Assignment #8: The Visual Heap

In this assignment you will visualize the functionality of a heap, stored in an array. You need to create a heap of pre-defined size. The user will enter a number, which has to be inserted into the heap. The heap should then be visualized (see below), such that the changes in the heap data structure are marked. If broken down into parts, this assignment is very simple and can most likely already be finished during the lab!

Remember: separate visualization from logic. Program the logic first. Insert simple print statements to check the functionality of your code. Then add the visualization.

Details:

- The heap must be stored as an array.
- Pre defined max. heap size: 20 elements.
- Visualization: show the array as a sequence of boxes in the natural order from left to right. After each insertion, show the new heap. Array cells that contain values that changed should be red, un-changed values should be green, empty cells should be gray.
- The user inserts a number, the new heap is visualized.
- This can repeat until the heap is full.

Example for visualization

Example heap before insertion:

The user inserts the value "1".
Heap after insertion:

Requirements:

This assignment only requires two classes, the Main class, and a Heap class. Heap contains the public methods "insert()" and "visualize()". You might want to have a private method "computeParentIndex(int child)".

Hints

- Keep two arrays! One is the actual, current heap. The other array is a copy. Before you insert a value into the heap, copy the heap. The visualization routine then simply prints all values of the heap in boxes. The color of the box can be determined by comparison of heap and its copy.
- You will need an index variable "lastIndex", indexing the next available position in the heap. This variable can also be used to determine when the gray boxes start.
- think first about the structure of your code. If you break it down correctly (Motaz will help in the lab!), it should be solvable in less than 2 hours. This assignment is much simpler than the previous one!
- the parent index of an odd child-index "c" is "c/2" (integer division, e.g. 1/2 = 0). The parent index of an even child-index "c" is "c/2 - 1". Using integer division, you don't need to distinguish between even and odd, it turns out that "(c-1)/2" in integer division (only then!) always returns the correct index.

Note:
The number of red boxes shows you the number of swaps during the insertion step. You can never have more than log_2(lastIndex) red boxes. Why?

Bonus points

- Get **3 bonus points** if you add deletion from the heap, too. Deletion would be triggered if the user enters "d" for deletion, instead of a number.
- Get an **additional 2 bonus points** if you add button functionality: button 1 is for insertion (of a random number then, no user input), button 2 is for deletion. This needs events based programming, since you want deletion and insertion at arbitrary times! These 2 bonus points are for event based button programming only!