Heaps and Heapsort

Ref: Chapter 6 of text.

Heapsort:
- An in-place sorting algorithm.
- Time: $O(n \log n)$.
- Based on (binary) heap: a useful data structure.

Heap: (or Max Heap)
- An implicit data structure (no pointers).
- An array $A$ that represents a binary tree.
- $A[1]$ has the value stored at the root of the binary tree.
- Binary tree is complete except possibly at the lowest level which is filled left to right.
- **Heap Property:** Except for the root, the value at a node is $\leq$ the value of its parent.

Example: To be presented in class.

Notes:
1. Heap size = No. of items stored.
2. For a node with index $i > 1$, $\text{Parent}(i) = \lfloor i/2 \rfloor$. (Root has no parent.)
3. For a node with index $i$, $\text{Left}(i) = 2i$ and $\text{Right}(i) = 2i + 1$, provided the indices $2i$ and $2i + 1$ are within the heap.

**Restatement of Heap Property:** For every node $i$ other than the root,

$$A[\text{Parent}(i)] \geq A[i].$$

**Consequence:** Maximum value is $A[1]$ (i.e., at the root).

**Definition 1:**

1. The **height** of a node $x$ in a tree is the number of edges in a longest path from $x$ to a leaf.
2. The **height of a heap** is the height of the root.
Lemma 1: The height of a heap with \( n \) nodes is \( \lceil \log_2 n \rceil \).

Proof: Exercise.

Heap operations:

(a) \textsc{Heapify}(\( A, i \))

Given: Array \( A \) such that the binary trees rooted at \( \text{Left}(i) \) and \( \text{Right}(i) \) are heaps.

Required: Make the subtree rooted at \( i \) a heap.

Example: To be presented in class.

Outline of \textsc{Heapify}:

1. Find the largest of \( A[i] \), \( A[\text{Left}(i)] \) and \( A[\text{Right}(i)] \). Let \( m \) be the index of largest value.
2. If \( m = i \), the structure is already a heap. So, stop.
3. Otherwise, swap \( A[i] \) with \( A[m] \) and invoke \textsc{Heapify}(\( A, m \)).

Pseudocode for \textsc{Heapify}: Handout 7.1.

Running time of \textsc{Heapify}:

Let \( n \) be the number of nodes in the subtree rooted at node \( i \).

- \textsc{Heapify} spends \( O(1) \) time at node \( i \) and descends to a child of \( i \).
- \textsc{Heapify} is called only \( O(\log n) \) times since height of node \( i \) is \( O(\log n) \) (Lemma 1).
- So, time = \( O(\log n) \).

Note: Let \( h \) denote the height of node \( i \). The running time of \textsc{Heapify} when started at node \( i \) is \( O(h) \).

(b) \textsc{Build-Heap}(\( A \))

Given: Array \( A[1 .. n] \).

Required: Convert \( A \) into a heap.
Observation: Elements in \( A[[n/2] + 1 .. n] \) are all leaves.

Reason:
\[
\text{LEFT}([n/2] + 1) = 2([n/2] + 1) > n.
\]

Ideas behind Build-Heap:
- Each leaf is a heap by itself.
- To convert \( A \) into a heap, use \textsc{Heapify}(\( A, i \)) for \( i \) from \([n/2]\) down to 1.

Pseudocode for Build-Heap: Handout 7.1.

Running time of Build-Heap:

(1) Simple (but not tight) analysis:
- There are \([n/2]\) = \( O(n) \) calls to \textsc{Heapify}.
- Each call to \textsc{Heapify} uses \( O(\log n) \) time.
- So, running time = \( O(n \log n) \).

(2) A careful (tight) analysis:

Ideas:
- Running time of \textsc{Heapify}(\( A, i \)) = \( O(h) \), where \( h \) is the height of tree rooted at \( i \).
- As \( h \) increases, the no. of nodes with height \( h \) decreases.

Lemma 2: In a heap with \( n \) nodes, for any \( h \geq 0 \), the number of nodes at height \( h \) is at most \([n/2^{h+1}]\).

Proof: Exercise.

Theorem 1: For an array \( A \) with \( n \) elements, \textsc{Build-Heap} runs in \( O(n) \) time.

Proof: To be presented in class.

Heapsort:

Input: (Unsorted) Array \( A[1 .. n] \).

Requirement: Sort \( A \) into increasing order.
Idea:

1. Convert $A$ into a heap.
3. Convert $A[1..n-1]$ into a heap using $\text{Heapify}(A, 1)$.
4. Repeat the above steps (suitably) until $A$ is sorted.

Pseudocode: Handout 7.1.

Running time of Heapsort:

- Time for $\text{Build-Heap} = O(n)$.
- No. of iterations of for loop $= O(n)$.
- Time per iteration $= O(\log n)$ (because of $\text{Heapify}$).
- So, time spent in for loop $= O(n \log n)$.
- Overall running time $= O(n \log n)$.

Priority Queue:

- A data structure for maintaining a (multi) set $S$ of keys.
- Supports the following operations:
  (a) $\text{Insert}(S, x)$: Inserts key $x$ into $S$. (Does not check for duplicates.)
  (b) $\text{Maximum}(S)$: Returns the value of the largest key in $S$. (The key is not removed from $S$.)
  (c) $\text{Extract-Max}(S)$: Removes and returns the largest key in $S$.
  (d) $\text{Increase-Key}(S, x, k)$: Increases the value of the key of item $x$ to $k$. (Assume that $k$ is at least as large as the current key of $x$.)

Applications:

- Job scheduling.
- Event-driven simulators.
Heap as a priority queue:


Heap-Extract-Max:

- Idea: As in HEAPSORT.
- Pseudocode: Handout 7.1.
- Time: $O(\log n)$ (because of $\text{HEAPIFY}(A,1)$).

Heap-Increase-Key:

- Increase the key to the specified new value. This may violate the heap property.
- If there is a violation, move the new value up towards the root until its correct place is found.
- Pseudocode: Handout 7.1.

- Running time:
  - No. of iterations of the while loop $= O(\log n)$ (because height $= O(\log n)$).
  - Time per iteration $= O(1)$.
  - So, running time $= O(\log n)$.

Heap-Insert:

- Increment heap_size($A$) to create room for new key.
- Set the key of the newly created location to $-\infty$.
- Use $\text{HEAP-INCREASE-KEY}$ to increase the key from $-\infty$ to the new key value.
- Pseudocode: Handout 7.1.
- Running time: $O(\log n)$ – dominant part is due to $\text{HEAP-INCREASE-KEY}$.

Other implementations: Class discussion.