

Lab 5: stacks, priority queues and heaps

This week, you should try to get to do question 1. The challenge task of question 2 might be interesting to revisit later on, after the lecture next monday, but it should be accessible today; the first half should be straightforward and not too different in spirit from question 1.

You can ask clarification on the coursework, but not direct help with the exercises in there.

1. **Stacks** Open the file `StackExercises.java` available on canvas. You can see we have defined there a couple of interfaces for stacks:

- A minimal one called `MyStackMinimal<T>` that says that a stack should contain the push and pop operations
 - A slightly more expanded one called `MyStack<T>` with a couple extra operations.
- (a) Extend `MyArrayStackInt` to an implementation of `MyStack<Integer>`. Use that to run the commented code in the main function that corresponds to an example discussed during the lecture.
- (b) Taking inspiration from `MyArrayStackInt`, provide another implementation of `MyStack<T>` based on a linked list structure. You should use the class `DLList<T>` developed in the previous lab; if you have not that piece of code, you can use the `LinkedList<T>` class of java¹.
Test that your implementation works by replacing the array-based stack in the main function with your own class.
- (c) Now, we will see how to use stack for another algorithmic problem. Inputs will be strings that can contain
- Either letters or spaces
 - Or parentheses of several kind:

`'(' ')' '{' '}' '[' ']' '<' '>'`

Write a method

```
static boolean isWellParenthesized(String str)
```

that says whether a string is well-parenthesized or not. The goal will be to recognize strings that are well-parenthesized in the obvious way. Here are a couple of examples of well-parenthesized strings:

¹C.f. <https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/LinkedList.html>

```
"aa( aaa aa aa) { ( adcas ) x (af ( fa )ss ) [asf ]afa }aaa"  
"<(<<[]> [ [] ] []>>)"  
"()()aa(bb)()"  
"aaaa"
```

Here are some strings which are not well-parenthesized:

```
"(((())"  
"<)"  
")(())"
```

2. **Priority Queues** Now we are going to look at priority queues. Open the file `PriorityQueuesExercises.java` from canvas and take some time to recall the interface.

- (a) Fill the code of the class `ListPriorityQueue<T>` that aim to implement a list-based implementation of a priority queue (if you do not have the code for `DLList<T>` from the previous lab, use an attribute of type `LinkedList<T>` instead). What are the complexity of the operations you have implemented.
- (b) **(Challenge task)** Now the rest of the exercise is concerned with implementing a priority queue using a heap structure.

Heaps are structure that are easier to picture using tree-like structures, but which actually happen to be well-suited to array representations, because the underlying trees are balanced in a very rigid sense.

So mathematically, a heap will be a kind of labelled tree, where the labels contain an integer priority. We furthermore impose two constraints on those structures:

- The priority of any node should be inferior or equal to the priority of its children
- Below any node in the structure, the depth of the subheap on the left-hand side should be the same as the one on the right-hand side, or one level deeper.

In Figure 1, you have some example of heaps and non-heaps, where only the priorities are used to label the nodes.

Because of the heap structure, we know that the root contains an element with highest priority, so dequeuing will correspond to deleting the root. Doing so however breaks the heap structure; thankfully, there is a way to repair that using only $\mathcal{O}(\log(n))$ number of steps (where n is the number of nodes in a heap).

Similarly, to enqueue an element, we shall simply add a node at the most natural spot, and then repair the heap structure in $\mathcal{O}(\log(n))$ steps.

These two operations, enqueueing and dequeuing, are depicted in Figures 2 and 3 respectively. Enqueueing should be consulted first, as the

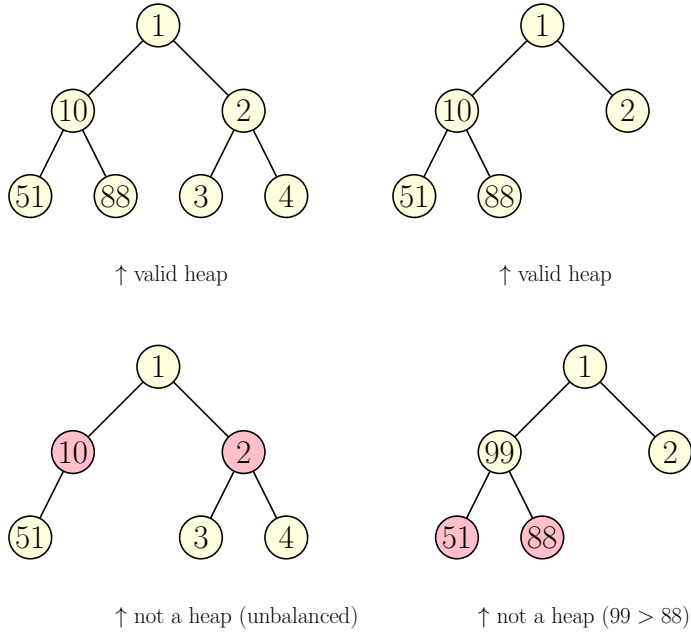


Figure 1: Some example of heaps and non-heaps

repair procedure for dequeuing actually calls the procedure for enqueueing.

Now in the provided code, the heaps are represented in dynamic arrays. The idea is that the label of every node is contained in a cell, and that the tree structure is implicit in the indexing:

- The root corresponds to the cell at index 0
- Regarding the node represented by the cell at index i , its left child is at index $2i + 1$ and its right child at cell $2i + 2$ (if they exist)
- The parent of the node represented by the cell at index j is at index $\left\lfloor \frac{j-1}{2} \right\rfloor$

With these notations, the two heaps in Figure 1 are represented by the following arrays:

$\{ 1, 10, 2, 51, 88, 3, 4 \}$
 $\{ 1, 10, 2, 51, 88 \}$

And all of the tree-like structures in Figure 2 may be represented by

$\{ 1, 10, 8, 51, 88, 11, 9, 99, 88 \}$
 $\{ 1, 10, 8, 51, 88, 11, 9, 99, 88, 7 \}$
 $\{ 1, 10, 8, 51, 7, 11, 9, 99, 88, 88 \}$
 $\{ 1, 7, 8, 51, 10, 11, 9, 99, 88, 88 \}$

Now the code for the heap-based queue is already somewhat pre-filled in

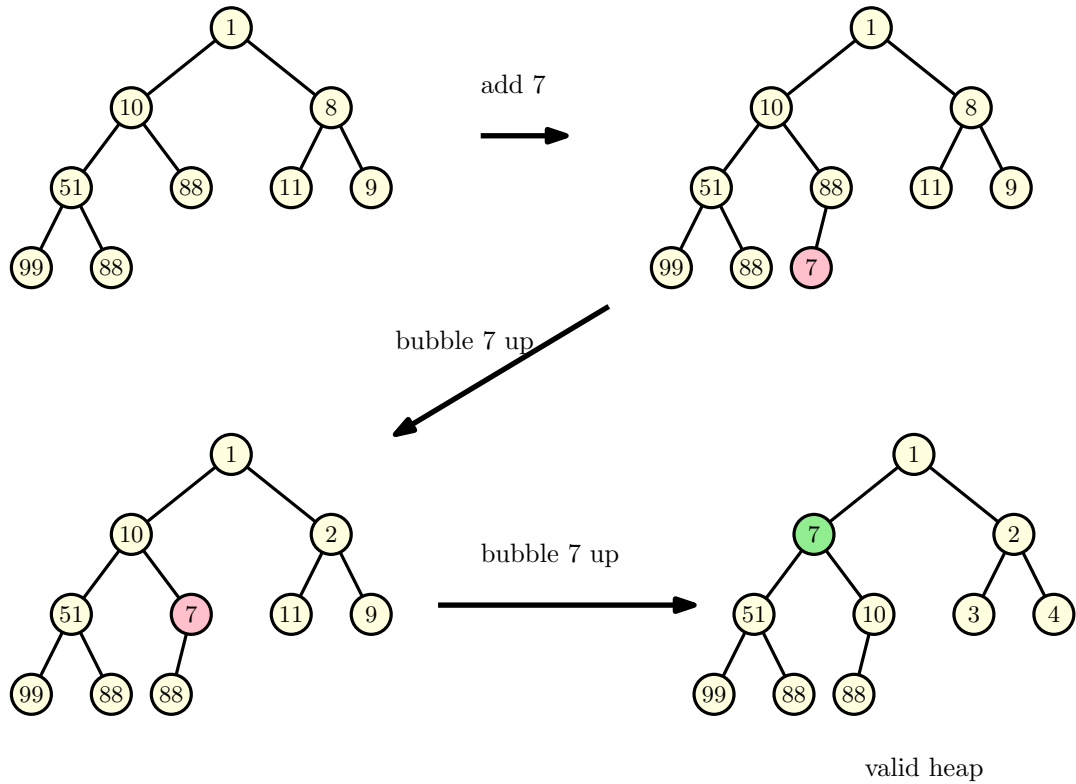


Figure 2: Inserting a new element with priority 9 in a heap and repairing.

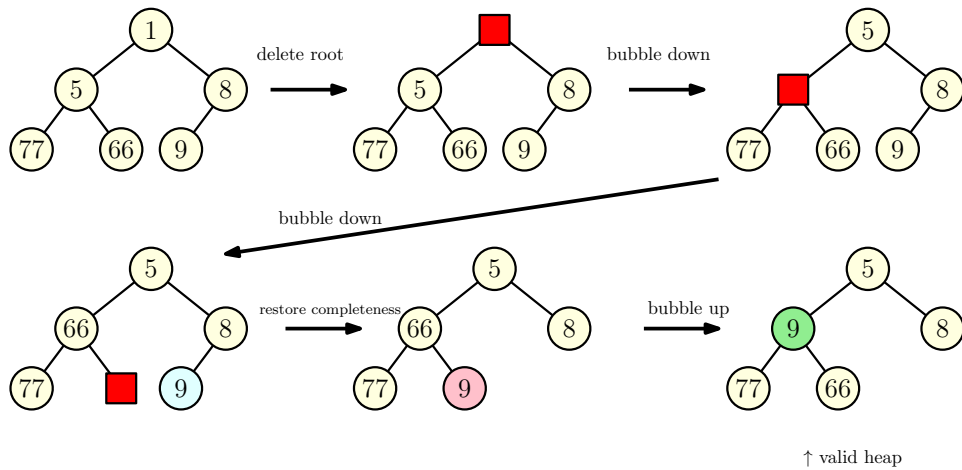


Figure 3: Removing the root of a heap and repairing. Note that the procedure is in two steps: first a “hole” is propagated down, and then, in order to restore the second invariant, the deepest rightmost node is put in place of the newly created hole, and then bubbled up using the same repair procedure as for enqueueing.

the file, except for the part that handle the repairs, handled by private methods

```
private void bubbleUp(int index);  
private int bubbleDown(int index);
```

Your task is to complete the code of these functions to obtain a working dequeue implementation as sketched in the handout!