UMC 201: Data Structures & Algorithms

Lecture 3: Heaps, Heapsort, Priority queues

Indian Institute of Science

Recap

- We used the *divide-and-conquer* technique to design a sorting algorithm, namely merge sort.
- The algorithm has a running time of $O(n \log n)$ but it does *not* sort the input array in place.

Recap

- We used the *divide-and-conquer* technique to design a sorting algorithm, namely merge sort.
- The algorithm has a running time of $O(n \log n)$ but it does *not* sort the input array in place.
- We discussed Strassen's $o(n^3)$ -time matrix multiplication algorithm, which also uses the divide-and-conquer strategy.

Recap

- We used the *divide-and-conquer* technique to design a sorting algorithm, namely merge sort.
- The algorithm has a running time of $O(n \log n)$ but it does *not* sort the input array in place.
- We discussed Strassen's $o(n^3)$ -time matrix multiplication algorithm, which also uses the divide-and-conquer strategy.
- Finally, we saw the *recursion tree method* for solving recurrence relations.

This lecture

- In this lecture, we'll introduce another sorting algorithm, namely heapsort, which has running time $O(n \log n)$ and also *sorts the array in place*.
- The algorithm uses a data structure called heap to manage information.

This lecture

- In this lecture, we'll introduce another sorting algorithm, namely heapsort, which has running time $O(n \log n)$ and also *sorts the array in place*.
- The algorithm uses a data structure called heap to manage information.

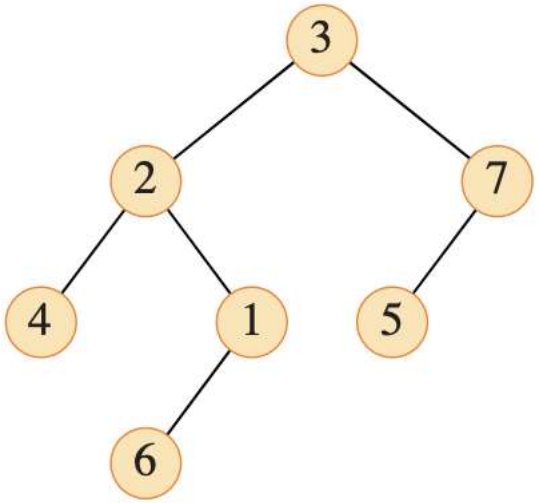
- The heap data structure is *an array* that can be viewed as a *nearly* complete binary tree.

Binary Trees

Binary trees

- Definition. A binary tree is a structure defined on a finite set of nodes that is composed of a root node, a binary tree called its left subtree, and a binary tree called its right subtree.

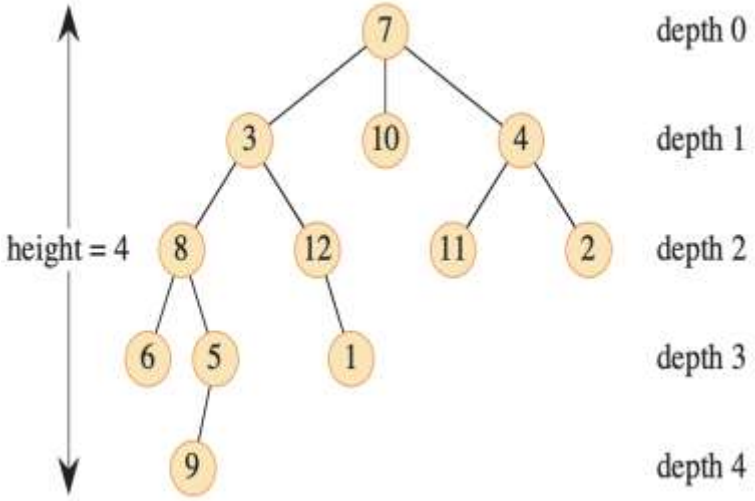
- The root of the left (right) subtree is the left (right) child of the root of the entire tree.



Binary trees

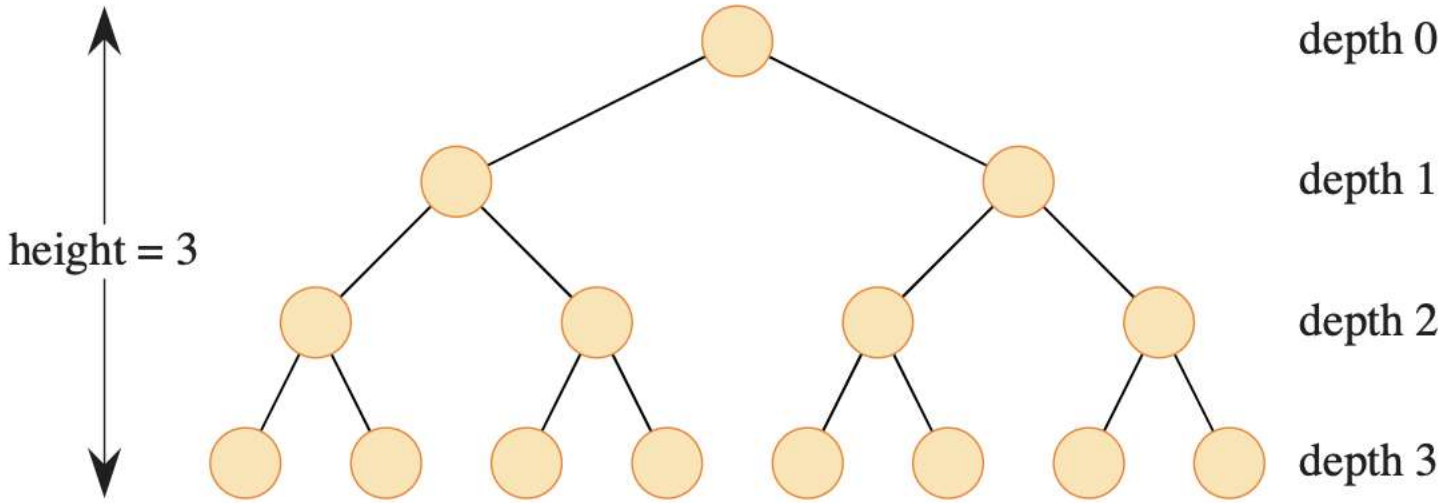
- The number of children of a node x is the degree of x. The length of the simple path from the root to a node x is the depth of x. A level of a tree consists of all nodes at the same depth.

- The height of a node is the number of edges on the longest simple downward path from the node to leaf.



Complete binary trees

- Definition. A complete binary tree is a binary tree in which *all leaves have the same depth* and *all internal nodes have degree two*.

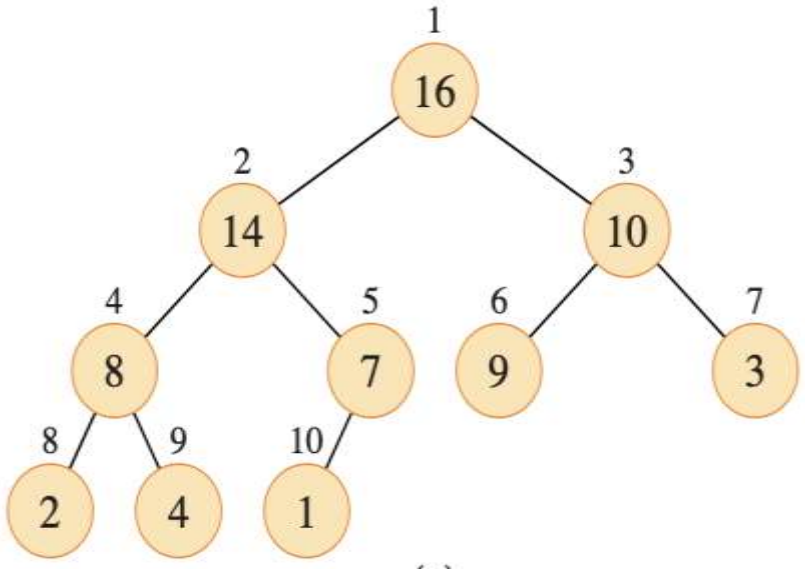


Heaps

The heap data structure

- The heap data structure is *an array* that can be viewed as a *nearly* complete binary tree.
- Each node of the tree corresponds to an element of the array.

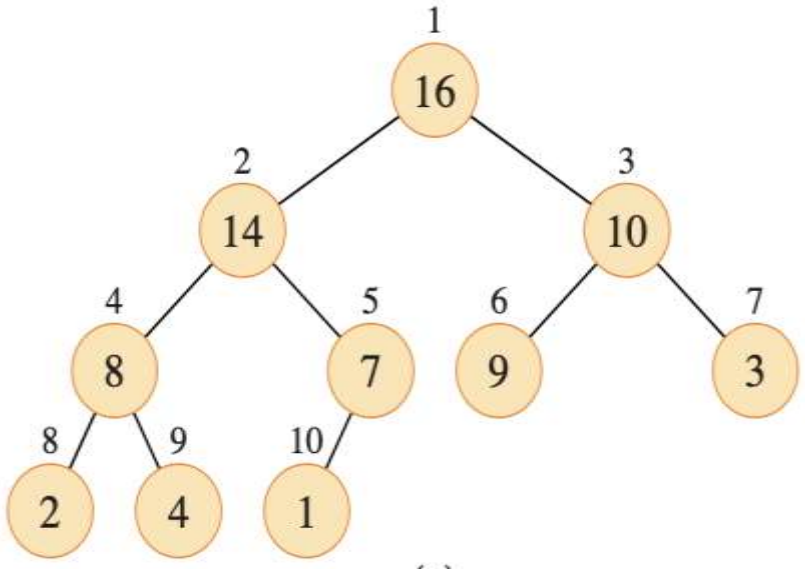
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.



The heap data structure

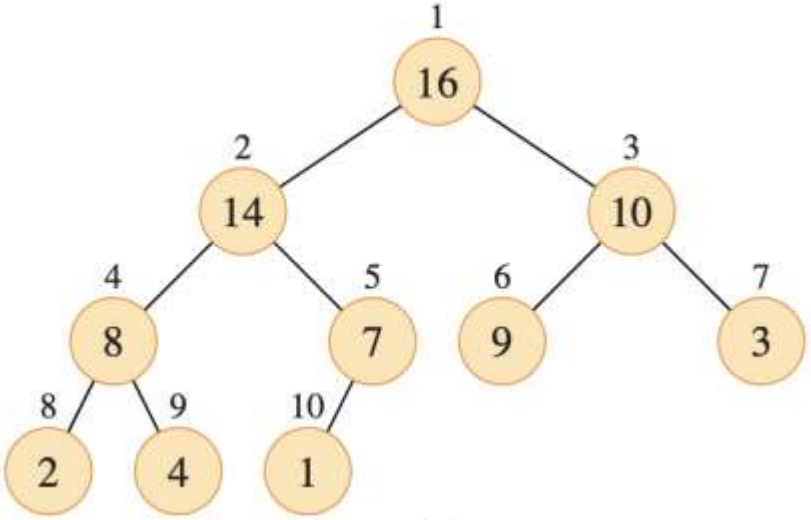
- The heap data structure is *an array* that can be viewed as a *nearly* complete binary tree.
- Each node of the tree corresponds to an element of the array.

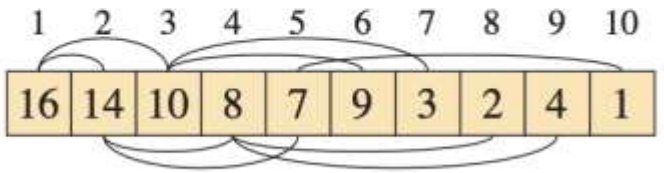
- Obs. Since a heap of n elements is based on a complete binary tree, its *height* is $\Theta(\log n)$, more precisely, $\lfloor \log n \rfloor$.



The heap data structure

- An array $A[1:n]$ that represents a heap is an object with an attribute $A.heap-size$, which represents the *number of nodes* in the heap.





- The root of the tree is $A[1]$.

Finding parent, left child & right child

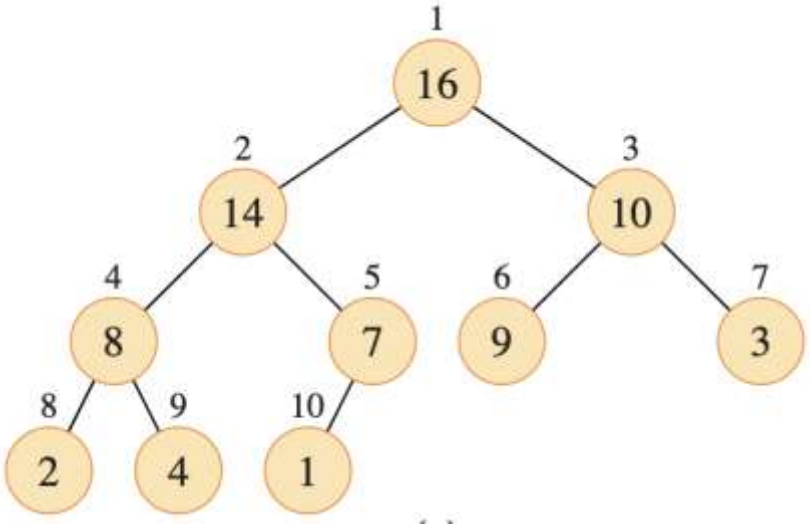
- Given the index i of a node, we can easily compute the indices of its *parent*, *left child*, and *right child*.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

- The proofs of correctness of these procedures follow easily *by induction*.

Max-heap and Min-heap

- In a max-heap, the *max-heap property* is that for every node i other than the root, $A[\text{PARENT}(i)] \geq A[i]$.
- Thus, the *largest element* in a max-heap is stored at the *root*, and the subtree rooted at a node contains values *no larger* than that contained at the node.



Max-heap and Min-heap

- In a max-heap, the *max-heap property* is that for every node i other than the root, $A[\text{PARENT}(i)] \geq A[i]$.
- Thus, the *largest element* in a max-heap is stored at the *root*, and the subtree rooted at a node contains values *no larger* than that contained at the node.
- In a min-heap, the *min-heap property* is that for every node i other than the root, $A[\text{PARENT}(i)] \leq A[i]$.
- The *smallest element* in a min-heap is at the *root*.

Max-heap and Min-heap

- In a max-heap, the *max-heap property* is that for every node i other than the root, $A[\text{PARENT}(i)] \geq A[i]$.
- Thus, the *largest element* in a max-heap is stored at the *root*, and the subtree rooted at a node contains values *no larger* than that contained at the node.
- In a min-heap, the *min-heap property* is that for every node i other than the root, $A[\text{PARENT}(i)] \leq A[i]$.
- The *smallest element* in a min-heap is at the *root*.
- Heapsort uses max-heaps. Both Max and Min-heaps are used to implement priority queues.

Two basic procedures

- The MAX-HEAPIFY procedure, which runs in $O(\log n)$ time on an array (heap) of size n, is the key to *maintaining the max-heap property*.

Two basic procedures

- The MAX-HEAPIFY procedure, which runs in $O(\log n)$ time on an array (heap) of size n, is the key to *maintaining the max-heap property*.

- The BUILD-MAX-HEAP procedure, which runs in $O(n)$ time, *produces a max-heap* from an unordered input array of size n.

Maintaining the max-heap property

- The inputs of MAX-HEAPIFY are an array A with the *heap-size* attribute and an index i into the array.

Maintaining the max-heap property

- The inputs of MAX-HEAPIFY are an array A with the *heap-size* attribute and an index i into the array.

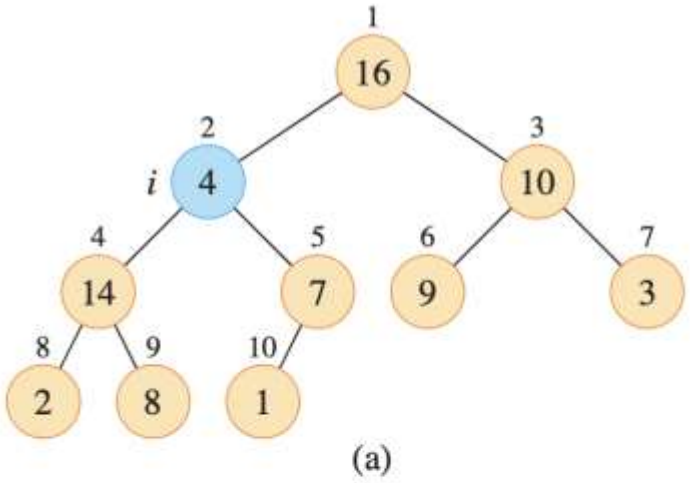
- When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ *might be smaller than its children*, thus violating the max-heap property.

Maintaining the max-heap property

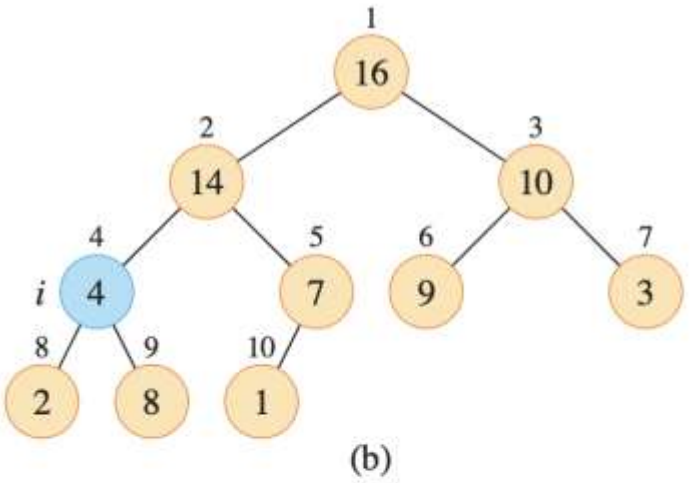
- The inputs of MAX-HEAPIFY are an array A with the *heap-size* attribute and an index i into the array.

- When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that $A[i]$ *might be smaller than its children*, thus violating the max-heap property.
- MAX-HEAPIFY lets the value at $A[i]$ "*float down*" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

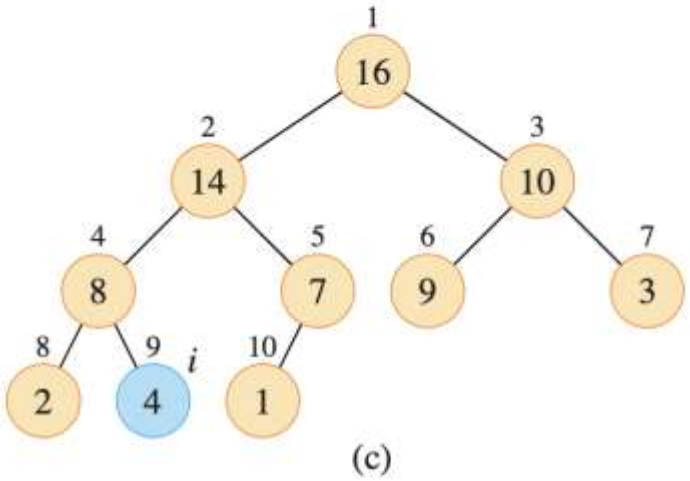
MAX-HEAPIFY: An example



(a)



(b)



(c)

MAX-HEAPIFY: Pseudocode

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

Compute the indices of the left and right children of i.

MAX-HEAPIFY: Pseudocode

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

Compute the indices of the left and right children of i.

Steps 3-7 determines the largest among $A[i]$, $A[\text{LEFT}(i)]$, & $A[\text{RIGHT}(i)]$ and stores the *index of the largest element* in $largest$.

MAX-HEAPIFY: Pseudocode

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

Compute the indices of the left and right children of i.

Steps 3-7 determines the largest among $A[i]$, $A[\text{LEFT}(i)]$, & $A[\text{RIGHT}(i)]$ and stores the *index of the largest element* in $largest$.

Positions i & $largest$ swap their contents, if required.

MAX-HEAPIFY: Pseudocode

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

Compute the indices of the left and right children of i.

Steps 3-7 determines the largest among $A[i]$, $A[\text{LEFT}(i)]$, & $A[\text{RIGHT}(i)]$ and stores the *index of the largest element* in $largest$.

Positions i & $largest$ swap their contents, if required.

- Time complexity. ***Steps 1-9*** take $\Theta(1)$ time. Suppose h is the height of i.

MAX-HEAPIFY: Pseudocode

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

Compute the indices of the left and right children of i.

Steps 3-7 determines the largest among $A[i]$, $A[\text{LEFT}(i)]$, & $A[\text{RIGHT}(i)]$ and stores the *index of the largest element* in $largest$.

Positions i & $largest$ swap their contents, if required.

- Time complexity. As the height of the subtree rooted at i decreases by 1 with each recursive call (***Step 10***), the overall running time is $\text{T}(h) = \text{T}(h-1) + \Theta(1) = \text{O}(h) = \text{O}(\log n)$.

Building a heap

- The procedure BUILD-MAX-HEAP coverts an array $A[1:n]$ into a max-heap by calling MAX-HEAPIFY in a *bottom-up manner*.

Building a heap

- The procedure BUILD-MAX-HEAP coverts an array $A[1:n]$ into a max-heap by calling MAX-HEAPIFY in a *bottom-up manner*.

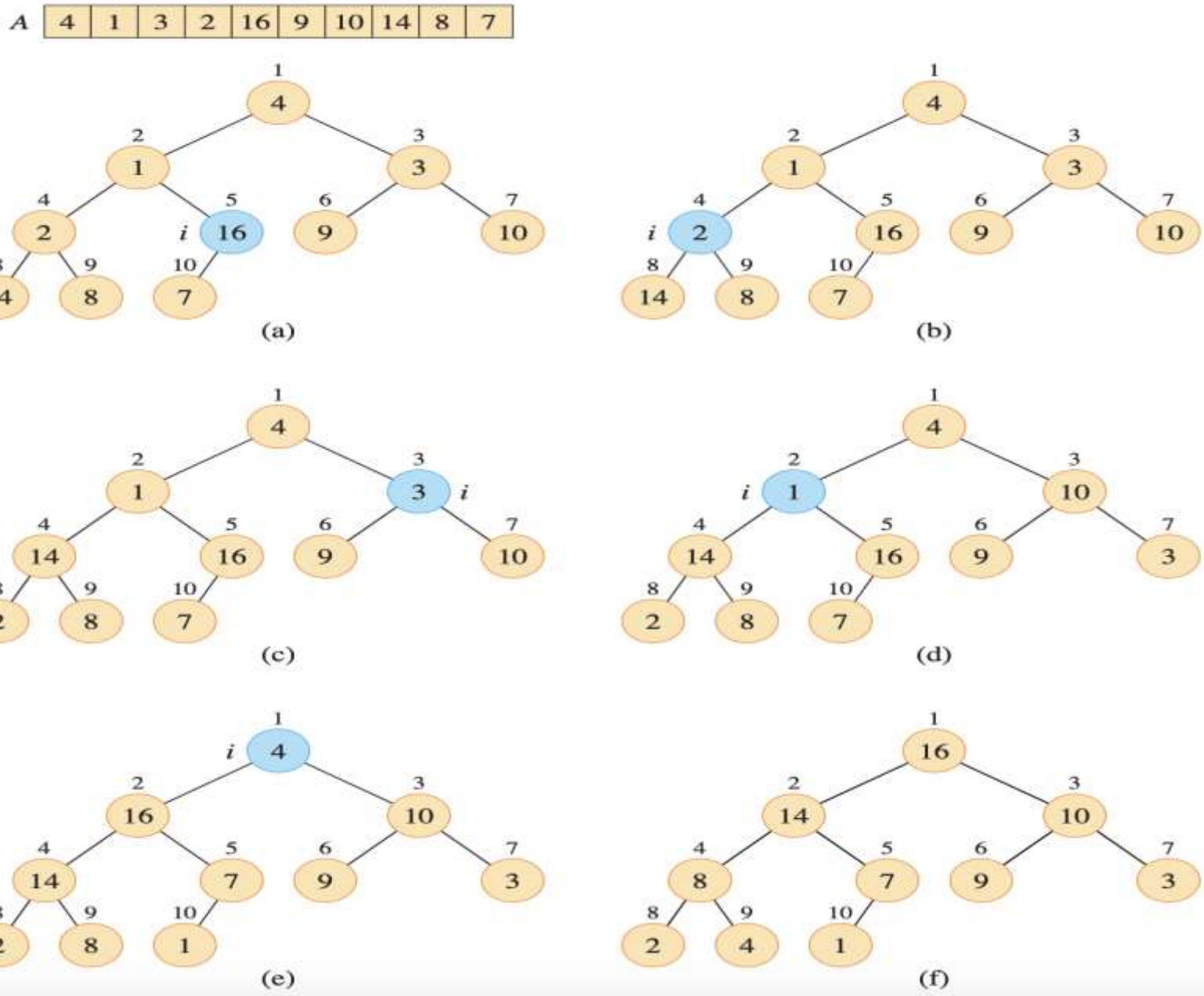
- Obs. The elements in the subarray $A[\lfloor n/2 \rfloor + 1 : n]$ are all *leaves* of the tree.
- *Proof.* Every node following the parent of the node indexed by n is a leaf. So, every index after $\text{PARENT}(n) = \lfloor n/2 \rfloor$ corresponds to a leaf node.

BUILD-MAX-HEAP: Pseudocode

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

- BUILD-MAX-HEAP goes through the *non-leaf nodes* of the tree and runs MAX-HEAPIFY on each one in a bottom-up manner.

BUILD-MAX-HEAP: An example



BUILD-MAX-HEAP: Time complexity

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

- Time complexity. As MAX-HEAPIFY takes $O(\log n)$ time, the above procedure takes $O(n \log n)$ time.
- We can actually give a *linear time bound* as follows…

BUILD-MAX-HEAP: Time complexity

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

- Obs. There are at most $n/2^h$ nodes of height h in any n-element heap.

- *Proof.* Suppose there are m nodes of height h. Observe that the subtree rooted at these nodes are *disjoint*. All (but one) of these subtrees are complete binary trees. Thus, $(m-1)(2^{h+1}-1)+2^h \leq n$. Simplifying, we get $m \leq n/2^h$.

BUILD-MAX-HEAP: Time complexity

```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

- Obs. There are at most $n/2^h$ nodes of height h in any n-element heap.

- Time complexity. BUILD-MAX-HEAP takes time $\leq \sum_{h=0}^{\lfloor \log n \rfloor} (n/2^h) \cdot ch$, where c is a constant.
- The above sum is $O(n)$.

Heapsort

Heapsort: The algorithm

- The algorithm starts by calling the BUILD-MAX-HEAP procedure on the input array $A[1:n]$.
- Since the max element is stored in $A[1]$, it *exchanges* $A[1]$ with $A[n]$, and reduces $A.heap-size$ by 1.

Heapsort: The algorithm

- The algorithm starts by calling the BUILD-MAX-HEAP procedure on the input array $A[1:n]$.
- Since the max element is stored in $A[1]$, it *exchanges* $A[1]$ with $A[n]$, and reduces $A.heap - size$ by 1.

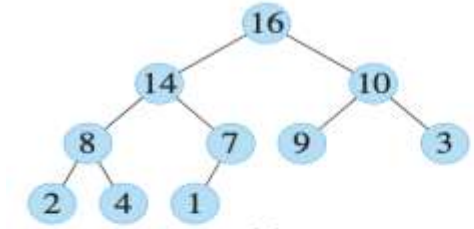
- To restore the max-heap property, it then calls MAX-HEAPIFY$(A, 1)$.
- The algorithm then repeats the process until the heap-size falls below 2.

Heapsort: Pseudocode

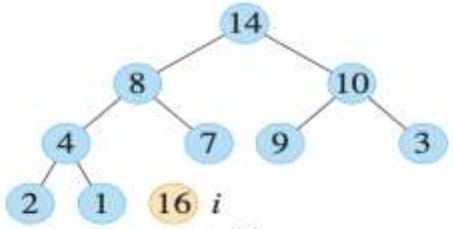
```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

- Note. HEAPSORT sorts the array A *in place*.

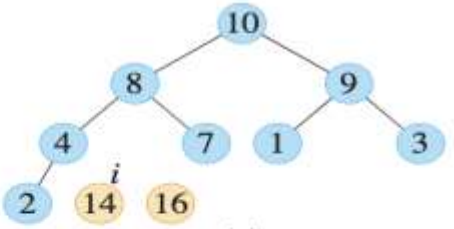
Heapsort: An example



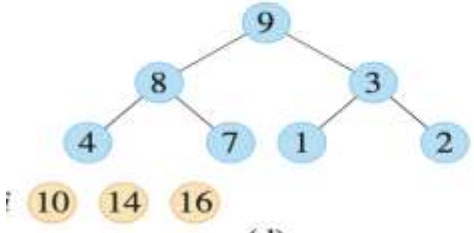
(a)



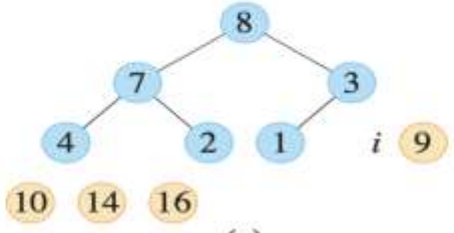
(b)



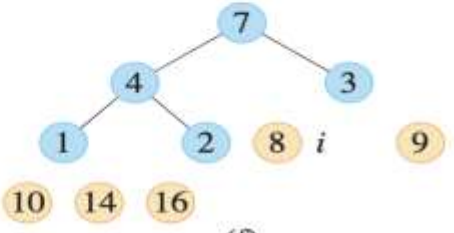
(c)



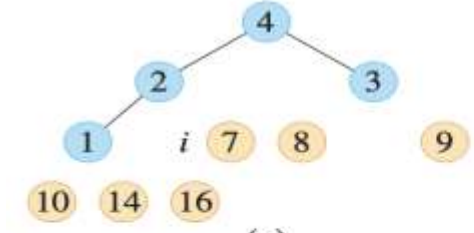
(d)



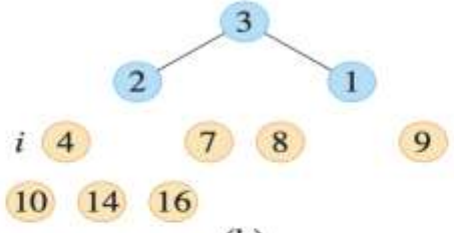
(e)



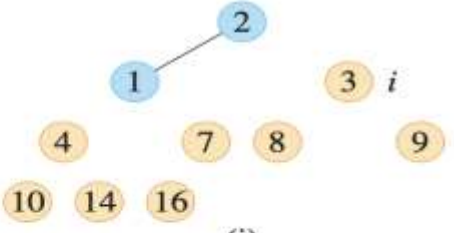
(f)



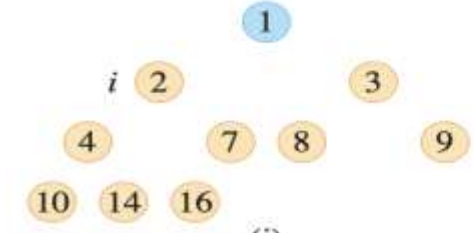
(g)



(h)



(i)



(j)

A	1	2	3	4	7	8	9	10	14	16

(k)

Heapsort: Time complexity

```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

- Time complexity. The HEAPSORT procedure takes $O(n \log n)$ time, since the call to BUILD-MAX-HEAP takes $O(n)$ time and each of the $n - 1$ calls to MAX-HEAPIFY takes $O(\log n)$ time.

Priority Queues

Priority queue data structure

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

Priority queue data structure

- A priority queue is a data structure for *maintaining a set S of elements*, each associated with a key value.
- A max-priority queue supports the operations:
  - INSERT(S, x, k) inserts the element x with key k into S.

Priority queue data structure

- A priority queue is a data structure for *maintaining a set S of elements*, each associated with a key value.
- A max-priority queue supports the operations:
  - INSERT(S, x, k) inserts the element x with key k into S.
  - MAXIMUM(S) returns the element of S with the max key.

Priority queue data structure

- A priority queue is a data structure for *maintaining a set S of elements*, each associated with a key value.
- A max-priority queue supports the operations:
  - INSERT(S, x, k) inserts the element x with key k into S.
  - MAXIMUM(S) returns the element of S with the max key.
  - EXTRACT-MAX(S) removes and returns the element of S with the maximum key.

Priority queue data structure

- A priority queue is a data structure for *maintaining a set S of elements*, each associated with a key value.
- A max-priority queue supports the operations:
  - INSERT(S, x, k) inserts the element x with key k into S.
  - MAXIMUM(S) returns the element of S with the max key.
  - EXTRACT-MAX(S) removes and returns the element of S with the maximum key.
  - INCREASE-KEY(S, x, k) increases the value of element x's key to a new value k, which is $\geq$ the current key value of x.

Priority queue: Application

- We can use max-priority queues to *schedule jobs on a computer* shared among multiple users.
- The max-priority queue keeps track of the jobs to be performed and their relative *priorities*.

Priority queue: Application

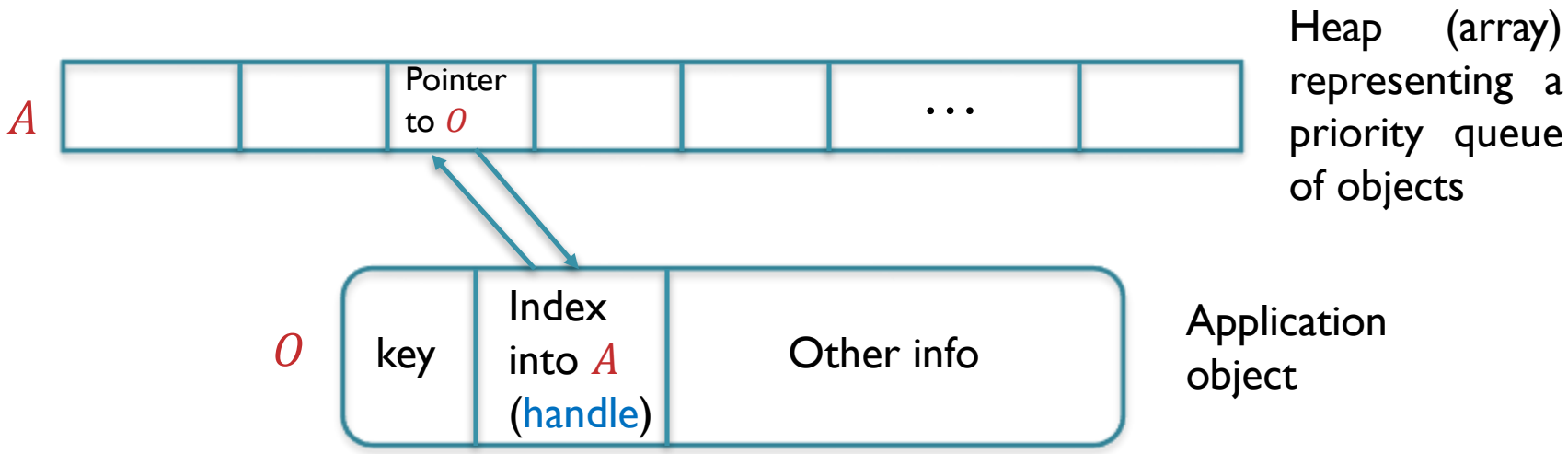
- We can use max-priority queues to *schedule jobs on a computer* shared among multiple users.
- The max-priority queue keeps track of the jobs to be performed and their relative *priorities*.
- When a job is finished or interrupted, the scheduler *selects the highest-priority job* from among those pending by calling EXTRACT-MAX.
- The scheduler can *add a new job* to the queue at any time by calling INSERT.

Priority queue: Using a heap

- Elements of a priority queue correspond to *objects* in an application. Each object contains a *key*.
- If a priority queue is implemented by a heap, we need to determine *which object corresponds to a given heap element, and vice versa*.

Priority queue: Using a heap

- Elements of a priority queue correspond to *objects* in an application. Each object contains a *key*.
- If a priority queue is implemented by a heap, we need to determine *which object corresponds to a given heap element, and vice versa*.



Priority queue: Using a heap

- Elements of a priority queue correspond to *objects* in an application. Each object contains a *key*.
- If a priority queue is implemented by a heap, we need to determine *which object corresponds to a given heap element, and vice versa*.
- A handle within an object *contains the corresponding index* into the heap array. Conversely, each element in the heap contains a *pointer to the corresponding object*. Thus, the overhead for mapping priority queue objects to heap indices is $O(1)$ *per access/update*.

PQ operations: Pseudocode

```
MAX-HEAP-MAXIMUM(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  return A[1]
```

- Time complexity. The above procedure implements the MAXIMUM operation in $\Theta(1)$ time.

PQ operations: Pseudocode

```
MAX-HEAP-EXTRACT-MAX(A)
1  max = MAX-HEAP-MAXIMUM(A)
2  A[1] = A[A.heap-size]
3  A.heap-size = A.heap-size − 1
4  MAX-HEAPIFY(A, 1)
5  return max
```

Copies $A[n]$ into $A[1]$, decreases the heap-size, and then calls "heapify" on the root.

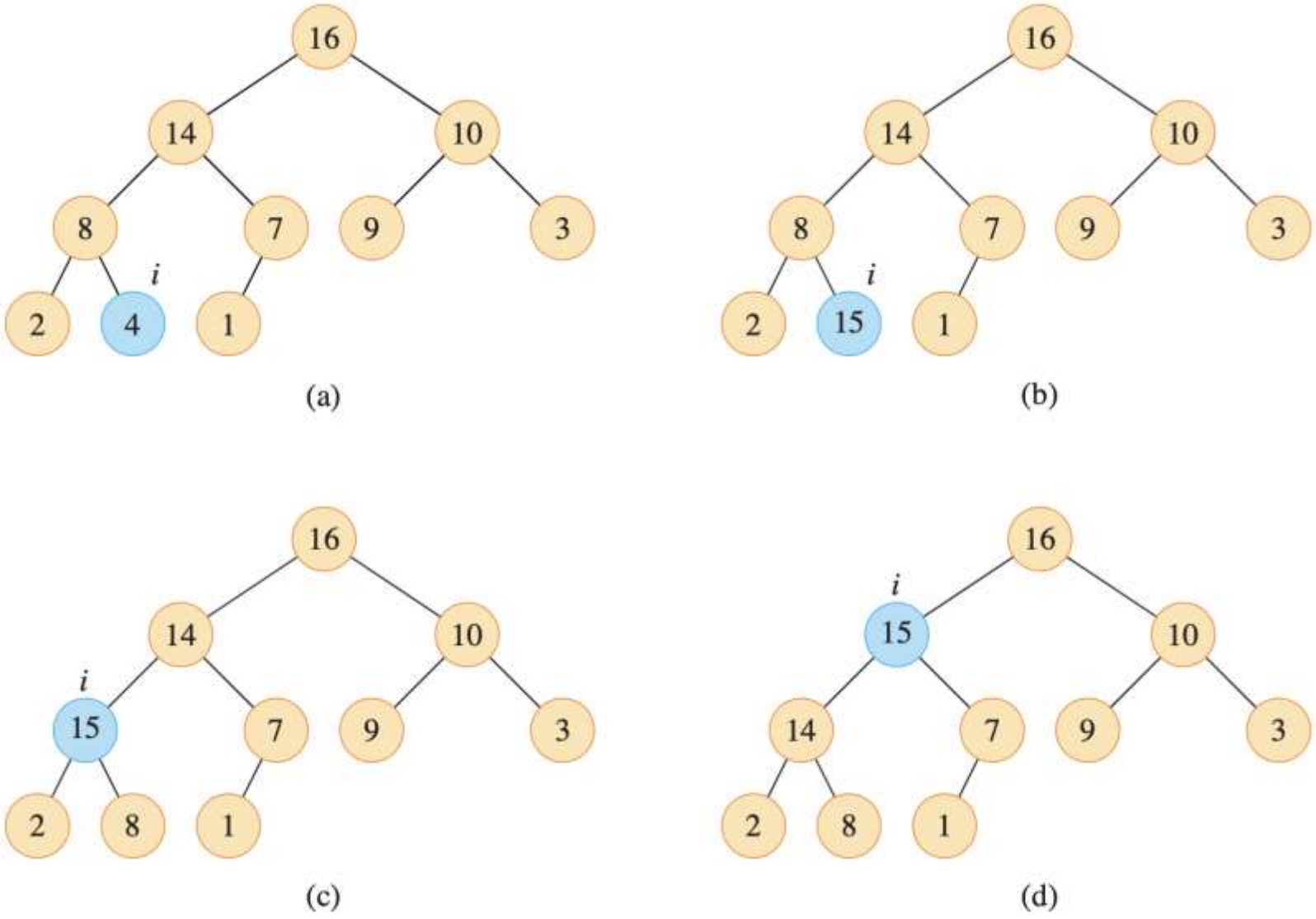
- Time complexity. The above procedure implements the EXTRACT-MAX operation in $\Theta(\log n)$ time.
- We implicitly assume that MAX-HEAPIFY compares priority queue objects *based on their key attributes*.

PQ operations: Pseudocode

```
MAX-HEAP-INCREASE-KEY(A, x, k)
1  if k < x.key
2       error "new key is smaller than current key"
3  x.key = k
4  find the index i in array A where object x occurs
5  while i > 1 and A[PARENT(i)].key < A[i].key
6       exchange A[i] with A[PARENT(i)], updating the information that maps
            priority queue objects to array indices
7       i = PARENT(i)
```

- Correctness. As increasing the key of $A[i]$ might violate the max-heap property, *Steps 5-7 traverses a simple path from the node i toward the root* to find a proper place for the newly increased key.

INCREASE-KEY: Example



PQ operations: Pseudocode

```
MAX-HEAP-INCREASE-KEY(A, x, k)
1  if k < x.key
2       error "new key is smaller than current key"
3  x.key = k
4  find the index i in array A where object x occurs
5  while i > 1 and A[PARENT(i)].key < A[i].key
6       exchange A[i] with A[PARENT(i)], updating the information that maps
            priority queue objects to array indices
7       i = PARENT(i)
```

- Time complexity. The above procedure implements the INCREASE-KEY operation in time $O(\log n)$ since the path from node i to the root has length at most $O(\log n)$.

PQ operations: Pseudocode

```
MAX-HEAP-INSERT(A, x, n)
1  if A.heap-size == n
2        error "heap overflow"
3  A.heap-size = A.heap-size + 1
4  k = x.key
5  x.key = −∞
6  A[A.heap-size] = x
7  map x to index heap-size in the array
8  MAX-HEAP-INCREASE-KEY(A, x, k)
```

Lowers the value of x to the smallest possible to ensure that the max-heap property is maintained when x is made the last element of the array

- Time complexity. The above procedure implements the INSERT operation in $O(\log n)$ time.

Practice problems

Practice Problem 1

Show that each child of the root of an n-node heap is the root of a subtree containing at most $2n/3$ nodes. What is the smallest constant α such that each subtree has at most αn nodes? How does that affect the recurrence (6.1) and its solution?

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 simple path from the root down to a leaf.)

Practice Problem 2

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

Practice Problem 3

Write pseudocode to implement a min-priority queue with a min-heap by writing the procedures MIN-HEAP-MINIMUM, MIN-HEAP-EXTRACT-MIN, MIN-HEAP-DECREASE-KEY, and MIN-HEAP-INSERT.

Practice Problem 4

Write pseudocode for the procedure MAX-HEAP-DECREASE-KEY(A, x, k) in a max-heap. What is the running time of your procedure?

Practice Problem 5

The operation MAX-HEAP-DELETE(A, x) deletes the object x from max-heap A. Give an implementation of MAX-HEAP-DELETE for an n-element max-heap that runs in $O(\lg n)$ time plus the overhead for mapping priority queue objects to array indices.

Practice Problem 6

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.)