Heaps

Heaps are **binary trees** that satisfy the **max-heap property** (or min-heap, depending on the type of heap). **Binary trees** are collections of nodes arranged into a tree with each node linking to at most two children. The **max-heap property** specifies that each node must be greater than or equal to its two children.

Operations

We can thus define several operations on heaps:

- MAX(H): Return the value of the maximum node in H.

  Implementation: Return the top node of $H \rightarrow O(1)$.

- MAX-HEAPIFY(H, x): For a node x, fix the subtree rooted in x, if the root can potentially violate the max-heap property.

  Implementation: Find the maximum of the two children of x, and swap x with that node. That node and its two children (x and its other child) are now correct. It's possible that the subtree which now has x as the root is now incorrect, so recursively call MAX-HEAPIFY on that subtree. The runtime is proportional to the height of the tree, since we might "trickle down" the entire tree $\rightarrow O(\log n)$.

- EXTRACT-MAX(H): Remove the maximum value node in H (and return its value).

  Implementation: Swap the root node (the max) of H with the last leaf in the array representation (more on this later). The max node, which is now a leaf, may be removed and returned without upsetting the tree structure. Finally, call MAX-HEAPIFY on the new root. In total $\rightarrow O(\log n)$.

- INCREASE-KEY(H, x, k): Increase the value of the node x to k.

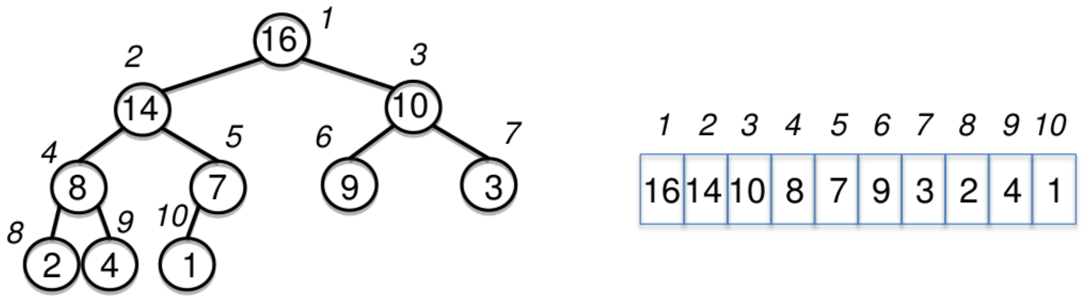
  Implementation: Increase the key of x to k, then "trickle up" the node by swapping with its parent as long as the parent is smaller than it. This takes time proportional to the height of the tree $\rightarrow O(\log n)$.

- INSERT(H, k): Insert a node with value k into H.

  Implementation: Add a leaf x with value $-\infty$, then run INCREASE-KEY$(H, x, k) \rightarrow O(\log n)$.

Array representation

In most cases, the underlying storage of a heap is actually an array. To allow an array representation, we require that the binary tree must be filled from left to right. As we increase index, we move from left to right, and top to bottom (as you would when reading an english book):



This representation allows us to mathematically define parent and children links without any additional structure:

- $\text{PARENT}(x) = \lfloor x/2 \rfloor$
- $\text{LEFT}(x) = 2x$
- $\text{RIGHT}(x) = 2x + 1$

Remark: in Python (and almost all other programming languages as well) lists are indexed starting from zero. So, for a node i the children will be $2i + 1$ and $2i + 2$, respectively. Moreover, the parent is $\lfloor (i-1)/2 \rfloor$.

Examples:

- $[5, 3, 2, 1, 2, 1]$ is a valid heap, but is not sorted;
- $[10, 8, 5, 6, 1, 1, 2, 1, 1]$; decrease 10 to 4;
- the same heap; increase the first 1 to 9;
- the same heap; perform EXTRACT-MAX.

Remark. Note that it is easy to come up with an approach that performs one of the EXTRACT-MAX and INSERT operations in $O(1)$ and the other one in linear time in the number of elements. However, using heap data structure we can perform both operations in $O(\log n)$.

A remark about $\log n$ vs n

Let us do the following experiment. We add n random values to a list and then perform EXTRACT-MAX n times. The first implementation is naive: it just performs linear scan of a list to find the current maximum, while the second implementation uses binary heaps. The following table summarizes the running times for various values of n (for all n operations, that is, we compare $n \log n$ vs n^2).

n	slow implementation	fast implementation
1024	0.14 seconds	0.08 seconds
2048	0.35 seconds	0.08 seconds
4096	1.15 seconds	0.10 seconds
8192	4.33 seconds	0.11 seconds
16384	17.36 seconds	0.14 seconds
32768	71.24 seconds	0.22 seconds
1048576	**extrapolated:** around 20 hours!	6.63 seconds

Building heaps

To work with heaps, we must first be able to construct them from an arbitrary array of elements. We define BUILD-MAX-HEAP(A) to accept an arbitrary array A and reorder the elements to produce a valid max-heap:

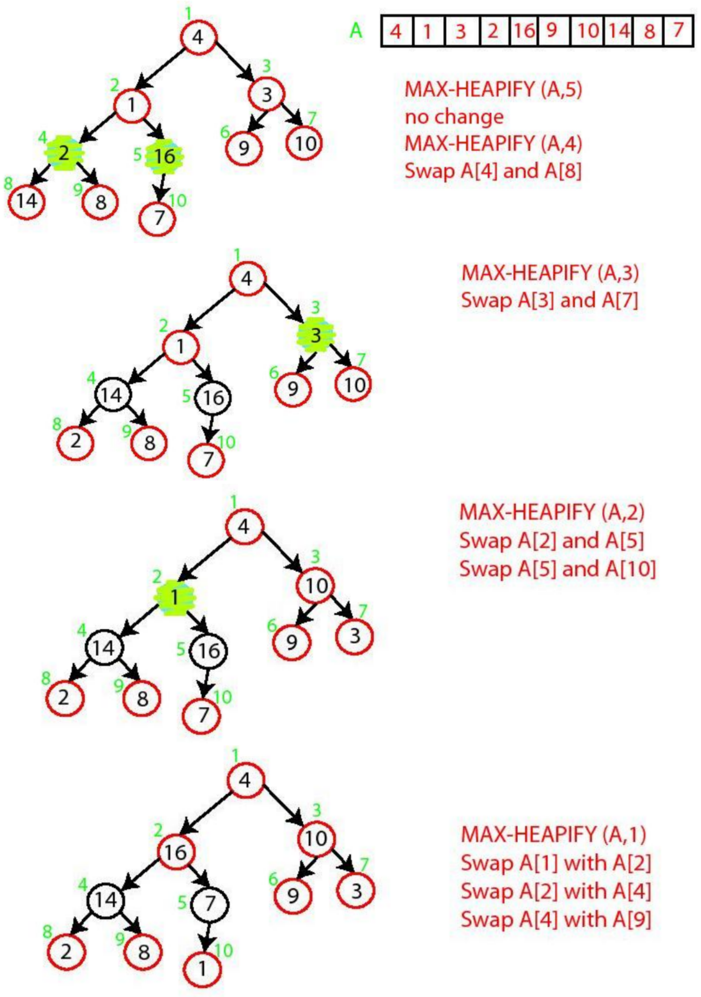
- The last $\lceil n/2 \rceil$ elements are leaves. Thus they are already "heapified".
- Work our way upwards from the leaves and call MAX-HEAPIFY on each successive node. Because all nodes below it have already been max-heapified, this is a valid operation. Note that moving upwards through the tree in this order is the same as moving linearly through the array in reverse order.
- Once we've iterated through all the nodes, the array is max-heapified!

At first glance, the runtime seems to be $n \times O(\log n) = O(n \log n)$. At closer look, we can see that we're actually over-counting, because the cost is not $O(\log n)$ at all levels, it's actually $O(h)$, where h is the height of the particular node. We can thus do some math to get a tighter runtime analysis:

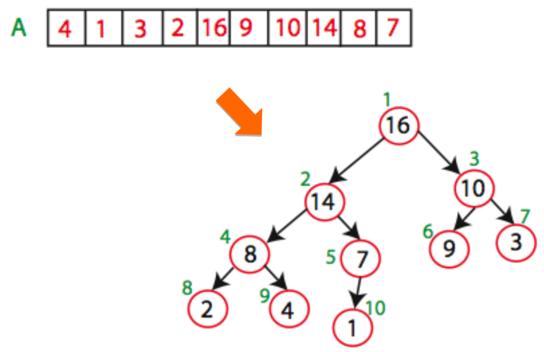
$$\begin{aligned}\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} O(h) &= O(n \sum_{h=0}^{\log n} \frac{h}{2^{h+1}}) \\ &\le O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) \\ &= O(n)^1\end{aligned}$$

[1]The summation turns out to be equal to a constant. See CLRS Appendix A: A.8 for more details.

We can see a pictoral example of this operation on the array $[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$ below:



Which results in a final output of:



Exercise 1 – Manually perform BUILD-MAX-HEAP on $A = [5, 2, 4, 8, 9]$, to get a sense for how BUILD-MAX-HEAP works.

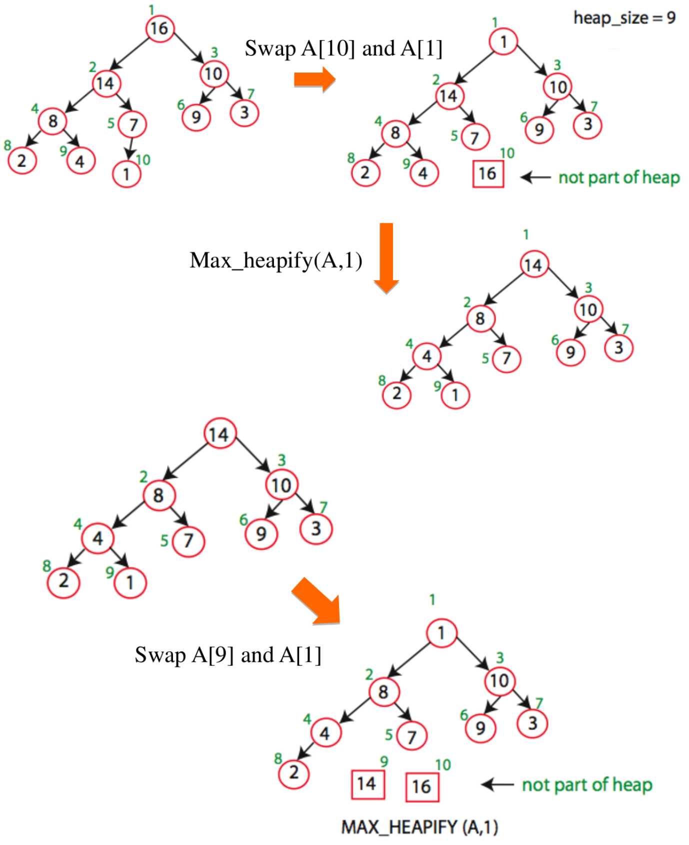
Heap sort

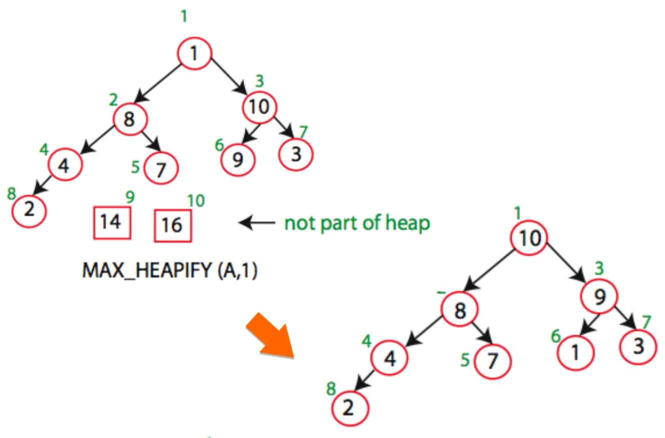
Heap sort is actually a very straight-forward sort given the heap operations. Given input A:

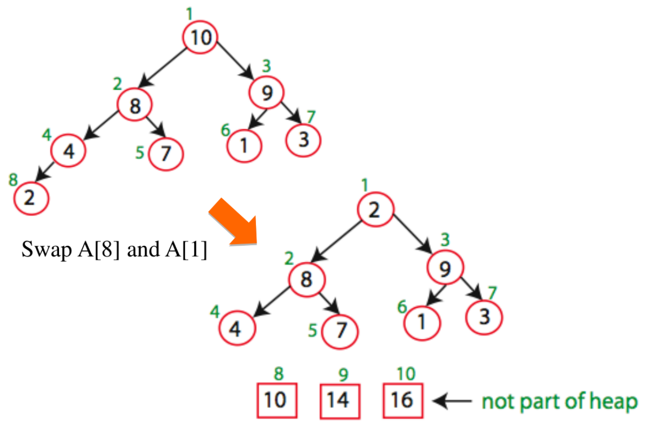
- Call BUILD-MAX-HEAP$(A) \rightarrow O(n)$.
- Repeatedly EXTRACT-MAX until the heap is empty, and store the extracted values in reverse order $\rightarrow O(n \log n)$.

In total, we get a runtime of $O(n \log n)$, which is the same as that of merge sort. One advantage of heap sort over merge sort is that there is a very clean way to perform this algorithm *in-place*. Rather than removing the max element from the array and shrinking the array by one at each step, just place the max element at the end of the heap, and decrement a value tracking the endpoint of the heap within the overall array (conveniently, the EXTRACT-MAX operation swaps the max element with the last element in the heap anyway). At the end of the algorithm, our array is sorted.

We can see an example of this below:







The remainder of the algorithm is omitted for brevity, but it is clear how to continue from here.

Exercise 2 – Manually perform heap sort on $A = [4, 10, 6, 8, 3]$.

An Application of Heap in Algorithm Design

Points on a Plane

Consider n points in a 2-dimensional plane, each specified by a set of (x, y) coordinates. Design an algorithm to return the k points which are closest to the origin, where $1 \leq k \leq n$.

There are a variety of solutions to this problem. Try to develop algorithms with the following runtimes.

1. $\Theta(n \log n)$

   Solution: Use merge-sort to sort the list of points by their distance to the origin and then take the k smallest elements. The total time is $O(n \log n + k) = O(n \log n)$.

2. $O(n + k \log n)$ time and $O(1)$ extra space.

   Note: You are allowed to modify the array A.

Solution: Use $build\_heap$ to turn A into a heap, in-place. Takes $O(n)$ time and $O(1)$ extra space. Now, run $extract\_min$ k times. Takes $O(k \log n)$ time and $O(1)$ extra space.

3. $\Theta(n \log k)$ and $O(k)$ extra space

   Note: you are not allowed to modify the array A.

   Solution: Let's do a bit of working backwards. You are given $O(k)$ space – so a natural thing to do is to maintain a heap of k elements. You are given $O(n \log k)$ time – so a natural solution idea is to walk through the list left to right, and do a heap operation every time. Should fit the time/space bounds. But what do we do in each step?

   Naturally, the heap contains the smallest k elements at any time. When we look at a new element, we have to decide whether to insert it into the heap (which we should, if it's smaller than the max of all elements in the heap) or ignore it and move on.

   In other words, we should be able to find the min of all the elements in the heap pretty quickly. So this should be a max-heap!! this is the slightly non-intuitive part of it – to get the min k elements, you maintain a max-heap.

   All of this should take $O(n \log k)$ time.