Heaps

6.006 Lecture 4

Image from www.telegraph.co.uk

Today's plan

- Priority Queues
- Heaps
  - Heapsort: Another $O(n \log n)$ sorting algorithm!

Priority Queue

An *abstract data structure* implementing a set S of *elements*, each associated with a *key*, supporting the following operations:

insert(S, x) : insert element x = (name(x), key(x)) into set S

max(S) : return element of S with largest key

extract_max(S) : return element of S with largest key and remove it from S

increase_key(S, x, k) : Change key-value of x to k (assumed to be increase)

Tons of applications!

e.g. triage in hospital based on how bad things are going

Priority Queue

$\text{insert}(S, x)$: insert element $x = (name(x), key(x))$ into set S

$\max(S)$: return element of S with largest key

$\text{extract\_max}(S)$: return element of S with largest key and remove it from S

$\text{increase\_key}(S, x, k)$: increase the value of element x' s key to new value k
(assumed to be as large as current value)

S:

$(Debayan, 25)$
$(Erik, 15)$

Insert(S, (Ronitt,5)):

$(Debayan, 25)$
$(Erik, 15)$
$(Ronitt, 5)$

Extract_max(S):

$(Erik, 15)$
$(Ronitt, 5)$

Increase_key(S, (Ronitt,33)):

$(Erik, 15)$
$(Ronitt, 33)$

How do we best implement it?

Priority Queue: First (slow) idea

Maintain an (unsorted) array?

insert(S, x) : insert x into S $O(1)$

max(S) : return largest element $O(n)$ to find

extract_max(S) : return largest element and remove

$O(n)$ to find

increase_key(S, x, k) : increase the value of element x' s key to new value k

(assumed to be as large as current value)

$O(n)$ to find

Priority Queue: Second (slow) idea

Maintain a sorted array (based on keys)?

insert(S, x) : insert x into S $O(n)$ to shift others right

max(S) : return largest element $O(1)$ to find

extract_max(S) : return largest element and remove

$O(1)$ to find, then $O(n)$ to shift others left

increase_key(S, x, k) : increase the value of element x' s key to new value k
(assumed to be as large as current value)

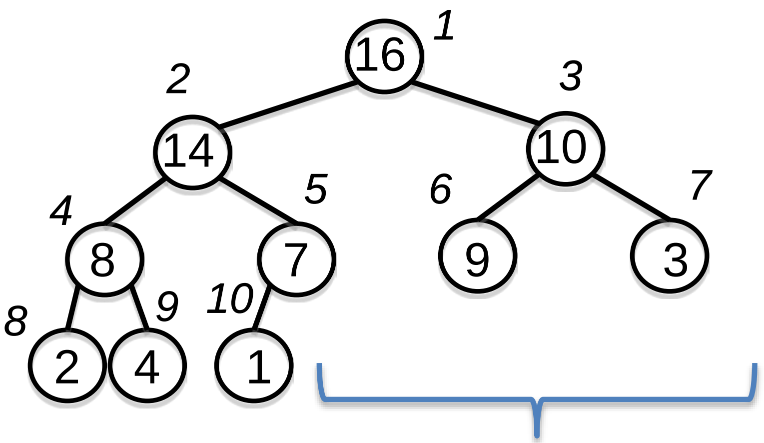
$O(n)$ to find, then $O(n)$ to shift left

Let's apply our 6.006 data structure superpowers!

New data structure

(Max) Heap

- A nearly complete binary tree
- Max Heap Property (MHP): key of a node ≥ keys of its children (Min Heap defined analogously)



Missing only rightmost nodes on bottom level

Important fact:
Height of the tree is always O(log n)

Heap Operations

insert:
max:
extract_max:
increase_key:

build_max_heap : produce a max-heap from an unordered array

max_heapify : correct a single violation of the heap property at the root (rest of heap is fine)

Heap operations can be used to implement priority queue!

Representation via *array*

root: first element in the array (**i=1**)

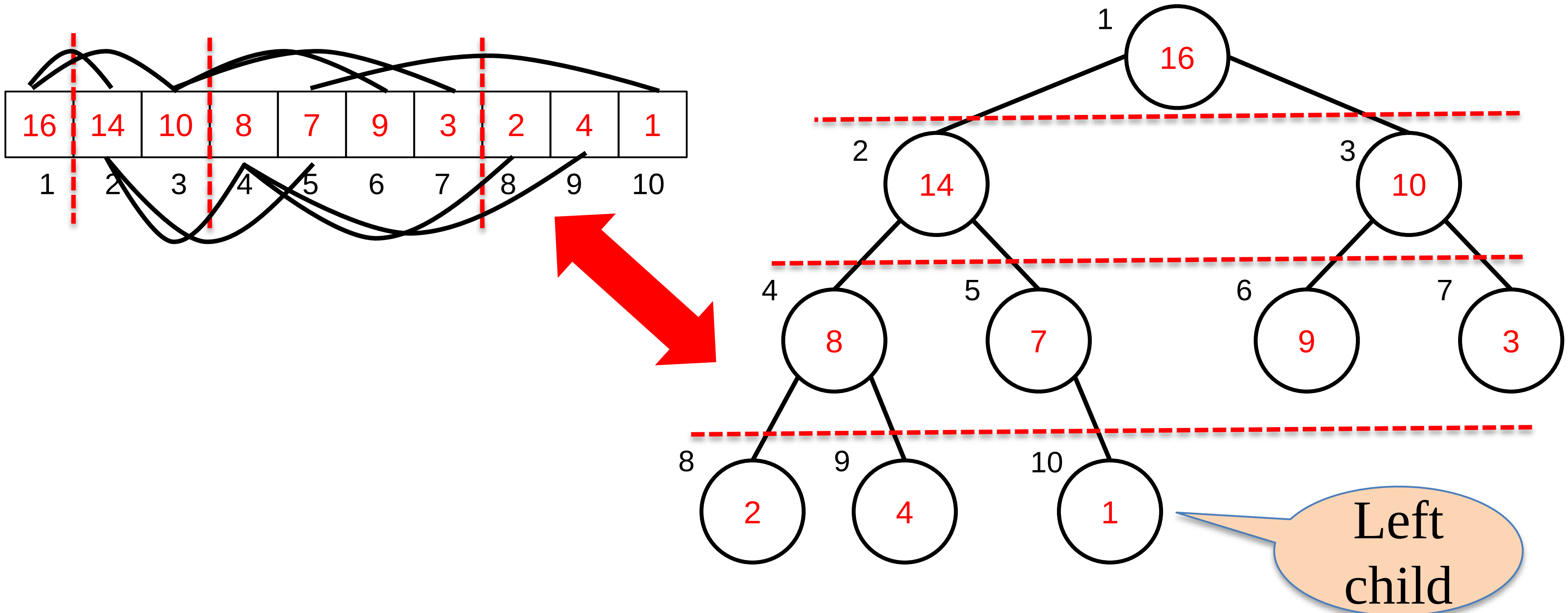
parent(i): **floor(i/2)** returns index of node's parent

left(i): **2i** returns index of node's *left* child

right(i): **2i+1** returns index of node's *right* child

(**Why?** No pointers needed!)

Important detail: index elements starting from **i=1**



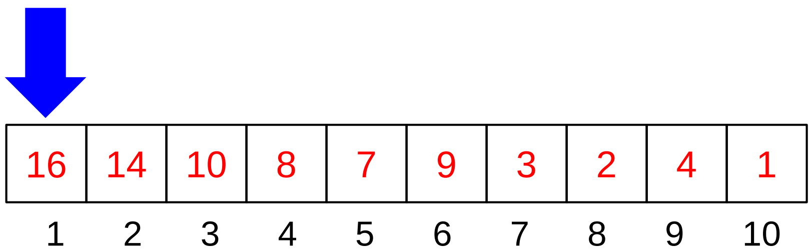
Why Heaps?

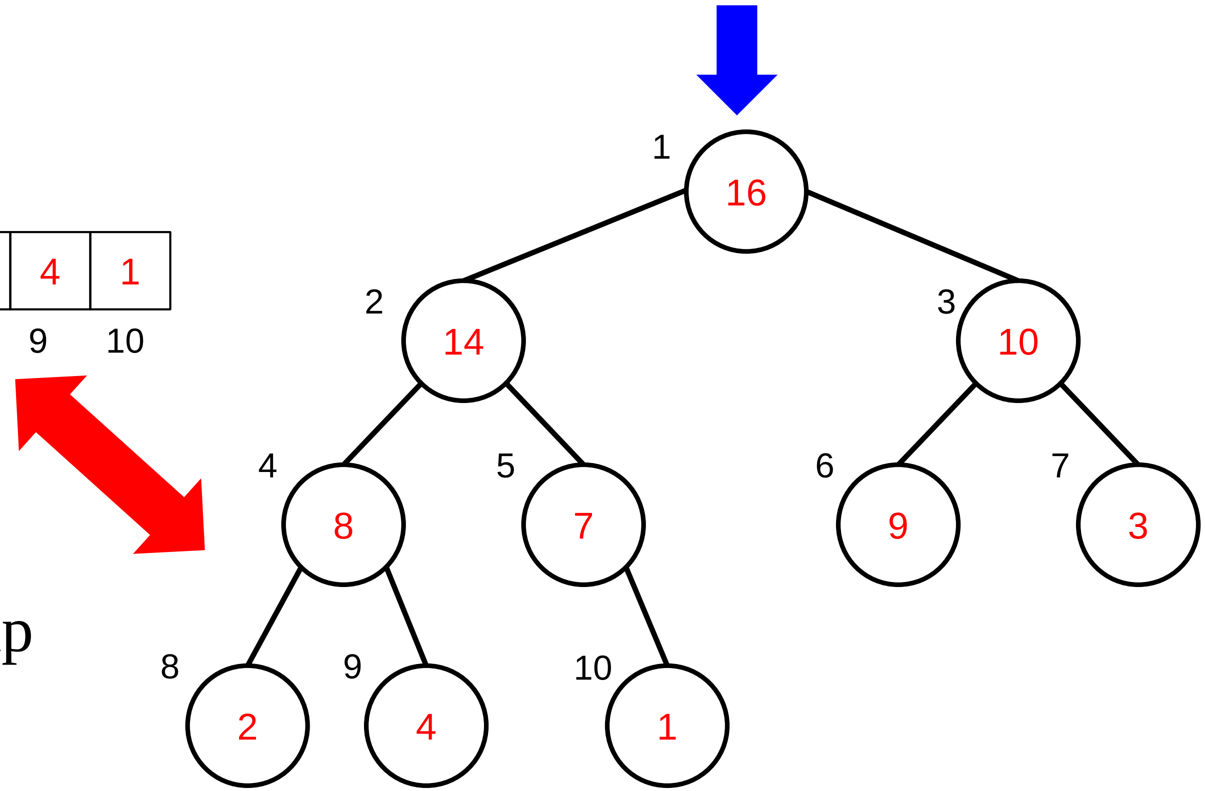
Key consequence of Max Heap Property:

Root/first element is always the max ➔ can do max(S) in **Θ(1) time**!

(**Note:** the array is not sorted though!)

But: How to maintain the Max Heap Property property after insert/extract_max/increase_key?





In fact:

How to build a Max Heap to begin with?

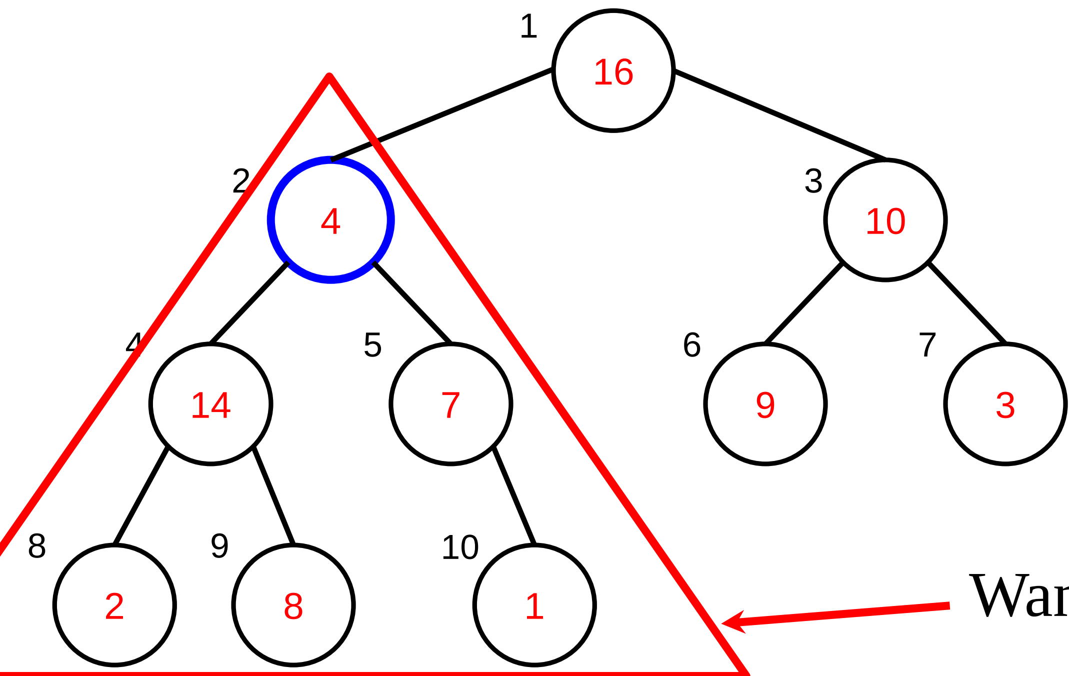
Key Primitive

max_heapify(A[i]): Corrects a **single** violation of Max Heap Property in a subtree rooted at **i** **only**

How to implement it?

➔ Assume that the trees rooted at left(i) and right(i) are Max Heaps

➔ If element A[i] violates the MHP, correct violation by "trickling" this element down the tree, making the subtree rooted at i a Max Heap (**Important:** Always swap with the larger of two children. **Why?**)



Want this to be a Max Heap

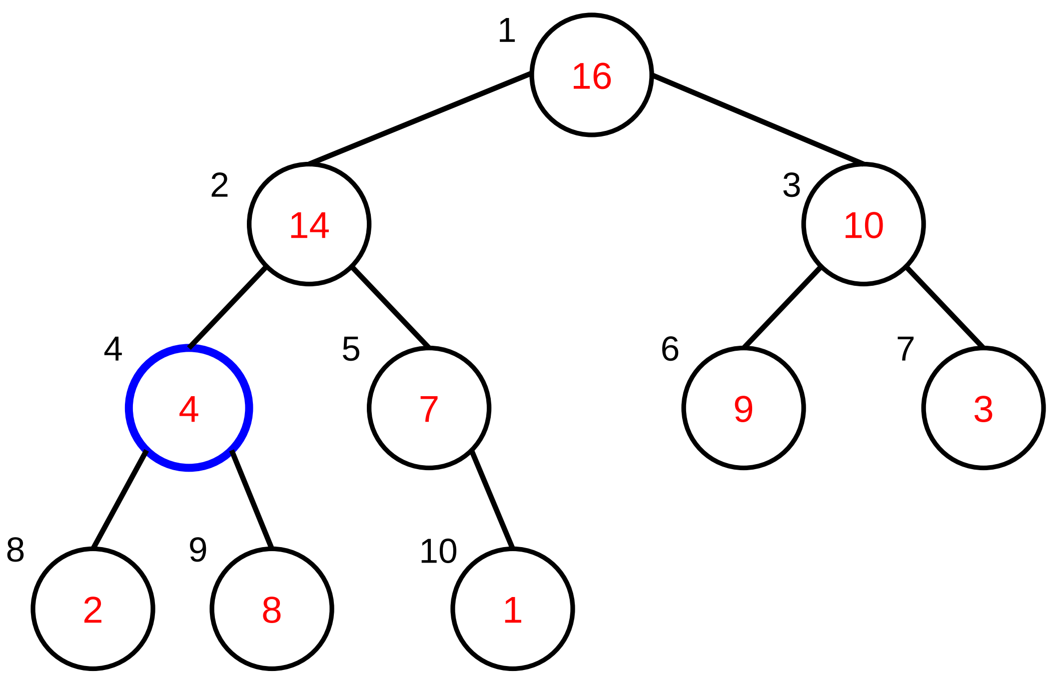
Key Primitive

max_heapify(A[i]): Corrects a **single** violation of Max Heap Property in a subtree rooted at **i** **only**

In other words:

➔Assume that the trees rooted at left(i) and right(i) are Max Heaps

➔If element A[i] violates the MHP, correct violation by "trickling" this element down the tree, making the subtree rooted at i a Max Heap (**Important:** Always swap with the larger of two children)



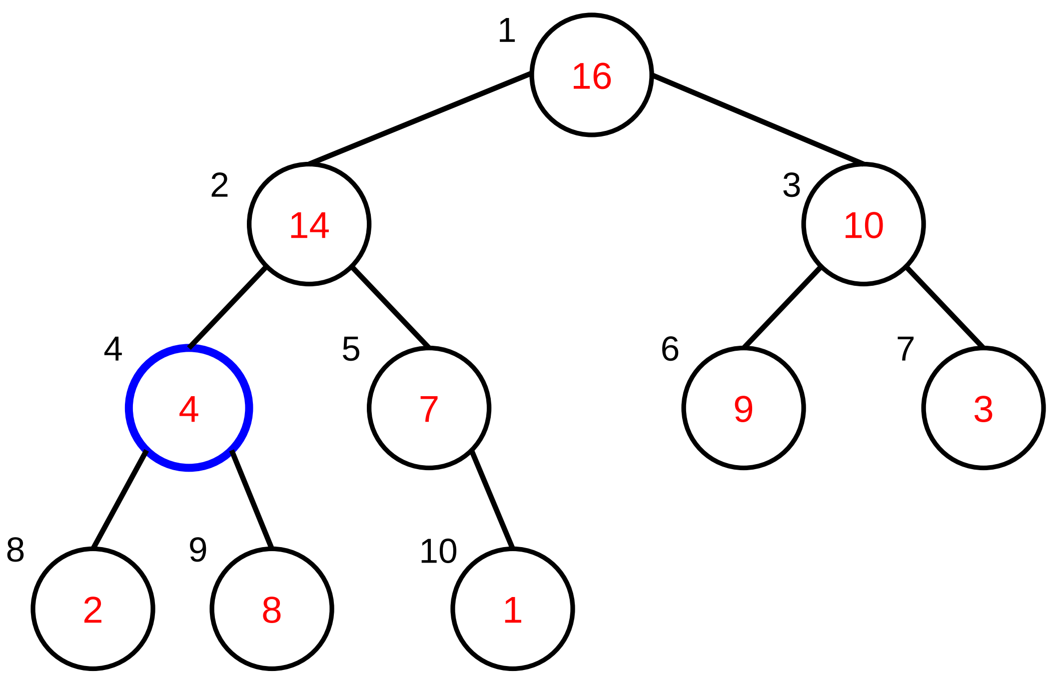
Key Primitive

max_heapify(A[i]): Corrects a **single** violation of Max Heap Property in a subtree rooted at **i** **only**

In other words:

➔Assume that the trees rooted at left(i) and right(i) are Max Heaps

➔If element A[i] violates the MHP, correct violation by "trickling" this element down the tree, making the subtree rooted at i a Max Heap (**Important:** Always swap with the larger of two children)



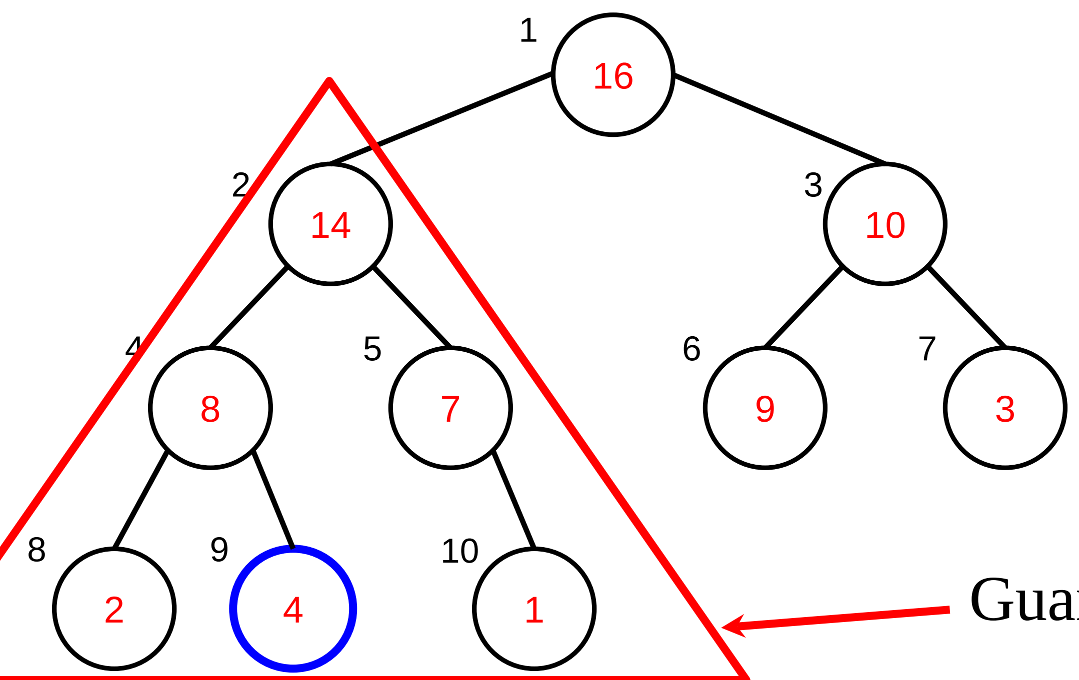
Key Primitive

max_heapify(A[i]): Corrects a **single** violation of Max Heap Property in a subtree rooted at **i** **only**

In other words:

➔ Assume that the trees rooted at left(i) and right(i) are Max Heaps

➔ If element A[i] violates the MHP, correct violation by "trickling" this element down the tree, making the subtree rooted at i a Max Heap (**Important:** Always swap with the larger of two children)



Run time?

➔ **Θ(1)** at each level

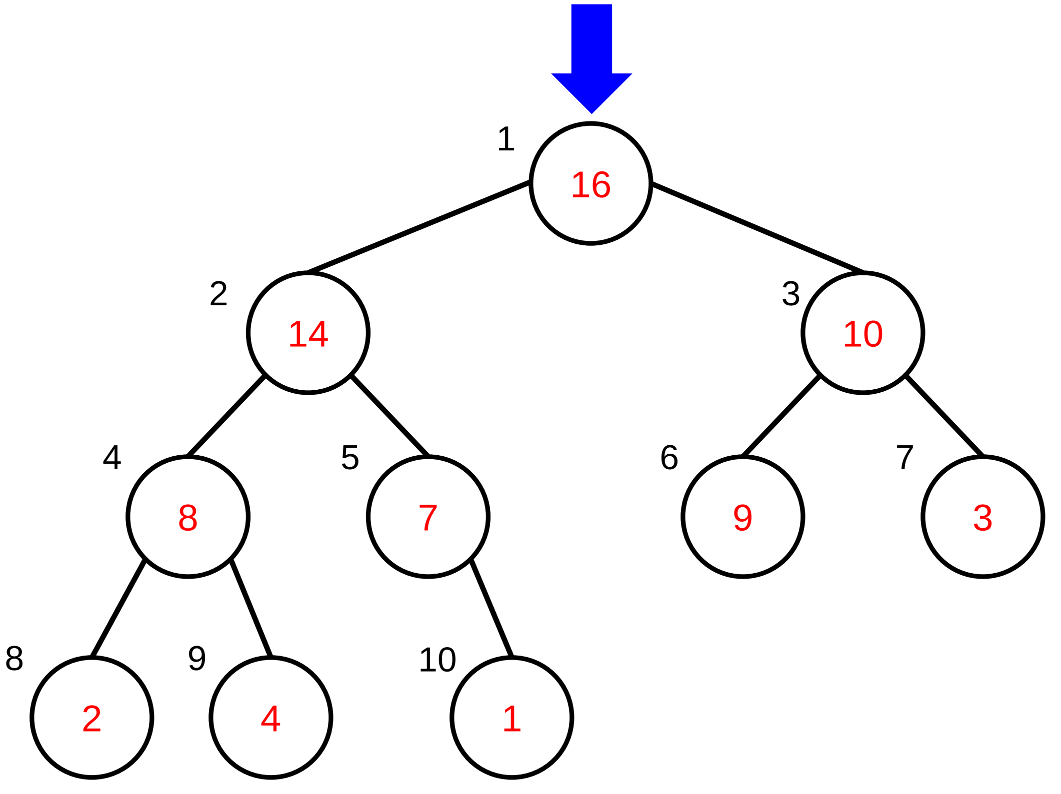
➔ total: **O(subtree height)**

= Θ (log n) (worst case)

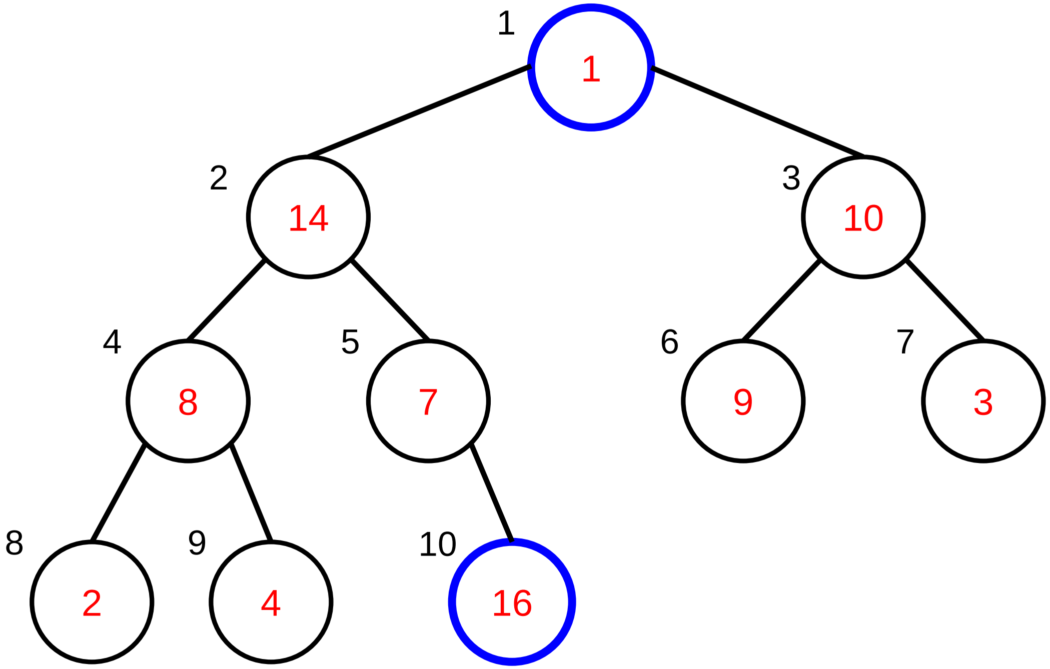
(**Recall:** tree is balanced)

Guaranteed to be a Max Heap now

Implementing Extract_Max

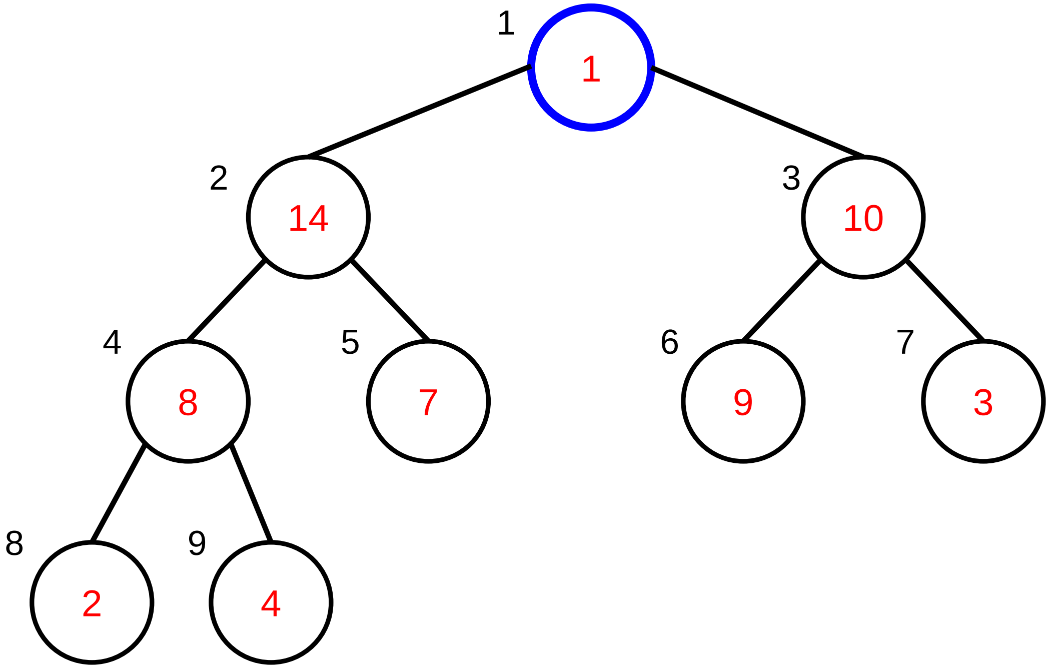


Implementing Extract_Max



- Swap the root with the last element of the heap

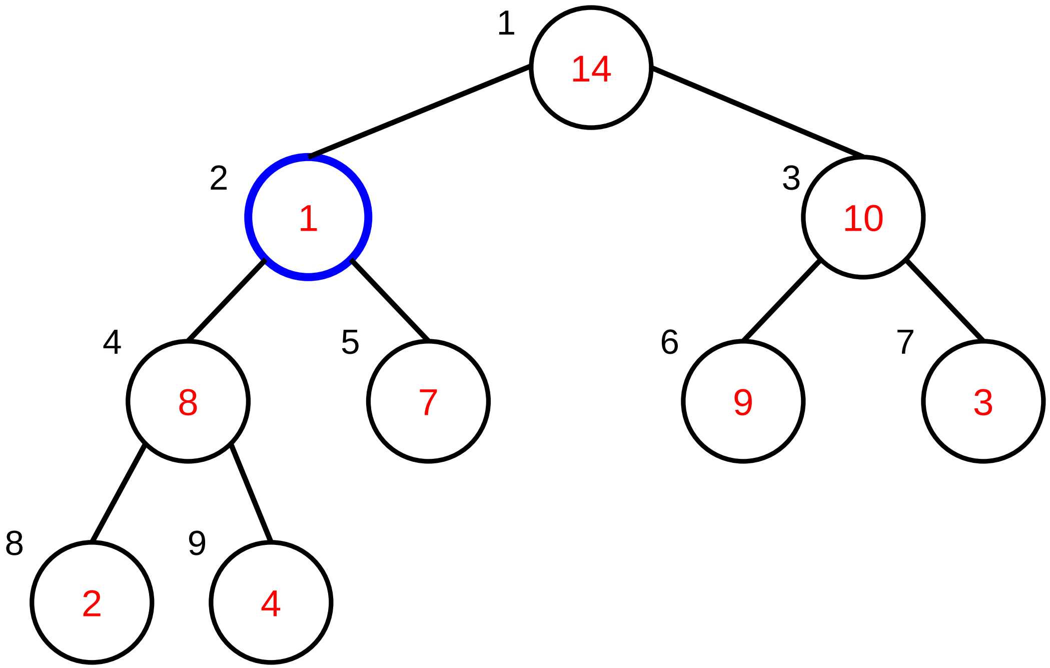
Implementing Extract_Max



- Swap the root with the last element of the heap
- Now we can remove it from the heap

  (decrease the heap size by one)
- **To fix MHP:** Max_heapify the root

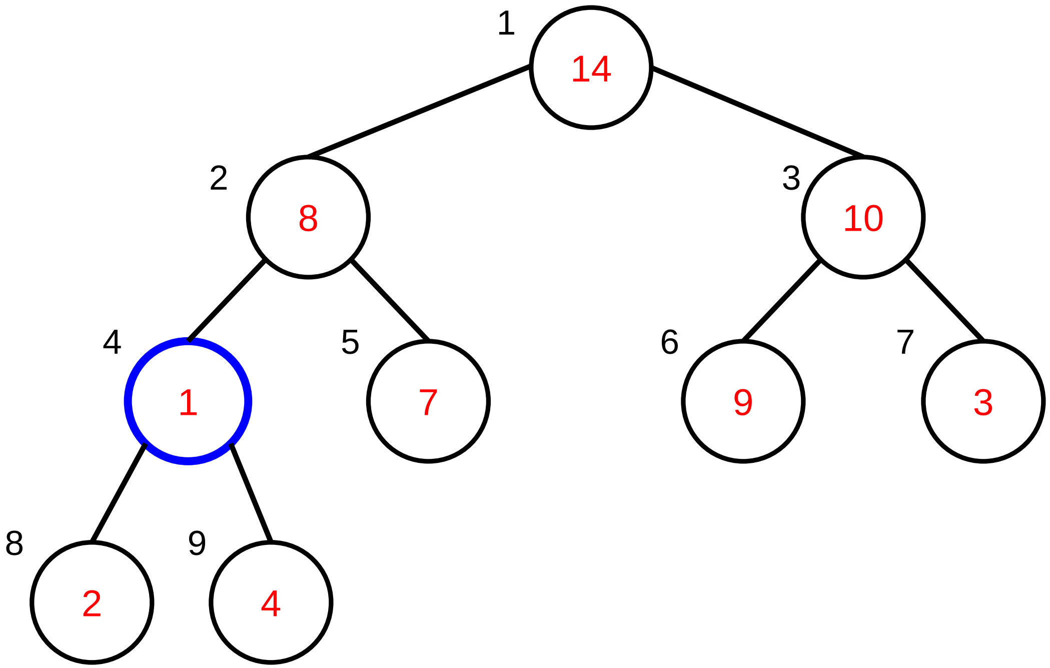
Implementing Extract_Max



- Swap the root with the last element of the heap
- Now we can remove it from the heap

  (decrease the heap size by one)
- **To fix MHP:** Max_heapify the root

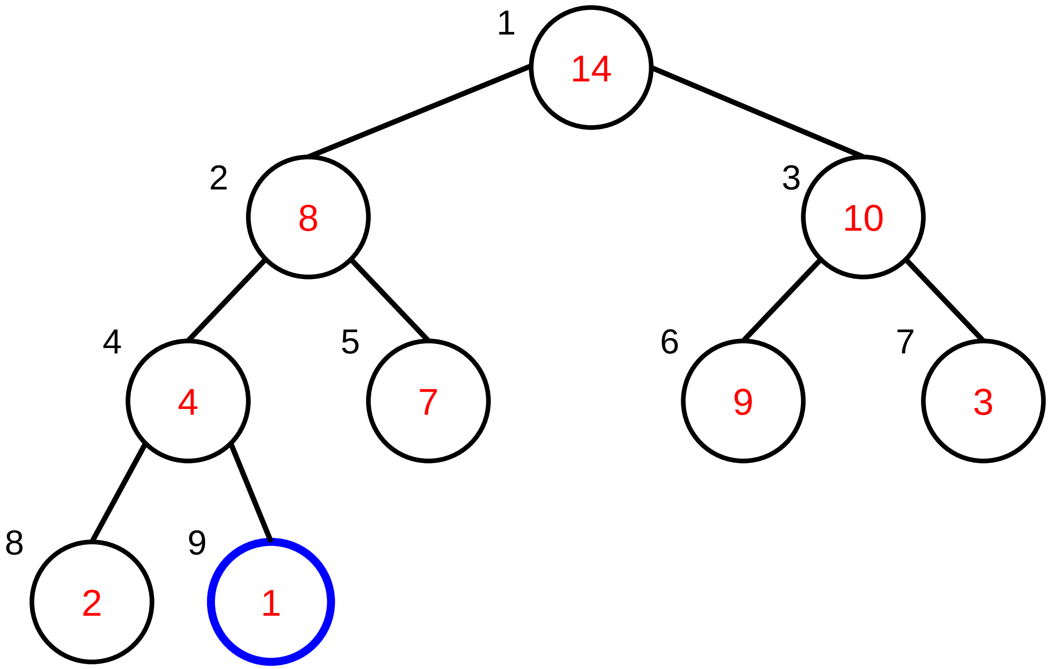
Implementing Extract_Max



- Swap the root with the last element of the heap
- Now we can remove it from the heap

  (decrease the heap size by one)
- **To fix MHP:** Max_heapify the root

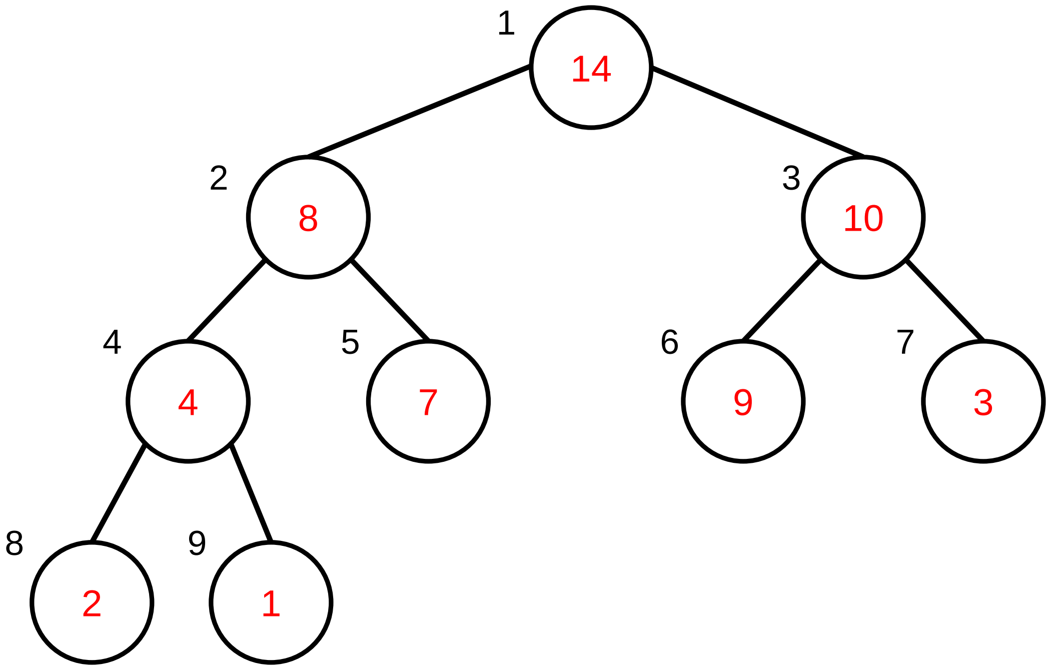
Implementing Extract_Max



- Swap the root with the last element of the heap
- Now we can remove it from the heap

  (decrease the heap size by one)
- **To fix MHP:** Max_heapify the root

Implementing Extract_Max



- Swap the root with the last element of the heap
- Now we can remove it from the heap

  (decrease the heap size by one)
- **To fix MHP:** Max_heapify the root
- Done!

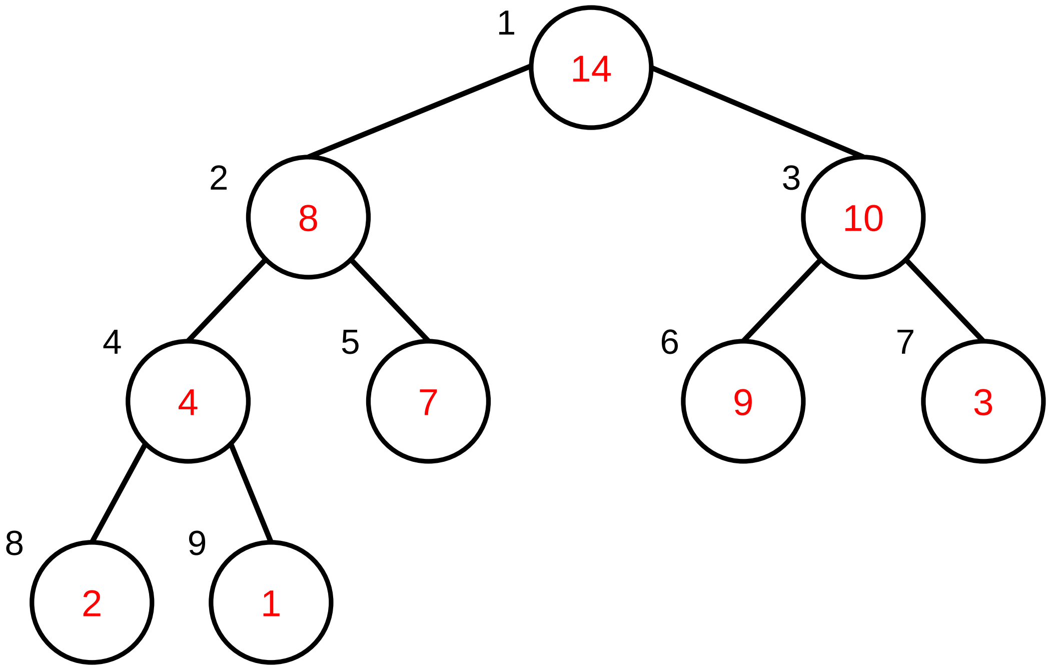
Run time?

➔ **$\Theta(1)$** (swapping) **+ $\Theta(1)$** (removal) **+ O (log n)** (max_heapify)

➔ total: **$\Theta(\log n)$ (worst case)**

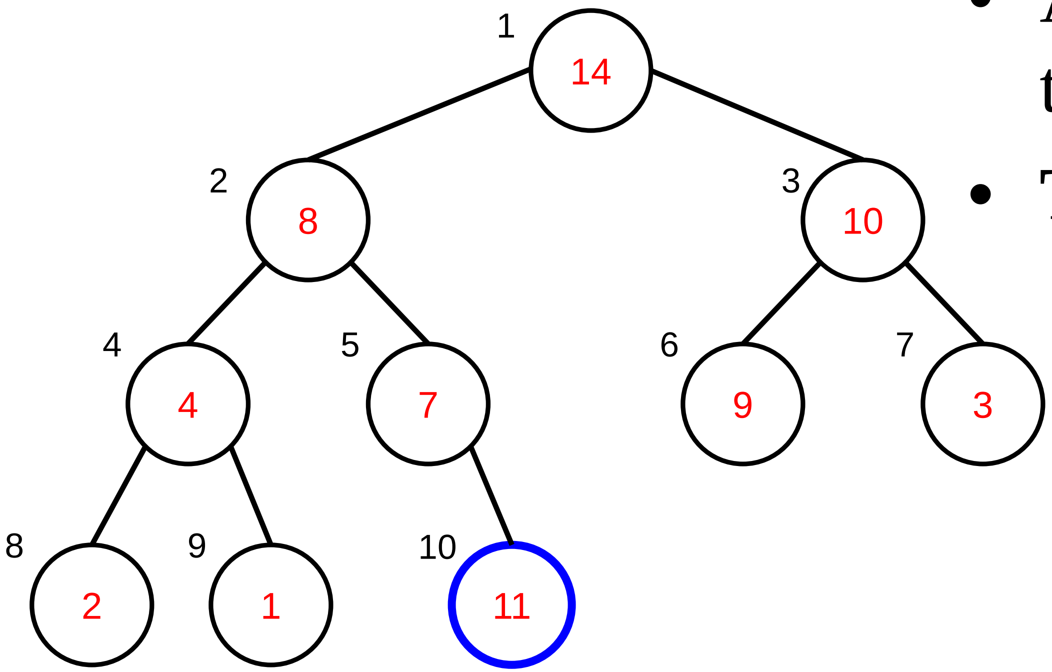
Implementing Insert

(**in a sense:** "reversing" the extract_max)



Implementing Insert

(in a sense: “reversing” the extract_max)



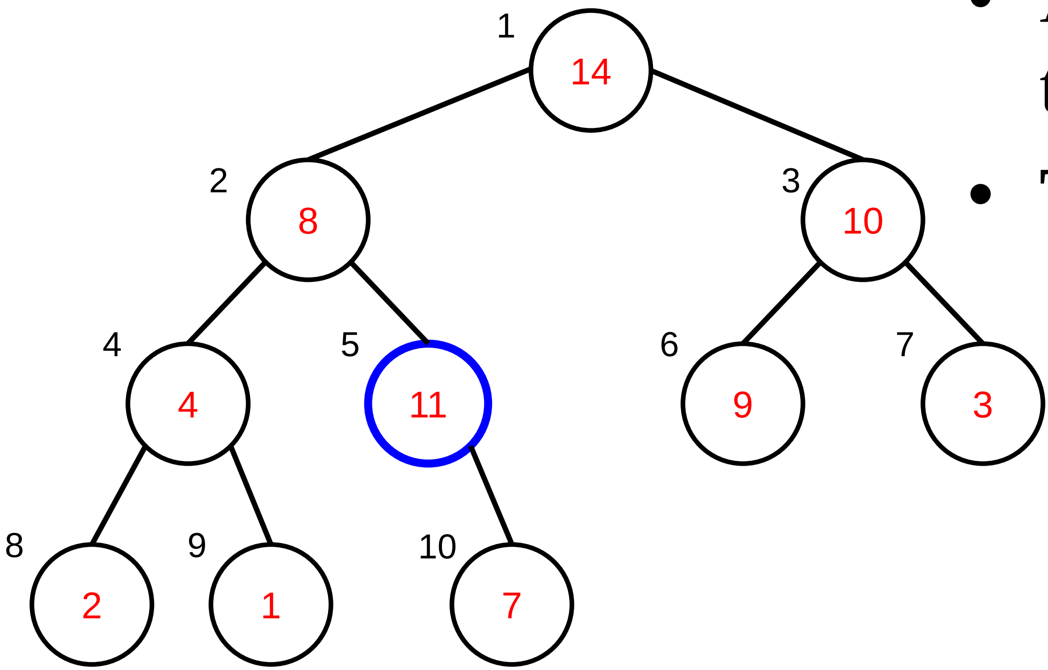
- Add the new element as the last one
- **To fix MHP:**

  “Promote” the new element up the tree

  (“reversed” max_heapify)

Implementing Insert

(in a sense: "reversing" the extract_max)



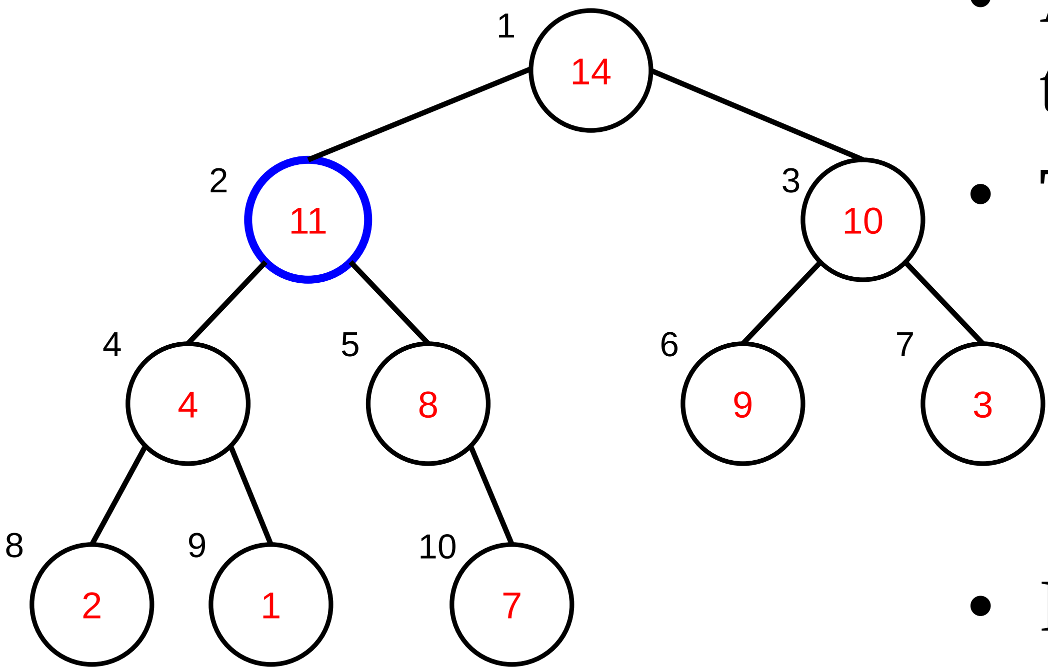
- Add the new element as the last one
- **To fix MHP:**

  "Promote" the new element up the tree

  ("reversed" max_heapify)

Implementing Insert

(**in a sense:** “reversing” the extract_max)



- Add the new element as the last one
- **To fix MHP:**
  “Promote” the new element up the tree
  (“reversed” max_heapify)
- Done!

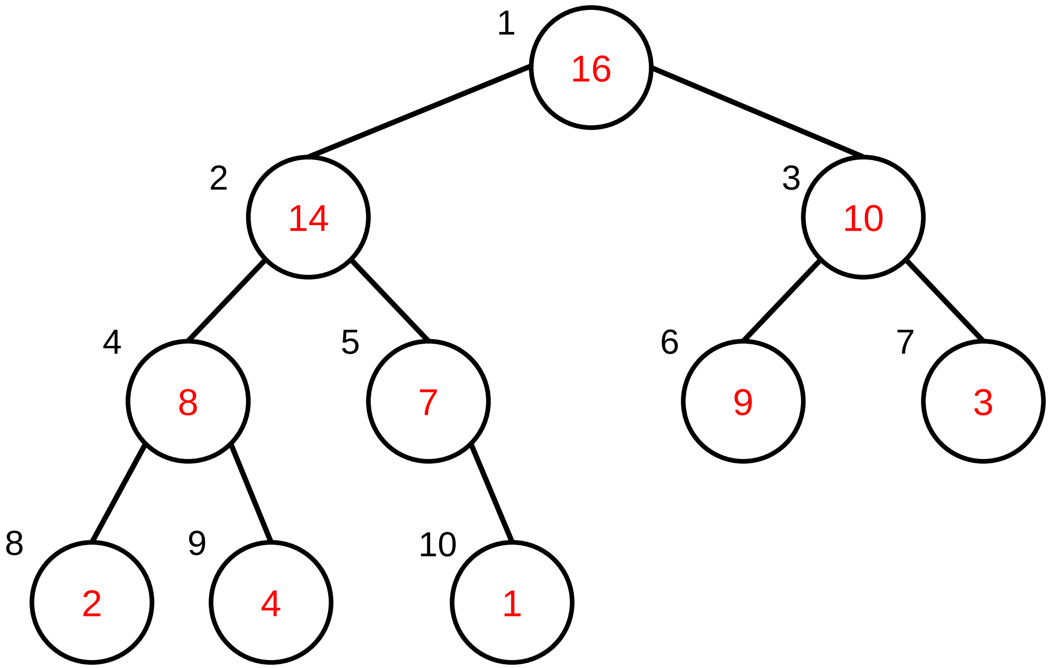
Run time?

➔ **Θ(1)** (addition) **+ O (log n)** (promotion up the tree)

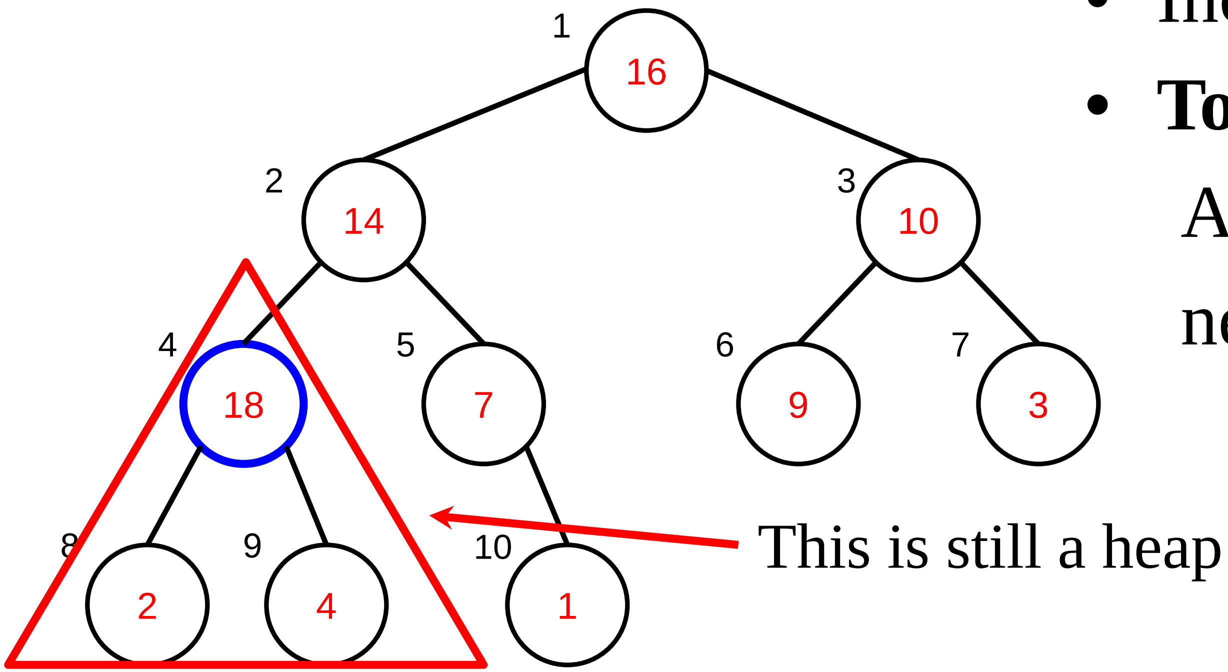
➔ total: **Θ(log n) (worst case)**

Implementing Increase_key

(Similar to Insert)



Implementing Increase_key
(Similar to Insert)

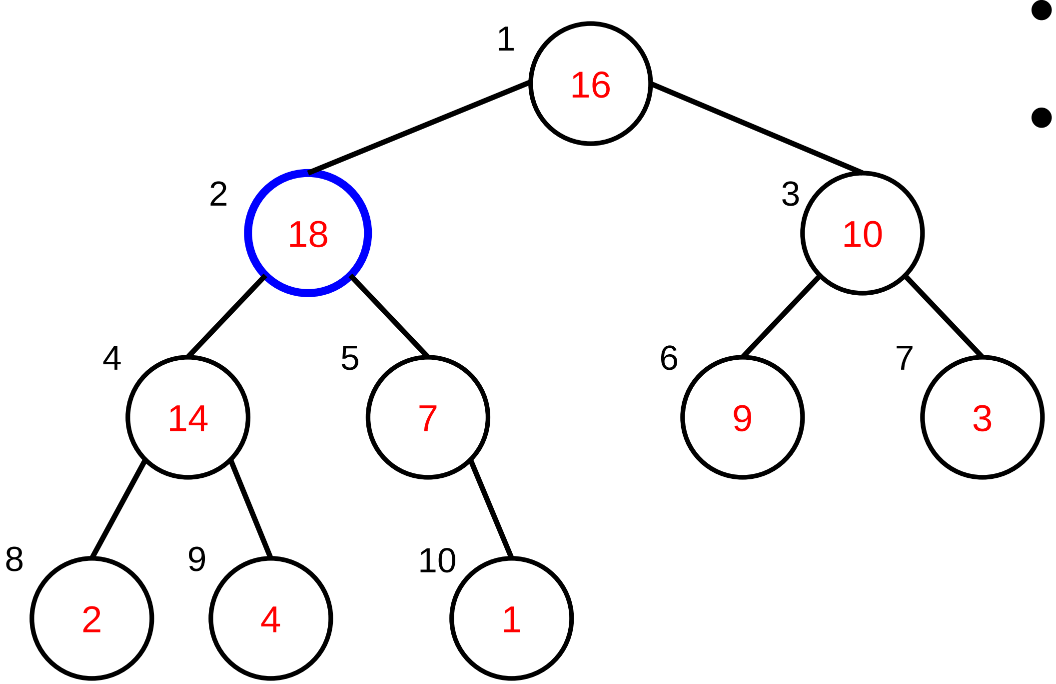


- Increase the key value
- **To fix MHP:**

  Again, "promote" the new element up the tree

Implementing Increase_key

(Similar to Insert)

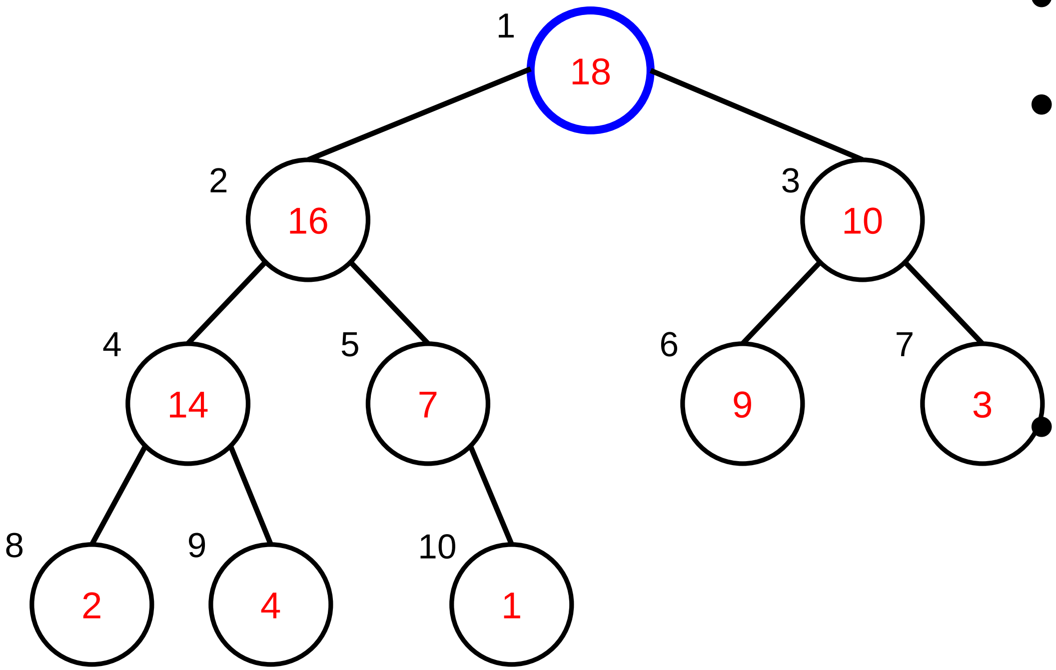


- Increase the key value
- **To fix MHP:**

  Again, "promote" the new element up the tree

Implementing Increase_key

(Similar to Insert)



- Increase the key value
- **To fix MHP:**
  Again, “promote” the new element up the tree
- Done!

Run time?

➔ **Θ(1)** (key value increase) **+ O (log n)** (promotion up the tree)

➔ total: **Θ(log n) (worst case)**

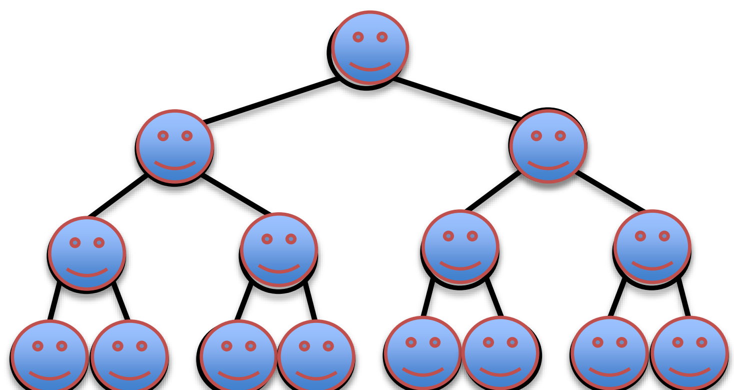
How to build a heap from a scratch?

Simple way:

➔ Start with an empty heap

➔ Insert all the **n** elements into it

➔ Total time: **Θ(n log n) (worst-case)**



Iterative (and in-place) way:

```
build_max_heap(A):
    for i=n downto 1
        do max_heapify(A,i)
```

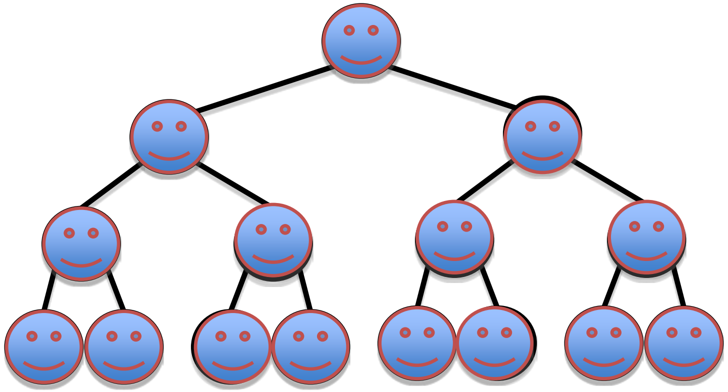
O(n log n)

➔Total time? **At first glance:** ~~**Θ(n log n)**~~ (cost of **n** max_heapify)

Actually: Θ(n)

Build_Max_Heap Analysis

Converts A[1…*n*] to a max heap

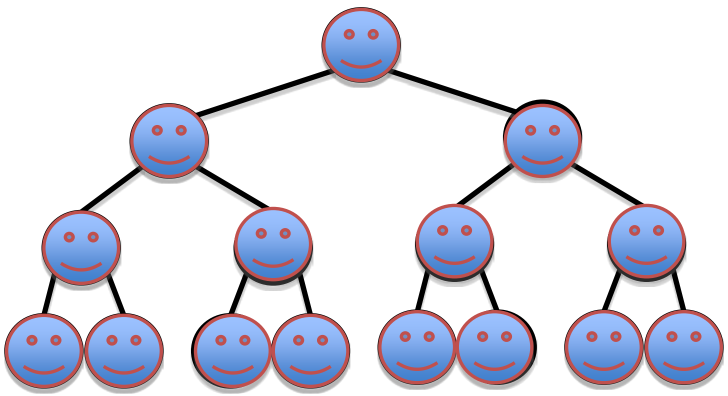


Build_Max_Heap(A):
for i=n/2 downto 1
do Max_Heapify(A, i)

O(*l*) time for nodes that are *l* levels above leaves.
We have n/4 nodes with level 1, n/8 with level 2,…

Build_Max_Heap Analysis

Converts A[1…n] to a max heap



Build_Max_Heap(A):
    for i=n/2 downto 1
        do Max_Heapify(A, i)

Total amount of work in the for loop can be summed as:

$n/4\ (1\ c) + n/8\ (2\ c) + n/16\ (3\ c) + \ldots + 1\ (\lg n\ c)$

Setting $n/4 = 2^k$ and simplifying we get:

$c\ 2^k(1/2^0 + 2/2^1 + 3/2^2 + \ldots (k+1)/2^k)$

The term is brackets is bounded by a constant!

This means that Build_Max_Heap is $O(n)$

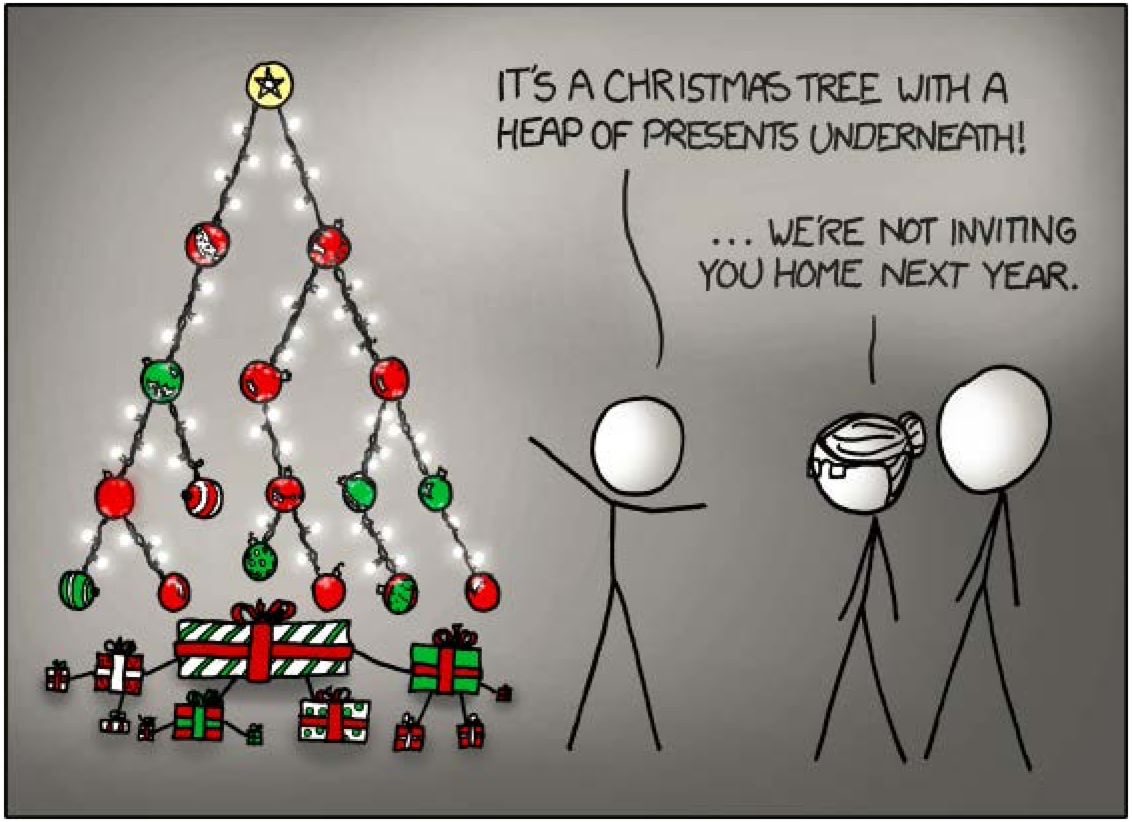
Cool application: Sorting

Heapsort: Sorting using a heap/priority queue
➔ Build a heap out of all elements
➔ Extract_max all elements one-by-one in an (inversely) sorted order!
➔ Total time: **Θ(n log n) (worst-case)**

This is a different algorithm than Merge sort!

In particular: Heapsort is actually an in-place algorithm (once we unravel the implementation of the heap)

More applications of heaps:



Source: xkcd.com