Heap
A priority queue data structure

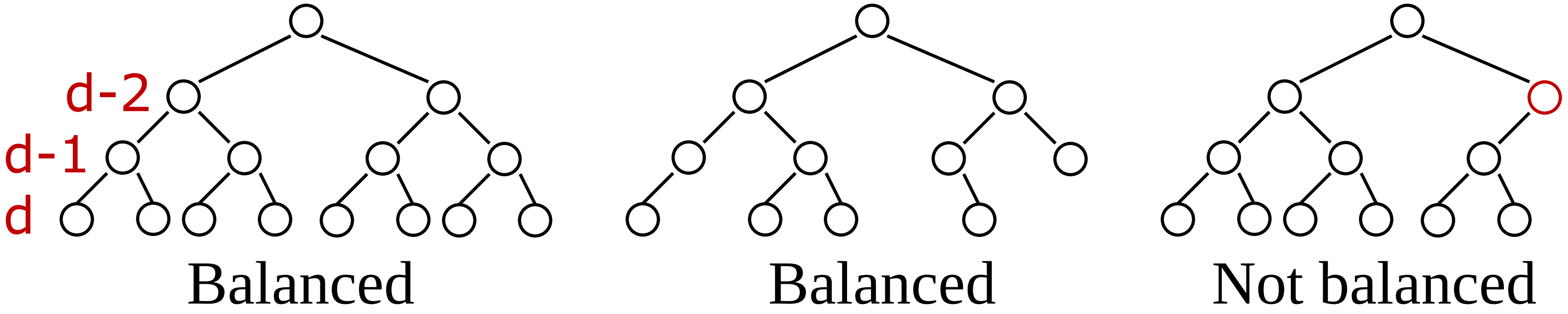
Sources:
David Matuszek, University Of Pennsylvania: www.cis.upenn.edu/~matuszek/
Dr. George Bebis, University of Nevado: Course page: www.cse.unr.edu/~bebis
Slides and figures have been collected from various publicly available Internet sources for preparing the lecture slides of IT2001 course. I acknowledge and thank all the original authors for their contribution to prepare the content.

What is a "Heap"?

- Definitions of heap:
  1. A large area of memory from which the programmer can allocate blocks as needed, and deallocate them (or allow them to be garbage collected) when no longer needed
  2. A balanced, left-justified binary tree in which no node has a value greater than the value in its parent
- These two definitions have little in common
- Heap data structure uses the second definition

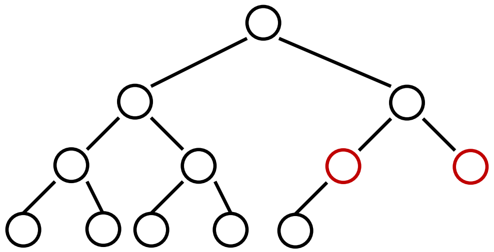
Balanced binary trees

- Recall:
  - The depth of a node is its distance from the root
  - The depth of a tree is the depth of the deepest node
- A binary tree of depth d is balanced if all the nodes at depths 0 through d-2 have two children

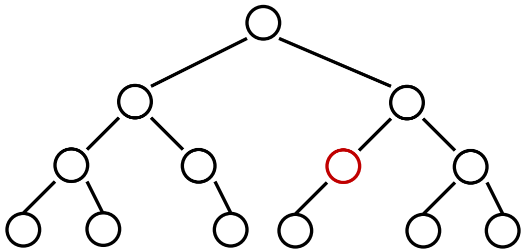


Left-justified binary trees

- A balanced binary tree is left-justified if:
  - all the leaves are at the same depth, or
  - all the leaves at depth d are to the left of all the nodes at depth d-1



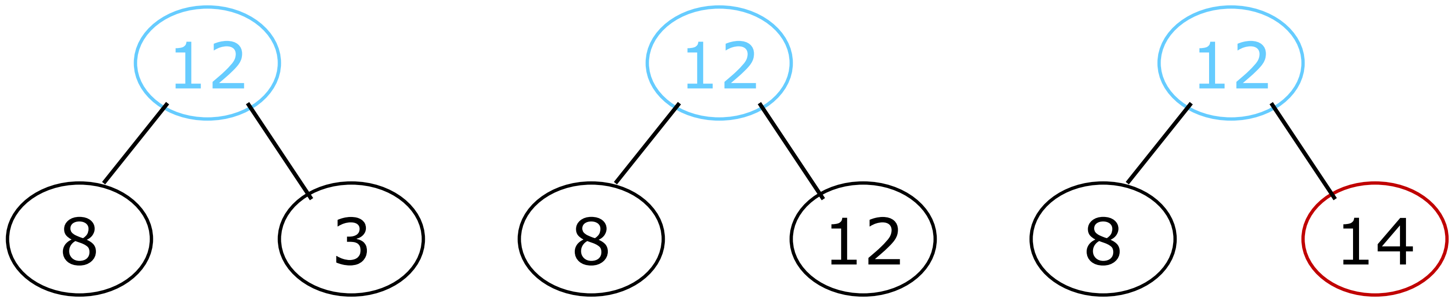
Left-justified



Not left-justified

The heap property

- A node has the heap property if the value in the node is as large as or larger than the values in its children



Blue node has heap property

Blue node has heap property

Blue node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

Heap Types

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

$$A[PARENT(i)] \geq A[i]$$

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

$$A[PARENT(i)] \leq A[i]$$

❑ We will consider Max-heaps only.

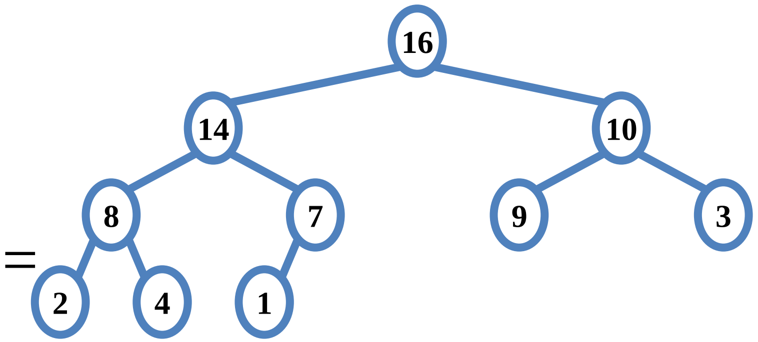
Heaps

- In practice, heaps are usually implemented as arrays:

A =

16	14	10	8	7	9	3	2	4	1

=



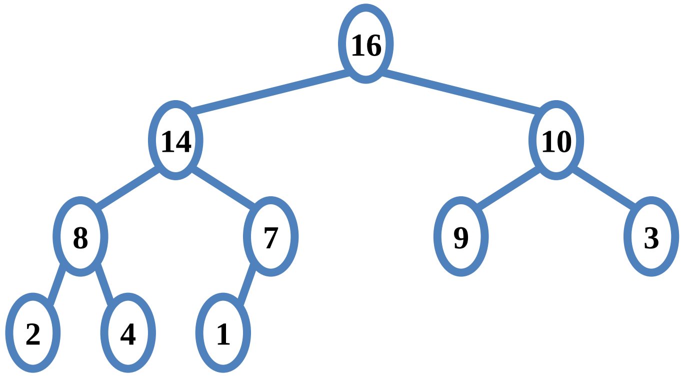
Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2] (note: integer divide)
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

A =

16	14	10	8	7	9	3	2	4	1

=



Referencing Heap Elements

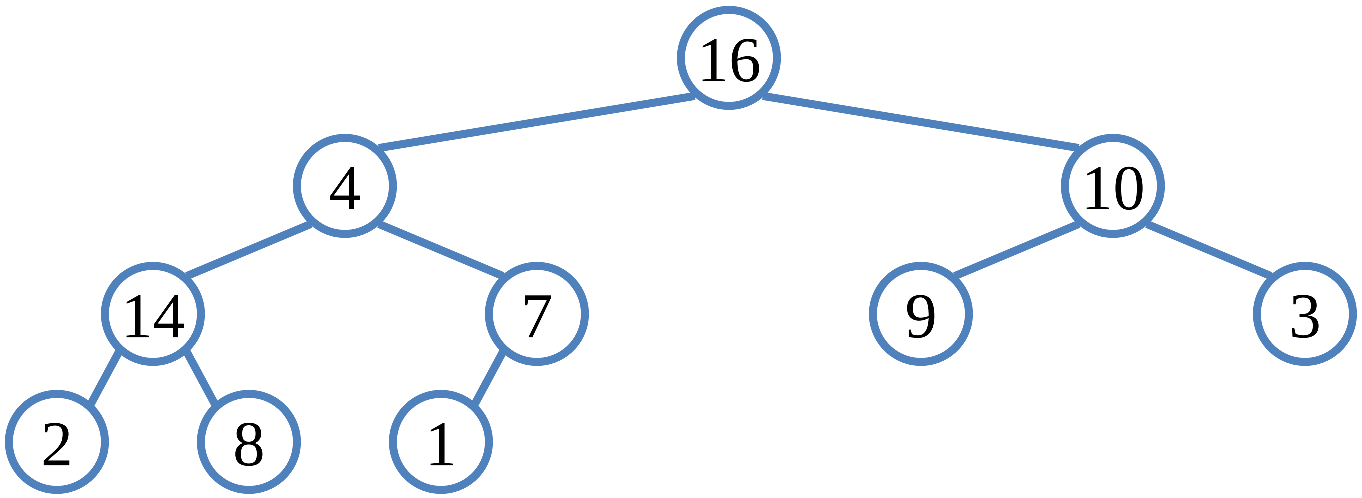
- So…

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

Heap Operations: Heapify()

- **`Heapify()`**: maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Problem: The subtree rooted at *i* may violate the heap property (*How?*)
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - *What do you suppose will be the basic operation between i, l, and r?*

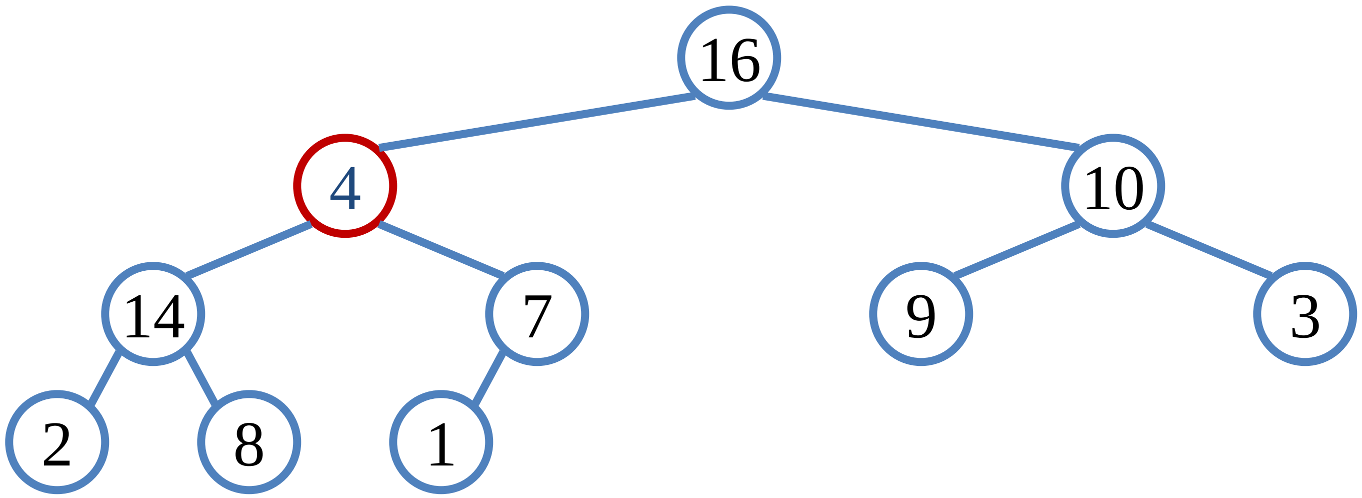
Heapify() Example



A =

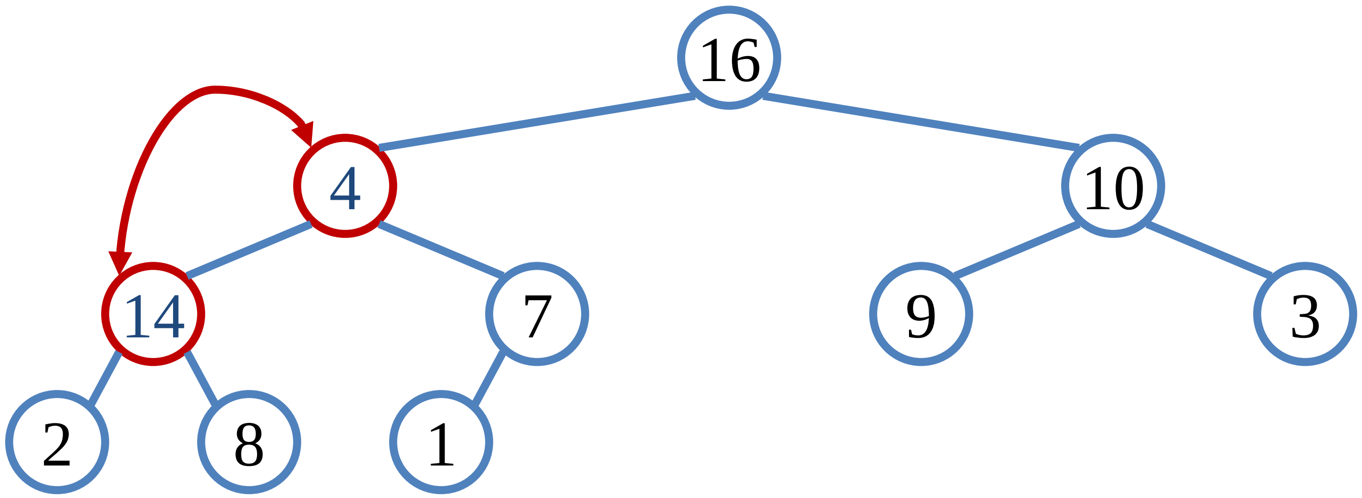
16	4	10	14	7	9	3	2	8	1

Heapify() Example



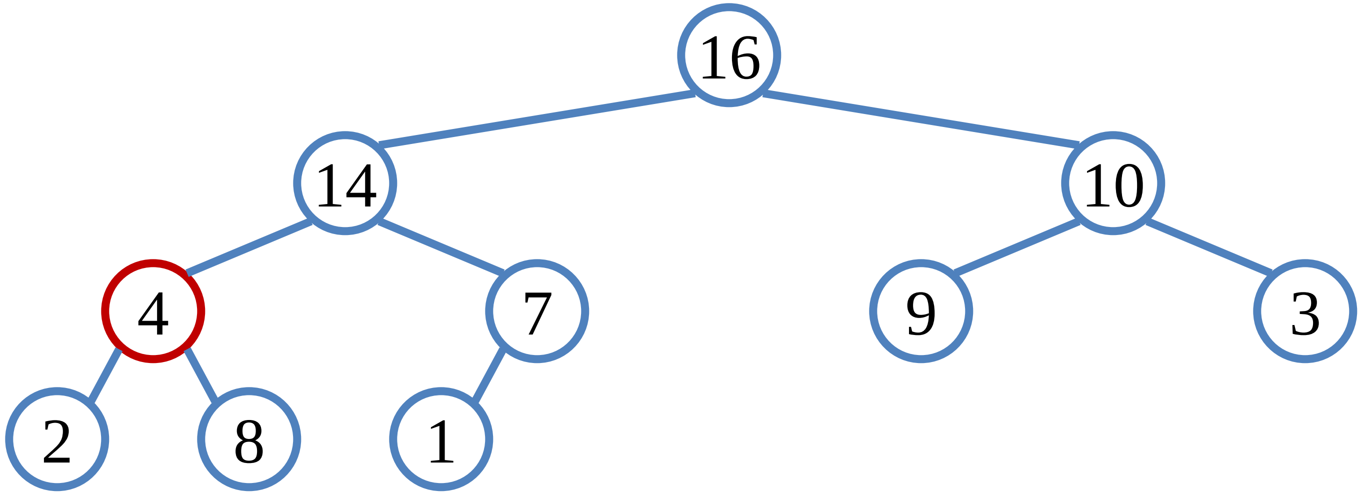
A =	16	4	10	14	7	9	3	2	8	1

Heapify() Example



A =	16	4	10	14	7	9	3	2	8	1

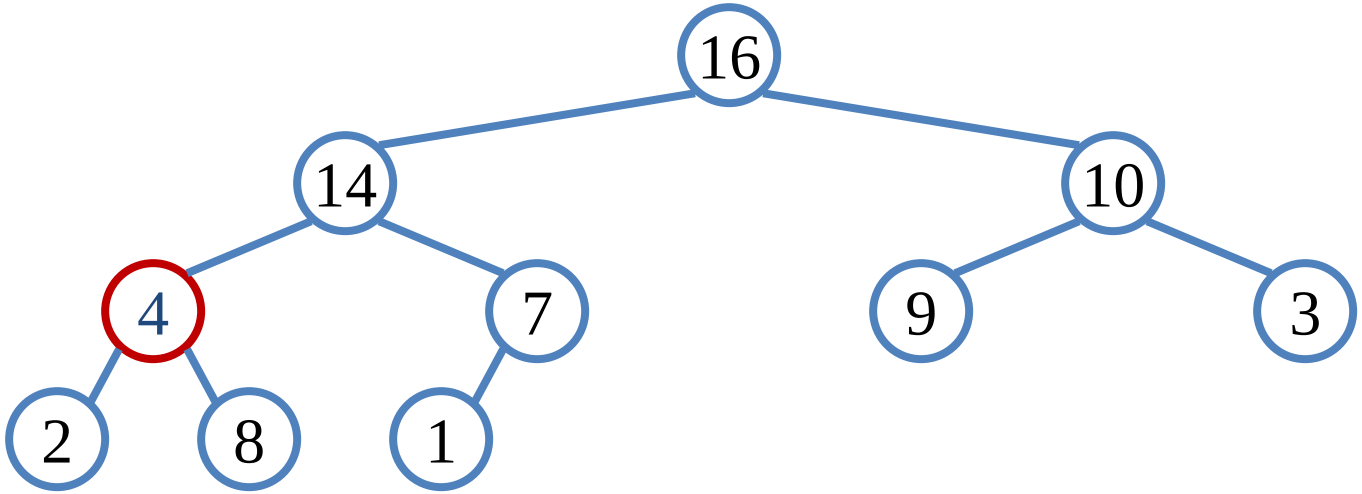
Heapify() Example



A =

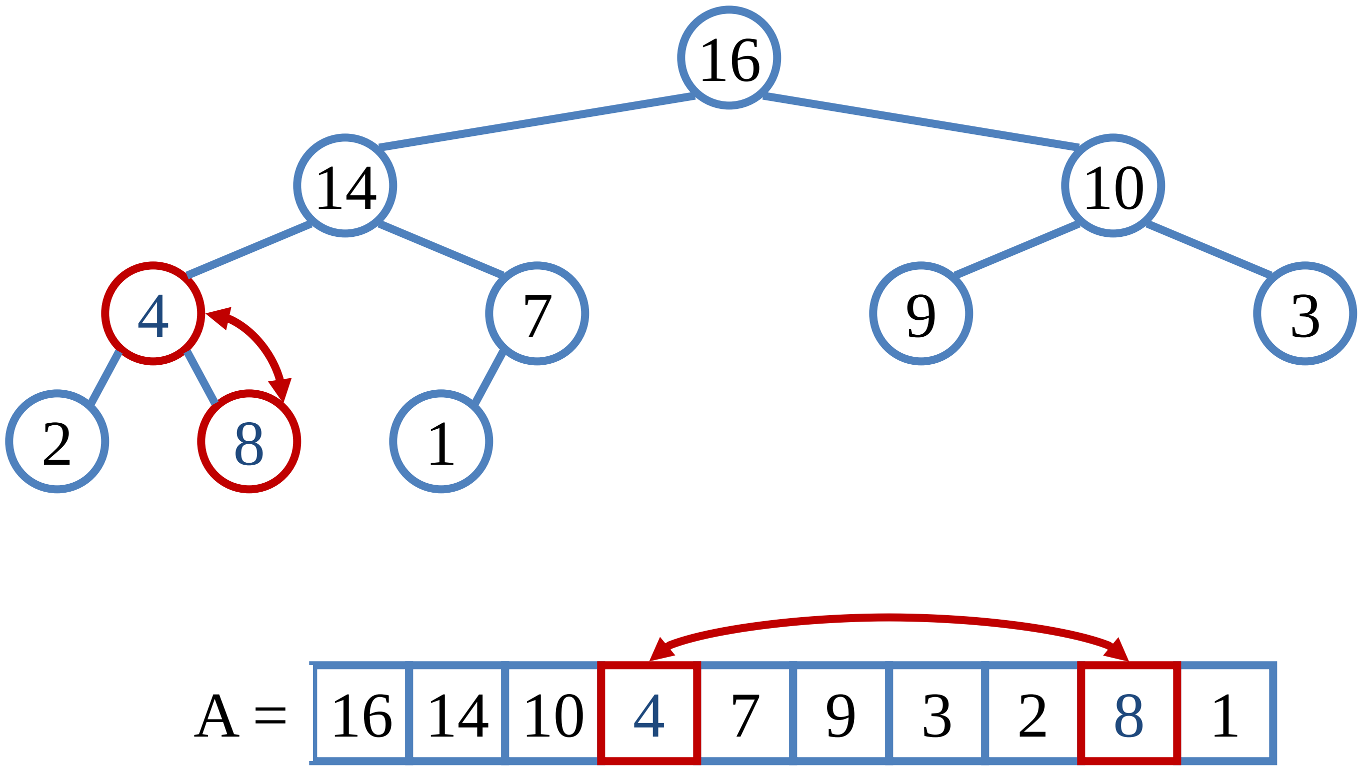
16	14	10	4	7	9	3	2	8	1

Heapify() Example

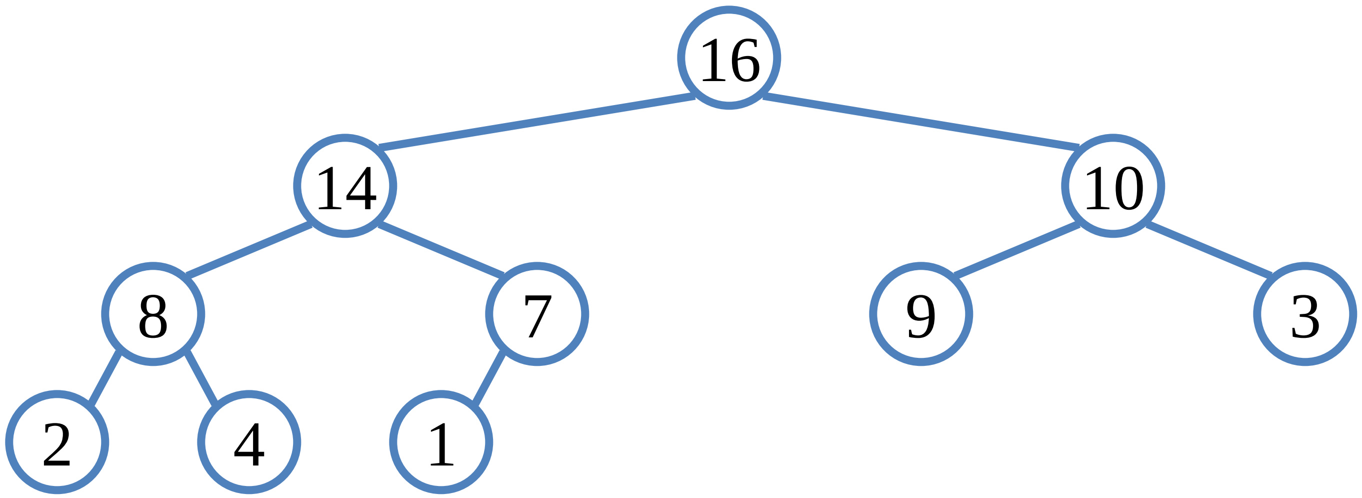


A =										
	16	14	10	4	7	9	3	2	8	1

Heapify() Example



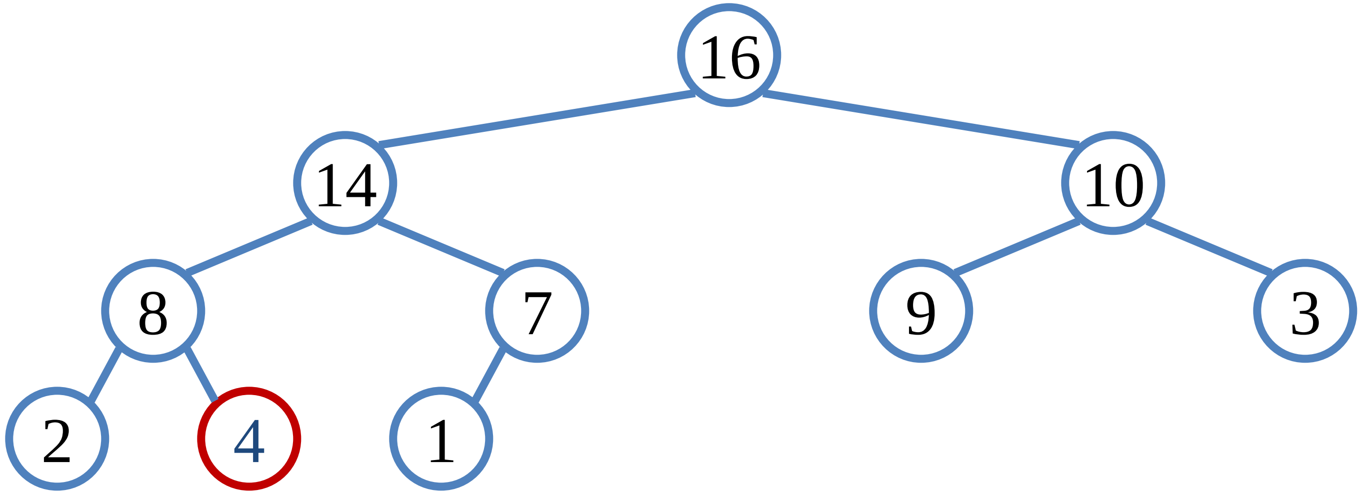
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

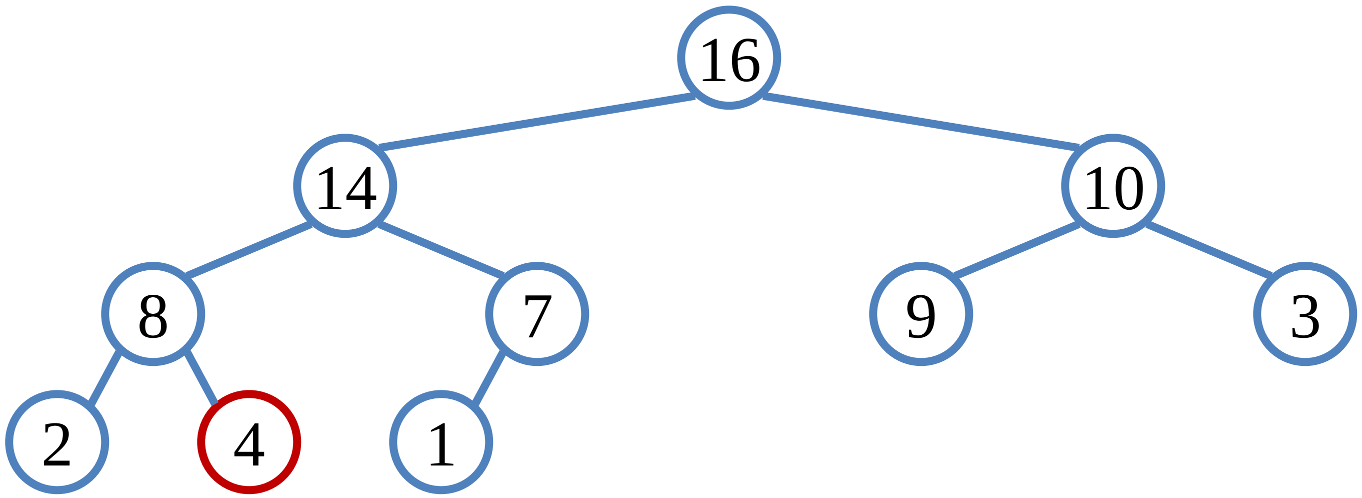
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =	16	14	10	8	7	9	3	2	4	1

Heap Operations: Heapify()

```
MAX-Heapify(A, i, Heap_size) // MAX-Heapify operation for MAX-Heap
{
  l = Left(i); r = Right(i);
  if (l <= heap_size(A) && A[l] > A[i])
      largest = l;
  else              //find the largest between A[l] and A[r]
      largest = i;
  if (r <= heap_size(A) && A[r] > A[largest])
      largest = r;
  if (largest != i)
      Swap(A, i, largest);
      MAX-Heapify(A, largest, Heap_size);
}
```

Analyzing Heapify(): Informal

- Aside from the recursive call, what is the running time of **`Heapify()`**?
- How many times can **`Heapify()`** recursively call itself?
- What is the worst-case running time of **`Heapify()`** on a heap of size n?

Analyzing Heapify(): Formal

- Fixing up relationships between *i*, *l*, and *r* takes $\Theta(1)$ time
- If the heap at *i* has *n* elements, how many elements can the subtrees at l or r have?
  - Draw it
- Answer: $2n/3$ (worst case: bottom row 1/2 full, (2n/3+n/3=n)
- So time taken by **`Heapify()`** is given by

$$T(n) \le T(2n/3) + \Theta(1)$$

Analyzing Heapify(): Formal

- So we have

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

- By case 2 of the Master Theorem,

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

- Thus, **`Heapify()`** takes logarithmic time

Heap Operations: BuildHeap()

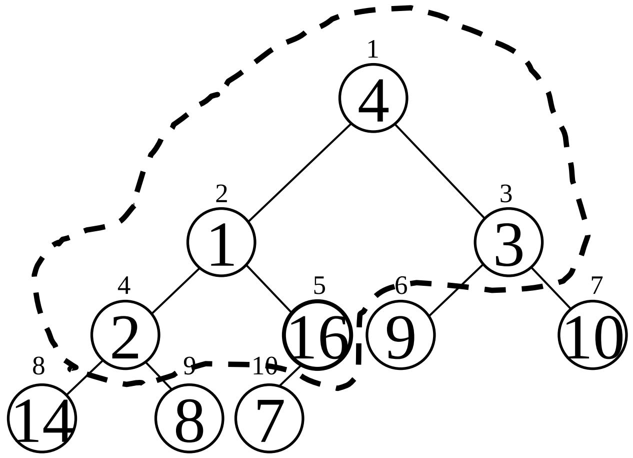
- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length n, all elements in range $A[\lfloor n/2 \rfloor + 1 .. n]$ are heaps (*Why?*)
  - The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

Building a Heap

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

Alg: BUILD-MAX-HEAP(A)

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

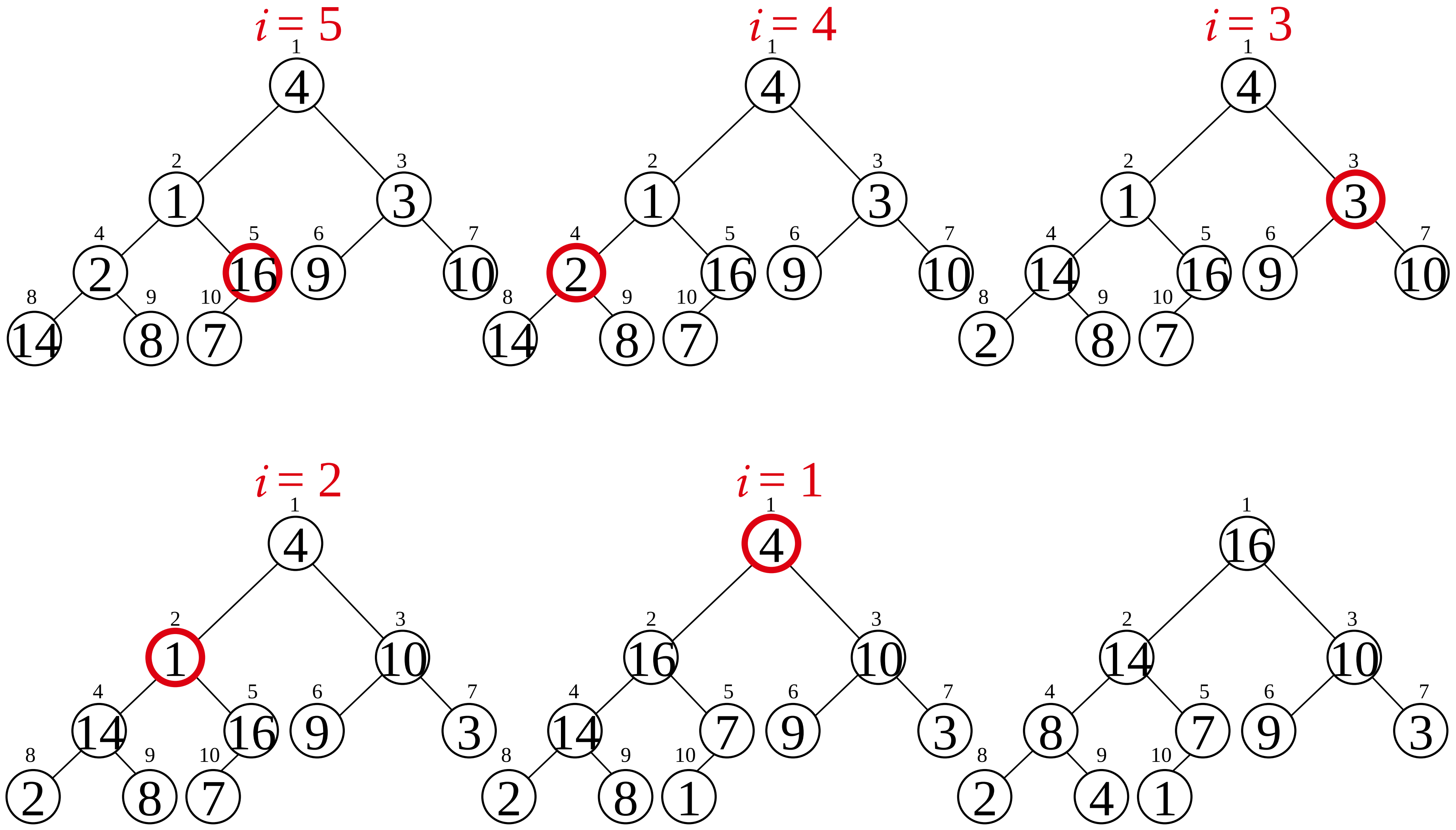


A:

4	1	3	2	16	9	10	14	8	7

Example: A

4	1	3	2	16	9	10	14	8	7



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) $O(\lg n)$

$O(n)$

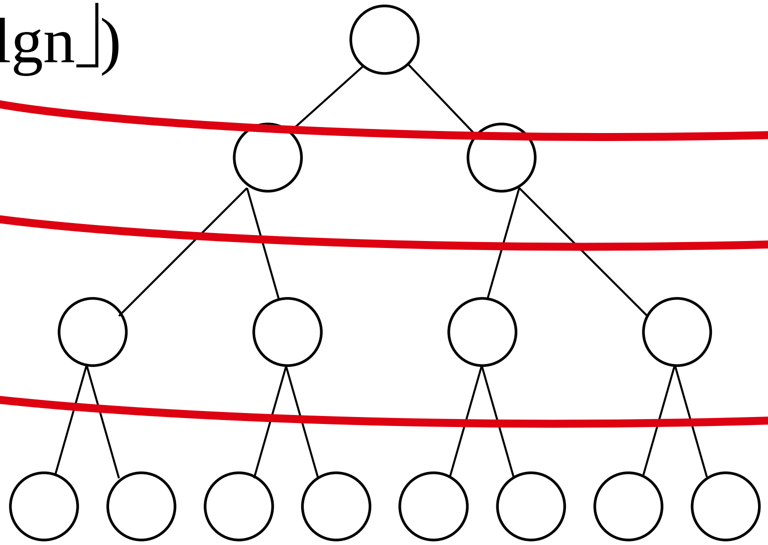
$\Rightarrow$ Running time: $O(n\lg n)$

- 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 $\boxed{\frac{1}{2^{-i}}}$

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

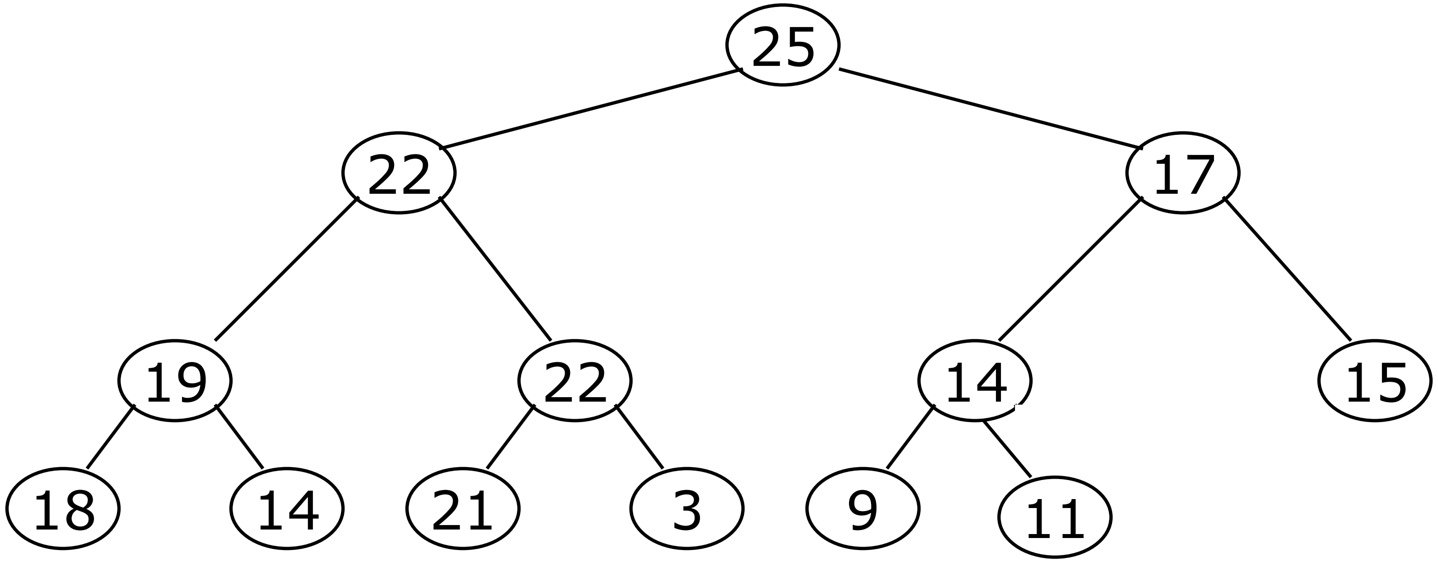
$$\le n \sum_{k=0}^{\infty} \frac{k}{2^k}$$ The sum above is smaller than the sum of all elements to ∞ and h = lgn

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

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

Removing the root

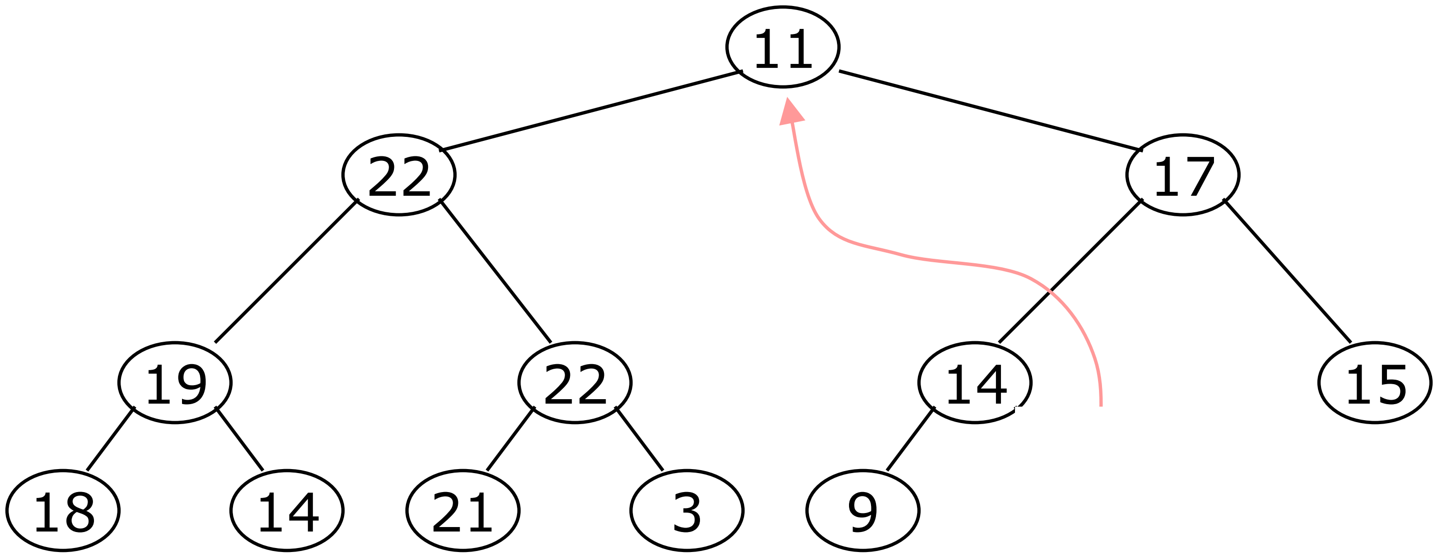
- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- How can we fix the binary tree so that it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

Removing the root

- Notice that the largest number is now in the root
- Suppose we *discard* the root:



- How can we fix the binary tree so it is once again *balanced and left-justified?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root

Removing and replacing the root

- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...

1	2	3	4	5	6	7	8	9	10	11	12	13
25	22	17	19	22	14	15	18	14	21	3	9	11

1	2	3	4	5	6	7	8	9	10	11	12	13
11	22	17	19	22	14	15	18	14	21	3	9	25

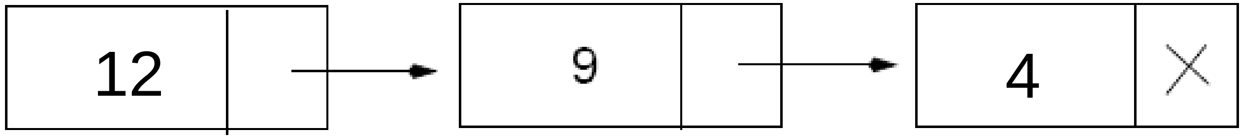
- ...And pretend that the last element in the array no longer exists—that is, the “last index” is 12 (9)

Removing and replacing the root

```
DEL_HEAP(A, N)
// the procedure deletes the max from heap A[1:N] and
stores it in ITEM
{
    if N = 0 then "heap is empty"
     SWAP (A, 1, N)
    MAX_HEAPIFY(A,1,N-1)
}
```

Priority Queues

- Each element is associated with a value (Priority)
- A priority queue is different from a "normal" queue, because instead of being a "first-in-first-out" data structure, values come out in order by priority.
- The key with highest (Lowest) priority is dequeued first



Operations on Priority Queues

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

HEAP-MAXIMUM

Goal:

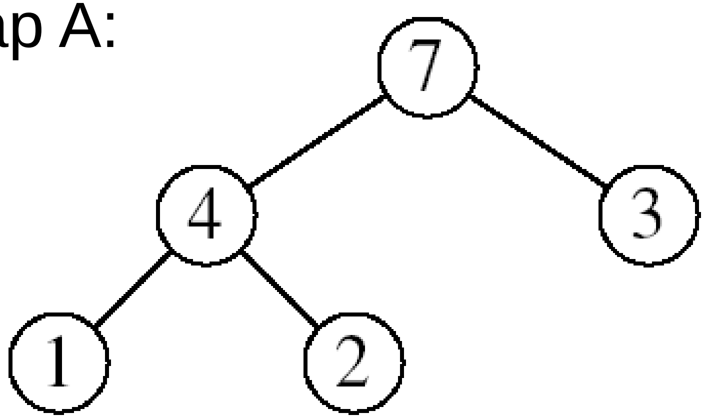
– Return the largest element of the heap

Running time: $O(1)$

Alg: HEAP-MAXIMUM(A)

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

Heap A:



Heap-Maximum(A) returns 7

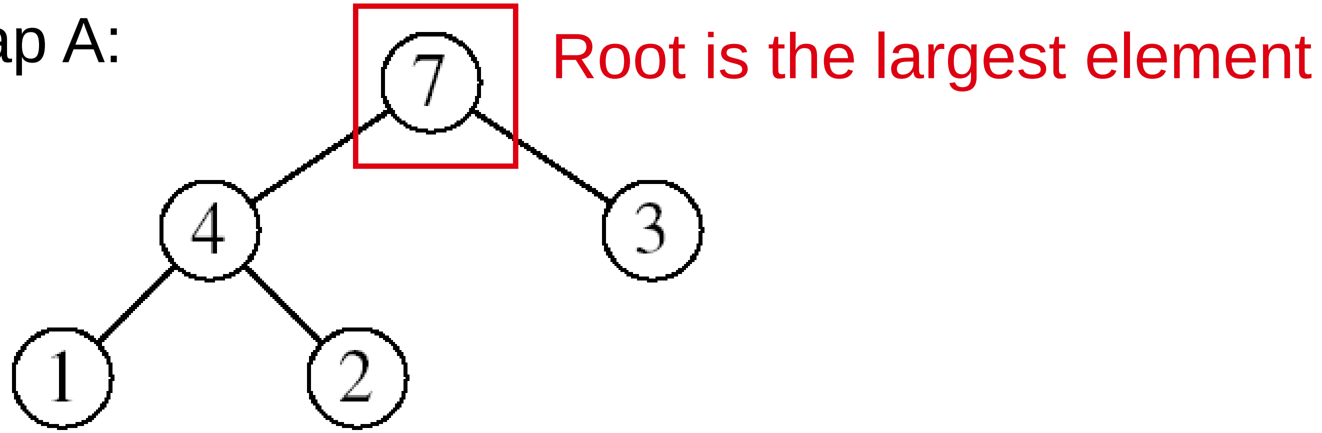
HEAP-EXTRACT-MAX

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

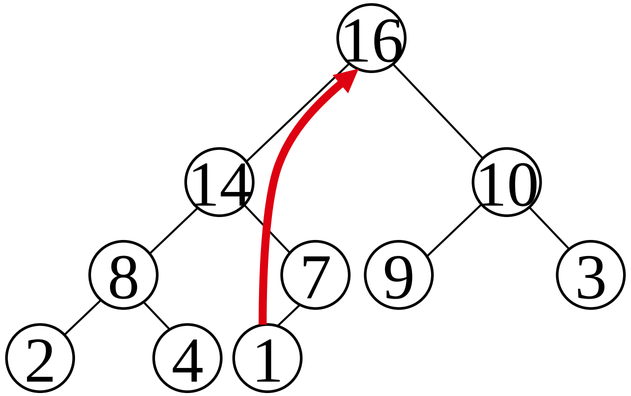
Idea:

- 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

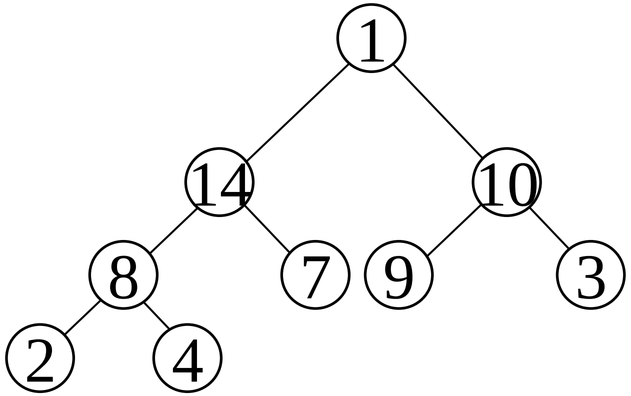
Heap A:



Example: HEAP-EXTRACT-MAX

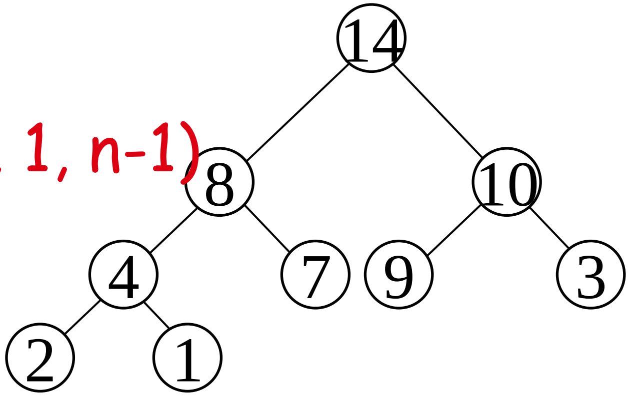


max = 16



Heap size decreased by 1

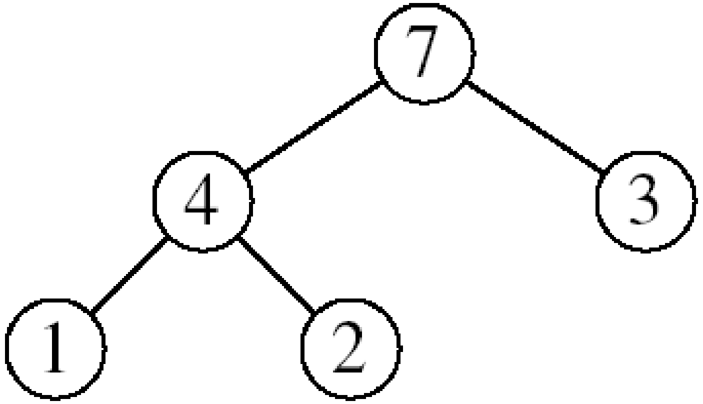
Call MAX-HEAPIFY(A, 1, n-1)



HEAP-EXTRACT-MAX

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

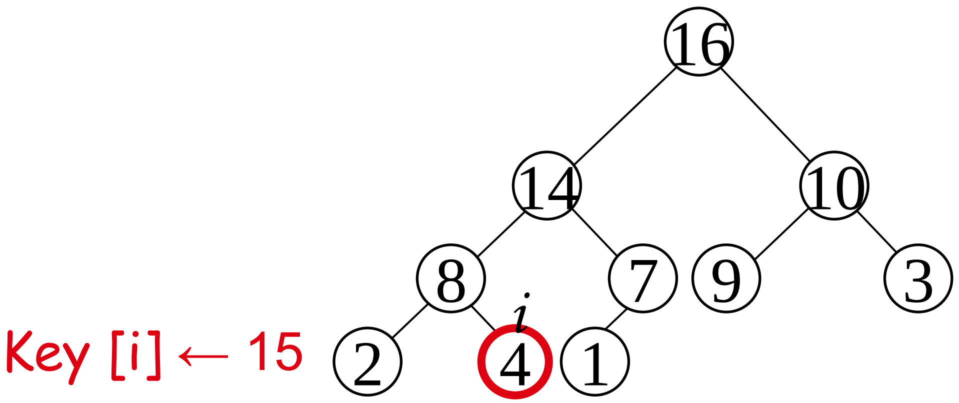
1. **if** $n < 1$
2. **then error** "heap underflow"
3. $max \leftarrow A[1]$
4. $A[1] \leftarrow A[n]$
5. MAX-HEAPIFY(A, 1, n-1) //remakes heap
6. **return** max



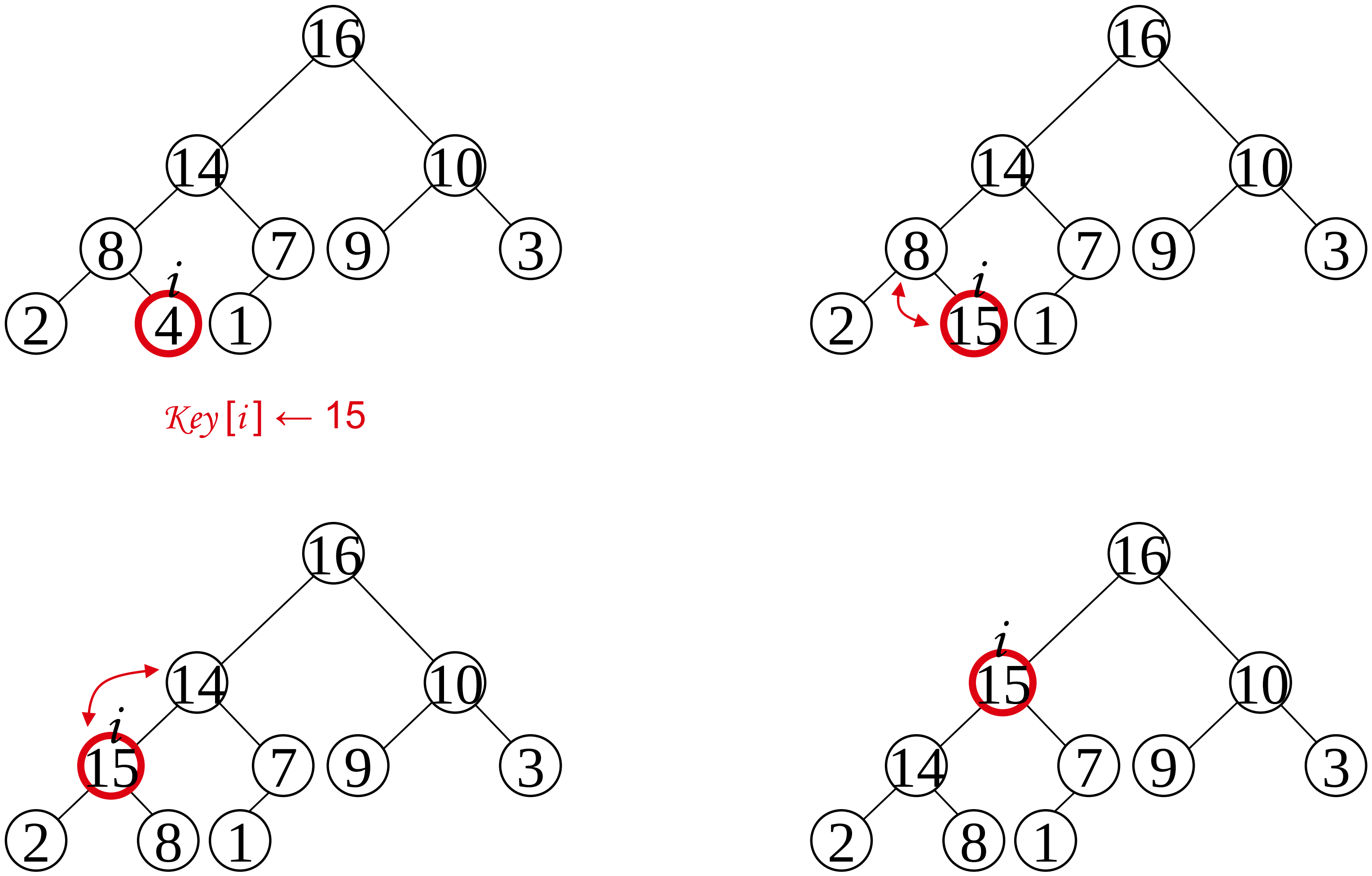
Running time: $O(\lg n)$

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

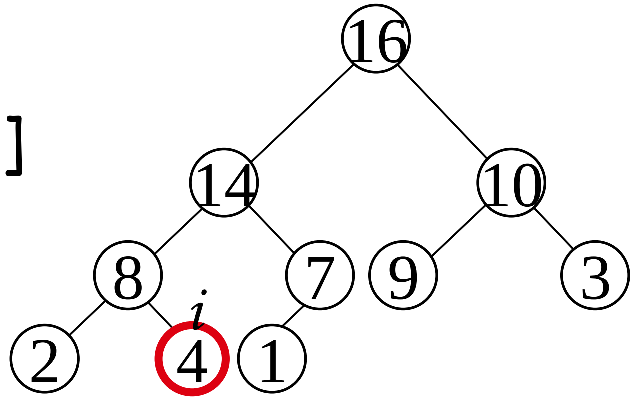


HEAP-INCREASE-KEY

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

```
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ↔ A[PARENT(i)]
        i ← PARENT(i)
```

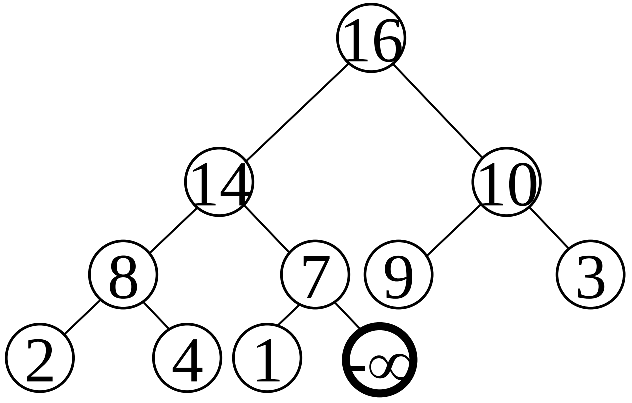
- Running time: $O(\lg n)$

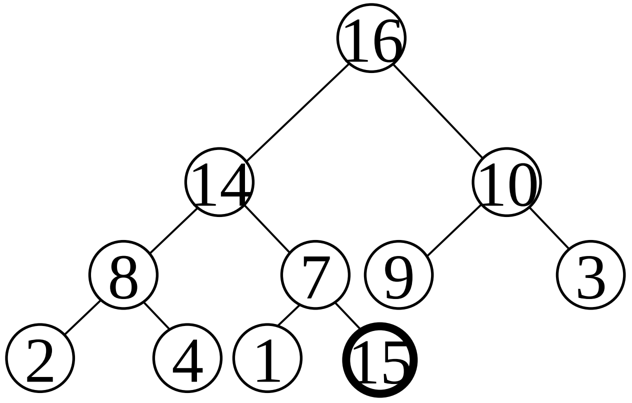


Key [i] ← 15

MAX-HEAP-INSERT

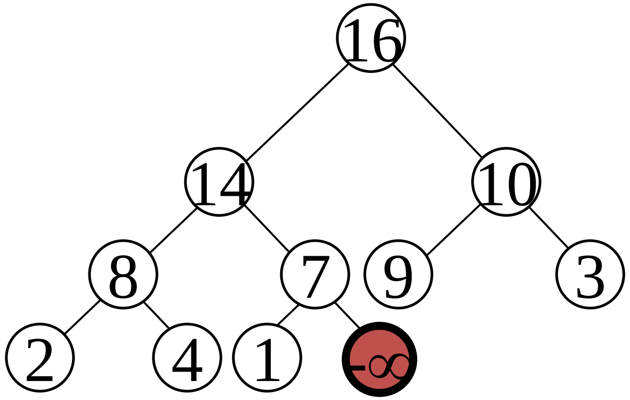
- Goal:
  - Inserts a new element into a max-heap
- Idea:
  - Expand the max-heap with a new element whose key is -∞
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property



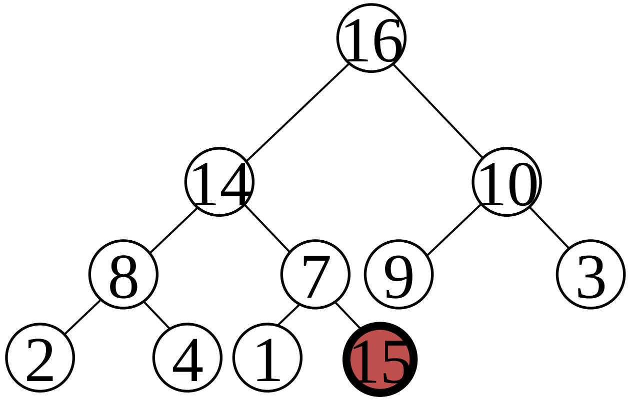


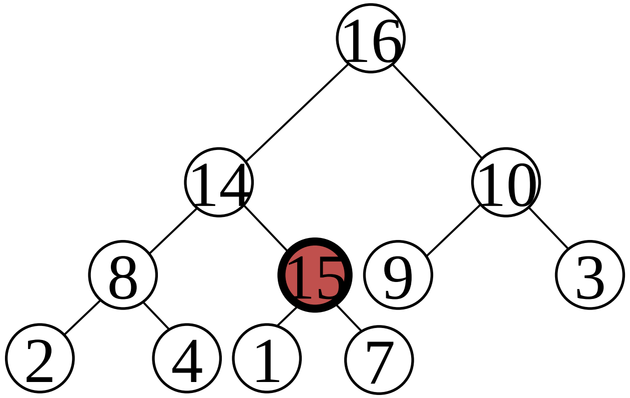
Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

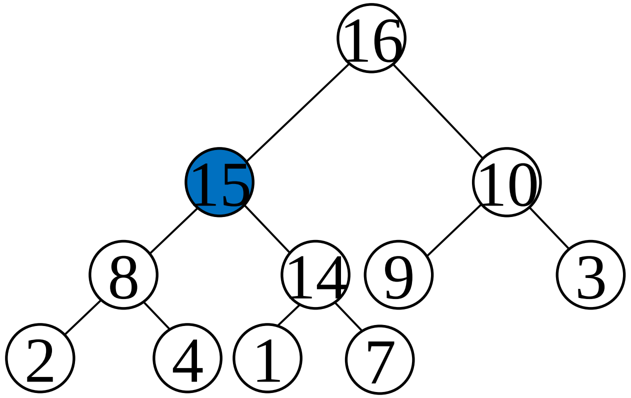


Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15





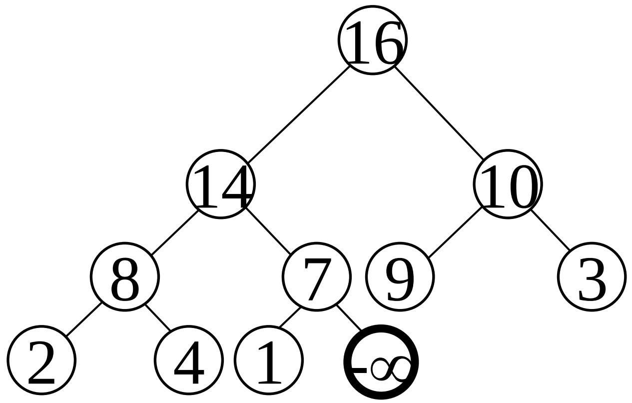
The restored heap containing
the newly added element



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)



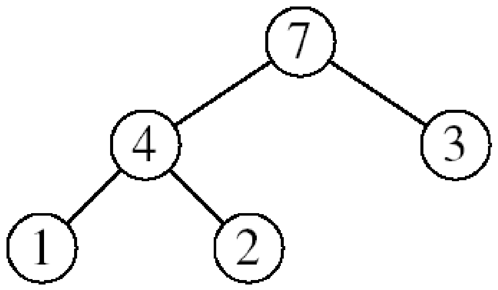
Running time: $O(\lg n)$

Why study Heapsort?

- It is a well-known, traditional sorting algorithm you will be expected to know
- Heapsort is *always* O(n log n)
  - Quicksort is usually O(n log n) but in the worst case slows to $O(n^2)$
  - Quicksort is generally faster, but Heapsort is better in time-critical applications

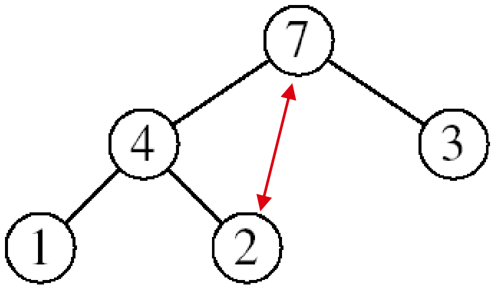
Heapsort

- Goal:
  - Sort an array using heap representations

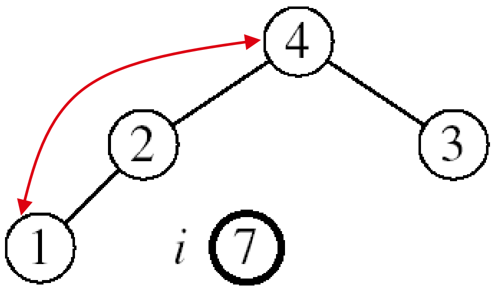


- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

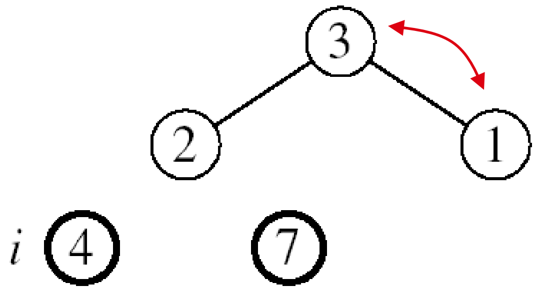
Example: A=[7, 4, 3, 1, 2]



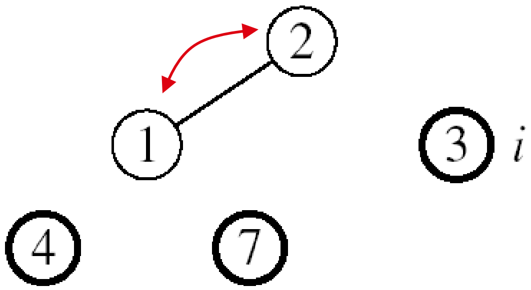
MAX-HEAPIFY(A, 1, 4)



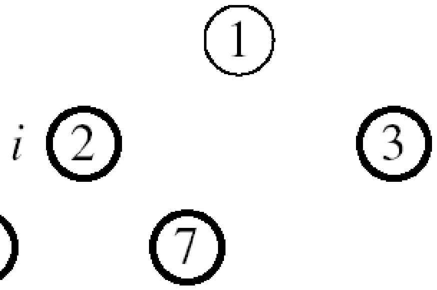
MAX-HEAPIFY(A, 1, 3)



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 $A[1] \leftrightarrow A[i]$
4. MAX-HEAPIFY(A, 1, i - 1) $O(\lg n)$

n-1 times

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

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 $O(1)$

References

- David Matuszek, University Of Pennsylvania
  – www.cis.upenn.edu/~matuszek/
- Dr. George Bebis, University of Nevado
  – Course page: www.cse.unr.edu/~bebis