Heapsort Algorithm For Arrays, Reading

The first n log n algorithm we'll study is heapsort. It works by arranging n values in an array into a heap, then dequeueing the values. Enqueueing into a heap is O(log n), and we have to do n of them. That's n log n. Then the next step is to deque -- each deque is O(log n) and there are n of them -- another (added) n log n. So the process is n log n.

The algorithm works on arrays, and it does so without using any extra arrays to store intermediate values. It just does compares and swaps in the existing array.

Arrange An Existing Array Into A Heap

Recall that a "heap" is a binary tree representation in which each child is less than or equal to its parent. And it's stored in an array with no "holes", filling each row from the left to the right, one row at a time. Refer to the earlier module on priority queues where heaps were introduced.

Basically, you separate the array into a "heapified" left side, and a the remainder values to the right. The left side starts with one value in it (the 11 below):

11 6 4 1 3 7 12 8

Then enqueue the 6. Knowing what we know about priority queues, we get this:

11 6 4 1 3 7 12 8

The 4, 1, and 3 are enqueued without any swapping in this example. But when 7 gets enqueued, it's parent is 4, and it swaps, with this result:

11 6 **7** 1 3 **4** 12 8

12's next -- it's parent is 7. It swaps with 7, and then swaps with it's new parent, 11:

12 6 **11** 1 3 4 **7** 8

Finally, 8. It's parent is 1. It swaps with 1, and then swaps with it's new parent, 6, and we had a heap:

12 **8** 11 **6** 3 4 7 **1**

Borrowing the enqueue algorithm from the priority queue module, and placing it in a for-loop to enqueue the values at indexes 1 through size - 1, we get this:

```
for index = 1; index < size; index++
  start loop
    parentIndex = (index + 1) / 2 - 1
    if parentIndex < 0, exit loop
    if value at parentIndex >= value at index, exit loop
    swap values at parentIndex and index
    set index = parentIndex
  repeat to top of loop
```

Note that array doubling is not included because the full array already exists!

Dequeue Into A Sorted Array

Now dequeue the top-most (highest) value, vacating the last value in the array, and putting the dequeued value there. Repeat, like this:

12 8 11 6 3 4 7 1

Swap the 12 into the end of the array (where it belongs). With no place to put the 1 that's already there, one idea is to put it where the 12 was. Now the 12 is in its correct place, and the remaining values are in an in-need-of-repair heap:

1 8 11 6 3 4 7 **12**

Now fix the heap, starting with the 1 and its (new) children, 8 and 11. These 3 are candidates for the top spot. 11 wins, so swap them:

11 8 **1** 6 3 4 7 12

We're not done fixing -- starting next with the newly-relocated 1 and its (new) children, 4 and 7. These 3 are candidates for the top spot. 7 wins, so swap them:

11 8 **7** 6 3 4 **1** 12

Now the heap's fixed, because with the last swap, 1's now a "leaf" -- that is, is has no children in the tree structure. Now repeat by moving the highest remaining value into it's proper location -- where the 1 is. After fixing the remaining heap with repeated swaps, we get this:

8 6 7 1 3 4 11 12

Continue until the sorted right-half of the array takes over the whole array. Here's that algorithm:

```
for index = size - 1; index > 0; index--
  swap values at zero and index
  parentIndex = 0
  start loop
    leftChildIndex = 2 * parentIndex + 1
    rightChildIndex = 2 * parentIndex + 2
    if leftChildIndex >= index, exit loop // leaf at parentIndex
    if rightChildIndex >= index // one-child for parentIndex
      if value at leftChildIndex <= value at parentIndex, exit loop
      swap values at leftChildIndex and parentIndex
      parentIndex = leftChildIndex
    else if value at rightChildIndex < value at leftChildIndex
     if value at leftChildIndex <= value at parentIndex, exit loop
     swap values at leftChildIndex and parentIndex
     parentIndex = leftChildIndex
   else
     if value at rightChildIndex <= value at parentIndex, exit loop
     swap values at rightChildIndex and parentIndex
     parentIndex = rightChildIndex
 repeat to top of loop
```

And the array's now sorted!

More Than One Way

There are other algorithms for enqueueing and dequeueing in a heap sort, that are equally valid and have the same big oh characteristics. For example, the dequeue shown here did a three-way compare among a parent and its children. Another way does *not* elevate the displaced value to the top of the heap -- it holds it in a temporary position and only has to compare children to promote to the vacated parent spot. But this results in a "hole" at the bottom of the heap when the bottom level is reached in the promotion sequence, requiring *another* loop to fix it. Which is better? Hard to say -- they are just... *different*.

Watch This! (https://www.youtube.com/watch?v=EreoMaOBTzE)

(https://www.youtube.com/watch?v=EreoMaOBTzE)