import java.util.Comparator; import java.util.PriorityQueue;

public class App {

```
/**
 * The main method is the entry point of the program.
 * It demonstrates the usage of a Max Heap Priority Queue
 * by performing various operations. Add, Remove, Change Priority,
 *
 * @param args The command line arguments.
 * @throws Exception If an error occurs during the execution of the program.
 */
public static void main(String[] args) throws Exception {
    // Creating a Max Heap Priority Queue
    //PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder());

    MaxHeap<Integer> pq = new MaxHeap<Integer>();

    // Inserting elements
    int[] elements = {45, 20, 14, 12, 31, 7, 11, 13, 7};
    for (int element : elements) {
        pq.add(element);
    }

    // Displaying the elements of priority queue
    System.out.println("Priority Queue: " + pq);

    // Displaying the node with maximum priority
    System.out.println("Node with maximum priority: " + pq.getMax());
    // Extracting max and displaying priority queue
    pq.removeMax(); pq.heapSort();
    System.out.println("Priority Queue after extracting max: " + pq);

    // Change the priority of element at index 2 to 49
    pq.remove(elements[2]);
    pq.add(49); // 49 is the new element
    //pq.heapSort();
    System.out.println("Priority Queue after changing priority[2]: " + pq);

    // Remove the element at index 3 and display
    pq.remove(elements[3]);
    pq.heapSort();
    System.out.println("Priority Queue after removing element[3]: " + pq);
}

private static <T extends Comparable<? super T>>
```

```
void reheap(T[] heap, int rootIndex, int lastIndex){ boolean done = false; T
orphan = heap[rootIndex]; int leftChildIndex = 2 * rootIndex + 1; while (!done
&& (leftChildIndex <= lastIndex)){ int largerChildIndex = leftChildIndex; int
rightChildIndex = leftChildIndex + 1; if ( (rightChildIndex <= lastIndex) &&
heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0){ largerChildIndex
= rightChildIndex; } // end if if (orphan.compareTo(heap[largerChildIndex])
< 0){ heap[rootIndex] = heap[largerChildIndex]; rootIndex = largerChildIndex;
leftChildIndex = 2 * rootIndex + 1; } else done = true; } // end while
heap[rootIndex] = orphan; }

}
```