CET1022 - Data Structures and Algorithms with Java - Practicum 03

Topics Covered: Queue, Priority Queue, Binary Tree, and Heap

Learning Objectives:

- Understanding of the defining property of a priority queue
- Implementing a priority queue using binary heap

Deliverables:

- Submit a single zip file called `CET1022_P03_<Your_Name>.zip` (e.g. `CET1022_P03_John_Doe.zip`) containing your Java project.
- Note that a non-working submission will result in a zero

Background

Priority Queue ADT

A queue, where collection of objects are added and removed according to **first-in-first-out (FIFO) principle**, is an abstract data structure that we are familiar with. In this practicum, we will introduce a variant of queue known as ***priority queue***.

Priority queue is collection in which removal is not strictly a FIFO principle, but of the highest priority element in the collection, according to some method for comparing elements. If comparison uses the natural ordering, the highest priority element is the one with smallest value. Whereas for comparison using reverse of natural ordering, the highest priority is the one with the highest value.

Priority queue can be summarized with the following properties:

- Every element will have a key which represents the priority assigned to it.
- Element with the highest priority will be dequeued before an element with lower priority.
- Element will be dequeued according to the FIFO principle if both have the same priority assigned to it.

Implementing Priority Queue with a Heap

There are a couple of ways to implement a priority queue but the ones that we will be learning is the classic and more efficient way by using a data structure known as ***binary heap*** which allows us to perform both insertion and removal in logarithmic time *O(log n)*.

A heap is actually structured like a binary tree however for the implementation of a binary heap, we will be using an array for the internal representation. There are two variations of binary heap:

- **min heap** - smallest key value is the highest priority and thus always at the front of a queue
- **max heap** - largest key value is the highest priority and thus always at the front of a queue

In this practicum, we will be implementing the **min heap**. Thus, the following will be explained with min heap implementation in mind.

Nevertheless, in order for the heap to work efficiently and properly, it needs to satisfy two additional properties:

- Structure Property
  - A heap should be a complete binary tree where each level has all of its nodes with the exception of the bottom most level which should also be filled in from the left to right.
  - Because of this property, a complete tree can be represented using an array (more details on below).
- Heap Order Property
  - In a heap, for every position other than the root, the key value stored at *p* is greater than or equal to the key stored at *p*'s parent.
  - With that said, the smallest key value is always stored at the root of a min heap.

Array-Based Representation of Complete Binary Tree

Elements are stored in an array-based representation, such that a node at index position ***p***:

- If p is the root, then **p = 0**
- Left child of parent, p is at position **2p + 1**
- Right child of parent, p is at position **2p + 2**

Hence, for a tree of size *n*, the last position is always at index ***n* - 1**.

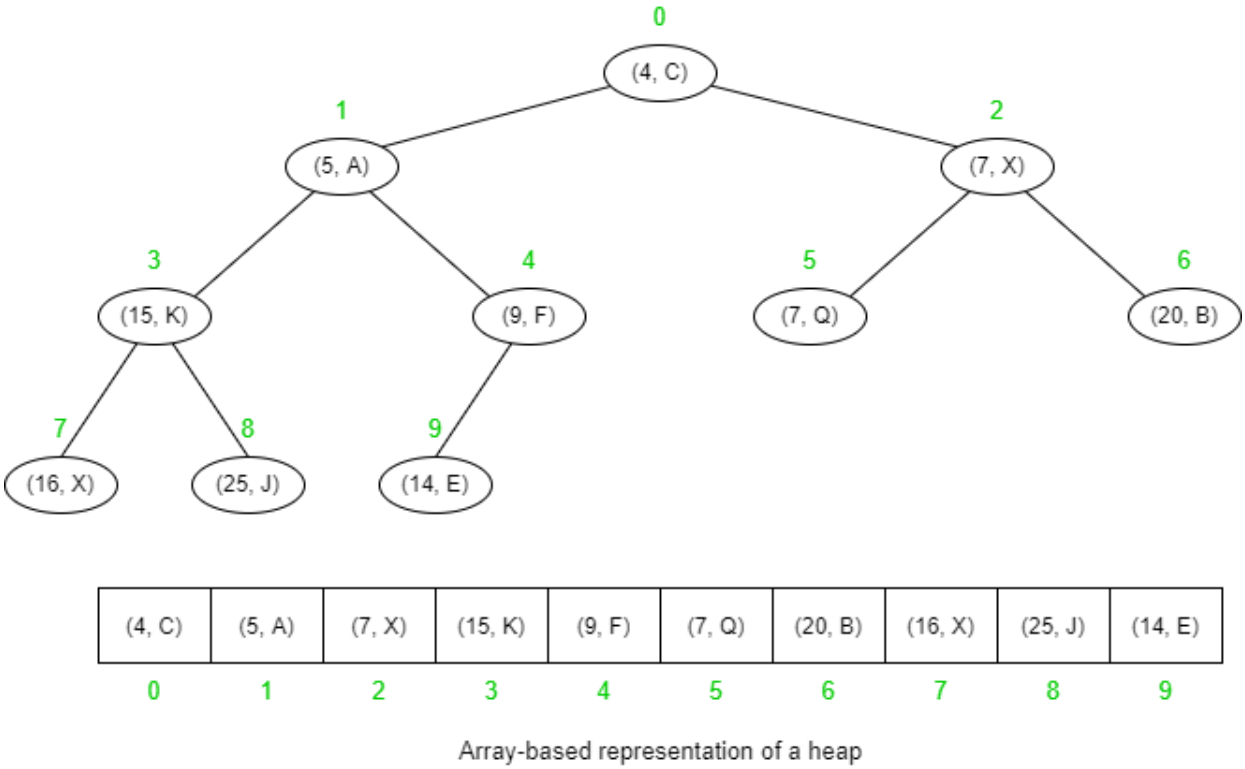


Figure 1: Array-based representation of a heap

Figure 1 above shows an example of a min heap priority queue implemented using an array. Notice that to get the left child of a parent at index position 3 (15, K), we can apply the formula above 2p + 1 which equals to 7, to get the index position of the left child.

Heap Operations

We will look at two heap operations which are insertion and removal of elements.

Insertion of new element to the queue should be placed to the end of the array or leftmost position at the bottom level of a tree. This will maintain the complete binary tree property but the bad news is that it will likely violate the heap order property.

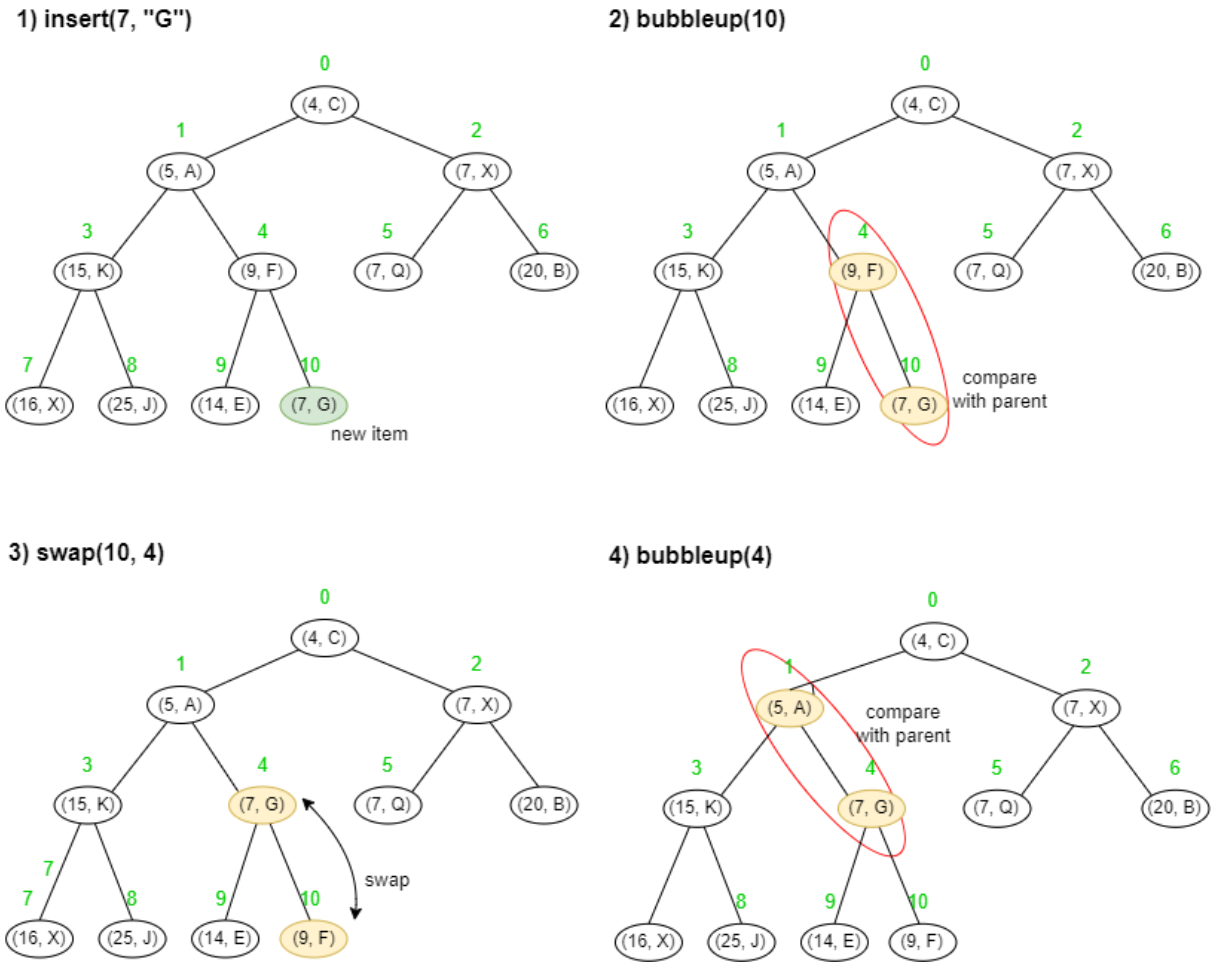


Figure 2: Example of insertion of new element into heap

Figure 2 shows the insertion of a new node `(7, G)`. Second sub-figure invokes the `bubbleup` method which compares the key of child and its parent. If child's key value is smaller than the parent, then a `swap` will take place. The process is repeated until no violation of the heap-order property occurs. This process of upward movement of newly inserted node is known as up-heap bubbling or some may refer to it as percolate up.

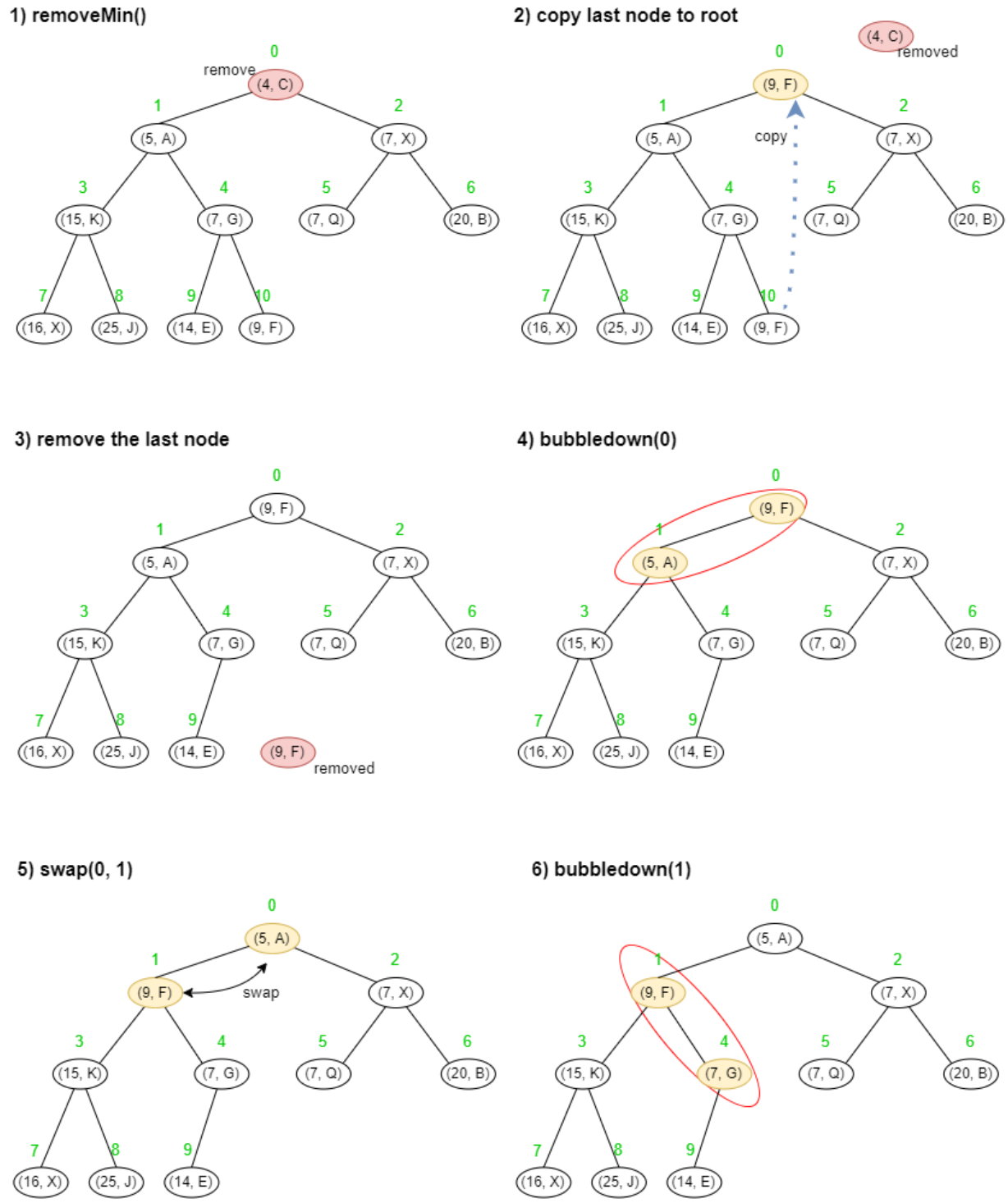
Next, we look at the removal of the element with the smallest key assigned. Finding the minimum is easy since we know that the smallest key will be stored in the root for a min heap tree. The hard part is to ensure that it complies with the heap structure and order properties after the root has been removed.

To restore the heap's properties, we can do it in two steps:

1. Restore the root by copying the last element in the queue to the root, in place of the element with minimal key that is being removed. The element at the last position is removed from the tree after this. With this step done, the heap order would probably have been destroyed. Proceed step 2 to restore it.
2. There are 3 scenarios to restore the heap order (here *p* initially denotes the root of tree):
    1. If the tree only has one node (the root), then the heap order is complied and the algorithm terminates

2. If ***p*** has no right child, let ***c*** be the left child of p. If $k_p <= k_c$, heap order is complied and algorithm terminates, otherwise, swapping of both elements happened to restore the heap order.
3. If ***p*** has both children, let ***c*** be a child of ***p*** with minimal key. When ***p*** has two children, we consider the smaller key of the two children.

After restoring the heap-order property for node **p** in relation to its descendants, there may be a violation of this property at **c**; hence, we may have to continue swapping down the tree until there is no violation of the heap-order property. This process is known as down heap bubbling. Refer Figure 3 below for an example of the removal process:



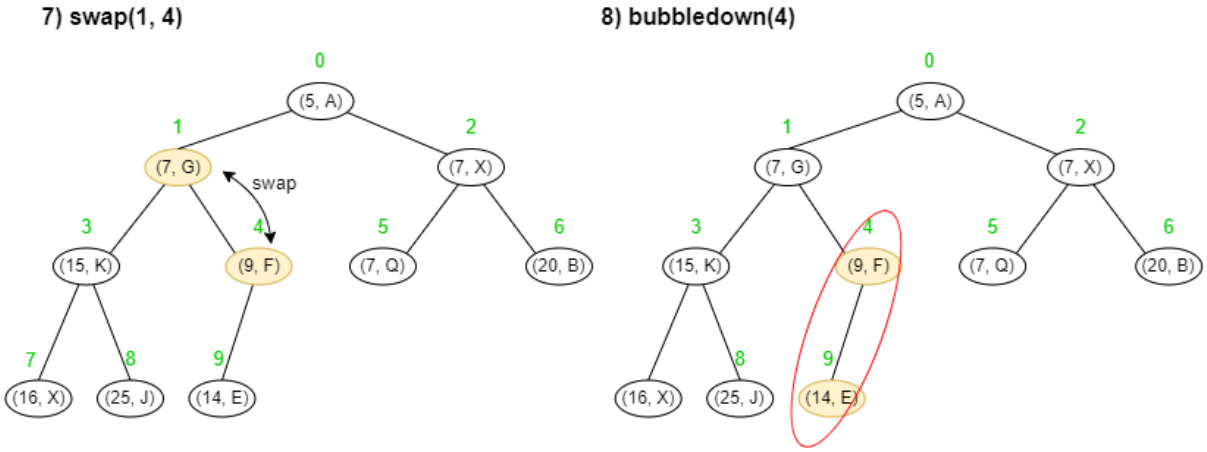


Figure 3: Example of removal of root node in heap

Task

You are given three files - `MinHeapPriorityQueue.java`, `KeyComparator.java` and `Node.java`. You are required to complete the implementation of this priority queue which uses the min heap.

`KeyComparator` class have been completed for you.

Your main task is to write code for the methods missing implementation in `MinHeapPriorityQueue` and `Node` class.

Use the `TestPriorityQueue.java` driver code included to test and check your implementation of the program.

you can put more test cases to test.

IMPORTANT:

- You are **NOT** required and **NOT** allowed to import any third party or Java packages for this practicum other than the ones that are automatically loaded by the JVM (aka no `import` statements).
- Do **NOT** modify any existing code in the given Java files unless stated.
- You are recommended to expand on the test cases on your own to test your program thoroughly to ensure that it is free of errors or bugs.

Rubrics

For the maximum allocation of marks, refer to the table below.

Description	Marks (%)
Successful implementation of the `MinHeapPriorityQueue` Class	80
Successful implementation of the `Node` Class	5
Proper and sufficient comments to explain code using Javadoc	10
Proper display outputs (easy to read, correct decimal places, etc.)	5