Heapsort (II)

Textbook, Chapter 7

CSCE310: Data Structures and Algorithms

`www.cse.unl.edu/~choueiry/S01-310/`

Berthe Y. Choueiry (Shu-we-ri)

Ferguson Hall, Room 104

`choueiry@cse.unl.edu`, Tel: (402)472-5444

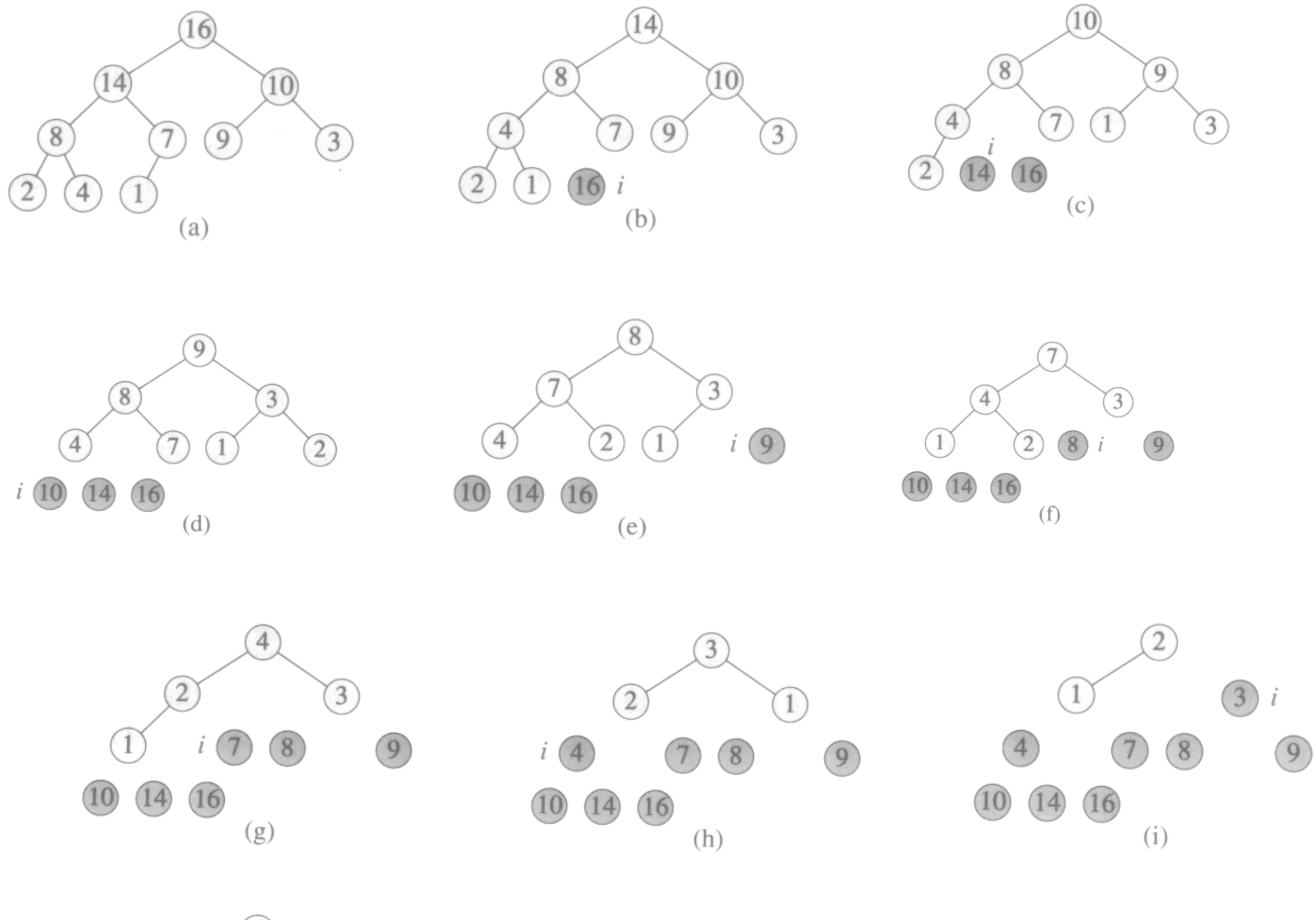
Heapsort: Principle and algorithm

Input: Array $A[1 \ldots n]$, where $n = length[A]$

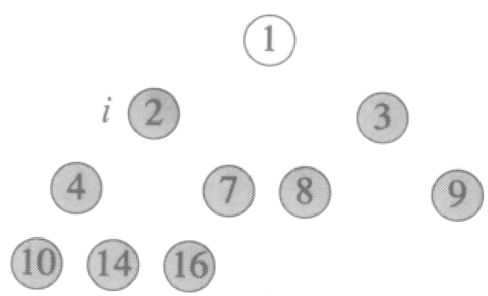
1. Build a heap out of the array
   the maximum element will be in $A[1]$, it should be last!!
2. Exchange $A[1]$ with $A[n]$
3. Reset heap size to $(n - 1)$
4. Heapify $A[1 \ldots (n-1)]$, repeat... down to a heap of size 2

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

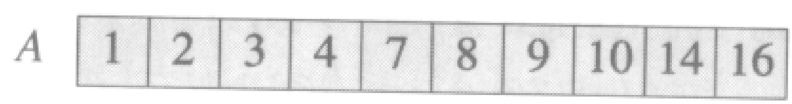
Heapsort: Example $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$



Heapsort: Example $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$ *(cont')*



(j)



(k)

Heapsort: Complexity

For an array A of size n

- `Build-Heap` is called once, its complexity id $O(n)$
- `Heapify` is called $n-1$ times, its complexity is $O(\lg n)$
- Complexity of `Heapsort` is $O(n + (n-1)\lg n) = O(n \lg n)$

Heapsort: Utility

- `Heapsort` is an excellent algorithm, however `Quicksort` is better in practice
- Heap data structure has enormous utility e.g., efficient priority queue

Priority queue as a data structure

- Data structure for maintaining a set S of elements, each with a value called key
- Supports operations:
  `Insert`(S, x) (i.e., $S \leftarrow S \cup \{x\}$)
  `Maximum`(S) returns element of S with largest key
  `Extract-Max`(S) removes and returns the element of S with largest key

Priority queue: Applications

1. Job scheduling on a shared resource (e.g., computer)
   Priority queue keeps track of jobs and their relative importance.
   `Insert` allows us to add new jobs at any time
   `Extract-Max` selects highest priority jobs when current job is finished/interrupted

2. Event-driven simulator
   Queue of events to be simulated, each has occurrence time → key value
   Occurrence of event causes other events to be simulated in future → `Insert` `Extract-Min` and `Minimum` instead of `Extract-Max` and `Maximum`

Implementation → a heap

```
Heap-Extract-Max(A)
If heap-size[A] < 1
   then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] − 1
Heapify(A, 1)
return max
```

→ Returns $A[1]$

→ Places the last element of the heap in $A[1]$

→ Decrements size of heap

→ Heapifies the array

→ Running time is in $O(\lg n)$ (constant effort and 1 heapify)

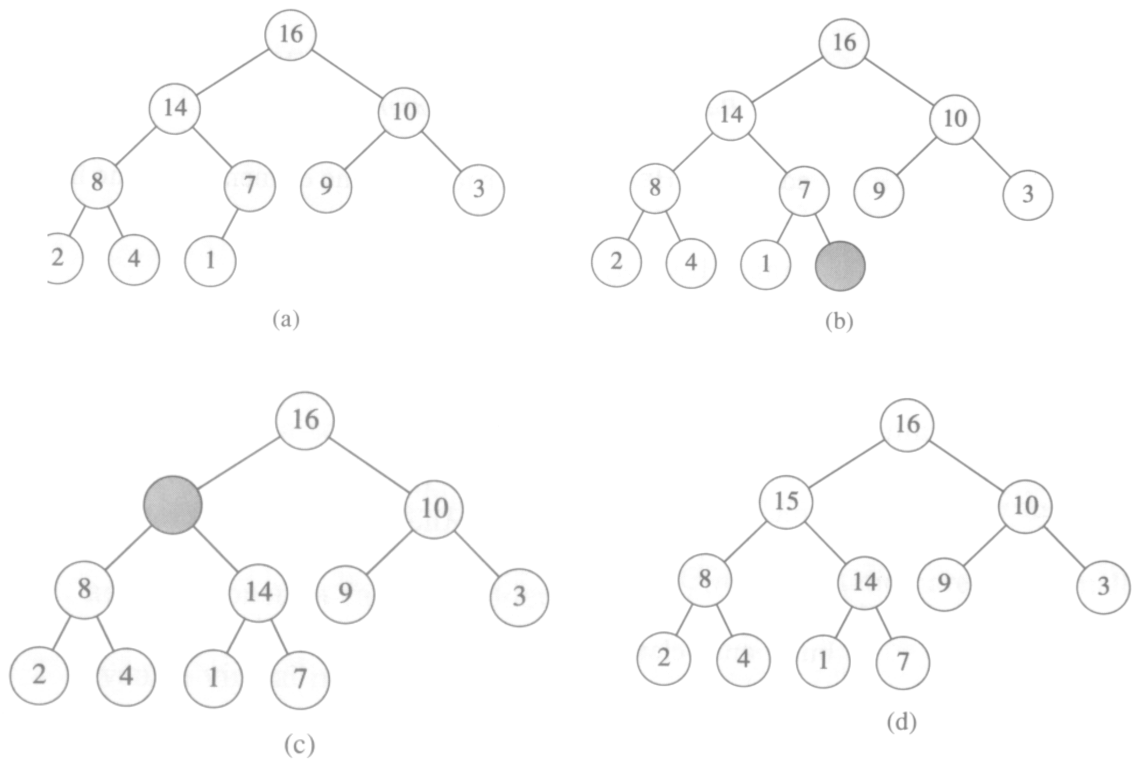
```
Heap-Insert(A, key)

heap-size[A] ← heap-size[A] + 1
i ← heap-size[A]
while i > 1 and A[Parent(i)] < key
      do A[i] ← A[Parent(i)]
         i ← Parent(i)
A[i] ← key
```

→ First expands the heap by adding 1 new leaf

→ Then traverses path from new leaf to root to find a proper place for the new element

Heap-Insert(A, key): Example

$key = 15$



Running time is $O(\lg n)$, since path from new leaf to root has length $\lg n$

Heap as a priority queue

For any set S of n elements:

- `Heap-Maximum`(A) is in $\Theta(1)$
- `Heap-Extract-Max`(A) is in $O(\lg n)$
- `Heap-Insert`(A, key) is in $O(\lg n)$

Bubble-sort

Notes of Dr. Cusack

- Go through the list in order, swapping two elements if their keys are out of order
- Repeat until no swaps are performed. The list is sorted
- n passes suffice
- Similar to Insertion-sort, but has lots of swaps

Bubble-Sort (A)
For $i \rightarrow n-1$ downto 1
    For $j \rightarrow 1$ to i
        When $A[j-1] > A[j]$, $A[j-1] \leftrightarrow A[j]$

Name	Upper bound	Lower bound	Tight bound
Selection-sort	$O(n^2)$	$\Omega(n)$	
Insertion-sort	$O(n^2)$	$\Omega(n)$	
Bubble-sort	$O(n^2)$		
Merge-sort			$\Theta(n \lg n)$
Heap-sort	$O(n \lg n)$		
Quick-sort	$O(n^2)$	$\Omega(n \lg n)$	