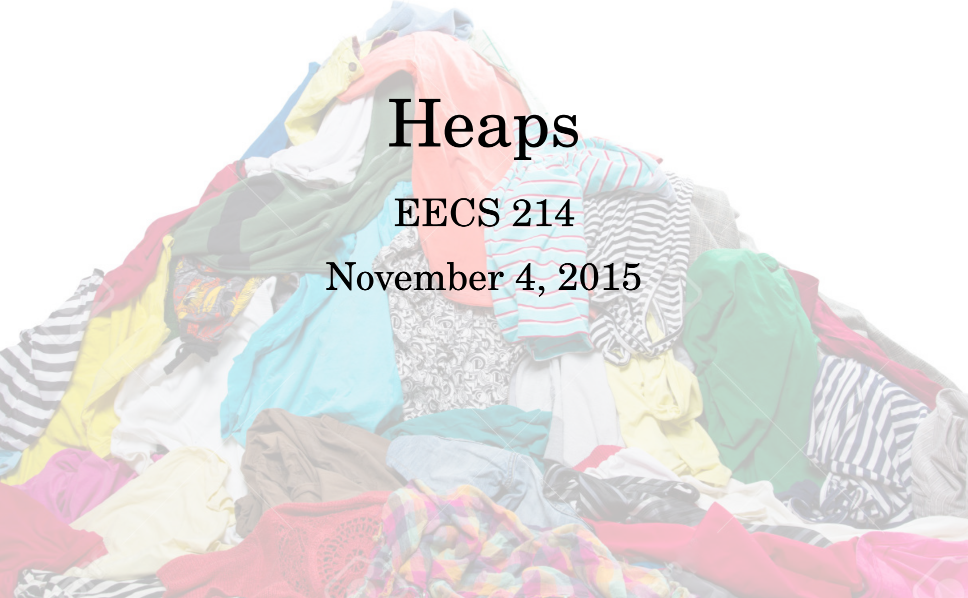
Heaps

EECS 214

November 4, 2015



Take-aways

- What is a priority queue is all about?
- How is the *heap property* defined?
- What does a binary heap look like?
- How do its operations work?
- What are their time complexities?

The *priority queue* ADT

$\textit{Empty}()$: PrioQ

$\textit{Empty?}(\textsf{PrioQ})$: Bool

$\textit{Insert}(\textsf{PrioQ}, \textsf{Element})$

$\textit{FindMin}(\textsf{PrioQ})$: Element

$\textit{RemoveMin}(\textsf{PrioQ})$

Note:

An Element has a *key*; keys are totally ordered

The *priority queue* ADT

representation / operation	linked list
$\mathit{Empty}()$: PrioQ	$\mathcal{O}(1)$
$\mathit{Empty}?(\text{PrioQ})$: Bool	$\mathcal{O}(1)$
$\mathit{Insert}(\text{PrioQ}, \text{Element})$	$\mathcal{O}(1)$
$\mathit{FindMin}(\text{PrioQ})$: Element	$\mathcal{O}(n)$
$\mathit{RemoveMin}(\text{PrioQ})$	$\mathcal{O}(n)$

Notes:

1. An Element has a *key*; keys are totally ordered
2. n is the number of elements

The *priority queue* ADT

representation / operation	linked list	sorted array
$\mathit{Empty}()$: PrioQ	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\mathit{Empty?}(\mathsf{PrioQ})$: Bool	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\mathit{Insert}(\mathsf{PrioQ}, \mathsf{Element})$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
$\mathit{FindMin}(\mathsf{PrioQ})$: $\mathsf{Element}$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
$\mathit{RemoveMin}(\mathsf{PrioQ})$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Notes:

1. An Element has a *key*; keys are totally ordered
2. n is the number of elements

The *priority queue* ADT

operation \ representation	linked list	sorted array
$\mathit{Empty}()$: PrioQ	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\mathit{Empty}?(\mathsf{PrioQ})$: Bool	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\mathit{Insert}(\mathsf{PrioQ}, \mathsf{Element})$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
$\mathit{FindMin}(\mathsf{PrioQ})$: Element	$\mathcal{O}(n)$	$\mathcal{O}(1)$
$\mathit{RemoveMin}(\mathsf{PrioQ})$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Notes:

1. An Element has a *key*; keys are totally ordered
2. n is the number of elements

The *priority queue* ADT

representation operation	linked list	sorted ring buffer
$\mathit{Empty}()$: PrioQ	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\mathit{Empty?}$(PrioQ) : Bool	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert(PrioQ, Element)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
$\mathit{FindMin}$(PrioQ) : Element	$\mathcal{O}(n)$	$\mathcal{O}(1)$
$\mathit{RemoveMin}$(PrioQ)	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Notes:

1. An Element has a *key*; keys are totally ordered
2. n is the number of elements

We can do better

A *heap* is a tree that satisfies the *heap property*

We can do better

A *heap* is a tree that satisfies the *heap property*: every element's key is less than all of its descendants' keys

We can do better

A *min-heap* is a tree that satisfies the *min-heap property*: every element's key is less than all of its descendants' keys

We can do better

A *max-heap* is a tree that satisfies the *max-heap property*: every element's key is greater than all of its descendants' keys

Heaps versus search trees

min-heap property:

for all nodes n,

- $n.key < n.left.key$, and
- $n.key < n.right.key$

BST property:

for all nodes n,

- for all of n's left-descendants ℓ,
  $\ell.key < n.key$, and
- for all of n's right-descendants r,
  $r.key > n.key$

Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

Definition: *complete tree*

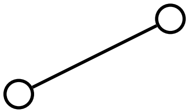
A tree is *complete* if the levels are all filled in left-to-right

Like this:

Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

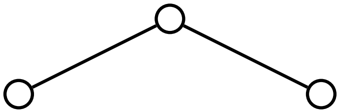
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

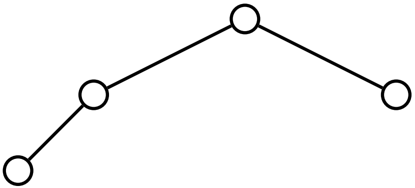
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

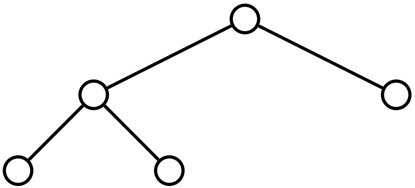
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

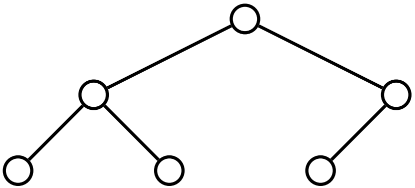
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

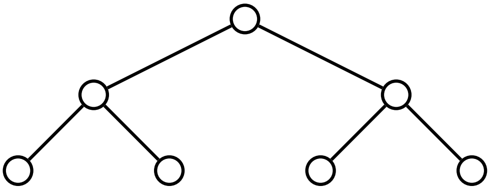
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

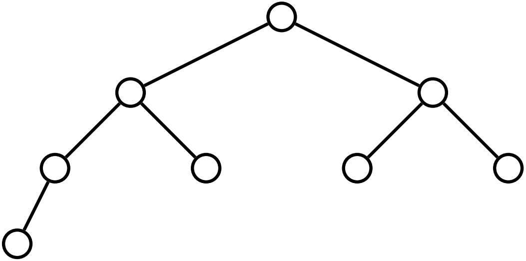
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

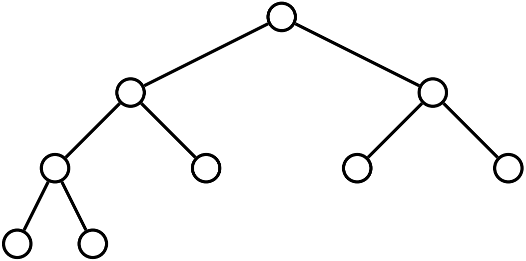
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

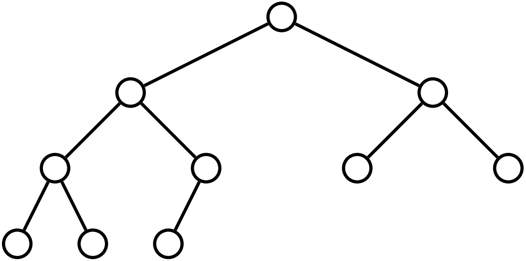
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

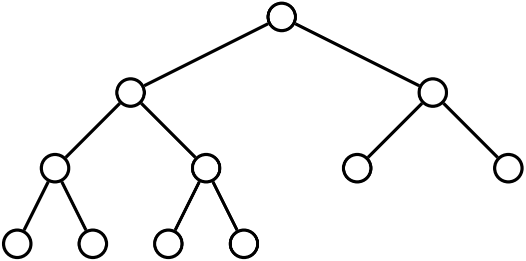
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

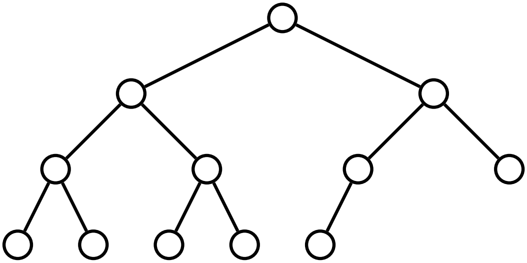
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

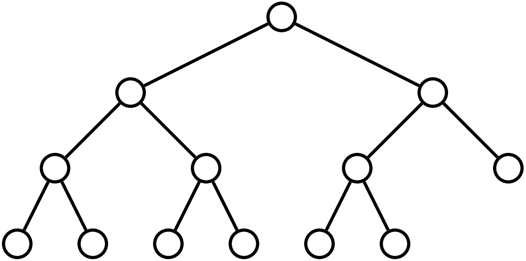
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

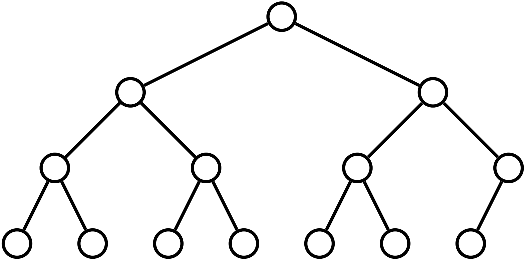
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

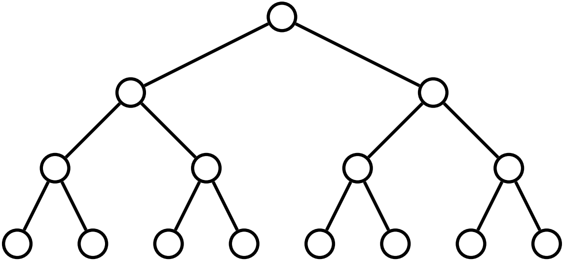
Like this:



Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

Definition: *binary heap*

A *binary heap* is a complete binary tree
satisfying the heap property

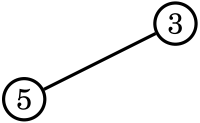
Like this:

3

Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

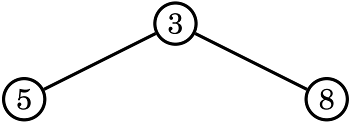
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

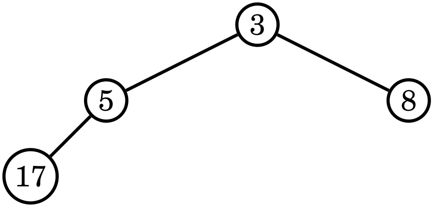
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

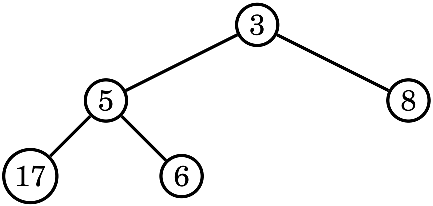
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

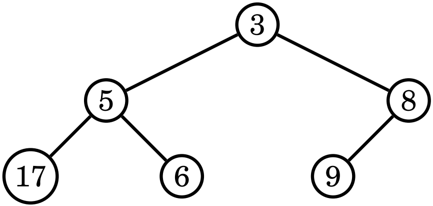
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

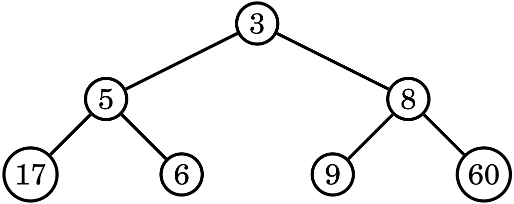
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

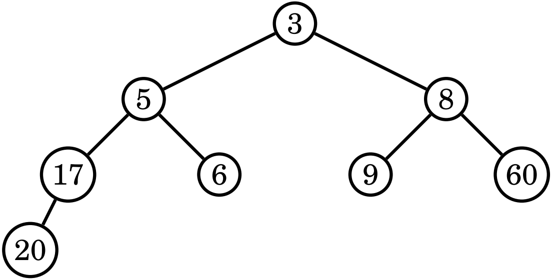
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

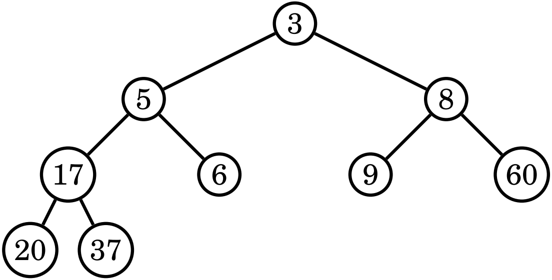
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

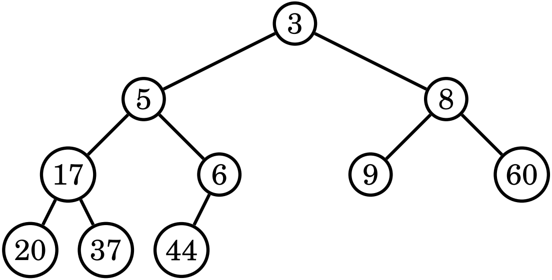
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

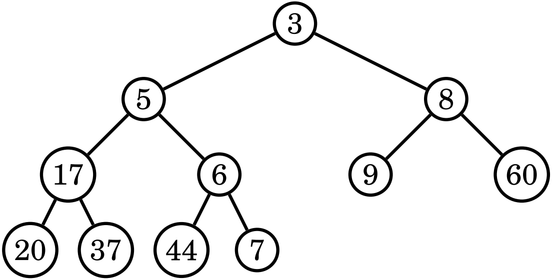
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

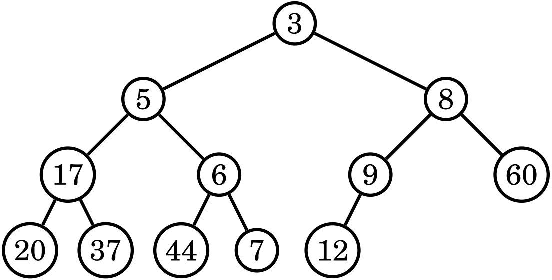
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

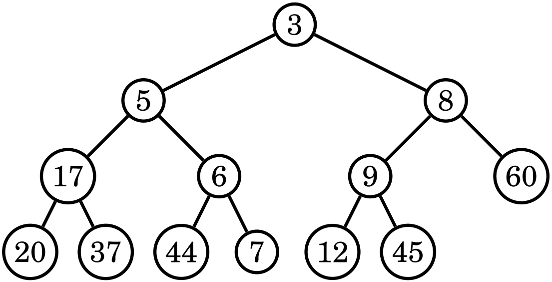
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

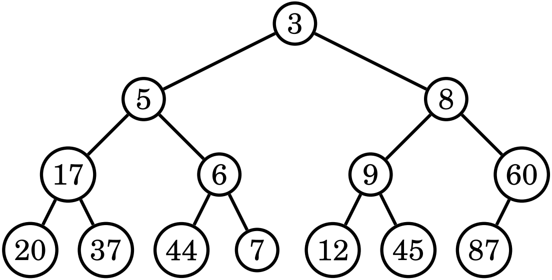
Like this:



Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

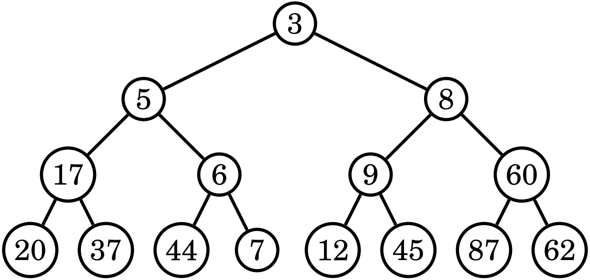
Like this:



Definition: *binary heap*

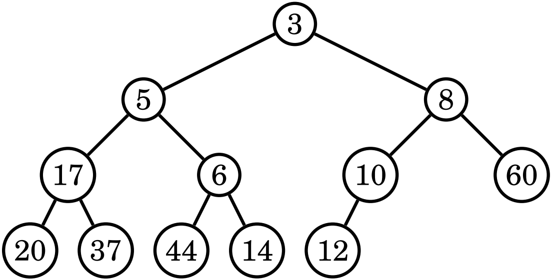
A *binary heap* is a complete binary tree satisfying the heap property

Like this:



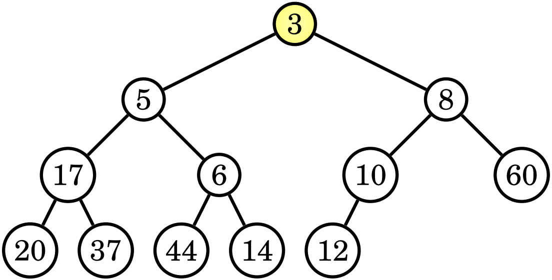
Operation *FindMin*

This one is easy:



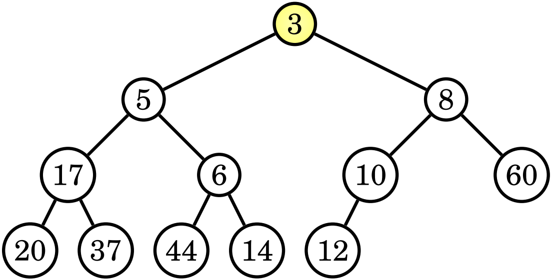
Operation *FindMin*

This one is easy:



Operation *FindMin*

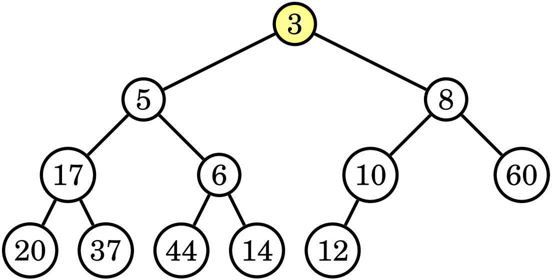
This one is easy:



How long does this take?

Operation *FindMin*

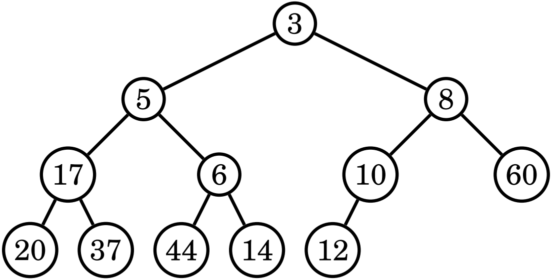
This one is easy:



How long does this take? $\mathcal{O}(1)$

Operation *Insert*

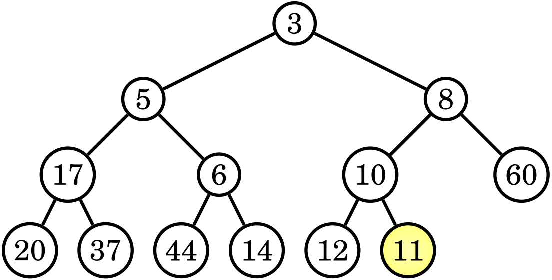
This one's a bit harder. Let's insert 11 into the heap.



Operation *Insert*

This one's a bit harder. Let's insert 11 into the heap.

Step 1: Add it at the end of the heap

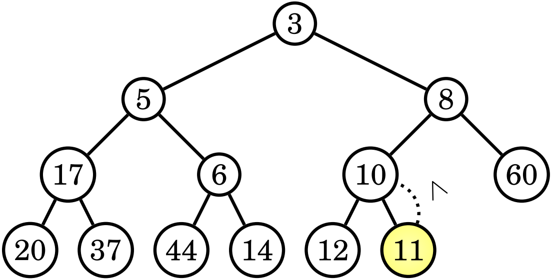


Operation *Insert*

This one's a bit harder. Let's insert 11 into the heap.

Step 1: Add it at the end of the heap

Step 2: Check if the heap condition is (locally!) preserved

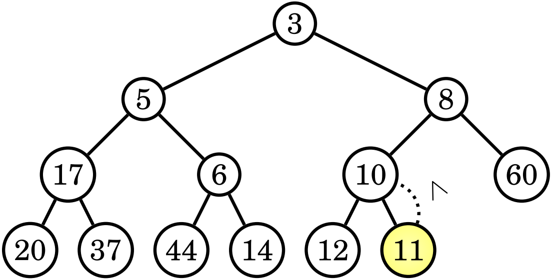


Operation *Insert*

This one's a bit harder. Let's insert 11 into the heap.

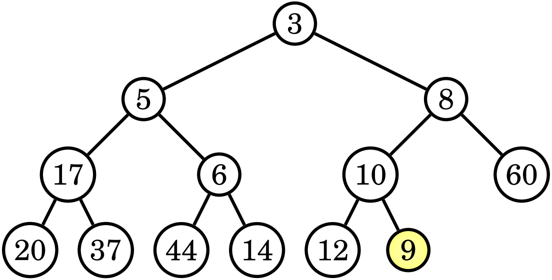
Step 1: Add it at the end of the heap
Step 2: Check if the heap condition is (locally!) preserved

It is, so we're done! Why is the local check sufficient?



Operation *Insert*

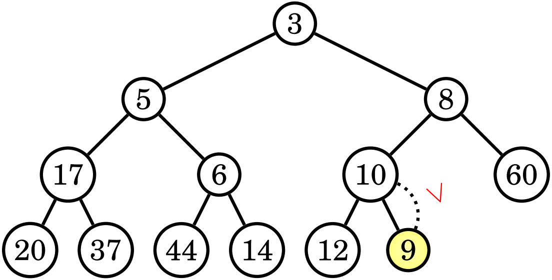
Okay, let's try inserting 9 instead.



Operation *Insert*

Okay, let's try inserting 9 instead.

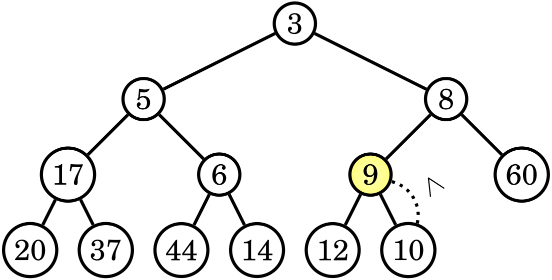
The local invariant is broken! How can we fix it?



Operation *Insert*

Okay, let's try inserting 9 instead.

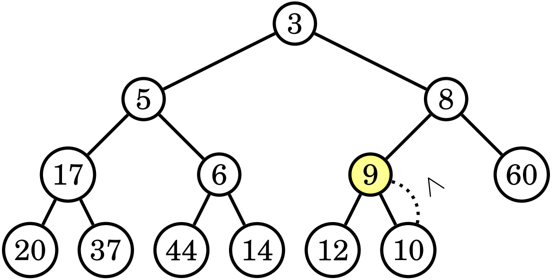
The local invariant is broken! How can we fix it?
Swap the troublesome node with its parent.



Operation *Insert*

Okay, let's try inserting 9 instead.

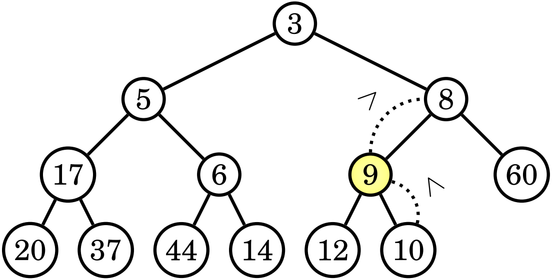
The local invariant is broken! How can we fix it?
Swap the troublesome node with its parent.
Now we check 9's new parent.



Operation *Insert*

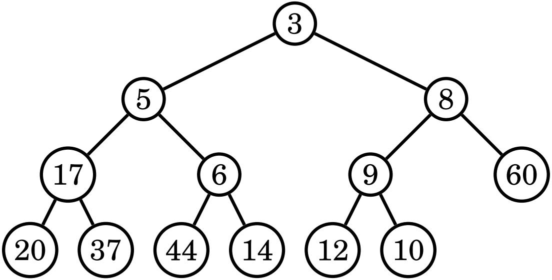
Okay, let's try inserting 9 instead.

The local invariant is broken! How can we fix it?
Swap the troublesome node with its parent.
Now we check 9's new parent. Looks good.



Operation *Insert*

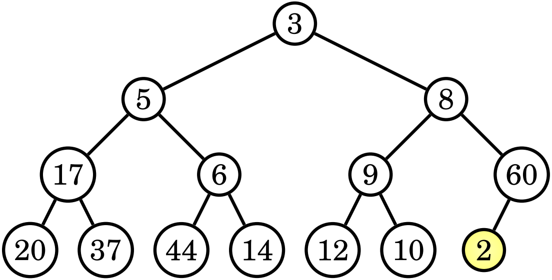
Okay, now let's insert 2.



Operation *Insert*

Okay, now let's insert 2.

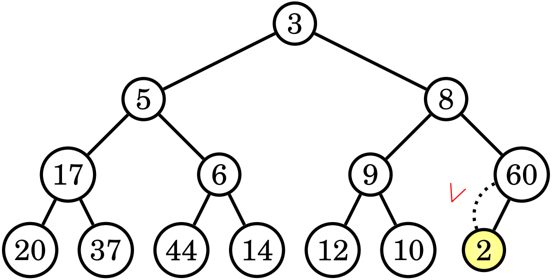
Check the local invariant.



Operation *Insert*

Okay, now let's insert 2.

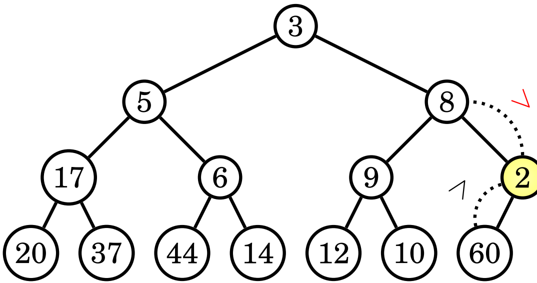
Check the local invariant. It's broken!



Operation *Insert*

Okay, now let's insert 2.

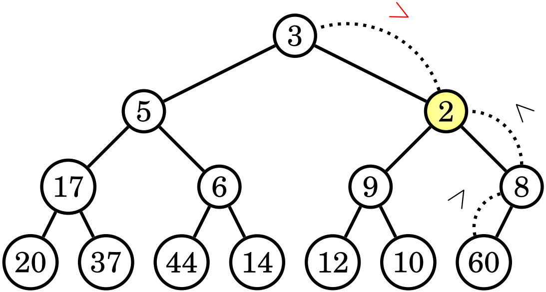
Check the local invariant. It's broken!
So swap with the parent. Still broken!



Operation *Insert*

Okay, now let's insert 2.

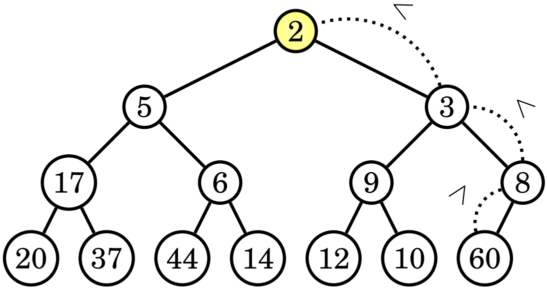
Check the local invariant. It's broken!
So swap with the parent. Still broken!
So "bubble up" until the invariant is restored.



Operation *Insert*

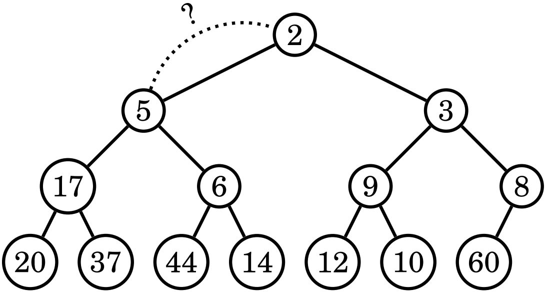
Okay, now let's insert 2.

Check the local invariant. It's broken!
So swap with the parent. Still broken!
So "bubble up" until the invariant is restored.



Operation *Insert*

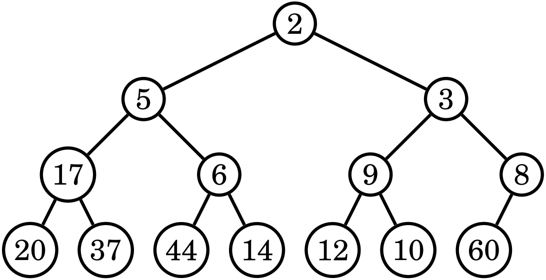
When bubbling up, why didn't we compare the node against its other child? (*E.g.*: comparing 2 to 5)



Operation *Insert*

When bubbling up, why didn't we compare the node against its other child? (*E.g.*: comparing 2 to 5)

Before swap, node $<$ parent, but also parent $<$ other child (by heap condition). Transitivity of $<$ tells us that node $<$ other child!

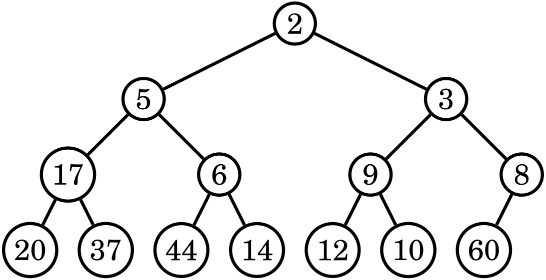


How long does this take?

Operation *Insert*

When bubbling up, why didn't we compare the node against its other child? (*E.g.*: comparing 2 to 5)

Before swap, node $<$ parent, but also parent $<$ other child (by heap condition). Transitivity of $<$ tells us that node $<$ other child!

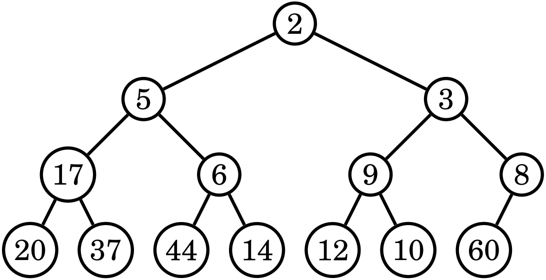


How long does this take? How tall is the tree?

Operation *Insert*

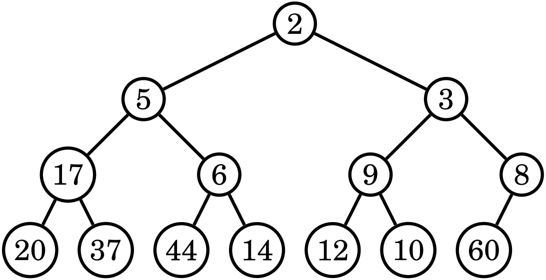
When bubbling up, why didn't we compare the node against its other child? (*E.g.*: comparing 2 to 5)

Before swap, node $<$ parent, but also parent $<$ other child (by heap condition). Transitivity of $<$ tells us that node $<$ other child!



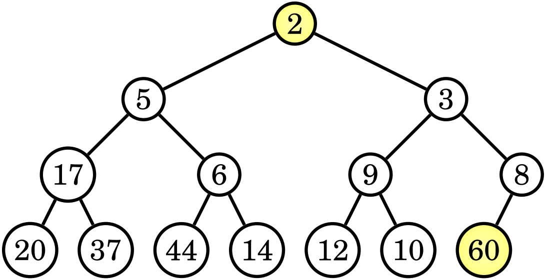
How long does this take? How tall is the tree? $\mathcal{O}(\log n)$

Operation *RemoveMin*



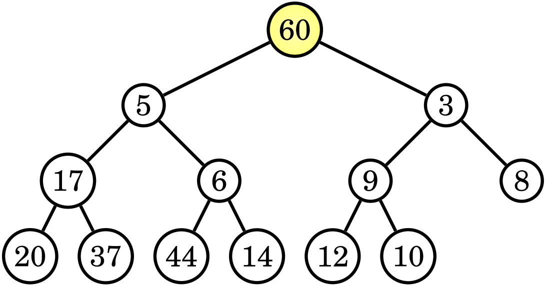
Operation *RemoveMin*

Step 1: Replace the root with the *last* node, and remove the last node.



Operation *RemoveMin*

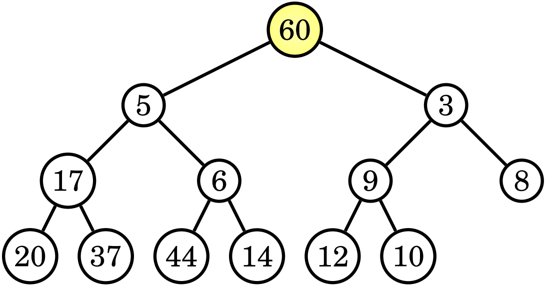
Step 1: Replace the root with the *last* node, and remove the last node. (This preserves tree completeness.)



Operation *RemoveMin*

Step 1: Replace the root with the *last* node, and remove the last node. (This preserves tree completeness.)

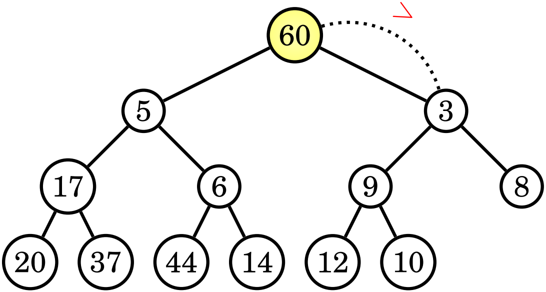
Step 2: Restore the invariant by "percolating down": swap new node with its *smaller* child until invariant is restored.



Operation *RemoveMin*

Step 1: Replace the root with the *last* node, and remove the last node. (This preserves tree completeness.)

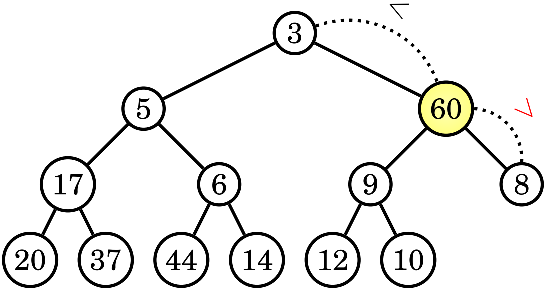
Step 2: Restore the invariant by "percolating down": swap new node with its *smaller* child until invariant is restored.



Operation *RemoveMin*

Step 1: Replace the root with the *last* node, and remove the last node. (This preserves tree completeness.)

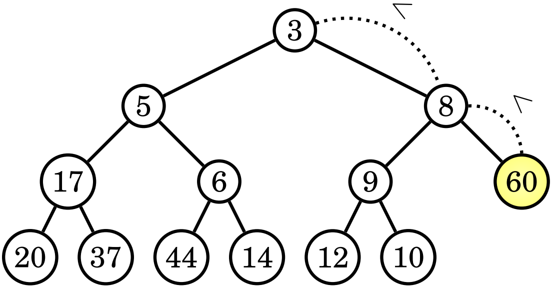
Step 2: Restore the invariant by "percolating down": swap new node with its *smaller* child until invariant is restored.



Operation *RemoveMin*

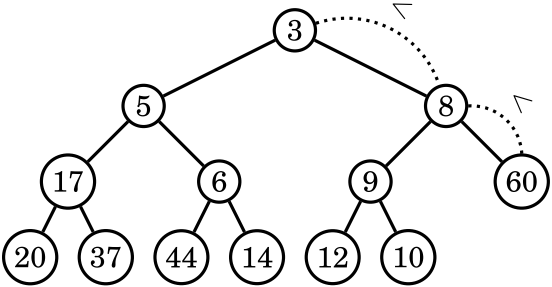
Step 1: Replace the root with the *last* node, and remove the last node. (This preserves tree completeness.)

Step 2: Restore the invariant by “percolating down”: swap new node with its *smaller* child until invariant is restored.



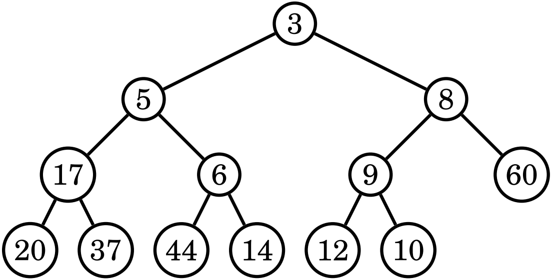
Operation *RemoveMin*

Why do we swap with the smaller child?



Operation *RemoveMin*

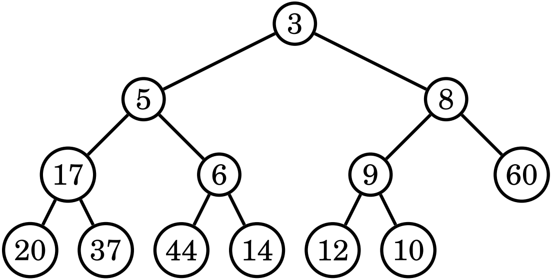
Why do we swap with the smaller child? Transitivity again!



Operation *RemoveMin*

Why do we swap with the smaller child? Transitivity again!

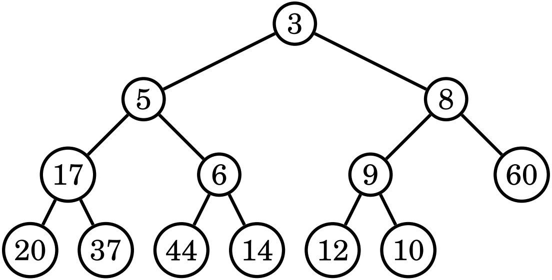
How long does this take?



Operation *RemoveMin*

Why do we swap with the smaller child? Transitivity again!

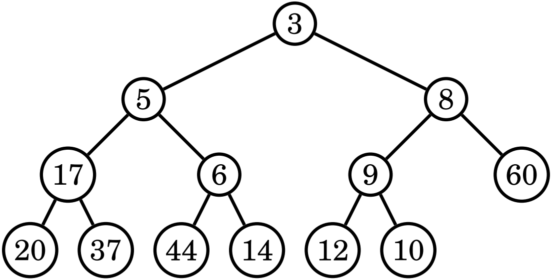
How long does this take? How tall is the tree?



Operation *RemoveMin*

Why do we swap with the smaller child? Transitivity again!

How long does this take? How tall is the tree? $\mathcal{O}(\log n)$



Take-aways

- What is a priority queue is all about?
- How is the *heap property* defined?
- What does a binary heap look like?
- How do its operations work?
- What are their time complexities?