طراحی و تحلیل الگوریتم ها

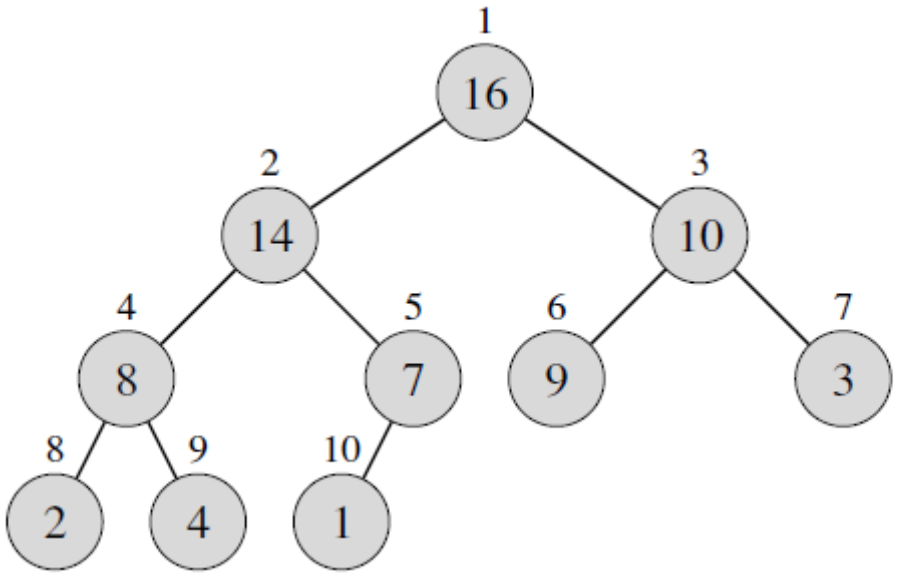
دکتر امیر لکی زاده

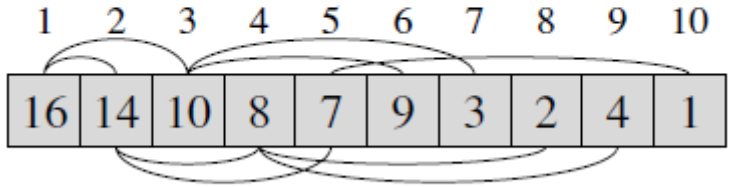
استادیار گروه مهندسی کامپیوتر دانشگاه قم

Heap sort

- In place sorting
- Stable sorting

Heap sort





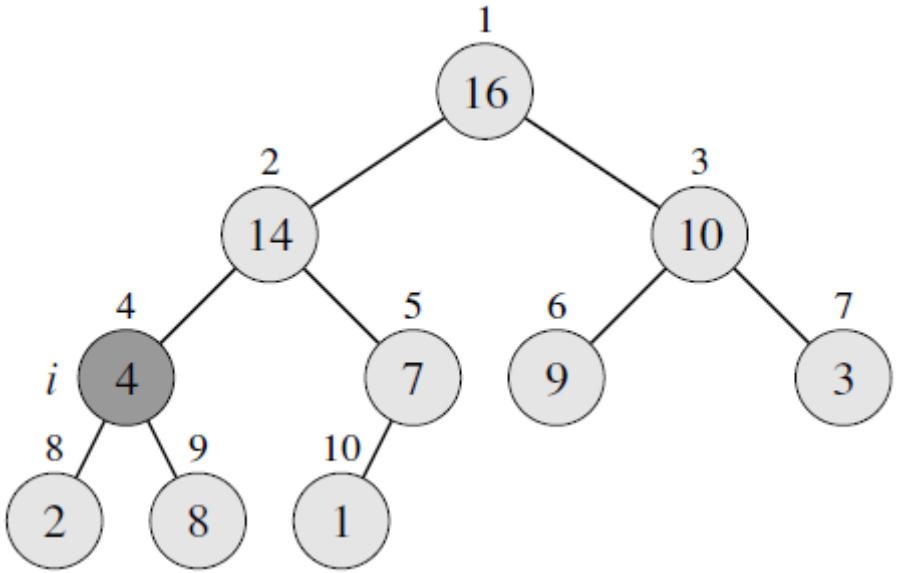
PARENT(i)
    return $\lfloor i/2 \rfloor$

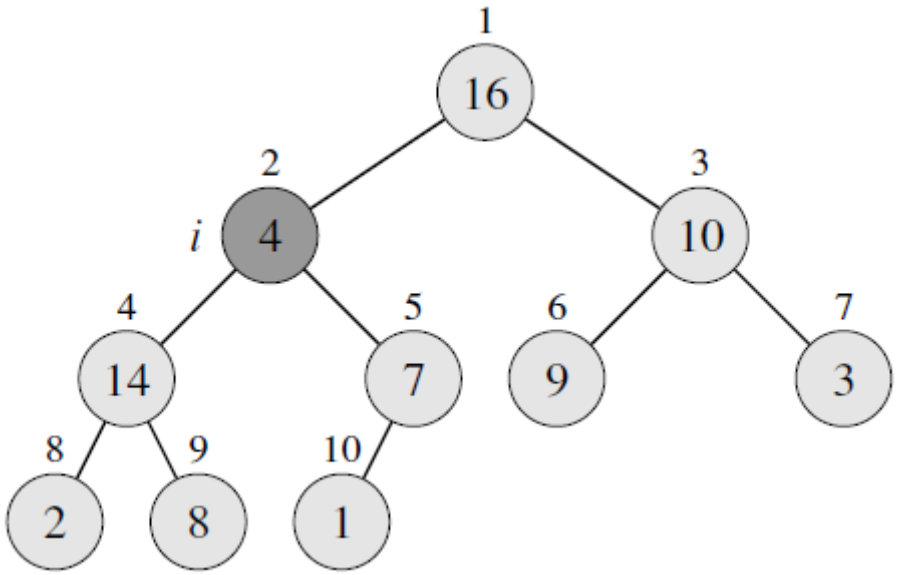
LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

Heap sort

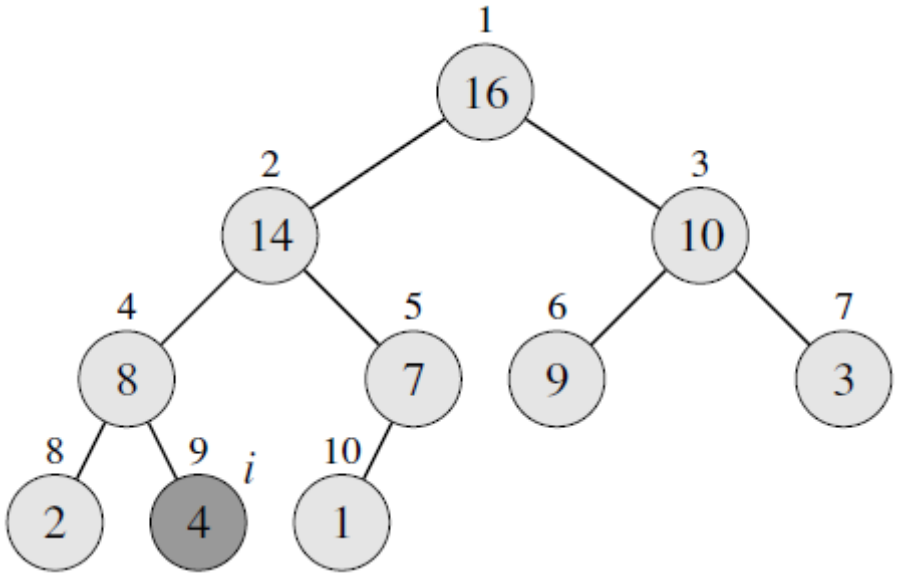
Heapify Alg





Heap sort

Heapify Alg



Heap sort

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```

Heap sort

Heapify Alg

6.2-6

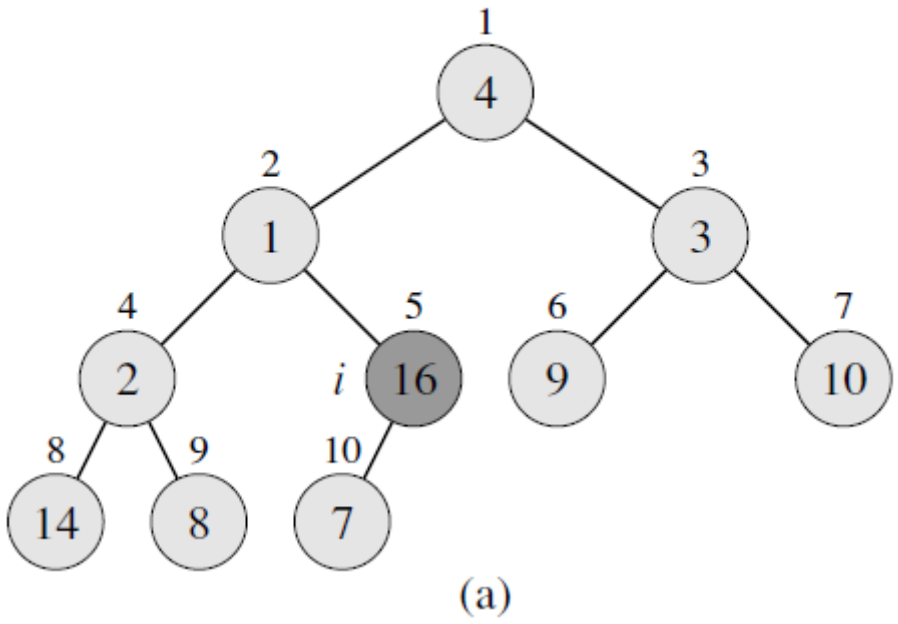
Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (*Hint:* For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a path from the root down to a leaf.)

$$T(n) \le T(2n/3) + \Theta(1) .$$

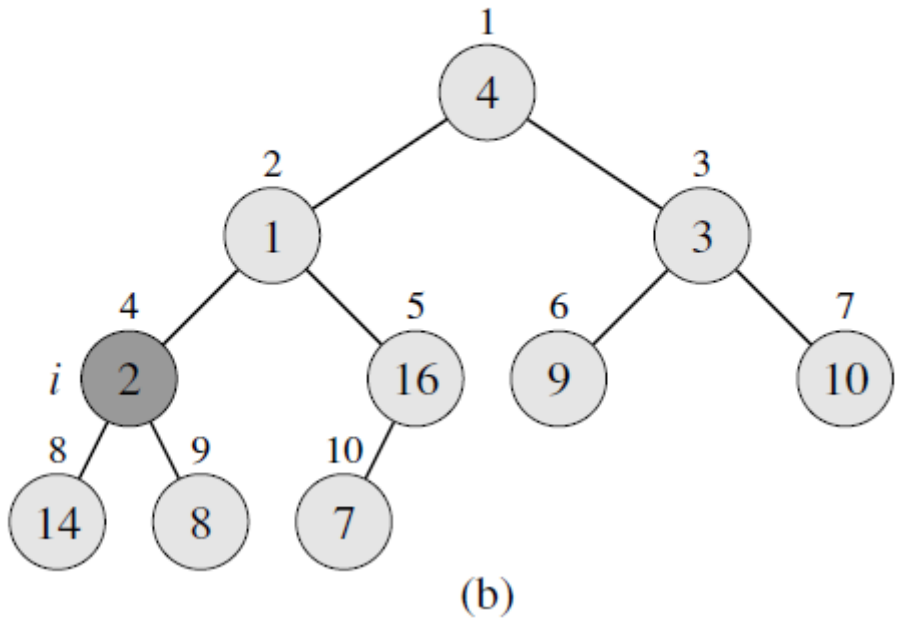
Heap sort

Building a heap

A	4	1	3	2	16	9	10	14	8	7



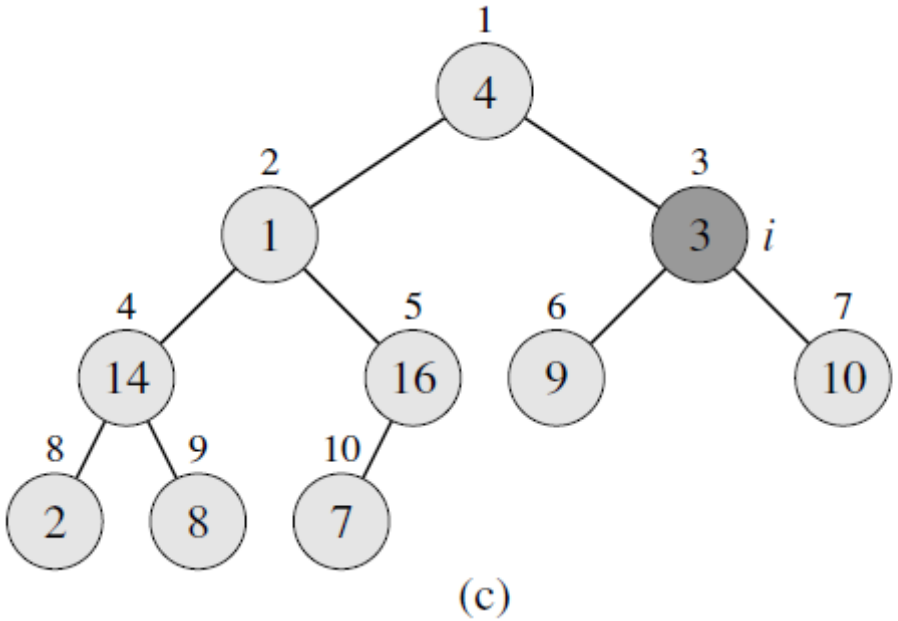
(a)



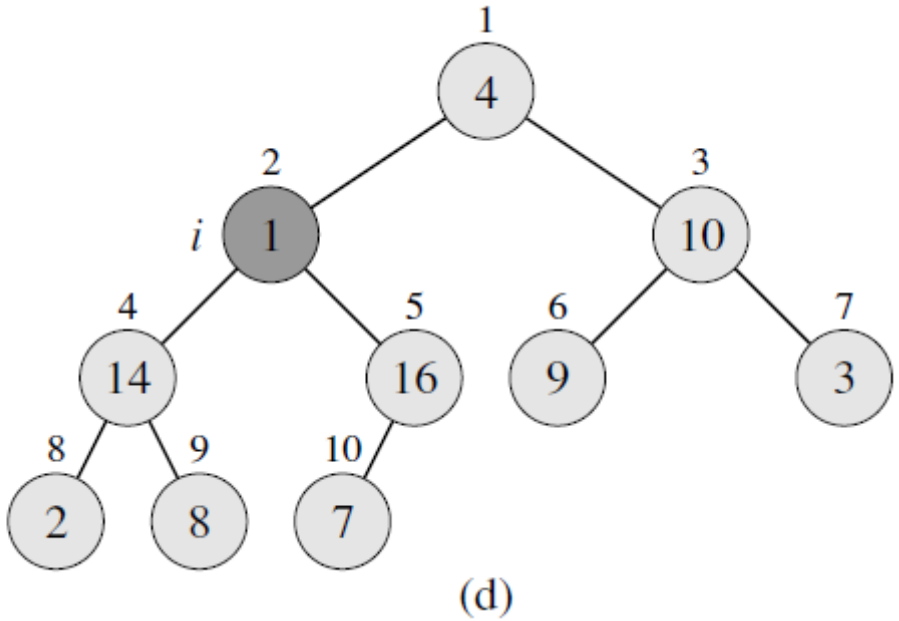
(b)

Heap sort

Building a heap



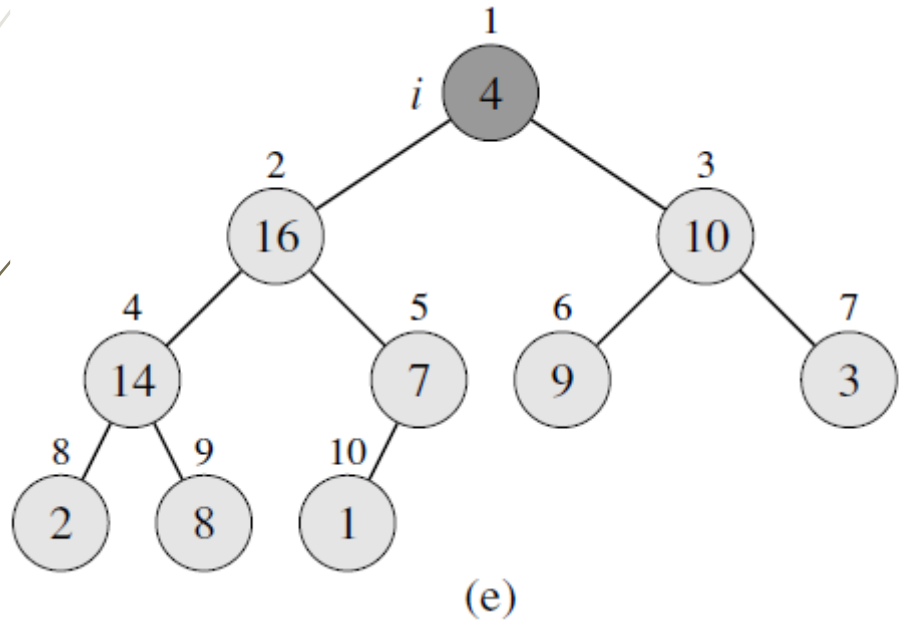
(c)



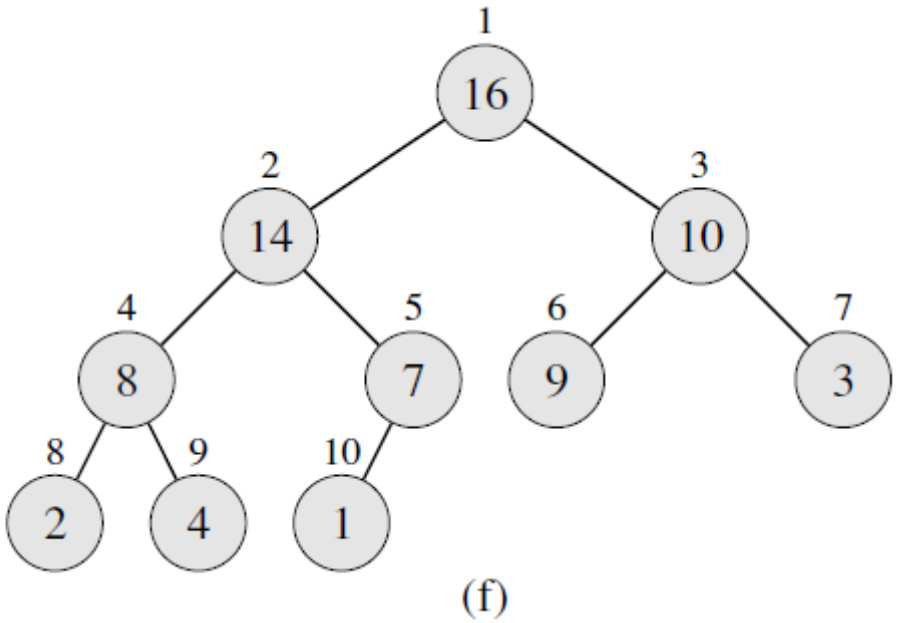
(d)

Heap sort

Building a heap



(e)



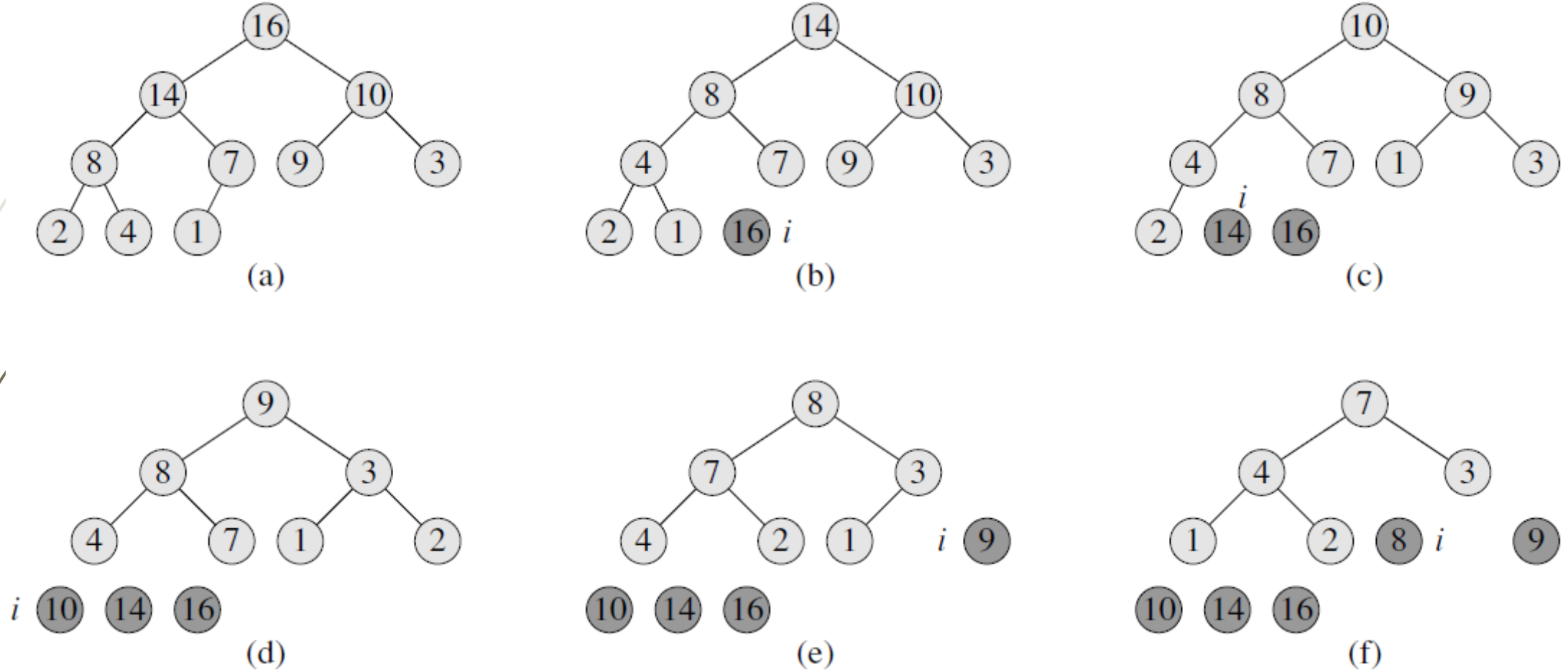
(f)

Heap sort

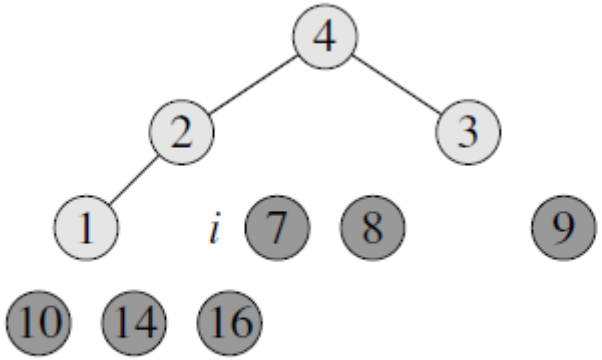
Building a heap

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

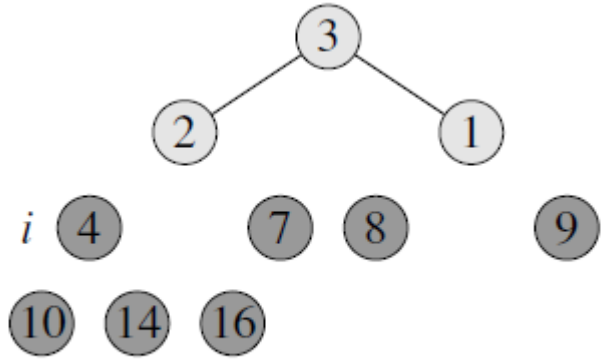
Heap sort


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

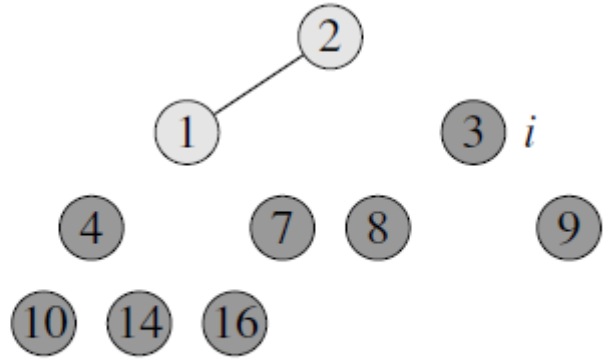
Heap sort



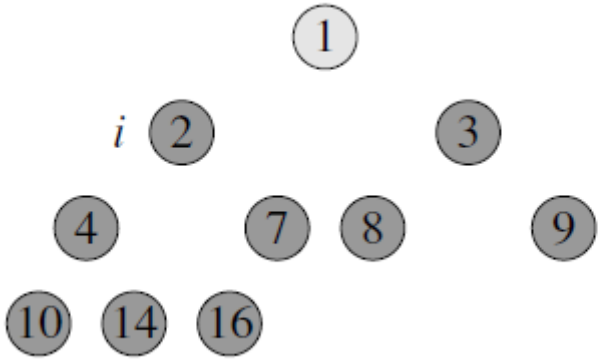
(g)



(h)



(i)



(j)

(k)

Heap sort

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

Heap sort

6.4-3

What is the running time of heapsort on an array A of length n that is already sorted in increasing order? What about decreasing order?

6.4-4

Show that the worst-case running time of heapsort is $\Omega(n \lg n)$.

Priority Queue

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations.

INSERT(S, x) inserts the element x into the set S. This operation could be written as $S \leftarrow S \cup \{x\}$.

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.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Priority Queue

One application of max-priority queues is to schedule jobs on a shared computer. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY. A min-priority queue can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program uses EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, they are inserted into the min-priority queue using INSERT. We shall see other uses for min-priority queues, highlighting the DECREASE-KEY operation, in Chapters 23 and 24.

Priority Queue

```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2      then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Priority Queue

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2     then error "new key is smaller than current key"
3  A[i] ← key
4  while i > 1 and A[PARENT(i)] < A[i]
5        do exchange A[i] ↔ A[PARENT(i)]
6           i ← PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
1  heap-size[A] ← heap-size[A] + 1
2  A[heap-size[A]] ← −∞
3  HEAP-INCREASE-KEY(A, heap-size[A], key)
```

Priority Queue

6.5-6

Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue. (Queues and stacks are defined in Section 10.1.)

6.5-7

The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n-element max-heap.

6.5-8

Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (*Hint:* Use a min-heap for k-way merging.)

Priority Queue

6-1 Building a heap using insertion

The procedure BUILD-MAX-HEAP in Section 6.3 can be implemented by repeatedly using MAX-HEAP-INSERT to insert the elements into the heap. Consider the following implementation:

```
BUILD-MAX-HEAP′(A)
1  heap-size[A] ← 1
2  for i ← 2 to length[A]
3        do MAX-HEAP-INSERT(A, A[i])
```

a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP′ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

b. Show that in the worst case, BUILD-MAX-HEAP′ requires $\Theta(n \lg n)$ time to build an n-element heap.

Priority Queue

6-2 Analysis of d-ary heaps

A ***d-ary heap*** is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.

a. How would you represent a d-ary heap in an array?

b. What is the height of a d-ary heap of n elements in terms of n and d?

c. Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

d. Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n.

e. Give an efficient implementation of INCREASE-KEY(A, i, k), which first sets $A[i] \leftarrow \max(A[i], k)$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.