Berikut adalah salah satu cara untuk mengimplementasikan **Priority Queue** menggunakan struktur data (Binary) Heap. Dengan struktur data Heap, operasi **add()** dan **remove()** pada priority queue menjadi logaritmik. Yang kita implementasikan disini adalah **Minimum Priority Queue** dmenggunakan **Min-Heap**.

Selain menggunakan array primitive milik Java, Anda juga bisa menggunakan Collection seperti Vector yang ada di Java. Silakan Anda rujuk kode implementasi yang ada pada slide kuliah SDA.

```
public interface Queue<E> {
	E peek ();   //findMin ()
	void add (E values);	// add new element
	E remove (); //remove Min
	boolean isEmpty ();
	int size ();
	void clear ();
}
```

```
import java.util.Arrays;
import java.util.NoSuchElementException;

public class MyPriorityQueue<E extends Comparable<? super E>>
                                                    implements Queue<E> {

	private int currentSize;  //banyaknya elemen pada Heap
	private E[] array;              //Heap array
	private static final int DEFAULT_CAPACITY = 100;

	public MyPriorityQueue () {
		currentSize = 0;
		array = (E[]) new Comparable [DEFAULT_CAPACITY];
	}

	//diberikan sebuah array yang belum tentu dalam kondisi Heap Order
	//bangun Heap dari array tersebut !
	public MyPriorityQueue (E[] arr) {
		array = Arrays.copyOf (arr, arr.length);
		currentSize = arr.length;
		heapify ();
	}

	//heap's methods, representasi internal
	//prinsip encapsulation, disembunyikan dari luar
	private int parentOf (int i) {
		return (i - 1) / 2;
	}

	private int leftChildOf (int i) {
		return 2 * i + 1;
	}

	private int rightChildOf (int i) {
		return 2 * (i + 1);
	}
```

```
private void percolateDown (int root) {
	E value = array[root];
	while (root < size ()) {
		int childPos = leftChildOf(root);
		if (childPos < size ()) {
			// choose the smallest child
			if ((rightChildOf(root) < size ()) &&
			    (array[childPos+1].compareTo(array[childPos]) < 0)) {
				childPos++;
			}

			if (array[childPos].compareTo (value) < 0) {
				array[root] = array[childPos];
				root = childPos; // keep moving down
			} else {
				array[root] = value;
				return;
			}
		} else { // at a leaf! insert and halt
			array[root] = value;
			return;
		}
	}
}

private void percolateUp (int leaf) {









}

//heapify ! - versi O(N)
private void heapify () {
	for (_____________________________________________________) {

		percolateDown (i);

	}
}

//doubling, jika array penuh !
private void doubleArray () {
	array = Arrays.copyOf(array, array.length * 2);
}
```

```
	//public interface untuk PriorityQueue
	public E peek () {
		if (isEmpty ()) {
			throw new NoSuchElementException();
		}

		return array[0];    //root, indeks 0 pada Heap
	}

	public void add (E value) {
		if (size () == array.length) {
			doubleArray ();
		}


		...

	}

	//removeMin - Min-HEAP
	public E remove () {









	}

	public boolean isEmpty () {
		return size () == 0;
	}

	public int size () {
		return currentSize;
	}

	public void clear () {
		currentSize = 0;
	}

}
```

Program Utama untuk Uji Coba

```
import java.util.Random;

public class Test {
	public static void main (String[] args) {
		Queue<Integer> pq = new MyPriorityQueue<Integer>();

		//add 300 random elements
		for (int i = 0; i < 300; i++) {
			Integer data = (int) (Math.random() * 1000);
			pq.add (data);
		}

		//remove all elements, lihat apakah terurut
		while (!pq.isEmpty()) {
			System.out.println (pq.remove());
		}
	}
}
```

Silakan buat program untuk menguji apakah method **heapify()** Anda sudah benar.