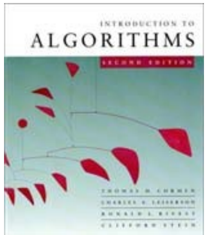
CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya



Best-case analysis

(For intuition only!)

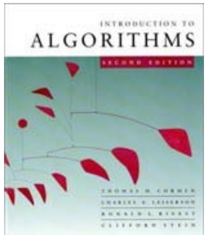
If we're lucky, PARTITION splits the array evenly:

$$
\begin{aligned}
T(n) &= 2T(n/2) + \Theta(n) \\
&= \Theta(n \lg n) \qquad \text{(same as merge sort)}
\end{aligned}
$$

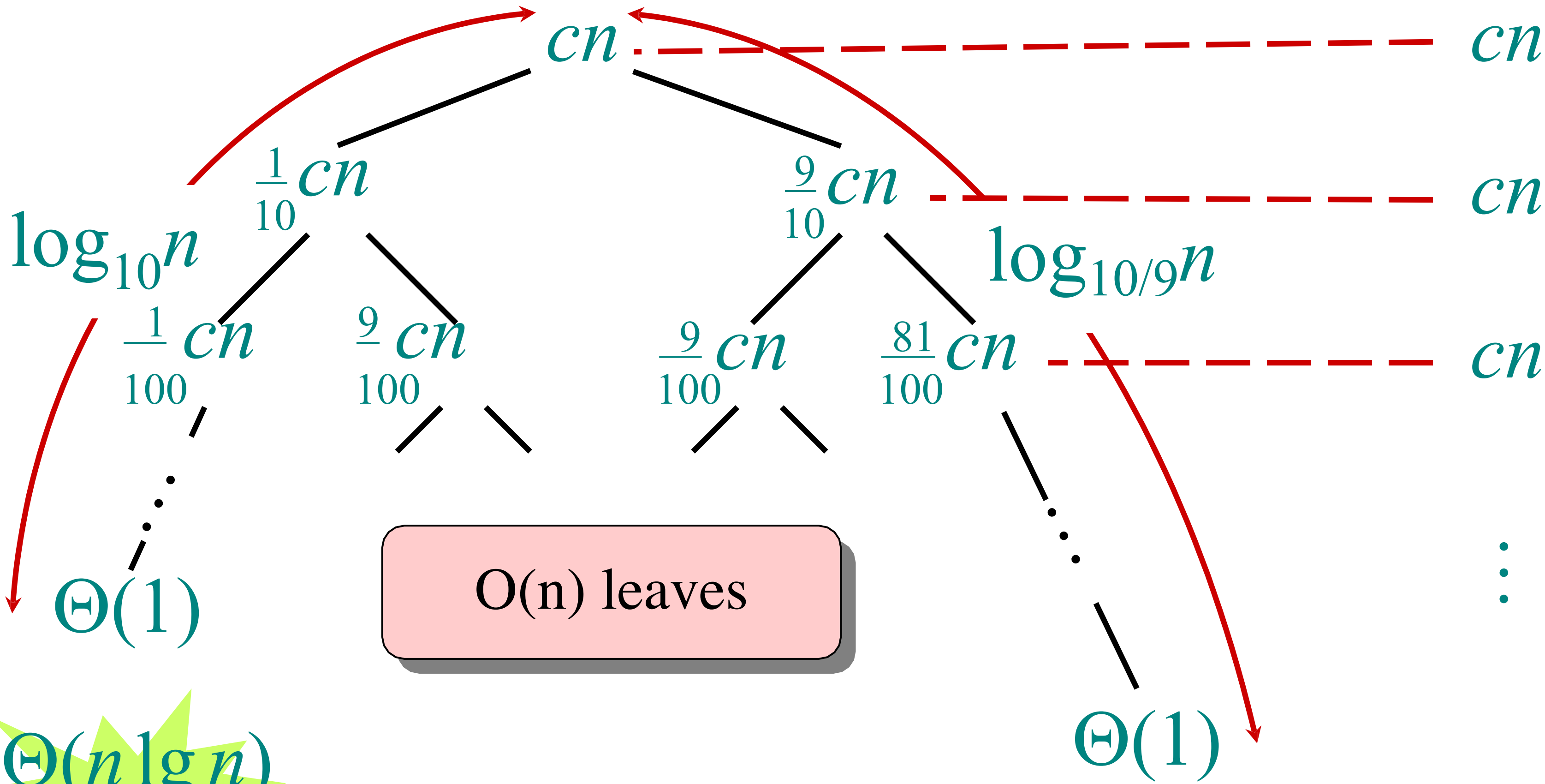
What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\tfrac{1}{10}n\right) + T\left(\tfrac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?



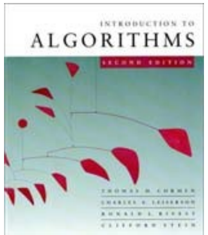
Analysis of “almost-best” case



$\Theta(n \lg n)$

Lucky!

$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$



More intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….

$$L(n) = 2U(n/2) + \Theta(n) \quad \textit{lucky}$$
$$U(n) = L(n-1) + \Theta(n) \quad \textit{unlucky}$$

Solving:

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

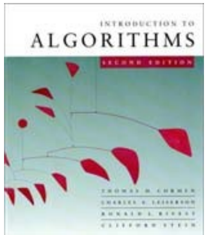
Lucky!

How can we make sure we are usually lucky?

Randomized quicksort

IDEA: Partition around a ***random*** element.

- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.



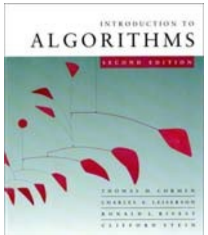
Randomized quicksort analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the ***indicator random variable***

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

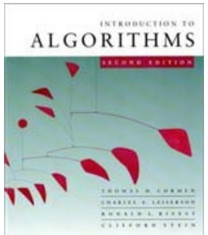
$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.



Analysis (continued)

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

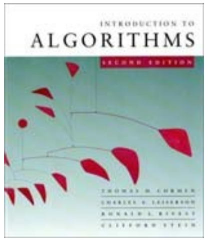
$$= \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n) \right)$$



Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

Take expectations of both sides.

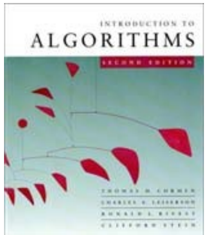


Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k\left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

Linearity of expectation.



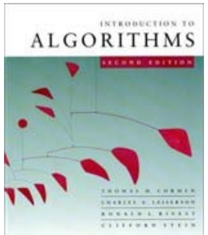
Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E\big[X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\big]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

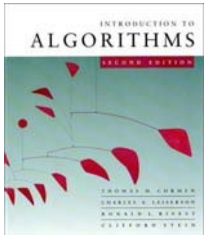
Independence of X_k from other random choices.



Calculating expectation

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k\big(T(k)+T(n-k-1)+\Theta(n)\big)\right] \\
&= \sum_{k=0}^{n-1} E\big[X_k\big(T(k)+T(n-k-1)+\Theta(n)\big)\big] \\
&= \sum_{k=0}^{n-1} E[X_k]\cdot E[T(k)+T(n-k-1)+\Theta(n)] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1}\Theta(n)
\end{aligned}
$$

Linearity of expectation; $E[X_k] = 1/n$.



Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

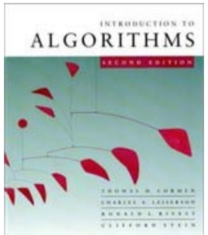
$$= \sum_{k=0}^{n-1} E\big[X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\big]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1}\Theta(n)$$

$$= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Summations have identical terms.



Hairy recurrence

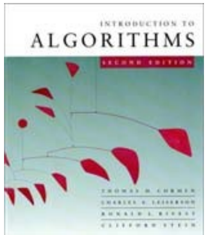
$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Prove: $E[T(n)] \le an \lg n$ for constant $a > 0$.

- Choose a large enough so that $an \lg n$ dominates $E[T(n)]$ for sufficiently small $n \ge 2$.

Use fact: $\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).



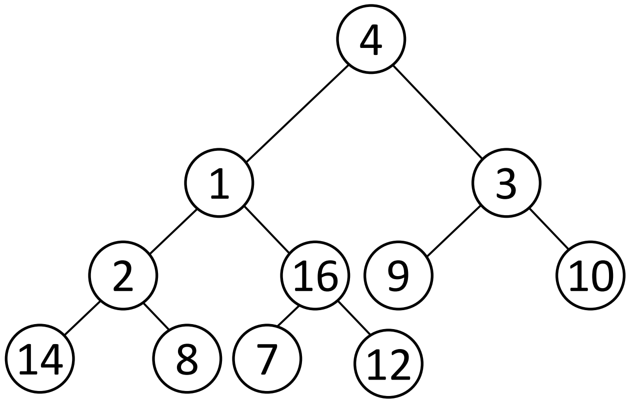
Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from ***code tuning***.
- Quicksort behaves well even with caching and virtual memory.

HEAP AND HEAPSORT

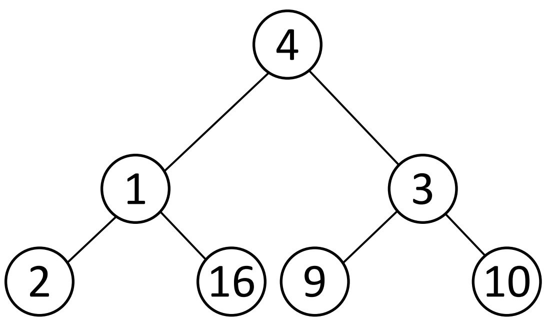
Special Types of Trees

- *Def:* Full binary tree = a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

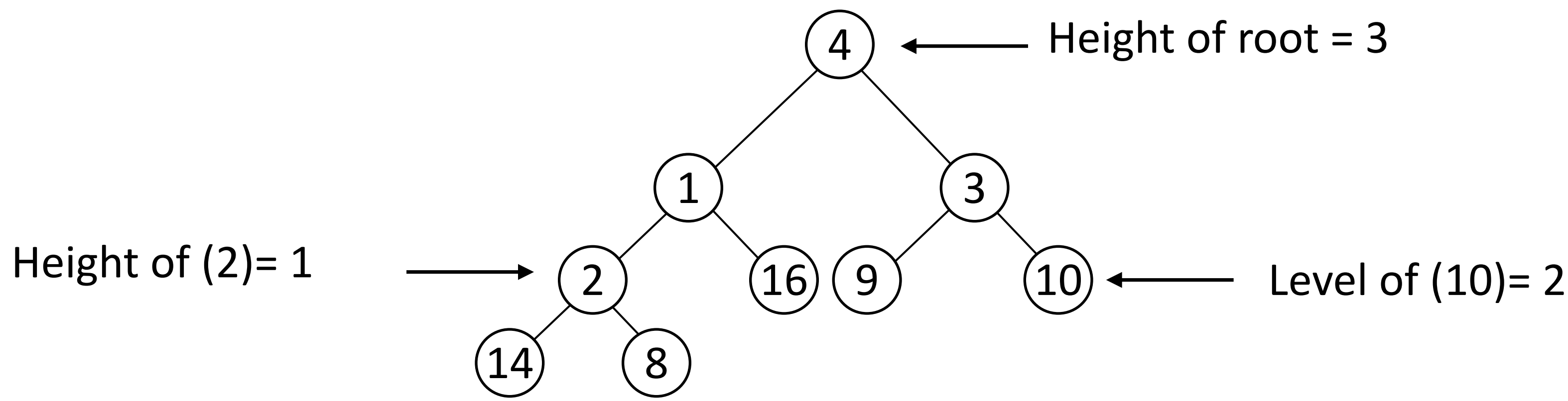
- *Def:* Complete binary tree = a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



Complete binary tree

Definitions

- **Height** of a node = the number of edges on the longest simple path from the node down to a leaf
- **Level** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node

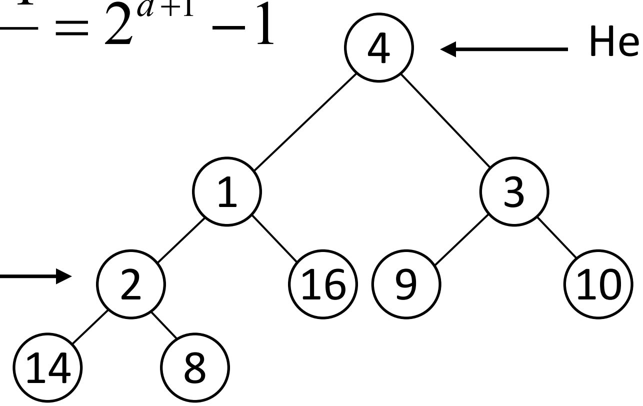


Useful Properties

- There are **at most** 2^l nodes at level (or depth) l of a binary tree
- A binary tree with height d has **at most** $2^{d+1} - 1$ nodes
- A binary tree with n nodes has height **at least** $\lfloor lgn \rfloor$

$$n \leq \sum_{l=0}^{d} 2^l = \frac{2^{d+1} - 1}{2 - 1} = 2^{d+1} - 1$$

Height of root = 3



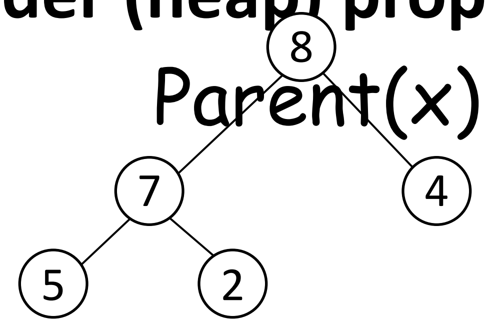
Height of (2)= 1

Level of (10)= 2

The Heap Data Structure

- *Def:* A **heap** is a <u>nearly complete</u> binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

$$\text{Parent}(x) \geq x$$



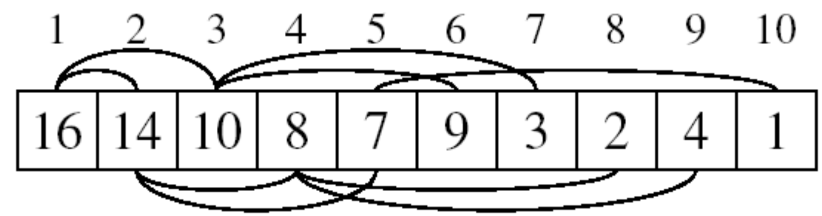
Heap

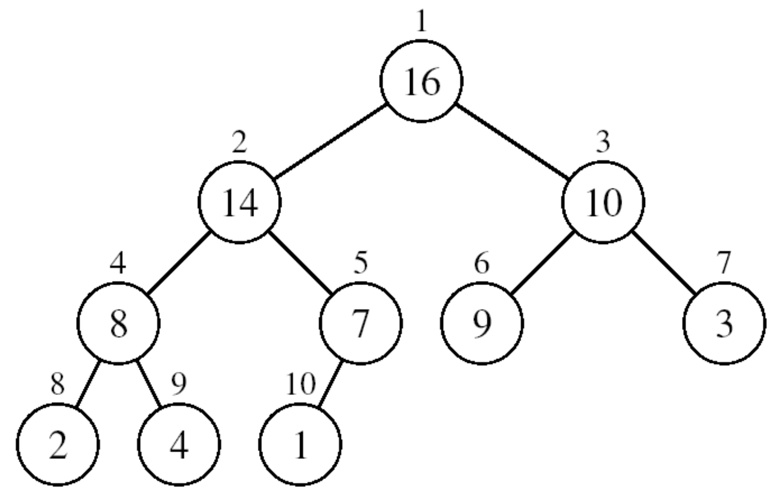
From the heap property, it follows that:
"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves





Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

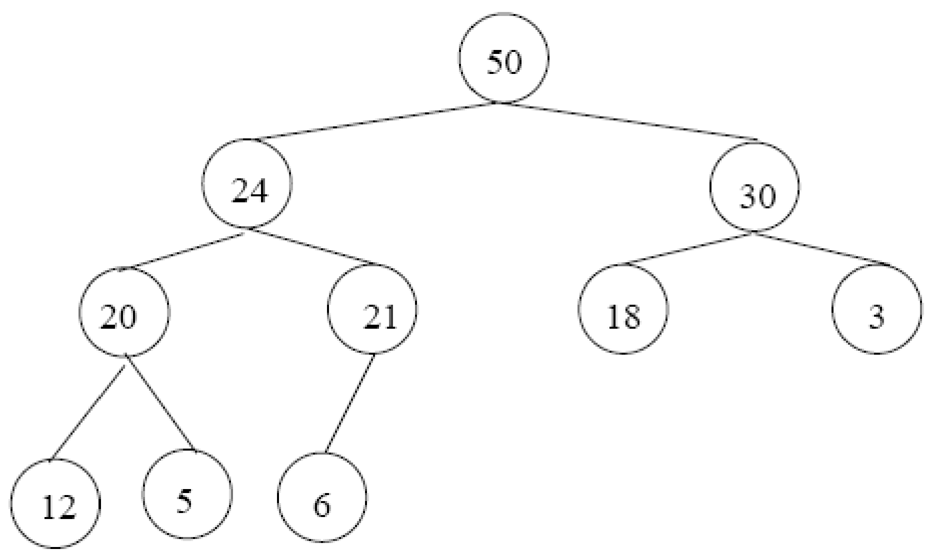
$$A[\mathrm{PARENT}(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

$$A[\mathrm{PARENT}(i)] \leq A[i]$$

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right

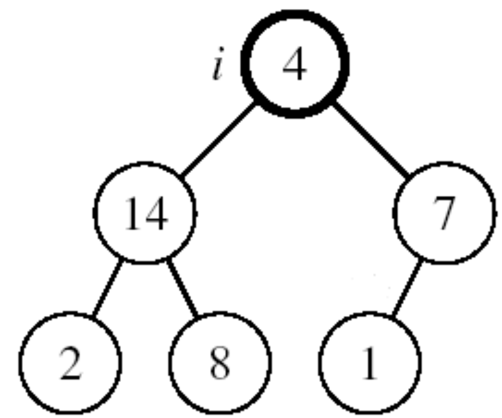


Operations on Heaps

- Maintain/Restore the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queues

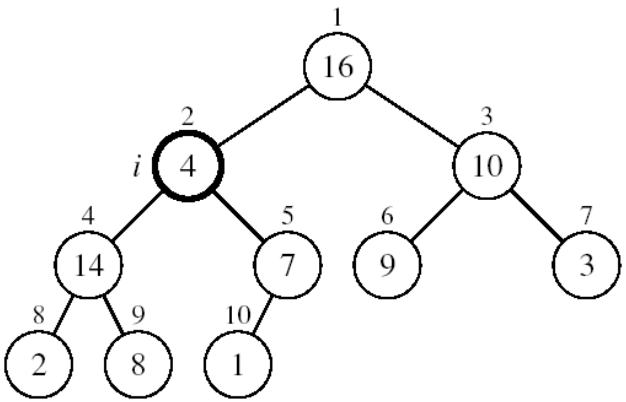
Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



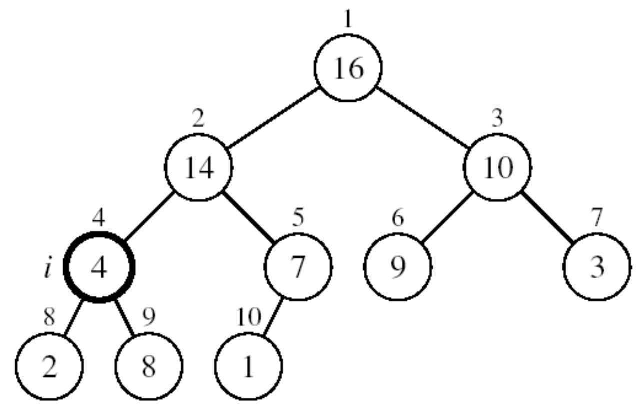
Example

<u>MAX-HEAPIFY(A, 2, 10)</u>



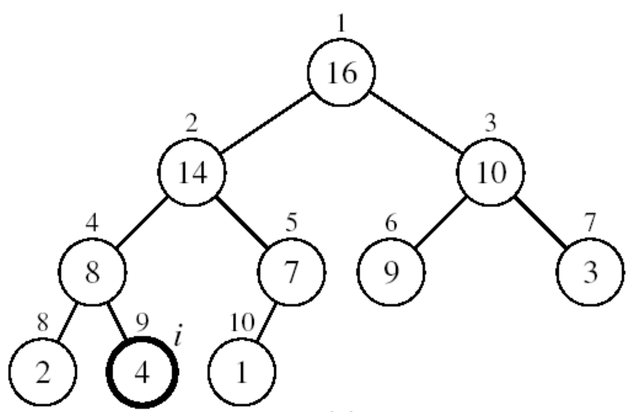
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

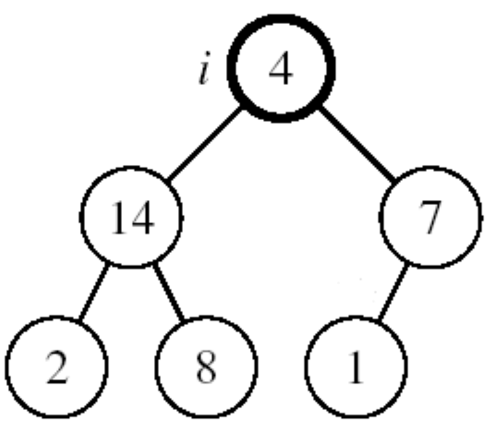
A[4] ↔ A[9]



Heap property restored

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of *i* are max-heaps
  - $A[i]$ may be smaller than



Alg: MAX-HEAPIFY(A, i, n)

1. $l \leftarrow$ LEFT(i)
2. $r \leftarrow$ RIGHT(i)
3. **if** $l \le n$ and $A[l] > A[i]$
4.   **then** $largest \leftarrow l$
5.   **else** $largest \leftarrow i$
6. **if** $r \le n$ and $A[r] > A[largest]$
7.   **then** $largest \leftarrow r$
8. **if** $largest \neq i$
9.   **then** exchange $A[i] \leftrightarrow A[largest]$
10.   MAX-HEAPIFY(A, $largest$, n)

MAX-HEAPIFY Running Time

- Intuitively:

  - It traces a path from the root to a leaf (longest path length: h)
  - At each level, it makes exactly 2 comparisons
  - Total number of comparisons is 2h
  - Running time is O(h) or $O(lgn)$

- Running time of MAX-HEAPIFY is $O(lgn)$

- Can be written in terms of the height of the heap, as being $O(h)$
  - Since the height of the heap is $\lfloor lgn \rfloor$

Building a Heap

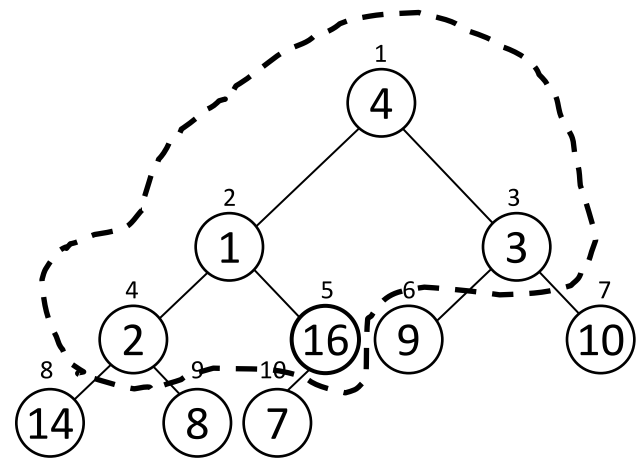
- Convert an array $A[1 \ldots n]$ into a max-heap (n = $length[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: <u>BUILD-MAX-HEAP(A)</u>

n = length[A]

for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1

do MAX-HEAPIFY(A, i, n)



A:

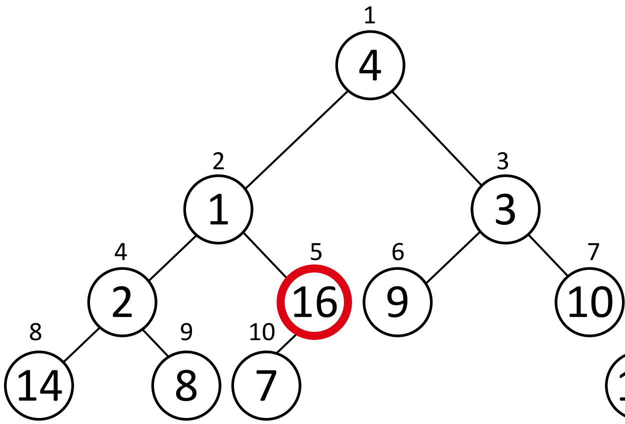
4	1	3	2	16	9	10	14	8	7

Example:

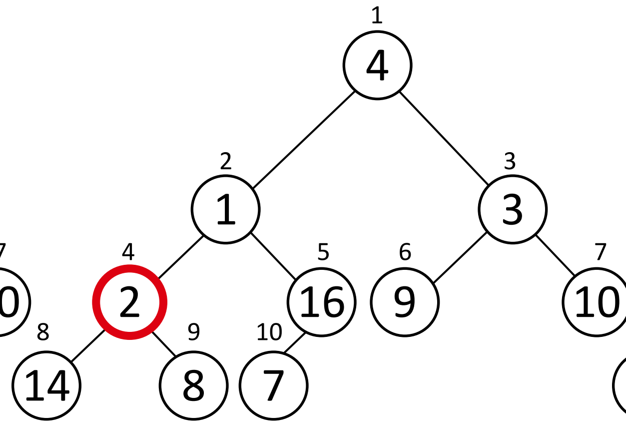
A

4	1	3	2	16	9	10	14	8	7

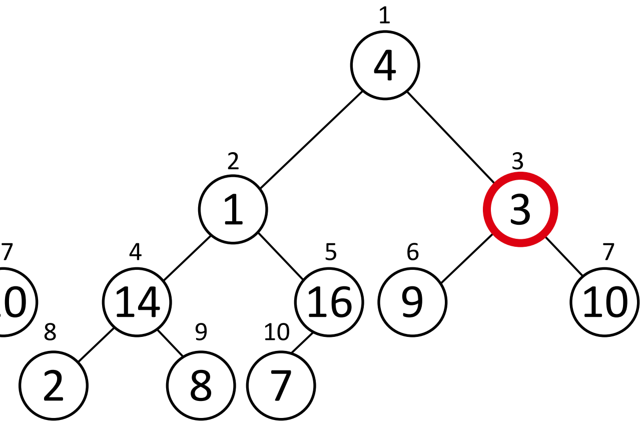
$i = 5$



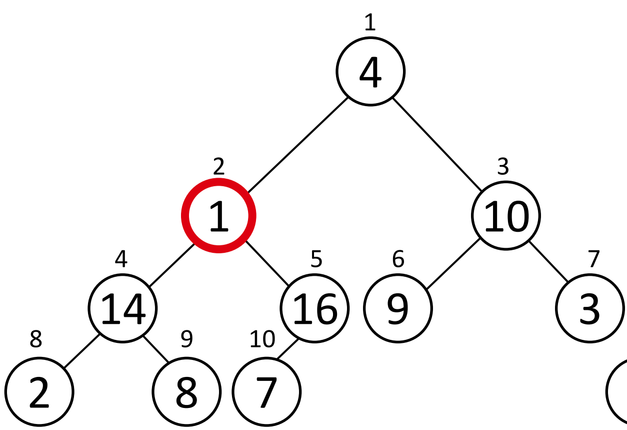
$i = 4$



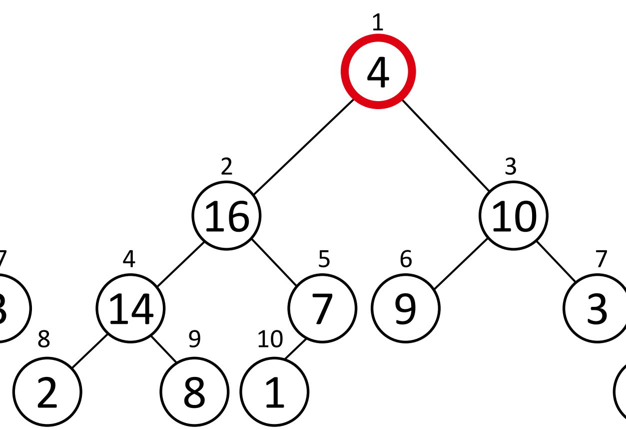
$i = 3$

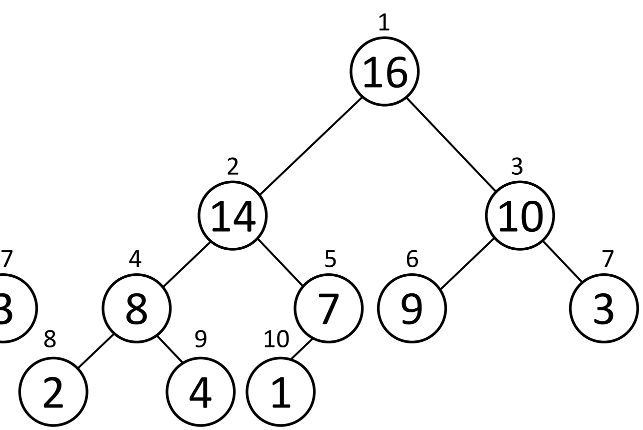


$i = 2$



$i = 1$





Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n) $O(\lg n)$

$\left.\begin{array}{l} \text{lines 2–3} \end{array}\right\}\ O(n)$

$\Rightarrow$ Running time: $O(n\lg n)$

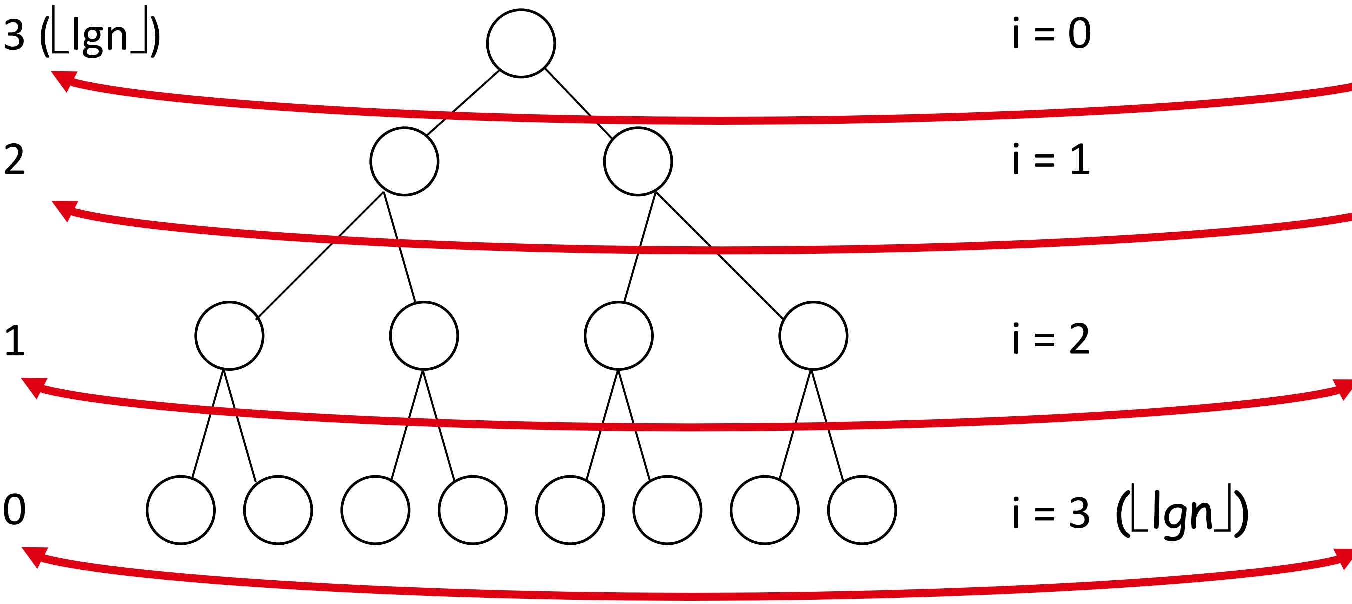
- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- HEAPIFY takes $O(h)$ $\Rightarrow$ the cost of HEAPIFY on a node **i** is proportional to the height of the node **i** in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$

Height	Level	No. of nodes
$h_0 = 3$ ($\lfloor \lg n \rfloor$)	i = 0	2^0
$h_1 = 2$	i = 1	2^1
$h_2 = 1$	i = 2	2^2
$h_3 = 0$	i = 3 ($\lfloor \lg n \rfloor$)	2^3



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

Running Time of BUILD MAX HEAP

$$T(n) = \sum_{i=0}^{h} n_i h_i$$ Cost of HEAPIFY at level i $*$ number of nodes at that level

$$= \sum_{i=0}^{h} 2^i (h - i)$$ Replace the values of n_i and h_i computed before

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$ Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$ Change variables: k = h - i

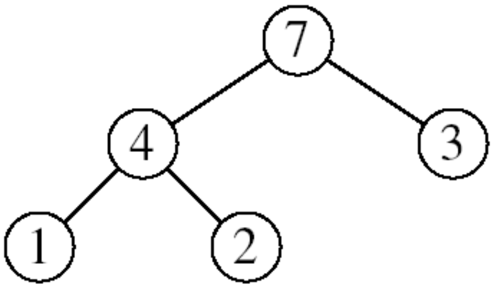
$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$ The sum above is smaller than the sum of all elements to ∞ and h = lgn

$$= O(n)$$ The sum above is smaller than 2

Running time of BUILD-MAX-HEAP: $T(n) = O(n)$

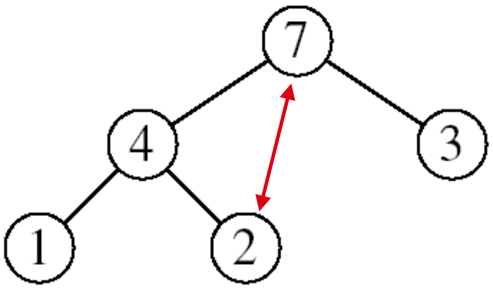
Heapsort

- Goal:
  - Sort an array using heap representations

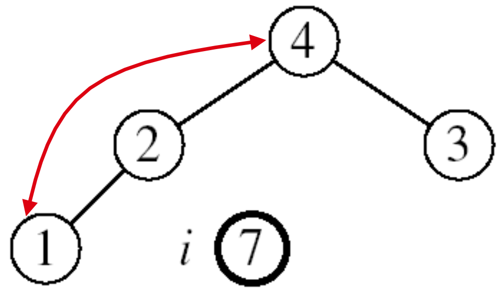


- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

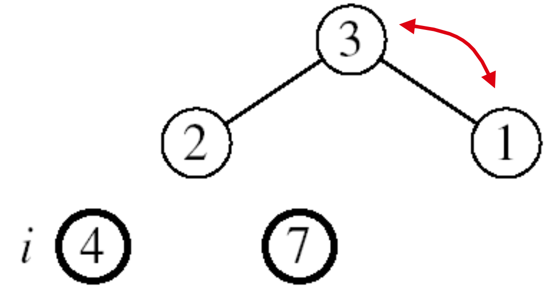
Example: A=[7, 4, 3, 1, 2]



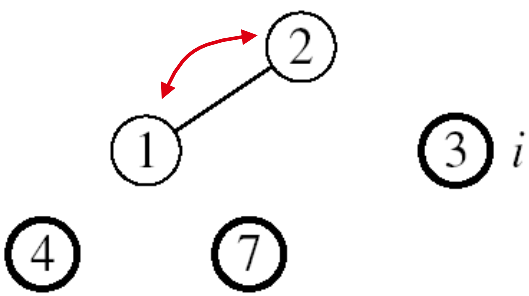
MAX-HEAPIFY(A, 1, 4)



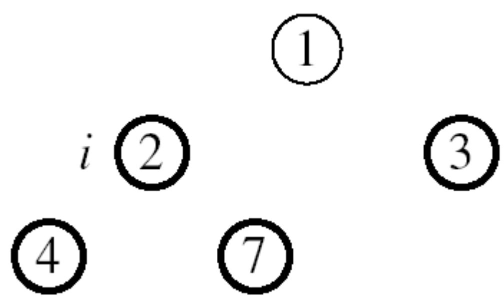
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)



A	1	2	3	4	7

Alg: HEAPSORT(*A*)

1. BUILD-MAX-HEAP(*A*) — $O(n)$
2. **for** i ← length[*A*] **downto** 2
3. **do** exchange *A*[1] ↔️ *A*[i] — $O(\lg n)$
4. MAX-HEAPIFY(*A*, 1, i - 1)

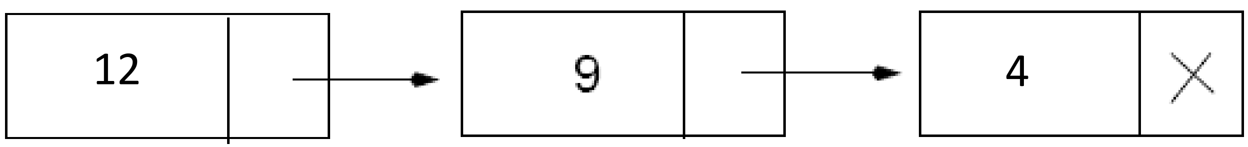
n-1 times

- Running time: $O(n\lg n)$ --- Can be shown to be $\Theta(n\lg n)$

Priority Queues

Properties

- Each element is associated with a value (priority)
- The key with the highest (or lowest) priority is extracted first



Operations on Priority Queues

- Max-priority queues support the following operations:
  - INSERT(S, x): inserts element x into set S
  - EXTRACT-MAX(S): removes and returns element of S with largest key
  - MAXIMUM(S): returns element of S with largest key
  - INCREASE-KEY(S, x, k): increases value of

HEAP-MAXIMUM

Goal:

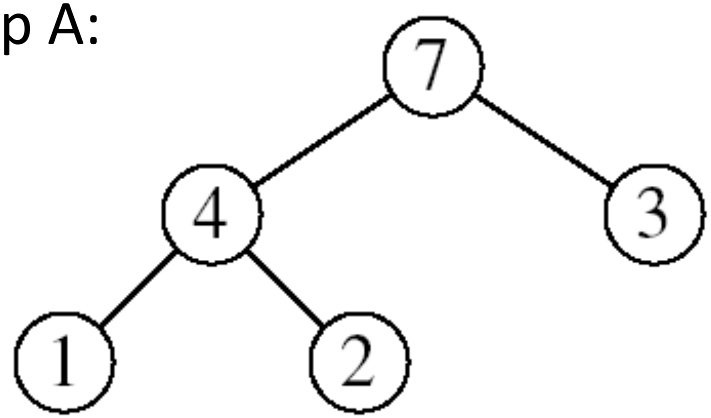
- Return the largest element of the heap

Running time: $O(1)$

Alg: HEAP-MAXIMUM(A)

1. **return** $A[1]$

Heap A:



Heap-Maximum(A) returns 7

HEAP-EXTRACT-MAX

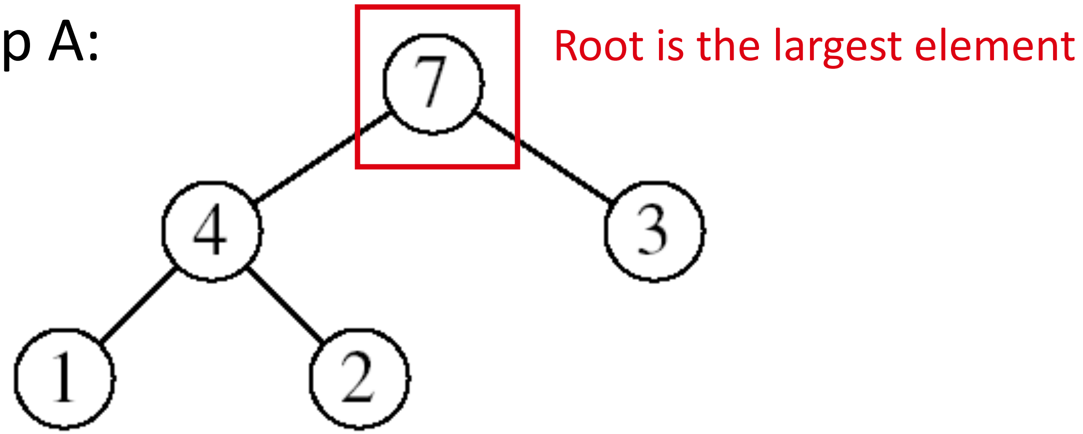
Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

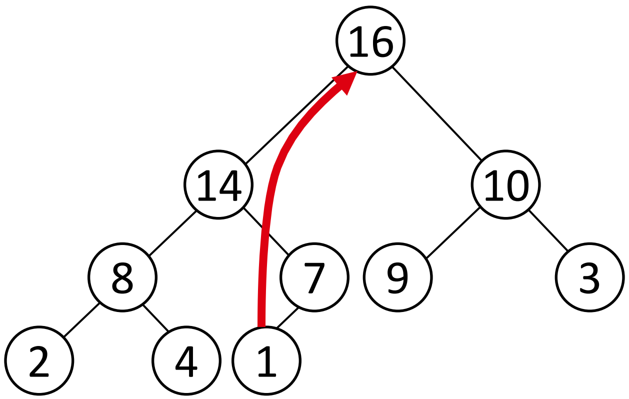
Idea:

- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1

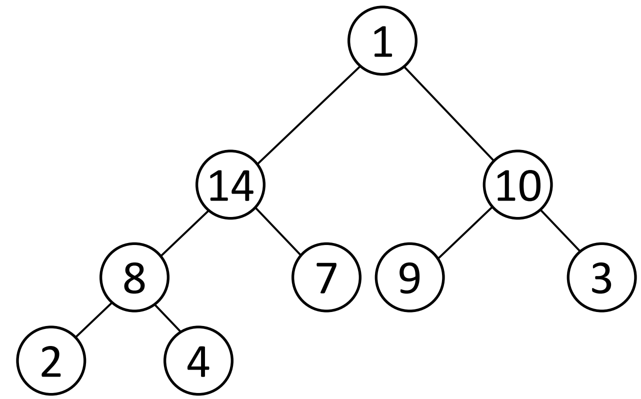
Heap A:



Example: HEAP-EXTRACT-MAX

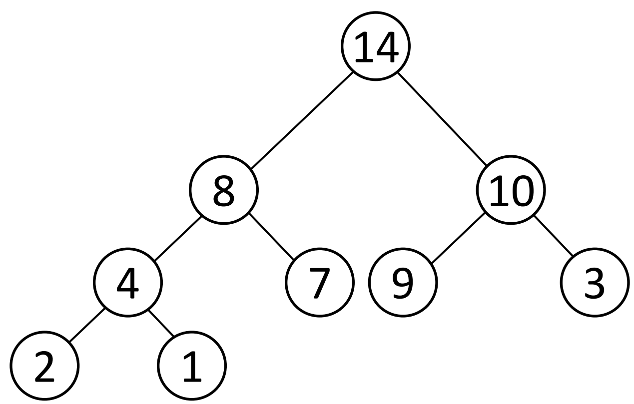


max = 16



Heap size decreased with 1

Call MAX-HEAPIFY(A, 1, n-1)



HEAP-EXTRACT-MAX

Alg: HEAP-EXTRACT-MAX(A, n)

if $n < 1$

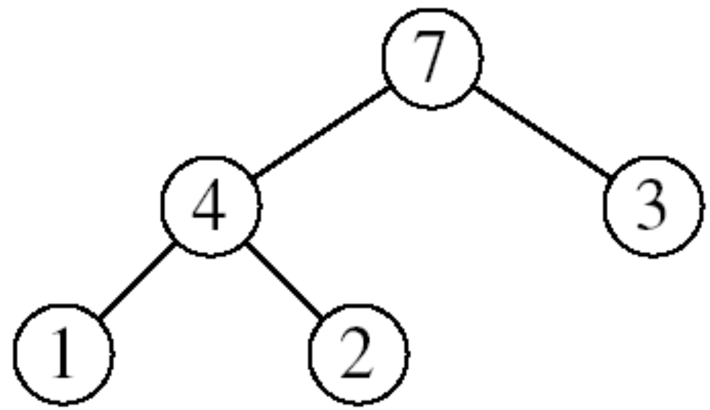
  then error “heap underflow”

$max \leftarrow A[1]$

$A[1] \leftarrow A[n]$

MAX-HEAPIFY(A, 1, n-1)   ▷ remakes heap

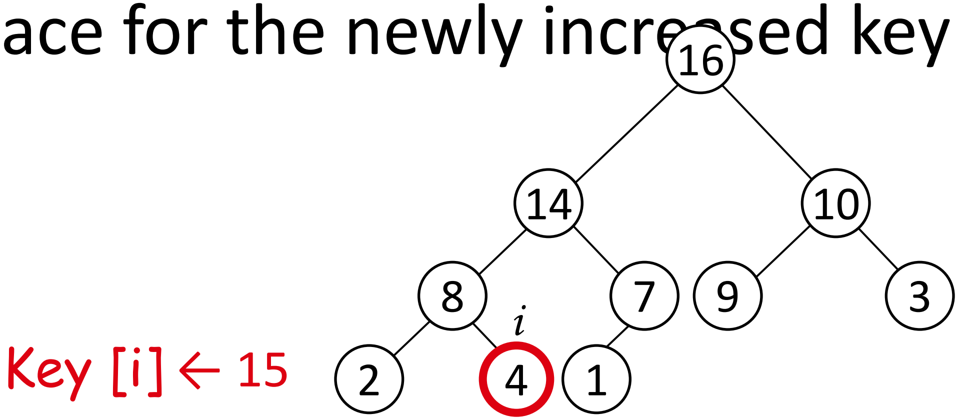
return max



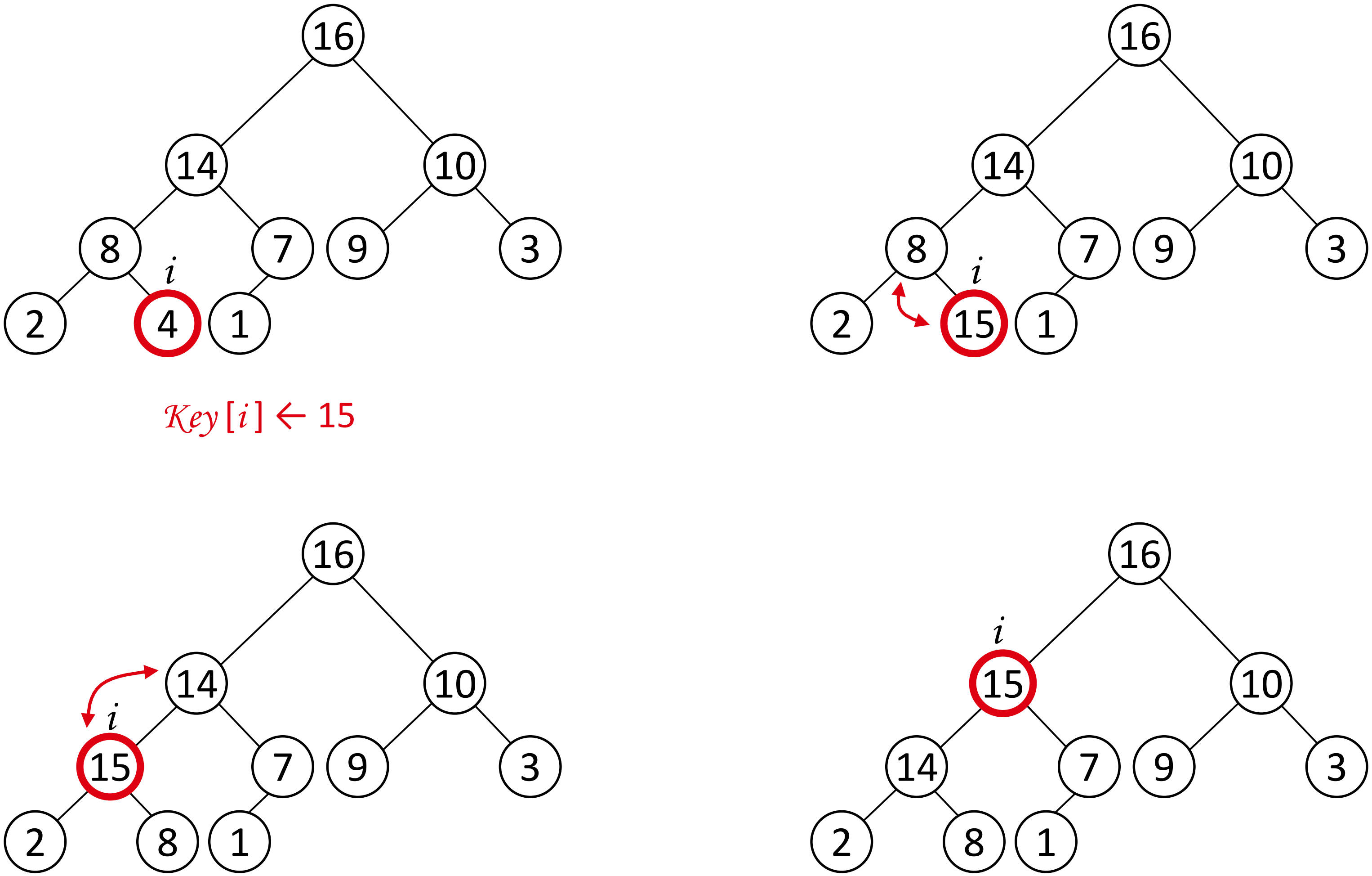
Running time: $O(\lg n)$

HEAP-INCREASE-KEY

- Goal:
  - Increases the key of an element i in the heap
- Idea:
  - Increment the key of $A[i]$ to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key



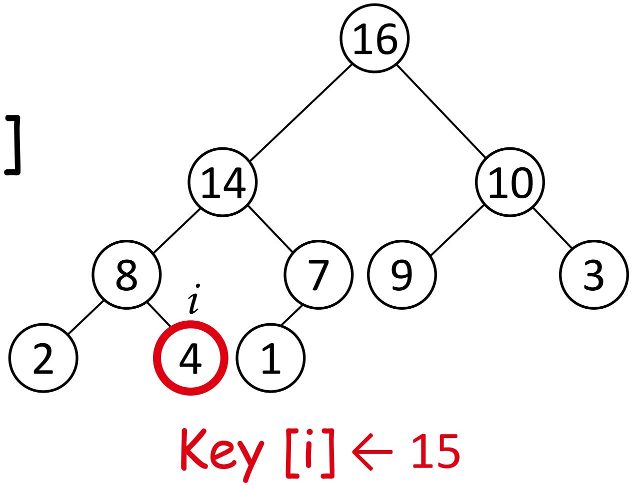
Example: HEAP-INCREASE-KEY



HEAP-INCREASE-KEY

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

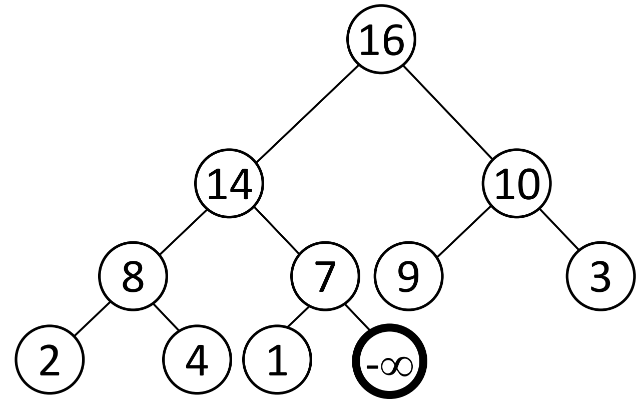
1. **if** key < A[i]
2. **then error** "new key is smaller than current key"
3. A[i] ← key
4. **while** i > 1 and A[PARENT(i)] < A[i]
5. **do** exchange A[i] ↔️ A[PARENT(i)]
6. i ← PARENT(i)



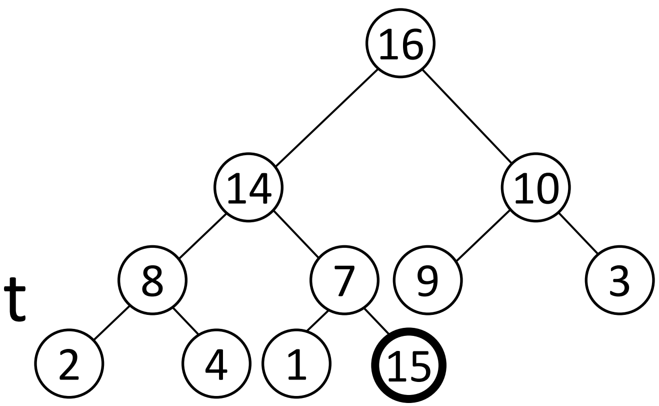
- Running time: $O(\lg n)$

MAX-HEAP-INSERT

- Goal:
  – Inserts a new element into a max-heap

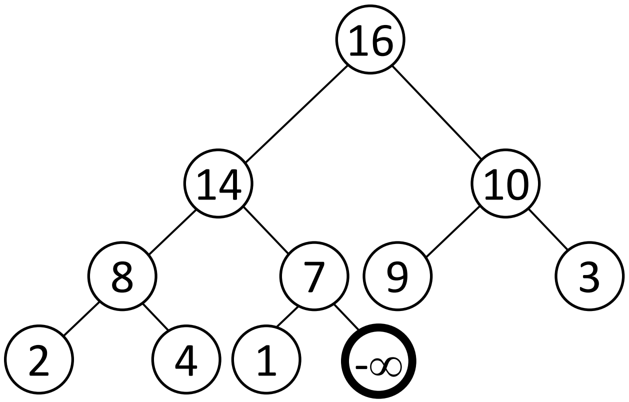


- Idea:
  – Expand the max-heap with a new element whose key is $-\infty$
  – Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property

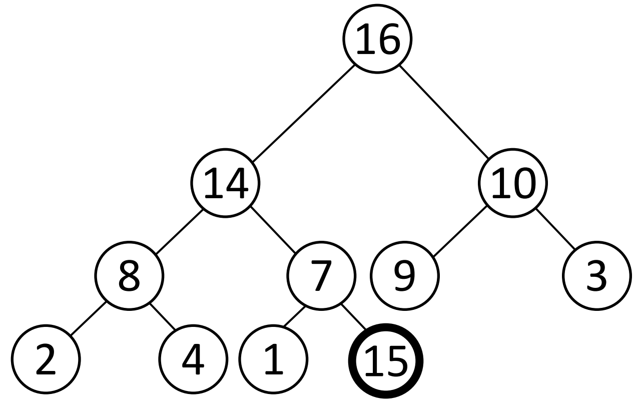


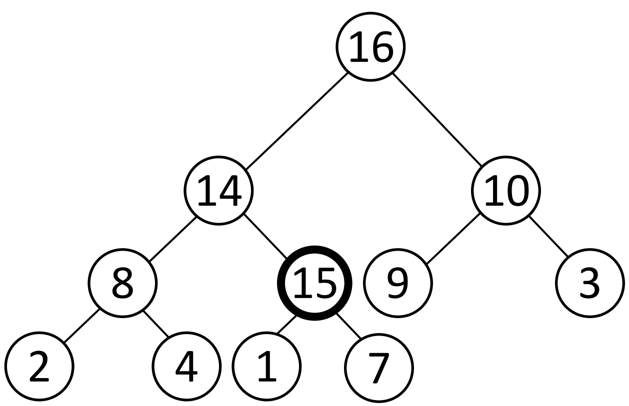
Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

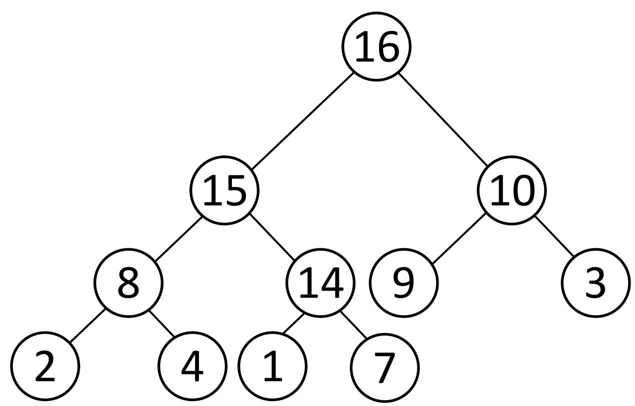


Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15



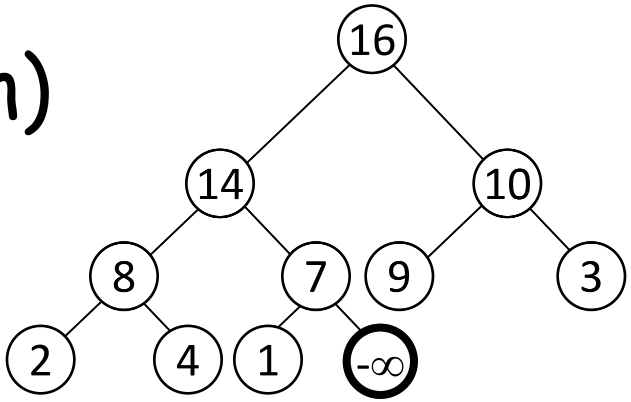


The restored heap containing the newly added element



MAX-HEAP-INSERT

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



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

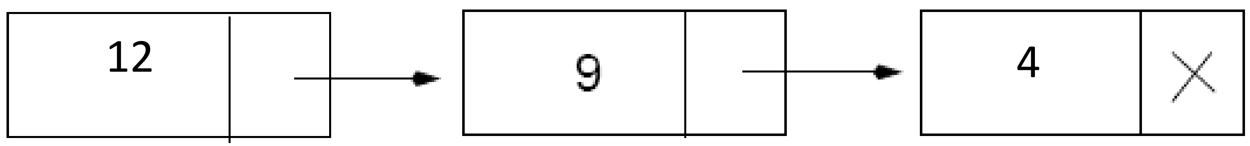
Running time: $O(\lg n)$

Summary

- We can perform the following operations on heaps:
  - MAX-HEAPIFY $O(\lg n)$
  - BUILD-MAX-HEAP $O(n)$
  - HEAP-SORT $O(n\lg n)$
  - MAX-HEAP-INSERT $O(\lg n)$
  - HEAP-EXTRACT-MAX $O(\lg n)$
  - HEAP-INCREASE-KEY $O(\lg n)$
  - HEAP-MAXIMUM $O(1)$

Average $O(\lg n)$

Priority Queue Using Linked List



Remove a key: O(1)

Insert a key: O(n)

Increase key: O(n)

Extract max key: O(1)

Average: O(n)