# Heap Sort

The heapSort is similar to the selectionSort, in that we look for the maximum value and swap it to the end. However, a maxHeap finds the maximum faster than the selectionSort, because the maximum is always at the root, for a performance of O(1). Now the heap needs to be re-formed, which we can do by repeated swapping down. (So the heapSort is also similar to the bubble sort, but works on a tree.) The reheap expense is O(log n). Since we must do this for each item, the heap sort is O(n log n).

Previous O(n log n) sorts were the MergeSort and the QuickSort. Let's compare all three sorts.

The **MergeSort** gives stable performances. It usually requires temporary storage space.

The **QuickSort**'s performance is "unstable" because it can degrade to O(n2) when you choose a bad pivot. Despite that problem, the QuickSort is actually faster than the other two sorts when sorting random arrays. The QuickSort was invented in 1962.

The **HeapSort** is stable and does not use any temporary storage space. It is not good for small n, because of the “overhead” of extra time it needs to rearrange random elements into the heap order. The HeapSort was invented in 1964.

This table summarizes the characteristics of all our O(n log n) sorts:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | best | average | worst | Extra Space | Stable | Overhead |
| **MergeSort** | O(n log n) | O(n log n) | O(n log n) | yes | yes | no |
| **QuickSort** | O(n log n) | O(n log n) | O(n2) | no | no | no |
| **HeapSort** | O(n log n) | O(n log n) | O(n log n) | no | yes | yes |

# Coding the HeapSort

# If you begin with random numbers, then the heap sorting takes 2 distinct phases. First, make the array into a maxHeap (this phase is the "overhead"). Second, sort the heap.

# First phase. Make the random array into a maxHeap. The basic idea is to visit each subtree and form each subtree into a heap. The most efficient algorithm starts at the middle of the array, does heapDown on that subtree, moves to the left in the array, does heapDown, and so on. How clever! Using the algorithm, show how the data moves as it turns the random array (on the left) into a maxheap (on the right):

1

2

3

4

5

6

8

9

7

1

2

3

4

5

6

8

9

7

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 30 | 1 | 2 | 16 | 80 | 84 | 85 | 17 | 99 |  |  |  | 99 | 80 | 85 | 17 | 30 | 84 | 2 | 1 | 16 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# Assume you have a working heapDown method. Write the code for makeHeap:

# public static void makeHeap(double[] array, int lastIndex) {

Phase 2. Now the elements are in a heap! Let’s sort it. You know that the greatest element is at the root. Swap the root and the last element. Then walk **down** the heap from the top, swapping with the larger child, until it is either in place or at the end. (You may do this either iteratively or recursively.) Reduce the last index by one, swap, and reheap down. After N-1 iterations, the array is ordered.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |  |  |  |  |
|  | 99 | 80 | 85 | 17 | 30 | 84 | 2 | 16 | 1 | start | | | |
|  | 1 | 80 | 85 | 17 | 30 | 84 | 2 | 16 | 99 | swap first and last | | | |
|  | 85 | 80 | 1 | 17 | 30 | 84 | 2 | 16 | 99 | heapDown | | | |
|  | 85 | 80 | 84 | 17 | 30 | 1 | 2 | 16 | 99 | heapDown continued | | | |
|  | 16 | 80 | 84 | 17 | 30 | 1 | 2 | 85 | 99 | swap first and last | | | |
|  | 84 | 80 | 16 | 17 | 30 | 1 | 2 | 85 | 99 | heapDown | | | |
|  | 2 | 80 | 16 | 17 | 30 | 1 | 84 | 85 | 99 | swap first and last | | | |
|  | 80 | 2 | 16 | 17 | 30 | 1 | 84 | 85 | 99 | heapDown | | | |
|  | 80 | 30 | 16 | 17 | 2 | 1 | 84 | 85 | 99 | heapDown continued | | | |
|  | 1 | 30 | 16 | 17 | 2 | 80 | 84 | 85 | 99 | swap first and last | | | |
|  | 30 | 1 | 16 | 17 | 2 | 80 | 84 | 85 | 99 | heapDown | | | |
|  | 30 | 17 | 16 | 1 | 2 | 80 | 84 | 85 | 99 | heapDown continued | | | |
|  | 2 | 17 | 16 | 1 | 30 | 80 | 84 | 85 | 99 | swap first and last | | | |
|  | 17 | 2 | 16 | 1 | 30 | 80 | 84 | 85 | 99 | heapDown | | | |
|  | 1 | 2 | 16 | 17 | 30 | 80 | 84 | 85 | 99 | swap first and last | | | |
|  | 16 | 2 | 1 | 17 | 30 | 80 | 84 | 85 | 99 | heapDown | | | |
|  | 1 | 2 | 16 | 17 | 30 | 80 | 84 | 85 | 99 | swap first and last | | | |
|  | 2 | 1 | 16 | 17 | 30 | 80 | 84 | 85 | 99 | heapDown | | | |
|  | 1 | 2 | 16 | 17 | 30 | 80 | 84 | 85 | 99 | swap first and last -- Done | | | |

# Ordering this array took 18 steps. By the O(n\*log n) formula, 9 \* log2 9 = 28.53.

# Assignment

Part 1: Given a heap (the 9 numbers shown above), display it, heap sort it, and display it again. Use these headers:

public void display(double[] array)

public void sort(double[] array)

public void swap(double[] array, int a, int b)

public void heapDown(double[] array, int k, int lastIndex)

public boolean isSorted(double[] arr)

Part 2: Generate 100 random numbers between 1 and 100, formatted to 2 decimal places, make a heap, sort it, and display it. You will need to write the methods:

public double[] createRandom(double[] array)

public void makeHeap(double[] array, int lastIndex)