Assignment 5 : Priority Queue

So, it's finally time for you to implement a class of your own: a **priority queue**, which is a variation on the standard queue described in the reader. The standard queue is a collection of elements managed in a first-in, first-out manner. The first element added to the collection is always the first element extracted; the second is second; so on and so on.

In some cases, a **FIFO (First-In-First-Out)** strategy may be too simplistic for the activity being modeled. A hospital emergency room, for example, needs to schedule patients according to priority. A patient who arrives with a more serious problem should preempt others even if they have been waiting longer. This is a priority queue, where elements are added to the queue in arbitrary order, but when the time comes to extract the next element, it is the highest priority element in the queue that is removed. Such an object would be useful in a variety of situations. In fact, you can even use a priority queue to implement sorting; insert all the values into a priority queue and extract them one by one to get them in sorted order. The main focus of this lab is to implement a priority queue class in several different ways. You'll have a chance to experiment with arrays and linked structures, and in doing so you'll hopefully master the pointer gymnastics that go along with it.

*The **VCPriorityQueue** Interface*

The priority queue will be a collection of Entry objects. Entry object (will be provided) simply has K(key) V(value) properties and your priority queue uses the K(key) property to keep the priority in the queue. The starter project includes the public interface of all priority queues and the Entry class.

enqueue is used to insert a new element to the priority queue. **dequeueMin** returns the value of highest priority (i.e., smallest key of Entry) element in the queue and removes it. **merge** destructively unifies the incoming queues and returns their union as a new queue.

Implementing the priority queue

While the external representation may give the illusion that we store everything in sorted order behind the scenes at all times, the truth is that we have a good amount of flexibility on what we choose for an internal representation. Sure, all of the operations need to work properly, and we want them to be fast. But we might optimize not for speed but for ease of implementation, or to minimize memory footprint. Maybe we optimize for one operation at the expense of others. This lab is all about client expectations, implementation and internal representation. Yes, you'll master arrays and linked lists in the process, but the takeaway point of the lab—or the most important of the many takeaway points—is that you can use whatever machinery you deem appropriate to manage the internals of a new, object-oriented container. You'll implement the priority queue in three (or four if you opt for the extra credit) different ways. Two are fairly straightforward, but the third is nontrivial.

Implementation 1

Optimized for simplicity and for the **enqueue** method by backing your priority queue by an unsorted **ArrayList**. **merge** is pretty straightforward, but **peek** and **dequeueMin** are expensive, but the expense might be worth it in cases where you need to get a version up and running really quickly for a prototype, or a proof of concept, or perhaps because you need to enqueue 50,000 elements and extract a mere 50. I don't provide much in terms of details on this one, as it's pretty straightforward.

Implementation 2

Optimized for simplicity and for the **dequeueMin** operation by maintaining a sorted **doubly linked** (next and prev pointers required) **list** of strings behind the scenes. **peek** and **dequeueMin** will run super fast, but **enqueue** will be slow, because it needs to walk the list from front to back to find the insertion point (and that takes time that's linear in the size of the priority queue itself. **merge** can (and should) be implemented to run in linear time, for much the same reason Merge from merge sort can be.

Implementation 3

Balance insertion and extraction times by implementing your priority queue in terms of a binary heap, discussed in detail below. When properly implemented, **peek** runs in **O(1)** constant time, **enqueue** and **dequeueMin** each run in **O(lg n)** time, and **merge** runs in **O(n)** time.

Detail Implementation

Implementation 1 - Unsorted ArrayList (`class ALPriorityQueue`)

This implementation is layered on top of the ArrayList class. **enqueue** simply adds the new element to the end. When it comes time to **dequeueMin** the minimum element (i.e. the one with the highest priority in our version), it performs a **linear search** to find it. This implementation is straightforward as far as layered abstractions go, and serves more as an introduction to the architecture of the lab than it does as an interesting implementation. Do this one first.

Implementation 2 - Sorted doubly-linked list (`class DLPriorityQueue`)

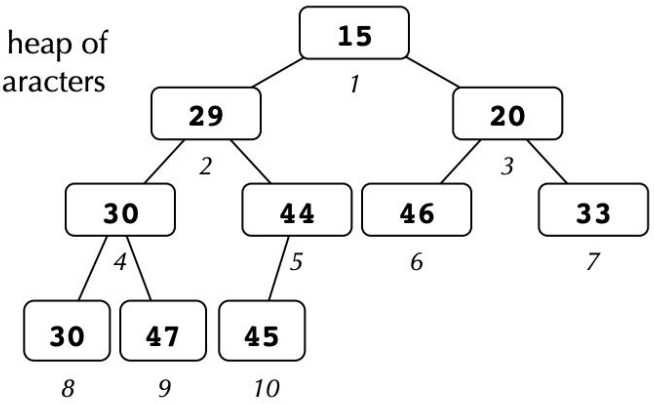
The linked list implementation is a doubly linked list of values, where the values are kept in sorted order (i.e., smallest to largest) to facilitate finding and removing the smallest element quickly. Insertion is a little more work, but made easier because of the decision to maintain both **prev(**ious) and **next** pointers. merge is conceptually simple, although the implementation can be tricky for those just learning linked lists for the first time.

Implementation 3 - Binary Heap (`class BHPriorityQueue`)

A heap is a binary tree that has these two properties:

- It is a complete binary tree, i.e. one that is full in all levels (all nodes have two children), except for possibly the bottom-most level which is filled in from left to right with no gaps.
- The value of each node is less than or equal to the value of its children.

Here's a conceptual picture of a small heap of integer strings (i.e. strings where all characters are digits)

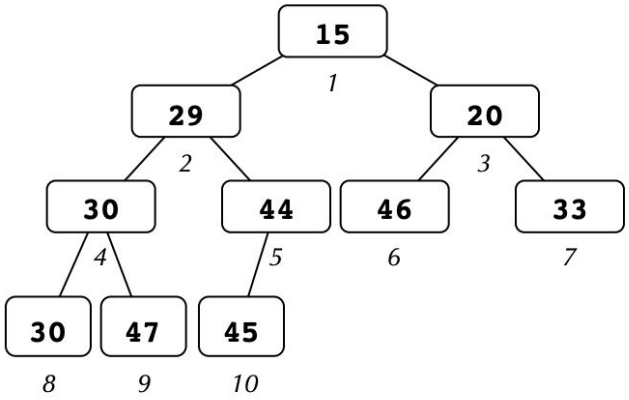


Note that a heap differs from a binary search tree in two significant ways. First, while a binary search tree keeps all the nodes in a sorted arrangement, a heap is ordered in a much weaker sense. Conveniently, the manner in which a heap is ordered is actually enough for the efficient performance of the standard priority queue operations. The second important difference is that while binary search trees come in many different shapes, a heap must be a complete binary tree, which means that every heap containing ten elements is the same shape as every other heap of ten elements.

Representing a heap using an array

One way to manage a heap would be to use a standard binary tree node definition and wire up left and right children pointers to all nodes. We can exploit the completeness of the tree and create a simple **array** representation and avoid the pointers. By default (default constructor), you will create an array of 100 elements and also provide a constructor to specify the size.

Consider the nodes in the heap to be numbered level by level like this:

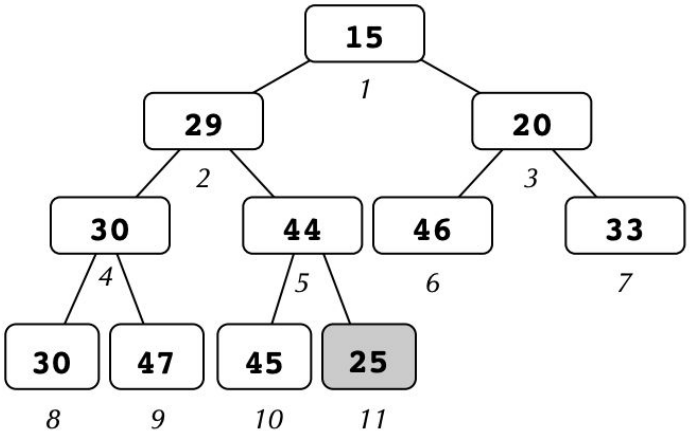


and now check out this array representation of the same heap:

15	29	20	30	44	46	33	30	47	45
1	2	3	4	5	6	7	8	9	10

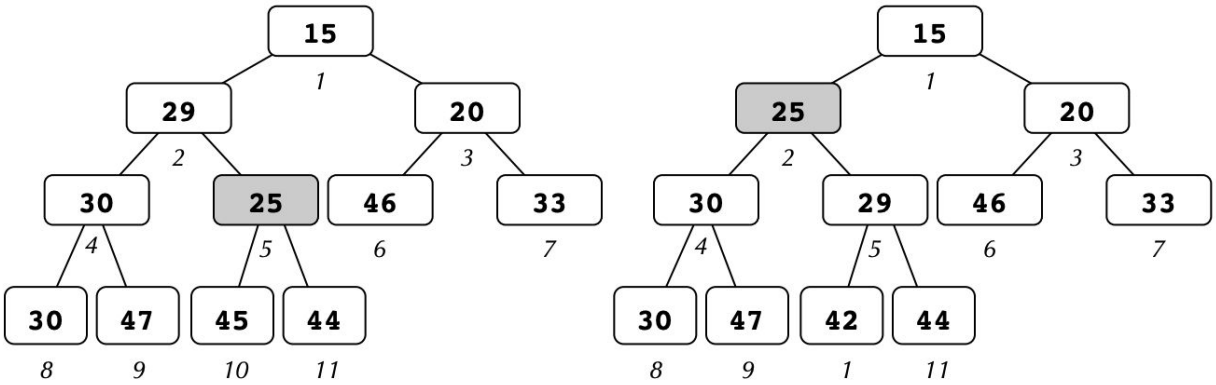
Heap insert

Inserting into a heap is done differently than its functional counterpart in a binary search tree. A new element is added to the very bottom of the heap and it rises up to its proper place. Suppose, for example, we want to insert "25" into our heap. We add a new node at the bottom of the heap (the insertion position is equal to the size of the heap):



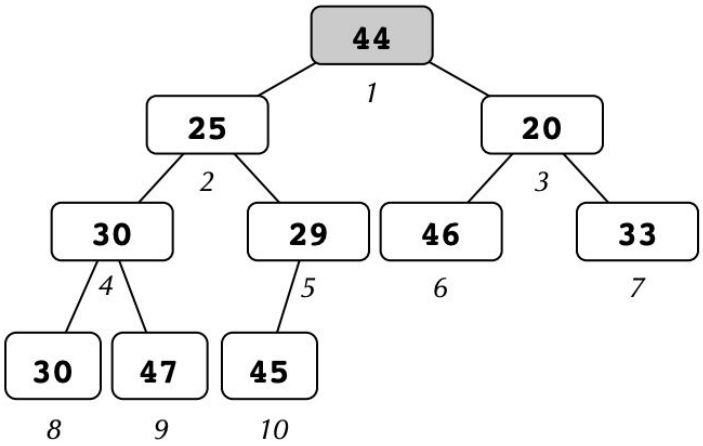
15	29	20	30	44	46	33	30	47	45	25
1	2	3	4	5	6	7	8	9	10	11

We compare the value in this new node with the value of its parent and, if necessary, exchange them. Since our heap is actually laid out in an array, we "exchange" the nodes by swapping array values. From there, we compare the moved value to its new parent and continue moving the value upward until it needs to go no further. This is sometimes called the bubble-up operation.

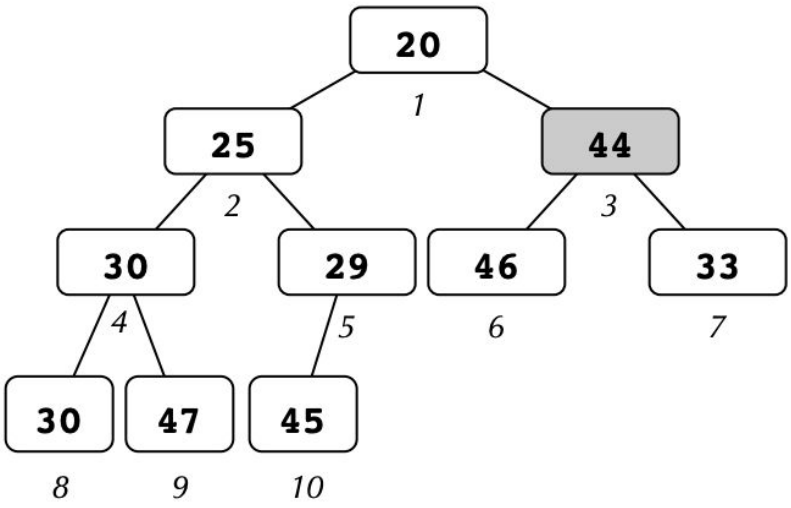


Heap dequeueMin

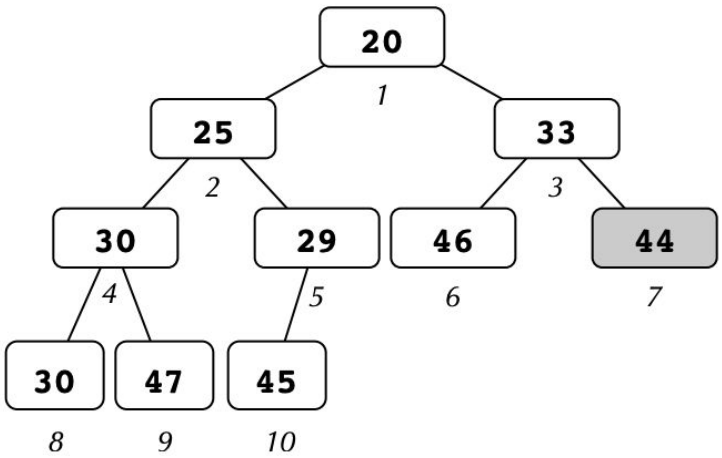
Where is the smallest value in the heap? Given heap ordering, the smallest value resides in the root(top node), where it can be easily accessed. However, after removing this value, you generally need to rearrange the nodes that remain. Remember the completeness property dictates the shape of the heap, and thus it is the bottommost node that needs to be removed from the structure. Rather than re-arranging everything to fill in the gap left by the root node, we can leave the root node where it is, copy the **value** from the last node to the root node, and remove the last node.



We have a complete tree again, and the left and right subtrees are heaps. The only potential problem: a violation of the heap ordering property, localized at the root. In order to restore the min-heap property, we need to trickle that value down to the right place. We will use an inverse to the strategy that allows us to float up the new value during the enqueue operation. Start by comparing the value in the root to the values of its two children. If the root's value is larger than the values of either child, swap the value in the root with that of the smaller child:



This step fixes the heap ordering property for the root node, but at the expense of possibly breaking the subtree of the child we switched with. The sub-tree is now another heap where only the root node is out of order, so we can iteratively apply the same reordering on the subtree to fix it up and so-on down through its sub-trees.



You stop trickling downwards when the value sinks to a level such that it is smaller than both of its children or it has no children at all. This recursive action of moving the out-of-place root value down to its proper place is often called heapify-ing. You should implement the priority queue as a heap array using the strategy shown above. This array should start small and grow dynamically as needed.

Merging two heaps

The merge operation—that is, destroying two heaps and creating a new one that's the logical union of the two originals—can be accomplished via the same heapify operation discussed above. Yes, you could just insert elements from the second into the first, one at a time, until the second is depleted and the first has everything. But it's actually faster—asymptotically so, in fact—to do the following:

- Create an array that's the logical concatenation of the first heap's array and the second heap's array, without regard for the heap ordering property. The result is a complete, array-backed binary tree that in all likelihood isn't even close to being a heap.
- Recognize that all of the leaf nodes—taken in isolation—respect the heap property.
- Heapify all sub-heaps rooted at the parents of all the leaves.
- Heapify all sub-heaps rooted at the grandparents of all the leaves.
- Continue heapify increasingly higher ancestors until you reach the root, and heapify that as well.

Binary Heap Implementation Notes

- **Think before you code.** The amount of code necessary to complete the implementation work is not large, but you will find it requires a bit of thinking & getting it to work correctly. It will help to sketch things on paper and work through the boundary cases carefully before you write any code.

Submission

- You **MUST** include all three classes **`ALPriorityQueue.java`, `DLPriorityQueue.java`, `BHPriorityQueue.java`** `and interface` **VCPriorityQueue.java, Entry.java** provided.
- You **MUST** include all necessary unit tests to cover your code.
- Presentation (Demo) will be required after the submission.