CS430-01
Introduction to Algorithms
4. Heap Sort

Michael Choi
Dept. of Computer Science
IIT

Sorting

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

The input sequence is usually an n-element array, although it may be represented in some other fashion, such as a linked list.

- Value is usually part of a collection of data called a **record**
- Each record contains a **key**, which is the value to be sorted
- The remainder of the record consists of **satellite data**, which are usually carried around with the key
- When a sorting algorithm permutes the keys, it must permute the satellite data as well

Why Sorting?

- Sometimes an application inherently needs to sort information. For example, in order to prepare customer statements, banks need to sort checks by check number.
- Algorithms often use sorting as a key subroutine. For example, a program that renders graphical objects which are layered on top of each other might have to sort the objects according to an "above" relation so that it can draw these objects from bottom to top. We shall see numerous algorithms in this text that use sorting as a subroutine.
- We can draw from among a wide variety of sorting algorithms, and they employ a rich set of techniques. In fact, many important techniques used throughout algorithm design appear in the body of sorting algorithms that have been developed over the years. In this way, sorting is also a problem of historical interest.

Why Sorting?

- We can prove a nontrivial lower bound for sorting (as we shall do in Chapter 8). Our best upper bounds match the lower bound asymptotically, and so we know that our sorting algorithms are asymptotically optimal. Moreover, we can use the lower bound for sorting to prove lower bounds for certain other problems.

- Many engineering issues come to the fore when implementing sorting algorithms. The fastest sorting program for a particular situation may depend on many factors, such as prior knowledge about the keys and satellite data, the memory hierarchy (caches and virtual memory) of the host computer, and the software environment. Many of these issues are best dealt with at the algorithmic level, rather than by "tweaking" the code.

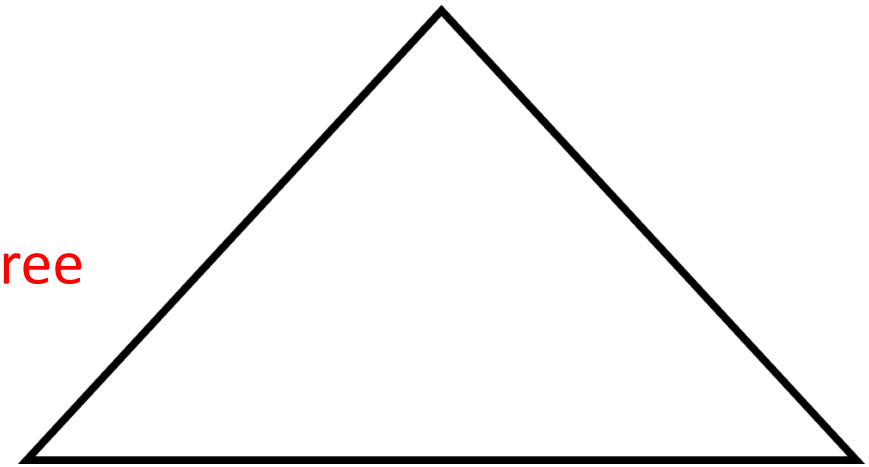
Heapsort

- (binary) heap data structure is an array object, as a nearly complete binary tree
- Trees: free tree vs. rooted/ordered tree vs. binary tree (in Appendix B.5)
- Heap structure requires complete BT

Definitions of Binary Trees

- **Full Binary Tree:** A binary tree in which all of the leaves are on the same level and every nonleaf node has two children

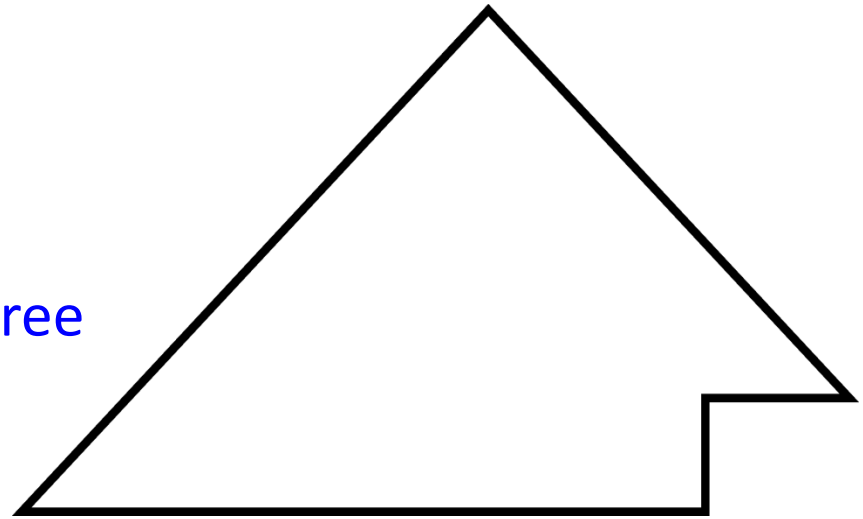
Full binary tree →
Or complete binary tree



Definitions of Binary Trees *(Cont'd)*

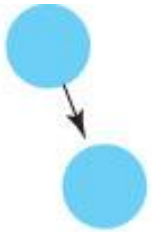
- **Complete Binary Tree:** A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible

Also calls near →
Complete binary tree

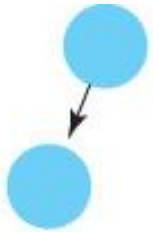


Examples of Different Types of Binary Trees

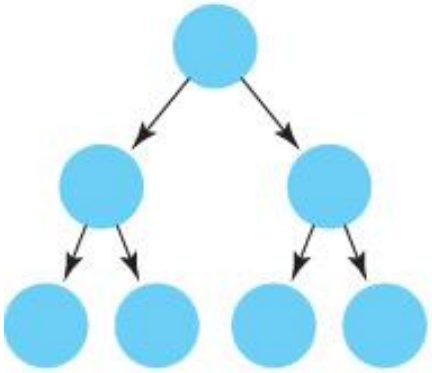
(a) Full and complete



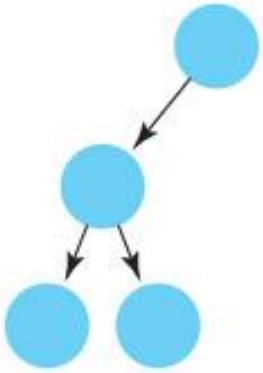
(b) Neither full nor complete



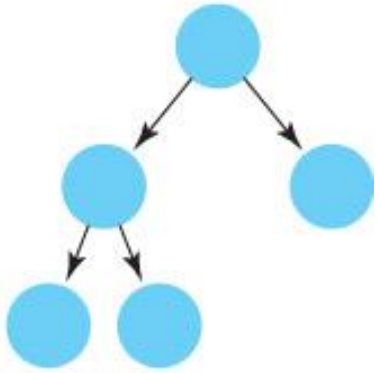
(c) Complete



(d) Full and complete



(e) Neither full nor complete



(f) Complete

Binary heap

A binary tree where the value of a parent is greater than or equal to the value of its children

Additional restriction: all levels of the tree are **complete** except the last level (or complete BT shape)

To be a heap structure, **complete BT + value properties**

Max heap vs. min heap

heap

Max heap

Parent node value must be bigger than children nodes (or subtrees)

- biggest value node? **Root node**

Typically use maxheap for heap sort

min heap $A[\text{PARENT}(i)] \leq A[i]$

- Root node value? Smallest

- the *shape property*: the tree must be a complete binary tree
- the *order property:* for every node in the tree, the value stored in that node is greater than or equal to the value in each of its children.

Binary heap - operations

Maximum(S) - return the largest element in the set

ExtractMax(S) – Return and remove the largest element in the set

Insert(S, val) – insert val into the set

IncreaseElement(S, x, val) – increase the value of element x to val

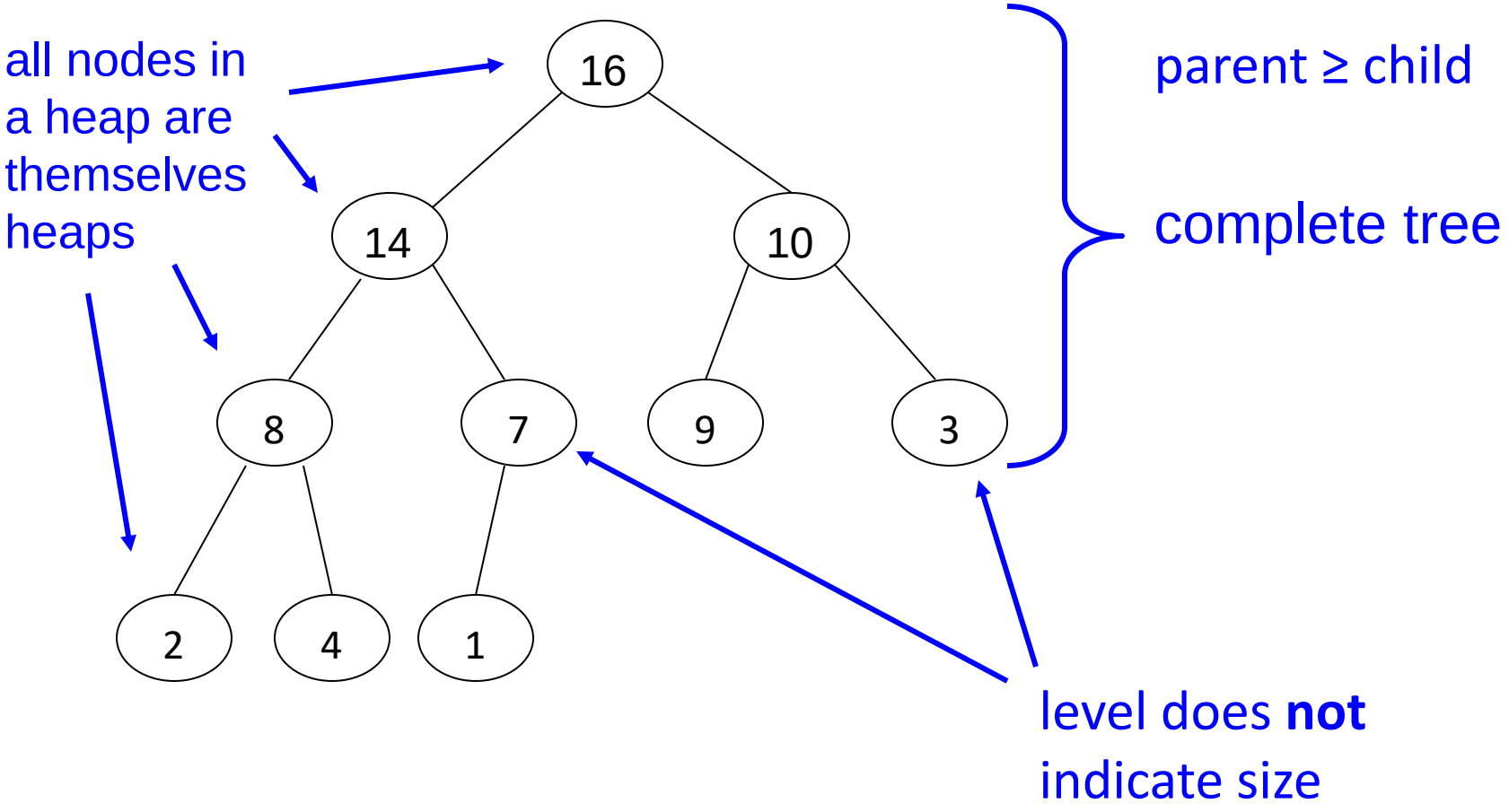
BuildHeap(A) – build a heap from an array of elements

Binary heap

How can we represent a heap?

Binary heap - references

Lnked list implementation



Heap structure implementations

Conceptual heap diagram

Array Implentation

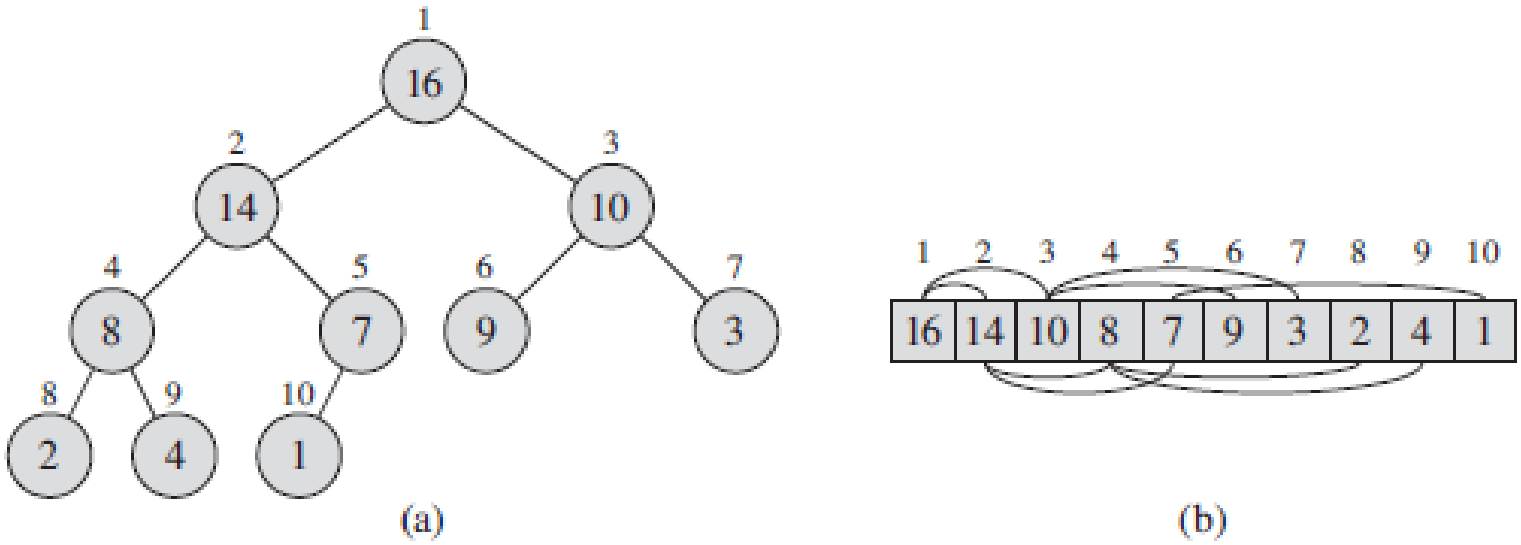
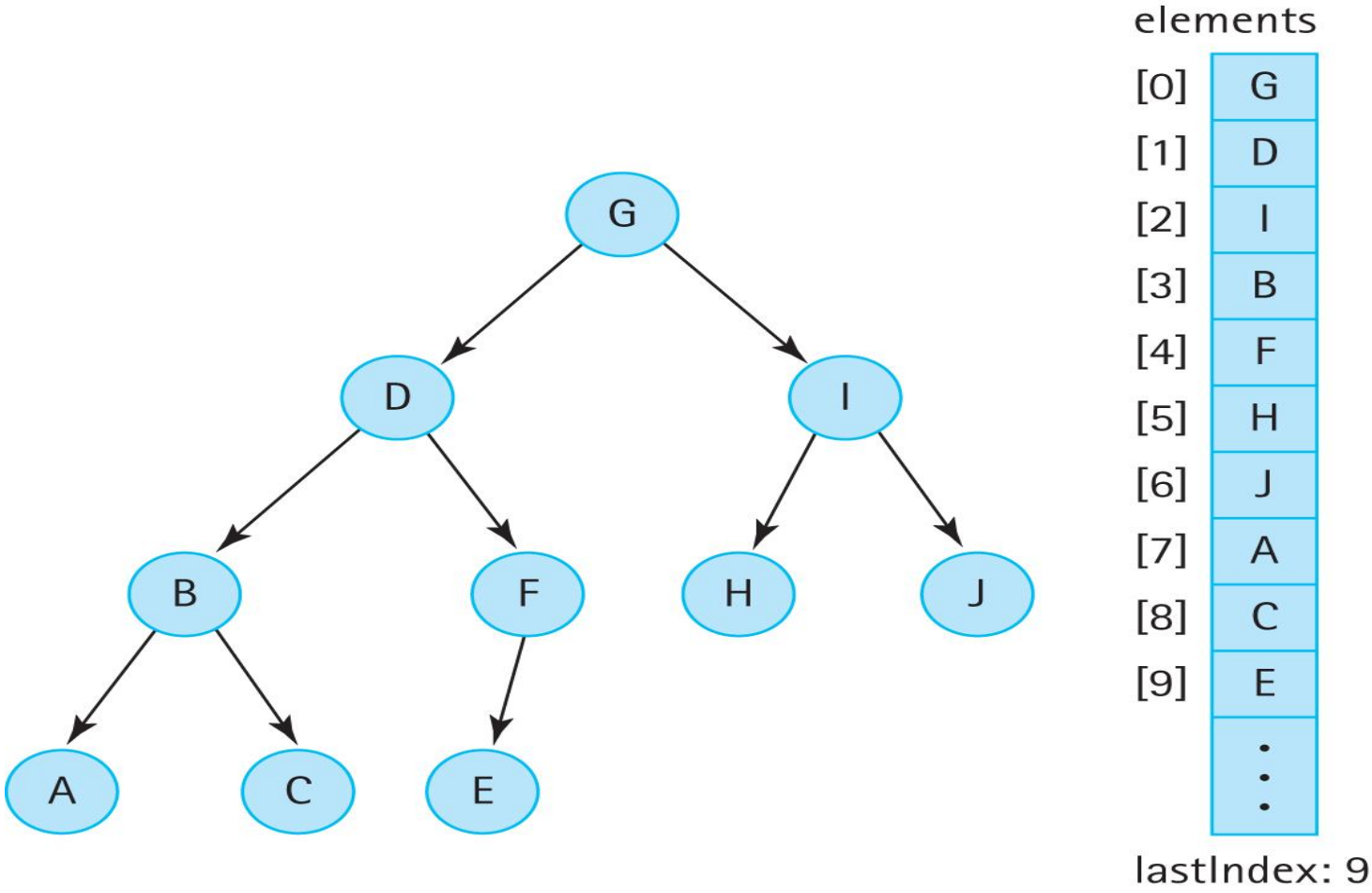


Figure 6.1 A max-heap viewed as (**a**) a binary tree and (**b**) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; 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.

Binary heap - array

```
PARENT(i)
    return ⌊i/2⌋

LEFT(i)
    return 2i

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

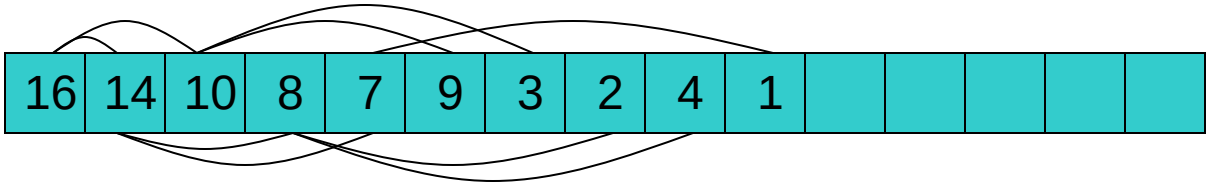
A Binary Tree and Its Array Representation



Array Representation continued

- To implement the algorithms that manipulate the tree, we must be able to find the left and right child of a node in the tree:
  - elements[index] left child is in elements[index*2 + 1]
  - elements[index] right child is in elements[index*2 + 2]
- We also can determine the location of its parent node:
  - elements[index]'s parent is in elements[(index – 1)/2].
- This representation works best, space wise, if the tree is complete (which it is for a heap)

Binary heap - array



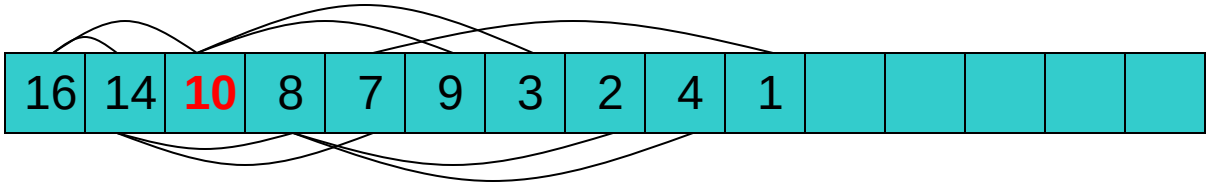
1 2 3 4 5 6 7 8 9 10

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

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

Binary heap - array



1 2 **3** 4 5 6 7 8 9 10

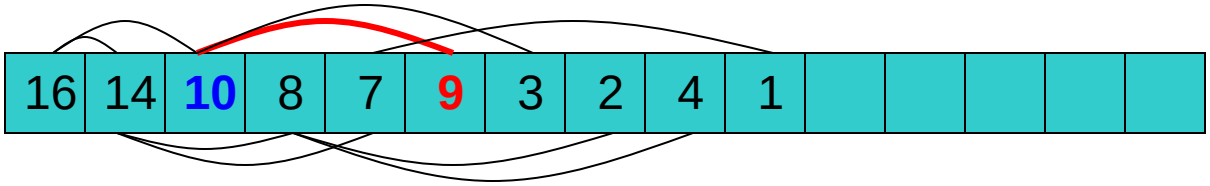
Left child of A[3]?

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

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

Binary heap - array



1 2 3 4 5 6 7 8 9 10

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

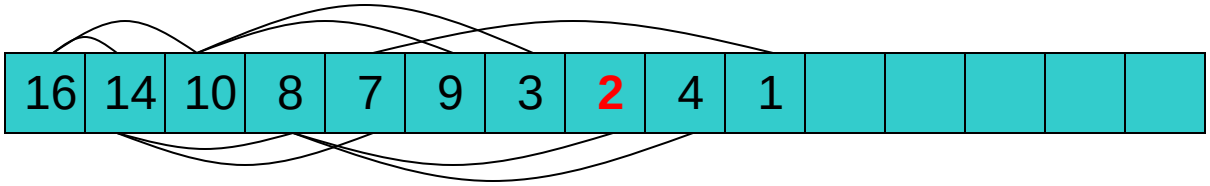
LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

Left child of A[3]?

2*3 = 6

Binary heap - array



1 2 3 4 5 6 7 **8** 9 10

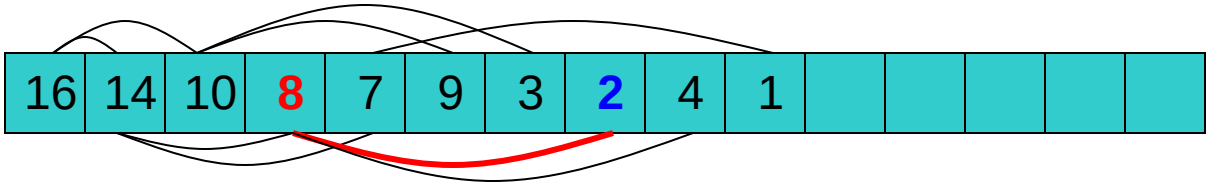
Parent of A[8]?

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

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

Binary heap - array



1 2 3 4 5 6 7 8 9 10

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

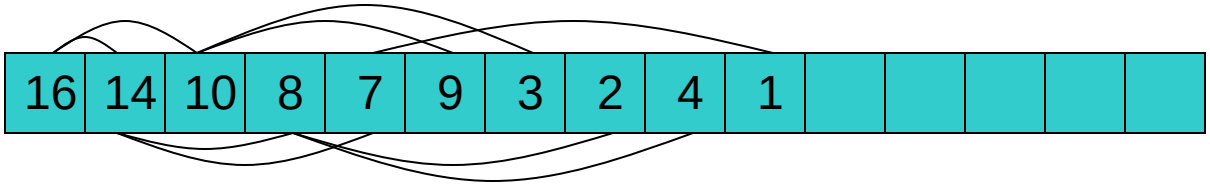
LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

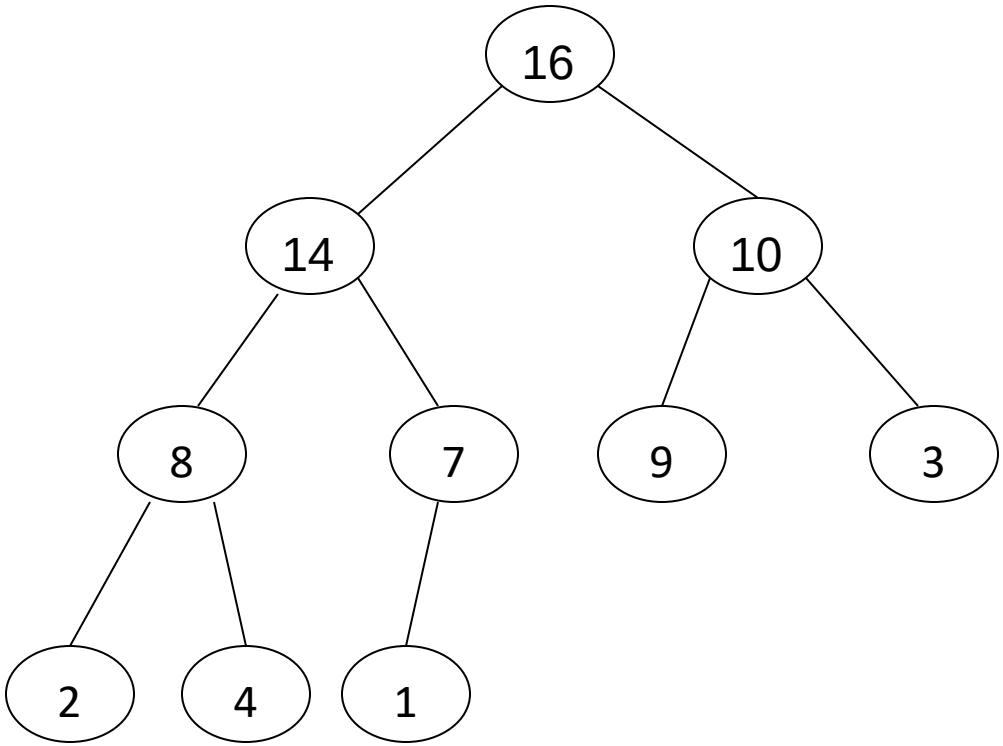
Parent of A[8]?

$$\lfloor 8/2 \rfloor = 4$$

Binary heap - array



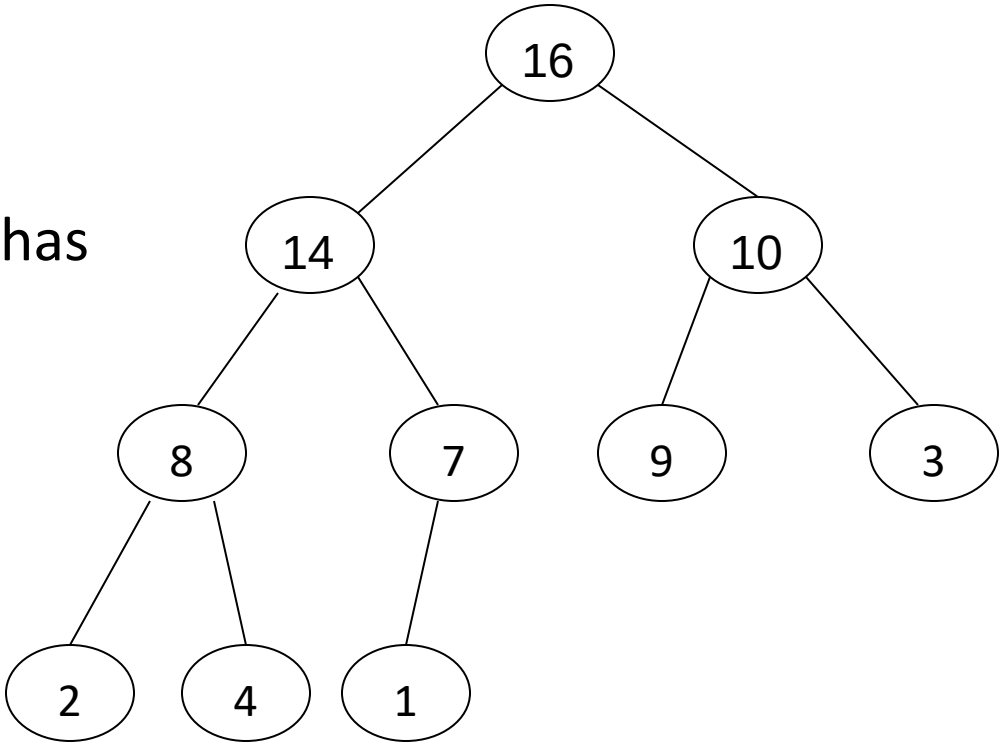
1 2 3 4 5 6 7 8 9 10



Height of a heap

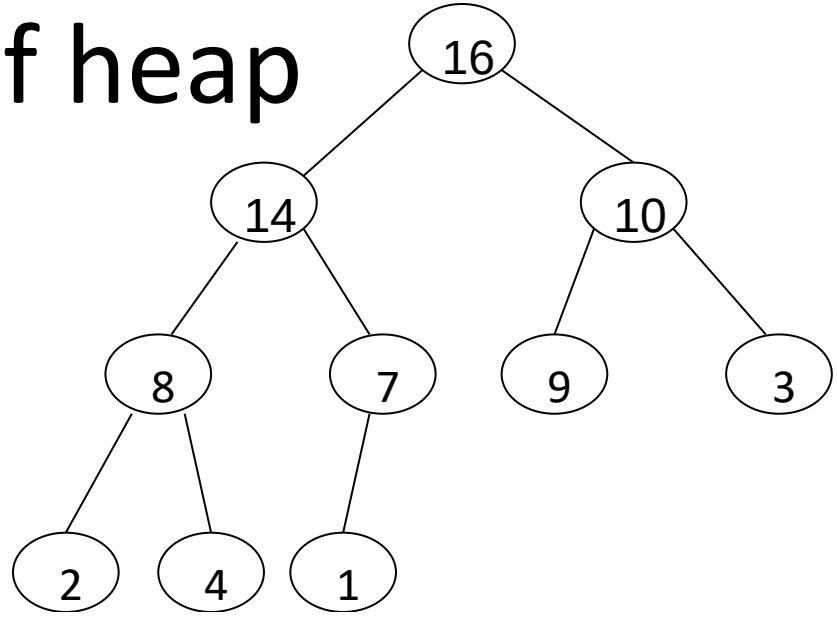
Height of a heap = ϴ(lg n)

Show an n-element heap has Height $\lfloor \lg n \rfloor$



Write $n = 2^m - 1 + k$ where m is as large as possible. Then the heap consists of a complete binary tree of height $m - 1$, along with k additional leaves along the bottom. The height of the root is the length of the longest simple path to one of these k leaves, which must have length m. It is clear from the way we defined m that $m = \lfloor \lg n \rfloor$.

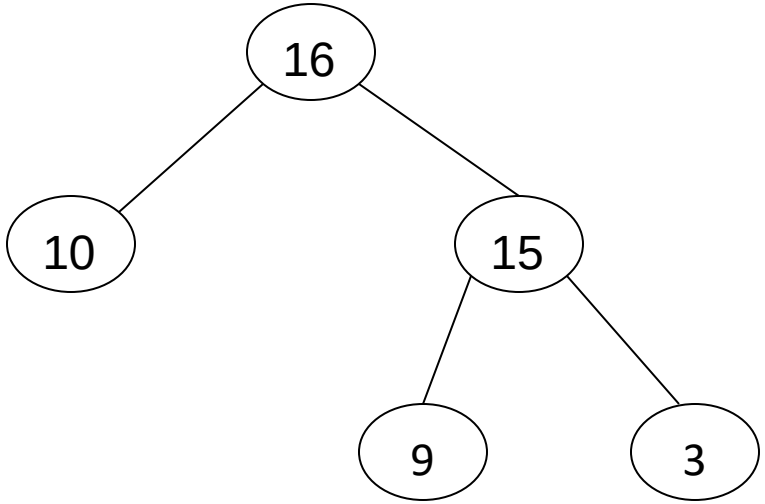
Operations of heap



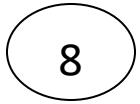
- The MAX-HEAPIFY procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Identify the valid heaps

[15, 12, 3, 11, 10, 2, 1, 7, 8]



[20, 18, 10, 17, 16, 15, 9, 14, 13]



Maintaining heap property

```
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)
```

Maintaining heap property

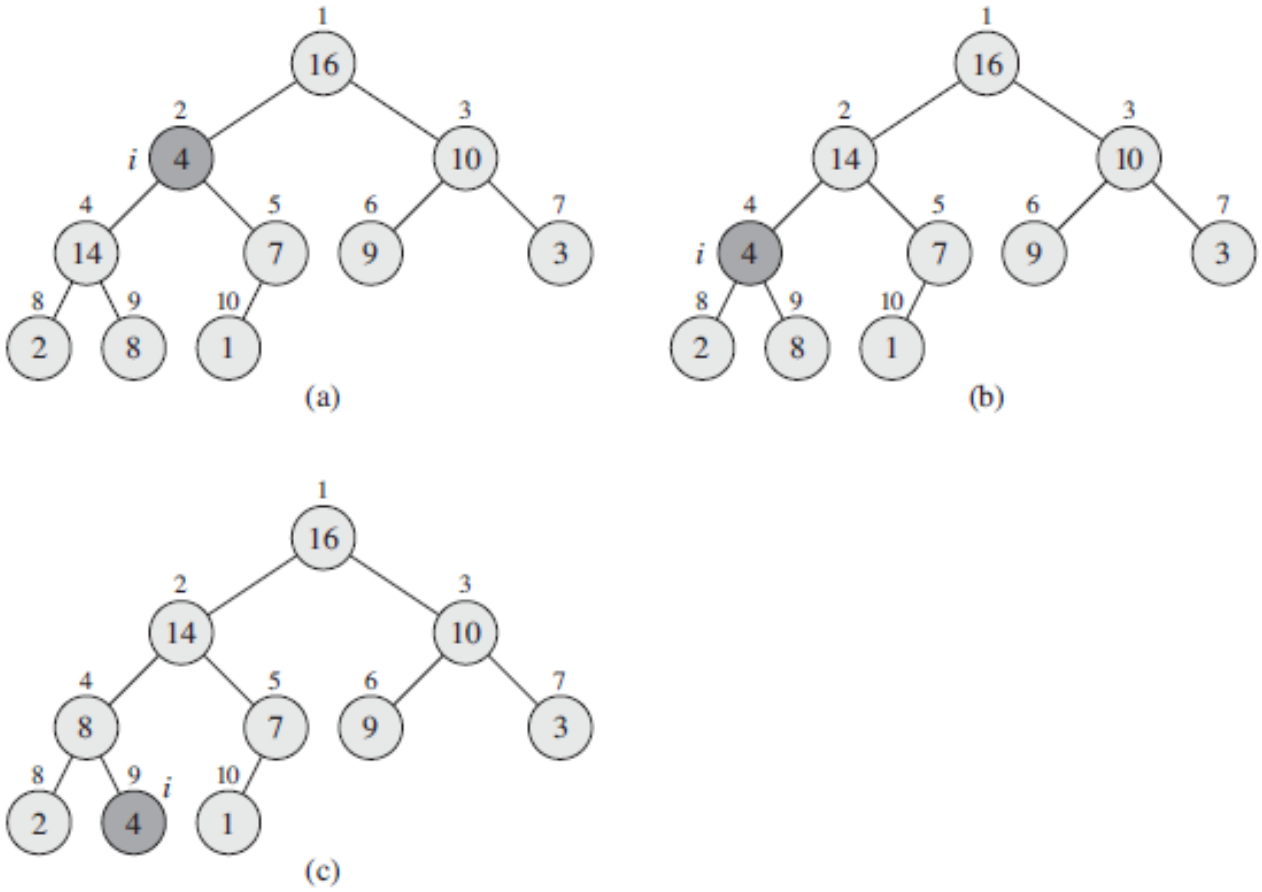
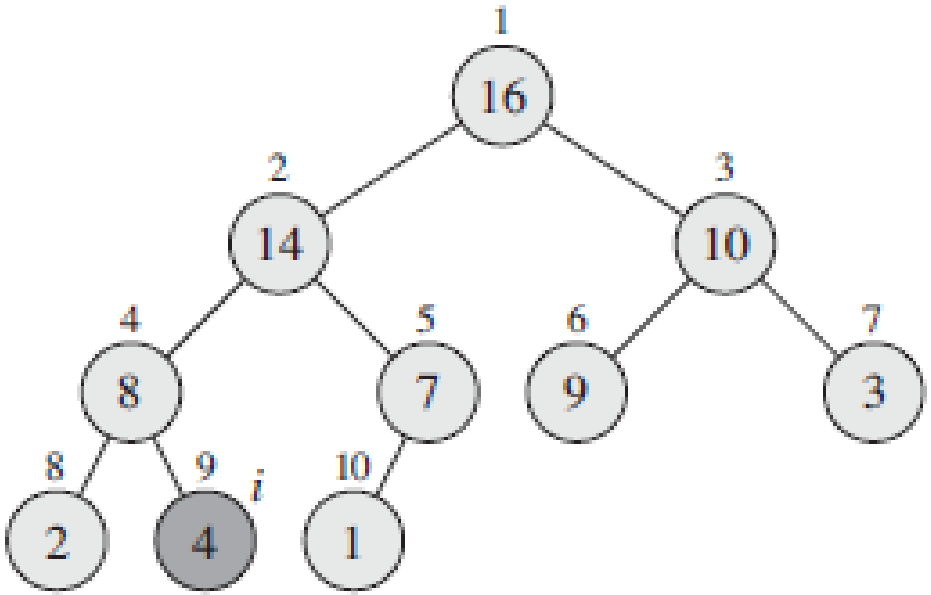


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, 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.

Maintaining heap property



The children's subtrees each have size at most $2n/3$—the worst case occurs when the bottom level of the tree is exactly half full—and therefore we can describe the running time of MAX-HEAPIFY by the recurrence

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

Building a Heap

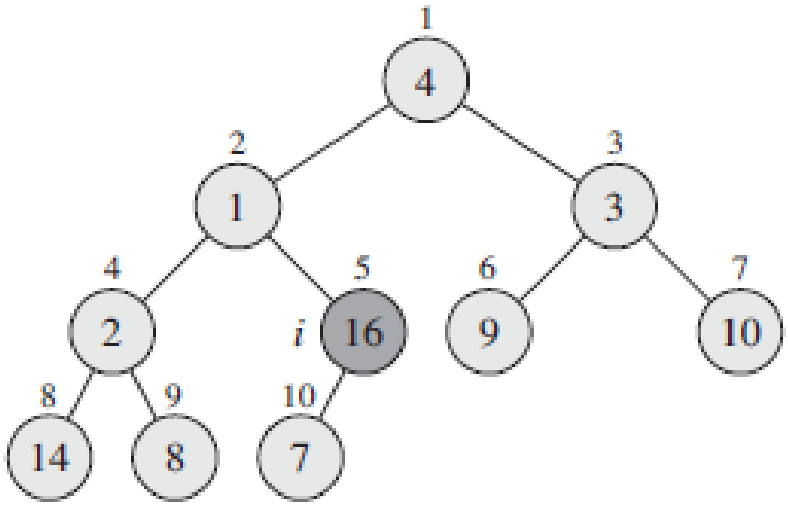
```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.

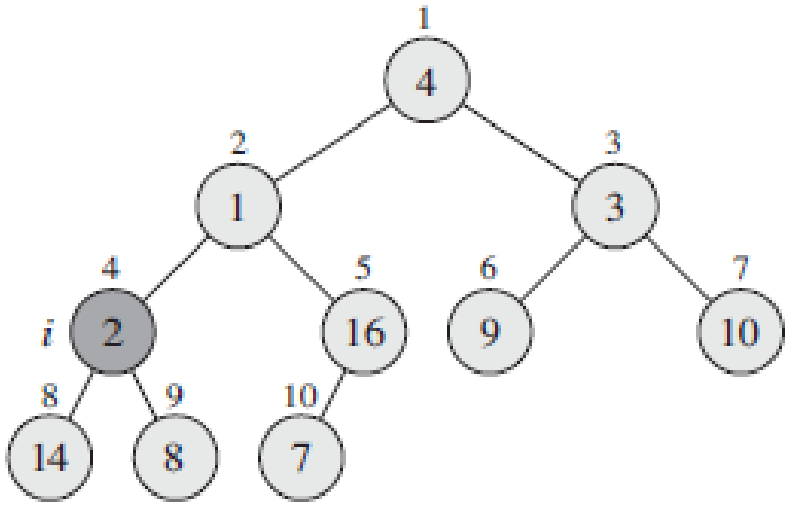
Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \ldots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.

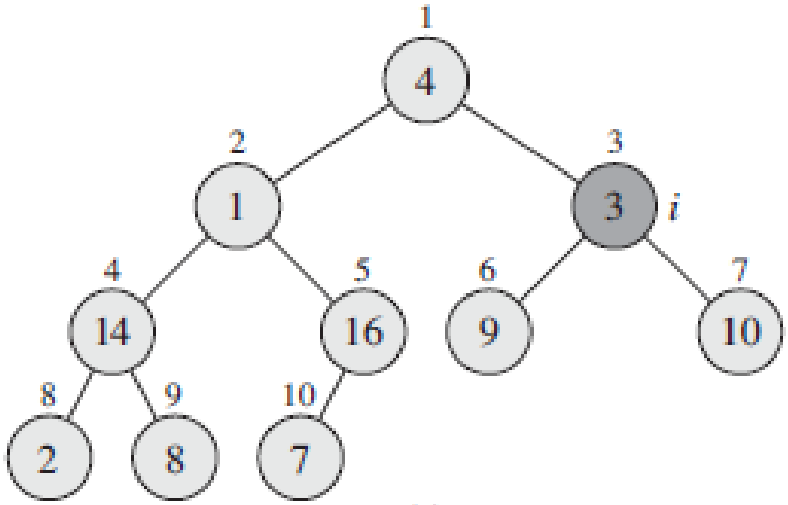
A	4	1	3	2	16	9	10	14	8	7



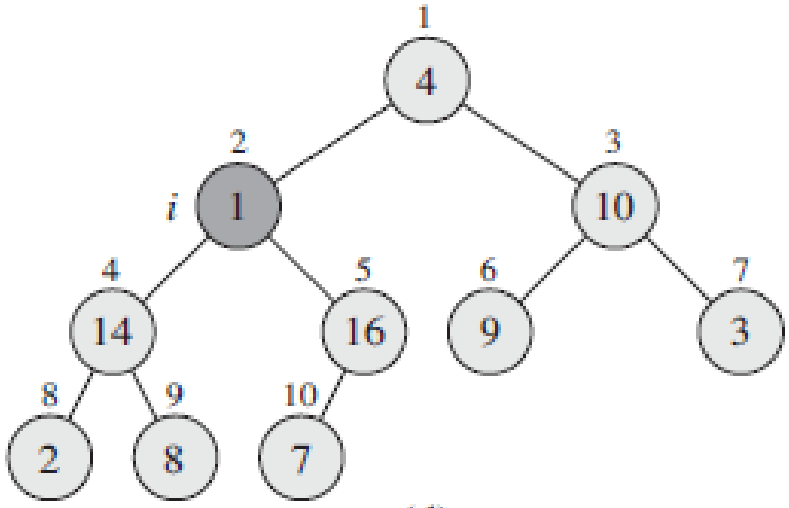
(a)



(b)



(c)



(d)

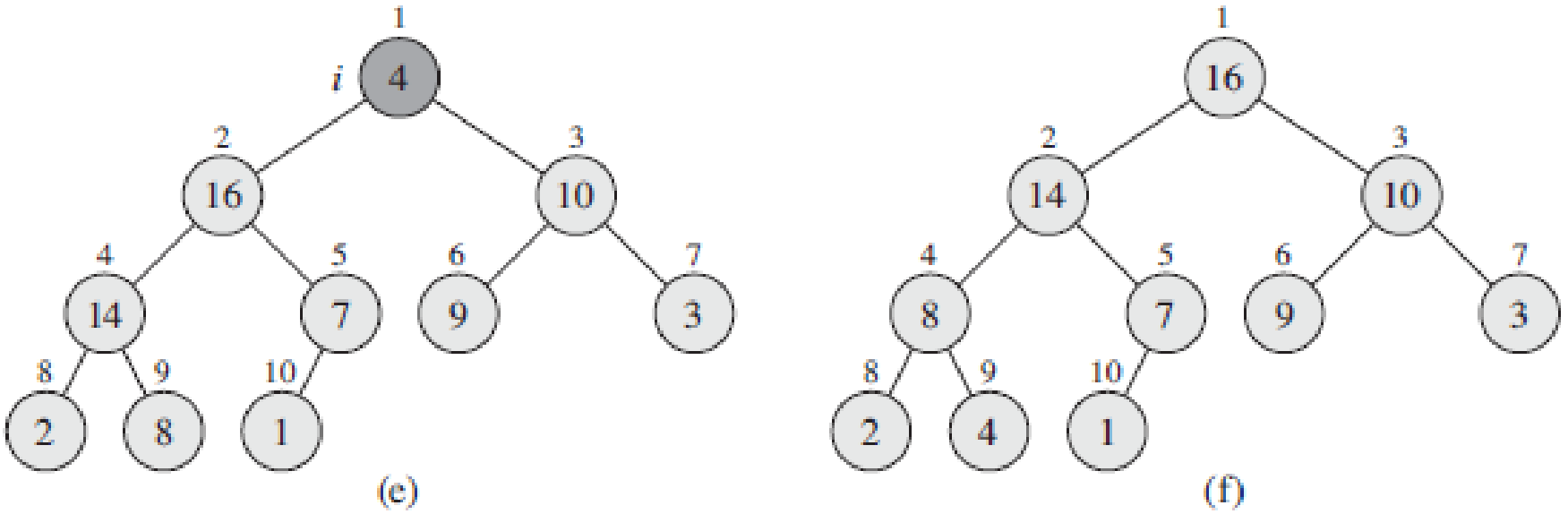
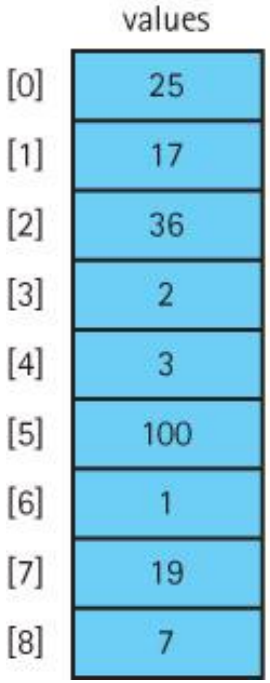
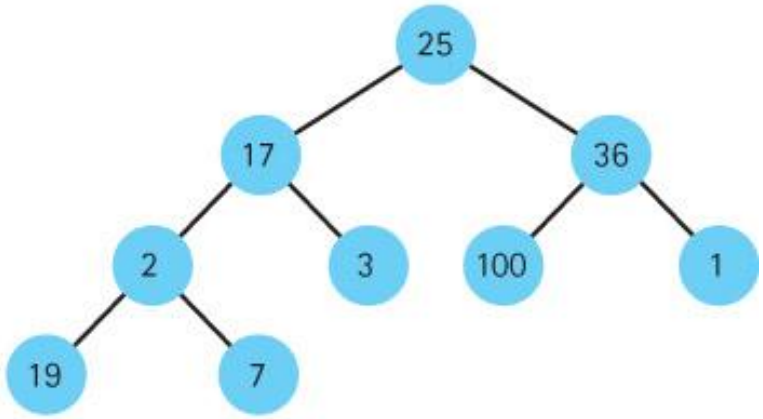


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. (a) A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). (b) The data structure that results. The loop index i for the next iteration refers to node 4. (c)–(e) Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. (f) The max-heap after BUILD-MAX-HEAP finishes.

Building a heap



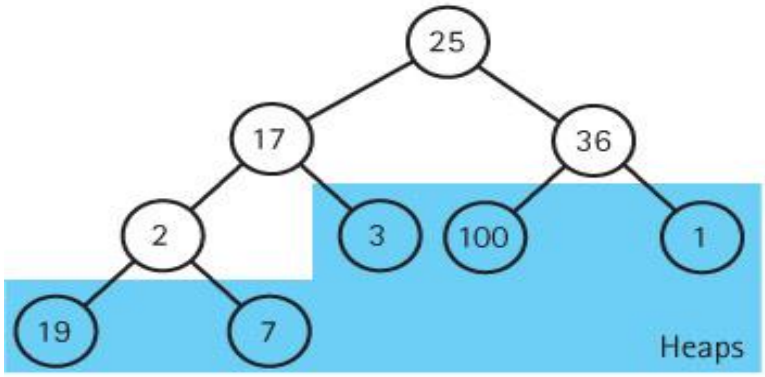


buildHeap

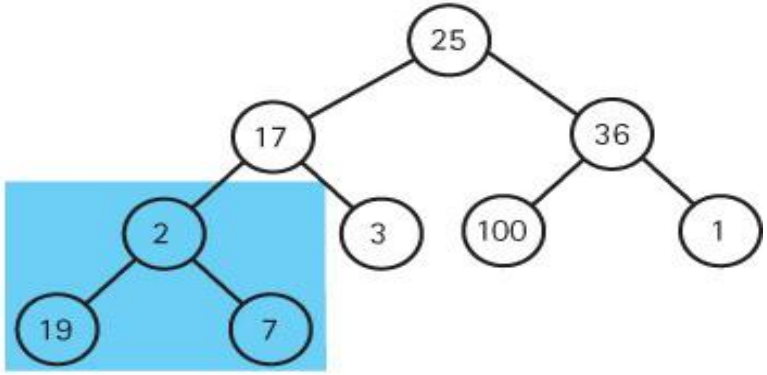
for index going from first nonleaf node up to the root node
    reheapDown(values[index], index)

See next slide …

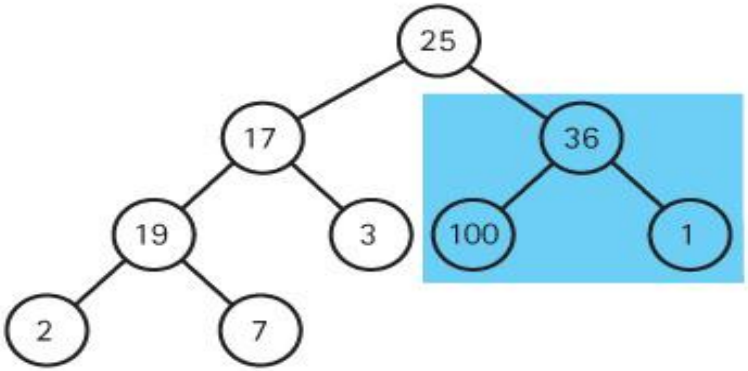
(a)



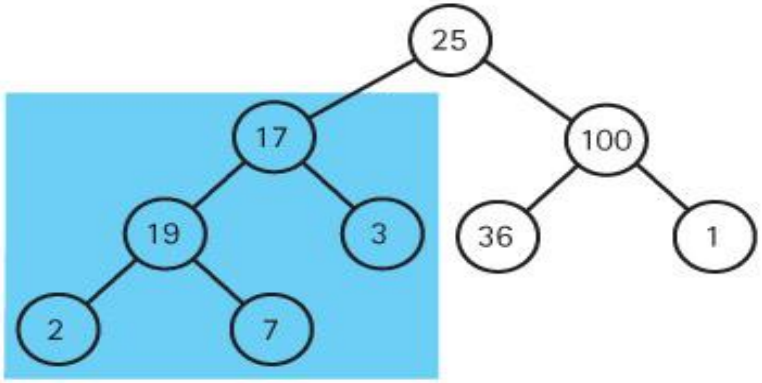
(b)



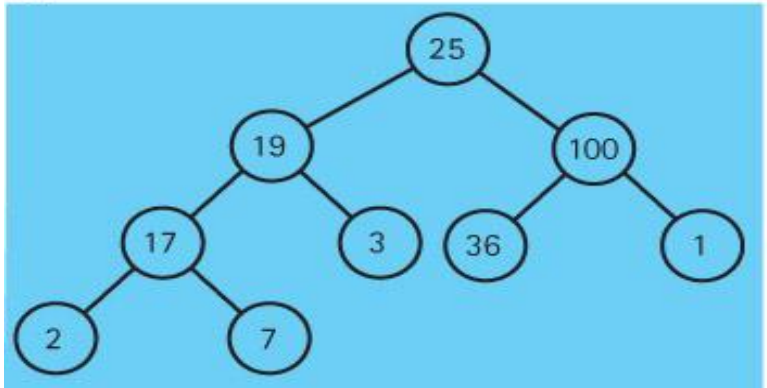
(c)



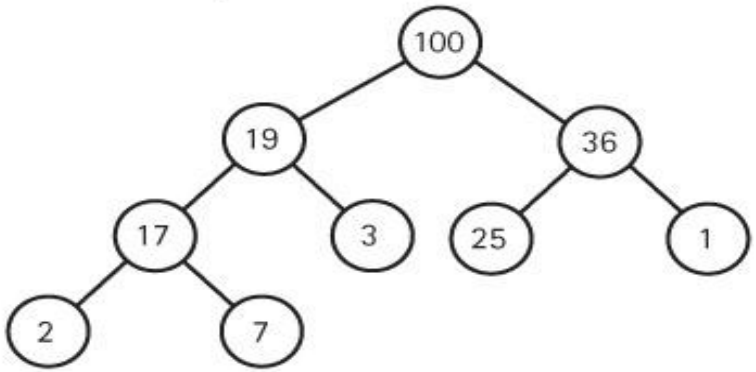
(d)



(e)



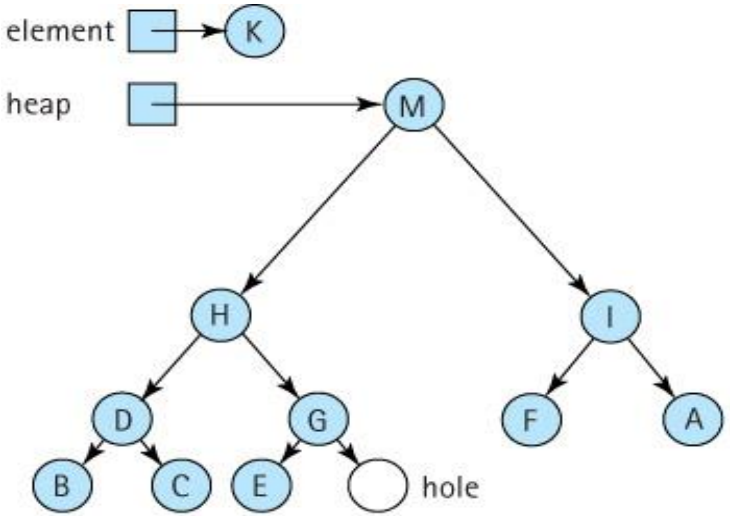
(f) Tree now represents a heap



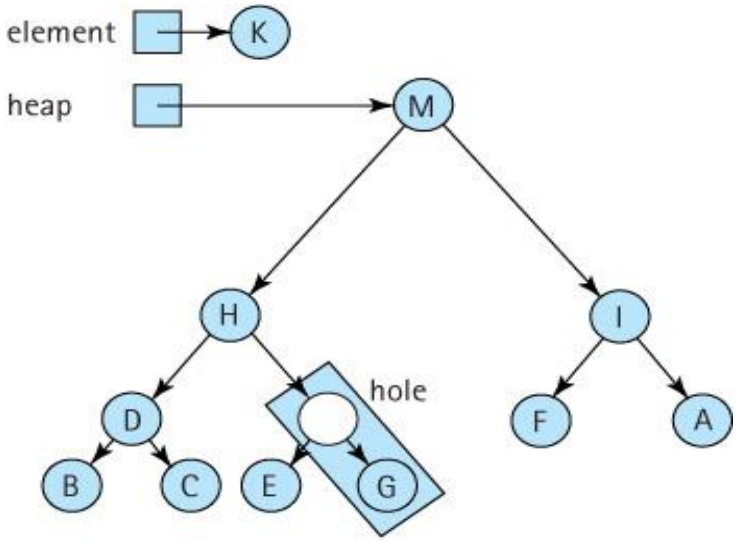
The changing contents of the array

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Original values	25	17	36	2	3	100	1	19	7
After reheapDown index = 3	25	17	36	19	3	100	1	2	7
After index = 2	25	17	100	19	3	36	1	2	7
After index = 1	25	19	100	17	3	36	1	2	7
After index = 0	100	19	36	17	3	25	1	2	7

Tree is a heap.

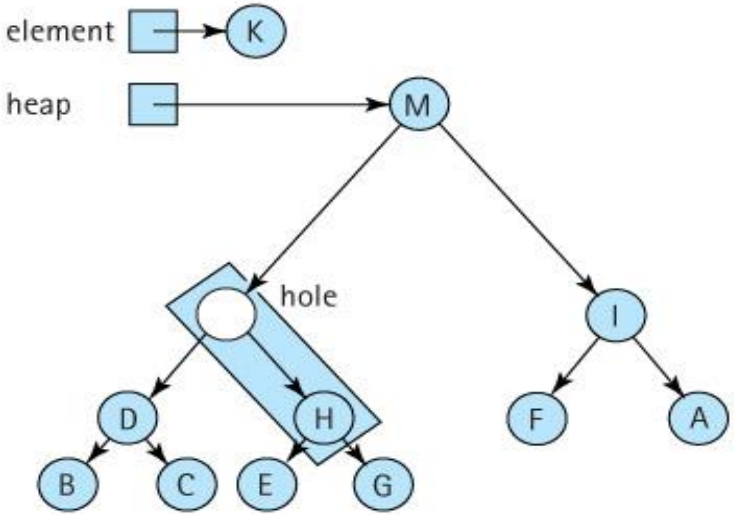


(a) Add K

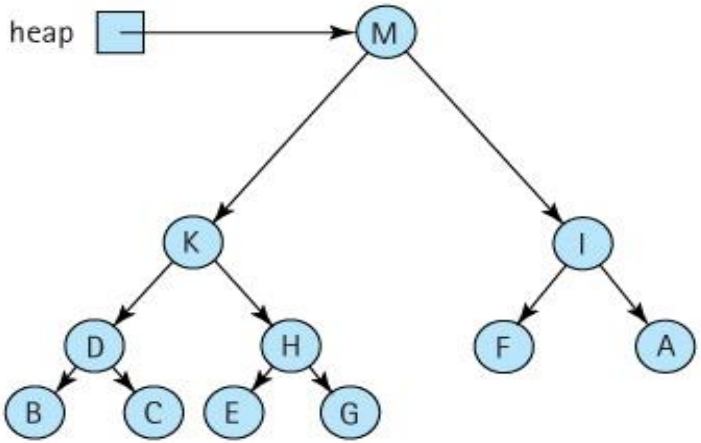


(b) Move hole up

Insert a new node to the heap



(c) Move hole up



(d) Place element into hole

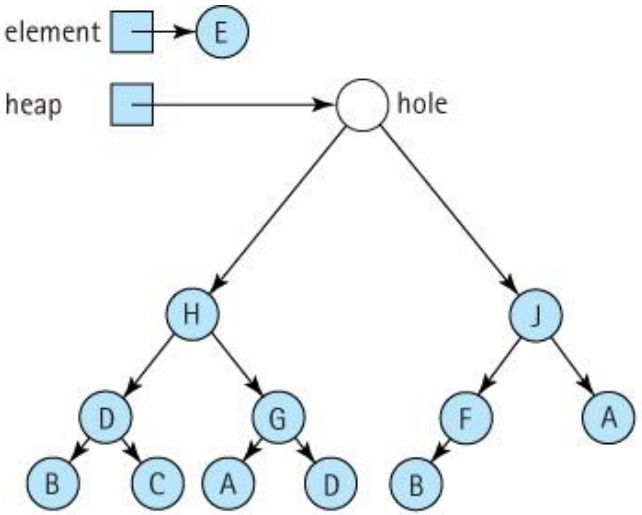
`reheapUp` operation

```
private void reheapUp(T element)
// Current lastIndex position is empty.
// Inserts element into the tree and ensures shape and order properties.
{
  int hole = lastIndex;
  while ((hole > 0)   // hole is not root and element > hole's parent
         &&
       (comp.compare(element, elements.get((hole - 1) / 2)) > 0))
  {
    elements.set(hole,elements.get((hole - 1) / 2));   // move hole's parent down
    hole = (hole - 1) / 2;                             // move hole up
  }
  elements.set(hole, element);                    // place element into final hole
}
```

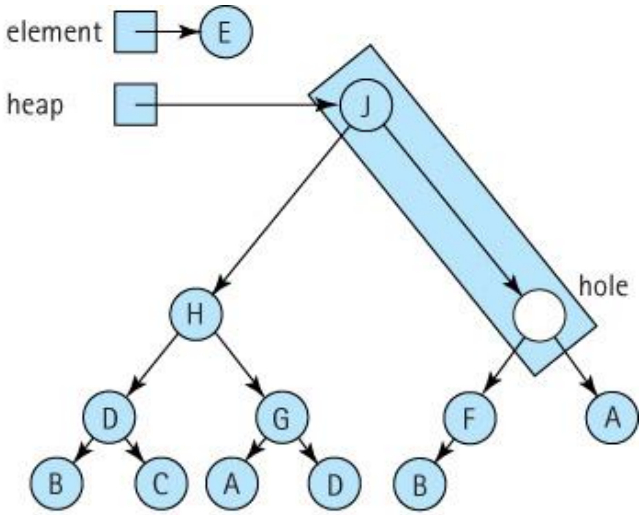
The `remove (dequeue)` method

```
public T dequeue() throws PriQUnderflowException
// Throws PriQUnderflowException if this priority queue is empty;
// otherwise, removes element with highest priority from this
// priority queue and returns it.
{
  T hold;      // element to be dequeued and returned
  T toMove;    // element to move down heap

  if (lastIndex == -1)
    throw new PriQUnderflowException("Priority queue is empty");
  else
  {
    hold = elements.get(0);               // remember element to be returned
    toMove = elements.remove(lastIndex); // element to reheap down
    lastIndex--;                          // decrease priority queue size
    if (lastIndex != -1)
       reheapDown(toMove);                // restore heap properties
    return hold;                          // return largest element
  }
}
```

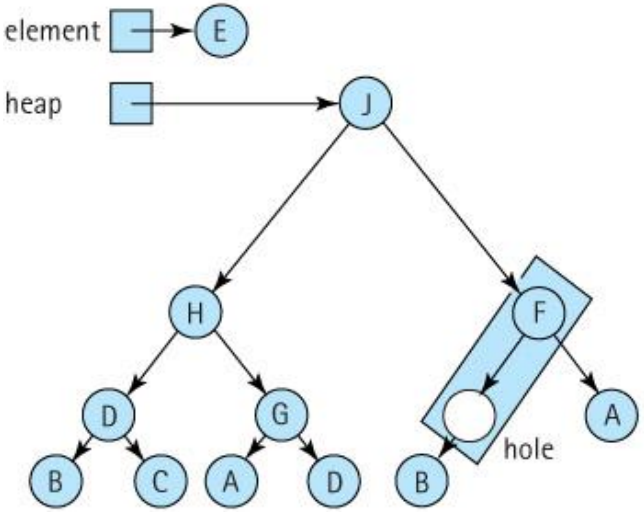
The `reheapDown` algorithm is pictured on the next slide



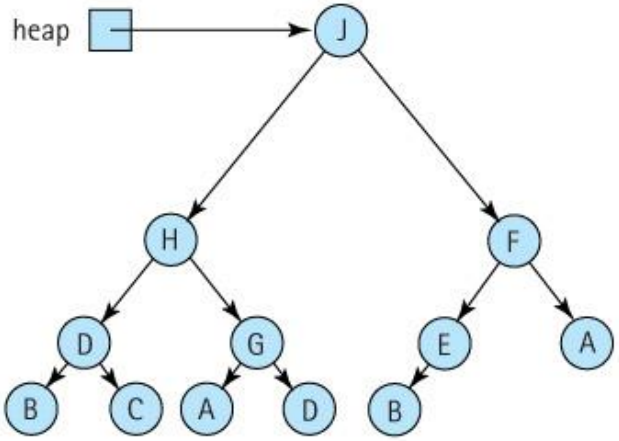
(a) reheapDown (E);



(b) Move hole down



(c) Move hole down



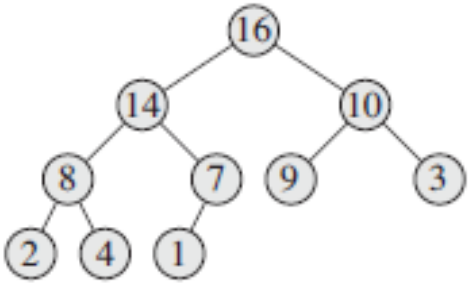
(d) Fill in final hole

`reheapDown` operation

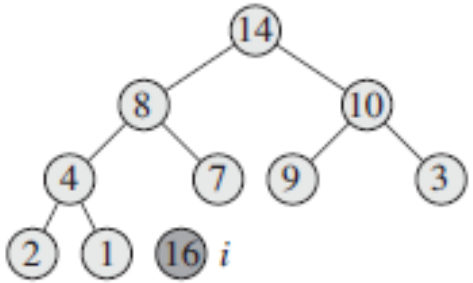
```
private void reheapDown(T element)
// Current root position is "empty";
// Inserts element into the tree and ensures shape and order properties.
{
  int hole = 0;       // current index of hole
  int next;           // next index where hole should move to

  next = newHole(hole, element);   // find next hole
  while (next != hole)
  {
    elements.set(hole,elements.get(next));  // move element up
    hole = next;                            // move hole down
    next = newHole(hole, element);          // find next hole
  }
  elements.set(hole, element);              // fill in the final hole
}
```

Heap sort

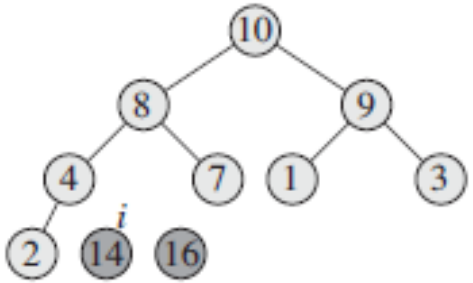
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```


16
14
10
8
7
9
3
2
4
1

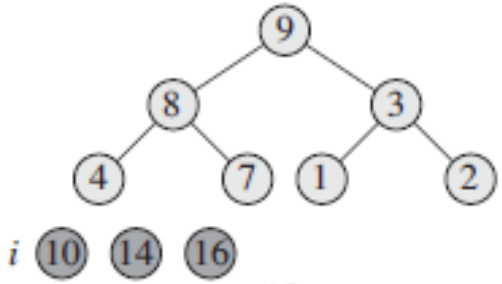
(a)


14
8
10
4
7
9
3
2
1
16
i

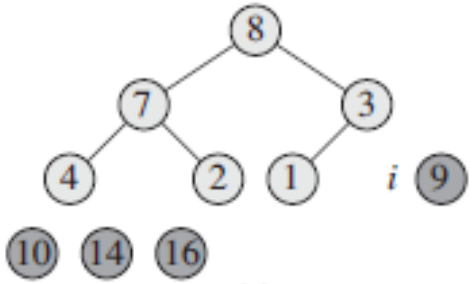
(b)


10
8
9
4
7
1
3
i
2
14
16

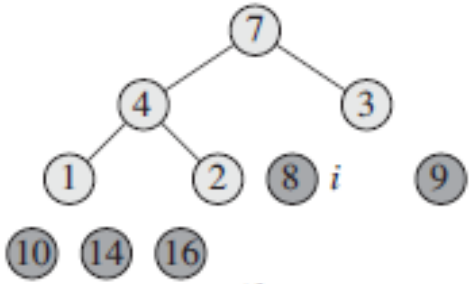
(c)


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)

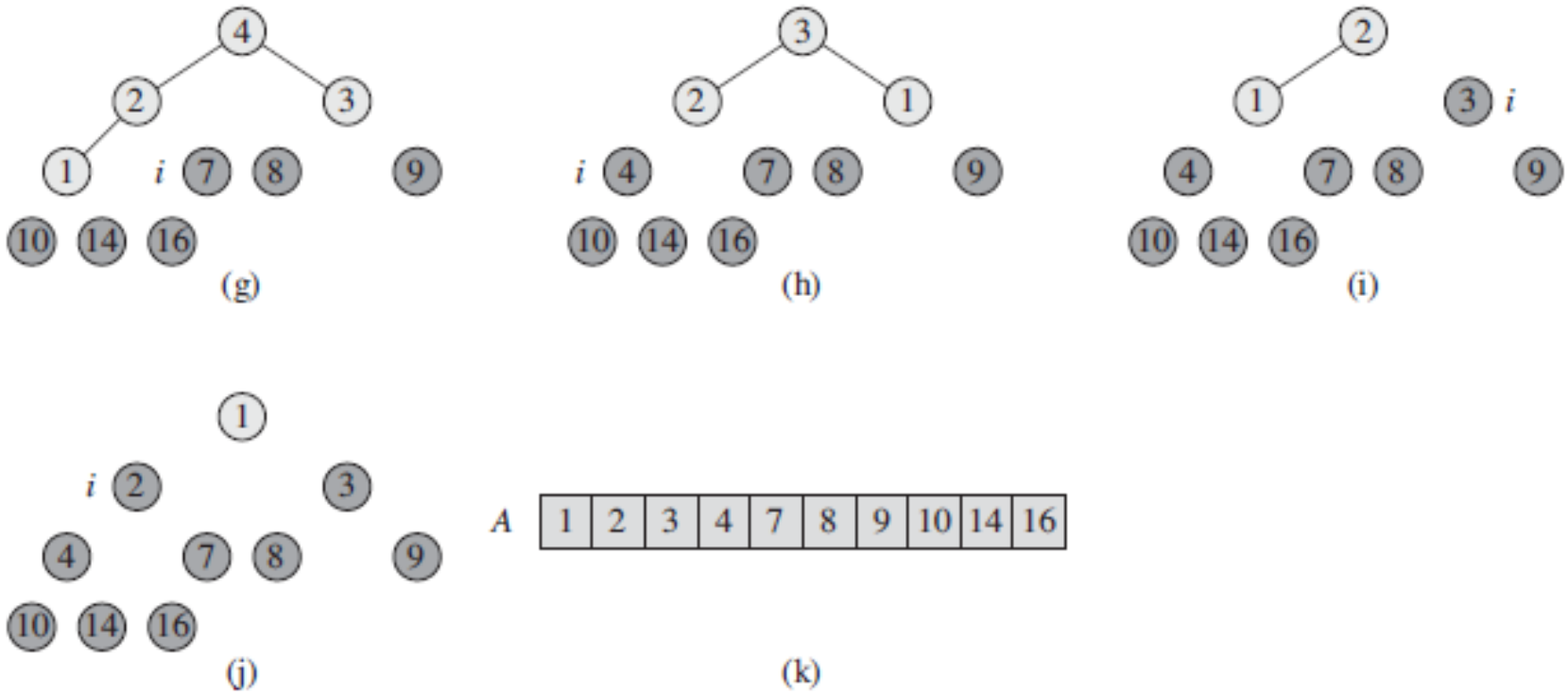


Figure 6.4 The operation of HEAPSORT. (a) The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. (b)–(j) The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. (k) The resulting sorted array A.

The Sort Nodes algorithm

Sort Nodes

for index going from last node up to next-to-root node
    Swap data in root node with values[index]
    reheapDown(values[0], 0, index 2 1)

The `heapSort` method

```
static void heapSort()
// Post: The elements in the array values are sorted by key
{
  int index;
  // Convert the array of values into a heap
  for (index = SIZE/2 - 1; index >= 0; index--)
    reheapDown(values[index], index, SIZE - 1);

  // Sort the array
  for (index = SIZE - 1; index >=1; index--)
  {
    swap(0, index);
    reheapDown(values[0], 0, index - 1);
  }
}
```

Priority Queue

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

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

Priority Queue

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

Priority Queue

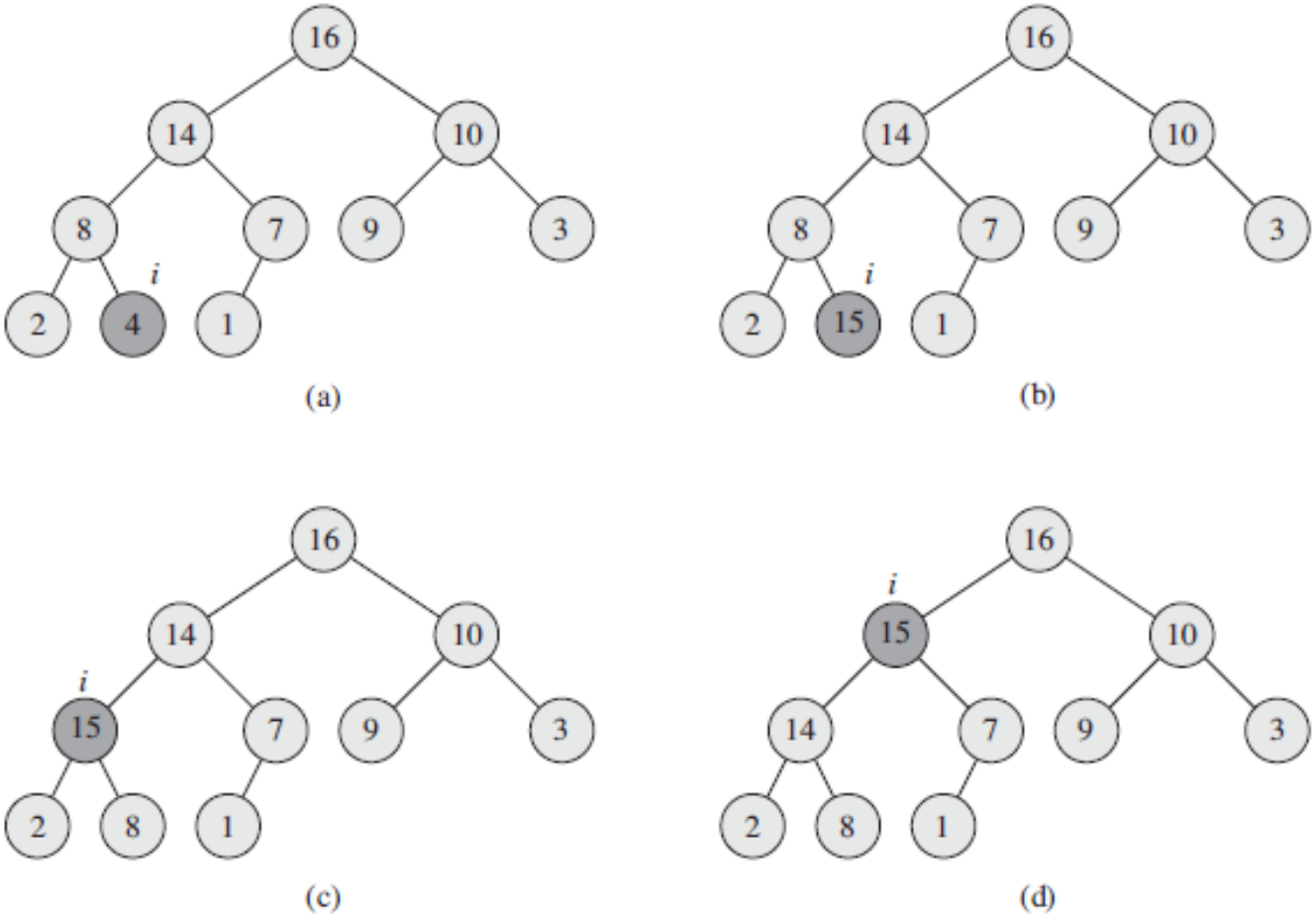


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

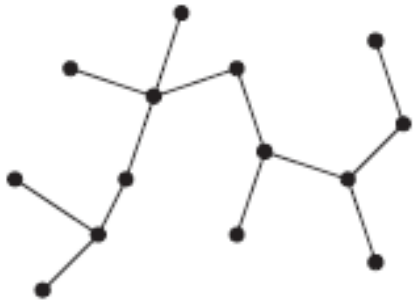
Heap sort steps

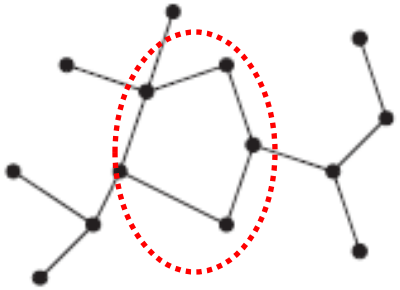
Heap Sort Animatiom

Appendix

Tree

- **Free Tree**: a connected, undirected graph





Graph, not tree

Tree

- **Rooted and ordered tree**

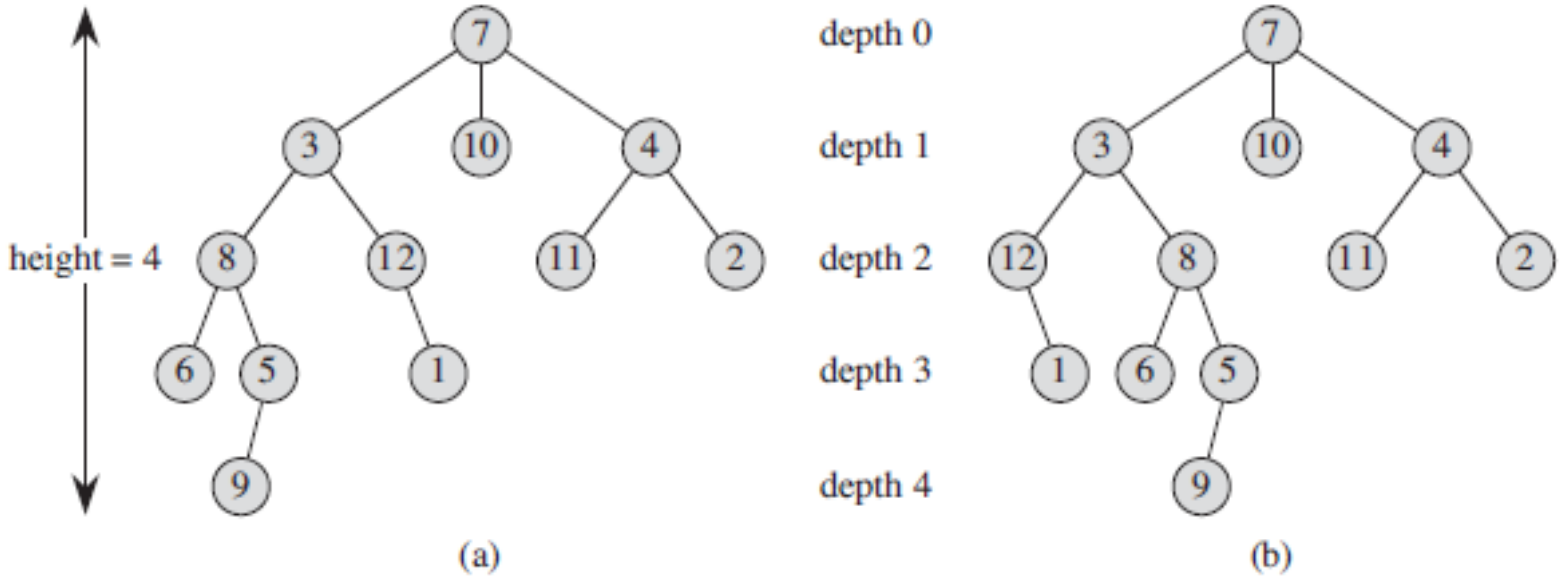
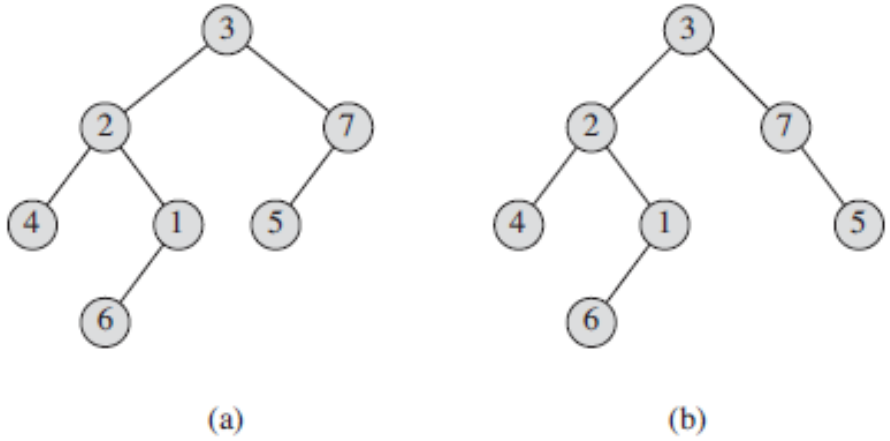


Figure B.6 Rooted and ordered trees. **(a)** A rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-to-right order of the children of a node matters; otherwise it doesn't. **(b)** Another rooted tree. As a rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of node 3 appear in a different order.

Tree

- **Binary and positional tree**
- Define binary tree recursively
- A structure defined on a finite set of nodes that either
  - Contains no node, or
  - Is composed of tree disjoint sets of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree



Tree

- A complete binary tree

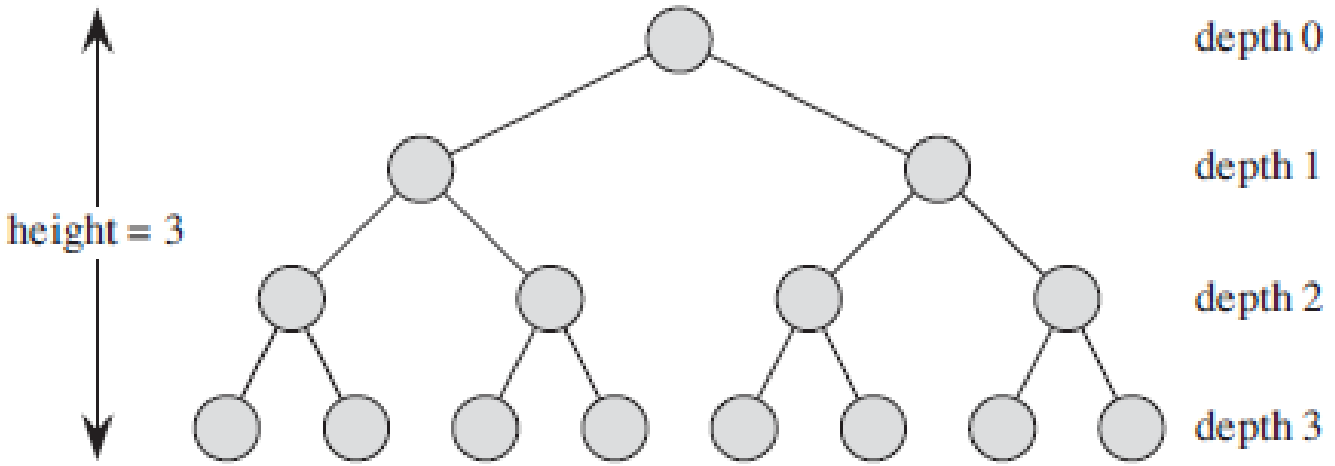


Figure B.8 A complete binary tree of height 3 with 8 leaves and 7 internal nodes.