# Heapsort

Mohamed Faouzi Atig

Uppsala University

# Sorting Algorithms

- Problem: Sort an array $A$ of $n$ elements in non-decreasing order

| Algorithm | Worst-Case | Average-Case | Best-Case | In place? |
|---|---|---|---|---|
| Insertion Sort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n)$ | Yes |
| Merge Sort | $\Theta(nlog(n))$ | $\Theta(nlog(n))$ | $\Theta(nlog(n))$ | No |
| Quicksort | $\Theta(n^2)$ | $\Theta(nlog(n))$ | $O(nlog(n))$ | Yes |

# Sorting Algorithms

- Problem: Sort an array $A$ of $n$ elements in non-decreasing order

| Algorithm | Worst-Case | Average-Case | Best-Case | In place? |
|---|---|---|---|---|
| Insertion Sort | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n)$ | Yes |
| Merge Sort | $\Theta(nlog(n))$ | $\Theta(nlog(n))$ | $\Theta(nlog(n))$ | No |
| Quicksort | $\Theta(n^2)$ | $\Theta(nlog(n))$ | $\Theta(nlog(n))$ | Yes |
| Heapsort | $O(nlog(n))$ | $O(nlog(n))$ | $O(nlog(n))$ | Yes |

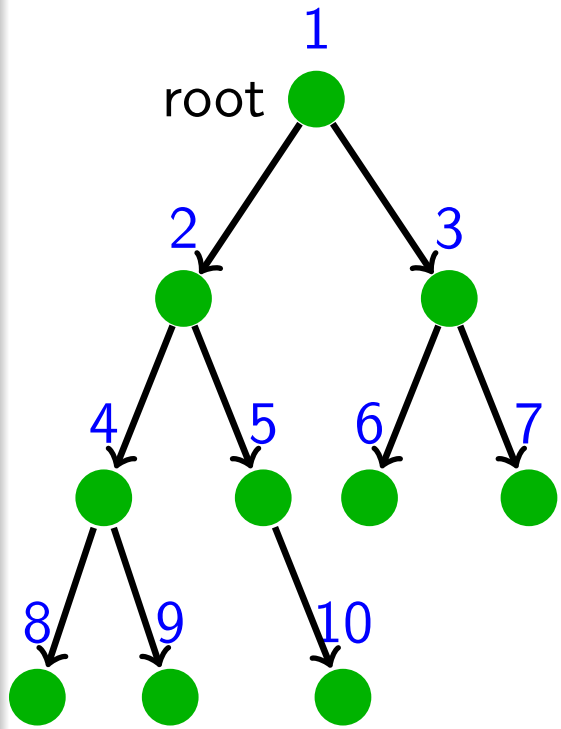
# HeapSort: Introduction

- HeapSort was invented by J. W. J. Williams in 1964.
- Based on a useful of data structure called heap
- Sorting in place algorithm

# Tree: Definition

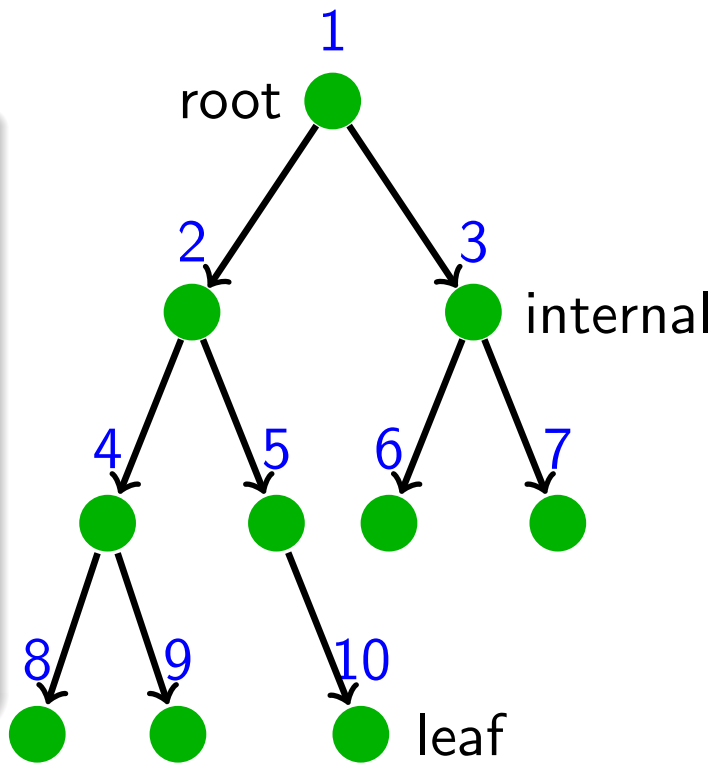
- A tree T is a directed graph $(N, E)$ where:
  - $N$ is a set of nodes.
  - $E \subset N \times N$ is a finite set of edges.

  such that the following properties hold:
  - $T$ is an acyclic connected graph.
  - For each $(n_1, n_2) \in E$, the node $n_1$ is the parent of $n_2$
    - Each node of $T$ has at most one parent.
    - There is exactly one node that does not have a parent called the root node.



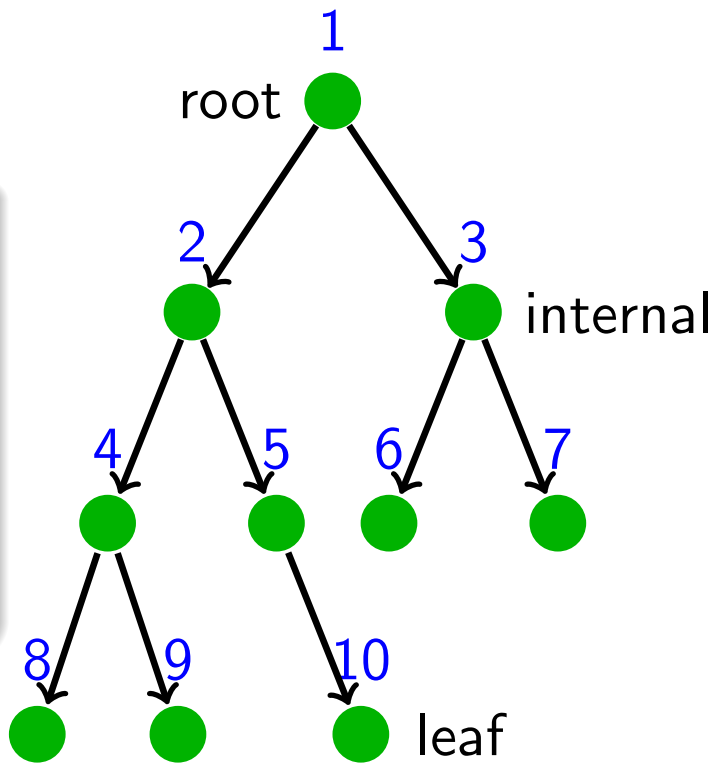
# Tree: Notations

- If a node $n_1$ is a parent of a node $n_2$ then $n_2$ is a child of $n_1$
- If two nodes have the same parent then they are siblings.
- A node with at least one child is an internal node.



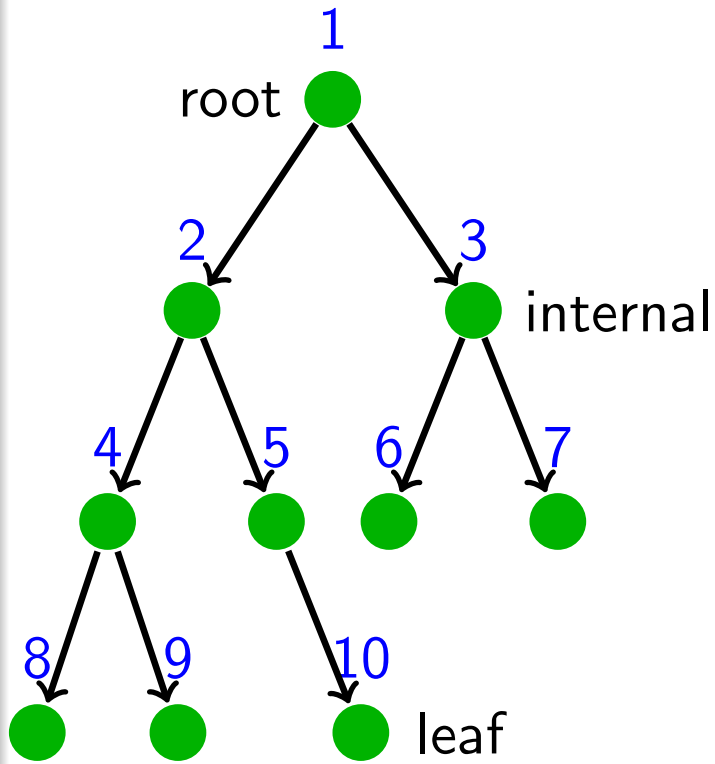
# Tree: Notations

- The node 1 is a parent of the node 2
- The node 4 is a child of the node 2
- The nodes 4 and 5 are siblings



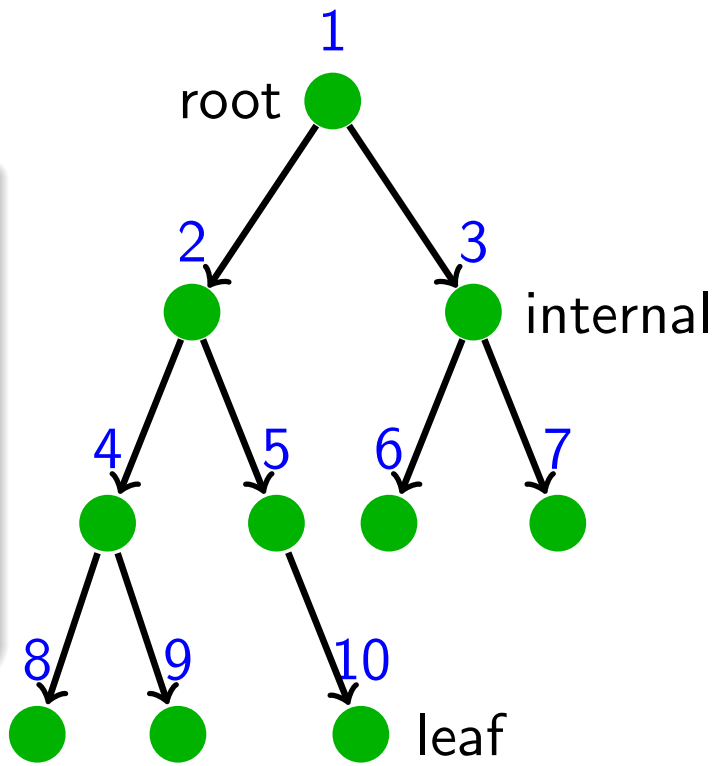
# Tree: Notations

- A path is a sequence of nodes $n_1, n_2, \ldots, n_m$ such that for all $i : 1 \leq i < m$, $(n_i, n_{i+1})$ is an edge.
- The height of a node $n$ is the number of edges of the longest path to a leaf from this node.
- The height of a tree is the height of its root node.
- The depth of a node $n$ is the number of edges in the path from the root to $n$.



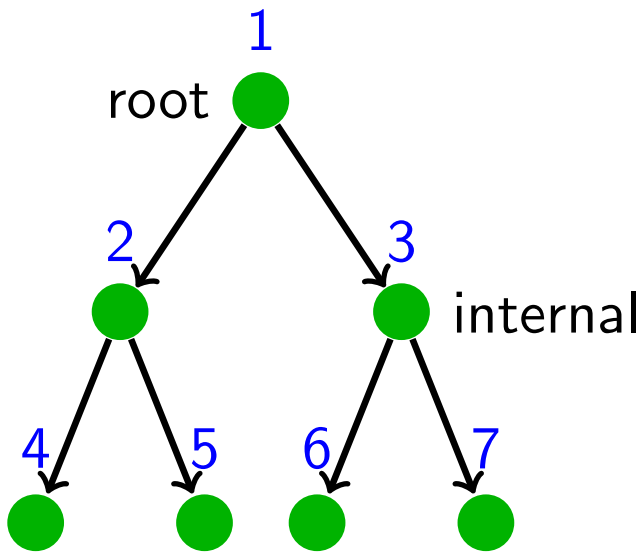
# Tree: Notations

- The sequence $1, 2, 4, 8$ is path
- The height of $2$ is $2$
- The height of the tree is $3$
- The depth (or level) of the node $7$ is $2$



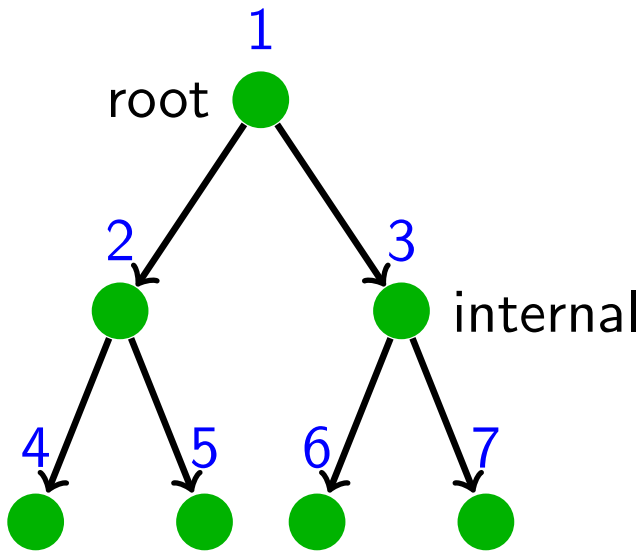
# Binary Trees

- A binary tree is a tree such that:
  - Each node has at most two child nodes, distinguished by left and right.
  - The left child always precedes the right child
- A full binary tree is a binary tree in which each internal node has exactly two children.
- A perfect binary tree is a full binary tree in which all the leaves have the same depth.



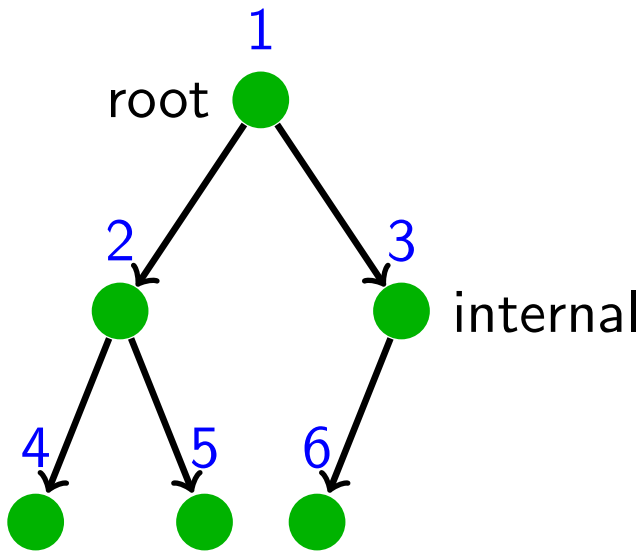
# Full Binary Trees: Properties

- The number of leaves is equal to the number of internal nodes plus 1.
- The number of internal nodes is equal to $\frac{n-1}{2}$ where n is totally number of nodes.
- The number of nodes at depth (or level) $i$ is $\leq 2^i$

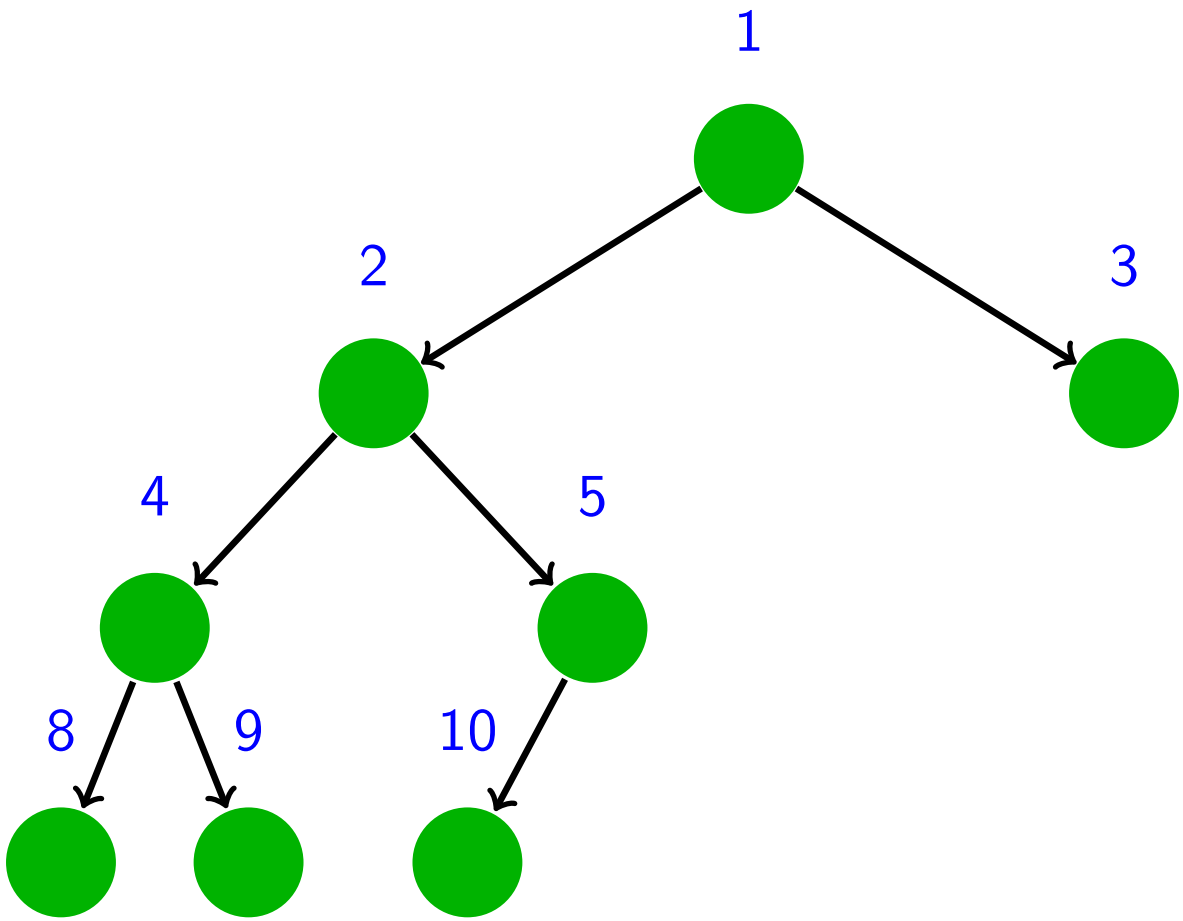


# Complete Binary Tree

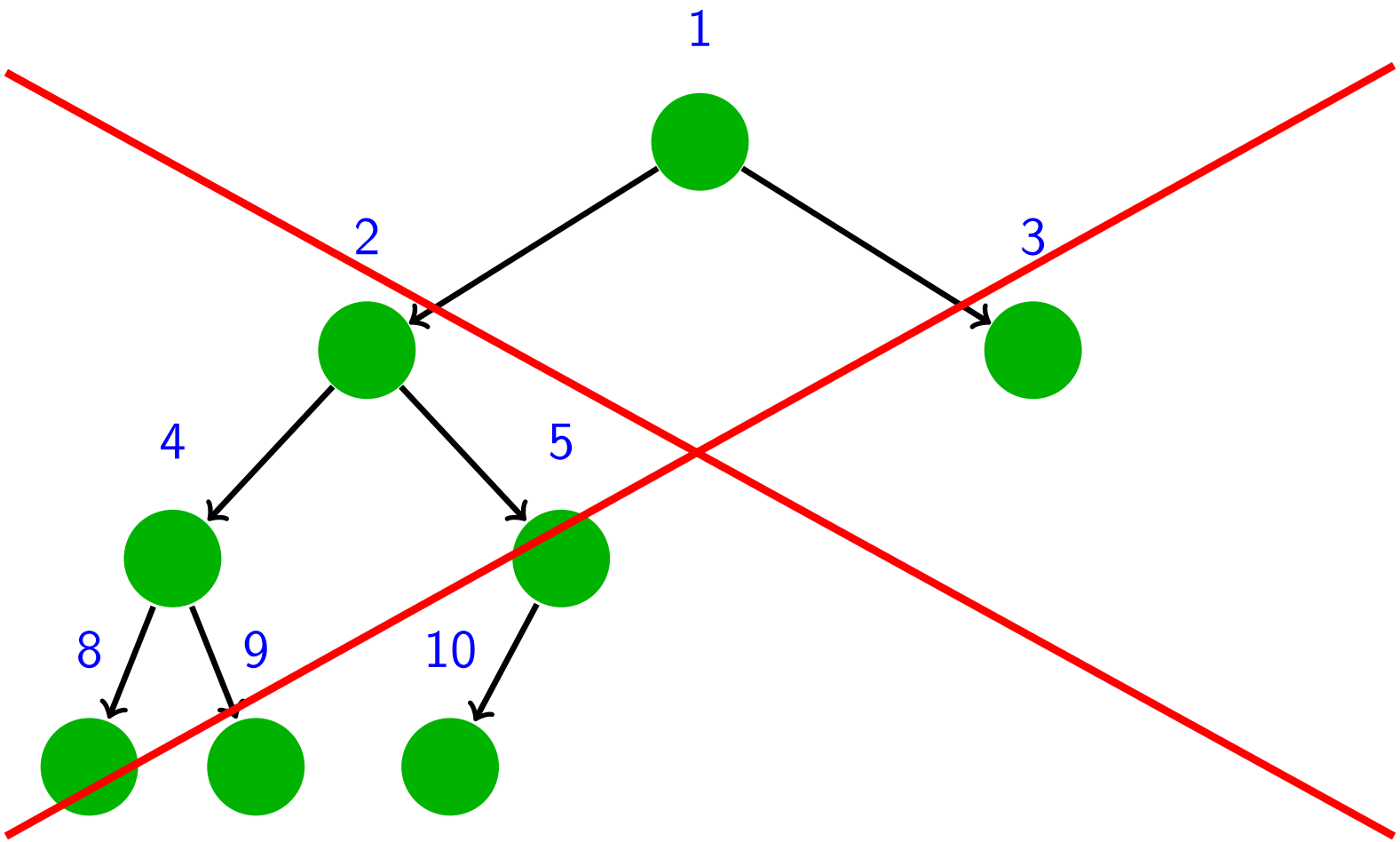
- A complete binary tree is a binary tree, which is completely filled at all the levels except possibly the highest, which is filled from left. Formally, we have
  - If $h$ is the height of the tree, then:
    - For all $i : 0 \leq i < h$, there is exactly $2^i$ nodes at depth $i$
    - A leaf node has a depth $h$ or $h - 1$
    - The leaves of depth $h$ are filled from left to right.



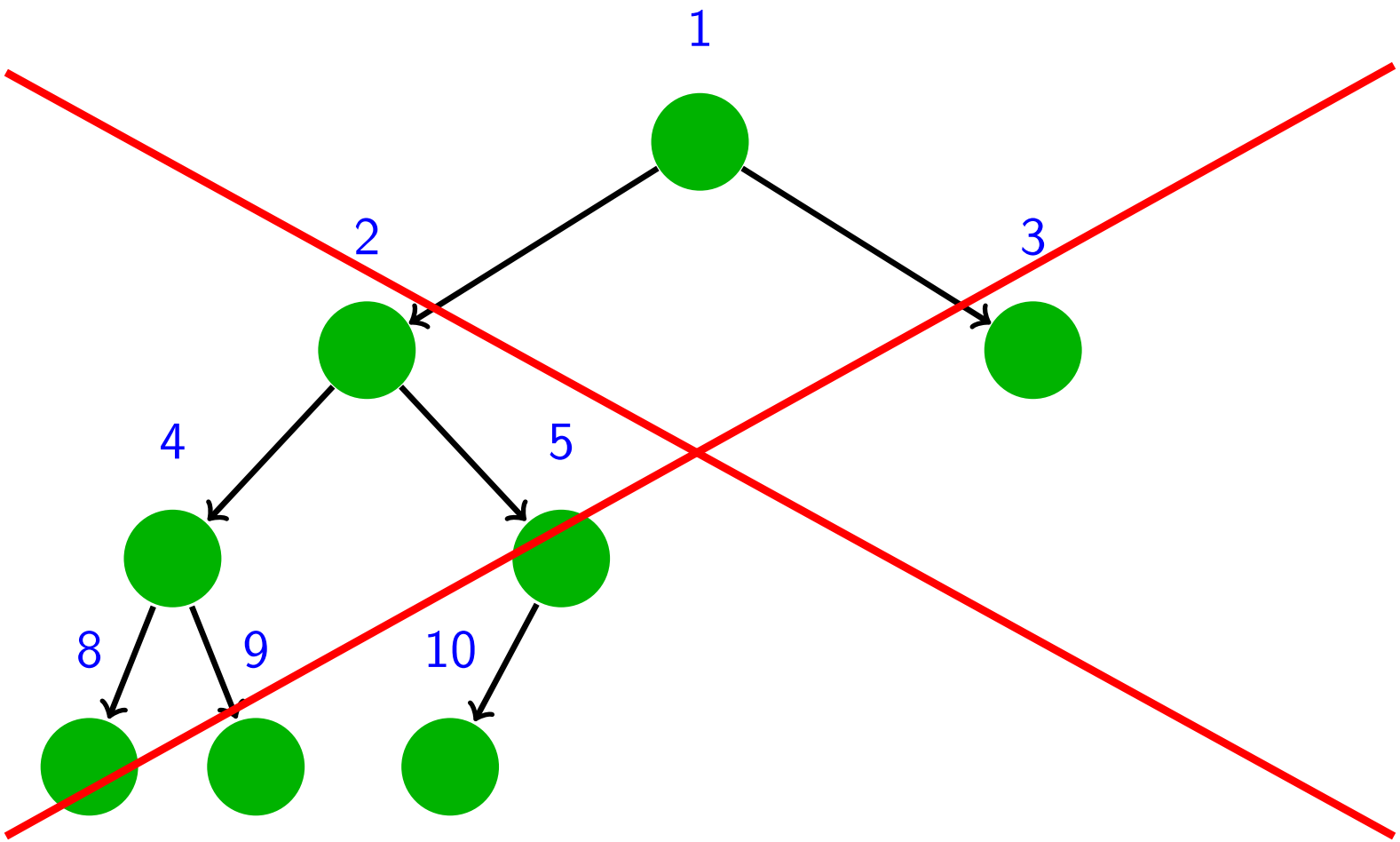
# Example (1/2)



# Example (1/2)

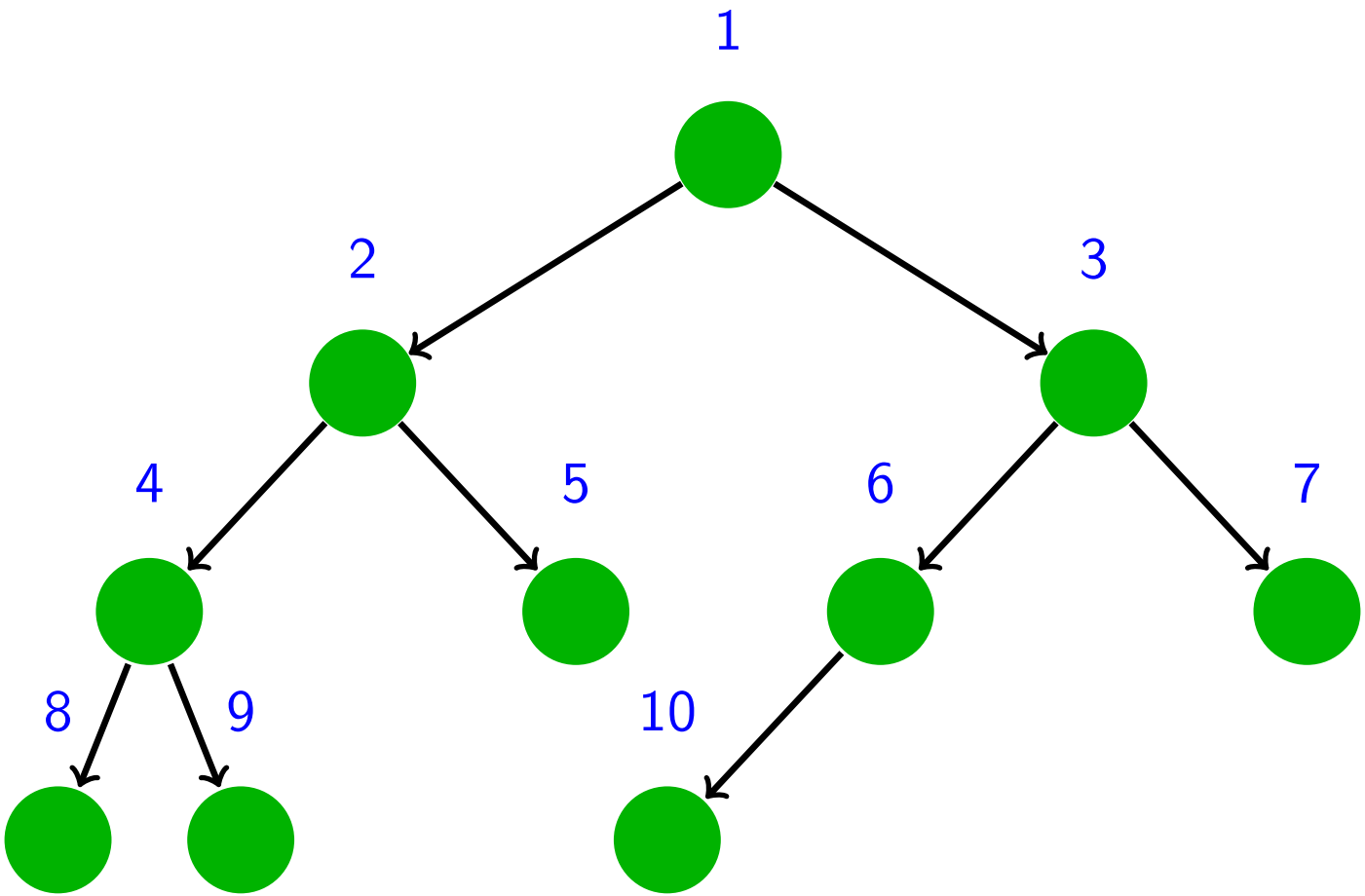


# Example (1/2)

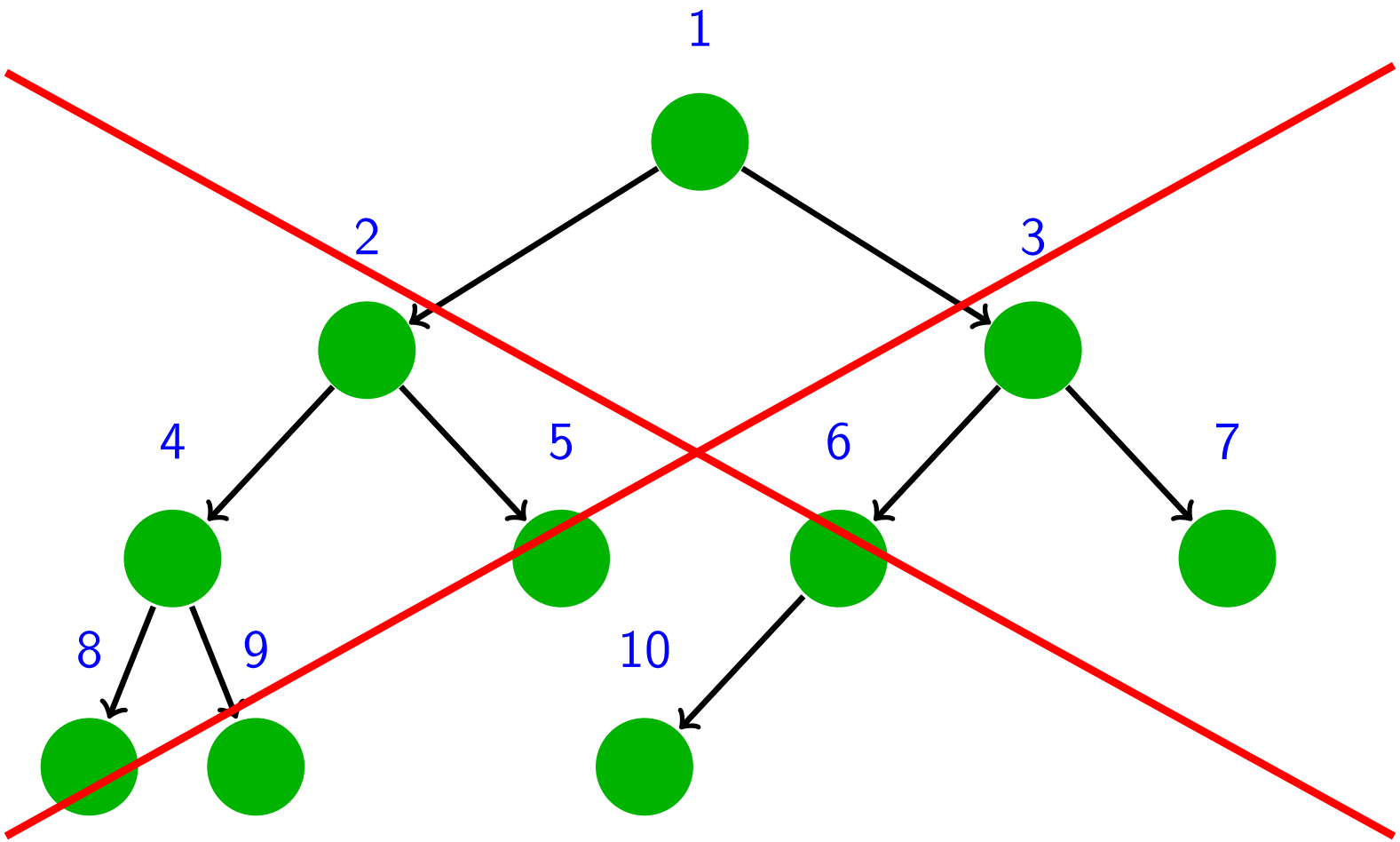



It is not completely filled at level 2

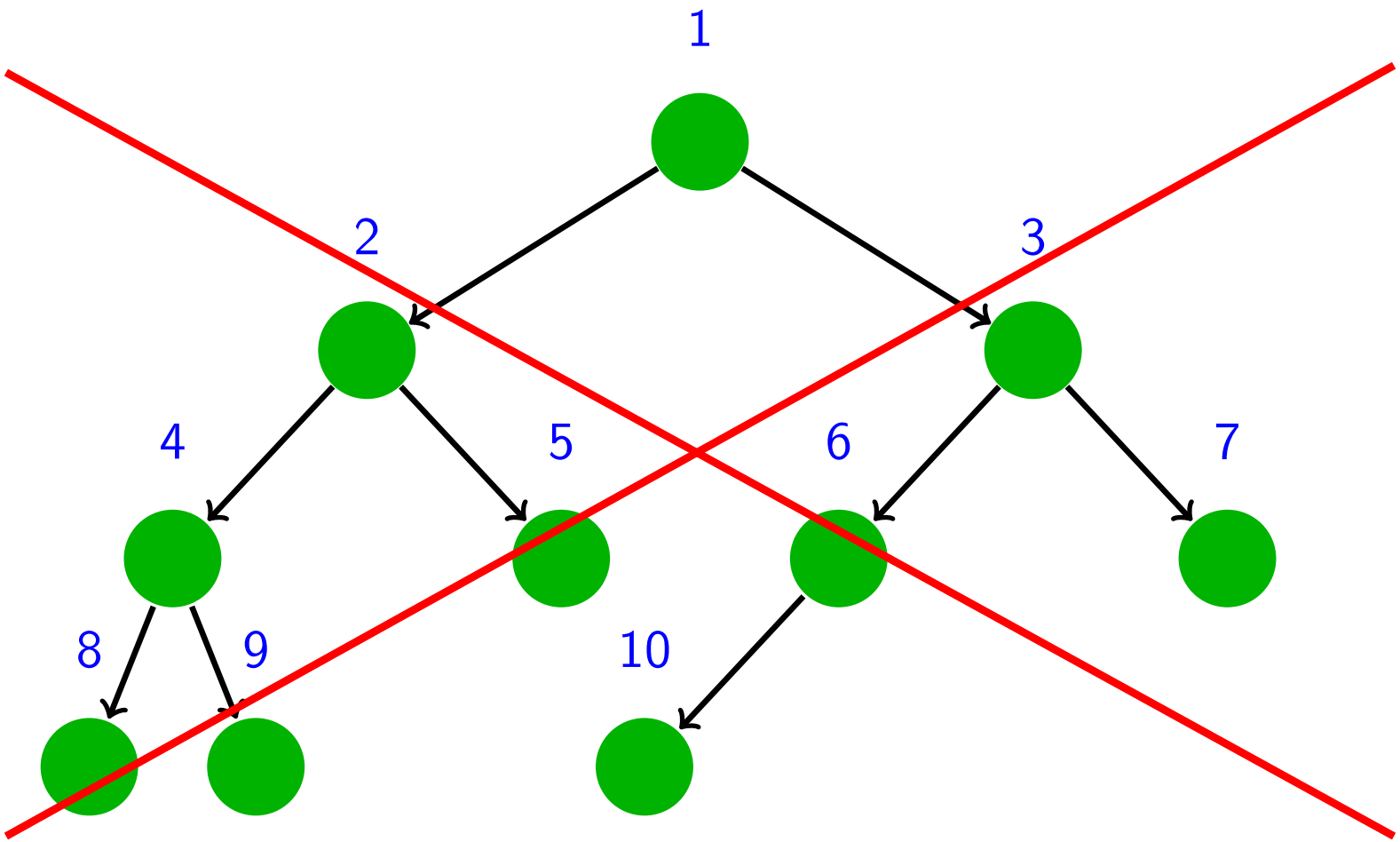
# Example (2/2)



# Example (2/2)



# Example (2/2)

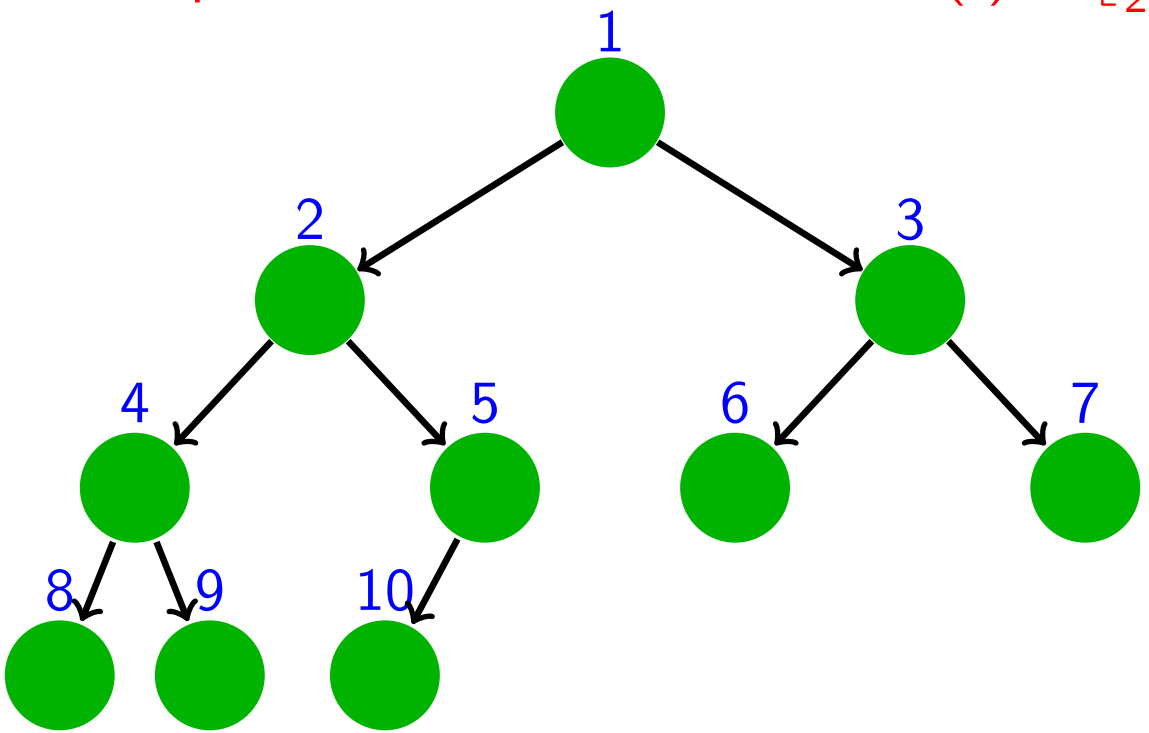



The leaves are not filled from left to right.

# Well-Indexed Tree

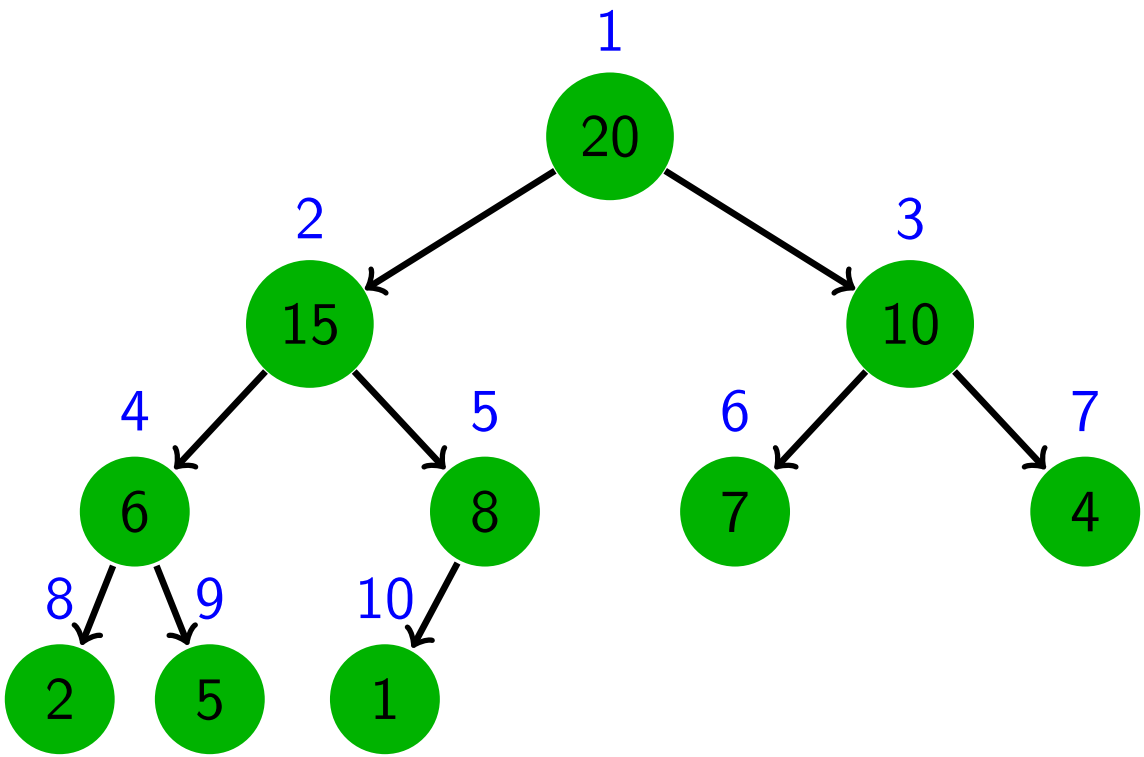
A well-indexed tree is a complete binary tree such that:

- The index of the root is 1
- The index of the left child of a node $i$ is $\textsc{Left}(i) = 2i$
- The index of the right child of a node $i$ is $\textsc{Right}(i) = 2i + 1$
- The index of the parent of a node $i$ is $\textsc{Parent}(i) = \lfloor \frac{i}{2} \rfloor$



# Heap: Definition

- A heap is a well-indexed tree such that :
  - For each node, we associate a value.



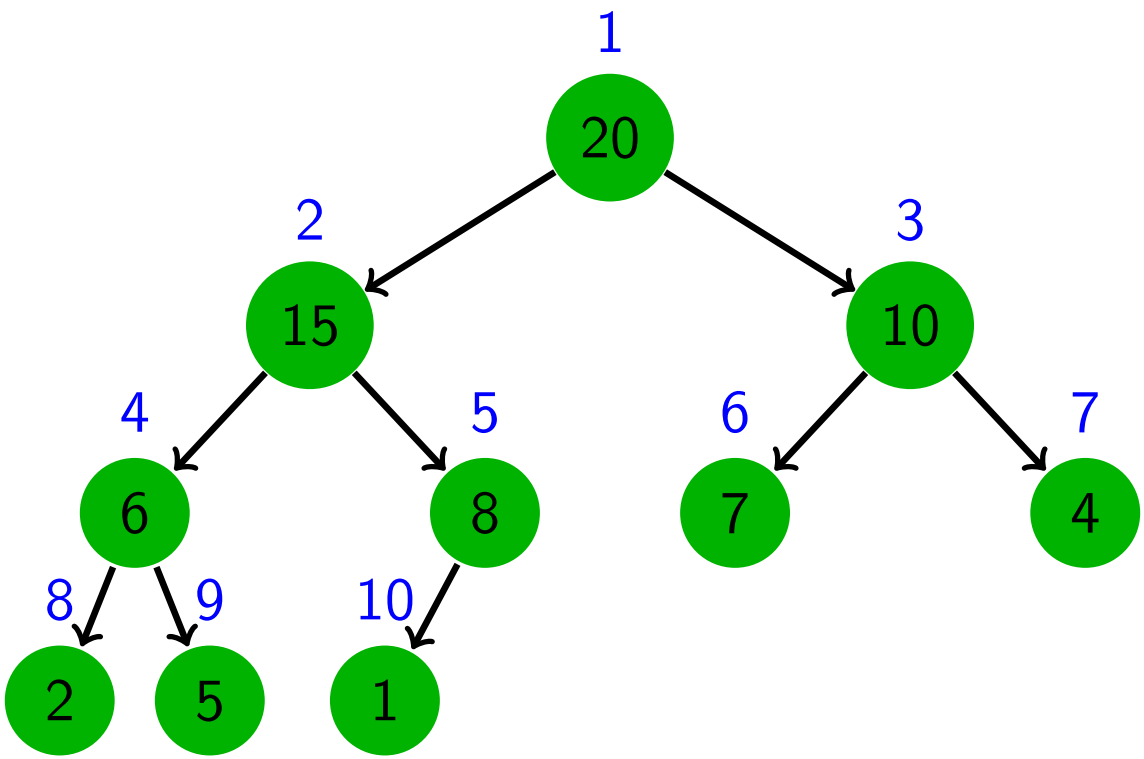
# Heap: Properties

- Let $T$ be a complete tree with $n$ is the number of nodes and $h$ is its height:
  - $n$ is greater or equal to the number of nodes in the perfect tree of height $h-1$ plus one (i.e., $n \geq 2^h$)
  - $n$ is less or equal than the number of nodes in the perfect tree of height $h$ (i.e., $n \leq 2^{h+1}-1$)

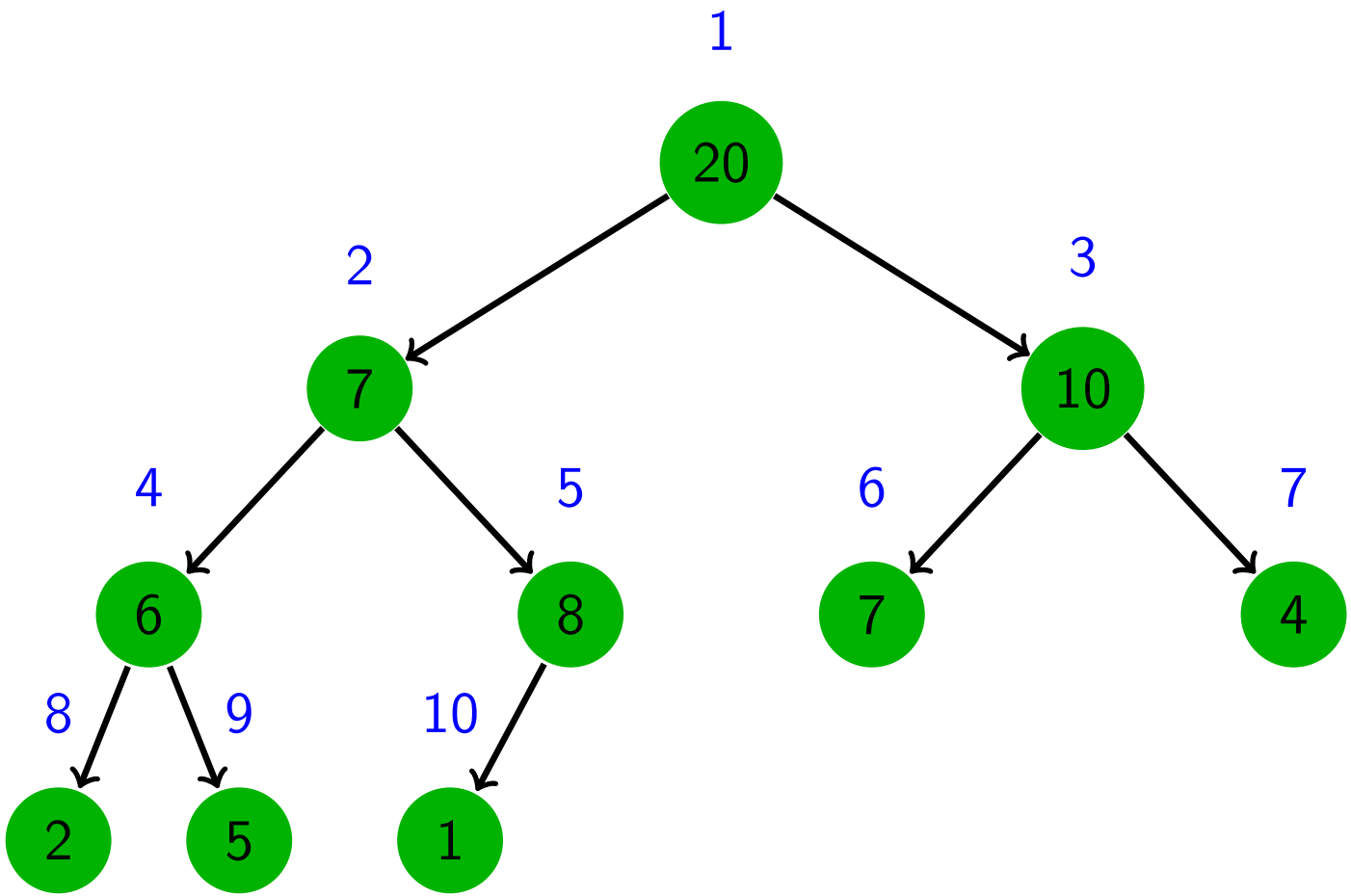
$$
\begin{aligned}
2^h \leq n \leq 2^{h+1}-1 \quad &\Rightarrow \quad 2^h \leq n < 2^{h+1} \\
&\Rightarrow \quad h \leq log_2(n) < h+1 \\
&\Rightarrow \quad h \leq log_2(n) < h+1 \\
&\Rightarrow \quad h = \lfloor log_2(n) \rfloor
\end{aligned}
$$

# Max-Heap: Definition

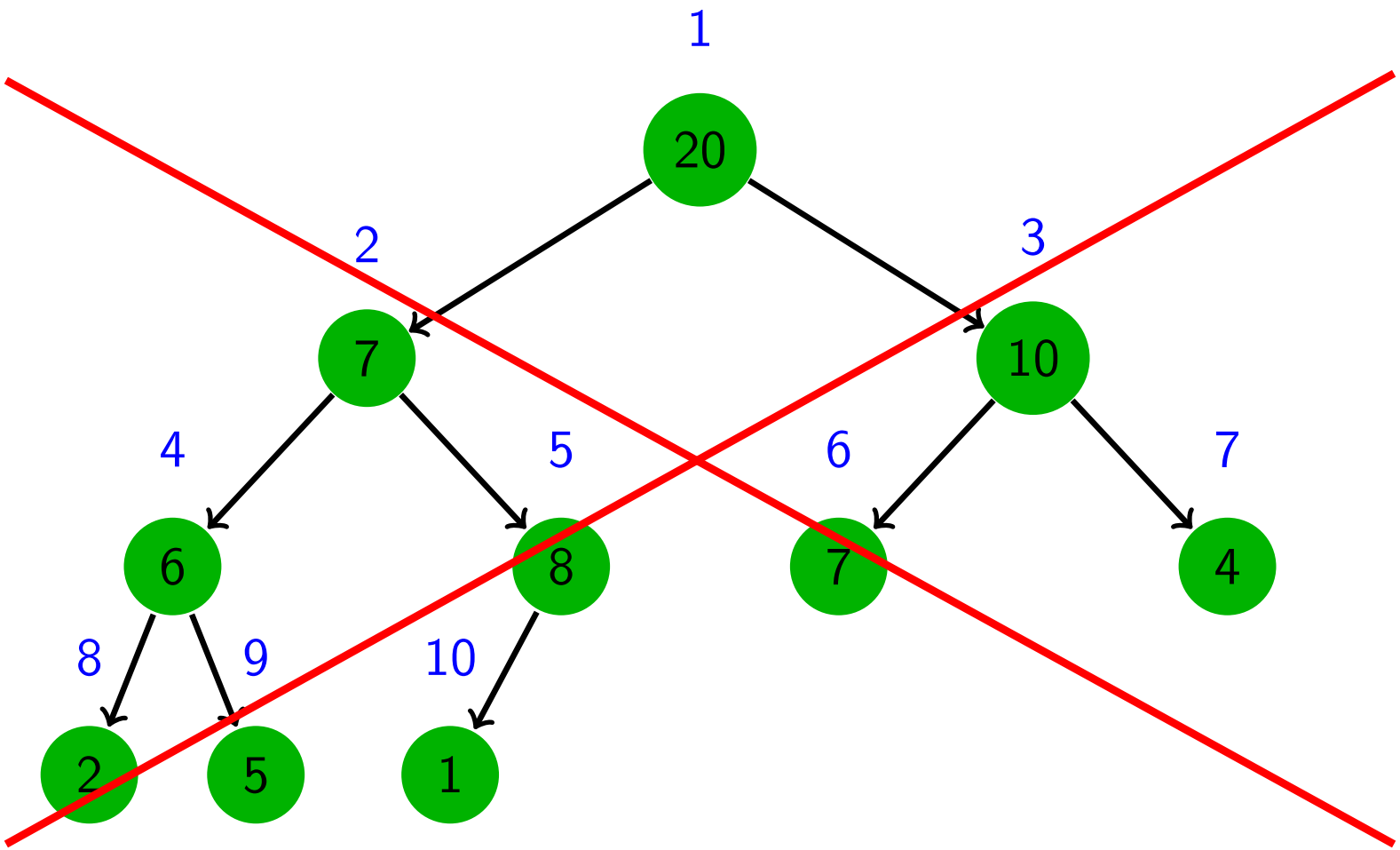
- A max-heap is a heap such that :
  - The value of node is at most the value of its parent.



# Max-Heap

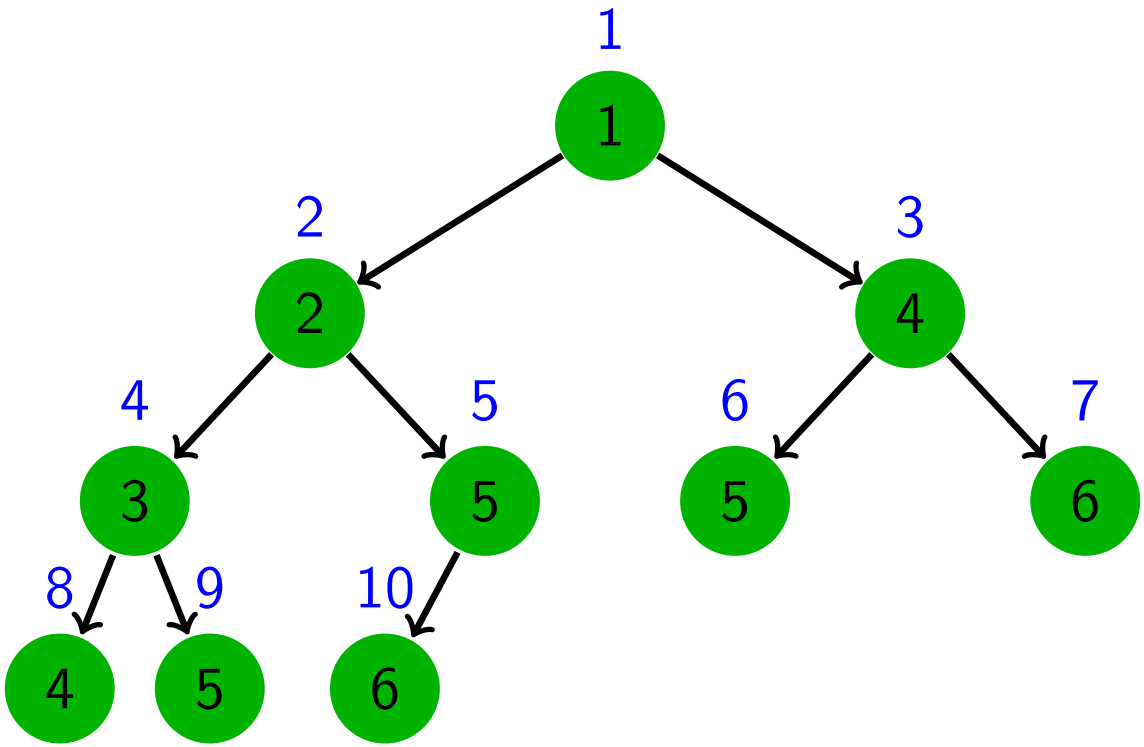


# Max-Heap

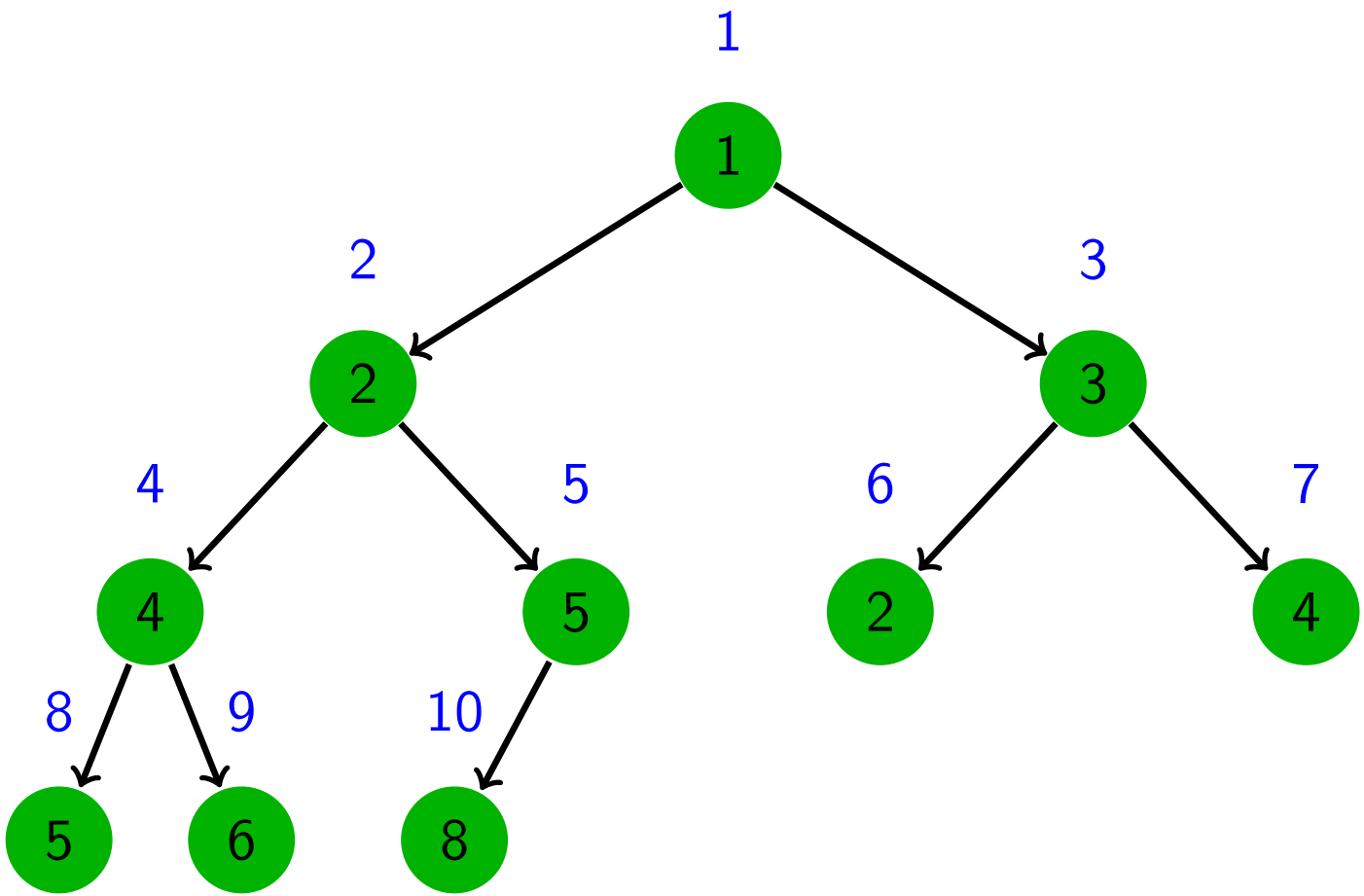


# Min-Heap: Definition

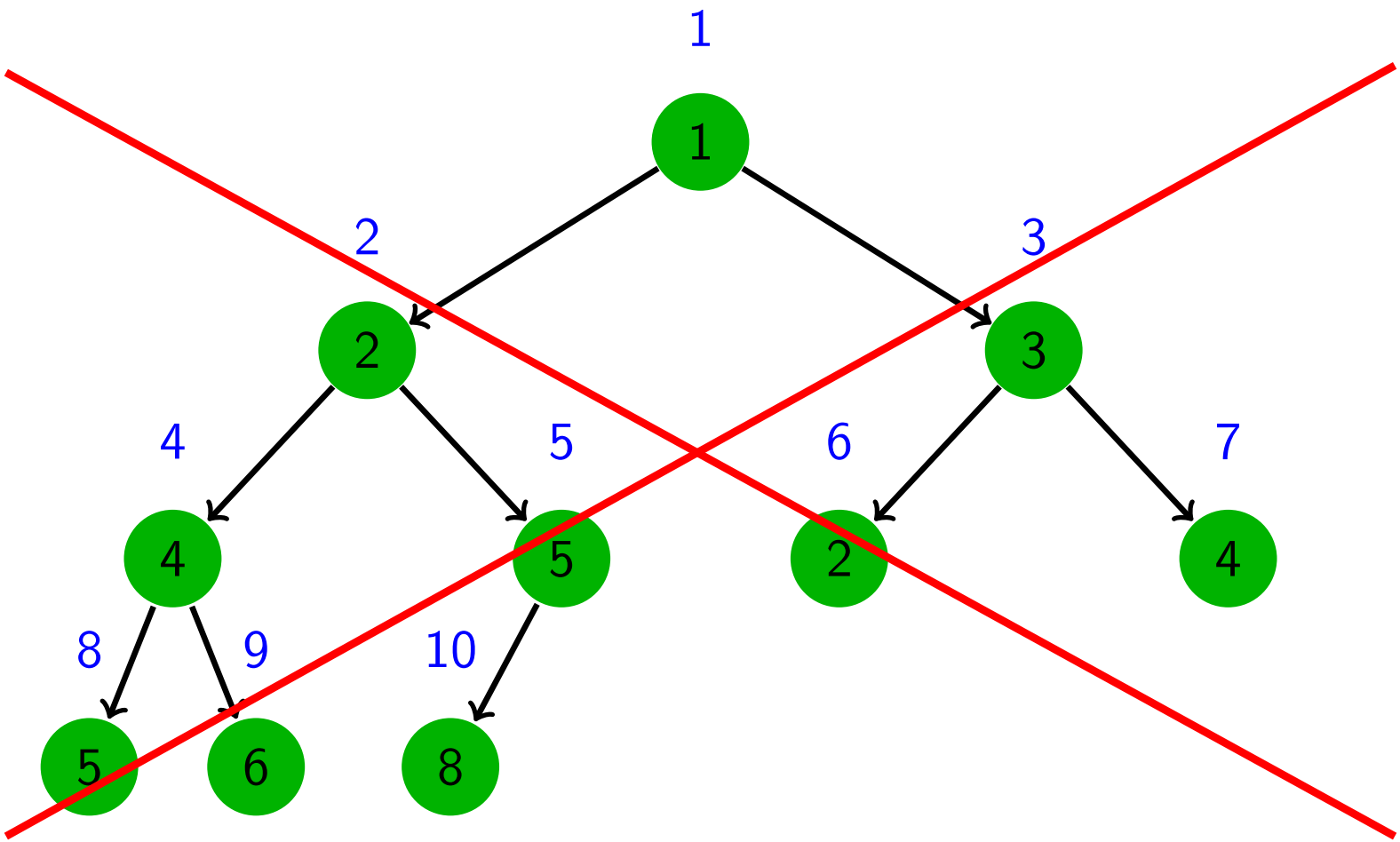
- A min-heap is a heap such that :
  - The value of node is at least the value of its parent.



# Min-Heap

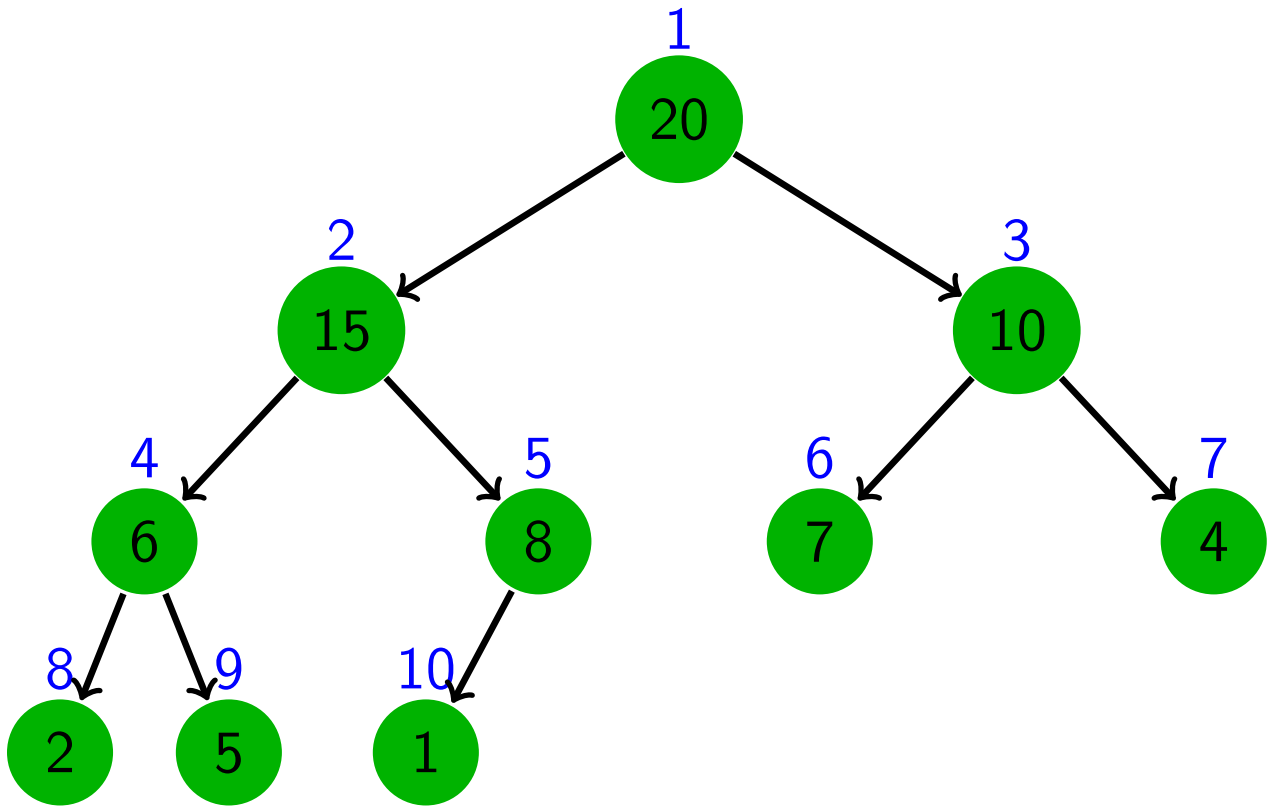


# Min-Heap



# Implementation of a Heap

A heap can be represented as an array $A$ such that the value of a node of index $i$ is $A[i]$.




A.*heap-size*=10

A.*length*=13

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 15 | 10 | 6 | 8 | 7 | 4 | 2 | 5 | 1 | 9 | 14 | 13 |

# Implementation of a Heap

An array A representing a heap has two attributes:

- A.length: The length of the array
- A.heap-size: length of the left subarray containing elements from the heap= number of nodes inside the heap.

A property of the array A representing a max-heap:

- For all $i : 2 \leq i \leq A.\,heap\text{-}size$, we have $A[\textsc{Parent}(i)] \geq A[i]$

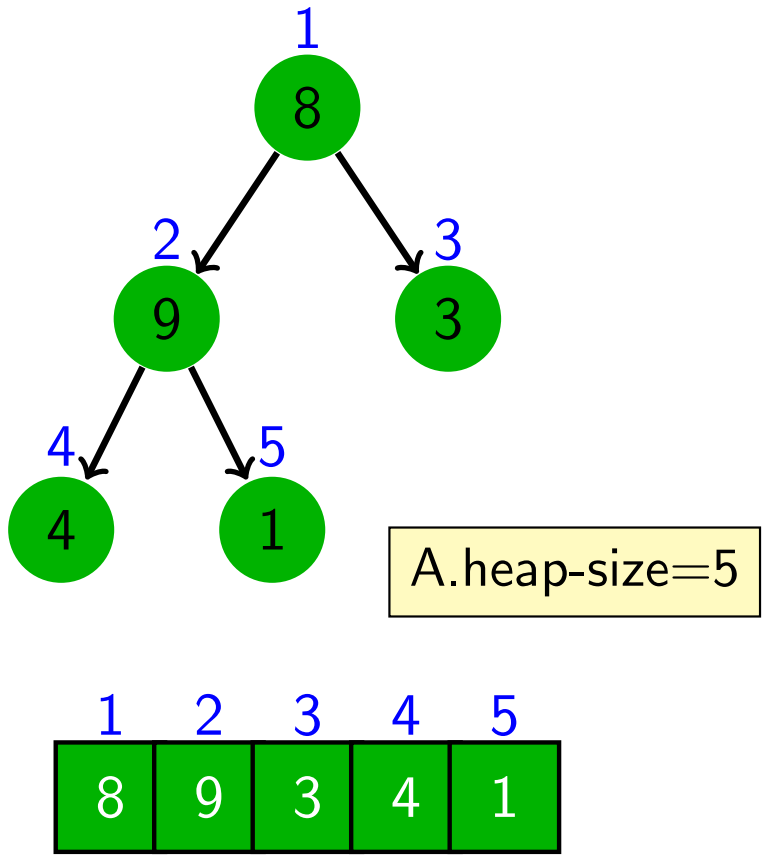
A property of the array A representing a min-heap:

- For all $i : 2 \leq i \leq A.\,heap\text{-}size$, we have $A[\textsc{Parent}(i)] \leq A[i]$

# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

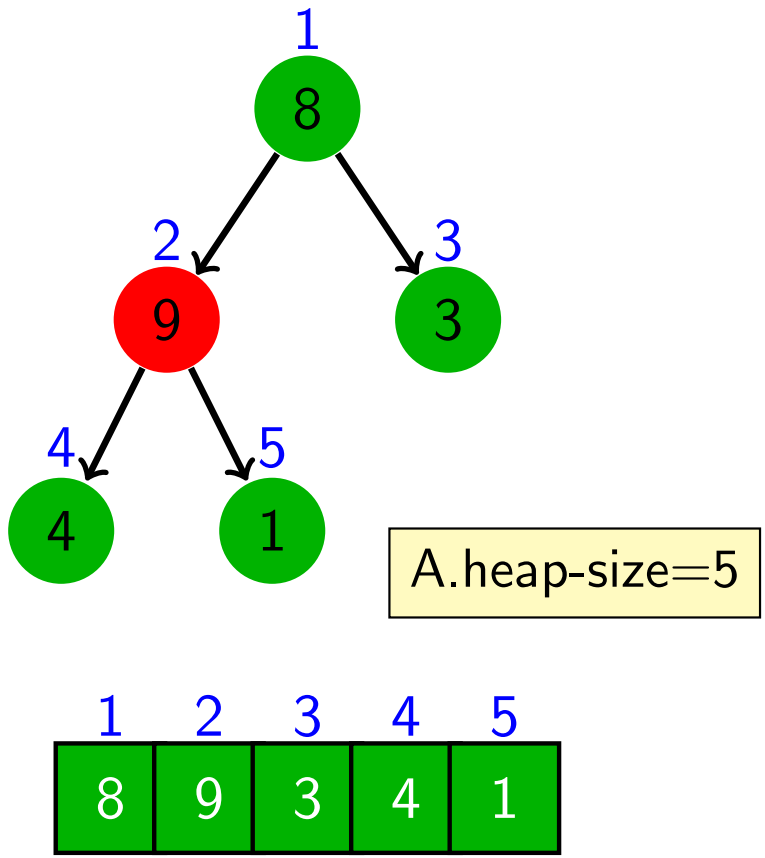


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))

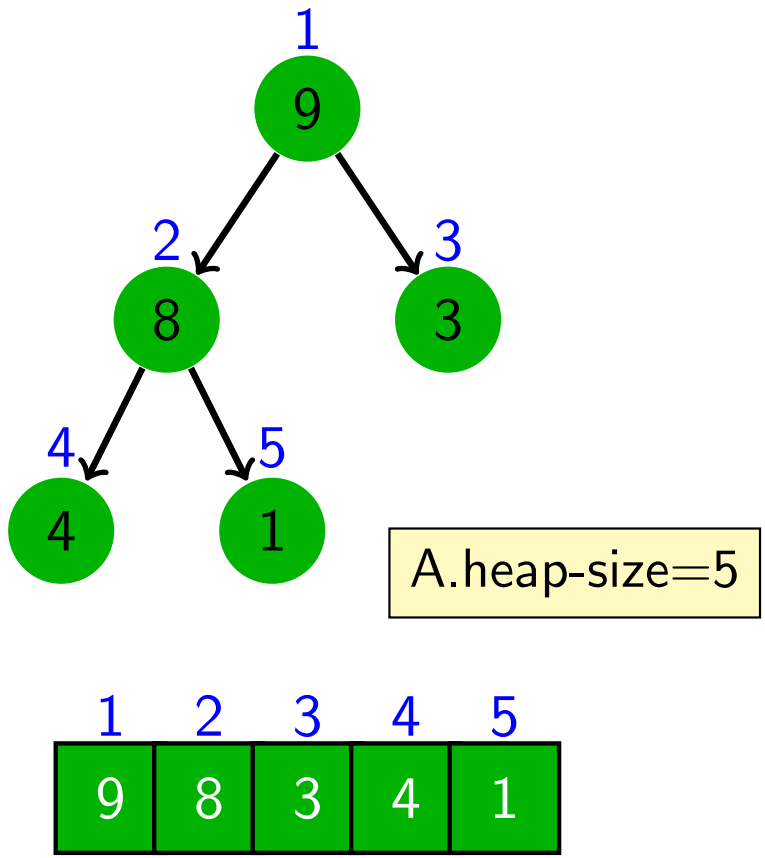


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))

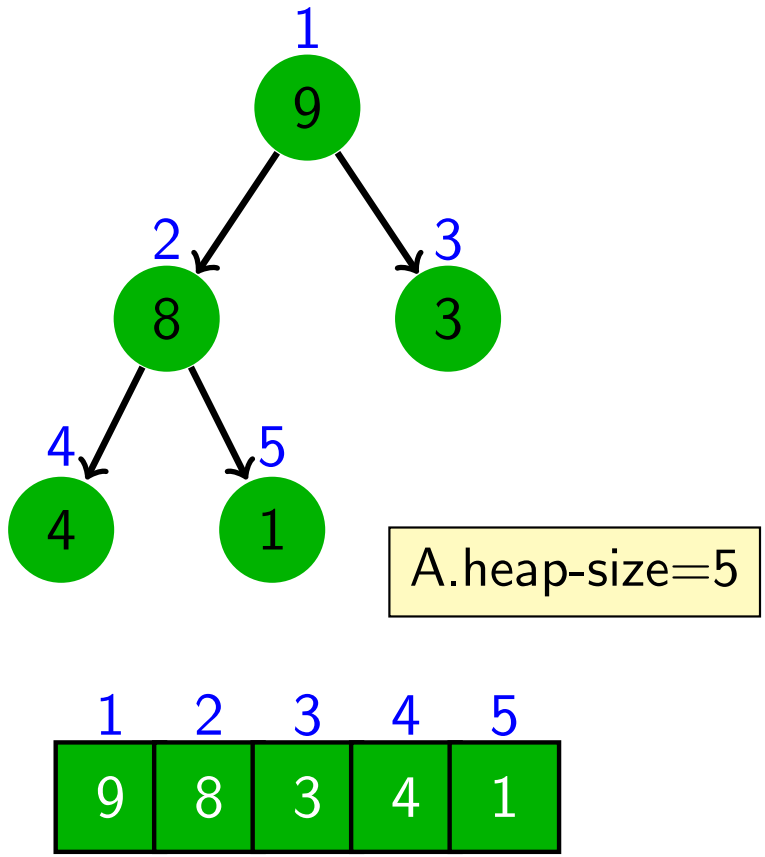


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:

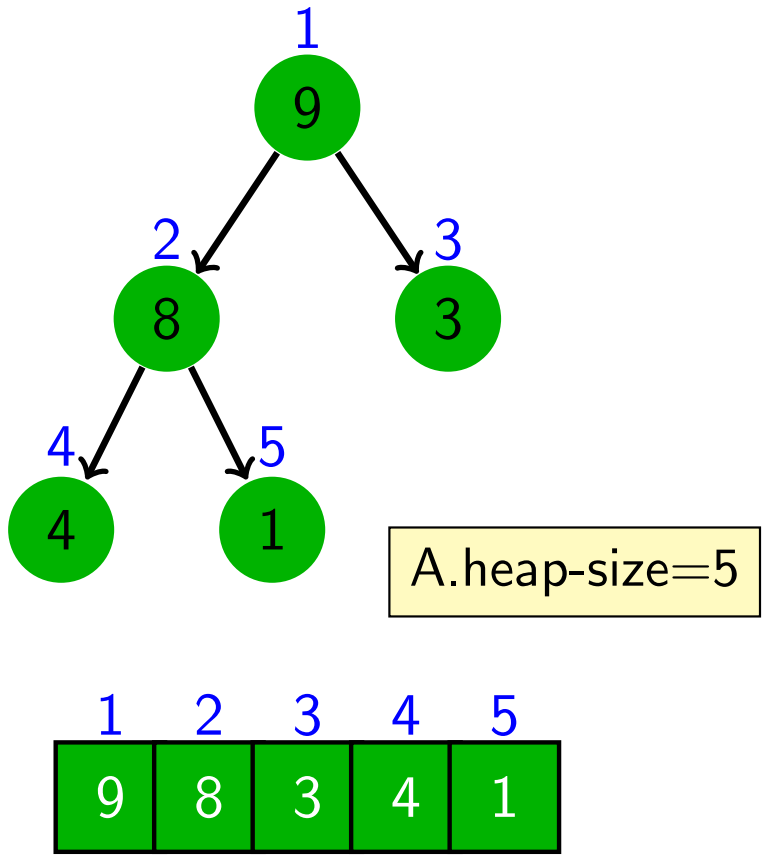


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )

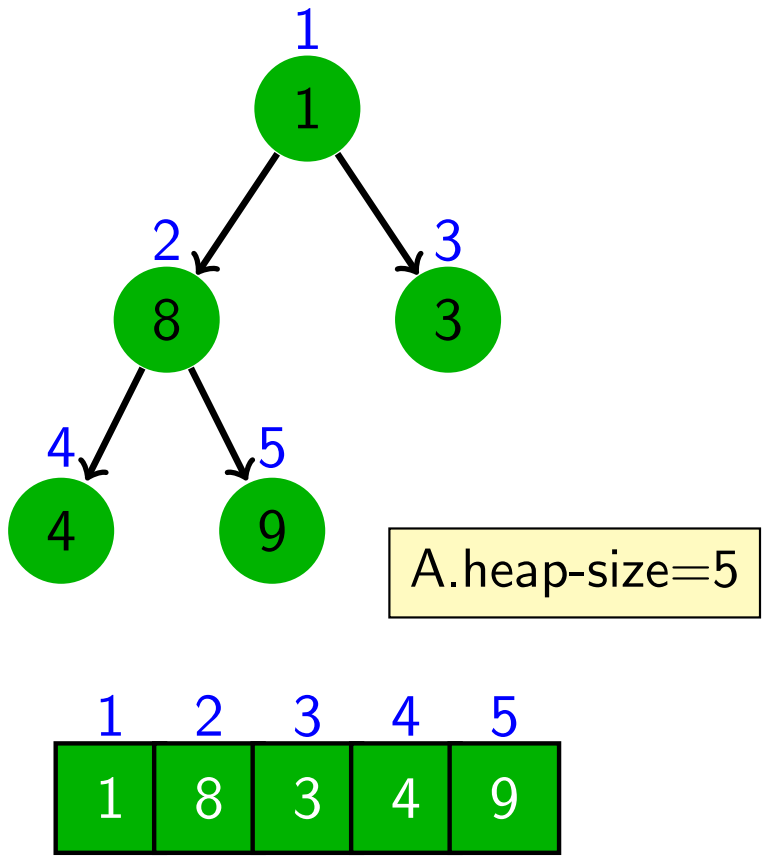


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )

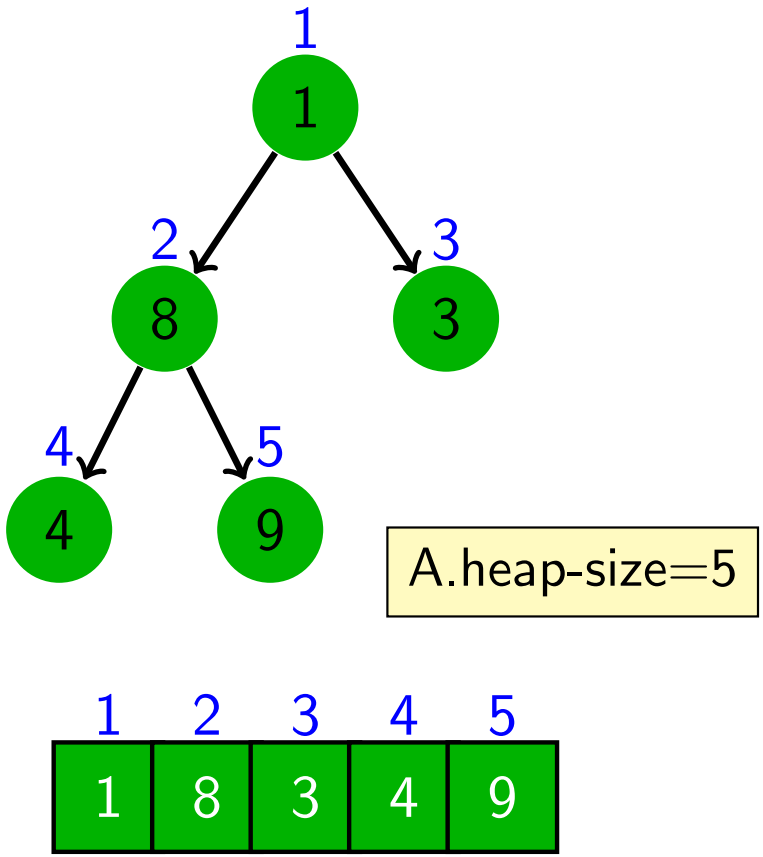


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size

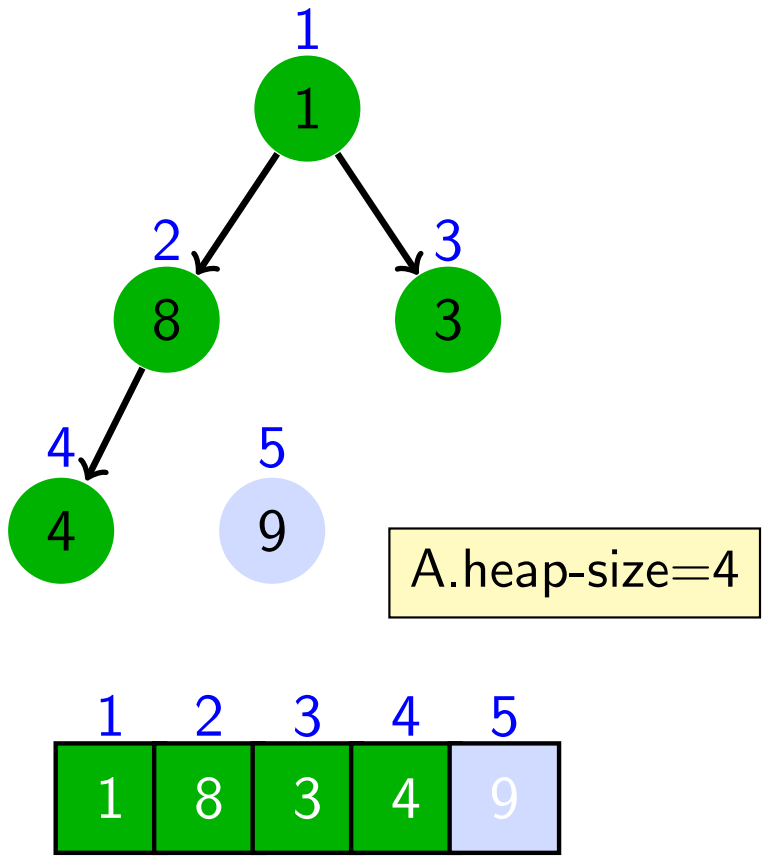


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size

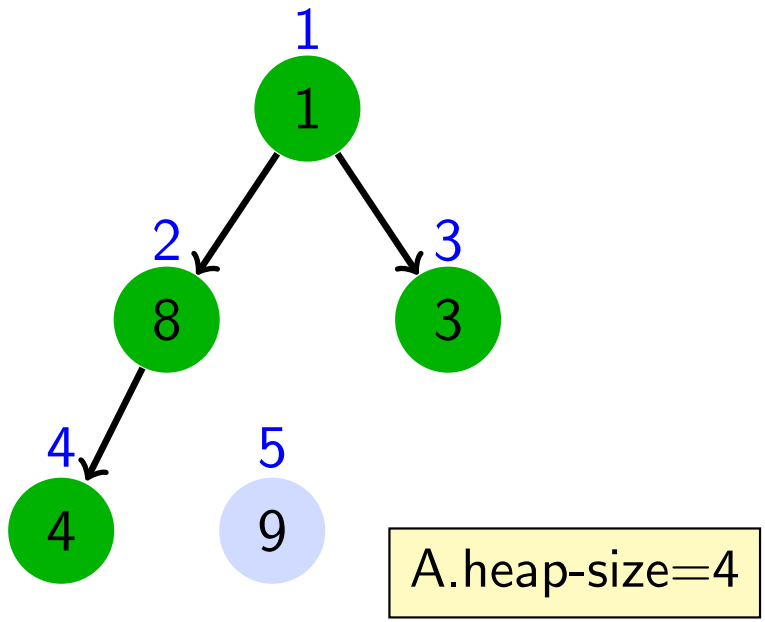


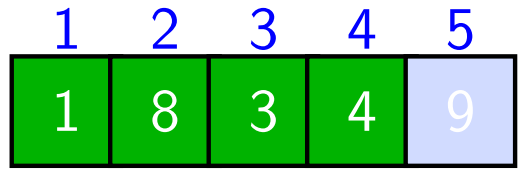
# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))



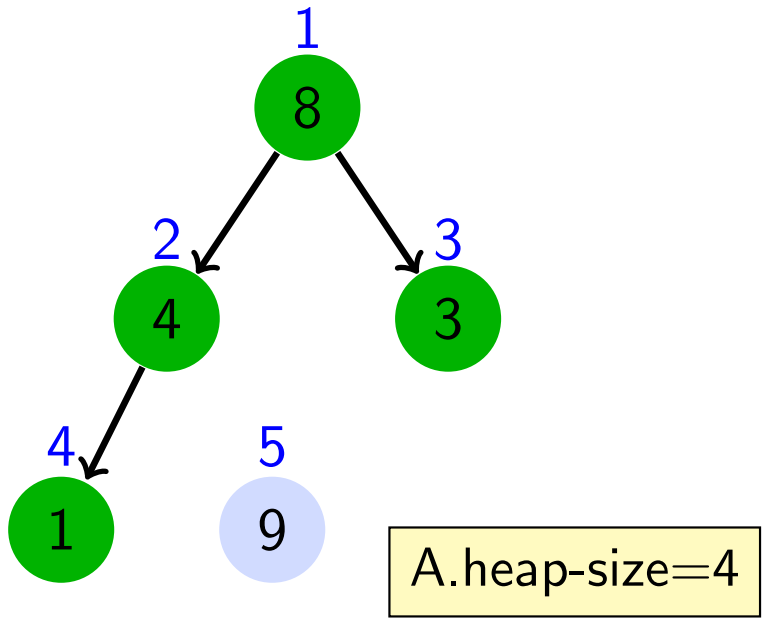



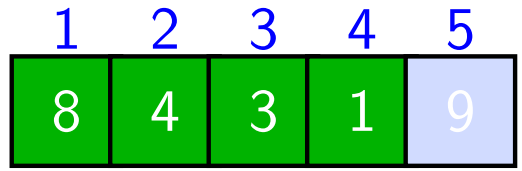
# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))



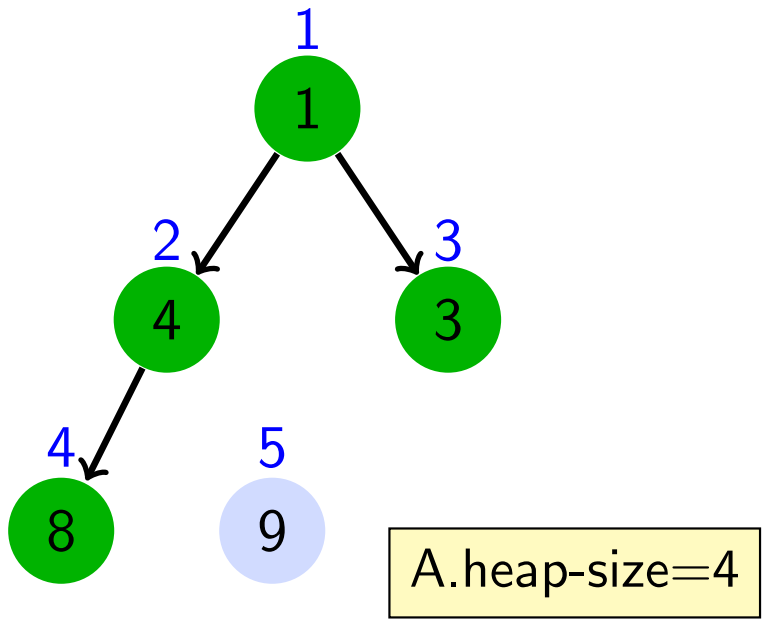



# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))

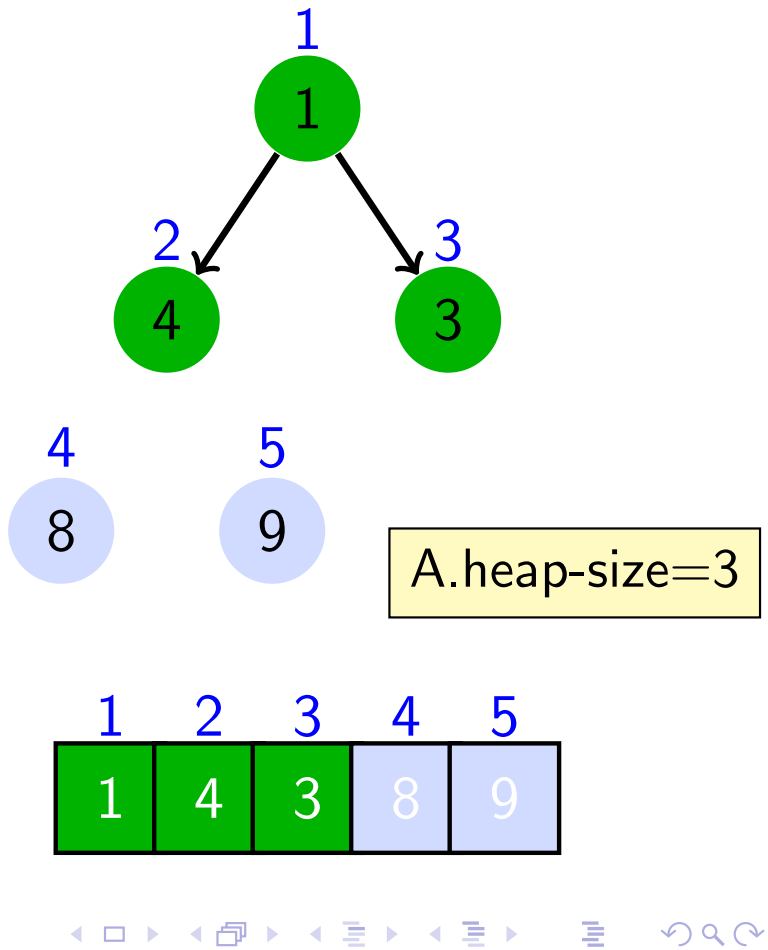


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))

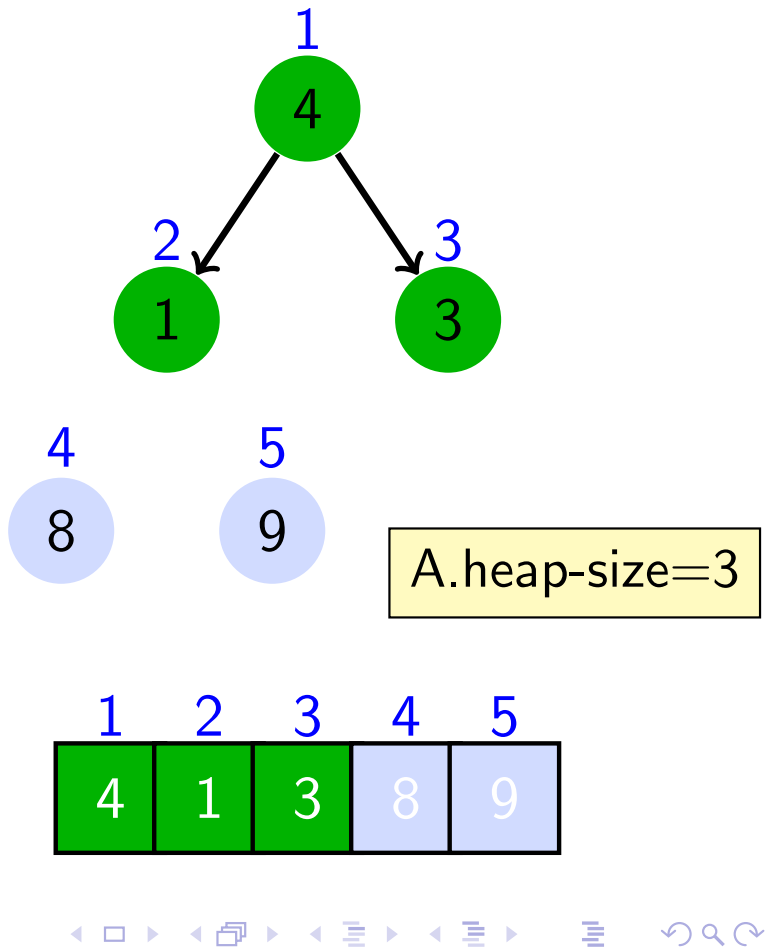


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))

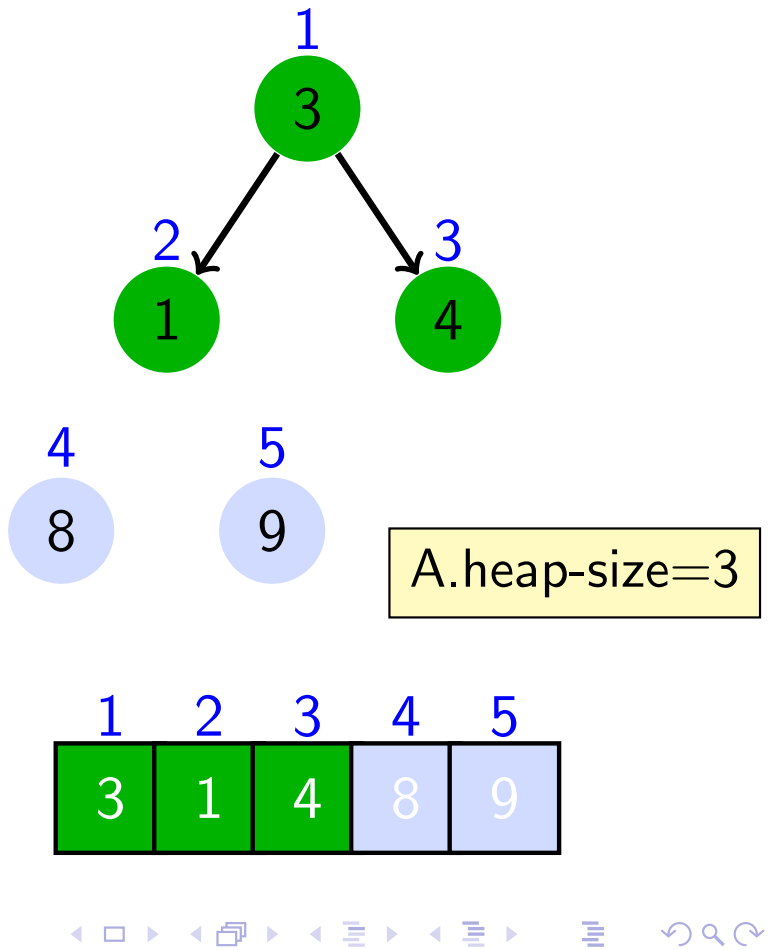


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))

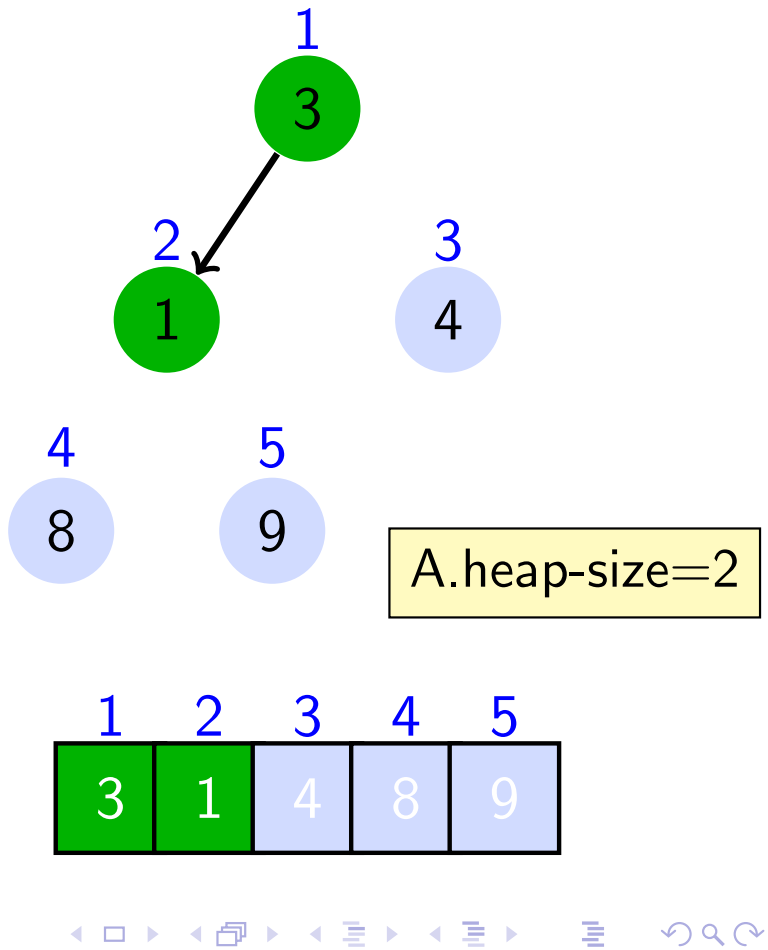


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))

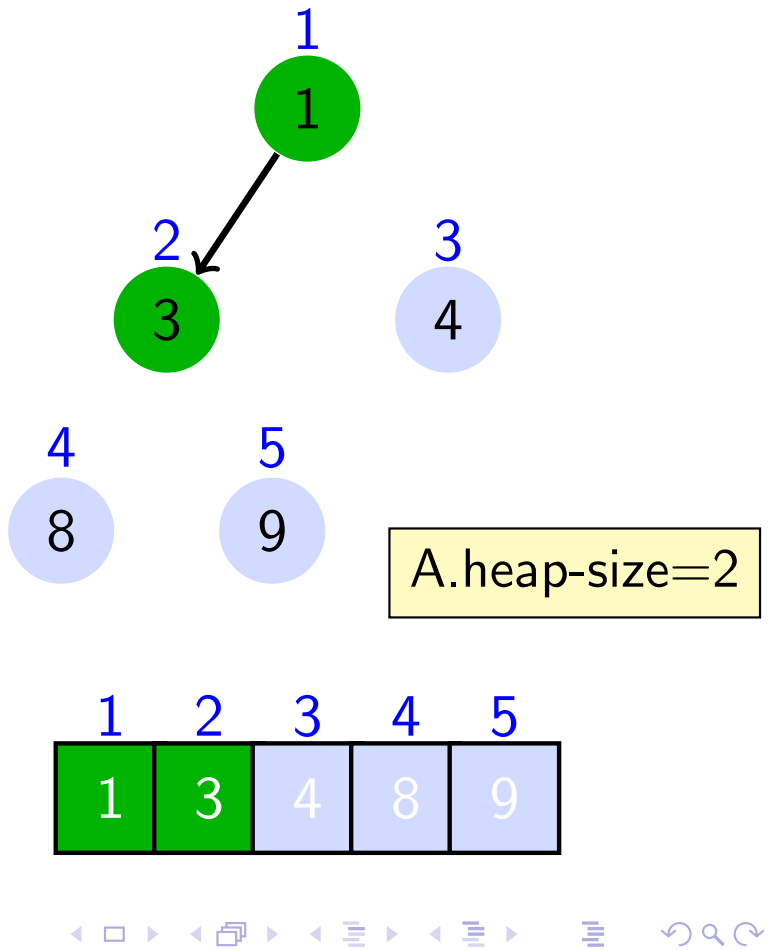


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))

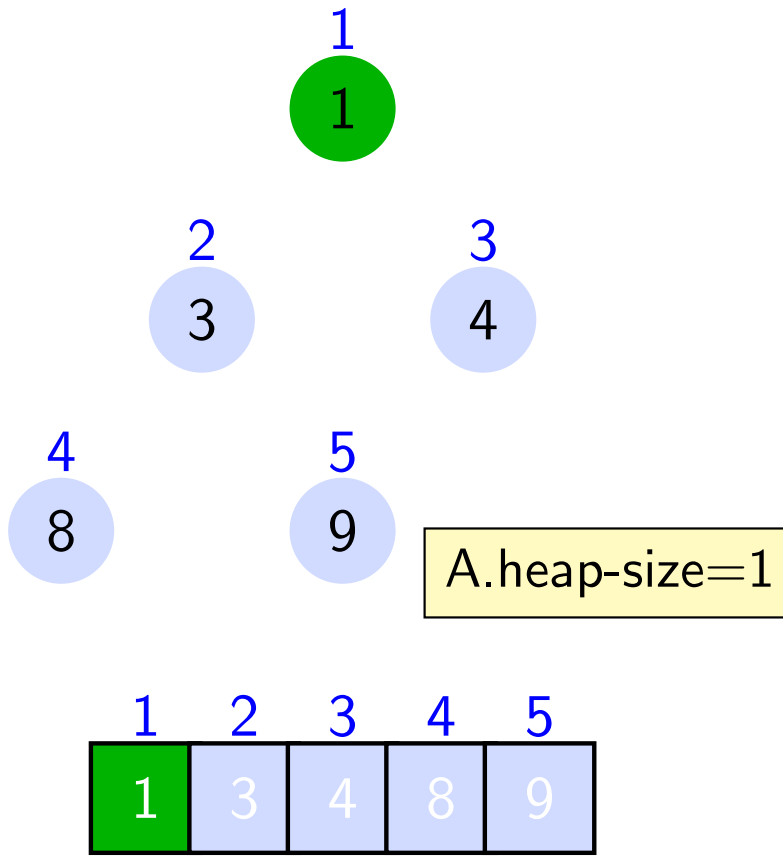


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))

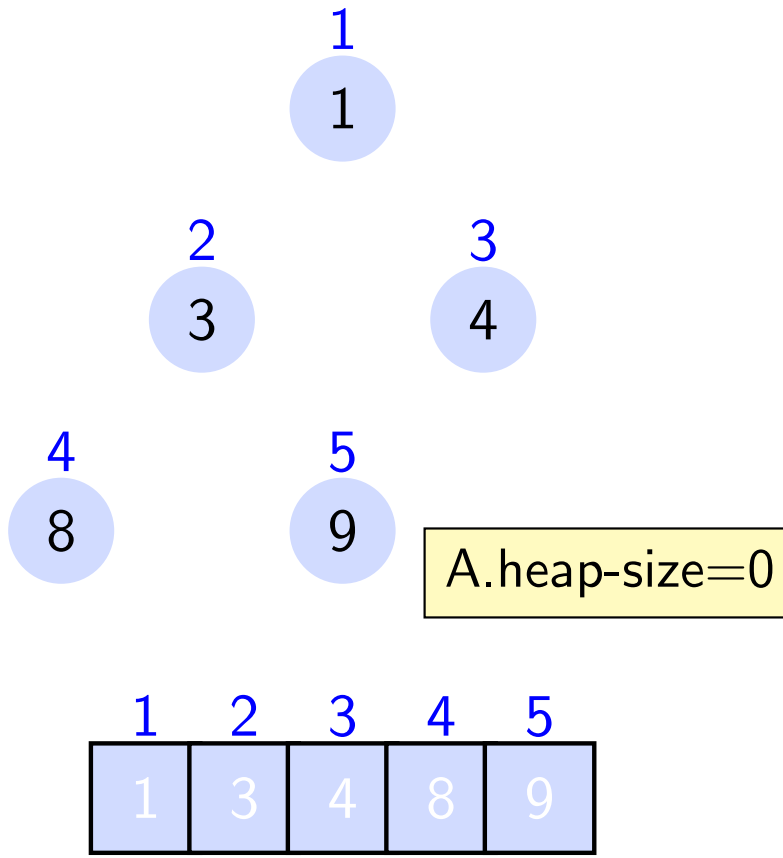


# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

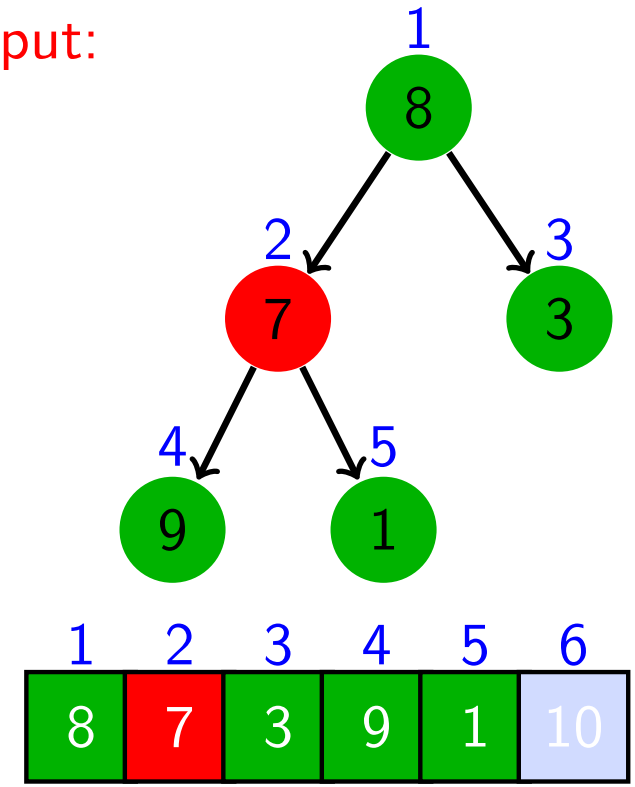
- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))



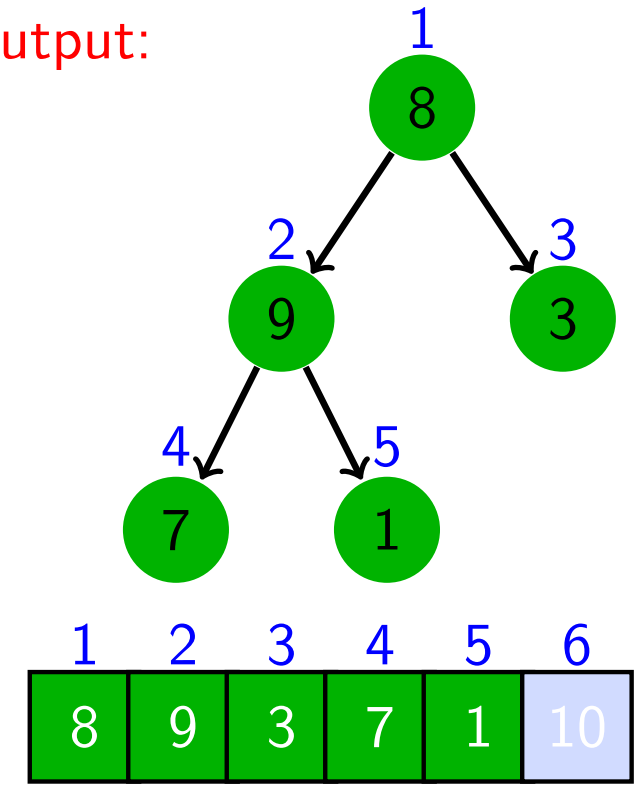
# MAX-HEAPIFY

- Input: An array $A$ and an index $i : 1 \leq i \leq A.heap - size$
- Assumption: The two sub-trees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are max-heaps
- Output: The sub-tree rooted at index $i$ is a max-heap.

Input:



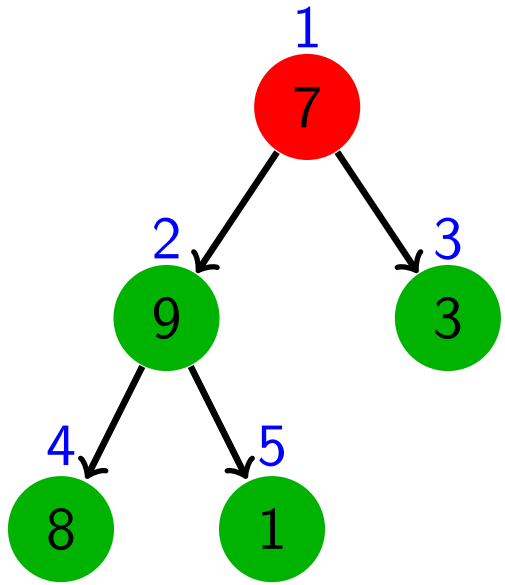

Output:

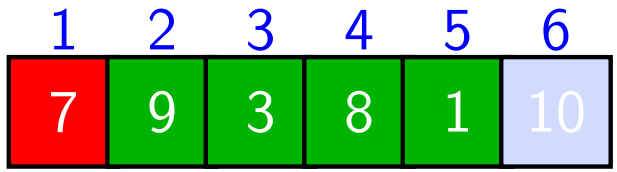


# Max-Heapify: Principle

## Max-Heapify

- Compare $A[i]$, $A[\textsc{Left}(i)]$, and $A[\textsc{Right}(i)]$

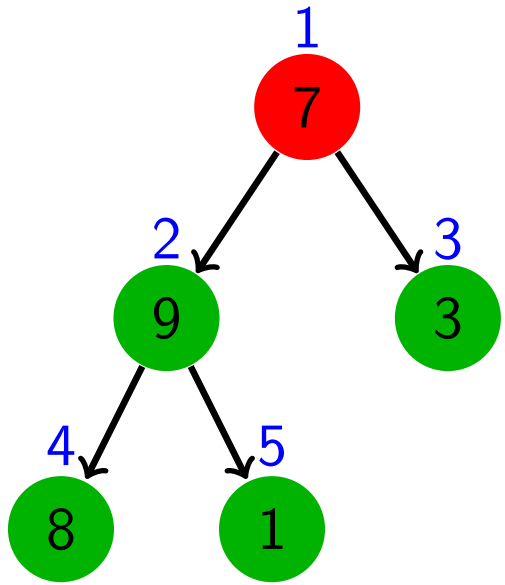


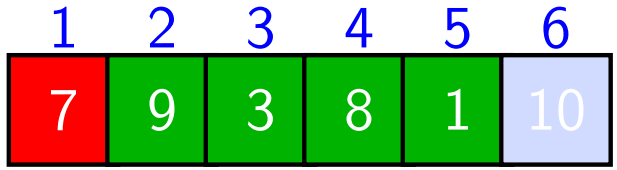



# Max-Heapify: Principle

## Max-Heapify

- Compare $A[i]$, $A[\textsc{Left}(i)]$, and $A[\textsc{Right}(i)]$
- If $A[i] < A[\textsc{Left}(i)]$ or $A[i] < A[\textsc{Right}(i)]$, swap $A[i]$ with the larger of the two children

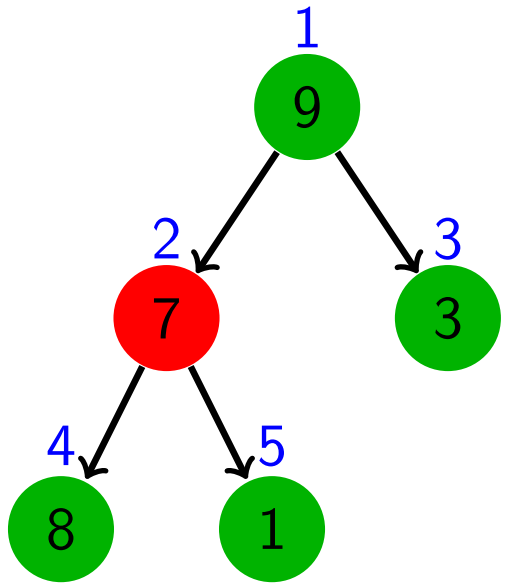


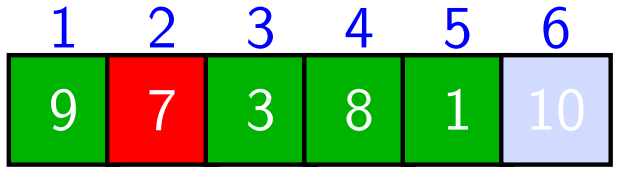



# Max-Heapify: Principle

## Max-Heapify

- Compare $A[i]$, $A[\textsc{Left}(i)]$, and $A[\textsc{Right}(i)]$
- If $A[i] < A[\textsc{Left}(i)]$ or $A[i] < A[\textsc{Right}(i)]$, swap $A[i]$ with the larger of the two children

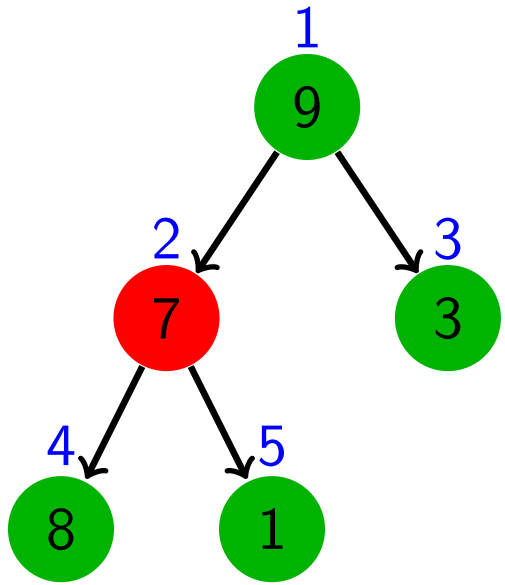


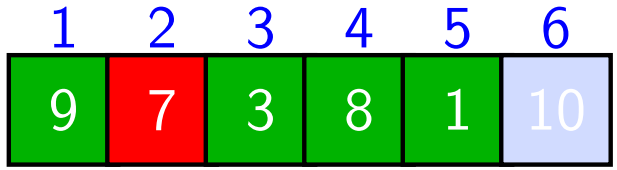



# Max-Heapify: Principle

## Max-Heapify

- Compare $A[i]$, $A[\textsc{Left}(i)]$, and $A[\textsc{Right}(i)]$
- If $A[i] < A[\textsc{Left}(i)]$ or $A[i] < A[\textsc{Right}(i)]$, swap $A[i]$ with the larger of the two children
- Continue this process of comparing and swapping down the heap, until subtree rooted at $i$ is a max-heap

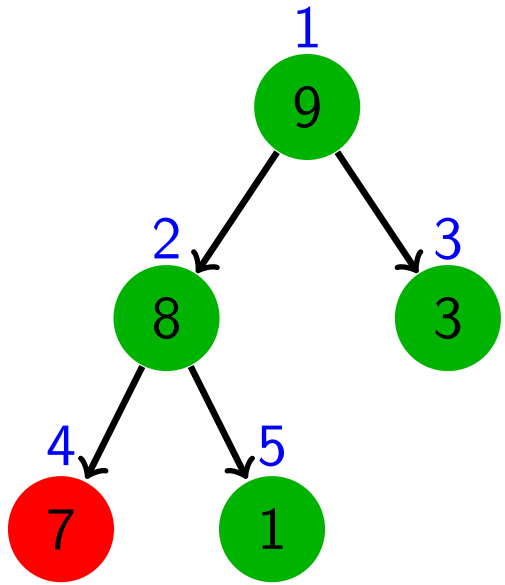


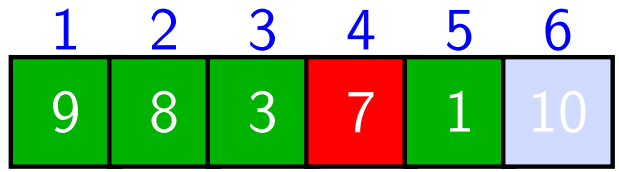



# Max-Heapify: Principle

## Max-Heapify

- Compare $A[i]$, $A[\textsc{Left}(i)]$, and $A[\textsc{Right}(i)]$
- If $A[i] < A[\textsc{Left}(i)]$ or $A[i] < A[\textsc{Right}(i)]$, swap $A[i]$ with the larger of the two children
- Continue this process of comparing and swapping down the heap, until subtree rooted at $i$ is a max-heap

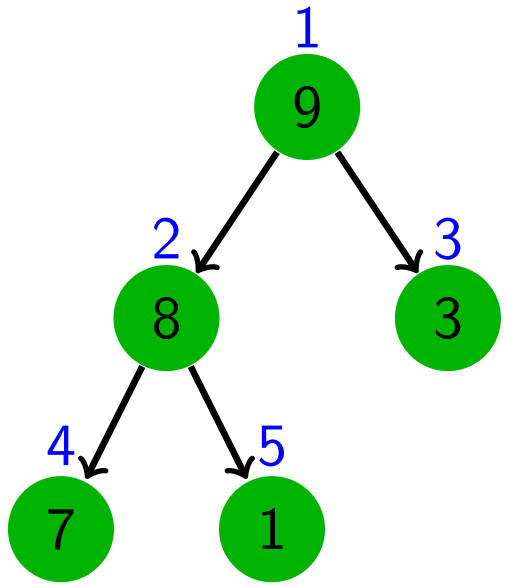


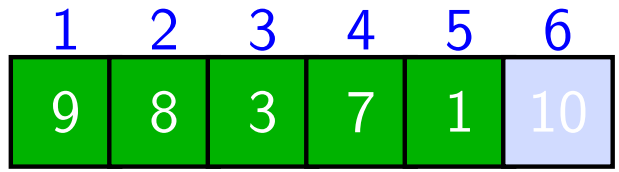



# Max-Heapify: Principle

## Max-Heapify

- Compare $A[i]$, $A[\textsc{Left}(i)]$, and $A[\textsc{Right}(i)]$
- If $A[i] < A[\textsc{Left}(i)]$ or $A[i] < A[\textsc{Right}(i)]$, swap $A[i]$ with the larger of the two children
- Continue this process of comparing and swapping down the heap, until subtree rooted at $i$ is a max-heap



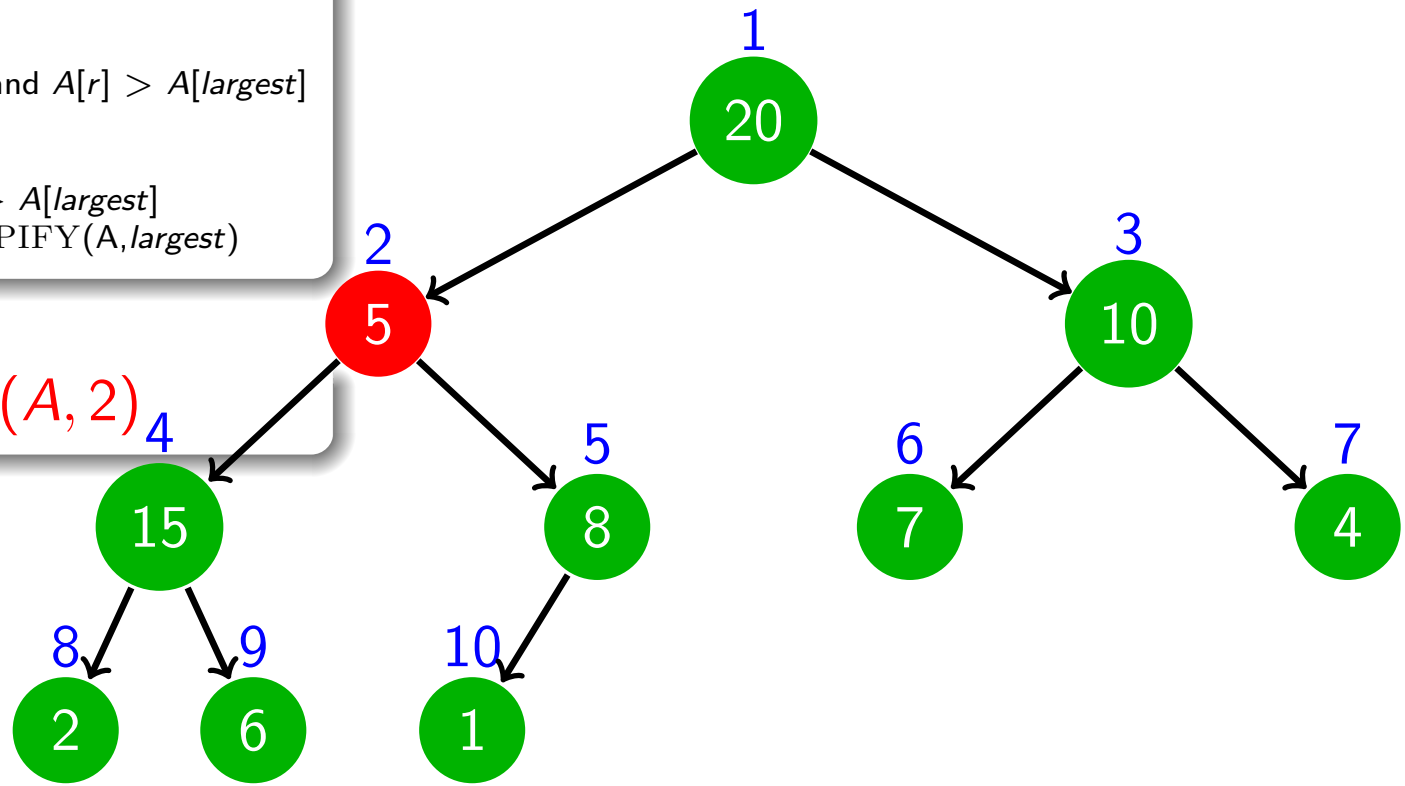



# MAX-HEAPIFY

MAX-HEAPIFY($A, i$)

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

MAX-HEAPIFY($A, 2$)



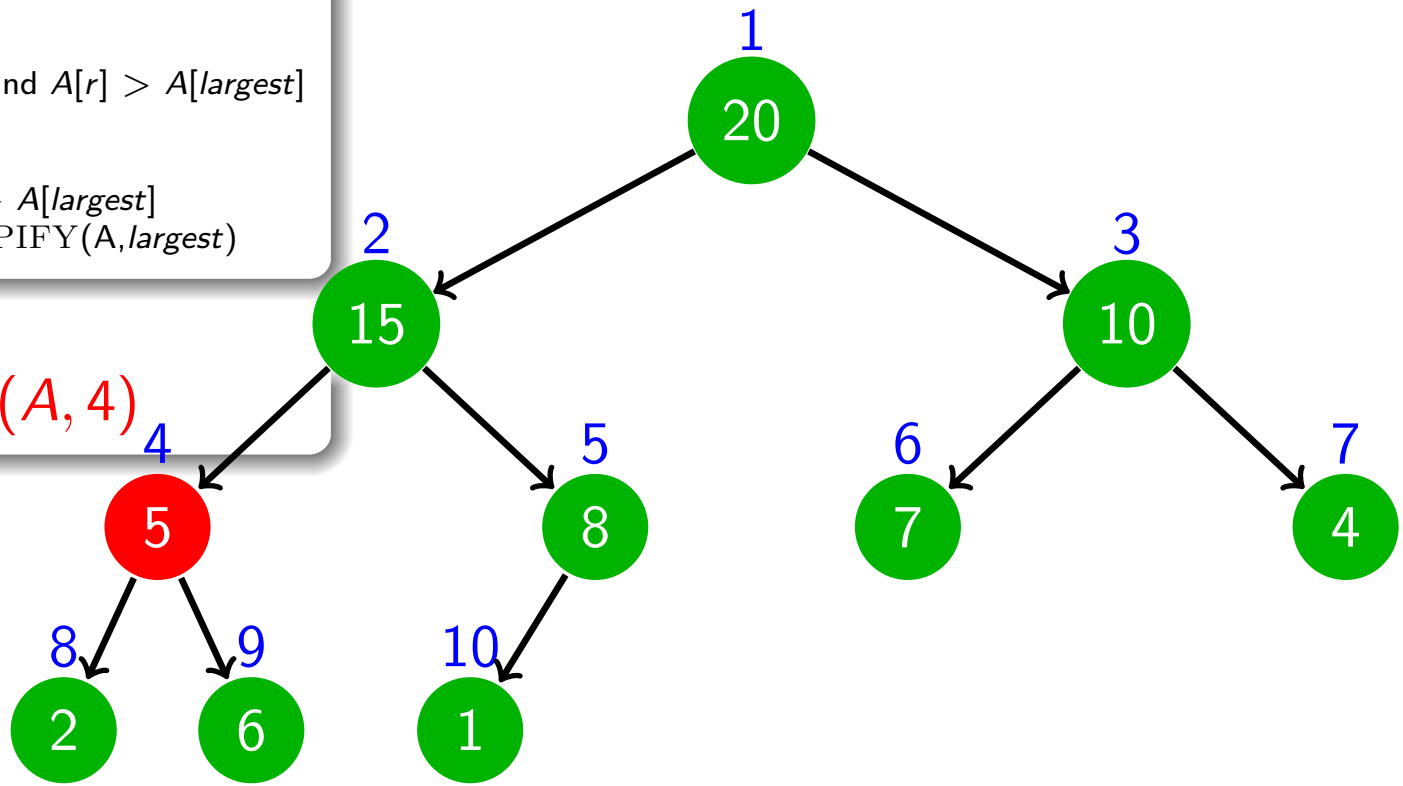

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 5 | 10 | 15 | 8 | 7 | 4 | 2 | 6 | 1 | 21 | 22 | 24 |

# MAX-HEAPIFY

MAX-HEAPIFY($A, i$)

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

MAX-HEAPIFY($A, 4$)



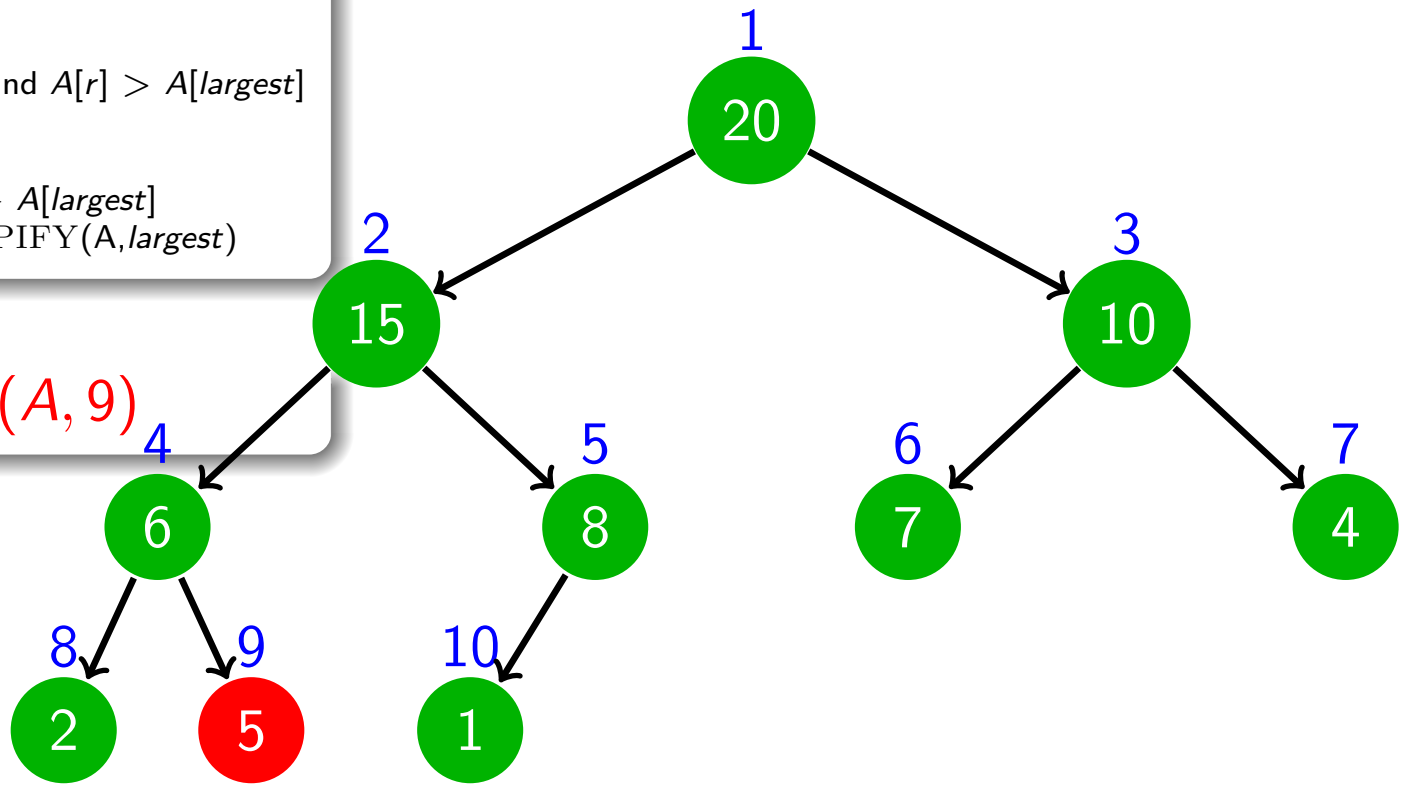

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 15 | 10 | 5 | 8 | 7 | 4 | 2 | 6 | 1 | 21 | 22 | 24 |

# MAX-HEAPIFY

MAX-HEAPIFY$(A, i)$

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

MAX-HEAPIFY$(A, 9)$

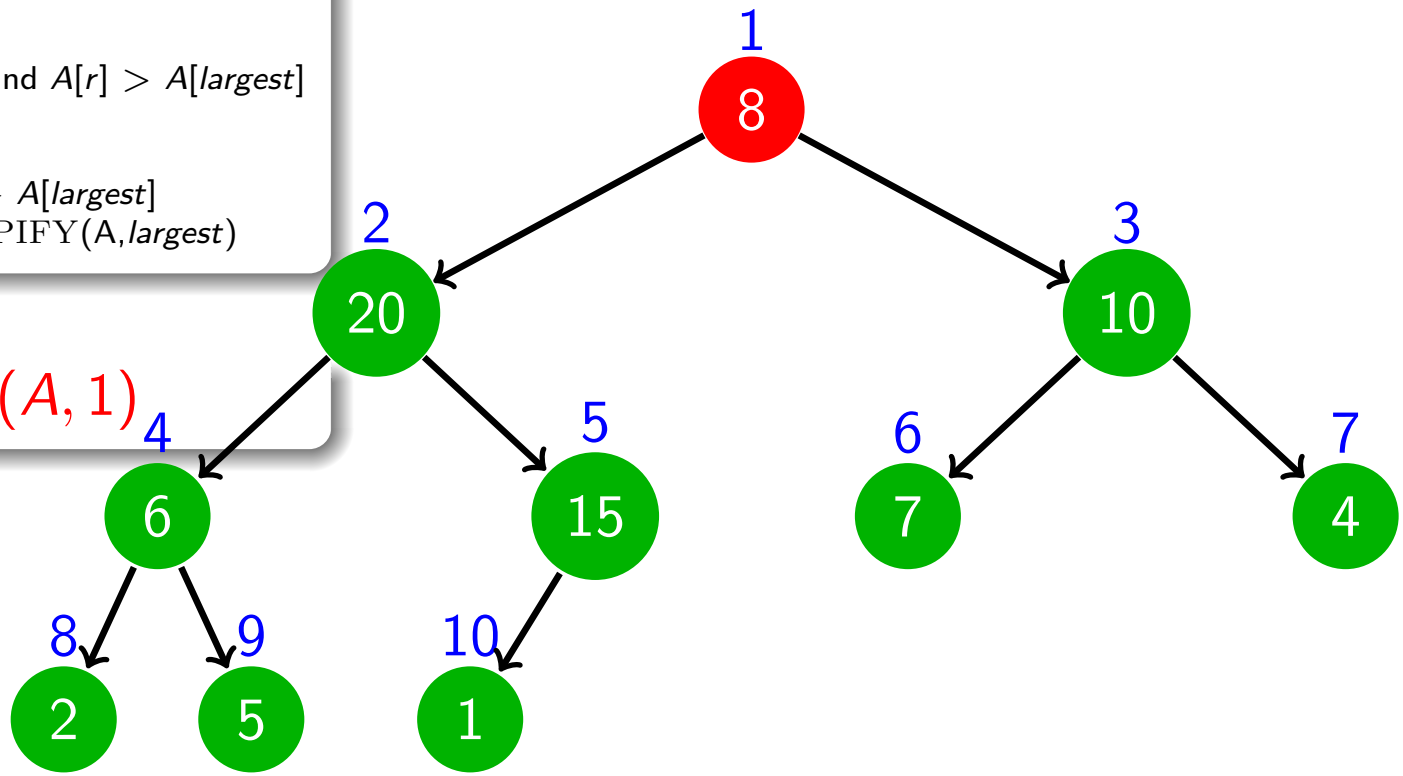



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 15 | 10 | 6 | 8 | 7 | 4 | 2 | 5 | 1 | 21 | 22 | 24 |

# MAX-HEAPIFY

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

MAX-HEAPIFY$(A, 1)$

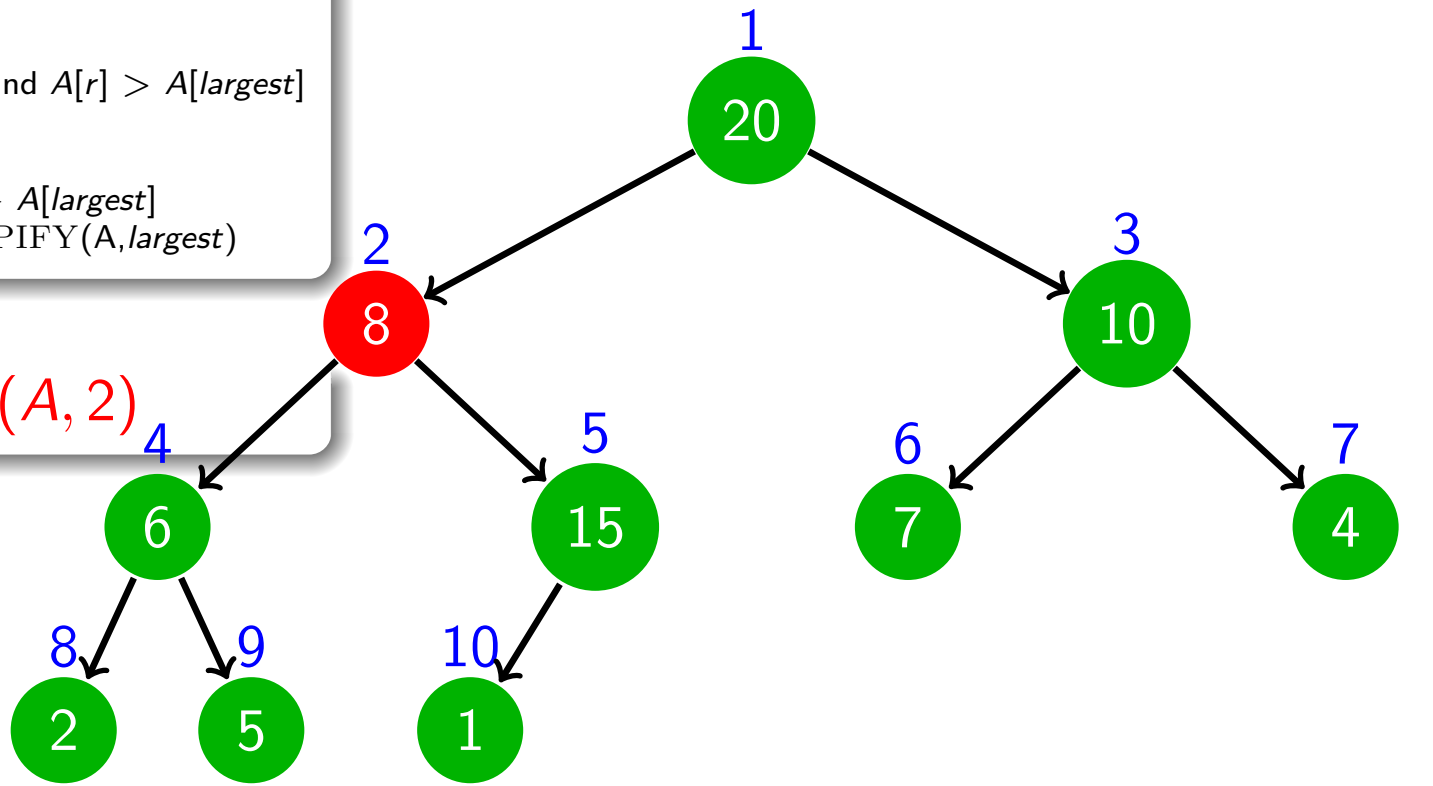



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 20 | 10 | 6 | 15 | 7 | 4 | 2 | 5 | 1 | 21 | 22 | 24 |

# MAX-HEAPIFY

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

MAX-HEAPIFY$(A, 2)$



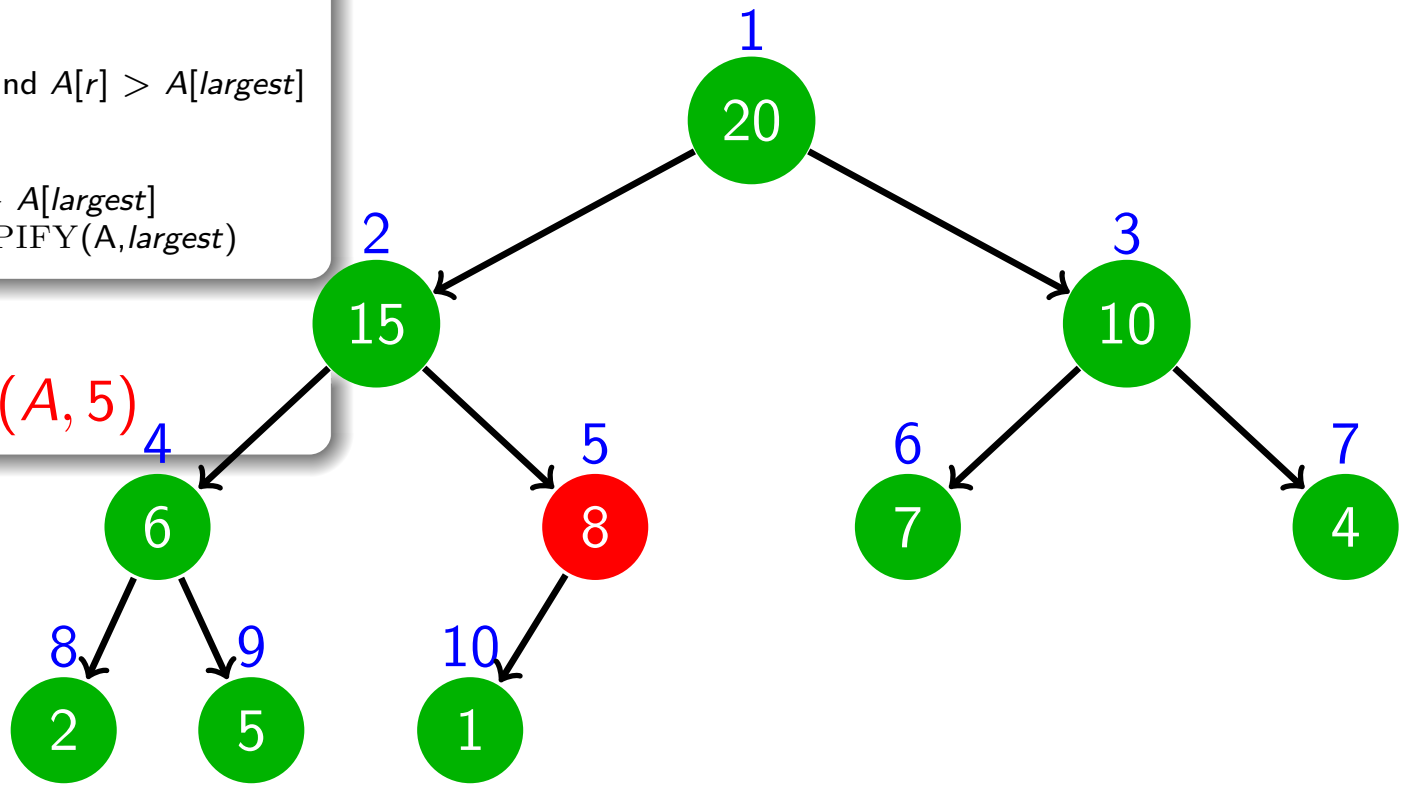

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 8 | 10 | 6 | 15 | 7 | 4 | 2 | 5 | 1 | 21 | 22 | 24 |

# MAX-HEAPIFY

MAX-HEAPIFY($A, i$)

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

MAX-HEAPIFY($A, 5$)



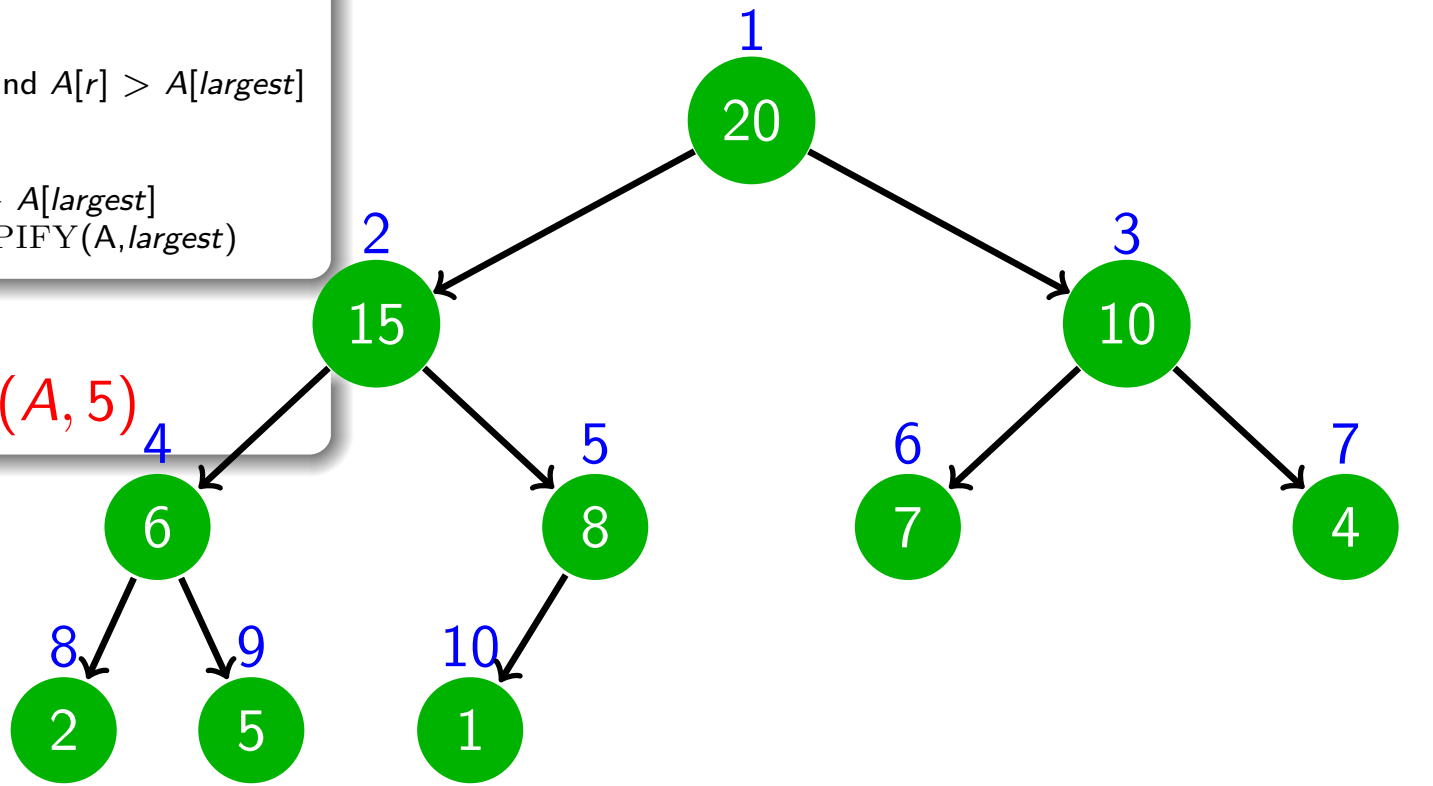

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 15 | 10 | 6 | 8 | 7 | 4 | 2 | 5 | 1 | 21 | 22 | 24 |

# MAX-HEAPIFY

MAX-HEAPIFY$(A, i)$

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

MAX-HEAPIFY$(A, 5)$




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 15 | 10 | 6 | 8 | 7 | 4 | 2 | 5 | 1 | 21 | 22 | 24 |

# MAX-HEAPIFY: Runtime

- Let $n$ be the number of nodes at sub-tree of the heap rooted at $i$
- Each of lines 1-9 takes constant time
- The number of calls at line 10 is bounded by the height $\lfloor \log_2(n) \rfloor$ of the sub-tree of the heap rooted at $i$

  $\Rightarrow$ Hence, $T(n) = O(log_2(n))$ since MAX-HEAPIFY$(A, i)$ should process $O(log_2(n))$ levels, with constant work at each level

MAX-HEAPIFY$(A, i)$

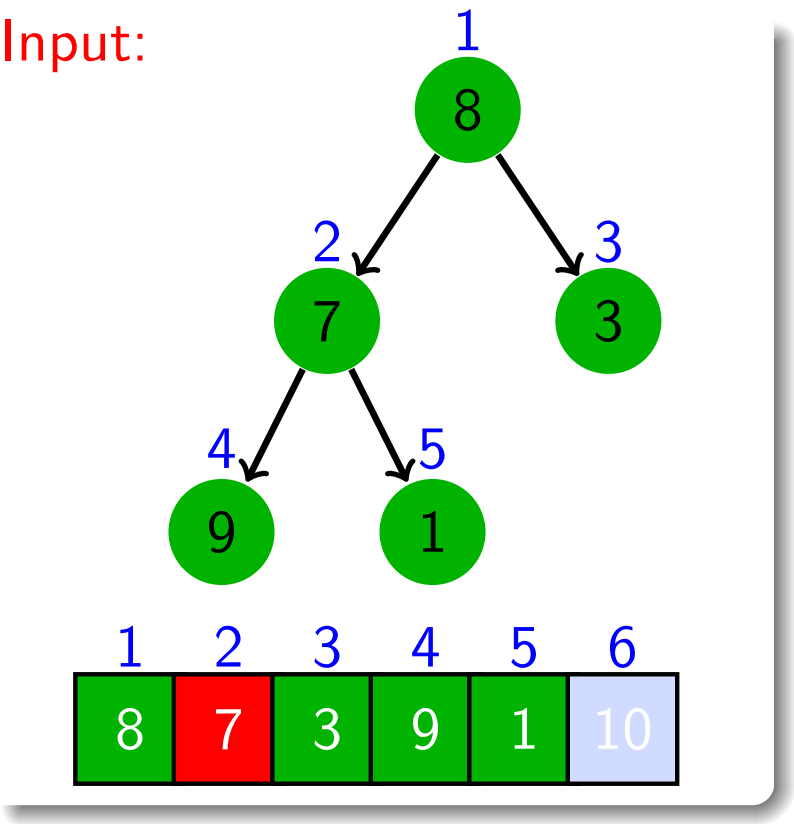
```
1   l ← LEFT(i)
2   r ← RIGHT(i)
3   if l ≤ A. heap-size and A[l] > A[i]
4       then largest ← l
5       else largest ← i
6   if r ≤ A. heap-size and A[r] > A[largest]
7       then largest ← r
8   if largest ≠ i
9       then swap A[i] ↔ A[largest]
10           MAX-HEAPIFY(A,largest)
```

$\Rightarrow$ $T(n)$ is linear in the height of the sub-tree of the heap rooted at $i$

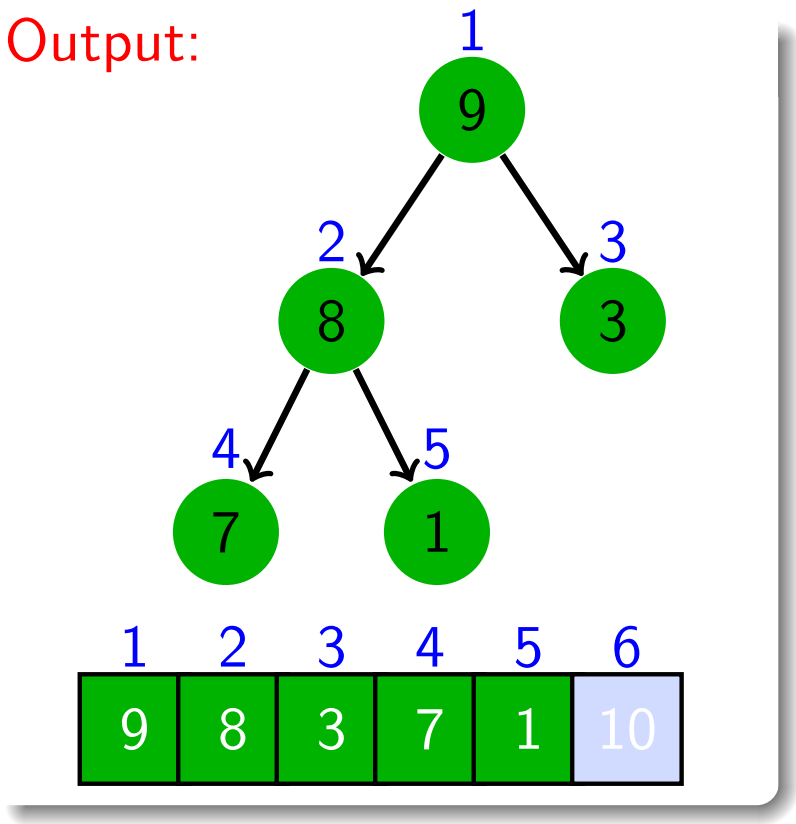
# Build-Max-Heap

- Input: An array $A$
- Output: A max-heap from $A$

Input:



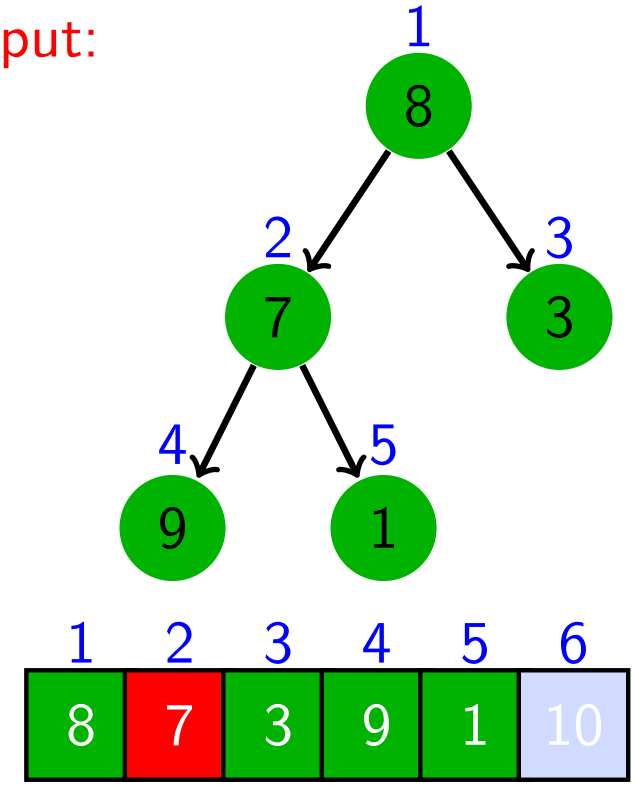

Output:



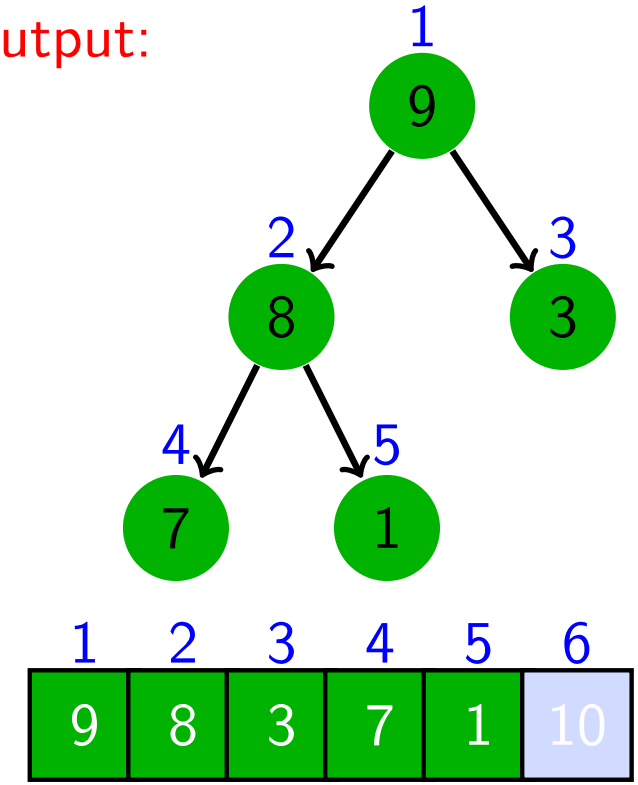
# Build-Max-Heap

- Input: An array $A$
- Output: A max-heap from $A$
- Idea: Use Max-Heapify in a bottom-up manner

Input:




Output:

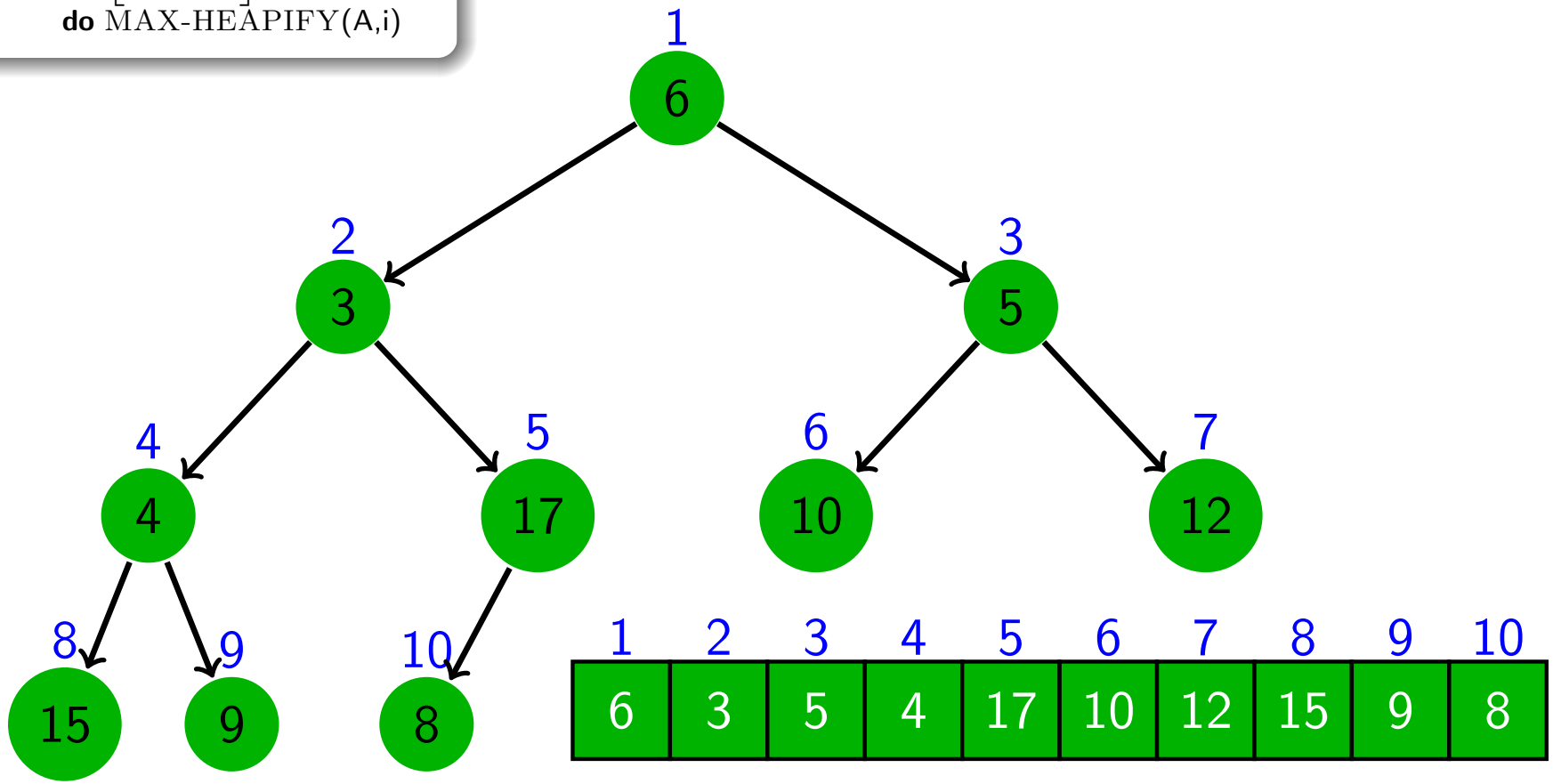


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 5$

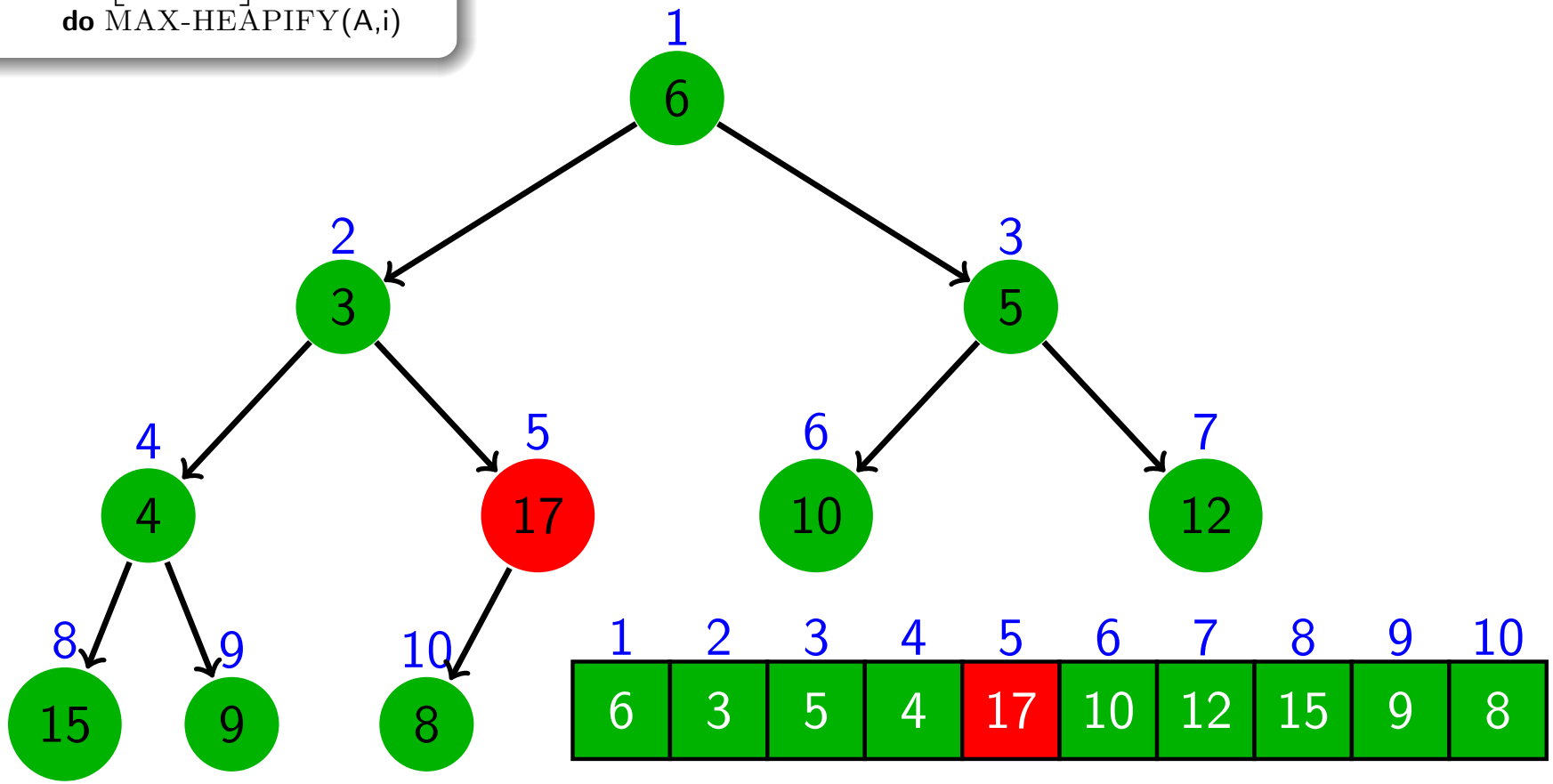


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 5$

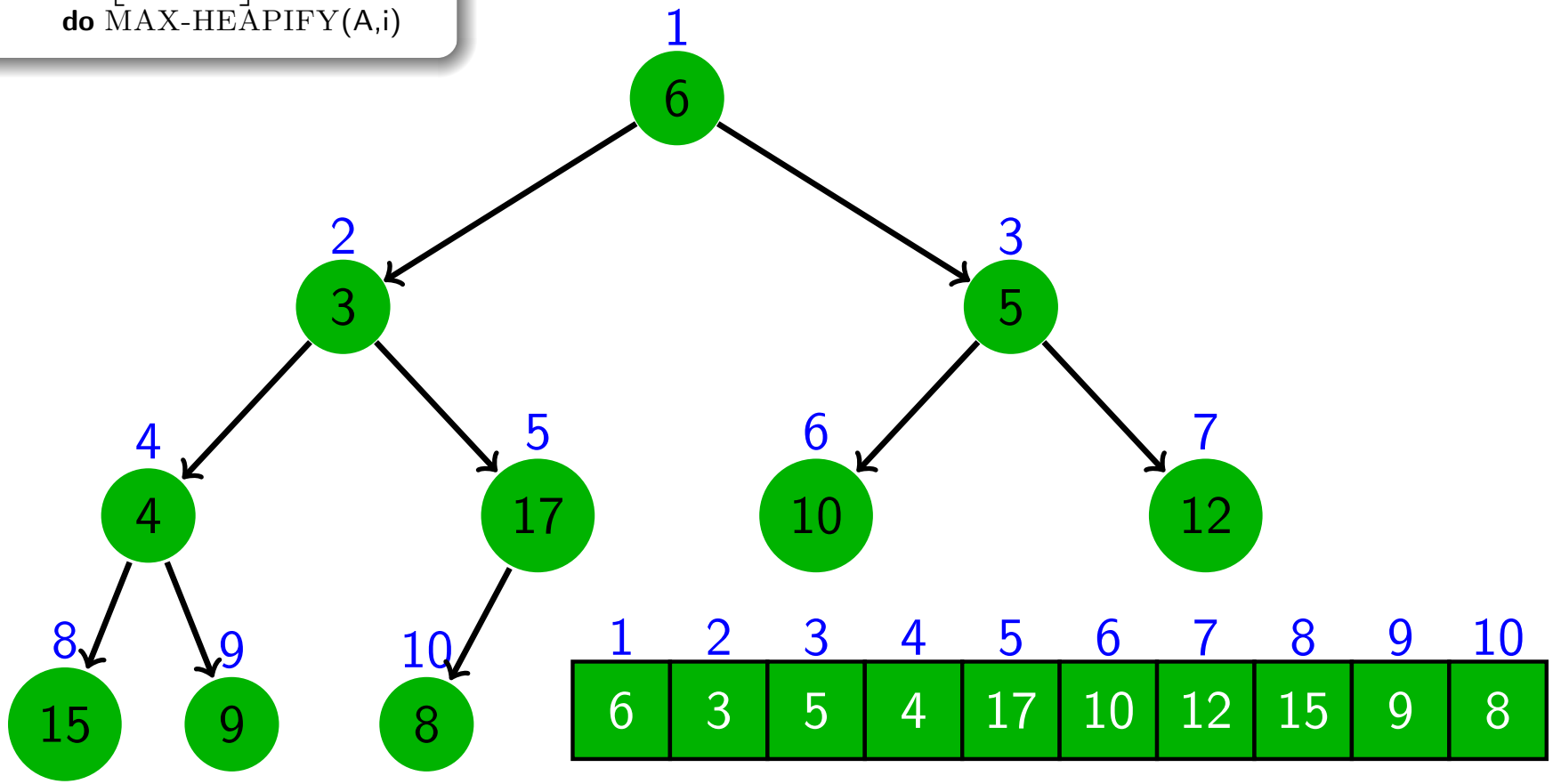


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 4$

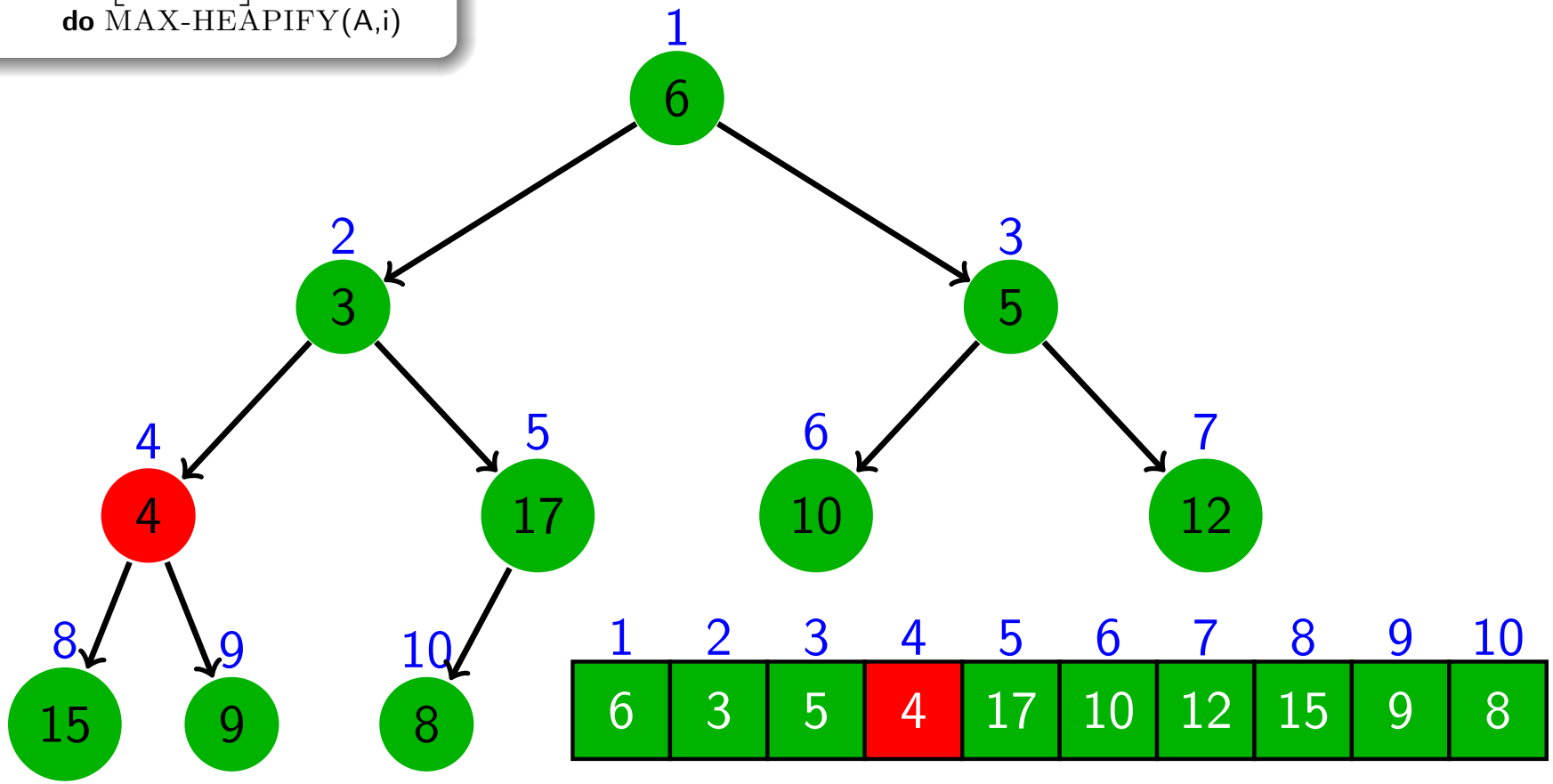


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 4$



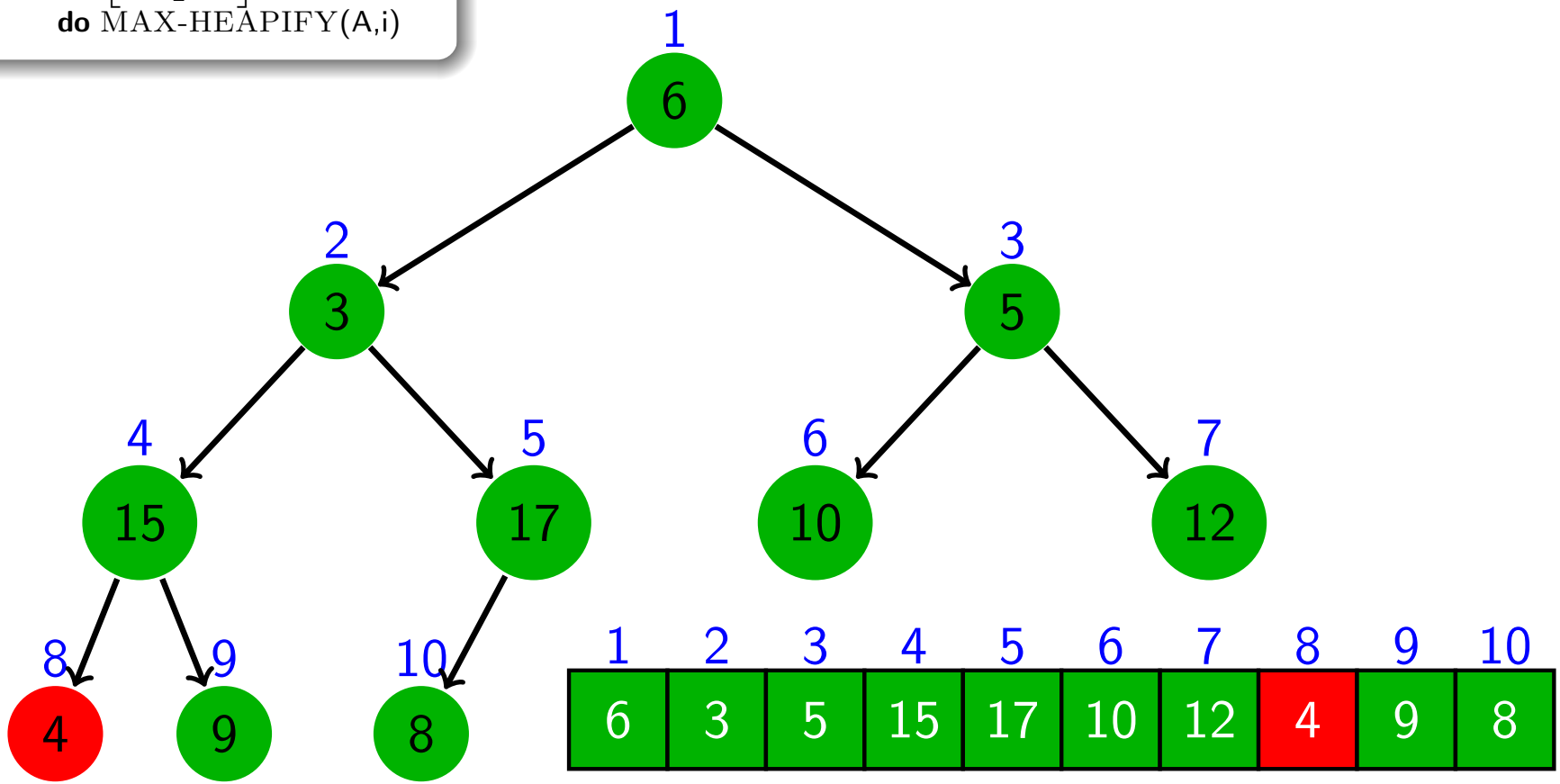
# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)

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

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\,length]$

$i = 4$



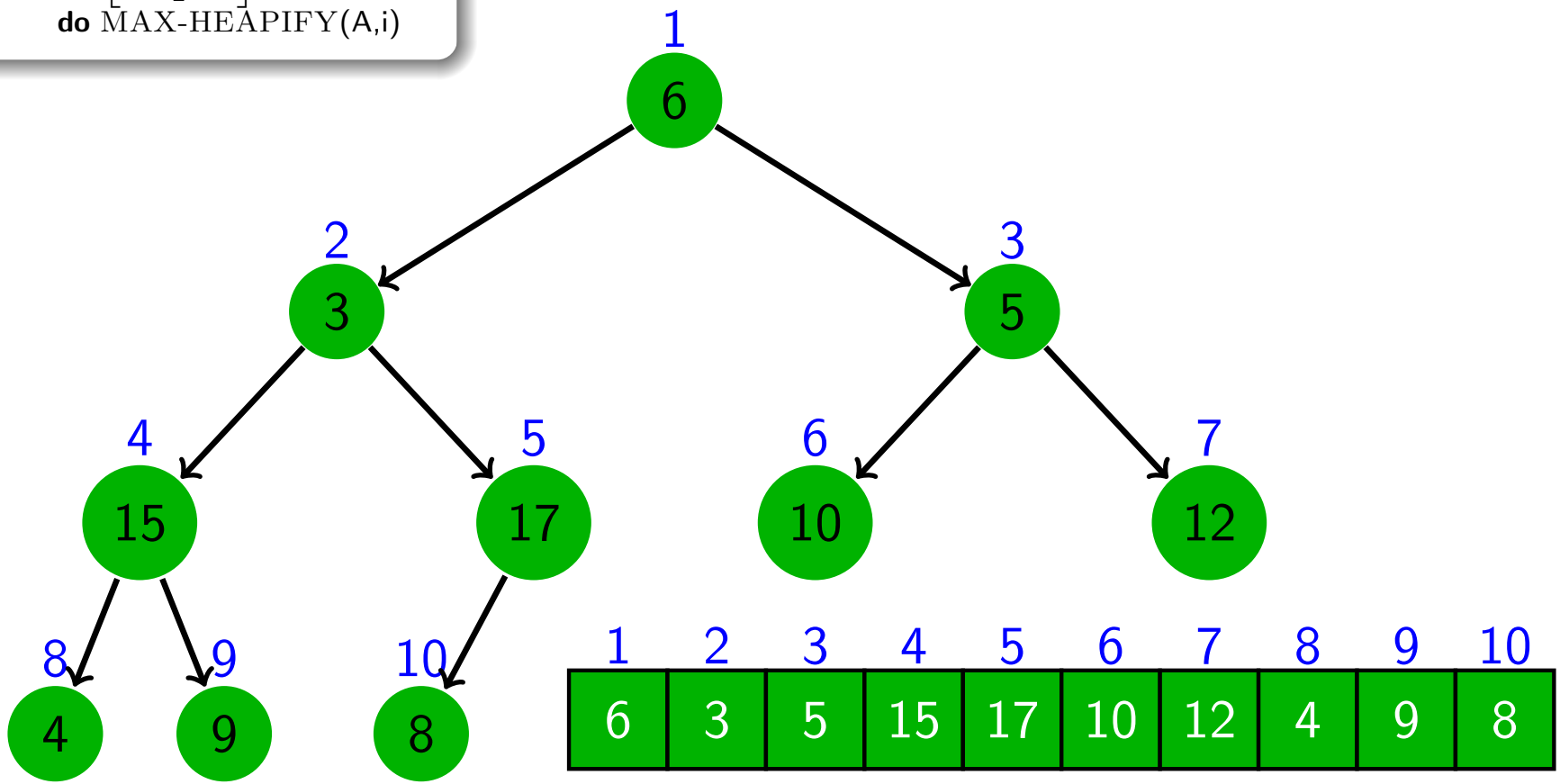
# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)

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

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\,length]$

$i = 3$

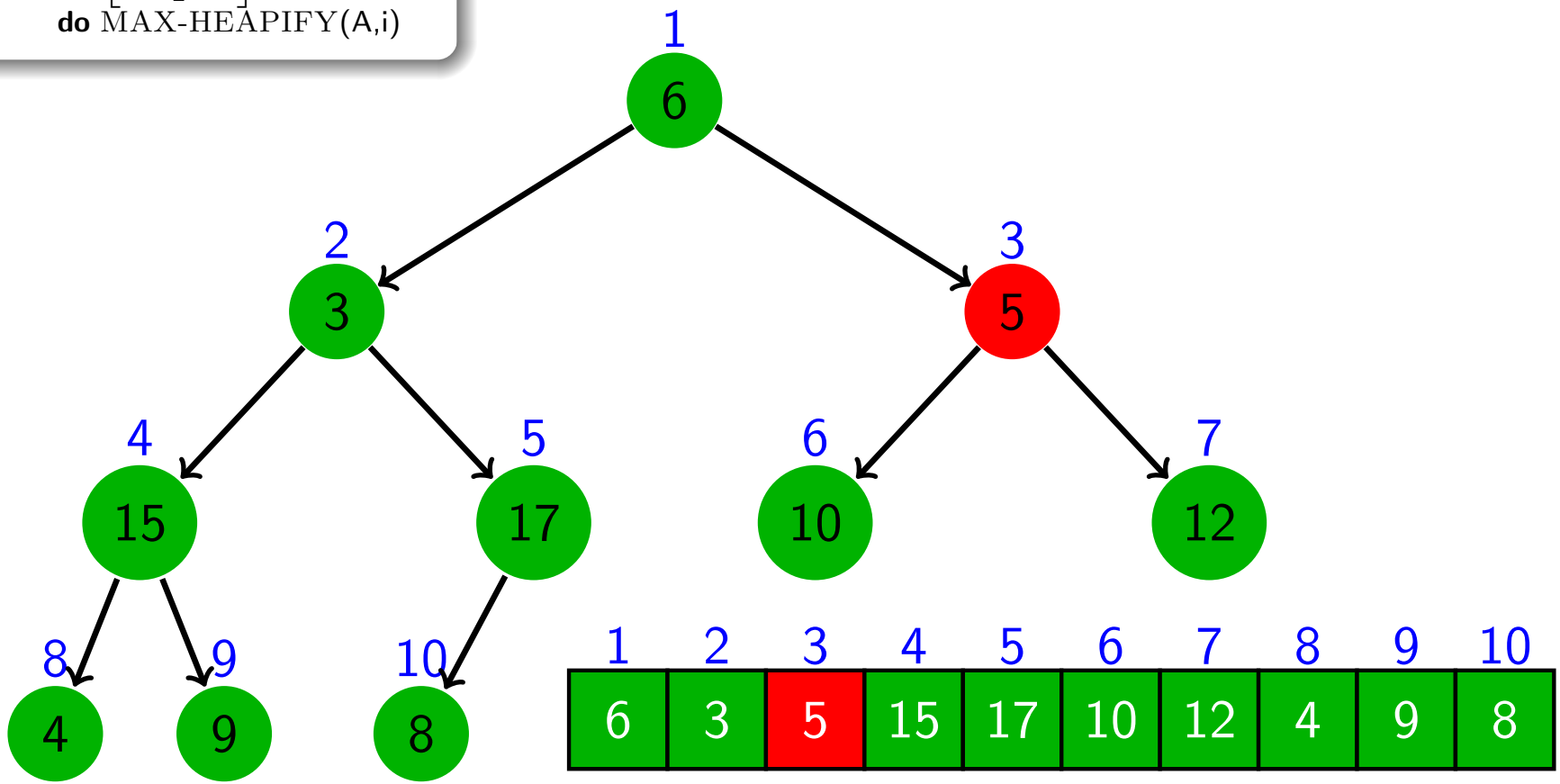


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 3$



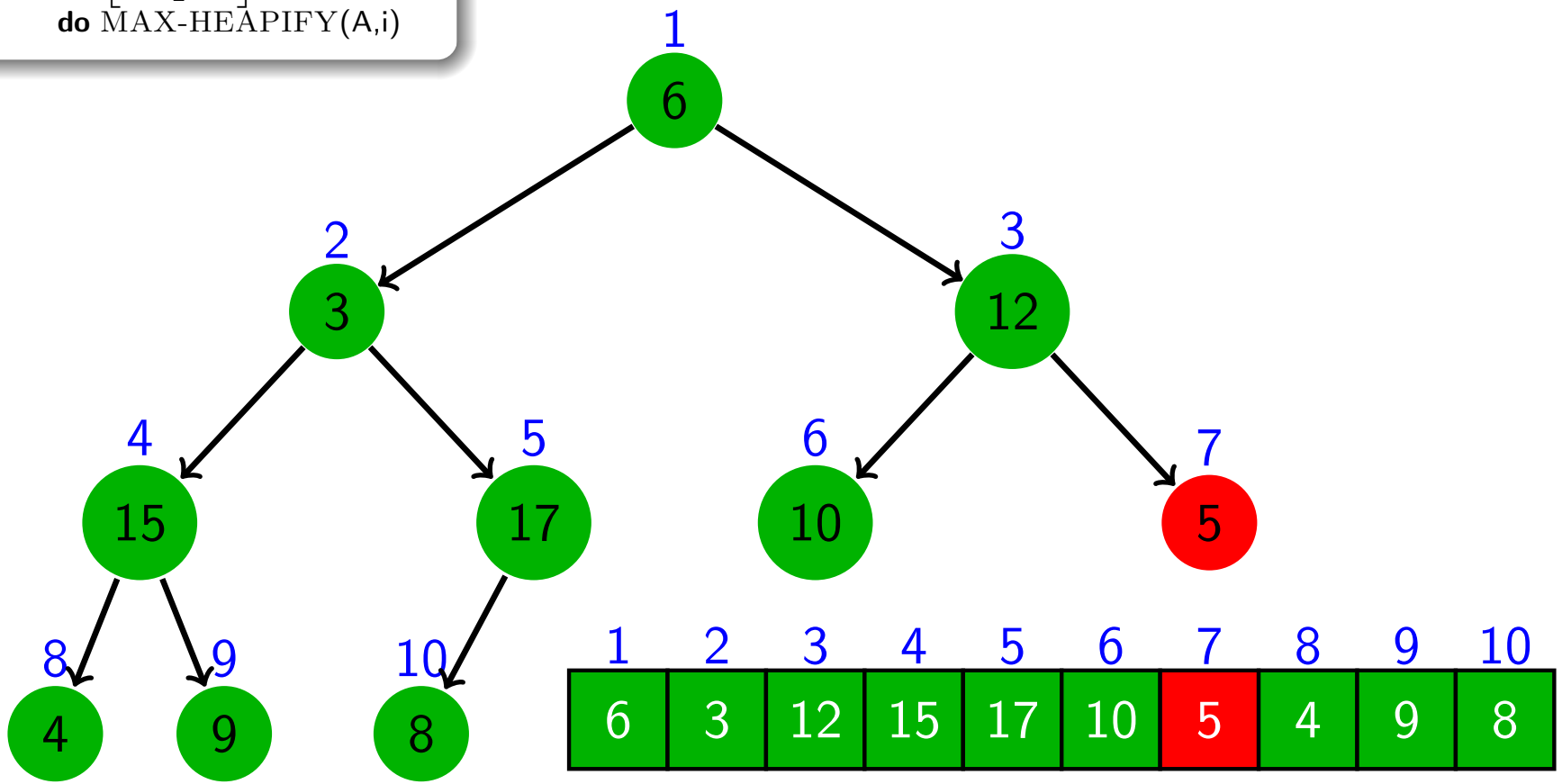
# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)

1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 3$

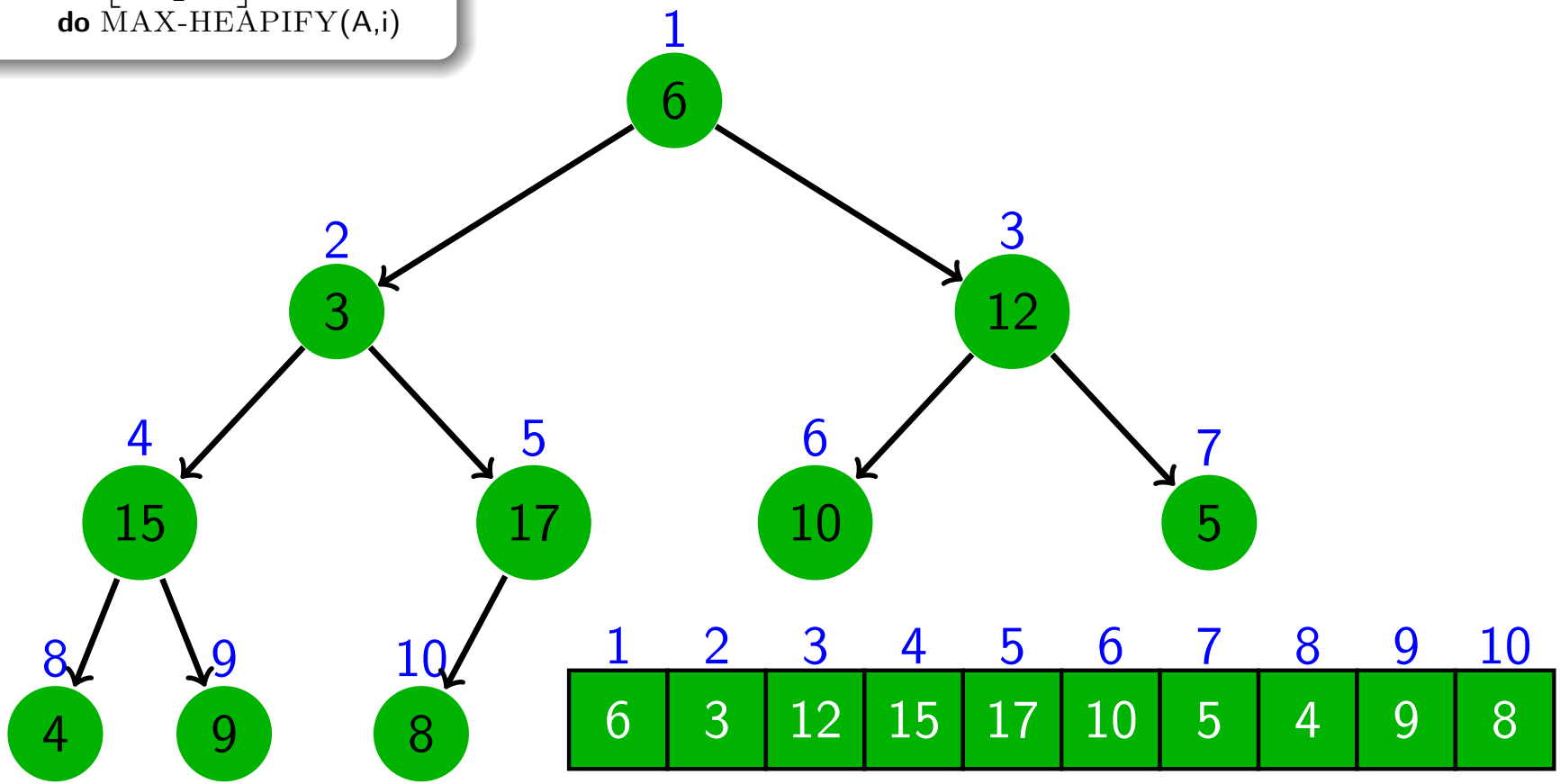


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 2$

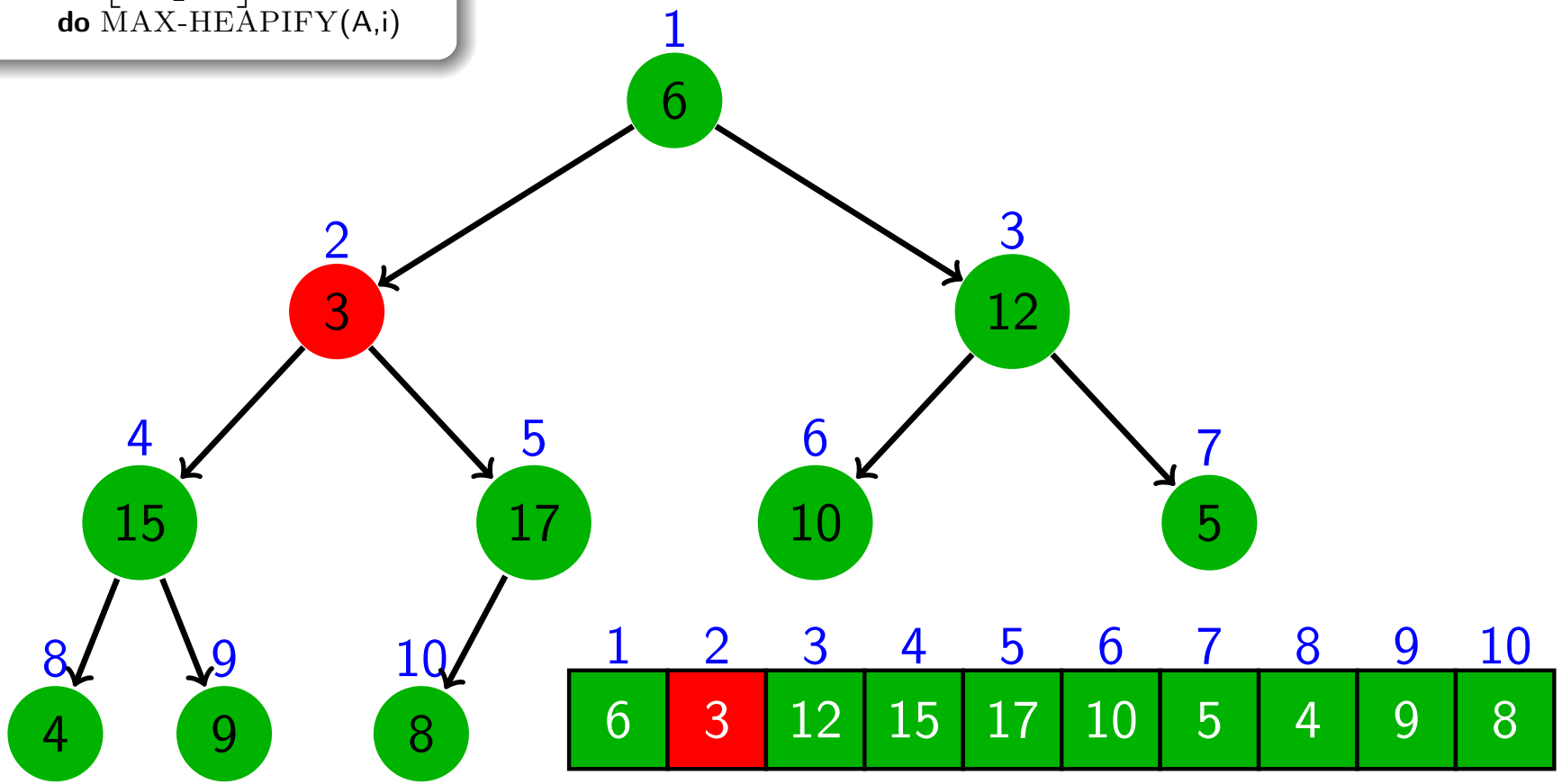


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 2$



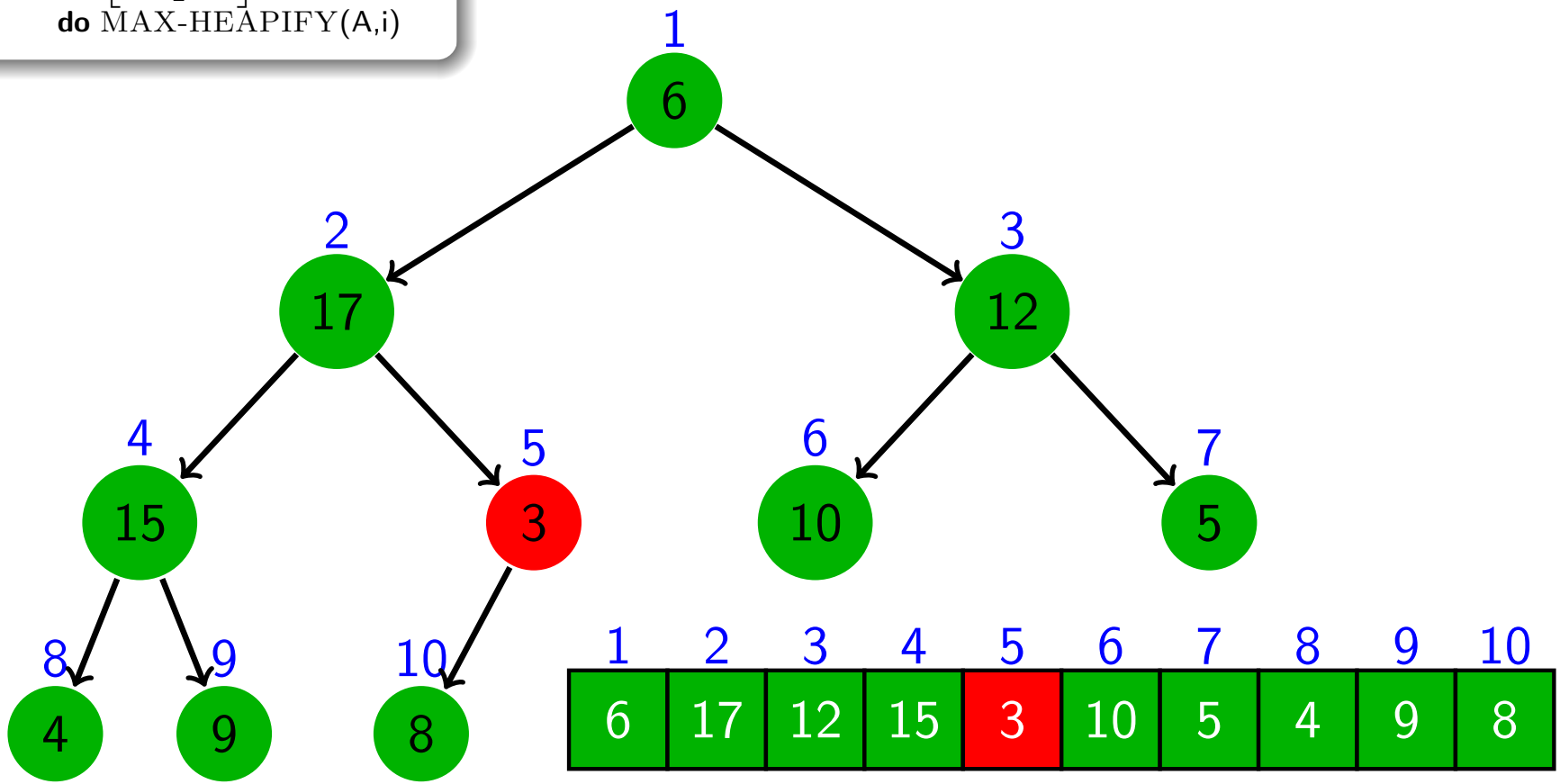
# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)

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

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\,length]$

$i = 2$



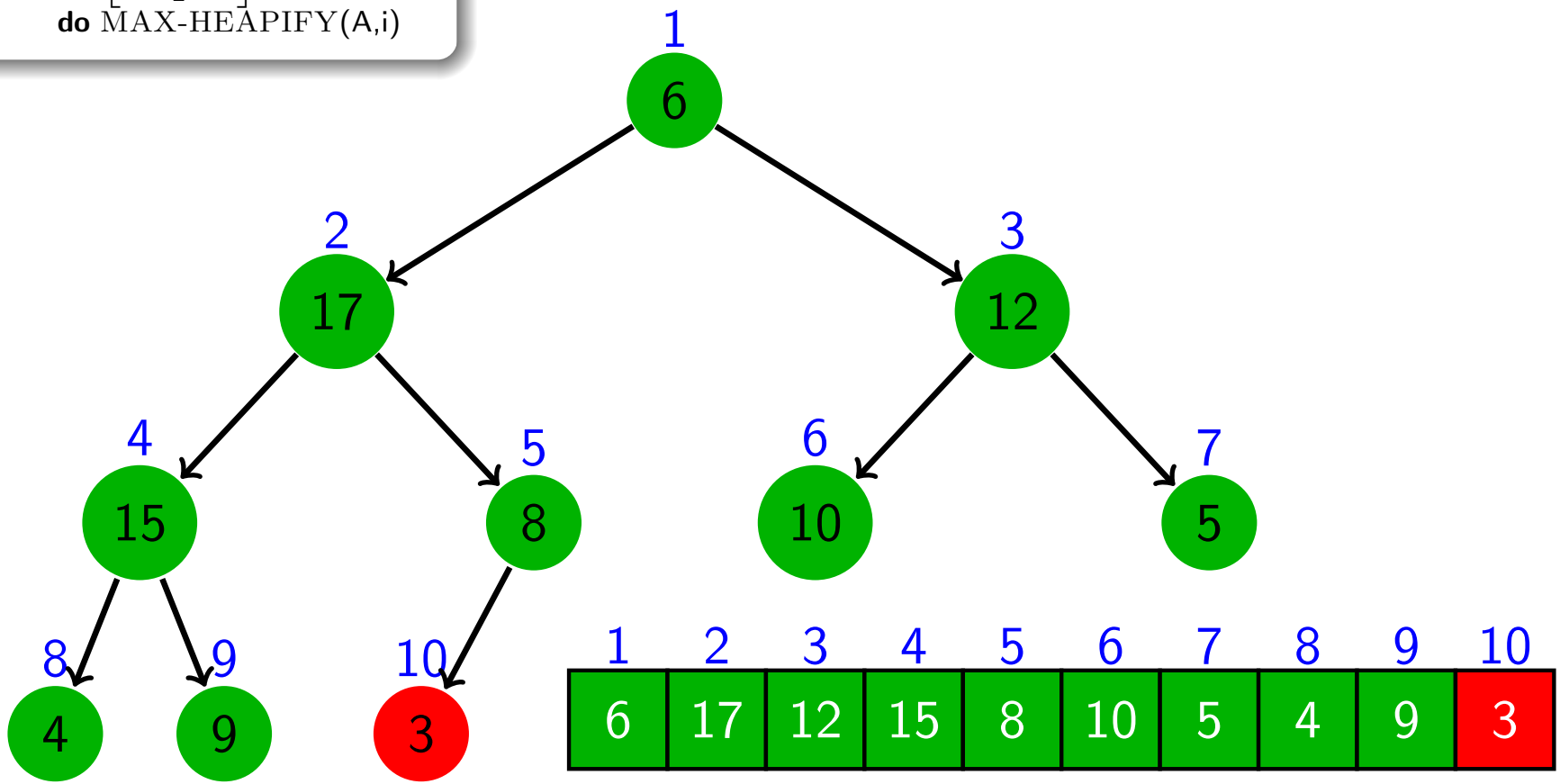
# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)

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

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\,length]$

$i = 2$

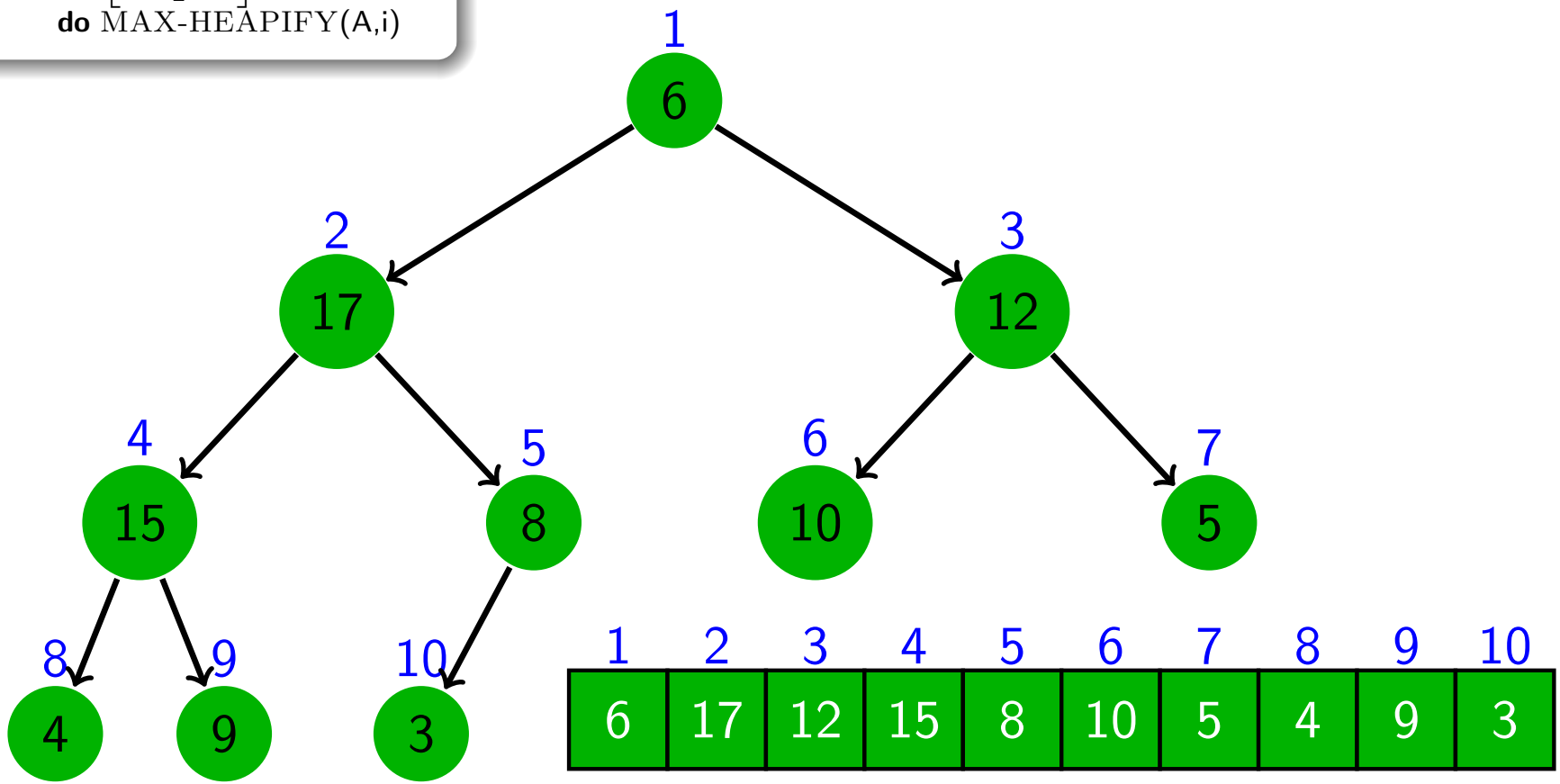


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 1$



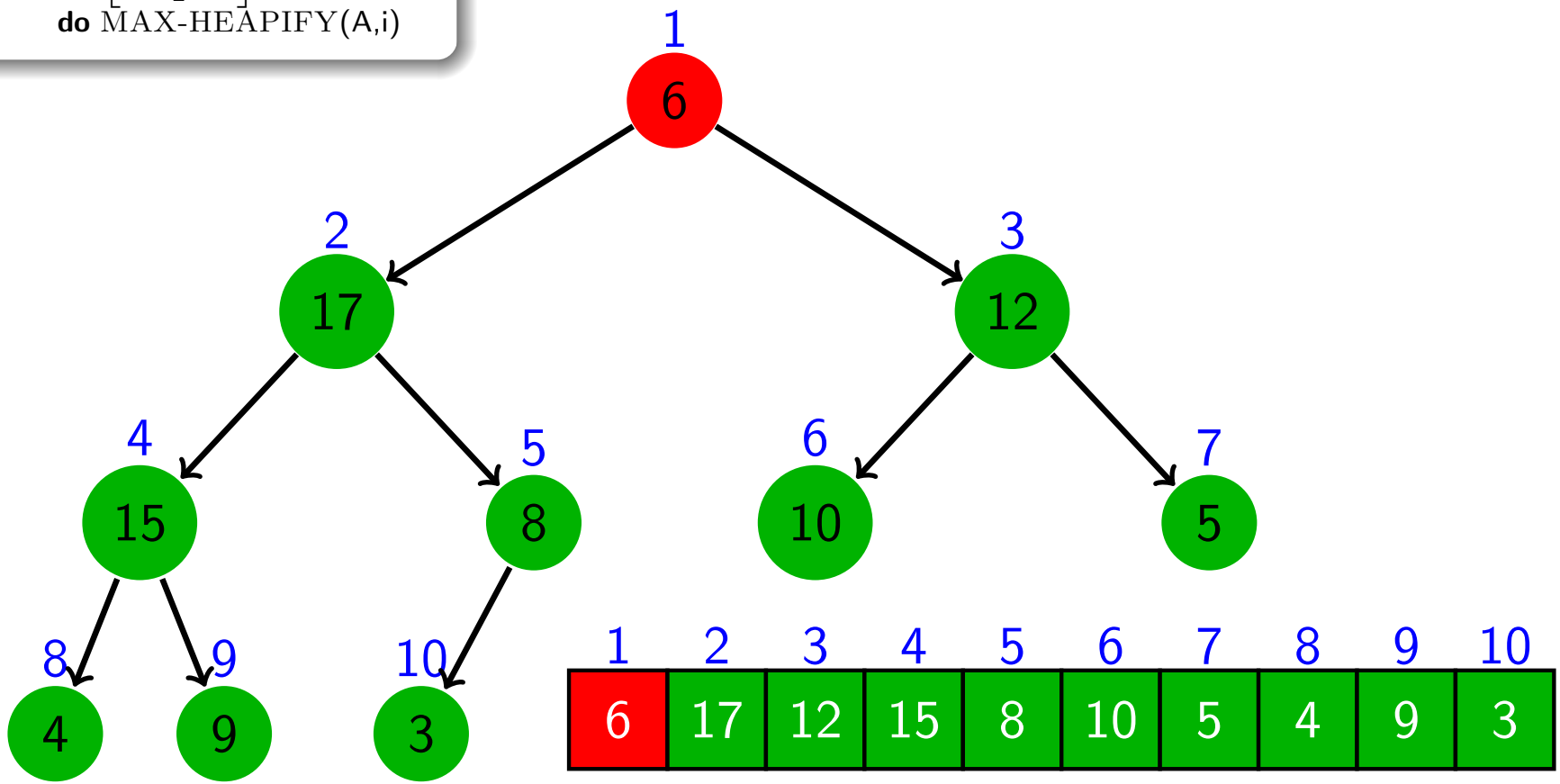
# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)

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

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\,length]$

$i = 1$

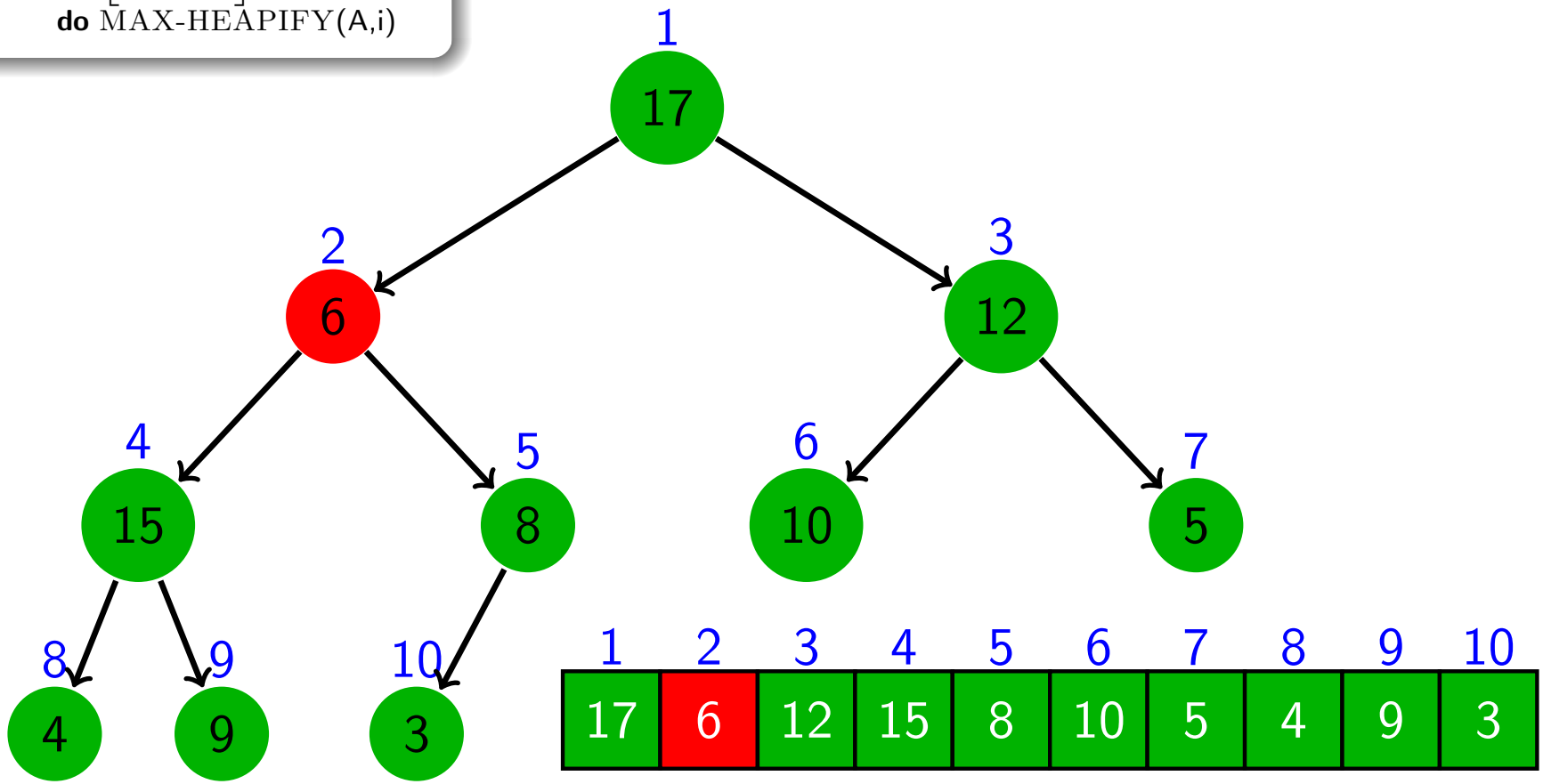


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 1$

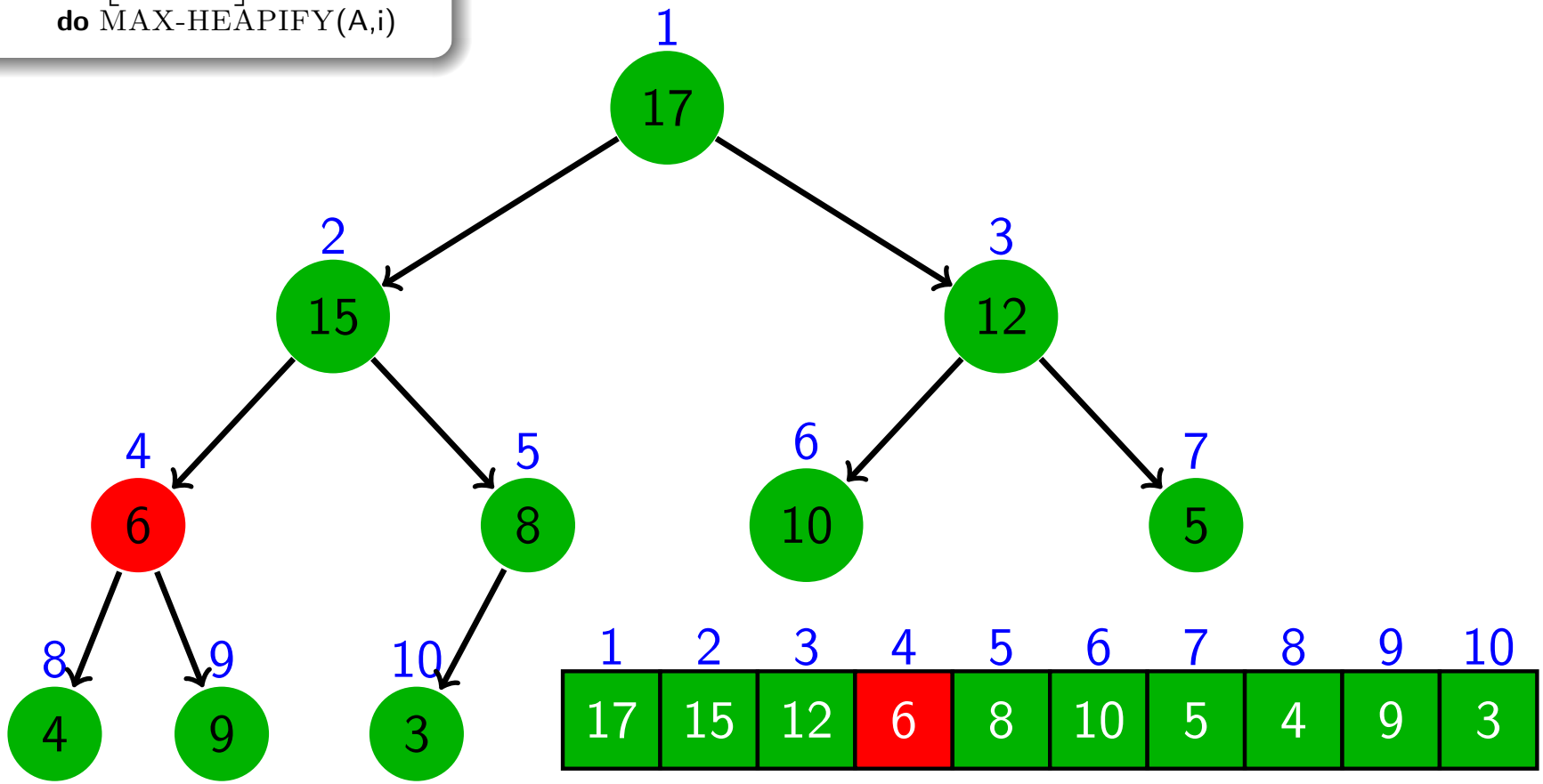


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

$i = 1$

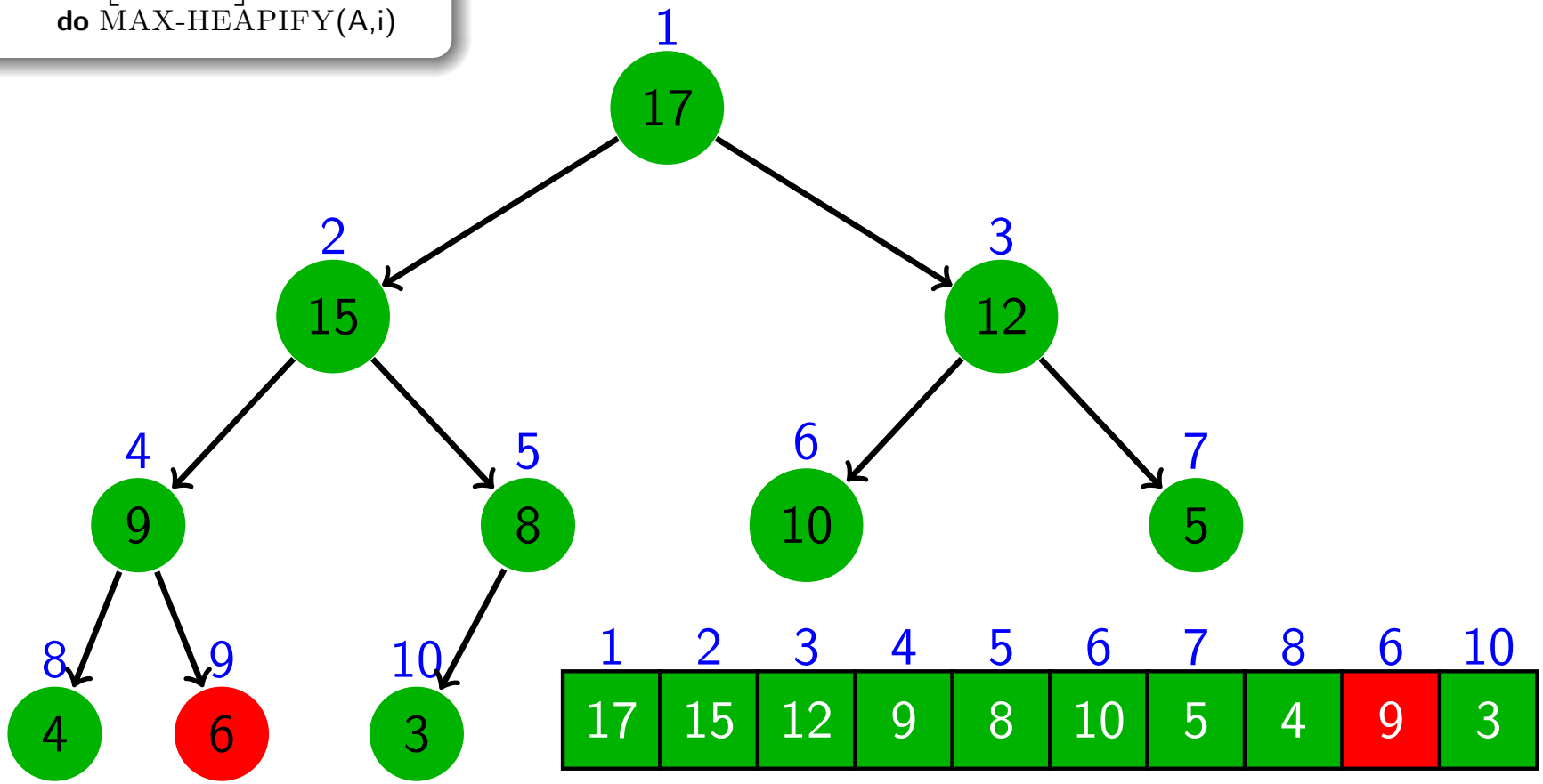


# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$

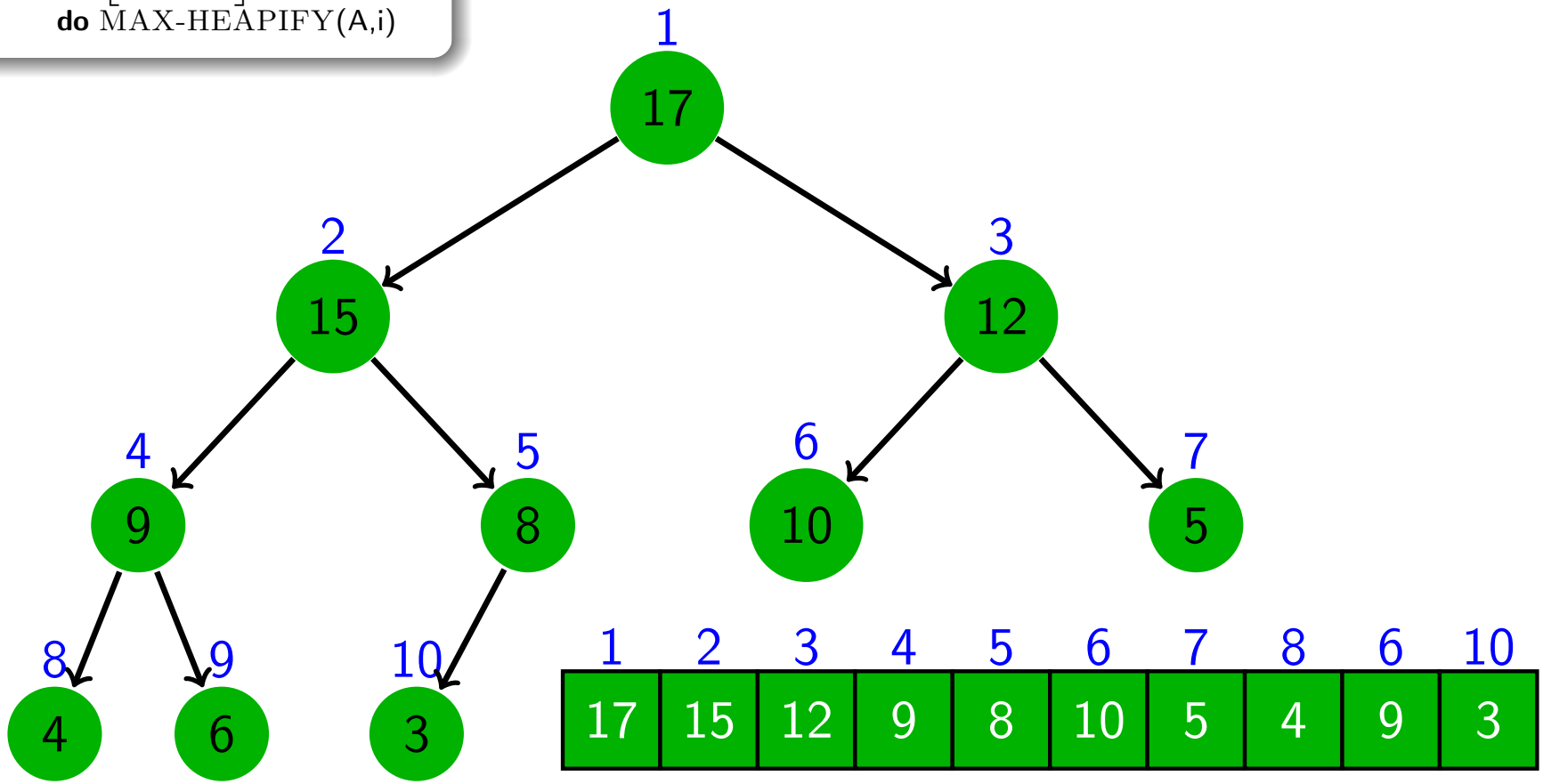
$i = 1$



# BUILD-MAX-HEAP

BUILD-MAX-HEAP($A$)
1 $A.\ heap\text{-}size \leftarrow A.\ length$
2 **for** $i \leftarrow \left\lfloor \frac{A.\ length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Remark: The leaves are the elements of $A[(\lfloor \frac{n}{2} \rfloor + 1)..A.\ length]$



# BUILD-MAX-HEAP: Runtime

- Let $n = A.length$
- $h = \lfloor log_2(n) \rfloor$ be the height of the heap
- Simple bound:
  - $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(log_2(n))$ time
    $\Rightarrow T(n) = O(n \cdot log_2(n))$

BUILD-MAX-HEAP($A$)
1 $A.heap\text{-}size \leftarrow A.length$
2 **for** $i \leftarrow \left\lfloor \frac{A.length}{2} \right\rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A,i)

- Tight bound:
  - We have at most $2^i$ nodes at depth $i$ ( height $h - i$)
  - We call MAX-HEAPIFY for each node of depth $i \Rightarrow O(h - i)$
  - The runtime of BUILD-MAX-HEAP is:

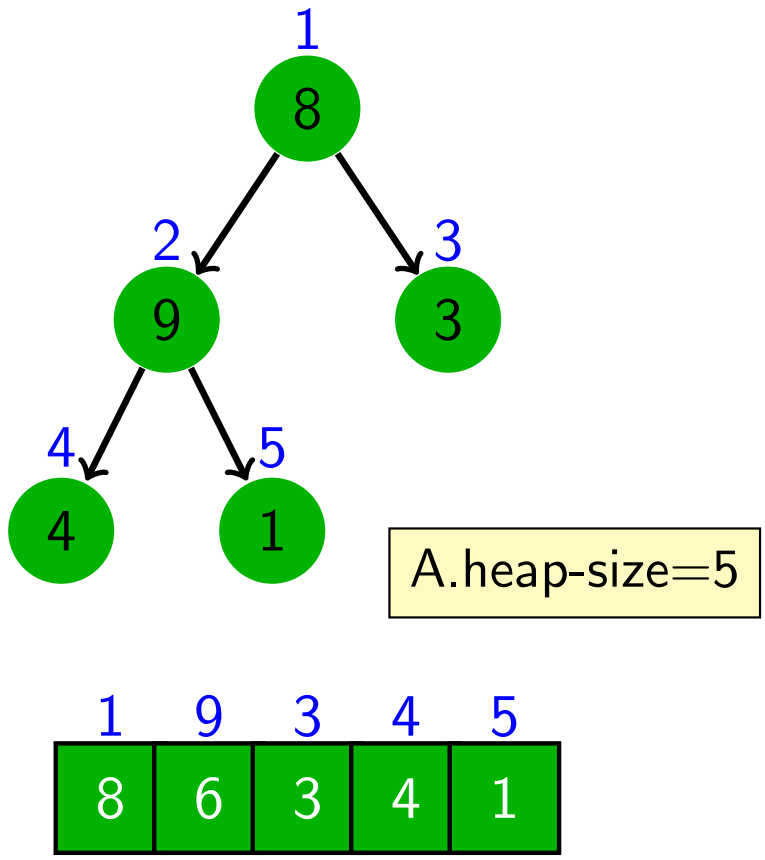
$$\begin{aligned} T(n) = \sum_{i=0}^{h-1} 2^i O(h-i) &= O\left(\sum_{i=0}^{h-1} 2^i (h-i)\right) \\ &= O\left(\sum_{j=1}^{h} \sum_{i=0}^{h-j} 2^i\right) \\ &= O(2^{h+1} - h - 2) \\ &= O(n) \end{aligned}$$

# HeapSort: Principle

Problem: Sort an array $A$ of $n$ elements in non-decreasing order

## HeapSort

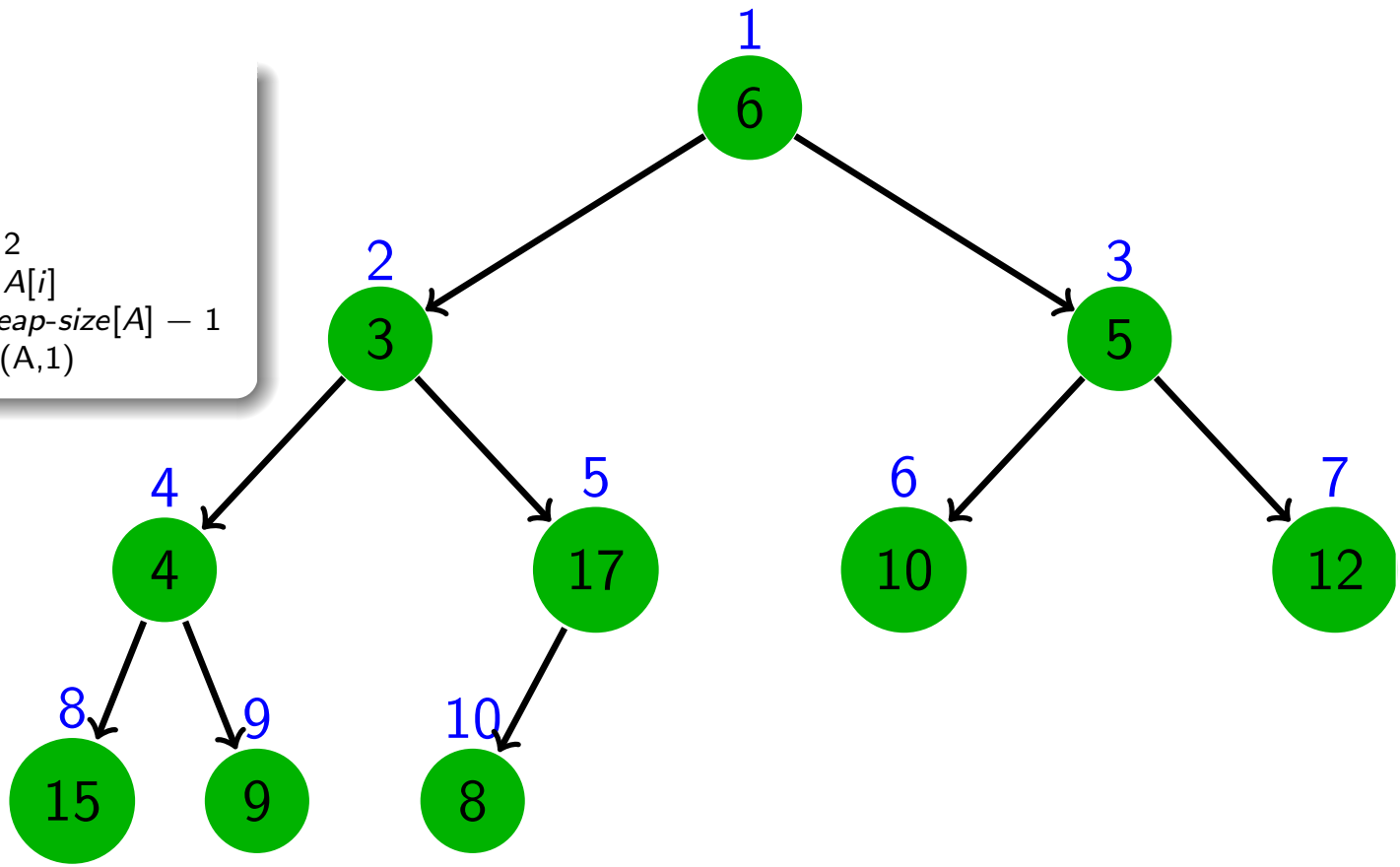
- Construct a max-heap from $A$ (call BUILD-MAX-HEAP(A))
- Repeat until the heap is of size one:
  - Swap the values of the root and the right-most leaf of the heap (i.e., Swap A[1] and $A[A.heap - size]$ )
  - Discard the right-most leaf from the heap by decreasing the heap size
  - Restore the max-heap property (call MAX-HEAPIFY(A,1))



# HEAP-SORT

HEAP-SORT($A$)

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

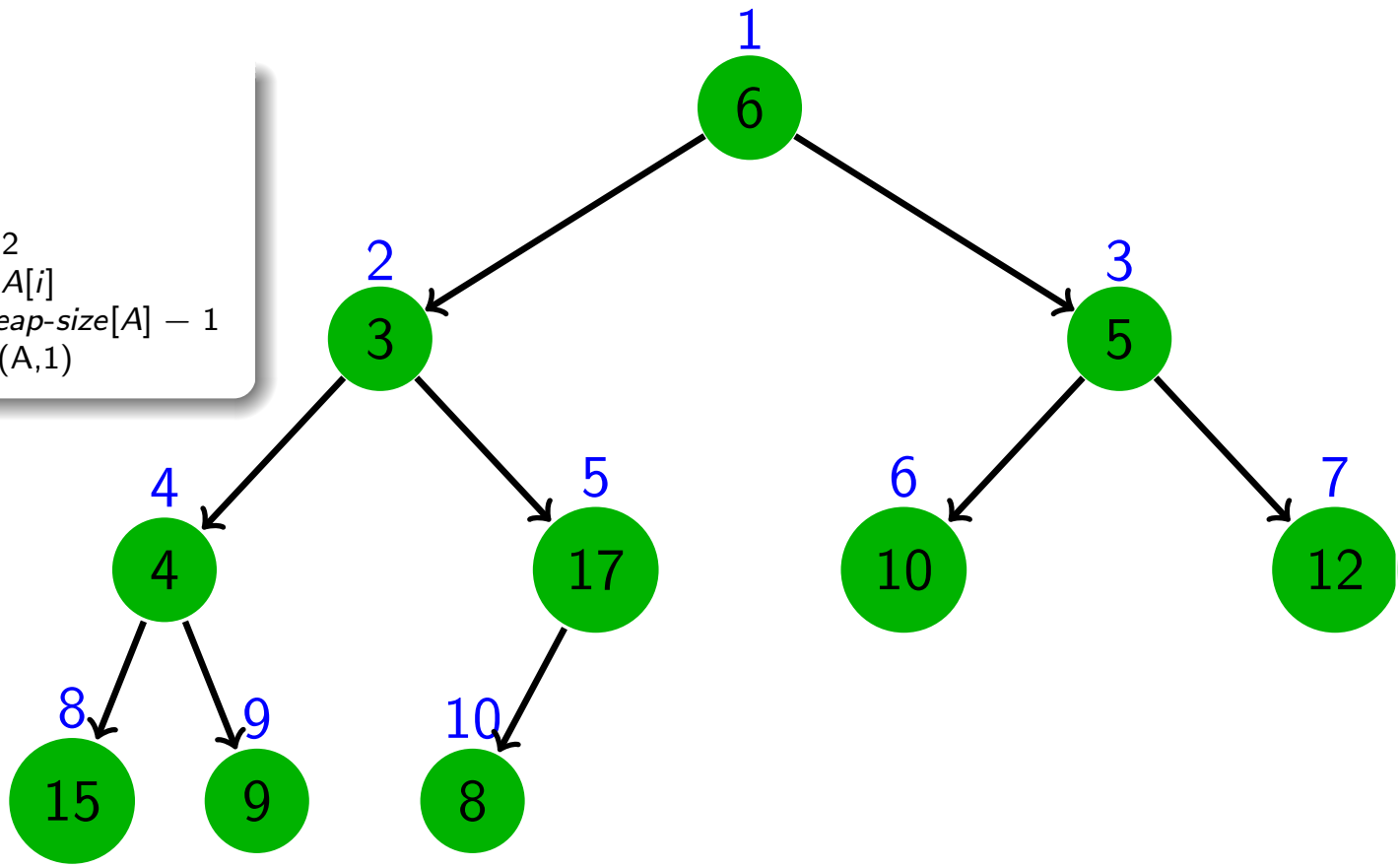



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 6 | 3 | 5 | 4 | 17 | 10 | 12 | 15 | 9 | 8 |

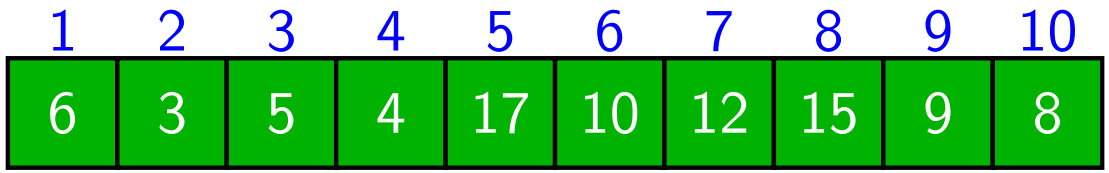
# HEAP-SORT

HEAP-SORT($A$)

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



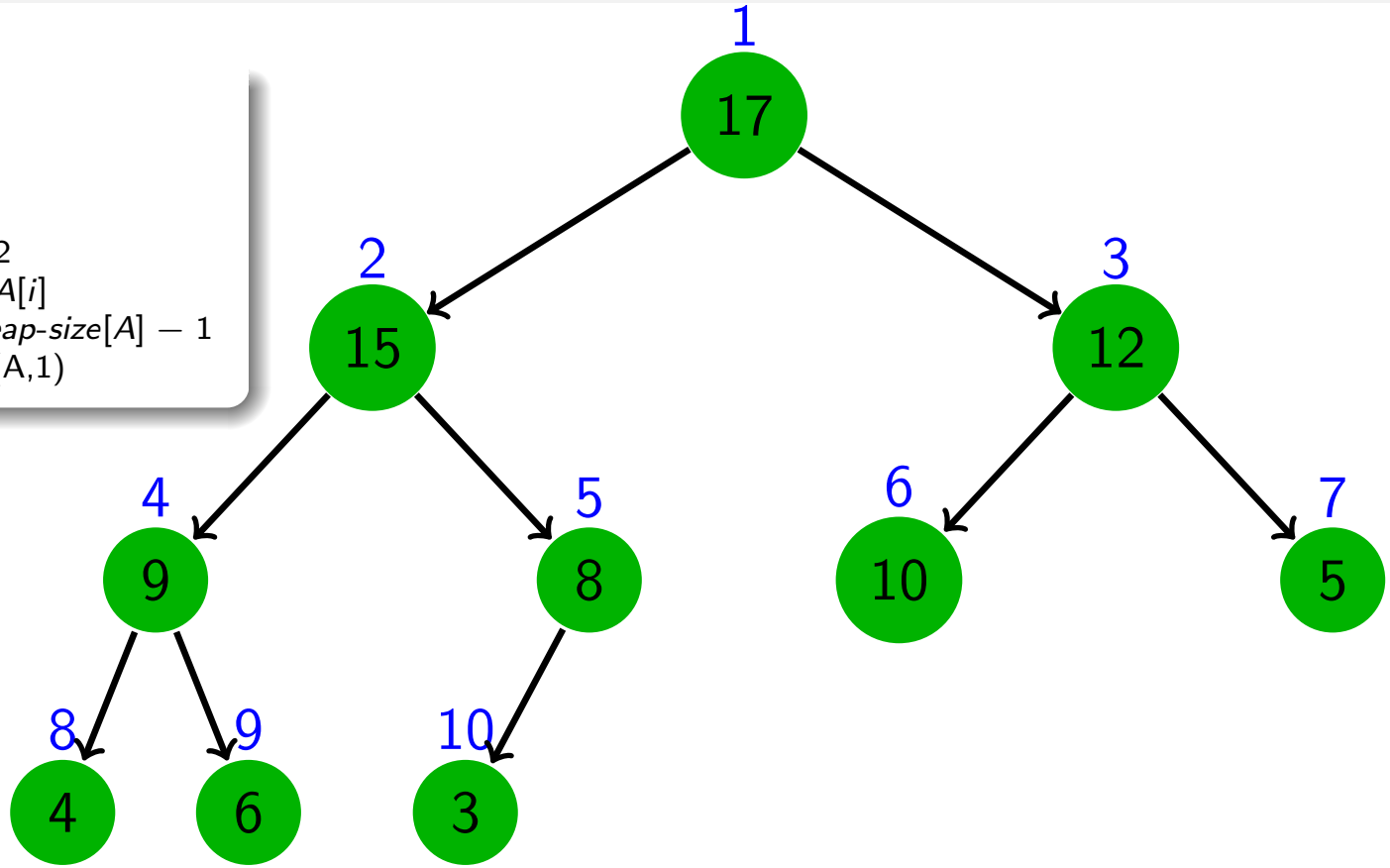

## BUILD-MAX-HEAP(A)



# HEAP-SORT

HEAP-SORT($A$)

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

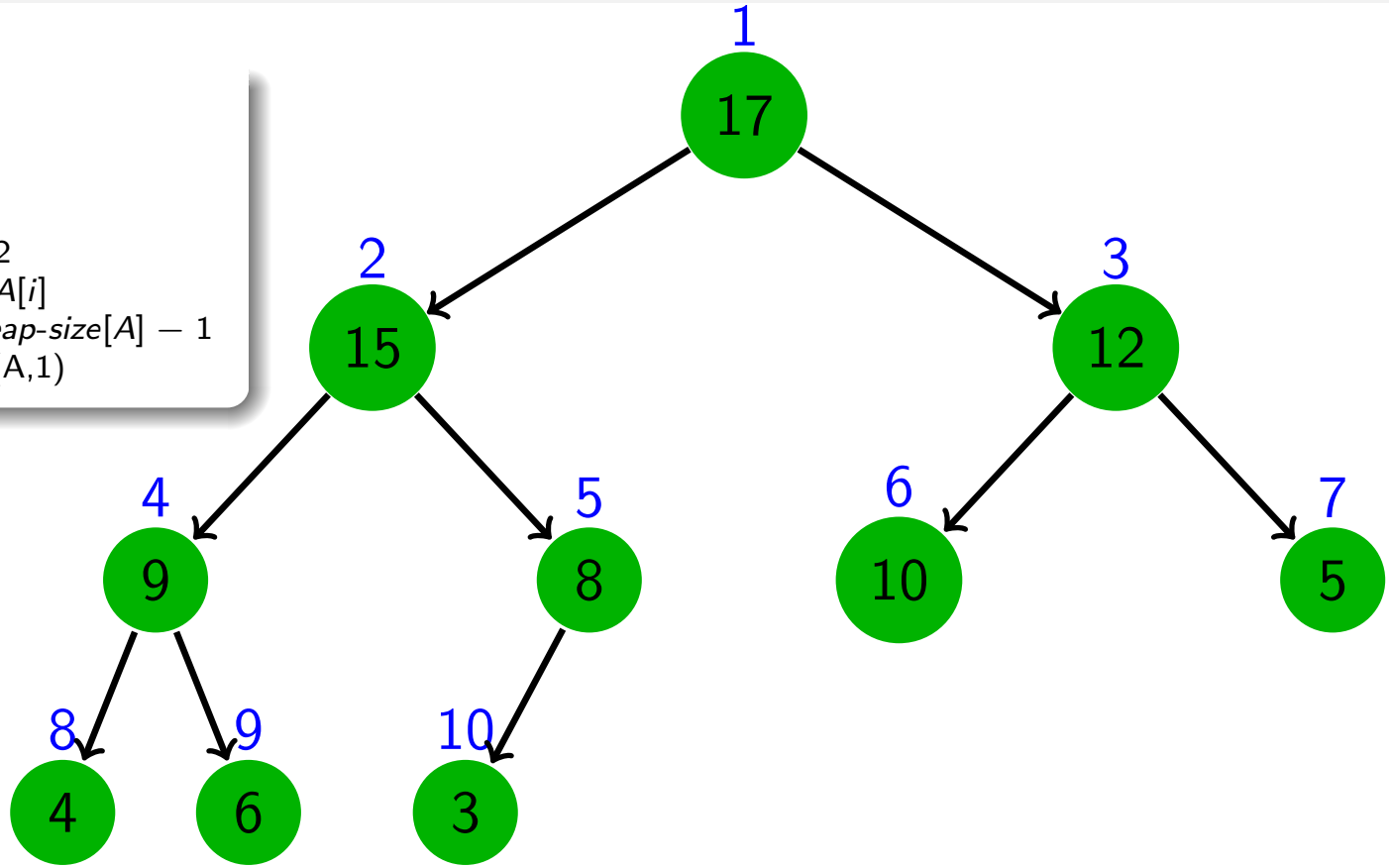



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 17 | 15 | 12 | 9 | 8 | 10 | 5 | 4 | 6 | 3 |

# HEAP-SORT

HEAP-SORT($A$)

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



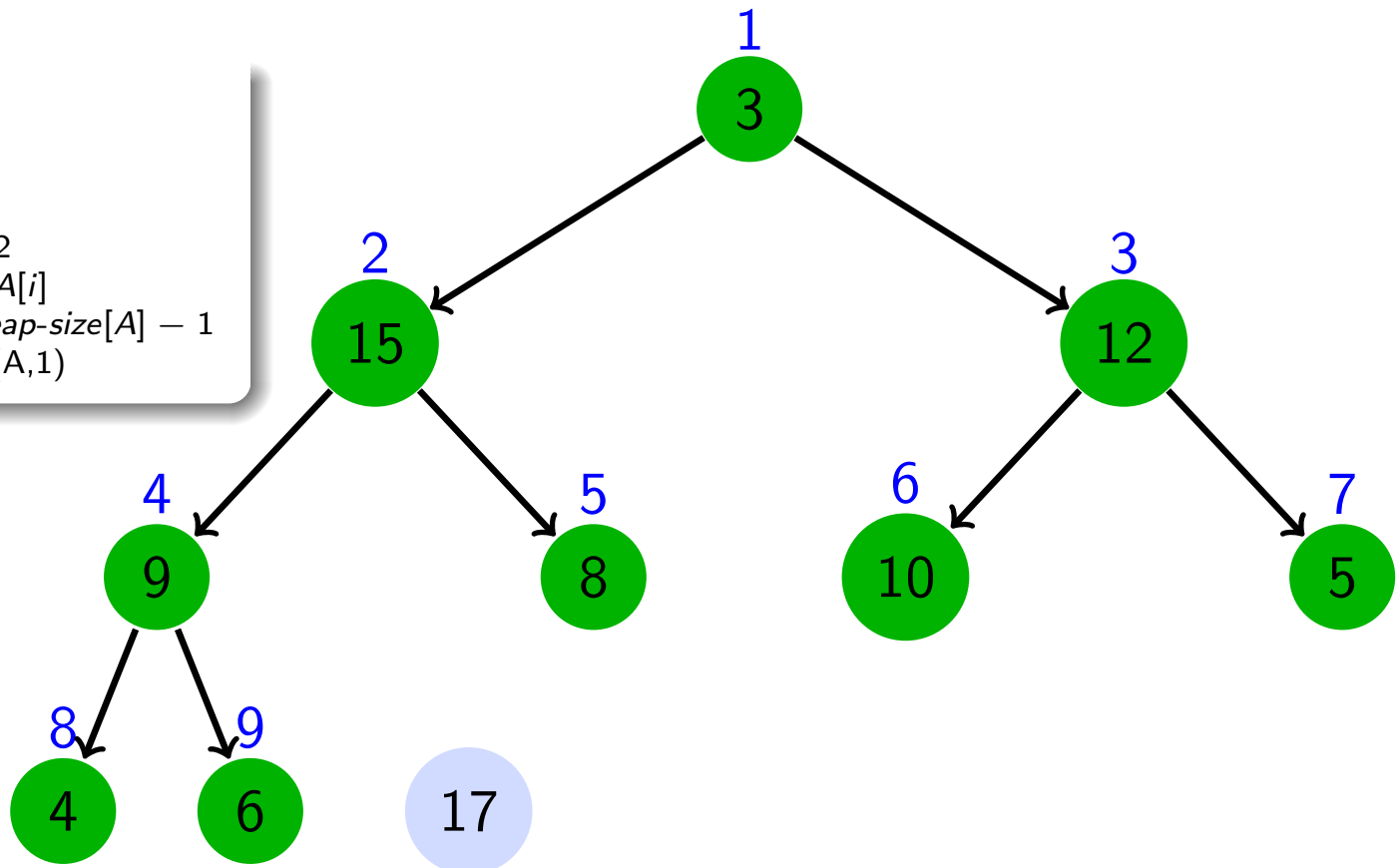

$i = 10$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 17 | 15 | 12 | 9 | 8 | 10 | 5 | 4 | 6 | 3 |

# HEAP-SORT

HEAP-SORT($A$)

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



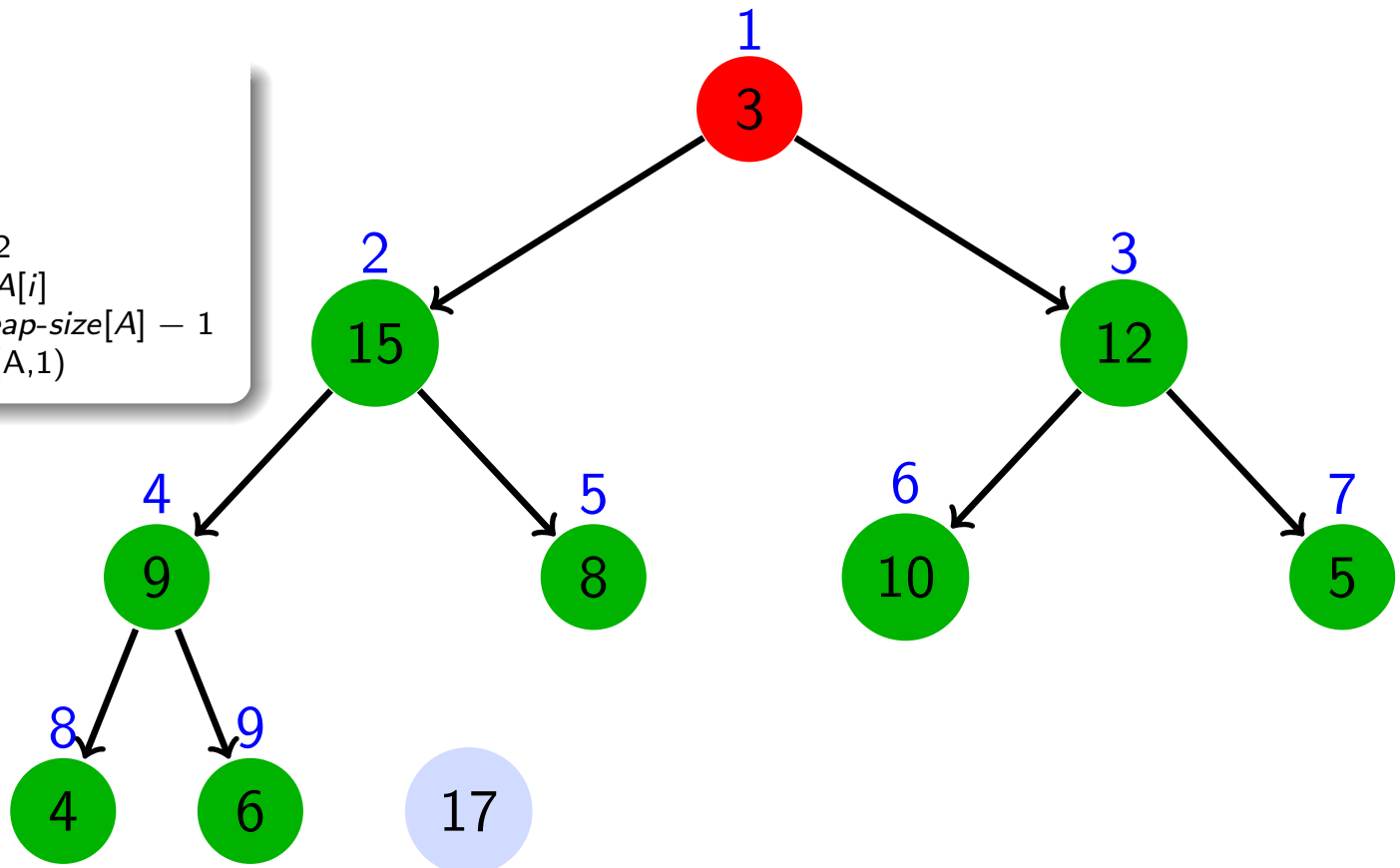

$i = 10$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 15 | 12 | 9 | 8 | 10 | 5 | 4 | 6 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



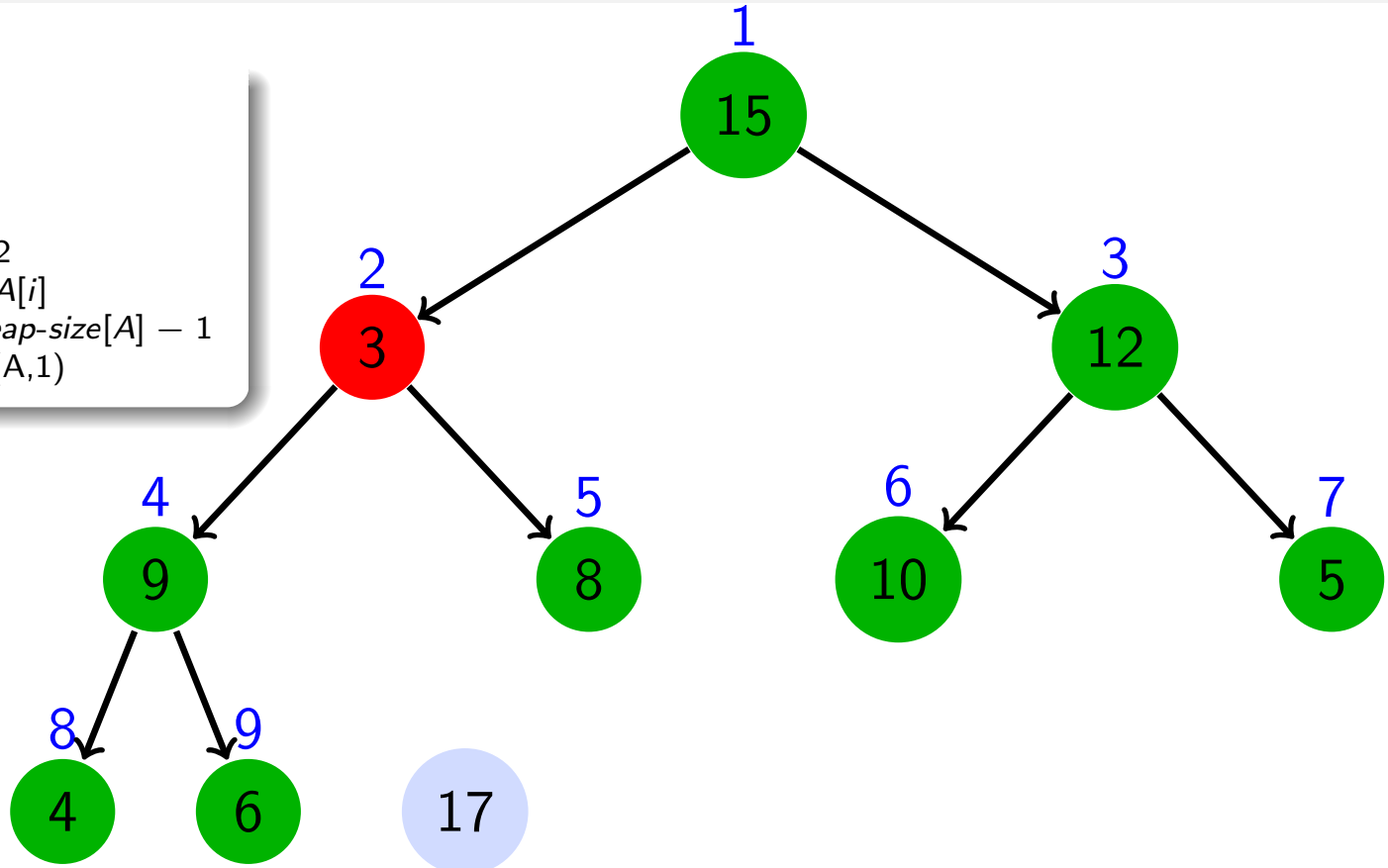

$i = 10$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 15 | 12 | 9 | 8 | 10 | 5 | 4 | 6 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



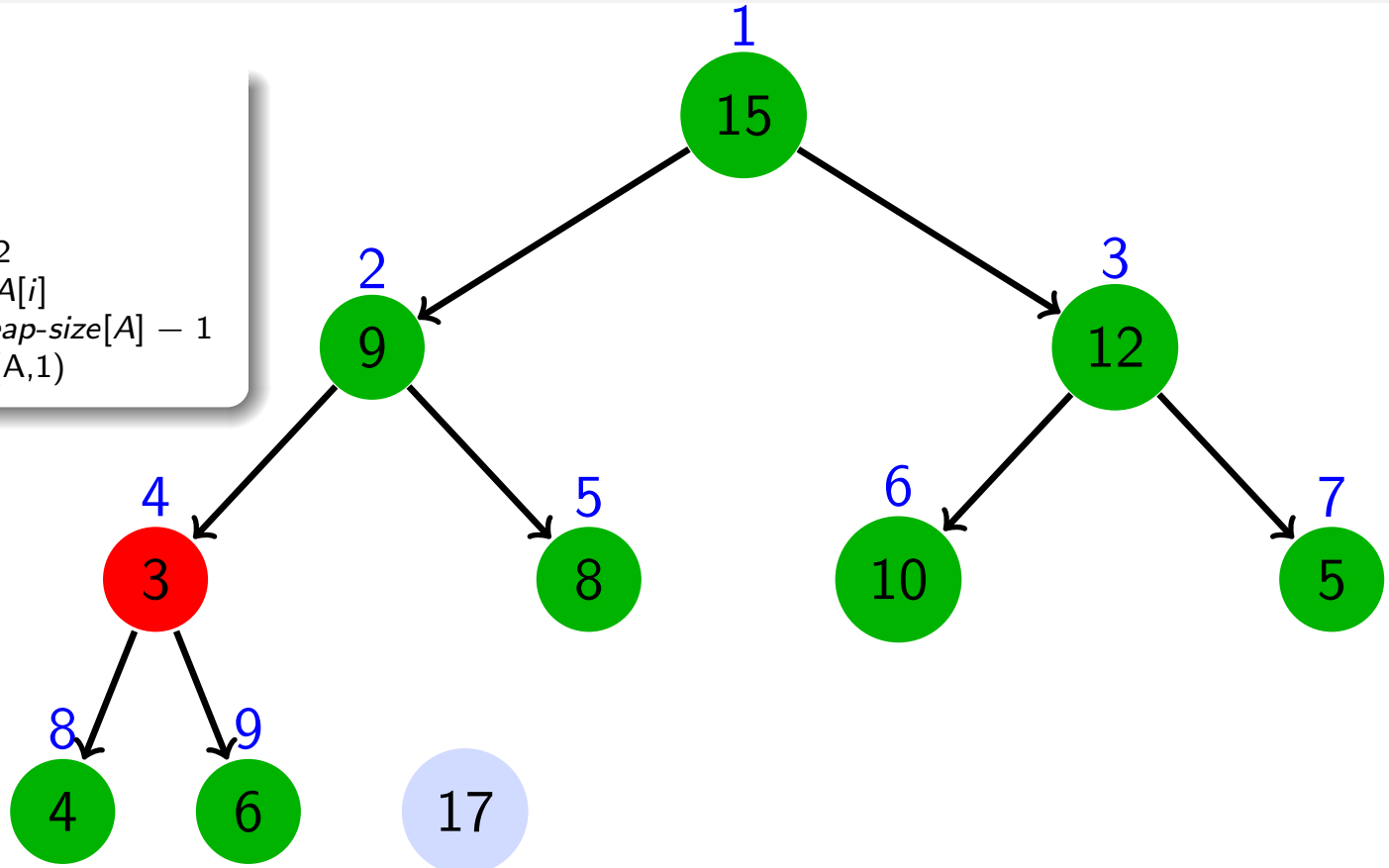

$i = 10$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 15 | 3 | 12 | 9 | 8 | 10 | 5 | 4 | 6 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



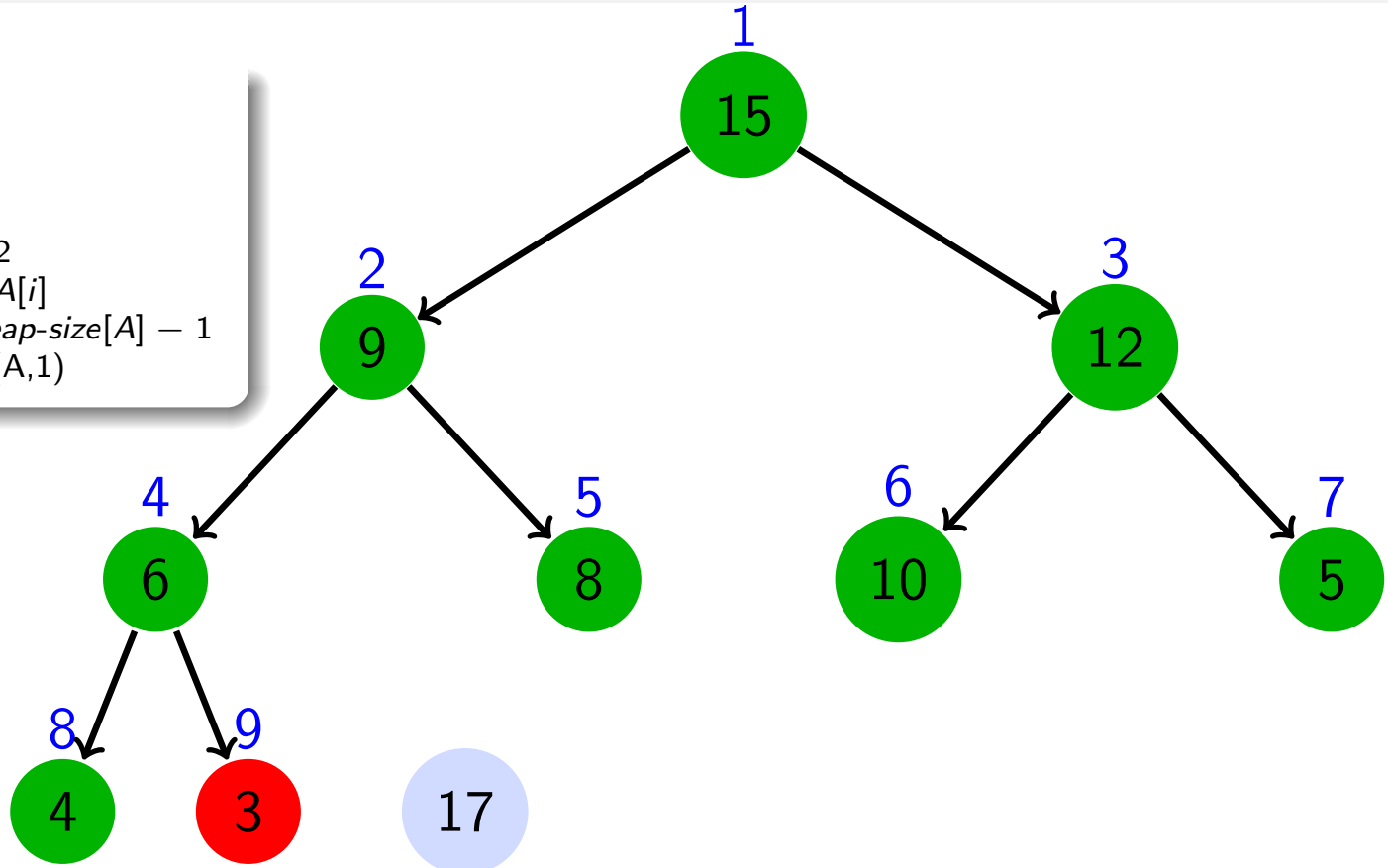

$i = 10$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 15 | 9 | 12 | 3 | 8 | 10 | 5 | 4 | 6 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



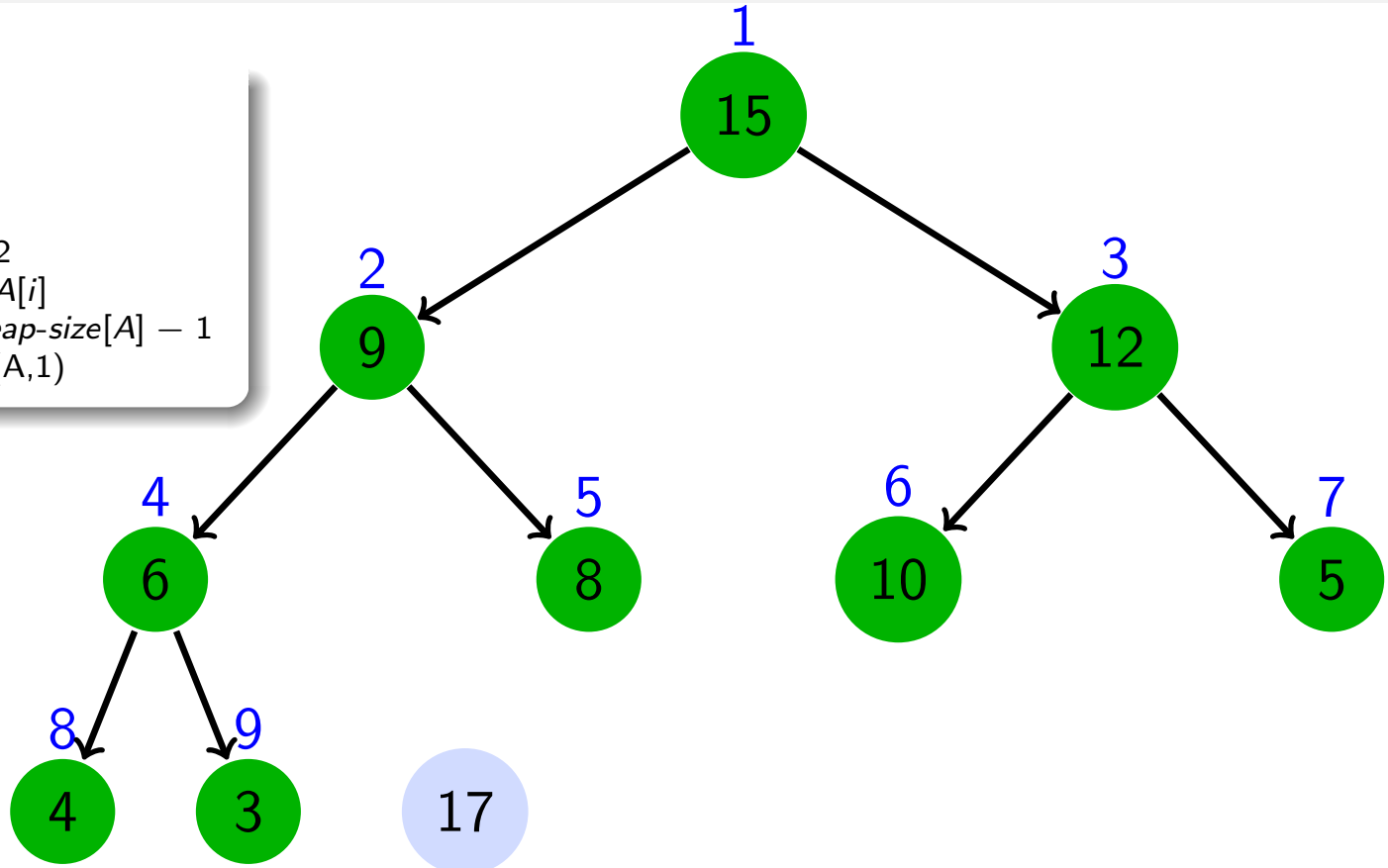

$i = 10$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 15 | 9 | 12 | 6 | 8 | 10 | 5 | 4 | 3 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



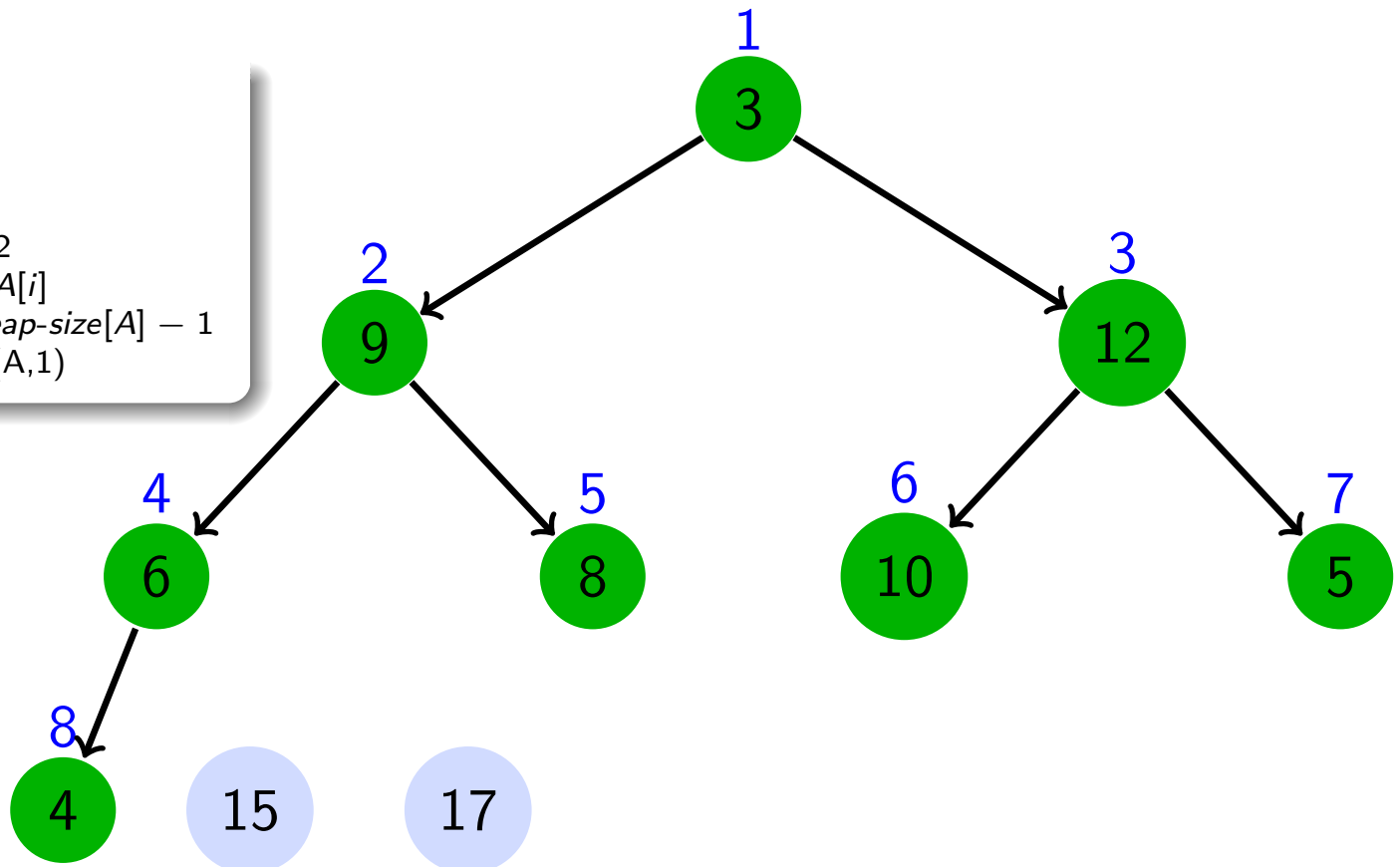

$i = 9$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 15 | 9 | 12 | 6 | 8 | 10 | 5 | 4 | 3 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



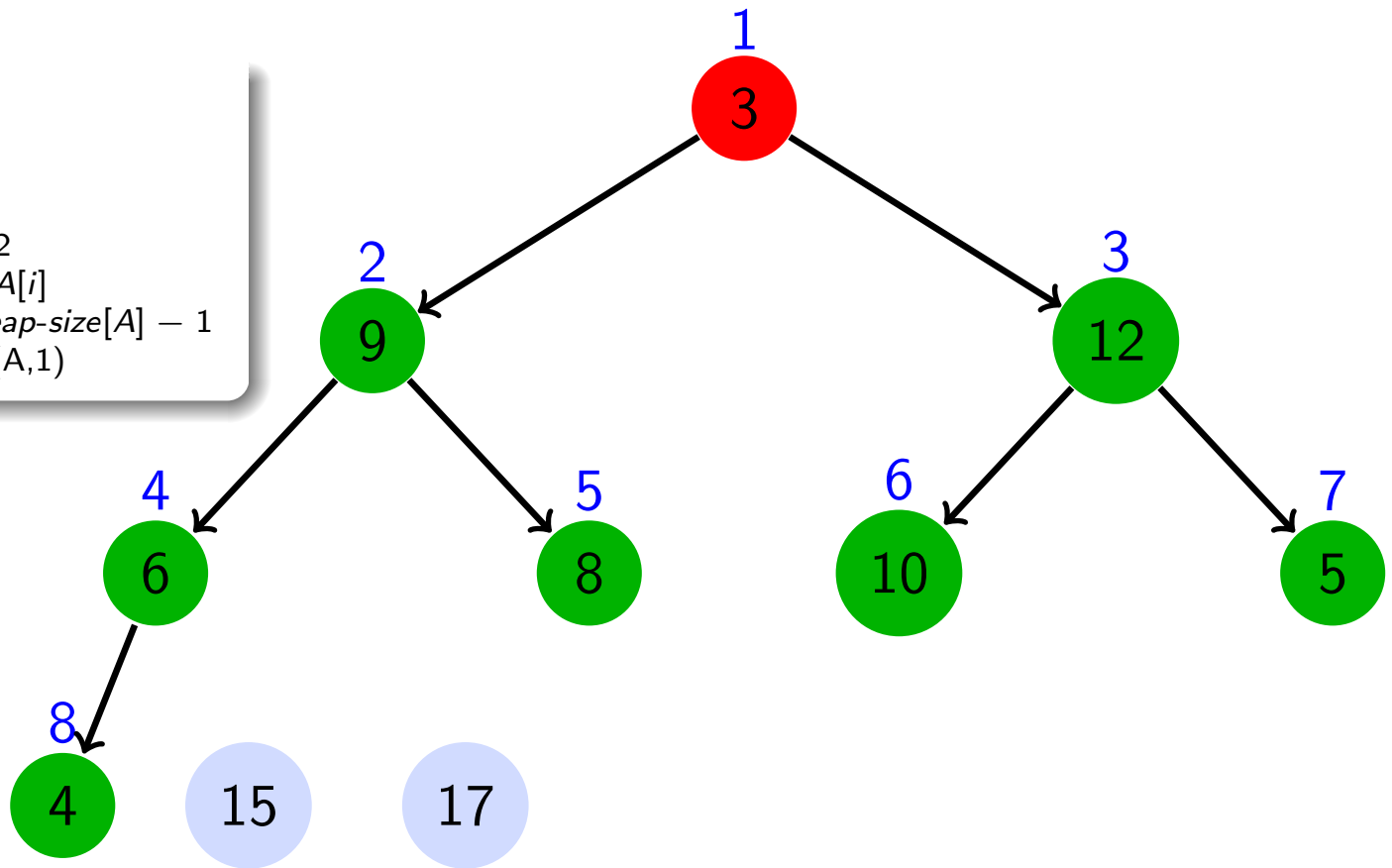

$i = 9$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 12 | 6 | 8 | 10 | 5 | 4 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



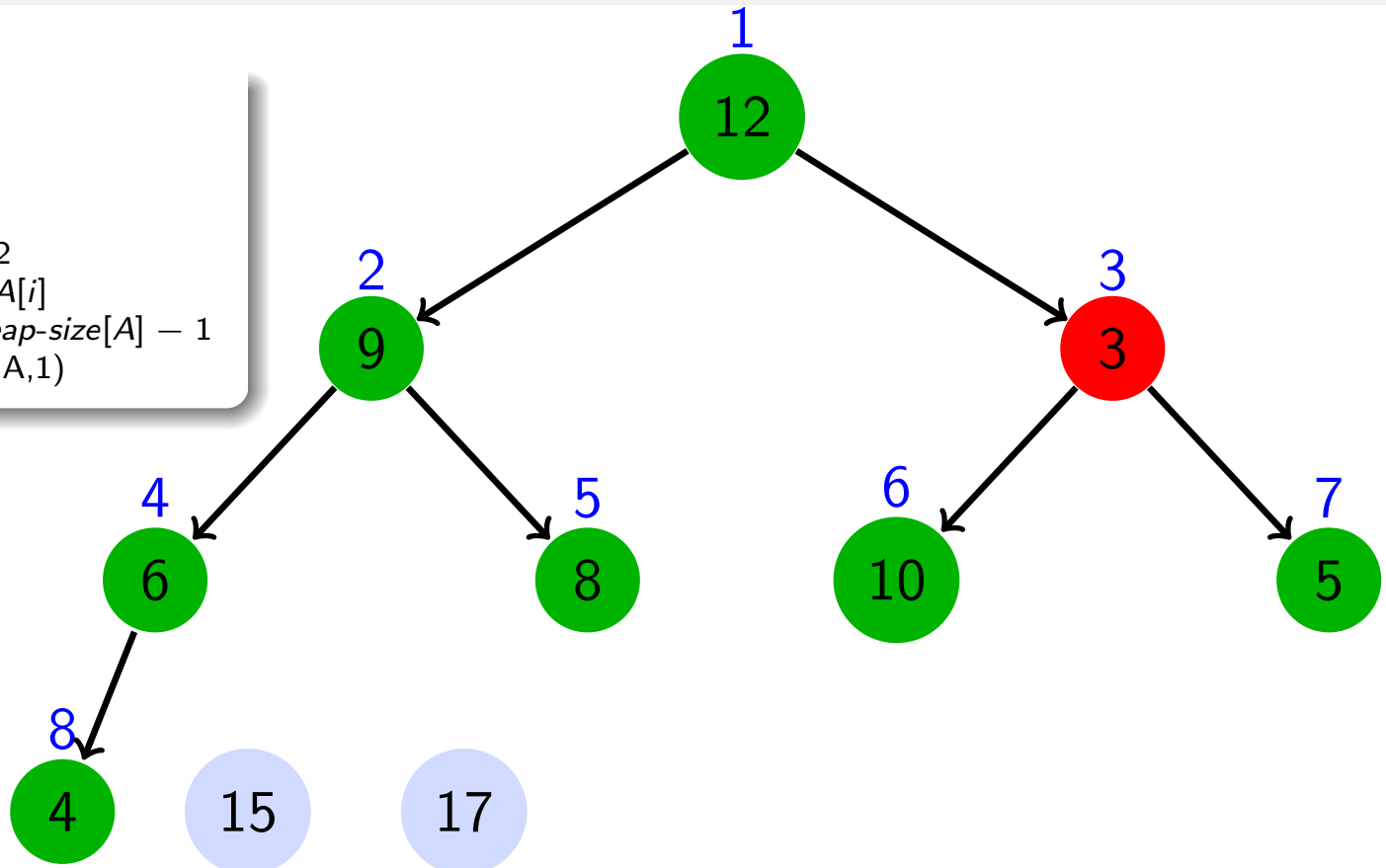

$i = 9$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 12 | 6 | 8 | 10 | 5 | 4 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



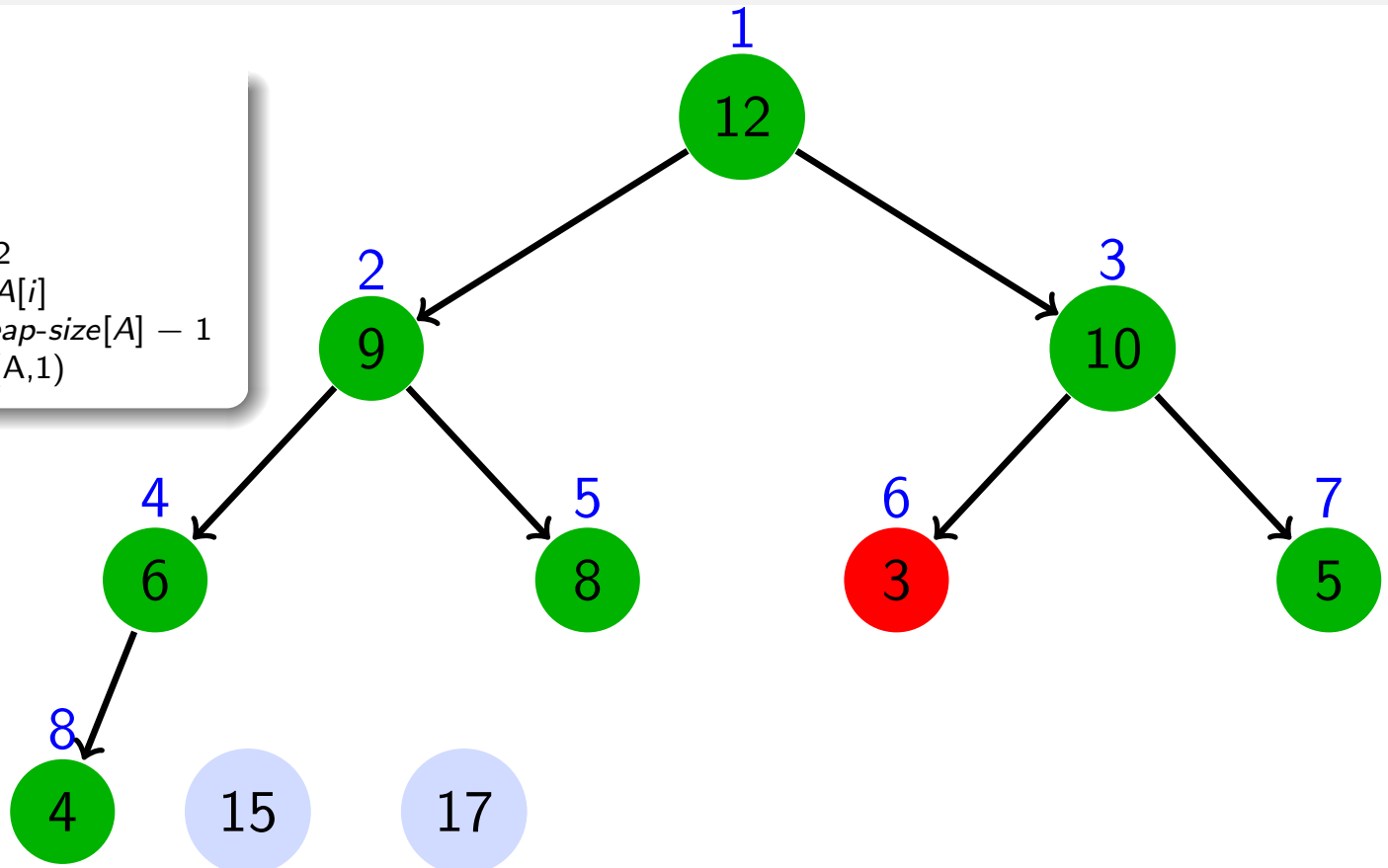

$i = 9$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 3 | 6 | 8 | 10 | 5 | 4 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



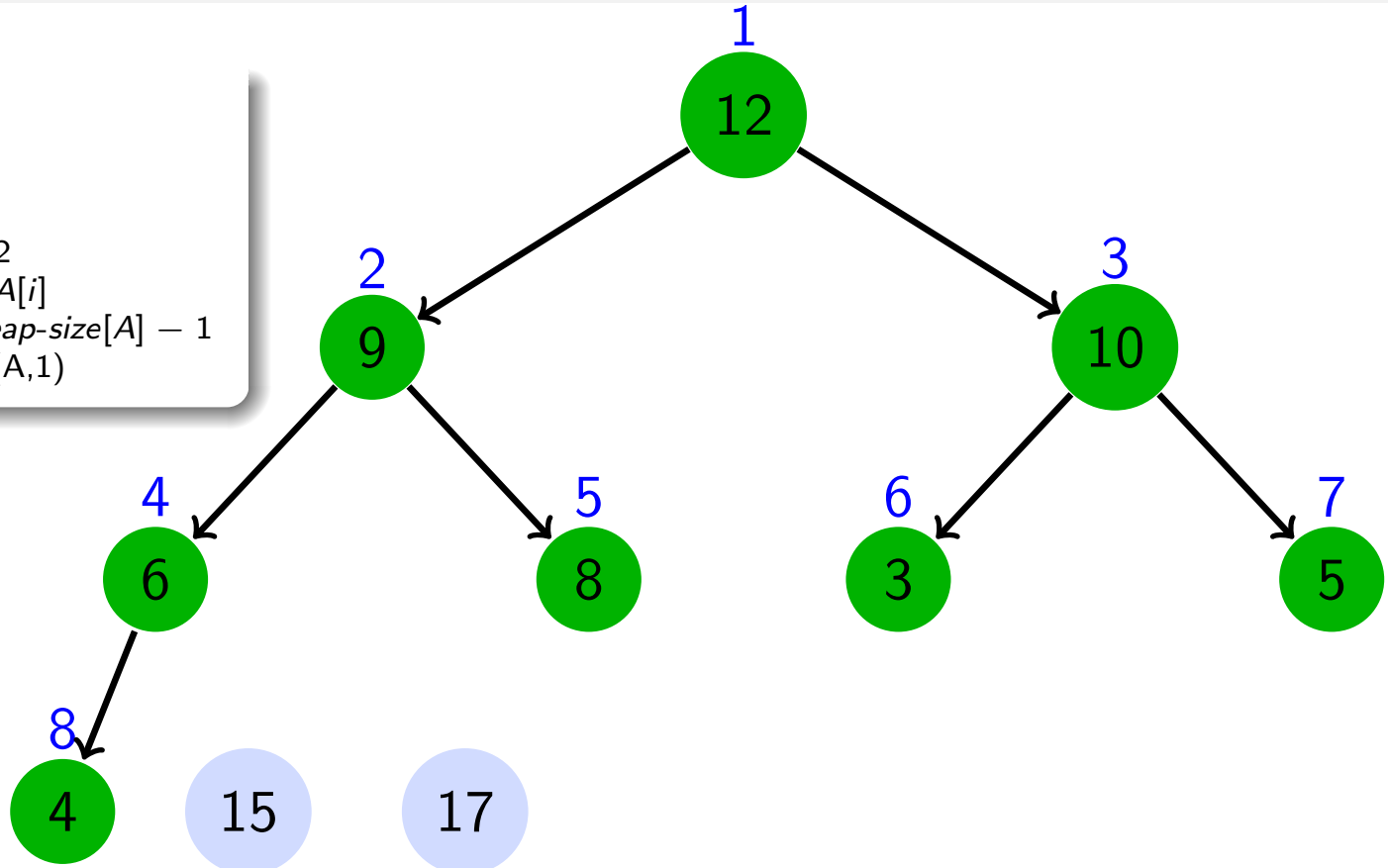

$i = 9$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 10 | 6 | 8 | 3 | 5 | 4 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



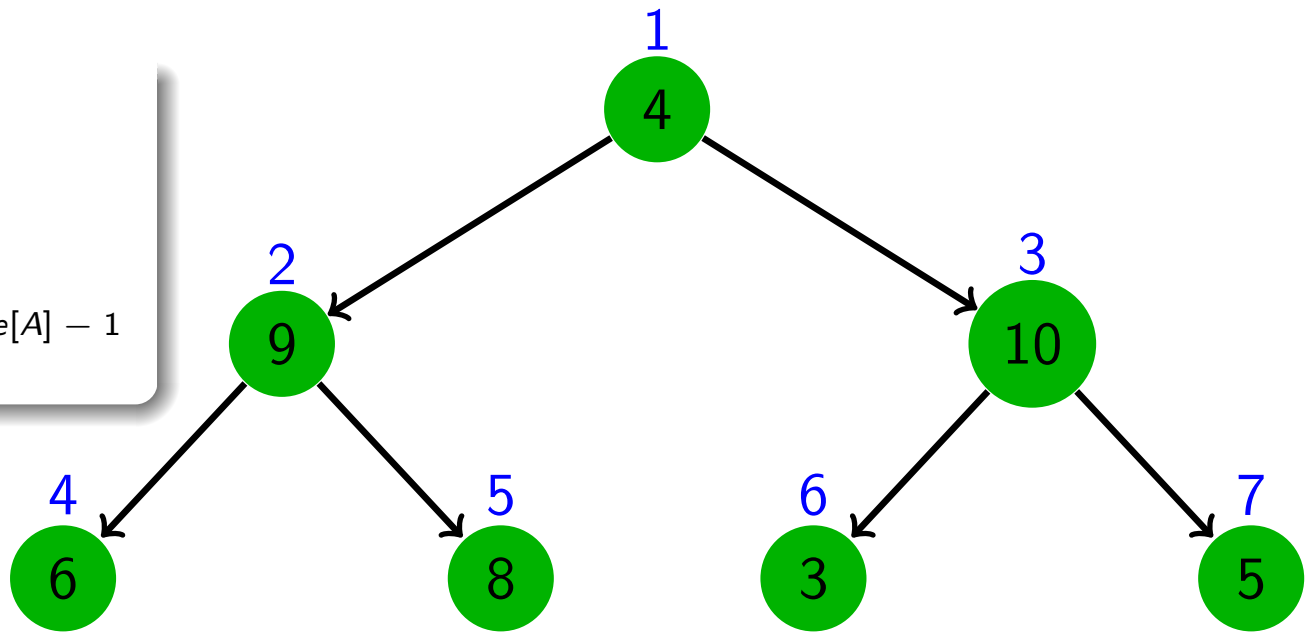

$i = 8$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 10 | 6 | 8 | 3 | 5 | 4 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




12 15 17

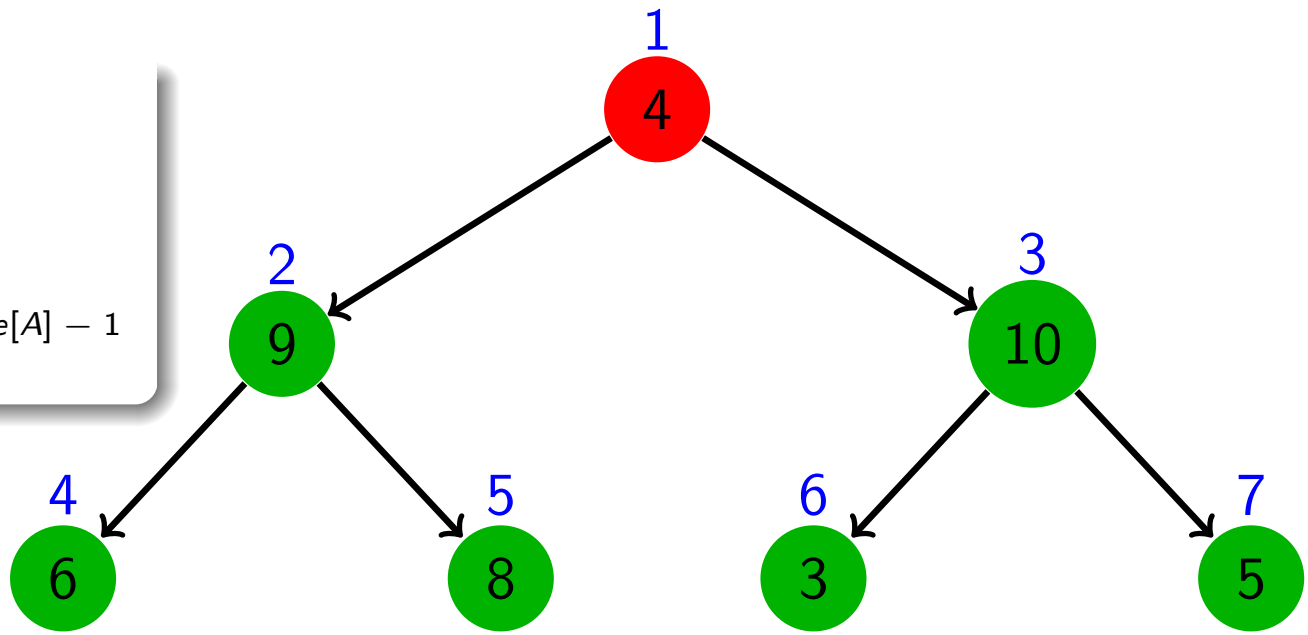
$i = 8$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 9 | 10 | 6 | 8 | 3 | 5 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




12 15 17

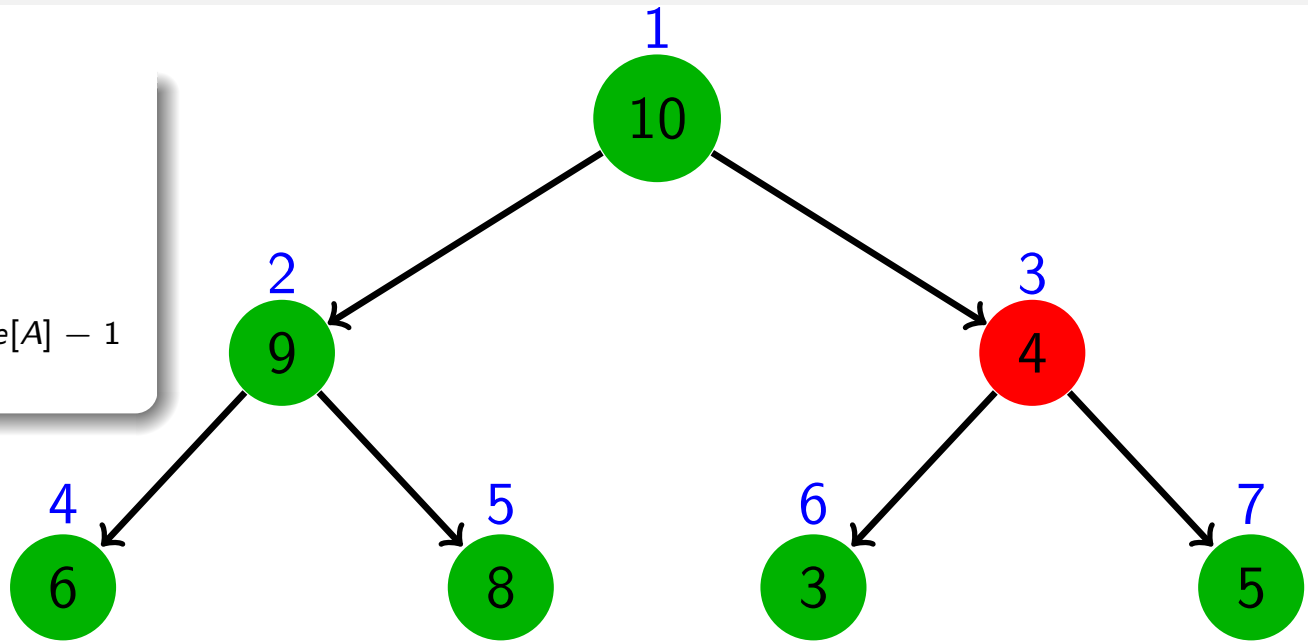
$i = 8$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 9 | 10 | 6 | 8 | 3 | 5 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




12 15 17

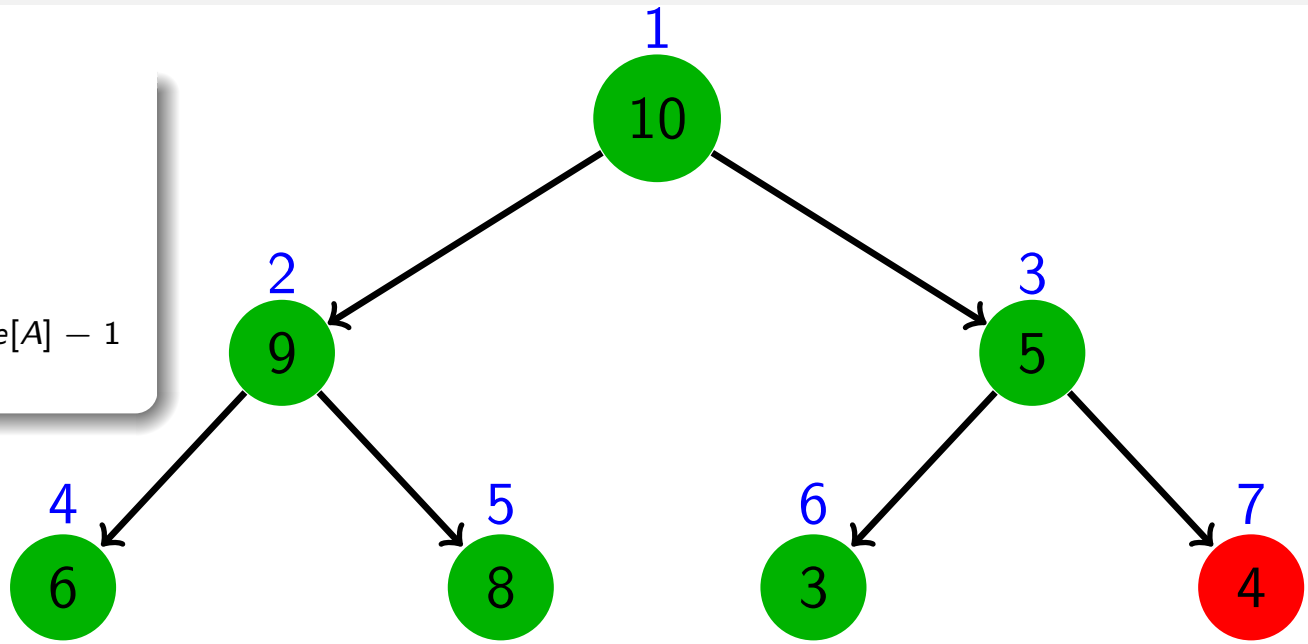
$i = 8$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 9 | 4 | 6 | 8 | 3 | 5 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




12 15 17

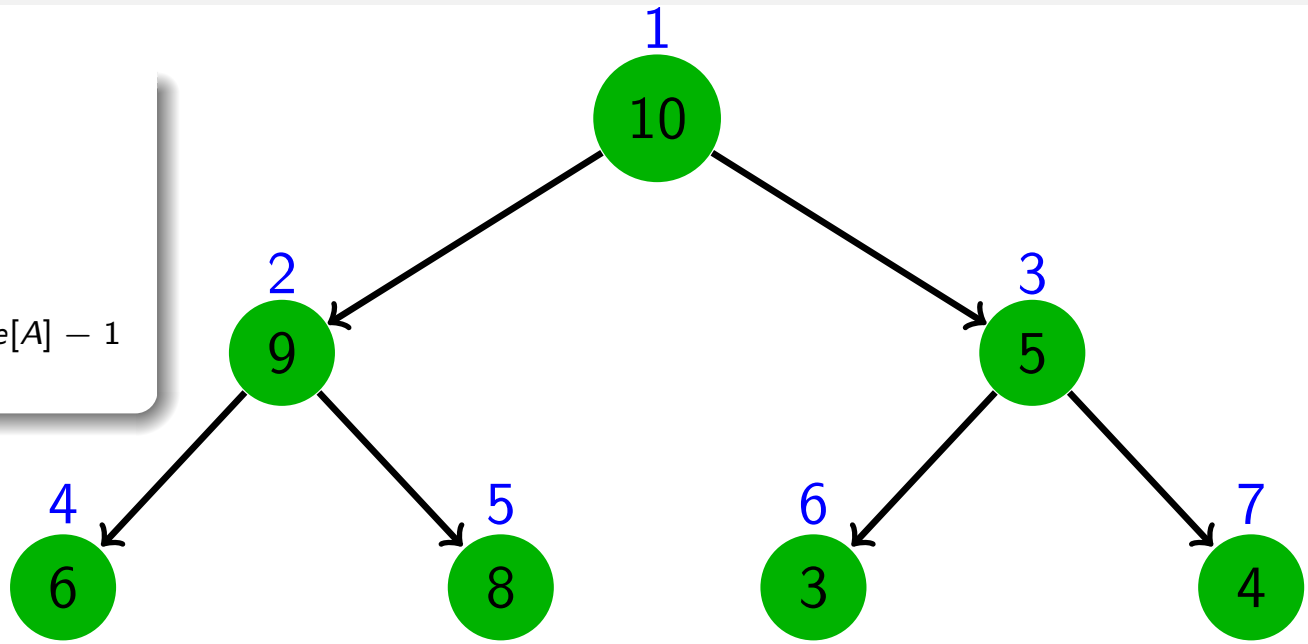
$i = 8$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 9 | 5 | 6 | 8 | 3 | 4 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




12 15 17

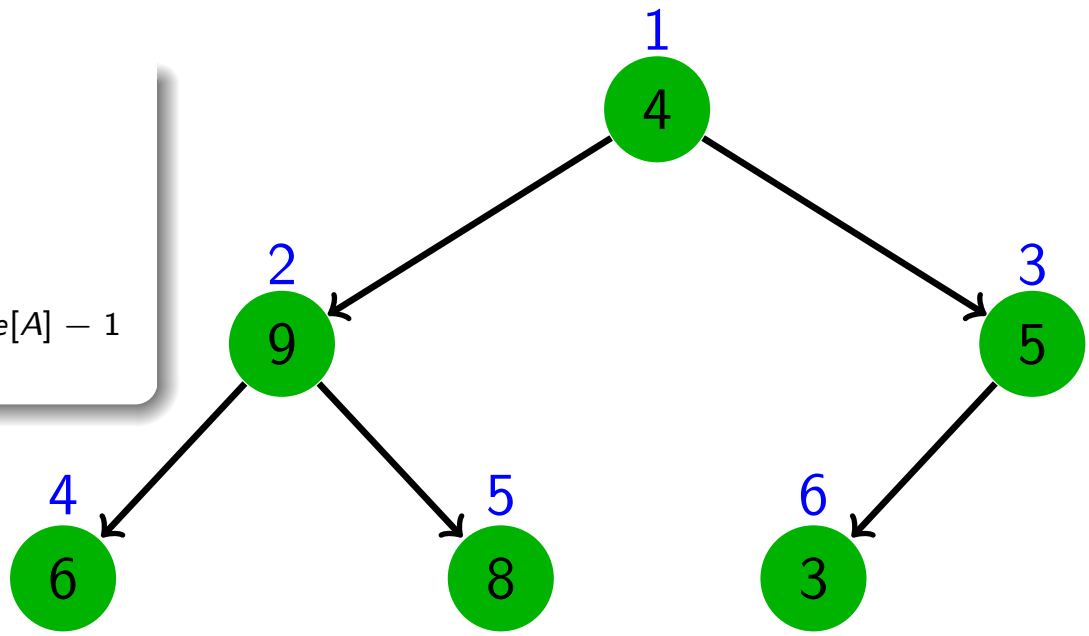
$i = 7$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 10 | 9 | 5 | 6 | 8 | 3 | 4 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




10 12 15 17

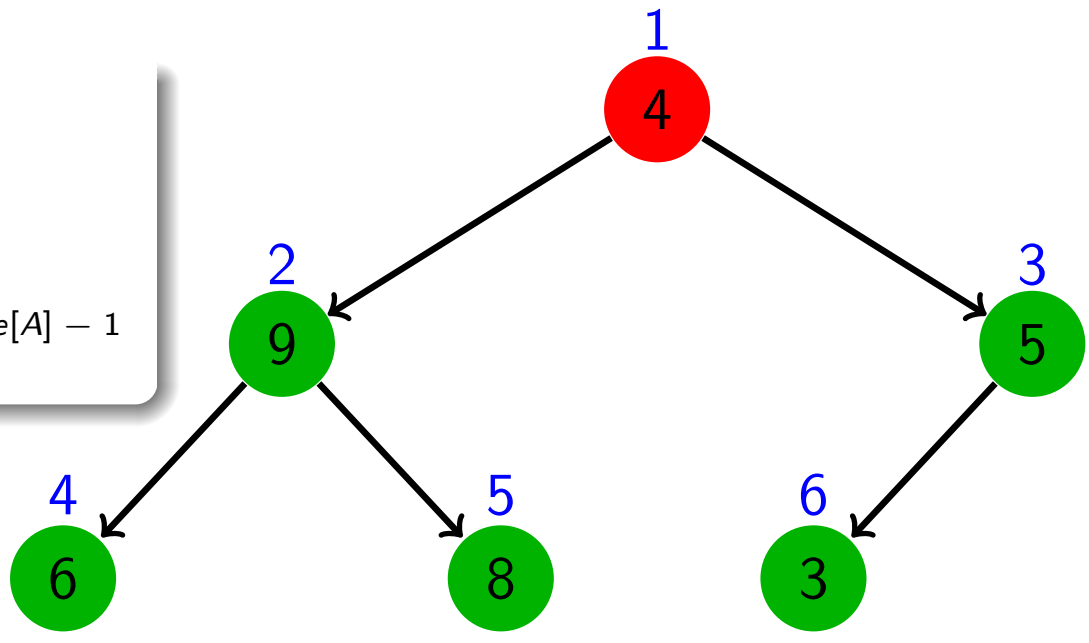
$i = 7$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 9 | 5 | 6 | 8 | 3 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




10 12 15 17

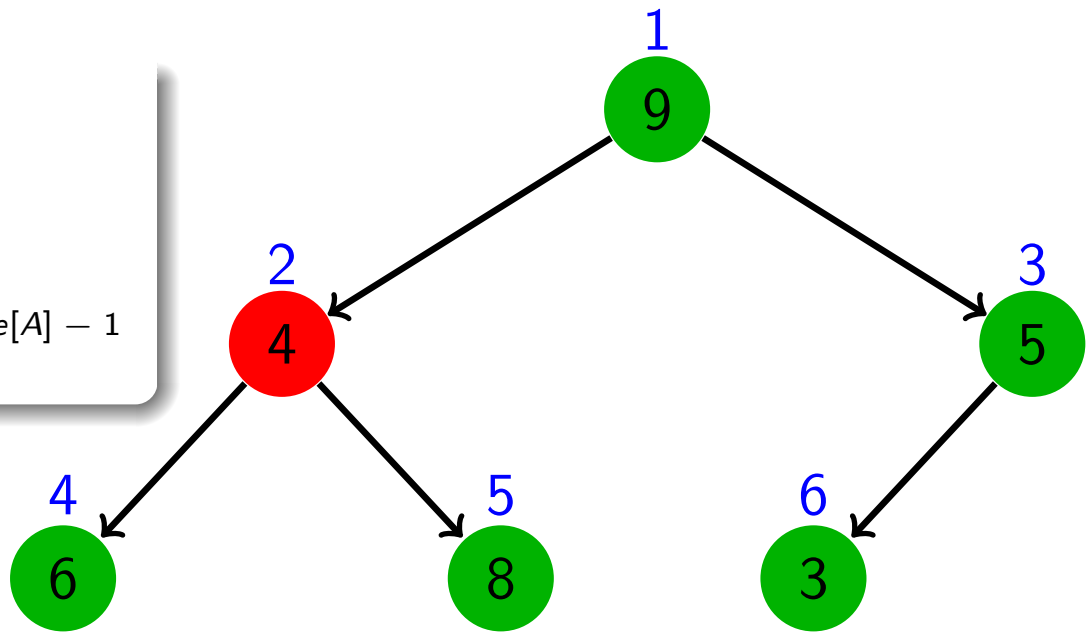
$i = 7$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 9 | 5 | 6 | 8 | 3 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




10 12 15 17

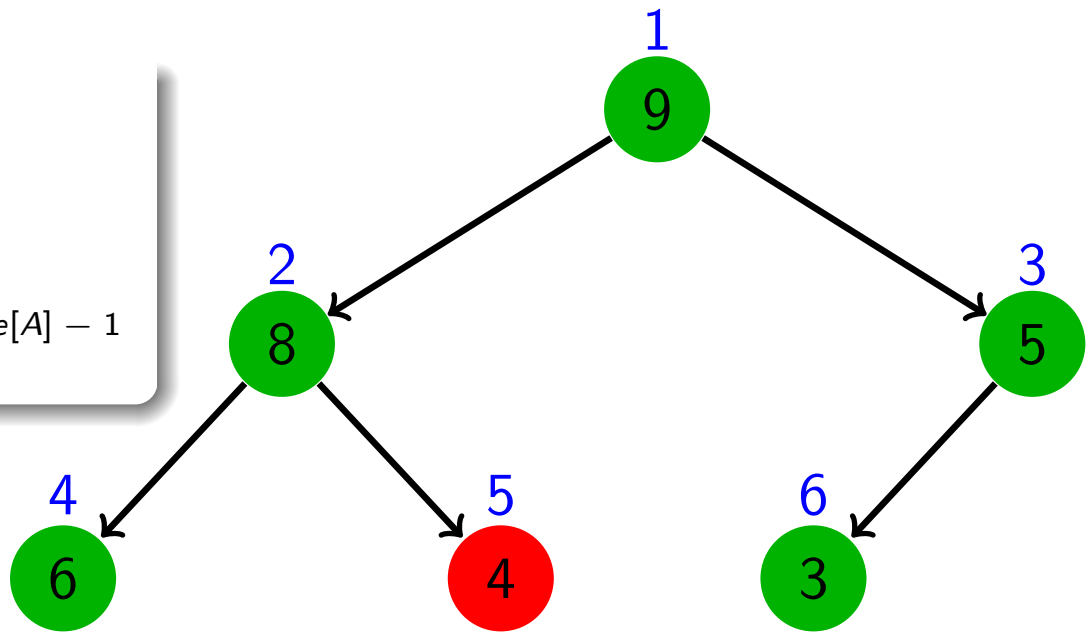
$i = 7$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 4 | 5 | 6 | 8 | 3 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




10 12 15 17

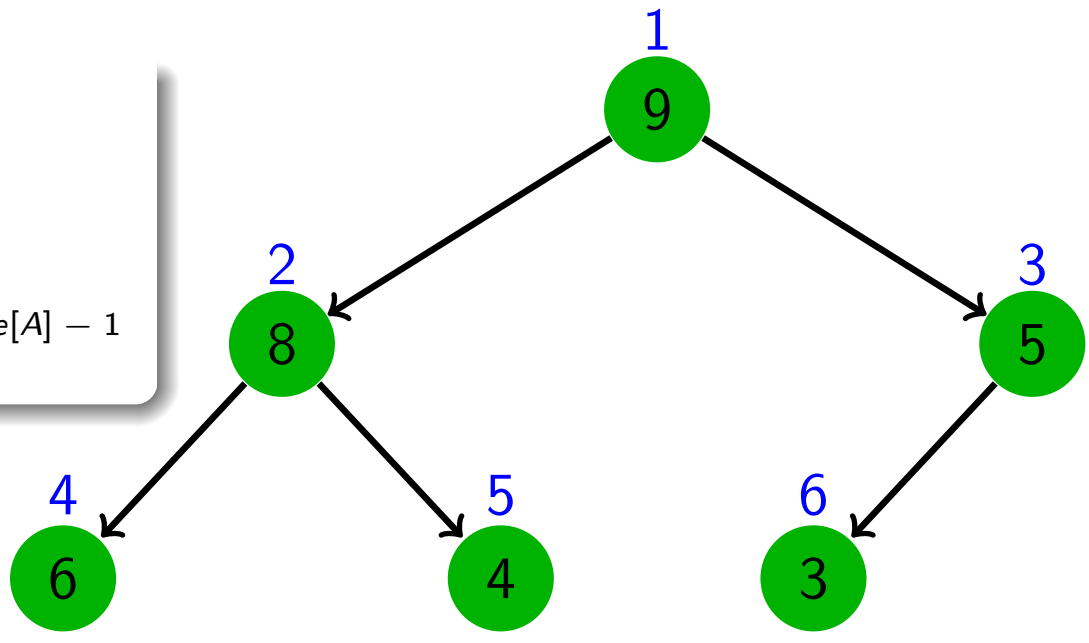
$i = 7$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 8 | 5 | 6 | 4 | 3 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




10 12 15 17

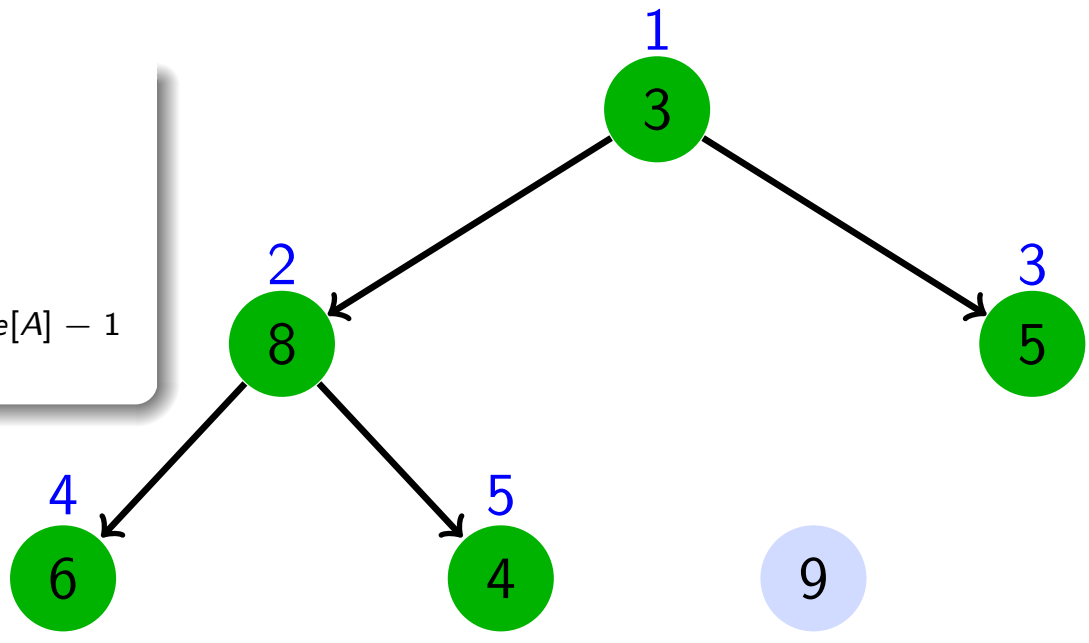
$i = 6$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 8 | 5 | 6 | 4 | 3 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



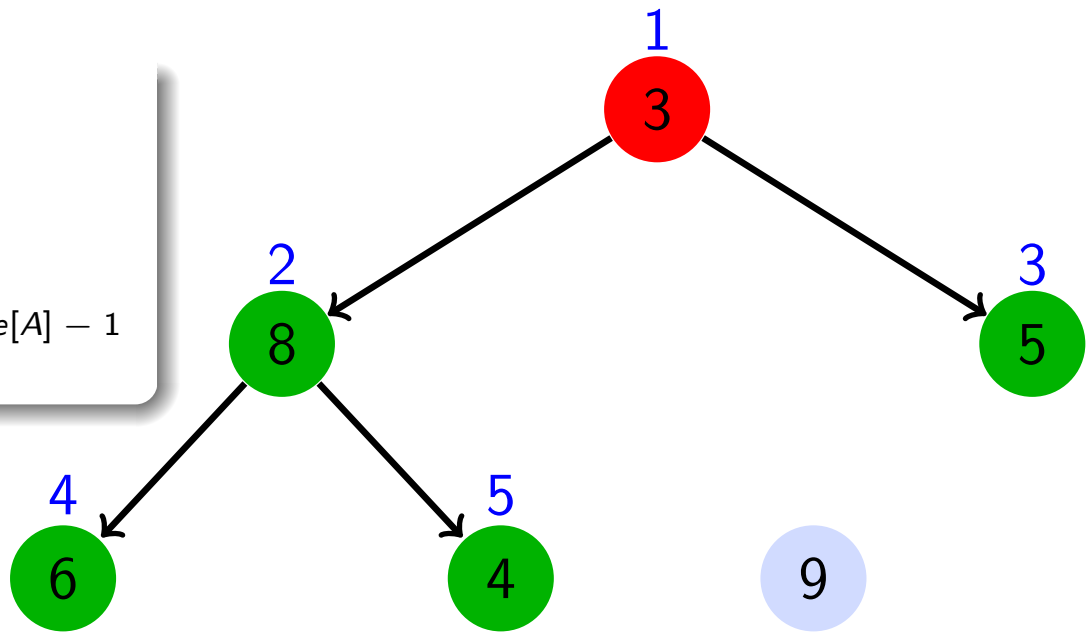

$i = 6$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 8 | 5 | 6 | 4 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



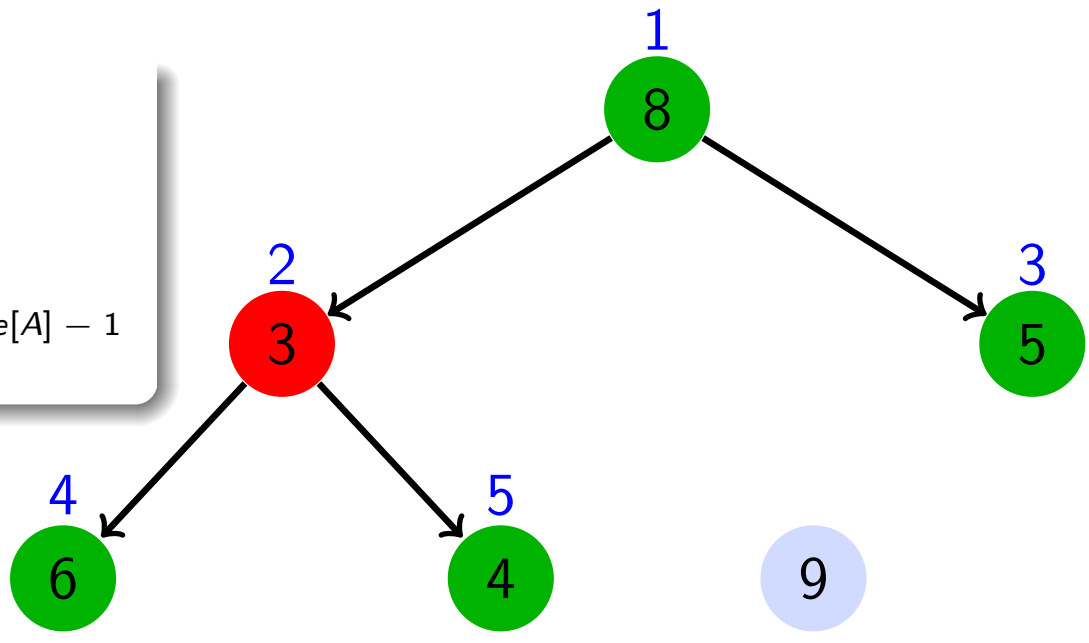

$i = 6$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 8 | 5 | 6 | 4 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



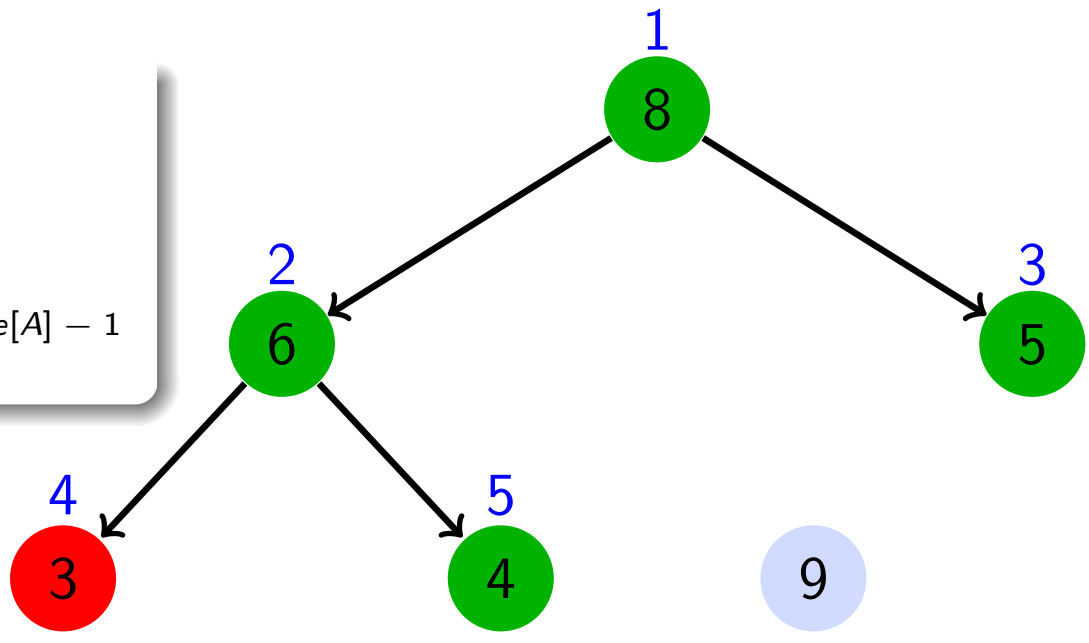

$i = 6$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 3 | 5 | 6 | 4 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




9 10 12 15 17

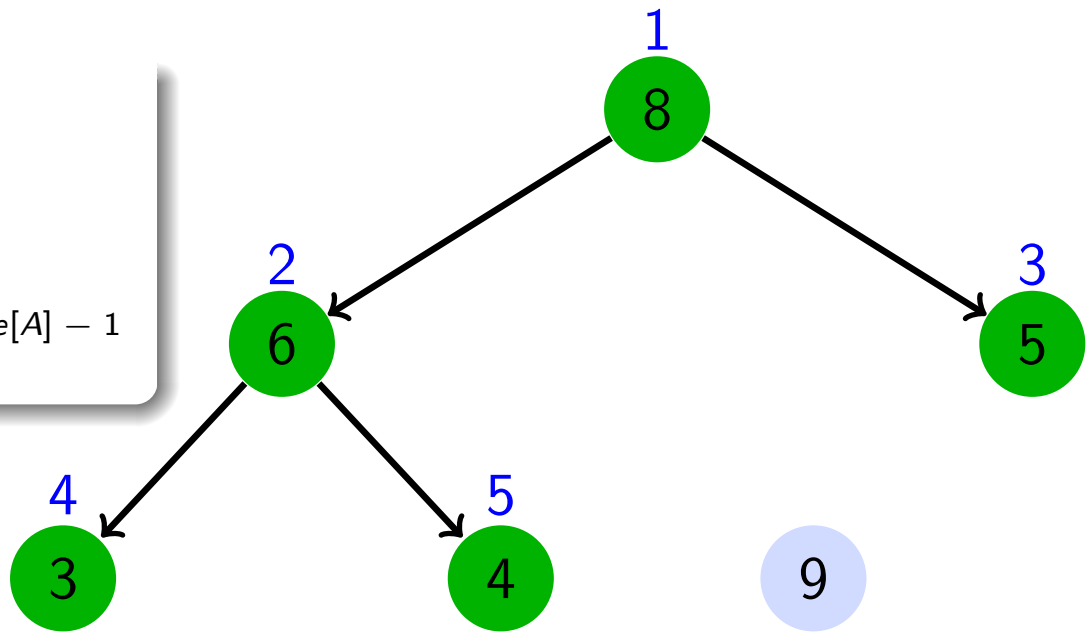
$i = 6$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 6 | 5 | 3 | 4 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




9 10 12 15 17

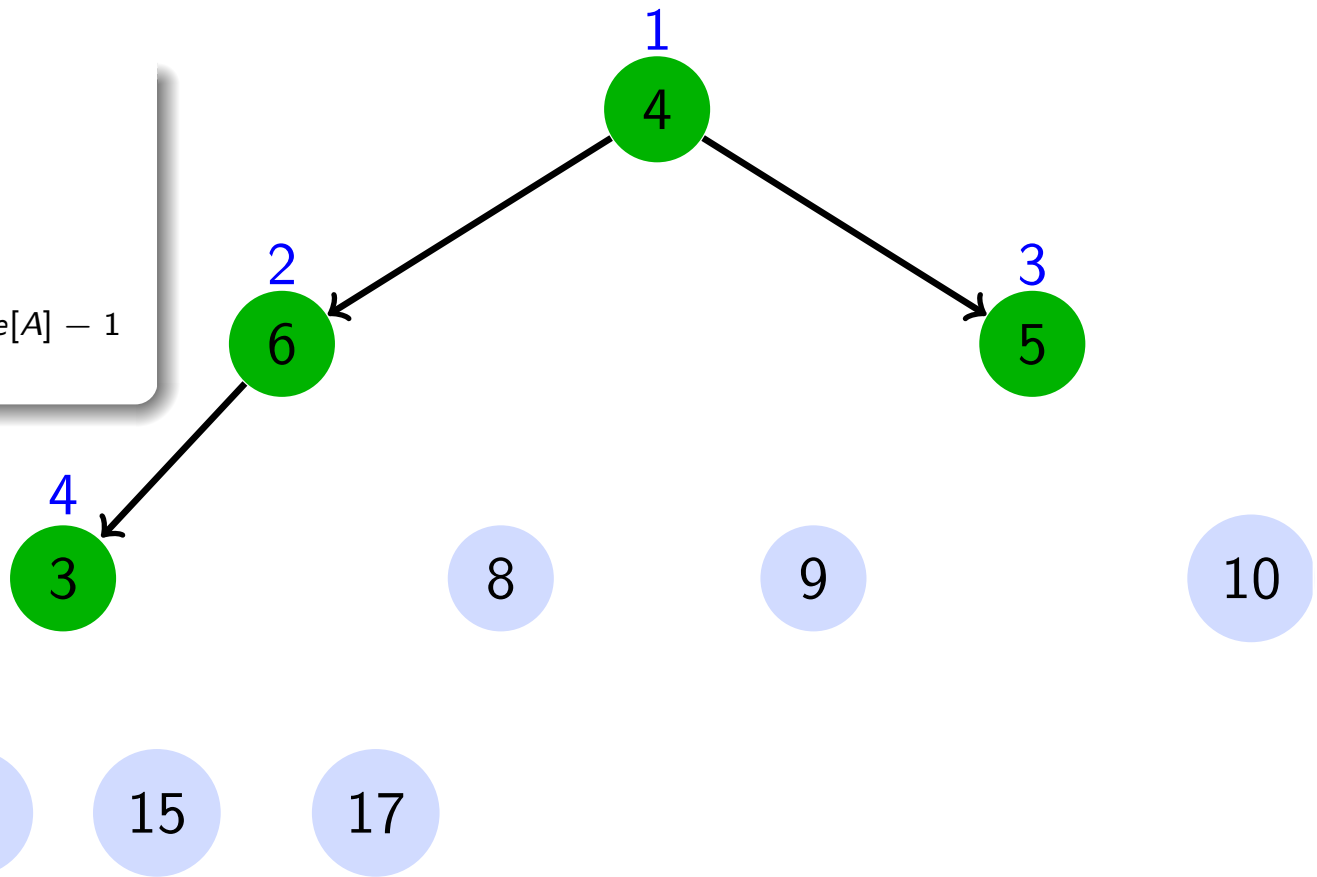
$i = 5$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 8 | 6 | 5 | 3 | 4 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



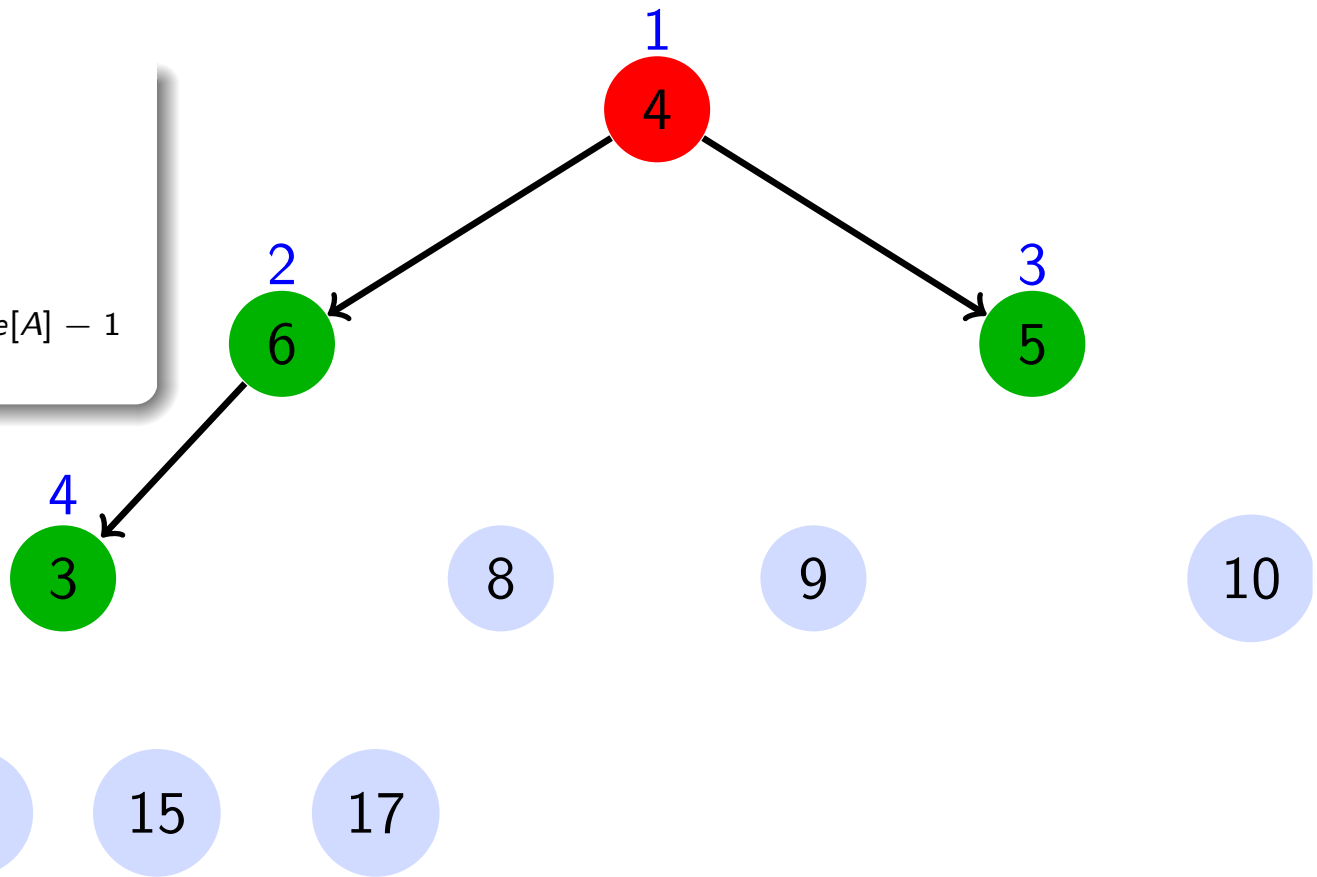

$i = 5$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | 5 | 3 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



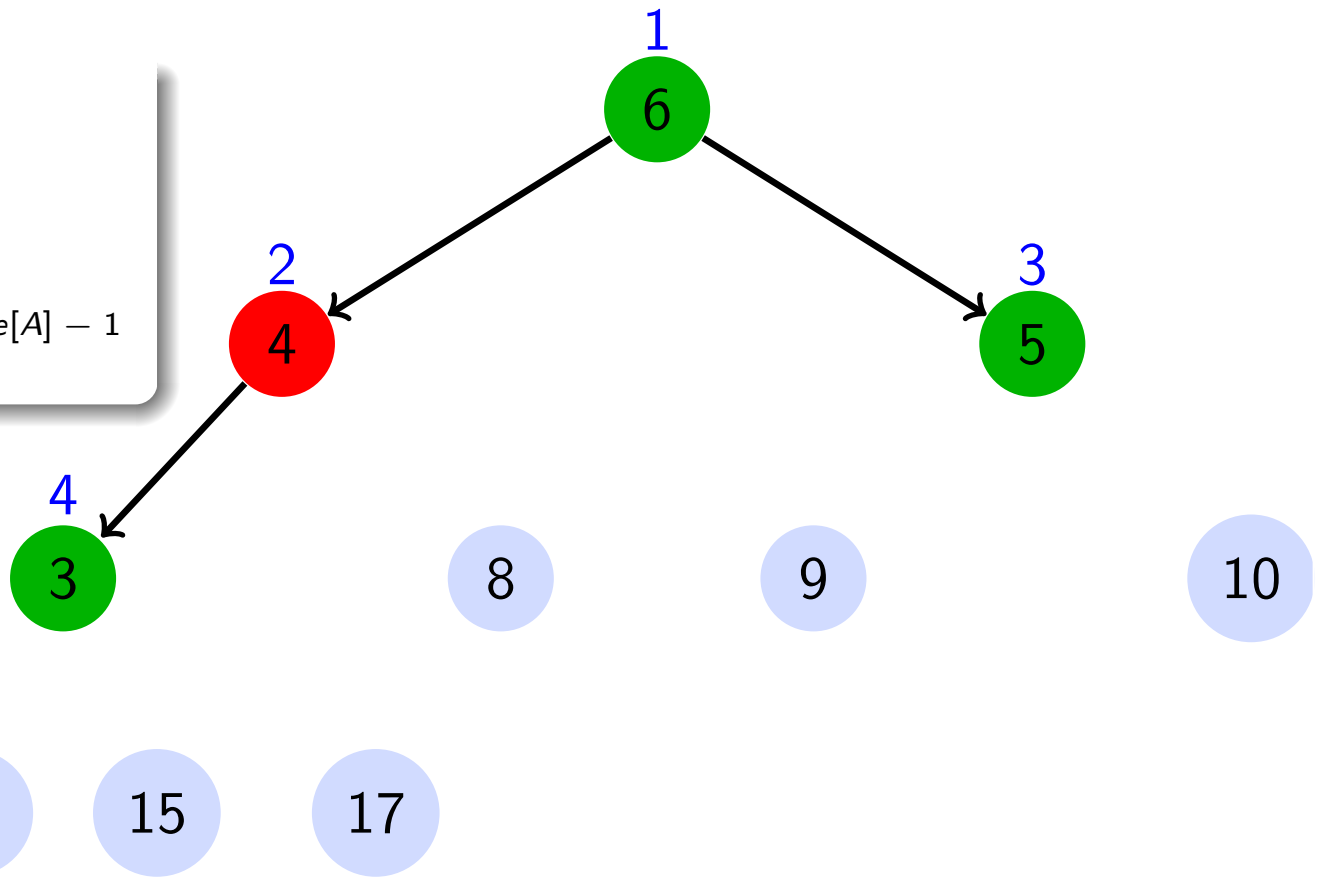

$i = 5$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | 5 | 3 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



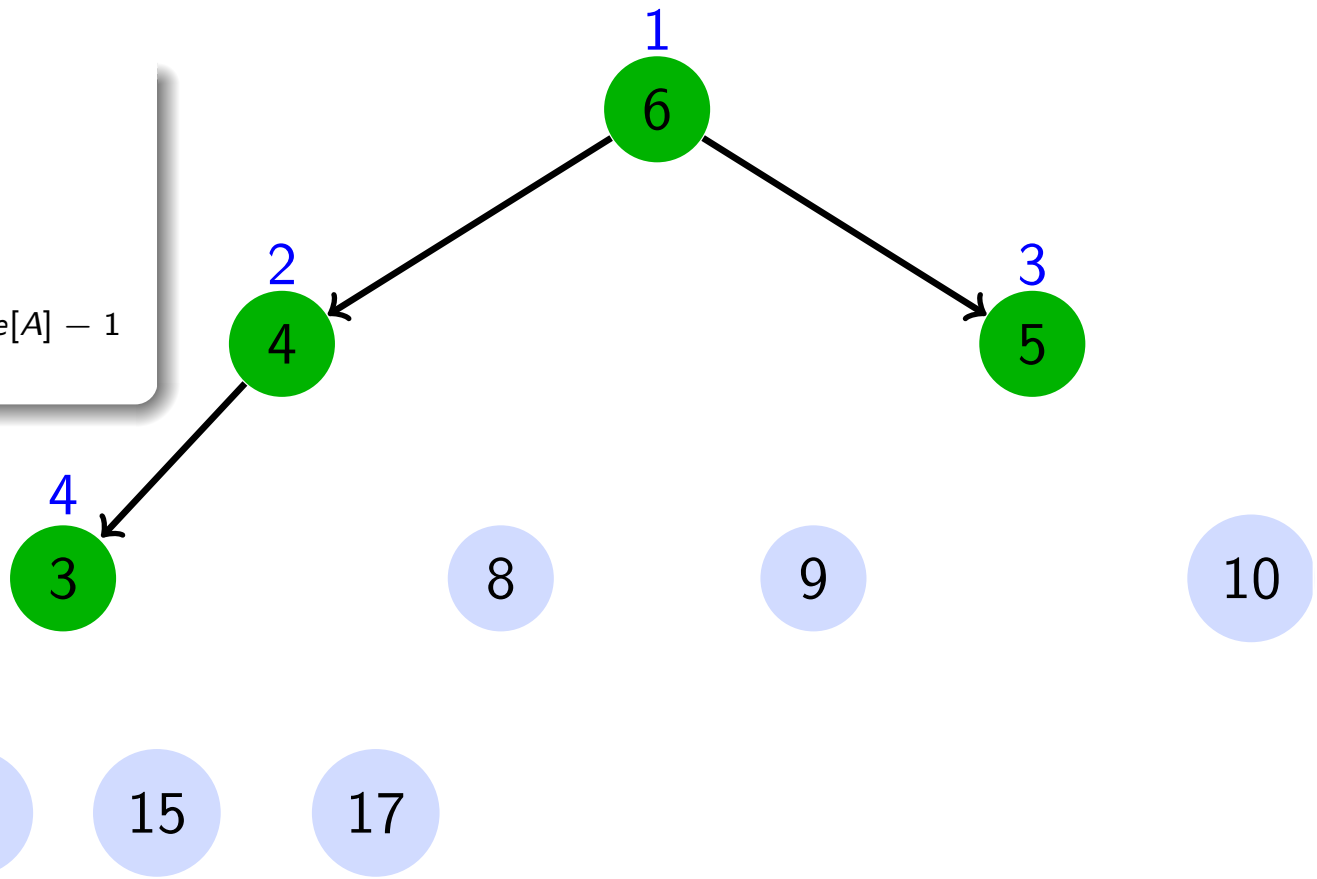

$i = 5$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 6 | 4 | 5 | 3 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



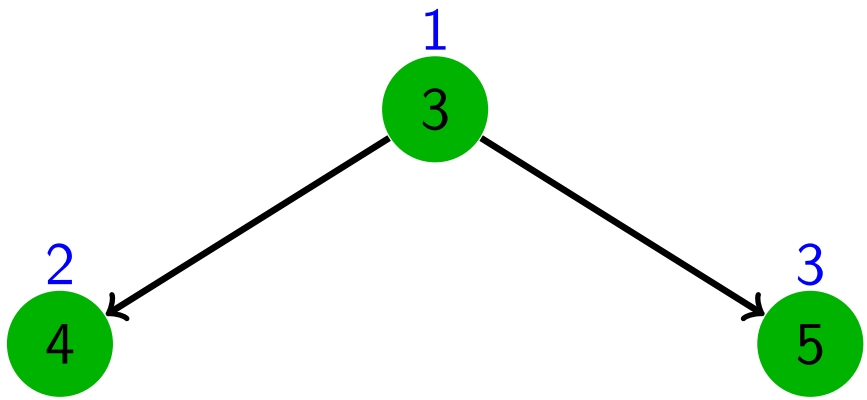

$i = 4$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 6 | 4 | 5 | 3 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




6 8 9 10

12 15 17

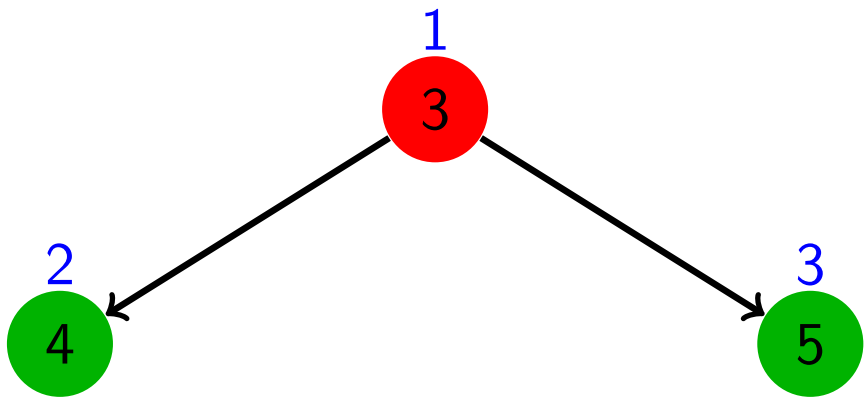
$i = 4$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




6 8 9 10

12 15 17

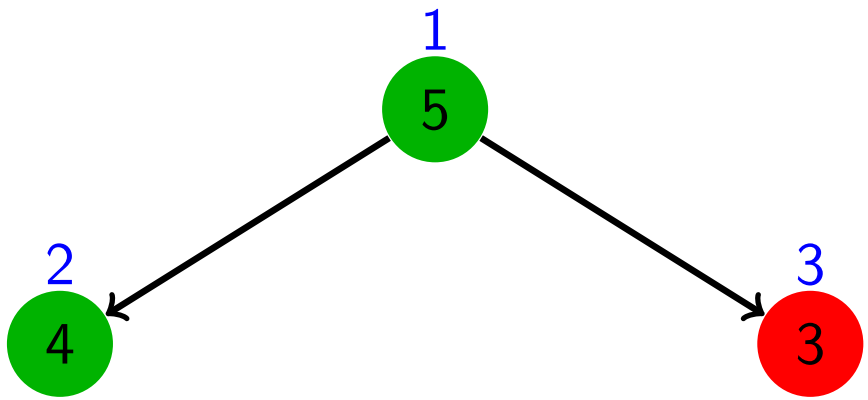
$i = 4$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




6 8 9 10

12 15 17

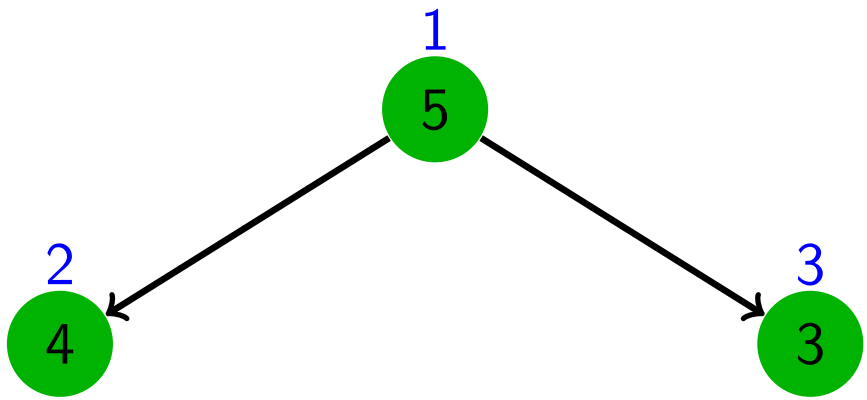
$i = 4$

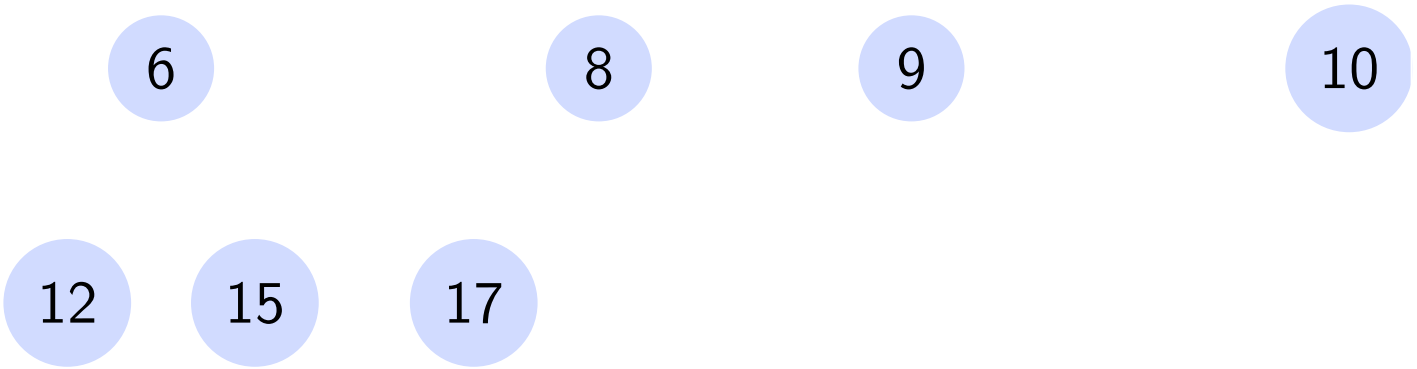
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 3 | 6 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



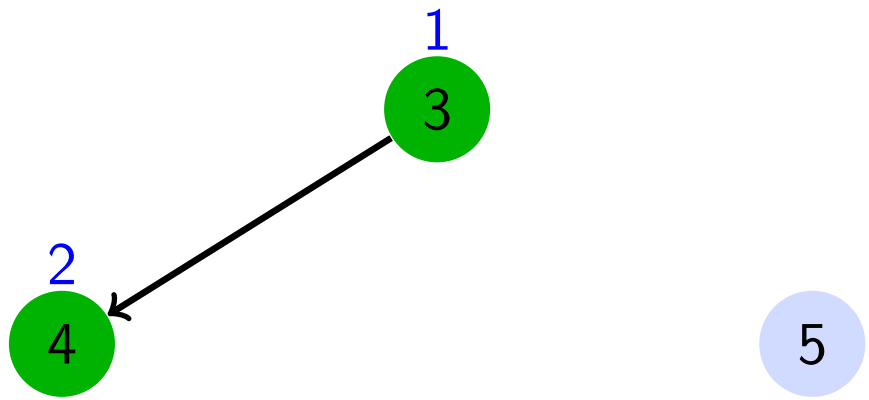

$i = 3$

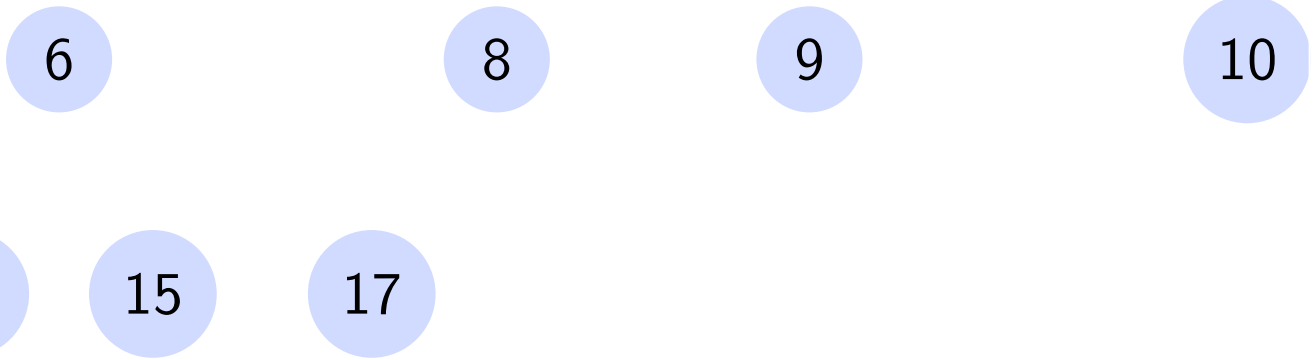
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 3 | 6 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

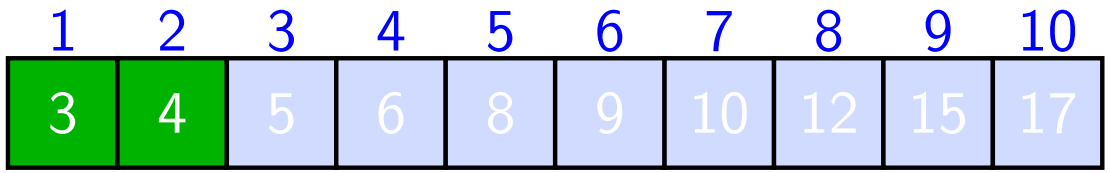
HEAP-SORT($A$)

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



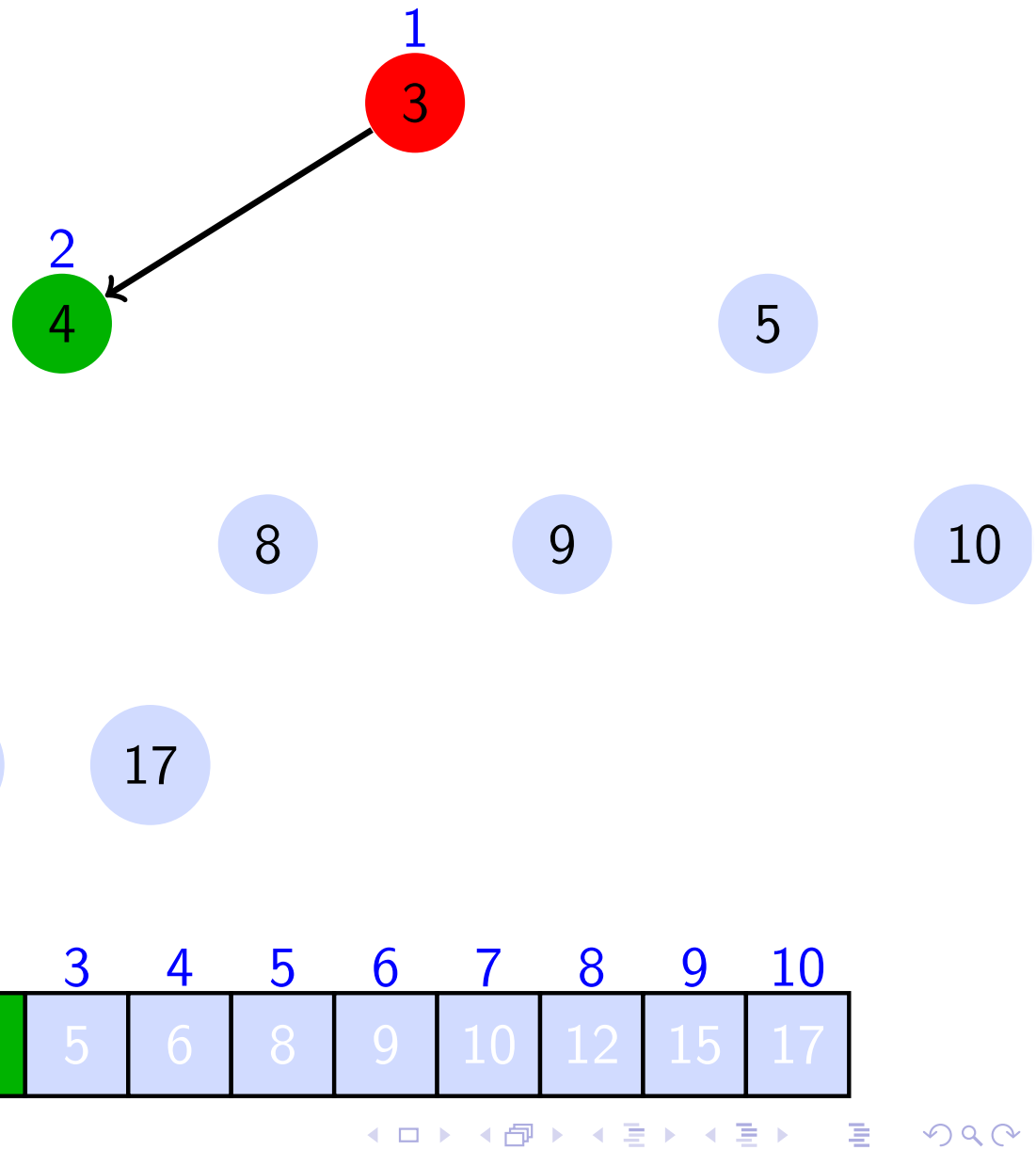

$i = 3$



# HEAP-SORT

HEAP-SORT($A$)

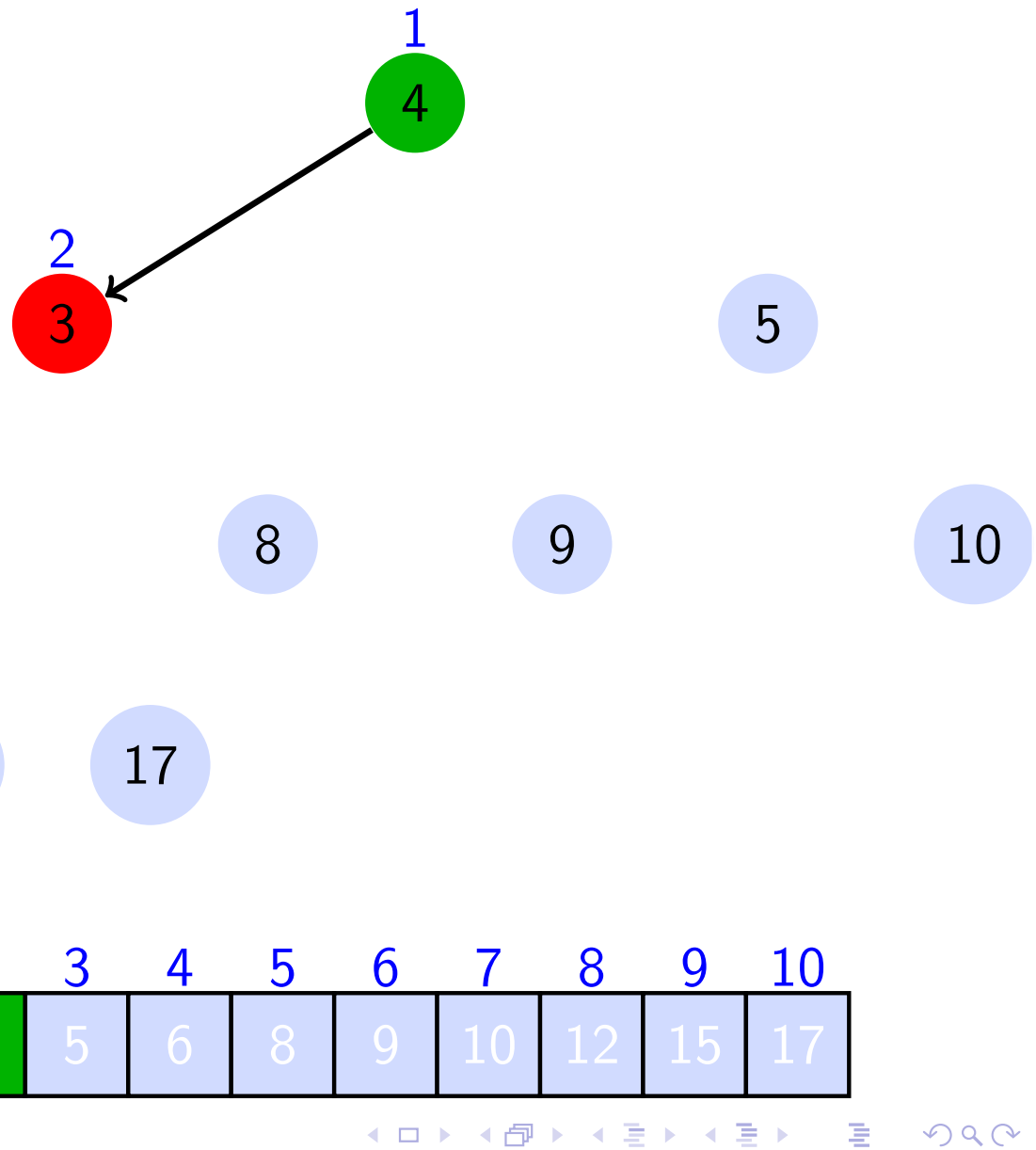
```
1  BUILD-MAX-HEAP(A)
2  for i ← A. length downto 2
3        do exchange A[1] ↔ A[i]
4           heap-size[A] ← heap-size[A] − 1
5           MAX-HEAPIFY(A,1)
```



# HEAP-SORT

HEAP-SORT($A$)

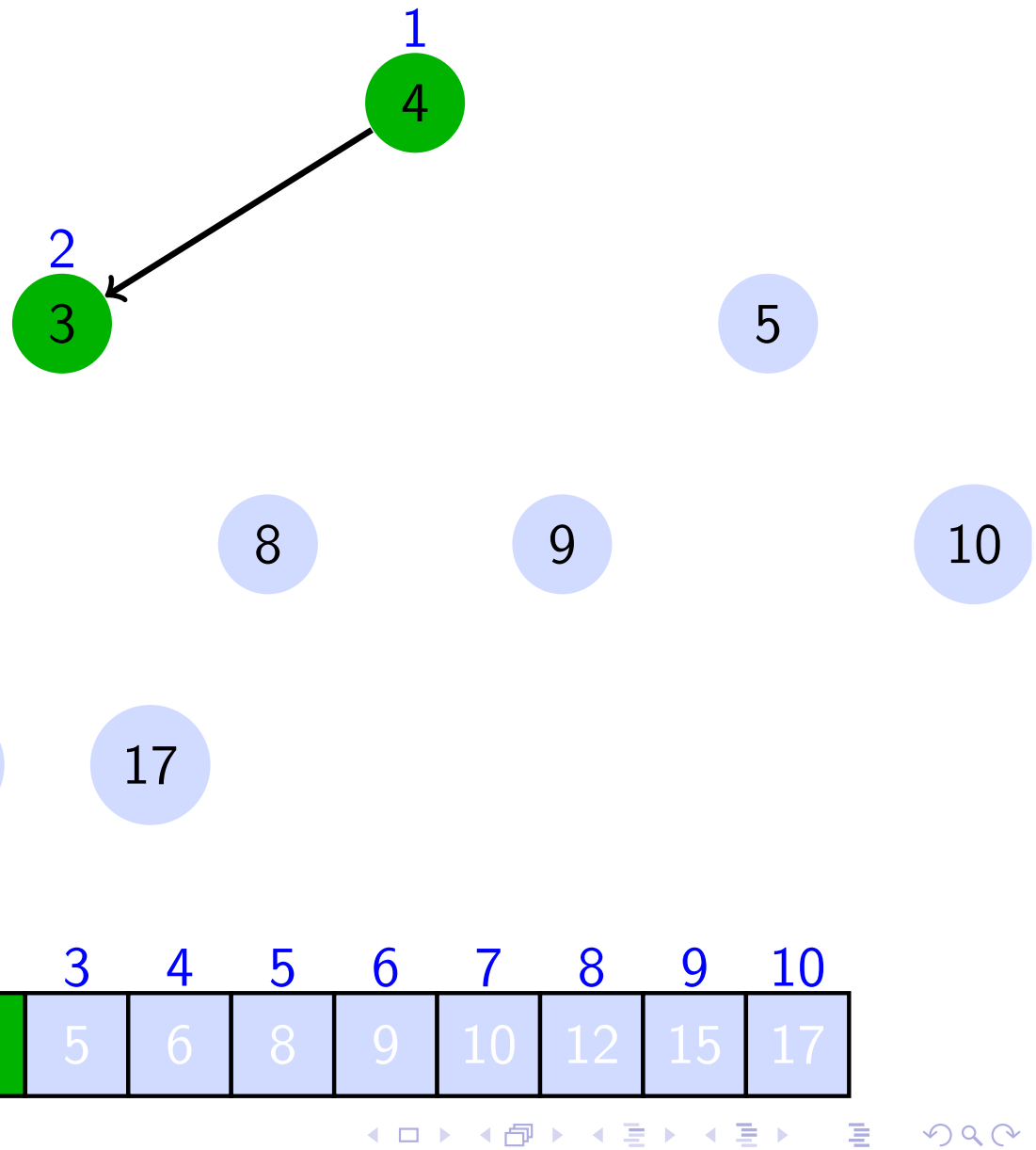
```
1  BUILD-MAX-HEAP(A)
2  for i ← A. length downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A,1)
```



# HEAP-SORT

HEAP-SORT($A$)

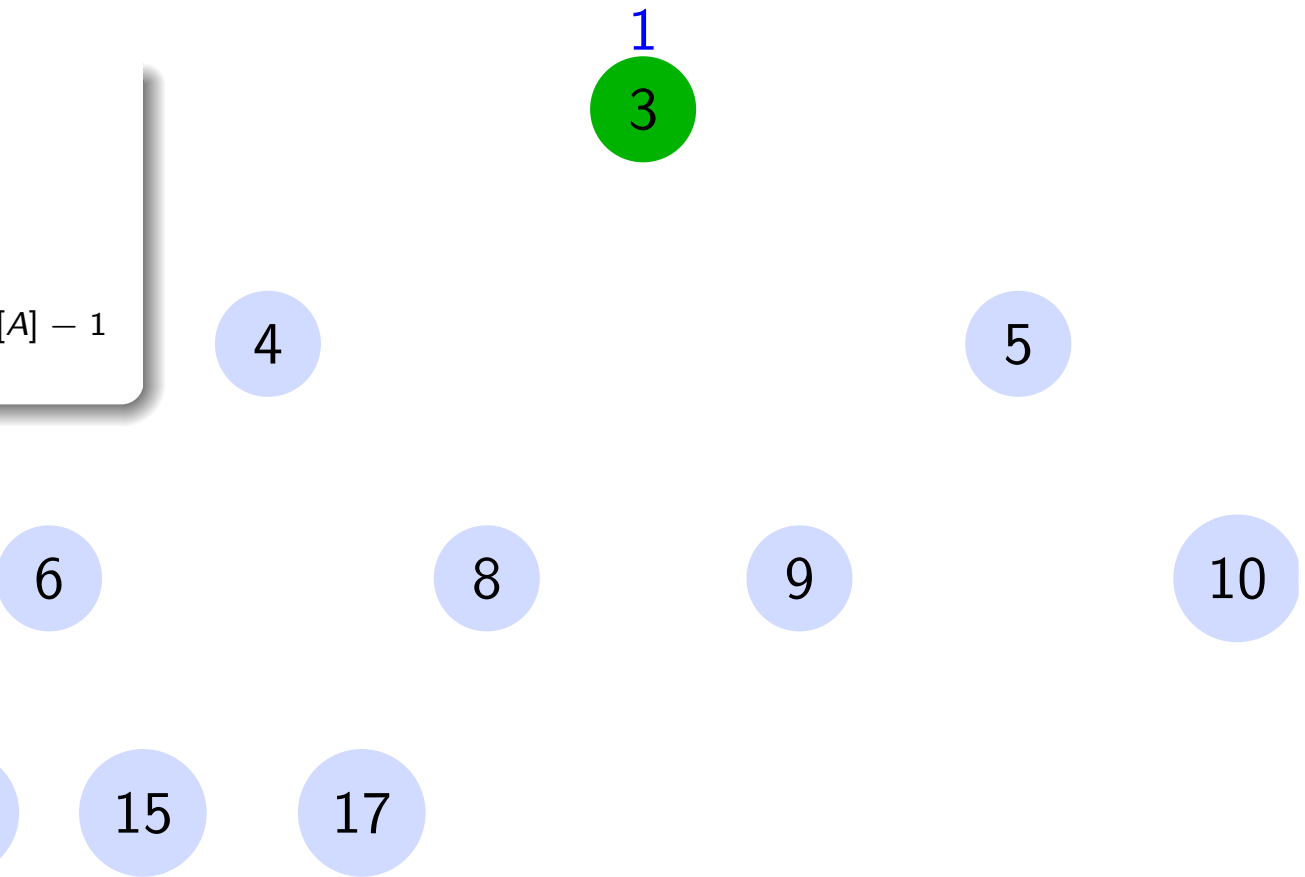
```
1  BUILD-MAX-HEAP(A)
2  for i ← A. length downto 2
3        do exchange A[1] ↔ A[i]
4           heap-size[A] ← heap-size[A] − 1
5           MAX-HEAPIFY(A,1)
```



# HEAP-SORT

HEAP-SORT($A$)

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



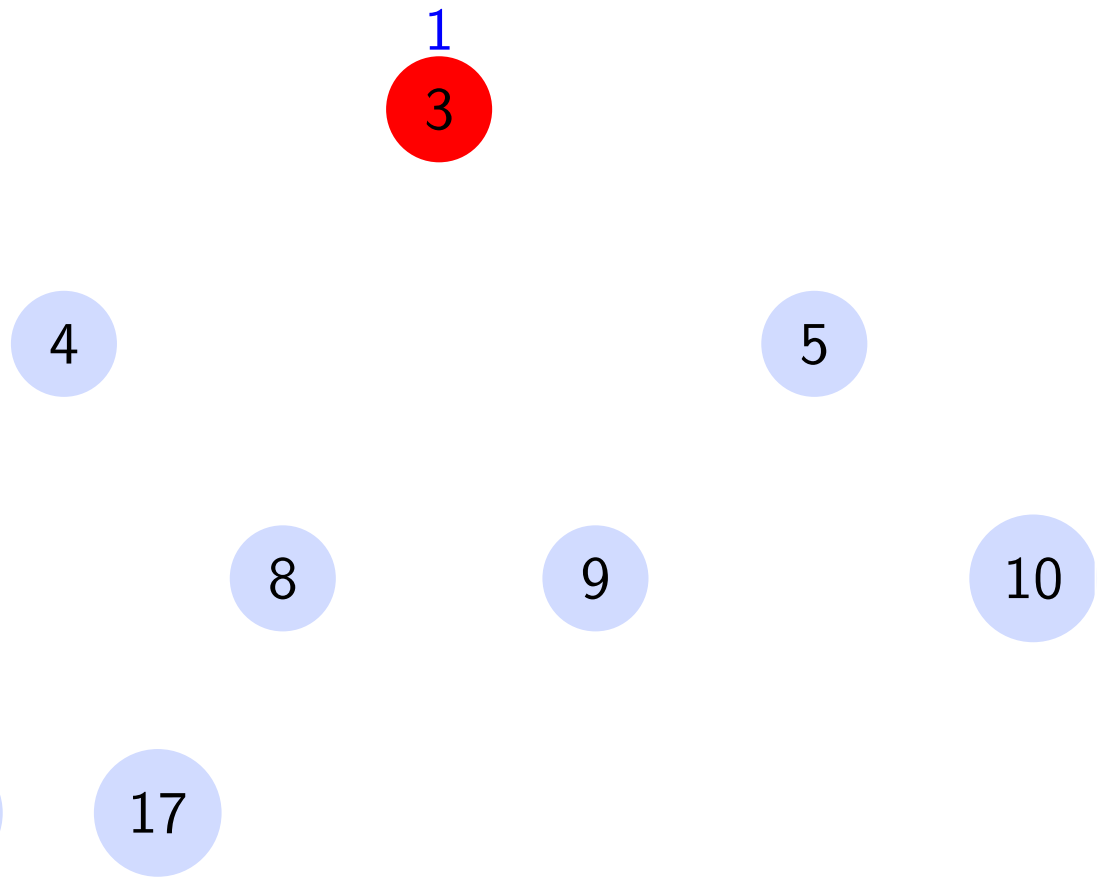

$i = 2$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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



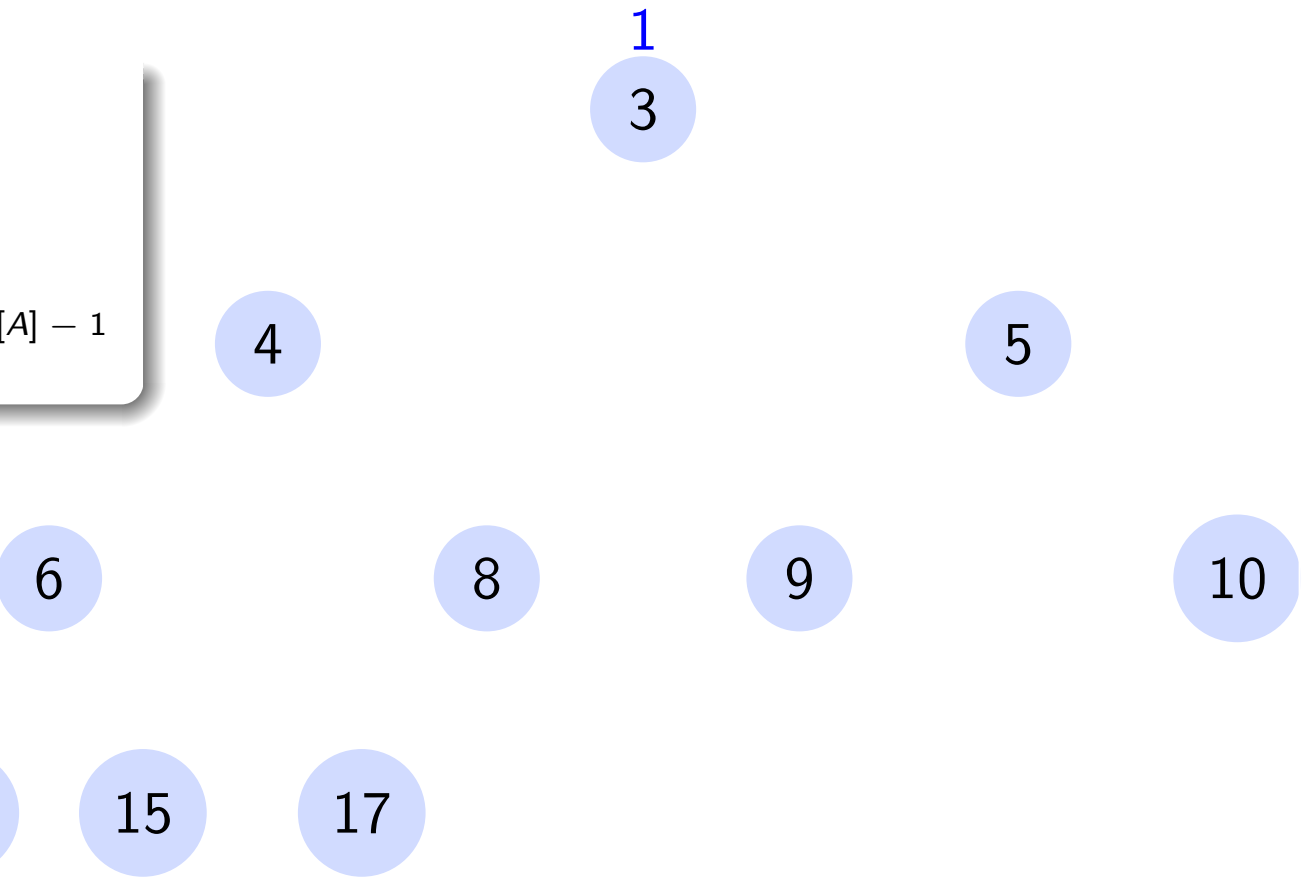

$i = 2$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT

HEAP-SORT($A$)

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




$i = 2$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 8 | 9 | 10 | 12 | 15 | 17 |

# HEAP-SORT: Runtime

- Let $n = A.\,length$
- Cost of BUILD-MAX-HEAP: $O(n)$

HEAP-SORT($A$)

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

- $(n-1)$ calls to MAX-HEAPIFY, each of which costs $O(log_2(n))$
- $T(n) = (n-1)O(log_2(n)) + O(n)$
  $= O(n \cdot log_2(n))$

# Priority Queues

- A priority queue is a data structure which represents a set $S$ of elements.
- Each element has a key (an integer)
- A priority queue should support the following operations:
  - INSERT$(S, x)$: inserts the element $x$ into the set $S$ (i.e., $S \leftarrow S \cup \{x\}$)
  - MAXIMUM$(S)$: returns the element of $S$ with largest key.
  - EXTRACT-MAX$(S)$: returns and removes the element of $S$ with largest key.
  - INCREASE-KEY$(S, i, x)$: finds the element with index $i$, and increases its value to $x$ ($x$ is assumed to be larger than the original value of the element)

# Priority Queues

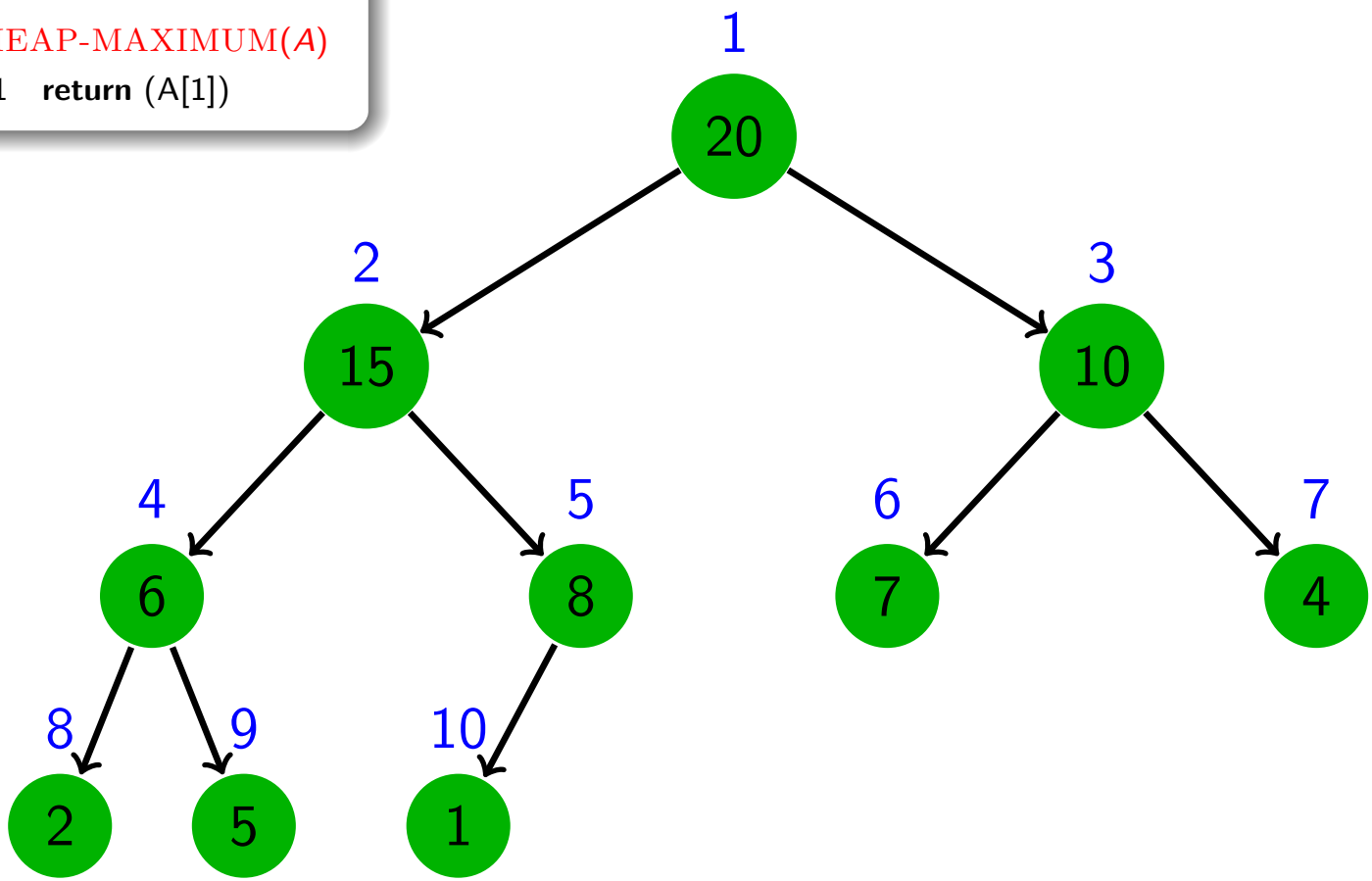
- A priority queue is a data structure which represents a set $S$ of elements.
- Each element has a key (an integer)
- A priority queue should support the following operations:
  - INSERT$(S, x)$: inserts the element $x$ into the set $S$ (i.e., $S \leftarrow S \cup \{x\}$)
  - MAXIMUM$(S)$: returns the element of $S$ with largest key.
  - EXTRACT-MAX$(S)$: returns and removes the element of $S$ with largest key.
  - INCREASE-KEY$(S, i, x)$: finds the element with index $i$, and increases its value to $x$ ($x$ is assumed to be larger than the original value of the element)

Max-Heaps efficiently implement priority queues.

# HEAP-MAXIMUM

HEAP-MAXIMUM($A$): returns the largest element of $A$: It is the root

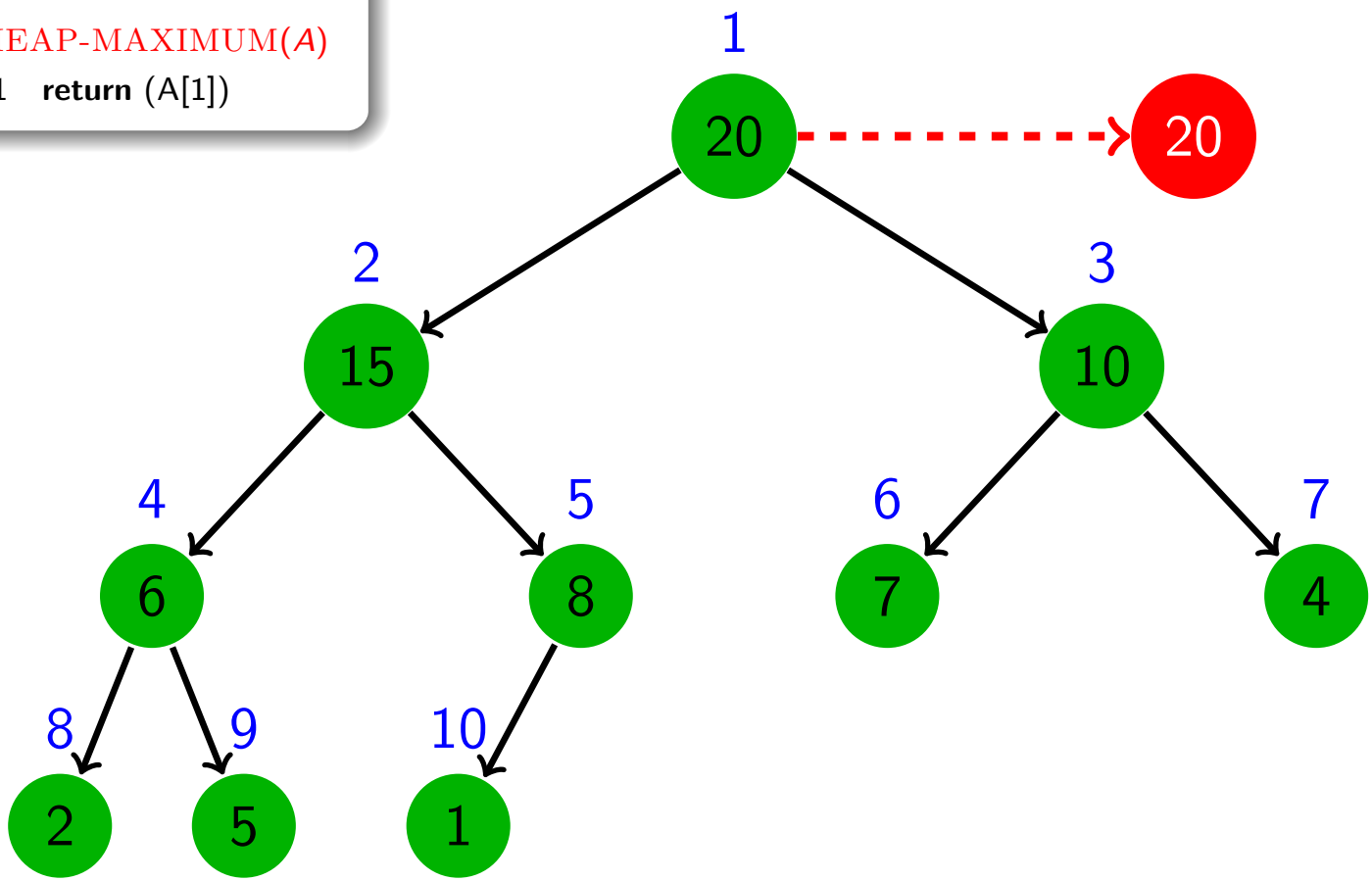
```
HEAP-MAXIMUM(A)
1  return (A[1])
```



# HEAP-MAXIMUM

HEAP-MAXIMUM($A$): returns the largest element of $A$: It is the root

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

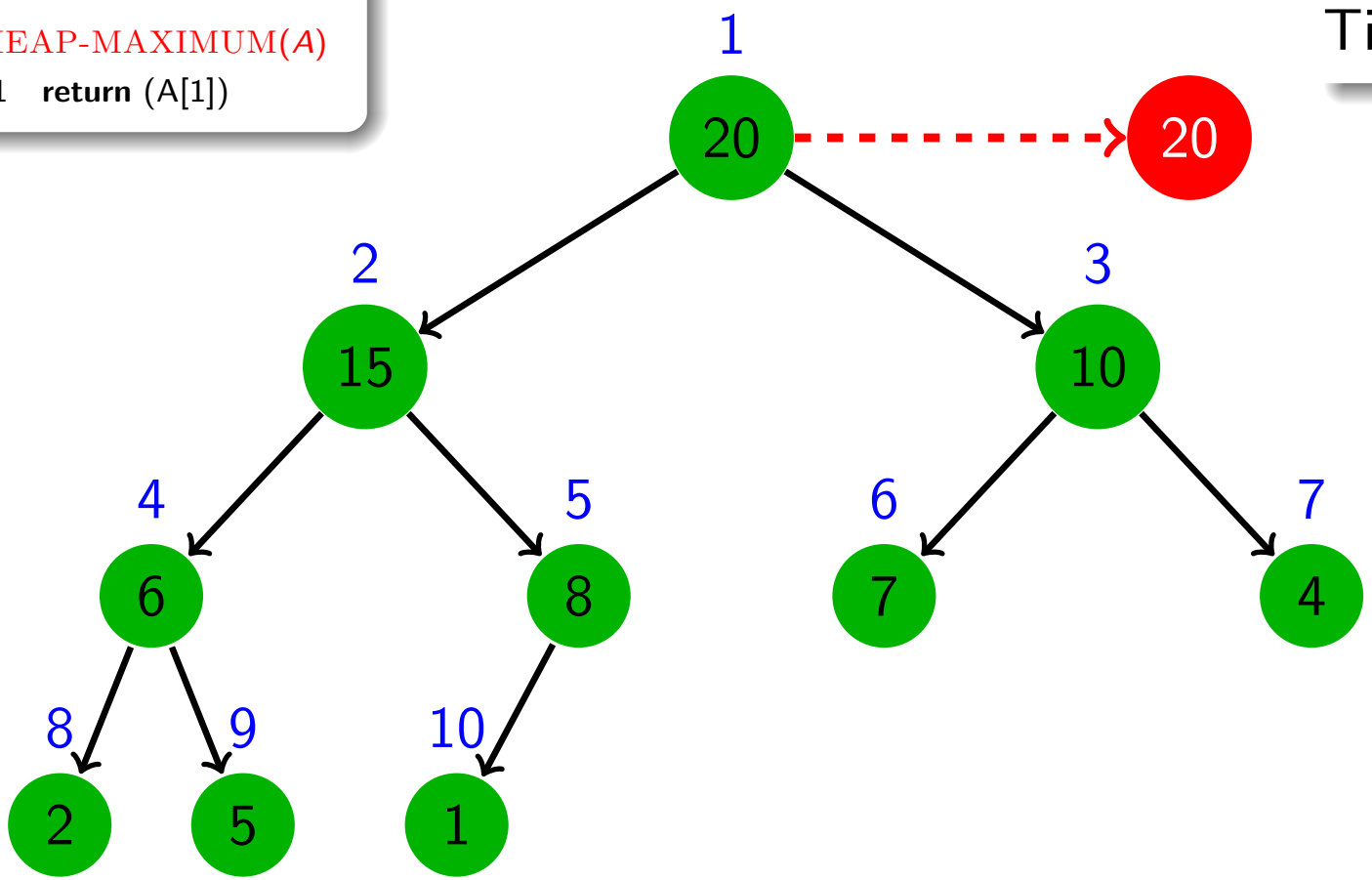


# HEAP-MAXIMUM

HEAP-MAXIMUM($A$): returns the largest element of $A$: It is the root

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

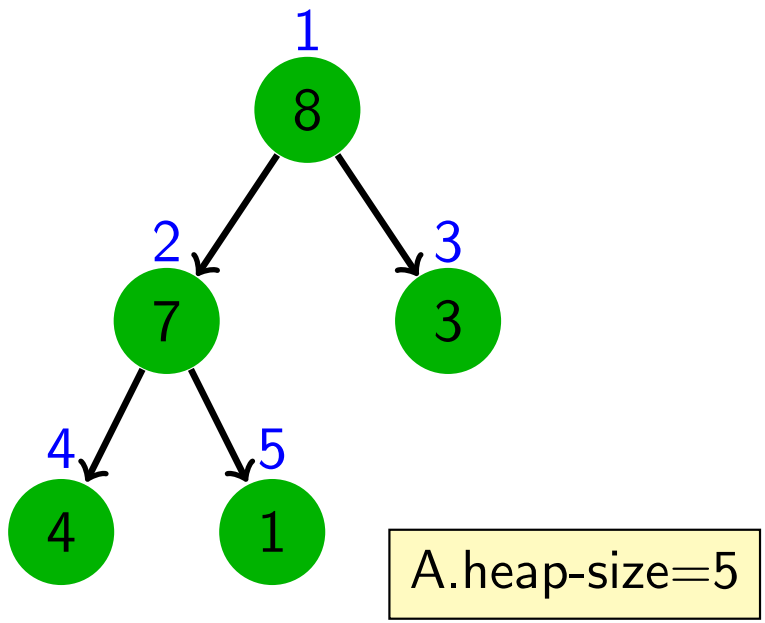
Time: $\Theta(1)$

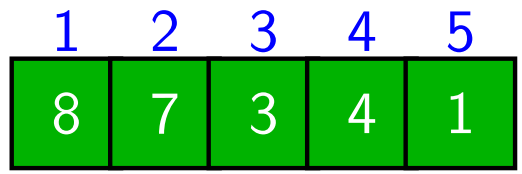


# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX



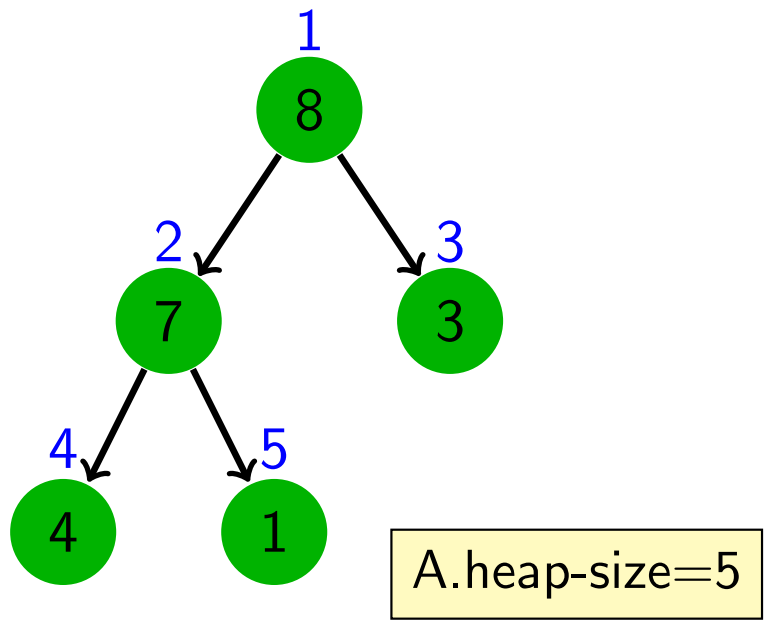



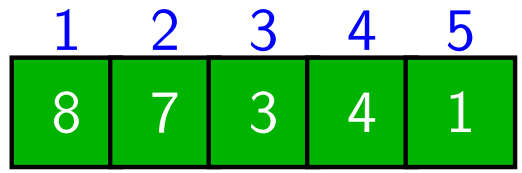
# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX

- Make sure that the max-heap is not empty.



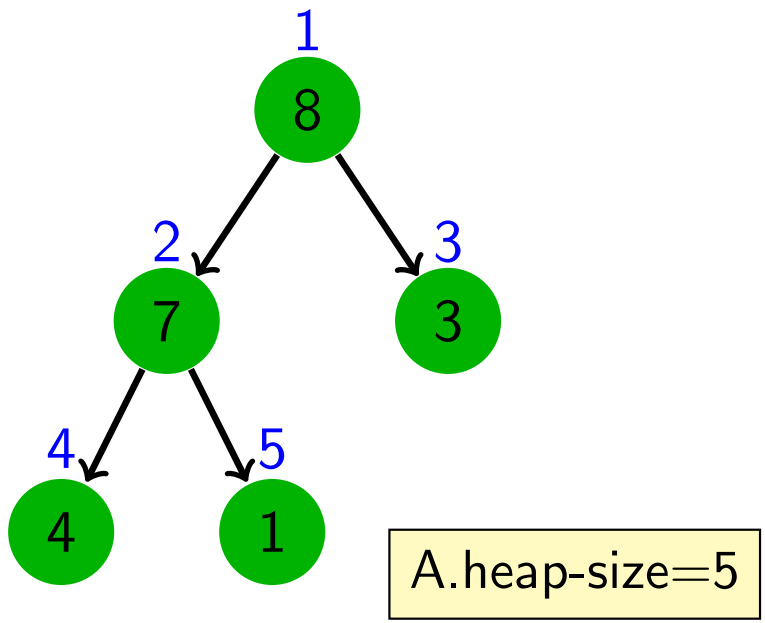



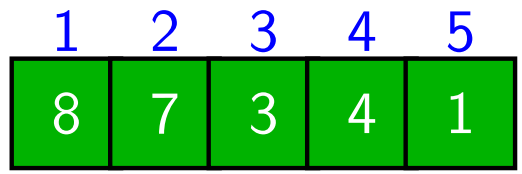
# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX

- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root)



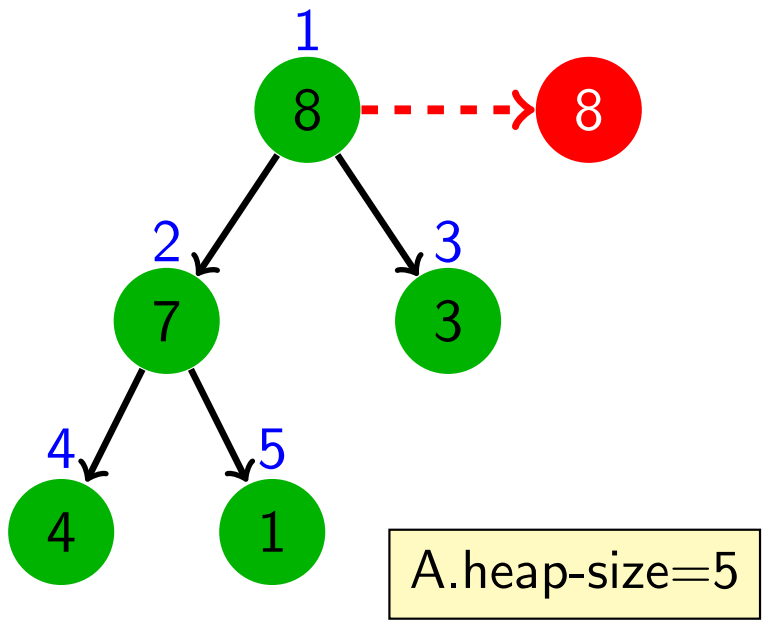



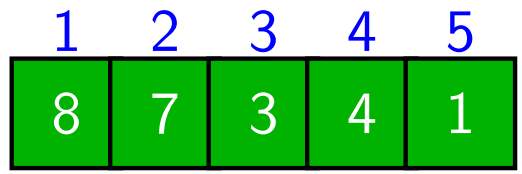
# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX

- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root)



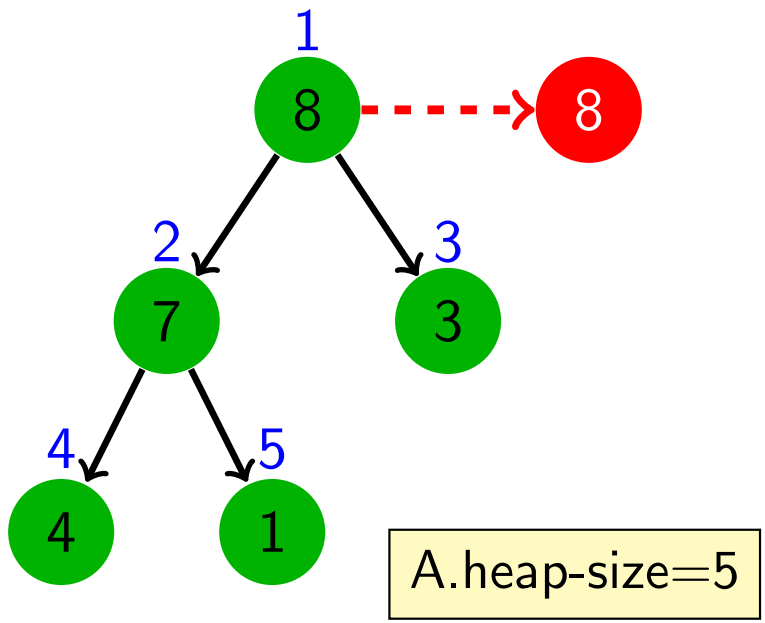



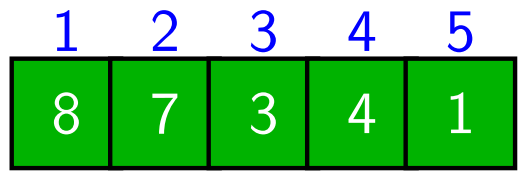
# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX

- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root)
- Make the last node of the heap the new root



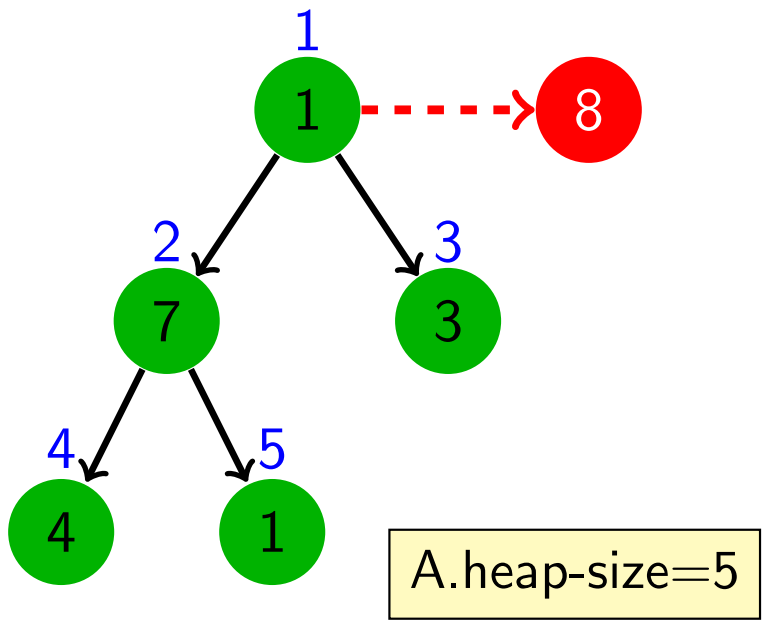



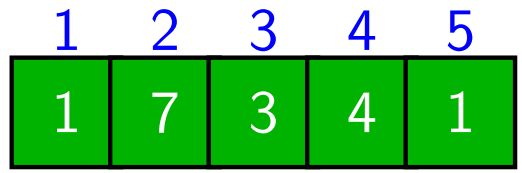
# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX

- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root)
- Make the last node of the heap the new root



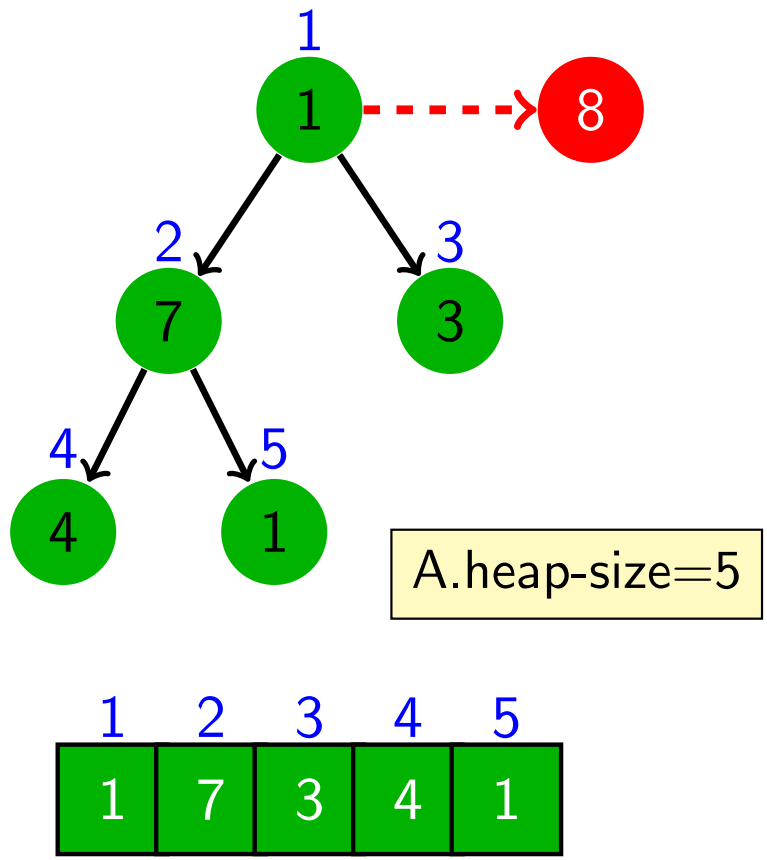



# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX

- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root)
- Make the last node of the heap the new root
- Discard the last node of the heap

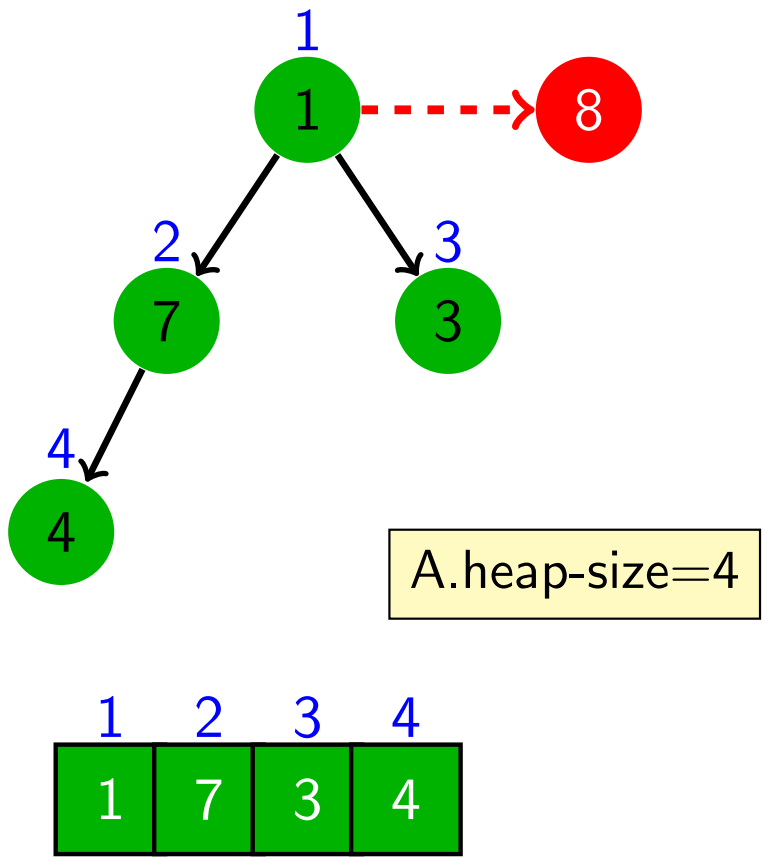


# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX

- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root)
- Make the last node of the heap the new root
- Discard the last node of the heap

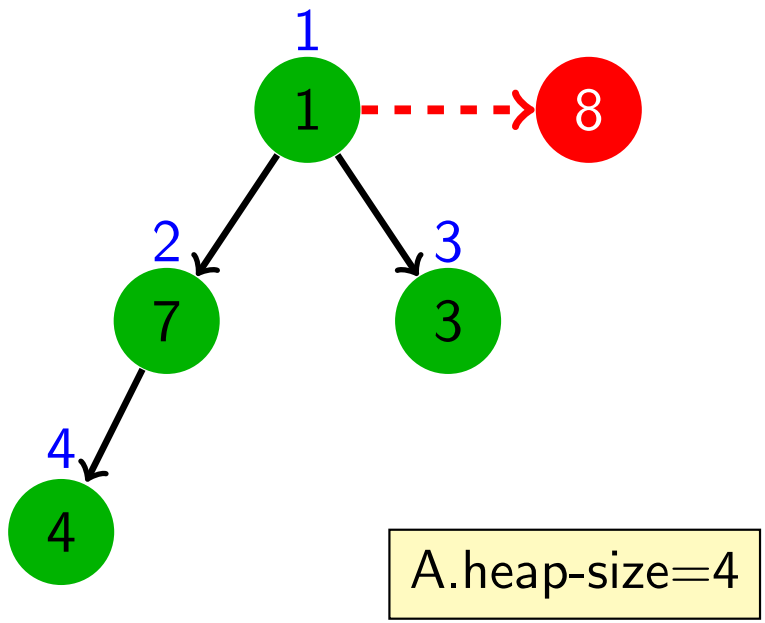


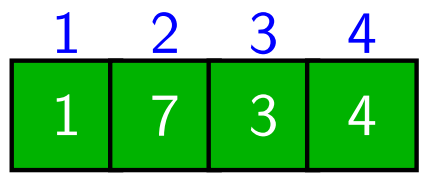
# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX

- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root)
- Make the last node of the heap the new root
- Discard the last node of the heap
- Restore the max-heap property



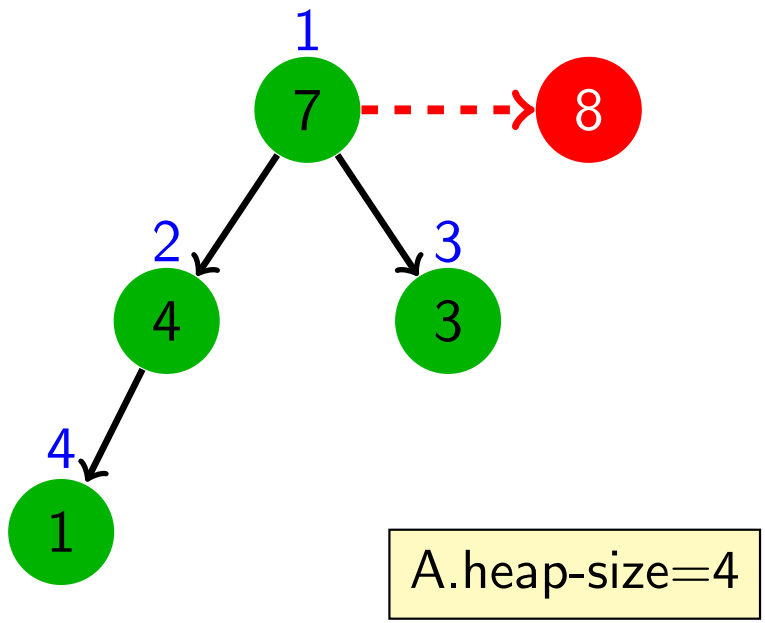



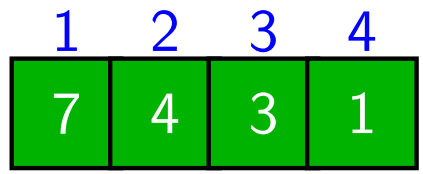
# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

## EXTRACT-MAX

- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root)
- Make the last node of the heap the new root
- Discard the last node of the heap
- Restore the max-heap property



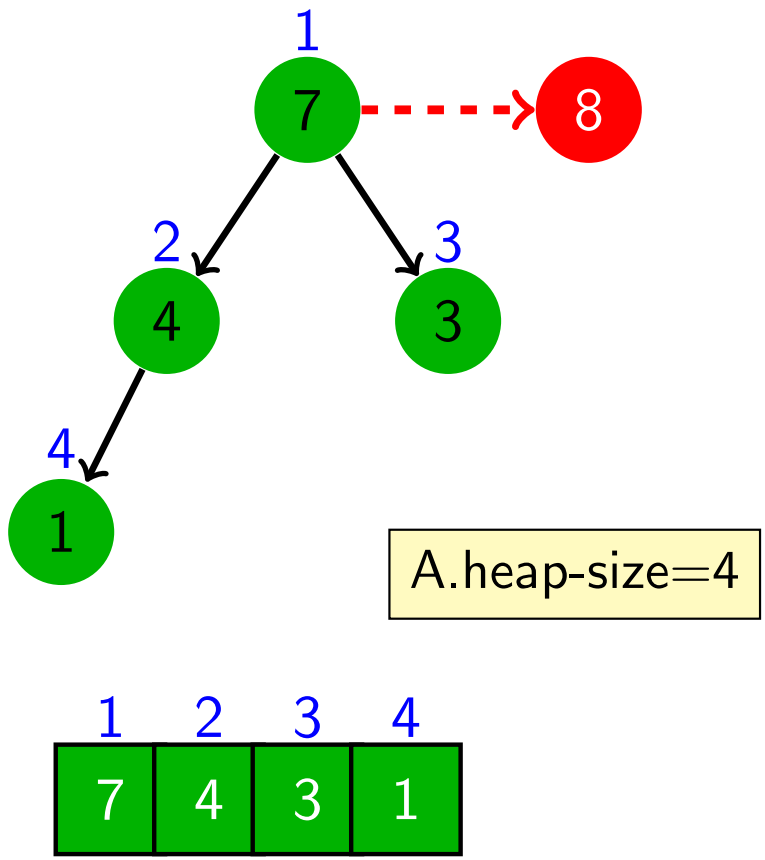



# EXTRACT-MAX: Principle

Problem: returns and removes the largest element of the array

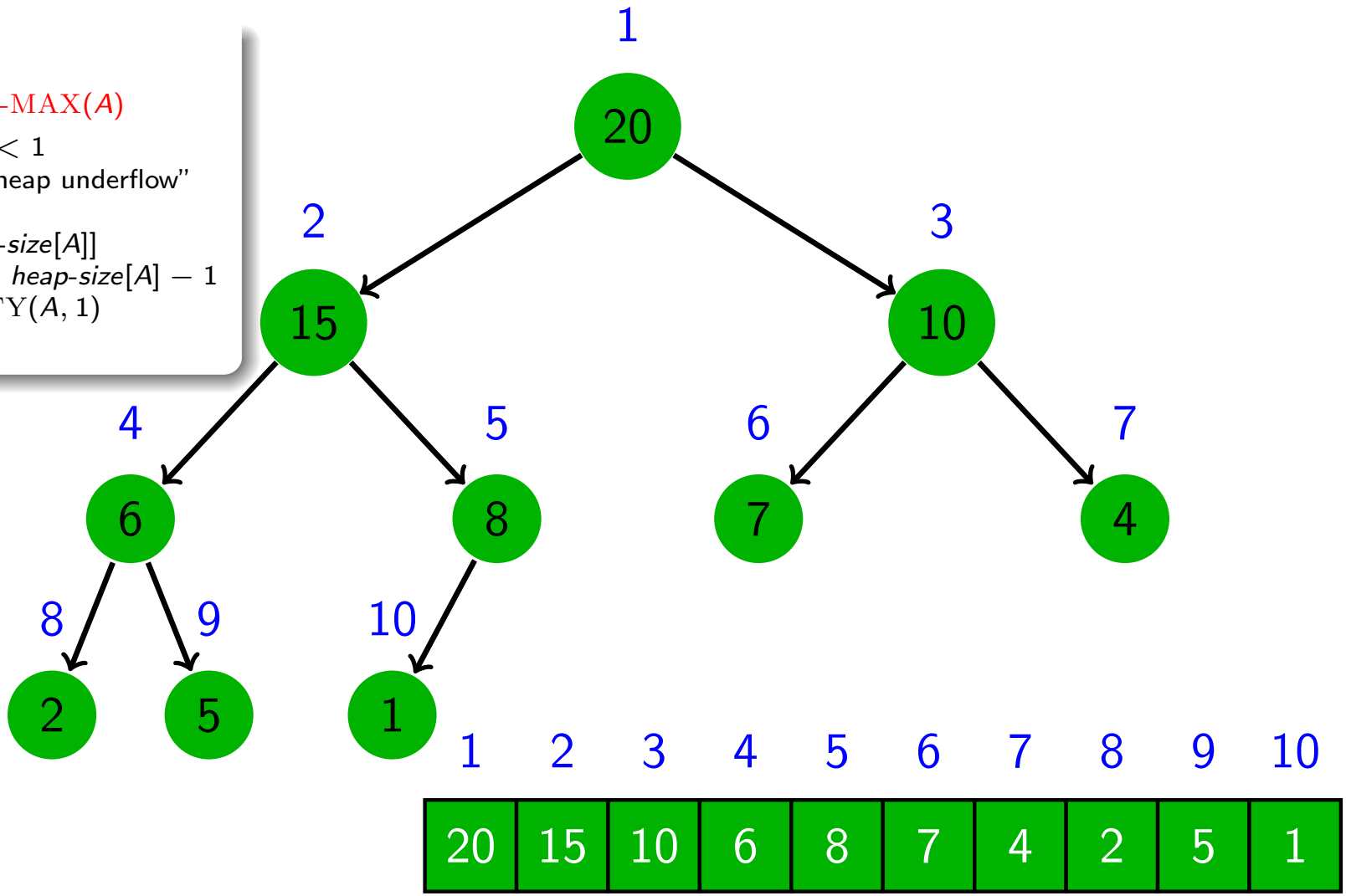
## EXTRACT-MAX

- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root)
- Make the last node of the heap the new root
- Discard the last node of the heap
- Restore the max-heap property
- Return the copy of the maximum element



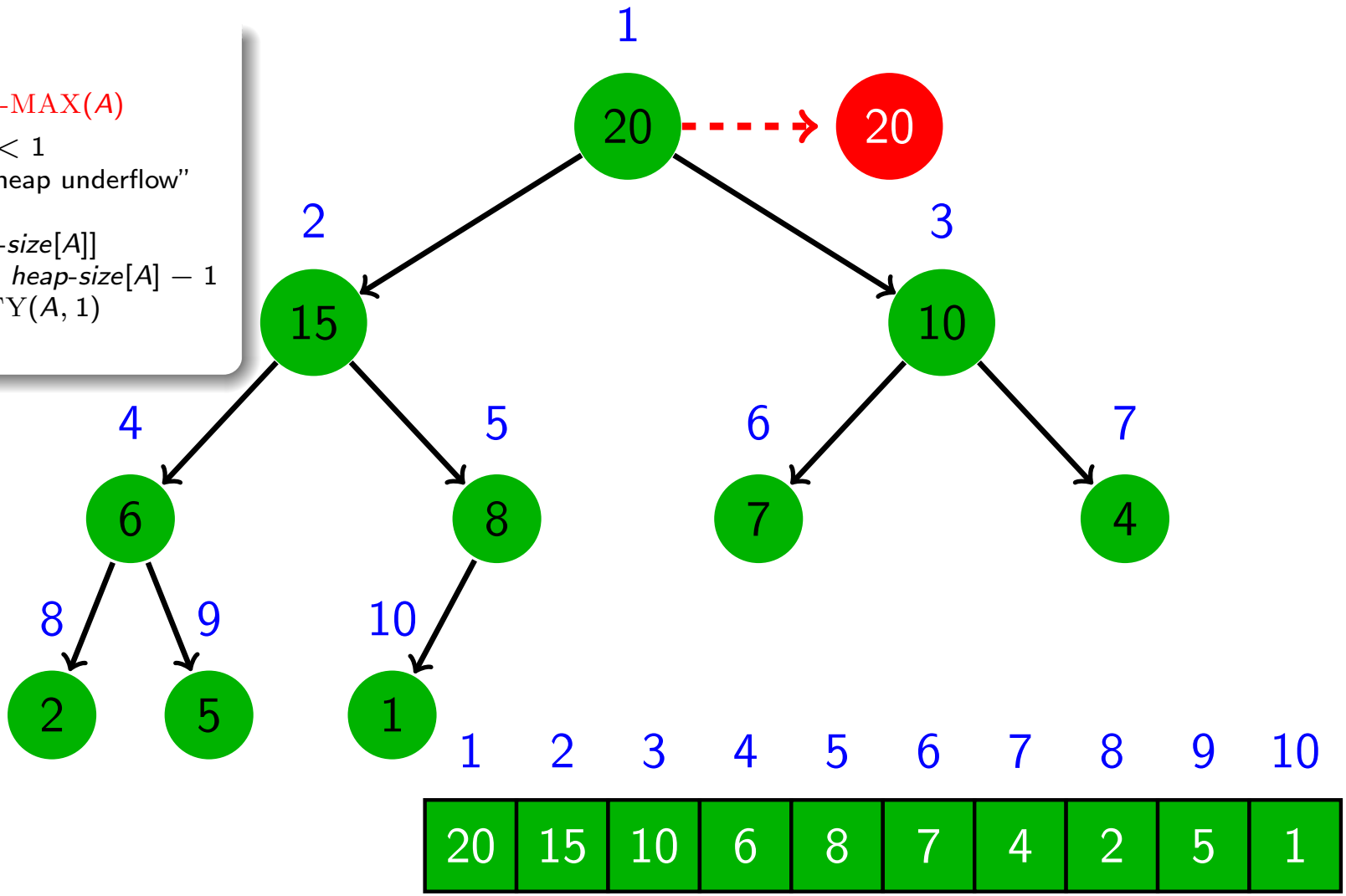
# HEAP-EXTRACT-MAX

```
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
```



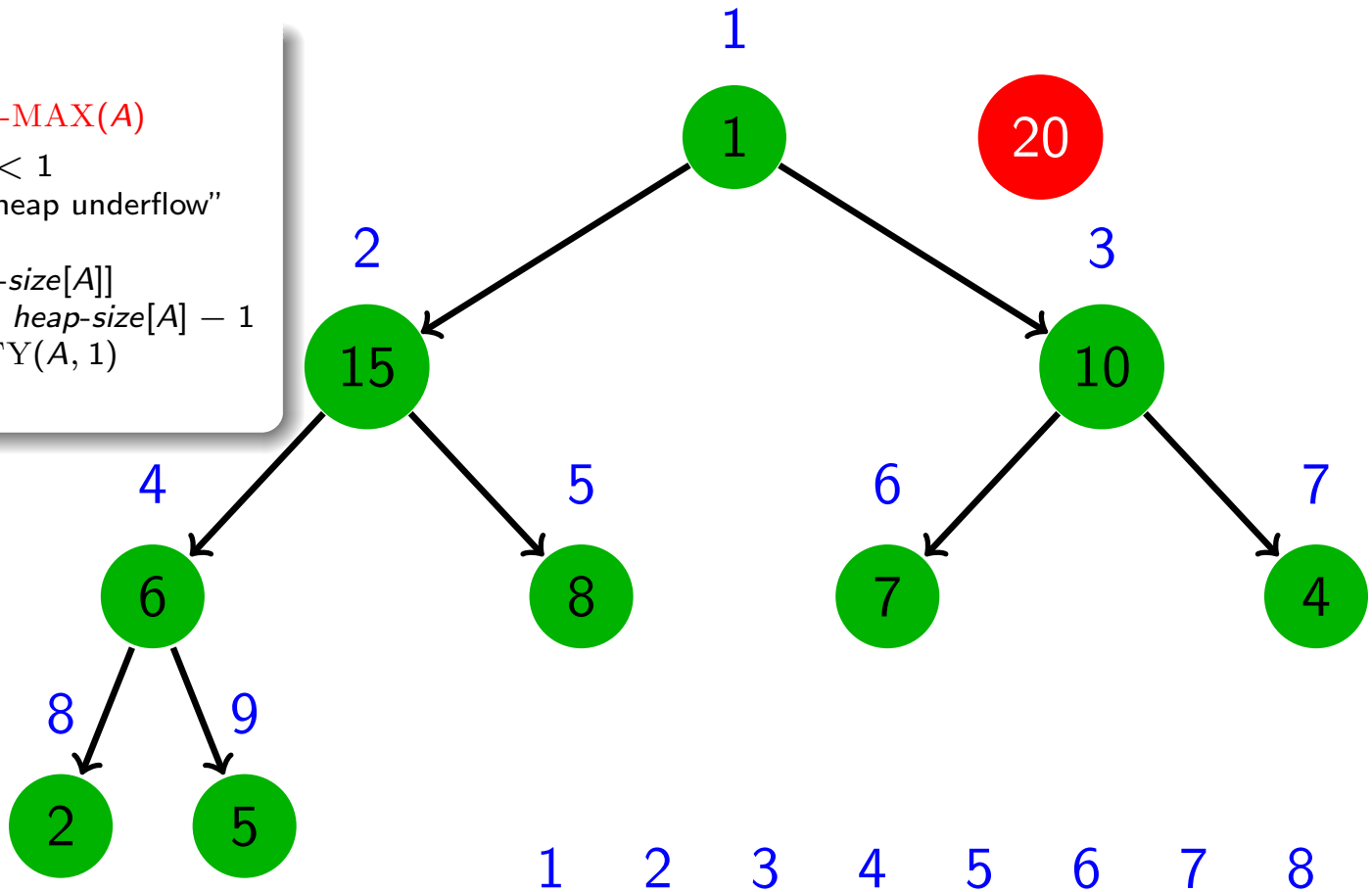
# HEAP-EXTRACT-MAX

```
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
```



# HEAP-EXTRACT-MAX

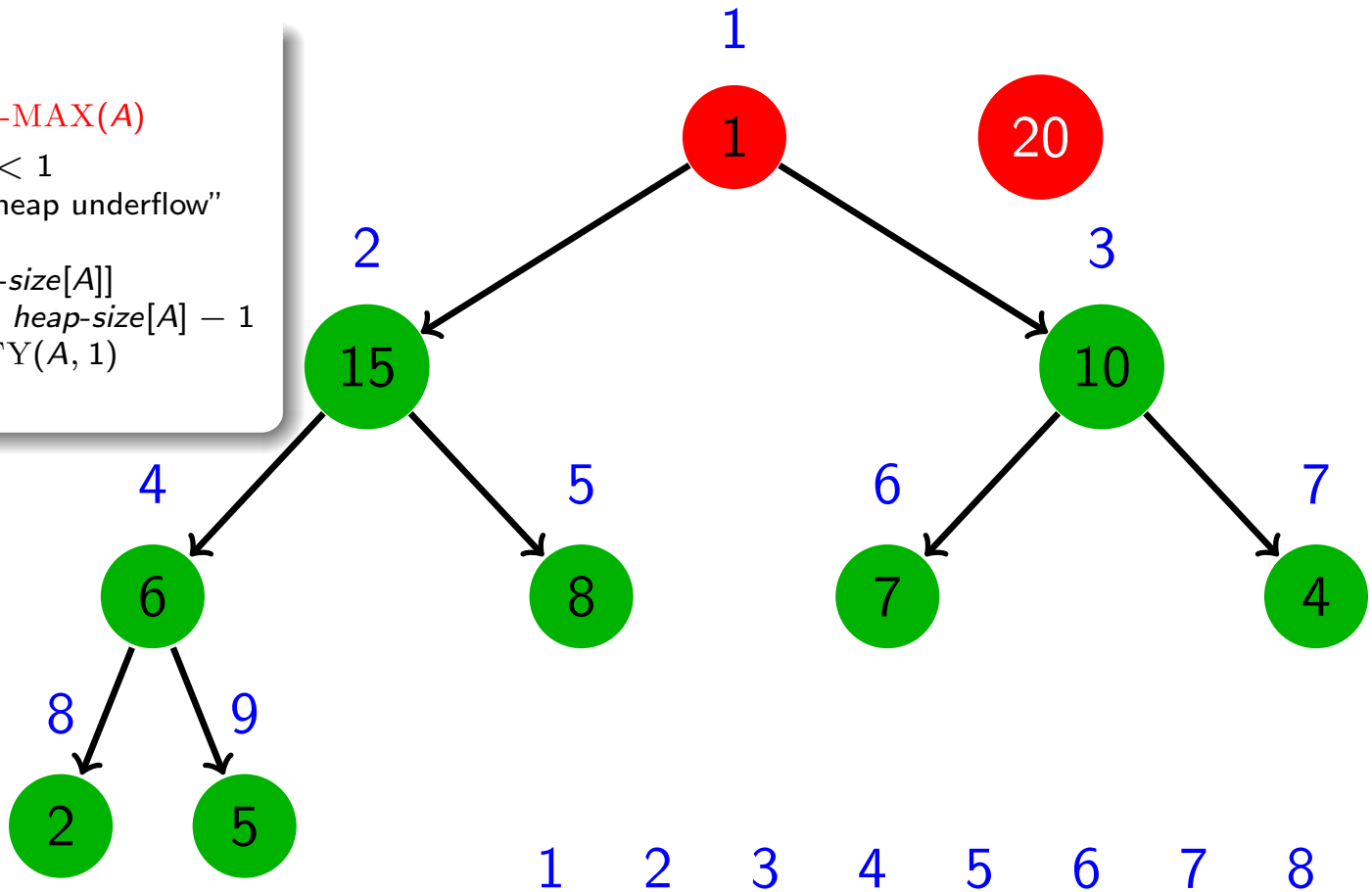
```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2      then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```




| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | 15 | 10 | 6 | 8 | 7 | 4 | 2 | 5 |

# HEAP-EXTRACT-MAX

```
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
```

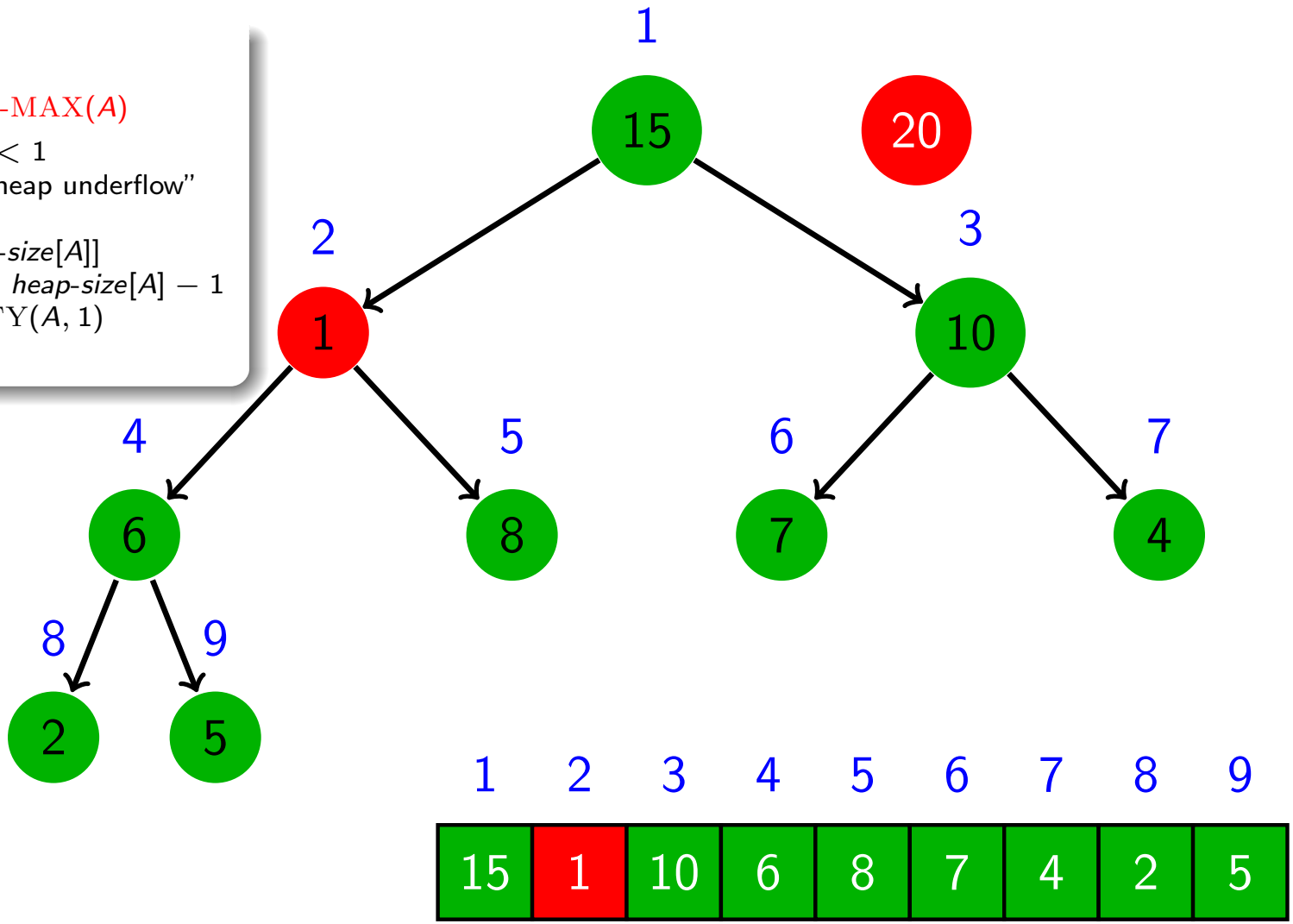



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 1 | 15 | 10 | 6 | 8 | 7 | 4 | 2 | 5 |

# HEAP-EXTRACT-MAX

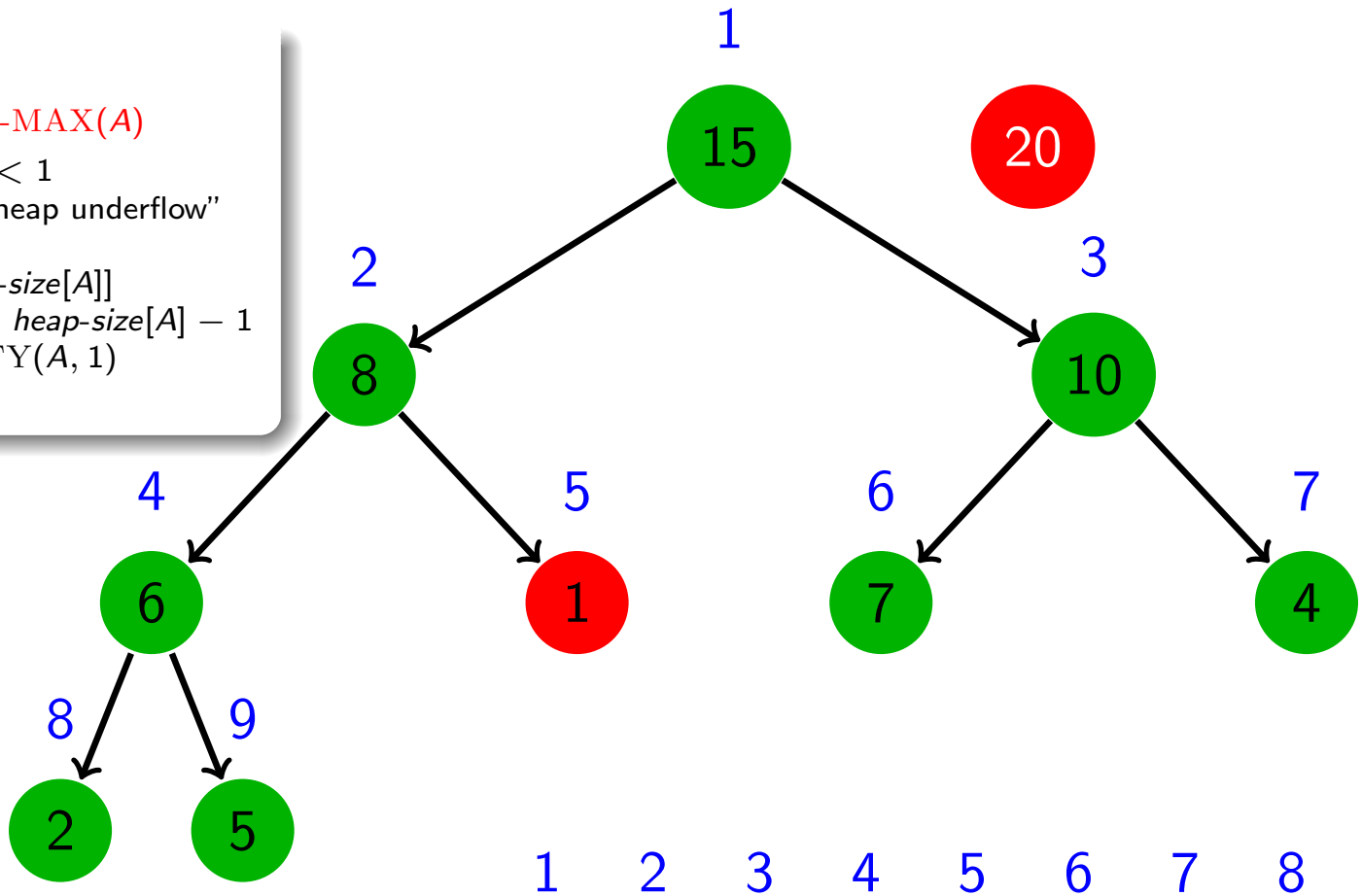
HEAP-EXTRACT-MAX($A$)

```
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
```



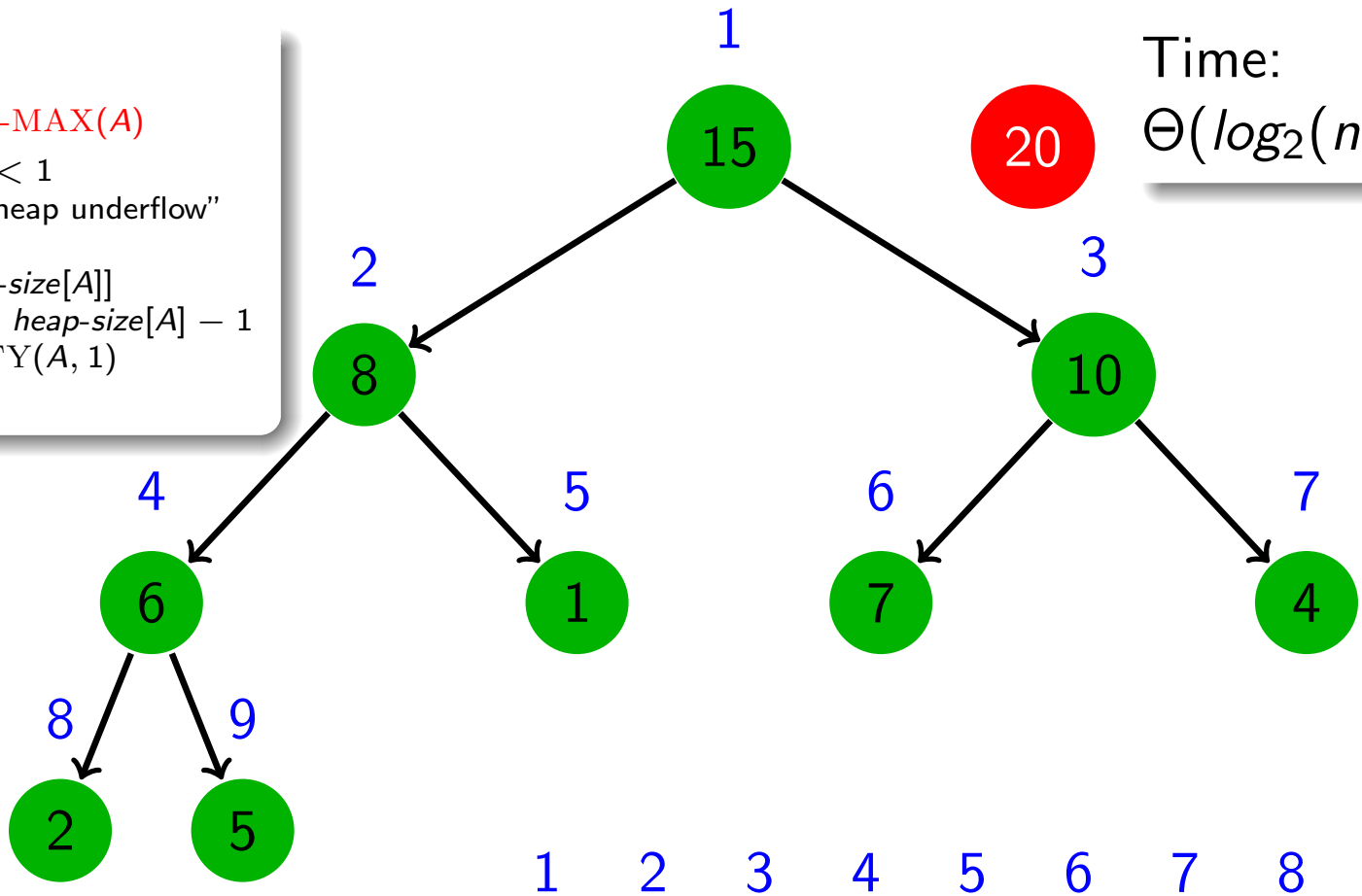
# HEAP-EXTRACT-MAX

```
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
```



# HEAP-EXTRACT-MAX

```
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
```



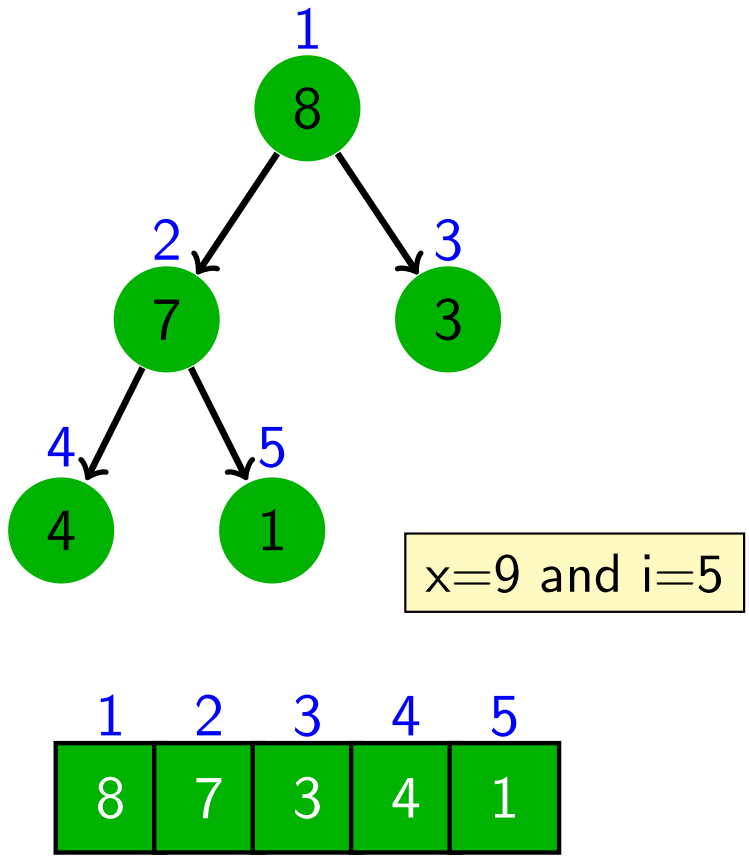

Time:
$\Theta(log_2(n))$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 15 | 8 | 10 | 6 | 8 | 7 | 4 | 2 | 5 |

# HEAP-INCREASE-KEY: Principle

Problem: Find the element with index $i$, and increase its value to $x$ ($x$ is assumed to be larger than the original value of the element)

## HEAP-INCREASE-KEY

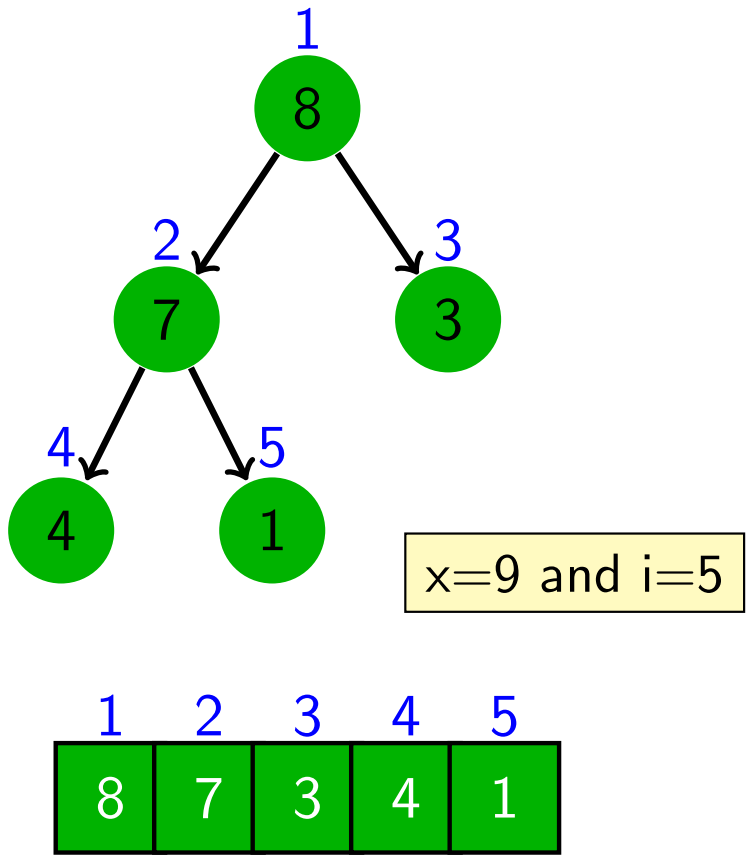


# HEAP-INCREASE-KEY: Principle

Problem: Find the element with index $i$, and increase its value to $x$ ($x$ is assumed to be larger than the original value of the element)

## HEAP-INCREASE-KEY

- Make sure that $x$ is larger than the original value at position $i$

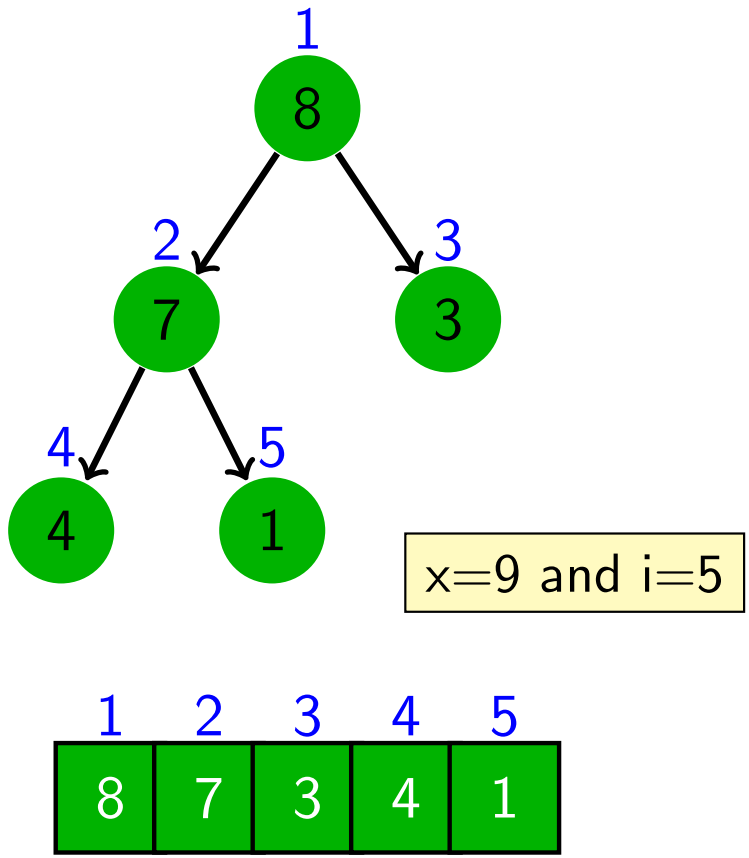


# HEAP-INCREASE-KEY: Principle

Problem: Find the element with index $i$, and increase its value to $x$ ($x$ is assumed to be larger than the original value of the element)

## HEAP-INCREASE-KEY

- Make sure that $x$ is larger than the original value at position $i$
- Increase the value of the element with index $i$ to $x$

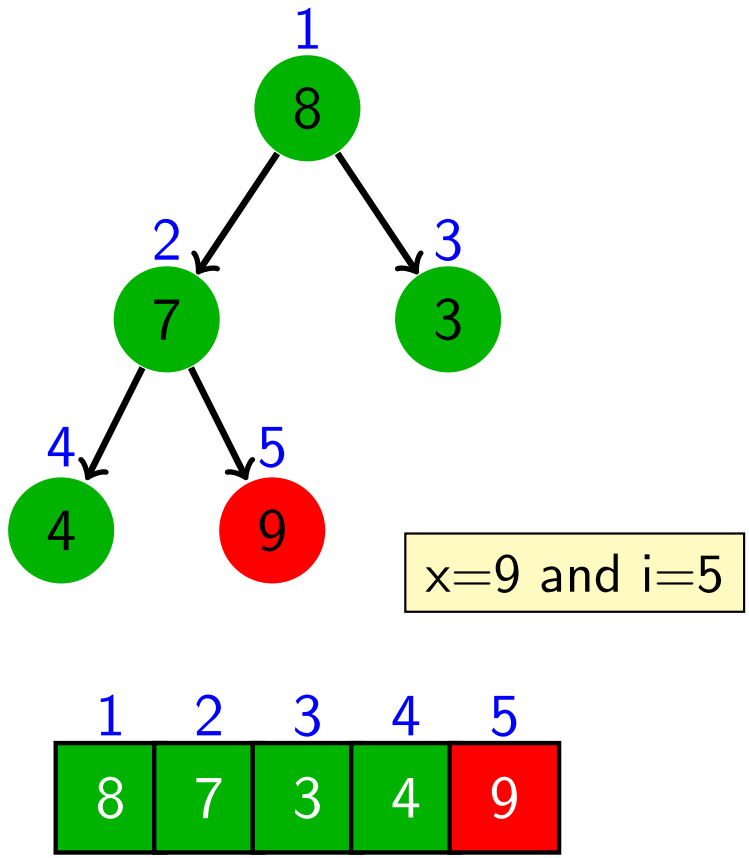


# HEAP-INCREASE-KEY: Principle

Problem: Find the element with index $i$, and increase its value to $x$ ($x$ is assumed to be larger than the original value of the element)

## HEAP-INCREASE-KEY

- Make sure that $x$ is larger than the original value at position $i$
- Increase the value of the element with index $i$ to $x$

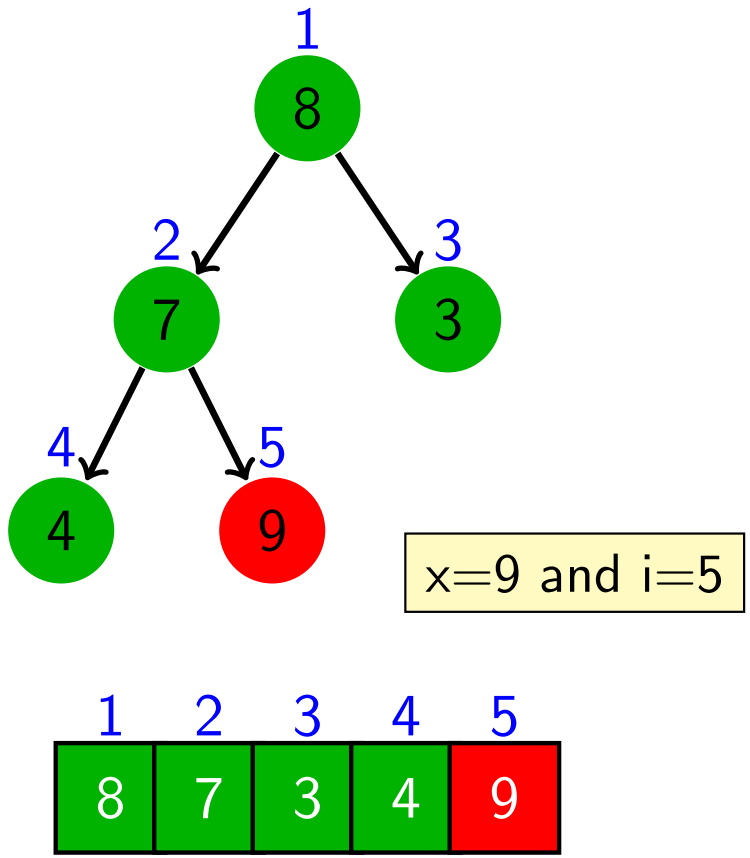


# HEAP-INCREASE-KEY: Principle

Problem: Find the element with index $i$, and increase its value to $x$ ($x$ is assumed to be larger than the original value of the element)

## HEAP-INCREASE-KEY

- Make sure that $x$ is larger than the original value at position $i$
- Increase the value of the element with index $i$ to $x$
- Traverse the tree upward comparing $x$ to its parent and swapping keys if necessary, until the max-heap property is restored

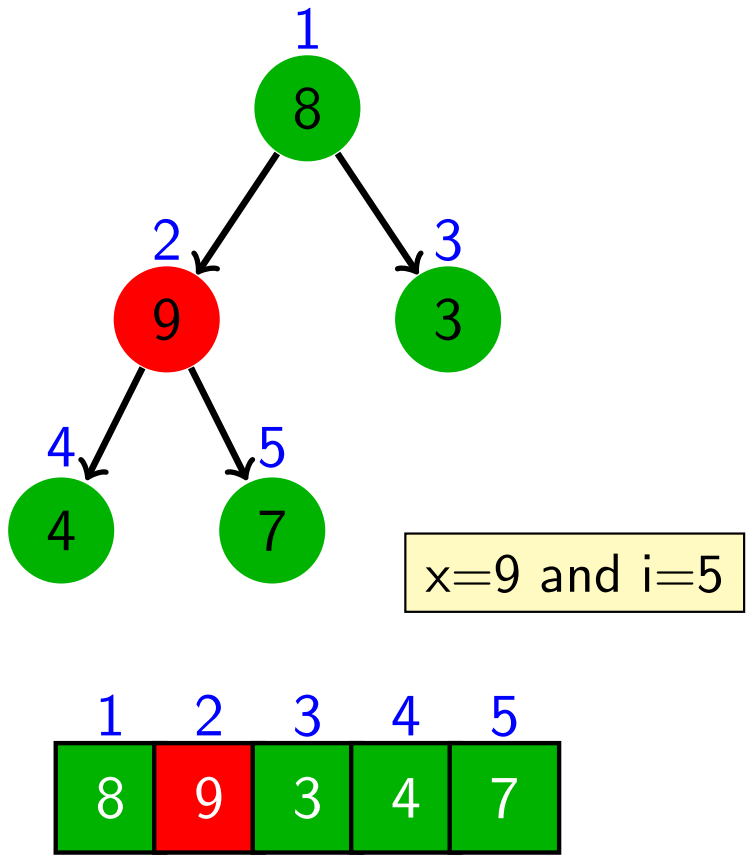


# HEAP-INCREASE-KEY: Principle

Problem: Find the element with index $i$, and increase its value to $x$ ($x$ is assumed to be larger than the original value of the element)

## HEAP-INCREASE-KEY

- Make sure that $x$ is larger than the original value at position $i$
- Increase the value of the element with index $i$ to $x$
- Traverse the tree upward comparing $x$ to its parent and swapping keys if necessary, until the max-heap property is restored

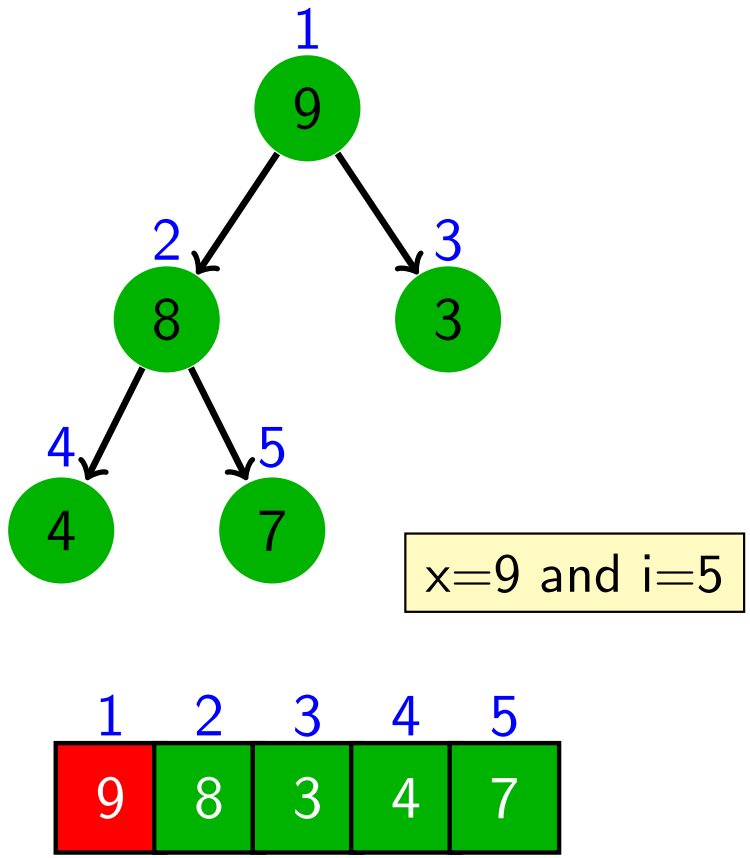


# HEAP-INCREASE-KEY: Principle

Problem: Find the element with index $i$, and increase its value to $x$ ($x$ is assumed to be larger than the original value of the element)

## HEAP-INCREASE-KEY

- Make sure that $x$ is larger than the original value at position $i$
- Increase the value of the element with index $i$ to $x$
- Traverse the tree upward comparing $x$ to its parent and swapping keys if necessary, until the max-heap property is restored

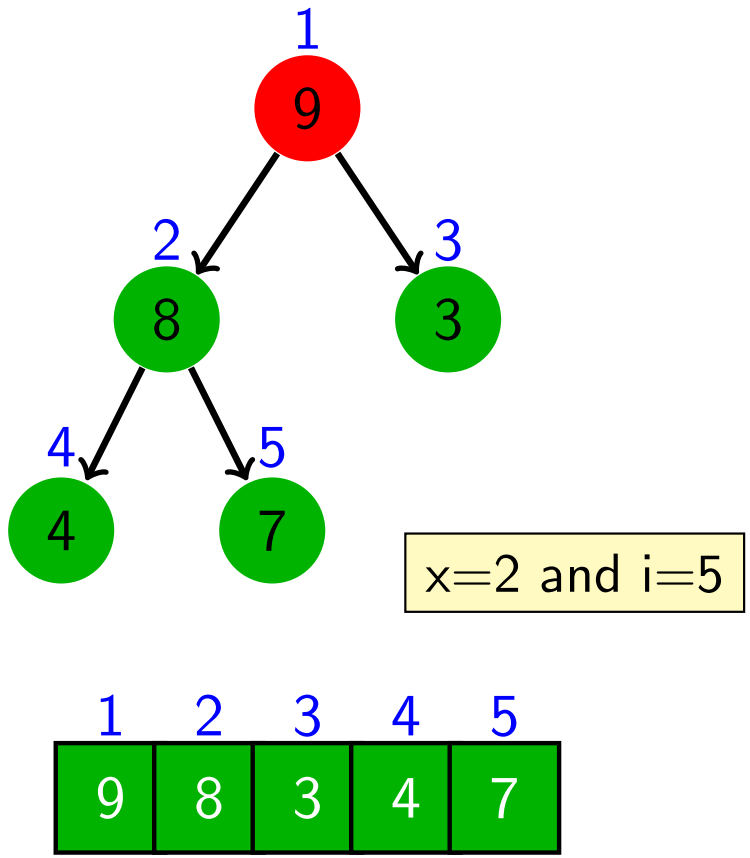


# HEAP-INCREASE-KEY: Principle

Problem: Find the element with index $i$, and increase its value to $x$ ($x$ is assumed to be larger than the original value of the element)

## HEAP-INCREASE-KEY

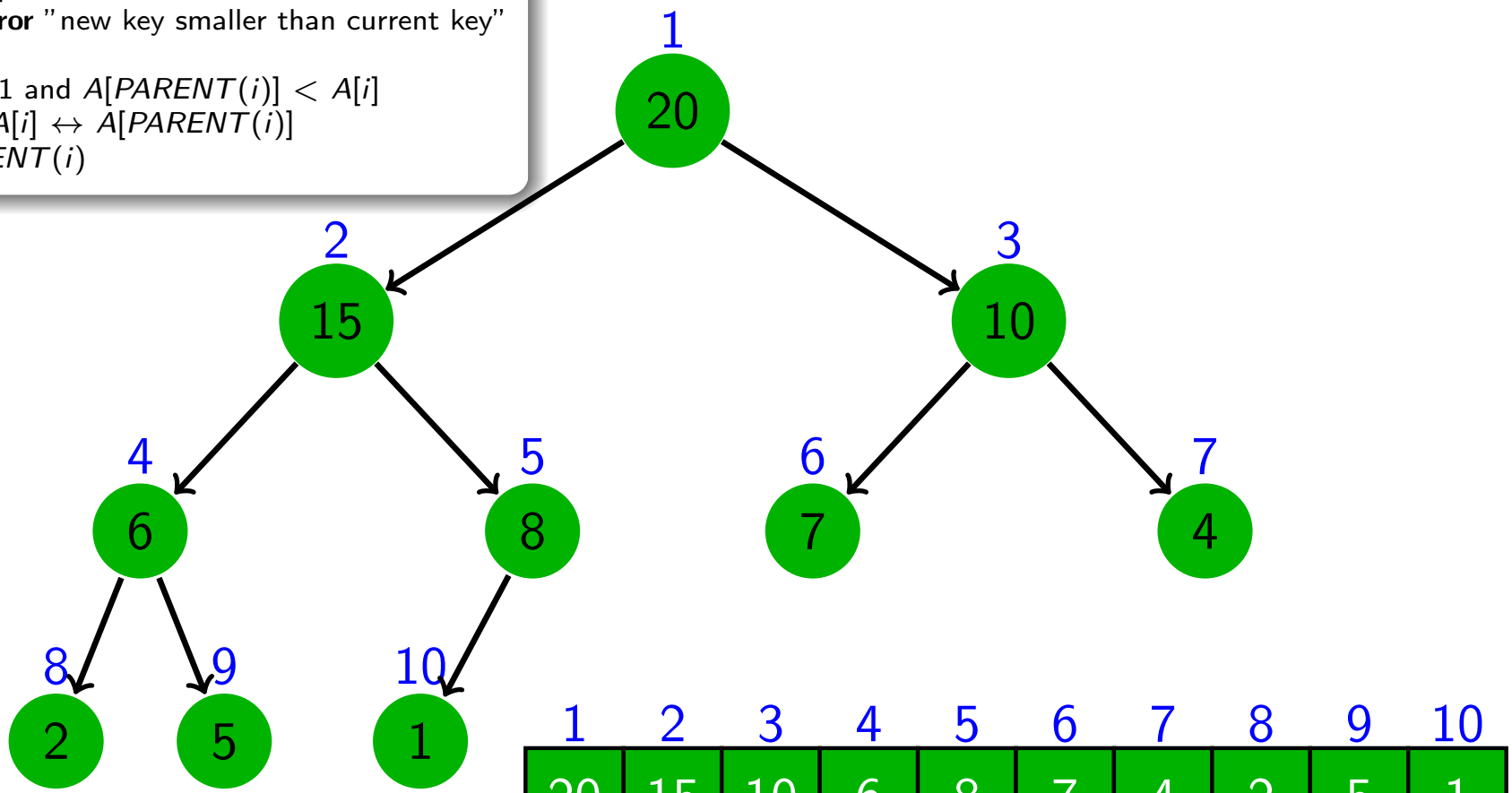
- Make sure that $x$ is larger than the original value at position $i$
- Increase the value of the element with index $i$ to $x$
- Traverse the tree upward comparing $x$ to its parent and swapping keys if necessary, until the max-heap property is restored



# HEAP-INCREASE-KEY

HEAP-INCREASE-KEY$(A, i, x)$

```
1  if x < A[i]
2      then error "new key smaller than current key"
3  A[i] ← x
4  while i > 1 and A[PARENT(i)] < A[i]
5  exchange A[i] ↔ A[PARENT(i)]
6  i ← PARENT(i)
```

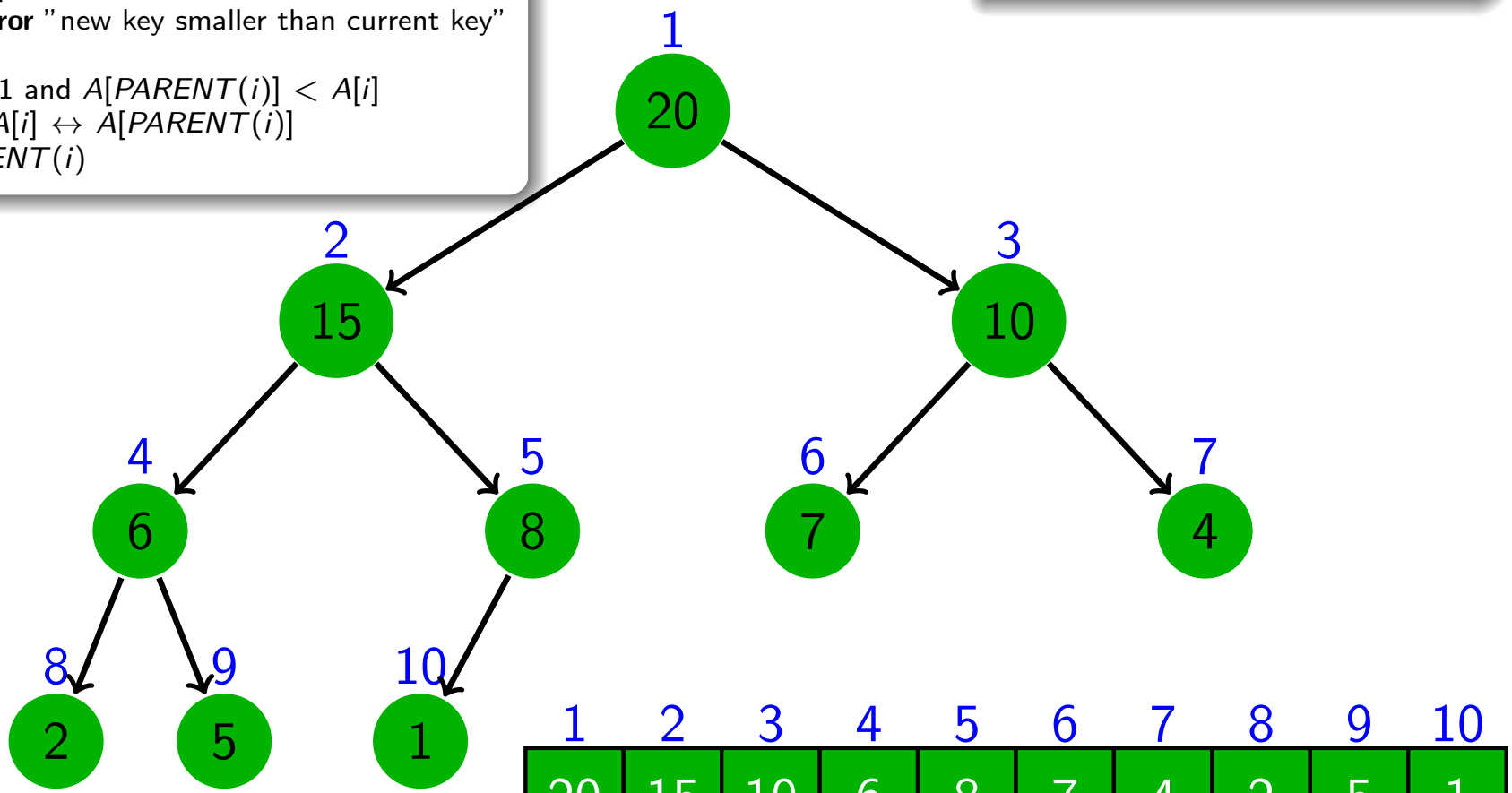


# HEAP-INCREASE-KEY

HEAP-INCREASE-KEY$(A, i, x)$

```
1  if x < A[i]
2      then error "new key smaller than current key"
3  A[i] ← x
4  while i > 1 and A[PARENT(i)] < A[i]
5  exchange A[i] ↔ A[PARENT(i)]
6  i ← PARENT(i)
```

HEAP-INCREASE-KEY$(A, 9, 18)$

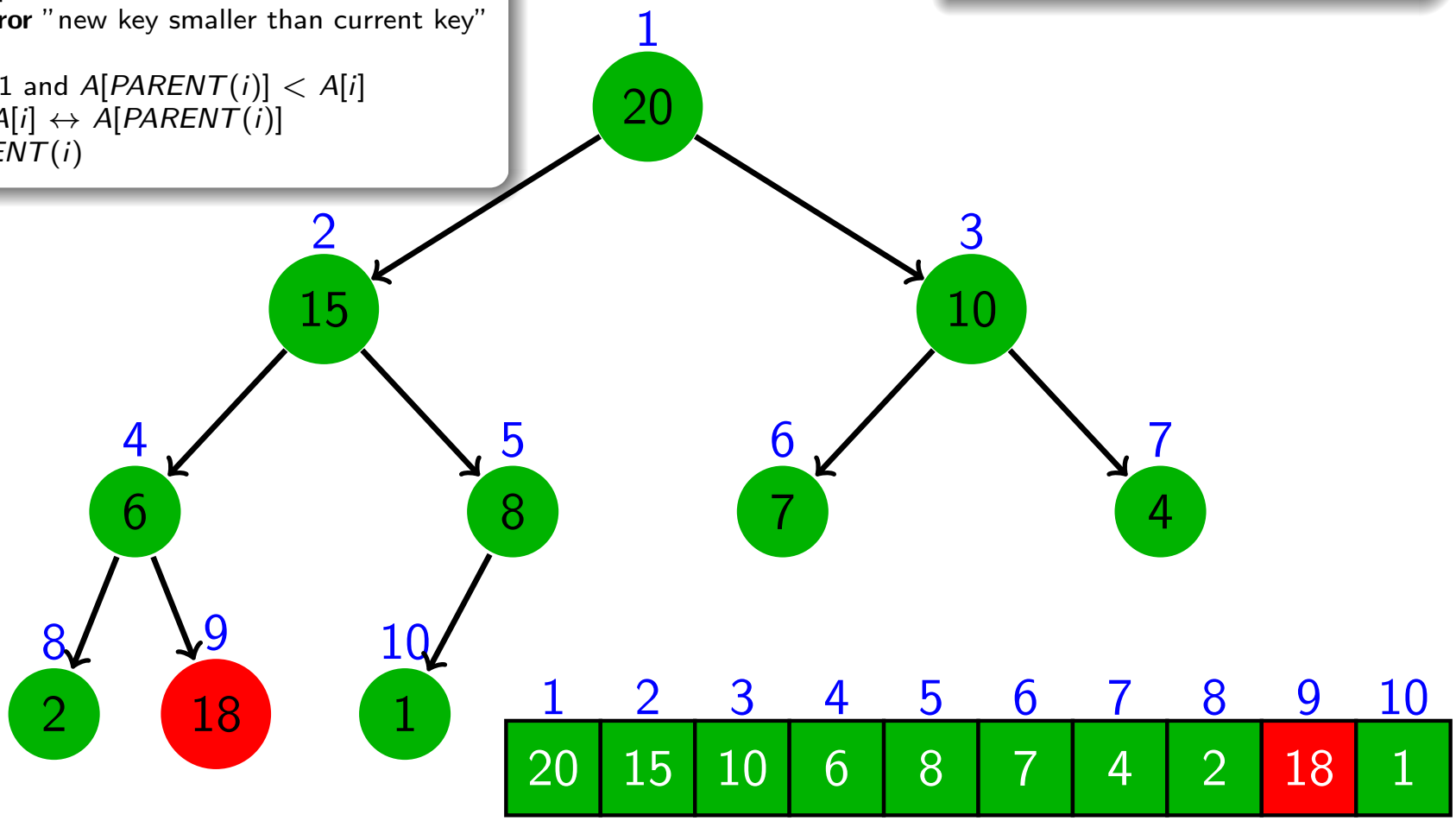


# HEAP-INCREASE-KEY

HEAP-INCREASE-KEY$(A, i, x)$

```
1  if x < A[i]
2      then error "new key smaller than current key"
3  A[i] ← x
4  while i > 1 and A[PARENT(i)] < A[i]
5  exchange A[i] ↔ A[PARENT(i)]
6  i ← PARENT(i)
```

HEAP-INCREASE-KEY$(A, 9, 18)$

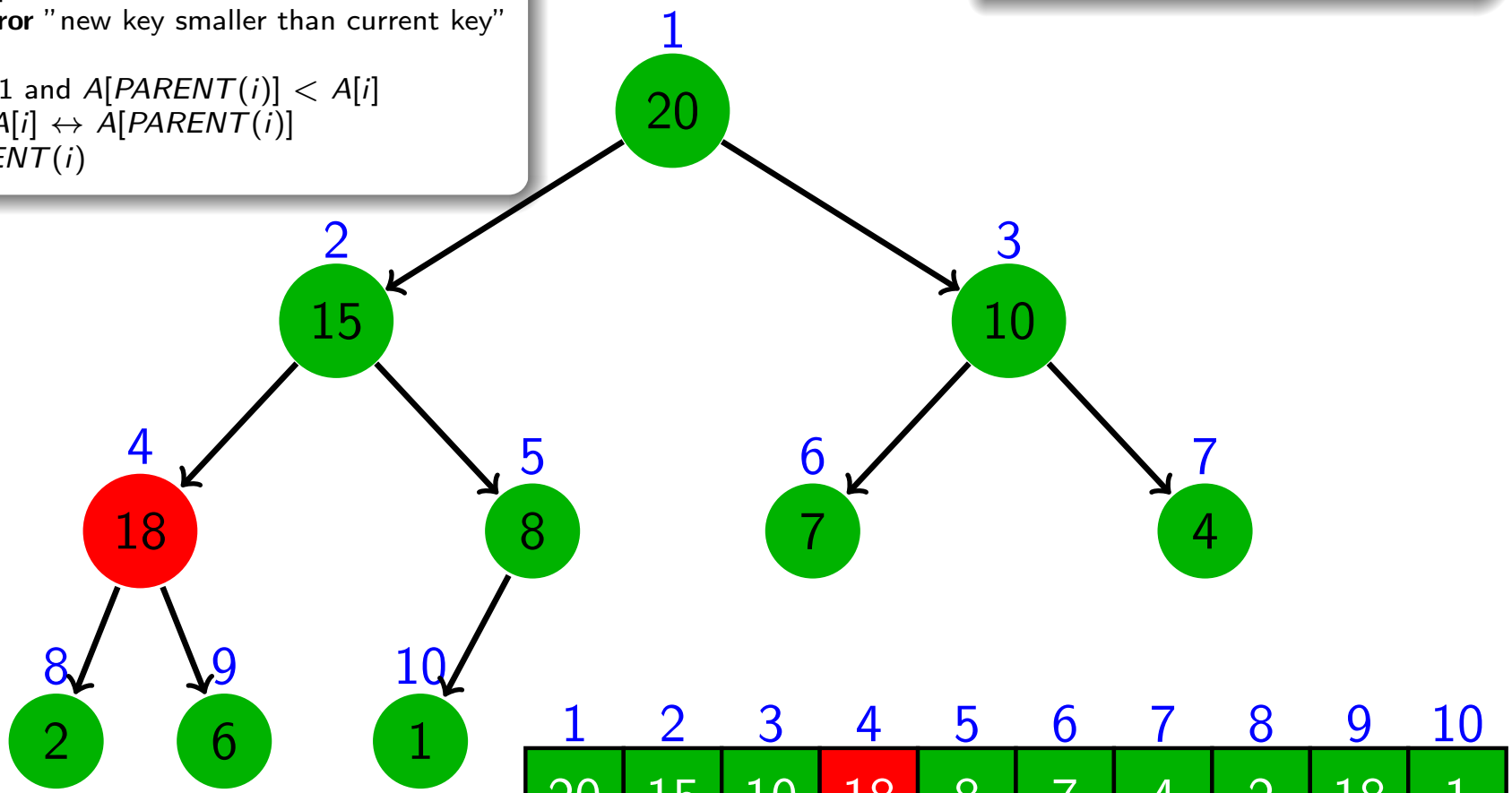


# HEAP-INCREASE-KEY

HEAP-INCREASE-KEY$(A, i, x)$

```
1  if x < A[i]
2      then error "new key smaller than current key"
3  A[i] ← x
4  while i > 1 and A[PARENT(i)] < A[i]
5  exchange A[i] ↔ A[PARENT(i)]
6  i ← PARENT(i)
```

HEAP-INCREASE-KEY$(A, 9, 18)$

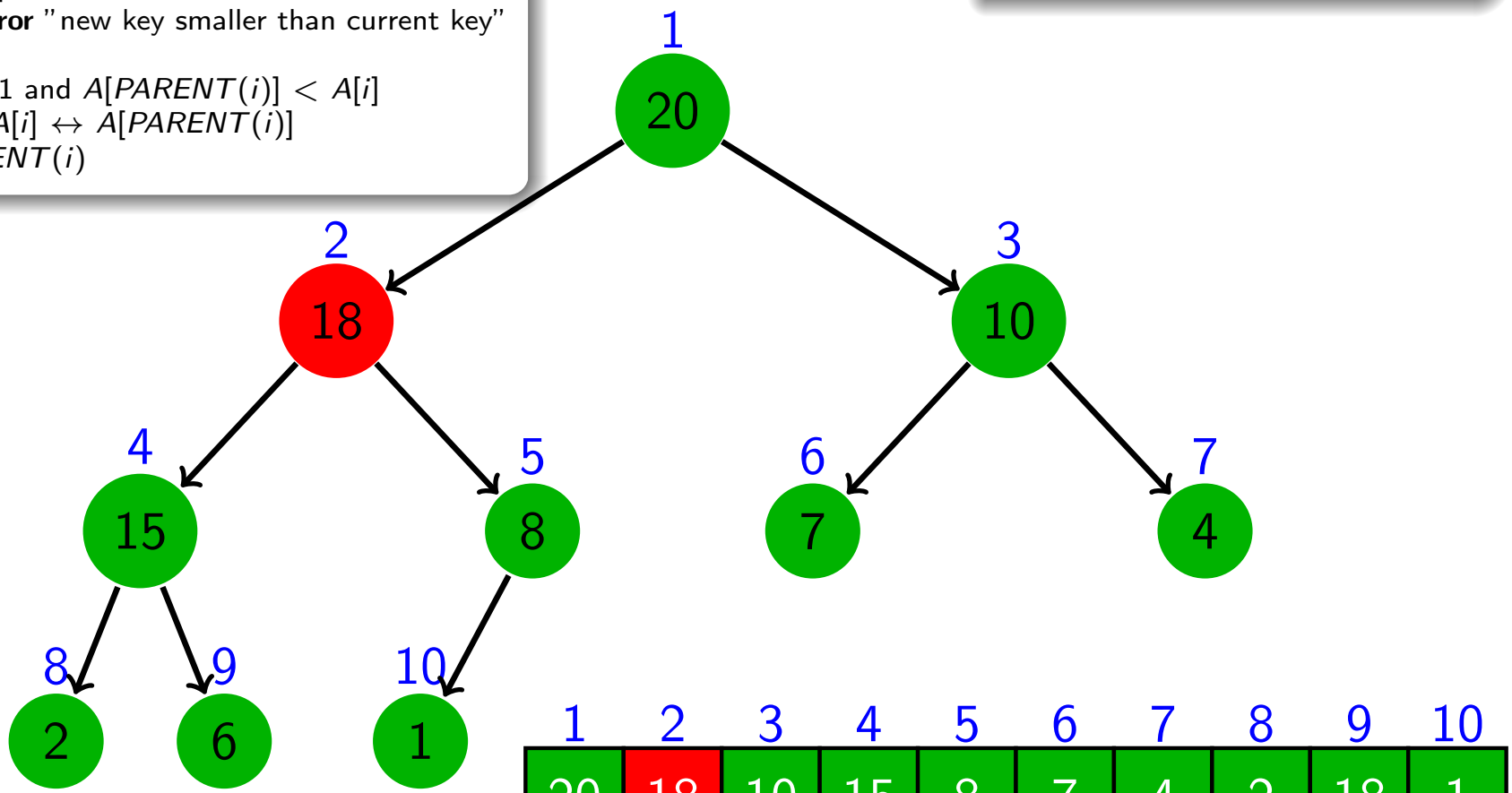


# HEAP-INCREASE-KEY

HEAP-INCREASE-KEY$(A, i, x)$

```
1  if x < A[i]
2      then error "new key smaller than current key"
3  A[i] ← x
4  while i > 1 and A[PARENT(i)] < A[i]
5  exchange A[i] ↔ A[PARENT(i)]
6  i ← PARENT(i)
```

HEAP-INCREASE-KEY$(A, 9, 18)$

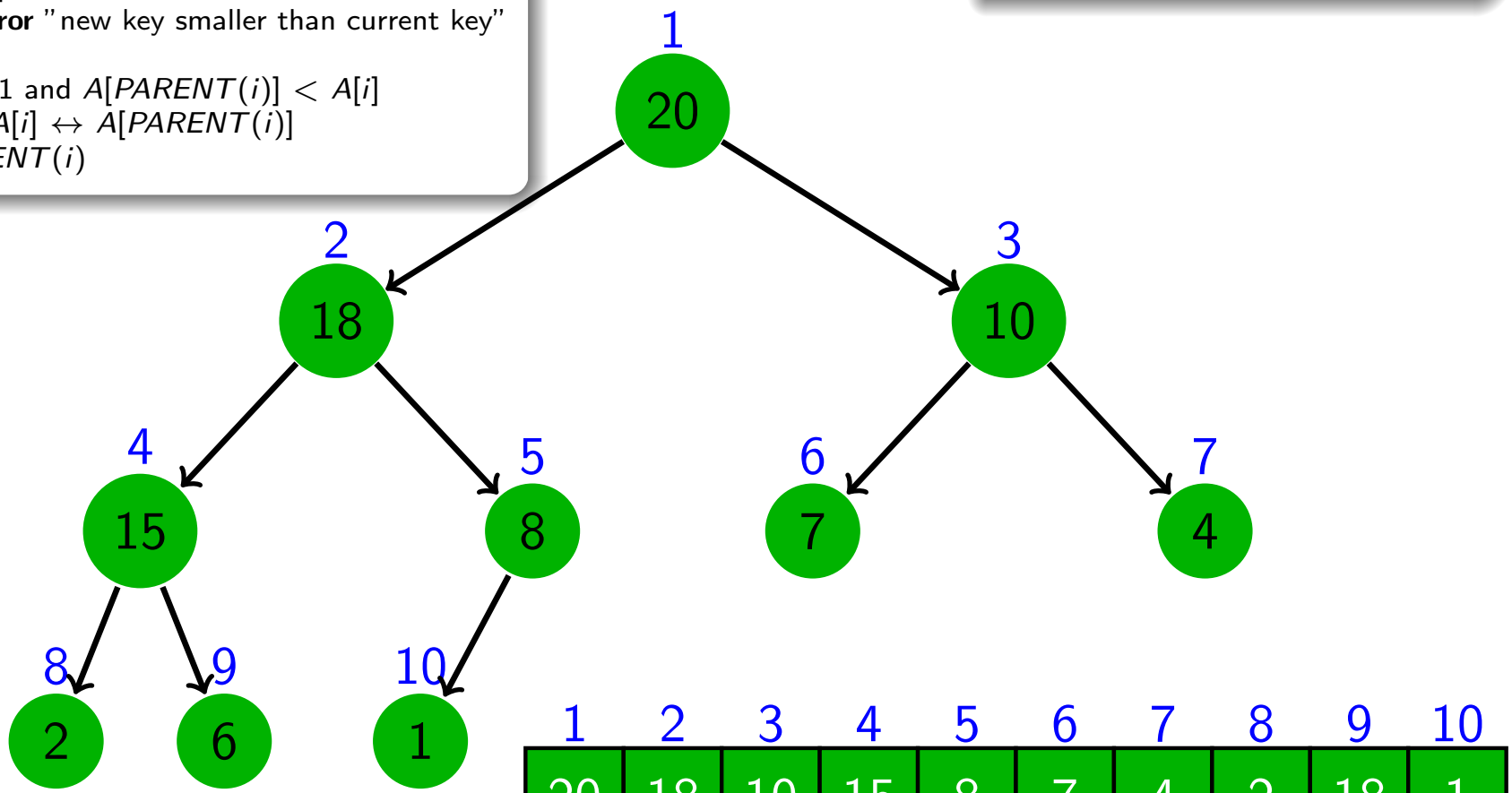


# HEAP-INCREASE-KEY

HEAP-INCREASE-KEY$(A, i, x)$

```
1  if x < A[i]
2      then error "new key smaller than current key"
3  A[i] ← x
4  while i > 1 and A[PARENT(i)] < A[i]
5  exchange A[i] ↔ A[PARENT(i)]
6  i ← PARENT(i)
```

HEAP-INCREASE-KEY$(A, 9, 18)$



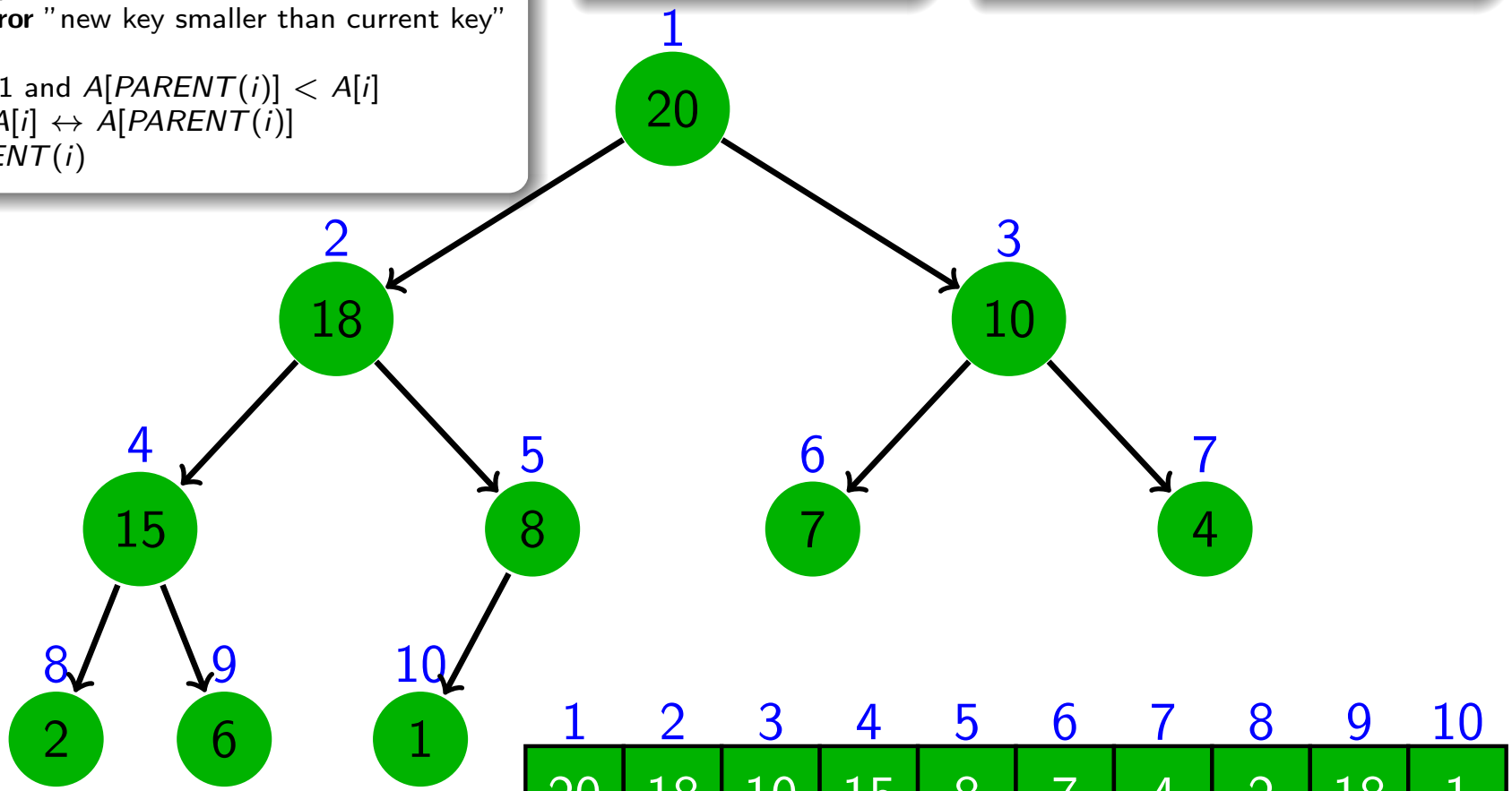
# HEAP-INCREASE-KEY

HEAP-INCREASE-KEY$(A, i, x)$

```
1  if x < A[i]
2      then error "new key smaller than current key"
3  A[i] ← x
4  while i > 1 and A[PARENT(i)] < A[i]
5  exchange A[i] ↔ A[PARENT(i)]
6  i ← PARENT(i)
```

Time: $\Theta(log_2 n)$

HEAP-INCREASE-KEY$(A, 9, 18)$

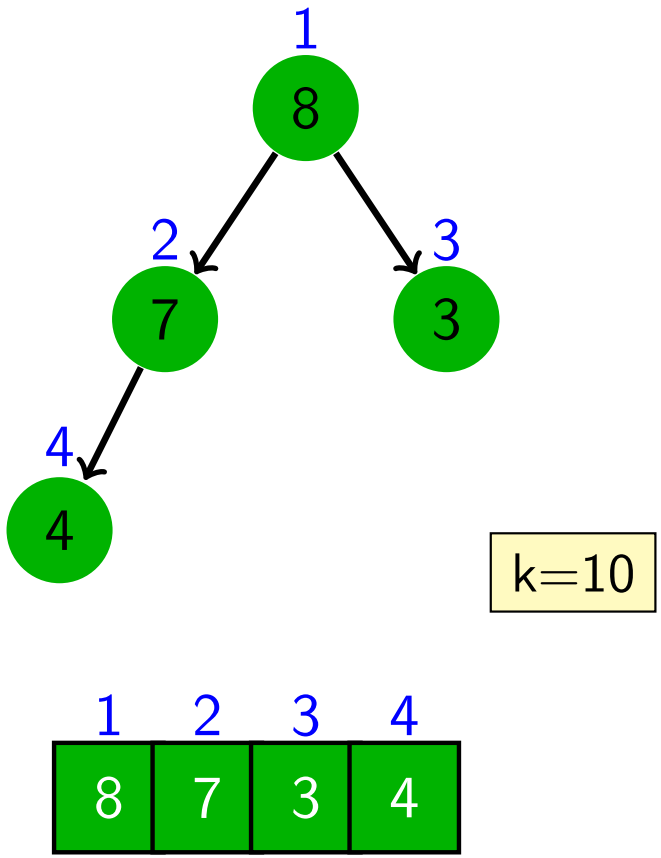



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 20 | 18 | 10 | 15 | 8 | 7 | 4 | 2 | 18 | 1 |

# MAX-HEAP-INSERT: Principle

Problem: inserts the value $k$ into the heap

MAX-HEAP-INSERT

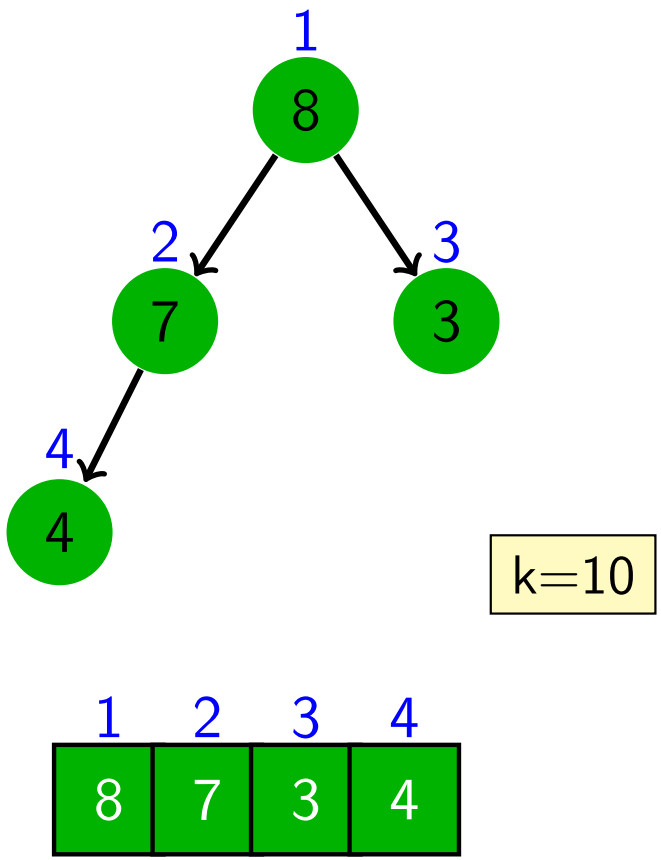


# MAX-HEAP-INSERT: Principle

Problem: inserts the value $k$ into the heap

## MAX-HEAP-INSERT

- Insert a new node in the very last position in the tree with key $-\infty$

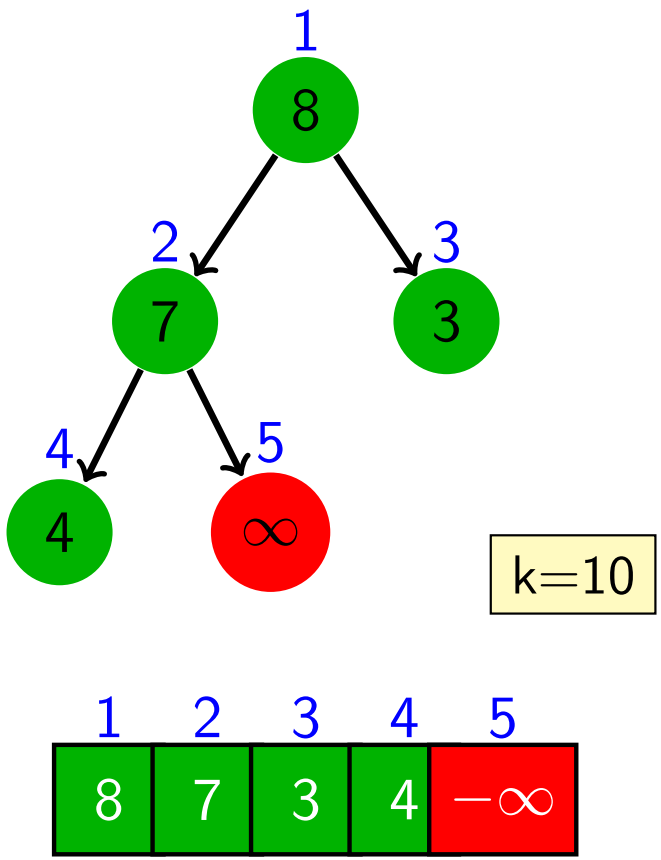


# MAX-HEAP-INSERT: Principle

Problem: inserts the value $k$ into the heap

## MAX-HEAP-INSERT

- Insert a new node in the very last position in the tree with key $-\infty$

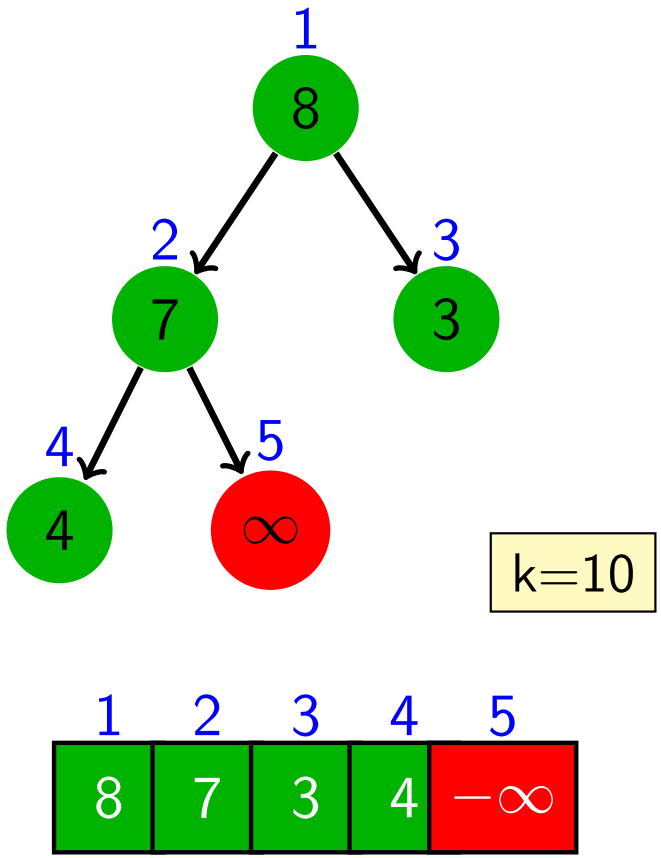


# MAX-HEAP-INSERT: Principle

Problem: inserts the value $k$ into the heap

## MAX-HEAP-INSERT

- Insert a new node in the very last position in the tree with key $-\infty$
- Increase the $-\infty$ value to $k$ using the HEAP-INCREASE-KEY procedure

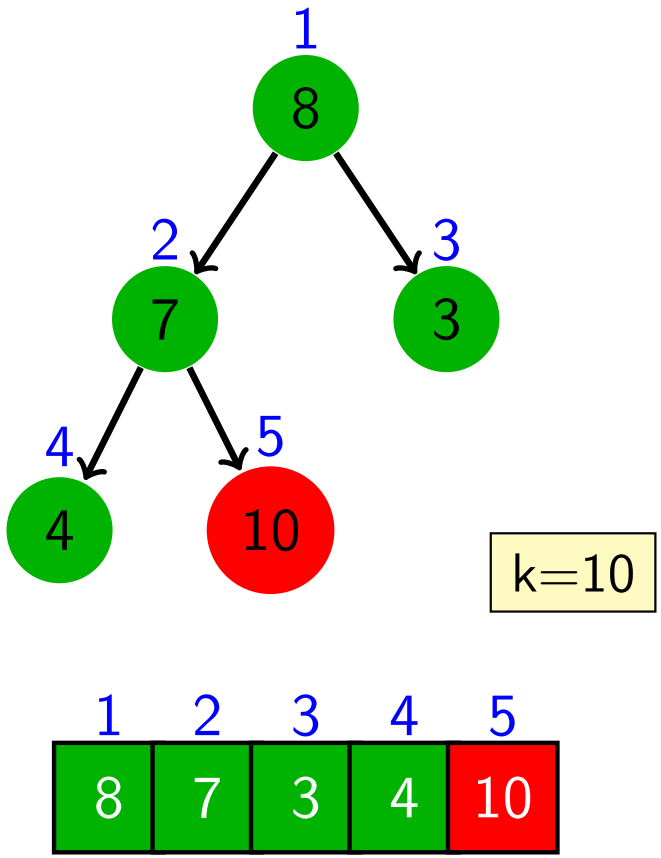


# MAX-HEAP-INSERT: Principle

Problem: inserts the value $k$ into the heap

## MAX-HEAP-INSERT

- Insert a new node in the very last position in the tree with key $-\infty$
- Increase the $-\infty$ value to $k$ using the HEAP-INCREASE-KEY procedure

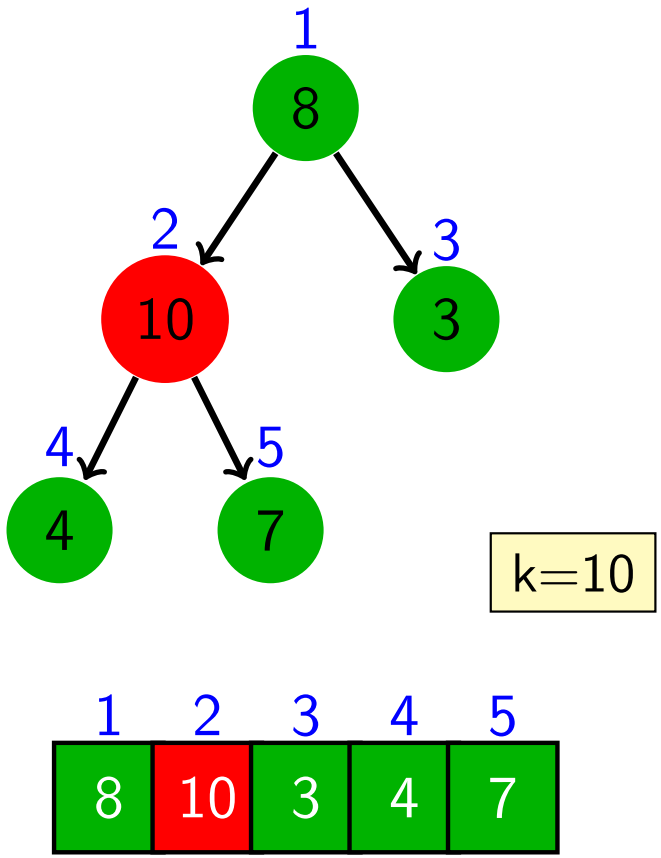


# MAX-HEAP-INSERT: Principle

Problem: inserts the value $k$ into the heap

## MAX-HEAP-INSERT

- Insert a new node in the very last position in the tree with key $-\infty$
- Increase the $-\infty$ value to $k$ using the HEAP-INCREASE-KEY procedure

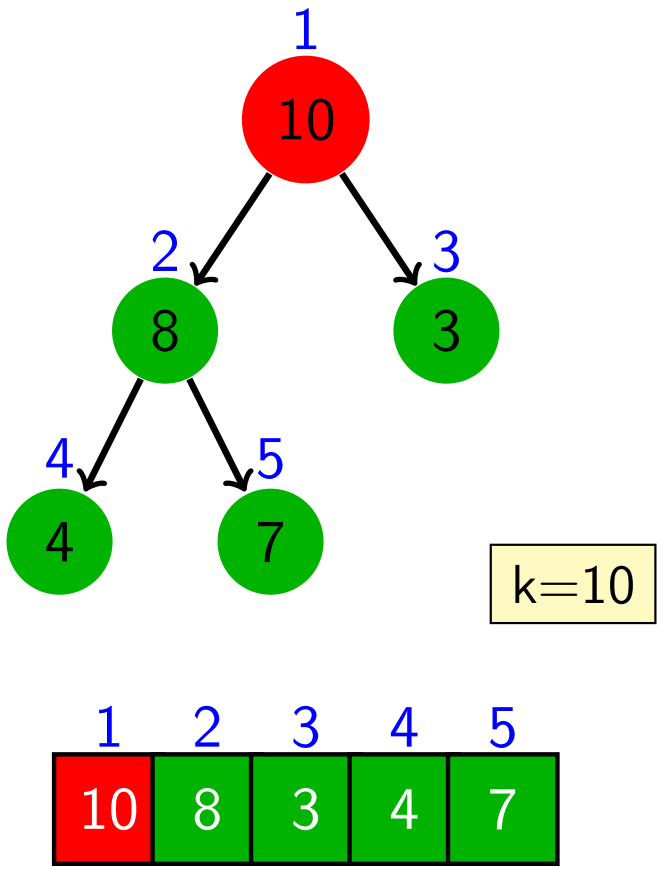


# MAX-HEAP-INSERT: Principle

Problem: inserts the value $k$ into the heap

## MAX-HEAP-INSERT

- Insert a new node in the very last position in the tree with key $-\infty$
- Increase the $-\infty$ value to $k$ using the HEAP-INCREASE-KEY procedure

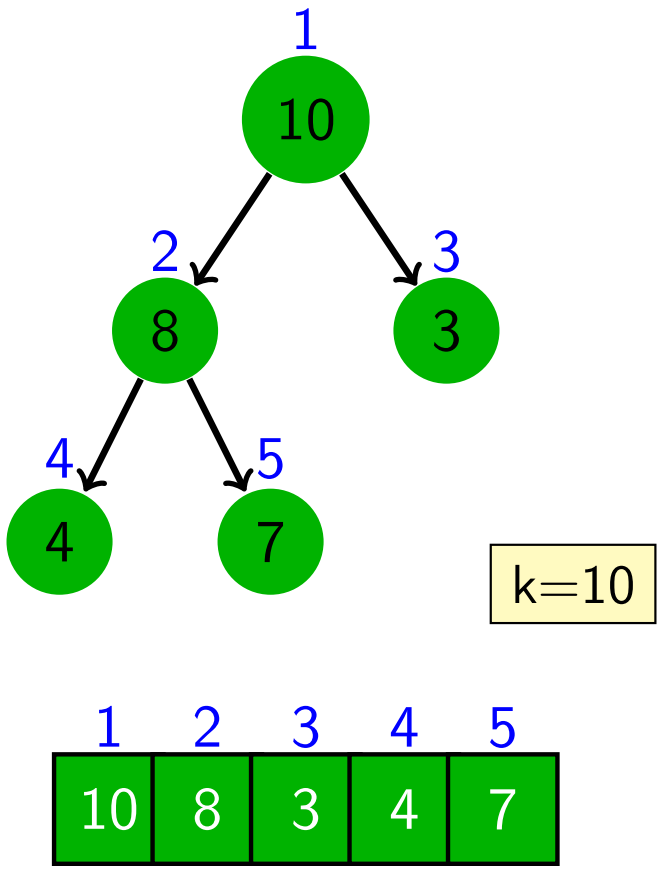


# MAX-HEAP-INSERT: Principle

Problem: inserts the value $k$ into the heap

## MAX-HEAP-INSERT

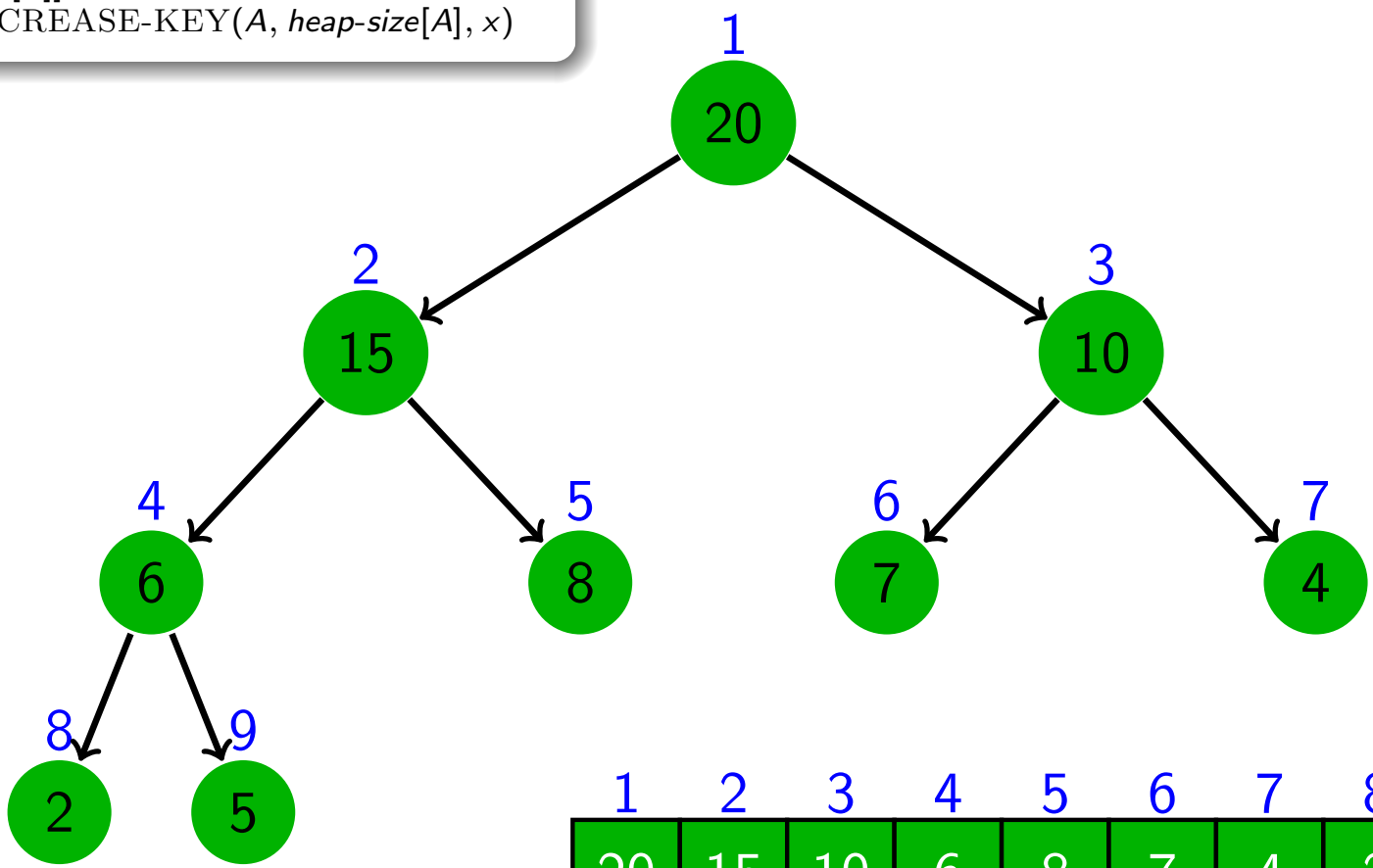
- Insert a new node in the very last position in the tree with key $-\infty$
- Increase the $-\infty$ value to $k$ using the HEAP-INCREASE-KEY procedure



# MAX-HEAP-INSERT

MAX-HEAP-INSERT($A, x$)

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



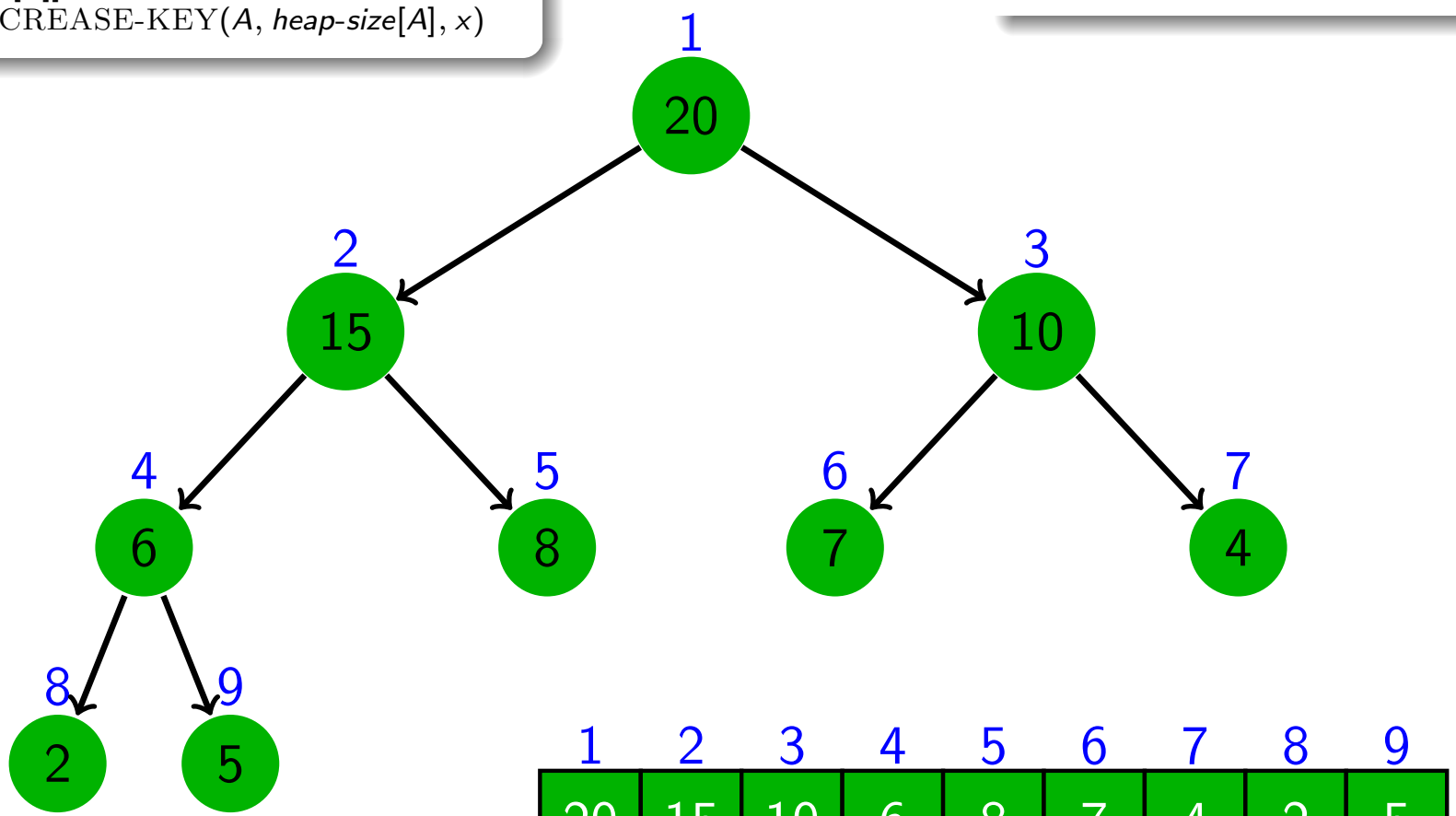

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 20 | 15 | 10 | 6 | 8 | 7 | 4 | 2 | 5 |

# MAX-HEAP-INSERT

MAX-HEAP-INSERT$(A, x)$

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

MAX-HEAP-INSERT$(A, 16)$

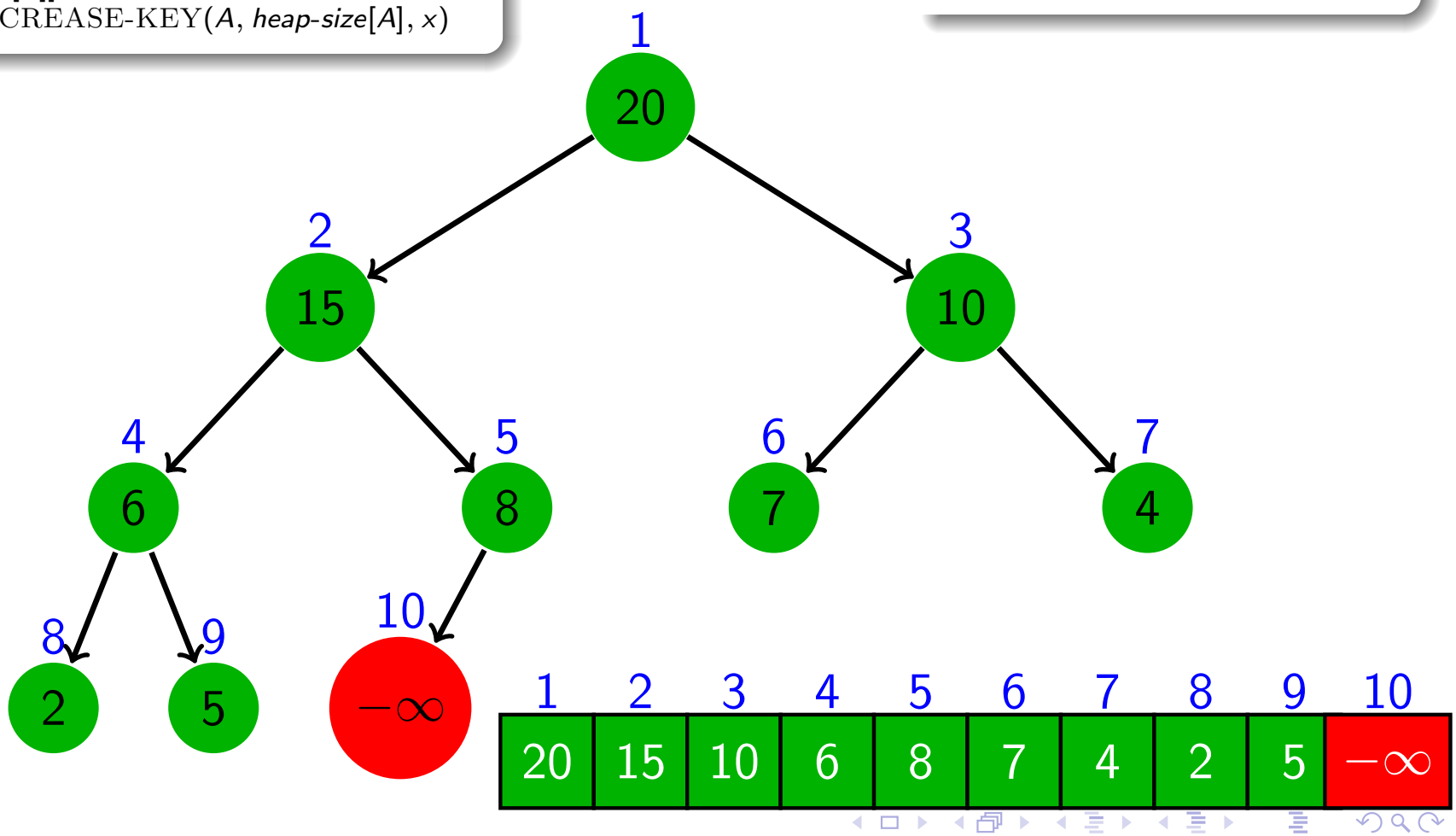


# MAX-HEAP-INSERT

MAX-HEAP-INSERT$(A, x)$

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

MAX-HEAP-INSERT$(A, 16)$

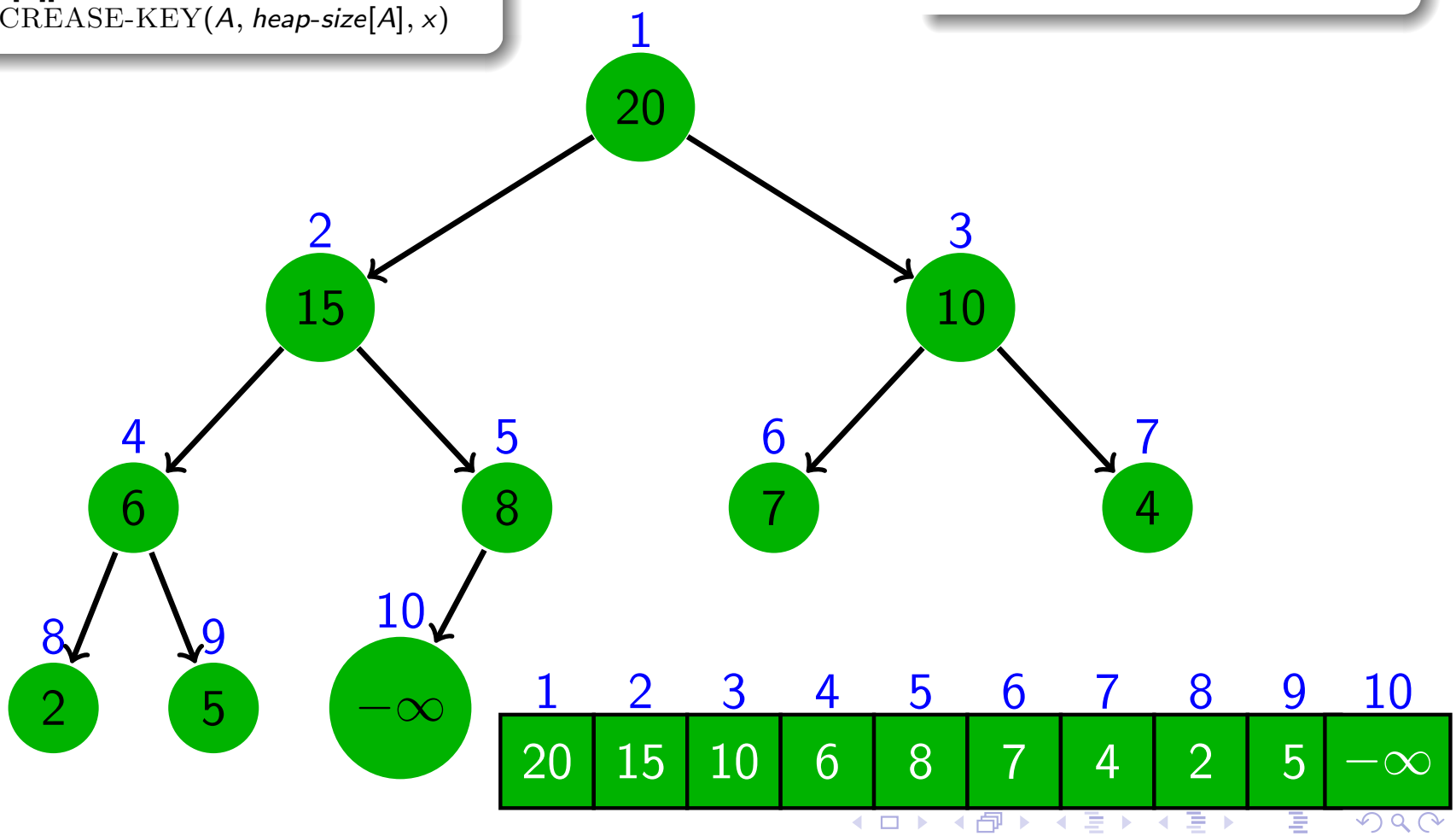


# MAX-HEAP-INSERT

MAX-HEAP-INSERT($A, x$)

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

MAX-HEAP-INSERT($A, 16$)

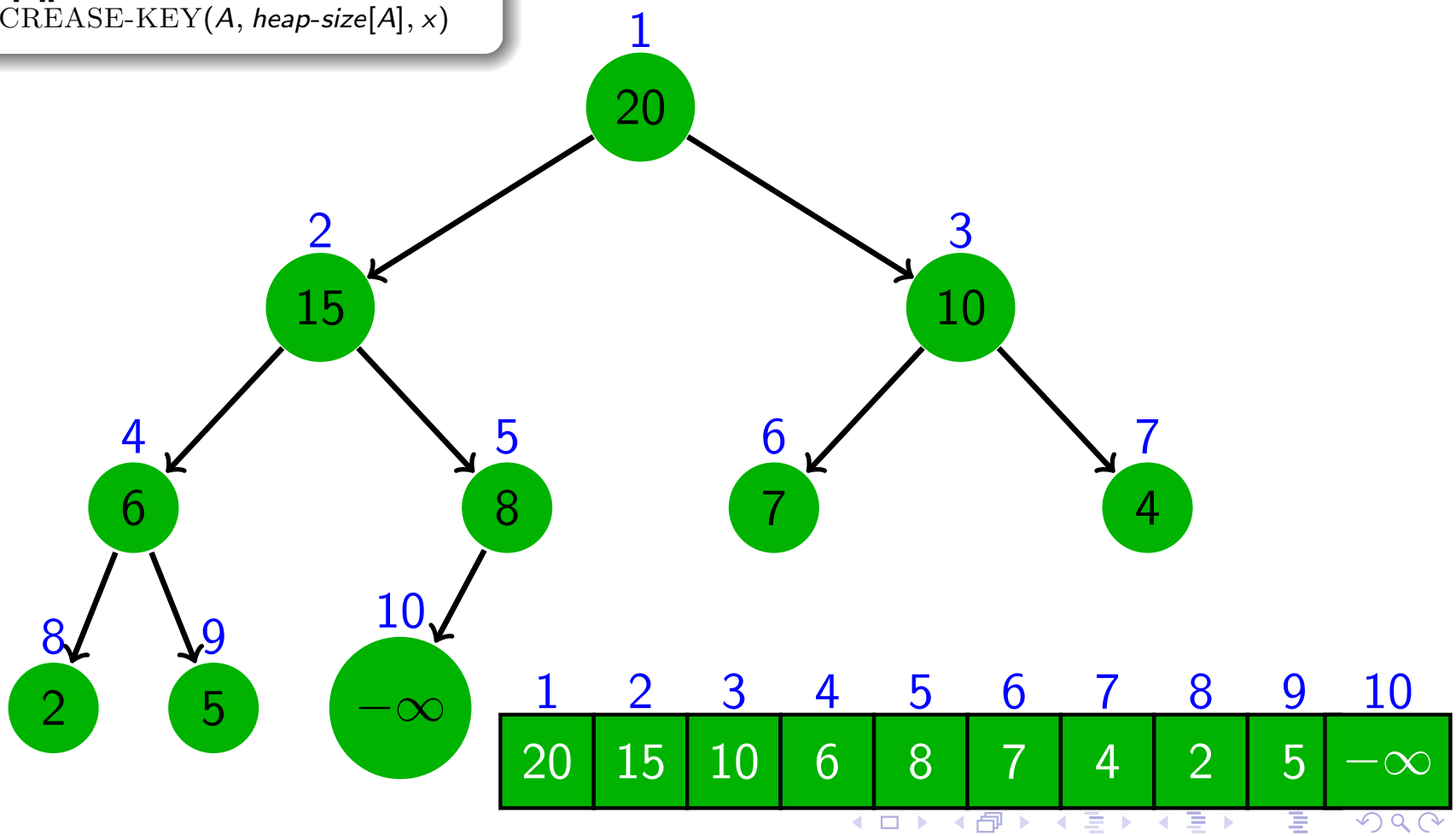


# MAX-HEAP-INSERT

MAX-HEAP-INSERT($A, x$)

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

HEAP-INCREASE-KEY($A, 10, 16$)

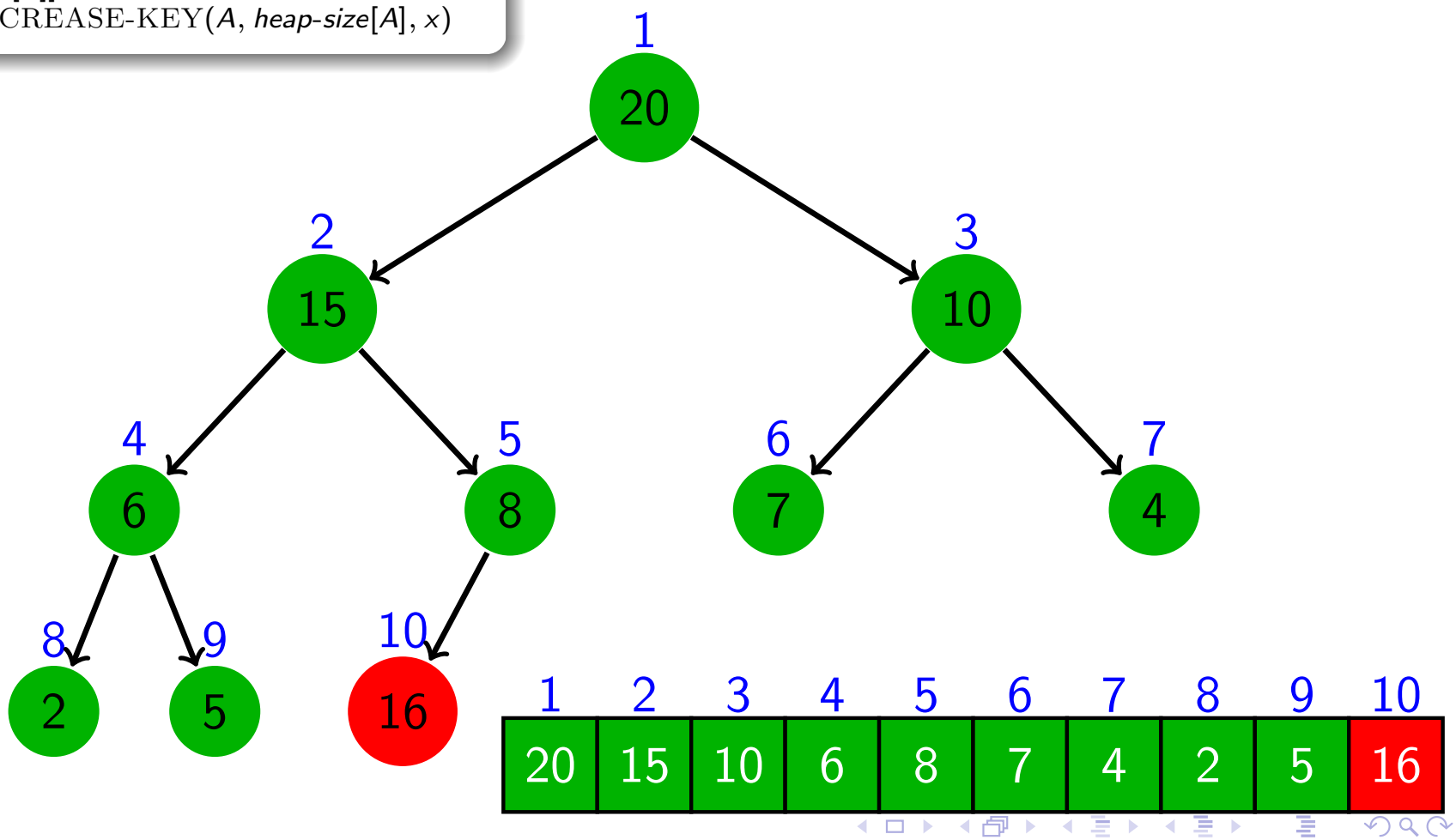


# MAX-HEAP-INSERT

MAX-HEAP-INSERT($A, x$)

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

HEAP-INCREASE-KEY($A, 10, 16$)

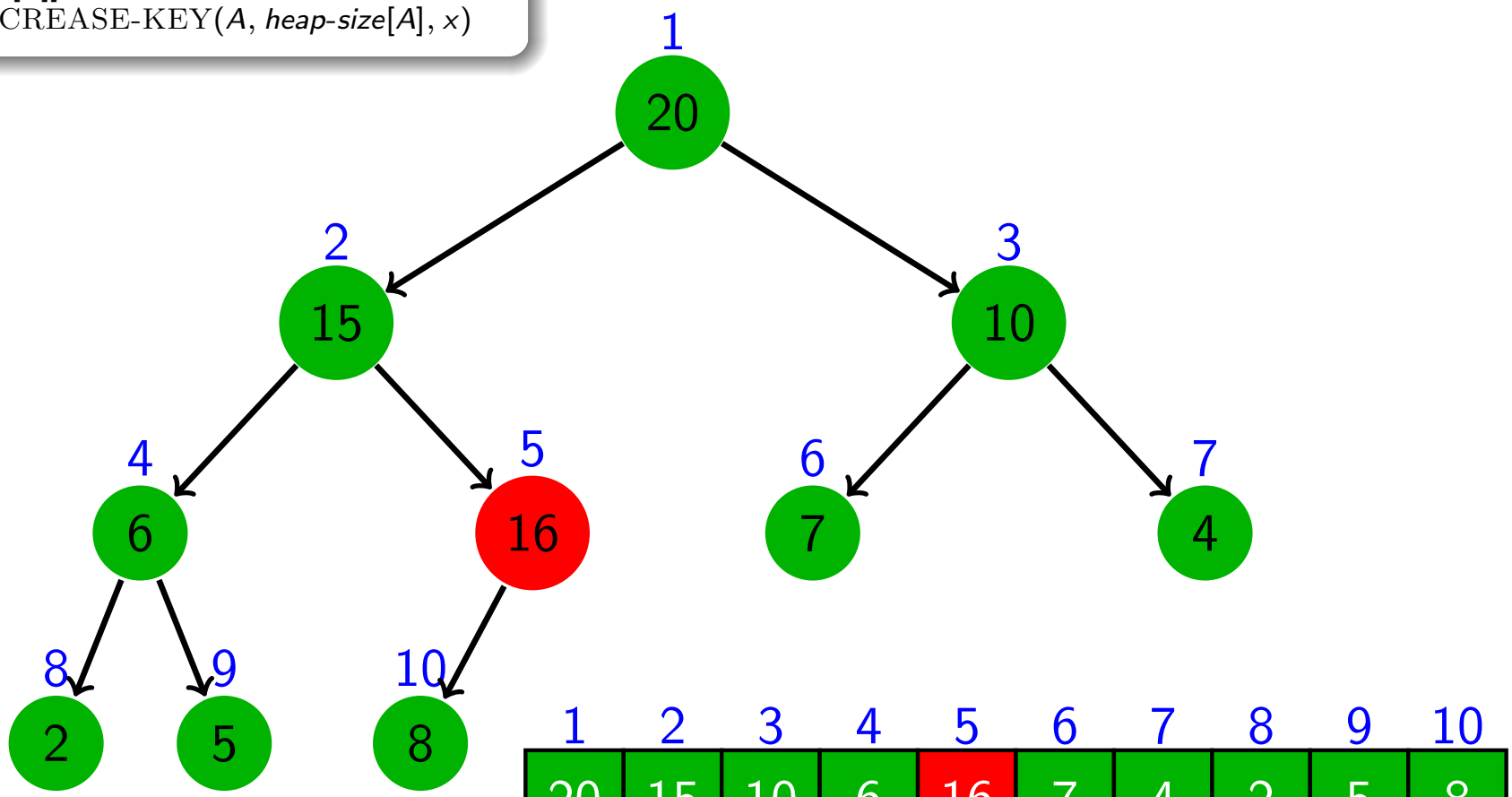


# MAX-HEAP-INSERT

MAX-HEAP-INSERT($A, x$)

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

HEAP-INCREASE-KEY($A, 10, 16$)

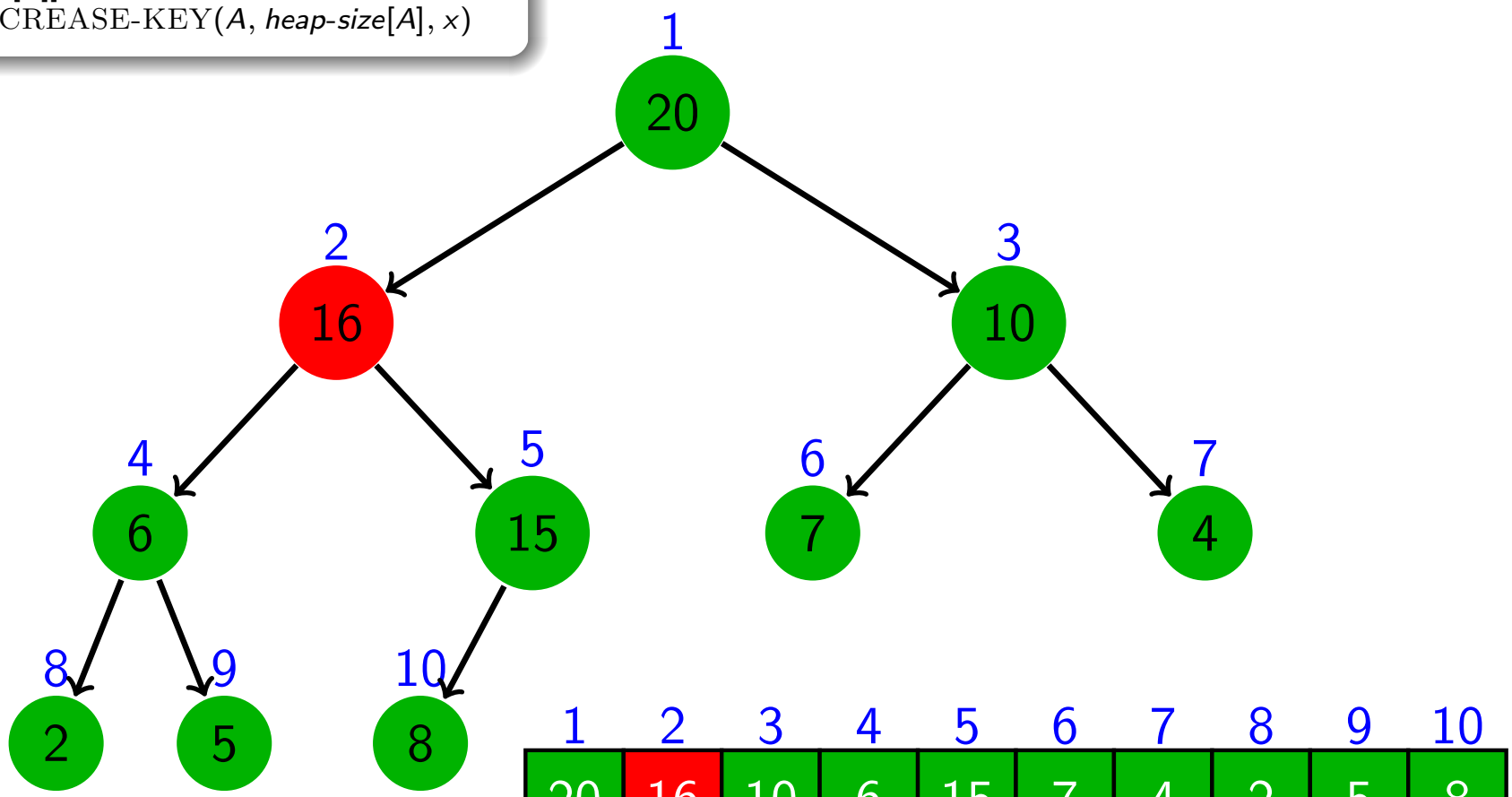


# MAX-HEAP-INSERT

MAX-HEAP-INSERT$(A, x)$

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

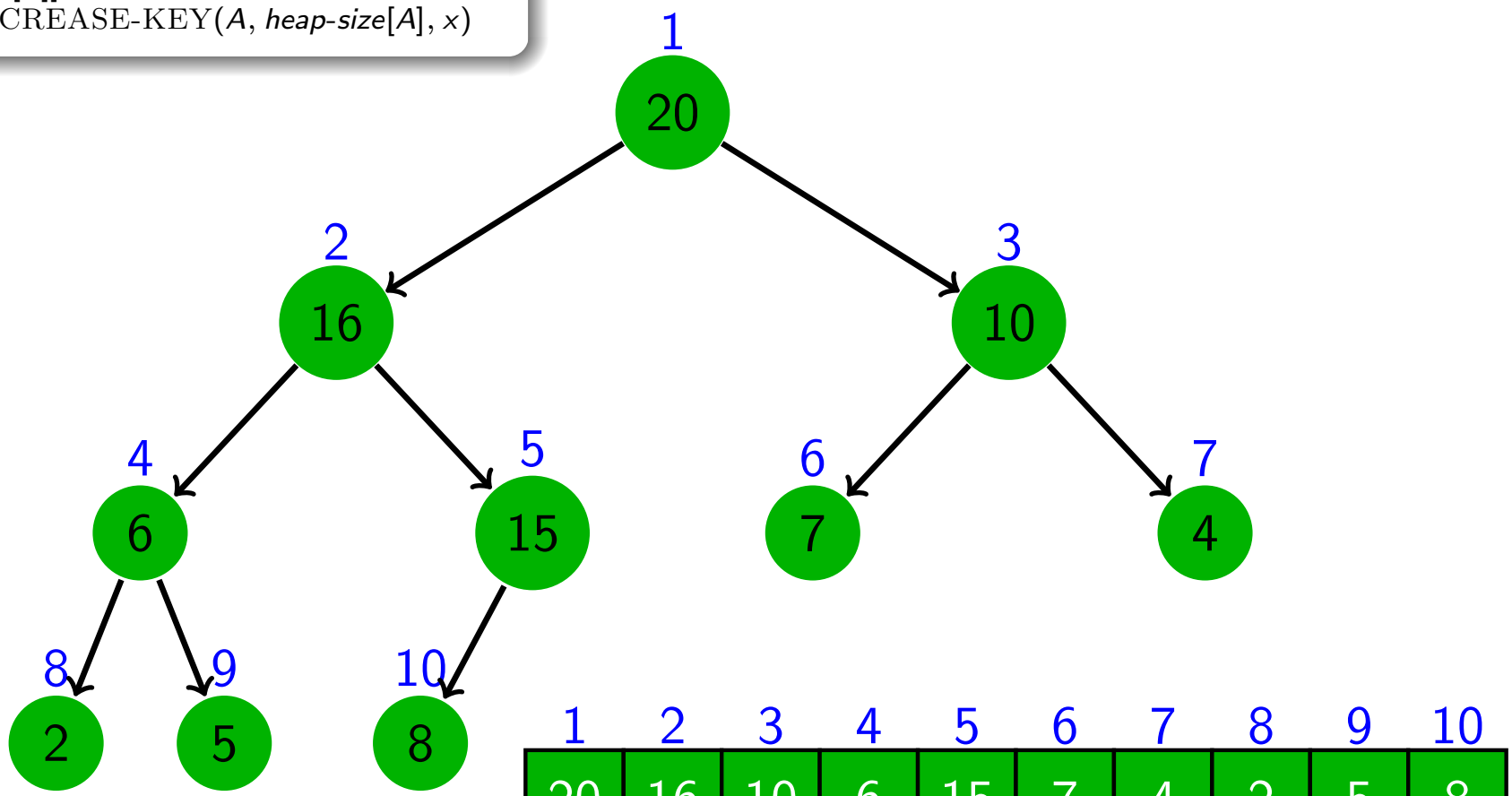
HEAP-INCREASE-KEY$(A, 10, 16)$



# MAX-HEAP-INSERT

MAX-HEAP-INSERT($A, x$)

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



# MAX-HEAP-INSERT

MAX-HEAP-INSERT$(A, x)$

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

Time:
$\Theta(log_2 n)$