CSCB63 Winter 2021

Week 4 Lecture 2 - Heaps, Priority Queues, Graphs

Anna Bretscher

February 4, 2021

Today

Priority Queues

Heaps

Heap Sort

Graph Definitions

Priority Queues

Collection of priority-job pairs; priorities are comparable.

- **insert(*p*, *j*)**: *insert* job *j* with priority *p*
- **max()**: *read*(-only) job of max priority
- **extract-max()**: *read* and *remove* job *j* of max priority
- **increase-priority(*j*, *p′*)**: *increase* priority of job *j* to *p′*

Priority Queues

Collection of priority-job pairs; priorities are comparable.

- **insert(*p*, *j*)**: *insert* job *j* with priority *p*
- **max()**: *read*(-only) job of max priority
- **extract-max()**: *read* and *remove* job *j* of max priority
- **increase-priority(*j*, *p′*)**: *increase* priority of job *j* to *p′*

Applications:

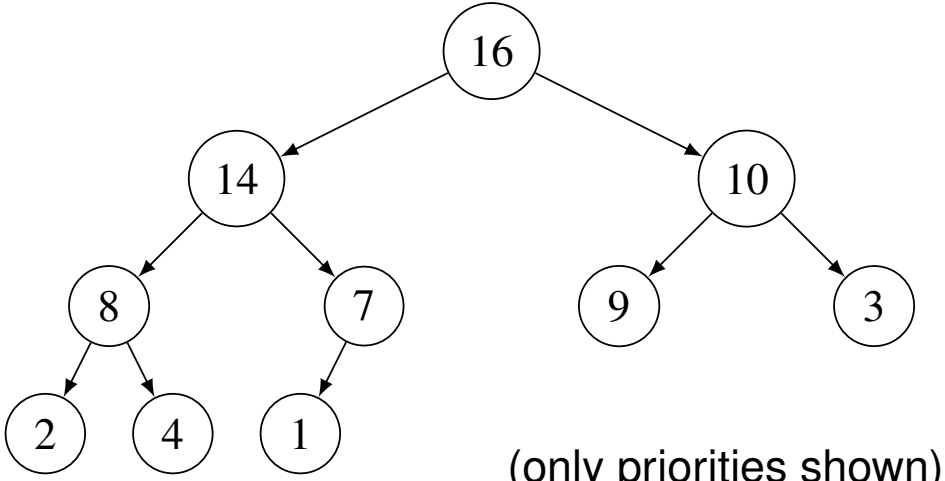
- Priority for getting covid vaccine
- Job scheduling in operating systems
- Printer queues
- Event-driven simulation algorithms
- Greedy algorithms
- Your ordering of things to study

Heap?

HEAP

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

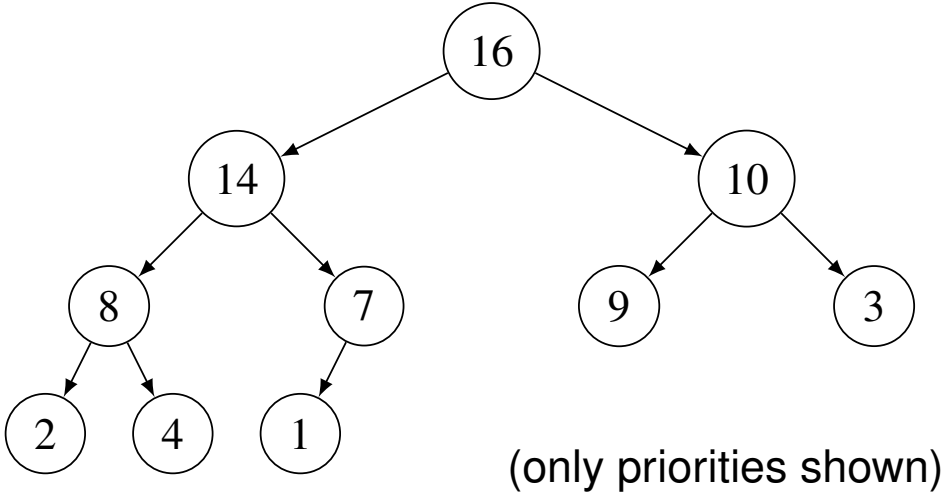
- a *binary tree*



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 fills in *left to right*

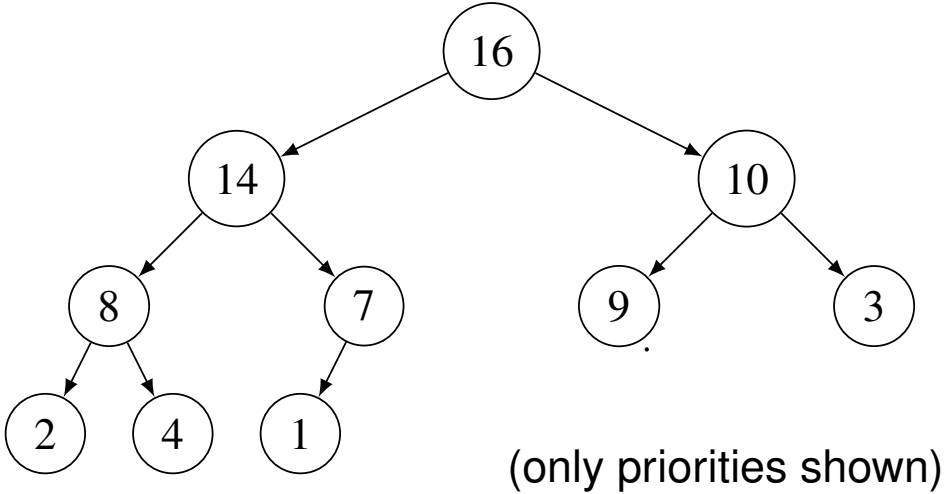


(only priorities shown)

HEAP

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

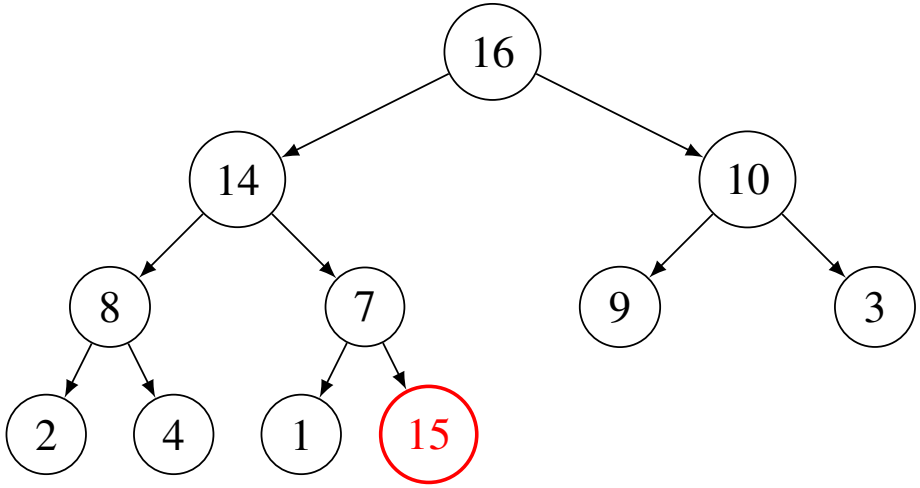
- a *binary tree*
- "*nearly complete*": every level i has 2^i nodes, except the bottom level; the bottom fills in *left to right*
- at each node: its *priority* is $\geq$ both *children's priorities*



(only priorities shown)

HEAP INSERT: EXAMPLE

Q. How do we insert a key with *priority 15*?

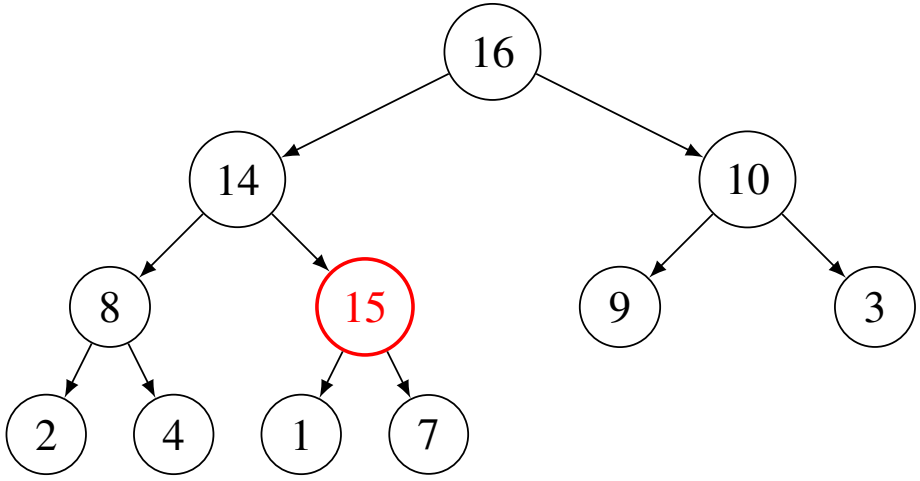


✓ The tree is still “nearly-complete”.

✗ Order of priorities bad. Fix?

HEAP INSERT: EXAMPLE

Q. How do we insert a key with *priority 15*?

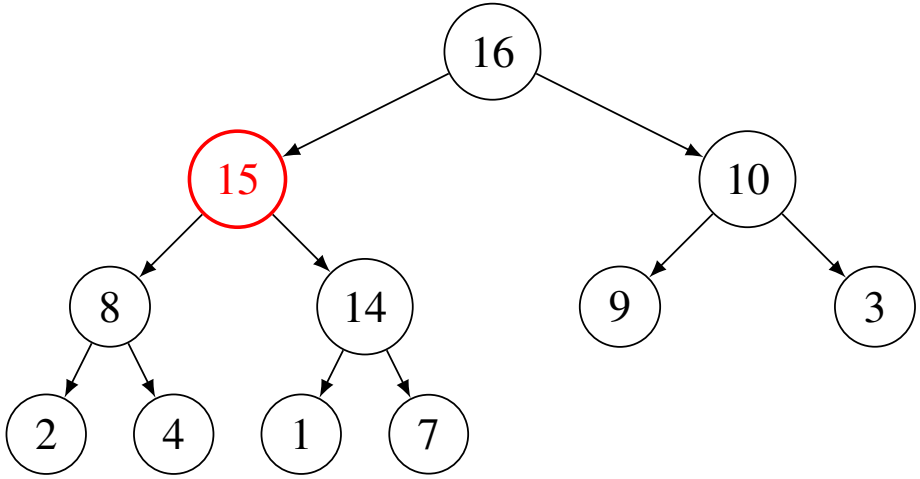


✓ The tree is still “nearly-complete”.

✗ Order of priorities bad. Fix? *swap* with parent.

HEAP INSERT: EXAMPLE

Q. How do we insert a key with *priority 15*?



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

HEAP INSERT: SUMMARY

1. Create new leaf at *bottom level*, *leftmost* open location $\Leftarrow$ maintains requirement that the tree is "*nearly-complete*"

HEAP INSERT: SUMMARY

1. Create new leaf at *bottom level*, *leftmost* open location $\Leftarrow$ maintains requirement that the tree is "*nearly-complete*"
2. Assign *priority* (and job) in the new node

HEAP INSERT: SUMMARY

1. Create new leaf at *bottom level*, *leftmost* open location $\Leftarrow$ maintains requirement that the tree is "*nearly-complete*"
2. Assign *priority* (and job) in the new node
3. v = the new node

HEAP INSERT: SUMMARY

1. Create new leaf at *bottom level*, *leftmost* open location $\Leftarrow$ maintains requirement that the tree is "*nearly-complete*"
2. Assign *priority* (and job) in the new node
3. v = the new node
4. **Percolate** v up:
   while v has *parent* with *smaller* priority:
   swap them
   $v = v$.parent

HEAP INSERT: SUMMARY

1. Create new leaf at *bottom level*, *leftmost* open location $\Leftarrow$ maintains requirement that the tree is "*nearly-complete*"
2. Assign *priority* (and job) in the new node
3. v = the new node
4. **Percolate** v up:
   while v has *parent* with *smaller* priority:
   swap them
   v = v.parent

Percolate?

HEAP INSERT: SUMMARY

1. Create new leaf at *bottom level*, *leftmost* open location $\Leftarrow$ maintains requirement that the tree is "*nearly-complete*"
2. Assign *priority* (and job) in the new node
3. v = the new node
4. **Percolate** v up:
   while v has *parent* with *smaller* priority:
   swap them
   $v = v$.parent

HEAP INSERT: SUMMARY

1. Create new leaf at *bottom level*, *leftmost* open location $\Leftarrow$ maintains requirement that the tree is "*nearly-complete*"
2. Assign *priority* (and job) in the new node
3. v = the new node
4. **Percolate** v up:
   while v has *parent* with *smaller* priority:
   swap them
   v = v.parent

Complexity?

HEAP INSERT: SUMMARY

1. Create new leaf at *bottom level*, *leftmost* open location $\Leftarrow$ maintains requirement that the tree is "*nearly-complete*"
2. Assign *priority* (and job) in the new node
3. v = the new node
4. **Percolate** v up:
   while v has *parent* with *smaller* priority:
   swap them
   $v = v$.parent

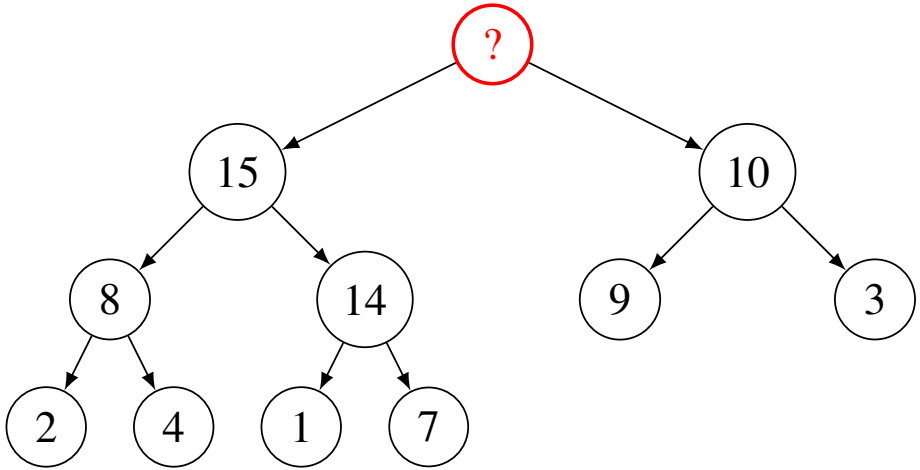
Complexity?

Worst case time $\Theta(\mathit{height})$.

Later we will see why $\mathit{height} = \lfloor \lg n \rfloor + 1$.

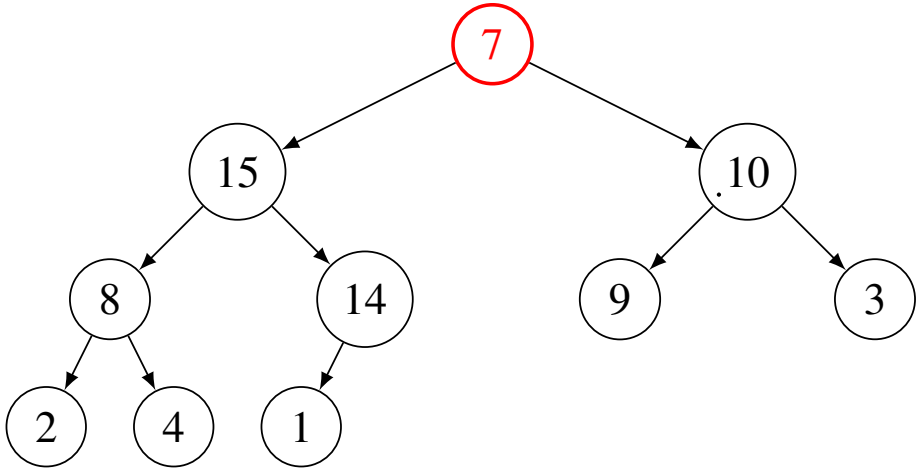
Therefore worst case time $\Theta(\lg n)$.

HEAP EXTRACT-MAX EXAMPLE



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

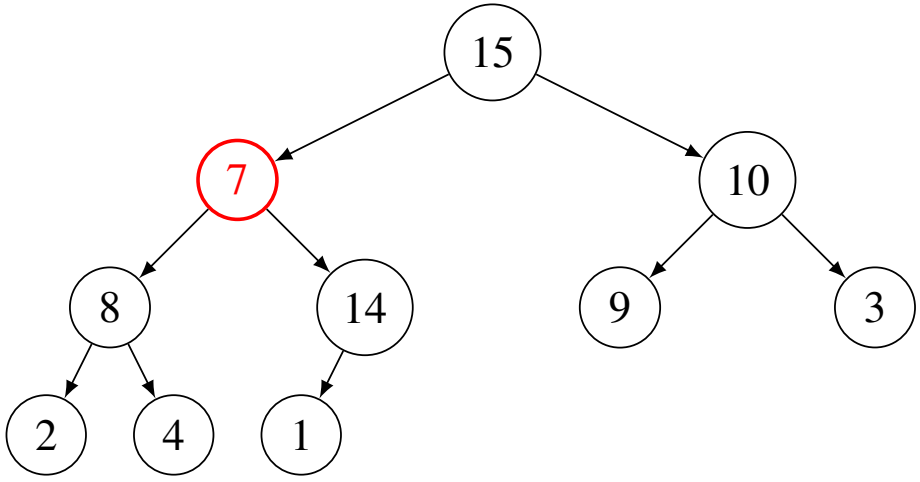
HEAP EXTRACT-MAX EXAMPLE



Replace by the *bottom level*, *rightmost* item. Then **heapify**!

- ✓ The tree is still "*nearly-complete*".
- ✗ Order of priorities bad. Fix?

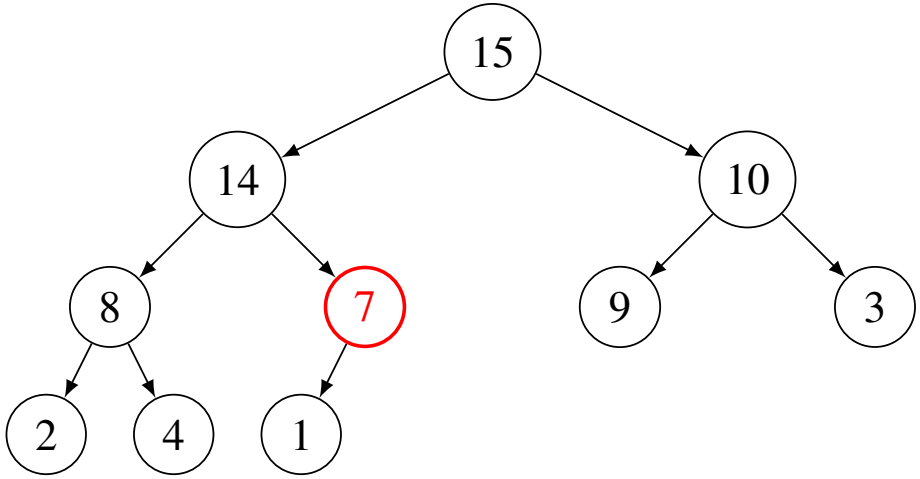
HEAP EXTRACT-MAX EXAMPLE



Replace by the *bottom level*, *rightmost* item. Then **heapify**!

- ✓ 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



Replace by the *bottom level*, *rightmost* item. Then **heapify**!

- ✓ The tree is still "*nearly-complete*".
- ✓ Order of priorities good.

HEAP EXTRACT-MAX: SUMMARY

1. *Replace* root by *bottom level*, *rightmost item* $\Leftarrow$ keeps the tree "nearly-complete".
2. v := *root*
3. **Heapify**:
   while v has *larger* child:
   swap with the *largest* child
   v := that *child* node

Worst case $\Theta(\textit{height})$ time.

Next we will see why $\textit{height} = \lfloor \lg n \rfloor + 1$.

Therefore worst case time $\Theta(\lg n)$.

The steps in line 3 are an algorithm we will use again called **Heapify**.

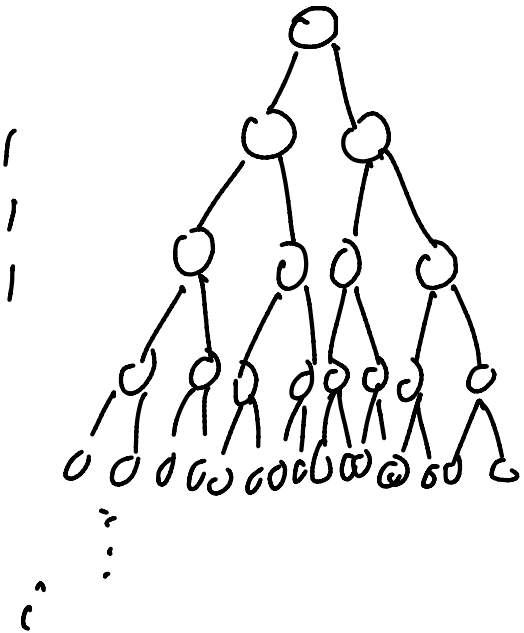
Binary Tree Size

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

Q. How many *nodes* does a *full* binary tree of *height h* have?

A.

h	n
1	1
2	$3 = 2^2 - 1$
3	$7 = 2^3 - 1$
4	$15 = 2^4 - 1$
5	31
i	$2^i - 1$



Binary Tree Size

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

Q. How many *nodes* does a *full* binary tree of *height h* have?

A.

h	*n*
1	1
2	3
3	
4	
5	
i	

Binary Tree Size

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

Q. How many *nodes* does a *full* binary tree of *height h* have?

A.

h	*n*
1	1
2	3
3	7
4	
5	
i	

Binary Tree Size

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

Q. How many *nodes* does a *full* binary tree of *height h* have?

A.

h	*n*
1	1
2	3
3	7
4	15
5	
i	

Binary Tree Size

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

Q. How many *nodes* does a *full* binary tree of *height h* have?

A.

h	*n*
1	1
2	3
3	7
4	15
5	31
i	

Binary Tree Size

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

Q. How many *nodes* does a *full* binary tree of *height h* have?

A.

h	*n*
1	1
2	3
3	7
4	15
5	31
i	2^i -1

Binary Tree Size

Q. How many *nodes* does a *full* binary tree of *height h* have?

A.

h	n
1	1
2	3
3	7
4	15
5	31
i	2^i -1

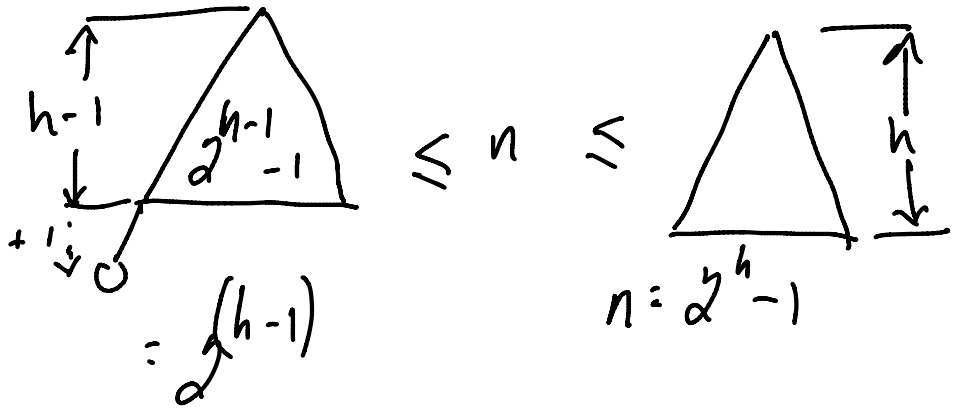
So a binary tree of *height h* has *at most* $2^h - 1$ nodes.

HEAP: HEIGHT

We will bound n to find h. A tree of height h can have a *range* of values for n.

Q. What is this *range*?

A.

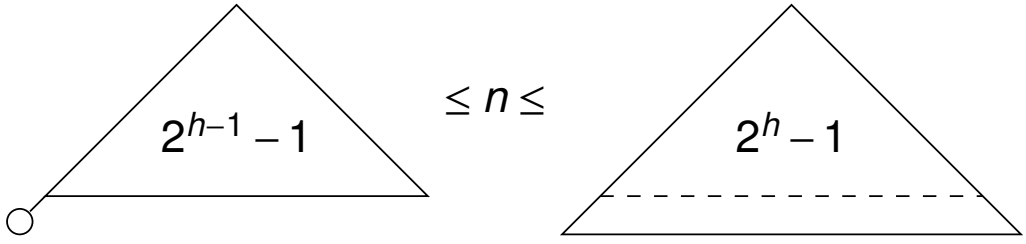


HEAP: HEIGHT

We will bound n to find h. A tree of height h can have a *range* of values for n.

Q. What is this *range*?

A.



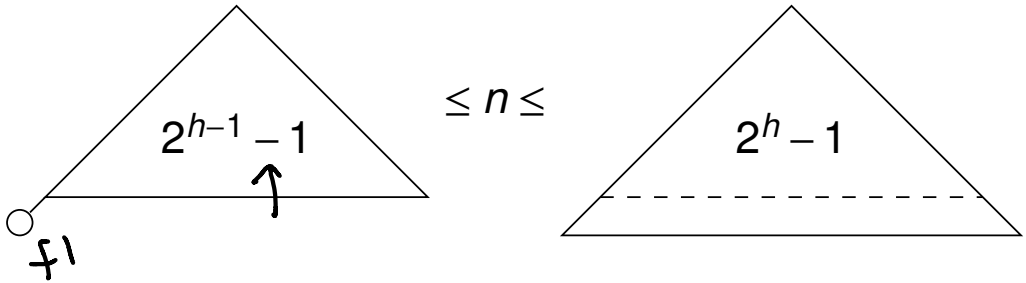
Left tree has $2^{h-1}-1+1$ keys.

HEAP: HEIGHT

We will bound n to find h. A tree of height h can have a *range* of values for n.

Q. What is this *range*?

A.



Left tree has $2^{h-1}-1+1$ keys.

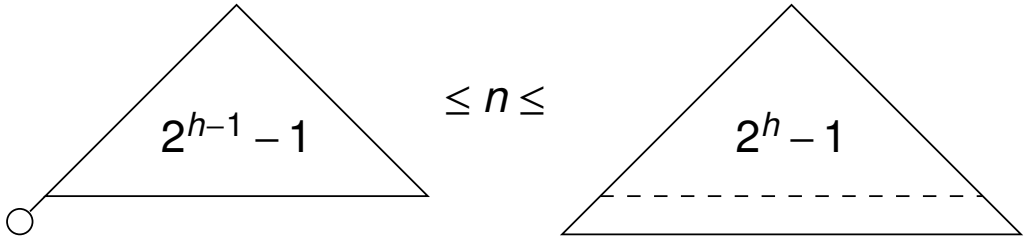
$$(2^{h-1}-1)+1 \leq n \leq 2^h-1$$

HEAP: HEIGHT

We will bound n to find h. A tree of height h can have a *range* of values for n.

Q. What is this *range*?

A.



Left tree has $2^{h-1}-1+1$ keys.

$$(2^{h-1}-1)+1 \leq n \leq 2^h-1$$

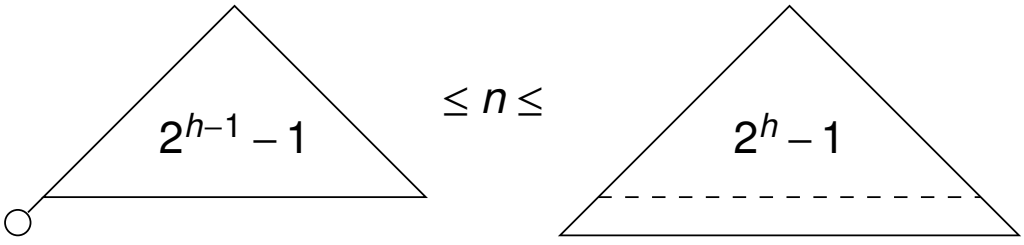
$$2^{h-1} \leq n < 2^h$$

HEAP: HEIGHT

We will bound n to find h. A tree of height h can have a *range* of values for n.

Q. What is this *range*?

A.



Left tree has $2^{h-1}-1+1$ keys.

$$(2^{h-1}-1)+1 \leq n \leq 2^h-1$$
$$2^{h-1} \leq n < 2^h$$
$$h-1 \leq \lg n < h$$

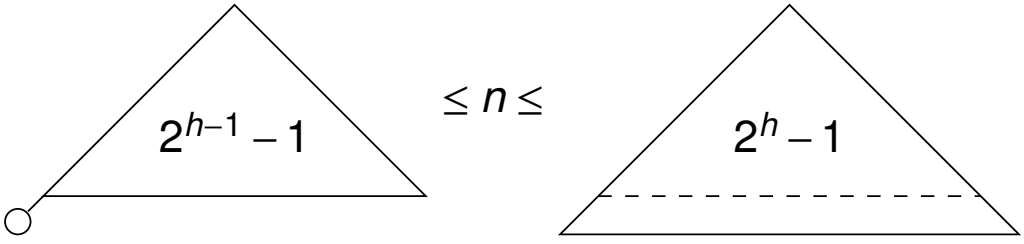
$h =$

HEAP: HEIGHT

We will bound n to find h. A tree of height h can have a *range* of values for n.

Q. What is this *range*?

A.



Left tree has $2^{h-1}-1+1$ keys.

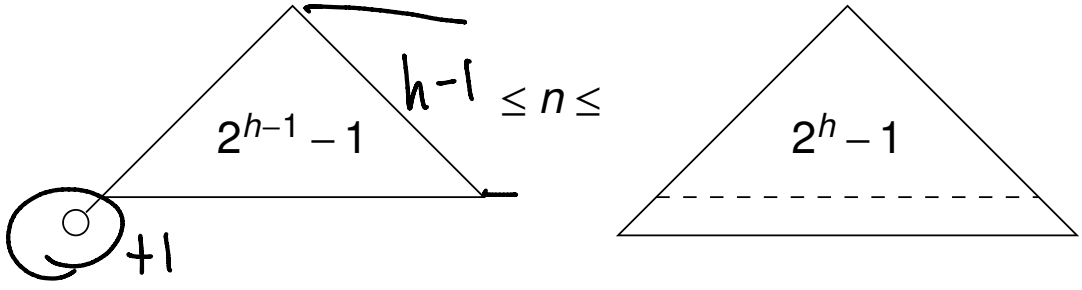
$$
\begin{aligned}
(2^{h-1}-1)+1 \leq n &\leq 2^h-1 \\
2^{h-1} \leq n &< 2^h \\
h-1 \leq \lg n &< h \\
h \leq (\lg n)+1 &< h+1
\end{aligned}
$$

HEAP: HEIGHT

We will bound n to find h. A tree of height h can have a *range* of values for n.

Q. What is this *range*?

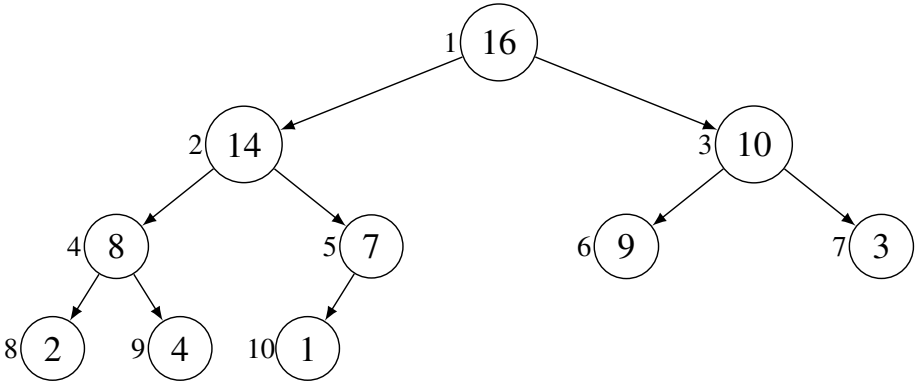
A.



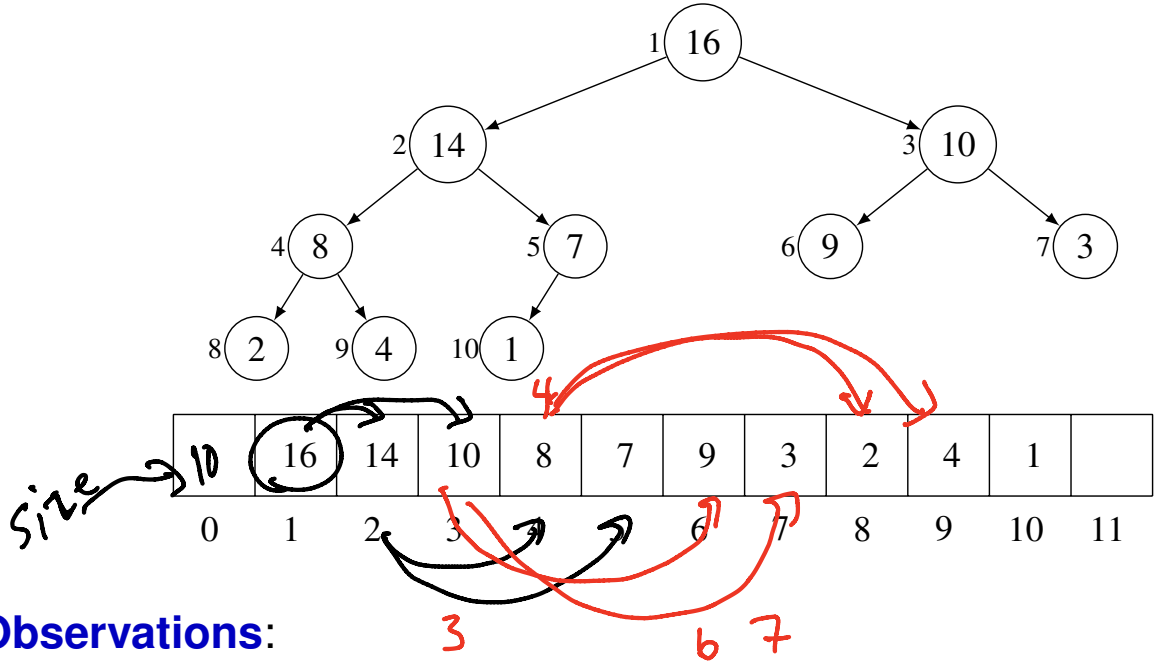
Left tree has $2^{h-1}-1+1$ keys.

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

Implementing a Heap

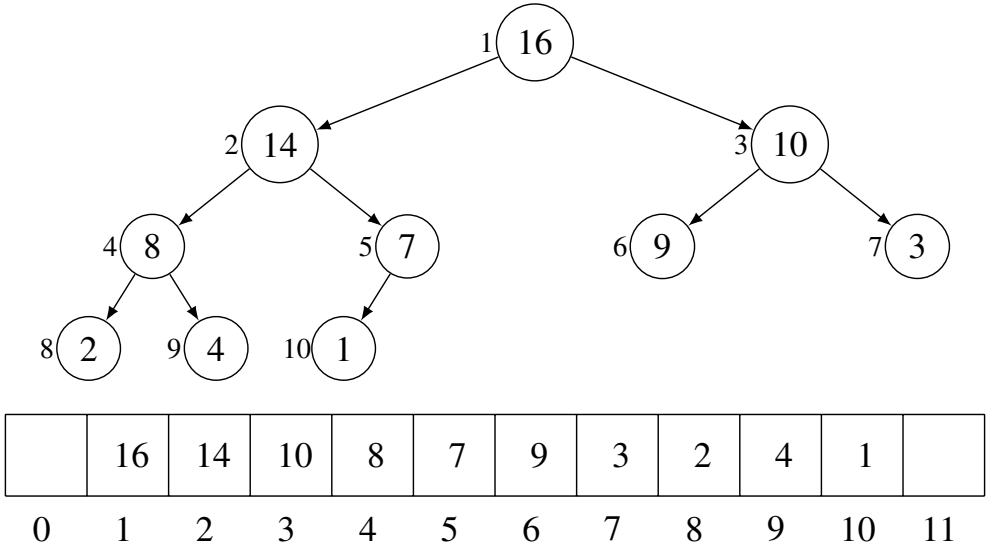


Implementing a Heap



Observations:

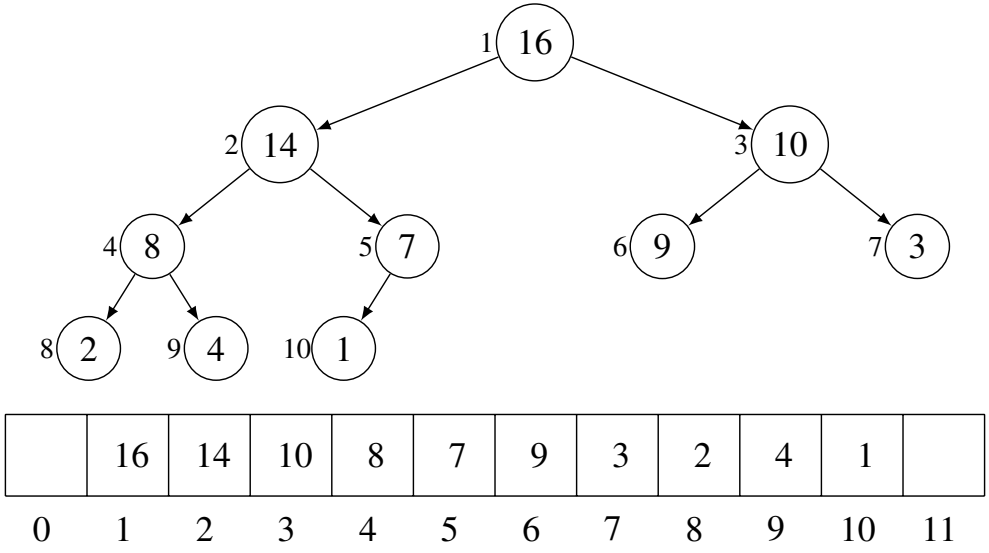
Implementing a Heap



Observations:

- *Start* index is 1. May need to store *heap size*.

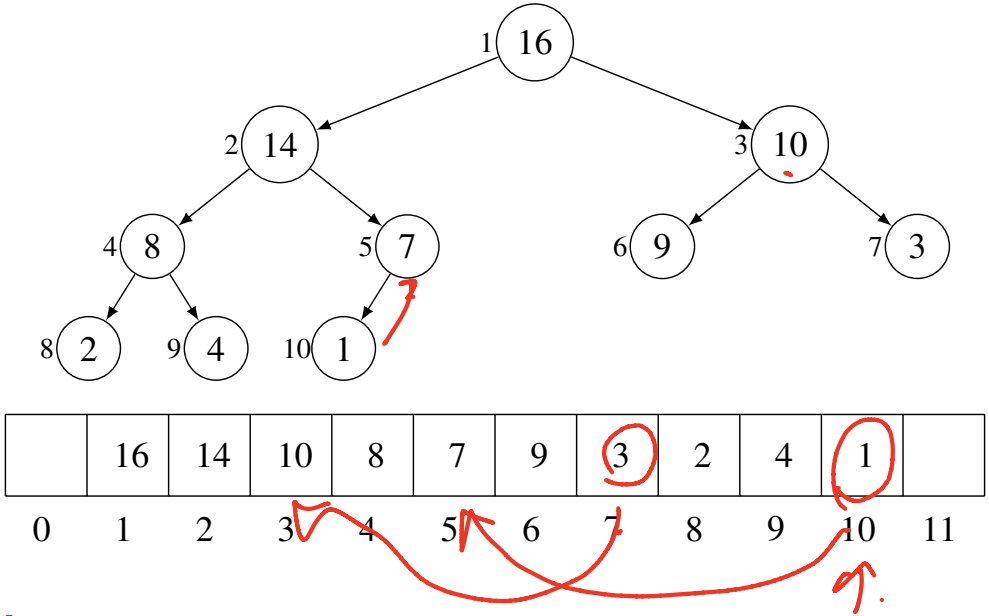
Implementing a Heap



Observations:

- *Start* index is 1. May need to store *heap size*.
- *Left child* of node *i* is at *2i*.

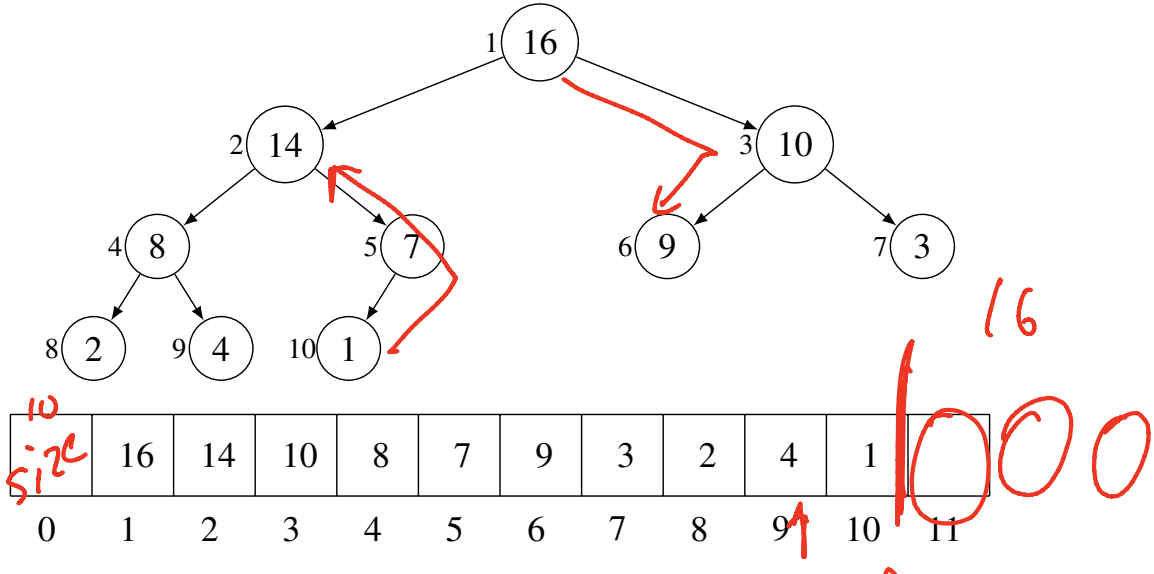
Implementing a Heap



Observations:

- *Start* index is 1. May need to store *heap size*.
- *Left child* of node i is at $2i$.
- *Right child* of node i is at $2i+1$.

Implementing a Heap



Observations:

- *Start* index is 1. May need to store *heap size*.
- *Left child* of node i is at $2i$.
- *Right child* of node i is at $2i+1$.
- *Parent* of node i is at $\lfloor \frac{i}{2} \rfloor$.

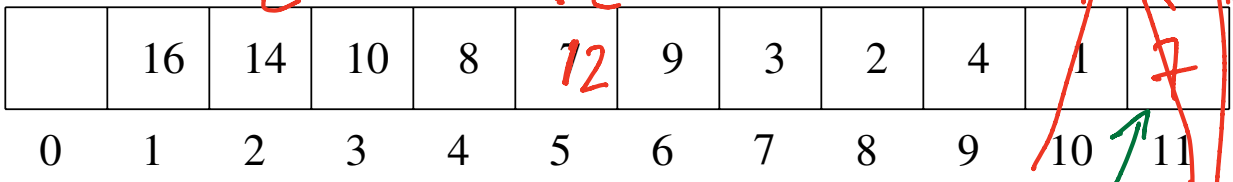
HEAP IN ARRAY/VECTOR

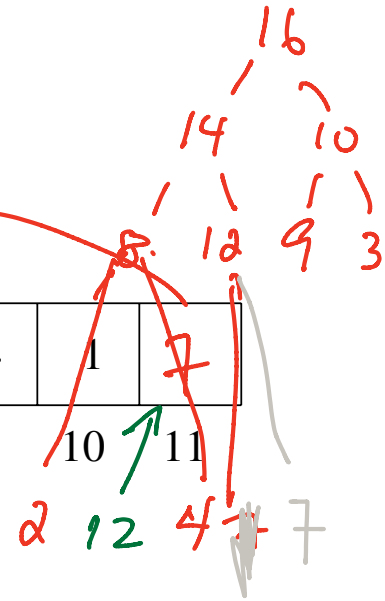
	16	14	10	8	7	9	3	2	4	1	X
0	1	2	3	4	5	6	7	8	9	10	11

Q. How do we *insert* into a heap with an *array*?

A.

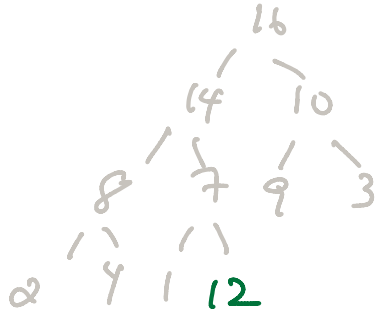
Heap in Array/Vector





Q. How do we *insert* into a heap with an *array*?

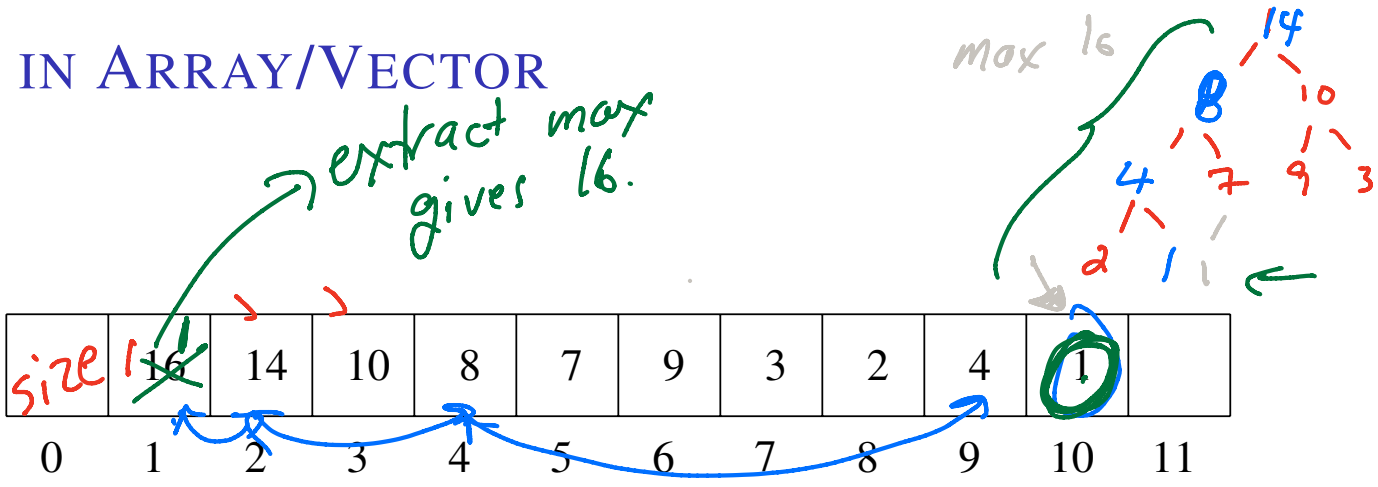
A. Simply **append** new item to the *end of the array* and update *heap size* if using. **Percolate** new node up.



Q. How do we *extract max*?

A.

Heap in Array/Vector

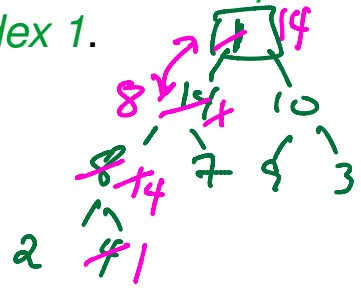


Q. How do we *insert* into a heap with an *array*?

A. Simply **append** new item to the *end of the array* and update *heap size* if using. **Percolate** new node up.

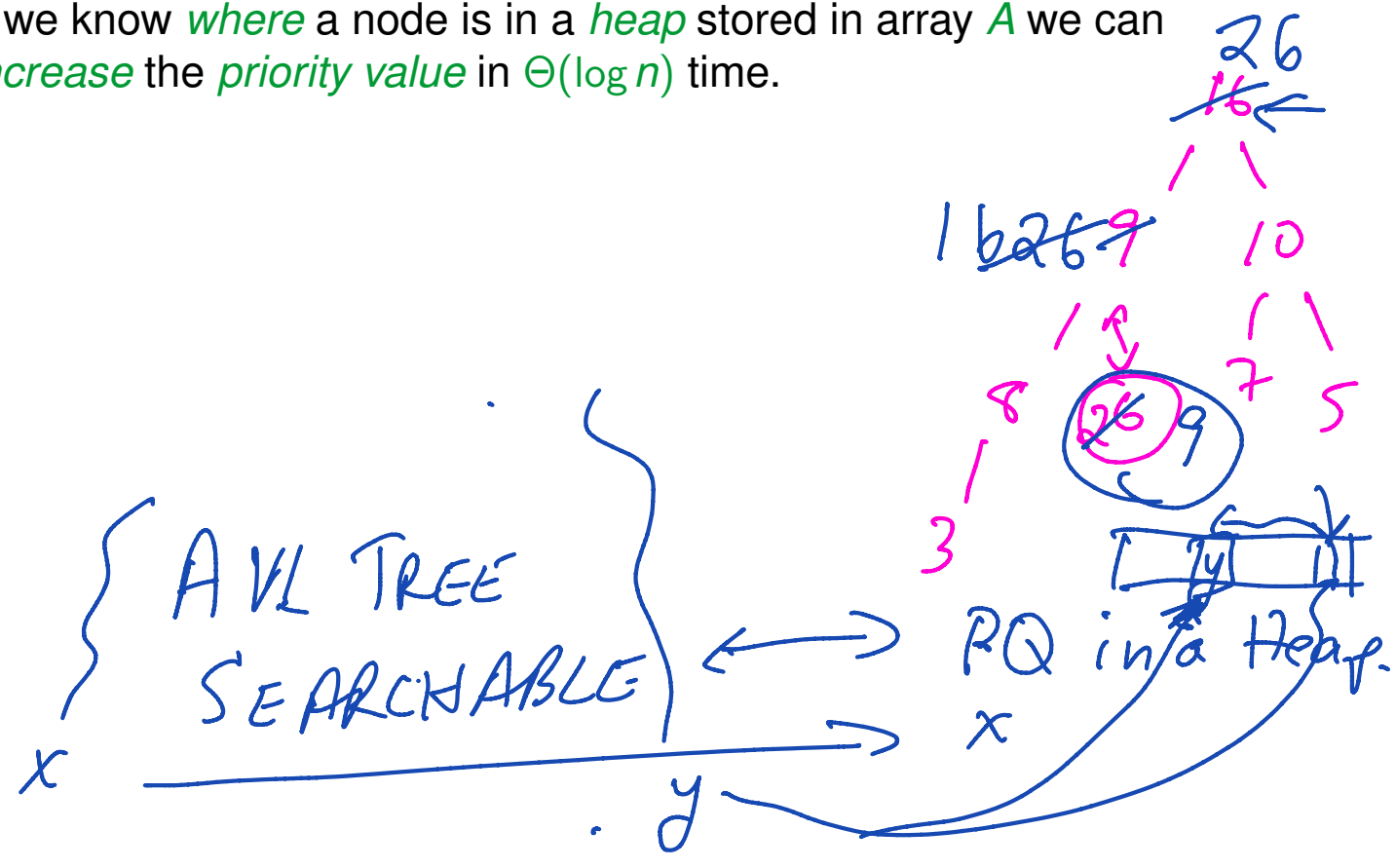
Q. How do we *extract max*?

A. **Read** and **replace** the item at *index 1* with the item at index *heap size* and decrement *heap size*. **Heapify** starting at *index 1*.



Increase a Priority

If we know *where* a node is in a *heap* stored in array A we can *increase* the *priority value* in $\Theta(\log n)$ time.



Increase a Priority

If we know *where* a node is in a *heap* stored in array A we can *increase* the *priority value* in $\Theta(\log n)$ time.

We set the new p value and then *percolate* the node up.

Increase a Priority

If we know *where* a node is in a *heap* stored in array *A* we can *increase* the *priority value* in $\Theta(\log n)$ time.

We set the new *p* value and then *percolate* the node up.

```
increase_priority(A, index, p):
   # array A, int index, priority p
   if p > A[index]:
      # priority is not better so do nothing
      return
   A[index] = p
   while A[⌊index/2⌋] < A[index]:
      swap them
      index = ⌊index/2⌋
```

BUILDING HEAPS

Given an array *A* of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view *A* as a *complete binary tree*...

...but not necessarily a *heap*...

Q. How can we turn *A* into a *heap*?

BUILDING HEAPS

Given an array *A* of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view *A* as a *complete binary tree*...

...but not necessarily a *heap*...

Q. How can we turn *A* into a *heap*?

1. *Sort A* from highest priority element to lowest. **Complexity**?

BUILDING HEAPS

Given an array A of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view A as a *complete binary tree*...

...but not necessarily a *heap*...

Q. How can we turn A into a *heap*?

1. *Sort* A from highest priority element to lowest. **Complexity**? $\Theta(n \log n)$.

BUILDING HEAPS

Given an array *A* of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view *A* as a *complete binary tree*...

...but not necessarily a *heap*...

Q. How can we turn *A* into a *heap*?

1. *Sort A* from highest priority element to lowest. **Complexity**? $\Theta(n \log n)$.
2. *Create* a new array *B* that represents a *heap* and go through every element of *A* and `insert` it into *B*.

   Complexity?

BUILDING HEAPS

Given an array A of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view A as a *complete binary tree*...

...but not necessarily a *heap*...

Q. How can we turn A into a *heap*?

1. *Sort* A from highest priority element to lowest. **Complexity**? $\Theta(n \log n)$.
2. *Create* a new array B that represents a *heap* and go through every element of A and `insert` it into B.

   Complexity? $\Theta(n \log n)$.

BUILDING HEAPS

Given an array A of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view A as a *complete binary tree*...

...but not necessarily a *heap*...

Q. How can we turn A into a *heap*?

1. *Sort* A from highest priority element to lowest. **Complexity**? $\Theta(n \log n)$.
2. *Create* a new array B that represents a *heap* and go through every element of A and `insert` it into B.

   Complexity? $\Theta(n \log n)$.
3. Use **`heapify`**:

BUILDING HEAPS

Given an array A of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view A as a *complete binary tree*...

...but not necessarily a *heap*...

Q. How can we turn A into a *heap*?

1. *Sort* A from highest priority element to lowest. **Complexity**? $\Theta(n \log n)$.
2. *Create* a new array B that represents a *heap* and go through every element of A and `insert` it into B.

   Complexity? $\Theta(n \log n)$.
3. Use **`heapify`**:
   - Use the fact that each subtree of a heap is a heap.

BUILDING HEAPS

Given an array A of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view A as a *complete binary tree*...

...but not necessarily a *heap*...

Q. How can we turn A into a *heap*?

1. *Sort* A from highest priority element to lowest. **Complexity**? $\Theta(n \log n)$.
2. *Create* a new array B that represents a *heap* and go through every element of A and `insert` it into B.

   Complexity? $\Theta(n \log n)$.
3. Use **`heapify`**:
   - Use the fact that each subtree of a heap is a heap.
   - Start with the *smallest subtrees*, turn into heaps using *heapify*.

BUILDING HEAPS

Given an array A of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view A as a *complete binary tree*...

...but not necessarily a *heap*...

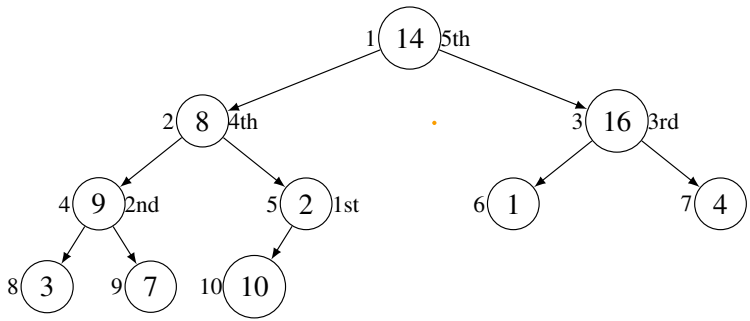
Q. How can we turn A into a *heap*?

1. *Sort* A from highest priority element to lowest. **Complexity**? $\Theta(n \log n)$.
2. *Create* a new array B that represents a *heap* and go through every element of A and `insert` it into B.

   Complexity? $\Theta(n \log n)$.
3. Use **`heapify`**:
   - Use the fact that each subtree of a heap is a heap.
   - Start with the *smallest subtrees*, turn into heaps using *heapify*.
   - Work *up* the tree.

HEAPIFY

```
heapify(A,i, size):   # array A, start index i, int size = heapsize
   max = i
   if (2i<=size and A[2i]>A[i]):
      max = 2i
   if (2i+1 ≤ size and A[2i+1]>A[max]):
      max = 2i+1
   if ( max != i ):
      swap A[i] and A[max]
      heapify(A,max,size)
```

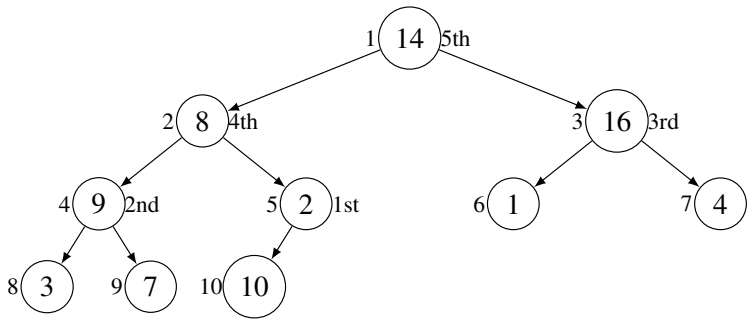


Q. On which *nodes* should we call *heapify*?

A.

HEAPIFY

```
heapify(A,i, size):   # array A, start index i, int size = heapsize
   max = i
   if (2i<=size and A[2i]>A[i]):
      max = 2i
   if (2i+1 ≤ size and A[2i+1]>A[max]):
      max = 2i+1
   if ( max != i ):
      swap A[i] and A[max]
      heapify(A,max,size)
```



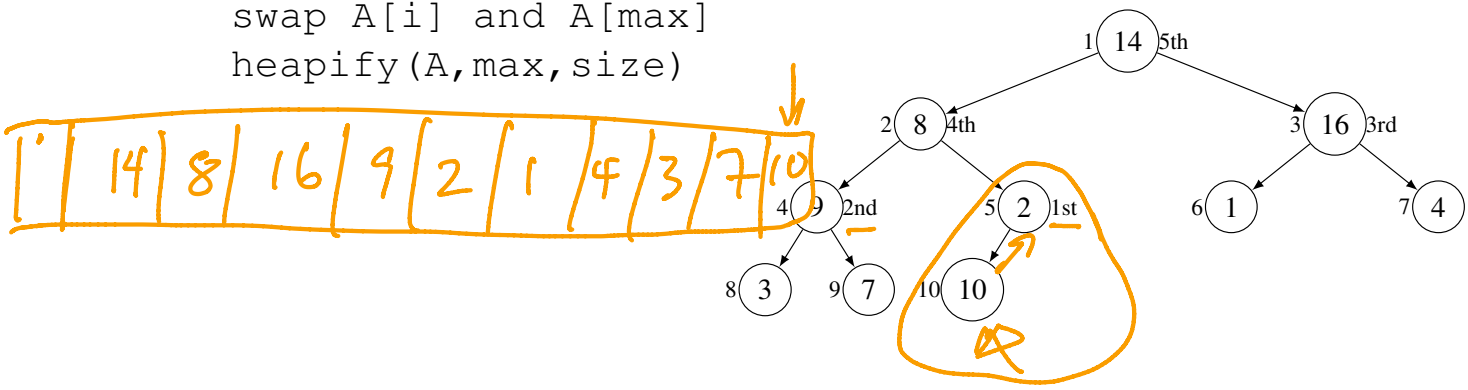
Q. On which *nodes* should we call *heapify*?

A. start at first node that is the *root* of a tree of height at least *2*. How do we calculate this node?

A.

HEAPIFY

```
heapify(A,i, size):    # array A, start index i, int size = heapsize
   max = i
   if (2i<=size and A[2i]>A[i]):
      max = 2i
   if (2i+1 ≤ size and A[2i+1]>A[max]):
      max = 2i+1
   if ( max != i ):
      swap A[i] and A[max]
      heapify(A,max,size)
```

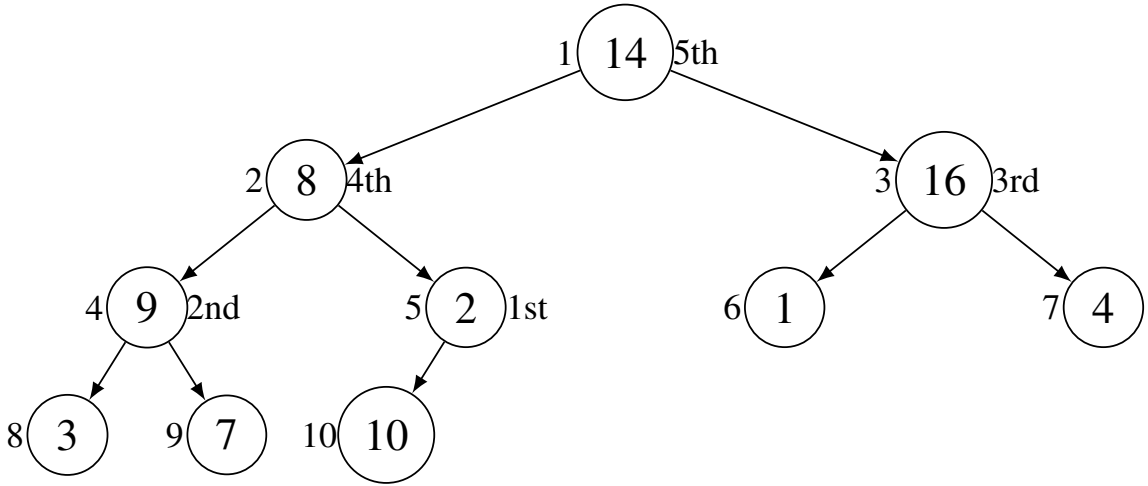


Q. On which *nodes* should we call *heapify*?

A. start at first node that is the *root* of a tree of height at least *2*. How do we calculate this node?

A. take the *last child's parent*, so $\lfloor \frac{heapsize}{2} \rfloor$.

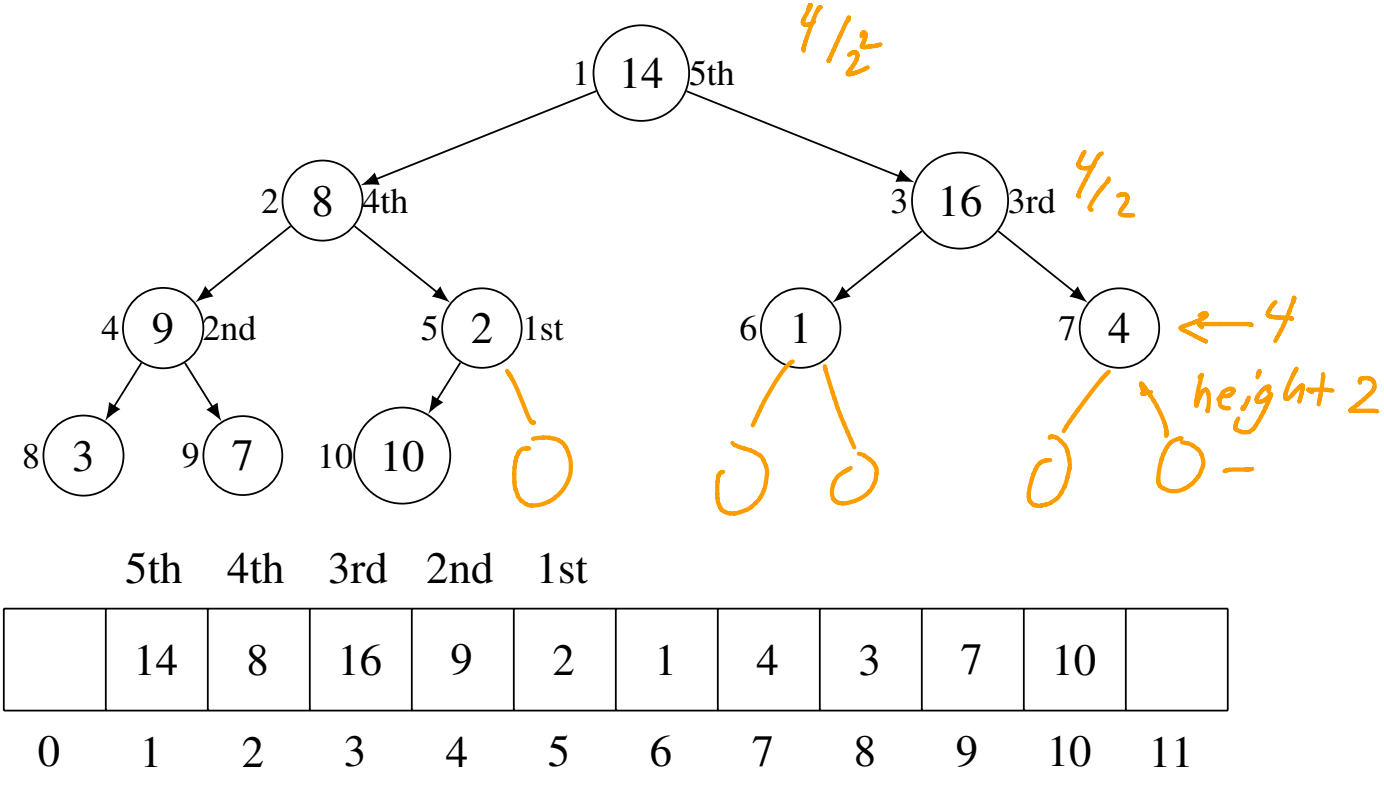
BUILDING A HEAP - EXAMPLE



	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 i = ⌊(heapsize)/2⌋ down to 1:
   heapify node i.
```

BUILD A HEAP COMPLEXITY



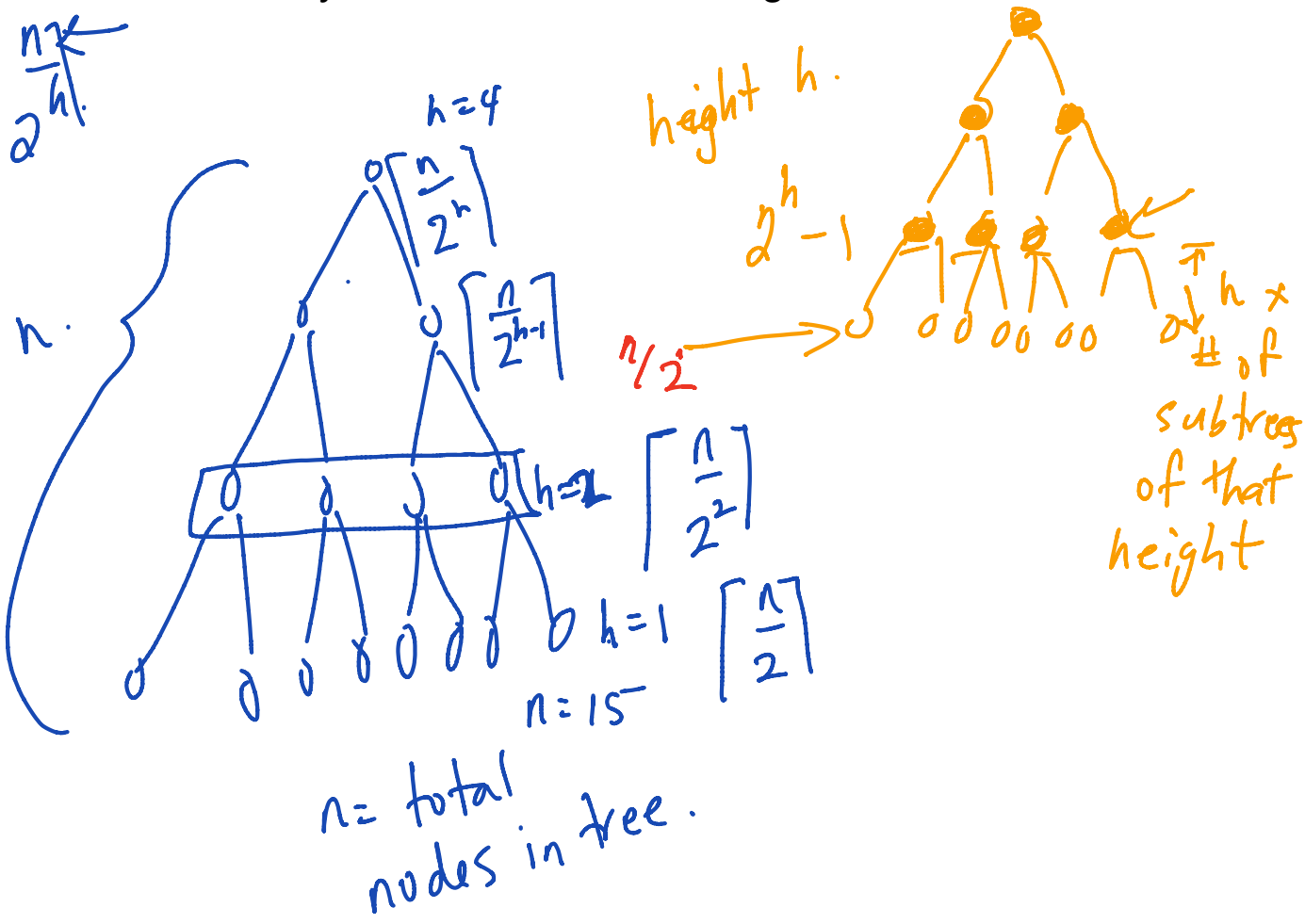
A node at height h takes $h-1$ steps to fix.

Q. At most how many *nodes* are there at height h?

Build a Heap Complexity

Q. At most how many *nodes* are there at height *h*?

A. $\left\lceil \frac{n}{2^h} \right\rceil$

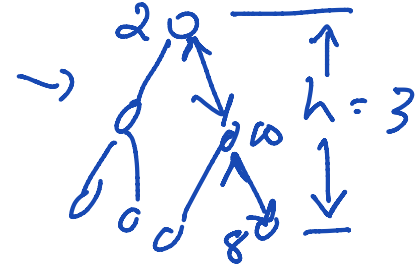


Build a Heap Complexity

Q. At most how many *nodes* are there at height h?

A. $\frac{n}{2^h}$. Why?

We need to sum the cost of each call to `heapify`.

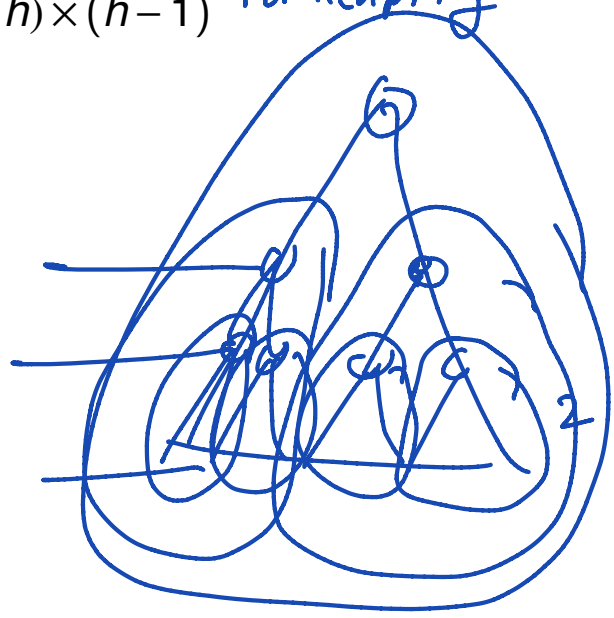


$$\sum_{h=2}^{\lfloor \lg n \rfloor + 1} (\text{num trees of height } h) \times (h-1)$$

$n/2^h$

amount of work for heapify

$$=$$



Build a Heap Complexity

Q. At most how many *nodes* are there at height h?

A. $\frac{n}{2^h}$. Why?

We need to sum the cost of each call to `heapify`.

$$\sum_{h=2}^{\lfloor \lg n \rfloor + 1} (\text{num trees of height } h) \times (h-1)$$

$$= \sum_{h=2}^{\lfloor \lg n \rfloor + 1} \frac{n}{2^h} \times (h-1)$$

work

$$\leq$$

Build a Heap Complexity

Q. At most how many *nodes* are there at height h?

A. $\frac{n}{2^h}$. Why?

We need to sum the cost of each call to `heapify`.

$$
\begin{aligned}
& \sum_{h=2}^{\lfloor \lg n \rfloor + 1} (\text{num trees of height } h) \times (h-1) \\
= & \sum_{h=2}^{\lfloor \lg n \rfloor + 1} \frac{n}{2^h} \times (h-1) \\
\leq & \; n \times \sum_{h=2}^{\infty} \frac{h-1}{2^h} \\
= &
\end{aligned}
$$

Build a Heap Complexity

Q. At most how many *nodes* are there at height h?

A. $\frac{n}{2^h}$. Why?

We need to sum the cost of each call to `heapify`.

cost to build a heap:

$$\begin{aligned}
&\sum_{h=2}^{\lfloor \lg n \rfloor + 1} (\text{num trees of height } h) \times (h-1) \\
&= \sum_{h=2}^{\lfloor \lg n \rfloor + 1} \frac{n}{2^h} \times (h-1) \qquad b \cdot n \le n \cdot \sum \cdots \\
&\le n \times \sum_{h=2}^{\infty} \frac{h-1}{2^h} = n \left(\sum_{h=2}^{\infty} \frac{h}{2^h} - \sum_{h=2}^{\infty} \frac{1}{2^h} \right) \\
&= n \times \text{constant} \qquad \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots \\
&\le 2
\end{aligned}$$

Since $\sum_{h=1}^{\infty} \frac{h}{2^h} \le 2$ the *series converges*. So $\Theta(n)$ time.

$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} \cdots$

Heaps and Sorting

Q. Given an *array*, how can we use a *heap* to efficiently *sort* the array?

A.

Heaps and Sorting

Q. Given an *array*, how can we use a *heap* to efficiently *sort* the array?

A.

- *Convert* the array into a heap - $\Theta(n)$ time.

Heaps and Sorting

Q. Given an *array*, how can we use a *heap* to efficiently *sort* the array?

A.

- *Convert* the array into a heap - $\Theta(n)$ time.
- Repeatedly *extract-max* swapping the *max* value at position *heap-size*, decrement *heap-size*

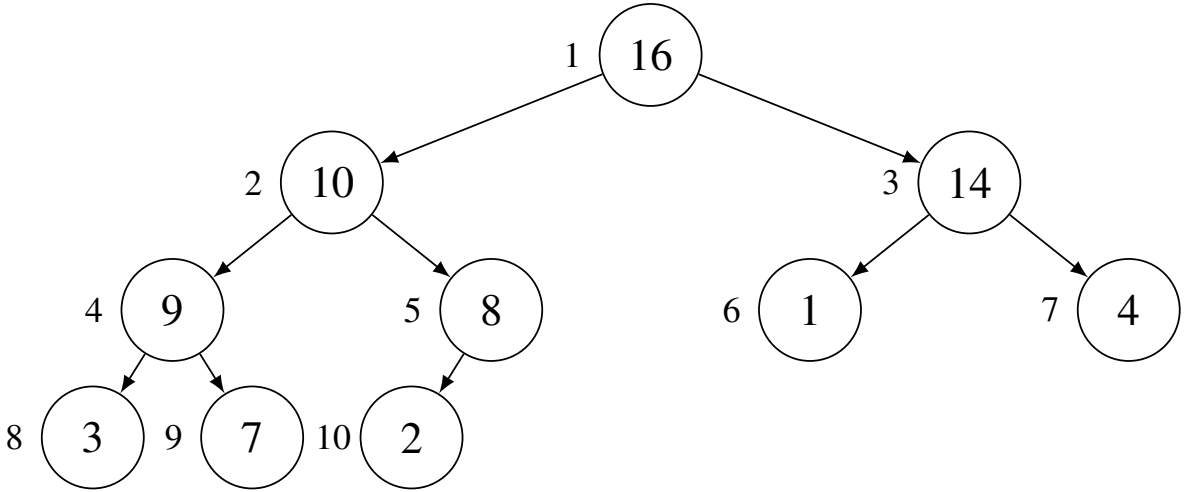
Heaps and Sorting

Q. Given an *array*, how can we use a *heap* to efficiently *sort* the array?

A.

- *Convert* the array into a heap - $\Theta(n)$ time.
- Repeatedly *extract-max* swapping the *max* value at position *heap-size*, decrement *heap-size*
- Total time - $\Theta(n \lg n)$ and *in place*.

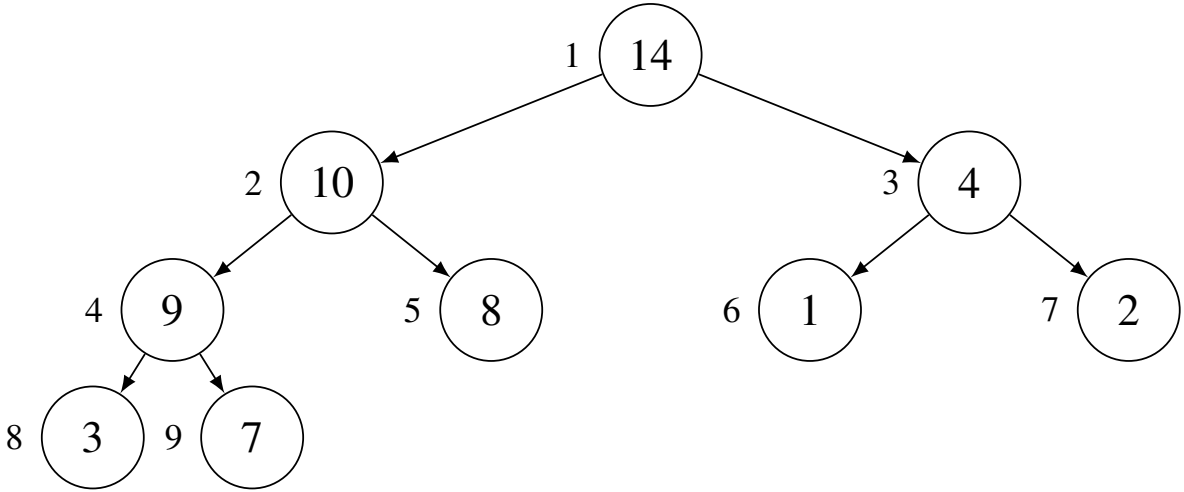
REPEATEDLY EXTRACT-MAX



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

```
while heap-size > 1:
   A[heapsize] = extract-max()
   heap-size = heap-size - 1
```

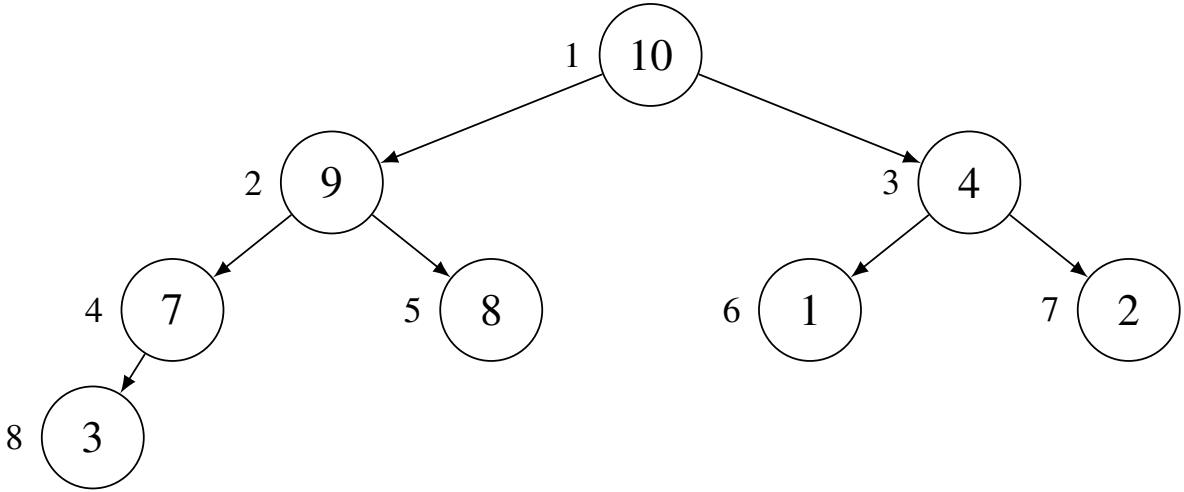
REPEATEDLY EXTRACT-MAX



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

```
while heap-size > 1:
   A[heapsize] = extract-max()
   heap-size = heap-size - 1
```

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

```
while heap-size > 1:
   A[heapsize] = extract-max()
   heap-size = heap-size - 1
```

Max vs Min

We have been storing *larger priorities* near the top to support

max, extract-max, increase-priority

$\Rightarrow$ These are *max priority queues* and *max-heaps*.

Max vs Min

We have been storing *larger priorities* near the top to support

max, extract-max, increase-priority

$\Rightarrow$ These are *max priority queues* and *max-heaps*.

But you could store *smaller priorities* near the top to support

min, extract-min, decrease-priority

$\Rightarrow$ These are *min priority queues* and *min-heaps*.

Max vs Min

We have been storing *larger priorities* near the top to support

max, extract-max, increase-priority

$\Rightarrow$ These are *max priority queues* and *max-heaps*.

But you could store *smaller priorities* near the top to support

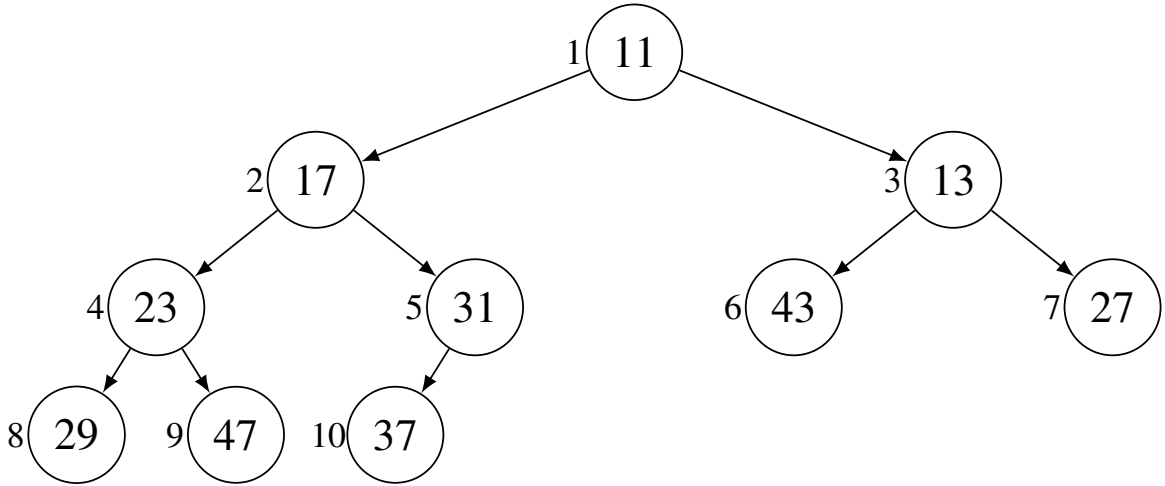
min, extract-min, decrease-priority

$\Rightarrow$ These are *min priority queues* and *min-heaps*.

Example of *min priority queue*: A todo-list with start times.

Another application: coming soon!

MIN-HEAP EXAMPLE



	11	17	13	23	31	43	27	29	47	37	
0	1	2	3	4	5	6	7	8	9	10	11