# Design and Analysis of Algorithms

## Lecture 3—Heapsort

Lei Wang

Dalian University of Technology

September 22, 2008

# Probabilistic Analysis and Randomized Algorithms

# Heapsort

- $O(n \lg n)$ worst case—like merge sort.
- Sorts in place—like insertion sort.
- Combines the best of both algorithms.

To understand heapsort, we'll cover heaps and heap operations, and then we'll take a look at priority queues.

# Heap data structure

## Heap as a tree:

Heap $A$ (not garbage-collected storage) is a nearly complete binary tree.

- ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
- ***Height*** of heap = height of root = $\Theta(\lg n)$.

## Heap as a array: A heap can be stored as an array $A$.

- Root of tree is $A[1]$.
- Parent of $A[i] = A[i/2]$.
- Left child of $A[i] = A[2i]$.
- Right child of $A[i] = A[2i+1]$.
- Computing is fast with binary representation implementation.

# Heap data structure (cont.)

**Heap property:**

- For max-heaps (largest element at root), ***max-heap property***: for all nodes $i$, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property***: for all nodes $i$, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

By induction and transitivity of $\leq$, the max-heap property guarantees that the maximum element of a max-heap is **at the root**. Similar argument for min-heaps.

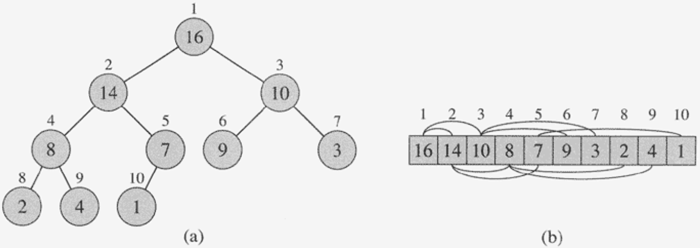



**Figure 6.1** A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

# Maintaining the heap property

MAX-HEAPIFY is important for manipulating max-heaps—to maintain the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of $i$ are max-heaps.
- After MAX-HEAPIFY, subtree rooted at $i$ is a max-heap.

## MAX-HEAPIFY

```
MAX-HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  if l ≤ heap-size[A] and A[l] > A[i]
     then largest ← l
     else largest ← i
  if r ≤ heap-size[A] and A[r] > A[largest]
     then largest ← r
  if largest ≠ i
     then exchange A[i] ↔ A[largest]
          MAX-HEAPIFY(A, largest)
```

## The way MAX-HEAPIFY works

- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue until subtree rooted at $i$ is max-heap.

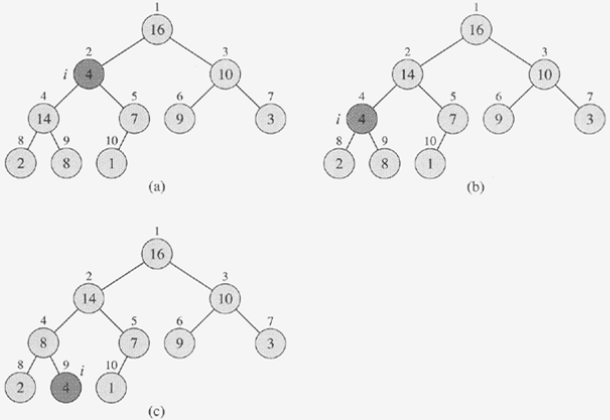



**Figure 6.2** The action of MAX-HEAPIFY($A$, 2), where *heap-size*[$A$] = 10. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY($A$, 4) now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY($A$, 9) yields no further change to the data structure.

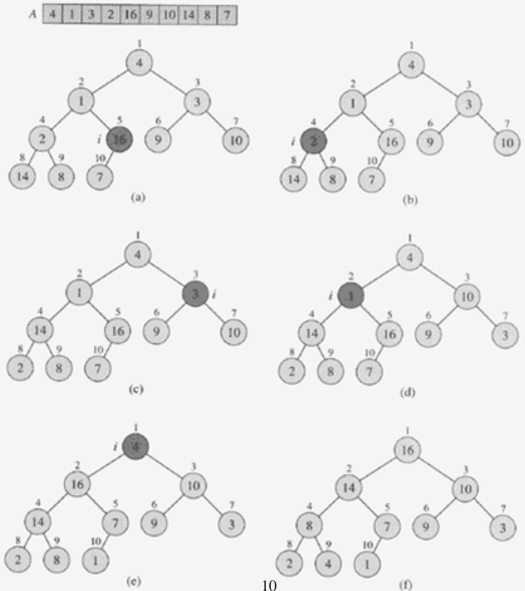
# Building a heap

## BUILD-MAX-HEAP

BUILD-MAX-HEAP$(A)$

```
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

- ***Simple bound:*** $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$.


A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

# The Heapsort Algorithm

- Builds a max-heap from the array.
- Starting with the root, the algorithm places the maximum element into the correct place in the array by swapping it with the last array element.
- "Discard" this last node by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node remains.

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
       heap-size[A] ← heap-size[A] − 1
       MAX-HEAPIFY(A, 1)
```

***Analysis:***

- BUILD-MAX-HEAP: $O(n)$
- for loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$.

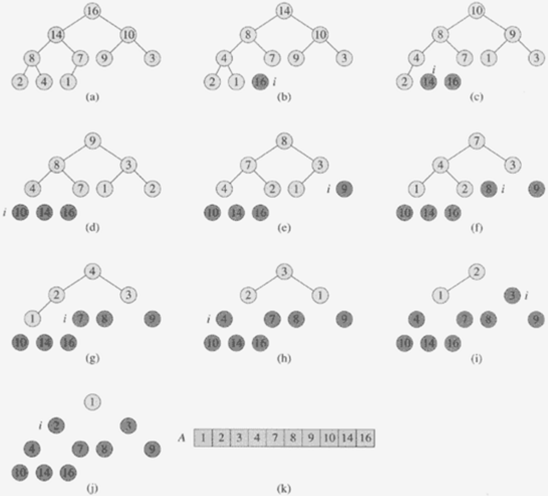



**Figure 6.4** The operation of HEAPSORT. **(a)** The max-heap data structure just after it has been built by BUILD-MAX-HEAP. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5. The value of $i$ at that time is shown. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array $A$.

# Priority queue

- Maintains a dynamic set $S$ of elements.
- Each set element has a key—an associated value.
- Max-priority queue supports dynamic-set operations:
  - $\textsc{Insert}(S, x)$: inserts element $x$ into set $S$.
  - $\textsc{Maximum}(S)$: returns element of $S$ with largest key.
  - $\textsc{Extract-Max}(S)$: removes and returns element of $S$ with largest key.
  - $\textsc{Increase-Key}(S, x, k)$: increases value of element $x$'s key to $k$. Assume $k \geq x$'s current key value.
  - Example max-priority queue application: **schedule jobs on shared computer**.