that we can add to our collection of algorithms to be evaluated.

We start by reviewing the lecture on heap-based priority queue and the heapsort algorithm.

Recall the properties of the *Heap* from the lecture:

- A *heap* is a complete binary tree. It means that every level is filled, except for the last level. The last level is filled from the left.

  So a heap with five nodes will have the following shape:



  The nodes are labeled by levels from left to right, starting at 1.

- The value in each node is greater than (or equal) to the value in either of its children. So the item with the highest value (highest priority) is at the root of the tree.
- The label of the parent of node i is $i/2$
- The label of the left child of node i is $2i$
- The label of the right child of node i is $2i + 1$

If it helps you, draw the picture of the tree and manipulate stickies or pieces of paper to see how the algorithms for insertion and deletion of nodes work.

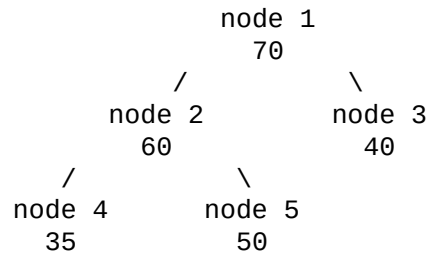
Typically, we represent this *heap* as an `ArrayList` with the first item (at index 0) unused (maybe `null`).

Tasks

A. Add `ExamplesHeaps` class to the package `student` (remember to include `package student;` as the first line.

B. Make three examples of the following *heap*s — by defining `new ArrayList<Integer>()` for each case and using an `initHeaps` method to add the values to them in the appropriate order.

So, to build the following *heap*

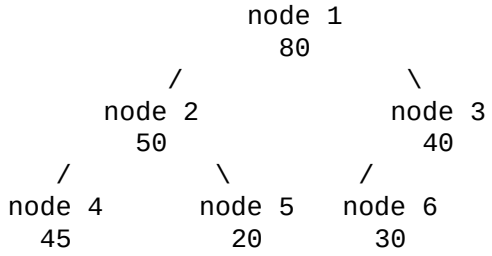


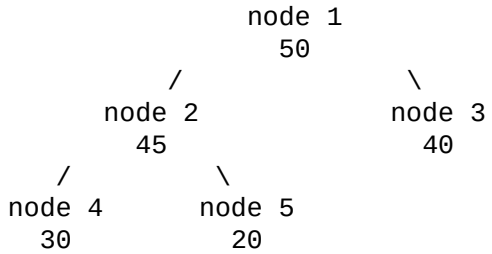
we would proceed as follows:

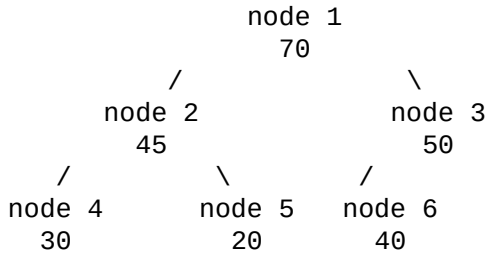
```
ArrayList<Integer> myheap = new ArrayList<Integer>();

void initMyheap(){
  this.myheap.add(null); // the unused first item
```

```
  this.myheap.add(70);
  this.myheap.add(60);
  this.myheap.add(40);
  this.myheap.add(35);
  this.myheap.add(50);
}
```

The three heaps to define:







C. Define the class `PriorityQueue<T>` that will represent the *heap-based* priority queue. It has two fields: an `ArrayList<T>` and a `Comparator<T>` that determines the ordering of the priorities.

D. Design the method `isLeaf` that consumes a node label (an `int`) and returns `true` if the node has no children.

E. Design the method `higherPriorityChild` that consumes the label of a node that is not a *leaf* and produces the index of its child with the higher priority.

Note: If the node has only one child, then this will be the one with the higher priority, of course.

F. Design the method `insert` that inserts a new node into the *heap*.

Here is what we did in class yesterday:

- insert the new item at the next position in the bottom level (or the first position on the next level, if the previous level is filled). Say, this is position k.
- Upheap from k:
  While $k > 1$ and $heap(k) > heap(k/2)$ {
  swap $heap(k)$ and $heap(k/2)$
  set k to $k/2$ }

G. Design the method `remove` that removes a node with the highest priority from the *heap*.

Here is what we did in class yesterday:

- save the top item as a temporary
- move the last item into the top position
- Downheap from $k = 1$:
  While k is not a leaf {
  find ck the node with the larger child of the node k
  if $heap(k) < heap(ck)$ {
  swap $heap(k)$ and $heap(ck)$
  set k to ck }
  else stop
  }

H. Implement a simple variant of *heapstort* as follows:

- In the first step insert the given data into your `PriorityQueue`, one item at a time.
- In the second step, remove the data from your `PriorityQueue` and insert them into the resulting `ArrayList`, one at a time.

  If you just add each item at the end, you will end up with a list ordered in descending order. If you wish to get the correct ordering, insert each item at the index 0.