Analysis of Algorithms

COMP.4040, Summer 2019

Chapter 6: Heap and its Application

By: Sirong Lin, PhD

University of Massachusetts Lowell

UMASS LOWELL

Learning with Purpose

HW6

HW6 (heap):

- Due 06-17-2019 (M), BEFORE the class starts

Outline

Data structure — Heap

Operations on Heap

Heapsort

Priority Queues

Heaps

Basics

Operations

Heap Sort

Priority Queues

Heap

a container of objects that have *keys*

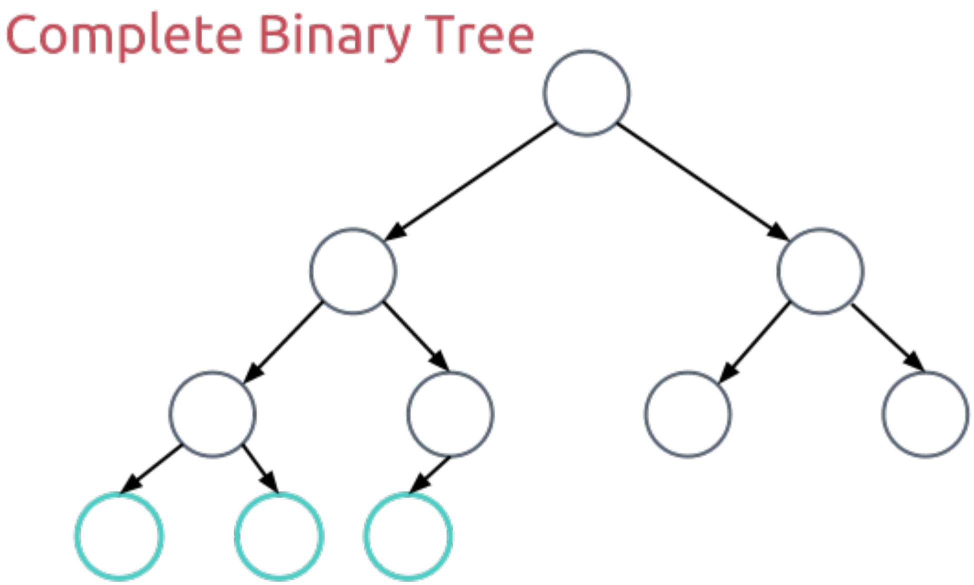
keys: usually numbers that can be *compared*, e.g., SSN, timestamp, weight of an edge

Objects	Keys
employee records	SSN
events	the time at which that an event is meant to occur
network edges	length or weight of an edge

Heap — two views

1st view: <u>tree view</u> (visualization):

a (nearly) **complete binary tree***, each node contains a key

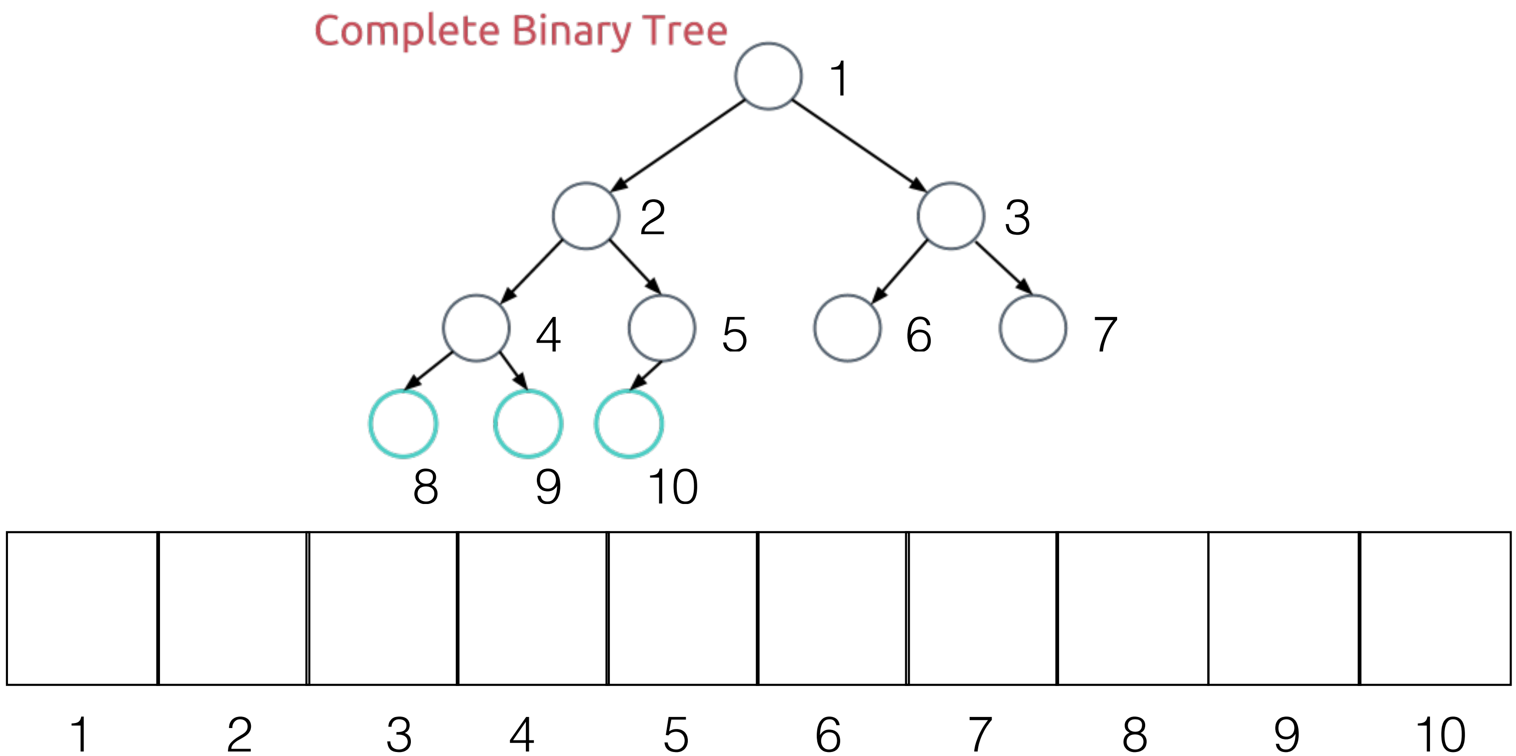


*a <u>complete binary tree</u> is a tree in which all levels are complete full, except of the last level (leaves), where all nodes are placed on the left first

Heap — two views (Cont'd)

2nd view: array view (implementation):

tree nodes are stored in an array, and satisfy a ***heap property*** (depending on the heap type)



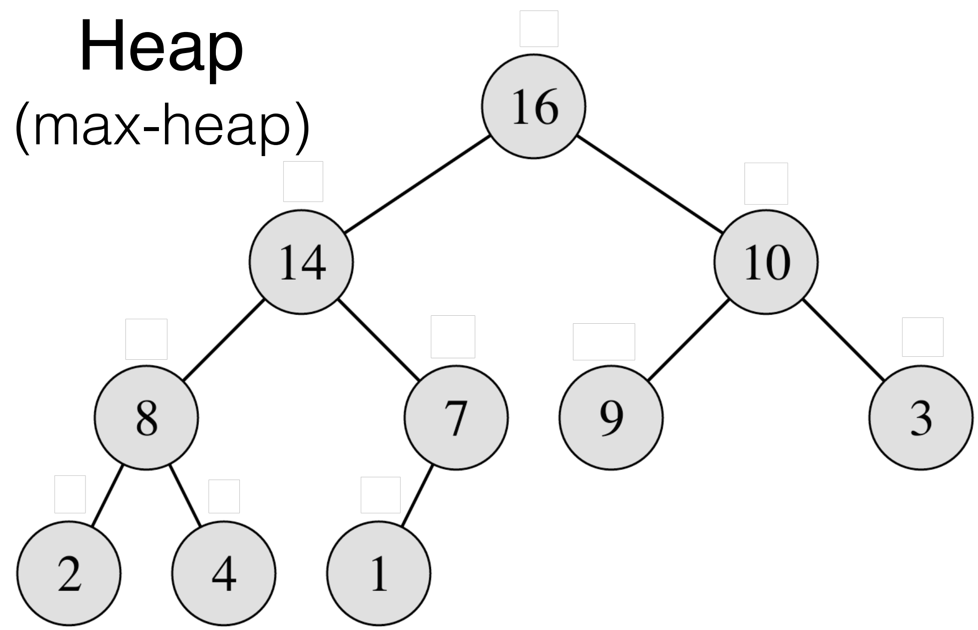
Heap — max-heap

max-heap:

every node is no larger than its parent (or at most the value of the parent)

every node ≥ all its descendants

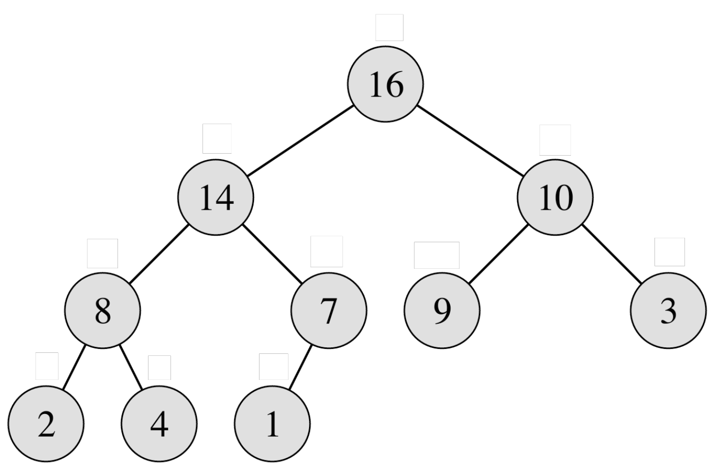
root node contains the maximum value key



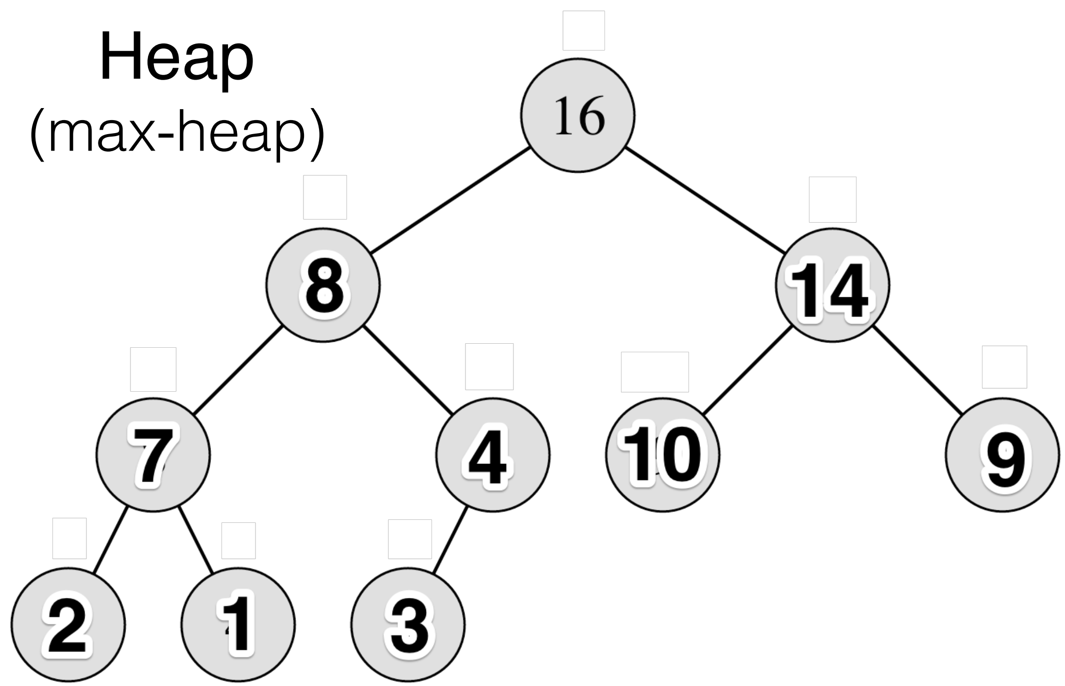
Heap — max-heap (Cont'd)

alternative heap

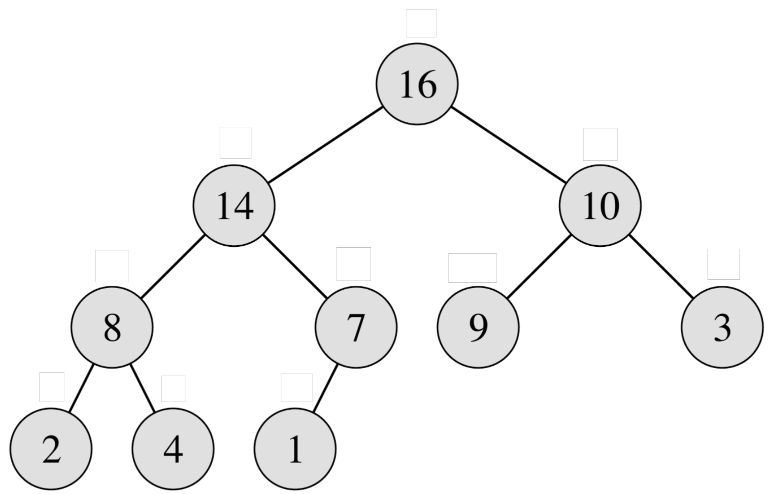
key values can be arranged differently to generate alternative heaps

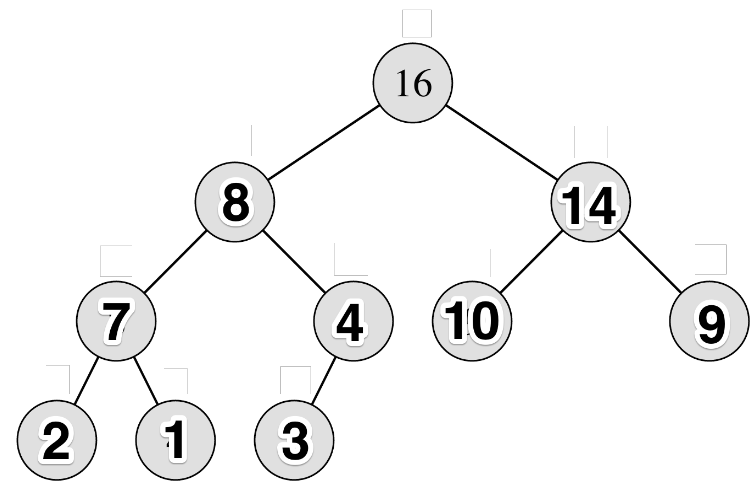


Heap
(max-heap)



Heap — max-heap (Cont'd)





for any n-node tree, the **shape is same**

(just the value of each node may be different)

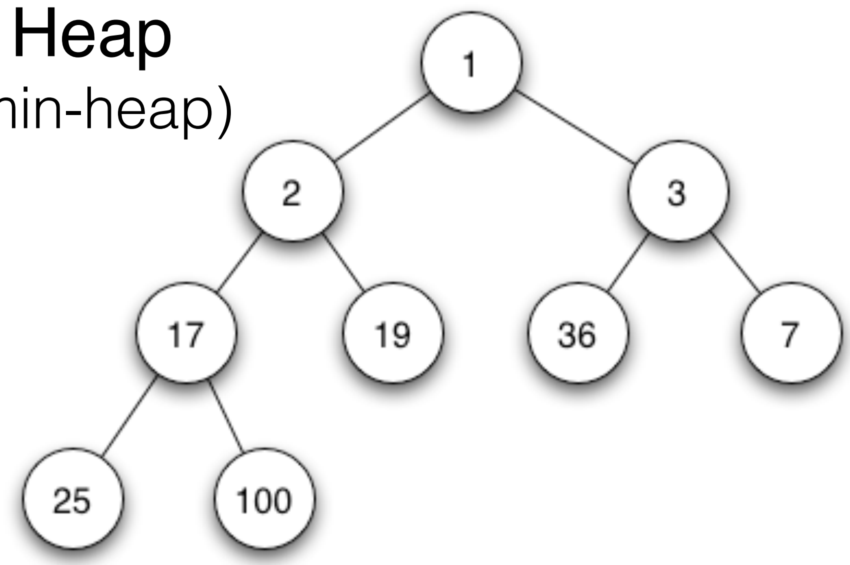
Heap — min-heap

min-heap

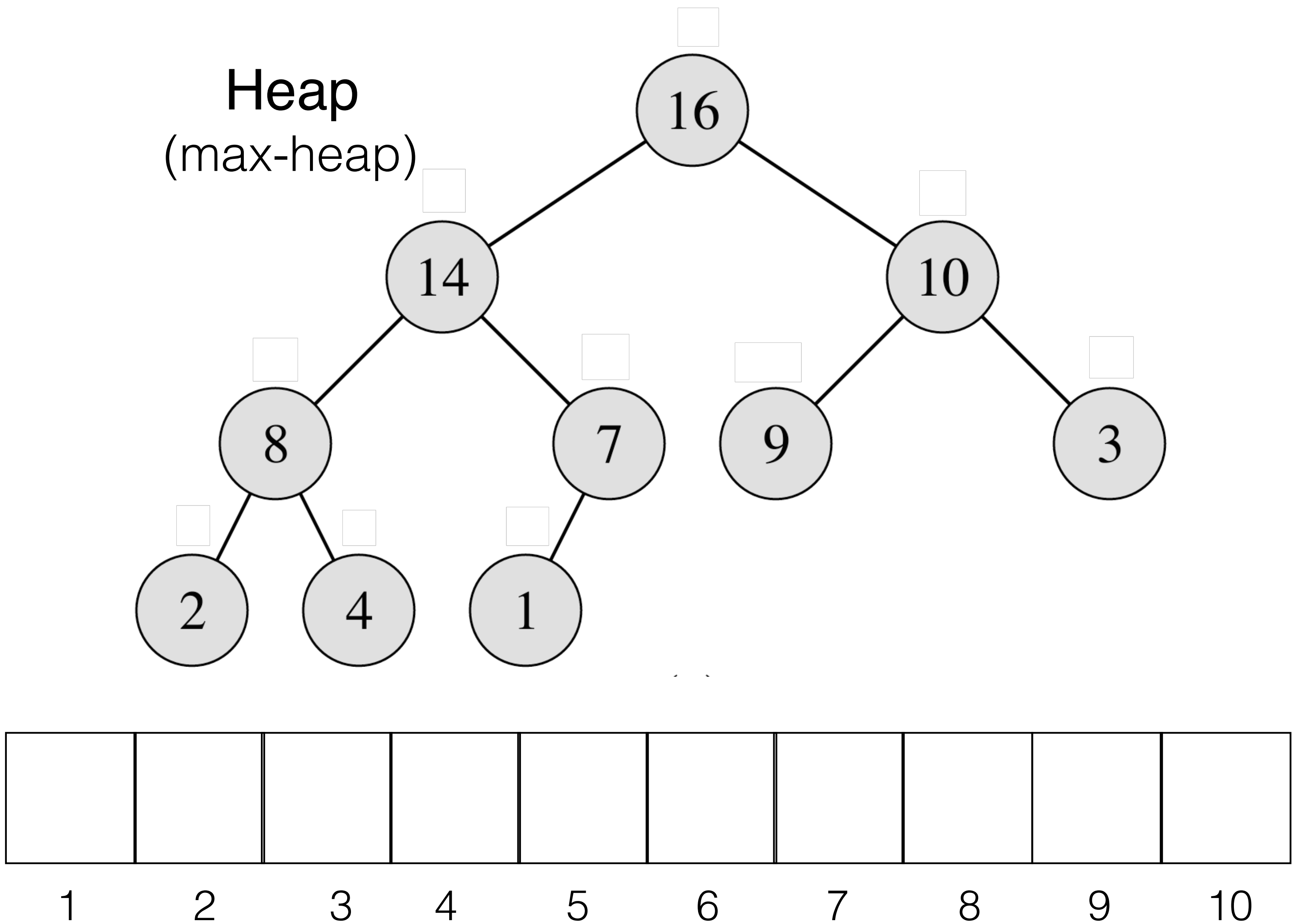
every node is no less than its parent (or at least the value of the parent)

root node contains the minimum value key

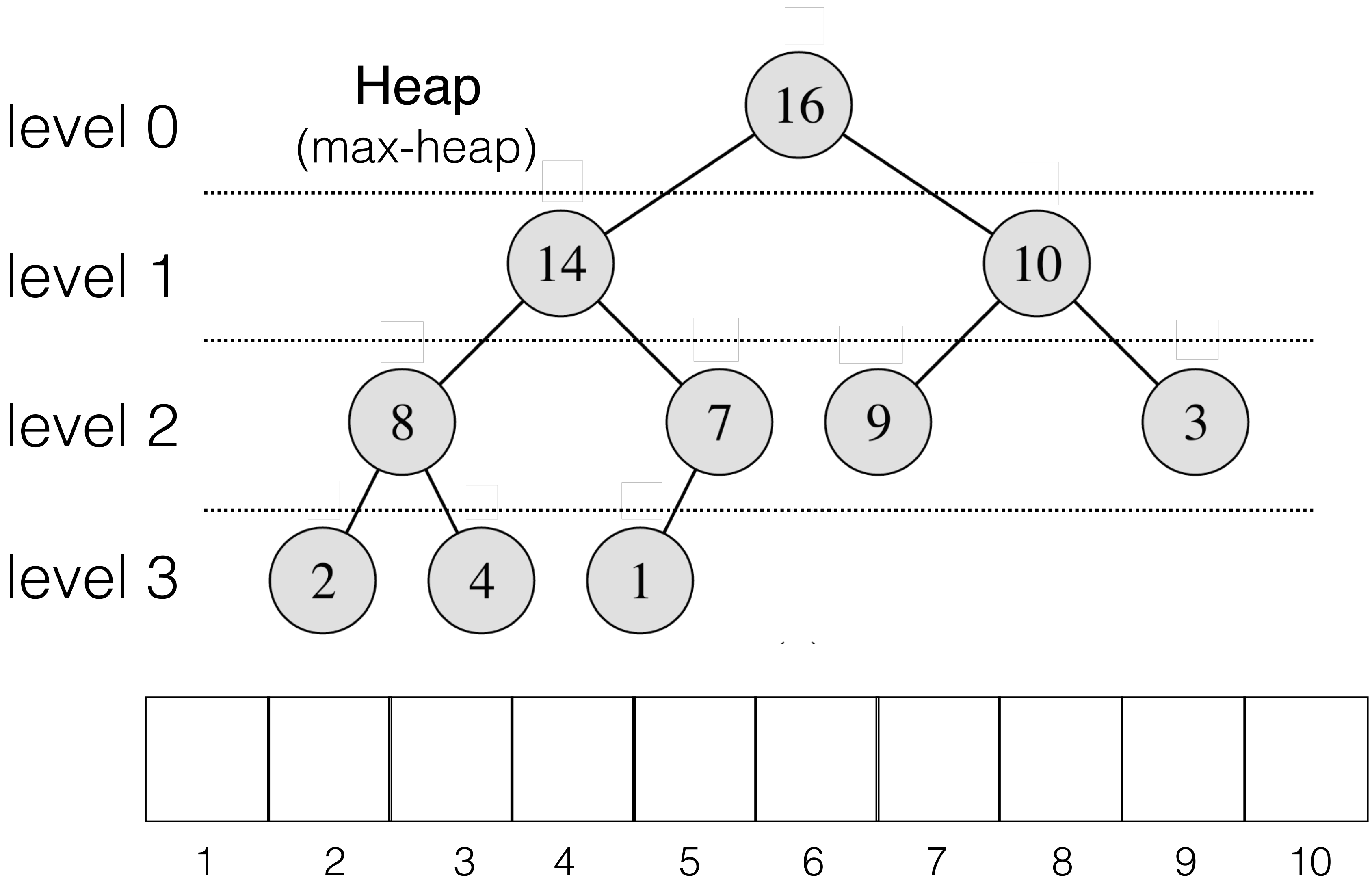
Heap
(min-heap)



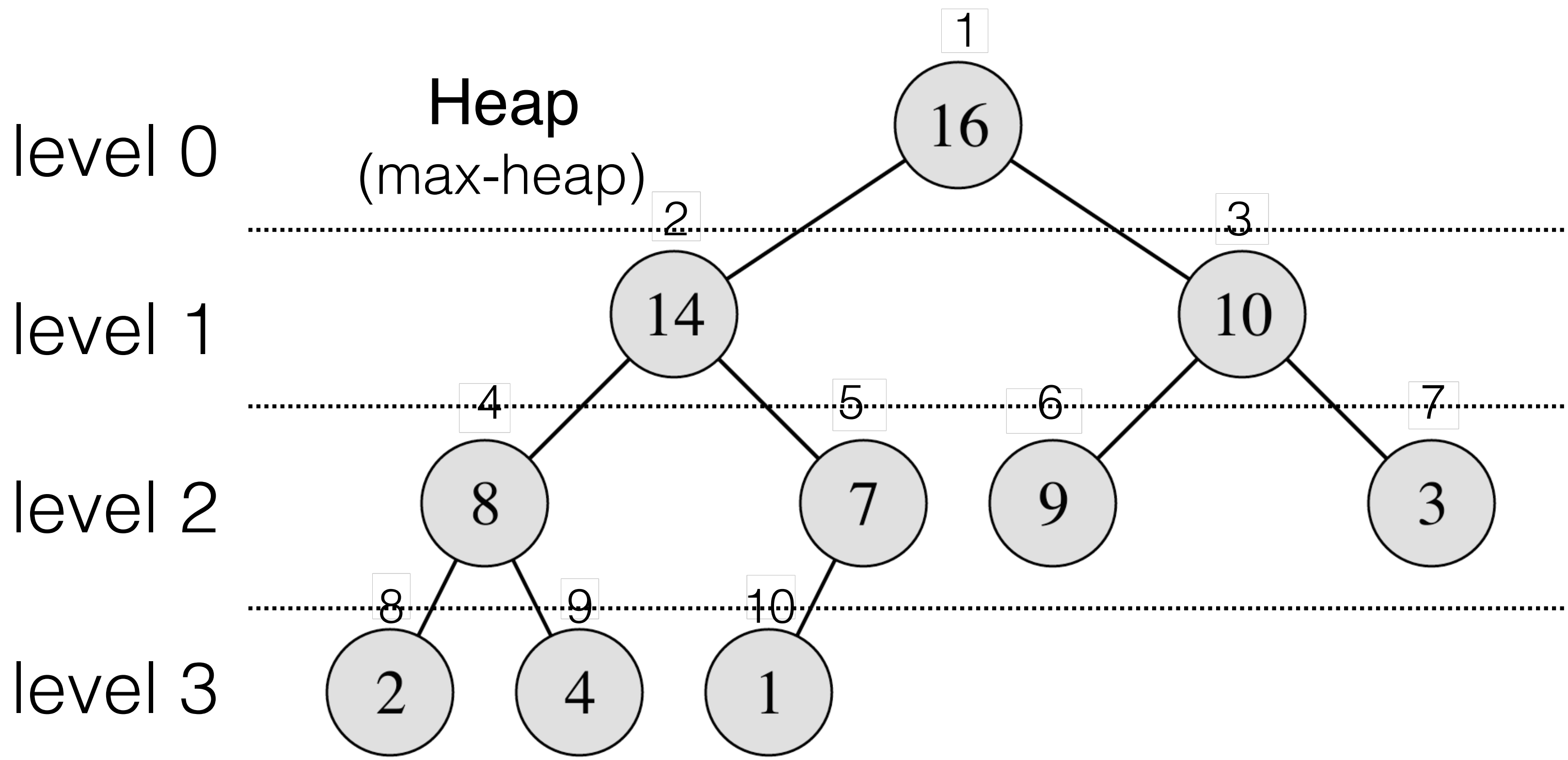
Heap — map tree to array



Heap — map tree to array (Cont'd)

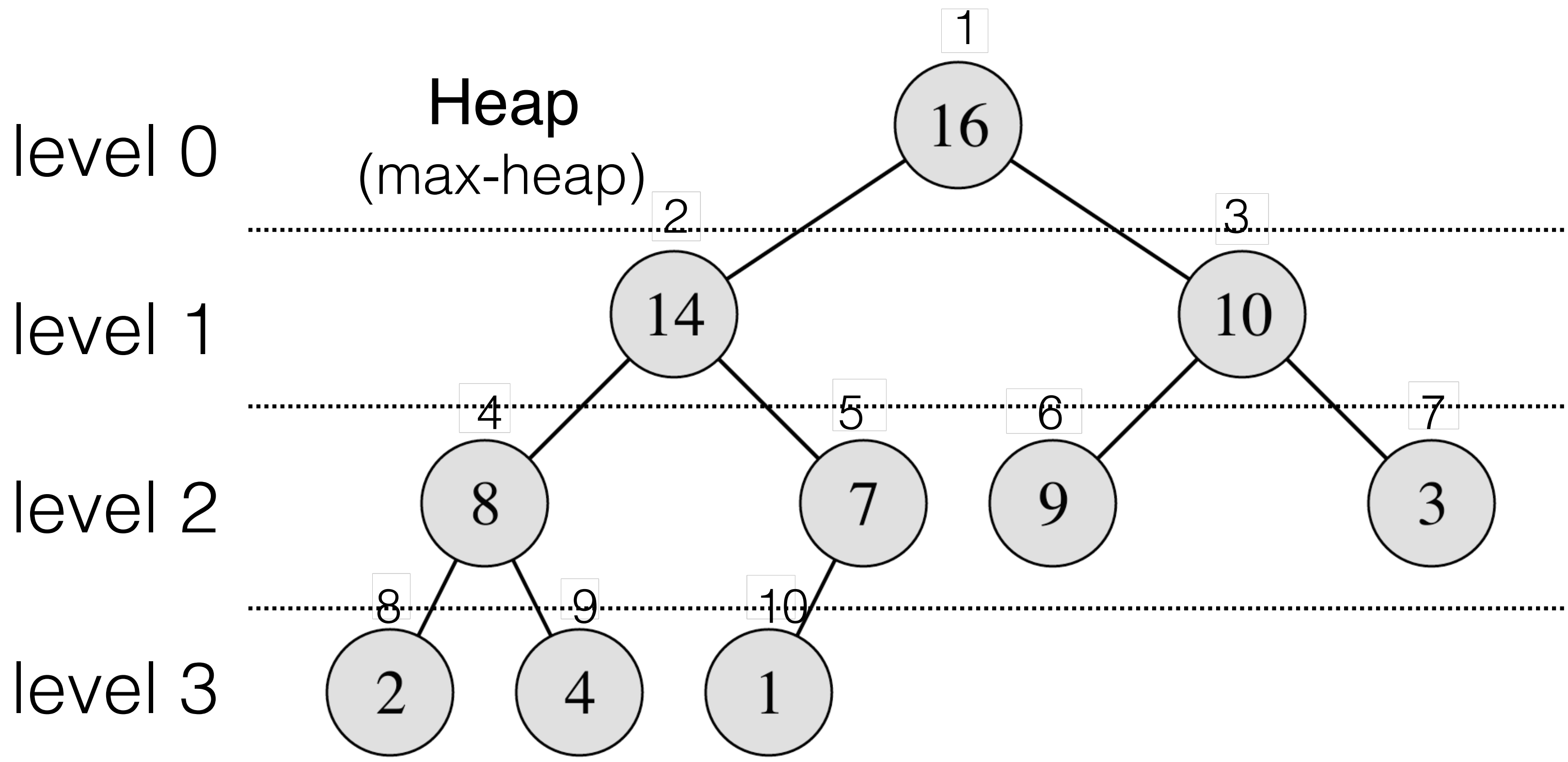


Heap — map tree to array (Cont'd)



1	2	3	4	5	6	7	8	9	10

Heap — map tree to array (Cont'd)



16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

Heap — map tree to array (Cont'd)

Can you think of another heap for these numbers {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}?

16	8	14	7	4	10	9	2	1	3
1	2	3	4	5	6	7	8	9	10

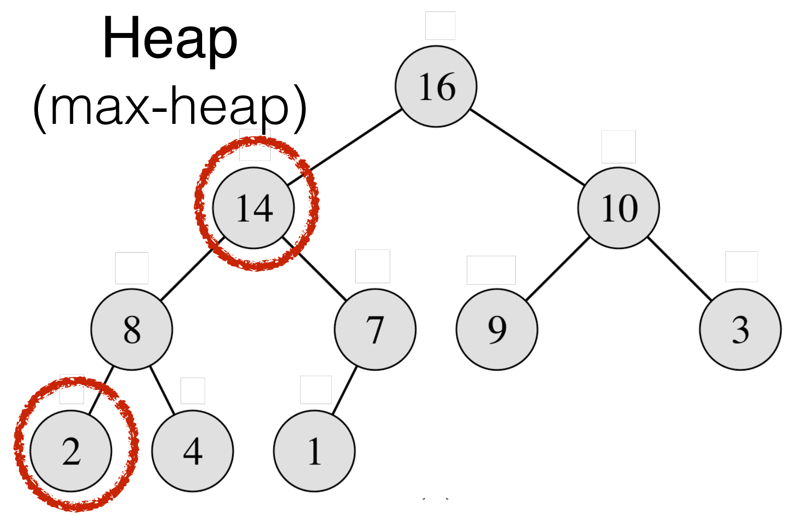
Heap — heap property

height of a node

# of edges on a longest simple path from the node down to a leaf

e.g., height of node 14 is 2,
height of node 2 is 0

Heap
(max-heap)



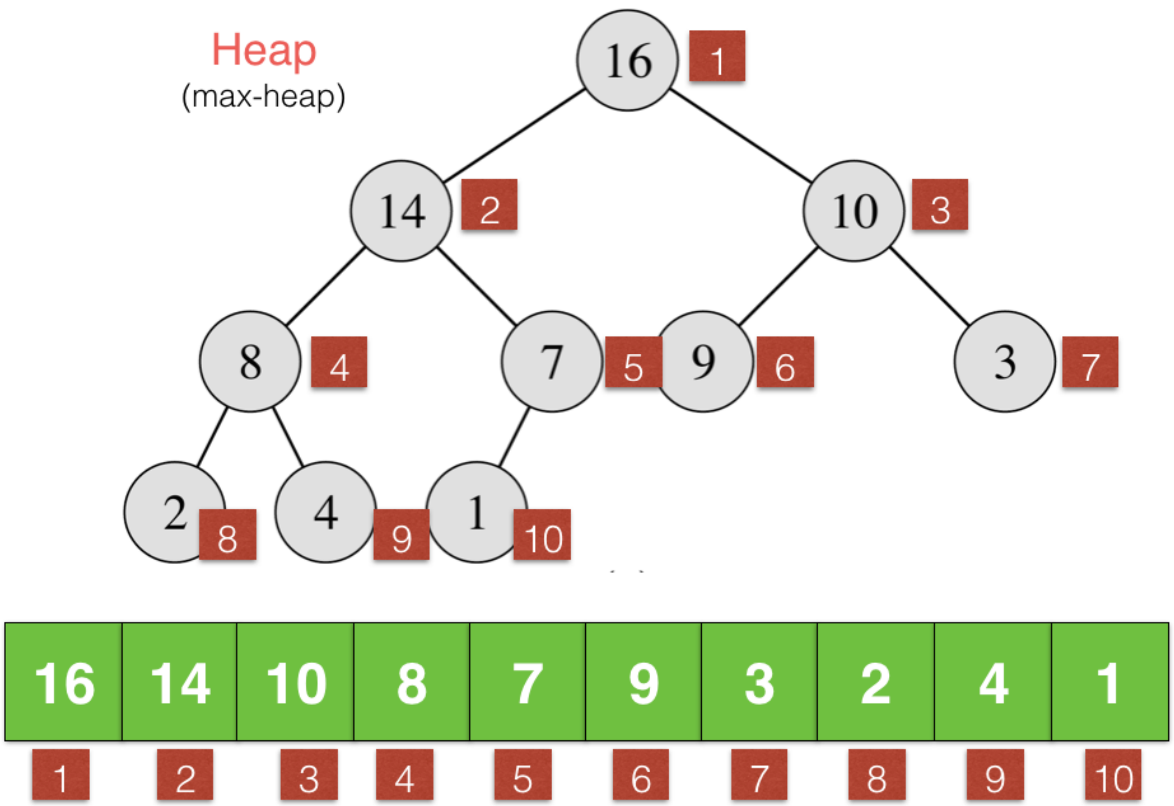
height of heap, *h*

h is the height of the root, e.g., 3, for given heap

$h = \lfloor \lg n \rfloor$, for a n-node (binary) tree, $\Theta(\lg n)$

heaps can be **k-ary tree** instead of binary

Heap — tree nodes in array



Observe the following nodes in the array, what is the index?

- Root of tree
- Parent of $A[i]$
- Left child of $A[i]$
- Right child of $A[i]$

Heap — tree nodes in array (Cont'd)

Root of tree is $A[1]$

Parent of $A[i]$: $A[\lfloor i/2 \rfloor]$

Left child of $A[i]$: $A[2i]$

Right child of $A[i]$: $A[2i+1]$

It's easy to find parent and children, no need to use pointers (save space, no traversing)

Heap — subroutines

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$

implementation:
could use bit shifting

Heap — heap property

max-heap property:

A[parent(i)] ≥ A[i]

min-heap property:

A[parent(i)] ≤ A[i]

i is the index for every node except of the root in the array

Heap — heap property (Cont'd)

How many nodes at each level?

level 0: 1 node

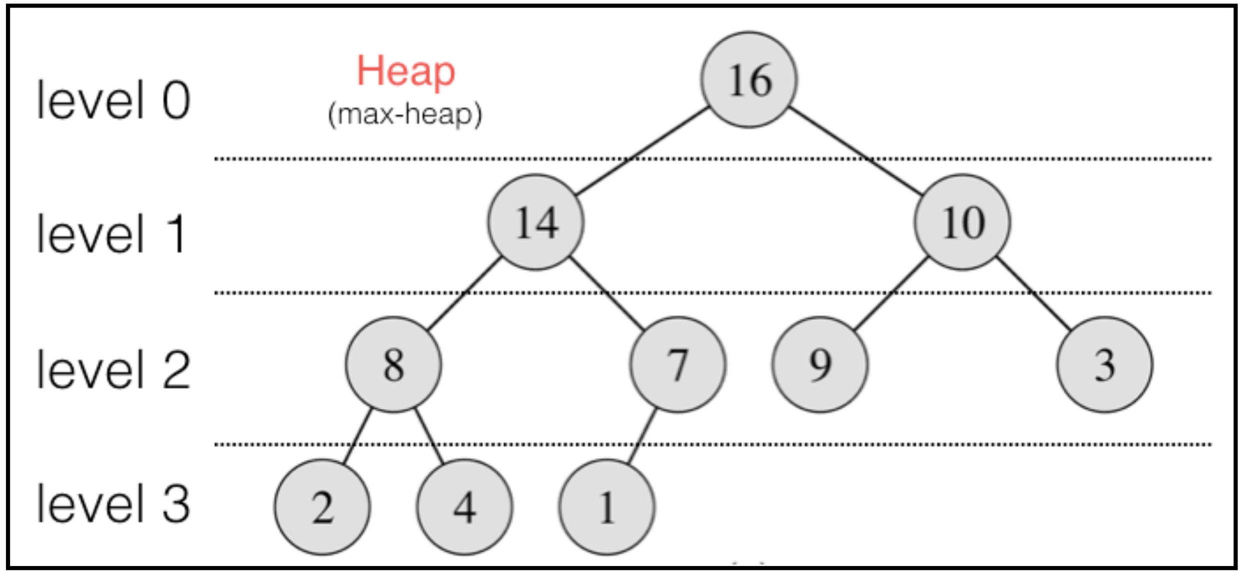
level 1: 2 nodes

level 2: 2^2=4 nodes

…

level i-1: 2^{i-1} nodes

level i (last level): 2^i nodes (at most)



in a n-node binary tree, how many nodes have

height of 0, 1, 2 ?

Heap — heap property (Cont'd)

In a n-node binary tree, how many nodes have

an height of *0*? — $n/2$

they are leave nodes, either at the last level or the 2nd last last level

an height of *1*? — $n/2^2$

height of *2*? — $n/2^3$

…

an height of *h*? — $n/2^{h+1}$

*proof in notes

Heaps

- Basics
- **Operations**
- Heap Sort
- Priority Queues

Heap — Operations

Operations on heap:

- HEAPIFY
- building a heap
- insertion
- deletion (extract-max for max-heap, extract-min for min-heap)
- increasing key value for max-heap /decreasing key value for min-heap

Heap — Maintain property

It is very important to **maintain the heap property** when perform operations:

maintain a complete binary tree as much as possible

maintain the max-heap property for max-heap, so that each node should be no less than its children (or the min-heap property for min-heap)

Operations on Heap — **MAX_HEAPIFY**

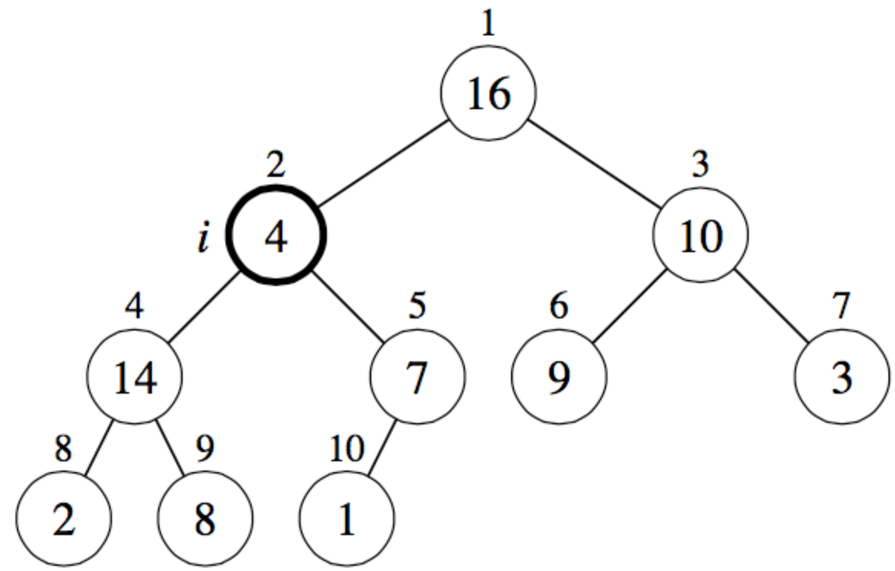
HEAPIFY: restore the heap property for the following case —

binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but A[i] might be smaller than its children

Operations on Heap — MAX_HEAPIFY (Cont'd)

Example:

Node #2 (i.e., 4) violates the heap property, how should we restore the heap property? (notes)



Operations on Heap — MAX_HEAPIFY (Cont'd)

HEAPIFY:

- compare A[i] with two children (A[LEFT(i)] and A[RIGHT(i)])
- swap A[i] with the larger child if needed
- repeat (1) & (2) of comparing and swapping down the heap, until subtree rooted at i is max-heap; this step can be done by a recursive call to HEAPIFY on that subtree

Operations on Heap — MAX_HEAPIFY (Cont'd)

Running Time of MAX_HEAPIFY?

height of the heap, O(lg *n*) (because the height of the heap is the floor of lg*n*), so at most process lg*n levels, with constant comparisons and swaps at each level*

Operations on Heap — MAX_HEAPIFY (Cont'd)

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

Operations on Heap —
Building a max-heap

example: build a max-heap for the following unsorted array <4, 1, 3, 2, 16, 9, 10, 14, 8, 7>

step 1: map it to a tree view

step 2: apply MAX_HEAPIFY to subtrees rooted for <u>all internal nodes</u>, i.e., 16, 2, 3, 1, 4, respectively

can we do the reversed order of 4, 1, 3, 2, 16?

(see notes for details, or Figure 6.3 on P158)

Operations on Heap — Building a max-heap (Cont'd)

Build max_heap Algorithm

$$
\begin{array}{l}
\textsc{Build-Max-Heap}(A, n) \\
\quad \textbf{for } i = \lfloor n/2 \rfloor \textbf{ downto } 1 \\
\qquad \textsc{Max-Heapify}(A, i, n)
\end{array}
$$

Running Time of building a max-heap:

simple upper bound: O($n$lg$n$), not a tight bound

Tight bound Analysis of building a max-heap

Let us consider the operations we need to do for each node:

- # of nodes of height *h*: $n/2^{h+1}$
- for each node, at most *h* levels of processing (constant # of comparisons and swaps)
- T(n) = $$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n) .$$
- use formula A.8

therefore, we can build a max-heap from an unordered array in linear time

Tight bound Analysis of building a max-heap (Cont'd)

lesson learned from the analysis:

- the lowest three levels in the tree contain 87.5% nodes
- when designing algorithms that operate on trees, it is important to be most efficient on the bottommost levels of the tree since that is the most of the weight of the tree resides

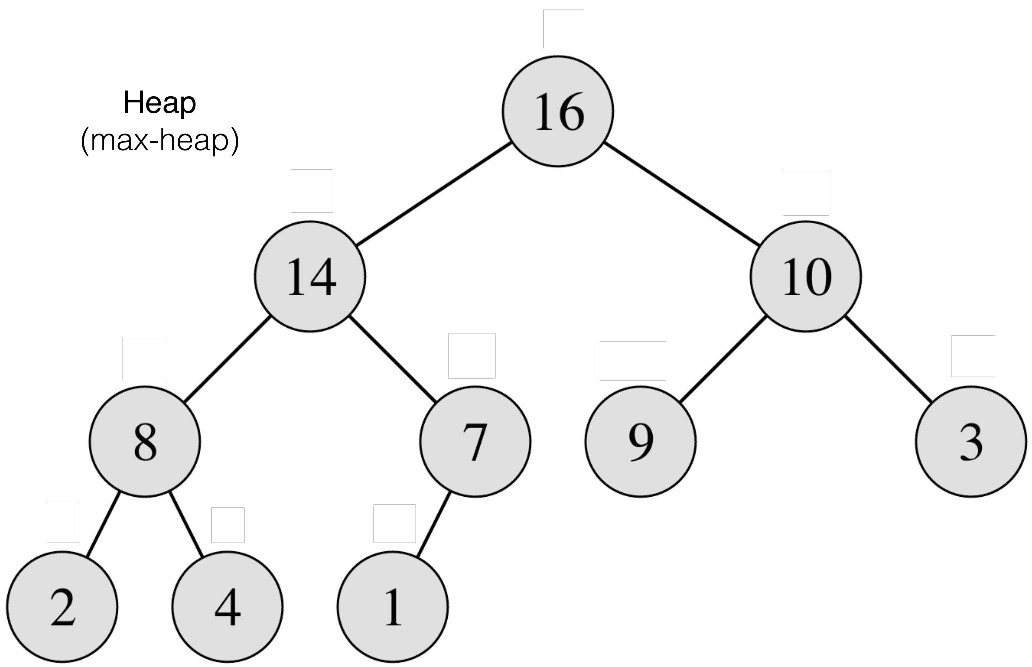
Heap Operations — **extract max**

Extract maximum for max-heap: removes and returns the largest key

- after the maximum element (root) is removed, still a max-heap?
- how to restore the complete the tree shape and heap property?
  - find a new root (easiest one is to use the last element)
  - re-heapify the heap, with one fewer node

Heap Operations — **extract max**

example: extract max from the given heap (notes)



similar process for extracting min for a min-heap

Heap Operations — extract max (Cont'd)

- Extract maximum for max-heap Algorithm

```
HEAP-EXTRACT-MAX (A, n)
    if n < 1                        //make sure the heap is not empty
        error "heap underflow"
    max = A[1]
    A[1] = A[n]                     //make the last node in the tree the new root
    n = n-1                         //decrement the heap size n (i.e., remove the last element),
                                    //assume it is passed by reference
    MAX-HEAPIFY(A, 1, n)            //re-heapify the heap, with one fewer node
    return max
```

Heap Operations — extract max (Cont'd)

Extract maximum for max-heap Analysis

$T(n)$ = constant time + time for MAX-HEAPIFY

$= T(1) + O(\lg n) = O(\lg n)$

Heap Operations — **find maximum/minimum**

Find maximum/minimum: return the largest key for max-heap, or smallest key for min-heap

it is A[1]

Running time: $\Theta(1)$

HEAP-MAXIMUM(A)

return $A[1]$

Heap Operations — **increase key value**

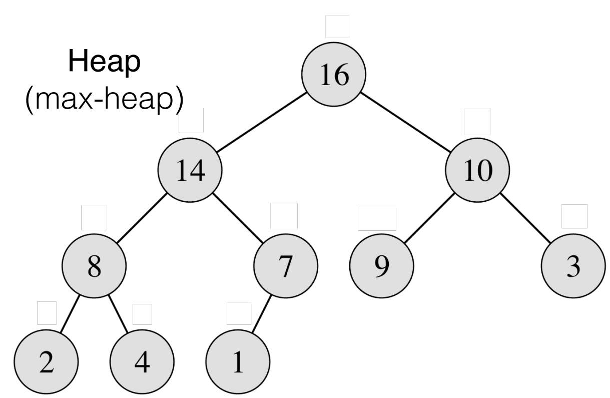
Increase key value for max-heap: for element x, increase its key value to k ($k \geq x$)

traverse the tree upward,

comparing x to its parent and swapping keys if necessary,

until x's key is smaller than its parent's key

Example: increase key of node 8 to 25



similar process for decreasing key value for a min-heap

Heap Operations — increase key value (Cont'd)

Algorithm

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 $A[\text{PARENT}(i)] < A[i]$
    exchange $A[i]$ with $A[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

Analysis:

Upward path from node i has length O(lgn) in an n-element heap

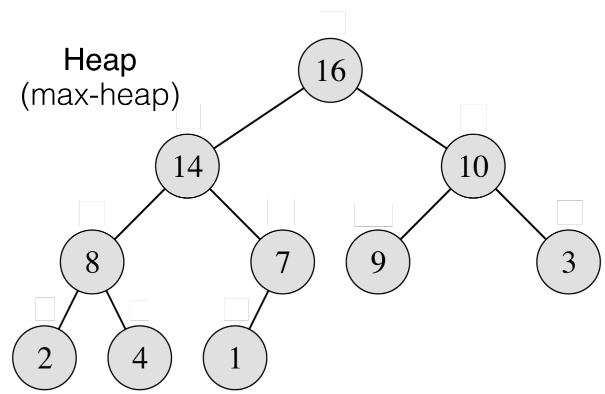
Heap Operations — **insert into heap**

Insertion in max-heap: Insert a key *k* to the heap

Where to add the new node?

insert the new node in the last position

Example: insert key 12 into the heap



similar process for inserting a key value into a min-heap

Heap Operations — insert into heap (Cont'd)

Algorithm

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)

Analysis:

constant time for increasing the size of the heap, and constant time for assignment

O(lgn) for Heap-increase-key

hence, T(n) = O(lgn)

Heap Summary

When to use a heap?

when you find that your program is doing repeated maximum (minimum) computations, especially via exhaustive search (e.g., find a job with the highest priority)

using heap can speed up tremendously

e.g., selection sort ($\Theta(n^2)$) vs. heap sort $O(n\lg n)$

Application of max-heap — Heapsort

Heapsort (ascending order)

Builds a max-heap from the array

Starting with the root, places the maximum element into the correct place in the array by swapping it with the element in the last position in the array

“discard” the last node by decreasing the heap size by 1, and **applies MAX-HEAPIFY on the new root** for the reduced subarray

repeat this “discarding” process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

Heapsort

Algorithm:

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    MAX-HEAPIFY$(A, 1, i-1)$

Analysis:

T(n) = O(n) + (n-1) O(lgn) = O(nlgn)

slower than Quicksort in practice

Application — Priority Queue

Priority Queue with Heap

priority queue: data structure for maintaining a set of elements (S), each with associated value called a key

Heaps efficiently implement priority queues

Priority Queue with Heap (Cont'd)

Event-driven simulator (discrete-event)

- models the operation of a system as discrete events occurs; each event occurs at a particular moment and changes the state in the system
- a *min priority queue:* schedule the events based on their <u>timestamp</u> (as the key) of events
- the simulator calls EXTRACT-MIN at each step to choose the next event to simulate

Priority Queue with Heap (Cont’d)

Job Scheduler in Operating System

- Usually, FIFO queue is used
- but some work has a higher priority, or takes much less time

Priority Queue with Heap

min-priority queue:

insertion(S, x): inserts x into S

MINIMUM(S): get the smallest key

EXTRACT-MIN(S): removes and returns the smallest key

DECREASE-KEY(S, x, k): decrease the value of x to k ($k \leq x$)

Priority Queue with Heap (Cont'd)

max-priority queue:

- insertion(S, x): inserts x into S
- MAXIMUM(S): get the largest key
- EXTRACT-MAX(S): removes and returns the largest key
- INCREASE-KEY(S, x, k): increase the value of x to k ($k \geq x$)

Chapter Summary

Heap: two views (the property of a complete binary tree)

What operations can (binary) heap support and their running time?

- HEAPIFY: O(*lgn*)
- building a heap with HEAPIFY: O(*n*)
- insertion: O(lg*n*)
- extract-max for max-heap, extract-min for min-heap: O(lg*n*)
- increasing key value for max-heap /decreasing key value for min-heap O(lg*n*)

Sorting Introduction

Sorting Algorithms Comparison

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)