# Design and Analysis of Algorithm (DAA)

## Heap sort and Priority Queue

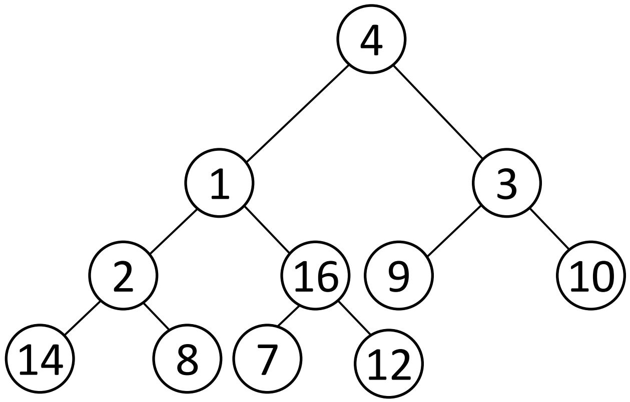
**[Module 2]**

Dr. Dayal Kumar Behera

School of Computer Engineering
KIIT Deemed to be University, Bhubaneswar, India

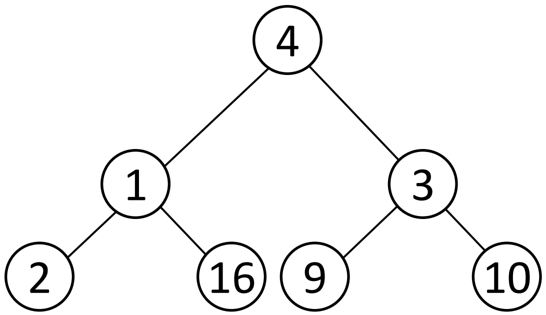
# Special Types of Trees

- **Full binary tree:** a binary tree in which each node is either a leaf or has degree exactly 2.




Full binary tree

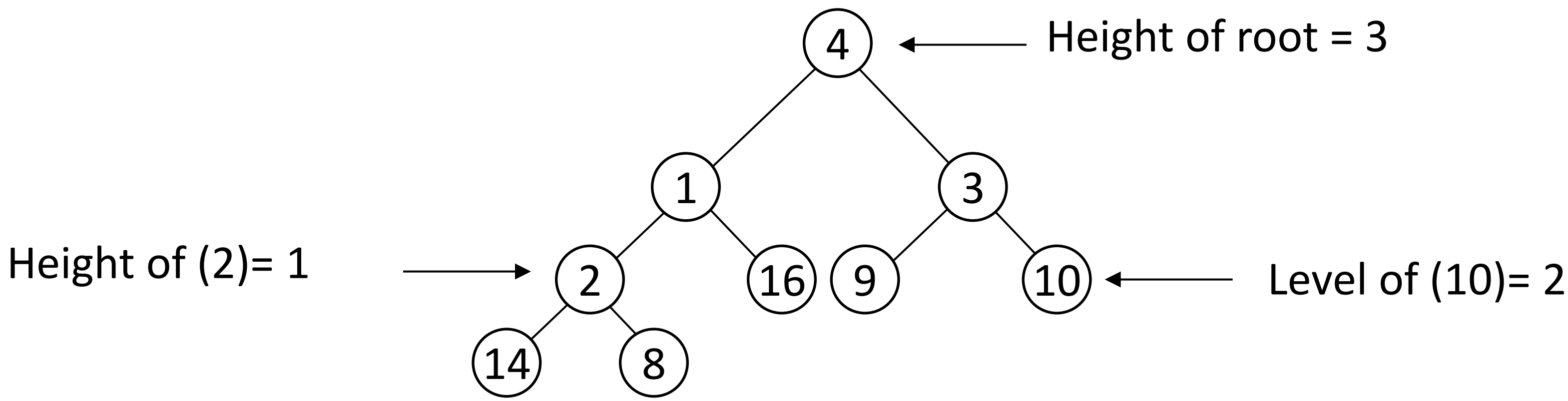
- **Complete binary tree:** a binary tree in which all leaves are on the same level and all internal nodes have degree 2.




Complete binary tree

# Definition

- **Height** of a node = the number of edges on the longest simple path from the node down to a leaf
- **Level** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node

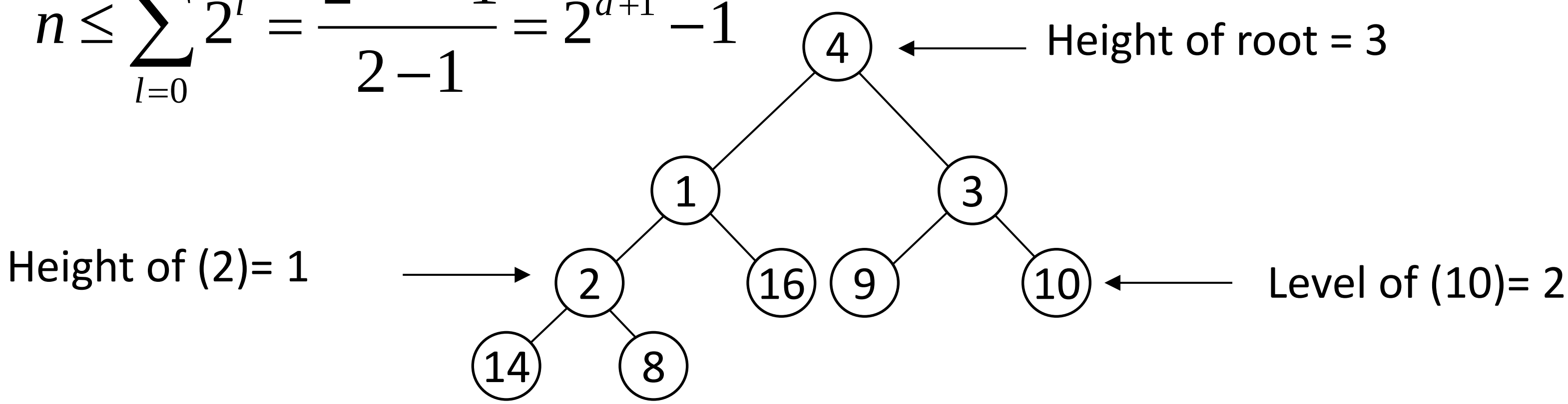



The depth of a node is its distance from the root. E.g. Level or depth of (10) = 2

# Useful Properties

- There are **at most** $2^l$ nodes at level (or depth) $l$ of a binary tree
- A binary tree with height $d$ has **at most** $2^{d+1}-1$ nodes
- A binary tree with $n$ nodes has height **at least** $\lfloor lgn \rfloor$

$$n \le \sum_{l=0}^{d} 2^l = \frac{2^{d+1}-1}{2-1} = 2^{d+1}-1$$

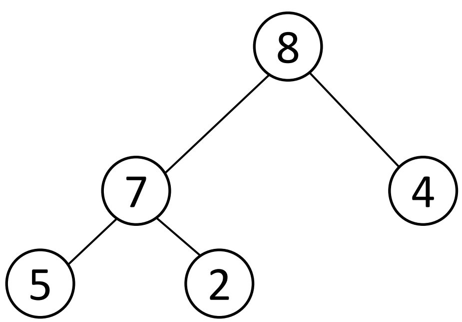



Note# height and depth of tree are same, whereas for node it may differ.

# The Heap Data Structure

- *Definition:* A **heap** is a <u>nearly or almost complete</u> binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node $x$

$$A[parent(x)] \geq A[x]$$ ← Max-heap




Heap

From the heap property, it follows that:
"The root is the maximum element of the heap!"

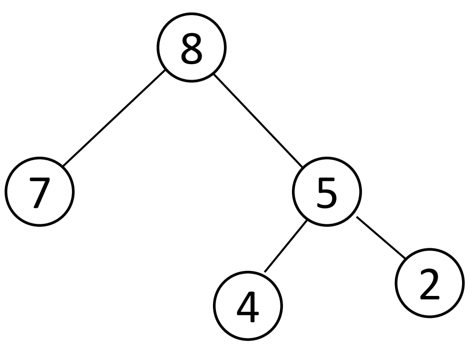
A heap is a binary tree that satisfies the heap properties.

# Types of Binary Heap

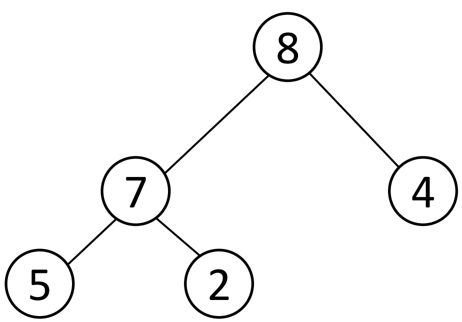
- *Two types* : Max-heap and Min-heap

  - **Max-heap** (largest element at root)**:** for every node, value of parent node is greater than or equals to the value of child nodes.
  - **Max-heap property:** for any node $x$

$$A[parent(x)] \geq A[x] \quad \leftarrow \text{Max-heap}$$




Not a Heap




Max-Heap

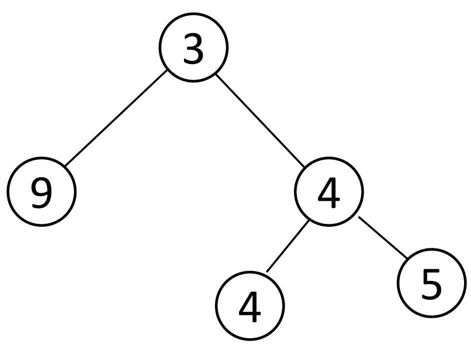
# Types of Binary Heap

- *Two types* : Max-heap and Min-heap

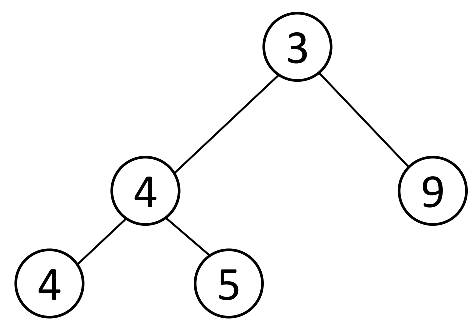
  – **Min-heap** (Smallest element at root)**:** for every node, value of parent node is less than or equals to the value of child nodes.

  – **Min-heap property:** for any node $x$

$$A[\text{parent}(x)] \le A[x] \quad \leftarrow \text{Min-heap}$$



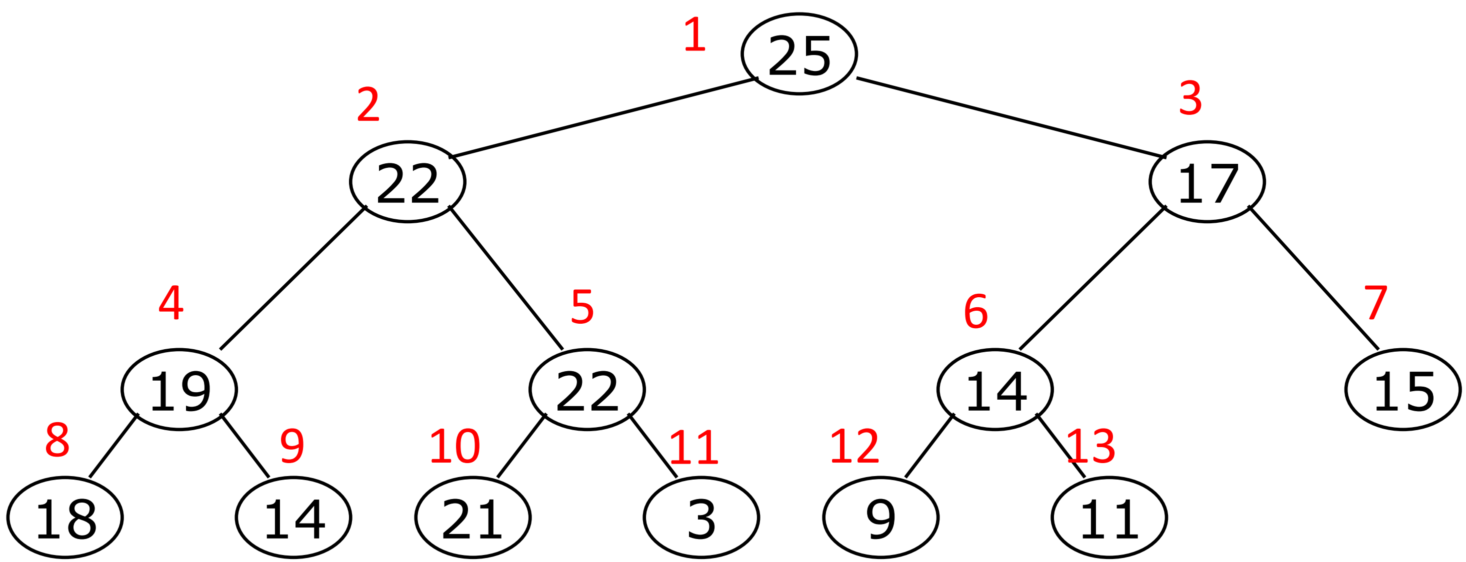

Not a Heap




Min-Heap

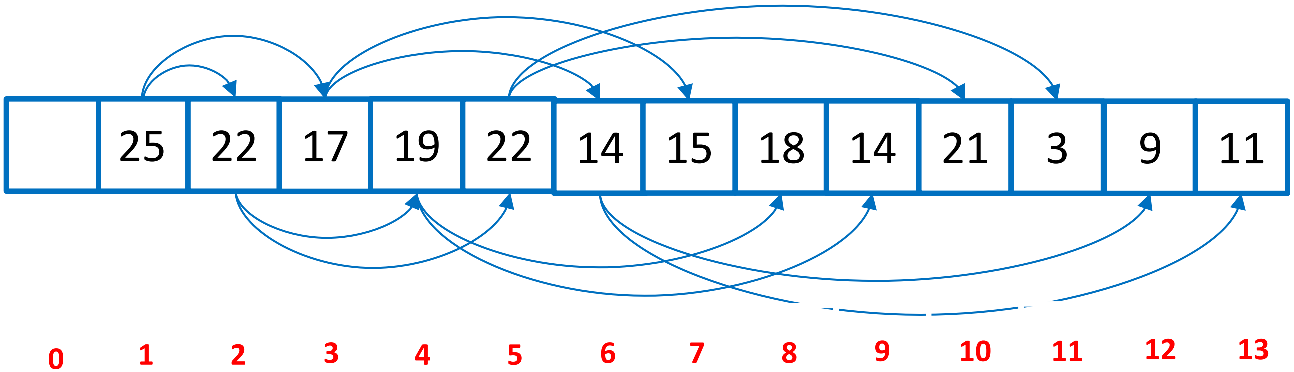
# Array Representation of Binary Heap

For simplicity heap is represented as an array




| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **25** | **22** | **17** | **19** | **22** | **14** | **15** | **18** | **14** | **21** | **3** | **9** | **11** |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |

# Array Representation of Heaps




Parent of i => $\lfloor i/2 \rfloor$

Left of i => 2 i

Right of i => 2 i +1

MAX-Heap

A(PARENT(i)) >= A(i)

MIN-Heap

A(PARENT(i)) <= A(i)

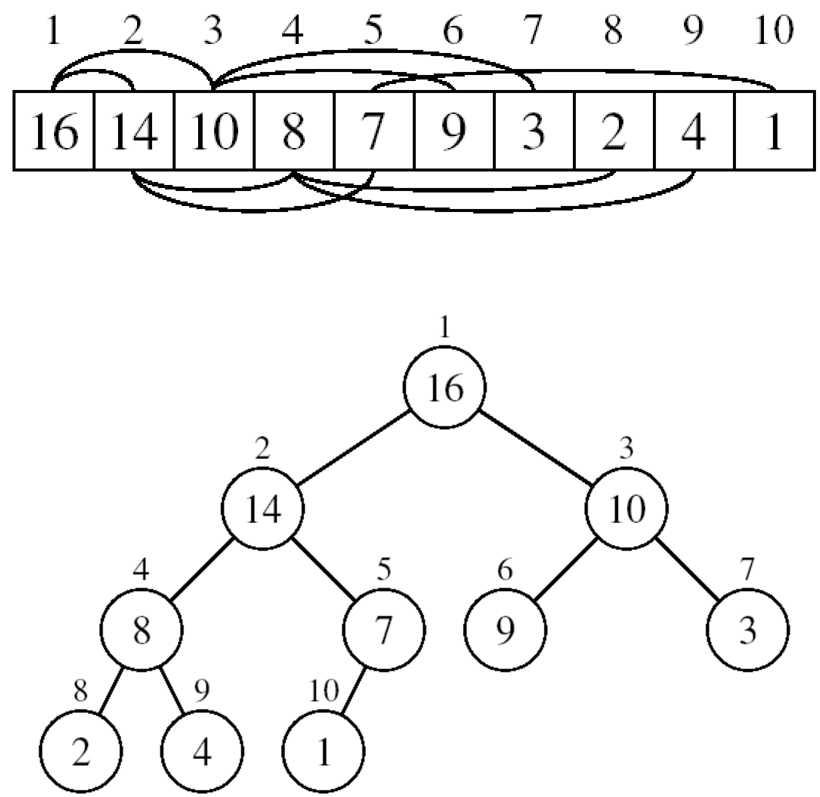
```
PARENT(i)
{
  return(i/2)
}

LEFT(i)
{
  return(2i)
}

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

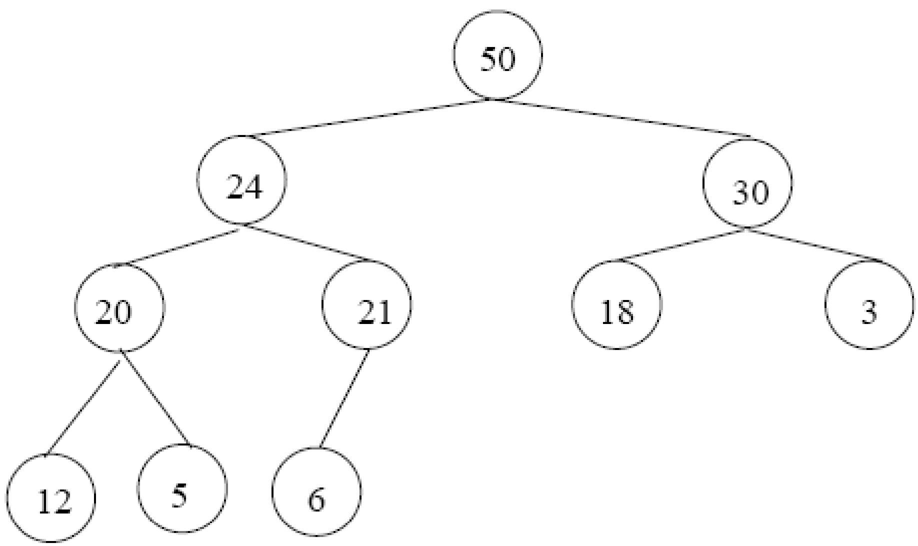
# Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2*i]$
  - Right child of $A[i] = A[2*i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - length[A] ← no. of elements in the array
  - Heapsize[A] ← no. of elements in the heap stored within array A
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves



# Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

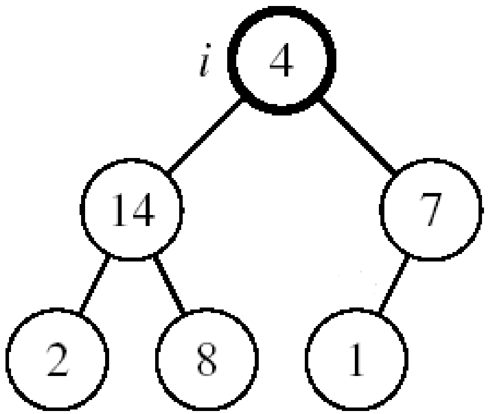


# Operations on Max-Heap

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queues

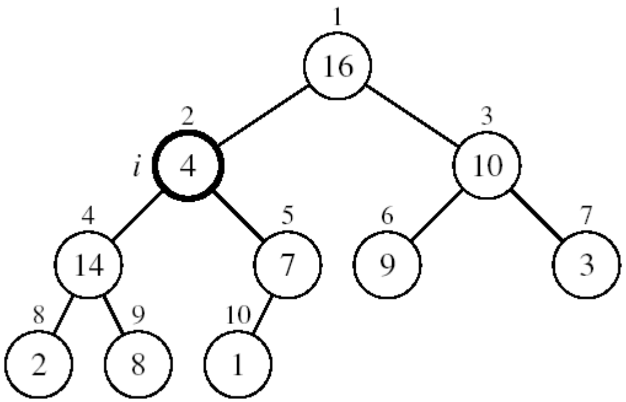
# Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of 'i' are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until the heap property not maintained.



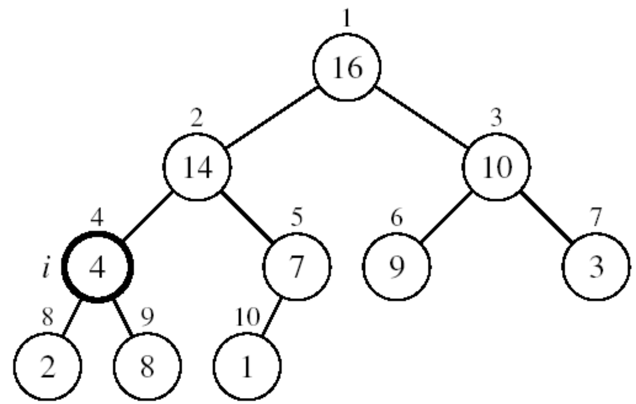
# Example

MAX-HEAPIFY(A, 2)



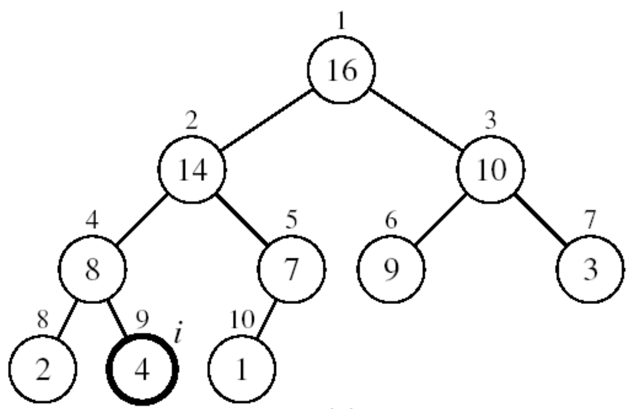

A[2] violates the heap property

A[2] ↔ A[4]




A[4] violates the heap property

A[4] ↔ A[9]

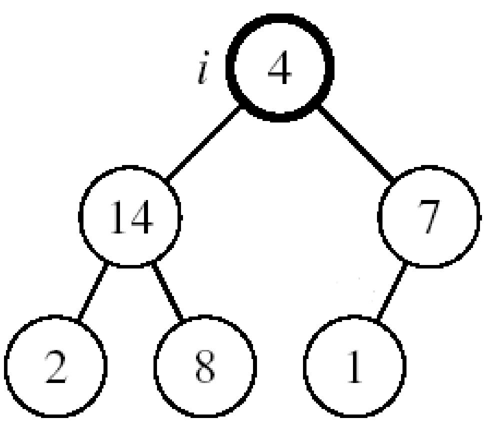



Heap property restored

# Maintaining the Heap Property

## Assumptions:

- Left and Right sub-trees of i are max-heaps
- $A$[i] may be smaller than its children




*Algorithm:* MAX-HEAPIFY($A$, i)

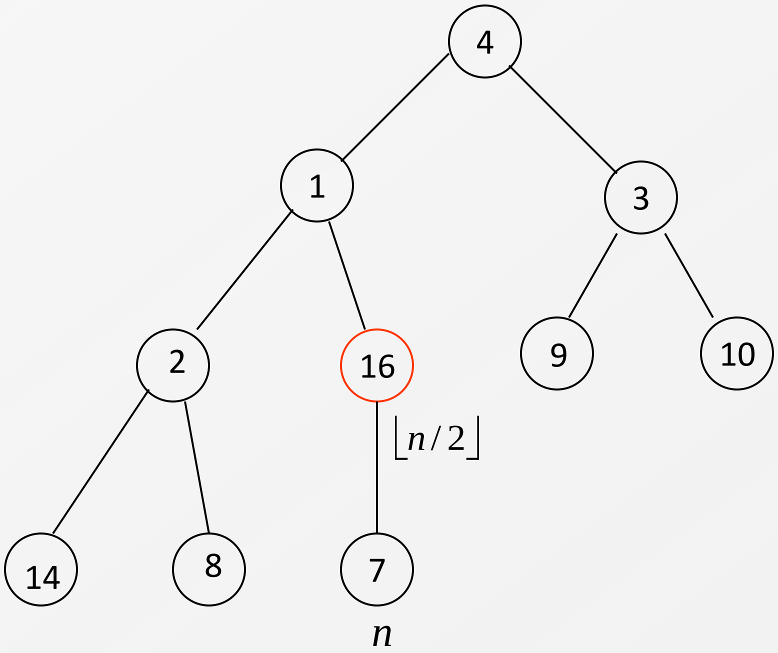
l $\leftarrow$ LEFT(i)<br>
r $\leftarrow$ RIGHT(i)<br>
**if** l $\le$ Heap-Size($A$) and $A$[l] > $A$[i]<br>
    **then** largest $\leftarrow$ l<br>
**else** largest $\leftarrow$ i<br>
**if** r $\le$ Heap-Size($A$) and $A$[r] > $A$[largest]<br>
    **then** largest $\leftarrow$ r<br>
**if** largest $\neq$ i<br>
    **then** exchange $A$[i] $\leftrightarrow$ $A$[largest]<br>
        MAX-HEAPIFY($A$, largest)

# MAX-HEAPIFY Running Time

- Intuitively:
  - It traces a path from the root to a leaf.
  - At each level it makes exactly 2 comparisons.
  - Total number of comparisons is 2h.
  - Running time is O(h) or O(lg n).
- Running time of MAX-HEAPIFY is $O(\lg n)$
- Can be written in terms of the height of the heap, as being $O(h)$
  - Since the height of the heap is $\lfloor \lg n \rfloor$

# Building a Heap

The last location who has a child is $\lfloor n/2 \rfloor$.

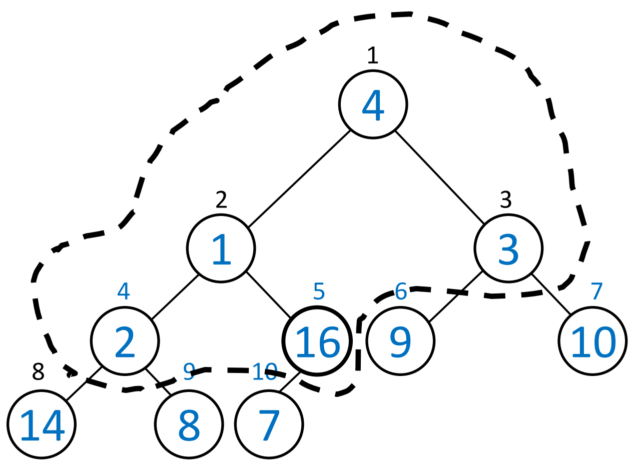


# Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap ($n = length[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) \,..\, n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

*Algorithm:* BUILD-MAX-HEAP($A$)

1. Heap-Size($A$) = length[A]
2. **for** i ← ⌊ length[A] /2⌋ **downto** 1
3. **do** MAX-HEAPIFY($A$, i)



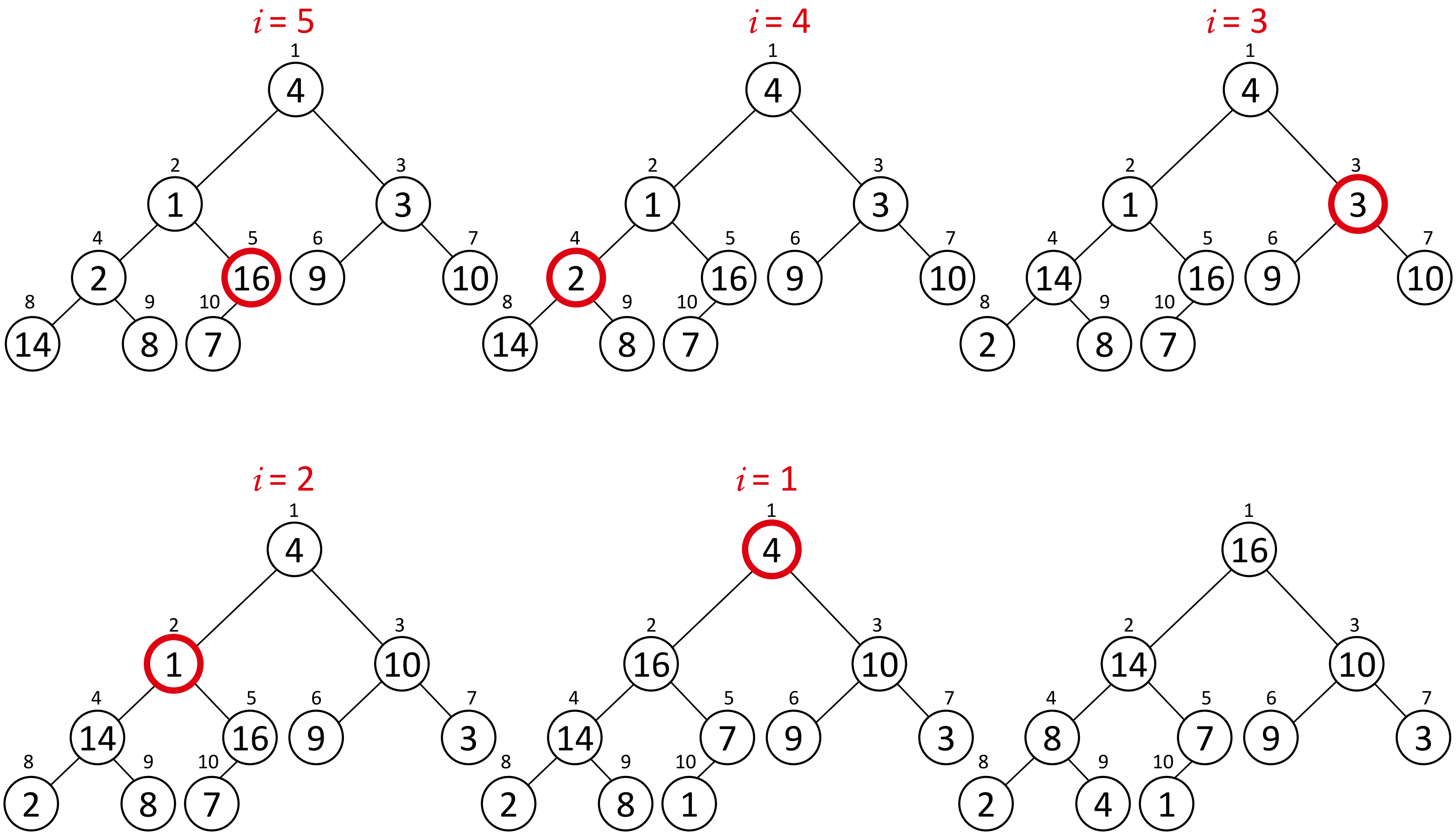

A:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |

# Example

A

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |



# Running Time of BUILD-MAX-HEAP

*Algorithm:* <u>BUILD-MAX-HEAP($A$)</u>

```
Heap-Size(A) = length[A]
for i ← ⌊ length[A] /2 ⌋ downto 1                    O(n)
    do MAX-HEAPIFY(A, i)                O(lgn)
```

$\Rightarrow$ Running time: $O(n\lg n)$

- This is not an asymptotically tight upper bound

# Running Time of BUILD-MAX-HEAP

- build-max-heap algorithm executes bottom-to-top.
- Let the size of heap = n
- Max$^{m}$ no. of elements with height h, $= \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- When max-heapify is called to a node having height h, the cost is O(h)
- For all nodes of height h, the total cost = $\left\lceil \frac{n}{2^{h+1}} \right\rceil * O(h)$

# Running Time of BUILD-MAX-HEAP

For all nodes with varying height, the time complexity T(n)

$$= \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil * O(h)$$

$$= O\left( n \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{h}{2^h} \right\rceil \right)$$

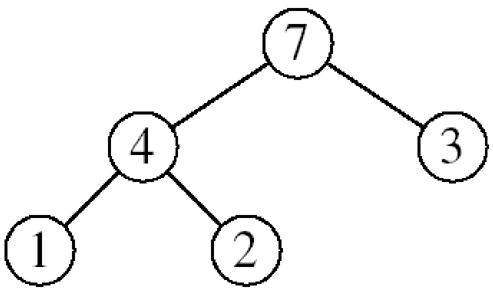
$$\leq \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= \; O(n)$$

$$But \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

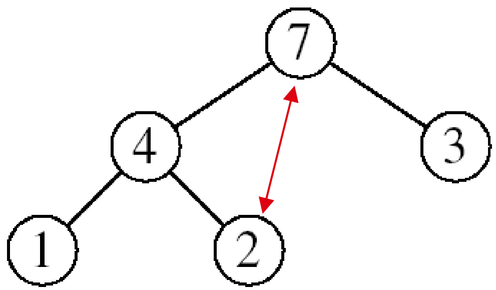
# Heap sort

- Goal:
  - Sort an array using heap representations



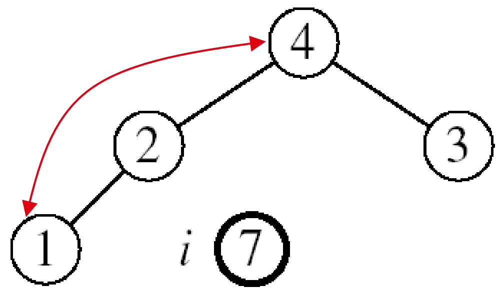

- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

# Example: A=[7, 4, 3, 1, 2]



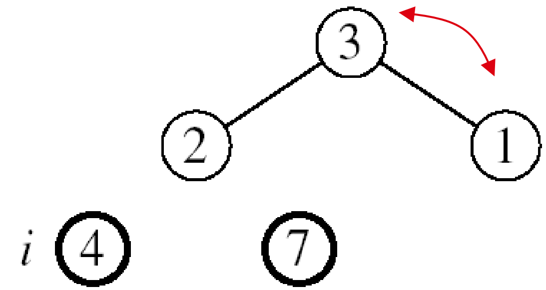
EXCHANGE A[1] ↔ A[5]
MAX-HEAPIFY(A, 1)

| | | | | |
|---|---|---|---|---|
| 7 | 4 | 3 | 1 | 2 |



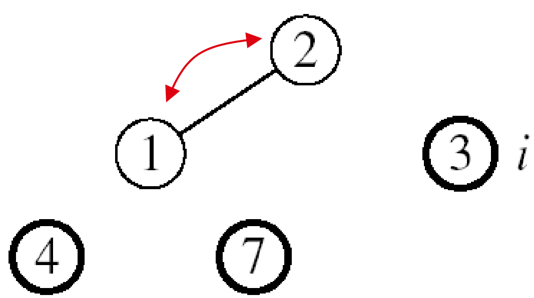

EXCHANGE A[1] ↔ A[4]
MAX-HEAPIFY(A, 1)

| | | | | |
|---|---|---|---|---|
| 4 | 2 | 3 | 1 | 7 |



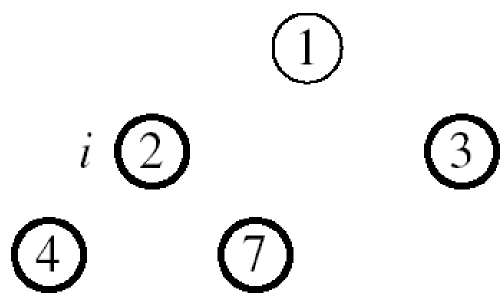

EXCHANGE A[1] ↔ A[3]
MAX-HEAPIFY(A, 1)

| | | | | |
|---|---|---|---|---|
| 3 | 2 | 1 | 4 | 7 |




EXCHANGE A[1] ↔ A[2]
MAX-HEAPIFY(A, 1)

| | | | | |
|---|---|---|---|---|
| 2 | 1 | 3 | 4 | 7 |




| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 7 |

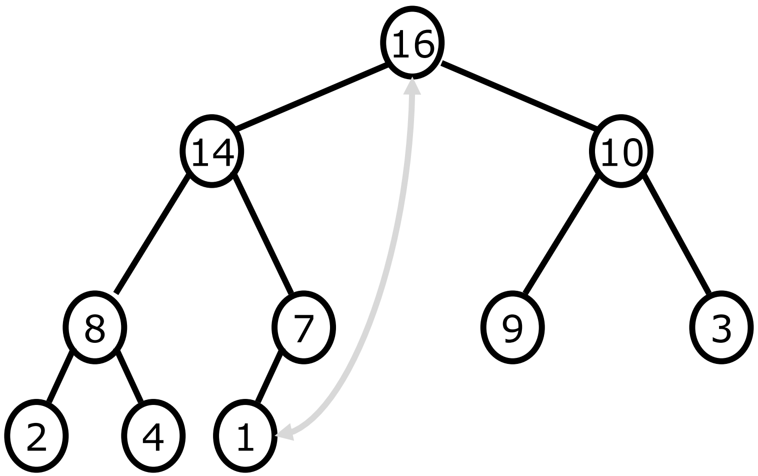
# HEAPSORT(*A*)

*Algorithm: HEAPSORT(A)*

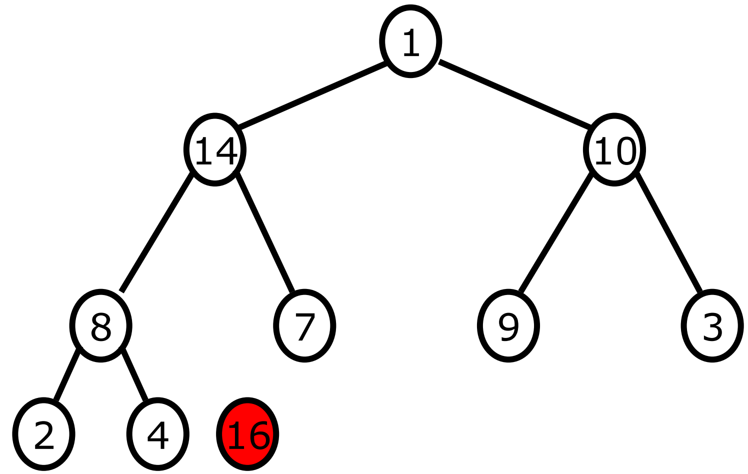
| | | |
|---|---|---|
| 1. BUILD-MAX-HEAP($A$) | $O(n)$ | |
| 2. **for** $i \leftarrow length[A]$ **downto** 2 | | |
| 3. **do** exchange $A[1] \leftrightarrow A[i]$ | $\theta(1)$ | n-1 times |
| 4. Heap-Size($A$) = Heap-Size($A$) -1 | $\theta(1)$ | |
| 5. MAX-HEAPIFY($A$, 1) | $O(\lg n)$ | |

- Running time: $O(n\lg n)$

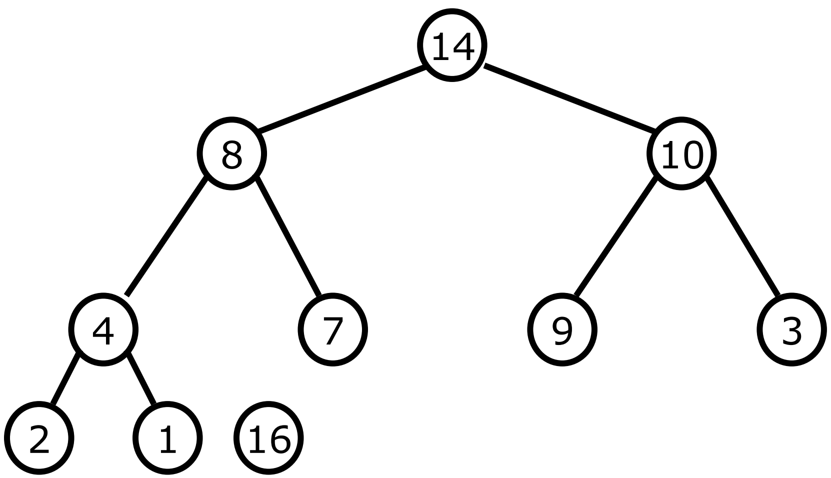
# Example 2



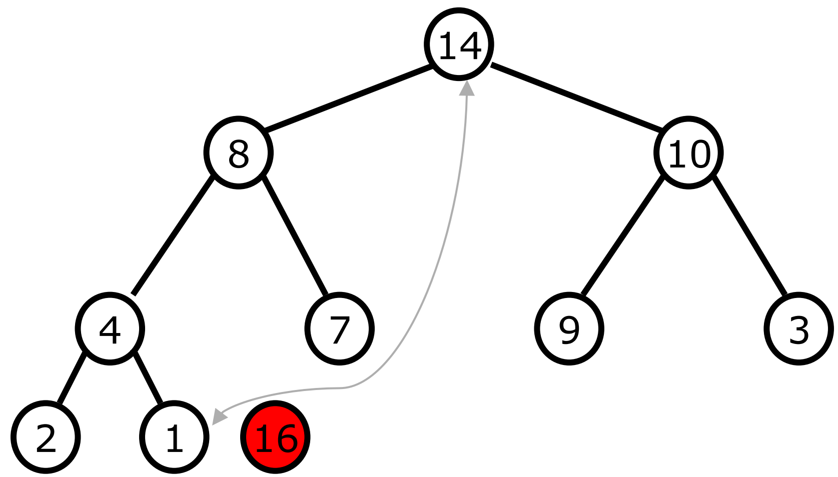
16
14
10
8
7
9
3
2
4
1

1
14
10
8
7
9
3
2
4
16

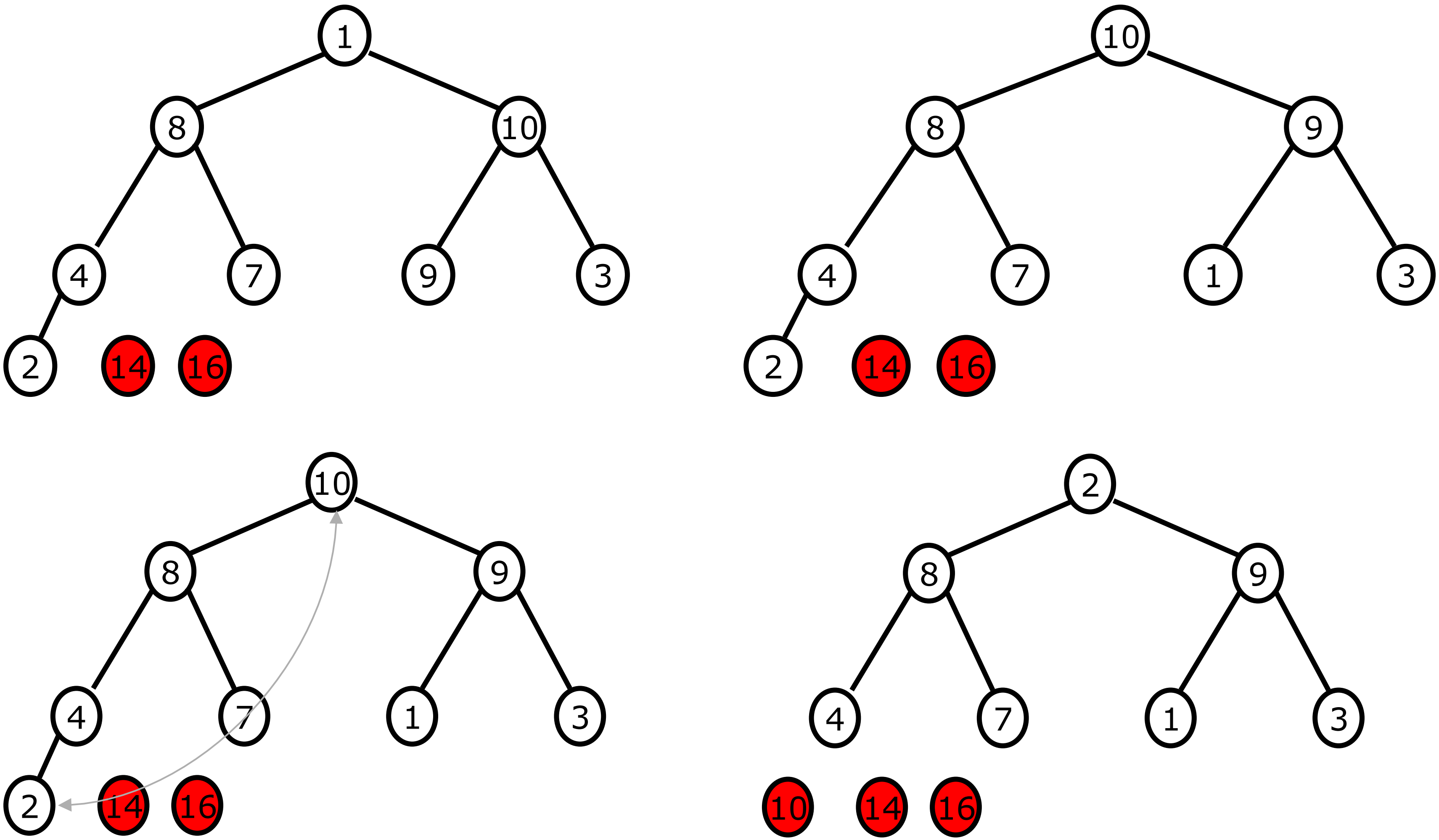



14
8
10
4
7
9
3
2
1
16

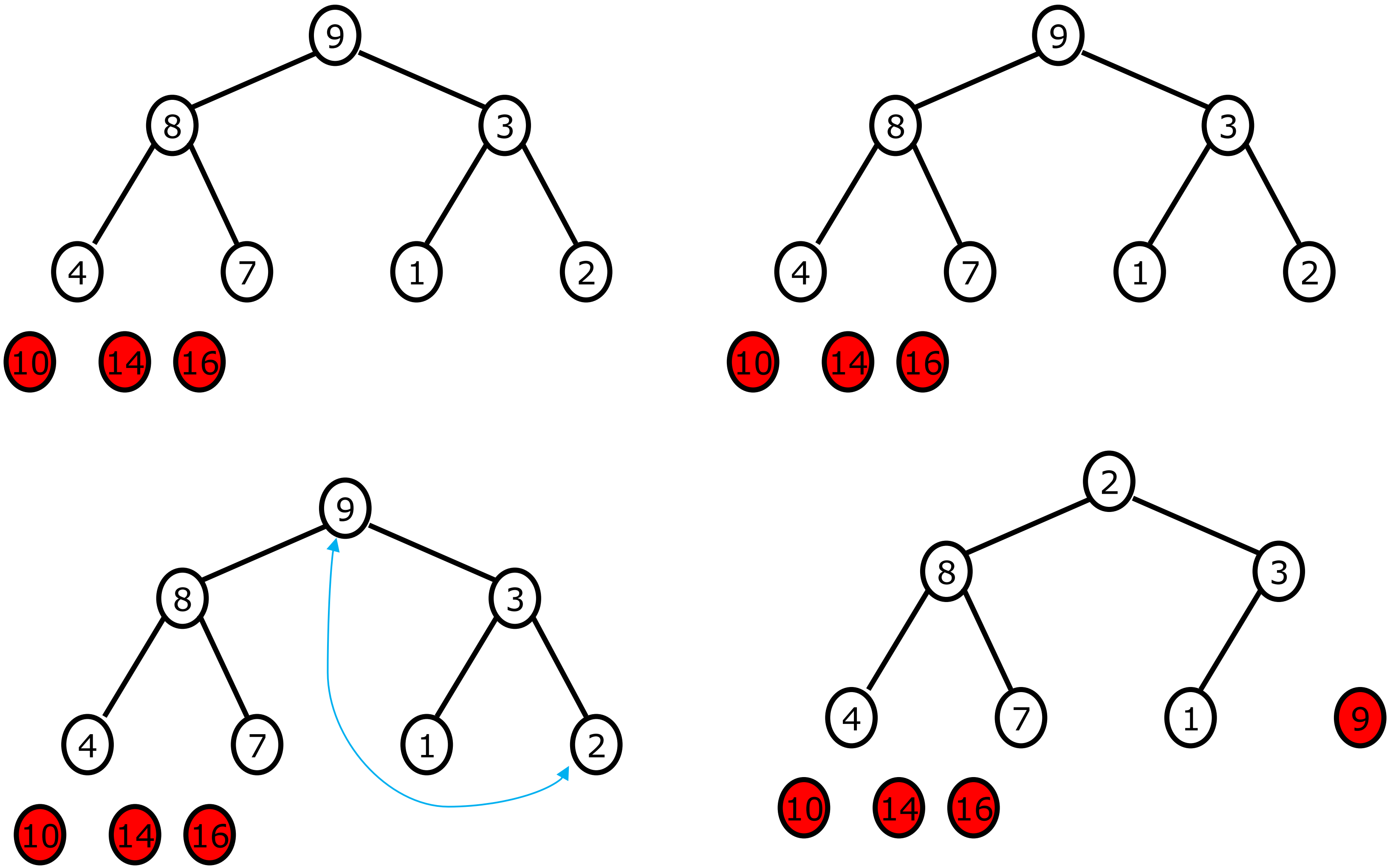



14
8
10
4
7
9
3
2
1
16

# Example 2

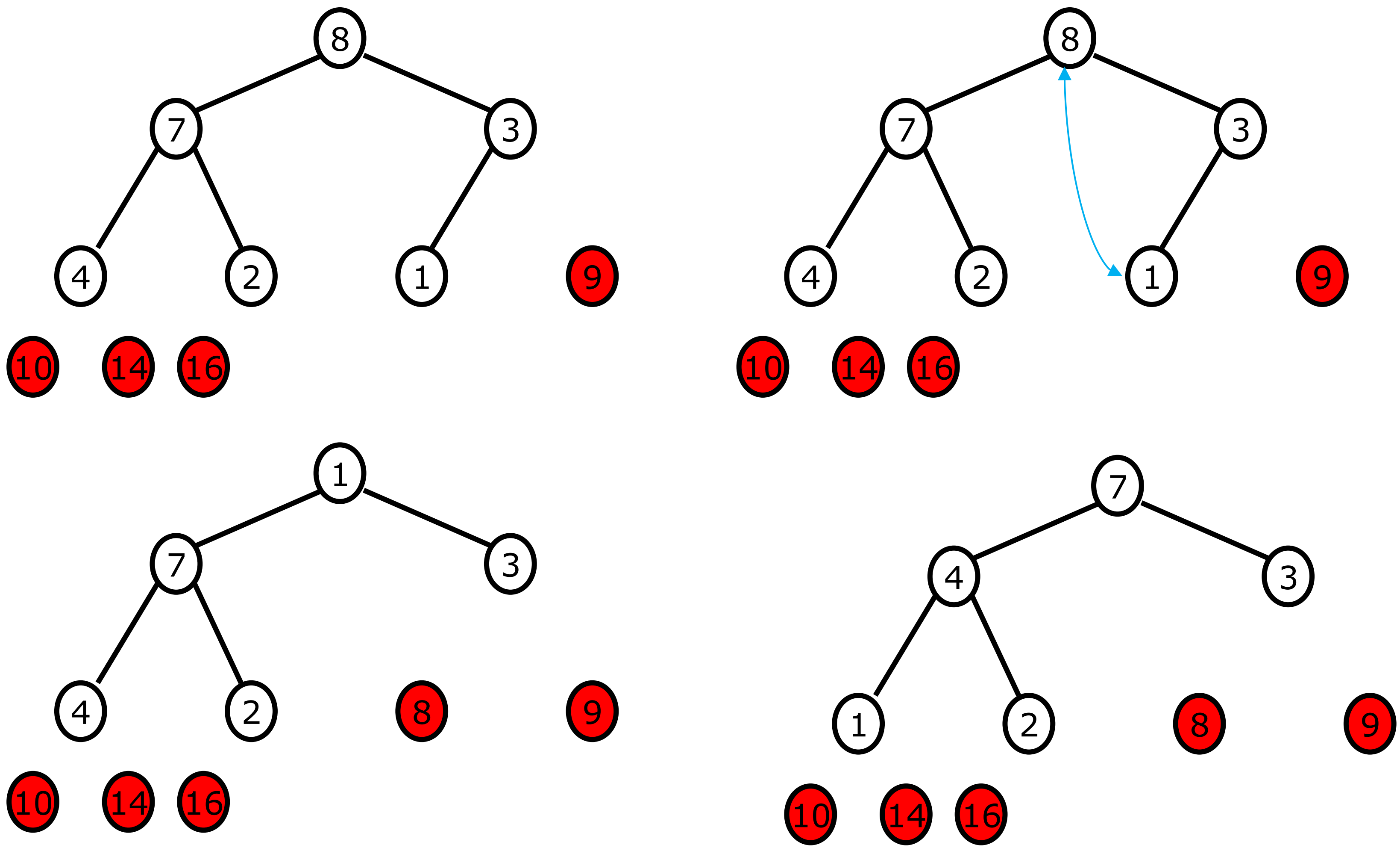

1 8 10 4 7 9 3 2 14 16
10 8 9 4 7 1 3 2 14 16
10 8 9 4 7 1 3 2 14 16
2 8 9 4 7 1 3 10 14 16

# Example 2

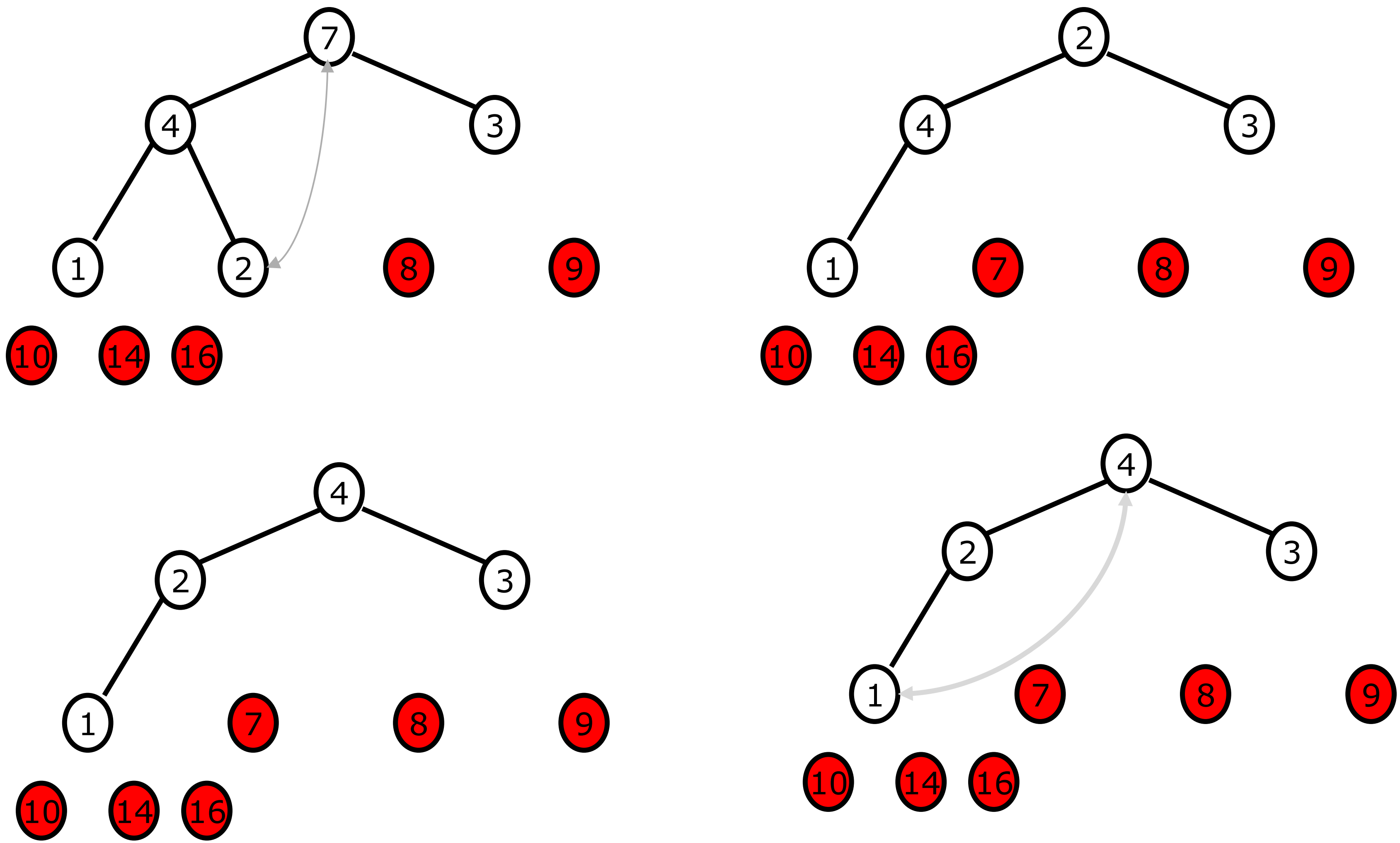


9
8
3
4
7
1
2
10
14
16
9
8
3
4
7
1
2
10
14
16
9
8
3
4
7
1
2
10
14
16
2
8
3
4
7
1
9
10
14
16

# Example 2

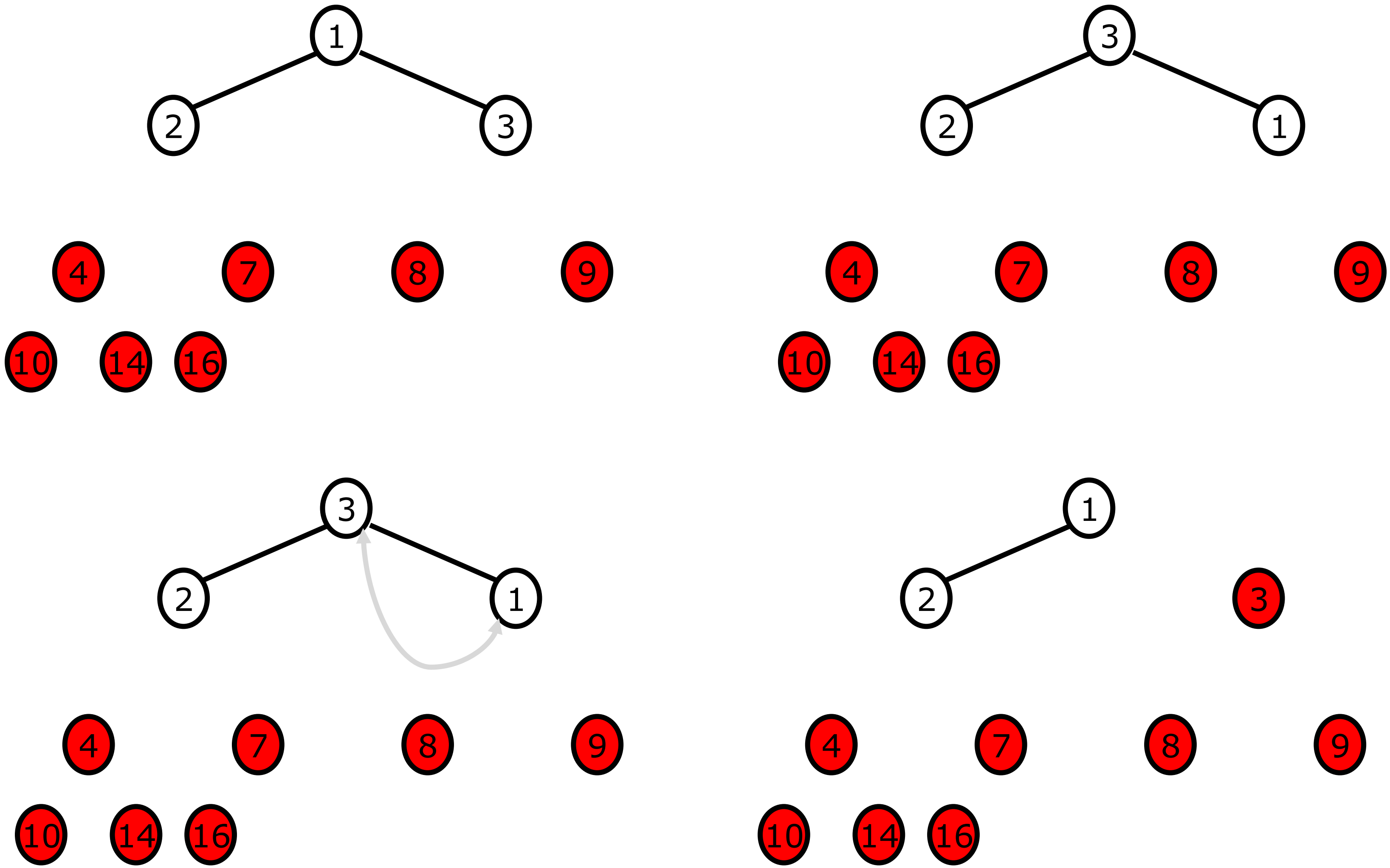

8
7
3
4
2
1
9
10
14
16
8
7
3
4
2
1
9
10
14
16
1
7
3
4
2
8
9
10
14
16
7
4
3
1
2
8
9
10
14
16

# Example 2

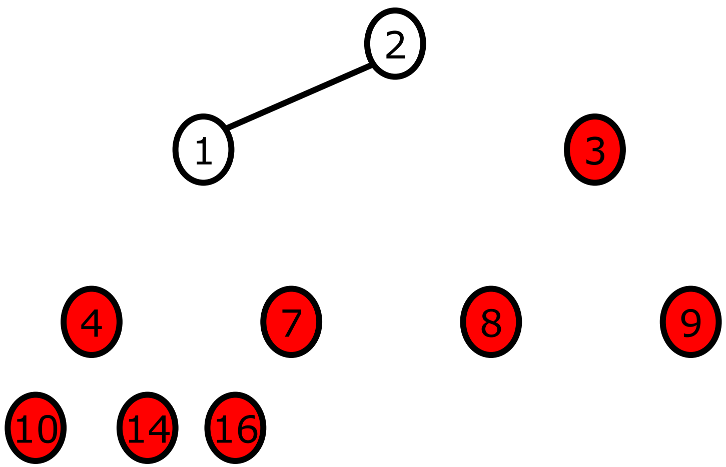


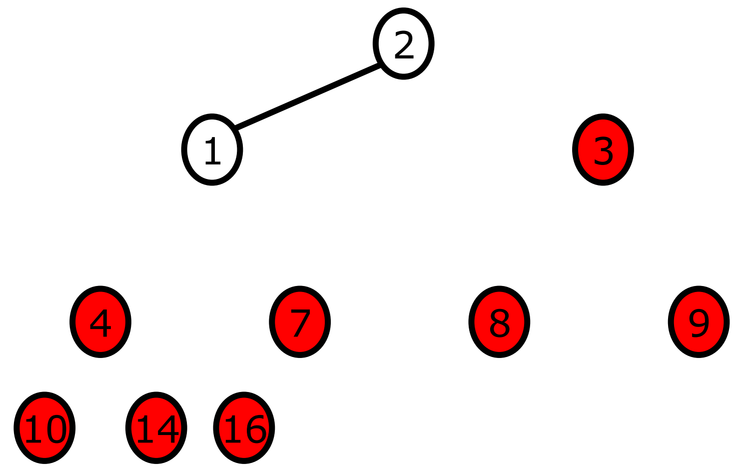
7
4
3
1
2
8
9
10
14
16
2
4
3
1
7
8
9
10
14
16
4
2
3
1
7
8
9
10
14
16
4
2
3
1
7
8
9
10
14
16

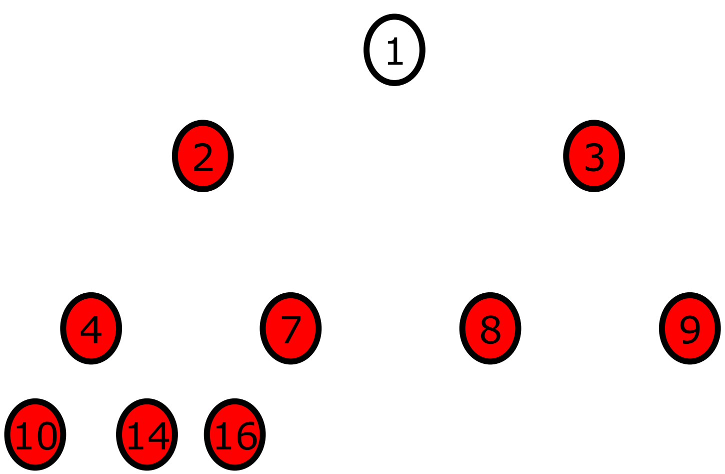
# Example 2

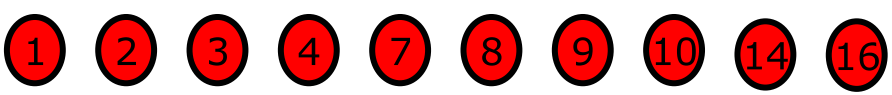

1
2
3
4
7
8
9
10
14
16
3
2
1
4
7
8
9
10
14
16
3
2
1
4
7
8
9
10
14
16
1
2
3
4
7
8
9
10
14
16

# Example 2




**Sorted Array**

# Priority Queues

A priority queue is a data structure for maintaining a set “S” where each element of S is associated with a key (priority).

**Properties:**

- Each element is associated with a value (priority).
- The  key with the highest (or lowest) priority is extracted first.

# Types of Priority Queue

There are two kinds of Priority Queue

- Max-Priority Queue – based on max-heap
  - Application(s): Job Scheduling, Load Balancing, Real-Time Systems, …
- Min-Priority Queue – based on min-heap
  - Application(s): Event-driven simulator, Huffman code, Dijkstra Algorithm, Prim's Algorithm, Task Scheduling…

# Operations on Max-Priority Queues

Max-Priority queue supports the following operations:

- INSERT(S, x): <u>inserts</u> element x into set S
- EXTRACT-MAX(S): <u>removes and returns</u> element of S with largest key
- MAXIMUM(S): <u>returns</u> element of S with largest key
- INCREASE-KEY(S, x, k): <u>increases</u> value of element x's key to k (Assume k ≥ x's current key value)

# HEAP-MAXIMUM

## Goal:

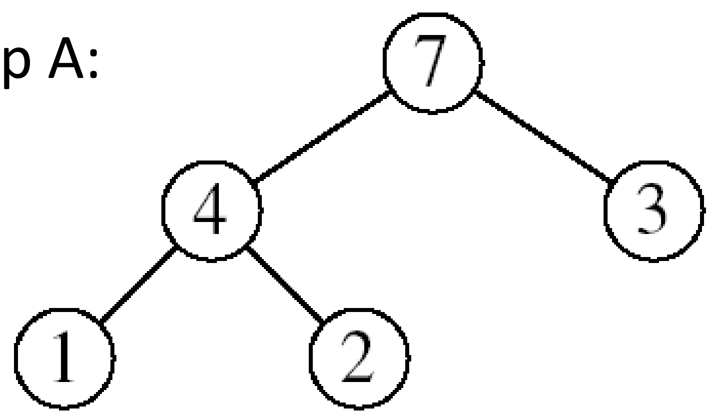
– Return the largest element of the heap

*Algorithm:* HEAP-MAXIMUM($A$)

Running time: $O(1)$

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

Heap A:



Heap-Maximum(A) returns 7

# HEAP-EXTRACT-MAX

Goal:

– Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

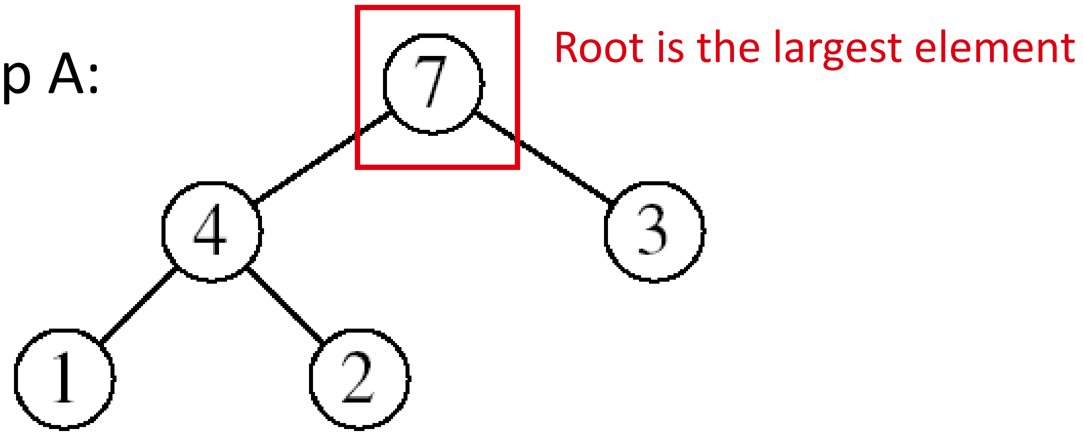
Idea:

– Exchange the root element with the last

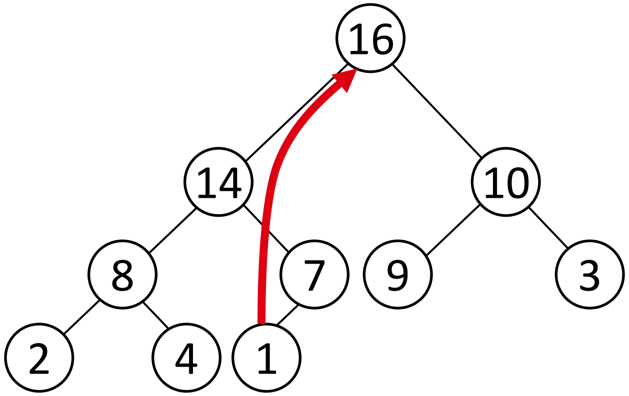
– Decrease the size of the heap by 1 element

– Call MAX-HEAPIFY on the new root, on a heap of size n-1

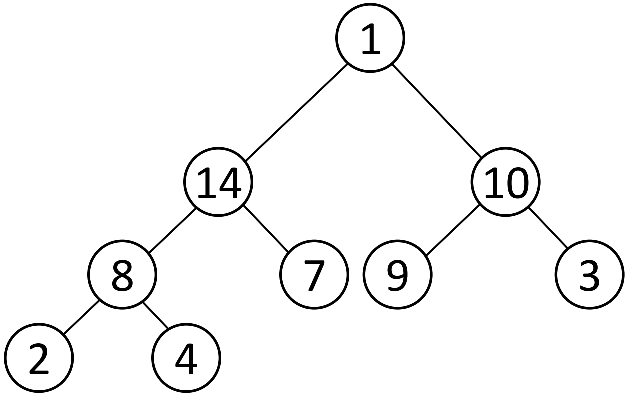
Heap A:



# Example: HEAP-EXTRACT-MAX

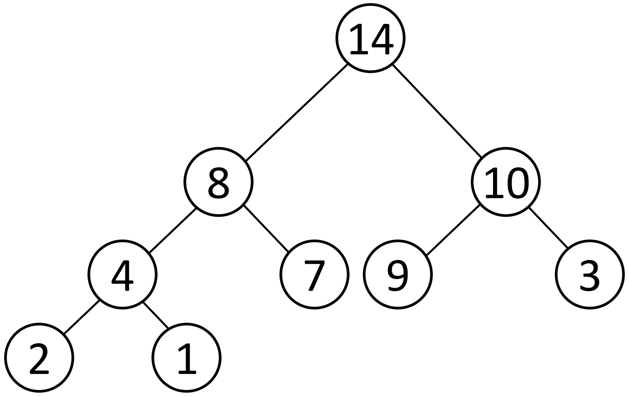



max = 16




Heap size decreased with 1

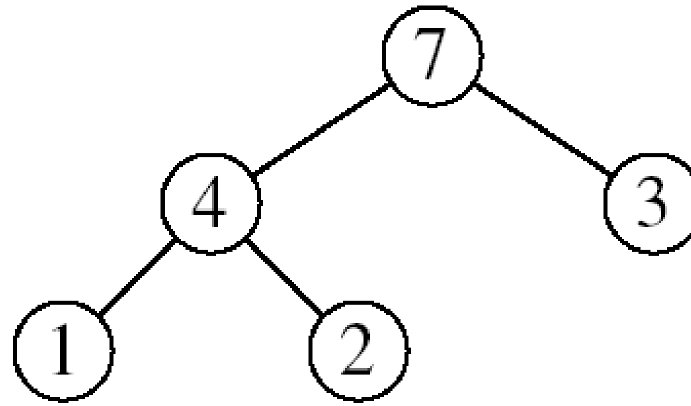
Call MAX-HEAPIFY($A$, 1)



# HEAP-EXTRACT-MAX

*Algorithm:* 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) ▷ remakes heap
7. **return** max

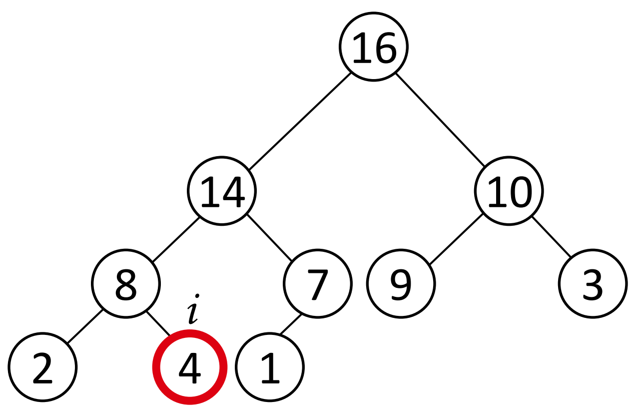



Running time: $O(\lg n)$

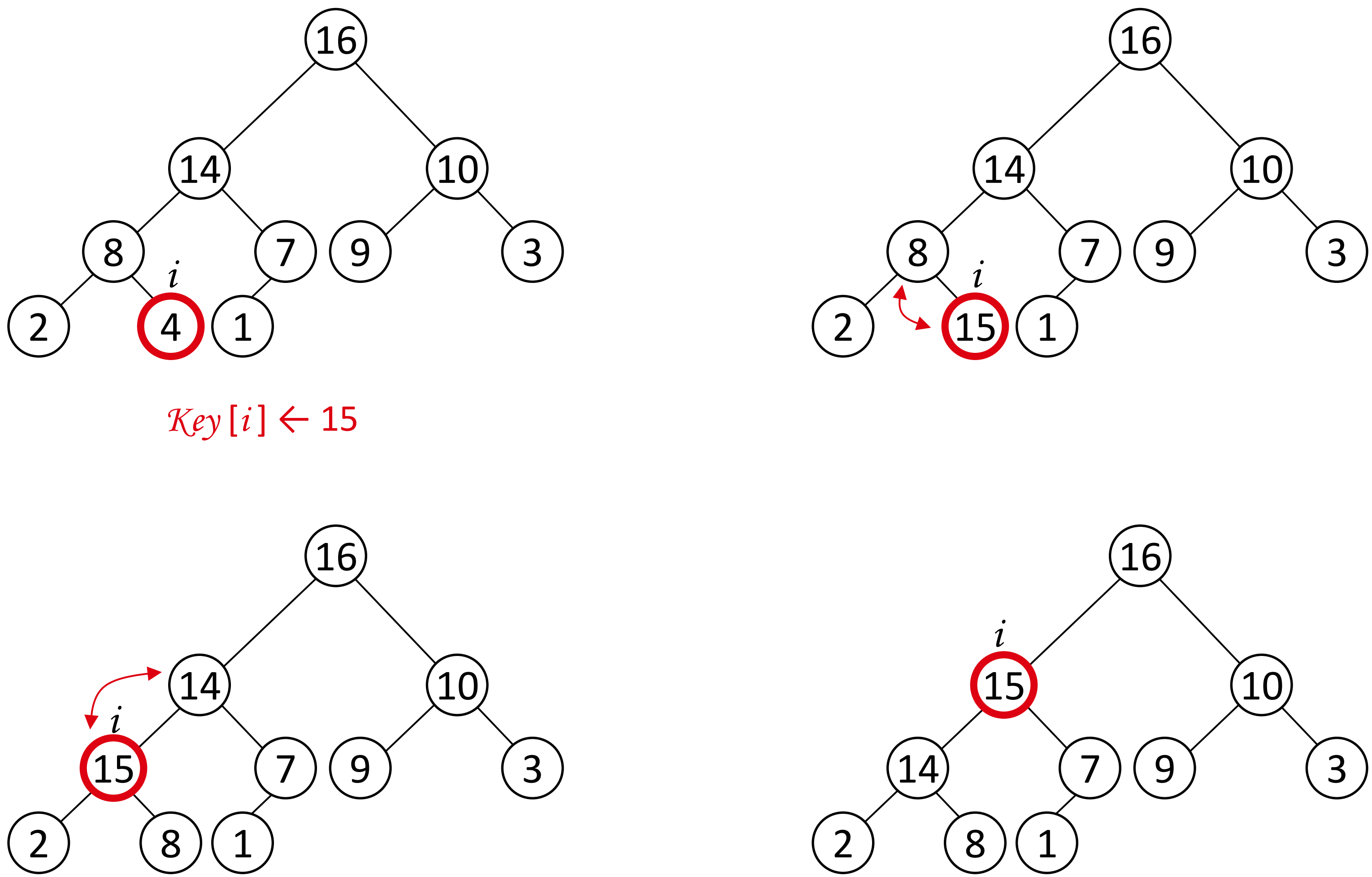
# HEAP-INCREASE-KEY

- Goal:
  - Increases the key of an element i in the heap
- Idea:
  - Increment the key of $A[i]$ to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key

Key [i] ← 15



# Example: HEAP-INCREASE-KEY

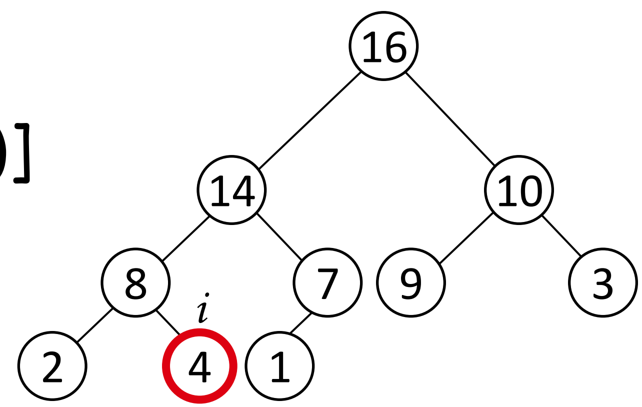


# HEAP-INCREASE-KEY

*Algorithm:* HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ↔ A[PARENT(i)]
        i ← PARENT(i)
```

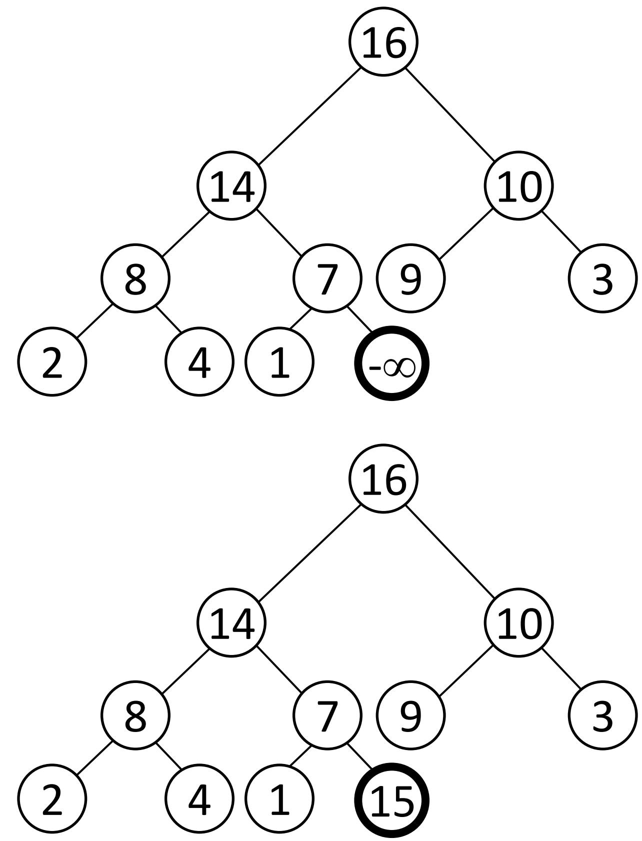
- Running time: $O(\lg n)$




Key [i] ← 15

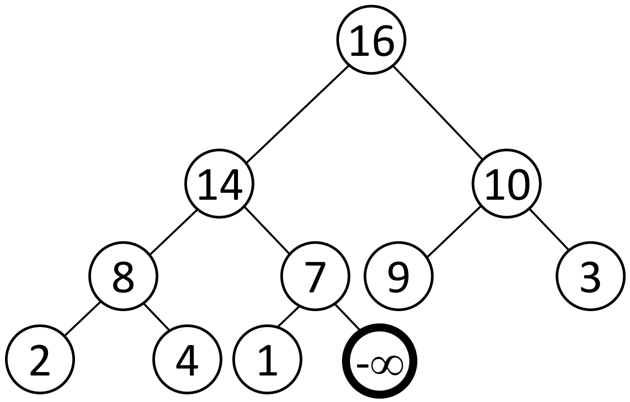
# MAX-HEAP-INSERT

- Goal:
  - Inserts a new element into a max-heap
- Idea:
  - Expand the max-heap with a new element whose key is $-\infty$
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property

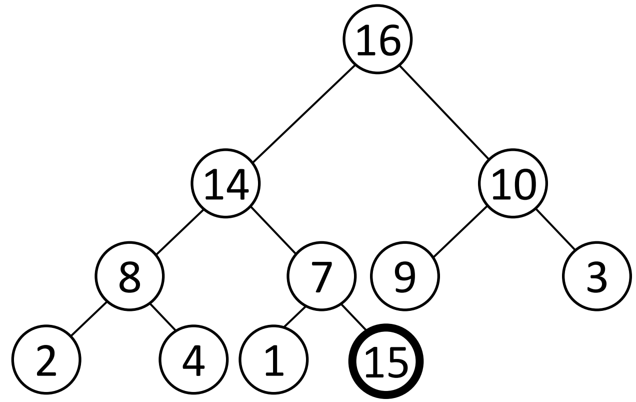


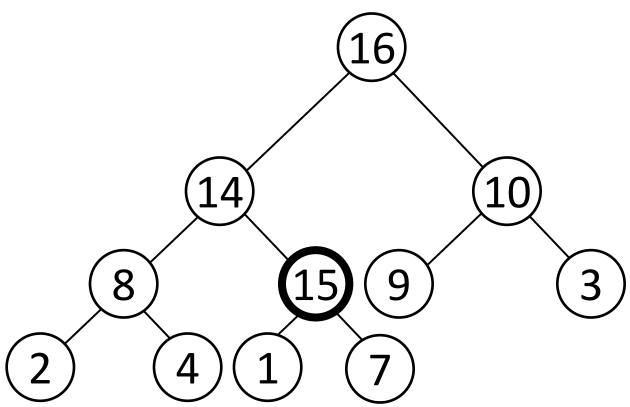
# Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

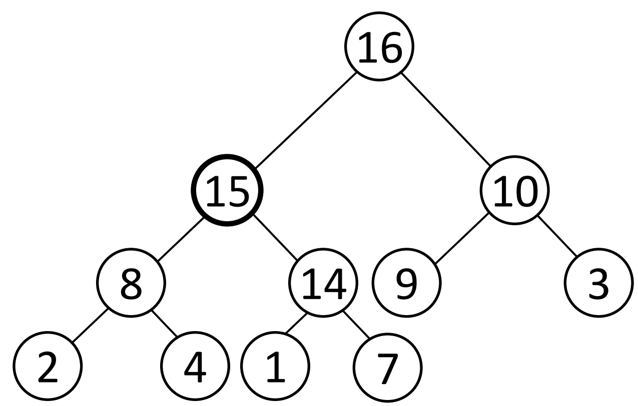



Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15



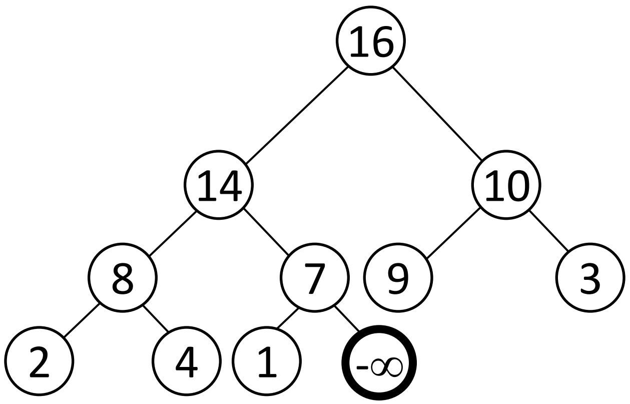

The restored heap containing
the newly added element



# MAX-HEAP-INSERT

*Algorithm:* 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)




Running time: $O(\lg n)$

# Summary

- We can perform the following operations on max-heaps:
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n \lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Average $O(\lg n)$

# Home work

- Illustrate the operations of HEAP-EXTRACT-MAX on the heap A=<15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>.

- Illustrate the functioning of the MAX-HEAP-INSERT(A, 10) operation subsequent to the execution of the HEAP-EXTRACT-MAX operation.

- Write pseudo code for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.


Each of your
actions will
have an
impact on your
future.


Once you know
who is walking
with you on your path.
you will never
be afraid.

# Thank you