6.3 Build-Heap

We can create a heap from an unordered array by calling Heapify on the internal nodes starting at the bottom and working our way up.

Let \( n = \text{heap-size}[A] = \text{length}[A] \) and observe that \( \lfloor n/2 \rfloor \) is the index of the last (i.e. rightmost) internal node, since it is the parent of the last leaf.

Thus the elements of \( A[\lfloor n/2 \rfloor + 1 \ldots n] \) are all leaves and hence each is already a heap.

We process the internal nodes \( A[1 \ldots \lfloor n/2 \rfloor] \) in an order which guarantees that the subtree rooted at each child is already a heap.

**Build-Heap** \((A)\)

1. \( n \leftarrow \text{heap-size}[A] \leftarrow \text{length}[A] \)
2. for \( i \leftarrow \lfloor n/2 \rfloor \) down to 1
3. Heapify \((A, i)\)

**Ex.**

\[
A_1, \ldots, A_6, A_7, \ldots, A_{12},
\]

Internal nodes \( \overbrace{A_1 \ldots A_6} \) \( A_7 \ldots A_{12} \)

Leaves
The run time of Build-Heap is $O(n \log n)$ since each call to Heaipify costs $O(\log n)$ and there are $\lfloor \frac{m}{2} \rfloor = \Theta(n)$ such calls.

This bound is not tight however, observe that the cost of calling Heaipify on a node of height $h$ is $\Theta(h)$, and the heights of most nodes are small.

**Lemma (Problem 6.3-2, p. 135)**

An ACRST on $n$ nodes has at most $\lfloor \frac{n}{2^{h+1}} \rfloor$ nodes at height $h$.

**Proof**

Recall the last (i.e., rightmost) internal node is the parent of the last leaf, and hence has index $\lfloor \frac{n}{2} \rfloor$.

\[
\begin{array}{cccccccc}
A_1 & \ldots & A_{\lfloor \frac{n}{2} \rfloor} & A_{\lfloor \frac{n}{2} \rfloor + 1} & \ldots & A_n \\
\text{internal nodes} & & & & \text{leafs} & \\
\end{array}
\]

Therefore, the number of leaves (nodes at height 0) is

\[n - \lfloor \frac{n}{2} \rfloor\]

If we delete these leaves we obtain an ACRST on $\lfloor \frac{n}{2} \rfloor$ nodes which therefore
has \( \lfloor \frac{n}{2} \rfloor - \lfloor \frac{\lfloor n/2 \rfloor}{2} \rfloor = \lfloor n/2 \rfloor - \lfloor n/2^2 \rfloor \) leaves.

Each of these leaves was at height 1 in the original ACRT, therefore an ACRT on \( n \) nodes has exactly

\[ \lfloor n/2 \rfloor - \lfloor n/2^2 \rfloor \]

nodes at height 1. Now delete all leaves in this new tree to obtain an ACRT on \( \lfloor n/2 \rfloor \) nodes which therefore has \( \lfloor n/2^2 \rfloor - \lfloor n/2^3 \rfloor \) leaves, which are precisely those nodes at height 2 in the original tree.

Continuing in this manner, we see that an ACRT on \( n \) nodes has exactly

\[ \lfloor n/2^h \rfloor - \lfloor n/2^{h+1} \rfloor \]

nodes at height \( h \). (This part of the proof could be phrased as a finite induction on \( h \) for \( h = 0, 1, \ldots, \lfloor \log n \rfloor \).)

**Exercise**

Show \( \forall x \in \mathbb{R}: \lfloor x \rfloor \leq \lfloor x/2 \rfloor + \lfloor x/2 \rfloor \leq \lceil x \rceil \).

In particular, \( \lfloor x \rfloor - \lfloor x/2 \rfloor \leq \lceil x/2 \rceil \), so letting \( x = n/2^h \), we see that in an ACRT on \( n \) nodes:

\[ (\# \text{ nodes at height } h) = \lfloor n/2^h \rfloor - \lfloor n/2^{h+1} \rfloor \leq \lceil n/2^{h+1} \rceil \].
Let $T(n)$ denote the worst case run time of Build-Heap on an array of length $n$. Build-Heap calls Heapify on the last internal node ($h = 1$), then backs through the array to the root ($h = \lfloor \log n \rfloor$). Each call to Heapify on a node at height $h$ costs $\Theta(h)$, and there are at most $\lceil n/2^{h+1} \rceil$ such nodes.

Therefore,

$$T(n) \leq \sum_{h=1}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil \cdot \Theta(h)$$

Now $\Theta(h)$ stands for a function which is bounded above by $c h$ for large $h$, and $\Gamma x^2 \leq 2 x$ for $x \geq 1$. Thus,

$$T(n) \leq \sum_{h=1}^{\lfloor \log n \rfloor} \frac{n}{2^h} \cdot c h \leq cn \sum_{h=1}^{\infty} \frac{2^h}{h} \cdot \left(\frac{1}{2}\right)^h$$

Formula A.8 on p. 1061 says:

$$\sum_{k=1}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

Exercise: Prove this.

Thus $T(n) \leq cn \cdot \frac{1}{(1-\frac{1}{2})^2} = 2cn = O(n)$.

\[ \therefore T(n) = O(n) \]

Exercise: Show $T(n) = \Omega(n)$, whence $T(n) = \Theta(n)$. 
6.4 HeapSort

HeapSort takes as input an unordered array A. First, it calls Build-Heap(A) to order A as a heap. It then exchanges the largest element A[n] with A[1] (where n = length[A]), then excludes A[n] from the heap by decrementing heap-size[A] to n-1.

At this point the subarray A[1,...,(n-1)] may not be a heap, but the left and right subtrees rooted at A_2 and A_3 are heaps (since they were before the swap). Thus a single call Heapify(A,1) makes A[1,...,(n-1)] into a heap.

Now exchange A_1 ↔ A_{n-1}, decrement heap-size[A] to n-2, call Heapify(A,1), and repeat the process until heap-size[A] is 1.

**HeapSort(A)**

1. Build-Heap(A)
2. For i ← length[A] down to 2
4. heap-size[A] ← (heap-size[A] - 1)
5. Heapify(A, 1)