Heap property: for every node $i$, the value in $i$ is less or equal the value in $p(i)$.

The embedding is defined by:

- $\text{root} := 1$
- $l(i) := 2i$
- $r(i) := 2i + 1$
- $p(i) := \left\lfloor \frac{i}{2} \right\rfloor$

Heap property is: $A[i] \leq A[p(i)]$ for all $i$.

The height of a node is the number of edges on the longest downward path starting at the node.

The height of a heap is the height of its root.

**Claim.** \( \log(n+1) - 1 \leq h \leq \log n \)

**Proof.** \[
\sum_{i=0}^{h-1} 2^i + 1 \leq n \leq \sum_{i=0}^{h} 2^i
\]
\[
2^h \leq n \leq 2^{h+1} - 1
\]
\[
\log(n+1) - 1 \leq h \leq \log n
\]

**Maintaining Heap Property:**

**Down-heap** extends the heap-property by one more node.

**Procedure** `Down-heap(i)`:

\[
\text{max} := i; \quad \text{if } l(i) \leq \text{heap-size}[A] \text{ and } A[\text{max}] < A[l(i)]
\]
\[
\quad \text{then } \max := l(i)
\]

\[
\text{endif}
\]
\[
\text{if } r(i) \leq \text{heap-size}[A] \text{ and } A[\text{max}] < A[r(i)]
\]
\[
\quad \text{then } \max := r(i)
\]

\[
\text{endif}
\]
\[
\text{if } \max \neq i \text{ then } A[i] \leftrightarrow A[\max]
\]
\[
\text{Down-heap} (\max)
\]
\[
\text{endif}
\]

**Cost is** \( O(h) \)
An iterative version is:

Procedure Downheap(i);
repeat max := i;
  if L(i) ≤ heap-size(A) and A[max] < A[L(i)]
    then max := L(i)
  endif
  if r(i) ≤ heap-size(A) and A[max] < A[r(i)]
    then max := r(i)
  endif
  A[i] ← A[max]; i ← max
until i := max

Building a heap

The idea is to construct it from bottom up.

Procedure Build-heap (n);
  for i := n downto 1 do Downheap(i) endfor
(Assumption: A is global; n = heap-size[A])

It is easy to show that this takes $O(n \log n)$ time. But, a tighter analysis is possible.
The amount of time to build the heap is at most
\[ \sum_{i=0}^{h} 2^i \cdot \log(h-i) = O(\frac{h \sum 2^i}{i=0} - \sum_{i=0}^{h} 2^i) \]
\[ = O(h \cdot 2^{h+1} - h - (h+1) \cdot 2^{h+1} + 2^{h+2}) \]
\[ = O(n) \]
[used the fact: \( \sum_{i=0}^{h} i \cdot 2^i = (h+1)2^{h+1} - 2^{h+2} + 2 \)]

**HeapSort algorithm**

The input is an unsorted array \( A[1...n] \), \( n = \text{length}(A) \). After the construction of heap, the maximum is repeatedly moved while shrinking it.

**Procedure HeapSort**(n);
Buildheap(n); heap-size\([A]\) := n;
for i := n downto 2 do 
    heap-size\([A]\) := i-1
    Downheap(1)
endfor

Complexity is \( O(n \log n) \)
Heap as Priority Queue

A priority queue stores a multiset $S$ of keys and supports operations:

- **Insert ($x$):** $S := S \cup \{x\}$
- **Delete ($i$):** remove element at location $i$
- **Max:** return the largest key
- **Extract Max:** return the largest key and remove it.

**Procedure** Insert ($x$):
- heap-size($A$) := heap-size($A$) + 1
- $i :=$ heap-size($A$)
- $A[i] := x$
- **upheap** ($i$)

**Procedure** Upheap ($i$):
- while $i > 1$ and $A[i] > A[p(i)]$ do
  - $A[i] \leftarrow A[p(i)]$
  - $i := p(i)$
- endwhile

**Procedure** Delete ($i$):
- $A[i] := A[\text{heap-size}(A)]$
- heap-size($A$) := heap-size($A$) - 1
- if $A[i] < A[p(i)]$ then **Downheap** ($i$)
  - else **upheap** ($i$)
- endif

**function** ExtractMax:
- ExtractMax := $A[1]$; Delete ($i$)

All take $O(\log n)$ time.