Heap Data Structure

Zahoor Jan

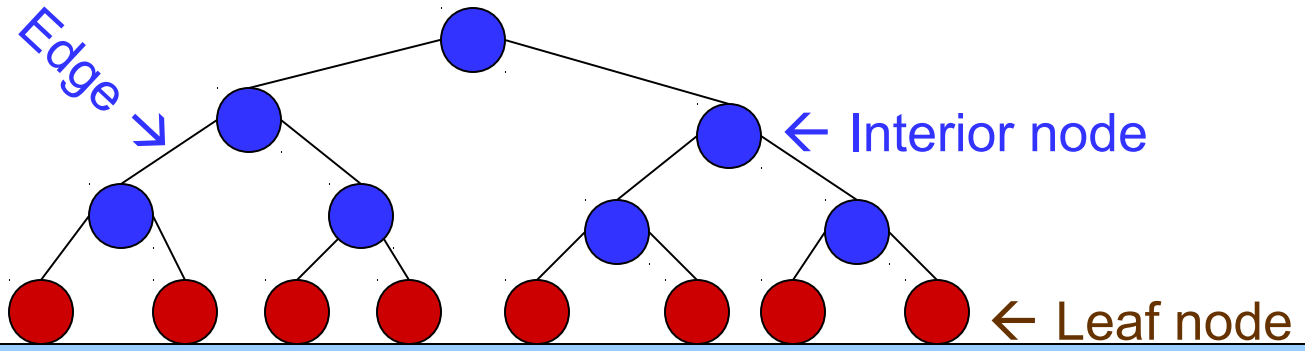
Heaps

A *(binary) heap* data structure is an array object that can be viewed as a complete binary tree

Each node of the tree corresponds to an element of the array that stores the value in the node

A complete binary tree is one where all the internal nodes have *exactly* 2 children, and all the leaves are on the same level. The tree is completely filled on all levels except possibly the lowest, which is filled from left to right up to a point

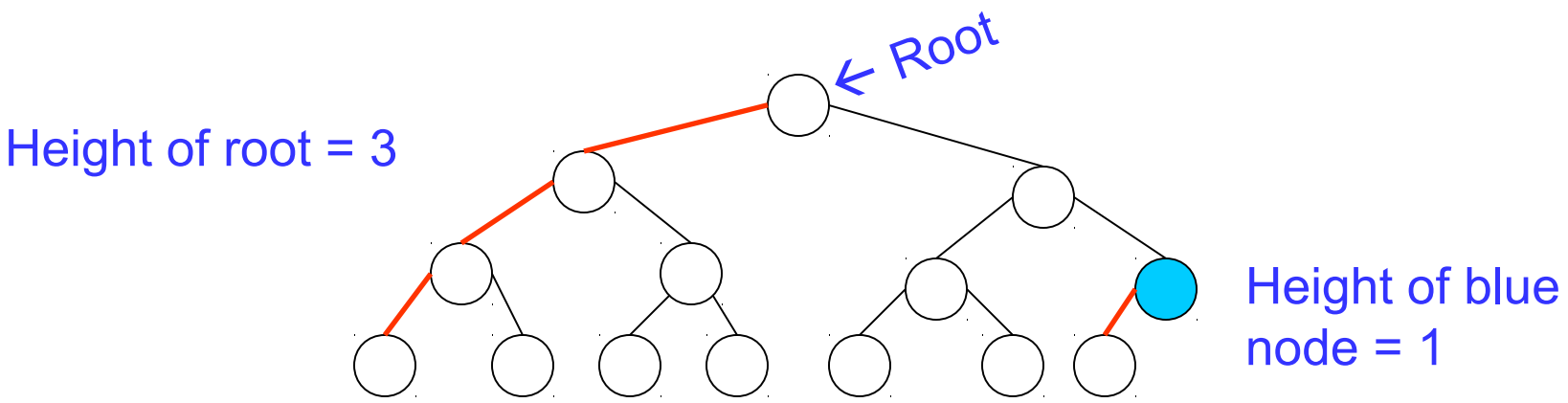
Leaf nodes are nodes without children.



Heaps

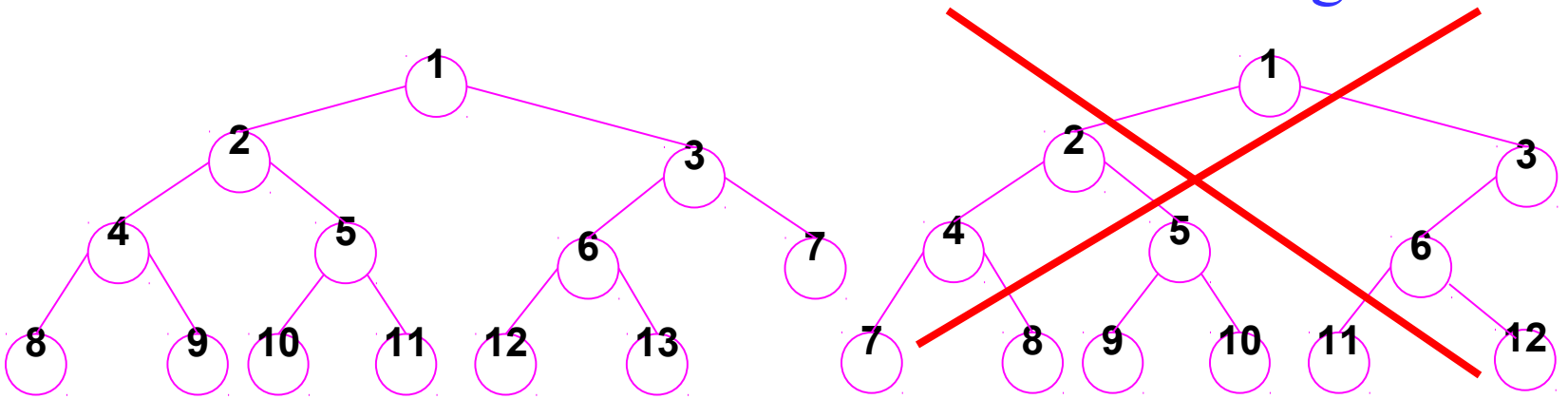
Height of a node: The number of edges starting at that node, and going down to the furthest leaf.

Height of the heap: The maximum number of edges from the root to a leaf.



Heaps

Complete binary tree– if not full, then the only unfilled level is filled in from left to right.



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

Left(i)
return 2i

Right(i)
return 2i + 1

Heaps

The heap property of a tree is a condition that must be true for the tree to be considered a heap.

Min-heap property: for min-heaps, requires

$A[parent(i)] \leq A[i]$

*So, the root of any sub-tree holds the **least** value in that sub-tree.*

Max-heap property: for max-heaps, requires

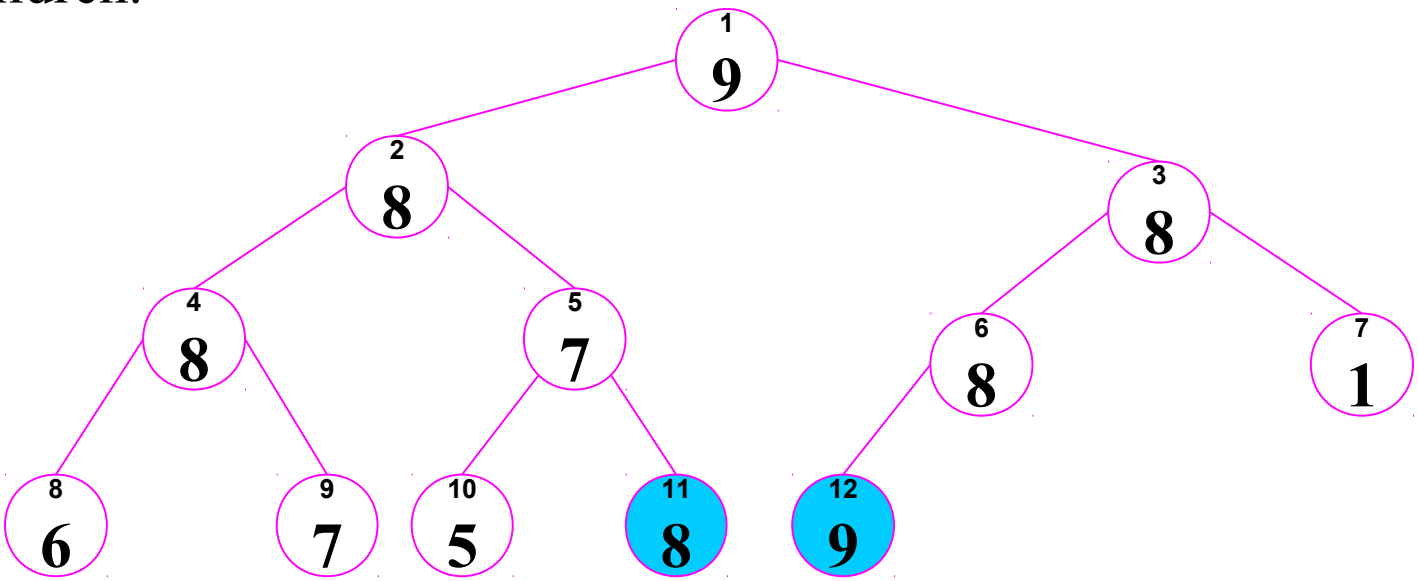
$A[parent(i)] \geq A[i]$

*The root of any sub-tree holds the **greatest** value in the sub-tree.*

Heaps

Looks like a max heap, but max-heap property is violated at indices 11 and 12. Why?

because those nodes' parents have smaller values than their children.

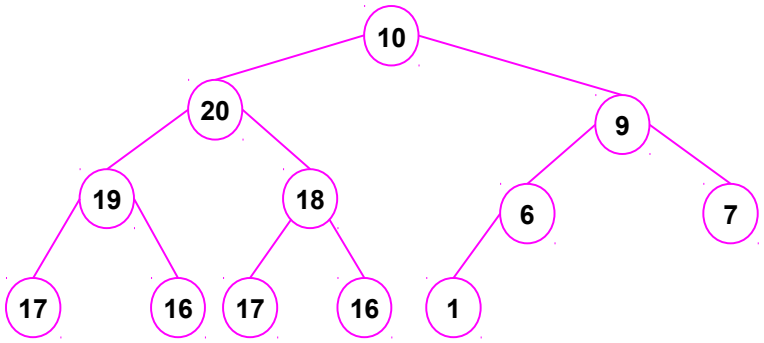


Heaps

Assume we are given a tree represented by a linear array A, and such that i ≤ length[A], and the trees rooted at Left(i) and Right(i) are heaps. Then, to create a heap rooted at A[i], use procedure Max-Heapify(A,i):

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 swap(A[i], A[largest])
   Max-Heapify(A, largest)



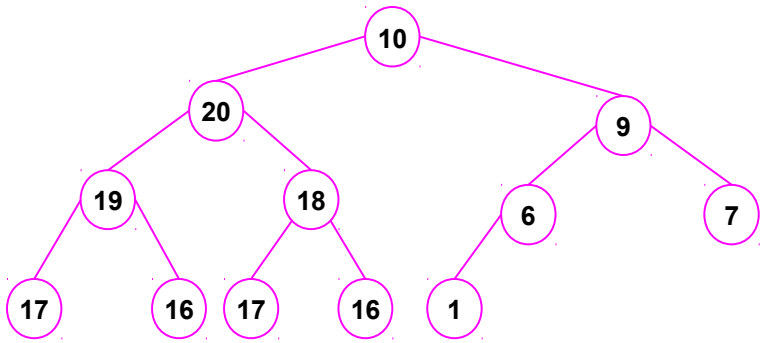
Heaps

If either **child** of A[i] is greater than A[i], the greatest child is exchanged with A[i]. So, A[i] moves down in the heap.

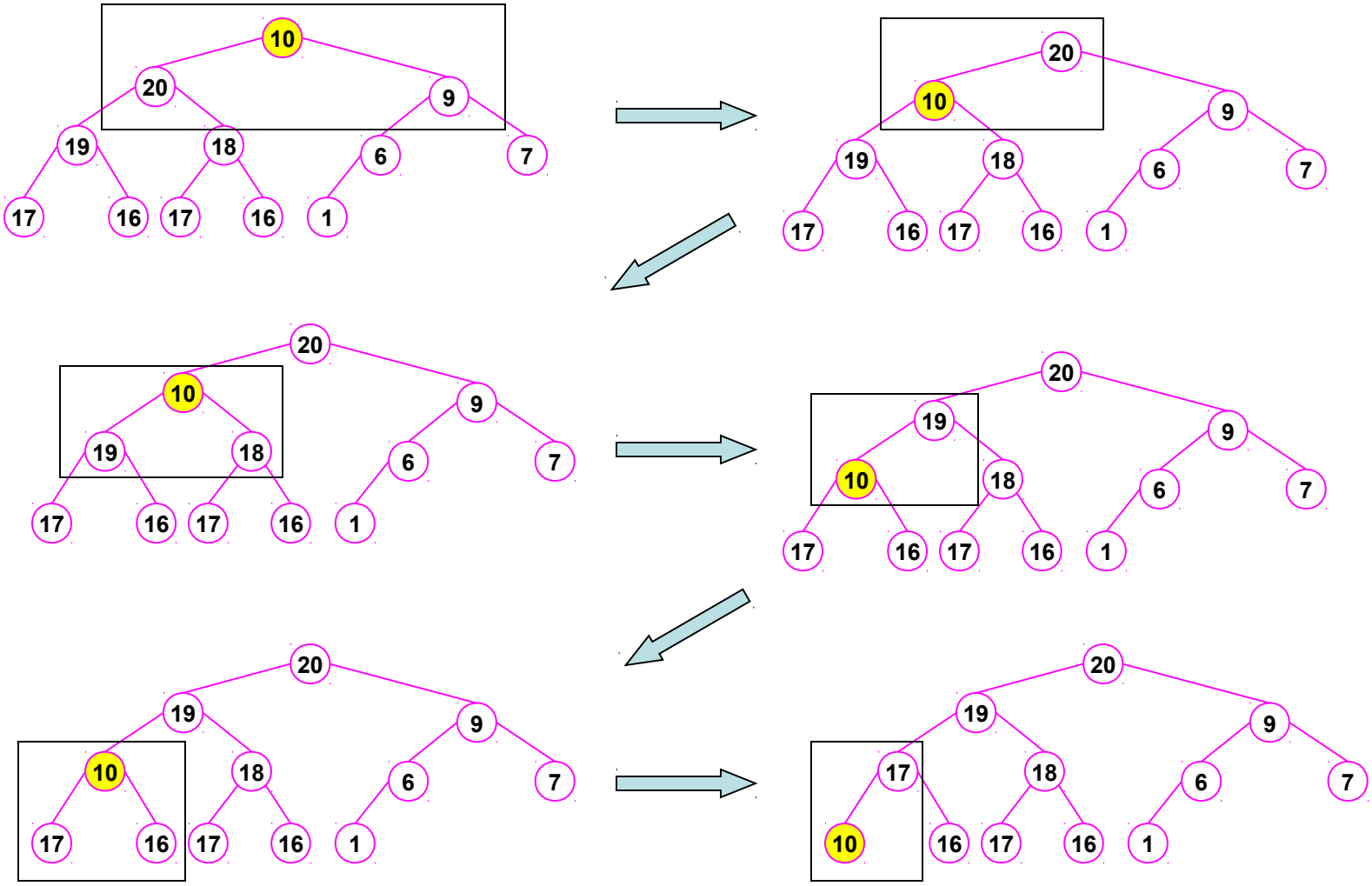
The move of A[i] may have caused a violation of the max-heap property at it's new location.

So, we must recursively call Max-Heapify(A,i) at the location i where the node "lands".

The above is the top-down approach of Max-Heapify(A,i)



Max-Heapify(A,1)



Run Time of Max-Heapify()

Run time of Max-Heapify().

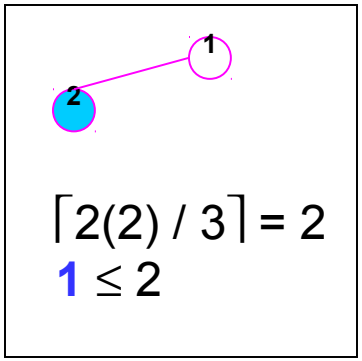
How many nodes, in terms of n, is in the sub-tree with the most nodes?

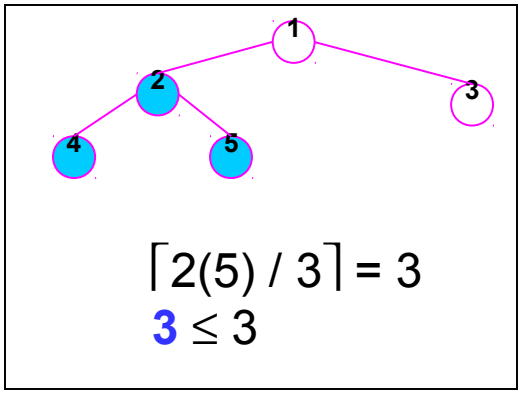
2 n / 3

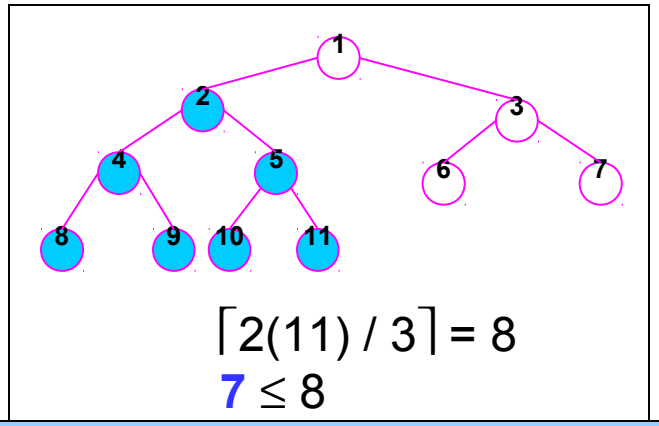
So, worst case would follow a path that included the most nodes

T(n) = T(2n/3) + Θ(1)

Why Θ(1)? Note that all the work in lines 1-7 is simple assignments or comparisons, so they are constant time, or Θ(1).







Heaps

So, what must we do to build a heap?

We call Max-Heapify(A,i) for every i starting at last node and going to the root.

Why Bottom-up?

Because Max-Heapify() moves the larger node upward into the root. If we start at the top and go to larger node indices, the highest a node can move is to the root of that sub-tree. But what if there is a violation between the sub-tree's root and its parent?

So, we must start from the bottom and go towards the root to fix the problems caused by moving larger nodes into the root of sub-trees.

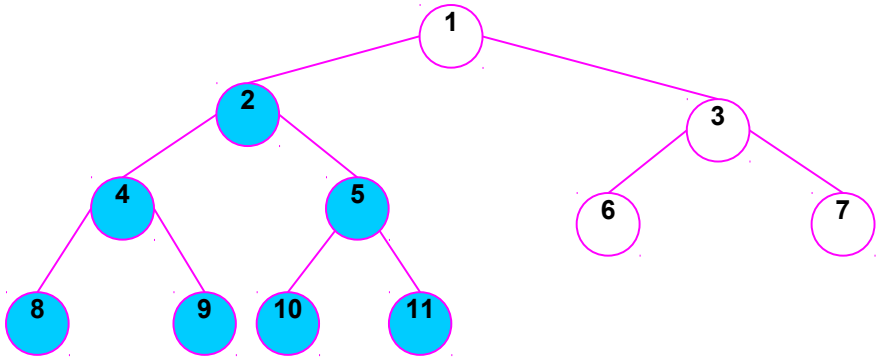
Heaps

Bad-Build-Max-Heap(A)

1. *heap-size*[A] ← *length*[A]
2. for i ← *length*[A] downto 1
3. do Max-Heapify(A,i)

Can this implementation be improved?

Sure can!



Heaps

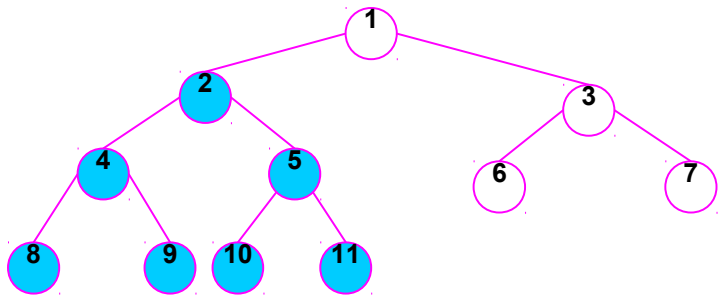
Without going through a formal proof, notice that there is no need to call Max-Heapify() on leaf nodes. At most, the internal node with the largest index is at most equal to length[A]/2.

Since this may be odd, should we use the floor or the ceiling?

In the case below, it is clear that we really need the floor of length[A]/2 which is 5, and not the ceiling, which is 6.

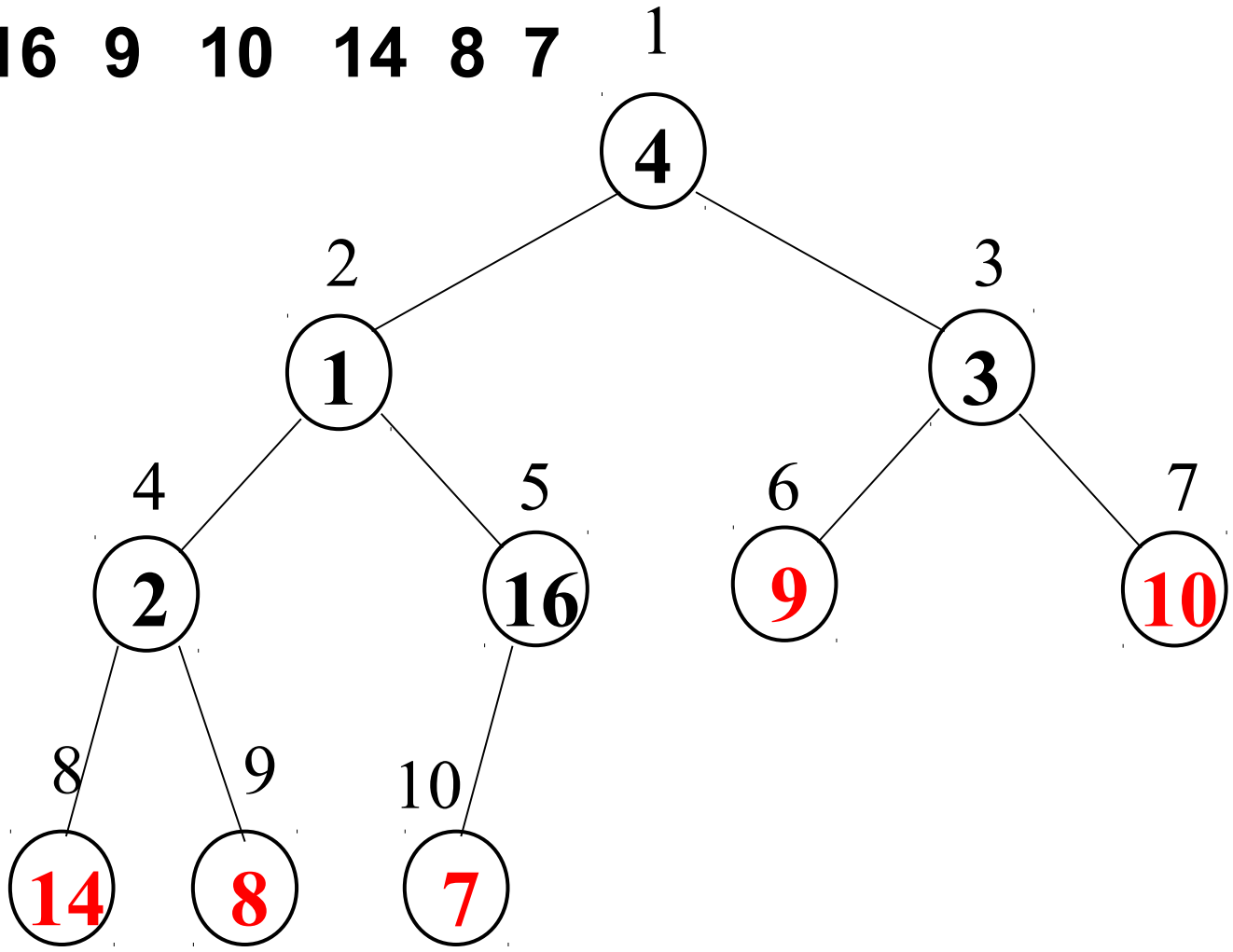
Build-Max-Heap(A)

1. *heap-size*[A] ← *length*[A]
2. for i ← ⌊*length*[A]/2⌋ downto 1
3. do Max-Heapify(A,i)

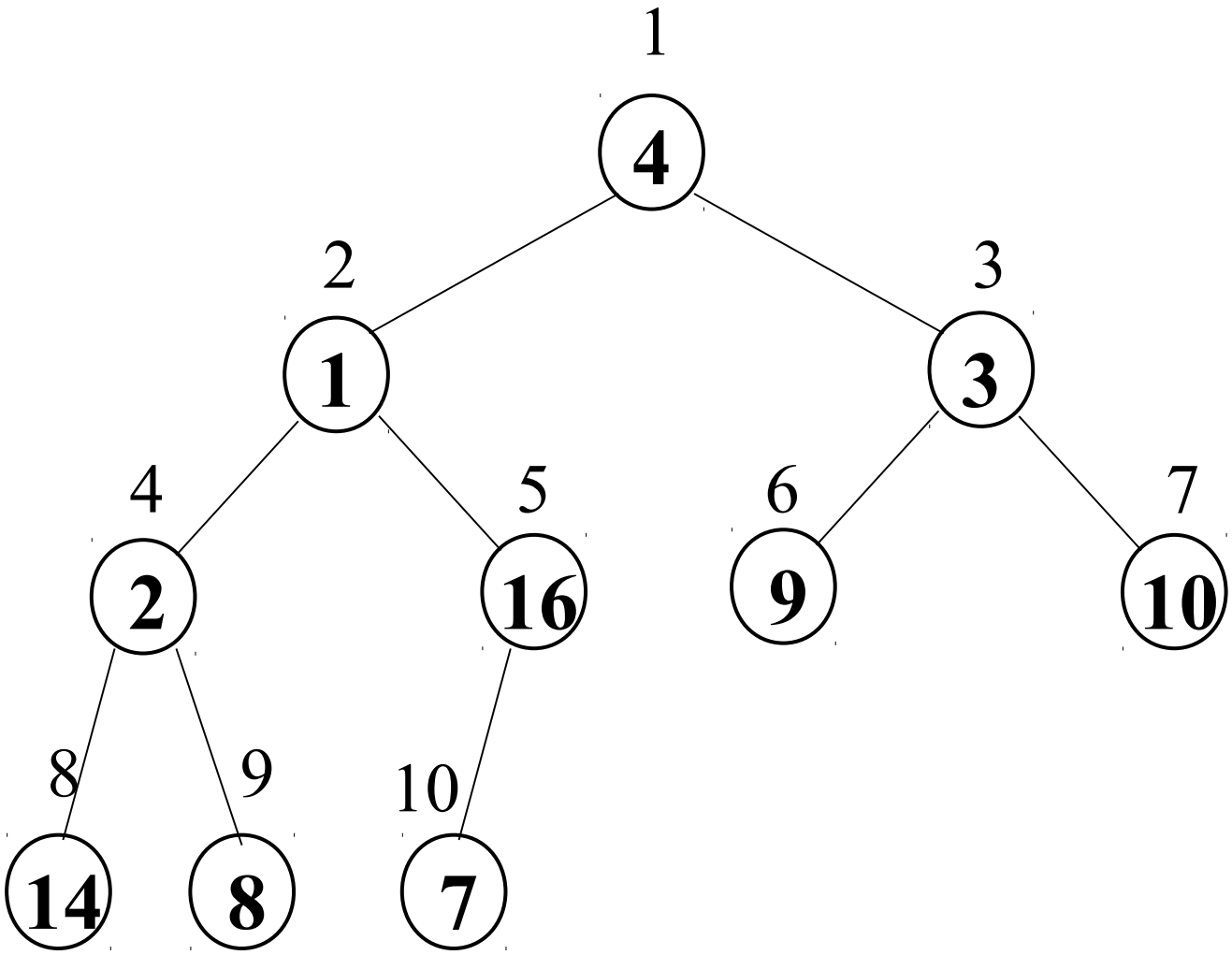


Example: building a heap (1)

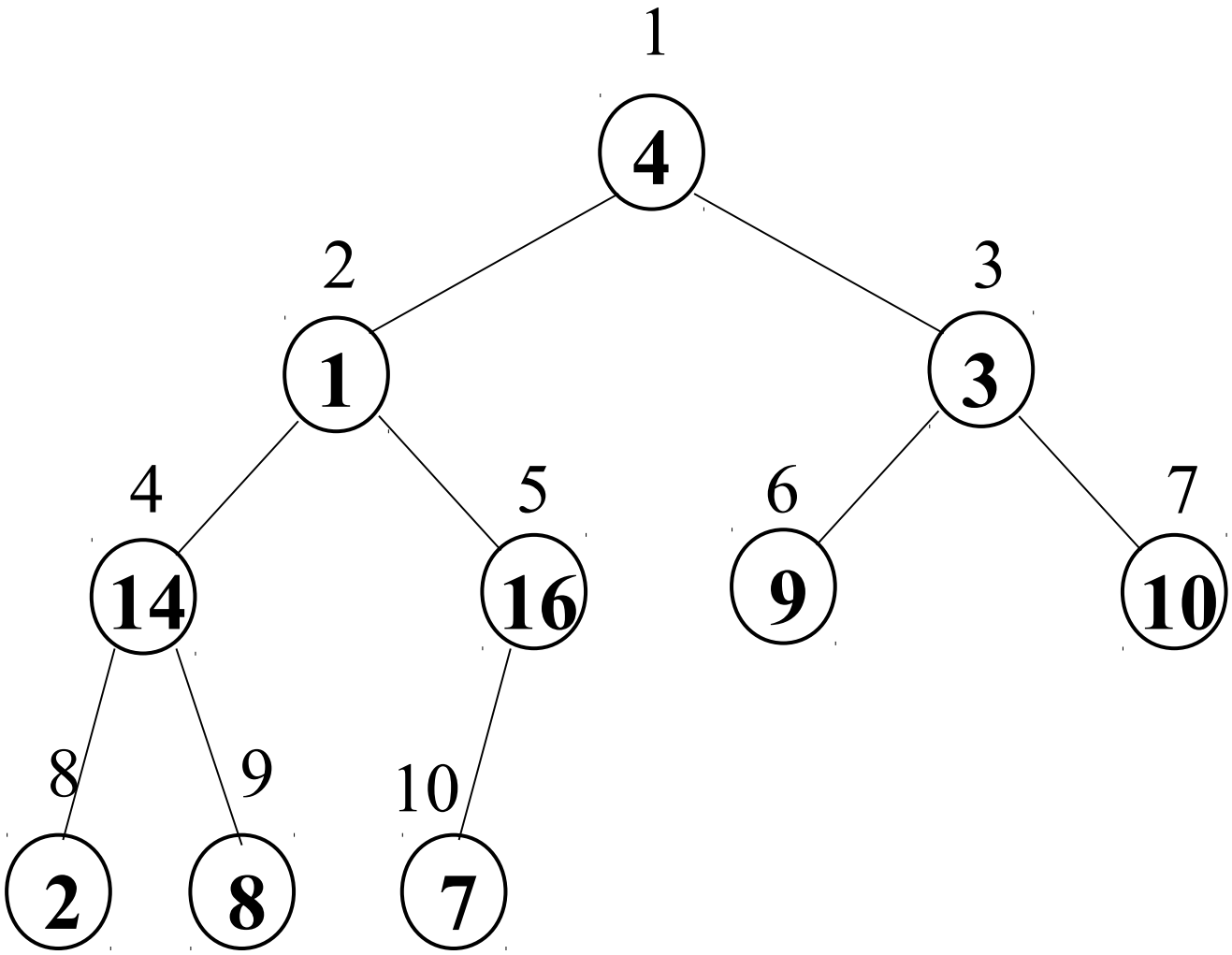
1	2	3	4	5	6	7	8	9	10
4	**1**	**3**	**2**	**16**	**9**	**10**	**14**	**8**	**7**



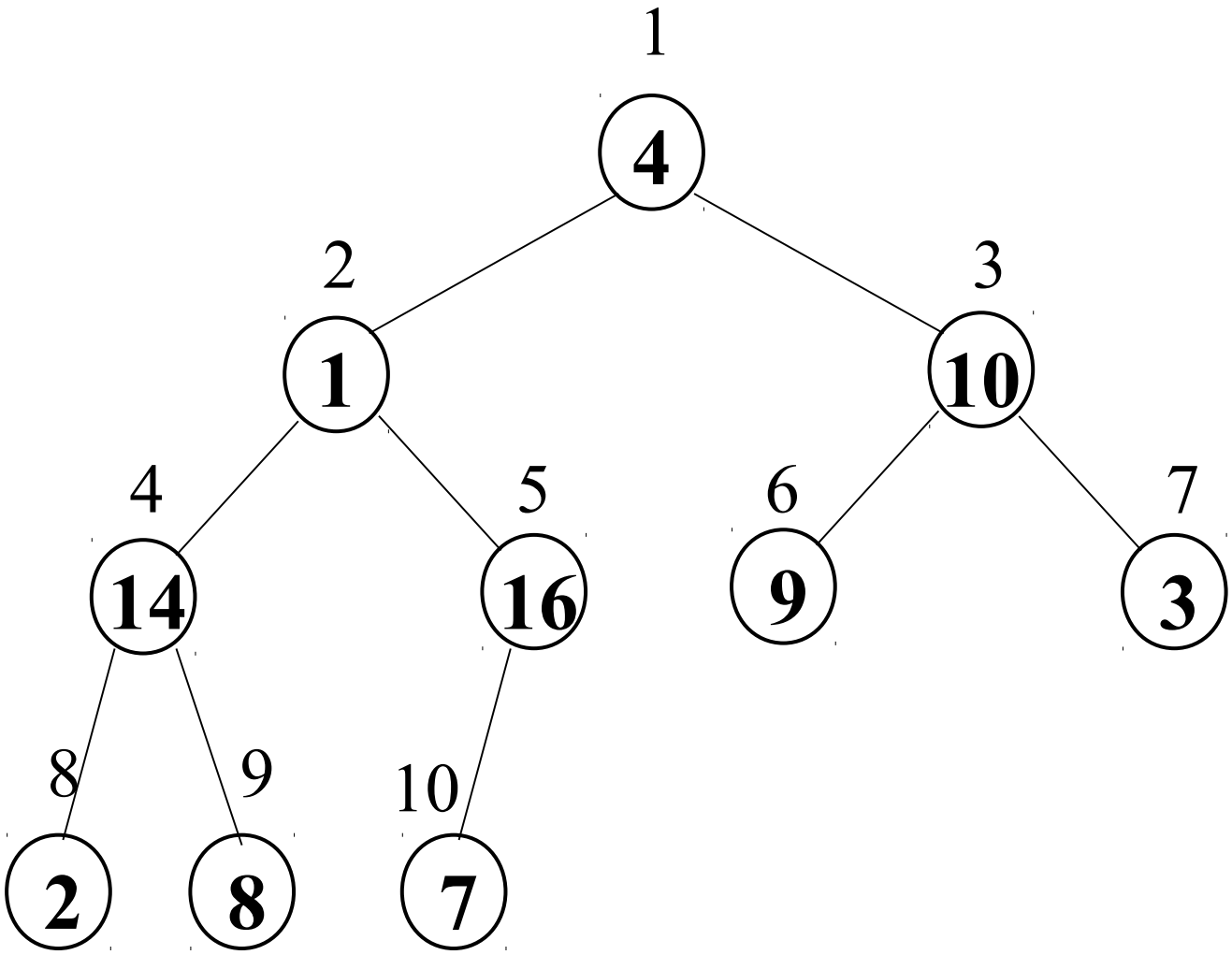
Example: building a heap (2)



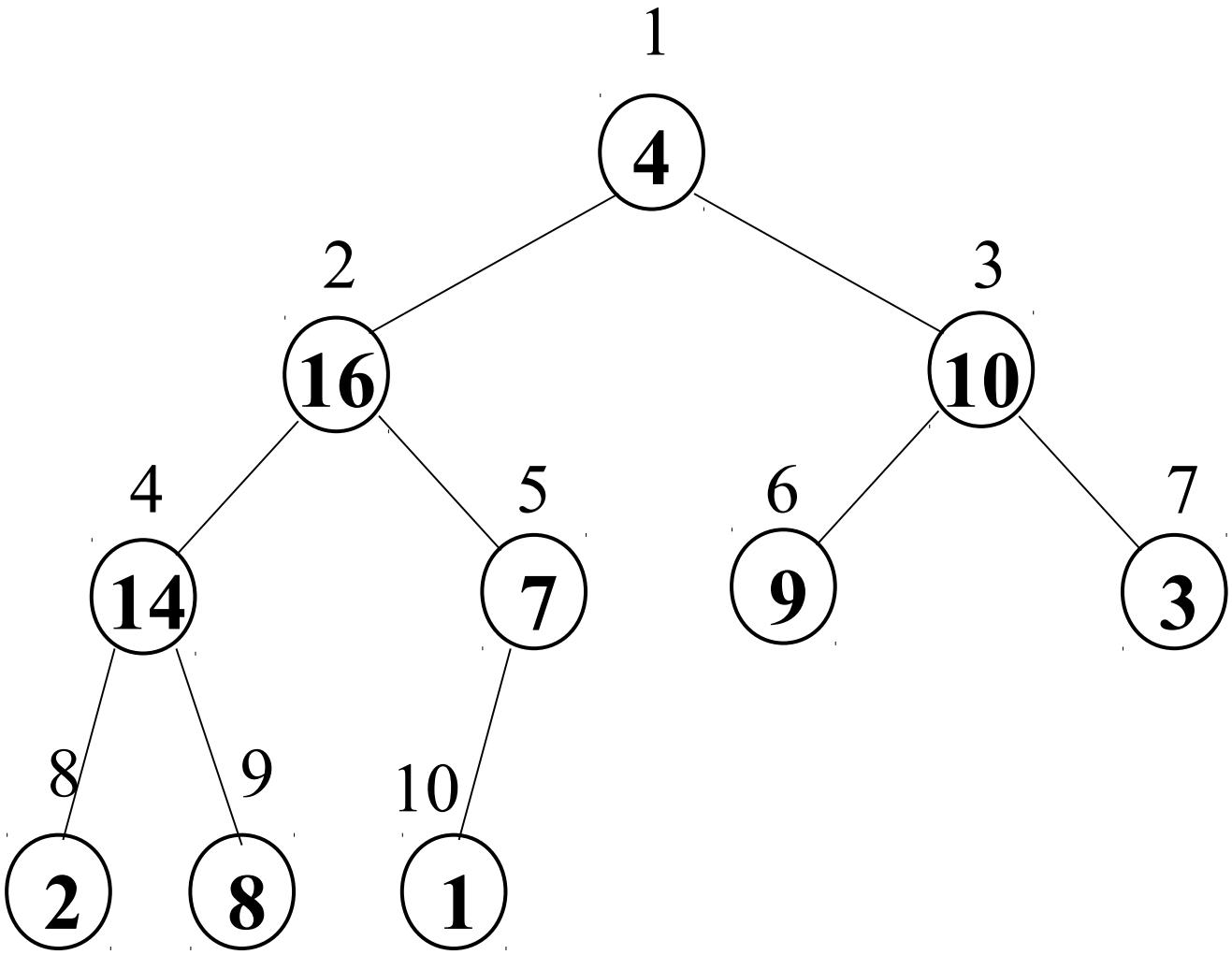
Example: building a heap (4)



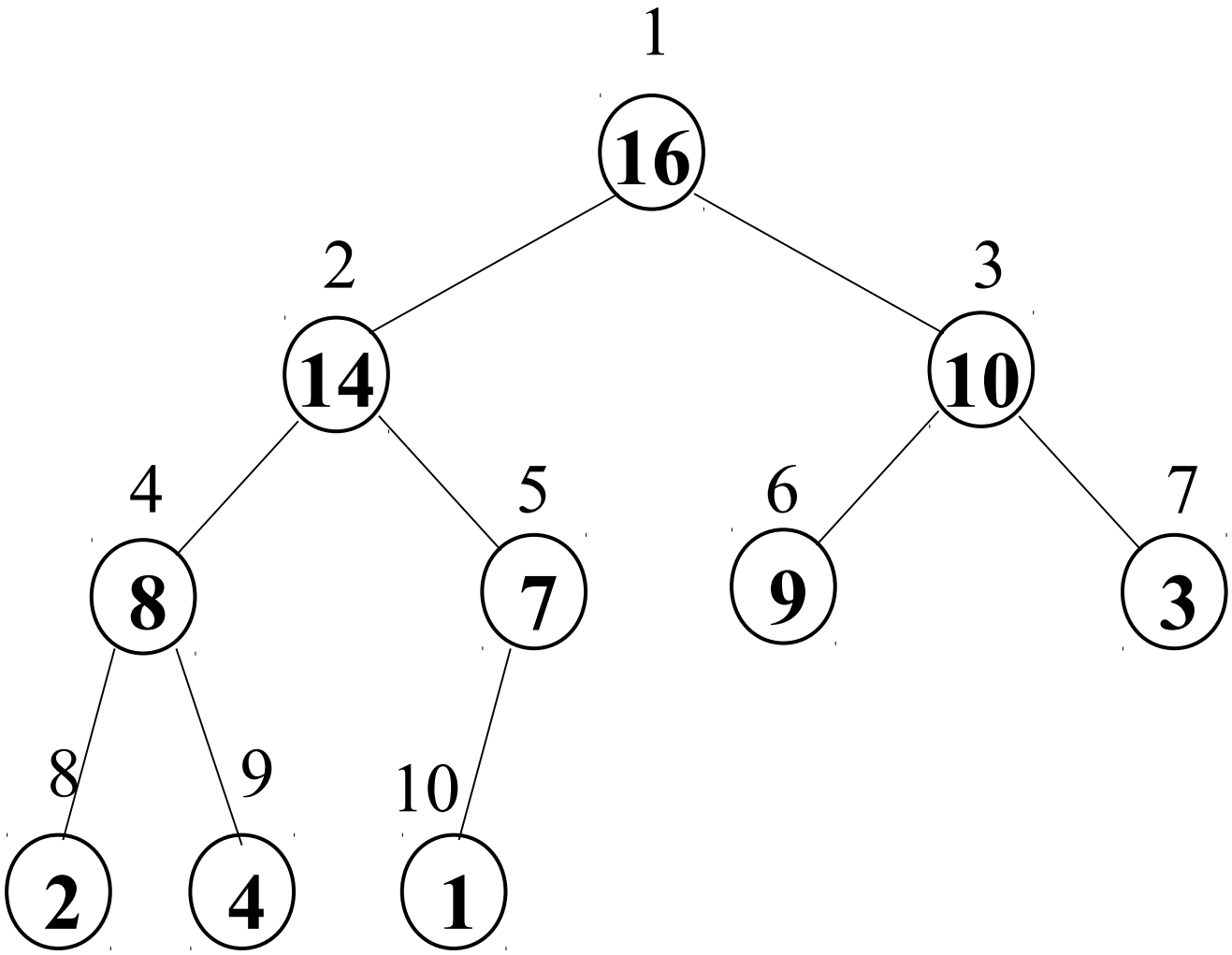
Example: building a heap (5)



Example: building a heap (6)



Example: building a heap (7)



Priority Queues

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key.

We will only consider a max-priority queue.

If we give the key a meaning, such as priority, so that elements with the highest priority have the highest value of key, then we can use the heap structure to extract the element with the highest priority.

Priority Queues

Max priority queue supports the following operations:

Max-Heap-Insert(A,key): insert key into heap, maintaining heap property

Heap-Maximum(A): returns the heap element with the largest key

Heap-Extract-Max(A): returns and removes the heap element with the largest key, and maintains the heap property

Heap-Increase-Key(A,i,key): used (at least) by
Max-Heap-Insert() to set A[i] ← A[key], and then maintaining the heap property.

Priority Queues

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 Queues

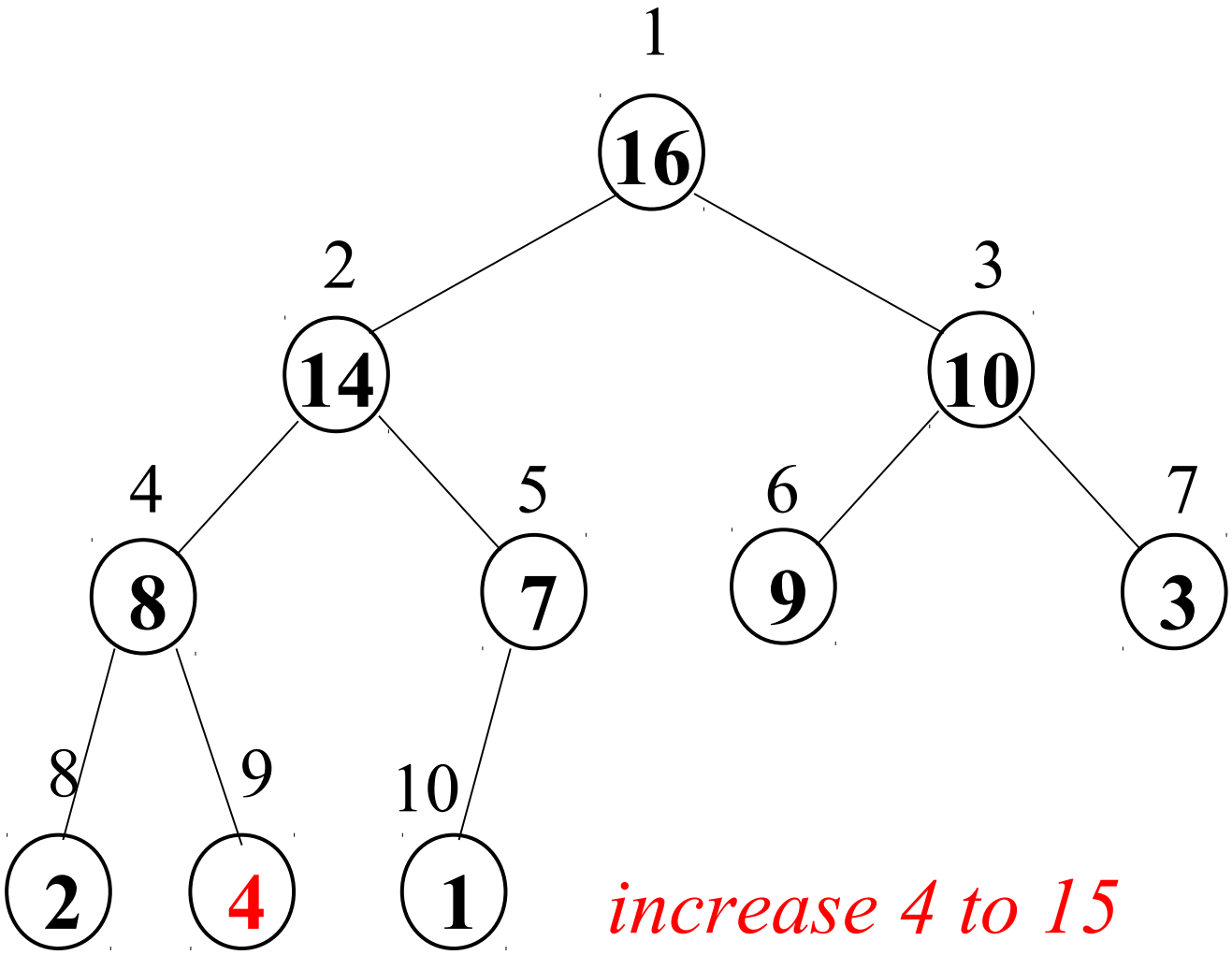
Heap-Increase-Key(A,i,key)

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)

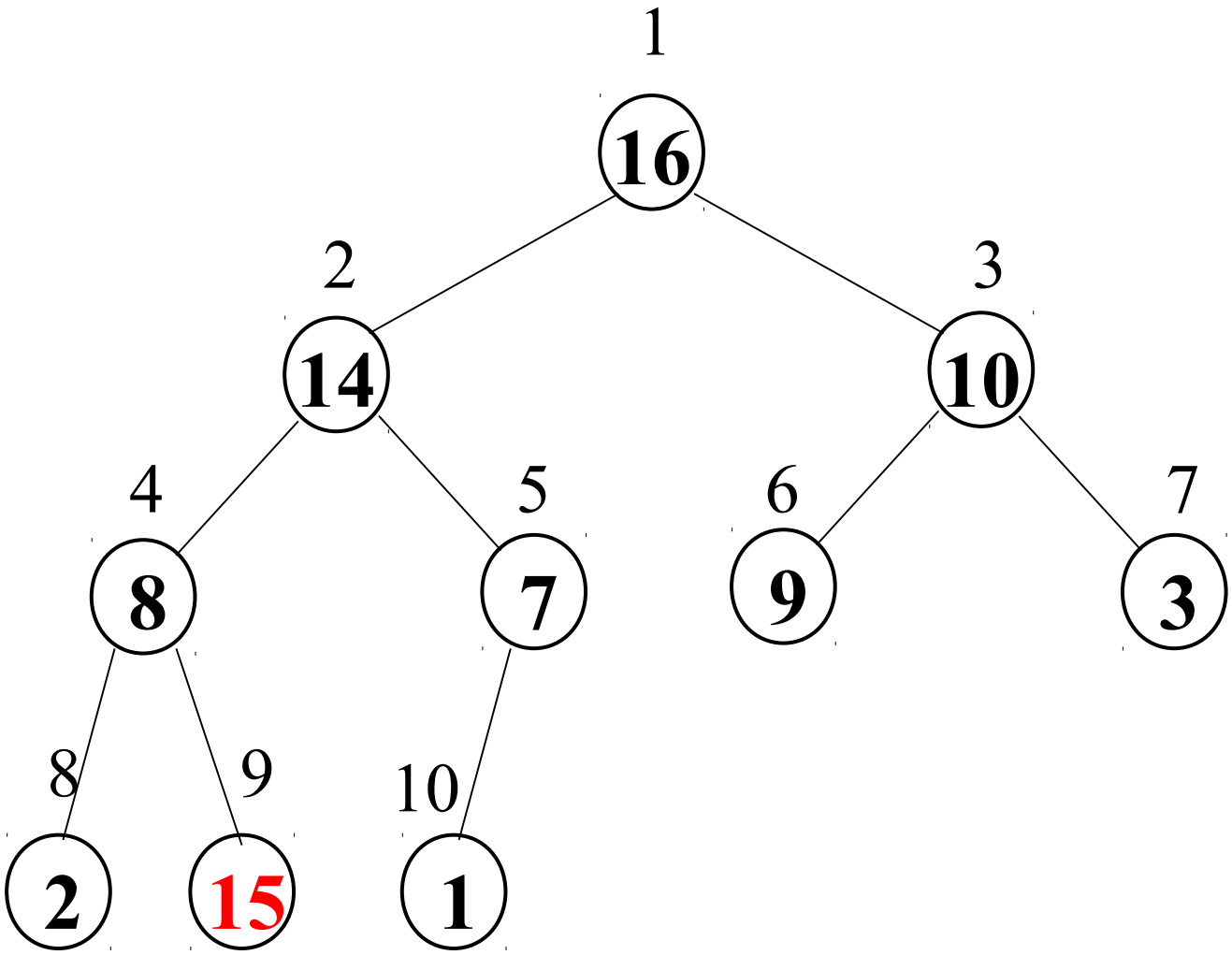
Max-Heap-Insert(A,key)

1. *heap-size*[A] ← *heap-size*[A]+1
2. A[*heap-size*[A]] ← $-\infty$
3. Heap-Increase-Key(A, *heap-size*[A], key)

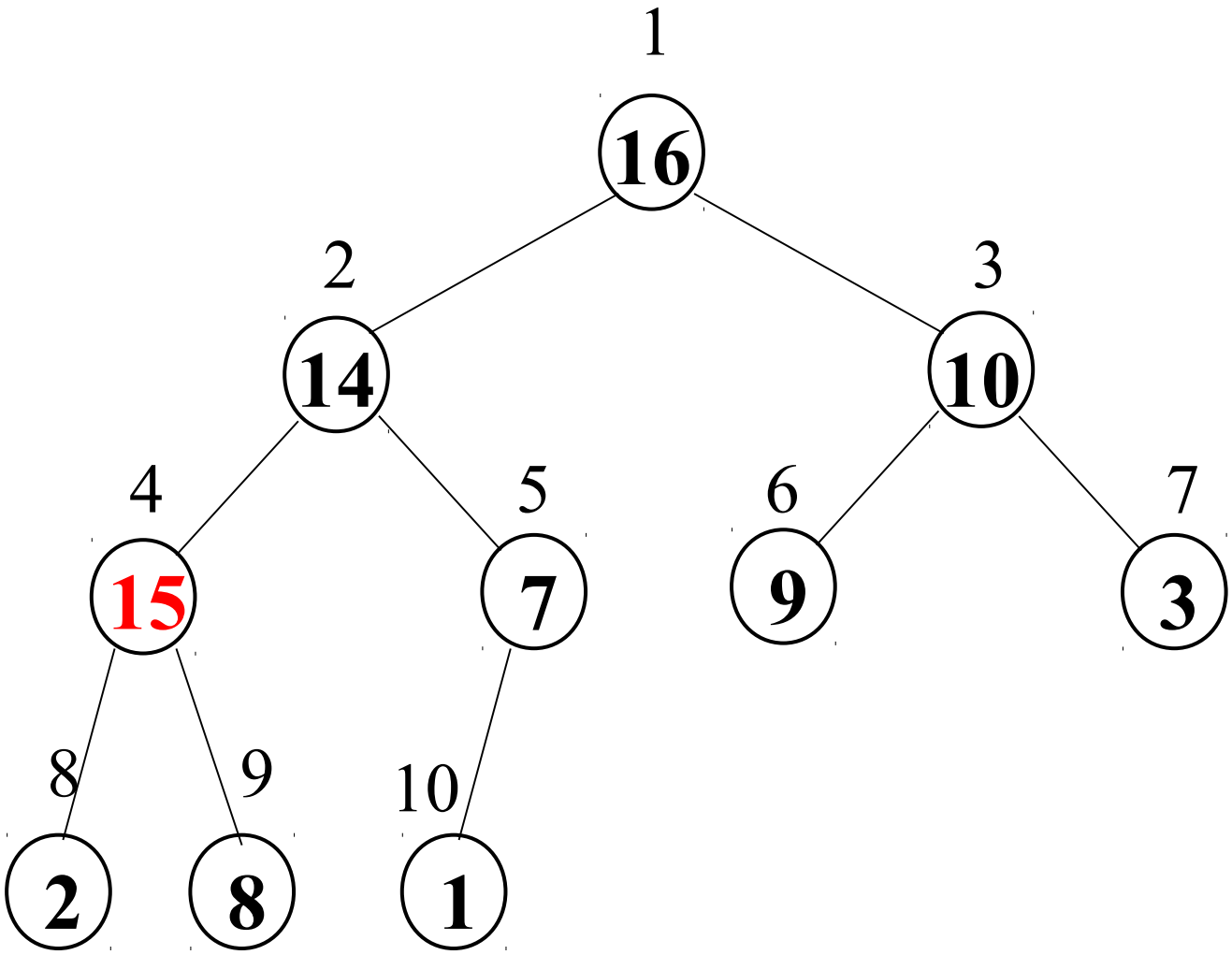
Example: increase key (1)



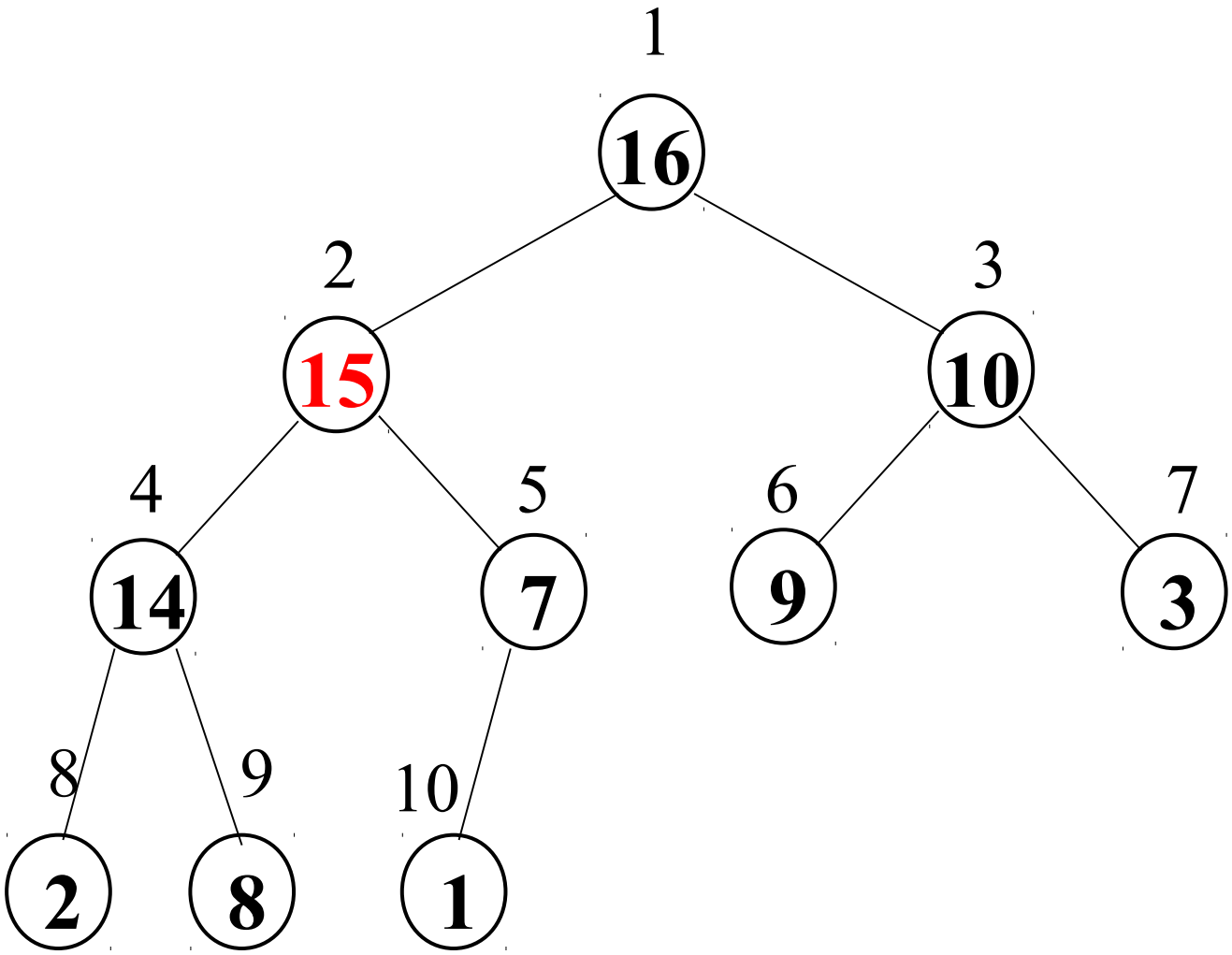
Example: increase key (2)



Example: increase key (3)



Example: increase key (4)



Heap Sort

Finally, how can we get the keys sorted in an array A? We know that a heap is not necessarily sorted as the following sequence illustrates:

A = < 100, 50, 25, 40, 30 >

Use the algorithm Heapsort():

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)

Build-Max-Heap() is O(n), Max-Heapify() is O(lg n), but is executed n times, so runtime for Heapsort(A) is O(nlgn).

Heapsort

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)

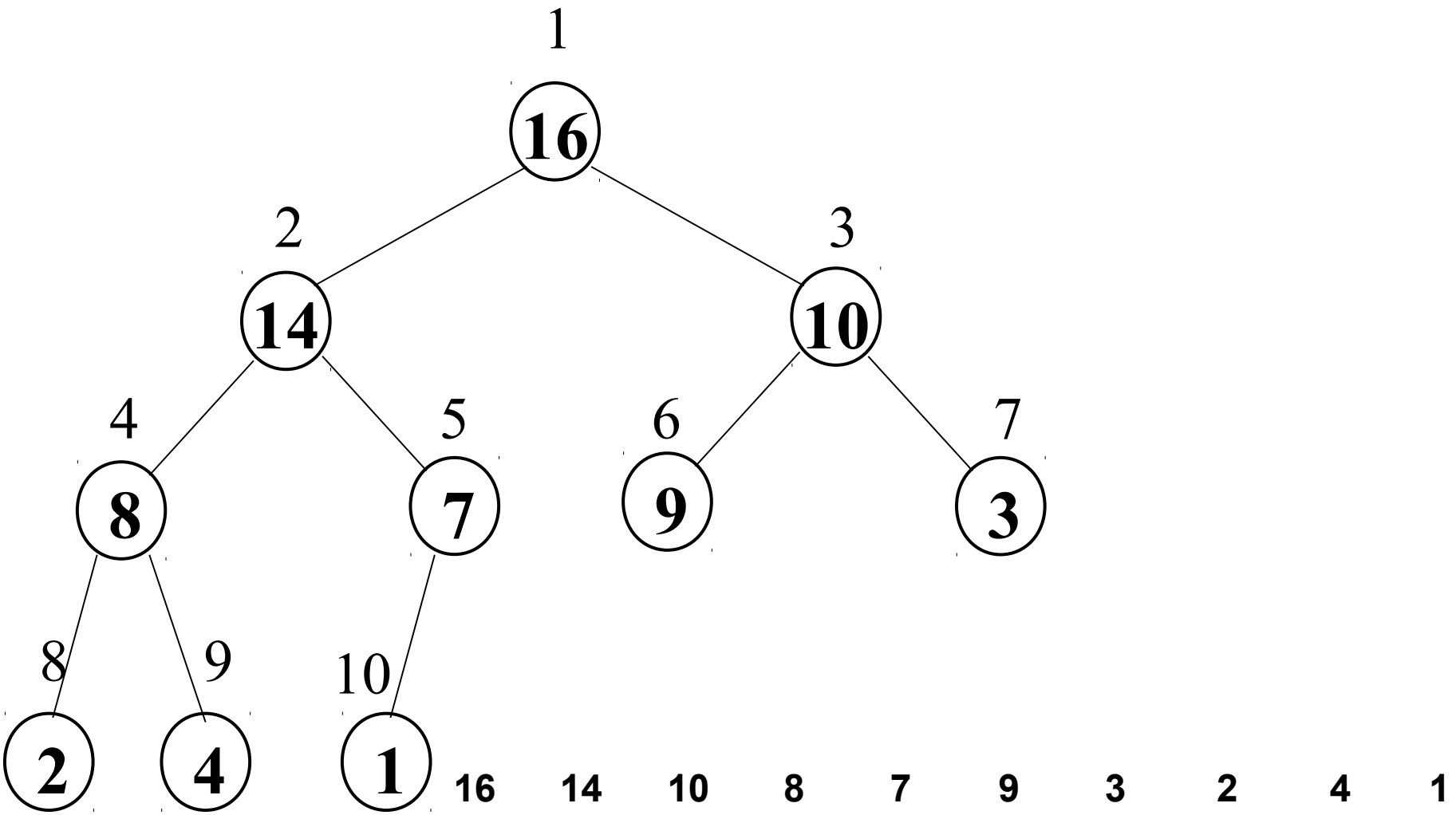
On line 1, we create the heap. Keep in mind that the largest element is at the root. So, why not put the element that is currently in the root at the last index of A[]?

We will exchange the last element in A, based on the value of heap-size[A], with A[1], as per line 3.

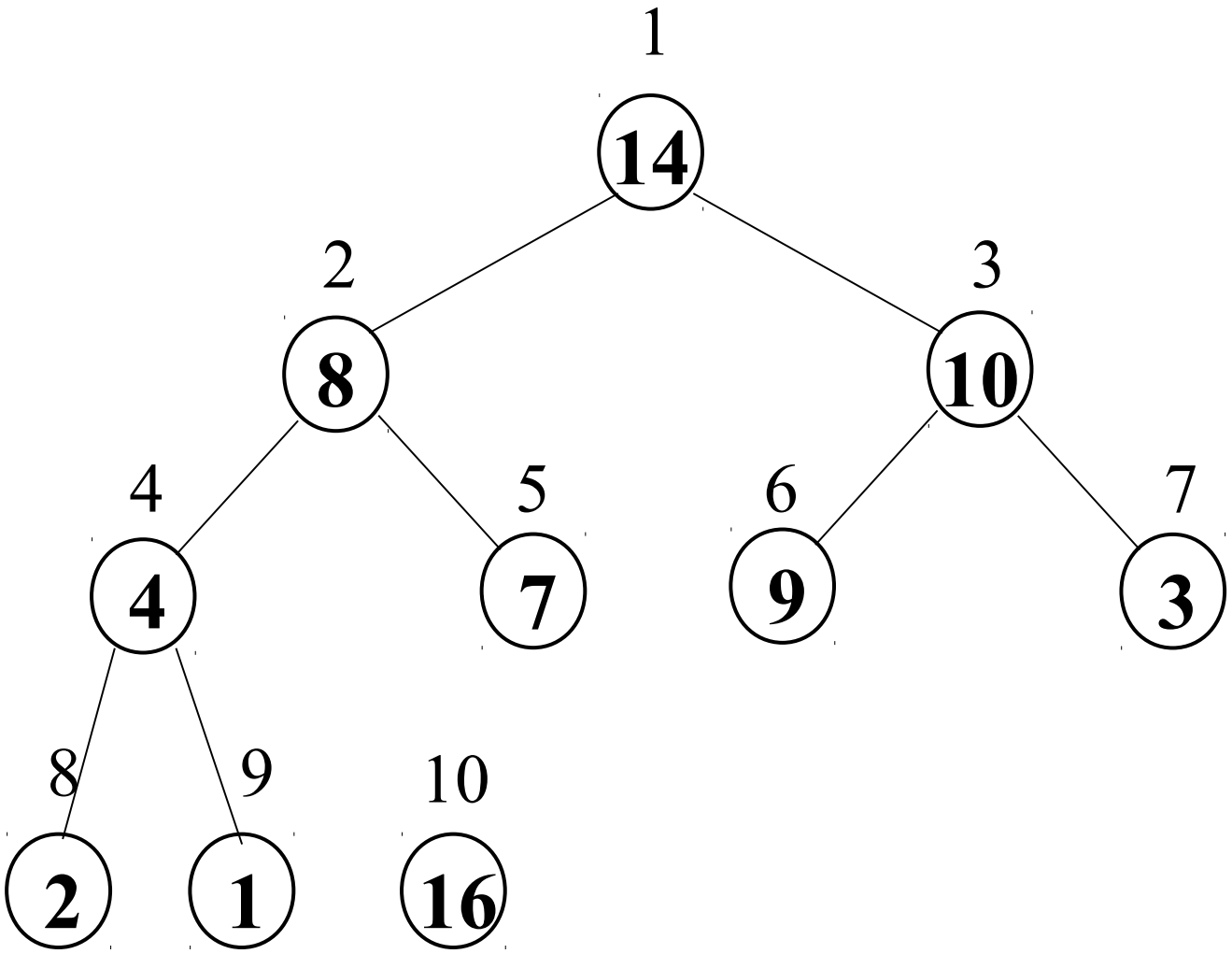
Then we will reduce heap-size[A] by one so that we can make sure that putting A[heap-size] into A[1] from line 3 doesn't violate the heap property. But we don't want to touch the max element, so that's why heap-size is reduced.

Continue in this fashion until i=2. Why don't we care about i=1? It's already done

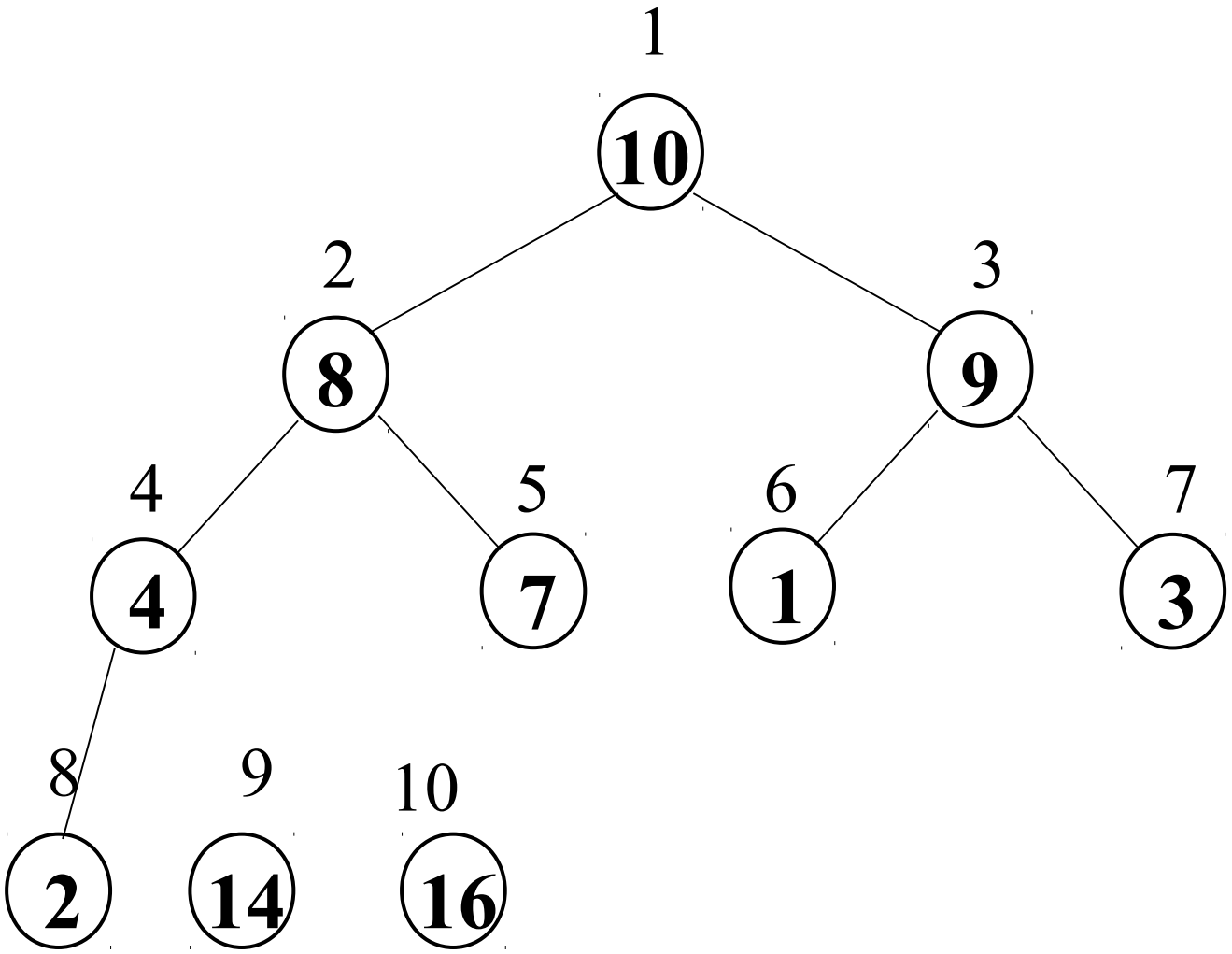
Example: Heap-Sort



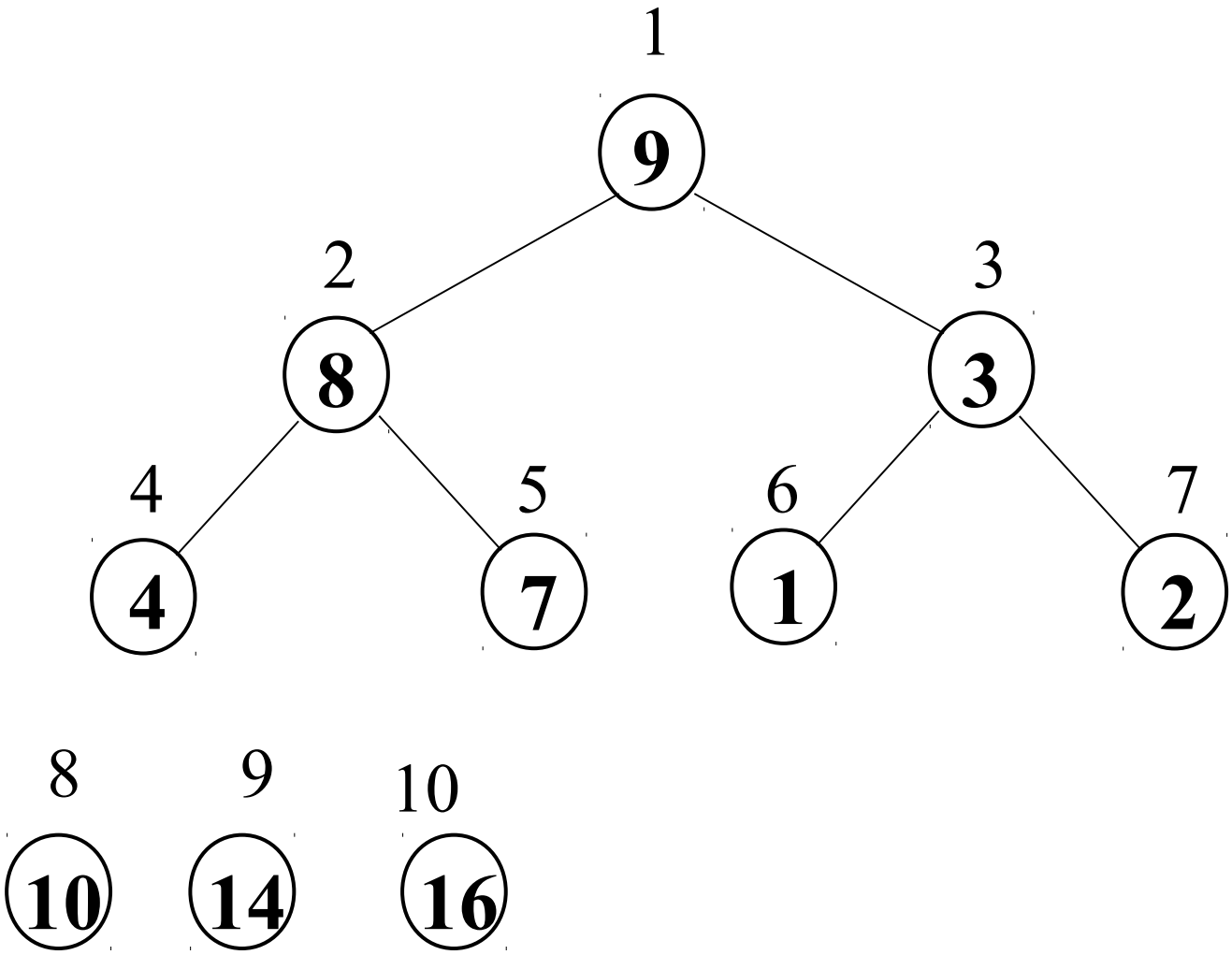
Example: Heap-Sort (2)



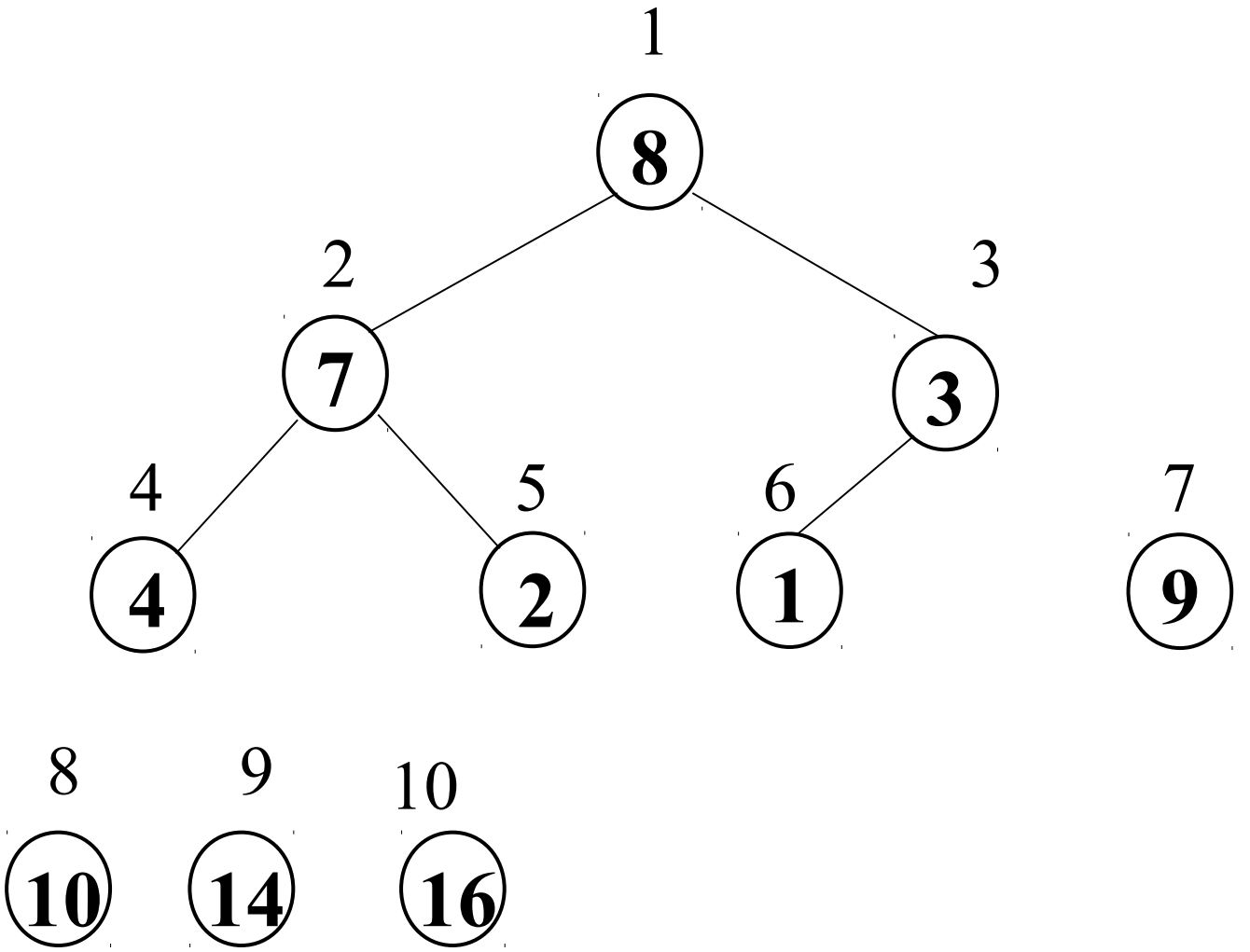
Example: Heap-Sort (3)



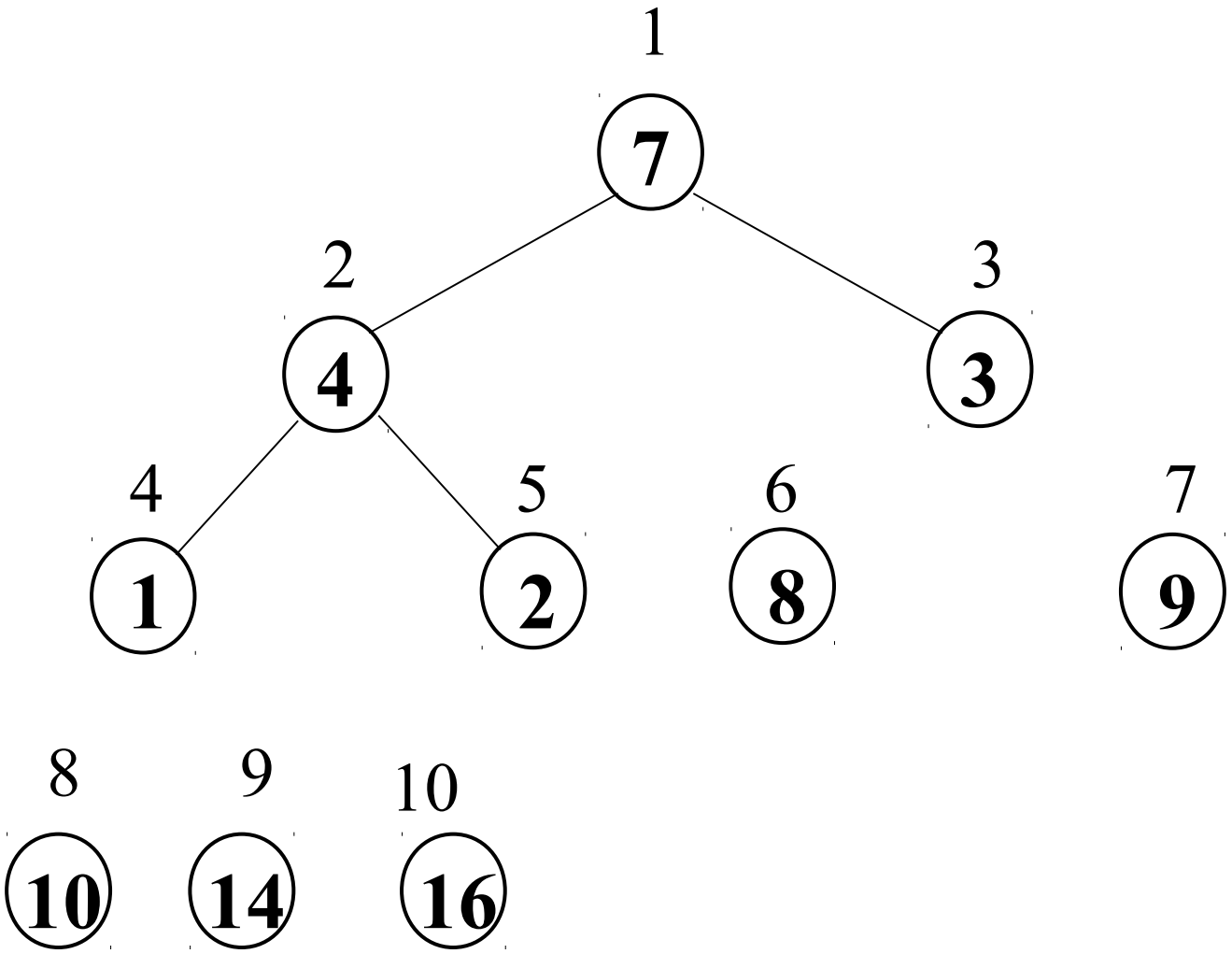
Example: Heap-Sort (4)



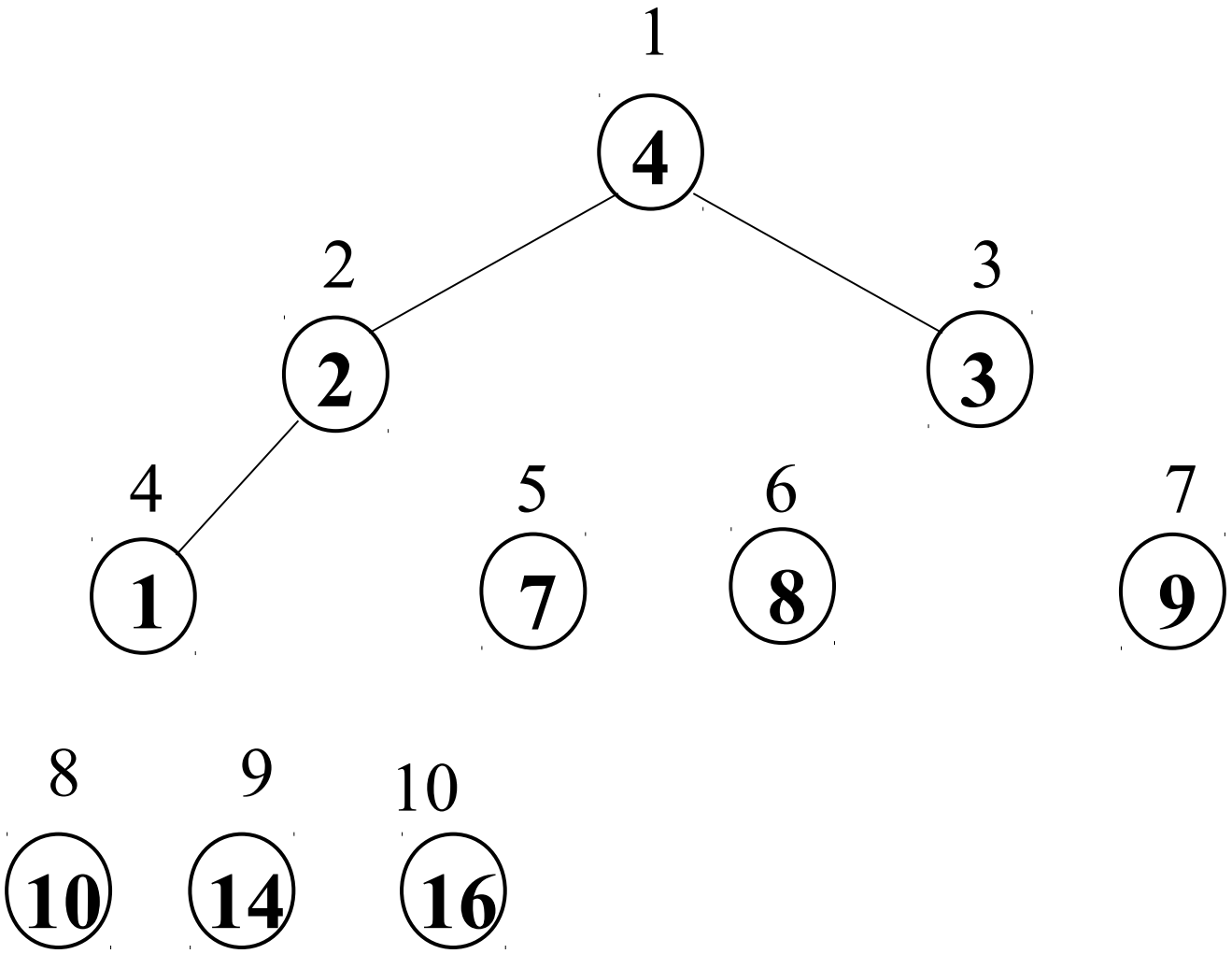
Example: Heap-Sort (4)



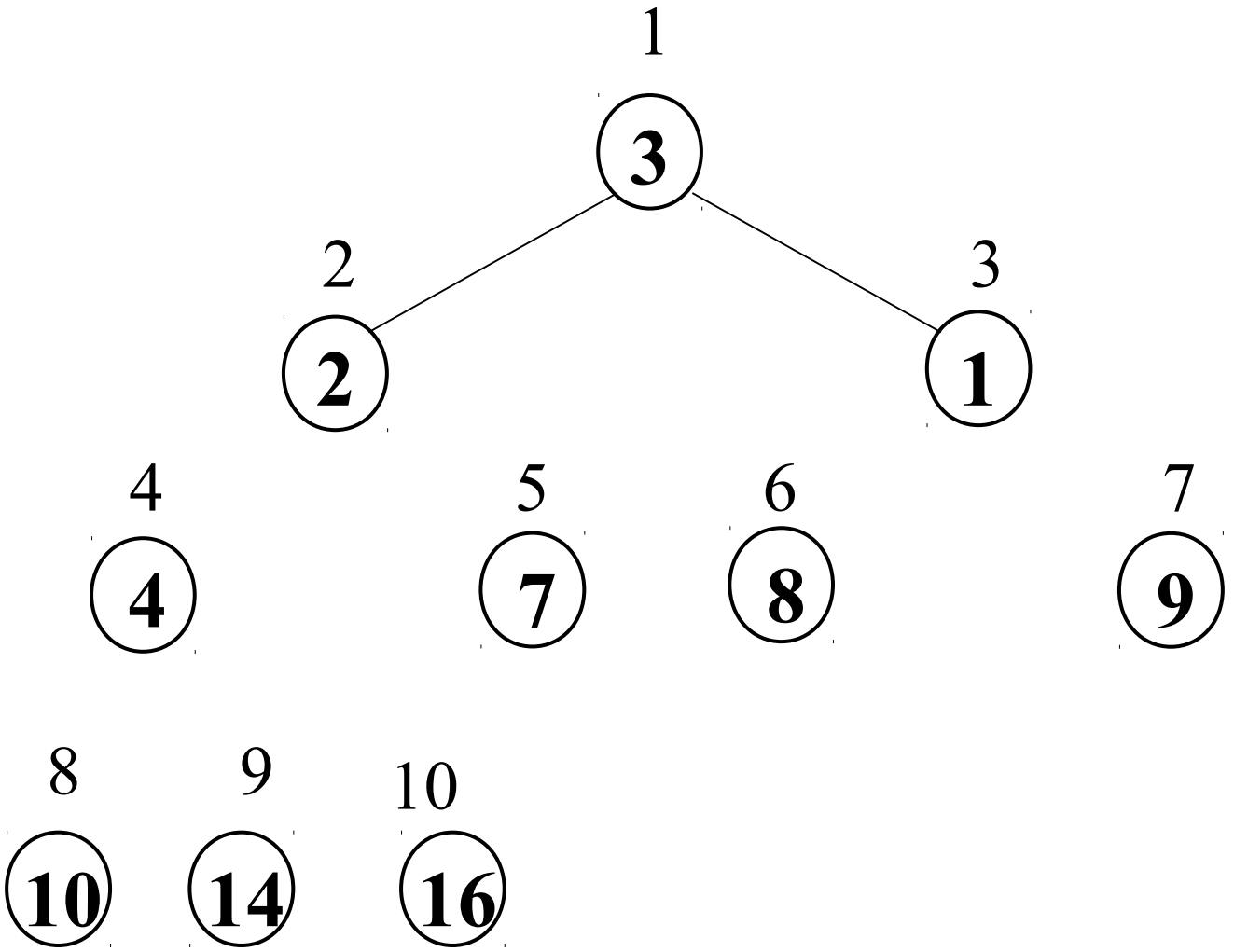
Example: Heap-Sort (5)



Example: Heap-Sort (6)



Example: Heap-Sort (7)



Example: Heap-Sort (8)