```java
/**
 * COMP215-Programming Project 2: Multiple Sort Analysis.
 * HEAPSORT is the Java implementation of the heap sort algorithm.
 * This implementation is based on psuedocode from "CLRS Algorithms."
 * This class is timed using the STOPWATCH class.
 * @author Andrew Parsons
 * @version 27 February 2017
 */
class HeapSort implements Sorter {

    /* --- INSTANCE VARIABLES --- */
    private Stopwatch stopwatch;
    private long elapsedTime;
    private static boolean debug = MainApp.debug;
    private static int heapSize;


    /* --- METHODS --- */

    /** (package-private): SORT begins timing, runs through the algorithm, and then stops timing.
     * SORT returns a sorted array. Implementation of algorithm from "CLRS Algorithms" */
    @Override
    public Comparable[] sort(Comparable[] dataset) {

        stopwatch = new Stopwatch();

        /* INVARIANT: At the start of each iteration of the for loop on line two,
        the subarray A[1…i] is a max-heap containing the i smallest elements of A,
        and the subarray A[i+1…n] containing the n-i- largest elements of A sorted. */

        /*INIT: Initially, i = n. By line 1, A[1…n] is a max-heap containing the n smallest
        elements of A.
        At the start, the subarray A[i+1…n] is empty, and the loop invariant holds trivially. */

        buildMaxHeap(dataset);

        /*MAIN: Before an iteration, A[1…i] is a max-heap containing the i smallest elements of
        A,
        and the subarray A[i+1…n] contains the n-i largest elements of A sorted. This holds by
        the loop invariant. */

        for (int i = dataset.length-1; i >= 0; i--) {

            /*MAIN: By exchanging A[1] with A[i] preserves the order of the elements in A[i…n].
            The elements of A[1…i-1] are all smaller than A[i…n]. */
            swap(dataset, 0, i);

            /*MAIN: Once A.heapsize is decremented, the method call to MAX-HEAPIFY(A,1) makes
            A[1…i-1] a max-heap.
            This restores the loop invariant.*/
            heapSize = heapSize - 1;
            maxHeapify(dataset,0,heapSize);
        }

        /*TERM: When i = 1, A[1…1] is a max-heap containing the smallest element of A,
        and the subarray A[2…n] contains the n-1 largest elements of A sorted. */

        elapsedTime = stopwatch.elapsedTime();
        return dataset;
    }

    /**
     * BUILDMAXHEAP constructs a max-heap by repeatedly calling MAXHEAPIFY
     * @param dataset
     */
    private void buildMaxHeap(Comparable[] dataset) {

        /* INVARIANT: At the start of each iteration of the for loop of lines two and three,
        each node i+1, i+2,…,n is the root of a max-heap. */

        heapSize = dataset.length - 1;
        /*INIT: everything is a leaf, so everything is a max-heap */
        if (debug)
            assert assertionIsMaxHeap(dataset, heapSize);
        int i;
        for (i = heapSize / 2; i >= 0; i--) {
            maxHeapify(dataset, i, heapSize);
            /*MAIN: maxHeapify preserves invariant*/
            if (debug)
                assert assertionIsMaxHeap(dataset, i);
        }
        /*TERM: when i==0, all nodes are max-heaps */
        if (debug) {
            assert assertionIsMaxHeap(dataset,i);
        }
    }

    /**
     * MAXHEAPIFY preserves the max-heap property (each node is greater than or equal to its
     children)
     * @param dataset the array on which MAXHEAPIFY is called
     * @param i the root node
     * @param heapMaxIndex where to begin
     */
    private void maxHeapify(Comparable[] dataset, int i, int heapMaxIndex) {
        int left = leftIndex(i);
        int right = rightIndex(i);
        int largest = i;
        if (left < heapMaxIndex && dataset[left].compareTo(dataset[i]) > 0) {

            largest = left;
        }
        if (right < heapMaxIndex && dataset[right].compareTo(dataset[largest]) > 0) {
            largest = right;
        }
        if (largest != i) {
            swap(dataset, i, largest);
            maxHeapify(dataset, largest, heapMaxIndex);
        }
    }

    /**
     * SWAP simply exchanges elements in an array
     * @param dataset the array on which a swap should occur
     * @param a one of the two indices to element-exchange
     * @param b one of the two indices to element-exchange
     */
    private void swap(Comparable[] dataset, int a, int b) {
        Comparable z = dataset[a];
        dataset[a] = dataset[b];
        dataset[b] = z;
    }

    /**
     * method to establish the LEFT index.
     * @param i the root
     * @return int, the LEFT index
     */
    private static int leftIndex(int i) {
        return 2 * i + 1;
    }

    /**
     * method to establish the RIGHT index.
     * @param i the root
     * @return int, the RIGHT index
     */
    private static int rightIndex(int i) {
        return 2 * i + 2;
    }

    /** (package-private): getter method for timing */
    @Override public long getElapsedTime() {
        return elapsedTime;
    }

    /** check if subarray is a max heap (called by assert) */
    private static boolean assertionIsMaxHeap(Comparable[] dataset, int index) {
        if (dataset[index].compareTo(dataset[index*2]) == -1) {
            return false;
        } return true;
    }

    @Override // DO NOT USE THIS METHOD
    public Comparable[] sort(Comparable[] dataset, int indexBegin, int indexEnd) {
        return new Comparable[0];
    }
}
```