Design and Analysis of Algorithms

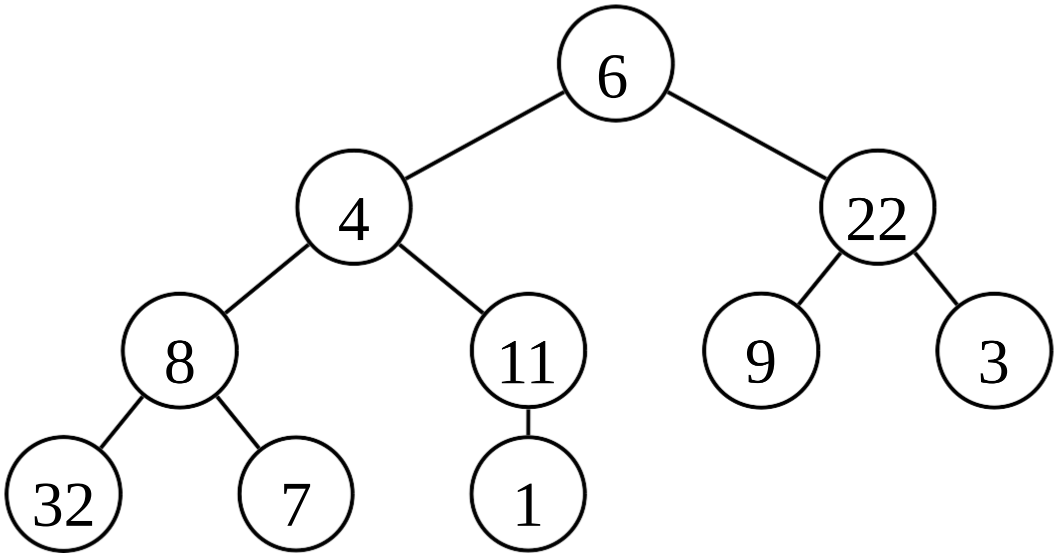
Chapter 3

Data structures as a tool for algorithm design: heaps, heapsort, and priority queues

Heaps [CLRS 6.1]

A heap is a data structure that organizes data in an essentially complete rooted tree,
i.e. a rooted tree that is completely filled on all levels except possibly on the lowest, *which is* filled from the left up to a point.

Example: Binary heap, storing numbers (keys) at the nodes of the tree

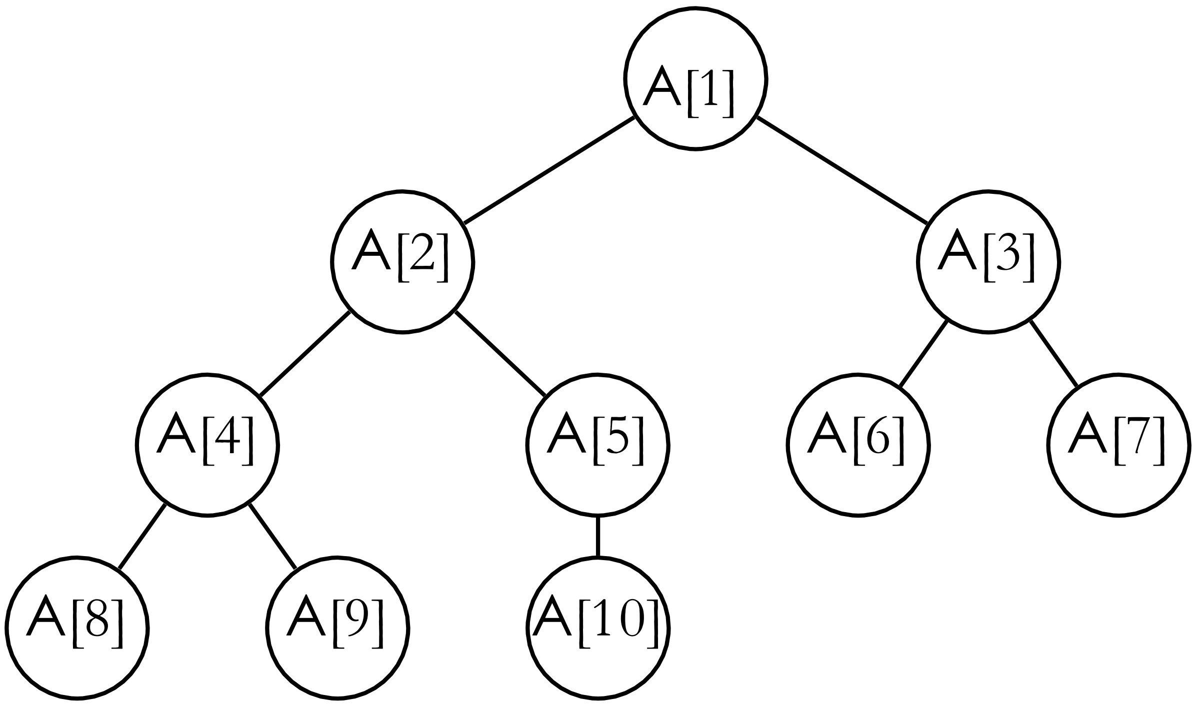


The *height* of a tree is the longest simple path from the root to a leave. A binary heap with n nodes has height $\lfloor \lg n \rfloor$.

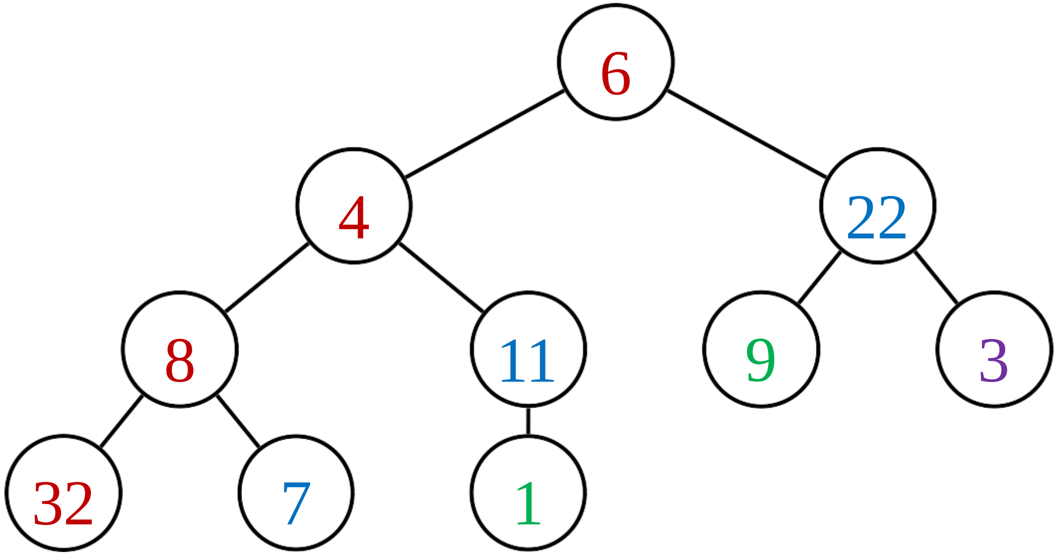
Implementation with arrays

A heap can be implemented by an array without any explicit pointers.
In particular, a *binary* heap can be implemented by an array A as follows:

Root of the binary tree is $A[1]$
Left child of $A[i]$ is $A[2i]$
Right child of $A[i]$ is $A[2i+1]$.
Hence, for $i > 0$, the parent of node i is the node $\text{Parent}(i) = \lfloor i/2 \rfloor$



Example



The heap is stored as the following array:

$A =$

6	4	22	8	11	9	3	32	7	1

Max-heaps

A *max-heap* is a heap that satisfies the

Max-Heap Property: The key of a node (except the root) is less than or equal to the key of its parent.

In the array implementation, the Max-Heap Property Reads:
For all $1 < i \le A.heap\text{-}size$: $A[i] \le A[\lfloor i/2 \rfloor]$.

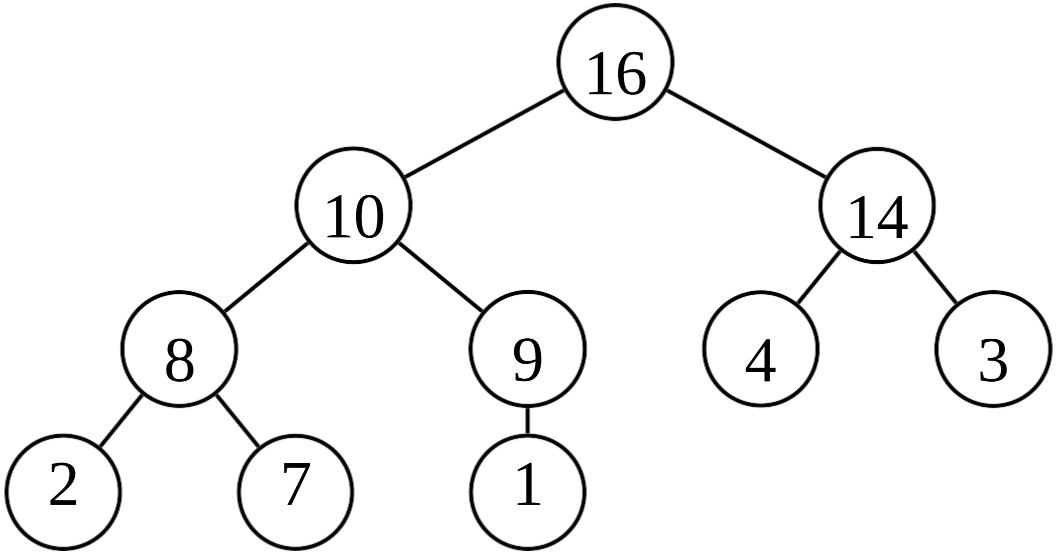
Remarks:

The maximum element of a max-heap is at the root.

In the following we will focus on *binary max-heaps*.
Generally, a max-heap may be k-ary.

One could also define *min-heaps*, where the key of each node (except the root) is larger than or equal to the key of its parent.

Example



This is a max-heap. It can be stored in the array

$A =$

16	10	14	8	9	4	3	2	7	1

Note that the array A is *not sorted*:
it does *not* satisfy the property $A[i] \leq A[i-1]$ for every $i > 1$.
However, A satisfies the max-heap property

$$A[i] \leq A[\lfloor i/2 \rfloor] \text{ for every } i > 1.$$

Building a max-heap

Given an array A, there is a procedure to turn A into a max-heap: MAKE-MAX-HEAP(A)

Takes an array A of n integers and rearranges it into a max-heap of size n.

In turn, MAKE-MAX-HEAP is based on the following procedure: MAX-HEAPIFY(A, i)

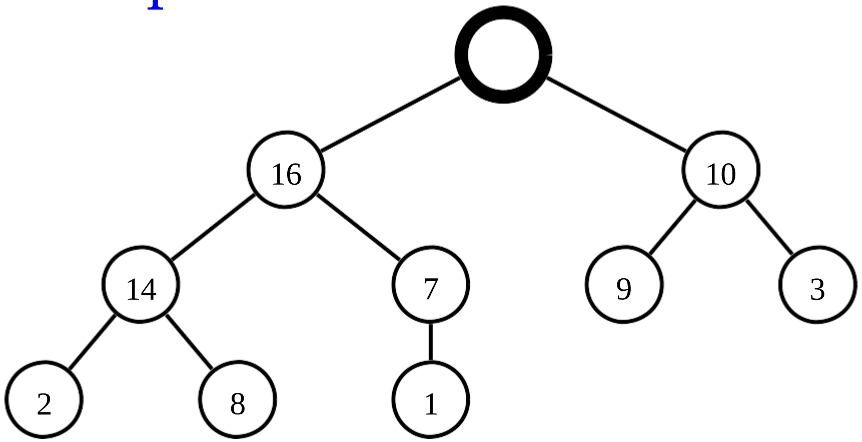
Assuming that the left and right subtrees of node i are max-heaps, MAX-HEAPIFY transforms the subtree rooted at the node i to a max-heap.

MAX-HEAPIFY [CLRS 6.2]

Idea: compare the key at node i with the keys of its children, and rearrange them in order to satisfy the max-heap property.

Example:

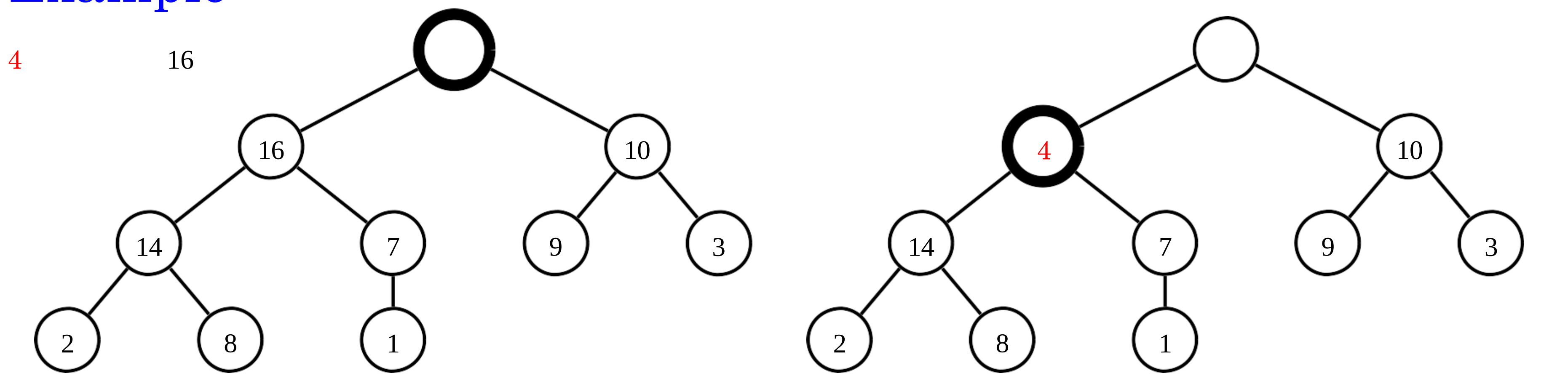
4



MAX-HEAPIFY [CLRS 6.2]

Idea: compare the key at node i with the keys of its children, and rearrange them in order to satisfy the max-heap property.

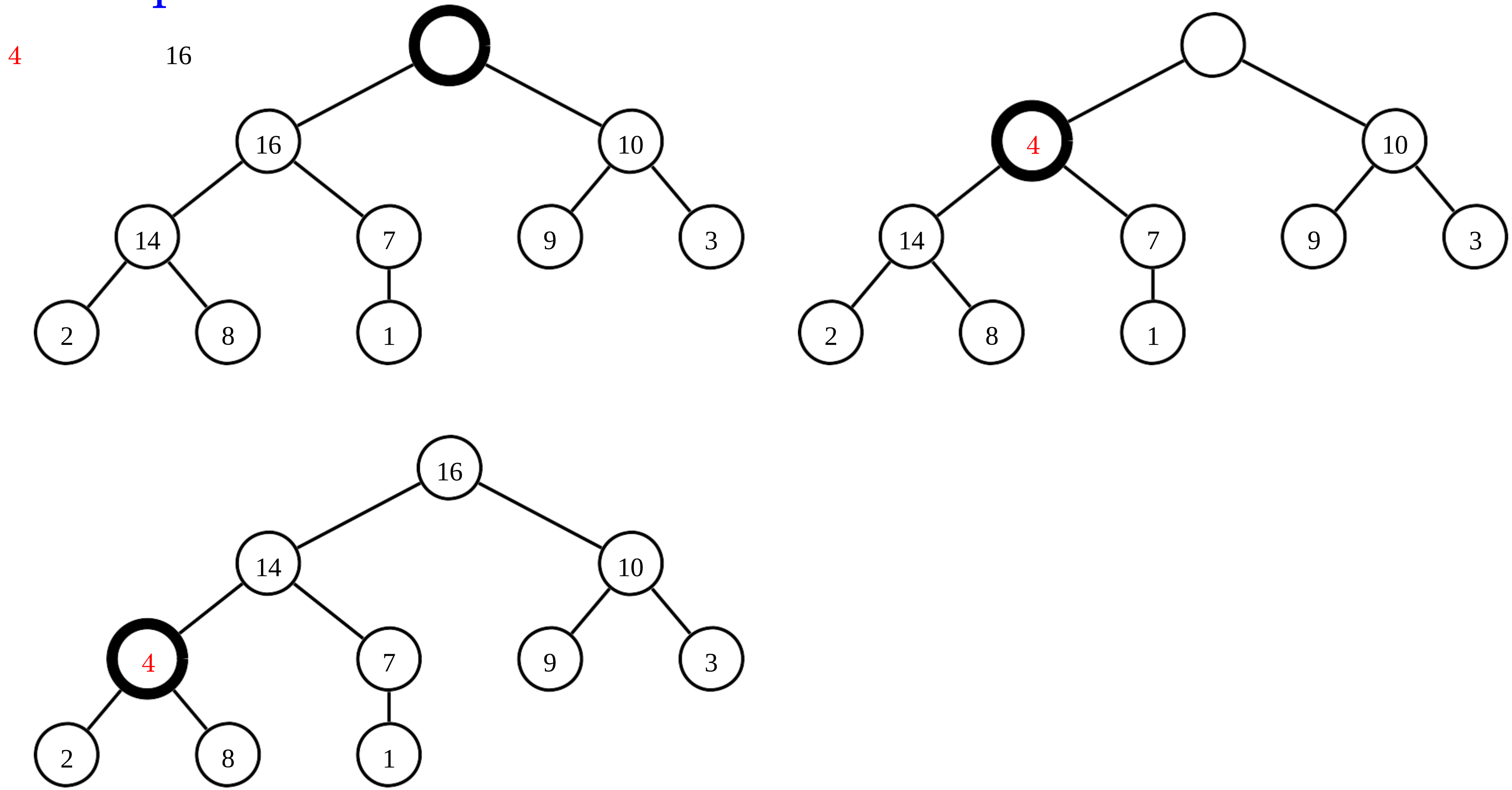
Example:



MAX-HEAPIFY [CLRS 6.2]

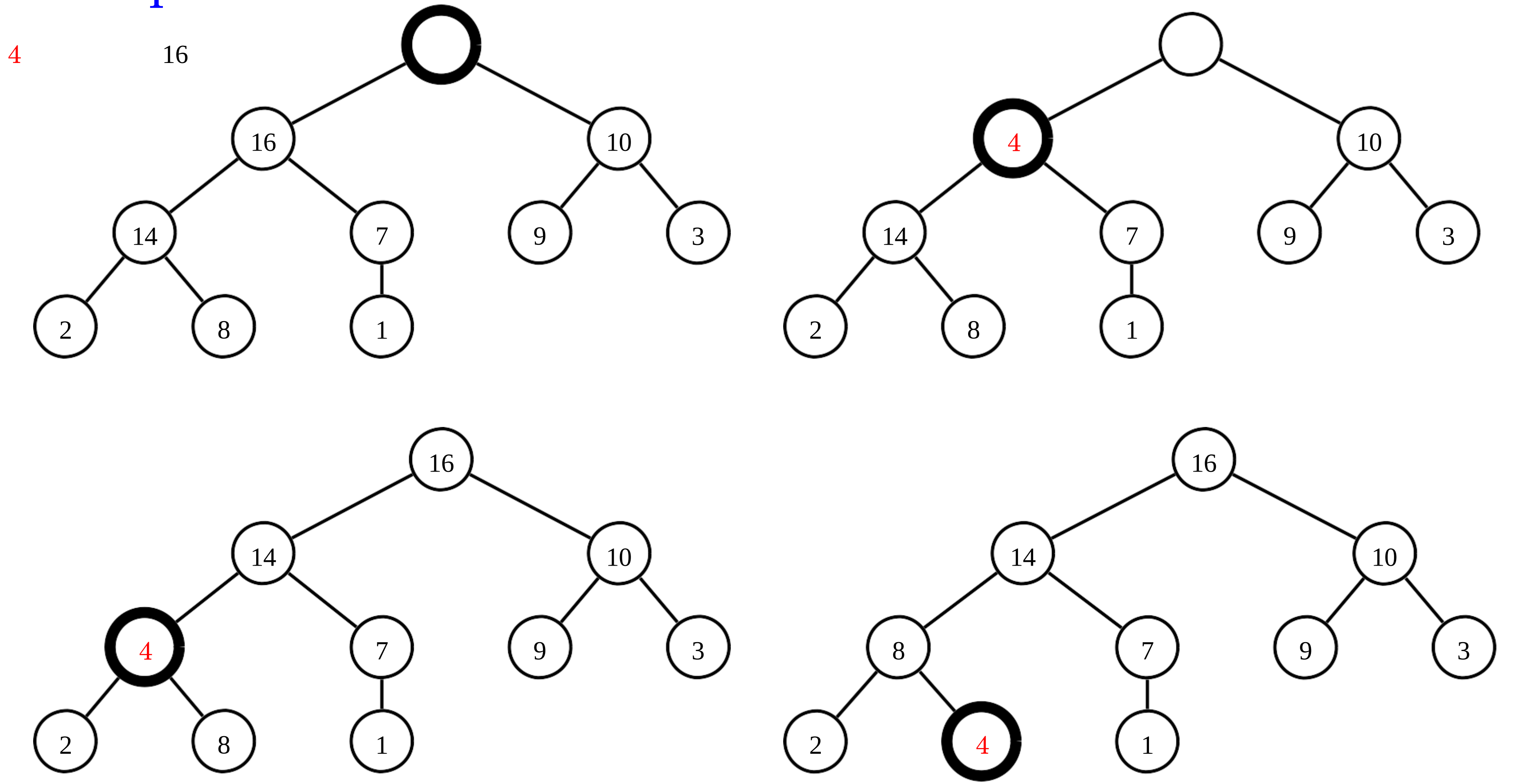
Idea: compare the key at node i with the keys of its children, and rearrange them in order to satisfy the max-heap property.

Example:



Idea: compare the key at node i with the keys of its children, and rearrange them in order to satisfy the max-heap property.

Example:



MAX-HEAPIFY in pseudocode

MAX-HEAPIFY(A, i)

Input: Assume left and right subtrees of i are max-heaps.
Output: Subtree rooted at i is a max-heap.

```
1   n = A.heap-size
2   l = 2i                                  / A[l] is the left-child of A[i]
3   r = 2i + 1                              / A[r] is the right-child of A[i]
4   if l ≤ n and A[l] > A[i]                / Lines 4-8: Determine
5       largest = l                         / largest among A[i], A[l] and A[r].
6   else largest = i
7   if r ≤ n and A[r] > A[largest]
8       largest = r
9   if largest ≠ i
10      exchange A[i] with A[largest]
11      MAX-HEAPIFY(A, largest)
```

Running time of MAX-HEAPIFY

MAX-HEAPIFY a subtree of size n at node i

$\Theta(1)$ to find the largest among $A[i]$, $A[2i]$ and $A[2i+1]$.

The subtree rooted at a child of node i has size upper bounded by $2n/3$ (Exercise. Prove this fact.
Proof idea: the worst case is when last row of tree is exactly half full).

Thus $T(n) \leq T(2n/3) + \Theta(1)$.

By the Master Theorem, we have

$$T(n) = O(n^0 \log n) = O(\log n).$$

Alternative reasoning:

Define the *height* of a node to be the number of edges on the longest simple downward path from the node to a leaf.

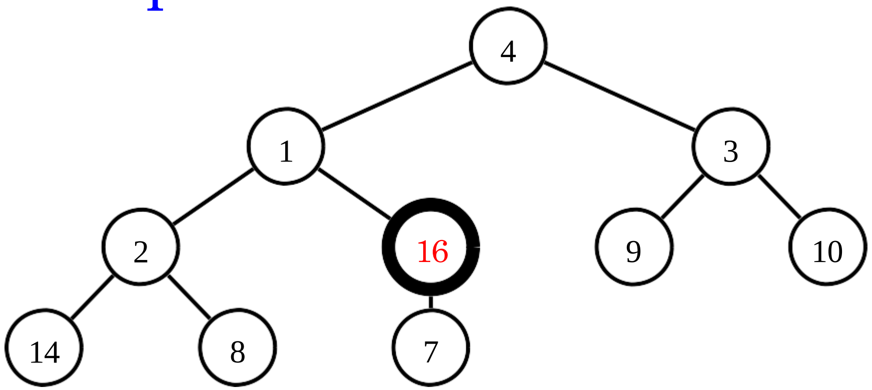
On a node of height h, MAX-HEAPIFY runs for $O(h)$ time at most.

The height of the root of a heap of size n is $\lfloor \lg n \rfloor$, so $T(n) = O(\log n)$.

MAKE-MAX-HEAP [CLRS 6.3]

Idea: starting from the last *non-leave* node,
apply MAX-HEAPIFY to the subtree based at that node. Repeat the same procedure for all the previous nodes.

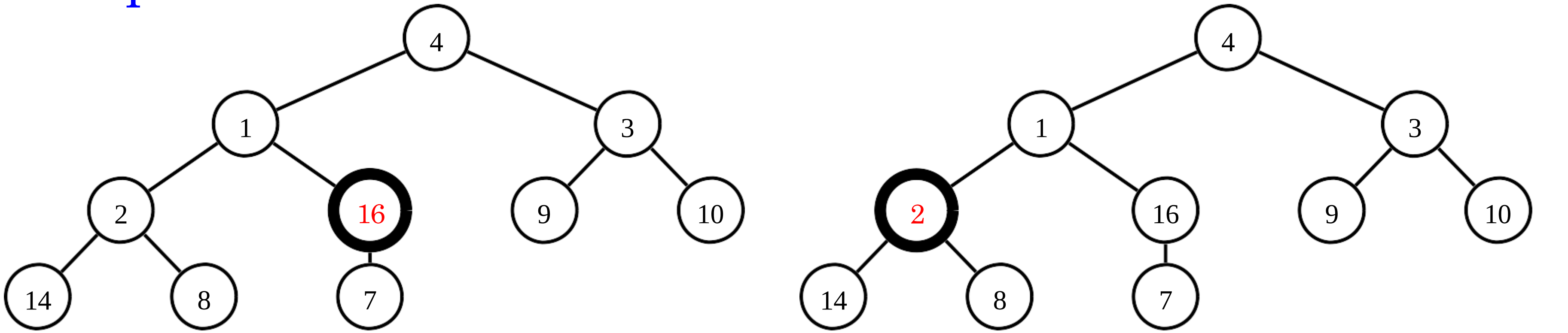
Example



MAKE-MAX-HEAP [CLRS 6.3]

Idea: starting from the last *non-leave* node,
apply MAX-HEAPIFY to the subtree based at that node. Repeat the same procedure for all the previous nodes.

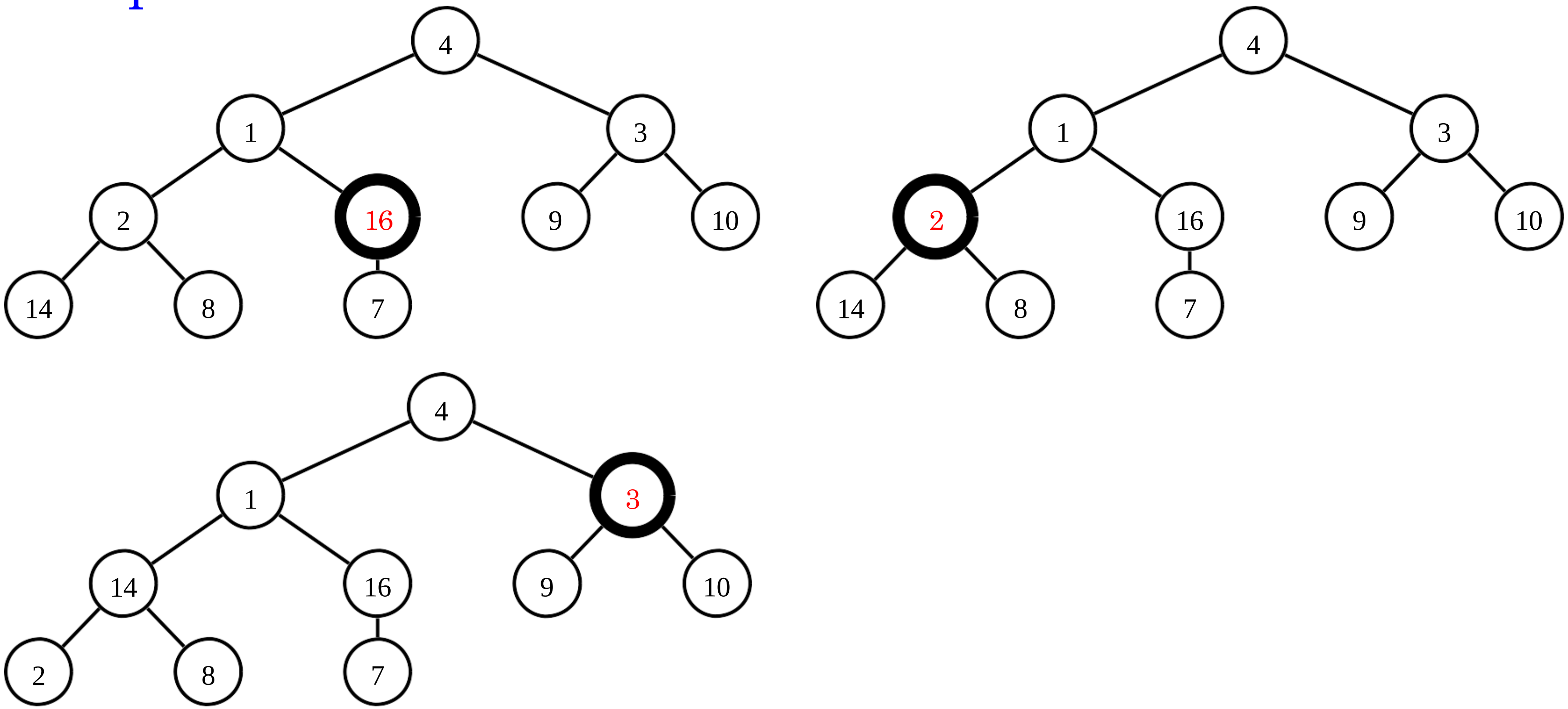
Example



MAKE-MAX-HEAP [CLRS 6.3]

Idea: starting from the last *non-leave* node,
apply MAX-HEAPIFY to the subtree based at that node. Repeat the same procedure for all the previous nodes.

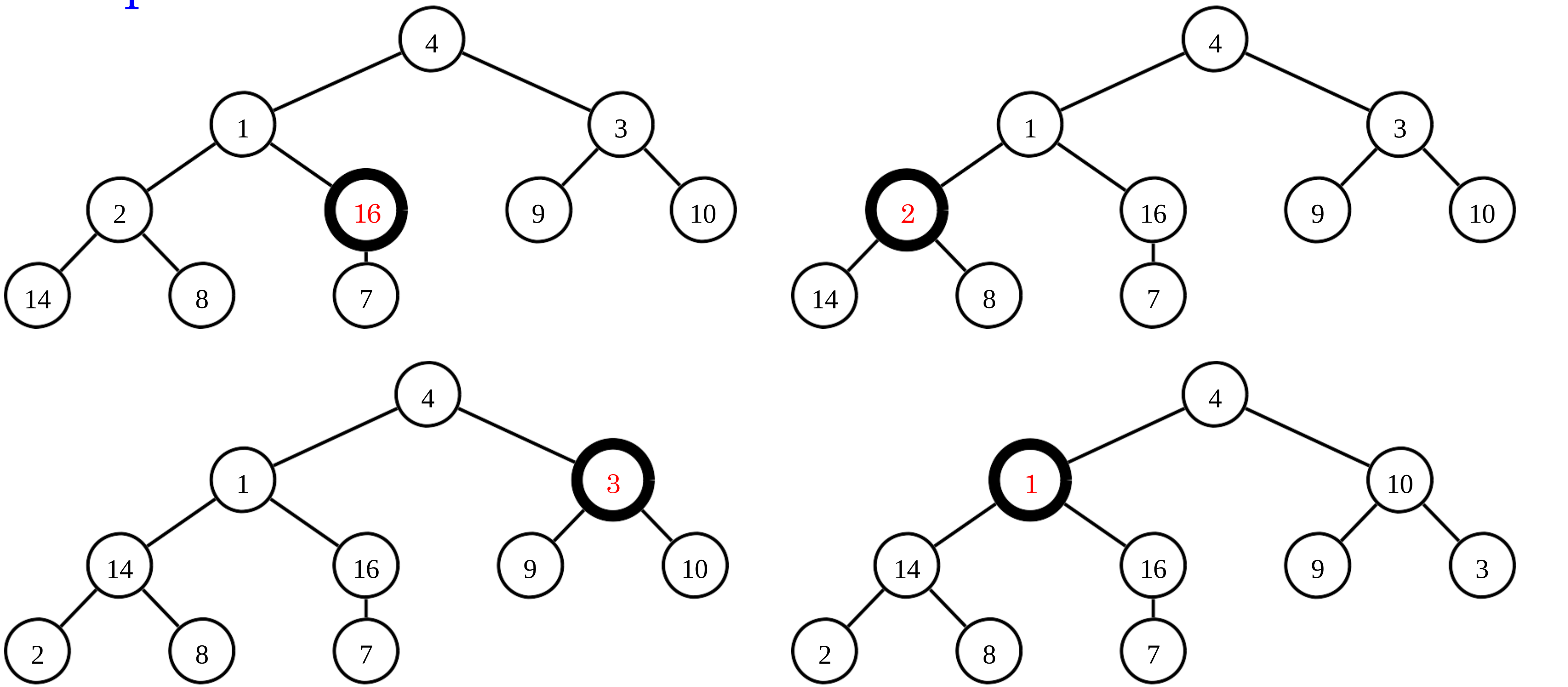
Example



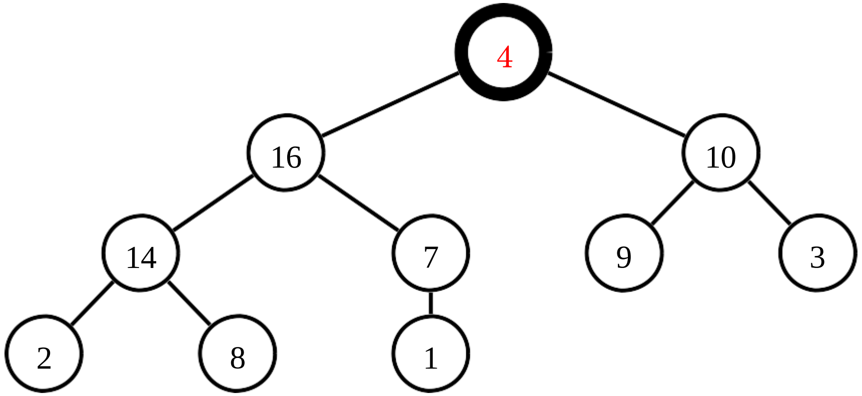
MAKE-MAX-HEAP [CLRS 6.3]

Idea: starting from the last *non-leave* node,
apply MAX-HEAPIFY to the subtree based at that node. Repeat the same procedure for all the previous nodes.

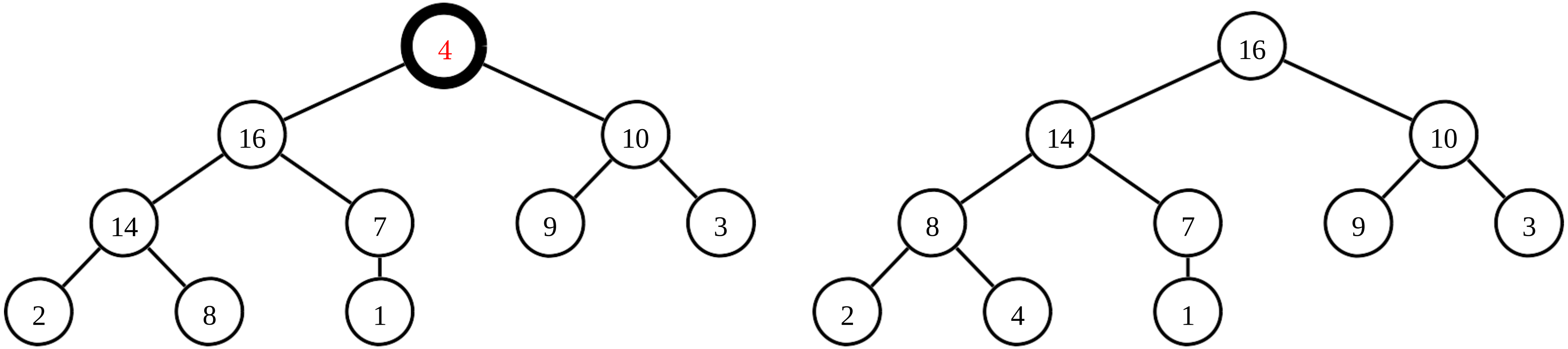
Example



MAKE-MAX-HEAP (example continued)



MAKE-MAX-HEAP (example continued)



Note that the procedure works because at every step the left and right subtrees are max-heaps.

Pseudocode

Recall that the leaves are the array elements indexed by

$$\lceil \tfrac{n+1}{2} \rceil, \lceil \tfrac{n+1}{2} \rceil + 1, \cdots, n.$$

MAKE-MAX-HEAP(A)

Input: An (unsorted) integer array A of length n.

Output: A heap of size n.

```
A.heap-size = A.length
for i = ⌈(n+1)/2⌉ − 1 downto 1
    MAX-HEAPIFY(A, i)
```

Correctness

Loop invariant: *Each node $i+1, i+2, \cdots, n$ is the root of a max-heap.*

Initialization

Each node $\lceil \frac{n+1}{2} \rceil, \lceil \frac{n+1}{2} \rceil + 1, \cdots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lceil \frac{n+1}{2} \rceil - 1$ before the first iteration, the invariant is initially true.

Maintenance

Suppose $i = i_0 \geq 1$ and assume each node $i_0+1, i_0+2, \cdots, n$ is the root of a max-heap. Executing MAX-HEAPIFY(A, i) causes i_0 to be the root of a new max-heap. Hence each node $i_0, i_0+1, \cdots, n$ is now the root of a max-heap, meaning that the loop invariant holds after i has been decremented from i_0 to $i_0 - 1$.

Termination

When $i = 0$ (i.e. after the counter becomes less than 1) the loop terminates. By the loop invariant, each node, in particular node 1, is the root of a max-heap.

Running time analysis

Simple (but loose) bound: $O(n \log n)$.

We have $O(n)$ calls to MAX-HEAPIFY, each taking $O(\log n)$ time.

Tighter analysis: $O(n)$.

MAX-HEAPIFY takes linear time in the height of the node it runs on, and "most nodes have small heights".

Fact. The number of nodes of height h is upper bounded by $n/2^h$, and the cost of MAX-HEAPIFY on a node of height h is $\leq ch$, for some $c > 0$.

Hence, the cost of MAKE-MAX-HEAP is

$$\boxed{T(n) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch \;\leq cn \left(\sum_{h=0}^{\infty} \frac{h}{2^h} \right) = 2cn\,,}$$

Note. For $|x| < 1$, one has $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$. Differentiating and multiplying by x, we get $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$

Applications of heaps

Sorting: *heapsort*, an *in-place* sorting algorithm with worst-case complexity $O(n \log n)$.

Efficient implementation of *priority queues*:

Max-heap $\longrightarrow$ max-priority queue.

Min-heap $\longrightarrow$ min-priority queue.
Max-priority queues can be used to schedule jobs on a shared computer.
Min-priority queues can be used to simulate events in time.

Remark. Actual implementations often have a *handle* in each heap element that allows access to an object in the application, and objects in the application often have a handle (likely an array index) to access the heap element.

Heapsort [CLRS 6.4]

A sorting algorithm based on the heap data structure.

Idea. Given an input array,

Build a max-heap using MAKE-MAX-HEAP.

Starting from the root (maximum element), place the maximum element into the correct place in the array by swapping it with the element in the last position in the array.

"Discard" this last node – decrement the heap size, and call MAX-HEAPIFY on the smaller structure with the possibly incorrectly-placed root.

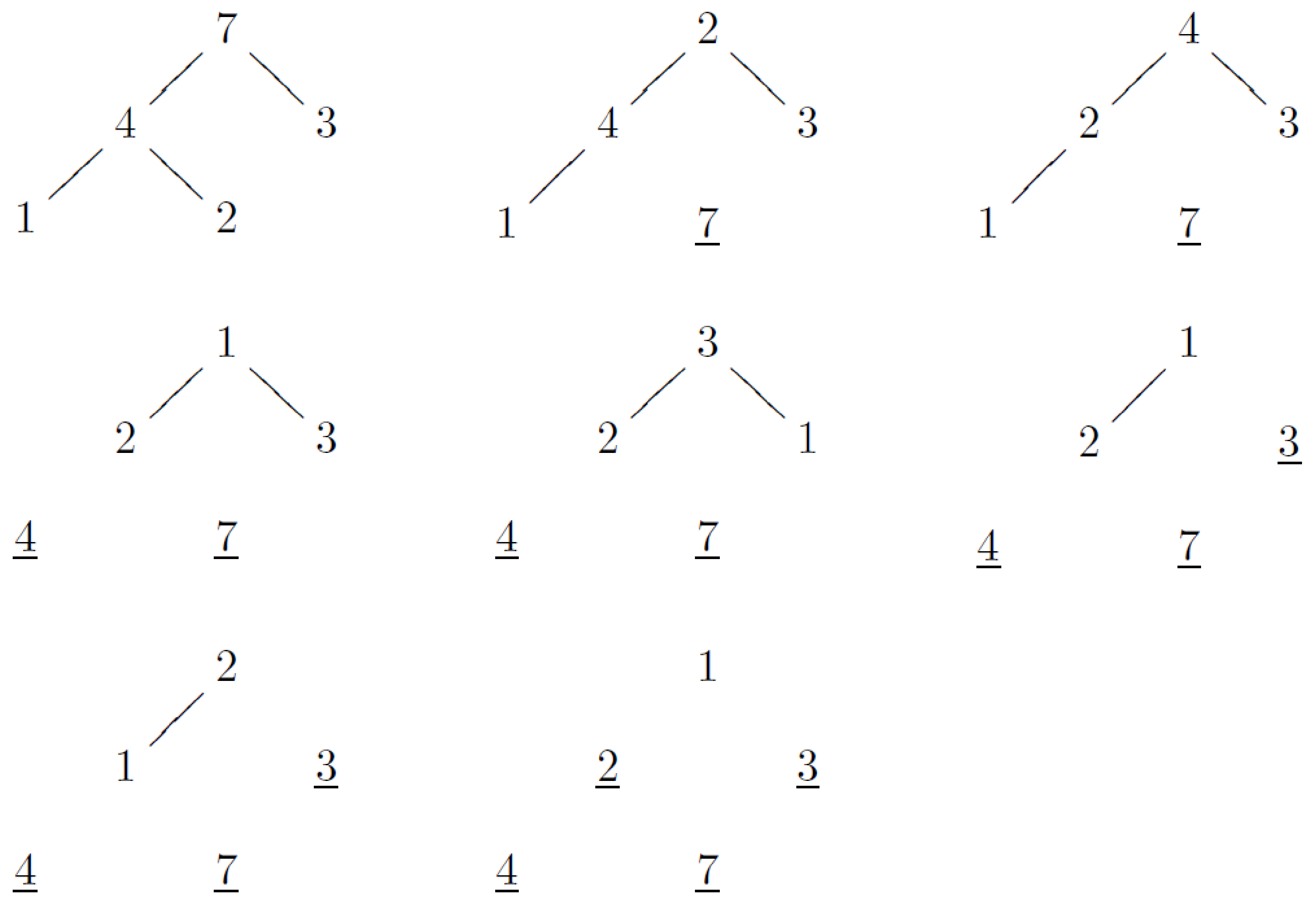
Repeat this discarding process until only one node (the minimum) remains, and is therefore in the correct place in the array.

Features:

$O(n \log n)$ worst case – like merge sort.

Sorts *in place* – like insertion sort.

Example: heapsort



The algorithm

HEAPSORT(A)

```
MAKE-MAX-HEAP(A)
for i = A.heap-size downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

Loop invariant: subarray $A[i+1\,..\,n]$ is sorted, and the remaining elements in $A[1\,..\,i]$ are ≤ than the elements in $A[i+1\,..\,n]$.

Running time

- MAKE-MAX-HEAP takes $O(n)$
- The for -loop is executed $O(n)$ times.
- Exchange operation takes $O(1)$.
- MAX-HEAPIFY takes $O(\log n)$. Total time: $O(n \log n)$.

Priority queues [CLRS 6.5]

A Priority queue is an *abstract data structure* for maintaining a set of elements, each with an associated value called a *key*.
Max-priority queues give priority to the elements with larger keys,
Min-priority queues give priority to the elements with smaller keys.

Operations supported by a max-priority queue:
1. INSERT(S, x, k) inserts element x with key k into set S.
2. MAXIMUM(S) returns the element of S with the largest key.
3. EXTRACT-MAX(S) removes and returns the element of S with the largest key.
4. INCREASE-KEY(S, x, k) increases value of x's key to k. Requires k to be at least as large as x's current key value.

Operations supported by a Min-priority queue supports INSERT(S, x), MINIMUM(S), EXTRACT-MIN(S) and DECREASE-KEY(S, x, k).

Implementation by unordered-sequence

Store the elements e and their keys k (as pairs(e, k)) in an unordered sequence, implemented as an array or a *doubly-linked list*.

Implement $\textsc{Insert}(S, e, k)$ by inserting (e, k) at the end of the sequence; takes $O(1)$ time.

Implement $\textsc{Extract-Max}(S)$ by inspecting all elements of the sequence and removing the maximum; takes $\Theta(n)$ time.

We can do better with a heap implementation!

Implementation by heap

A heap offers a good compromise between insertion and extraction. Both operations take $O(\log n)$ time.

For simplicity, in the following, we identify the element with its key.

Finding the maximum

HEAP-MAXIMUM(A)

return $A[1]$

Time: $\Theta(1)$

Extracting maximum

Check that the heap is non-empty.

Make a copy of the maximum element (root).

Make the last node in the tree the new root.

HEAPIFY the array, but *less the last node*.

Return the copy of the maximum.

HEAP-EXTRACT-MAX(A)

if $A.heap\text{-}size < 1$
    error "heap underflow"
$max = A[1]$
$A[1] = A[A.heap\text{-}size]$
$A.heap\text{-}size = A.heap\text{-}size - 1$
MAX-HEAPIFY(A, 1)
return max

Time: $O(\log n)$, where n is the size of the heap.

Increasing key value

Given set S, entry i, and new key value key:

1. Check that key is greater than or equal to i's current value.
2. Update i's key value to key.
3. Traverse the tree upward comparing i to its parent and swapping keys if necessary, until i's key is smaller than its parent's key.

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

```
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[Parent(i)] < A[i]
    exchange A[i] with A[Parent(i)]
    i = Parent(i)
```

Time. $O(\log n)$

Insertion

Given a key k to insert into the heap:

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

Increase the $-\infty$ key to k using HEAP-INCREASE-KEY

HEAP-INSERT(A, key)

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

Time. $O(\log n)$