A **Priority Queue** is a data structure maintaining a set $S$ of elements, each with a **key**. A **max-priority queue** supports the following operations:

- **Insert**($S$, $x$): inserts the element $x$ into the set $S$.

- **Maximum**($S$): returns the element of $S$ with the largest key.

- **Extract-Max**($S$): removes and returns the elements of $S$ with the largest key.

- **Increase-Key**($S$, $x$, $k$): increase the value of element $x$’s key to the new value $k$. 
Applications of Priority Queue

Job Scheduler in Operating Systems.

- Each process is assigned a priority. (each process has an item in a priority queue).
- When a new process comes in, system will assign it a priority. (Insert into the priority queue).
- System picks the process with the highest priority to run. (Maximum).
- When a process terminates, system needs to remove it from the queue. (Extract-Max)
- Sometimes system needs increase or decrease the priority of certain process. (Increase-Key).
First implementation: Sorted Array

- Insert: Need search for the proper place to insert and some elements need be moved; worst case: $\Theta(n)$.  

- Maximum: $\Theta(1)$  

- Extract-Max: $\Theta(1)$  

- Increase-Key: May need to find a new place to put; worst case: $\Theta(n)$
- **Heap** is a *complete binary tree*, namely, it’s filled at all levels except at the lowest level (filled from left to right).

- **(Max-Heap property)** The value sorted in a node is greater than or equal to the values stored at its children.
Since there are no nodes at level \( l \) unless level \( l - 1 \) is completely filled, a heap can be stored in an array level by level (beginning with the root), left to right within each level.

- The ROOT is always stored at \( A[1] \)
- \( \text{PARENT}(i) = \lfloor i/2 \rfloor \)
- \( \text{LEFT-CHILD}(i) = 2i \)
- \( \text{RIGHT-CHILD}(i) = 2i + 1 \)
- \( \text{Length}[A] \): number of elements in the array \( A \);
  \( \text{Heapsize}[A] \): number of elements in the heap stored within array \( A \).
Examples of Heap Representations

![Heap Diagrams]

1. [Binary Heap Diagram]
2. [Max Heap Diagram]

Array Representation:

```plaintext
[9 5 7 1 4 3 6]
```

```plaintext
[50 24 30 21 18 3 12 5 6]
```
Maintain(Restore) the heap property

If $A[i]$’s left subtree and right subtree are Max-Heaps, but $A[i]$ violates the heap property, i.e., $A[i]$ is smaller than its children, Max-Heapify($A$, $i$) is called to let $A[i]$ “float down” in the max-heap so that the subtree rooted at index $i$ becomes a Max-Heap.

Max-Heapify($A$, $i$)

\[
\begin{align*}
\text{max} & \;:= \; i; \\
\text{if} \; (2i \leq \text{Heapsize}[A] \; \text{AND} \; A[\text{max}] < A[2i]) & \; \text{max} \;:= \; 2i; \\
\text{if} \; (2i + 1 \leq \text{Heapsize}[A] \; \text{AND} \; A[\text{max}] < A[2i + 1]) & \; \text{max} \;:= \; 2i + 1; \\
\text{if} \; (i \neq \text{max}) & \; \text{exchange} \; A[i] \; \text{and} \; A[\text{max}]; \\
& \; \text{Max-Heapify}(A, \text{max});
\end{align*}
\]
Max-Heapify
Complexity for MAX-HEAPIFY


- The children's subtree each has size at most \( 2n/3 \) — the worst case occurs when the last row of the tree is exactly half full.

\[
T(n) \leq T(2n/3) + \Theta(1)
\]

\[
\Rightarrow T(n) = O(lgn) \quad \text{(Using Master Method)}
\]

- Another approach: the height of a complete binary tree with \( n \) elements is \( \Theta(lgn) \). Why is that?

Because for a complete binary tree with height \( h \), it has at most \( 2^{h+1} - 1 \) and at least \( 2^h - 1 + 1 = 2^h \) nodes.

\[
2^h \leq n \leq 2^{h+1} - 1 \Rightarrow \lg(n + 1) - 1 \leq h \leq lgn
\]

Because \( T(n) \leq h \Rightarrow T(n) = O(lgn) \)
Build a Heap

Use Max-Heapify in a bottom-up fashion to convert $A[1..n]$ to a Max-Heap.

**Build-Max-Heap**($A$)

$$\text{Heapsize}[A] = \text{Length}[A];$$

for $i := \lfloor \text{Length}[A]/2 \rfloor$ downto 1

Max-Heapify($A, i$);
An example of Build-Max-Heap
- Assume that the binary tree is a full binary tree (the proof is slightly more complicated if the binary tree is not full):

- \( T(n) = T(n/2) + T(n/2) + O(lgn) \), where the first \( T(n/2) \) is to Build the left sub heap, the second is for right sub heap. The \( O(lgn) \) is the complexity for Max-Heapify(A, 1), which makes the whole tree a heap.

- Based on Master-Method case 1, \( T(n) = \Theta(n) \).
**HEAPSORT***(A)***

Build-Max-Heap(A);  —— \( \Theta(n) \)

for \( i := n \) to 2


Max-Heapify (A, 1);  —— \( O(lgn) \)

Comments:

- \( A[1..\text{Heapsize}[A]] \) are the elements currently in the heap.
  When elements are removed from the heap one by one, Heapsize[A] is decremented. Length[A] does not change.

- Complexity: \( \Theta(n) + O(nlgn) = O(nlgn) \).
Examples of HeapSort

(a) 16 4 8 2 1 7 3 9 10
(b) 14 8 4 10 2 7 9 3
(c) 8 10 4 9 2 7 1 3
(d) 9 8 4 7 2 1 3
(e) 9 8 2 7 1 1 3
(f) 9 2 8 1 1 1 3
(g) 1 4 8 7 3 2 9
(h) 1 4 7 2 3 2 9
(i) 1 2 3 4 7 8 9 10 14 16
Priority Queue Operations: using Heaps

Maximum(A)
return A[1];

--- Complexity: $\Theta(1)$

Extract-Max(A)  // Remove and return the max.

MAX := A[1];
Heapsize := Heapsize - 1;
Max-Heapify(A, 1);
return MAX;

--- Complexity: $O(h) = O(lgn)$. 
Priority Queue Operations, Cont’d

Increase-Key(A, \(i\), \(key\))  // Increase the value of \(A[i]\) to \(key\).
                            // assume \(key\) is bigger than \(A[i]\).

\[ A[i] := key; \]
\[ \text{while } (i > 1 \text{ AND } A[\text{parent}(i)] < A[i]) \text{ DO} \]
\[ \text{exchange } A[i] \text{ and } A[\text{parent}(i)]; \]
\[ i := \text{parent}(i); \]

--- Complexity: \(O(h) = O(lgn)\).

Insert(A, \(key\))  // Insert \(key\) into A

\[ \text{Heapsize}[A] := \text{Heapsize}[A] + 1; \]
\[ A[\text{Heapsize}[A]] := -\infty; \]
\[ \text{Increase-Key}(A, \text{Heapsize}[A], key); \]

--- Complexity: \(O(h) = O(lgn)\).
### Summary: Complexities using Heaps

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max-Heapify</td>
<td>(O(lgn))</td>
</tr>
<tr>
<td>Build-Max-Heap</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Heap-Sort</td>
<td>(O(nlgn))</td>
</tr>
<tr>
<td>Maximum</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td>Extract-Max</td>
<td>(O(lgn))</td>
</tr>
<tr>
<td>Insert</td>
<td>(O(lgn))</td>
</tr>
<tr>
<td>Increase-Key</td>
<td>(O(lgn))</td>
</tr>
</tbody>
</table>