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 $\leftrightarrow$ lower key value
Priority Queue Implementation

Sorted linked 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
- Delete item with minimum key value: $O(\log n)$ time
  - Need to rebalance!
- 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 structure
- 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 are very low for heaps
• Supports useful extensions
  • Dynamic changing of priorities within heap
**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
- An array with these features is called a heap
Heap Example
Implementing Heap Operations

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

- Put $s$ in first available slot $k$ in $H[]$
- While $H[k].key < H[k].parent.key$, “percolate” up tree
  - Percolate: swap $H[k]$ with $H[k/2]$; set $k ← k/2$
- Called **Heapify-up** in text
Implementing Heap Operations II

Delete element $x$ in $H[1]$ (minimum key value) from heap

- Remove $x$ from $H[1]$
- Move last element $s$ of heap to $H[1]$
- Let $k = 1$
- While $H[k].key > \min\{H[2k].key, H[2k+1].key\}$, “sift” $H[k]$ down tree
  - Sift: swap $H[k]$ with child having smaller key value
  - Set $k$ to the location of the child $H[k]$ was swapped with ($2k$ or $2k+1$)
- Called $\text{Heapify-down}$ in text
Implementing Heap Operations III

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
HeapSort

A heap can be converted to a sorted array in $O(n \log n)$ time

• for $i = 0$ to $n-1$
  
  • let $x = H.\text{removeMin}()$
  
  • put $x$ in location $H[n-i]$

$H[]$ is now in (reverse) sorted order
Heapifying an Array

An array $H[]$ can be converted to a heap in $O(n)$ time

- for $i = n/2$ downTo 1
  - $H$.heapifyDown($i$)

$H[]$ is now in a heap. This is called bottom-up heapifying

Try this at home:

- Prove that bottom-up heapifying takes $O(n)$ time
Heapifying an Array

An array \( H[] \) can be converted to a heap by top-down heapifying

- for \( i = 2 \) to \( n \)
  - \( H.\text{heapifyUp}(i) \)

Try this at home:

- Prove that top-down heapifying takes \( O(n \log n) \) time