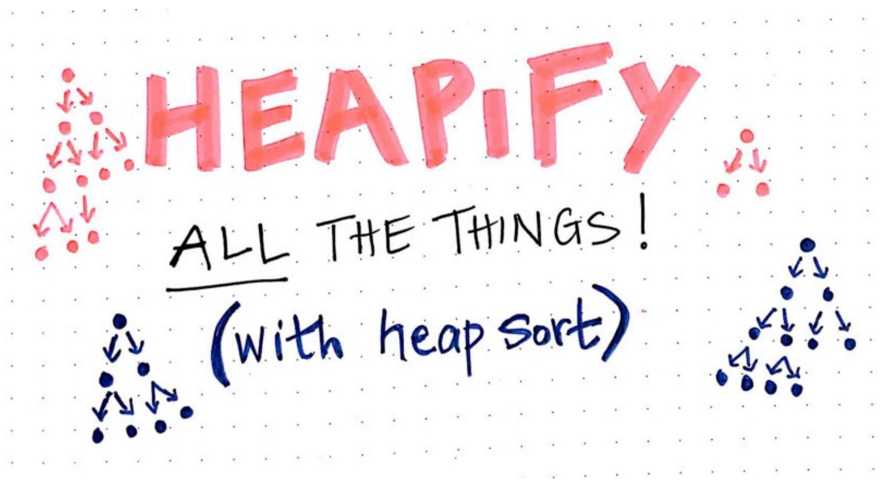
Vaidehi Joshi Follow
Writing words, writing code. Sometimes doing both at once.
Jul 13, 2017 · 11 min read

Heapify All The Things With Heap Sort



Heapify All Of The Things!

Someone once told me that everything important in computer science boils down to trees. Literally just trees. We can use them to build things, parse things, and interpret things (yes, there might be some foreshadowing happening here, don't worry about it if it doesn't make any sense to you just yet, because soon, it will!). And we can even use them to—you guessed it!—*sort things*.

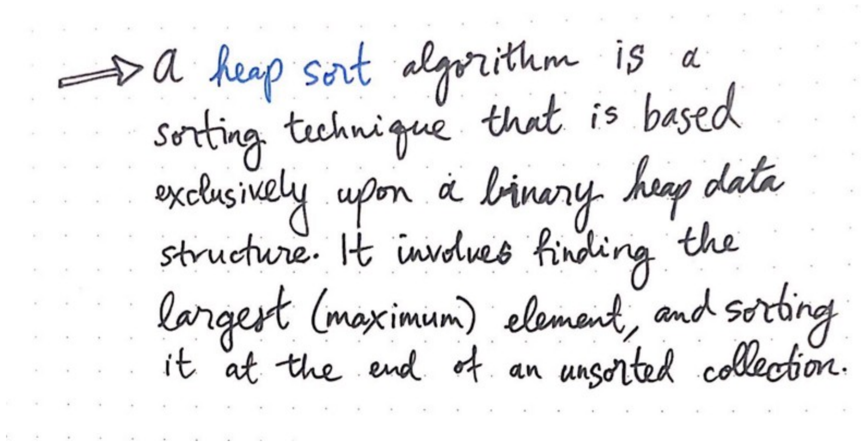
Ah, sorting. We've done so much of it in the past few weeks, but we're now nearing the end of our sorting adventures. However, it is impossible and unfair to talk about sorting without talking about a special kind of sorting that uses the newest data structure in our data structure tool belt.

We recently learned to love heaps, a special kind of binary tree that follows a strict set of rules, and are used to implement things like priority queues and background jobs. But these aren't the only things that heaps are good for. It turns out that binary heaps are often used for no other purpose than *efficient sorting*. Many programs will rely on heap sort since it happens to be one of the most efficient ways to sort an array. And now that we know what a heap is, we can try to understand why it works so well when it comes to the problem of sorting!

Heapify all the things!

Before we dive into heap sort, let's make sure that we have heaps straight in our heads. We might remember that a ***heap*** is really nothing more than a binary tree with some additional rules that it has to follow: first, it must always have a heap structure, where all the levels of the binary tree are filled up, from left to right, and second, it must either be ordered as a max heap or a min heap. For the purposes of heap sort, we'll be dealing exclusively with ***max heaps***, where every parent node (including the root) is greater than or equal to the value of its children nodes.

Okay, let's get to answering the question of the hour: how do we sort using heaps? Well, in order to answer that question, we'll need to understand what a heap sort algorithm *is*, first!

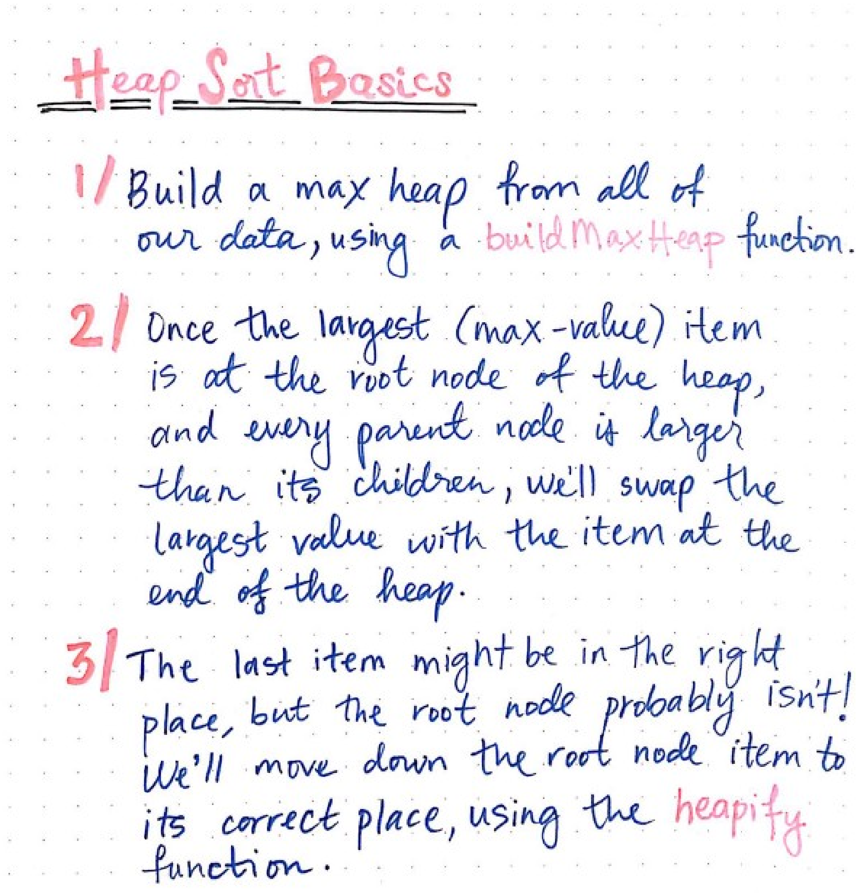


a heap sort algorithm is a sorting technique that is based exclusively upon a binary heap data structure. It involves finding the largest (maximum) element, and sorting it at the end of an unsorted collection.

Heap sort: a definition

A ***heap sort algorithm*** is a sorting technique that leans on binary heap data structures. Because we know that heaps must always follow a specific order, we can leverage that property and use that to find the largest, maximum value element, and sequentially sort elements by selecting the root node of a heap, and adding it to the end of the array.

We already know that heap sort is an efficient way of sorting an unsorted array; but what does an array have to do with a heap? And how do we sort an array using a heap? Well, there are are three key steps to how this actually works in practice. We'll look at these in more depth in a moment, but let's take a high level glance at what these three steps are, first.



Heap Sort Basics

1/ Build a max heap from all of our data, using a buildMaxHeap function.

2/ Once the largest (max-value) item is at the root node of the heap, and every parent node is larger than its children, we'll swap the largest value with the item at the end of the heap.

3/ The last item might be in the right place, but the root node probably isn't! We'll move down the root node item to its correct place, using the heapify function.

The basics of heap sort

1. To start, we have an unsorted array. The first step is to take that array and turn it into a heap; in our case, we'll want to turn it into a max heap. So, we have to transform and build a max heap out of our unsorted array data. Usually, this is encapsulated by a single function, which might be named something like `buildMaxHeap`.

2. Once we have our array data in a max heap format, we can be sure that the largest value is at the root node of the heap. Remember that, even though the entire heap won't be sorted, if we have built our max heap correctly and without any mistakes, every single parent node in our heap will be larger in value than its children. So, we'll move the largest value—located at the root node—to the end of the heap by swapping it with the last element.

3. Now, the largest item in the heap is located at the last node, which is great. We know that it is in its sorted position, so it can be removed from the heap completely. But, there's still one more step: making sure that the new root node element is in the correct place!

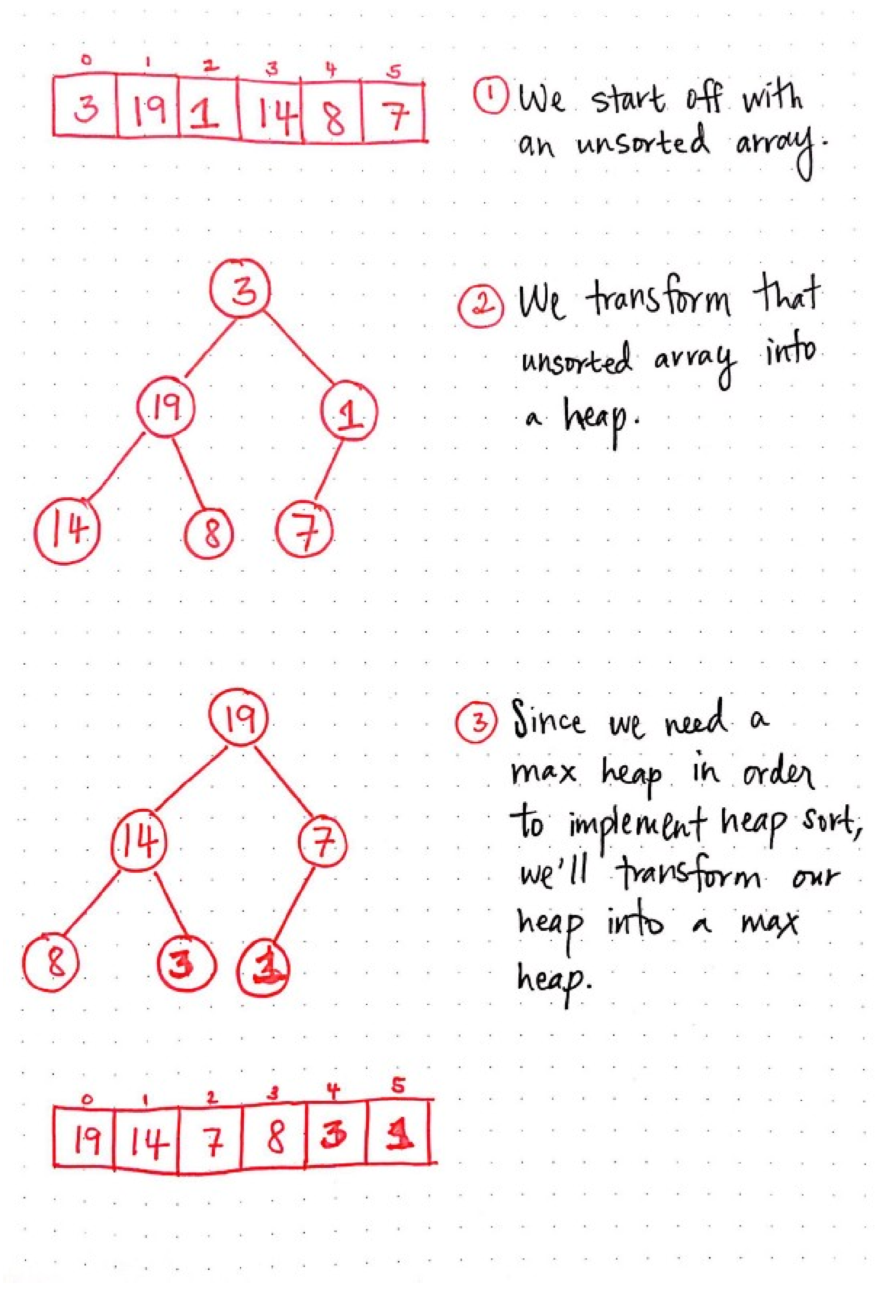
> It's highly unlikely that the item that we swapped into the root node position is in the right location, so we'll move down the root node item down to its correct place, using a function that's usually named something like `heapify`.

And that's basically it! The algorithm continues to repeat these steps until the heap is down to just one single node. At that point, it knows that all the elements in the unsorted array are in their sorted positions, and that the last node remaining will end up being the first element in the sorted array.

Okay, I know I said that these are the only three steps to heap sort. But if these three steps seem confusing, don't worry; they can be pretty complicated and difficult to understand until you see them play out in action. In fact, I think this algorithm makes much more sense with an illustrated example. Since heaps are a type of tree, it helps to visualize them, the same way we do with binary trees. So let's do that right now!

Have you ever looked under heap sort's hood?

Alright, it's time for my absolute favorite part of learning heap sort: drawing it out! Hooray! In order to understand what's going on under the heap sort hood, we'll work with a small, unsorted dataset.



0 1 2 3 4 5
3 19 1 14 8 7
① We start off with an unsorted array.
② We transform that unsorted array into a heap.
③ Since we need a max heap in order to implement heap sort, we'll transform our heap into a max heap.
0 1 2 3 4 5
19 14 7 8 3 1

Implementing heap sort, part 1

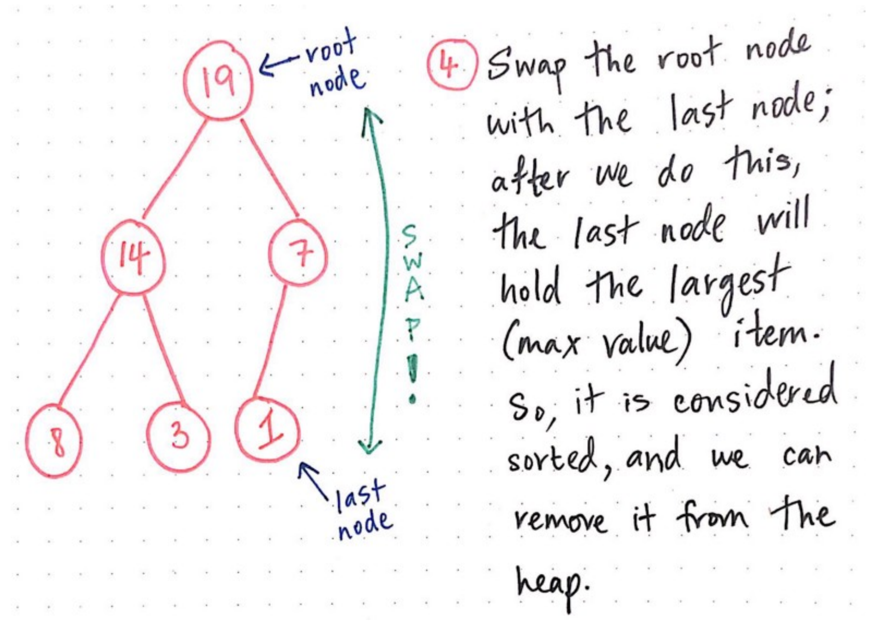
We'll start off with an unsorted array with five elements that are super out of order: `[3, 19, 1, 14, 8, 7]`.

Remember that, since this is heap sort that we're working with, we're going to need to turn that array into a heap, to start.

In the illustration shown here, you can see that the array has been morphed into a tree—it's not a heap just yet because it's still not in any max or min heap order! We can see that this is the case because `3` isn't the largest or smallest element, and yet, it is the root node at the moment. This is just a tree, with the elements from the array directly translated into a binary tree format.

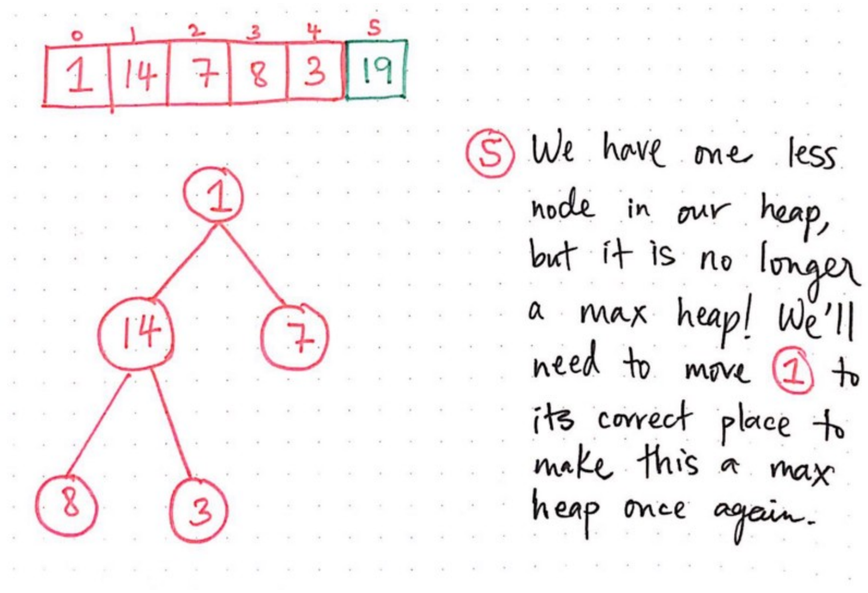
But, since we need to deal with a max heap, we'll need to transform our structure from a binary tree into a max heap. Notice how, in the max heap, the parent nodes are all larger than their children. Last week, we learned the algorithms that allow us to determine the child nodes from the index of an array; this week, we're seeing them in action. Those algorithms are what we are using to transform this array into a tree, and then into a heap.

Okay, now we have an *actual max heap*. Great! Now for the actual work of sorting.


19
root node
14
7
8
3
1
last node
SWAP!
4
Swap the root node with the last node; after we do this, the last node will hold the largest (max value) item. So, it is considered sorted, and we can remove it from the heap.

Implementing heap sort, part 2

Since we know that the largest element is at the root node, we know that we'll need to put it at the very end of the array, in the last available index spot. So, we'll swap the root node with the last node. Once we make this swap, our last node will hold the largest, max value item.

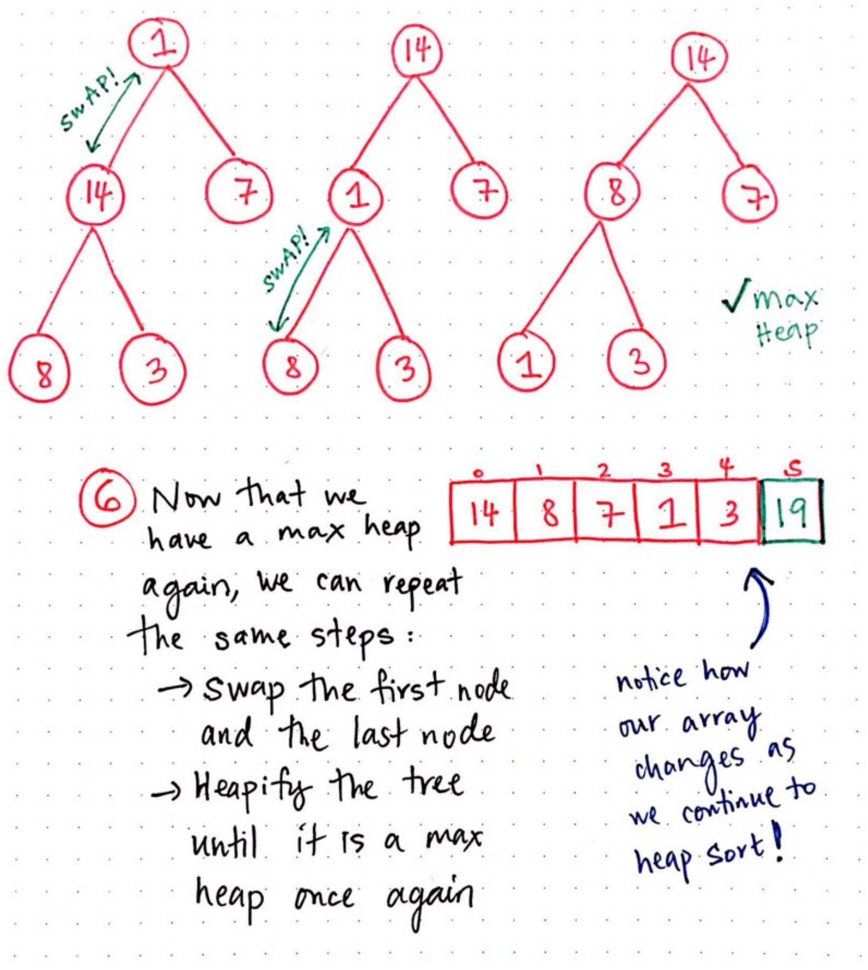

0 1 2 3 4 5
1 14 7 8 3 19
1
14 7
8 3
5 We have one less node in our heap, but it is no longer a max heap! We'll need to move 1 to its correct place to make this a max heap once again.

Implementing heap sort, part 3

Cool! Now we can see that `19` , the largest element, which used to be the root node, is now at the last position in the array. And, since it is effectively "sorted" relative to the rest of the elements, we can remove it completely from the heap.

Now, the good news is that we have one less node in our heap to sort! The bad news? Our heap isn't actually a heap anymore: it's totally violating its heap order rule, since it's not a max heap. Notice that `1` is the root node, but it's definitely *not* larger than it's two children nodes, `14` and `7` . So, we'll need to move it down to its correct place in the tree.

Let's heapify this tree and make it a max heap again!

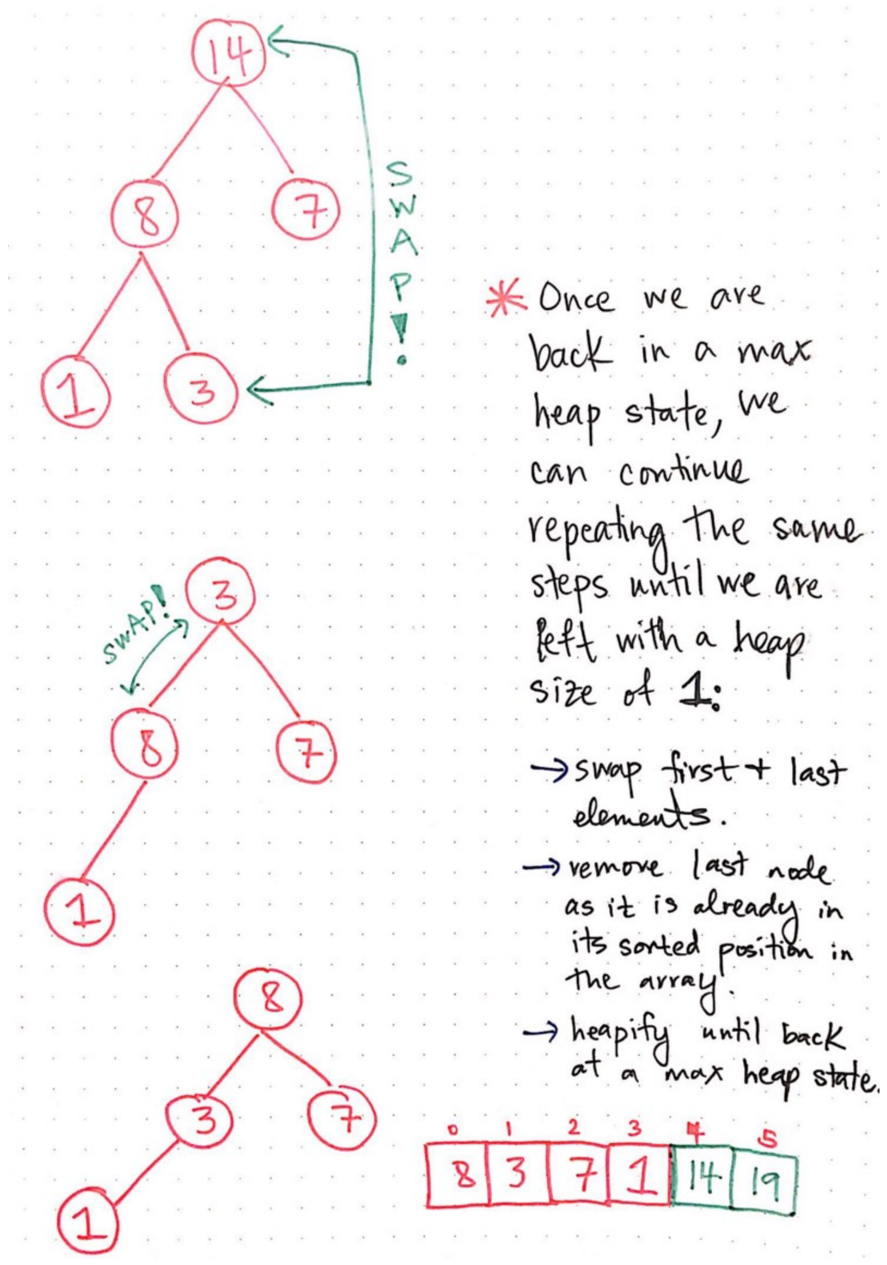

SWAP!
1
14
7
8
3
SWAP!
14
1
7
8
3
14
8
7
1
3
✓max Heap
6 Now that we have a max heap again, we can repeat the same steps:
→ Swap the first node and the last node
→ Heapify the tree until it is a max heap once again
0 1 2 3 4 5
14 8 7 1 3 19
notice how our array changes as we continue to heap sort!

Implementing heap sort, part 4

Awesome! In the illustration above, we can see that we first swapped `1` and `14`, and then we swapped `1` and `8`. Now, we're back to a proper max heap. We can repeat the same steps we did when sorting the element `19`:

→ We'll first swap the first and last nodes.
→ Then, we'll heapify the tree until it's a proper max heap again.

Let's do that with our new root node, the element `14`. Here's what our next two steps would look like:

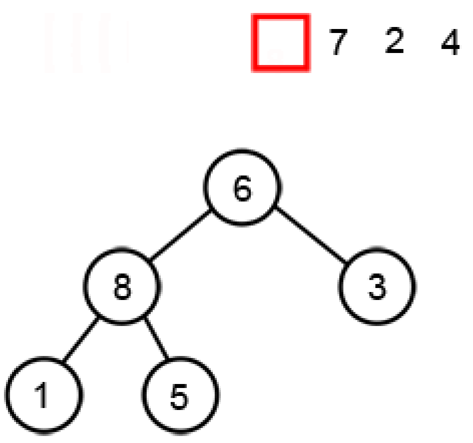

SWAP!
Once we are back in a max heap state, we can continue repeating the same steps until we are left with a heap size of 1:
→ swap first + last elements.
→ remove last node as it is already in its sorted position in the array.
→ heapify until back at a max heap state.
SWAP!
0 1 2 3 4 5
8 3 7 1 14 19

Implementing heap sort, part 5

Rad! We swapped the first and last nodes, and then we removed the last node, `14`, since it was in its sorted position. The only thing we had to do next was move the root node into its correct location, and heapify the element `3` until we were back at a max heap state.

We would continue doing this three more times. Eventually, we'd be left with just `1`, the last node in the heap. At this point, the heap sort algorithm would be finished, and we'd know that `1` would be the first element in the array, and we'd know that the array was finally sorted.

Here's a great visualization of the entire process we just walked through. Notice how, with each iterative sort, the largest unsorted element ends up in its correct place in the heap, and then in the array.



Heap sort visualized, Wikimedia Commons

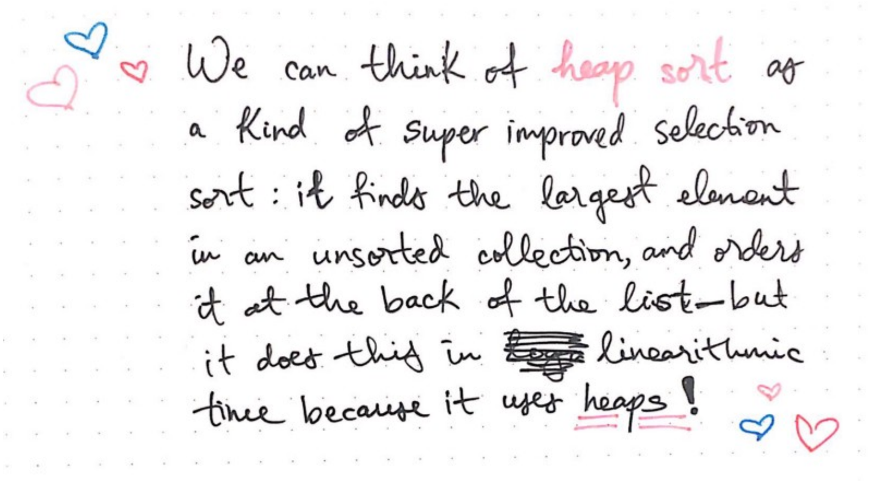
Heap sort: what is it good for?

When I was first reading about heap sort, something about the algorithm seemed oddly familiar to me. It was only after illustrating heap sort that I realized where my feeling of déjà vu was coming from: heap sort was almost exactly like selection sort! You might remember from earlier in the series that selection sort is a sorting algorithm that sorts through a list of unsorted items by iterating through a list of elements, finding the smallest one, and putting it aside into a sorted list. It continues to sort by finding the smallest unsorted element, and adding it to the sorted list.

Doesn't that sound a whole lot like heap sort, but just reversed?

> *It turns out that heap sort is a lot like selection sort in its logic: both algorithms find either the smallest or largest element, "select" it out, and put that item in its correct location in the sorted list.*

However, as similar as they are, heap sort is *much* better than selection sort in one massive way: its performance! Heap sort is basically a super-improved version of selection sort. Yes, it does find the largest element in an unsorted collection and orders it at the back of the list—however, it does all of this work so much faster than selection sort would!


We can think of heap sort as a kind of super improved selection sort: it finds the largest element in an unsorted collection, and orders it at the back of the list—but it does this in [illegible] linearithmic time because it uses heaps!

Heap sort: kind of like selection sort, but so much better!

Okay, so just *how* much faster is heap sort? And *why* is it faster?

Well, let's take a look at the code. There are various implementations of heap sort, and the code below is adapted from Rosetta Code's JavaScript implementation of heap sort. You'll remember that heap sort has two important parts to it: `buildMaxHeap` and `heapify`. We can see them in action in the version of `heapSort` below.

```
function heapSort(array) {
  // Build our max heap.
  buildMaxHeap(array);

  // Find last element.
  lastElement = array.length - 1;

  // Continue heap sorting until we have
  // just one element left in the array.
  while(lastElement > 0) {
    swap(array, 0, lastElement);
```

The `buildMaxHeap` function does the work of actually creating the max heap. Notice that even this function calls out to `heapify`, which does the work of moving one element at a time down to its correct location in the heap.

```
function buildMaxHeap(array) {
  var i;
  i = array.length / 2 - 1;
  i = Math.floor(i);

  // Build a max heap out of
  // all array elements passed in.
  while (i >= 0) {
    heapify(array, i, array.length);
```

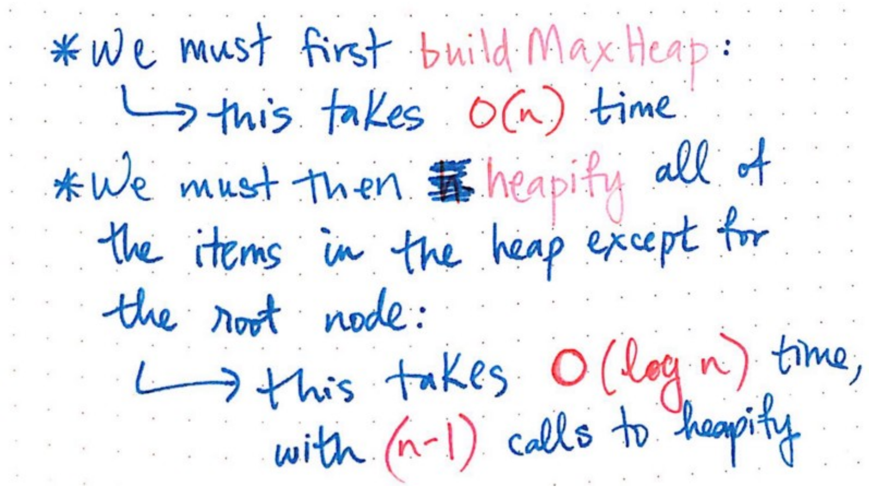
The `heapify` function is pretty important, so let's look at that. Notice that it is relying on the algorithms to determine the left and right child of a node, which we discussed last week when we first learned about heaps.

```
function heapify(heap, i, max) {
  var index, leftChild, righChild;

  while(i < max) {
    index = i;

    leftChild = 2*i + 1;
    righChild = leftChild + 1;

    if (leftChild < max && heap[leftChild] > heap[ind
      index = leftChild;
    }

    if (righChild < max && heap[righChild] > heap[ind
      index = righChild;
    }

```

And last but not least, the `swap` function, which we've seen before in other sorting algorithms, but is worth looking at quickly to remind ourselves what it does:

```
function swap(array, firstItemIndex, lastItemInde) {
  var tmp = array[firstItemIndex];

  // Swap first and last items in the array.
  array[firstItemIndex] = array[lastItemInde];
```

Okay, now that we've got some context for how these functions interact and invoke one another, let's get back to our original question of *how* and *why* heap sort is so much more efficient than selection sort! If we look deeply at the code, we'll notice two things: first, we must build the max heap once, passing in all of the elements of the array to it; second, we have to heapify all of the items in the heap again and again, with the exception of the first root node element.



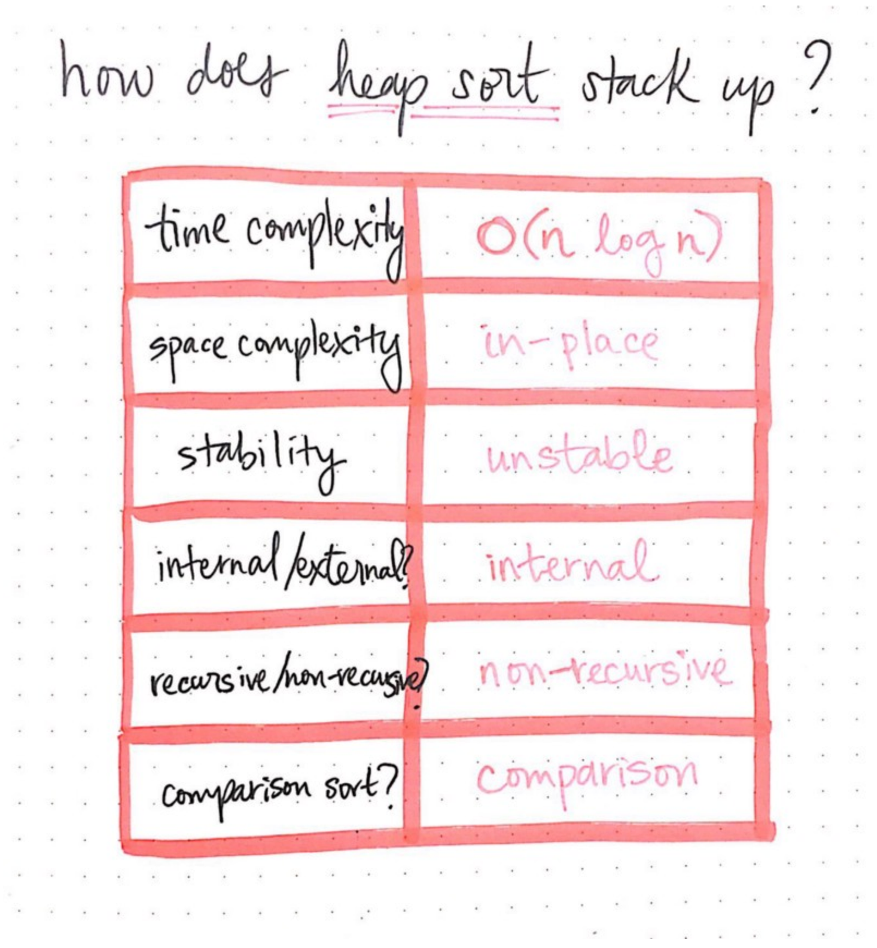
* We must first build MaxHeap:
↳ this takes O(n) time
* We must then heapify all of the items in the heap except for the root node:
↳ this takes O(log n) time, with (n-1) calls to heapify

Understanding heap sort's time complexity

These two observations are actually the key to the question of *how* and *why* heap sort is as fast as it is. Calling `buildMaxHeap` takes *O(n)* time, since every single item must be added to the heap, and a larger amount of elements mean a larger heap. However, remember that we are dealing with a binary tree, and binary trees are logarithmic in nature. So, even though we have to call `heapify` again and again, invoking this function is actually fairly fast, since it will run in logarithmic time, or *O(log n)*.

The combination of these two time complexities is something we've already seen before! Heap sort runs in *linearithmic* time, or in Big O notation, *O(n log n)*. So, even though heap sort *seems* so much like selection sort, it's a lot faster! Selection sort runs in quadratic time, or *O(n^2)*, which is so much less efficient than linearithmic time.

Let's quickly look at the other ways that heap sort compares to other sorting algorithms.



how does heap sort stack up?

time complexity	O(n log n)
space complexity	in-place
stability	unstable
internal/external?	internal
recursive/non-recursive?	non-recursive
comparison sort?	comparison

How does heap sort stack up?

Heap sort transforms the array that pass to it as it sorts; unlike some sorting algorithms, it doesn't create an entirely separate copy of the input data. This makes it an ***in-place*** sorting algorithm. Heap sort also doesn't need external memory, and is an ***internal*** sorting algorithm. It runs iteratively (and is thus ***non-recursive***), and compares two elements at a time when it swaps and calls the heapify function, making it a ***comparison*** sort algorithm.

However, because of the nature of heaps and the heapify function, if there are duplicate elements, we can't rely on elements maintaining their order! So, heap sort is ***unstable***; this is a major differentiator between merge sort and heap sort, which each rely on tree structures to perform so efficiently. However, merge sort wins in the battle of stability, whereas heap sort fails in this category.

Despite their differences, merge sort and heap sort can agree on one thing: without binary trees, they'd both be lost!

Resources

There are some really fantastic course notes and lectures on heap sorting, as well as a few good video tutorials. I did some googling so that you wouldn't have to! Here are some great places to start if you're interested in learning more about heap sort.

1. Introduction to Algorithms: Heap Sort, MIT
2. Algorithms: Heap Sort, Professor Ching-Chi Lin
3. Heap sort, Growing with the Web
4. Heap sort in 4 minutes, Michael Sambol
5. Heap sort: Max heap, strohtennis