Announcements

- **Weekly Reading Assignment:**
  Chapter 6 (CLRS)
- **Homework #3 is due on Thursday, October 6, 2005**
- **Extra help session next Thursday, September 29, 2005**

Heapsort

- Heapsort (Chapter 6)
  - Data Structure
  - Maintain the Heap Property
  - Build a Heap
  - Heapsort Algorithm
  - Priority Queue

Heap Data Structure

- Construct in $\Theta(n)$ time
- Extract maximum element in $\Theta(\lg n)$ time
- Leads to $\Theta(n \lg n)$ sorting algorithm:
  - Build heap
  - Repeatedly extract largest remaining element (constructing sorted list from back to front)
- Heaps useful for other purposes too

Properties

- Conceptually a complete binary tree
- Stored as an *array*
- *Heap Property*: for every node i other than the root,

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

  - Algorithms maintain heap property as data is added/removed

Array Viewed as Binary Tree

- Last row filled from left to right

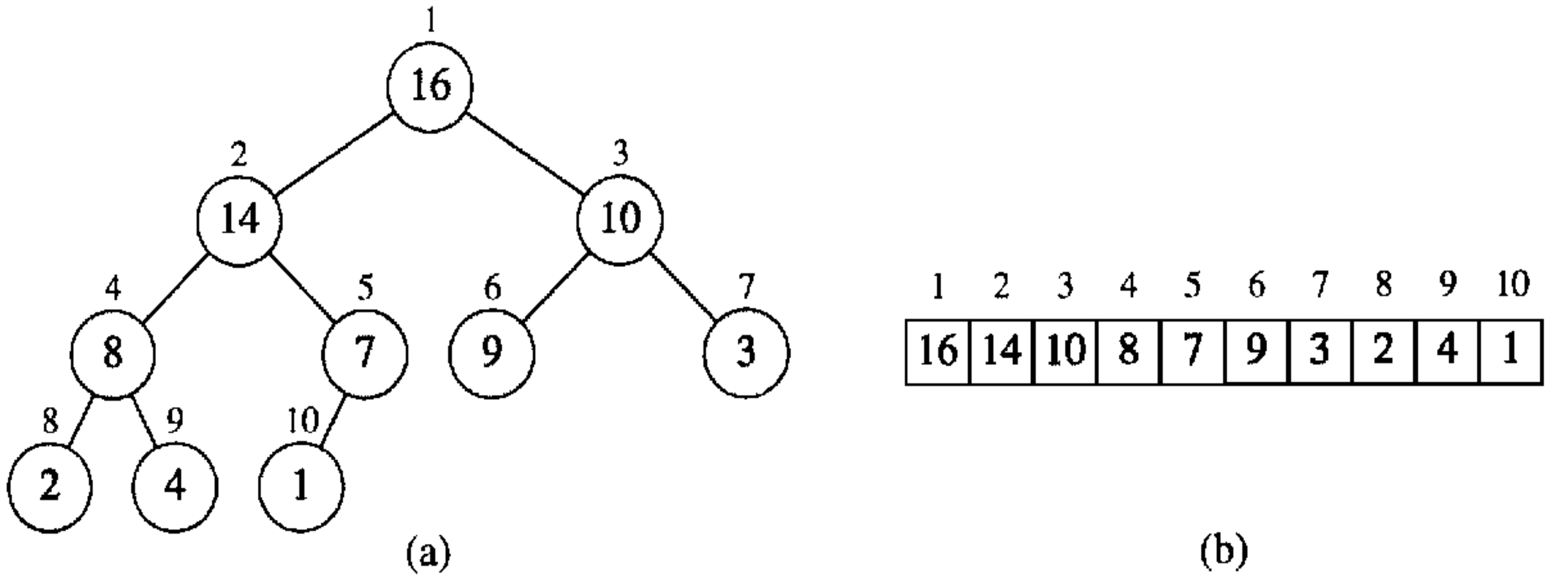

Figure 7.1 A heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number next to a node is the corresponding index in the array.

Basic Operations

- **Parent(i)**

  return $\lfloor i/2 \rfloor$

- **Left(i)**

  return $2i$

- **Right(i)**

  return $2i+1$

Height

- *Height of a node in a tree*: the number of edges on the longest simple downward path from the node to a leaf
- *Height of a tree*: the height of the root
- Height of the tree for a heap: $\Theta(\lg n)$
  - Basic operations on a heap run in $O(\lg n)$ time

Maintaining Heap Property

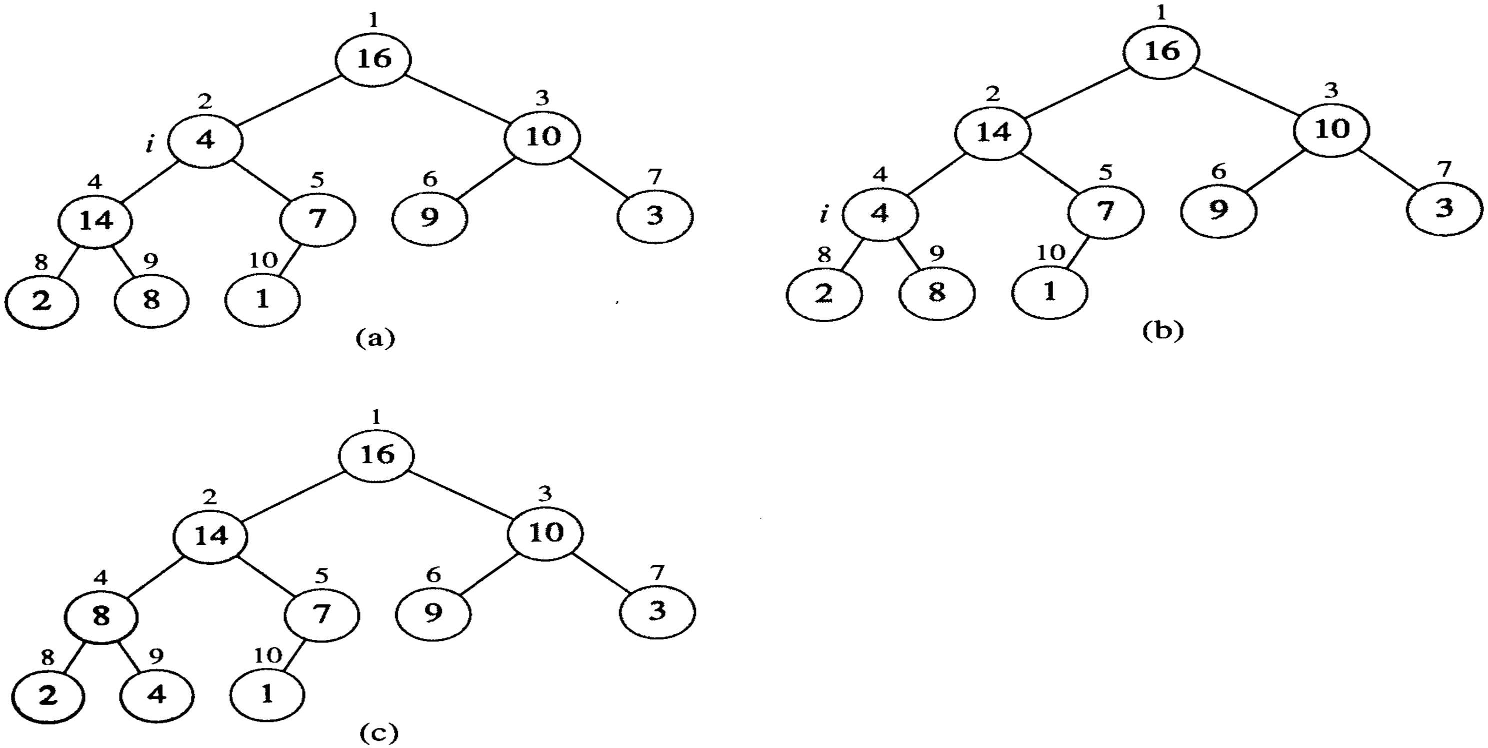

Figure 7.2 The action of HEAPIFY$(A, 2)$, where *heap-size*$[A] = 10$. **(a)** The initial configuration of the heap, with $A[2]$ at node $i = 2$ violating the heap property since it is not larger than both children. The heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the heap property for node 4. The recursive call HEAPIFY$(A, 4)$ now sets $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call HEAPIFY$(A, 9)$ yields no further change to the data structure.

Heapify (A, i)

$l \leftarrow$ left(i)
$r \leftarrow$ right(i)
if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
    then $largest \leftarrow l$
    else $largest \leftarrow i$
if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
    then $largest \leftarrow r$
if $largest \neq i$
    then exchange $A[i] \leftrightarrow A[largest]$
        Heapify(A, $largest$)

Running Time for Heapify(A, i)

$l \leftarrow$ left(i)
$r \leftarrow$ right(i)
if $l \leq$ *heap-size*[A] and $A[l] > A[i]$
    then *largest* $\leftarrow l$
    else *largest* $\leftarrow i$
if $r \leq$ *heap-size*[A] and $A[r] > A[$*largest*$]$
    then *largest* $\leftarrow r$
if *largest* $\neq i$
    then exchange $A[i] \leftrightarrow A[$*largest*$]$
        Heapify(A, *largest*)

T(i) =

$\Theta(1)$ +

T(*largest*)

Running Time for Heapify(A, n)

- So, $T(n) = T(largest) + \Theta(1)$
- Also, $largest \le 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- $T(n) \le T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$
- Alternately, Heapify takes $O(h)$ where h is the height of the node where Heapify is applied

Build-Heap(A)

1. $heap\text{-}size[A] \leftarrow length[A]$
2. for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
3. do Heapify(A, i)

Running Time

- The time required by Heapify on a node of height h is $O(h)$
- Express the total cost of Build-Heap as

$$\sum_{h=0 \text{ to } \lg n} \lceil n / 2^{h+1} \rceil O(h) = O(n \sum_{h=0 \text{ to } \lg n} h/2^h)$$

And, $\sum_{h=0 \text{ to } \infty} h/2^h = (1/2) / (1 - 1/2)^2 = 2$

Therefore, $O(n \sum_{h=0 \text{ to } \lg n} h/2^h) = O(n)$

  – Can build a heap from an unordered array in linear time

Heapsort (A)

Build-Heap(A)
for $i \leftarrow length[A]$ downto 2
  do exchange $A[1] \leftrightarrow A[i]$
    $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
    Heapify(A, 1)

Algorithm Analysis

- In-place
- Not Stable
- Build-Heap takes $O(n)$ and each of the $n\text{-}1$ calls to Heapify takes time $O(\lg n)$.
- Therefore, $T(n) = O(n \lg n)$

Priority Queues

- A data structure for maintaining a set S of elements, each with an associated value called a *key*.
- Applications: scheduling jobs on a shared computer, prioritizing events to be processed based on their predicted time of occurrence.
- Heap can be used to implement a priority queue.

Basic Operations

- **Insert(S, x) - inserts the element x into the set S, i.e. $S \rightarrow S \cup \{x\}$**

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

- **Extract-Max(S) - removes and returns the element of S with the largest key**

Heap-Extract-Max(A)

if *heap-size*[A] < 1
    then error "heap underflow"
max $\leftarrow$ $A[1]$
$A[1] \leftarrow A[\textit{heap-size}[A]]$
heap-size[A] $\leftarrow$ *heap-size*[A] - 1
Heapify(A, 1)
return max

Heap-Insert(*A*, *key*)

$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$

$i \leftarrow heap\text{-}size[A]$

while $i > 1$ **and** $A[\text{Parent}(i)] < key$

  do $A[i] \leftarrow A[\text{Parent}(i)]$

    $i \leftarrow \text{Parent}(i)$

$A[i] \leftarrow key$

Running Time

- Running time of Heap-Extract-Max is $O(\lg n)$.
  - Performs only a constant amount of work on top of Heapify, which takes $O(\lg n)$ time
- Running time of Heap-Insert is $O(\lg n)$.
  - The path traced from the new leaf to the root has length $O(\lg n)$.

Examples

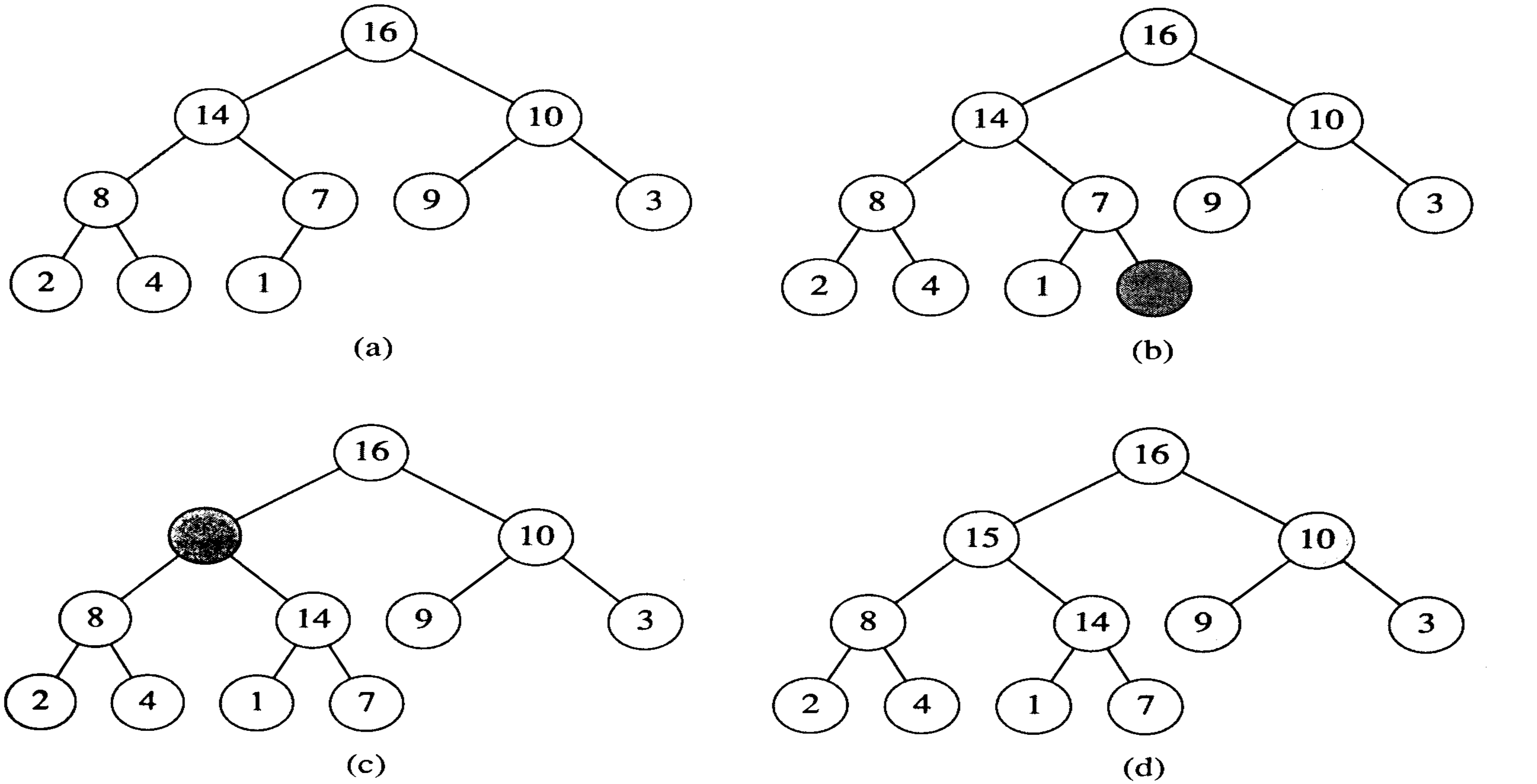

Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.