Priority Queues and Heaps
Priority Queues & Heaps

Many problems involve a dynamically changing set $S$ of objects that each have a priority.

Examples

• Process scheduling on computers

• Shortest path computations

• Selection of next node to explore in a search algorithm
Priority Queues & Heaps

Managing such a set typically involves the following operations on $S$

- Insert a new element
- Retrieve highest priority element
- Delete highest priority element

A structure supporting these methods is called a **priority queue**

- **Note:** Priorities are encoded as a ‘key’ value
  - Typically: higher priority $\rightarrow$ lower key value
Priority Queue Implementation

Sorted list (low to high key value)

- Insert: $O(n)$ time (must find insertion point)
- Retrieve item with minimum key: $O(1)$ time
- Delete item with minimum key: $O(1)$ time
- Initialize: $O(1)$ time
Priority Queue Implementation

Sorted array (low to high key value)

- Insert: $O(n)$ time (must move elements)
- Retrieve item with minimum key value: $O(1)$ time
- Delete item with minimum key value: $O(n)$ time
  - $O(1)$ time if array is ‘circular’
- Initialize: $O(n)$ time
Priority Queue Implementation

Balanced Binary Tree

- Assume root of every subtree has lowest value key in subtree
- Insert: $O(\log n)$ time (traversal and rebalancing)
- Retrieve item with minimum key value: $O(1)$ time
  - Need to rebalance!
- Delete item with minimum key value: $O(\log n)$ time
- Initialize: $O(1)$ time
Priority Queues via Heaps

A heap combines tree structure with array access

- Insert: $O(\log n)$ time (‘tree’ traversal & moves)
- Retrieve item with minimum key value: $O(1)$ time
- Delete item with minimum key value: $O(\log n)$ time
  - Element shifting to maintain balance
- Initialize: $O(n)$ time (initialize an array)
Priority Queues via Heaps

Benefits of heap over balanced binary tree

• Almost trivial to implement heaps

• Each operation is fast!

• The “Big-O” constants c are very low for heaps

• Supports useful extensions

• Dynamic changing of priorities within heap
Heap Example

```
H
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>11</td>
<td>17</td>
<td>14</td>
<td>30</td>
<td>21</td>
<td>35</td>
<td>24</td>
<td>19</td>
<td>22</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Structure of a Heap

Def’n: A binary tree $T$ with node priorities and depth $d$ has a **heap structure** if

- Every node has lower key value (higher priority) than its children ($T$ is in **heap order**),
- Every leaf is at depth $d$ or $d-1$,
- At each level $l < d$, there are $2^l$ nodes
- All nodes at level $d$ are in left-most positions
Structure of a Heap II

A binary tree $T$ with a **heap structure** can be efficiently stored in an array $H[1..n]$

- Nodes of $T$ are stored in $H$ level by level, from left to right, with root($T$) at $H[1]$

Features

- Node of $T$ at $H[i]$ has parent at $H[\lfloor i/2 \rfloor]$

- Node of $T$ at $H[i]$ has children at $H[2i]$ & $H[2i + 1]$

- No ‘holes’ in the array
Heap Example

The heap is represented as a binary tree where each parent node's value is greater than or equal to its child nodes.

Heap Example Table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>X</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>11</td>
<td>17</td>
<td>14</td>
<td>30</td>
<td>21</td>
<td>35</td>
<td>24</td>
<td>19</td>
<td>22</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Implementing Heap Operations

Insert element $s$ into the (min-)heap $H$

- Put $s$ in first available slot $k$ in $H[]$
- While $s.key < s.parent.key$, “percolate” $s$ up tree
  - Percolate: swap $H[k]$ with $H[k/2]$; set $k ← k/2$
- Called **Heapify-up** in text
Delete element x in H[1] (minimum key value) from heap

- Remove x from H[1]
- Move last element s of heap to H[1]
- While s.key > min{s.left.key, s.right.key}, “sift” s down tree
  - Sift: swap s with child having smaller key value
- Called Heapify-down in text
Heapify-up & Heapify-down can be used to **re-heapify** the structure if a single priority has been changed

- Assume that Heap Property holds everywhere except at $H[i]$
  - If $H[i] < H[i/2]$, Heapify-up($H, i$) percolates the value at $H[i]$ up until heap property is restored
  - If $H[i] > \min\{H[2i], H[2i+1]\}$, Heapify-down($H, i$) sifts the value at $H[i]$ down until heap property is restored

- Both operations take $O(\log n)$ time