1 Heaps

- **Complete binary tree**: a binary tree with exactly $2^h$ leaves, where $h$ is the height of the tree.

- **Nearly complete binary tree**: a binary tree obtained from a complete binary tree by deleting one or more of its “rightmost” leaves.

- A **heap** is an array $A[1..n]$ or its subarray $A[1..m]$, $m < n$, viewed as a complete or nearly complete binary tree.

- heap-size[$A$]: number of elements in the heap.

- parent($i$) = $\lfloor i/2 \rfloor$

- left($i$) = $2i$

- right($i$) = $2i + 1$

- **Max-heap**: $A[i] \leq A[parent(i)]$, for all $2 \leq i \leq \text{heap-size}[A]$

- **Min-heap**: $A[i] \geq A[parent(i)]$, for all $2 \leq i \leq \text{heap-size}[A]$
2 Max-Heapify

Problem: In a heap, suppose the subtrees rooted at left\((i)\) and right\((i)\) are max-heaps. We want to make the tree rooted at \(i\) a max-heap.

procedure MaxHeapify\((A, i)\)

\(l \leftarrow \text{left}(i)\)

\(r \leftarrow \text{right}(i)\)

\(\text{size} \leftarrow \text{heap-size}[A]\)

\(\text{largest} \leftarrow i\)

\(\text{if}\ (l \leq \text{size}) \text{ and } (A[l] > A[\text{largest}]) \text{ then } \text{largest} \leftarrow l\)

\(\text{if}\ (r \leq \text{size}) \text{ and } (A[r] > A[\text{largest}]) \text{ then } \text{largest} \leftarrow r\)

\(\text{if}\ (\text{largest} \neq i)\ \text{then}\)

\(\text{exchange } A[i] \leftrightarrow A[\text{largest}]\)

MaxHeapify\((A, \text{largest})\)

Running time: \(O(\log n)\).
3 Build a heap

Problem: Convert an array \( A[1..n] \) of integers into a max-heap.

procedure Build-Max-Heap\( (A[1..n]) \)

heap-size\[A\] \( \leftarrow \) \( n \)

for \( i \leftarrow \) parent\( (n) \) downto 1 do

MaxHeapify\( (A, i) \)

Running time: \( O(n \log n) \).
4 Heapsort


procedure Heapsort($A[1..n]$)
  Build-Max-Heap($A[1..n]$)
  for $i \leftarrow n$ downto 2 do
    exchange $A[1] \leftrightarrow A[i]$
    decrease heap-size[$A$] by 1
    MaxHeapify($A, 1$)

Running time: $O(n \log n)$.
5 Priority Queues

- A data structure for maintaining a set $S$ of elements with the following operations:
  - $\text{Maximum}(S)$: returns the element in $S$ with the largest key.
  - $\text{Insert}(S, x)$: inserts an element $x$ into $S$.
  - $\text{Extract-Max}(S)$: removes and returns the element in $S$ with the largest key.

- The following function is useful in many applications of priority queue.
  - $\text{Increase-Key}(S, i, k)$: increases $i$’s key to $k$.

- Basic idea: use a max-heap.

- All the above operations can be done in $O(\log n)$ time, where $n$ is the size of $S$.

**procedure** Increase-Key($A[1..n], i, k$)

/* increase $A[i]$ to $k$ */

if ($k < A[i]$) then error “new key smaller than current key”

$A[i] \leftarrow k$

while ($i > 1$) and ($A[\text{parent}(i)] < A[i]$) do

exchange $A[i] \leftrightarrow A[\text{parent}(i)]$

$i \leftarrow \text{parent}(i)$