

HEAPS, HEAPSORT & PRIORITY QUEUES

Pontus Ekberg

Uppsala University

(Based on previous material by Mohamed Faouzi Atig and Parosh Aziz Abdulla)

- 1 Introduction
- 2 Tree Definition
- 3 Heap Definition
- 4 MAX-HEAPIFY
- 5 BUILD-MAX-HEAP
- 6 HEAP-SORT
- 7 Priority Queues

Sorting Algorithms

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

Algorithm	Worst-Case	"Average-Case"	Best-Case	In place?
InsertionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	Yes
MergeSort	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	No
QuickSort	$\Theta(n^2)$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	Yes

Sorting Algorithms

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

Algorithm	Worst-Case	"Average-Case"	Best-Case	In place?
InsertionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	Yes
MergeSort	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	No
QuickSort	$\Theta(n^2)$	$\Theta(n \log(n))$	$\Theta(n \log(n))$	Yes
HeapSort	$\Theta(n \log(n))$	$\Theta(n \log(n))$	$\Theta(n \log(n))^1$	Yes

¹Assuming all distinct elements; with n identical elements HeapSort is $\Theta(n)$.

HeapSort: Introduction

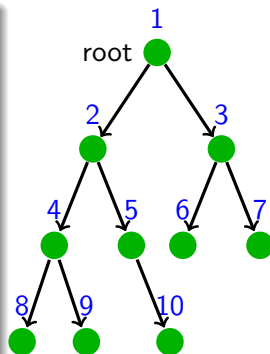
- HeapSort was invented by J. W. J. Williams in 1964.
- Based on a useful of data structure called **heap**
- Sorting in place algorithm

Tree: Definition

- A tree T is a directed graph (V, E) where:
 - V is a set of vertices (or nodes).
 - $E \subseteq V \times V$ is a finite set of edges (or arcs).

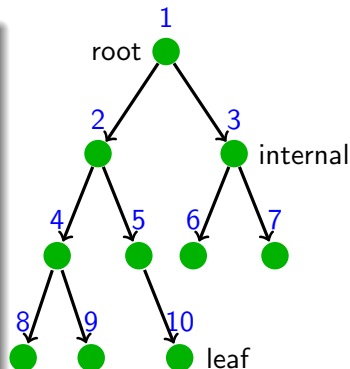
such that the following properties hold:

- 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.



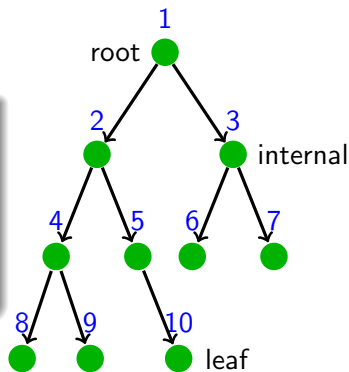
Tree: Notations

- 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.
- A node with no children is a **leaf**.



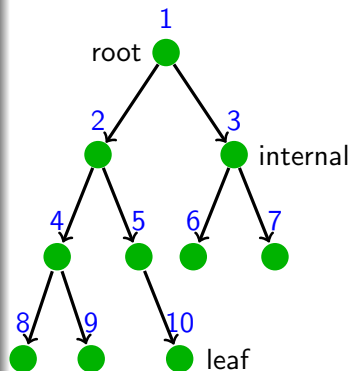
Tree: Notations

- 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



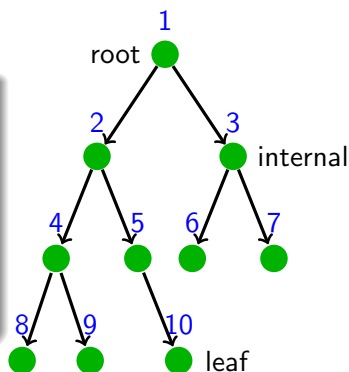
Tree: Notations

- A **path** is a sequence of nodes n_1, n_2, \dots, 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 .



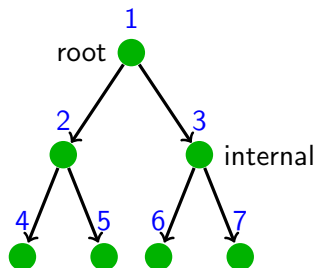
Tree: Notations

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



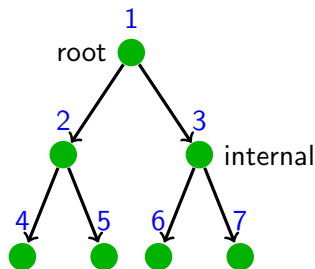
Binary Trees

- 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.



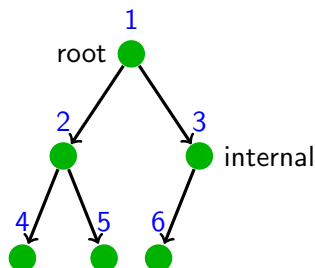
Full Binary Trees: Properties

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

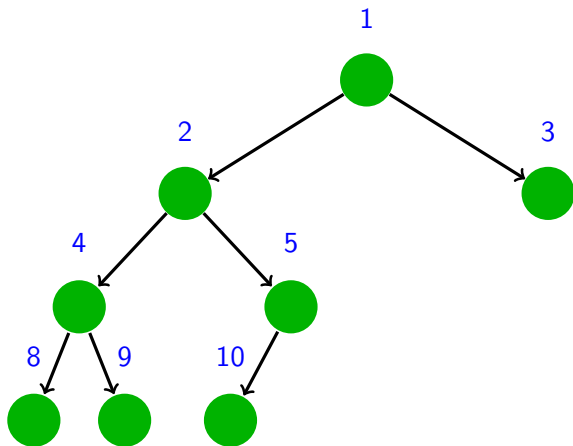


Complete Binary Tree

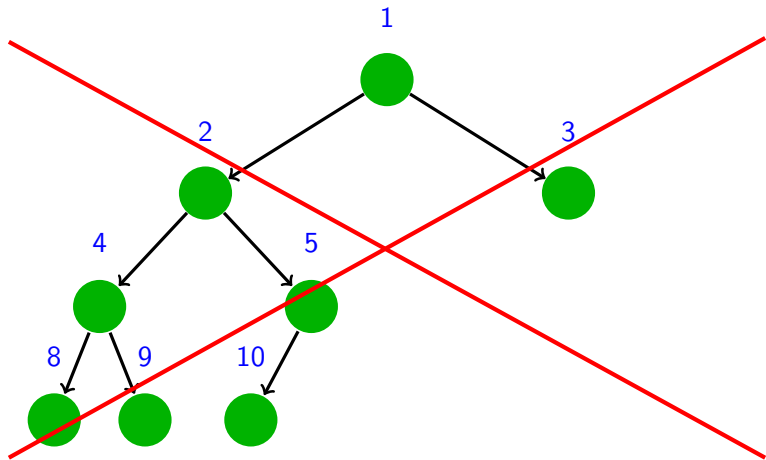
- A **complete** binary tree is a binary tree, which is completely filled at all the levels except possibly the highest, which is filled from left. Formally, we have
 - If h is the height of the tree, then:
 - For all $i : 0 \leq i < h$, there is exactly 2^i nodes at depth i
 - A leaf node has a depth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



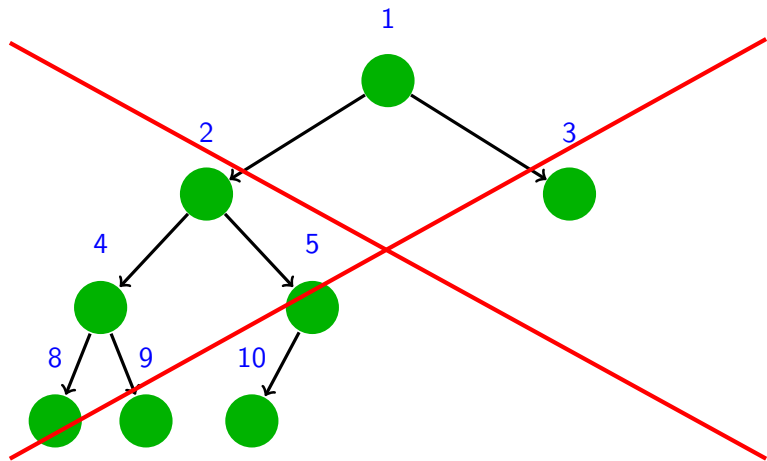
Example (1/2)



Example (1/2)

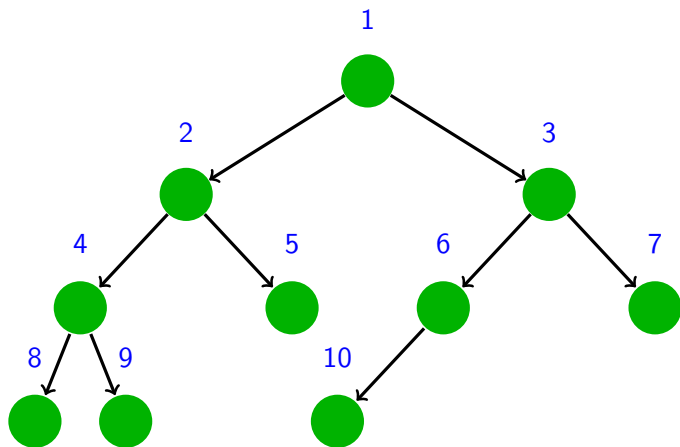


Example (1/2)

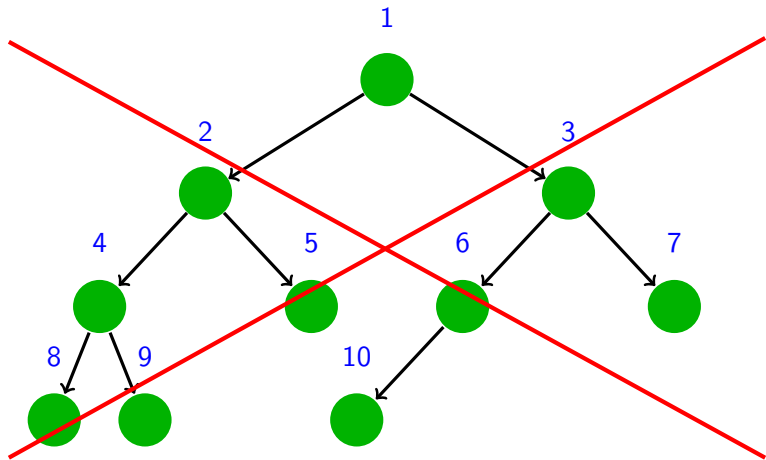


It is not completely filled at level 2

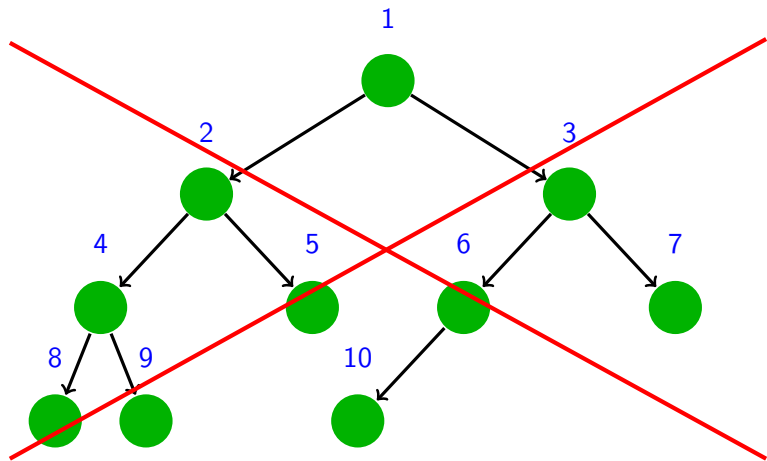
Example (2/2)



Example (2/2)



Example (2/2)



The leaves are not filled from left to right.

Complete binary tree: 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., $n \leq 2^{h+1} - 1$)

$$2^h \leq n \leq 2^{h+1} - 1 \Rightarrow 2^h \leq n < 2^{h+1}$$

$$\Rightarrow h \leq \log_2(n) < h + 1$$

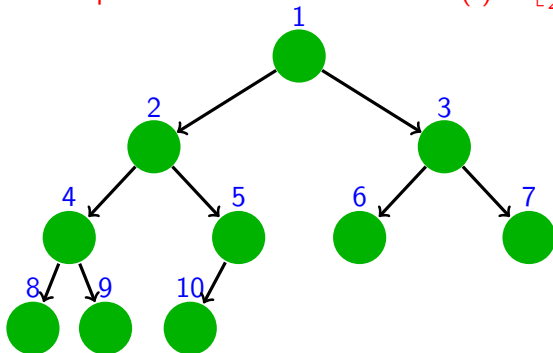
$$\Rightarrow h \leq \log_2(n) < h + 1$$

$$\Rightarrow h = \lfloor \log_2(n) \rfloor$$

Well-Indexed Tree

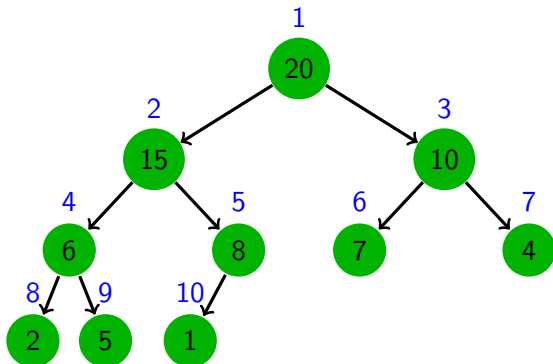
A well-indexed tree is a complete binary tree such that:

- The index of the root is **1**
- The index of the **left child** of a node i is $\text{LEFT}(i) = 2i$
- The index of the **right child** of a node i is $\text{RIGHT}(i) = 2i + 1$
- The index of the **parent** of a node i is $\text{PARENT}(i) = \lfloor \frac{i}{2} \rfloor$

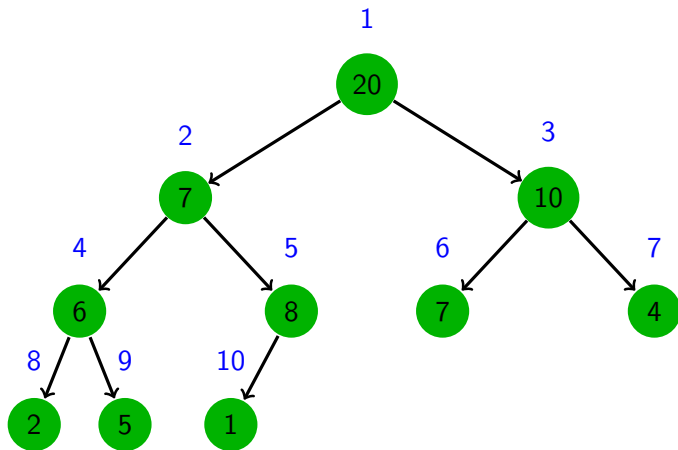


Max-Heap: Definition

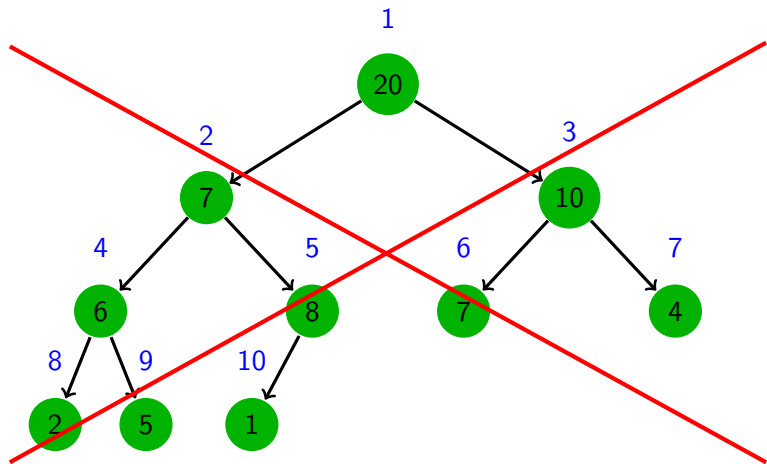
- A **max-heap** is a well-indexed tree such that :
 - Each node is associated with a value.
 - The value of a node is **at most** the value of its parent.



Max-Heap

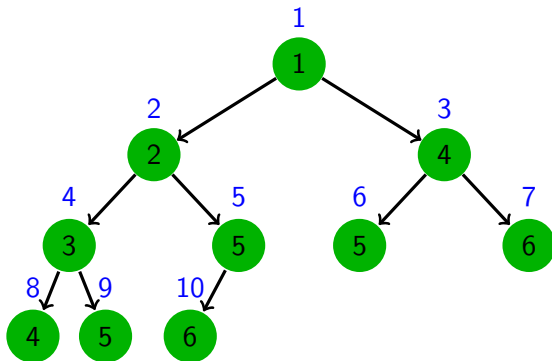


Max-Heap

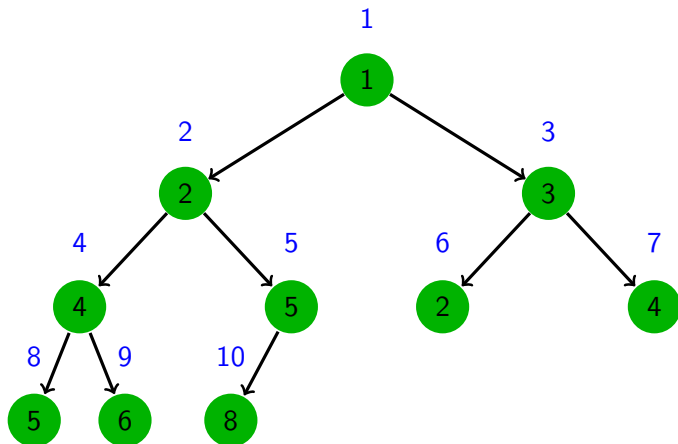


Min-Heap: Definition

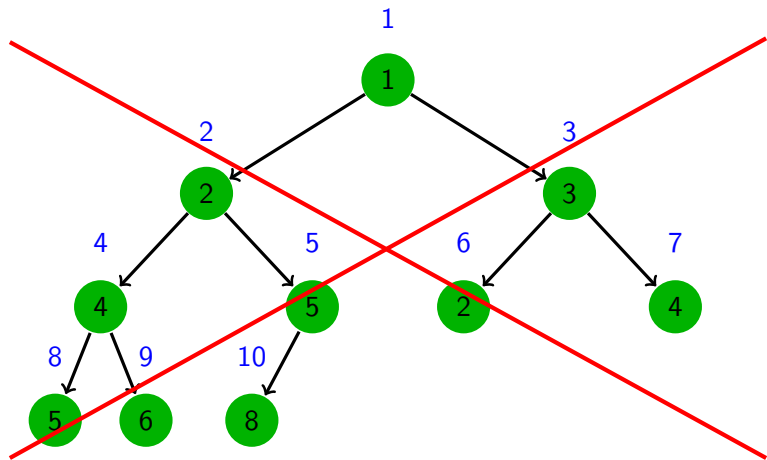
- A **min-heap** is a well-indexed tree such that :
 - Each node is associated with a value.
 - The value of a node is **at least** the value of its parent.



Min-Heap

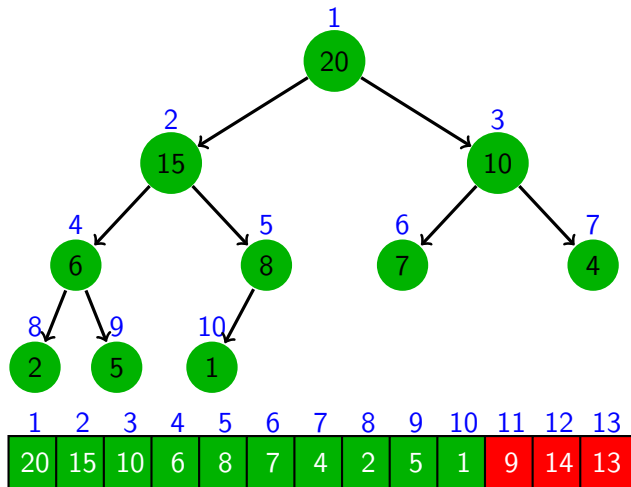


Min-Heap



Implementation of a Heap

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



$A.heap-size=10$

$A.length=13$

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 $i : 2 \leq i \leq A.heap-size$, we have $A[PARENT(i)] \geq A[i]$

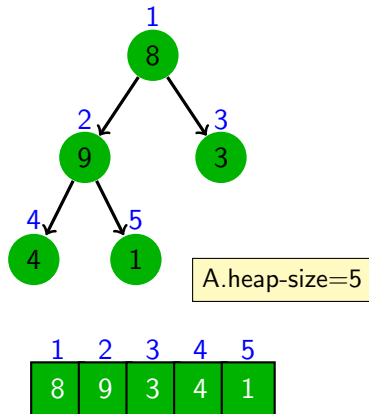
A property of the array A representing a min-heap:

- For all $i : 2 \leq i \leq A.heap-size$, we have $A[PARENT(i)] \leq A[i]$

HeapSort: Principle

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

HeapSort

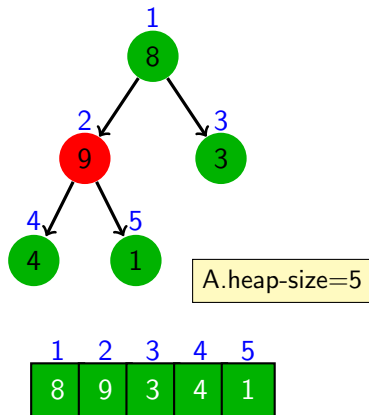


HeapSort: Principle

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

HeapSort

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

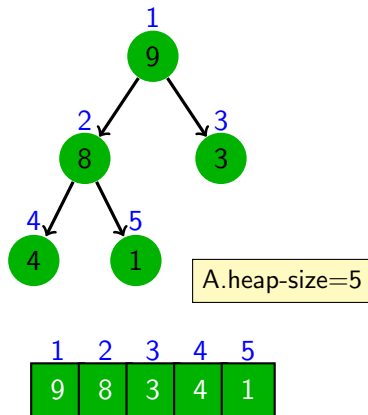


HeapSort: Principle

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

HeapSort

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

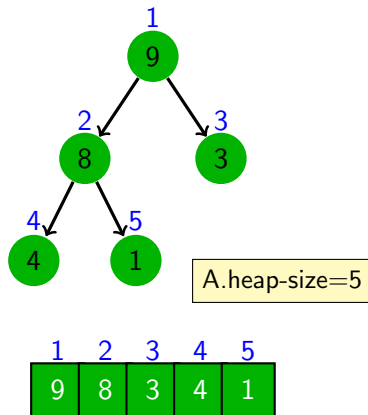


HeapSort: Principle

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:

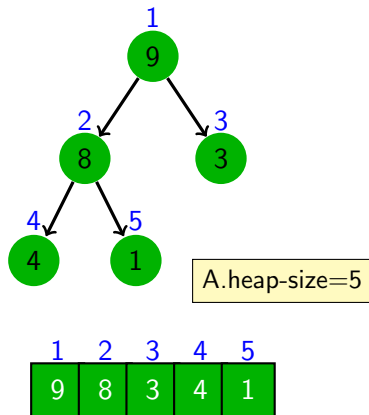


HeapSort: Principle

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 $A[A.\text{heap-size}]$)

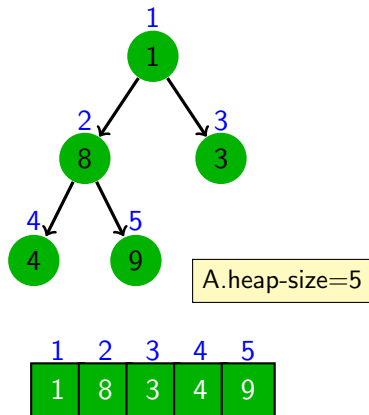


HeapSort: Principle

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 $A[A.\text{heap-size}]$)

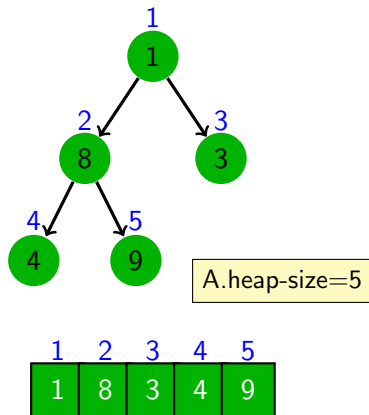


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size

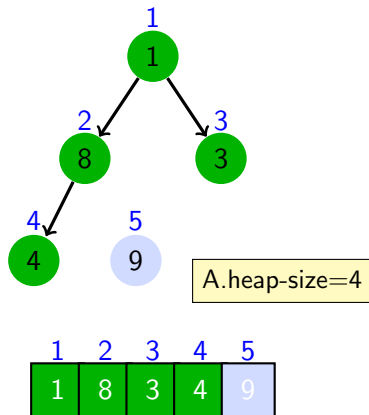


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size

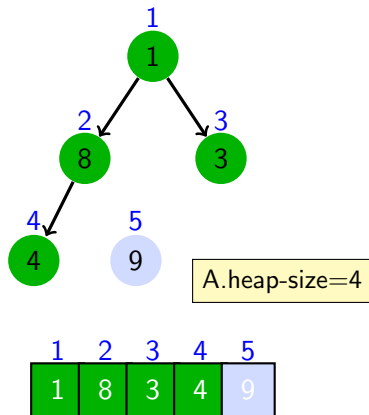


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap property (call **MAX-HEAPIFY**($A,1$))

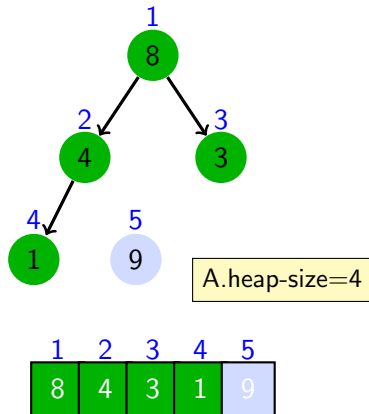


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap property (call **MAX-HEAPIFY**($A,1$))

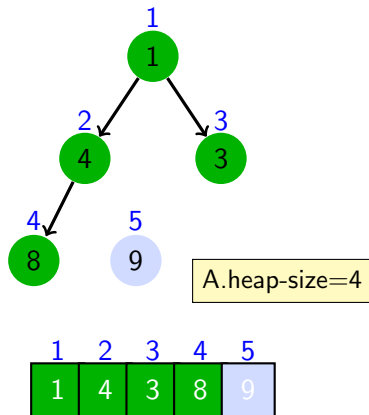


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap property (call **MAX-HEAPIFY**($A,1$))

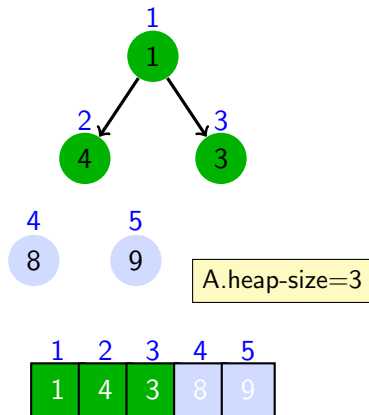


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap property (call **MAX-HEAPIFY**($A,1$))

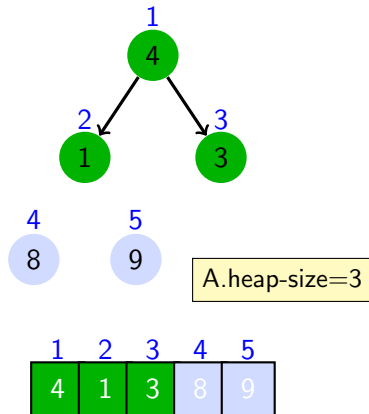


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap property (call **MAX-HEAPIFY**($A,1$))

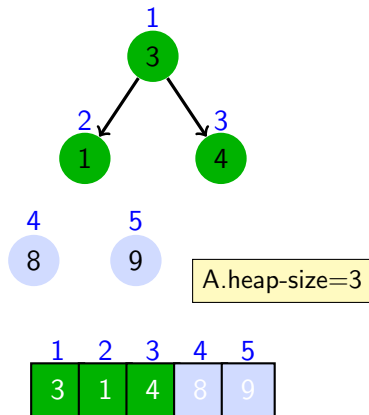


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap property (call **MAX-HEAPIFY**($A,1$))

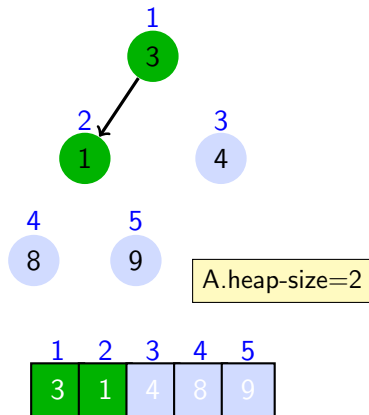


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap property (call **MAX-HEAPIFY**($A,1$))

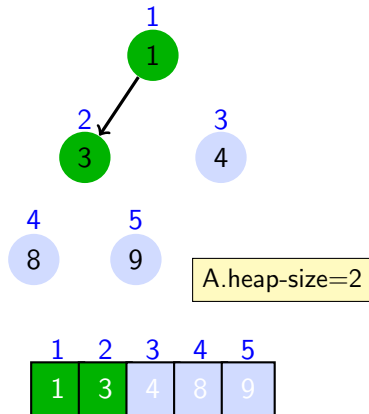


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap property (call **MAX-HEAPIFY**($A,1$))

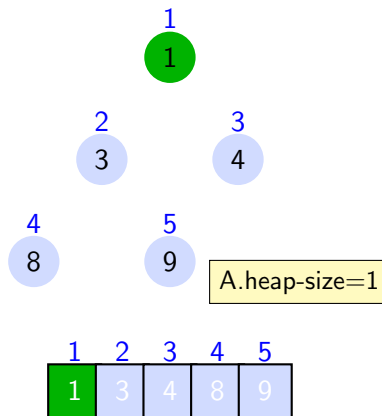


HeapSort: Principle

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 $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap property
(call **MAX-HEAPIFY**($A, 1$))

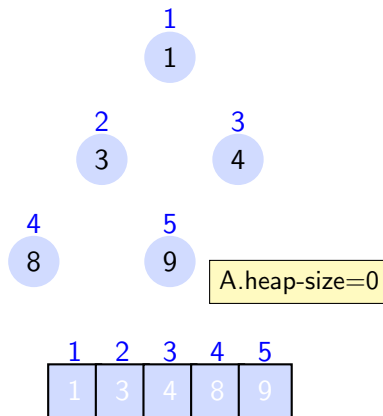


HeapSort: Principle

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

HeapSort

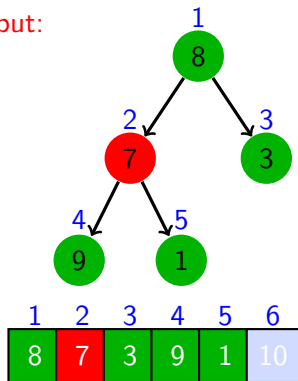
- Construct a max-heap from A
(call **BUILD-MAX-HEAP**(A))
- Repeat until the heap is of size one:
 - Swap the values of the root and the right-most leaf of the heap (i.e., Swap $A[1]$ and $A[A.\text{heap-size}]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the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



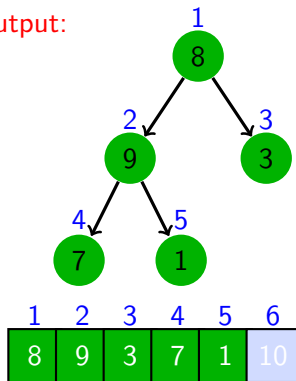
MAX-HEAPIFY

- **Input:** An array A and an index $i : 1 \leq i \leq A.\text{heap-size}$
- **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:



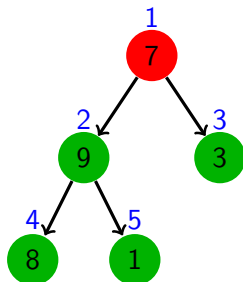
Output:



MAX-HEAPIFY: PRINCIPLE

MAX-HEAPIFY

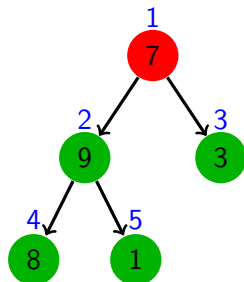
- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$



MAX-HEAPIFY: PRINCIPLE

MAX-HEAPIFY

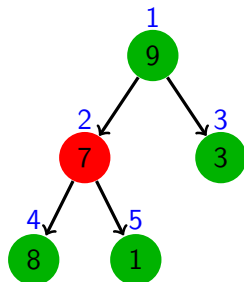
- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$
- If $A[i] < A[\text{LEFT}(i)]$ or $A[i] < A[\text{RIGHT}(i)]$, swap $A[i]$ with the larger of the two children



MAX-HEAPIFY: PRINCIPLE

MAX-HEAPIFY

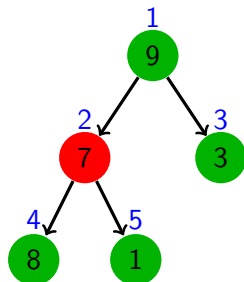
- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$
- If $A[i] < A[\text{LEFT}(i)]$ or $A[i] < A[\text{RIGHT}(i)]$, swap $A[i]$ with the larger of the two children



MAX-HEAPIFY: PRINCIPLE

MAX-HEAPIFY

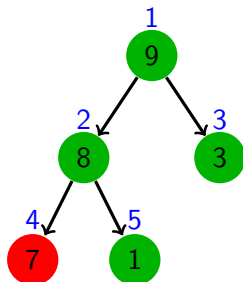
- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$
- If $A[i] < A[\text{LEFT}(i)]$ or $A[i] < A[\text{RIGHT}(i)]$, swap $A[i]$ with the larger of the two children
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is a max-heap



MAX-HEAPIFY: PRINCIPLE

MAX-HEAPIFY

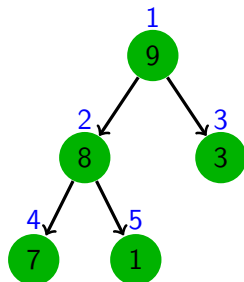
- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$
- If $A[i] < A[\text{LEFT}(i)]$ or $A[i] < A[\text{RIGHT}(i)]$, swap $A[i]$ with the larger of the two children
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is a max-heap



MAX-HEAPIFY: PRINCIPLE

MAX-HEAPIFY

- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$
- If $A[i] < A[\text{LEFT}(i)]$ or $A[i] < A[\text{RIGHT}(i)]$, 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

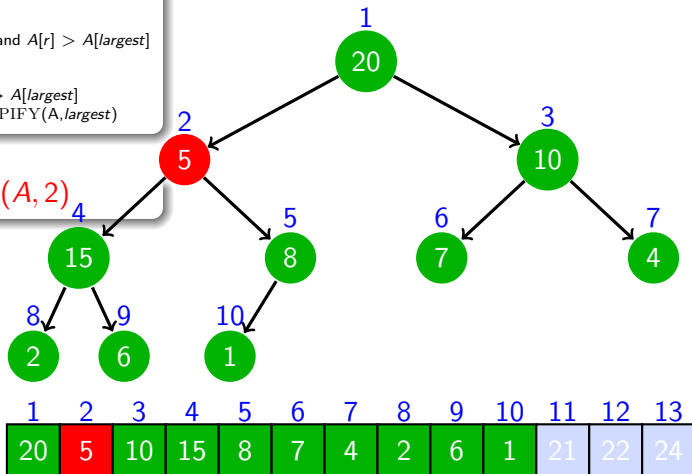


MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1  $l \leftarrow \text{LEFT}(i)$   
2  $r \leftarrow \text{RIGHT}(i)$   
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$   
4   then  $\text{largest} \leftarrow l$   
5   else  $\text{largest} \leftarrow i$   
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$   
7   then  $\text{largest} \leftarrow r$   
8 if  $\text{largest} \neq i$   
9   then swap  $A[i] \leftrightarrow A[\text{largest}]$   
10   MAX-HEAPIFY( $A, \text{largest}$ )
```

MAX-HEAPIFY($A, 2$)

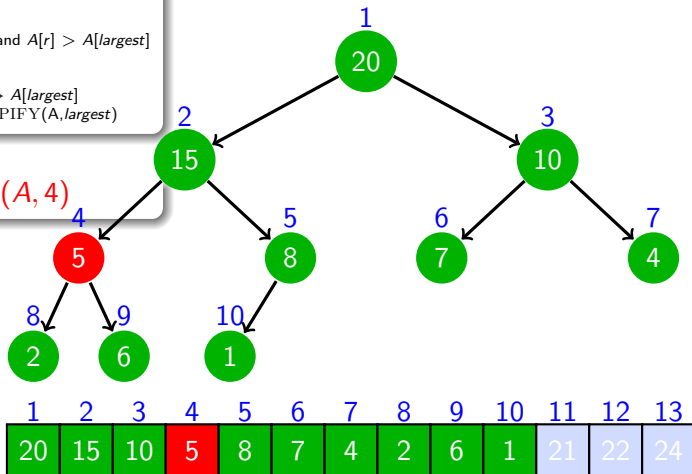


MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1  $l \leftarrow \text{LEFT}(i)$ 
2  $r \leftarrow \text{RIGHT}(i)$ 
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4   then  $\text{largest} \leftarrow l$ 
5   else  $\text{largest} \leftarrow i$ 
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7   then  $\text{largest} \leftarrow r$ 
8 if  $\text{largest} \neq i$ 
9   then swap  $A[i] \leftrightarrow A[\text{largest}]$ 
10    MAX-HEAPIFY( $A, \text{largest}$ )
```

MAX-HEAPIFY($A, 4$)

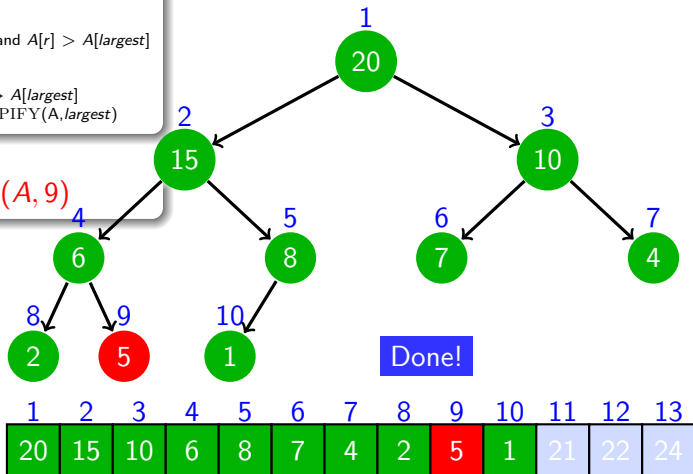


MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1  $l \leftarrow \text{LEFT}(i)$ 
2  $r \leftarrow \text{RIGHT}(i)$ 
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4   then  $\text{largest} \leftarrow l$ 
5   else  $\text{largest} \leftarrow i$ 
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7   then  $\text{largest} \leftarrow r$ 
8 if  $\text{largest} \neq i$ 
9   then swap  $A[i] \leftrightarrow A[\text{largest}]$ 
10   MAX-HEAPIFY( $A, \text{largest}$ )
```

MAX-HEAPIFY($A, 9$)

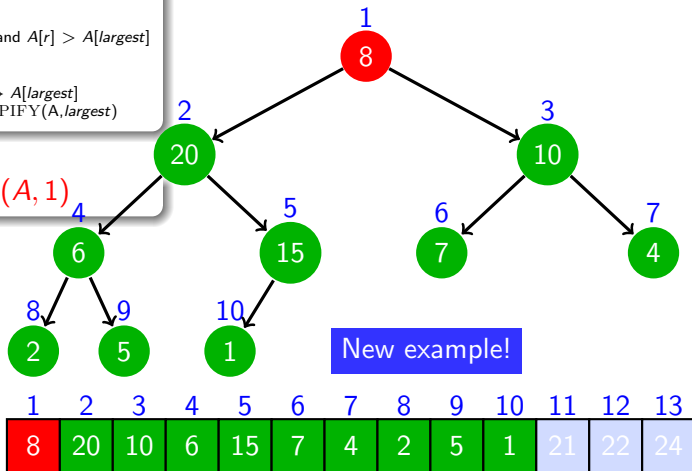


MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1  $l \leftarrow \text{LEFT}(i)$ 
2  $r \leftarrow \text{RIGHT}(i)$ 
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4   then  $\text{largest} \leftarrow l$ 
5   else  $\text{largest} \leftarrow i$ 
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7   then  $\text{largest} \leftarrow r$ 
8 if  $\text{largest} \neq i$ 
9   then swap  $A[i] \leftrightarrow A[\text{largest}]$ 
10   MAX-HEAPIFY( $A, \text{largest}$ )
```

MAX-HEAPIFY($A, 1$)

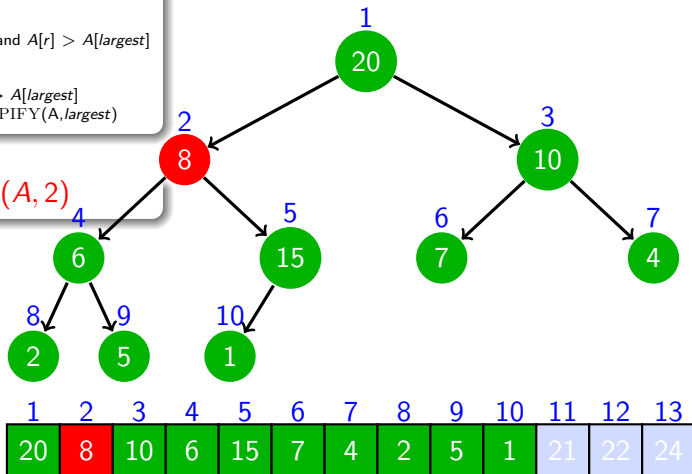


MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1  $l \leftarrow \text{LEFT}(i)$ 
2  $r \leftarrow \text{RIGHT}(i)$ 
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4   then  $\text{largest} \leftarrow l$ 
5   else  $\text{largest} \leftarrow i$ 
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7   then  $\text{largest} \leftarrow r$ 
8 if  $\text{largest} \neq i$ 
9   then swap  $A[i] \leftrightarrow A[\text{largest}]$ 
10    MAX-HEAPIFY( $A, \text{largest}$ )
```

MAX-HEAPIFY($A, 2$)

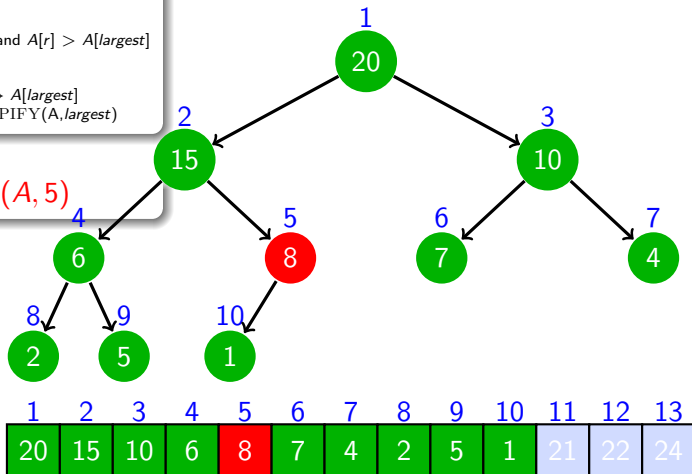


MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1  $l \leftarrow \text{LEFT}(i)$ 
2  $r \leftarrow \text{RIGHT}(i)$ 
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4   then  $\text{largest} \leftarrow l$ 
5   else  $\text{largest} \leftarrow i$ 
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7   then  $\text{largest} \leftarrow r$ 
8 if  $\text{largest} \neq i$ 
9   then swap  $A[i] \leftrightarrow A[\text{largest}]$ 
10    MAX-HEAPIFY( $A, \text{largest}$ )
```

MAX-HEAPIFY($A, 5$)

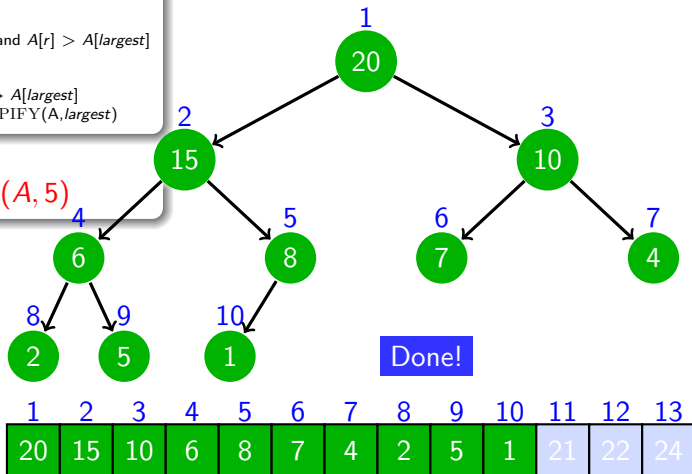


MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1  $l \leftarrow \text{LEFT}(i)$ 
2  $r \leftarrow \text{RIGHT}(i)$ 
3 if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4   then  $\text{largest} \leftarrow l$ 
5   else  $\text{largest} \leftarrow i$ 
6 if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7   then  $\text{largest} \leftarrow r$ 
8 if  $\text{largest} \neq i$ 
9   then swap  $A[i] \leftrightarrow A[\text{largest}]$ 
10    MAX-HEAPIFY( $A, \text{largest}$ )
```

MAX-HEAPIFY($A, 5$)



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 $\text{MAX-HEAPIFY}(A, i)$ should process $O(\log_2(n))$ levels, with constant work at each level

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

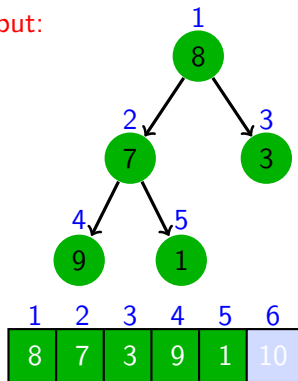
MAX-HEAPIFY(A, i)

```
1   $l \leftarrow \text{LEFT}(i)$ 
2   $r \leftarrow \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4    then  $\text{largest} \leftarrow l$ 
5  else  $\text{largest} \leftarrow i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7    then  $\text{largest} \leftarrow r$ 
8  if  $\text{largest} \neq i$ 
9    then swap  $A[i] \leftrightarrow A[\text{largest}]$ 
10     MAX-HEAPIFY( $A, \text{largest}$ )
```

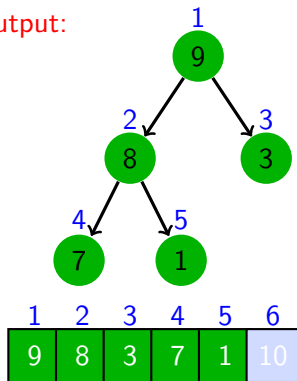
BUILD-MAX-HEAP

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

Input:



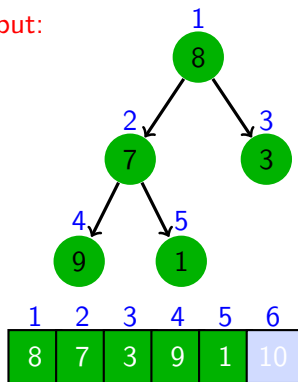
Output:



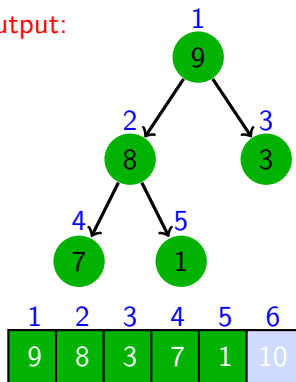
BUILD-MAX-HEAP

- **Input:** An array A
- **Output:** A max-heap from A
- **Idea:** Use MAX-HEAPIFY in a bottom-up manner

Input:



Output:



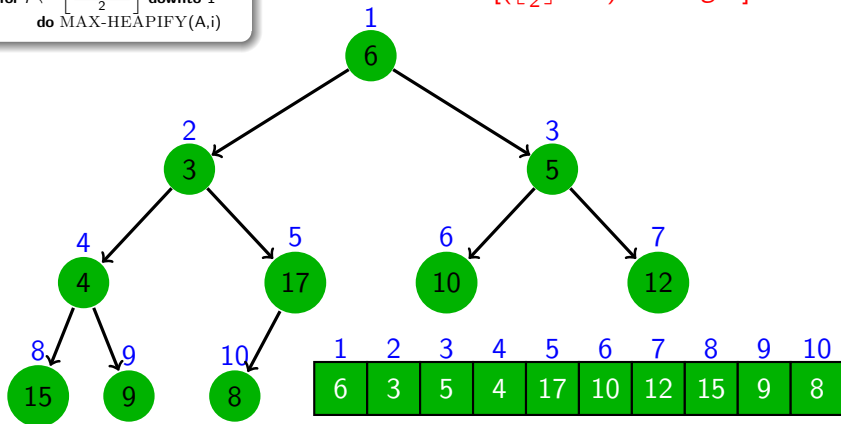
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 5$$



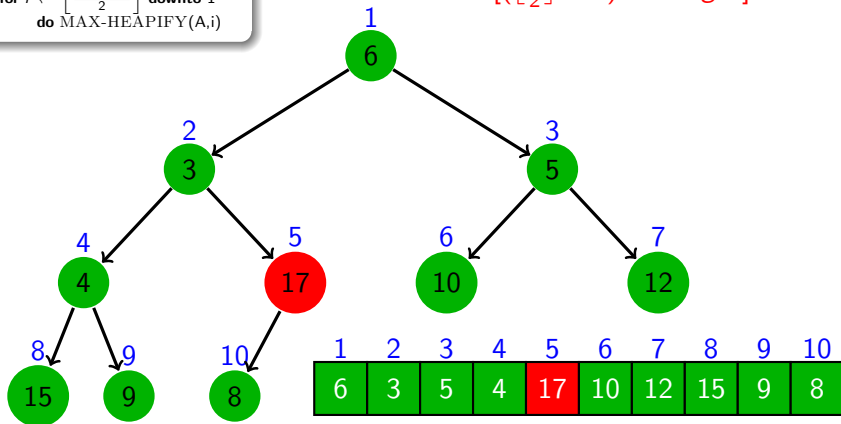
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 5$$



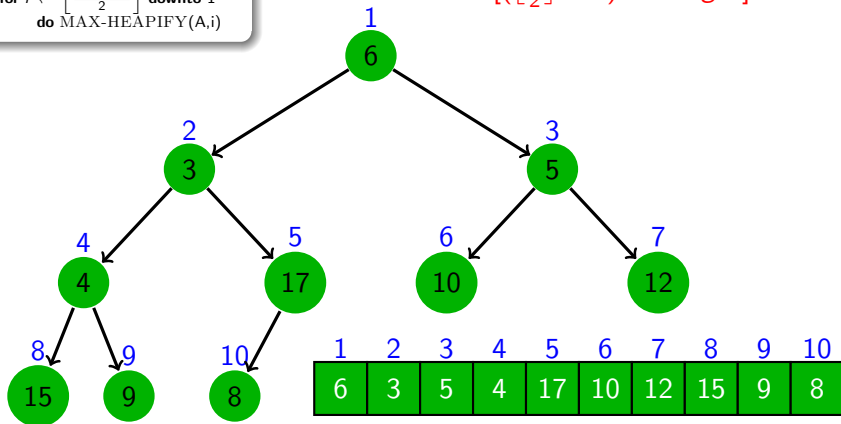
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 4$$



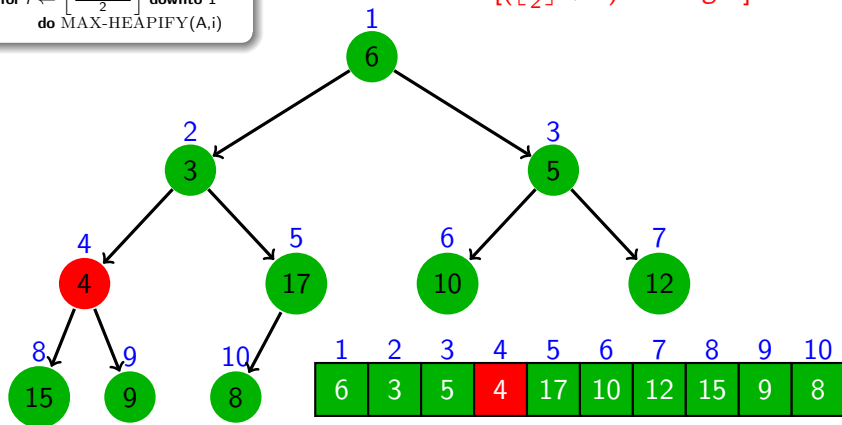
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 4$$



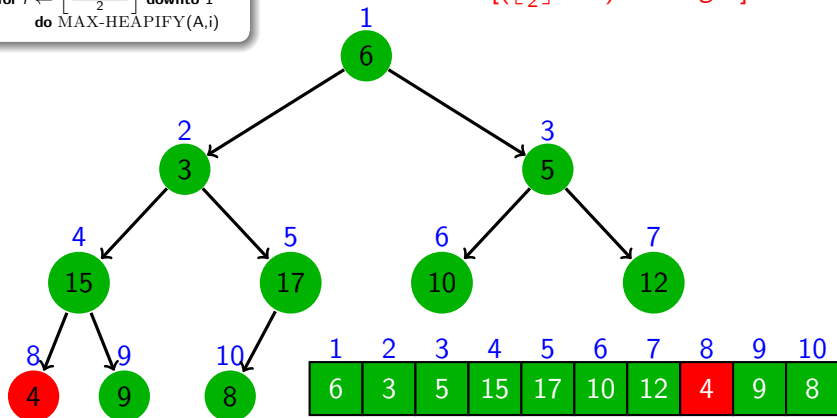
BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)

```
1  A.heap-size ← A.length
2  for i ←  $\lfloor \frac{A.length}{2} \rfloor$  downto 1
3      do MAX-HEAPIFY(A,i)
```

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

$$i = 4$$



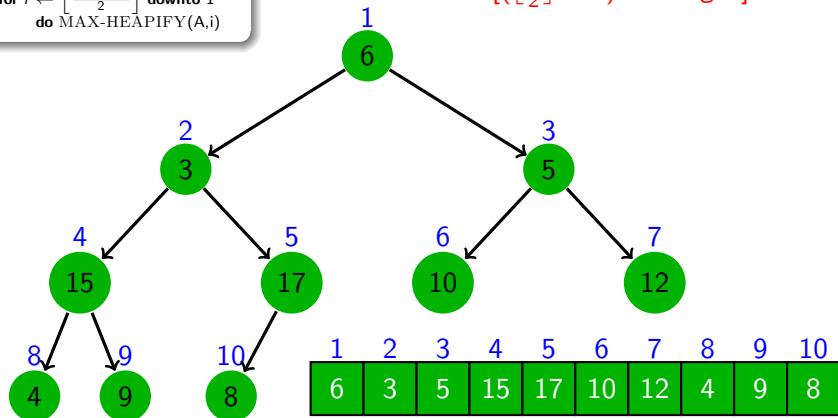
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 3$$



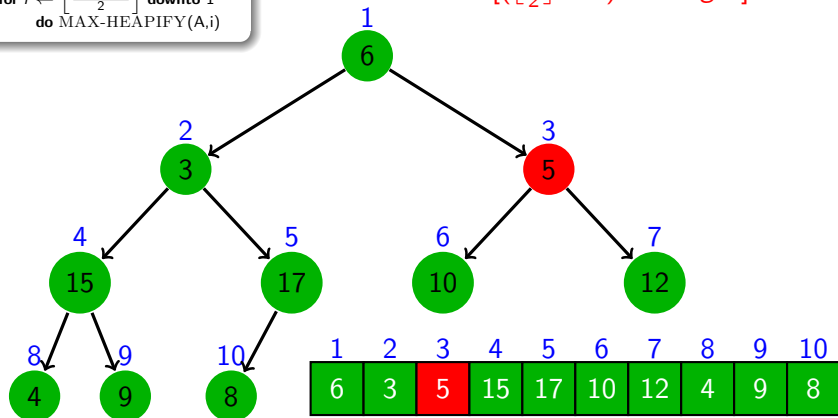
BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)

```
1  A.heap-size ← A.length
2  for i ←  $\lfloor \frac{A.length}{2} \rfloor$  downto 1
3      do MAX-HEAPIFY(A,i)
```

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

$i = 3$



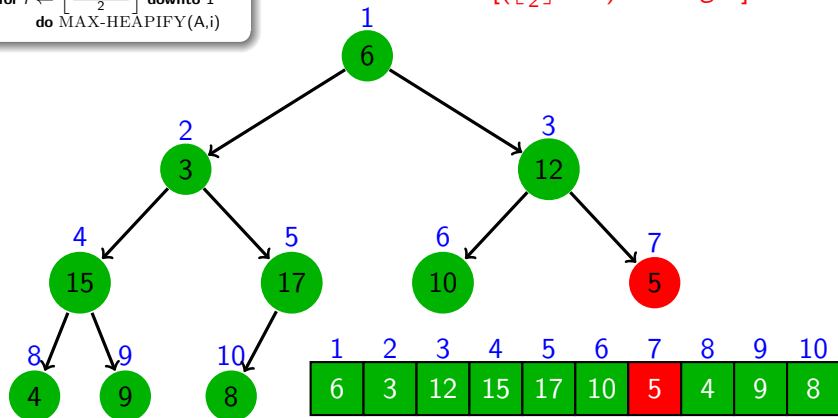
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 3$$



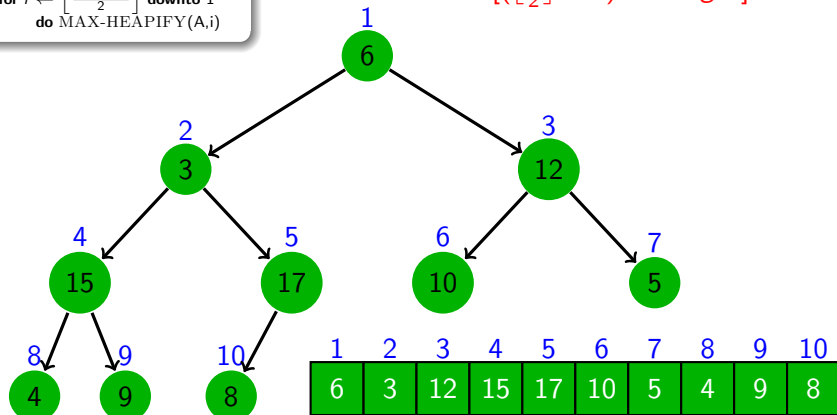
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 2$$



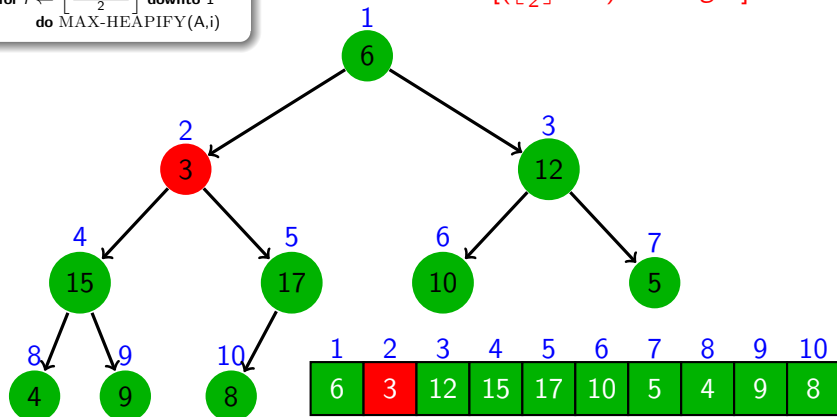
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 2$$



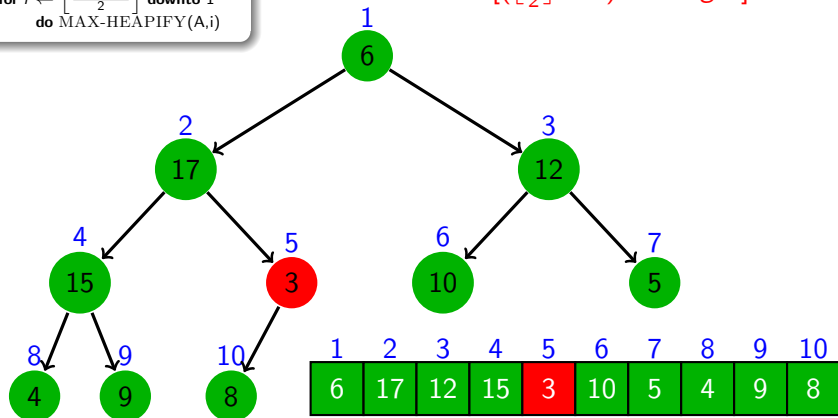
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

```
1 A.heap-size ← A.length  
2 for i ←  $\lfloor \frac{A.length}{2} \rfloor$  downto 1  
3   do MAX-HEAPIFY(A, i)
```

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

$$i = 2$$



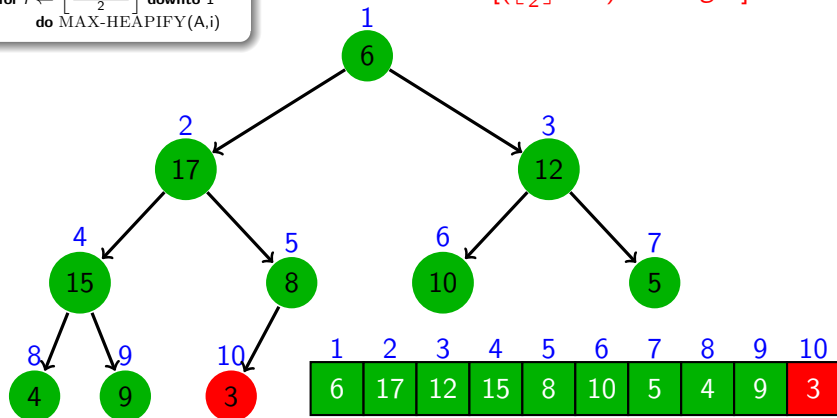
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 2$$



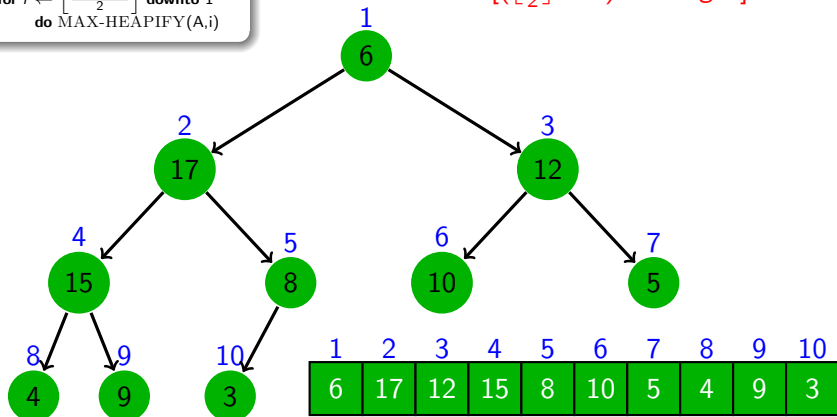
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$i = 1$



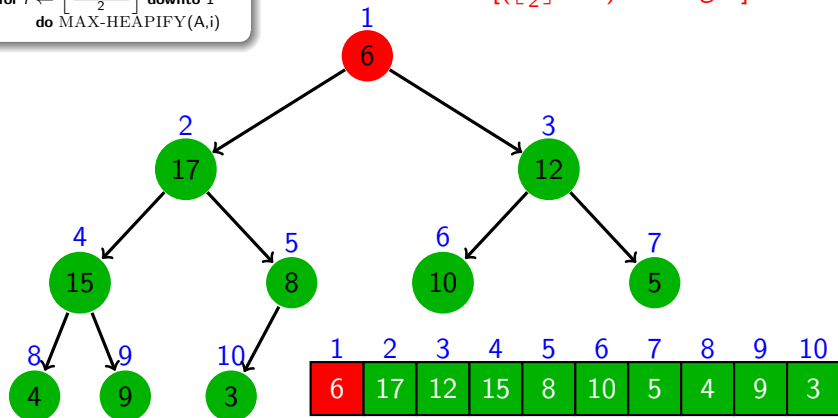
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 1$$



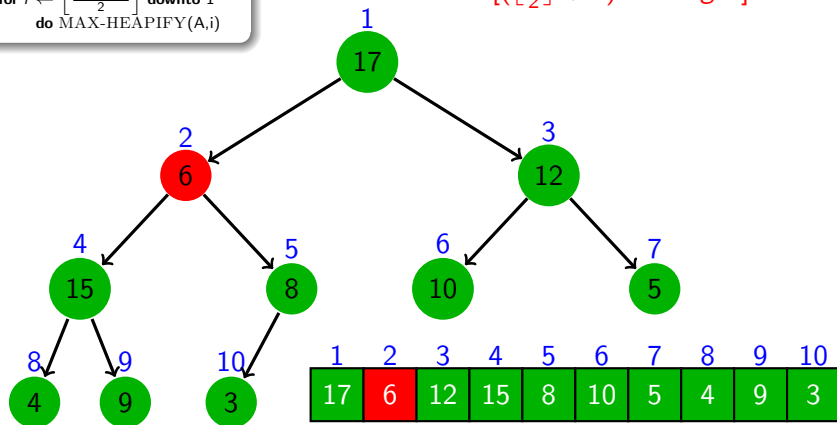
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

```
1 A.heap-size ← A.length
2 for i ←  $\lfloor \frac{A.length}{2} \rfloor$  downto 1
3   do MAX-HEAPIFY(A, i)
```

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

$$i = 1$$



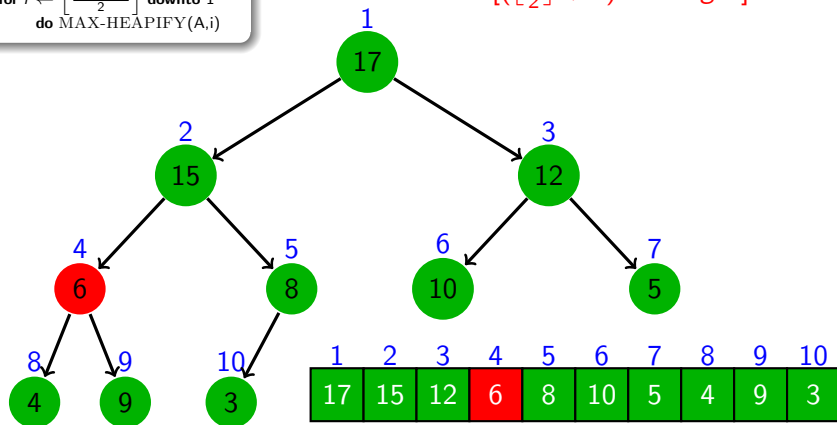
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

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

$$i = 1$$



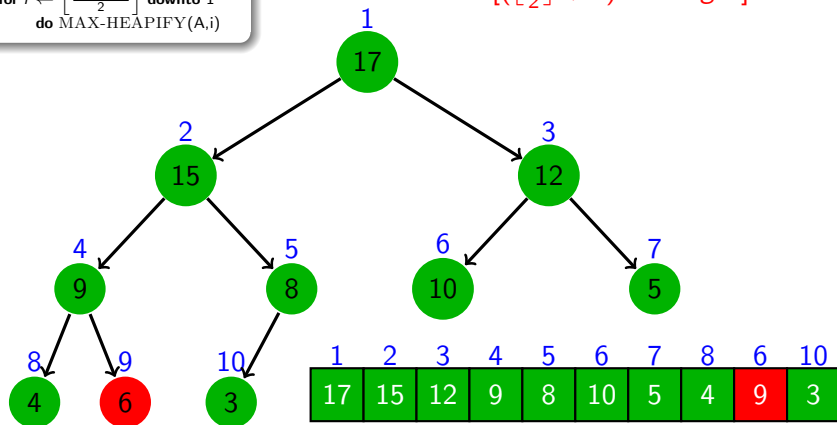
BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

```
1 A.heap-size ← A.length  
2 for i ←  $\lfloor \frac{A.length}{2} \rfloor$  downto 1  
3   do MAX-HEAPIFY(A, i)
```

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

$$i = 1$$

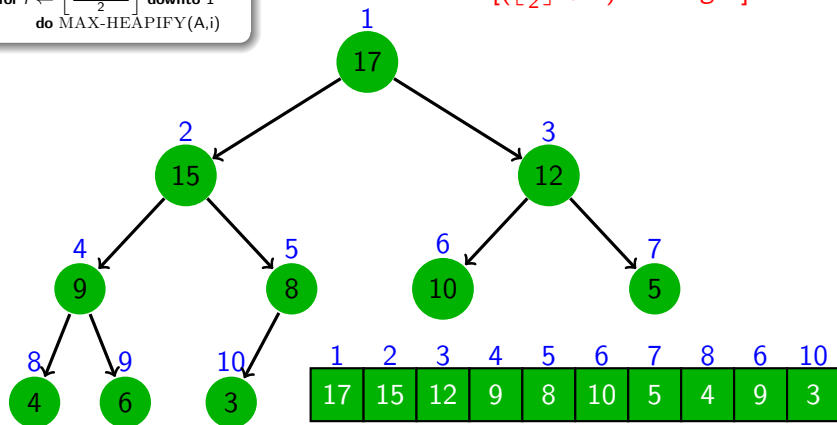


BUILD-MAX-HEAP

BUILD-MAX-HEAP(*A*)

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

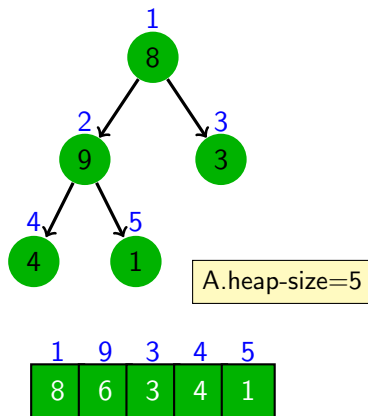
- Tight bound:
 - We have at most 2^i nodes at depth i (height $h - i$)
 - We call MAX-HEAPIFY for each node of depth $i \Rightarrow O(h - i)$
 - The runtime of BUILD-MAX-HEAP is:
$$\begin{aligned} T(n) &= \sum_{i=0}^{h-1} 2^i O(h - i) = O(\sum_{i=0}^{h-1} 2^i (h - i)) \\ &= O(\sum_{j=1}^h \sum_{i=0}^{h-j} 2^i) \\ &= O(2^{h+1} - h - 2) \\ &= O(n) \end{aligned}$$

HeapSort: Principle

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

HeapSort

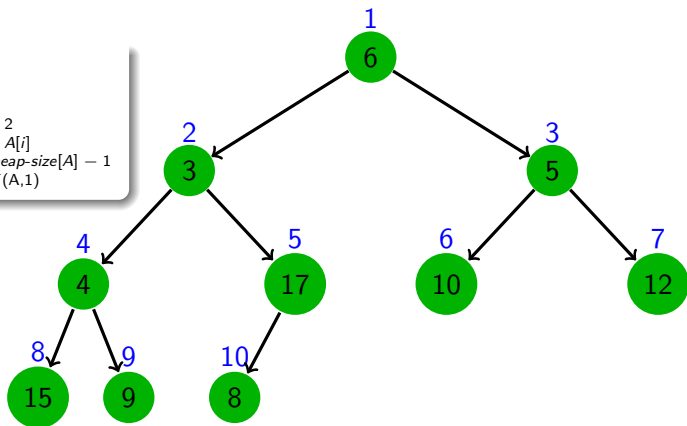
- Construct a max-heap from A (call **BUILD-MAX-HEAP**(A))
- Repeat until the heap is of size one:
 - Swap the values of the root and the right-most leaf of the heap (i.e., Swap $A[1]$ and $A[A.heap-size]$)
 - Discard the right-most leaf from the heap by decreasing the heap size
 - Restore the max-heap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

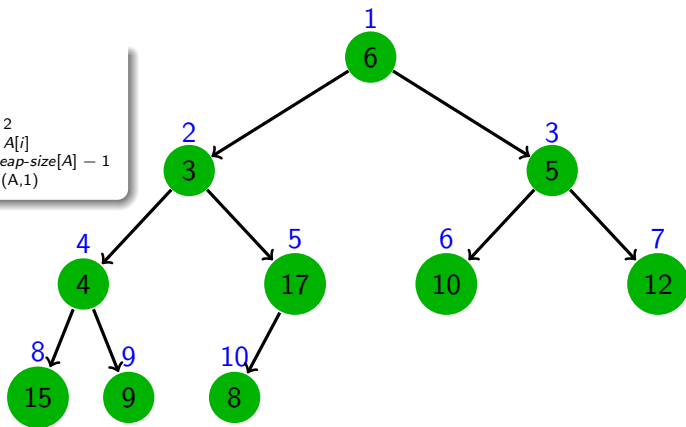


1	2	3	4	5	6	7	8	9	10
6	3	5	4	17	10	12	15	9	8

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)
```



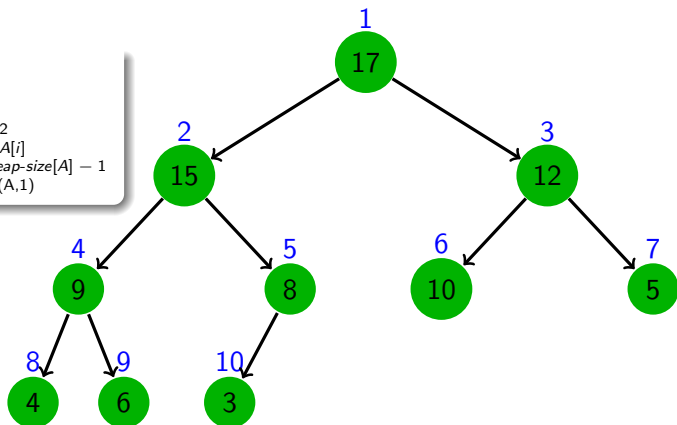
BUILD-MAX-HEAP(*A*)

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)
```

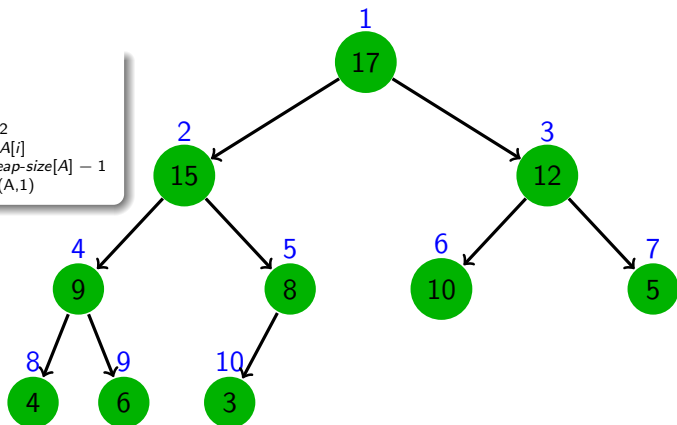


1	2	3	4	5	6	7	8	9	10
17	15	12	9	8	10	5	4	6	3

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)
```



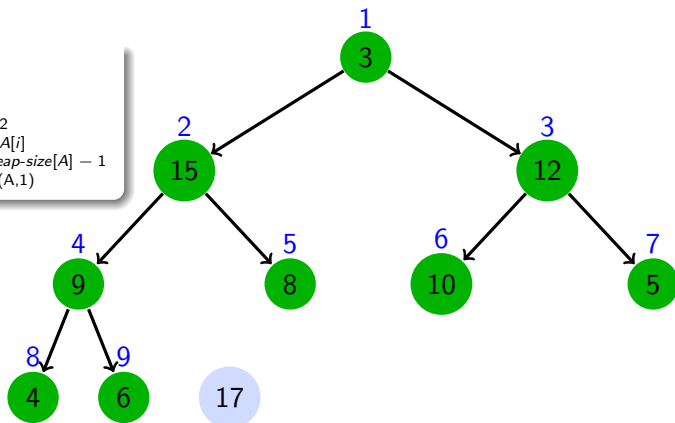
$i = 10$

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)
```



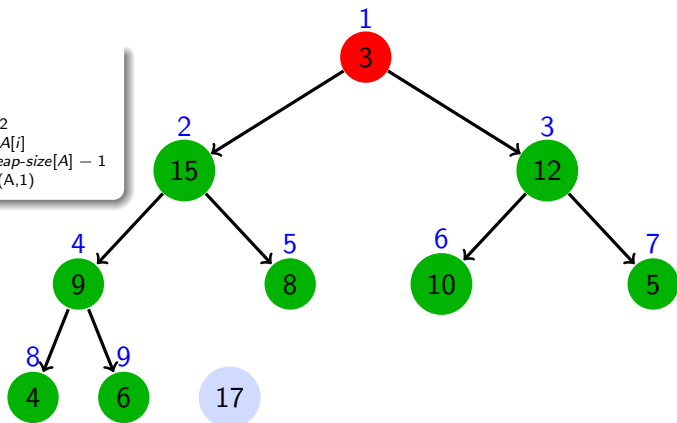
$i = 10$

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)
```



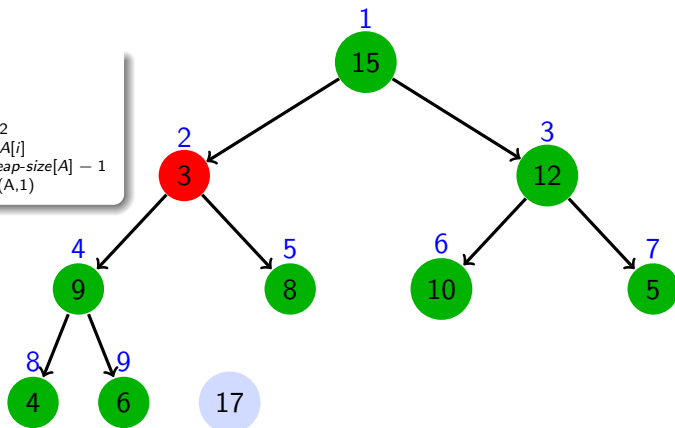
$i = 10$

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)
```



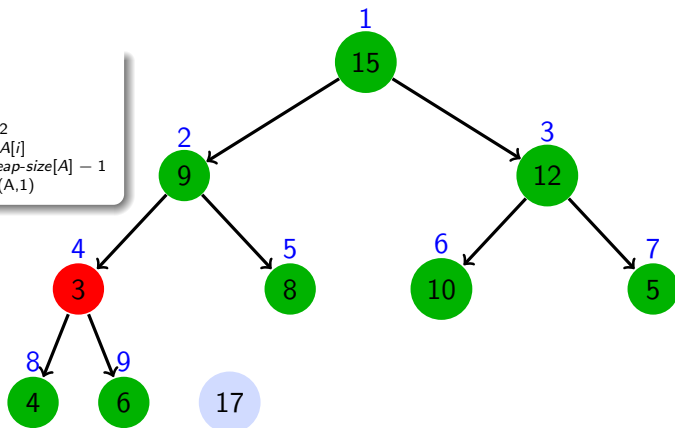
$i = 10$

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)
```



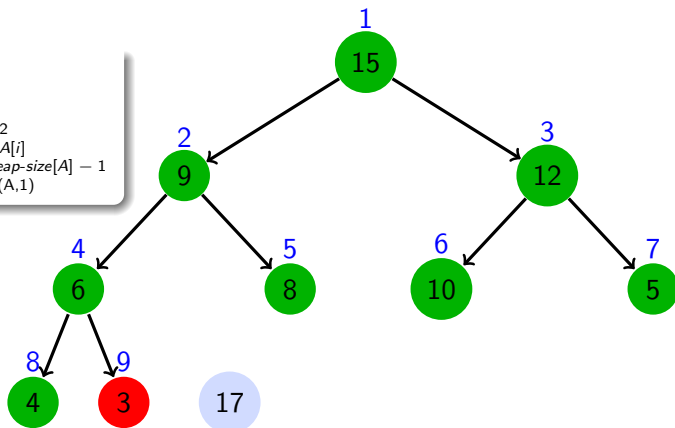
$i = 10$

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)
```



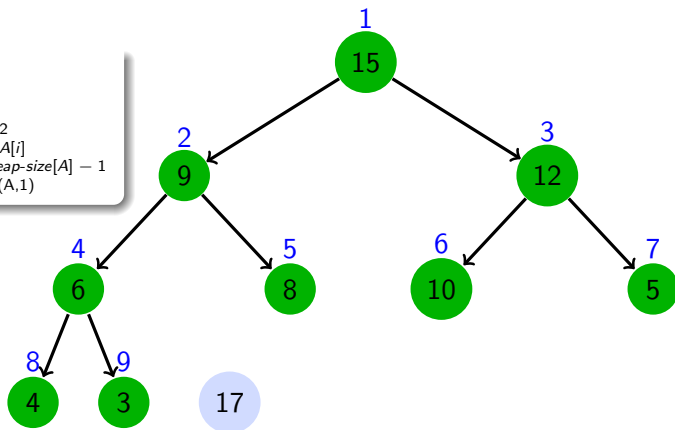
$i = 10$

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)
```



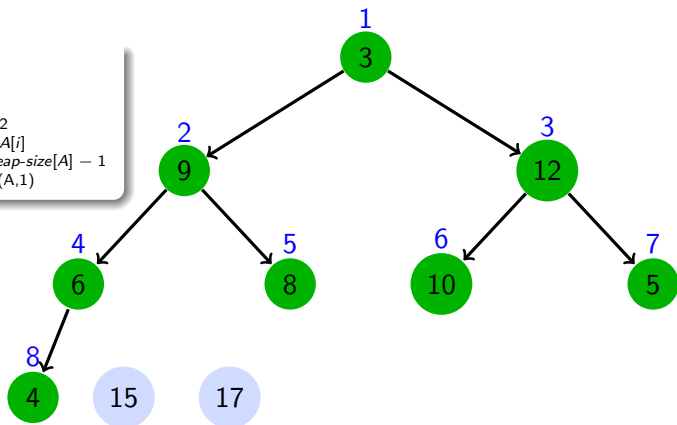
$i = 9$

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)
```



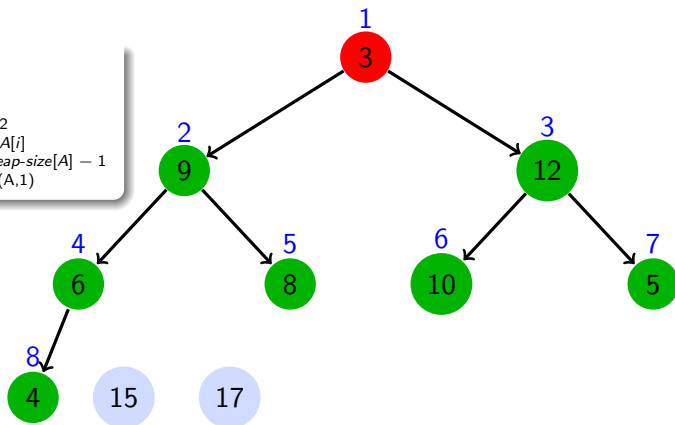
$i = 9$

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)
```



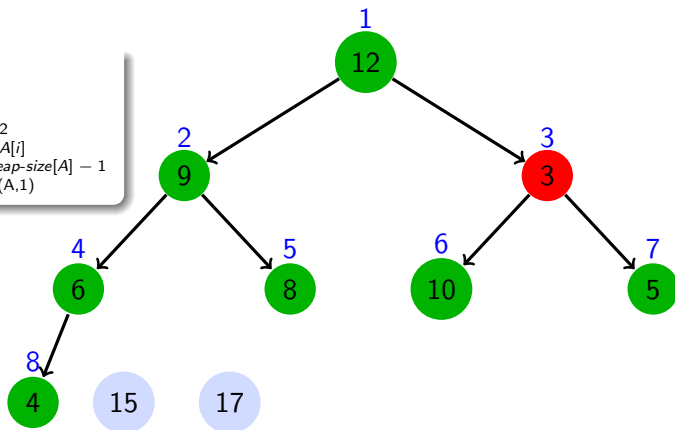
$i = 9$

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)
```



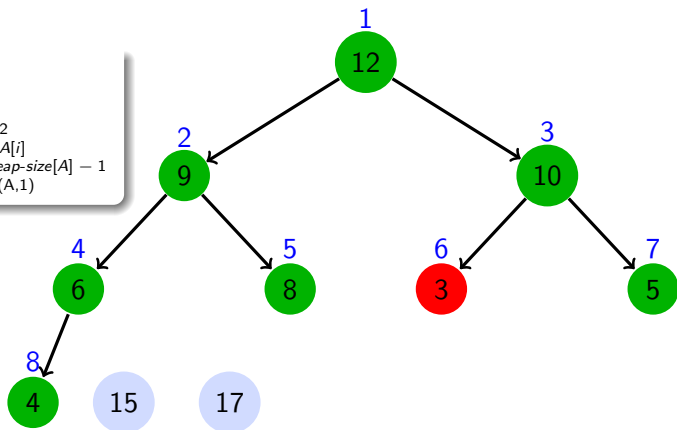
$i = 9$

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)
```



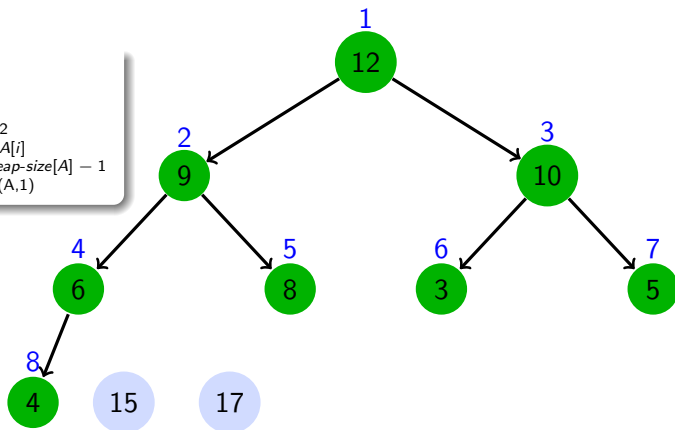
$i = 9$

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)
```



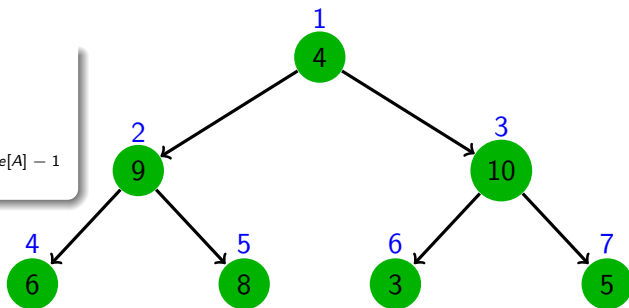
$i = 8$

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)
```



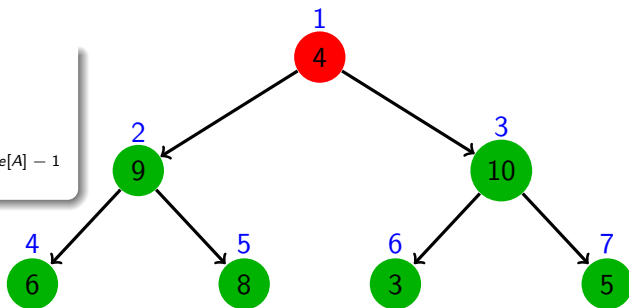
$i = 8$

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)
```



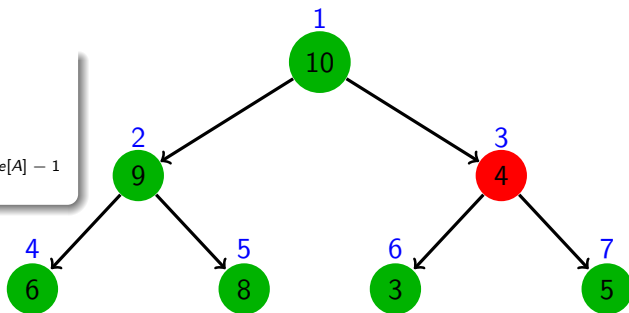
$i = 8$

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)
```



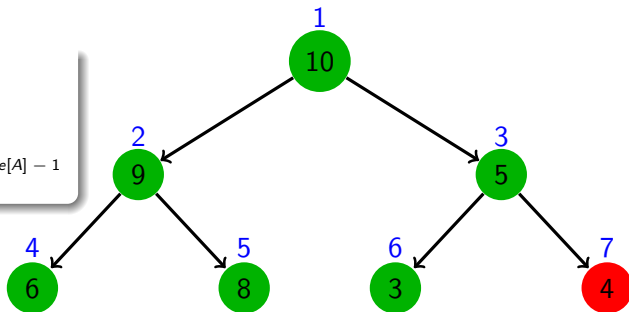
$i = 8$

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)
```



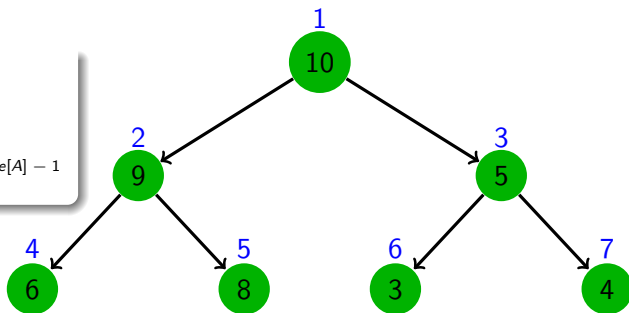
$i = 8$

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)
```



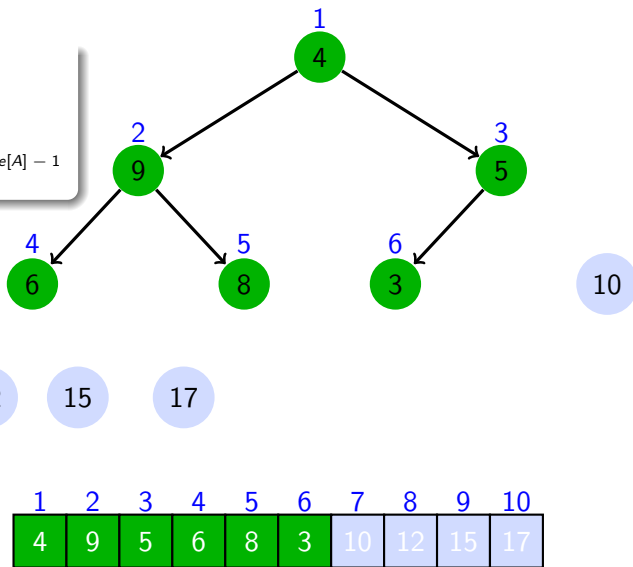
$i = 7$

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*)

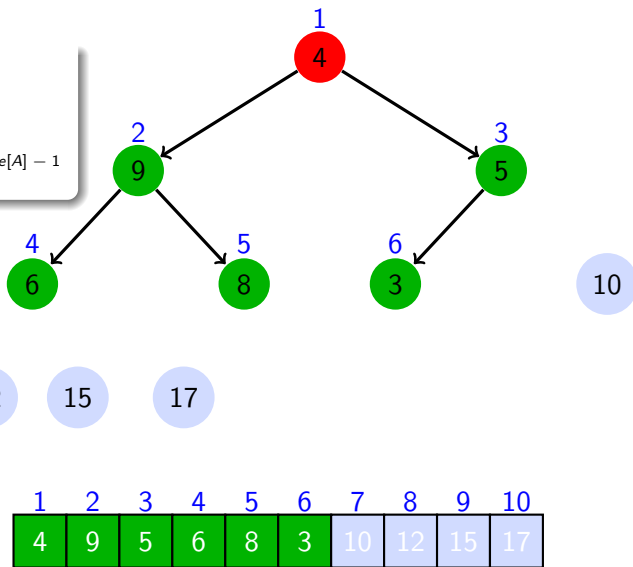
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

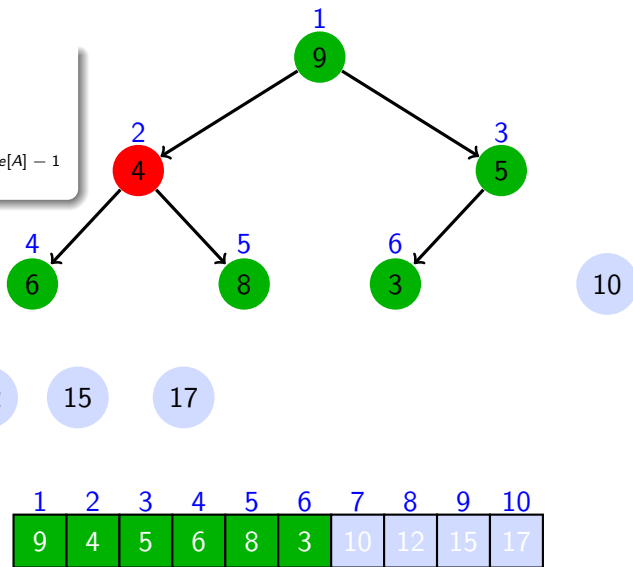
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4     heap-size[A] ← heap-size[A] - 1  
5     MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

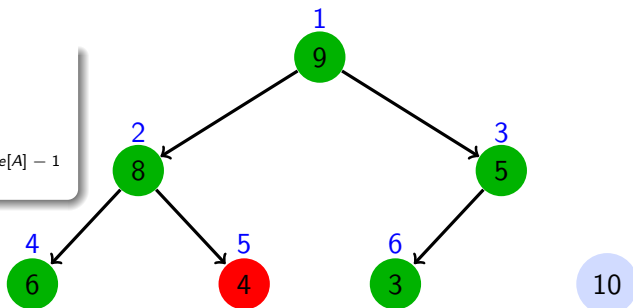
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



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)
```



12

15

17

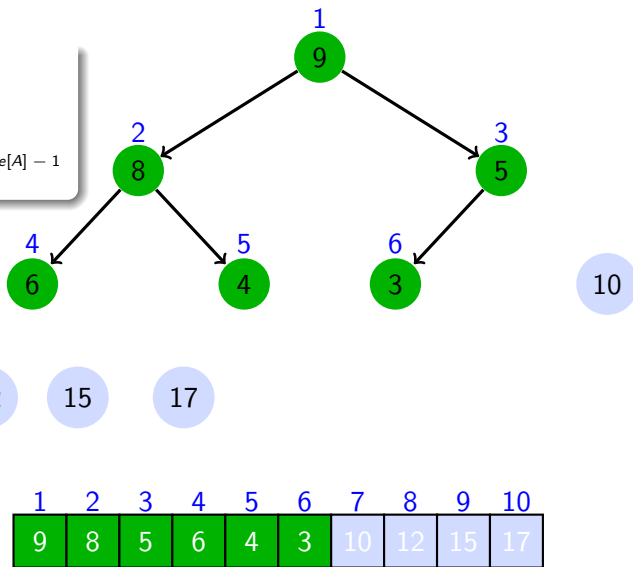
$i = 7$

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*)

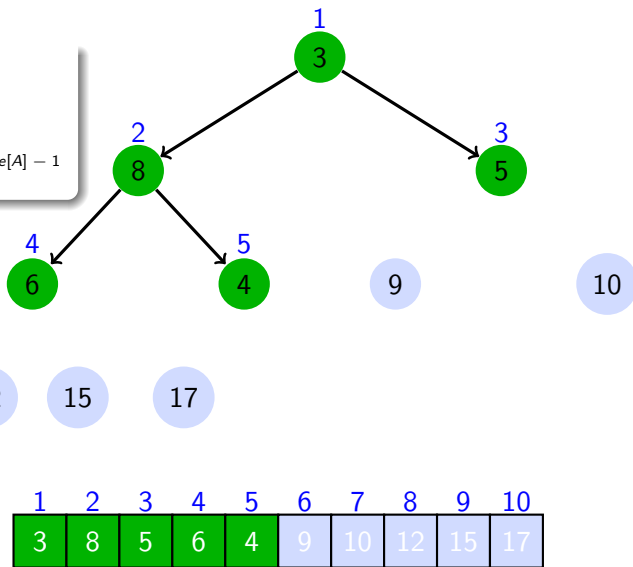
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

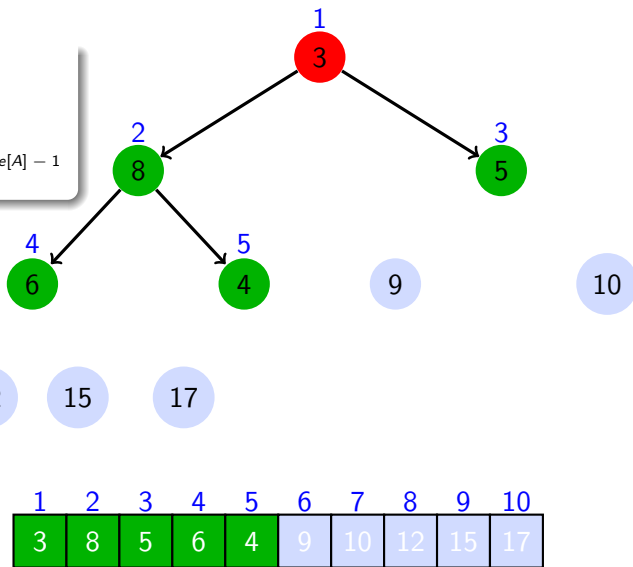
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4     heap-size[A] ← heap-size[A] - 1  
5     MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

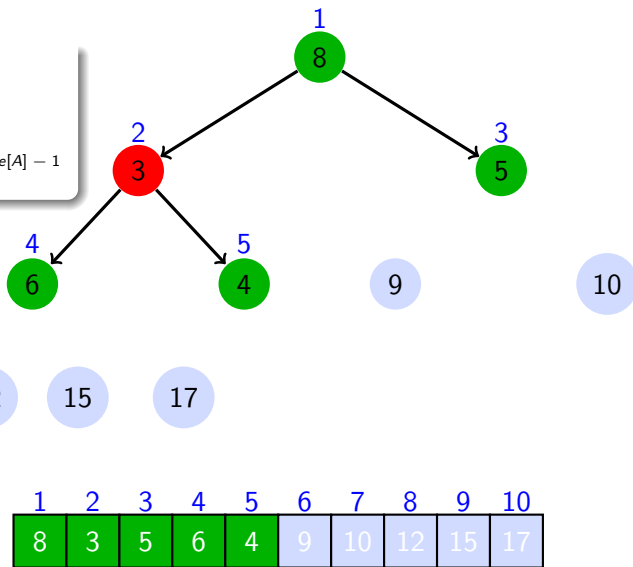
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4   heap-size[A] ← heap-size[A] - 1  
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

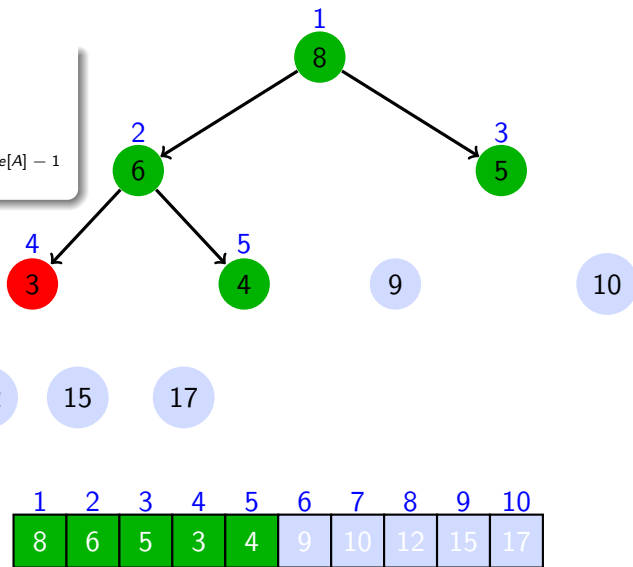
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4   heap-size[A] ← heap-size[A] - 1  
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(A)

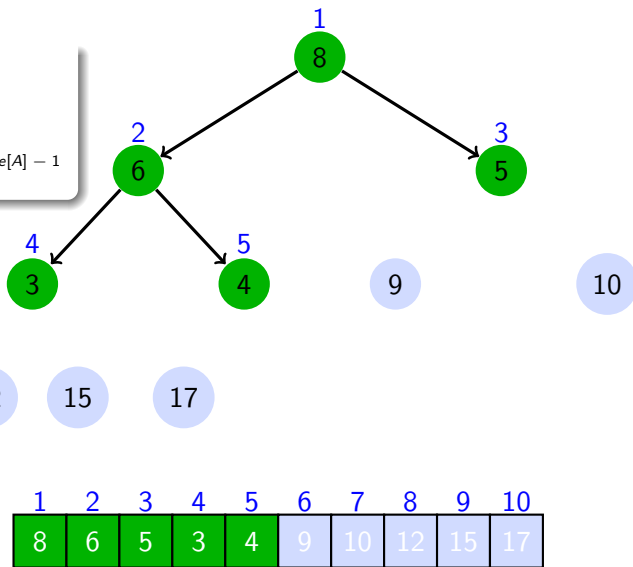
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

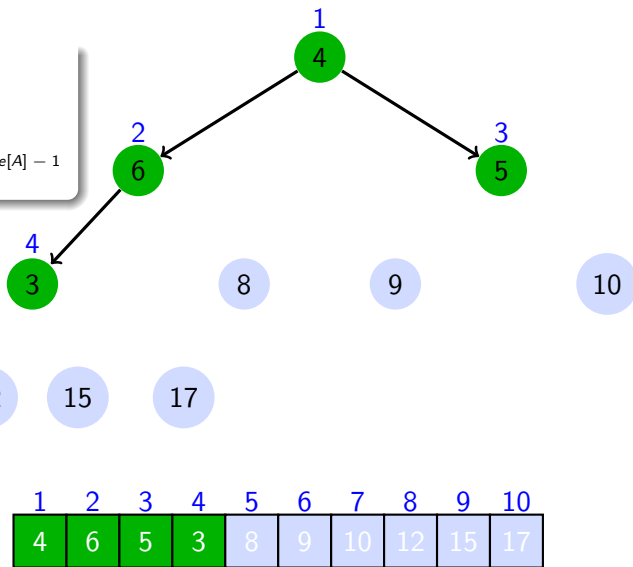
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4     heap-size[A] ← heap-size[A] - 1  
5     MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

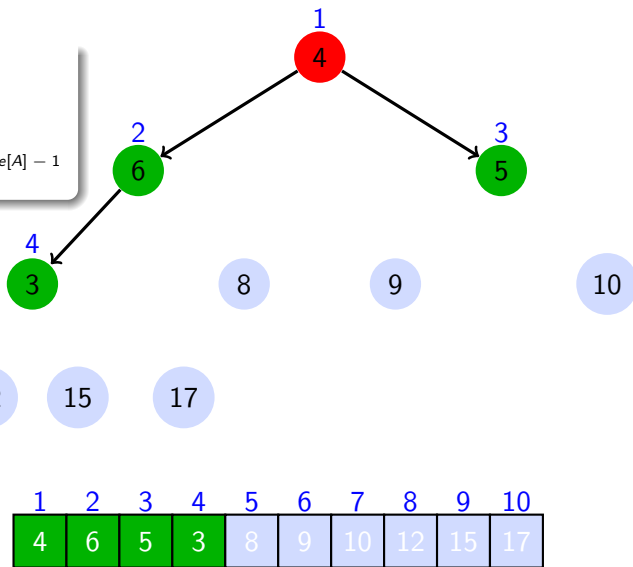
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

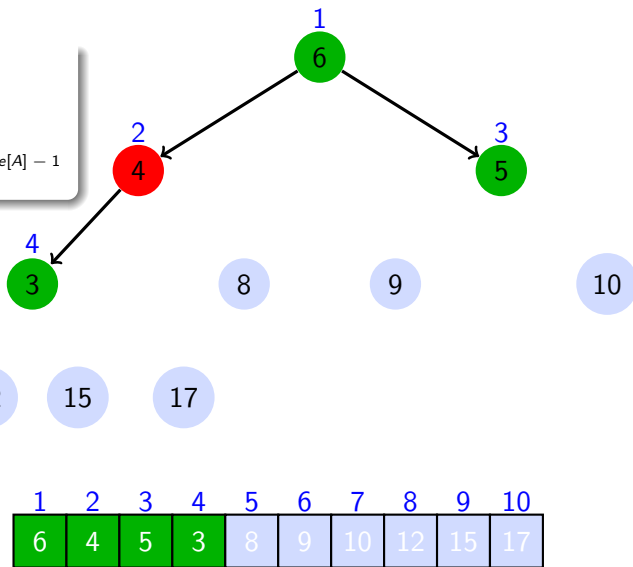
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

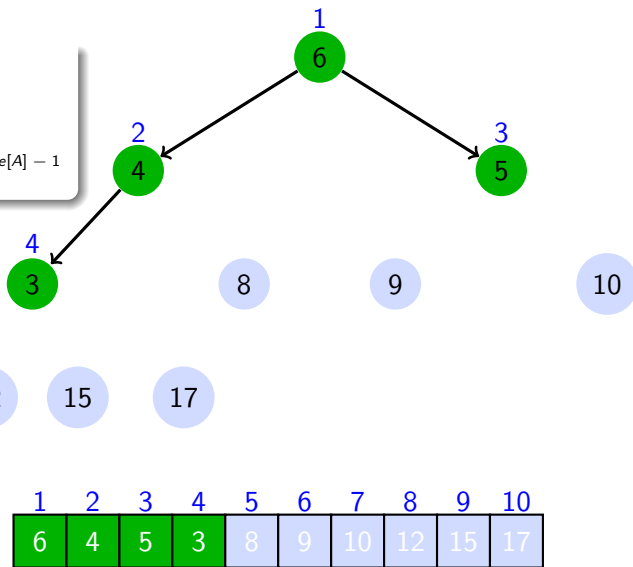
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

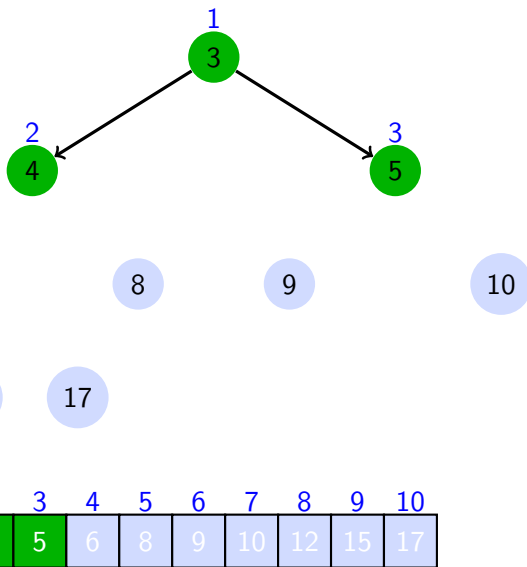
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4   heap-size[A] ← heap-size[A] - 1  
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

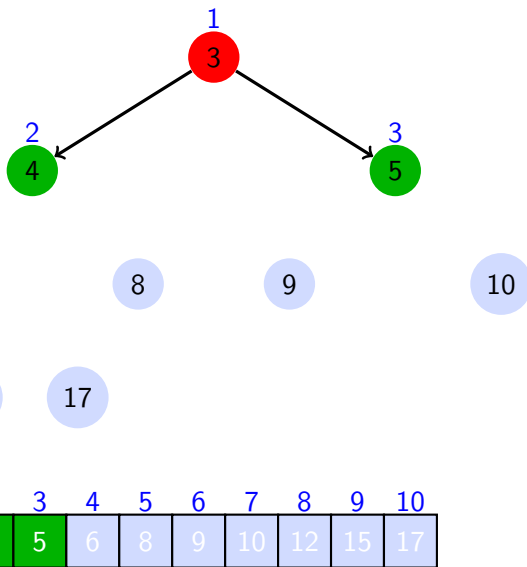
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

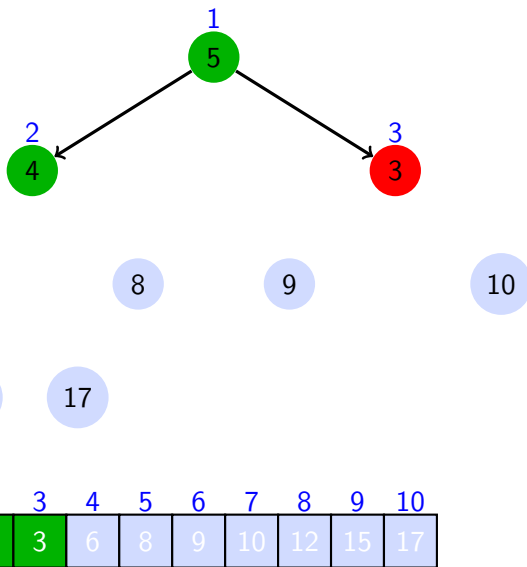
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4   heap-size[A] ← heap-size[A] - 1  
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

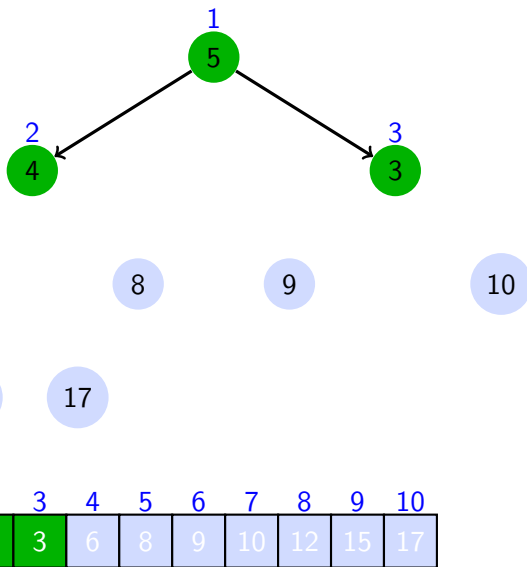
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4   heap-size[A] ← heap-size[A] - 1  
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

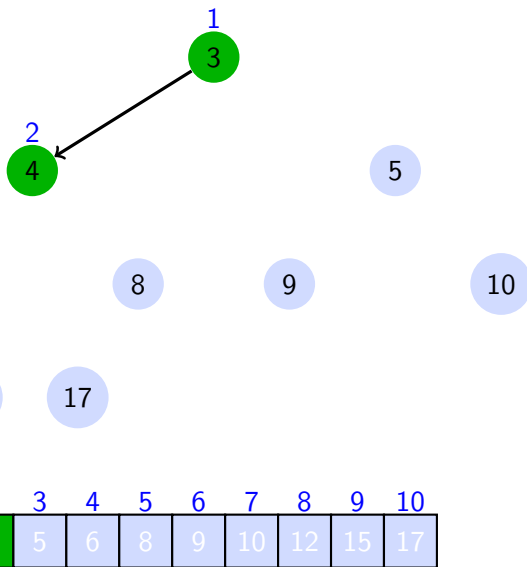
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4     heap-size[A] ← heap-size[A] - 1  
5     MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

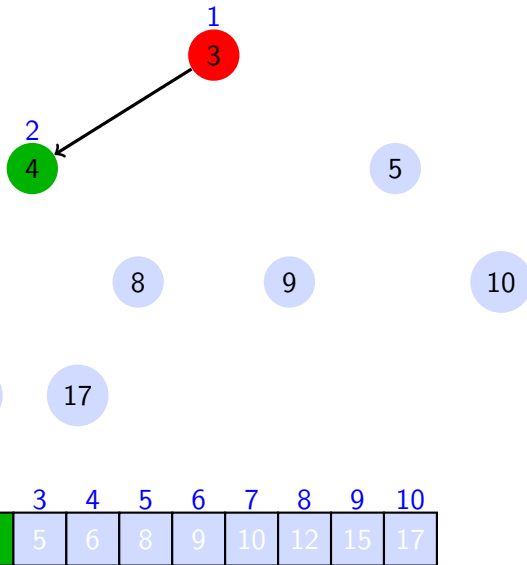
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

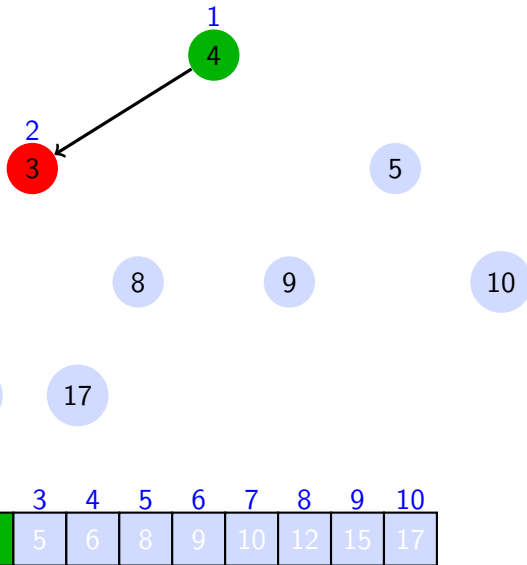
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4     heap-size[A] ← heap-size[A] - 1  
5     MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

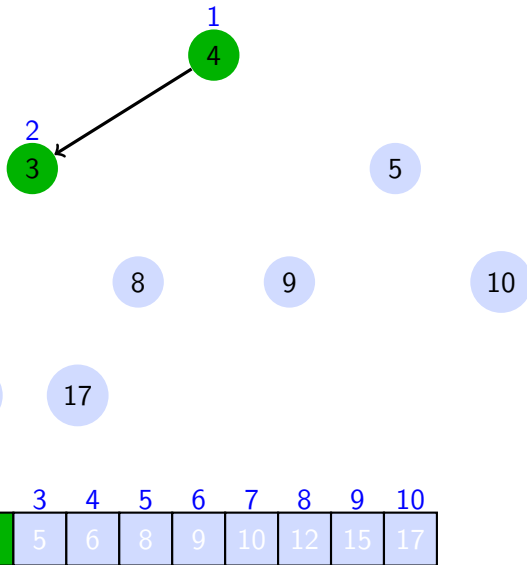
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4     heap-size[A] ← heap-size[A] - 1  
5     MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

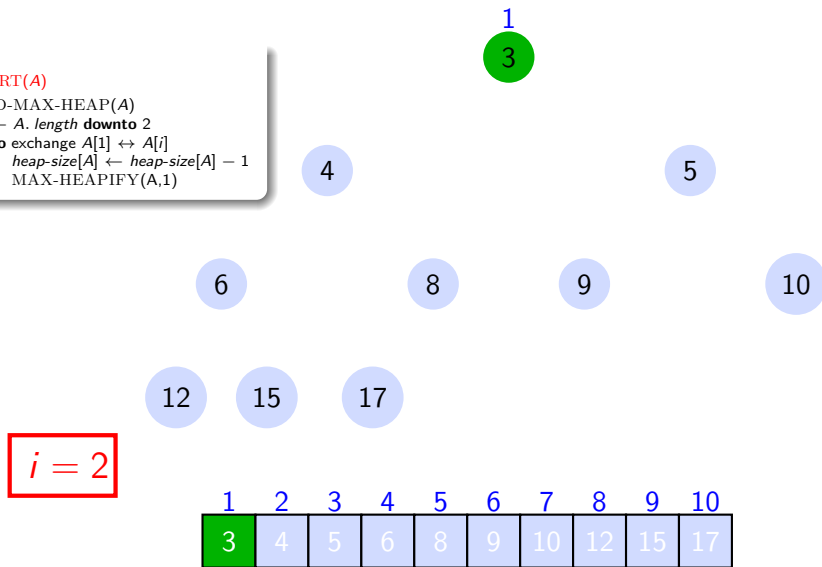
```
1 BUILD-MAX-HEAP(A)
2 for i ← A.length downto 2
3   do exchange A[1] ↔ A[i]
4   heap-size[A] ← heap-size[A] - 1
5   MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

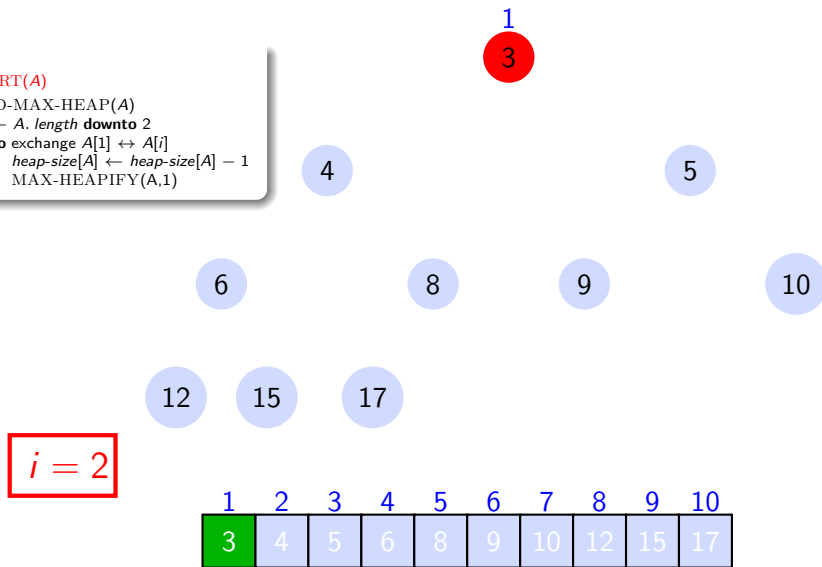
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4     heap-size[A] ← heap-size[A] - 1  
5     MAX-HEAPIFY(A,1)
```



HEAP-SORT

HEAP-SORT(*A*)

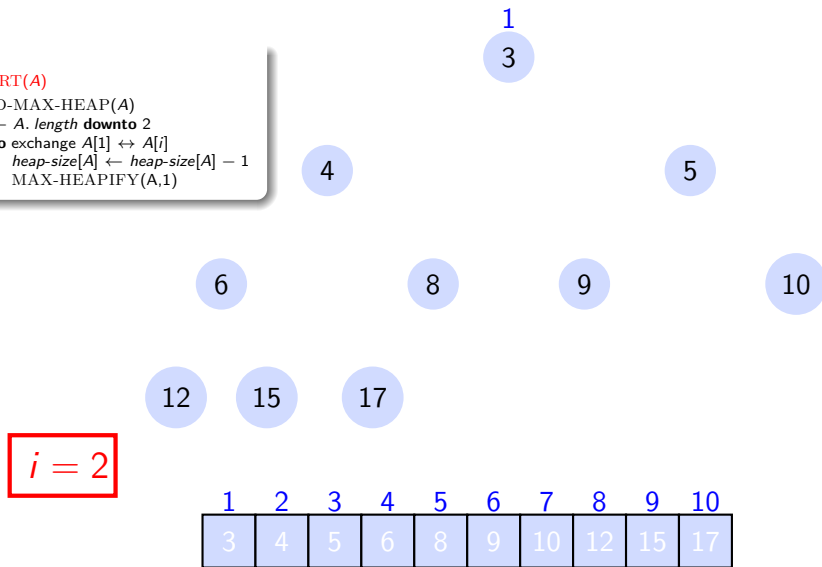
```
1 BUILD-MAX-HEAP(A)  
2 for i ← A.length downto 2  
3   do exchange A[1] ↔ A[i]  
4     heap-size[A] ← heap-size[A] - 1  
5     MAX-HEAPIFY(A,1)
```



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)
```



HEAP-SORT: Runtime

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

HEAP-SORT(A)

```
1 BUILD-MAX-HEAP( $A$ )
2 for  $i \leftarrow A.length$  downto 2
3     do exchange  $A[1] \leftrightarrow A[i]$ 
4          $heap-size[A] \leftarrow heap-size[A] - 1$ 
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 an **abstract data type** 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, x, k): increases the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

Priority Queues

- A **priority queue** is an **abstract data type** 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, x, k): increases the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

Max-Heaps efficiently implement priority queues.

Priority Queues

- A **priority queue** is an **abstract data type** 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, x, k): increases the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

Max-Heaps efficiently implement priority queues.

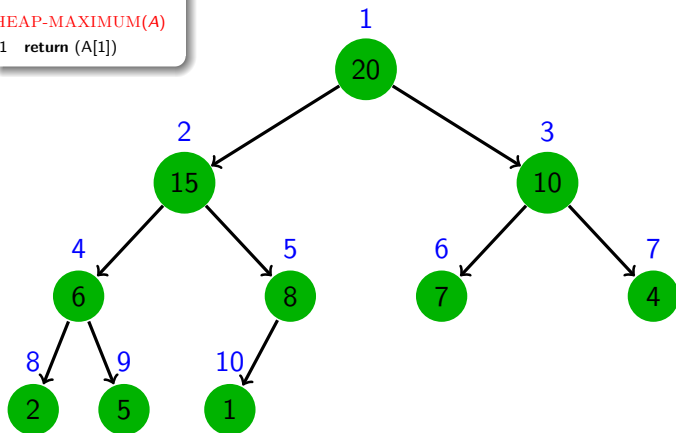
(In the following we depict only the keys, but is implied that each key is part of some element.)

HEAP-MAXIMUM

HEAP-MAXIMUM(*A*): returns the element of *A* with the largest key.

HEAP-MAXIMUM(*A*)

1 **return** (*A*[1])

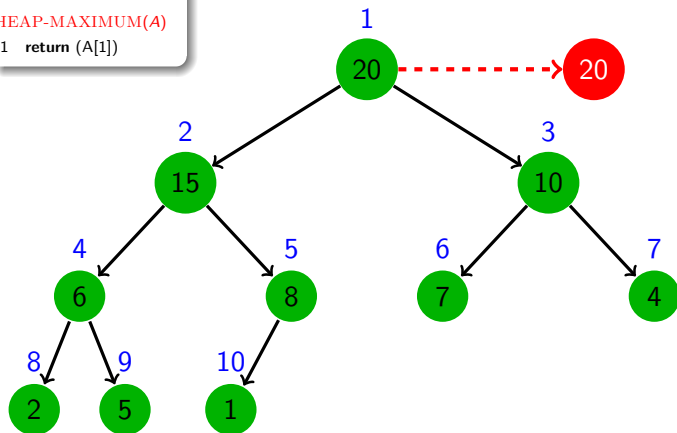


HEAP-MAXIMUM

HEAP-MAXIMUM(*A*): returns the element of *A* with the largest key.

HEAP-MAXIMUM(*A*)

1 **return** (*A*[1])



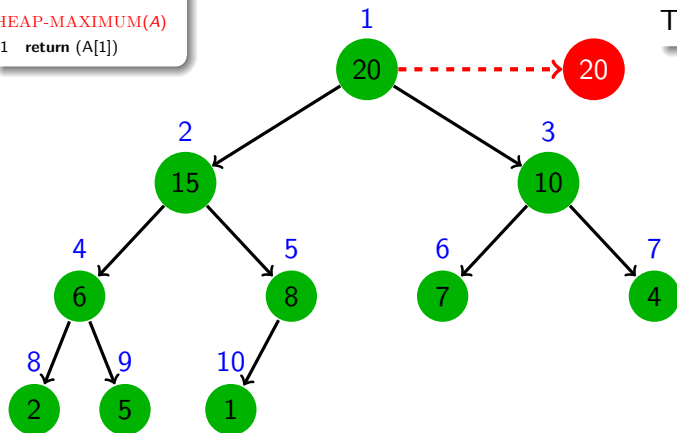
HEAP-MAXIMUM

HEAP-MAXIMUM(*A*): returns the element of *A* with the largest key.

HEAP-MAXIMUM(*A*)

```
1 return (A[1])
```

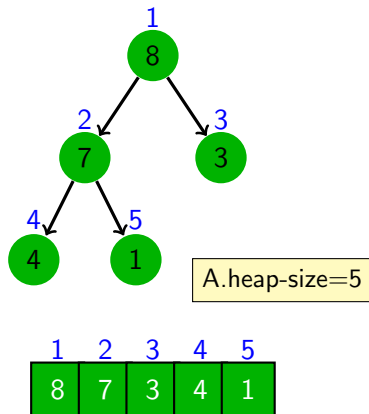
Time: $\Theta(1)$



HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of A with the largest key.

HEAP-EXTRACT-MAX

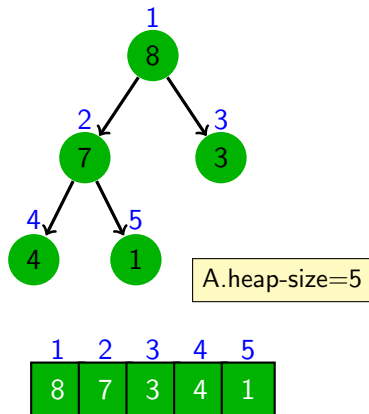


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of **A** with the largest key.

HEAP-EXTRACT-MAX

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

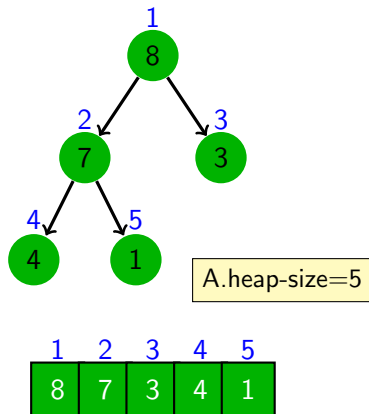


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of **A** with the largest key.

HEAP-EXTRACT-MAX

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

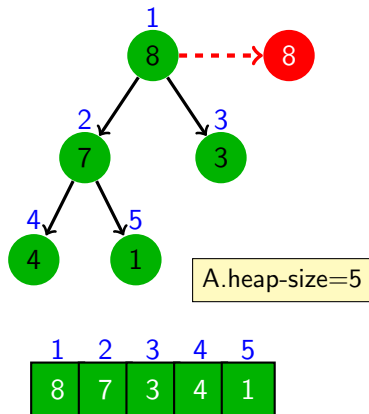


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of *A* with the largest key.

HEAP-EXTRACT-MAX

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

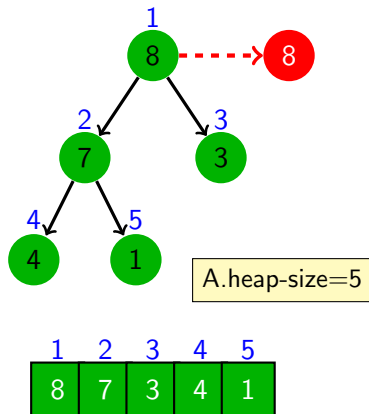


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of *A* with the largest key.

HEAP-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.

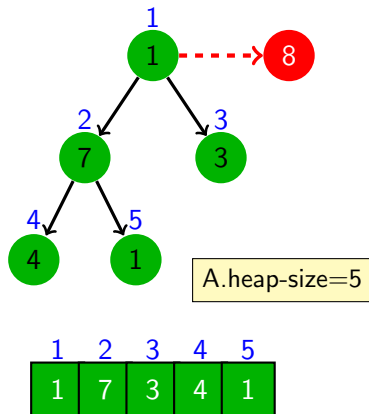


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of *A* with the largest key.

HEAP-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.

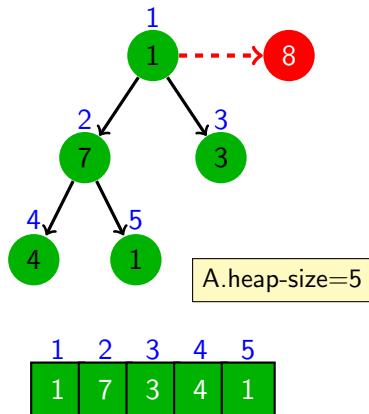


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of **A** with the largest key.

HEAP-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.

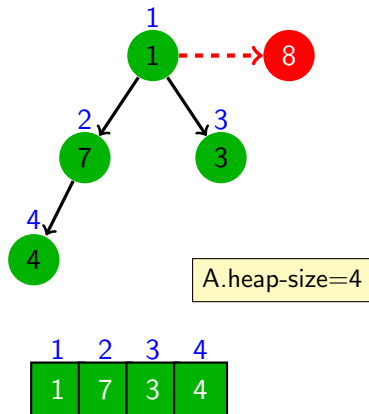


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of *A* with the largest key.

HEAP-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.

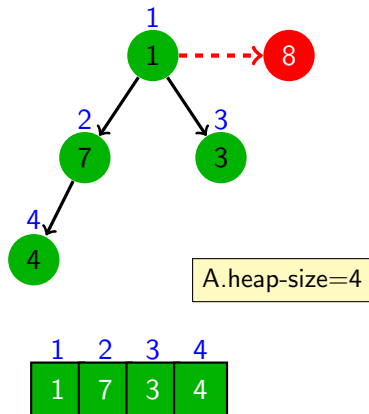


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of *A* with the largest key.

HEAP-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.

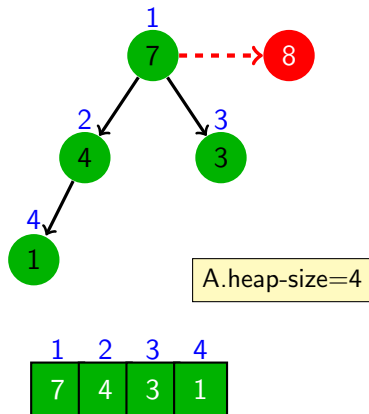


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of *A* with the largest key.

HEAP-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.

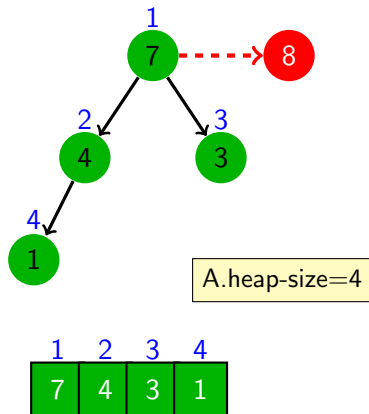


HEAP-EXTRACT-MAX: Principle

Problem: returns and removes the element of *A* with the largest key.

HEAP-EXTRACT-MAX

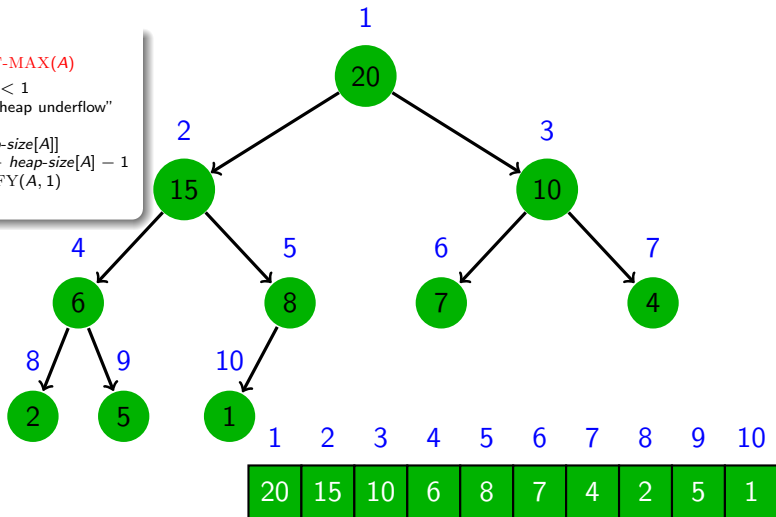
- Make sure that the max-heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node of the heap the new root.
- Discard the last node of the heap.
- Restore the max-heap property.
- Return the copy of the maximum element.



HEAP-EXTRACT-MAX

HEAP-EXTRACT-MAX(*A*)

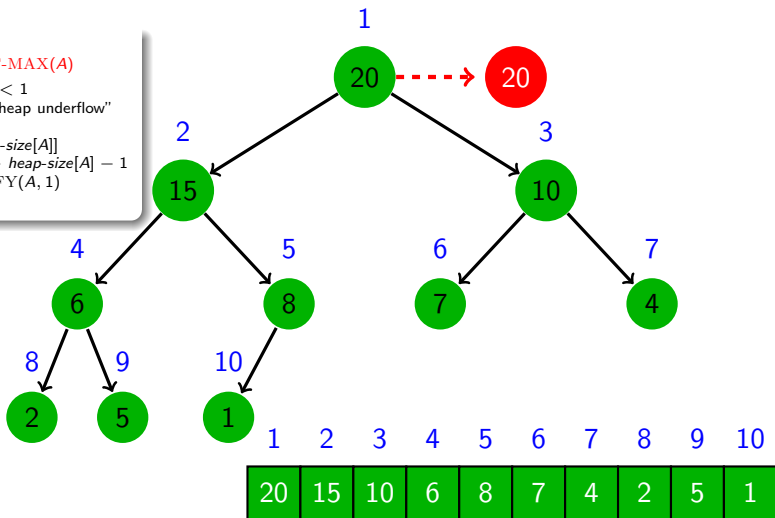
```
1 if heap-size[A] < 1
2   then error "heap underflow"
3 max ← A[1]
4 A[1] ← A[heap-size[A]]
5 heap-size[A] ← heap-size[A] - 1
6 MAX-HEAPIFY(A, 1)
7 return max
```



HEAP-EXTRACT-MAX

HEAP-EXTRACT-MAX(*A*)

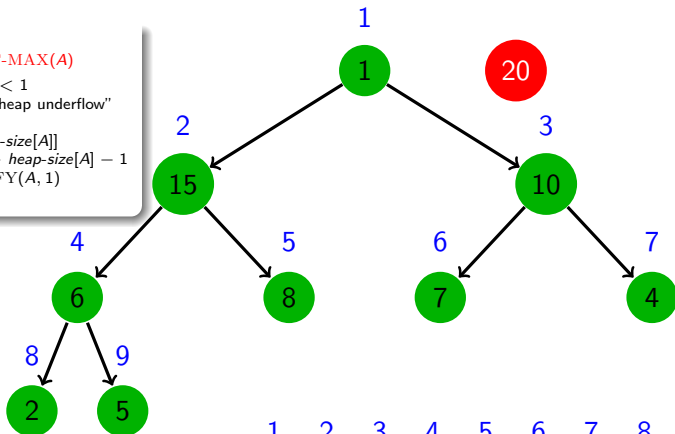
```
1 if heap-size[A] < 1
2   then error "heap underflow"
3 max ← A[1]
4 A[1] ← A[heap-size[A]]
5 heap-size[A] ← heap-size[A] - 1
6 MAX-HEAPIFY(A, 1)
7 return max
```



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
```

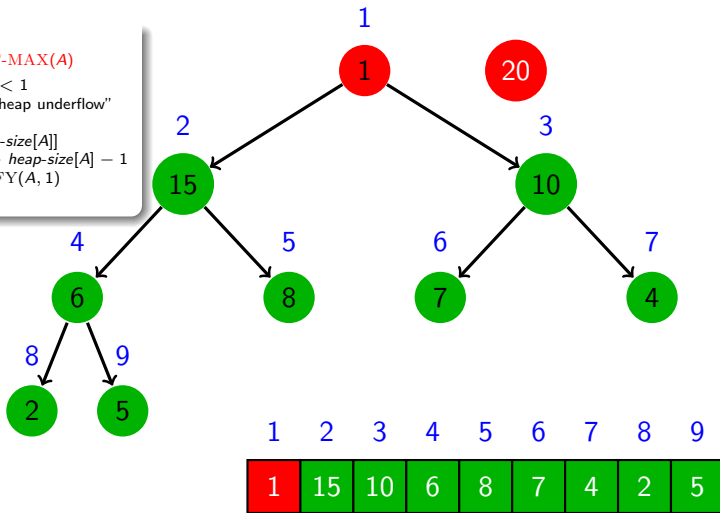


1	2	3	4	5	6	7	8	9
1	15	10	6	8	7	4	2	5

HEAP-EXTRACT-MAX

HEAP-EXTRACT-MAX(*A*)

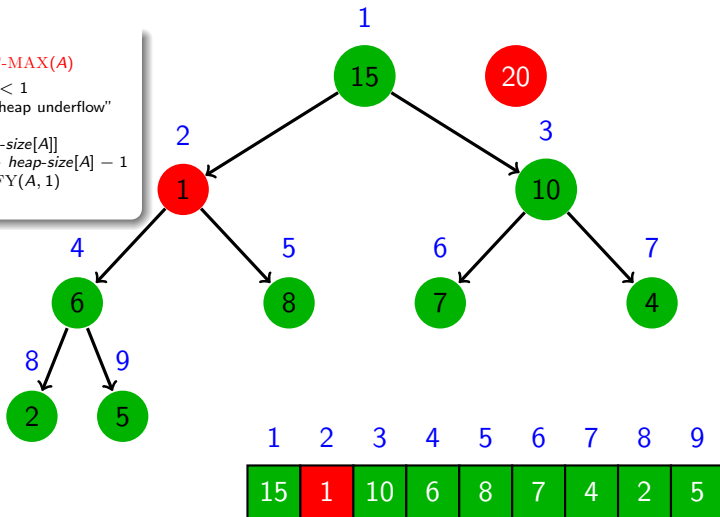
```
1 if heap-size[A] < 1
2   then error "heap underflow"
3 max ← A[1]
4 A[1] ← A[heap-size[A]]
5 heap-size[A] ← heap-size[A] - 1
6 MAX-HEAPIFY(A, 1)
7 return max
```



HEAP-EXTRACT-MAX

HEAP-EXTRACT-MAX(*A*)

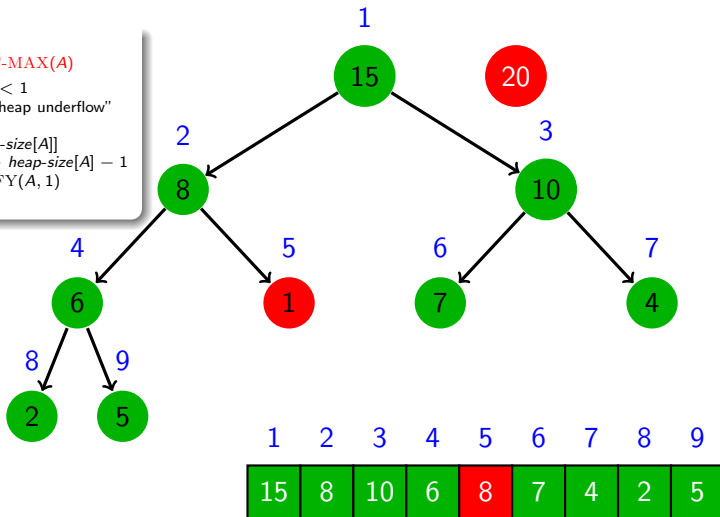
```
1 if heap-size[A] < 1
2   then error "heap underflow"
3 max ← A[1]
4 A[1] ← A[heap-size[A]]
5 heap-size[A] ← heap-size[A] - 1
6 MAX-HEAPIFY(A, 1)
7 return max
```



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
```

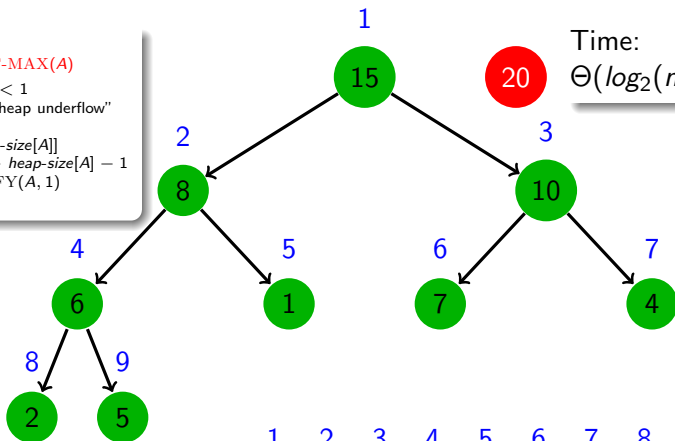


HEAP-EXTRACT-MAX

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

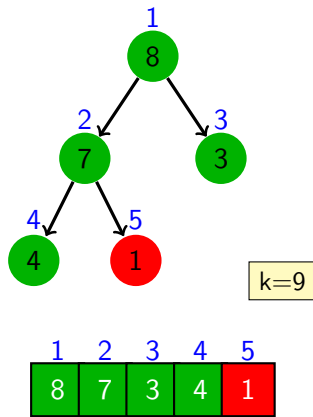


1	2	3	4	5	6	7	8	9
15	8	10	6	8	7	4	2	5

HEAP-INCREASE-KEY: Principle

Problem: Increase the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

HEAP-INCREASE-KEY

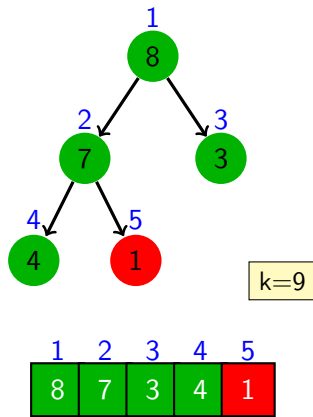


HEAP-INCREASE-KEY: Principle

Problem: Increase the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

HEAP-INCREASE-KEY

- Make sure that k is larger than the original key of element x .

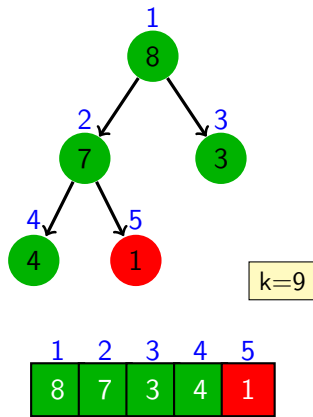


HEAP-INCREASE-KEY: Principle

Problem: Increase the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

HEAP-INCREASE-KEY

- Make sure that k is larger than the original key of element x .
- Increase the value of x 's key to k .

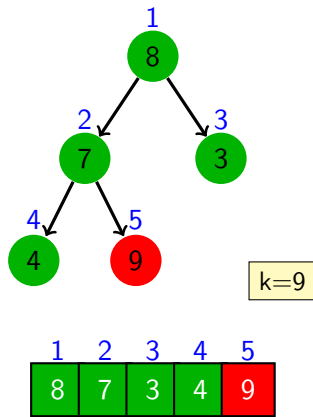


HEAP-INCREASE-KEY: Principle

Problem: Increase the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

HEAP-INCREASE-KEY

- Make sure that k is larger than the original key of element x .
- Increase the value of x 's key to k .

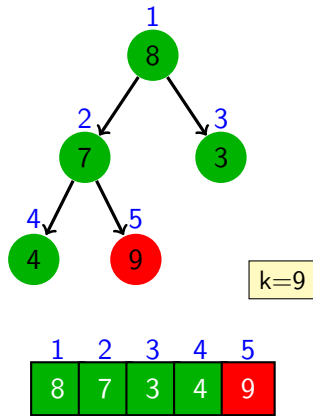


HEAP-INCREASE-KEY: Principle

Problem: Increase the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

HEAP-INCREASE-KEY

- Make sure that k is larger than the original key of element x .
- Increase the value of x 's key to k .
- Traverse the tree upward comparing x to its parent and swapping if necessary, until the max-heap property is restored.

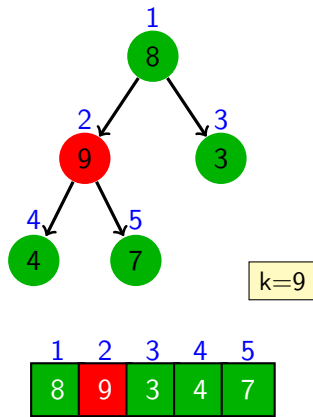


HEAP-INCREASE-KEY: Principle

Problem: Increase the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

HEAP-INCREASE-KEY

- Make sure that k is larger than the original key of element x .
- Increase the value of x 's key to k .
- Traverse the tree upward comparing x to its parent and swapping if necessary, until the max-heap property is restored.

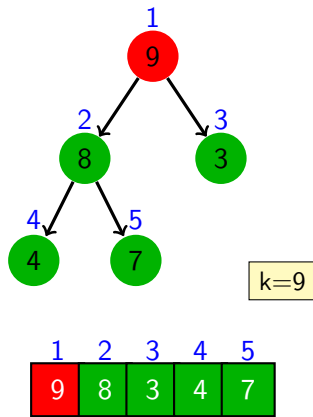


HEAP-INCREASE-KEY: Principle

Problem: Increase the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

HEAP-INCREASE-KEY

- Make sure that k is larger than the original key of element x .
- Increase the value of x 's key to k .
- Traverse the tree upward comparing x to its parent and swapping if necessary, until the max-heap property is restored.

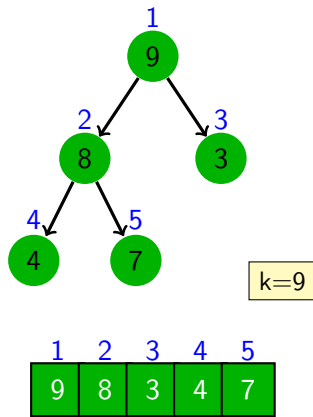


HEAP-INCREASE-KEY: Principle

Problem: Increase the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

HEAP-INCREASE-KEY

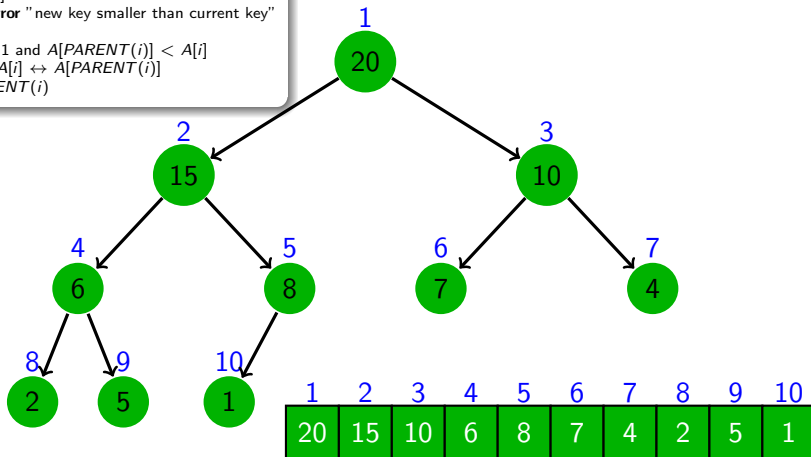
- Make sure that k is larger than the original key of element x .
- Increase the value of x 's key to k .
- Traverse the tree upward comparing x to its parent and swapping if necessary, until the max-heap property is restored.



HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(A, x, k)

```
1  $i \leftarrow$  the index where  $x$  is stored.  
2 if  $k < A[i]$   
3   then error "new key smaller than current key"  
4  $A[i] \leftarrow k$   
5 while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$   
6   exchange  $A[i] \leftrightarrow A[\text{PARENT}(i)]$   
7    $i \leftarrow \text{PARENT}(i)$ 
```

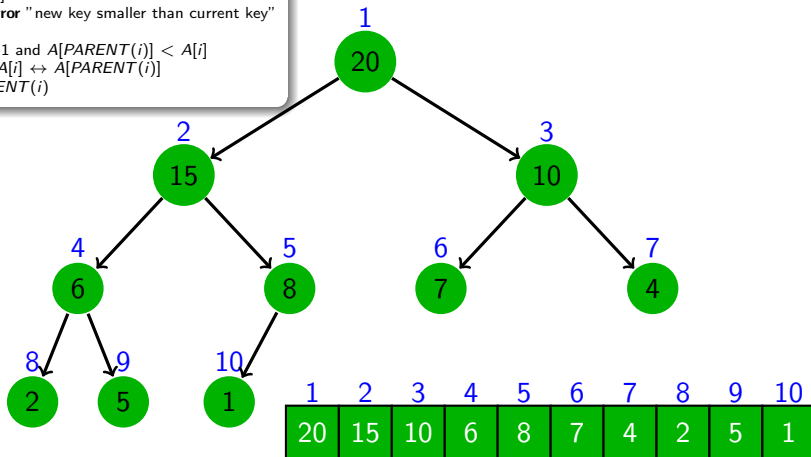


HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(A, x, k)

```
1  $i \leftarrow$  the index where  $x$  is stored.  
2 if  $k < A[i]$   
3   then error "new key smaller than current key"  
4  $A[i] \leftarrow k$   
5 while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$   
6   exchange  $A[i] \leftrightarrow A[\text{PARENT}(i)]$   
7    $i \leftarrow \text{PARENT}(i)$ 
```

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

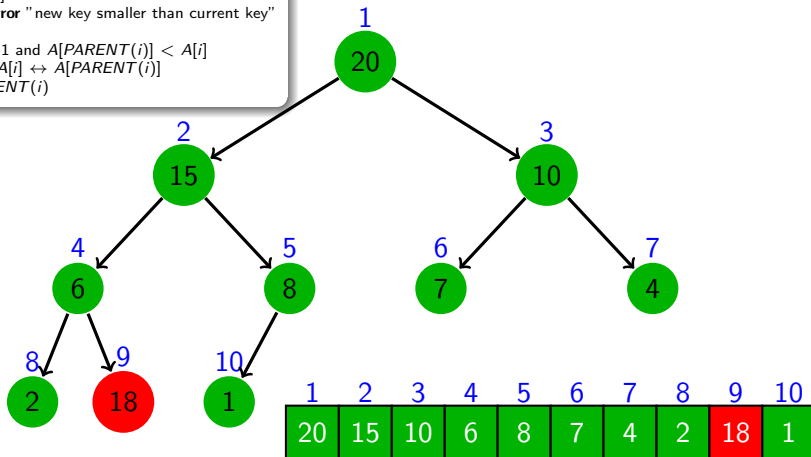


HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(A, x, k)

```
1  $i \leftarrow$  the index where  $x$  is stored.  
2 if  $k < A[i]$   
3   then error "new key smaller than current key"  
4  $A[i] \leftarrow k$   
5 while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$   
6   exchange  $A[i] \leftrightarrow A[\text{PARENT}(i)]$   
7  $i \leftarrow \text{PARENT}(i)$ 
```

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

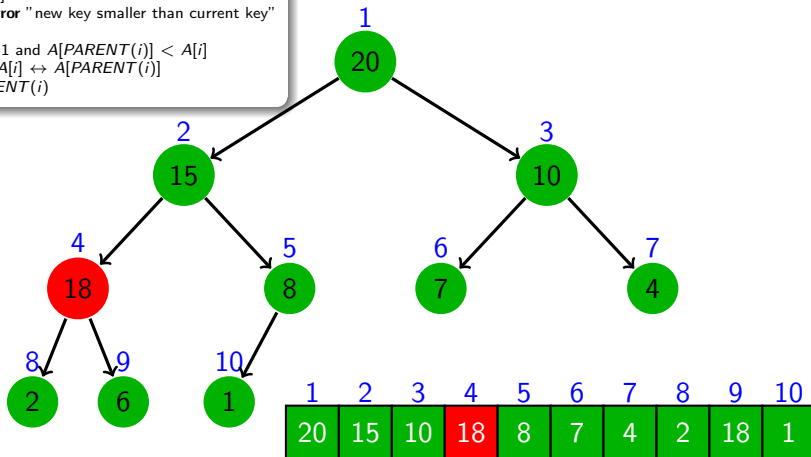


HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(A, x, k)

```
1  $i \leftarrow$  the index where  $x$  is stored.  
2 if  $k < A[i]$   
3   then error "new key smaller than current key"  
4  $A[i] \leftarrow k$   
5 while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$   
6   exchange  $A[i] \leftrightarrow A[\text{PARENT}(i)]$   
7  $i \leftarrow \text{PARENT}(i)$ 
```

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

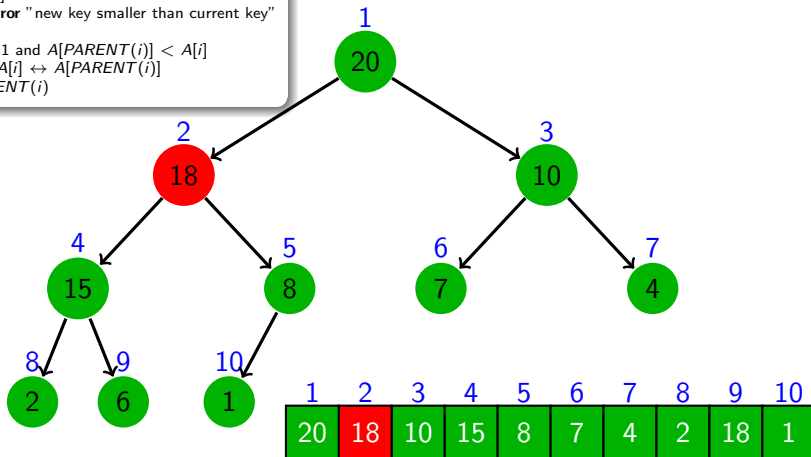


HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(A, x, k)

```
1  $i \leftarrow$  the index where  $x$  is stored.  
2 if  $k < A[i]$   
3   then error "new key smaller than current key"  
4  $A[i] \leftarrow k$   
5 while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$   
6   exchange  $A[i] \leftrightarrow A[\text{PARENT}(i)]$   
7  $i \leftarrow \text{PARENT}(i)$ 
```

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

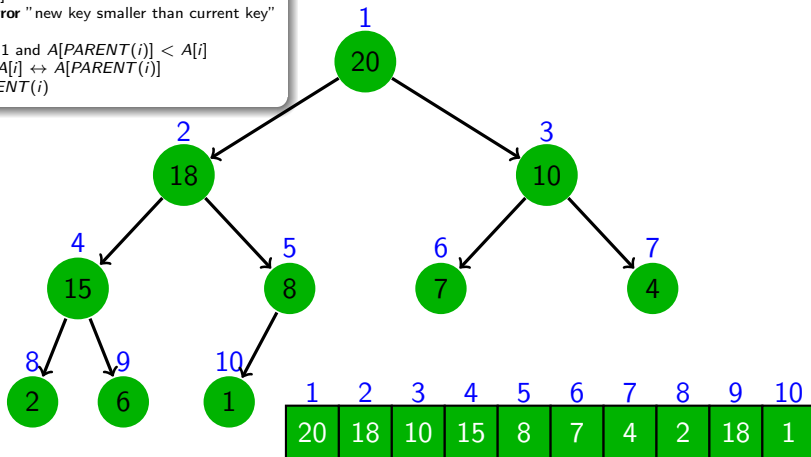


HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(A, x, k)

```
1  $i \leftarrow$  the index where  $x$  is stored.  
2 if  $k < A[i]$   
3   then error "new key smaller than current key"  
4  $A[i] \leftarrow k$   
5 while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$   
6   exchange  $A[i] \leftrightarrow A[\text{PARENT}(i)]$   
7    $i \leftarrow \text{PARENT}(i)$ 
```

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



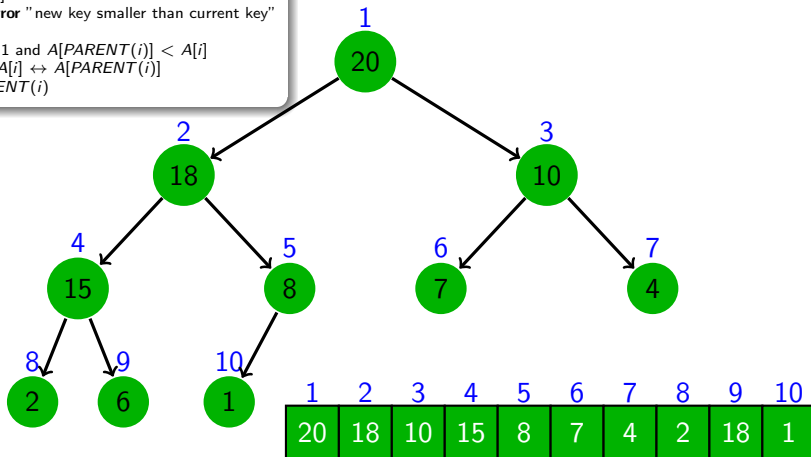
HEAP-INCREASE-KEY

HEAP-INCREASE-KEY(A, x, k)

```
1  $i \leftarrow$  the index where  $x$  is stored.  
2 if  $k < A[i]$   
3   then error "new key smaller than current key"  
4  $A[i] \leftarrow k$   
5 while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$   
6   exchange  $A[i] \leftrightarrow A[\text{PARENT}(i)]$   
7  $i \leftarrow \text{PARENT}(i)$ 
```

Time:
 $\Theta(\log_2 n)$

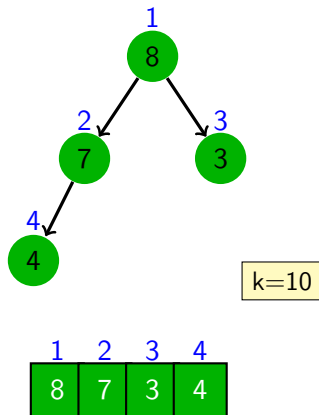
HEAP-INCREASE-KEY($A, 9, 18$)



MAX-HEAP-INSERT: Principle

Problem: insert the element x (with key k) into the heap

MAX-HEAP-INSERT

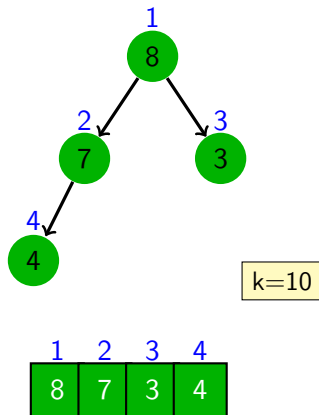


MAX-HEAP-INSERT: Principle

Problem: insert the element x (with key k) into the heap

MAX-HEAP-INSERT

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

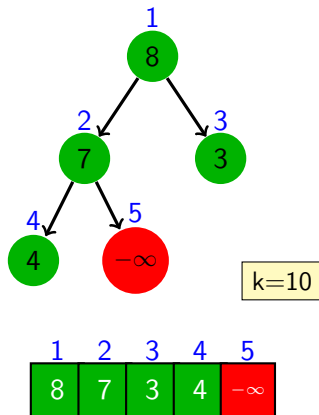


MAX-HEAP-INSERT: Principle

Problem: insert the element x (with key k) into the heap

MAX-HEAP-INSERT

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

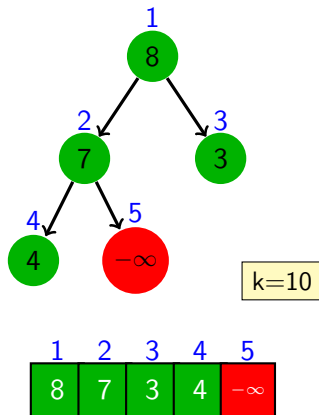


MAX-HEAP-INSERT: Principle

Problem: insert the element x (with key 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

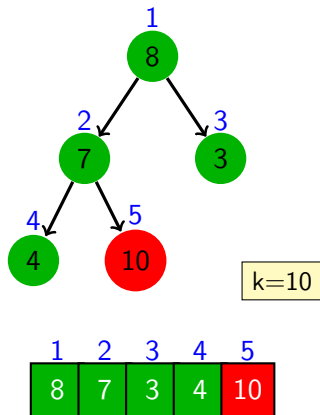


MAX-HEAP-INSERT: Principle

Problem: insert the element x (with key 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

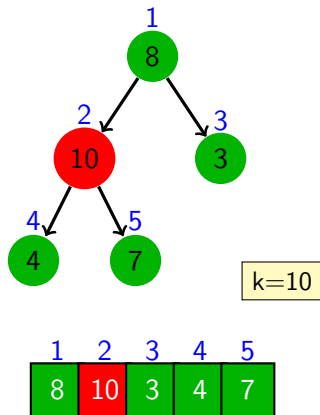


MAX-HEAP-INSERT: Principle

Problem: insert the element x (with key 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

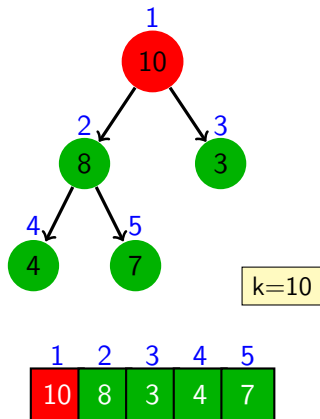


MAX-HEAP-INSERT: Principle

Problem: insert the element x (with key 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

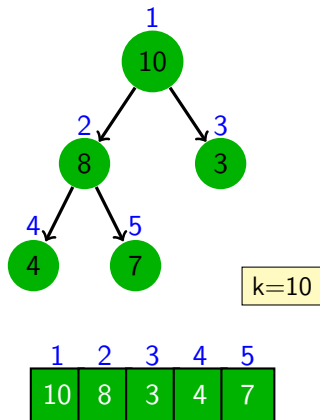


MAX-HEAP-INSERT: Principle

Problem: insert the element x (with key k) into the heap

MAX-HEAP-INSERT

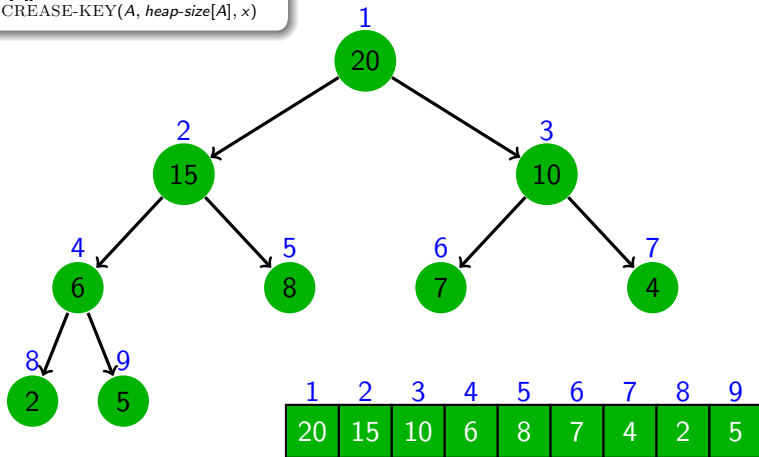
- Insert a new node in the very last position in the tree with key $-\infty$
- Increase the $-\infty$ value to k using the HEAP-INCREASE-KEY procedure



MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, x)

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

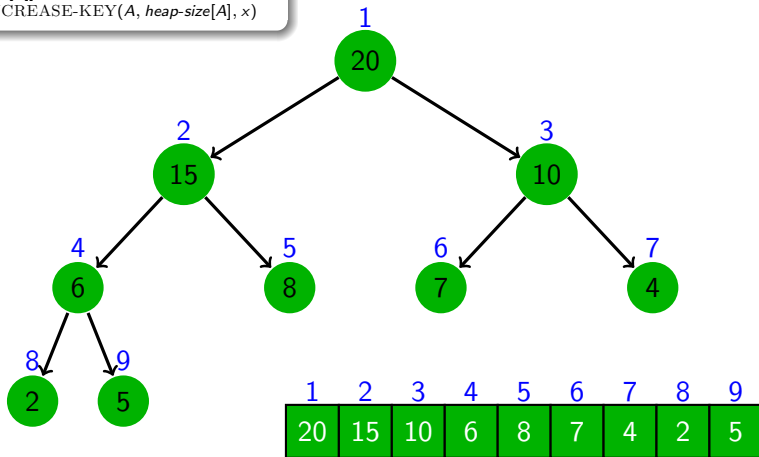


MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, x)

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

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

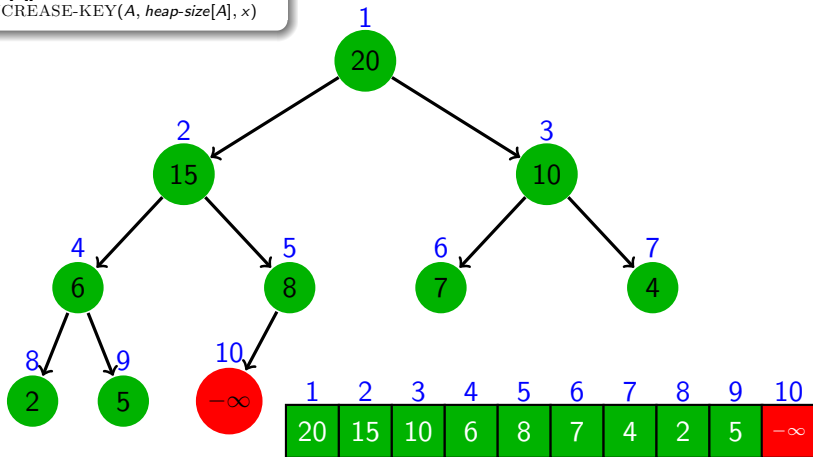


MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, x)

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

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

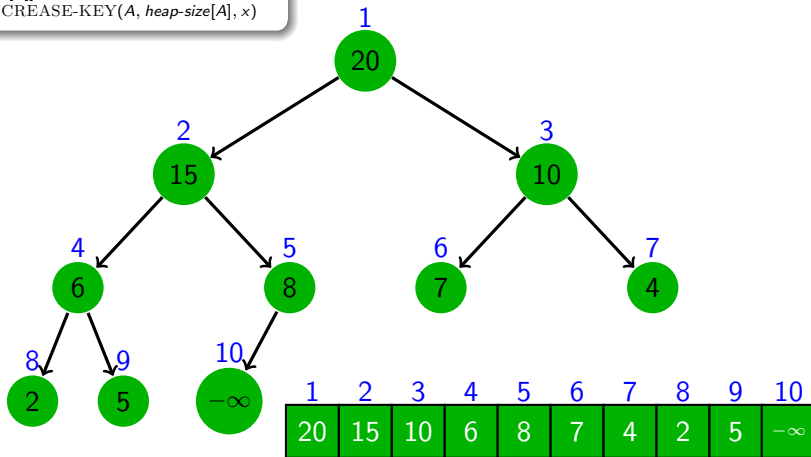


MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, x)

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

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

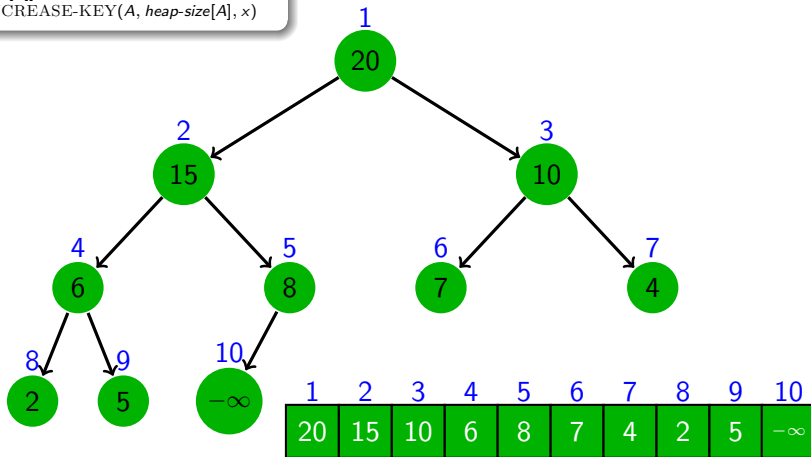


MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, x)

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

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

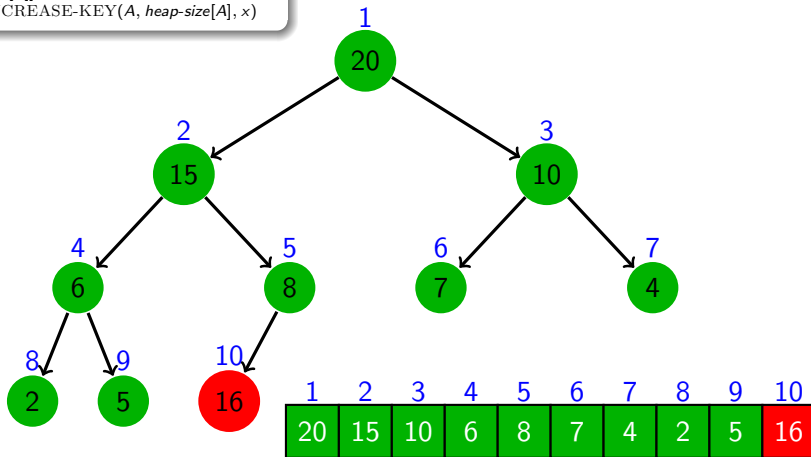


MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, x)

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

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

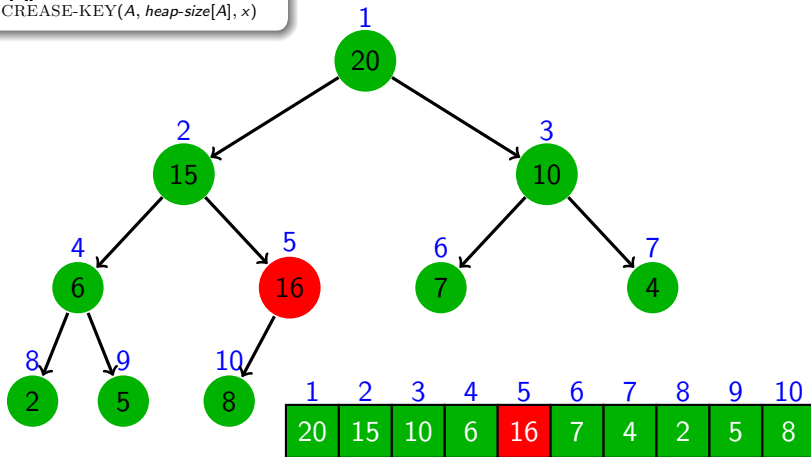


MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, x)

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

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

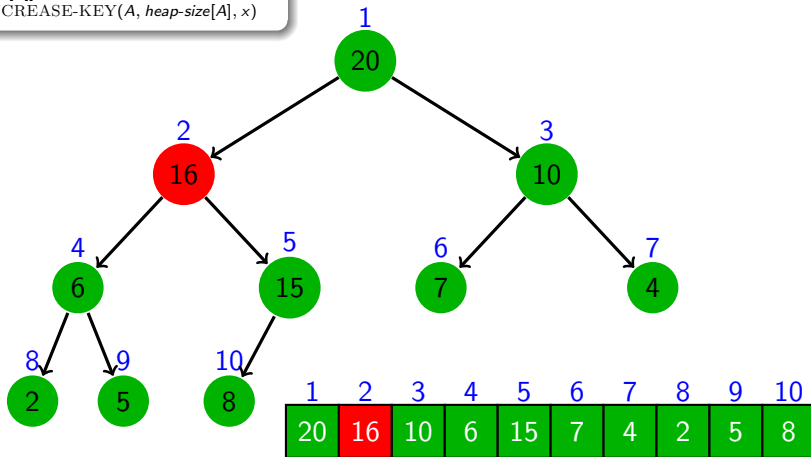


MAX-HEAP-INSERT

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($A, heap\text{-}size[A], x$)

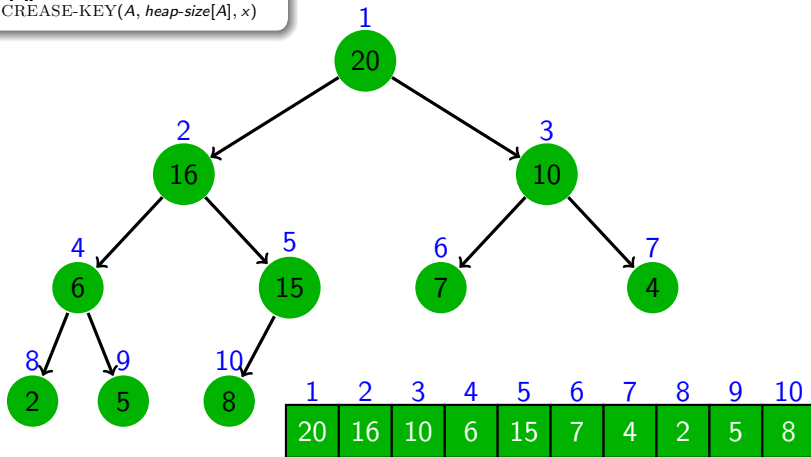
HEAP-INCREASE-KEY($A, 10, 16$)



MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, x)

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

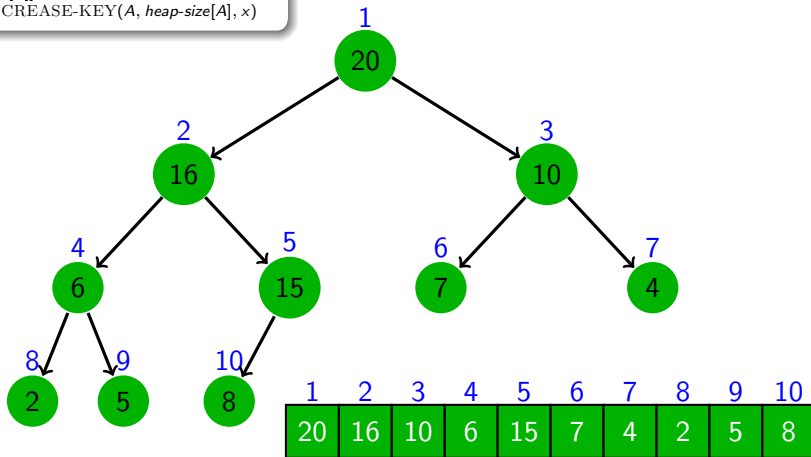


MAX-HEAP-INSERT

MAX-HEAP-INSERT(A, x)

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

Time:
 $\Theta(\log_2 n)$



Using Heaps to implement Priority Queues

- The operations have the following worst-case running times:
 - $\text{HEAP-INSERT}(S, x)$: $\Theta(\log n)$
 - $\text{HEAP-MAXIMUM}(S)$: $\Theta(1)$
 - $\text{HEAP-EXTRACT-MAX}(S)$: $\Theta(\log n)$
 - $\text{HEAP-INCREASE-KEY}(S, x, k)$: $\Theta(\log n)$