Heap Sort

Refer to Chap 6 CLRS

Sakeena | 25-01-2022

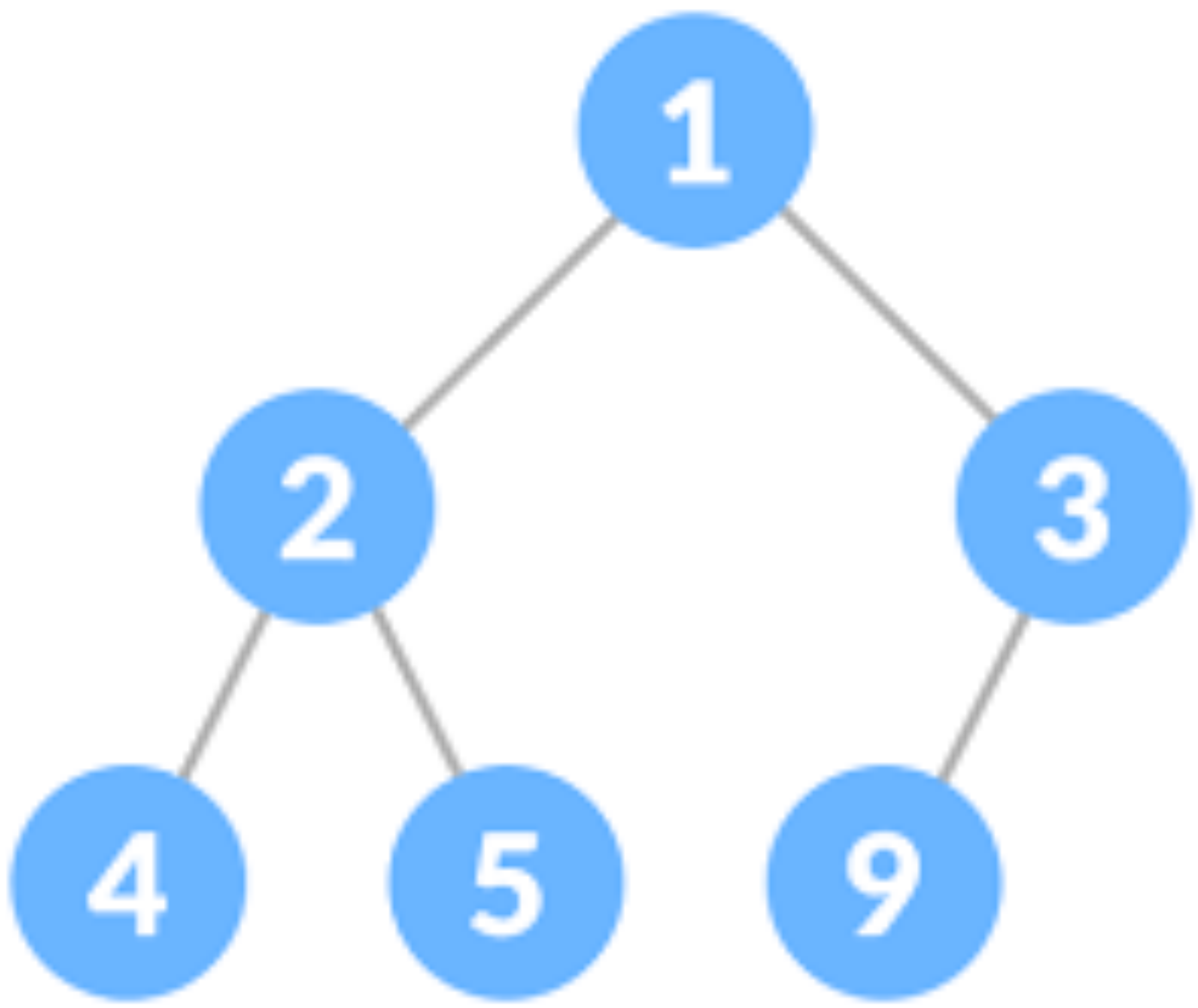
Heap Sort

- Comparison based sorting algorithm.
  - Individual keys of the input are compared to one another to perform sorting.
- Time complexity -> like merge sort [O(nlogn)]
- In-place -> like insertion sort.
- Makes use of heap data structure.

Combines the better attributes of the two.

Recall

Binary Heaps



Heap DS - Recall

- Binary heaps can be represented using arrays. Consider an array A representing a max heap.

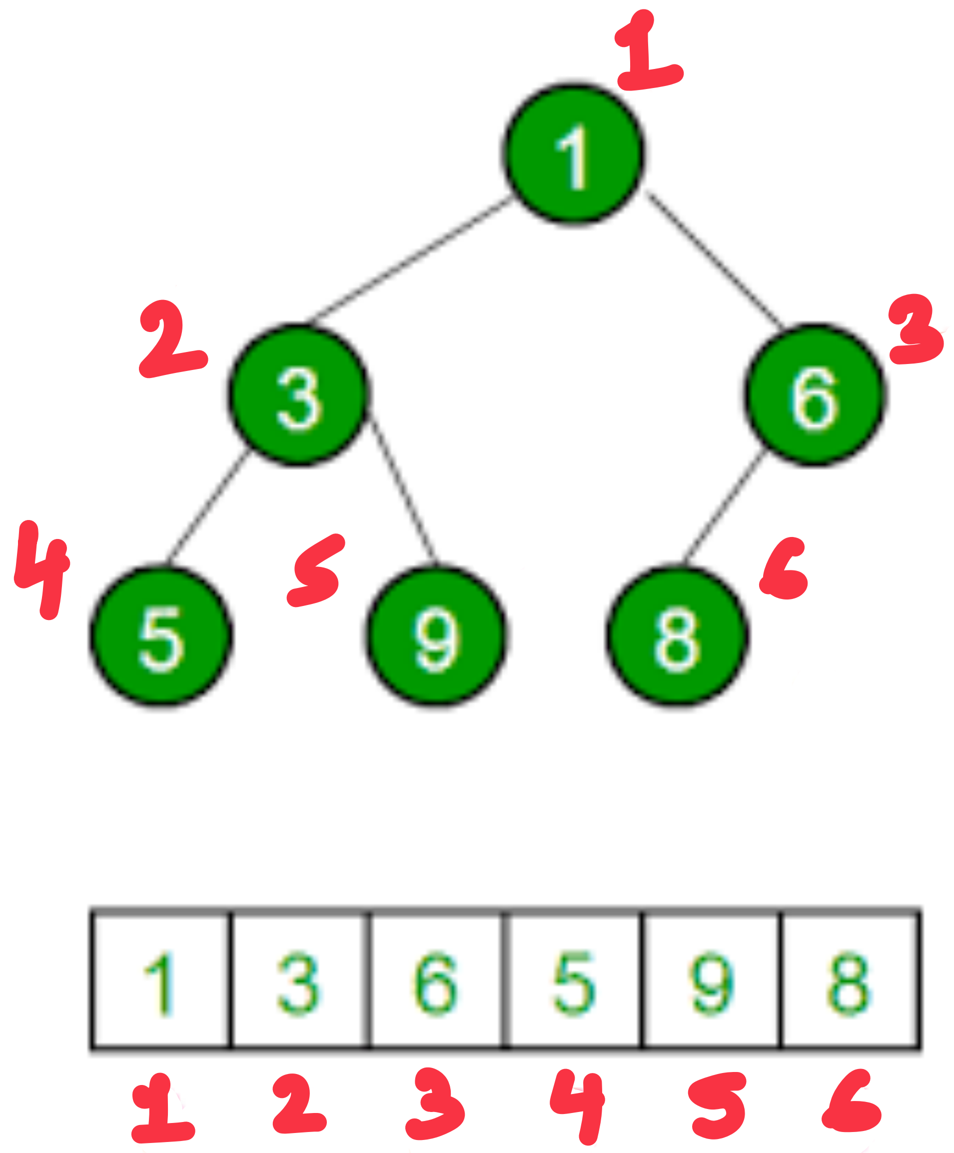
- The root of the heap is A[1].

- Given index of any node, we can compute the index of its parent, left child or right child in the array A by using the following:

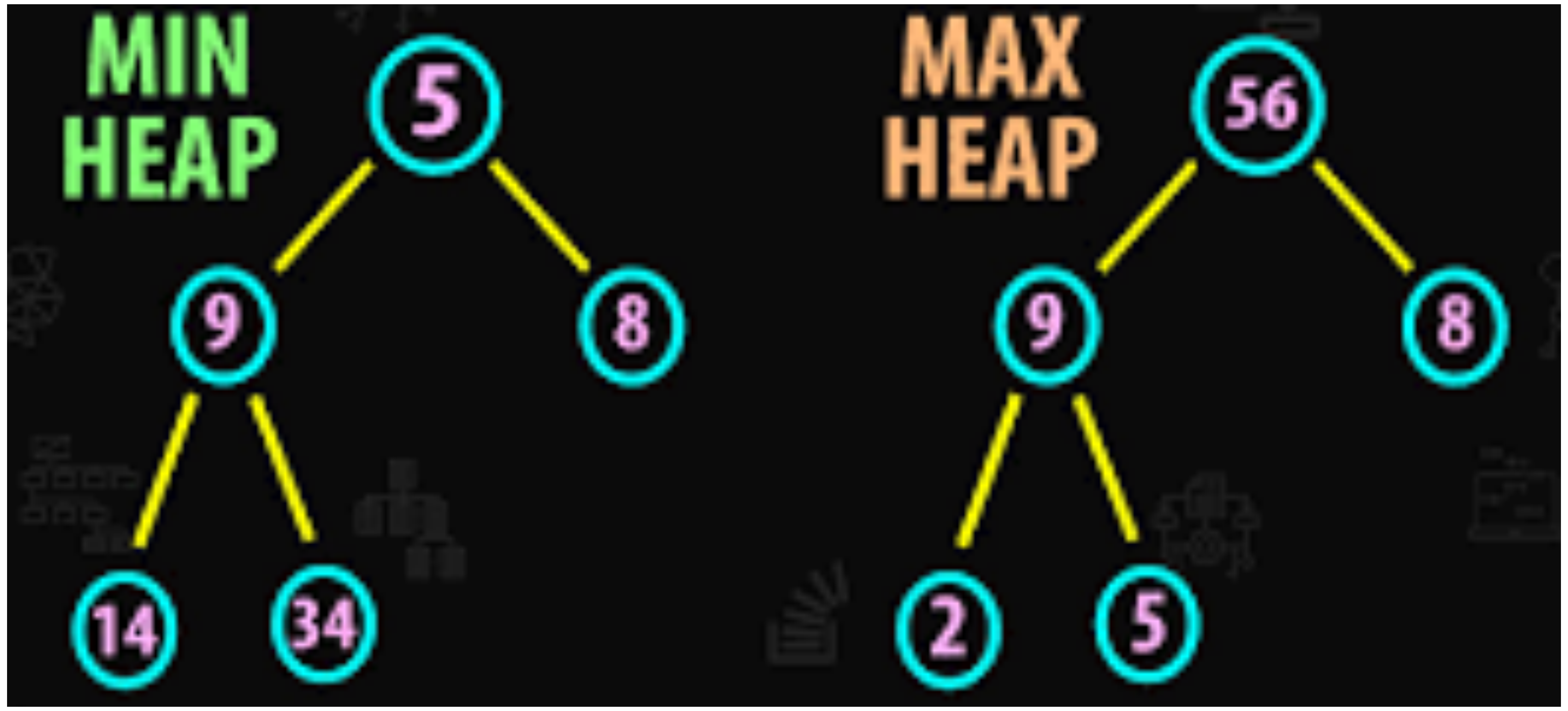
  - Parent(i) = $\lfloor i/2 \rfloor$

  - Left(i) = 2i

  - Right(i) = 2i+1



- Nearly complete binary tree
    - Completely filled on all levels except possibly the last.
    - Last level filled from left up to a point.
    - Satisfies heap property
- Two kinds:
    - Min heap (largest element at the root)
    - Max heap (smallest element at the root)



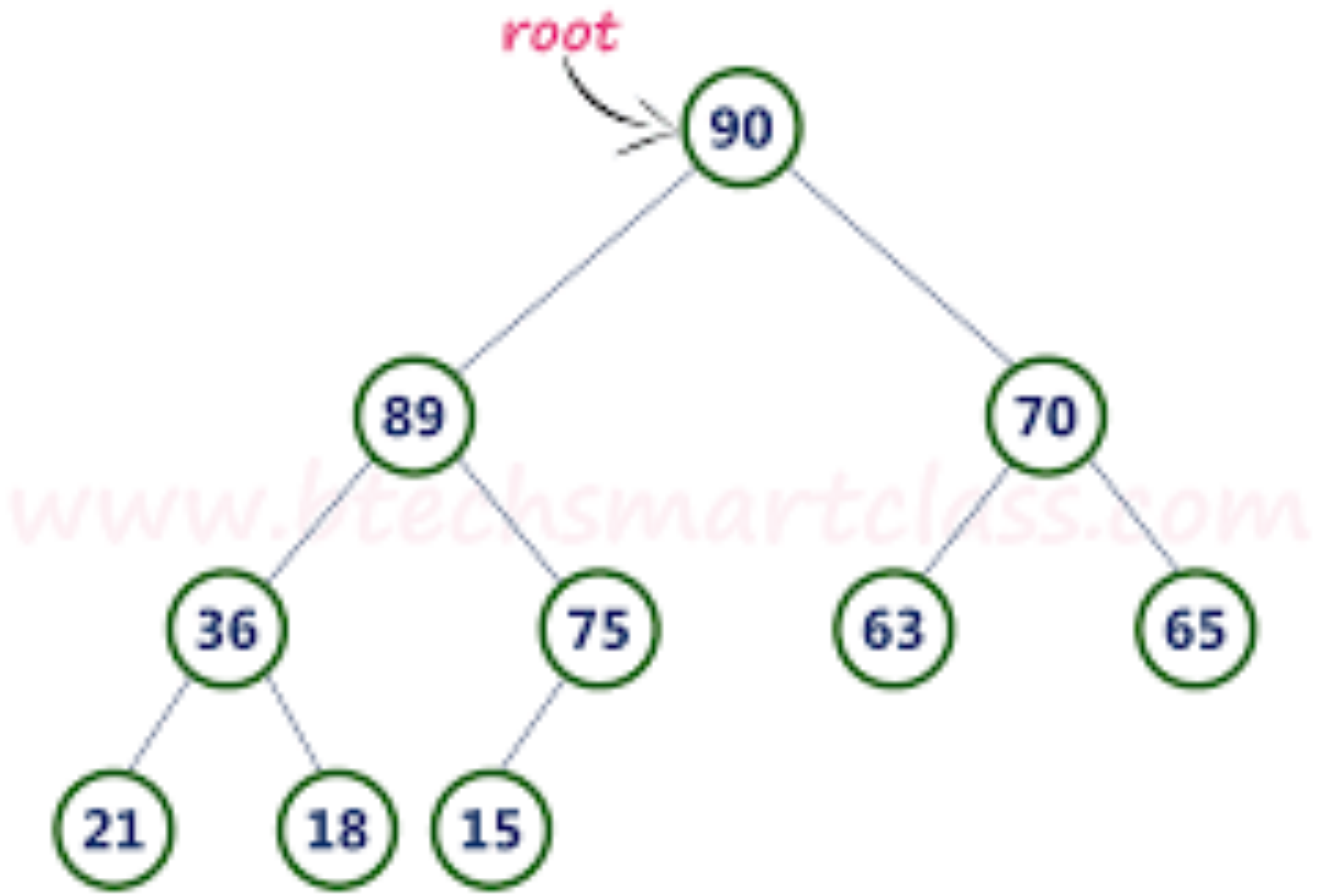
The heap property

- Max-heap property: For every node i in the tree other than the root, $A[\text{parent}(i)] \geq A[i]$
- Min-heap property: ?

Max heap is used for heap sort

Height

- Height of a node in a heap: number of edges on the longest path downward to the leaf
- Height of the heap: height of the root



Let's do it

- What is the minimum and maximum number of elements in a heap of height h?
- Prove that the height of a heap is O(logn).
- If all elements in a max-heap are distinct, where will the smallest value reside?
- Is a sorted array (ascending) a min-heap?
- Is the array {23, 17, 14, 6, 13, 10, 1, 5, 7, 12} a max-heap?

- Can you identify the indices of the leaf nodes in a heap? Note, it is given there are n elements in the heap.

- Can you identify the indices of the leaf nodes in a heap? Note, it is given there are n elements in the heap.

Ans: $\{\lfloor n/2 \rfloor + 1, \ldots, n\}$

Proof:

- Show that all nodes with indices $\{\lfloor n/2 \rfloor + 1, \ldots, n\}$ are childless -> they are leaves.

- Part 1: All nodes in the range of indices $\{\lfloor n/2 \rfloor + 1, \ldots, n\}$ are not having children in this range.

- Part 2: Any node not having children does fall in this range $\{\lfloor n/2 \rfloor + 1, \ldots, n\}$

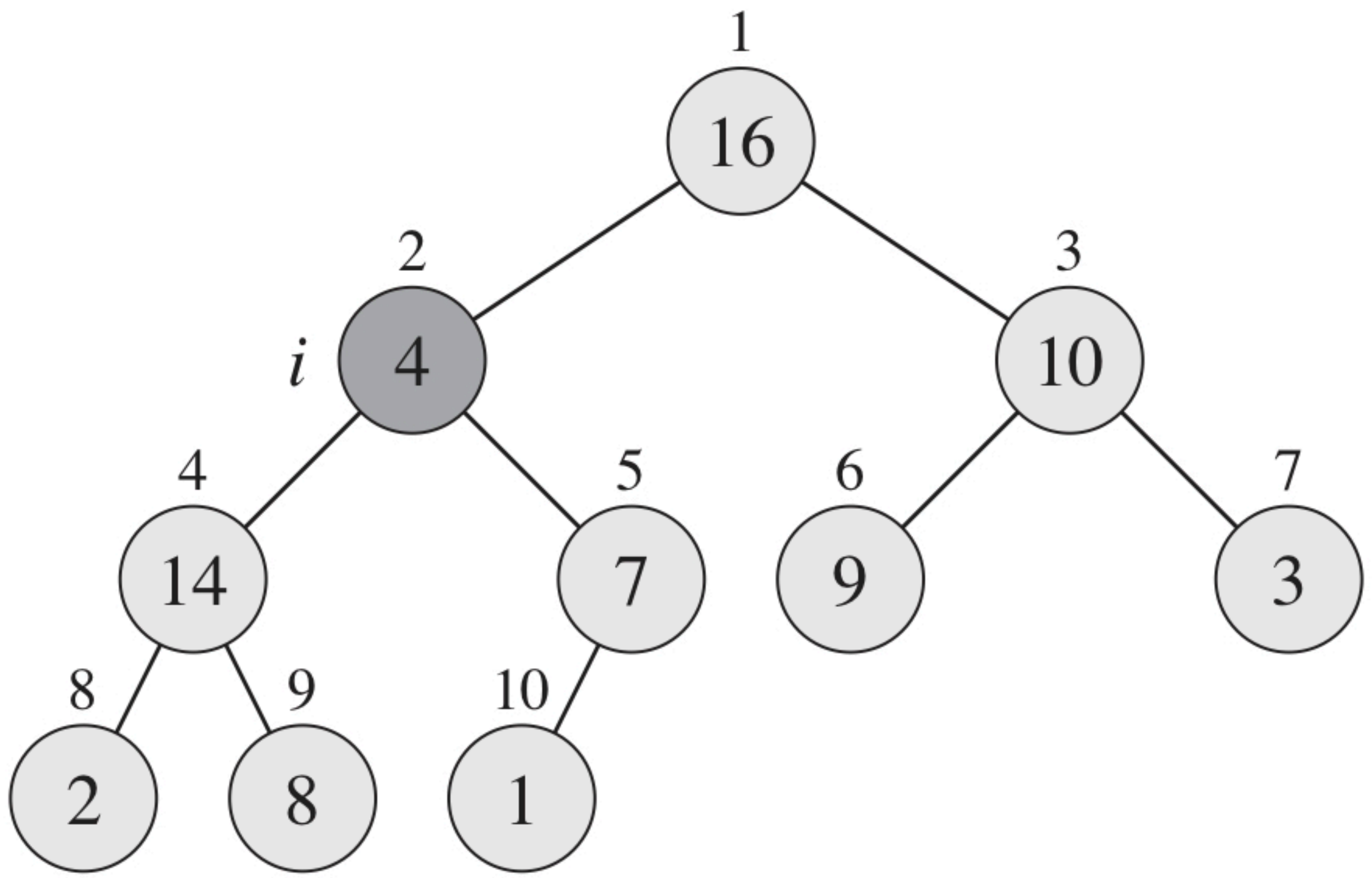
- Part 1: All nodes in the range of indices $\{\lfloor n/2 \rfloor + 1,\ldots,n\}$ are not having children in this range.
    - Let i be a node in this range. It's children are present at index 2i and 2i+1. Assuming, i = $\lfloor n/2 \rfloor + 1$, we have 2i = 2$\lfloor n/2 \rfloor + 2$ > n. Because the left child of the smallest i in the range $\lfloor n/2 \rfloor + 1,\ldots,n$ is >n => all nodes in range $\lfloor n/2 \rfloor + 1,\ldots,n$ are childless.
- Part 2: Any node not having children does fall in this range $\{\lfloor n/2 \rfloor + 1,\ldots,n\}$
    - Let i be a node with no children. That is, 2i and 2i+1 >n. => i>$\lfloor n/2 \rfloor$. => $i \in \{\lfloor n/2 \rfloor + 1,\ldots,n\}$

Maintaining the heap property

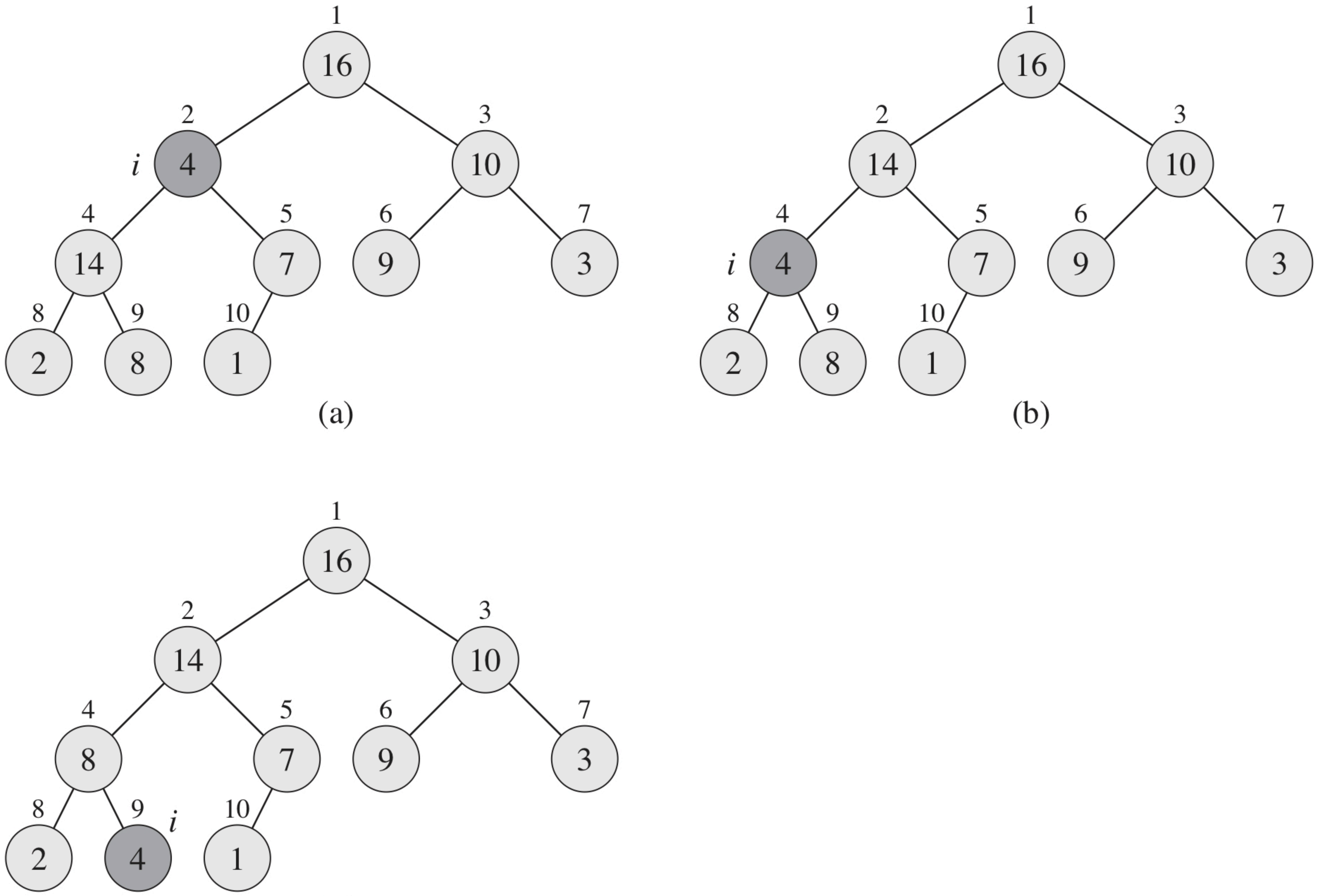
Maintaining the heap property

- A procedure is called : MAX-HEAPIFY
- Input to MAX-HEAPIFY - Array A and index i
- Function assumes left(i) and right(i) are roots of max-heaps, but A[i] might not be. Hence, it floats the value down until the tree rooted at index i becomes a max-heap.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```



(a)



(a) (b) (c)

Running time: MAX-HEAPIFY

- At a given subtree of size n rooted at node i, MAX-HEAPIFY needs $\theta(1)$ to fix the heap property in A[i], A[left(i)] and A[right(i)] + recursive calls to MAX-HEAPIFY on a subtree rooted at one of the children of i.

- The children's subtrees each have a size at most 2n/3 - the worst case occurs when the bottom level is exactly half full (why?)

- Therefore, recurrence relation for MAX-HEAPIFY: $T(n) \leq T(2n/3) + O(1)$

- Solving using master's theorem, the solution is: O(log n)

Let's do it

- Run MAX-HEAPIFY(A,3) on A = {27,17,3,16,13,10,1,5,7,12,4,8,9,0}
- Suppose A.heapSize is the number of elements in the heap. What will happen when you call MAX-HEAPIFY(A, i) for i > A.heapSize/2?
- Modify the heapify procedure for a min-heap/Write procedure for MIN-HEAPIFY.

Building the heap

Building the heap

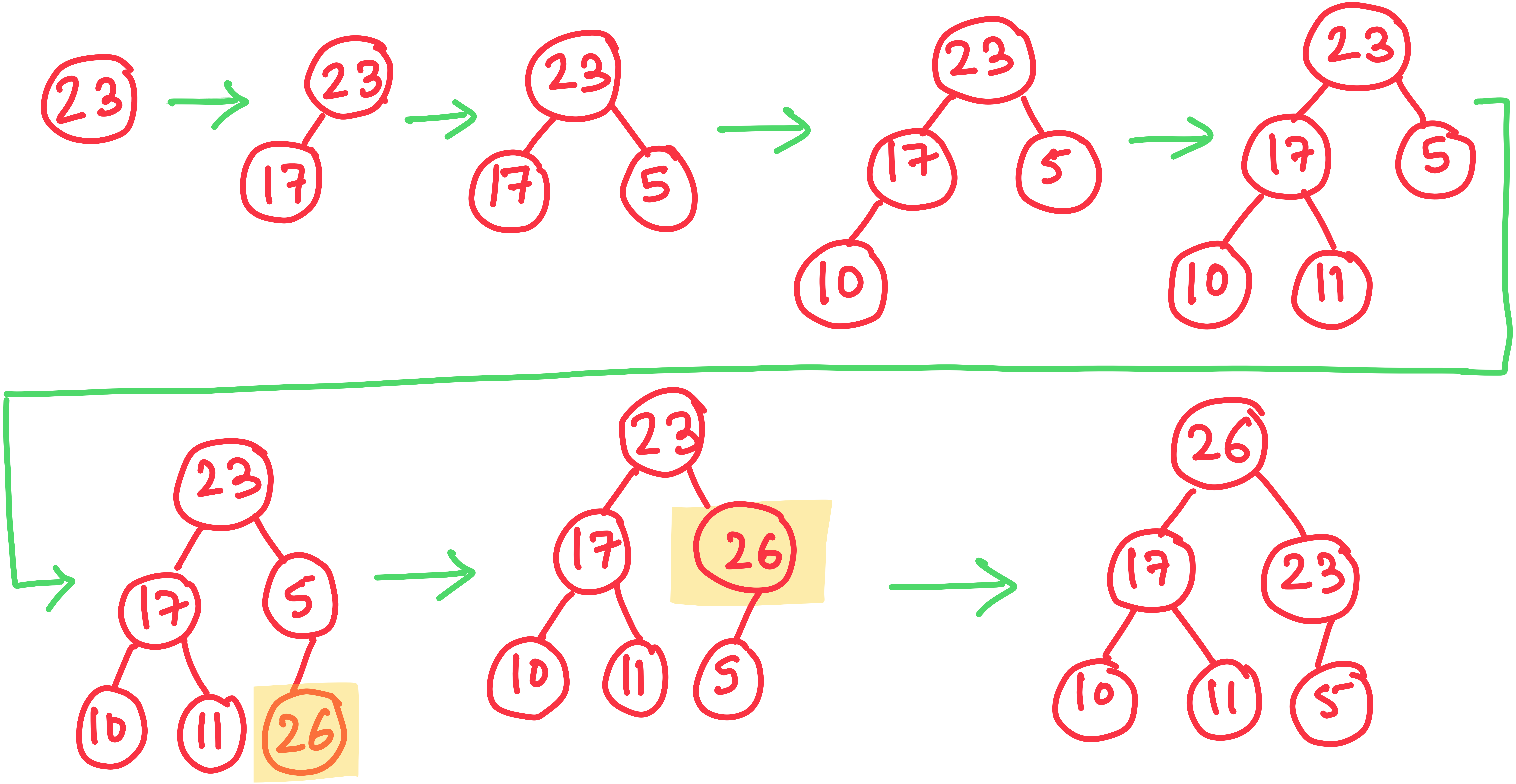
- Two ways:
  - William’s method
  - Floyd’s method (used in book)

Building a heap - William’s method

- Consider an input array: A = { 23, 17, 5, 10, 11, 26}

```
Williams Algorithm: top down
while not end of array,
    if heap is empty,
        place item at root;
    else,
        place item at bottom of heap;
        while (child > parent)
            swap(parent, child);
    go to next array element;
end
```

- Consider an input array: A = { 23, 17, 5, 10, 11, 26}



Running time: William’s method

- The number of operations required depends only on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a worst-case time complexity of O(log n)

- If we do this for every node, then we have O(n*logn) as the running time for William’s method -> not very efficient.

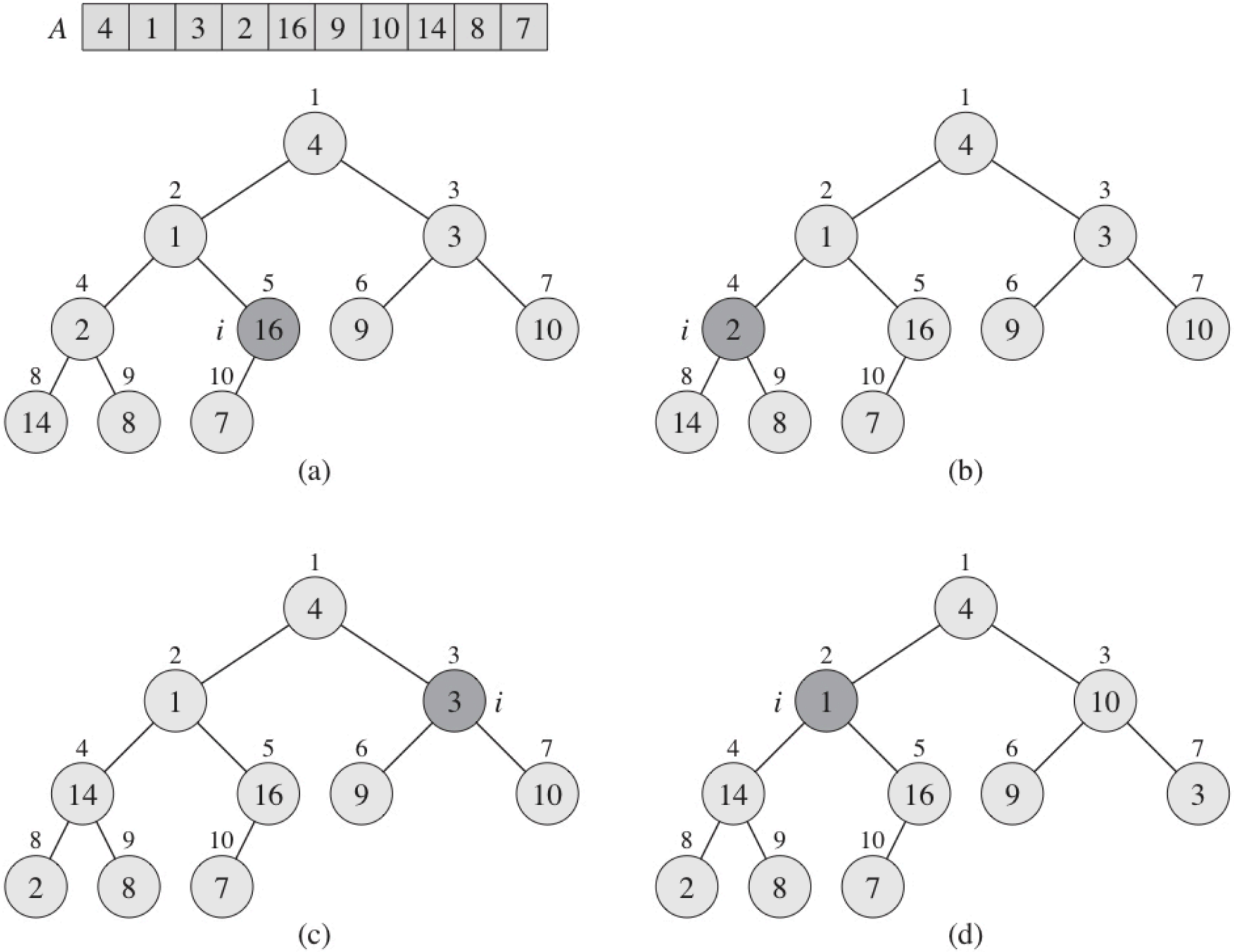
- Efficiency approach proposed by Floyd.

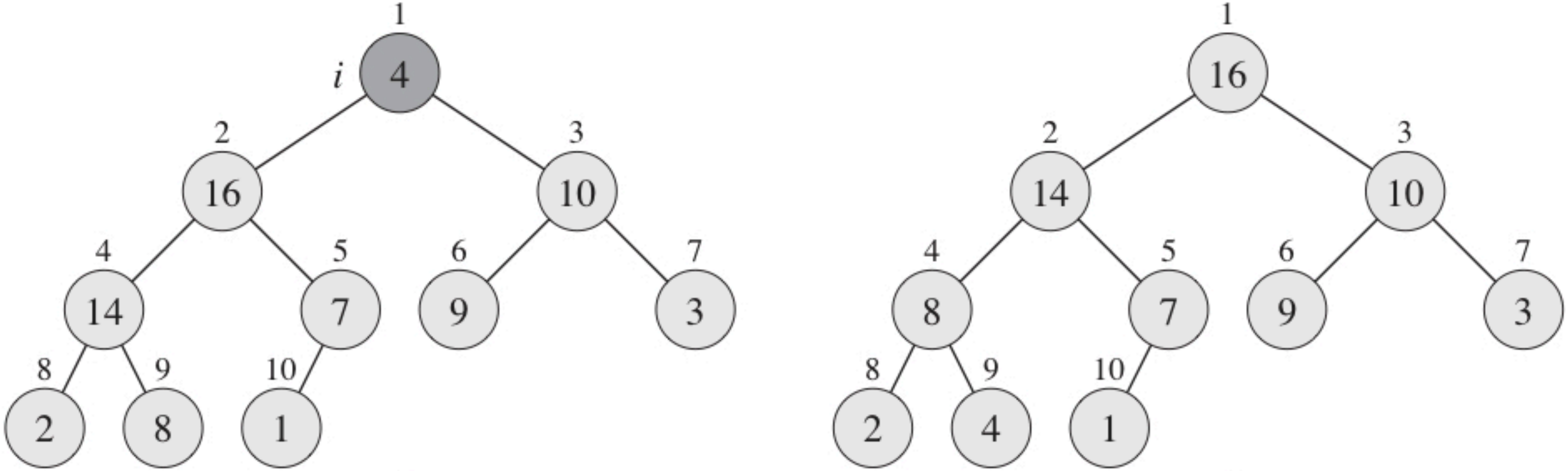
Building a heap - Floyd's method

- The elements in the sub array A[($\lfloor n/2 \rfloor$+1)…n] are all leaves and so each is a 1-element heap to begin with.
- This method runs the MAX-HEAPIFY procedure on all the remaining nodes.
- It is a bottom-up approach.

BUILD-MAX-HEAP(A)

1 A.heapSize = A.length
2 for i = $\lfloor A.length/2 \rfloor$ down to 1
3. MAX-HEAPIFY(A, i)


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



(e) (f)

Correctness of Heap sort

- Depends on the correctness of BUILD-MAX-HEAP procedure.

Correctness of BUILD-MAX-HEAP

- LOOP INVARIANT ??

Correctness of BUILD-MAX-HEAP

- LOOP INVARIANT

- **At the start of each iteration of the for loop, each node i+1, i+2, … , n is the root of a max-heap.**

- Need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and the invariant provides a useful property to show the correctness when the loop terminates.

BUILD-MAX-HEAP(A)

1 A.heapSize = A.length
2 for i = $\lfloor A.length/2 \rfloor$ down to 1
3. MAX-HEAPIFY(A, i)

Initialisation:

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

Maintenance:

- The children of node i are numbered higher than i.
- By the loop invariant, they are both roots of max-heaps.
- When we call MAX-HEAPIFY(A, i), it will make i a max-heap root. Moreover, it preserves the property that i+1, i+2 , .., n are all roots of max-heaps.
- Decrementing i in the floor loop update reestablishes the loop invariant for the next iteration.

```
BUILD-MAX-HEAP(A)

1 A.heapSize = A.length
2 for i = ⌊A.length/2⌋ down to 1
3.   MAX-HEAPIFY(A, i)
```

```
BUILD-MAX-HEAP(A)

A.heapSize = A.length
for i = ⌊A.length/2⌋ down to 1
  MAX-HEAPIFY(A, i)
```

Termination:

- At termination, i=0
- By the loop invariant each node 1, 2, …, n is the root of a max-heap. Node 1 is.

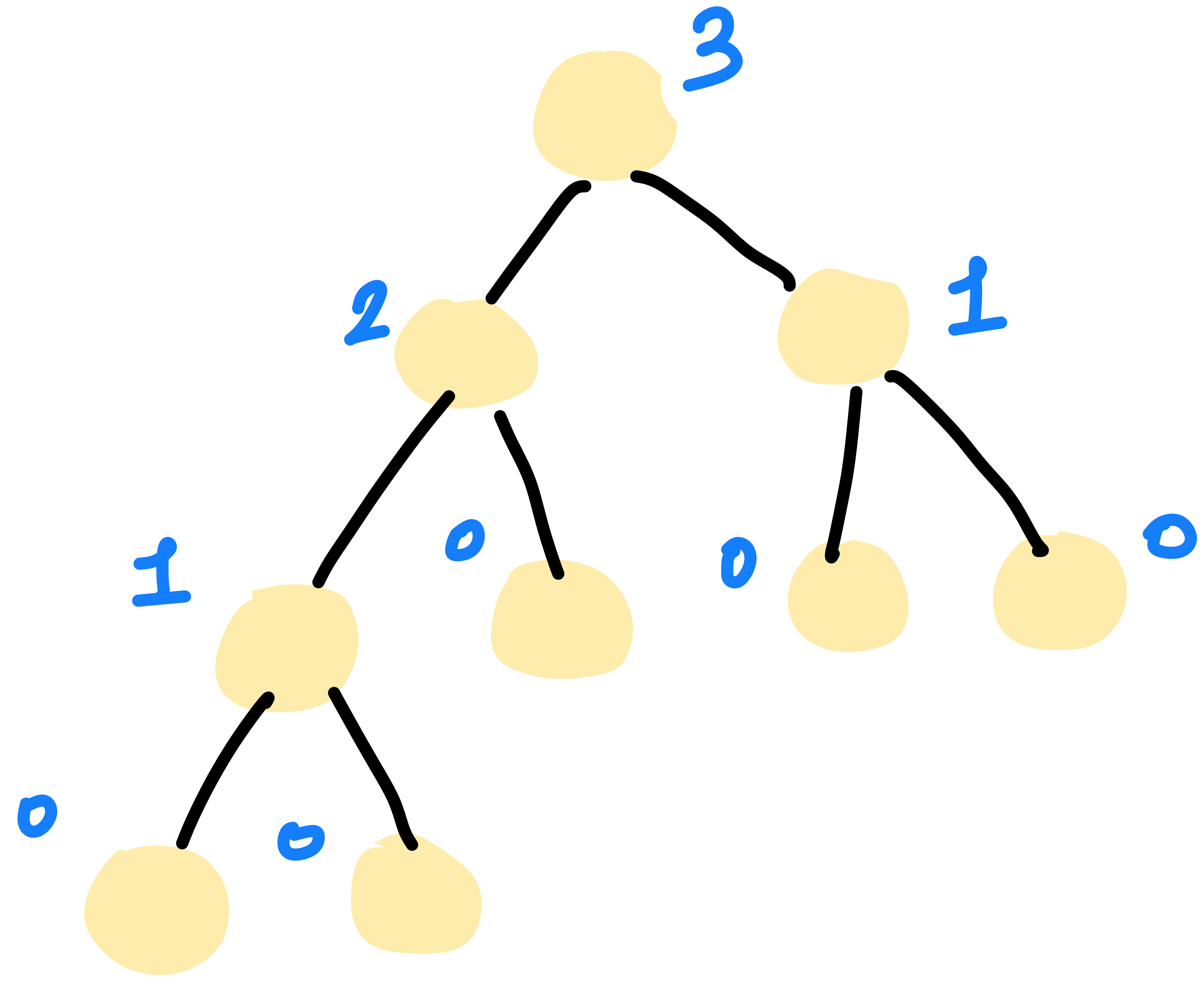
Running time: Floyd's method

Simpler to understand:

- Each call to MAX-HEAPIFY requires O(logn) time.
- We make O(n) such calls.
- Total = O(n*logn).
- Although correct, but not a tight bound.

Running time: Floyd's method

- The time for MAX-HEAPIFY is not always logn.
- It varies with the height of the node in the tree, and height of most nodes is small.
- The height of a n-node heap has a height of $\lfloor log n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.



- The time required by MAX-HEAPIFY when called on a node of height h is O(h).
- Thus, we can express the total time required by BUILD-MAX-HEAP as below:
- $\sum_{h=0}^{\lfloor logn \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h)$
- $O\left(n \sum_{h=0}^{\lfloor logn \rfloor} \frac{h}{2^h}\right)$
- We are already familiar with the series $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$
- Thus we have, $O\left(n \sum_{h=0}^{\lfloor logn \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$

Heap can be built in linear time

Let’s do it

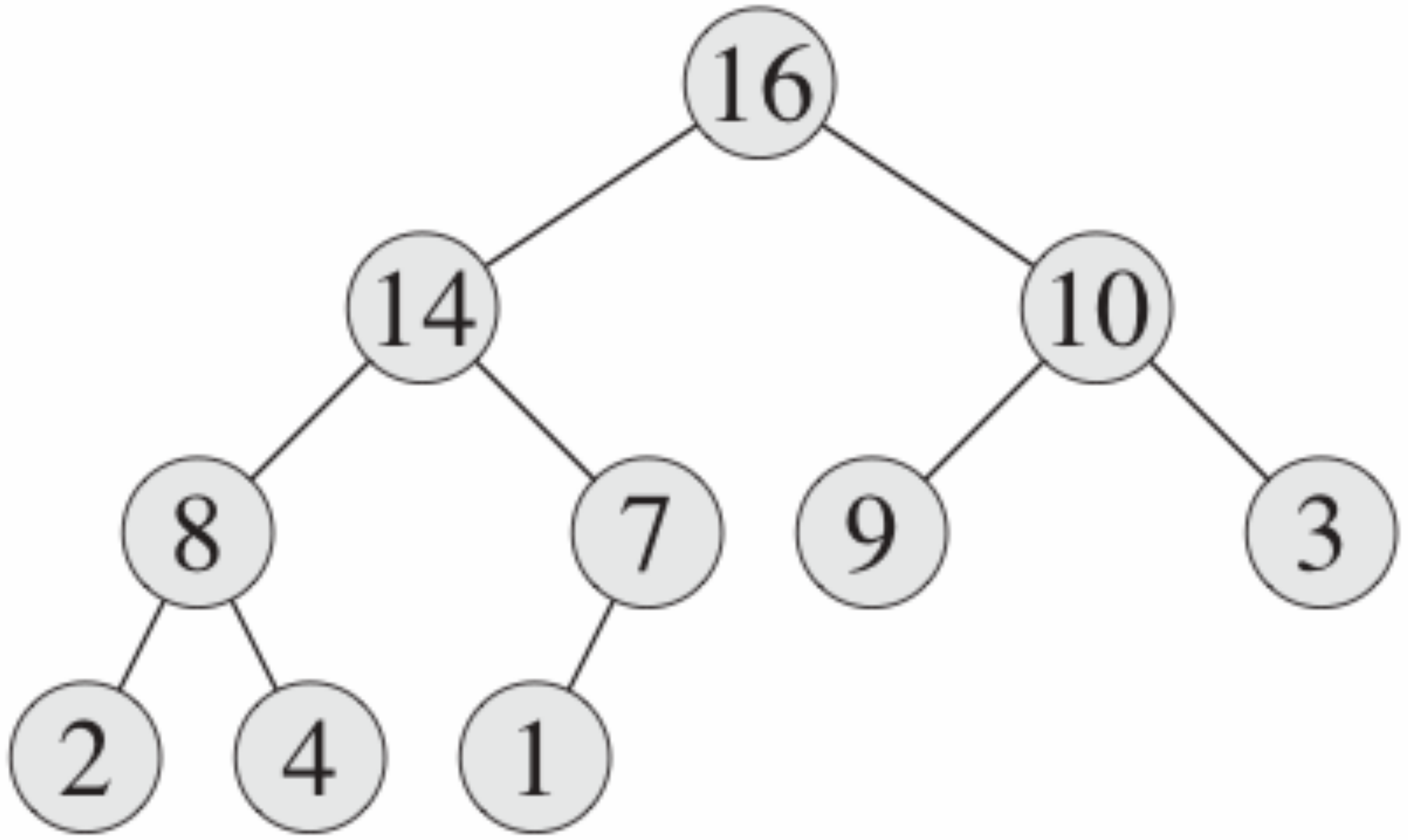
- Build a heap from the array A = {5, 3, 17, 10, 84, 19, 6, 22, 9}

Pulling it all together - The heap sort algorithm

Heap Sort Algorithm

HeapSort(A)

```
BUILD-MAX-HEAP(A)
for i = A.length down to 2
    Exchange A[1] with A[i]
    A.heapSize = A.heapSize-1
    MAX-HEAPIFY(A, 1)
```

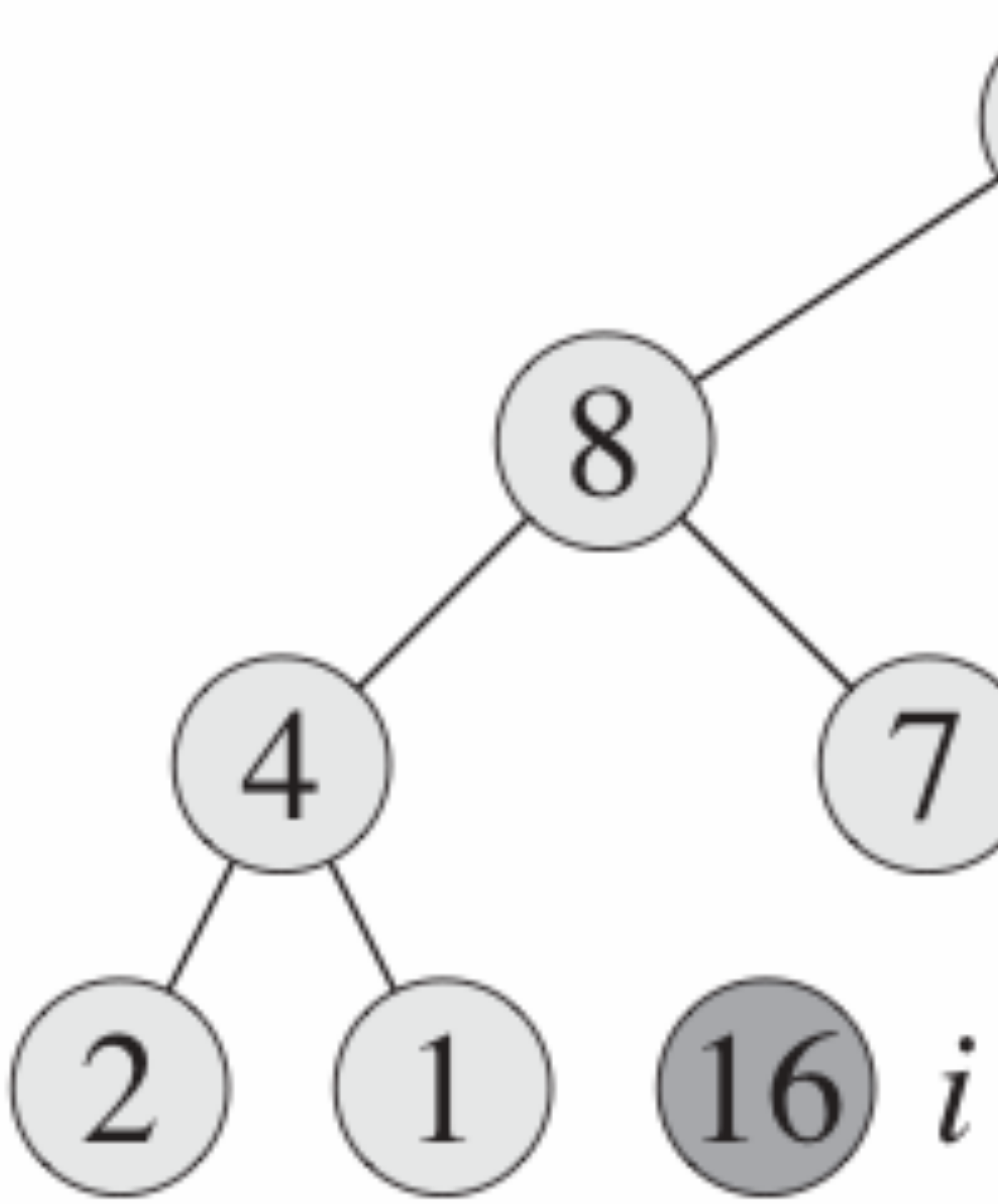


(a)

HeapSort(A)

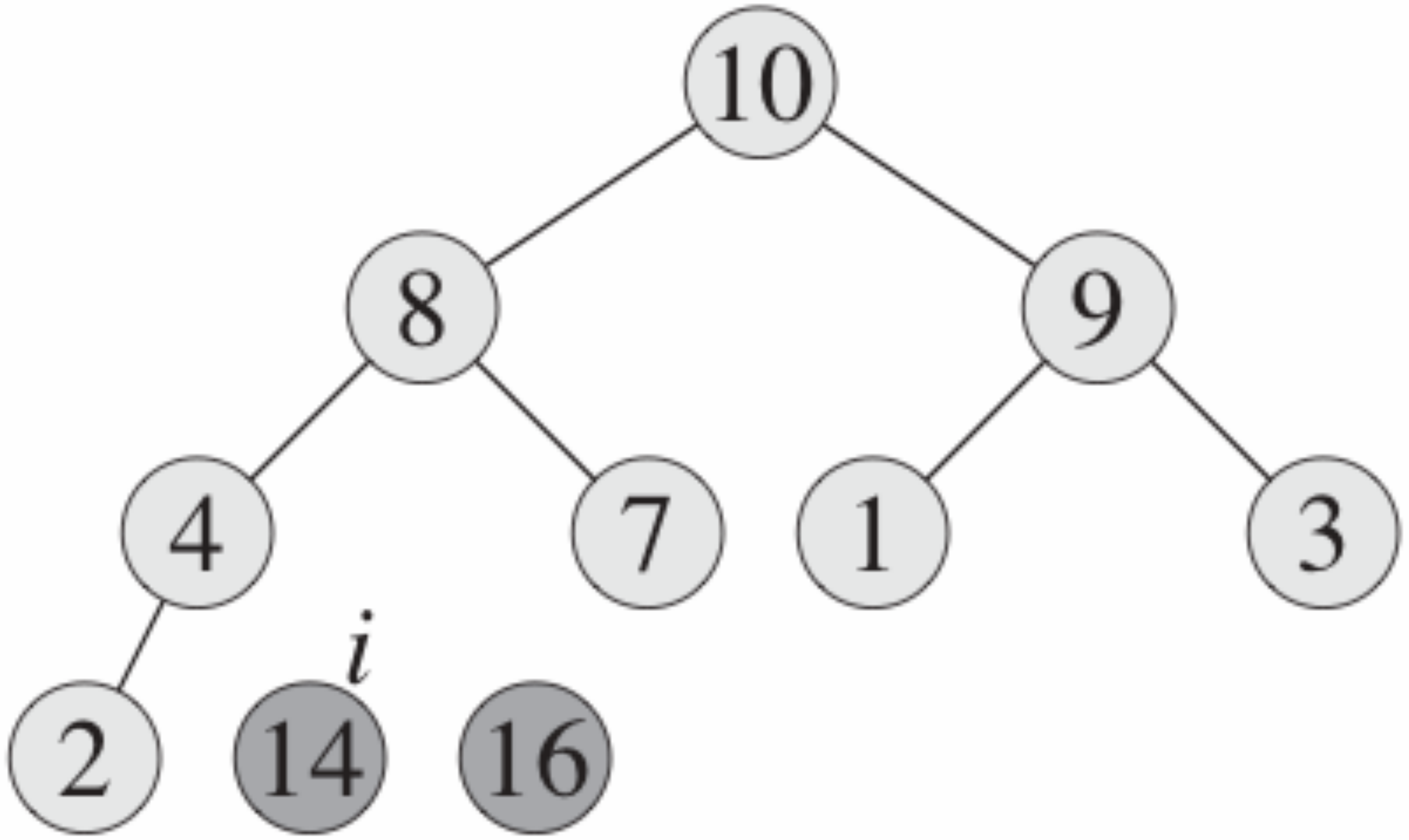
```
BUILD-MAX-HEAP(A)
for i = A.length down to 2
    Exchange A[1] with A[i]
    A.heapSize = A.heapSize-1
    MAX-HEAPIFY(A, 1)
```

$i = 10$



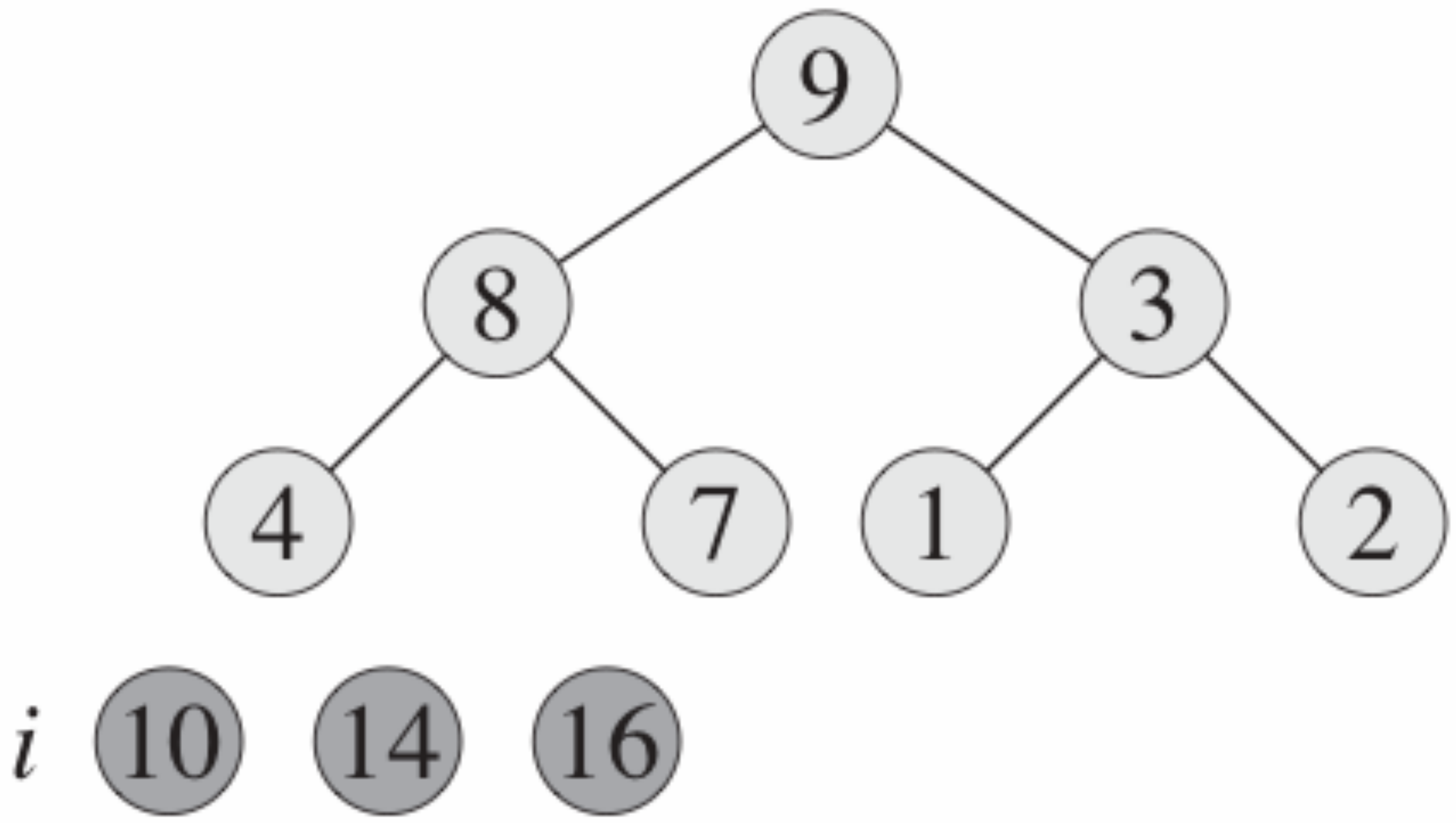
(b)

$i = 9$



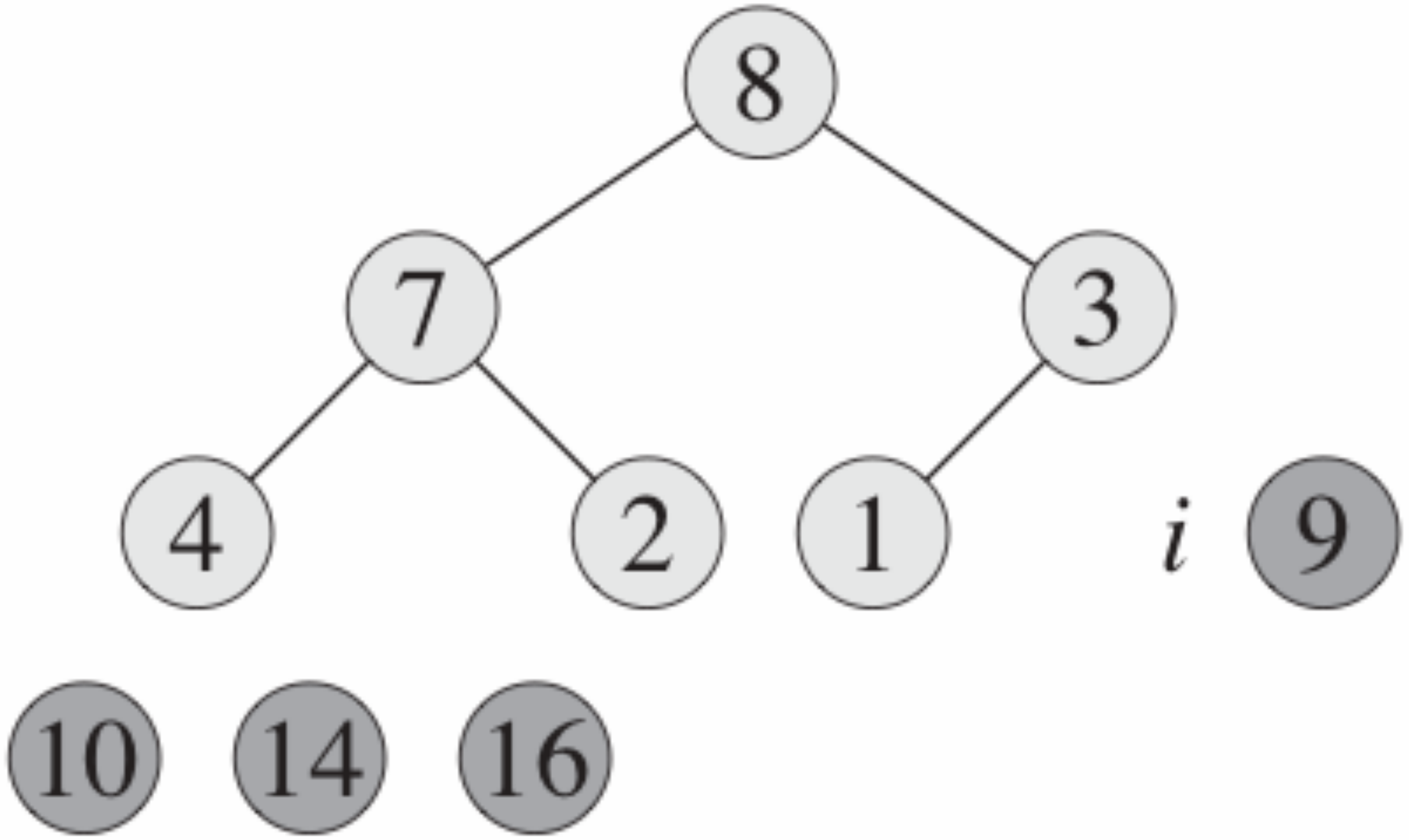
(c)

$i = 8$



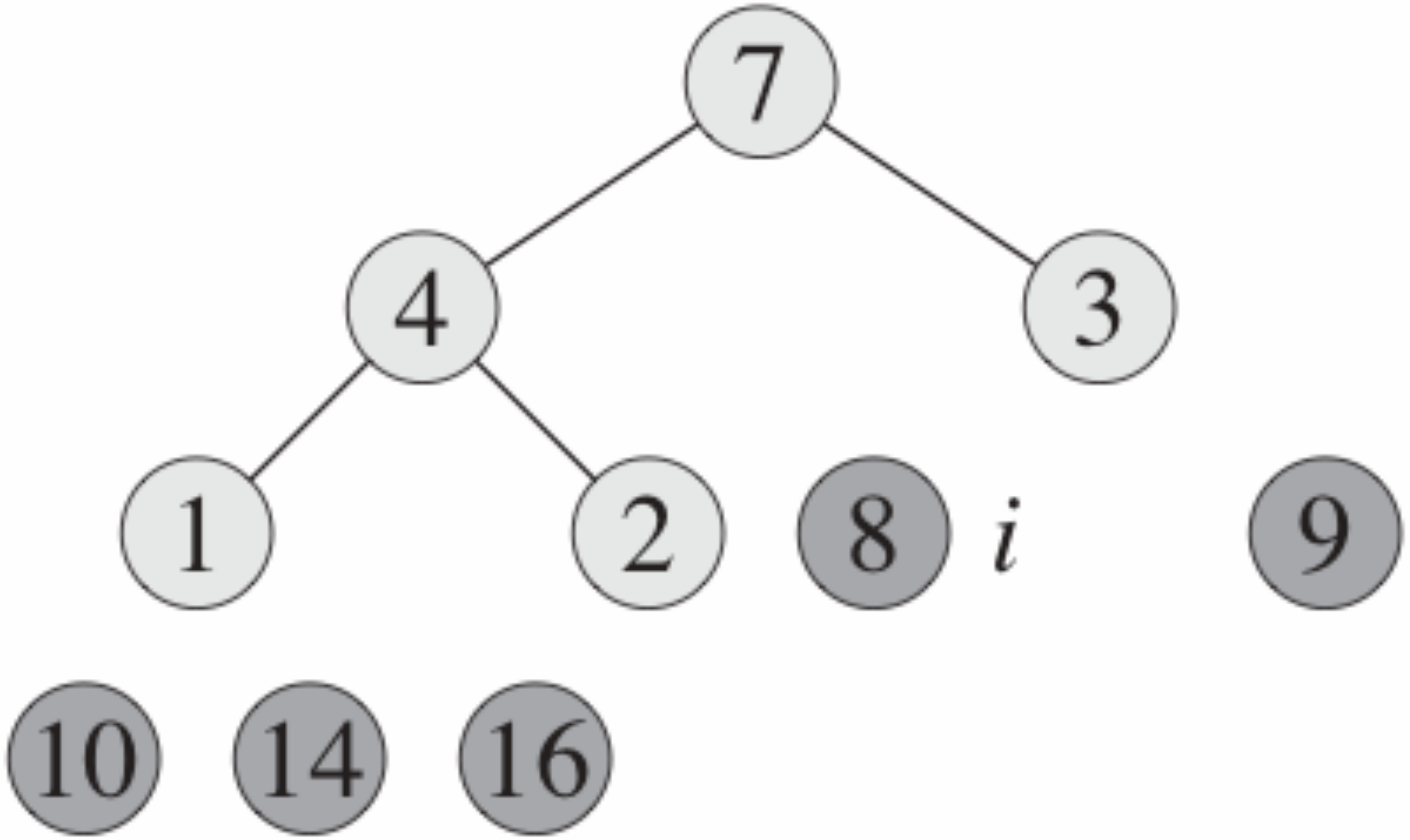
(d)

$i = 7$



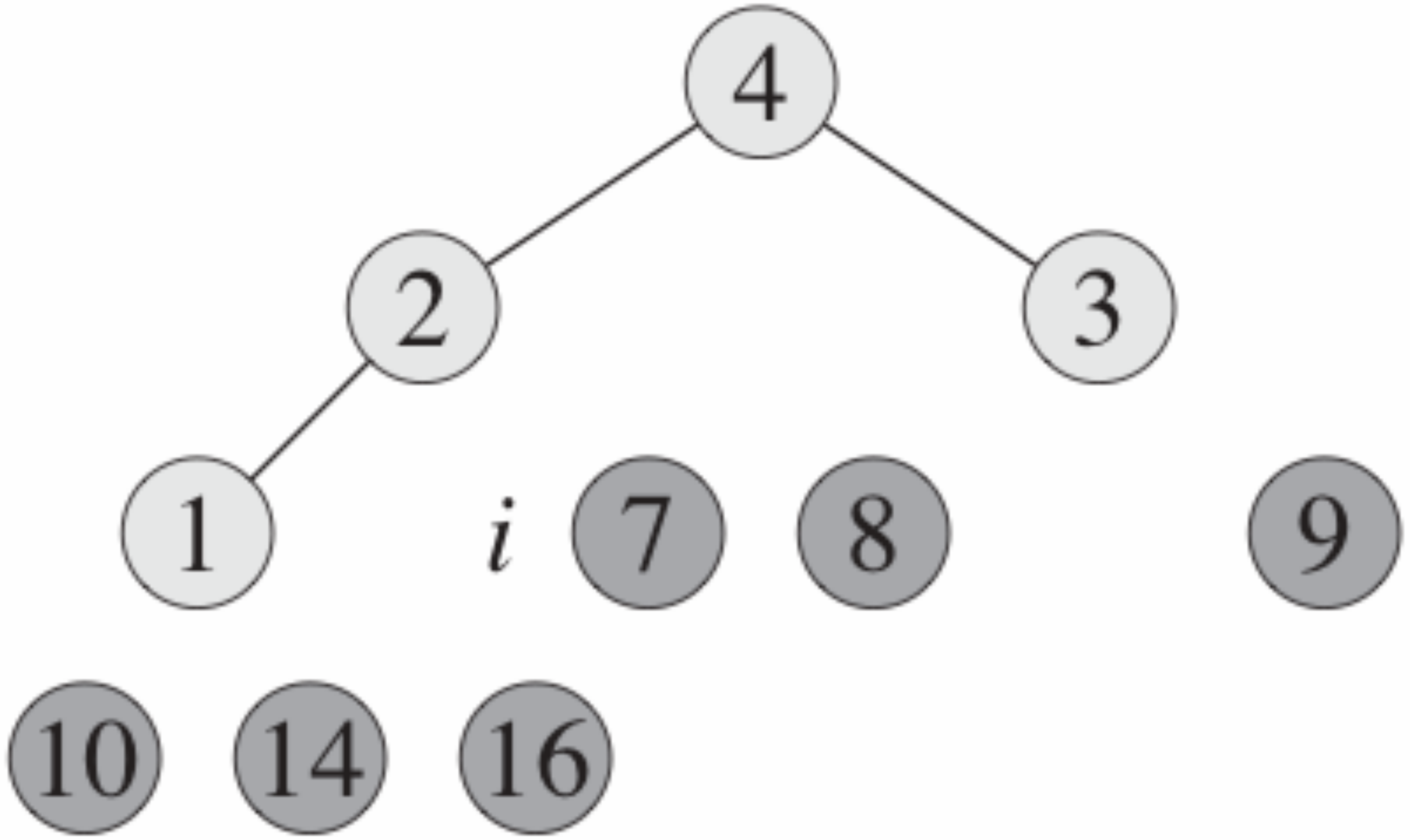
(e)

$i = 6$

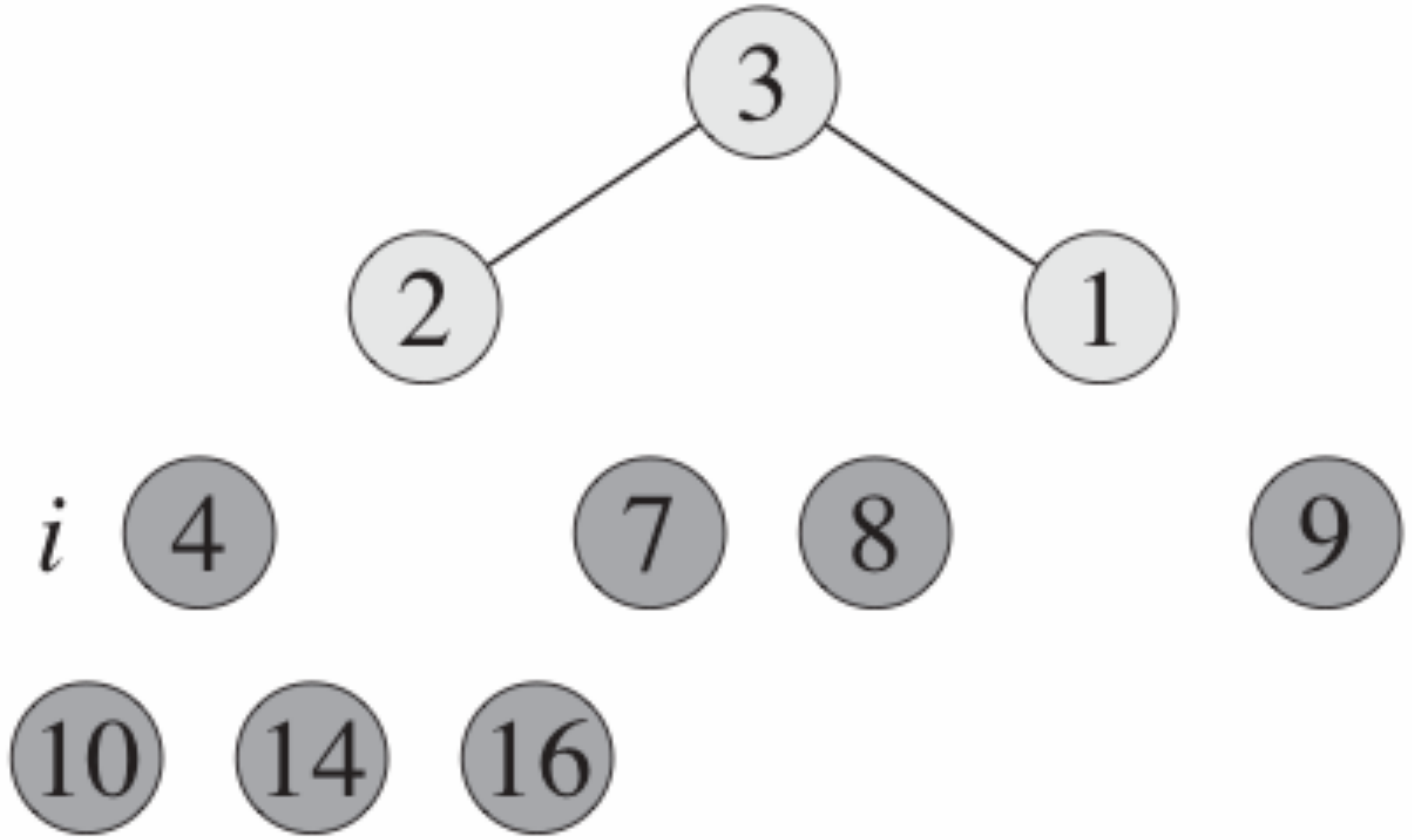


(f)

$i = 5$

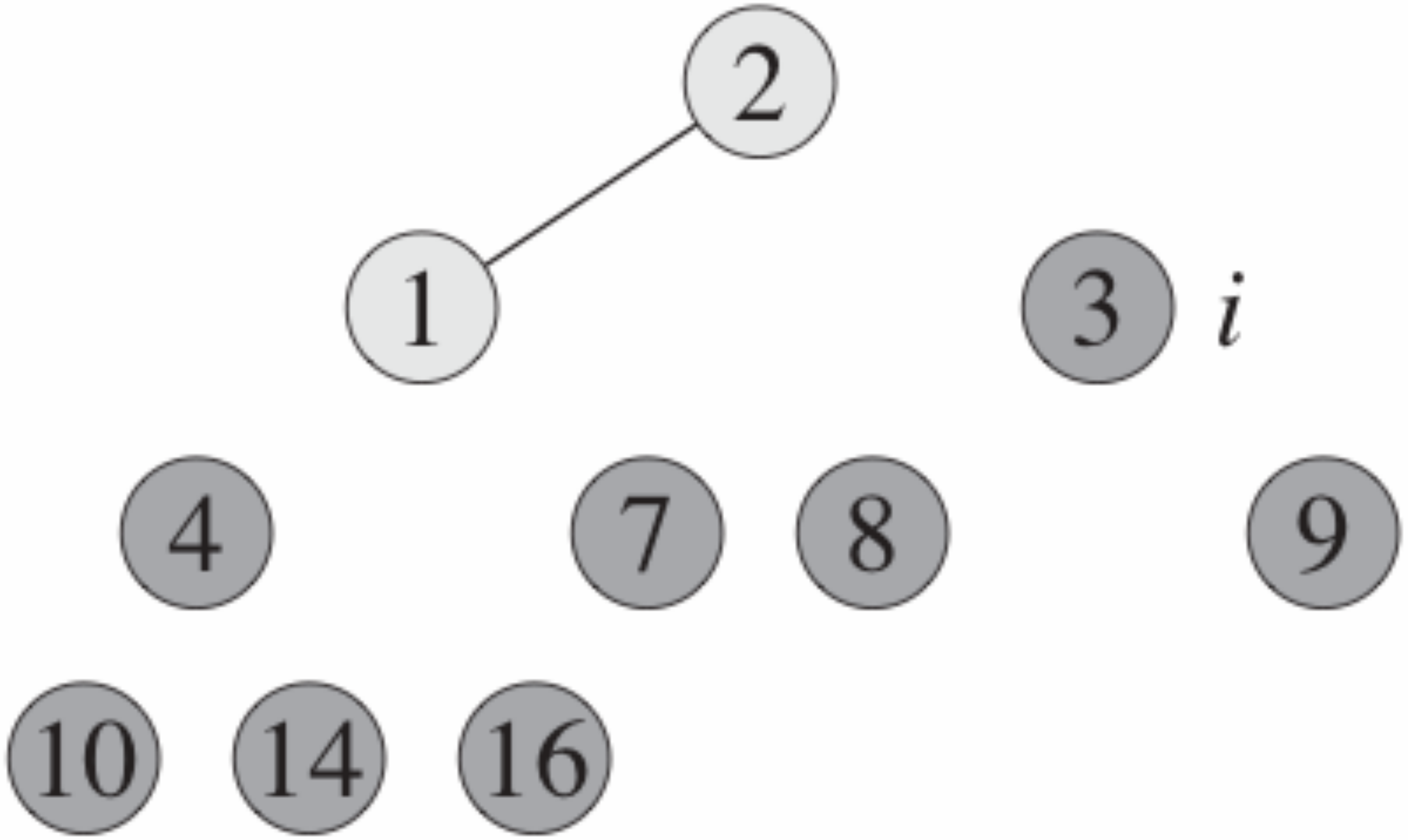


(g)



(h)

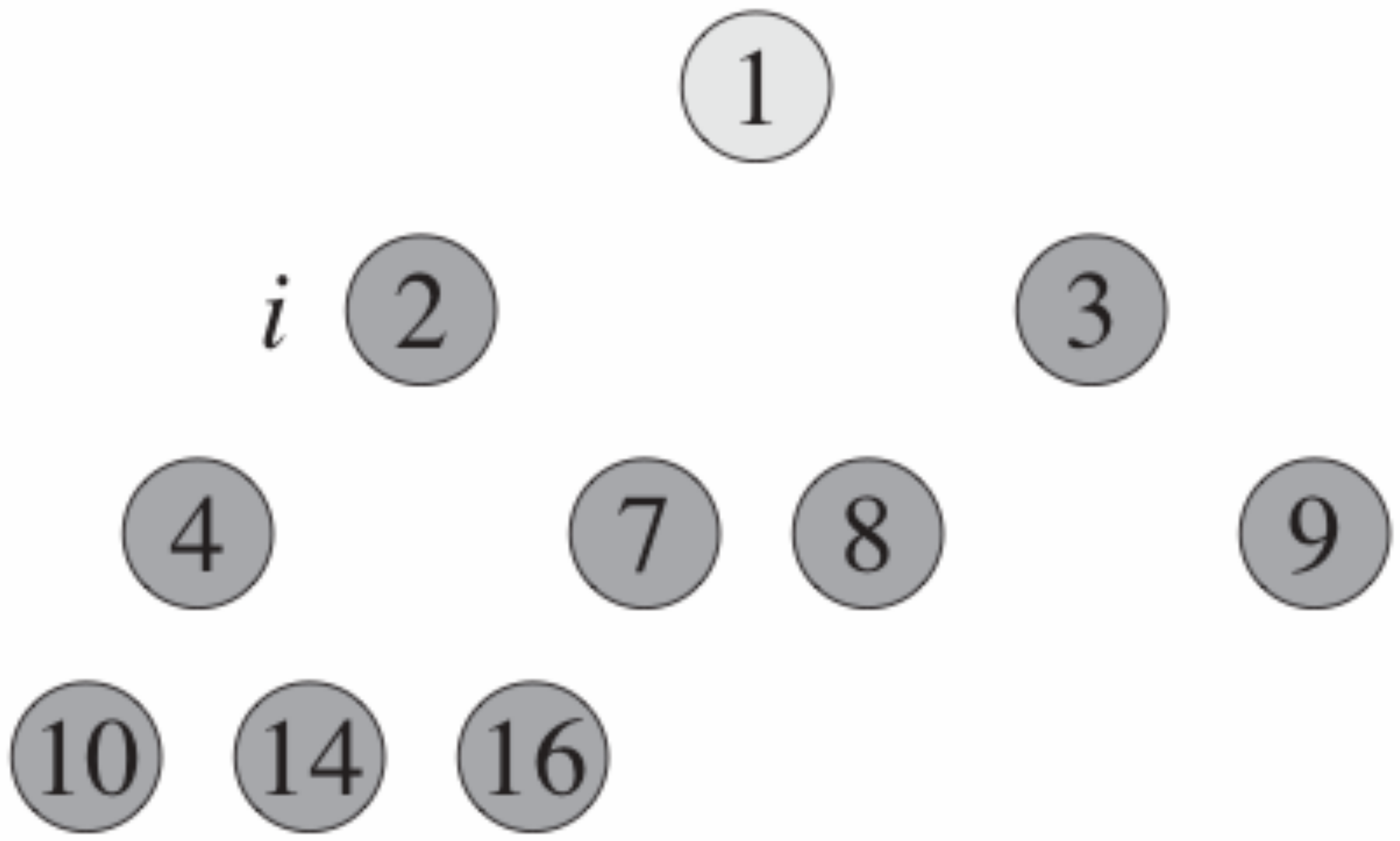
$i = 4$



(i)

$i = 3$

$i = 2$



A	1	2	3	4	7	8	9	10	14	16

Priority Queues

Priority Queues

- Priority queue is a DS for maintaining a set S of elements each with an associated value called key.
- Types:
  - Min priority queue (implemented using min heaps)
  - Max priority queue (implemented using max-heaps)

Max priority queue

- Supports the following functions:
  - INSERT(S, x): Inserts an element x into set S (equivalent to the operation S U {x})
  - MAXIMUM(S): returns element of S with 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 x’s key to the new value k (assumed to be as large as x’s current key value)

HEAP-MAXIMUM(A)

HEAP-MAXIMUM(A)

1 **return** $A[1]$

Time complexity?

HEAP-MAXIMUM(A)

HEAP-MAXIMUM(A)

1 **return** $A[1]$

Time complexity? $\Theta(1)$

HEAP-EXTRACT-MAX(A)

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Time complexity?

HEAP-EXTRACT-MAX(A)

HEAP-EXTRACT-MAX(A)

```
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Time complexity?

O(logn)

HEAP-INCREASE-KEY(A, i, key)

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
    error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
    exchange $A[i]$ with $A[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

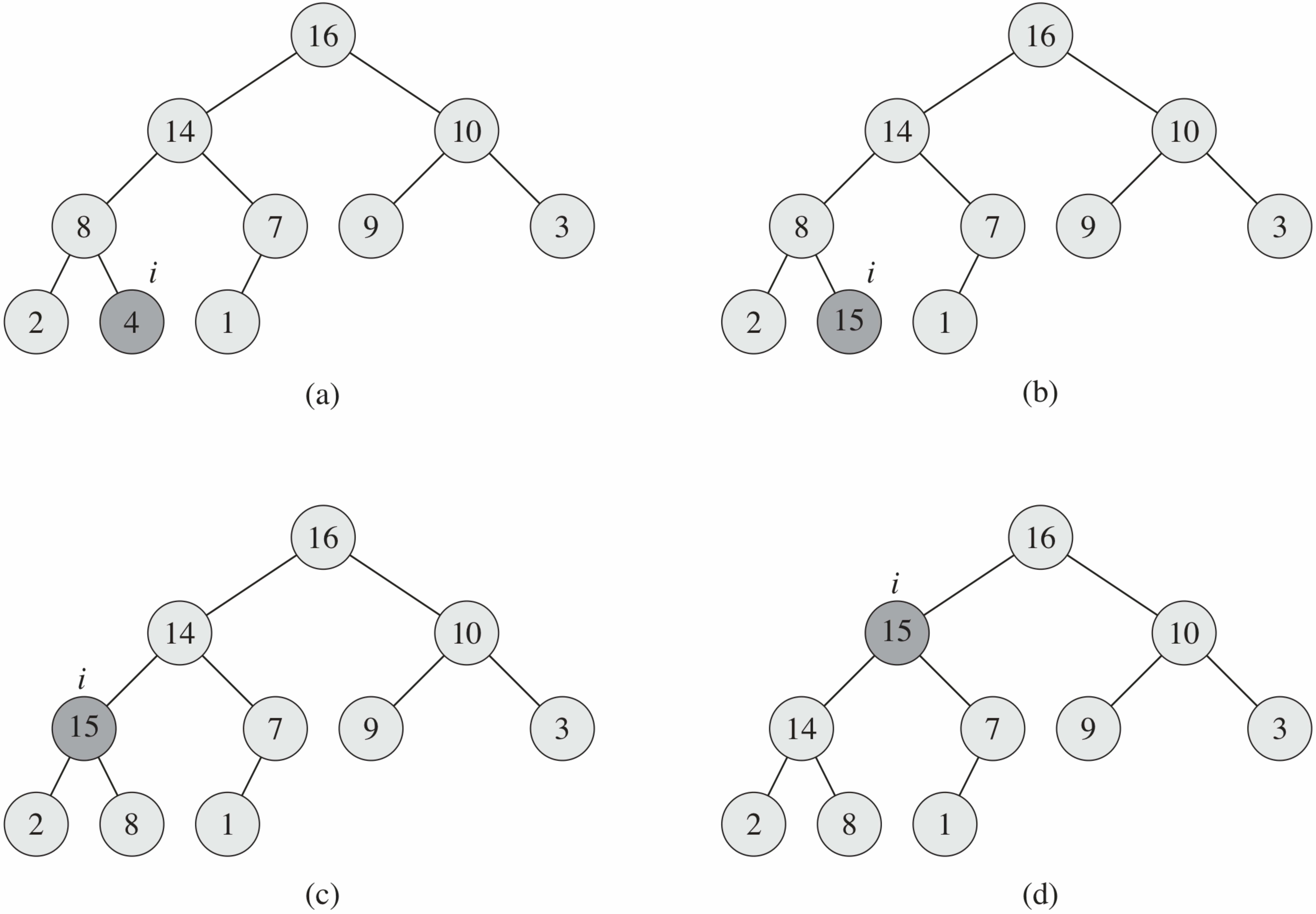
Time complexity?

HEAP-INCREASE-KEY(A, i, key)

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

Time complexity?

$O(\log n)$


16
14
10
8
7
9
3
i
2
4
1
(a)
16
14
10
8
7
9
3
i
2
15
1
(b)
16
14
10
i
15
7
9
3
2
8
1
(c)
16
i
15
10
14
7
9
3
2
8
1
(d)

MAX-HEAP-INSERT(A, key)

```
MAX-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = -∞
    HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Time complexity?

MAX-HEAP-INSERT(A, key)

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

Time complexity?

$O(\log n)$