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 → \(O(1)\).

- **MAX-HEAPIFY(H, x)**: Correct a single error in the max-heap property at node x.
  
  *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 → \(O(\log n)\).

- **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 → \(O(\log n)\).

- **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 → \(O(\log n)\).

- **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) → \(O(\log n)\).

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:

- \( \text{\textsc{parent}}(x) = \lfloor x/2 \rfloor \)
- \( \text{\textsc{left}}(x) = 2 \times x \)
- \( \text{\textsc{right}}(x) = 2 \times x + 1 \)

### Building heaps

To work with heaps, we must first be able to construct them from an arbitrary array of elements. We define \( \text{\textsc{build-max-heap}}(A) \) 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 \( \text{\textsc{max-heapify}} \) 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 would appear to be \( n \times O(\log n) = O(n \log n) \). At closer look, we can see that we’re actually overcounting, 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:

\[
\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} O(h) = O(n \sum_{h=0}^{\log n} \frac{h}{2^{h+1}}) \\
\leq O(n \sum_{h=0}^{\infty} \frac{h}{2^{h}}) \\
= O(n) 
\]

\(^1\)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:

![Heap diagram]

**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 straightforward 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 \( \rightarrow O(n \times \log n) \).

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 in fact a very clean way to perform this algorithm *in-place*. Rather than removing the maximum element from the array and shrinking the array by one at each step, just move the maximum value to the last value in the heap, and decrement a value tracking the endpoint of the heap within the overall array (conveniently, the **EXTRACT-MAX** operation swaps the maximum 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:

\[\text{Swap A[10] and A[1]}\]

\[\text{Max_heapify(A,1)}\]

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

The remainder of the algorithm is omitted for brevity, but it is clear how to continue from here.
**Change of Variables (Solving Recurrences)**

For some particularly tricky recurrences, it may be necessary to combine the existing methods we’ve looked at with a method known as **change of variables**. The idea behind change of variables is to rewrite a tricky recurrence in terms of a new variable that is somehow related to the old variable, solve the new recurrence, then change back to the original variable.

Consider, for example, the following recurrence containing a square root:

\[
T(n) = 2T(\sqrt{n}) + \Theta(\log n)
\]

None of our existing methods provide an easy way to solve this recurrence. To solve this we can perform a change of variables by defining a new variable \( m \) as \( n = 2^m \). Substituting this in gives:

\[
T(2^m) = 2T(\sqrt{2^m}) + \Theta(2^m) = 2T(2^{m/2}) + \Theta(m)
\]

This recursion still isn’t in the standard form that we expect for Master Theorem, so we define a new recurrence, \( S(m) \) as \( S(m) = T(2^m) = T(n) \). This gives us:

\[
S(m) = T(2^m) = 2T(2^{m/2}) + \Theta(m) = 2S(m/2) + \Theta(m)
\]

Suddenly, we have a form we can work with using any of the previous methods we’ve learned. Using Master Theorem, for example, gives us \( S(m) = m \log m \). Now we simply have to substitute back to make this meaningful in the context of \( T \) and \( n \):

\[
S(m) = \Theta(m \log m) \\
T(n) = \Theta(\log n \log \log n)
\]

**Exercise 3** – Suppose you are given \( T(\log_2 n) = 2T(\log_4 n) + \Theta(\log \log n) \). Compute a closed form for \( T(x) \) using change of variables.