Priority Queue

Collection of priority-job pairs; priorities are comparable.

- insert(p, j)
- max(): read(-only) job of max priority
- extract-max(): read and remove job of max priority
- increase-priority(i, p'): increase priority of pair i

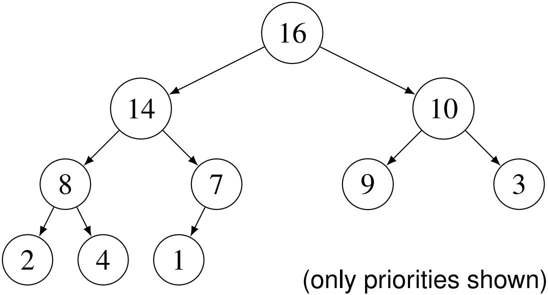
It's like:

- a hospital's emergency room
- an OS's ordering of things to do
- your ordering of things to study

Heap

A heap is one way to store a priority queue. A heap is:

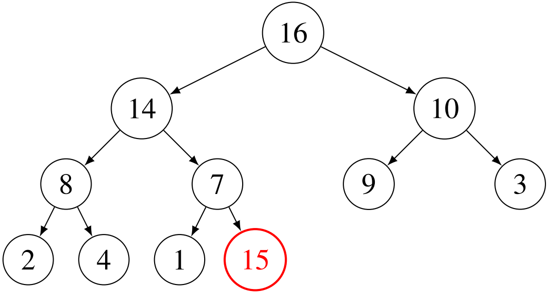
- a binary tree
- "nearly complete": every level i has 2^i nodes, except the bottom level; the bottom nodes flush to the left
- at each node: its priority $\geq$ both children's priorities



(only priorities shown)

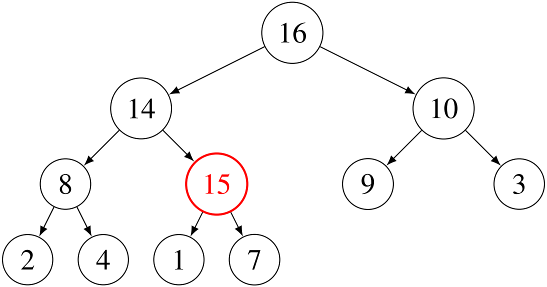
Heap insert: Example

Insert priority 15. At the bottom level, leftmost free space.



- √ The tree is still “nearly-complete”.
- ! Order of priorities bad. Fix: swap with parent.

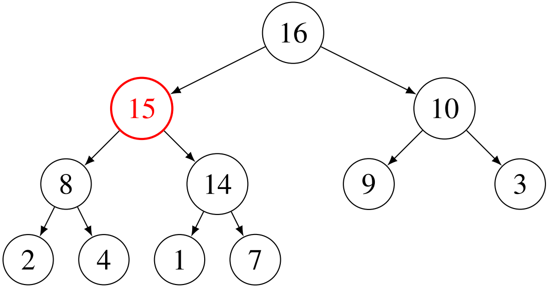
Heap insert: Example



√ The tree is still "nearly-complete".

! Order of priorities bad. Fix: swap with parent.

Heap insert: Example



- √ The tree is still “nearly-complete”.
- √ Order of priorities good.

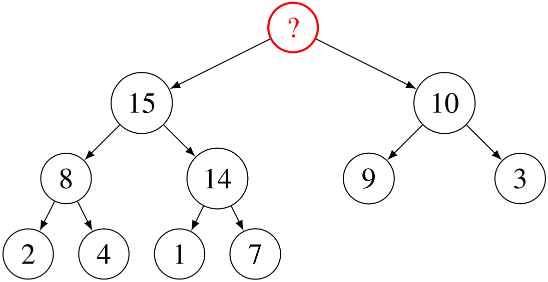
Heap insert: Summary

1. create new node at bottom level, leftmost free place
   (keep the tree "nearly-complete")
2. put priority (and job) in new node
3. v := that new node
4. "Float up as needed":
   while v has parent with smaller priority:
       swap them
       v := v.parent

Worst case time $\Theta(height)$.

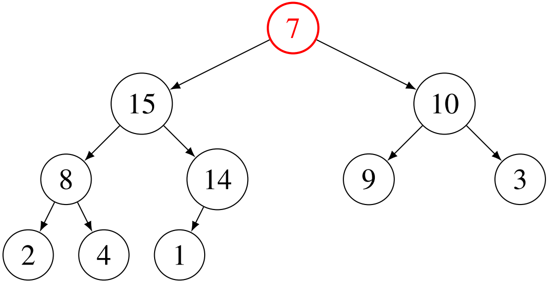
Later we will see why $height = \lfloor \lg n \rfloor$. Therefore worse case time $\Theta(\lg n)$.

Heap extract-max Example (aka Game of Thrones)



Someone has to ~~take the throne~~ replace the blank!

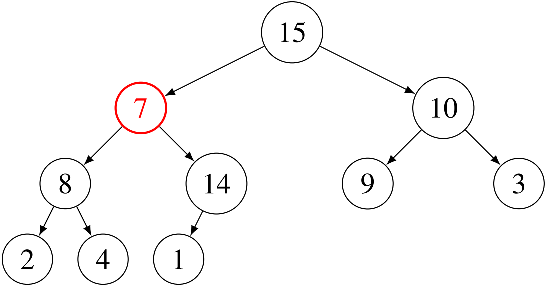
Heap extract-max Example (aka Game of Thrones)



Replace by the bottom level, rightmost item.

- √ The tree is still "nearly-complete".
- ! Order of priorities bad. Fix: swap with the larger child. (Why not the smaller child?)

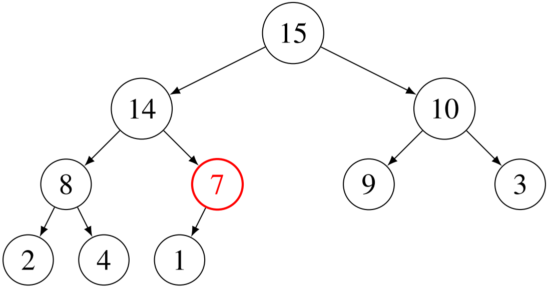
Heap extract-max Example (aka Game of Thrones)



Replace by the bottom level, rightmost item.

- √ The tree is still “nearly-complete”.
- ! Order of priorities bad. Fix: swap with the larger child. (Why not the smaller child?)

Heap extract-max Example (aka Game of Thrones)



Replace by the bottom level, rightmost item.

- √ The tree is still “nearly-complete”.
- √ Order of priorities good.

Heap extract-max: Summary

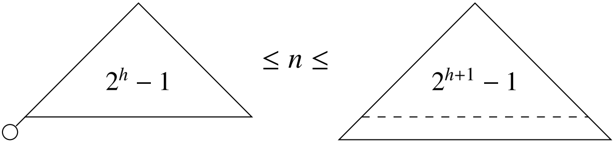
1. Replace root by bottom level, rightmost item
   (keep the tree "nearly-complete".)
2. v := root
3. "heapify at v":
   while v has larger child:
      swap with the largest child
      v := that child node

Worst case $\Theta(height)$ time.

Next we will see why $height = \lfloor \lg n \rfloor$. Therefore worse case time $\Theta(\lg n)$.

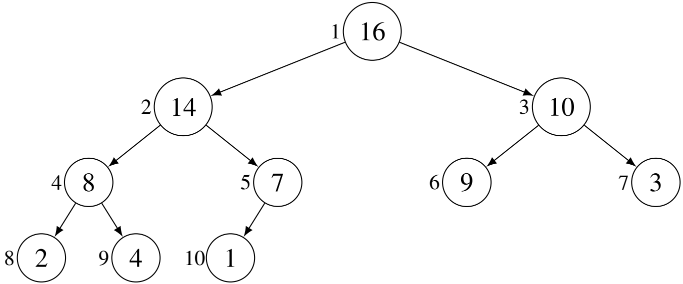
Heap: Height

Let n be the number of nodes, h be the height.



$$\begin{aligned}
(2^h - 1) + 1 \leq \quad & n && \leq 2^{h+1} - 1 \\
2^h \leq \quad & n && < 2^{h+1} \\
h \leq \quad & \lg n && < h + 1 \\
h = \quad & \lfloor \lg n \rfloor &&
\end{aligned}$$

Heap in Array/Vector



	16	14	10	8	7	9	3	2	4	1	
0	1	2	3	4	5	6	7	8	9	10	11

Heap in Array/Vector

	16	14	10	8	7	9	3	2	4	1	
0	1	2	3	4	5	6	7	8	9	10	11

Convenience:

- Where to insert/remove: simply at the end.
- Saves space. (No pointers to store.)

Formulas for:

- left child of index i: index $2 \times i$
- right child of index i: index $2 \times i + 1$
- parent of index i: index $\lfloor i/2 \rfloor$

Heapsort

Heapsort sorts an array via an intermediate max-heap.

Two stages:

1. "Build max-heap": Turn the array into max-heap form.

   Basic idea: heapify at nodes that have children, bottom-up order:

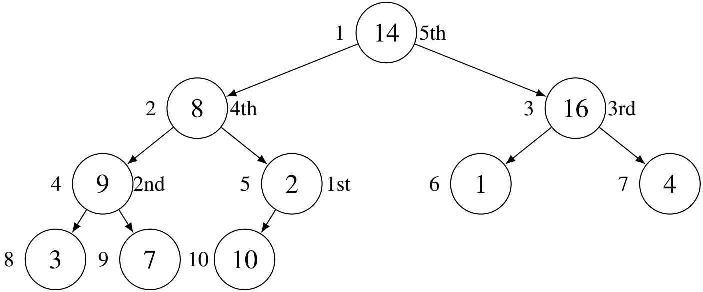
   for $v := \lfloor size/2 \rfloor$ down to 1:
       heapify at v

2. Repeatedly extract-max, put answer at the end.

   Basic idea: The array slot freed up by extract-max is exactly where you want the max to land at.

Turn Array Into Max-Heap

Below, "1st"… "5th" means order of getting heapified:

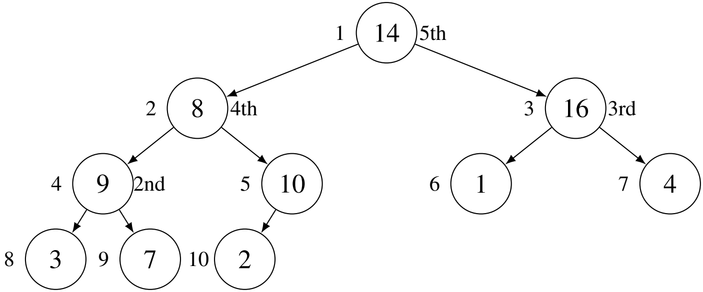


	5th	4th	3rd	2nd	1st						
	14	8	16	9	2	1	4	3	7	10	
0	1	2	3	4	5	6	7	8	9	10	11

for $v := \lfloor size/2 \rfloor$ down to 1: heapify at v.

Turn Array Into Max-Heap

Below, “1st”. . . “5th” means order of getting heapified:

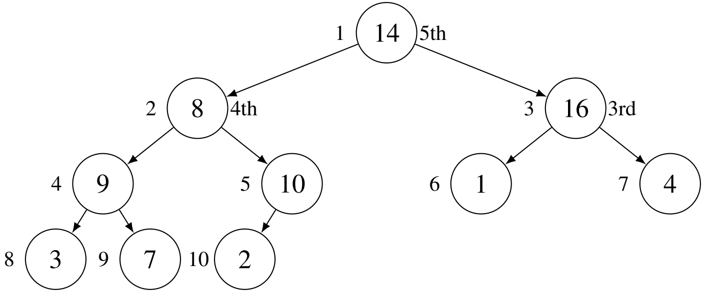


	5th	4th	3rd	2nd							
	14	8	16	9	10	1	4	3	7	2	
0	1	2	3	4	5	6	7	8	9	10	11

for $v := \lfloor size/2 \rfloor$ down to 1: heapify at v.

Turn Array Into Max-Heap

Below, “1st”. . . “5th” means order of getting heapified:

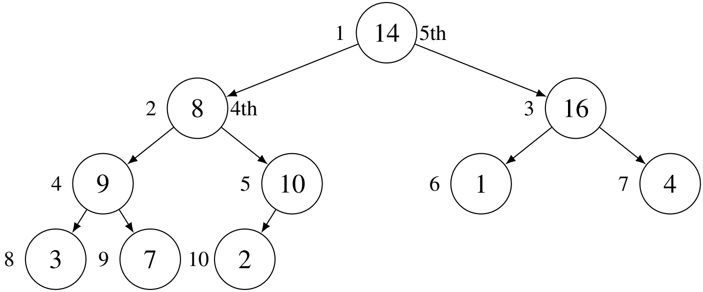


	5th	4th	3rd								
	14	8	16	9	10	1	4	3	7	2	
0	1	2	3	4	5	6	7	8	9	10	11

for $v := \lfloor size/2 \rfloor$ down to 1: heapify at v.

Turn Array Into Max-Heap

Below, "1st"... "5th" means order of getting heapified:

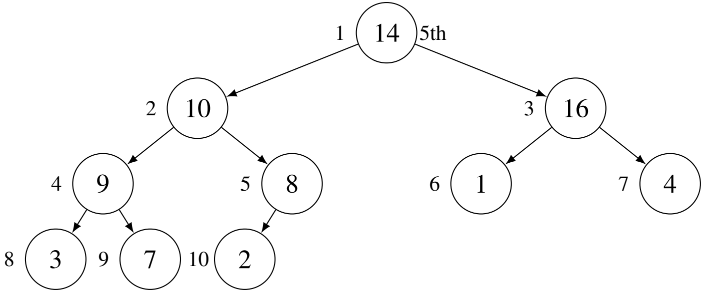


	5th	4th									
	14	8	16	9	10	1	4	3	7	2	
0	1	2	3	4	5	6	7	8	9	10	11

for $v := \lfloor size/2 \rfloor$ down to 1: heapify at v.

Turn Array Into Max-Heap

Below, "1st"... "5th" means order of getting heapified:



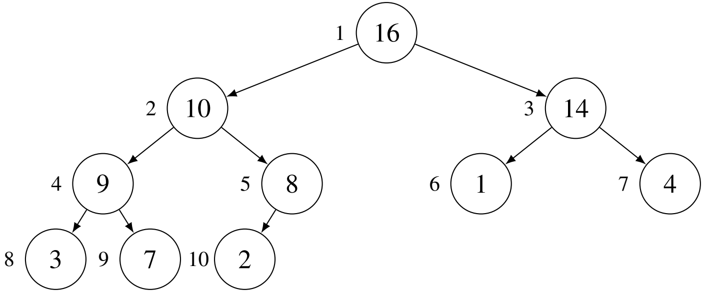
5th

	14	10	16	9	8	1	4	3	7	2	
0	1	2	3	4	5	6	7	8	9	10	11

for $v := \lfloor size/2 \rfloor$ down to 1: heapify at v.

Turn Array Into Max-Heap

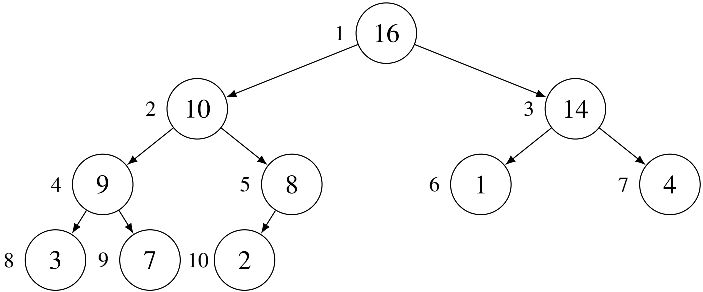
Below, "1st"…"5th" means order of getting heapified:



	5th										
	16	10	14	9	8	1	4	3	7	2	
0	1	2	3	4	5	6	7	8	9	10	11

for $v := \lfloor size/2 \rfloor$ down to 1: heapify at v.

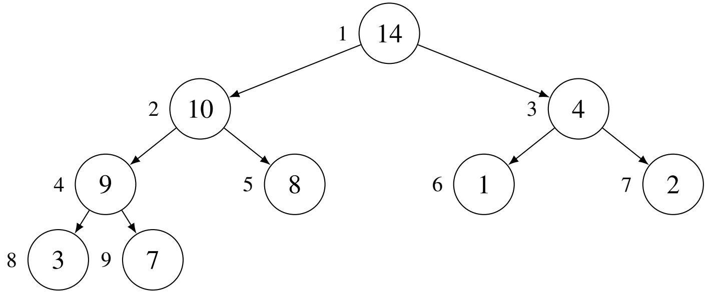
Repeatedly Extract-Max



	16	10	14	9	8	1	4	3	7	2	
0	1	2	3	4	5	6	7	8	9	10	11

for $i := size$ down to 1: $m :=$ extract-max(); $A[i] := m$

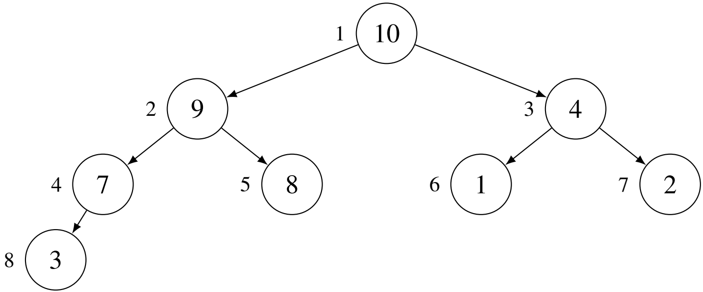
Repeatedly Extract-Max



0	1	2	3	4	5	6	7	8	9	10	11
	14	10	4	9	8	1	2	3	7	**16**	

for i := $size$ down to 1: m := extract-max(); $A[i]$:= m

Repeatedly Extract-Max



	10	9	4	7	8	1	2	3	**14**	**16**	
0	1	2	3	4	5	6	7	8	9	10	11

for $i := size$ down to 1: $m :=$ extract-max(); $A[i] := m$

Heapsort Time

1. Turn array into heap: A node at height h takes h iterations to fix; fewer than $n/2^h$ such nodes.

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} \times h \leq n \times \sum_{h=0}^{\infty} \frac{h}{2^h}$$

$$= n \times \text{constant} \qquad \text{(convergent series)}$$

   So $O(n)$ time. (Faster than n inserts.)

2. Repeatedly extract-max: $O(n \lg n)$ time.

Total $O(n \lg n)$ time.