# Outline

- Motivation: efficient sorting
- Trees
  - Binary Trees
- Heap
- Heapsort

# HeapSort (One Slide)

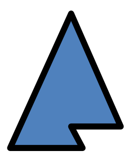
using a data-structure (**DS**) <u>as part of the algorithm design</u>

**DS-Sort**

Unsorted
Elements

**Build-DS**

For each element:
**Maintain DS-property**



Iteratively:
Retrieve maximum
&&
**Maintain DS-property**

Sorted
Elements

DS-property

# (Recall) The Sorting Problem

**Input**: sequence of *n* numbers $<a_1,a_2,..,a_n>$

**Output**: permutation (reordering) of the input

$\sigma(a_i) = b_j$ such that $b_1 \le b_2 \ldots \le b_n$

**Note:** the sets $\{a_1,a_2,..,a_n\} = \{b_1,b_2,..,b_n\}$

**Example**:

- input: <31,41,59,26,41,58>
- output: <26,31,41,41,58,59>

# Motivation For Sorting

- Among the most frequently used algorithms in CS
- Allows:
  - Binary search in *$O(\log N)$* time
  - *$O(1)$* time access to $k^{th}$ largest element
  - Easy detection of any duplicates

# Motivation For Heap Sorting

How to solve the sorting problem efficiently in terms of

- Time complexity
- Space complexity

**Insertion-sort** uses only constant number of extra storing cells (But what is the worst case running time?)

**Merge-sort** * takes *O(nlogn)* but it is not *in-place*

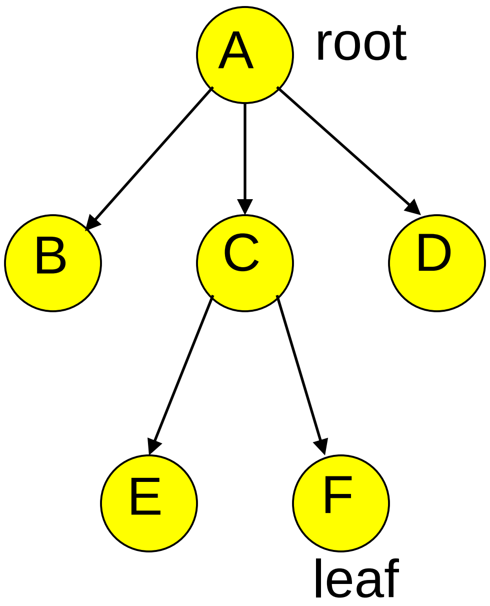
**We are seeking an efficient ADT that both operates *in-place* and takes *O(nlogn)* *time* !**

*coming soon

# Trees

A ***Tree*** is a data structure with the following properties:

- Consisting of layers
- Has a single element in the top layer
- Each element in layer *i* points to elements in layer *i+1*
- Only a single element in layer *i* points to an element in layer *i+1*




How many edges in a tree with *N* nodes?

*N-1* edges

# Tree
# Recursive Definition

A tree is a set of nodes, either

- It is an empty set of nodes, or
- It has one node called the root from which zero or more trees (subtrees) descend

# Tree Terms

***Child:*** B is a child of A iff A points to B

***Parent:*** A is a parent of B iff A points to B

***Sibling:*** A and B are siblings if they have the same parent

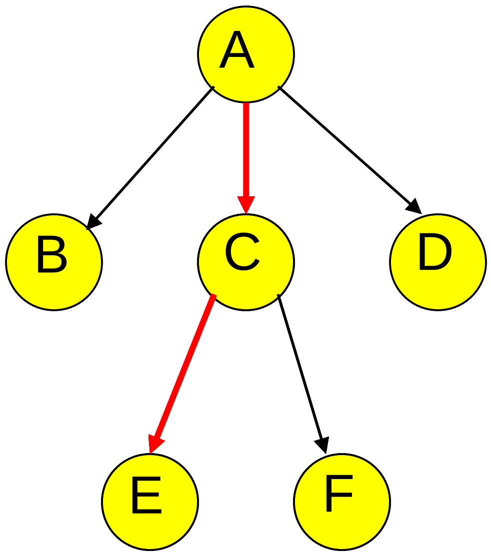
***Root:*** a node with no parent

***Leaf:*** a node with no children

***Path:*** sequence of connected nodes

# Examples of Tree Terms

A is the root
A-F are nodes
AD is an edge (one out of 5)
B,E,F,D are the leaves
A is the parent of C
E,F are children of C
B,C,D are siblings
A-C-E is a path
C is the root of the subtree consisting of C,E,F



# More Tree Terms

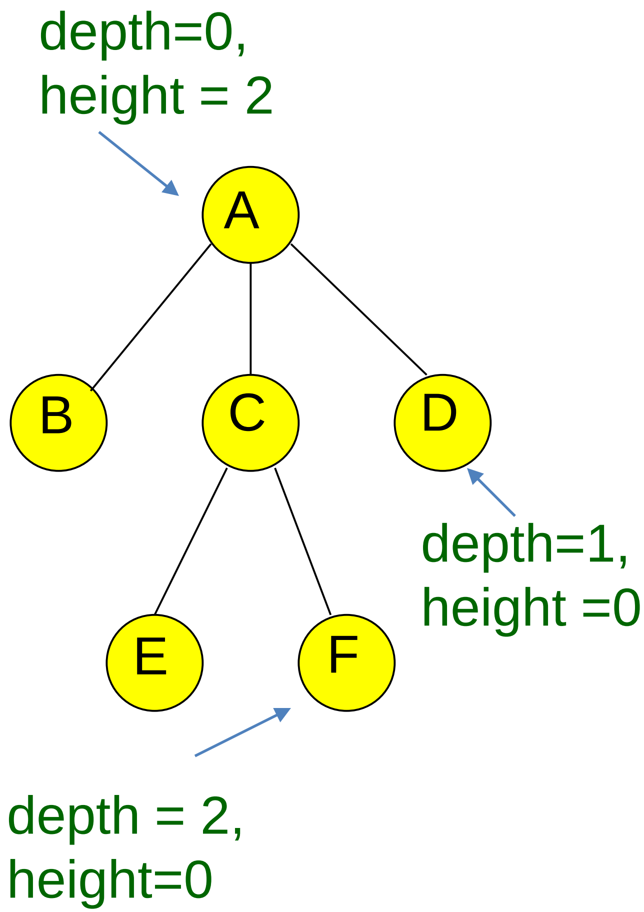
**Length** of a path = number of edges between two nodes

**Depth** of a node *A* = length of path from root to *A*

**Depth of tree** = depth of deepest node

**Height** of node *A* = length of longest path from *A* to a leaf

**Height of tree** = height of the root



# Implementation of Trees

Each node includes:

Option 1: a value + one pointer <u>to each child</u>

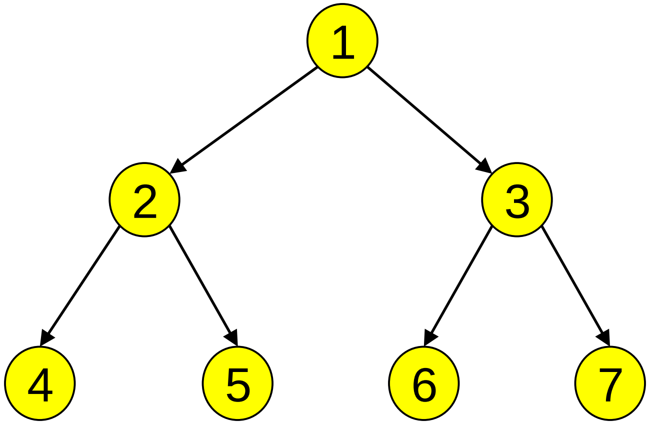
How many pointers should we allocate space for?

Other options?

# Binary Tree

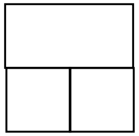
Each node has at most two children

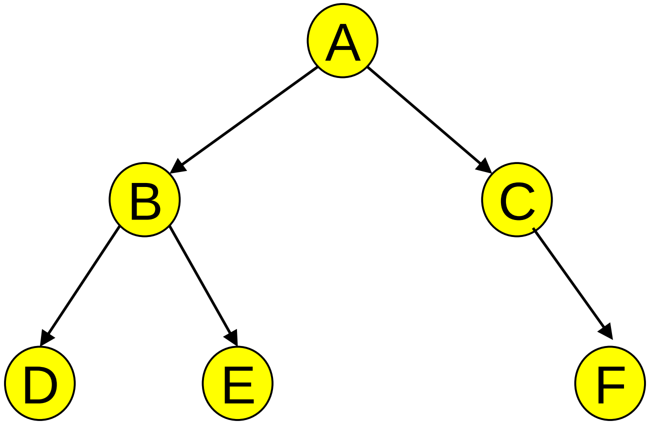
- Popular data structure in computer science
- Will talk about it again latter in the course

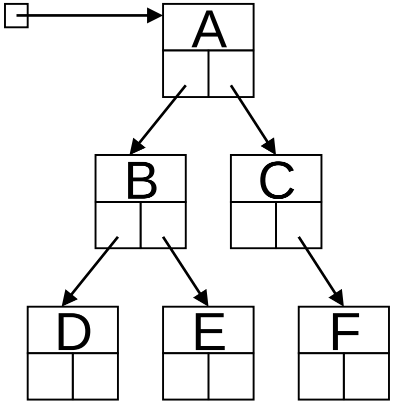


# Implementation of Binary Trees

Each node is implemented by a structure with
value (key) and two pointers (left child & right child)

Left child  Right child



# Binary Tree Trivia

Amongst all binary trees with ***N*** nodes:

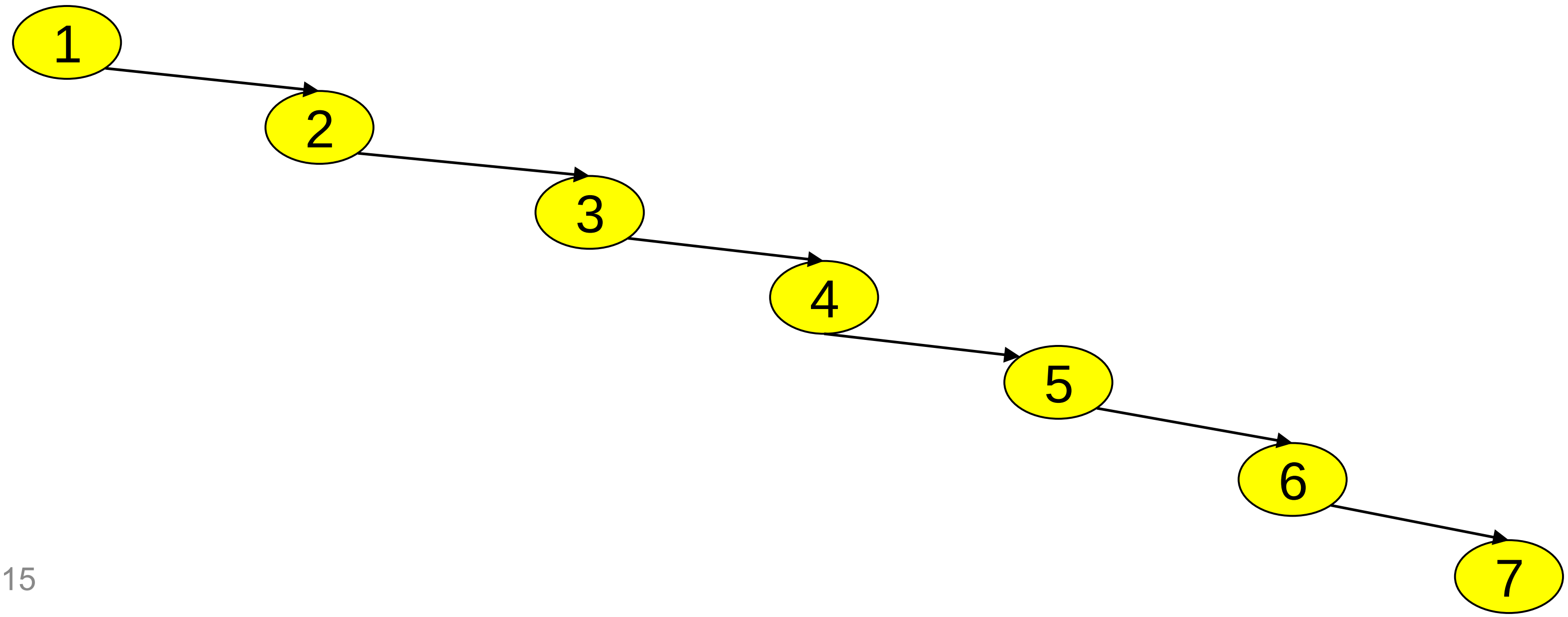
- Which tree has the maximal depth?
- Which tree has the minimal depth?

# Tree With Maximal Depth

Which tree has the maximal depth?

– Degenerate case: a linked list

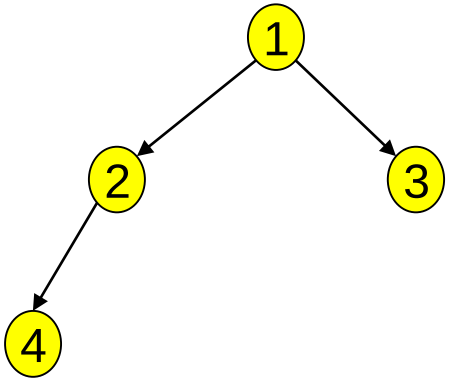
– Depth = *N-1*



# The Tree With Minimal Depth

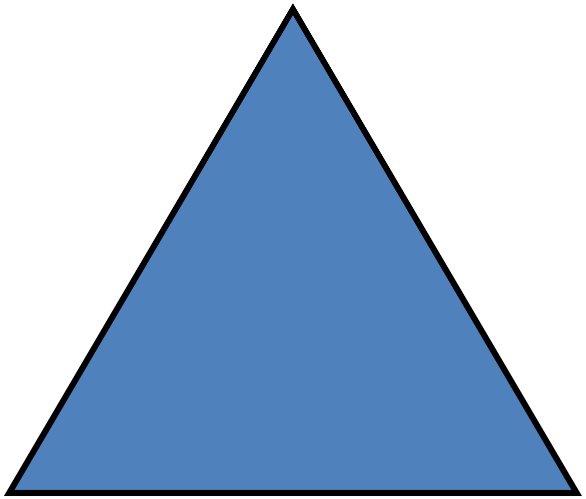
Which tree has the minimal depth?

Is it unique?



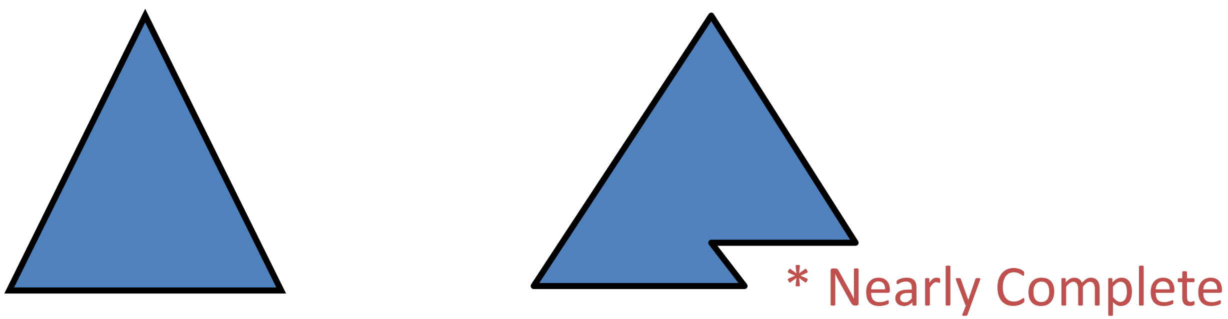
# <u>Complete</u> Binary Tree

<u>Complete</u> Binary Tree: All nodes are in use



# Nearly Complete Binary Tree

(Nearly) Complete Binary Tree: All nodes are in use (except for possibly the right part of the bottom row.)



# <u>Nearly Complete</u> Binary Tree Trivia

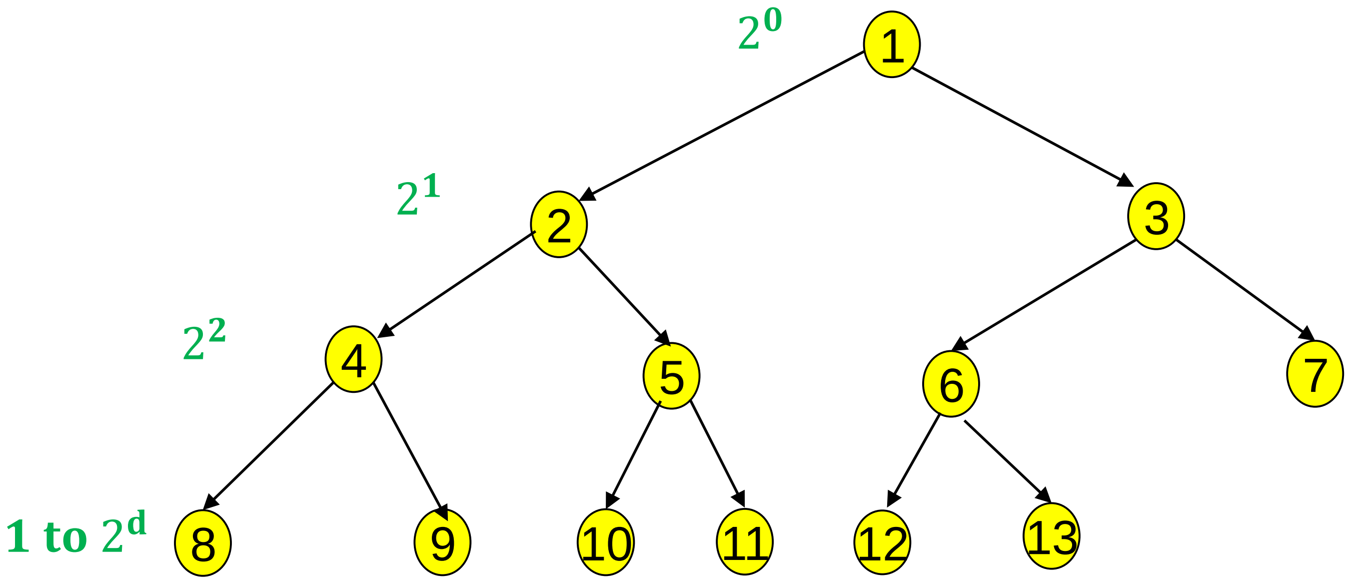
Consider a <u>nearly complete</u> binary tree with ***N*** nodes:

- How many nodes in depth ***i***?
- How many nodes in height ***j***?
- What is the height(=depth) of the tree?

# Nodes No. In Depth *i*

Consider a <u>nearly complete </u>binary trees with ***N*** nodes:

- How many nodes in depth ***i***?



# Nodes No. In Height *j* : Intuition

Consider a ~~nearly~~ complete binary tree with ***N*** nodes:

- How many nodes in total in a tree with depth d?

$$\boldsymbol{a}\left(\frac{r^n - 1}{\boldsymbol{r} - 1}\right)$$

$$\mathbf{N} = \sum_{i=0}^{d} 2^i = 2^{d+1} - 1$$

→ adding a complete level " ~ doubles" the number of elements

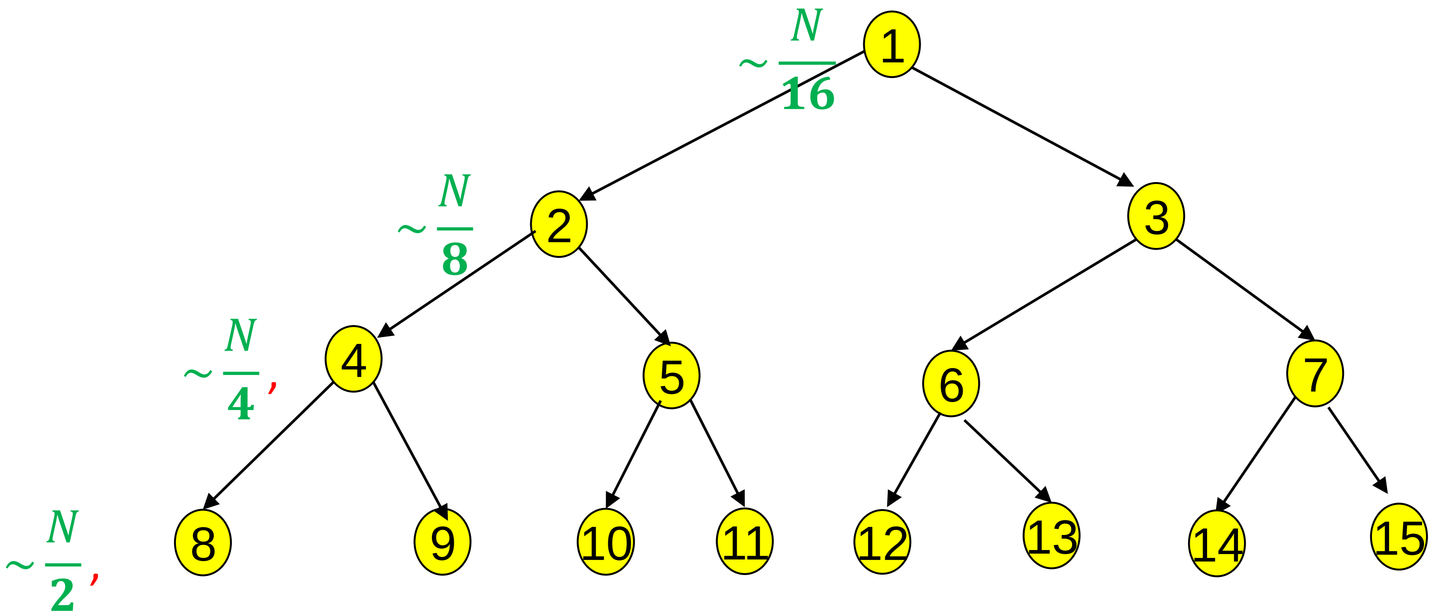
Equivalently: How many elements in the deepest (it is a complete) level **h=0**?

Answer: approximately half

# Nodes No. In Height *j* : Intuition

Consider a ~~nearly~~ complete binary trees with ***N*** nodes:

- How many nodes in height ***j***?

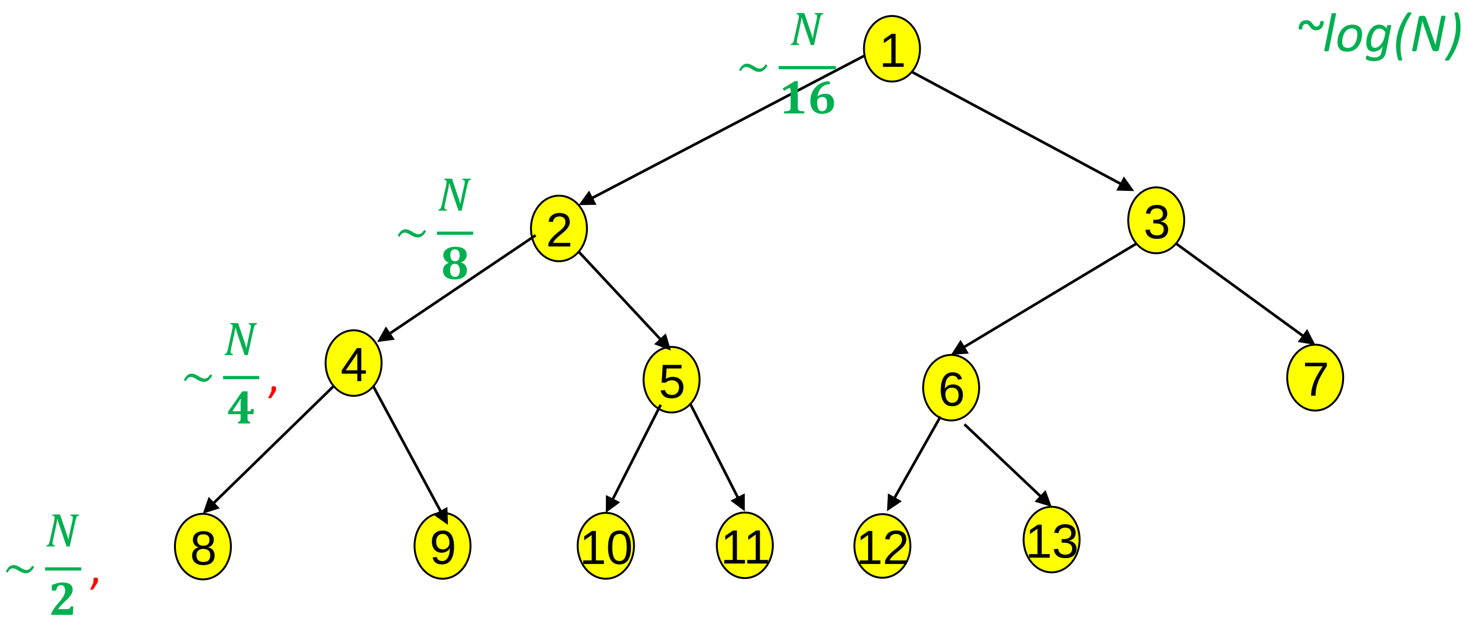



Formally: Height ***j*** includes at most $\left\lceil \frac{N}{2^{j+1}} \right\rceil$

# Tree Height : Intuition

Consider a nearly complete binary trees with ***N*** nodes:

- What is the height(=depth) of the tree?



# Tree Depth Analysis: Formally

Using the above observation:

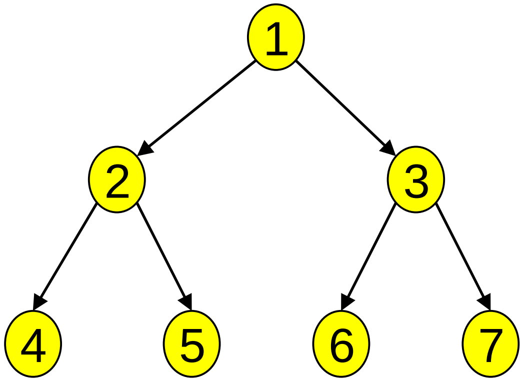
– At depth $i$, there are $2^i$ nodes

– At depth $d$ (tree depth), **there may be 1 to $2^d$ nodes**

Let N denote the total number of nodes:

$$\sum_{i=0}^{d-1} 2^i + 1 \leq N \leq \sum_{i=0}^{d-1} 2^i + 2^d$$

$$2^d \leq N \leq 2^{d+1} - 1$$



# Depth Analysis

$$2^d \le N \le 2^{d+1} - 1$$

- From the left inequality: $d \le \log N$
- From the right inequality: $\log(N+1) \le d+1$

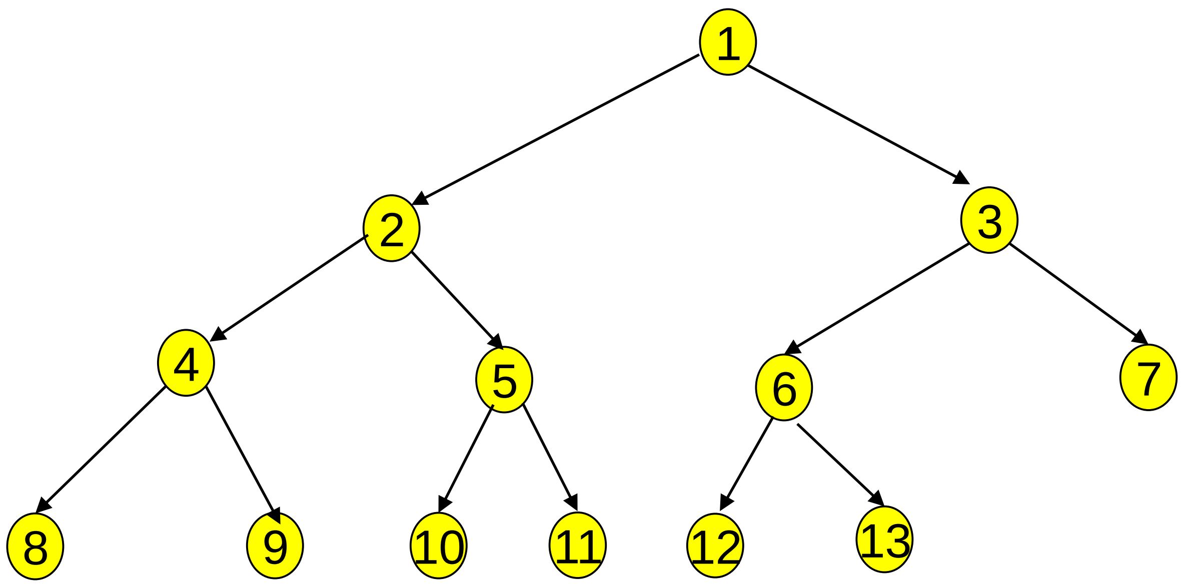
  and: $\log(N) < d+1$

In total: $\log(N) - 1 < d \le \log N$

Equivalently: $d = \lfloor \log_2(N) \rfloor$

# Nearly Complete Binary Trees

- How many nodes in depth ***d***? $\sim 2^d$
- How many nodes in height ***h***? $\frac{N}{2}, \frac{N}{2^2}, \ldots$ $\sim\left\lceil \frac{N}{2^{h+1}} \right\rceil$
- What is the height(=depth) of the tree? $\sim\lfloor log(N) \rfloor$

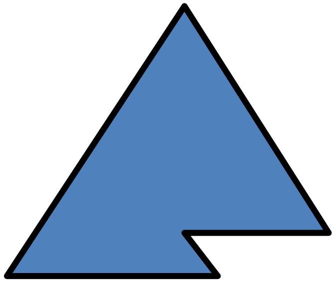


# Heap

It is convenient to view a (binary) heap
as a nearly complete binary tree.

The Max-heap property:

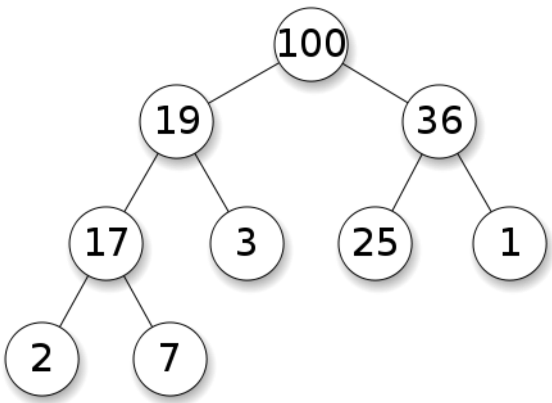
the key of the parent is equal or greater than the key of the children.



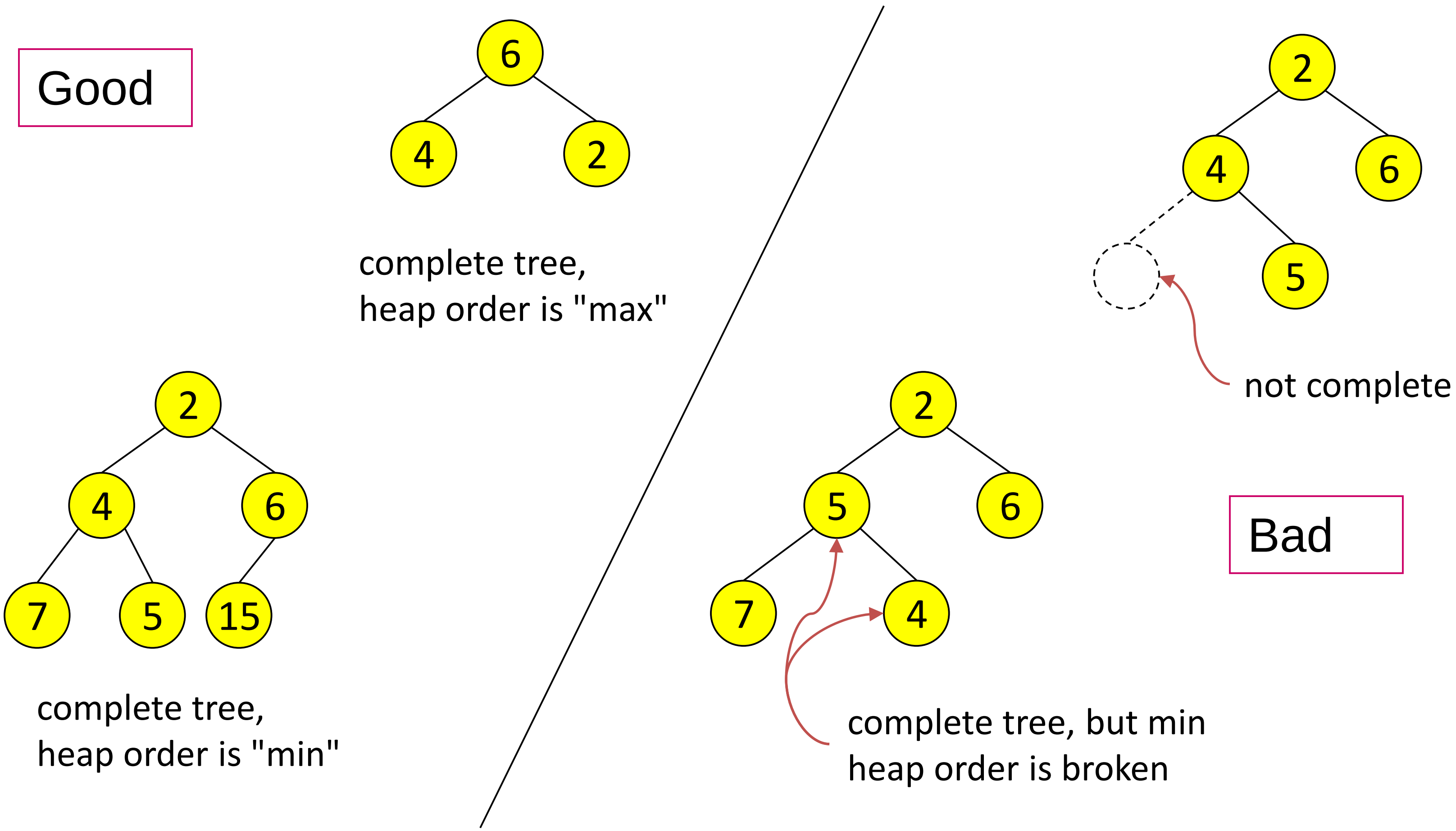
# Heap Properties

- Binary heaps provide limited ordering information
- Each *path* is sorted, but *siblings are not sorted*
- Binary heap is ≠ binary search tree (future …)

This is a binary heap



# Examples



Good
complete tree,
heap order is "max"
complete tree,
heap order is "min"
not complete
Bad
complete tree, but min
heap order is broken

# Min / Max Heap

We focus on Max-heaps.

By symmetry, the statements, procedures and definitions are relevant for Min-heaps.

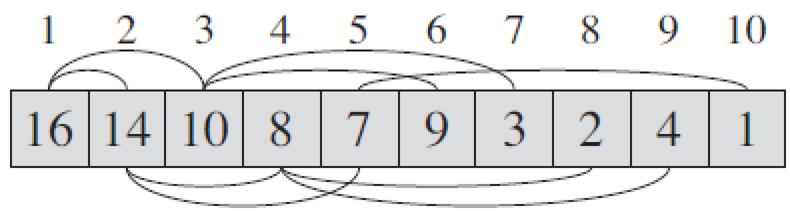
# ADT: Max-Heap

Operations:
Empty (T)
Insert (T,x)
Min(Q)
Del_Max(Q)

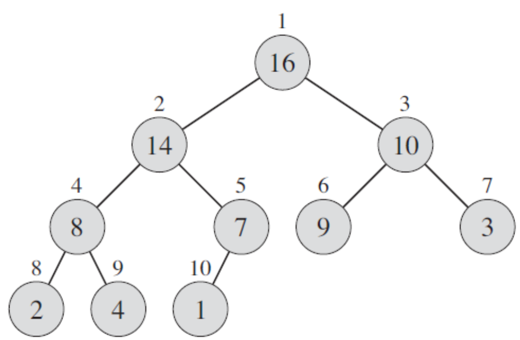
Max-heap property: the key of the parent is equal or greater than the keys of both children

# Heap Implementation

Binary Heap as <u>array</u> object:




The rules (of how to insert and delete elements) allow us to <u>view it</u> as a <u>nearly complete binary tree</u>.



# Heap as a Complete Binary Tree

- Basic operations run in time that is proportional to the root height*
- The height is $\Theta$(logN) (as proved above):

The Heap ADT is potentially useful for sorting in order O(NlogN) instead of O($N^2$)

* We will prove or demonstrate it later in this lecture

# Binary Heap Space Analysis

- Space needed for heap of at most *MaxN* nodes: *O(MaxN)*
  - An array of size *MaxN*, plus a variable to store the current size *N of the heap*

# HeapSort (One Slide)

Heap

using a data-structure (~~DS~~) as part of the algorithm design

HeapSort

~~DS-Sort~~

Unsorted Elements

BuildHeap

~~Build-DS~~

For each element:

~~Maintain DS-property~~

Maintain Heap-property

Iteratively:

Retrieve maximum

&&

~~Maintain DS-property~~

Maintain Heap-property

Sorted Elements

~~DS-property~~

Heap-property

# Max-Heapify Pseudocode

```
MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

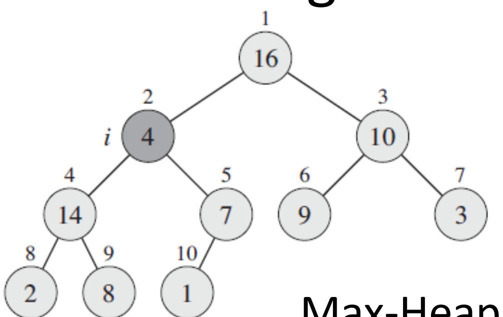
Largest =
Index of the node
with highest values

Percolate into deeper level

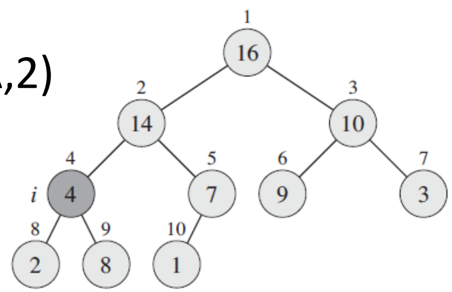
Cormen 6.2

# Max-Heapify Visualization

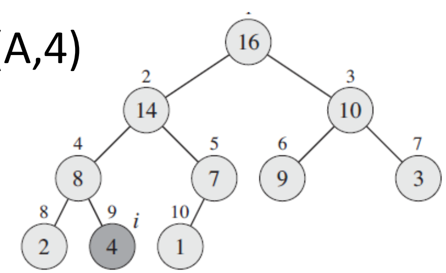
Running time of Max-Heapify on a node of <u>height ***h***</u> *is* ***O(h)***




Max-Heapify (A,2)




Max-Heapify (A,4)




Cormen 6.2

Max-Heapify (A,9)

# Build-Heap Pseudocode

BUILD-MAX-HEAP$(A)$

$A.heap\text{-}size = A.length$

**for** $i = \lfloor A.length/2 \rfloor$ **downto** 1

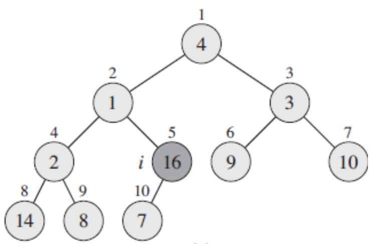
  MAX-HEAPIFY$(A, i)$

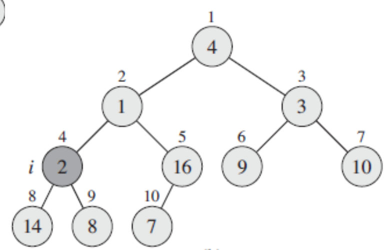
$\left\lfloor \frac{N}{2} \right\rfloor$ is the index of the last non- leaf

# Build Heap Visualization

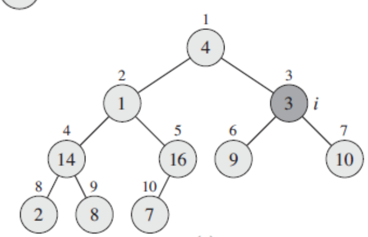
Idea: using Max-Heapify in a bottom-up manner

from $\left\lfloor \frac{N}{2} \right\rfloor$ to 1 (e.g. from 5 to 1, when N=10)

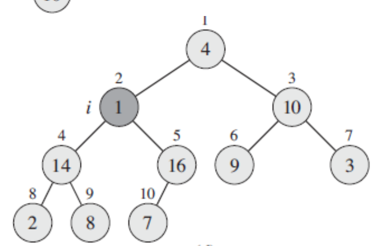




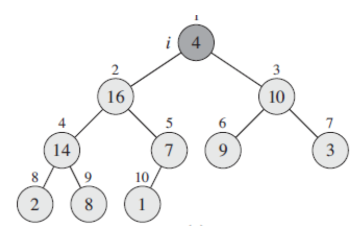
(b)



(c)

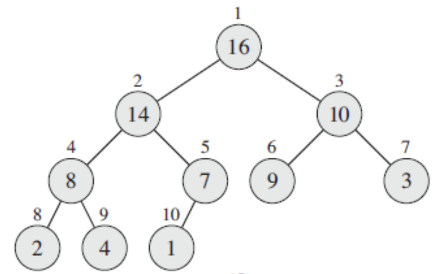


(d)



(e)

Result: Heap



(f)

Cormen 6.3

# Correctness of Build Heap

- For $n=1$, a tree with a single node is a heap.

Hence, trees rooted at $\left\lfloor \frac{n}{2} \right\rfloor < i \leq n$ are heaps.

- For $i \leq \left\lfloor \frac{n}{2} \right\rfloor$, children are heaps.

Hence, after calling Max-Heapify we obtain heap.

# Time Complexity Build Heap

Claim: The running time of Build-Max-Heap is *O(n)*

Proof :

**BuildHeap** performs **Heapify** on nodes with height $j \geq 1$

**Heapify** on node of height $j$ costs at most $cj$

Recall:

- at most $\left\lceil \frac{N}{2^{j+1}} \right\rceil$ elements at level in height **j**
- the height of the tree is at most $\lfloor log(N) \rfloor$

$$\sum_{j=1}^{\lfloor log(n) \rfloor} \left\lceil \frac{N}{2^{j+1}} \right\rceil cj$$

# Time Complexity of Build Heap

$$\sum_{j=1}^{\lfloor log(N) \rfloor} \left\lceil \frac{N}{2^{j+1}} \right\rceil cj \leq$$

$$\sum_{j=0}^{\lfloor log(N) \rfloor} \frac{N}{2^j} cj = cN \sum_{j=0}^{\lfloor log(N) \rfloor} \frac{j}{2^j}$$

$$< cN \sum_{j=0}^{\infty} \frac{j}{2^j} = 2cN$$

$$\sum_{j=0}^{\infty} \frac{\mathrm{j}}{2^{\mathrm{j}}} = 2$$

# Heapsort Pseudocode

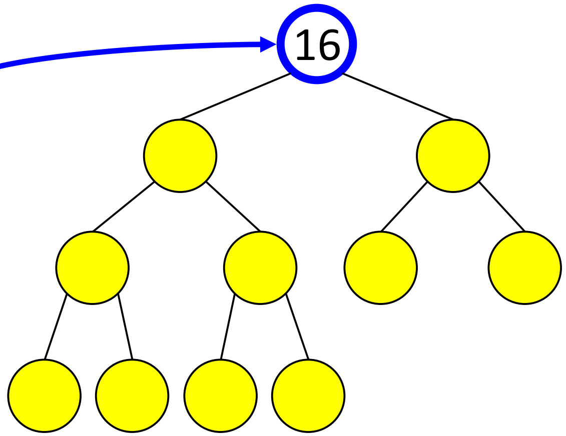
Iteratively:
Retrieve maximum
&&
**Maintain DS-property**

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

# Retrieve Maximum

*Retrieve maximum:* Easy!

- Return root value (that is *A[1]* )
- Run time = ?




**Maintain DS-property:** harder!

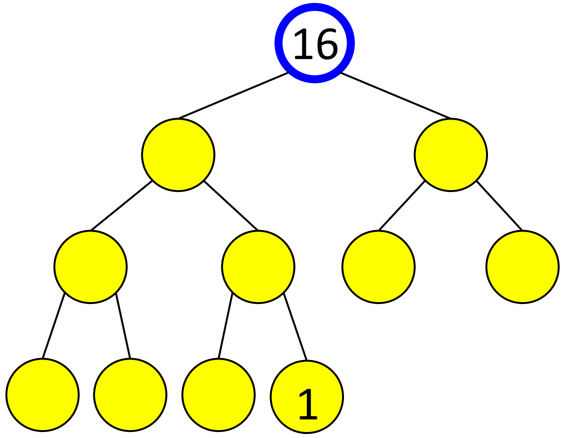
- why ?

# The Challenge of Retrieve Maximum

**<u>Challenge:</u>** Following retrieve maximum (remove root), the tree will have one less node → temporarily it is not a tree

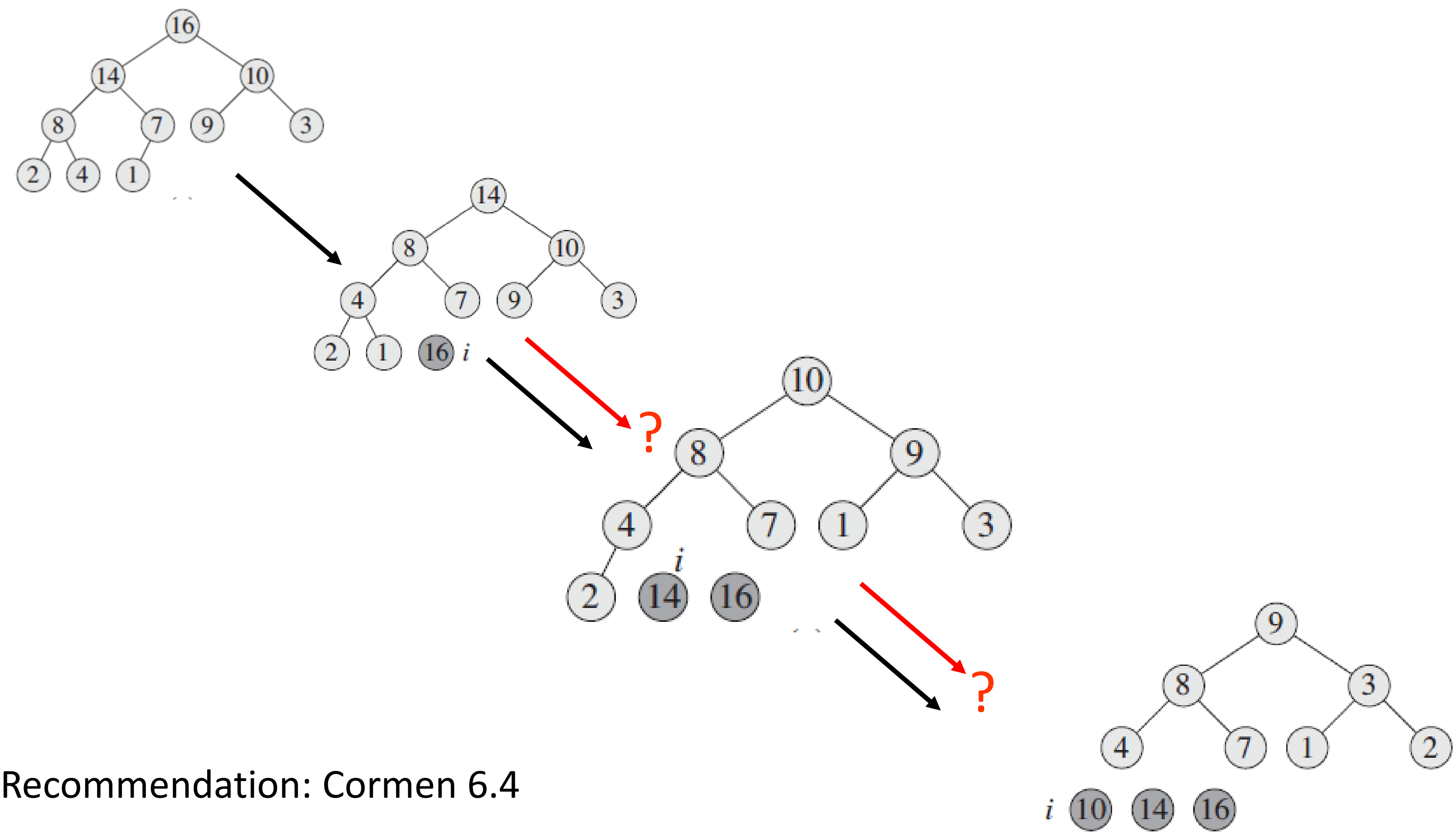
**<u>Solution:</u>** Replace the root with the "last element" yet not sorted

Why is it <u>not a Heap</u>?




Iteratively:
Retrieve maximum
&&
**<u>Maintain DS-property</u>**

# Heapsort Visualizaion




Recommendation: Cormen 6.4

# Time Complexity HeapSort

Claim: The running time of HeapSort is *O(nlogn)*

Proof :

- `HeapSort` performs `Max-Heapify` on n-1 nodes
- Each `Max-Heapify` costs at most ***clogn***

In total, `HeapSort` costs at most ***cnlogn.***

# Summary – What Is Heap?

**The Heapsort Idea:** using a data structure that

- returns the maximum in ***O(1)***
- maintains the heap property in ***O(logn)***.

Heap ~ Nearly Complete Tree

Max-heap property: the key of the parent is equal or greater than the keys of both children.

# Summary – HeapSort Analysis

using a data-structure(**DS**) <u>as part of the algorithm design</u>

**HeapSort**

***O(nlogn)***