Binary Heaps

CMPUT 275 - Winter 2018

University of Alberta

Binary Heaps

Recall in Dijkstra's algorithm we had to maintain a set of **events**: each was some *burning edge* and stored a value indicating when this edge finished burning.

We had to extract the next event that would happen.

Thus, a **heap** was introduced. A data structure that holds (*item*, *key*) pairs where we can extract the item with a minimum key efficiently.

Heaps

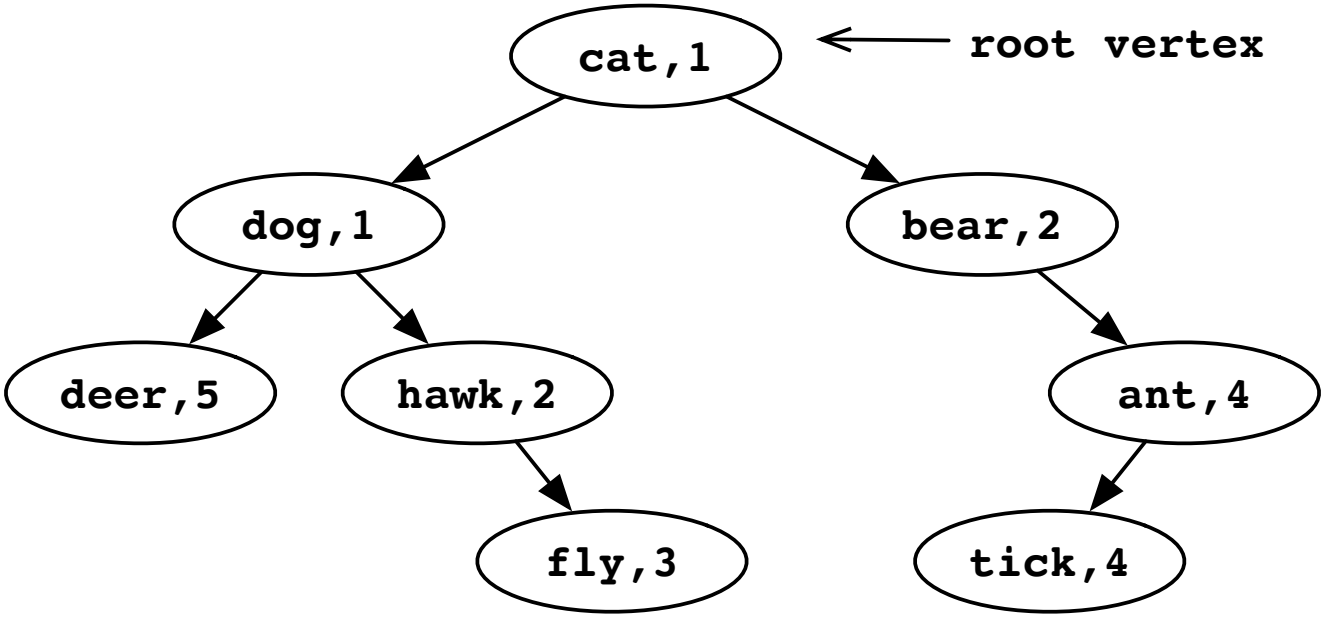
First, we discuss heaps generically. Then we discuss **binary heaps**: the type of heap we will implement together.

Recall a heap supports the following operations:

- `insert(x, key)`: add `x` to the heap with the given key value.
- `min()`: returns the pair `(x, key)` with the smallest key.
- `popmin()`: removes and returns the pair `(x, key)` with the smallest key.
- `size()`: number of items being held.

Heaps

Most heaps are **rooted trees**. Trees with a special **root** vertex.



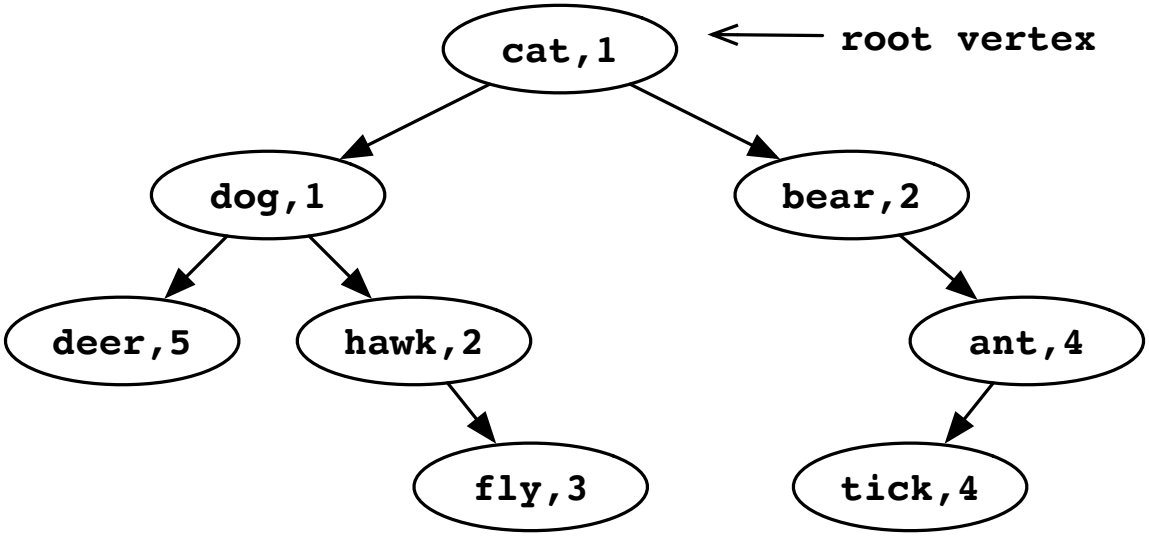
Each vertex holds an (*item*, *key*) pair.

(Funny how trees in computing science are frequently drawn **down** from the root!)

The Heap Property

We say a vertex v has the **heap property** if its key is *at most* the key of every child (vertices hanging directly under v).

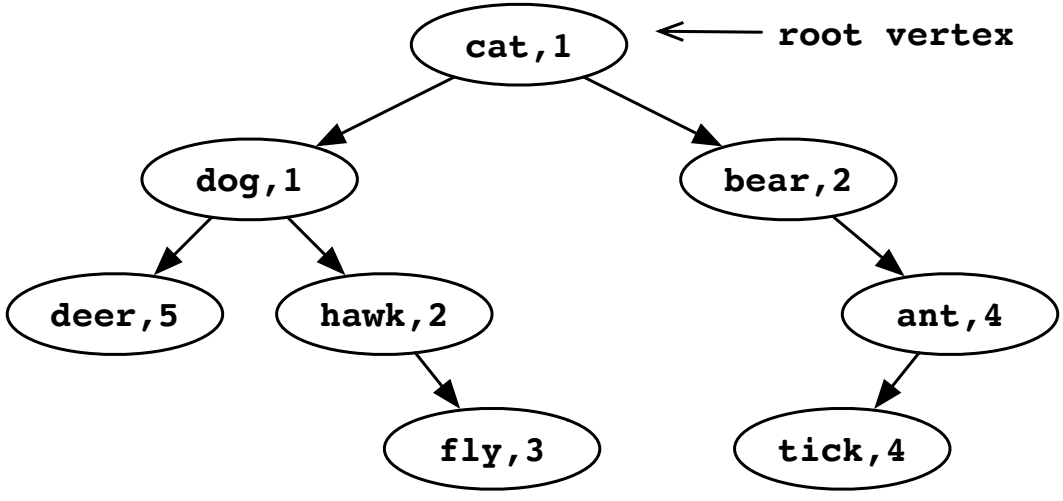
e.g. The children of `dog` are just `deer` and `hawk`. In particular, `fly` is not a child of `dog`: we would call it a **descendent** of `dog`.



In a heap, **every vertex** has the heap property.

The Heap Property

The root always contains a minimum-key item!



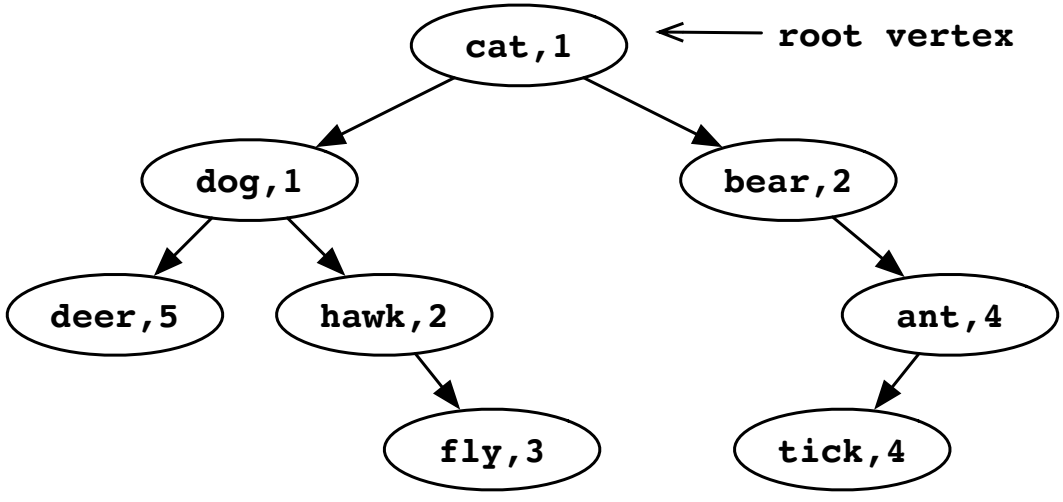
Proof

If not, let v_0 be the **root** and let $v_1, \ldots, v_k$ be a path in the tree where v_k has a minimum-key item (and the only minimum-key item on the path).

Then $k > 0$, so v_{k-1} does not have the heap property.

The Heap Property

So in any heap, we can easily support the `min()` operation: just return what is held by the root.

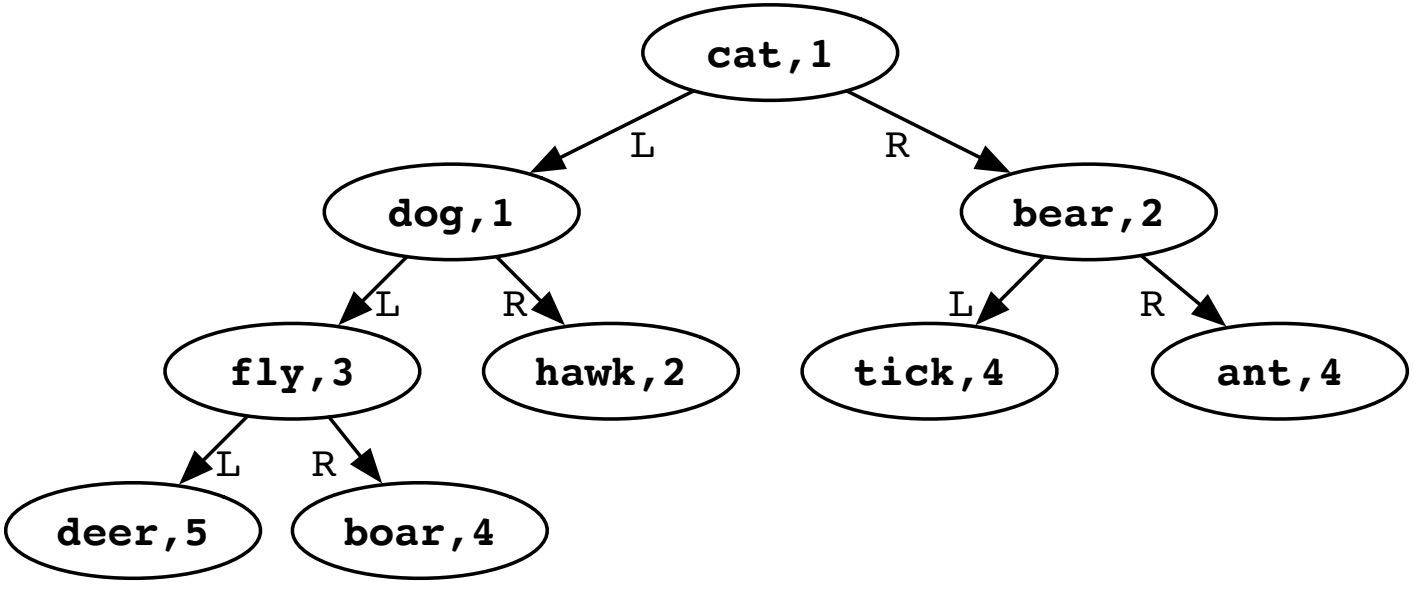


(picture: just return `cat, 1`)

What about popping the minimum? Much harder. We can remove the root from the tree but how should we restructure the tree so it is a heap again?

Binary Heaps

A particular way to implement a heap efficiently. The tree is a **complete binary tree**. Every vertex has at most 2 children, distinguished as the **left** and **right** child.

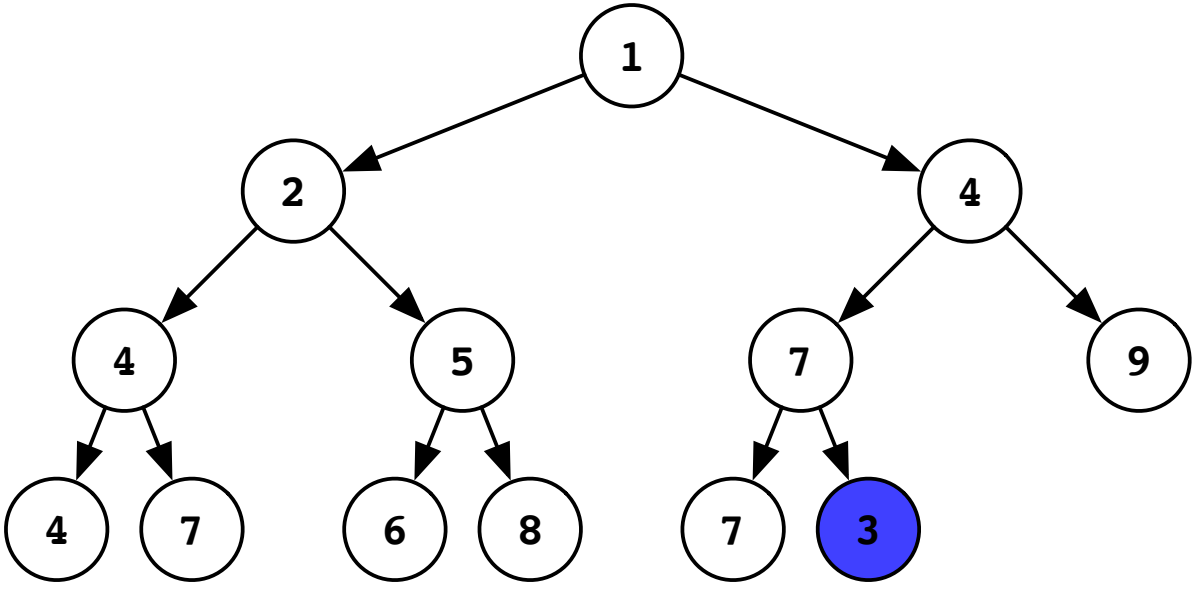


All levels except, perhaps, the last level are **full**. The last level has all its vertices in a *left-to-right* fashion.

Insertion

Note: Only keys will be shown in the figures from now on, but recall the vertices also hold items. The left/right labels are also missing.

To insert, just add the (*item*, *key*) pair into the next new location! You might have to make a new layer if the last one is full.

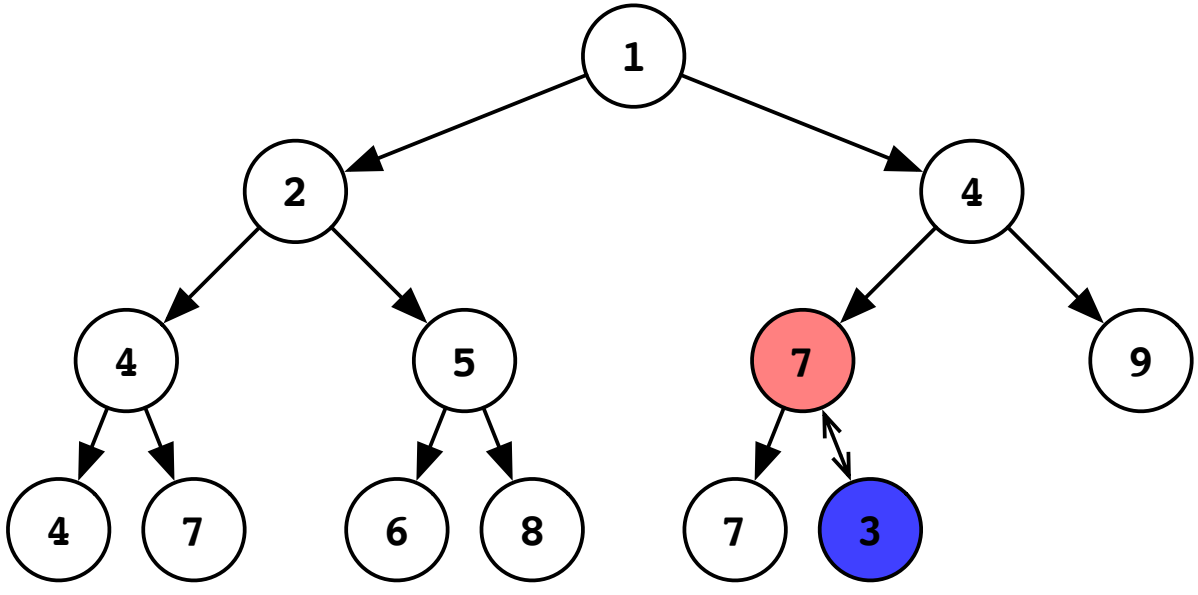


Pictured: Inserting an item with key 3. Put it at the end first (blue).

Insertion

This is almost a heap. The only vertex that may not satisfy the heap property is the parent u of the new vertex v.

If so, **fix by swapping the contents of these vertices**. Swap both the keys and items (they always travel together).

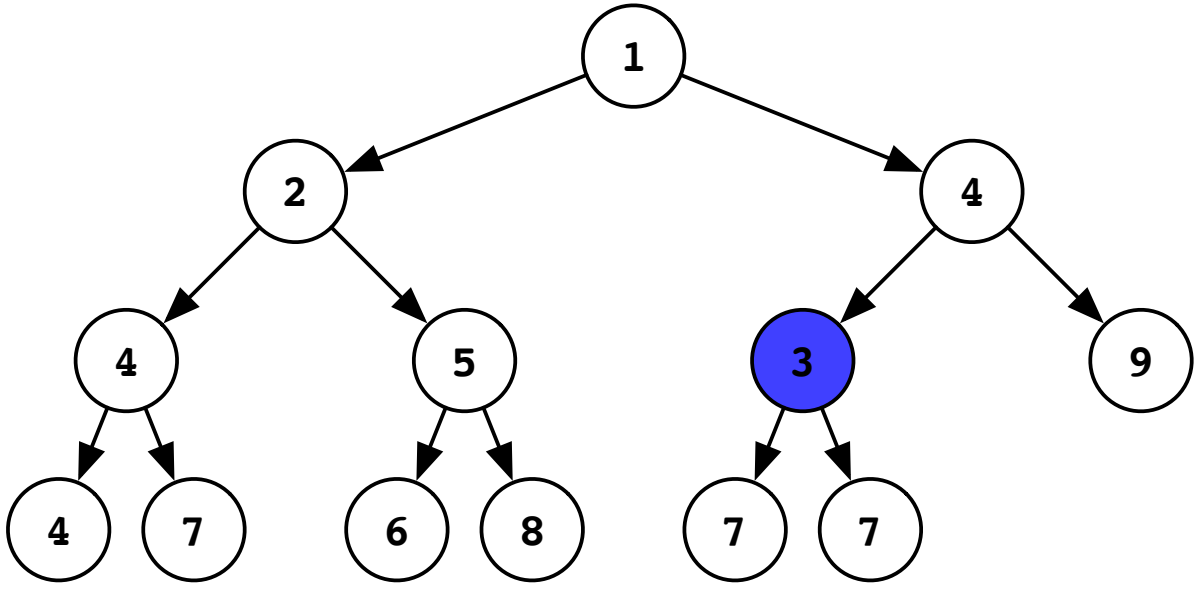


Pictured: We should swap the two coloured vertices.

Insertion

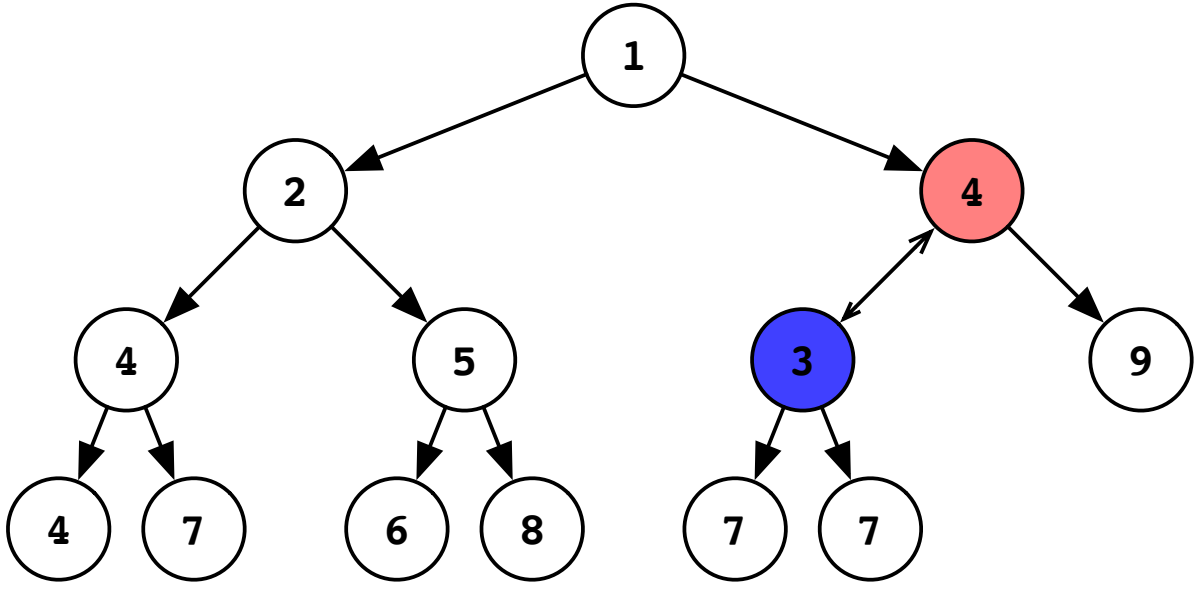
This is almost a heap. The only vertex that may not satisfy the heap property is the parent u of the new vertex v.

If so, **fix by swapping the contents of these vertices**. Swap both the keys and items (they always travel together).



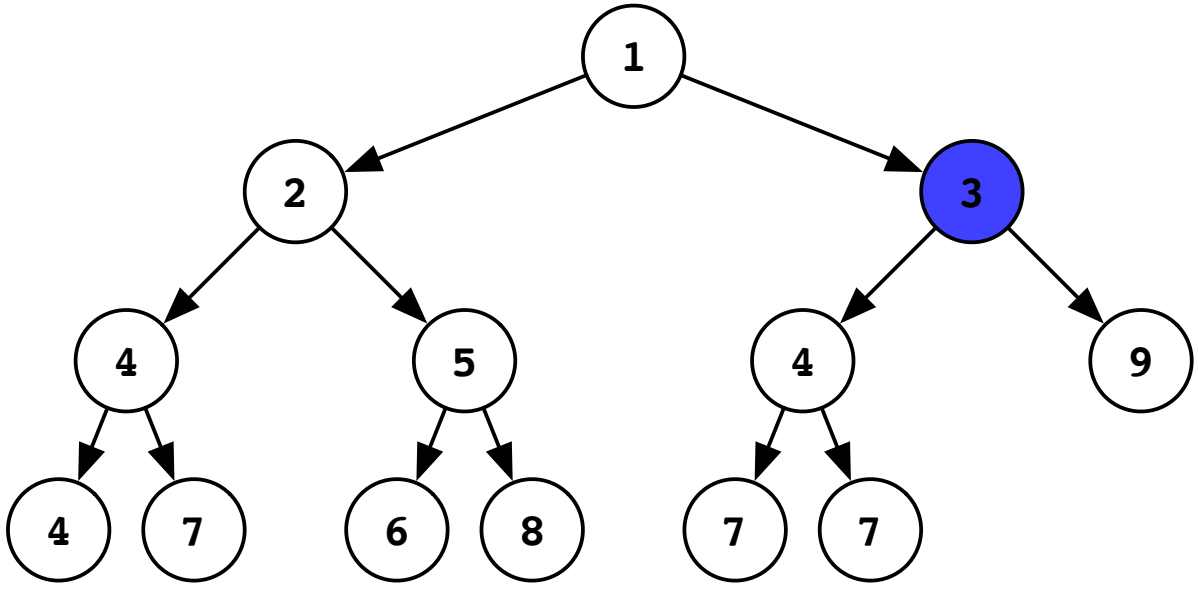
Pictured: After the swap.

Repeat with the new location of the newly-added item/key. Check it's parent, swap contents if necessary, and repeat.



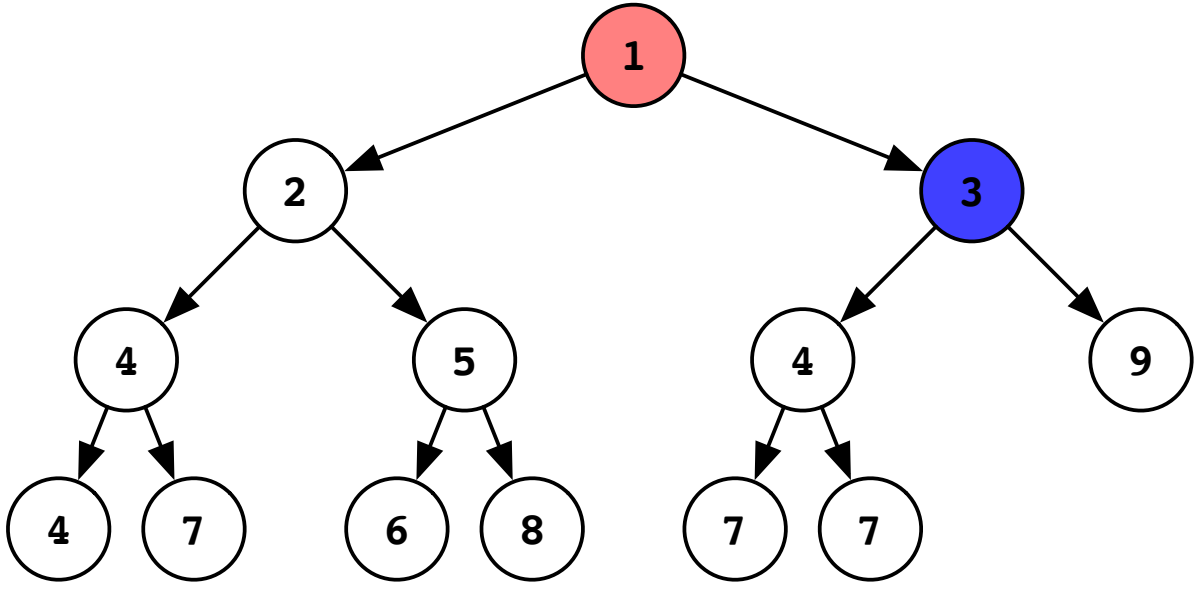
Pictured: The parent of the new item/key has a smaller key. We have to swap.

Repeat with the new location of the newly-added item/key. Check it's parent, swap contents if necessary, and repeat.



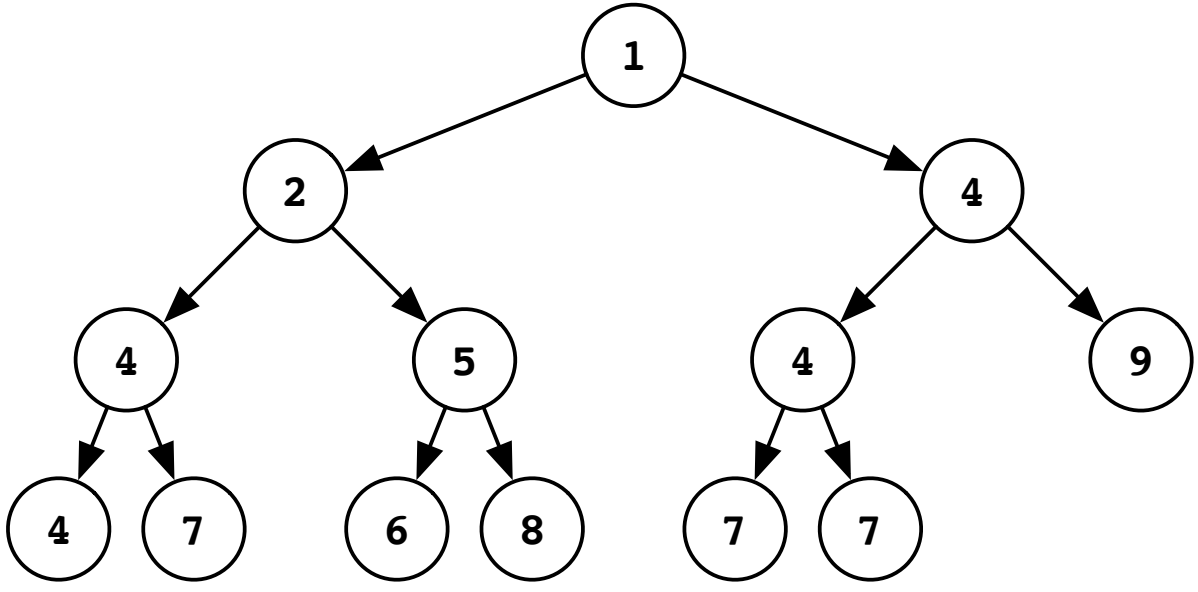
Pictured: After the swap.

Repeat with the new location of the newly-added item/key. Check it's parent, swap contents if necessary, and repeat.



Pictured: Checking the parent again. No problem: its key is $\leq$ the newly-added key.

Repeat with the new location of the newly-added item/key. Check it's parent, swap contents if necessary, and repeat.



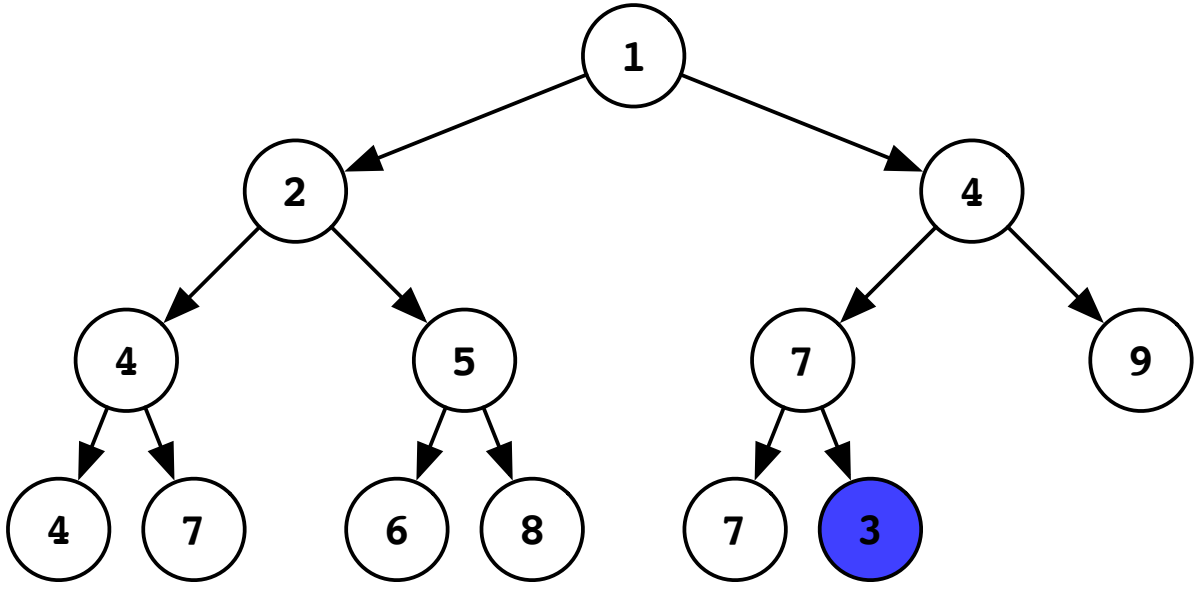
Pictured: The heap after completing the insertion. All vertices have the heap property.

Verification: Does This Always Work?

Invariant

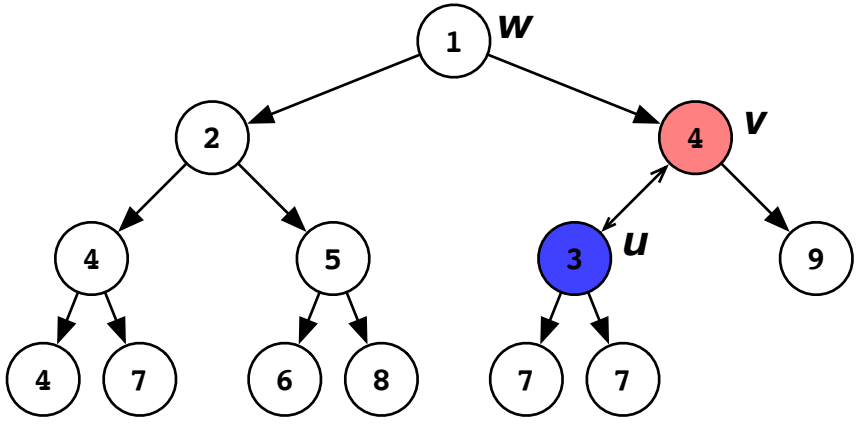
Initially and after each swap, all vertices except, perhaps, the parent of the vertex with the new item/key have the heap property satisfied.

Trivially, it is true initially:



Verification: After a Swap

Suppose u had the old key and swapped its contents with its parent v.



- The key of u did not increase beyond its children because they were already descendants of the new key. The heap property holds there.
- The key of v decreased and the only child that changed had its key increase. The heap property still holds at v.

The only other vertex that had its key or a child's key change is the parent w of v. **The invariant holds!**

Insertion

What is the running time? $O(\#\text{layers in the tree})$.

Fun Math

Express in terms of total # items. Say the heap had n items and ℓ layers. The number of vertices in each layer $0 \leq i < \ell$ is 2^i so,

$$n \geq 2^0 + 2^1 + 2^2 + \ldots + 2^{\ell-1} = 2^\ell - 1$$

Rearranging:

$$\ell \leq \log_2(n+1)$$

So the running time is $O(\log n)$.

Insertion Pseudocode

Let $key(v)$ denote the key held at a vertex v and $parent(v)$ be the parent vertex of v in the tree.

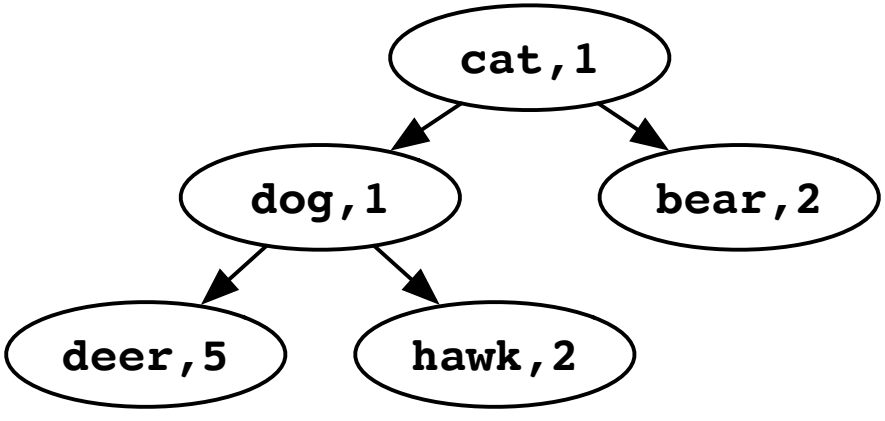
Algorithm 1 Inserting($item, key$) into the heap.

```
let v be a new vertex in the tree          # only one place to put v
give v the pair (item, key)
while v is not the root and key(v) < key(parent(v)) do
    swap the items and keys between v and parent(v)
    v ← parent(v)                          # crawl up the tree
```

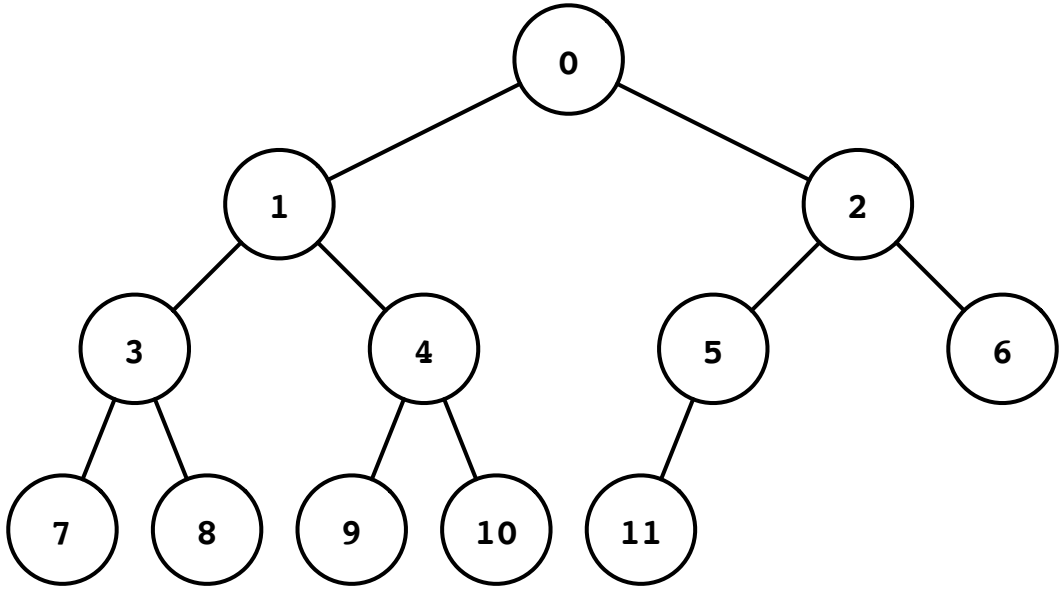
Implementation Consideration!

There is an *awesome* trick to make implementation of binary heaps a breeze!

Store the vertices as consecutive elements of an array/list, starting with the root at index 0.




```
[ (cat,1), (dog,1), (bear, 2), (deer, 5), (hawk, 2) ]
```



Pictured: Indices into the list/array for the corresponding vertex.

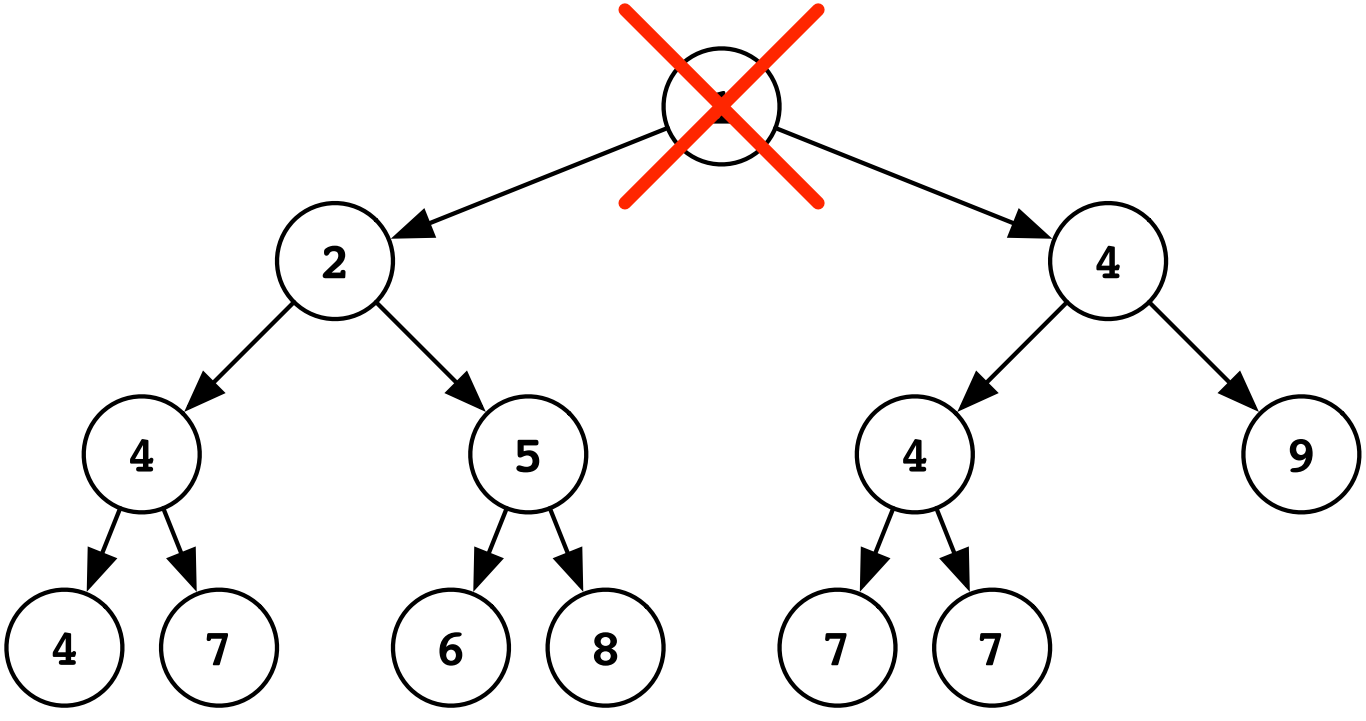
Left child of index i: $2i + 1$.
Right child of index i: $2i + 2$.
Parent of index $i > 0$: $\lfloor (i - 1)/2 \rfloor$ (the quotient).

Bonus: The last item in the last layer is just the end of the list.

Now Let's Code It!

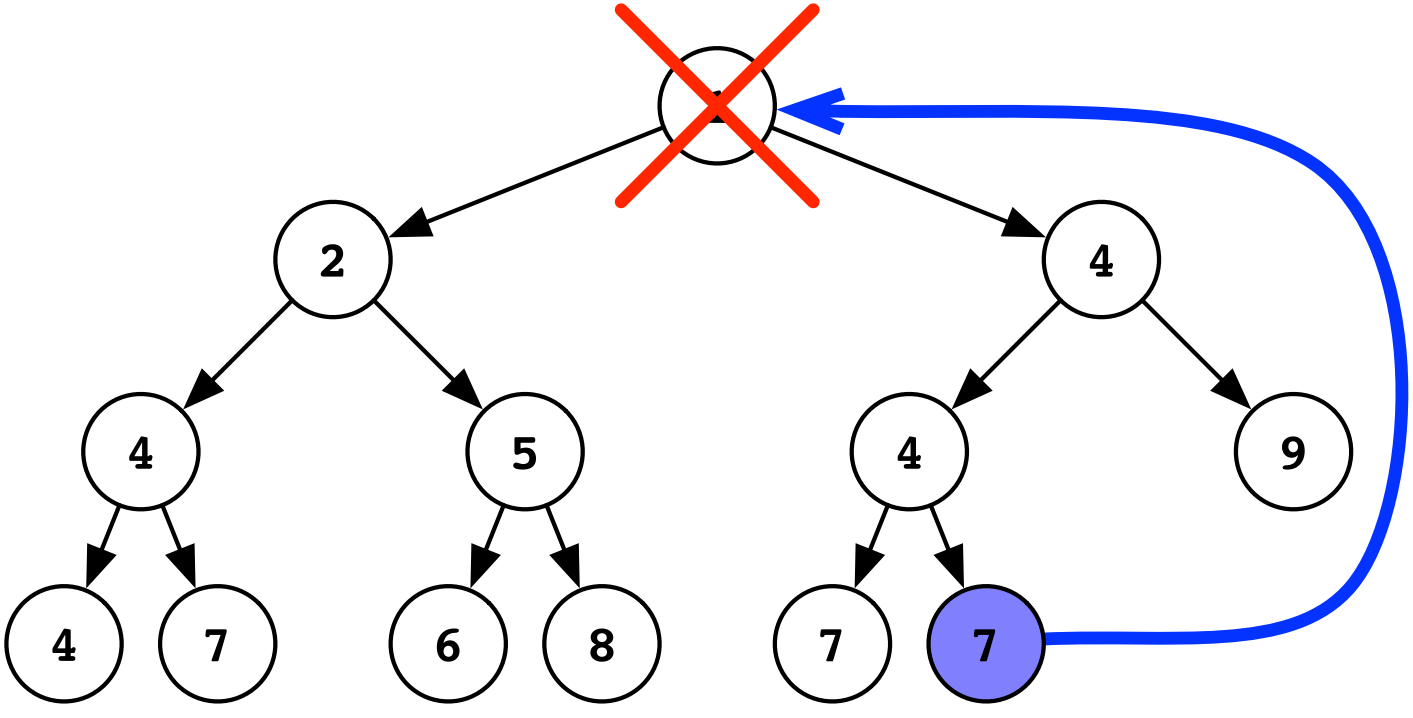
Popping

To pop the minimum value, we have to remove the root of the tree and then restructure it to look like a binary heap again.



Popping

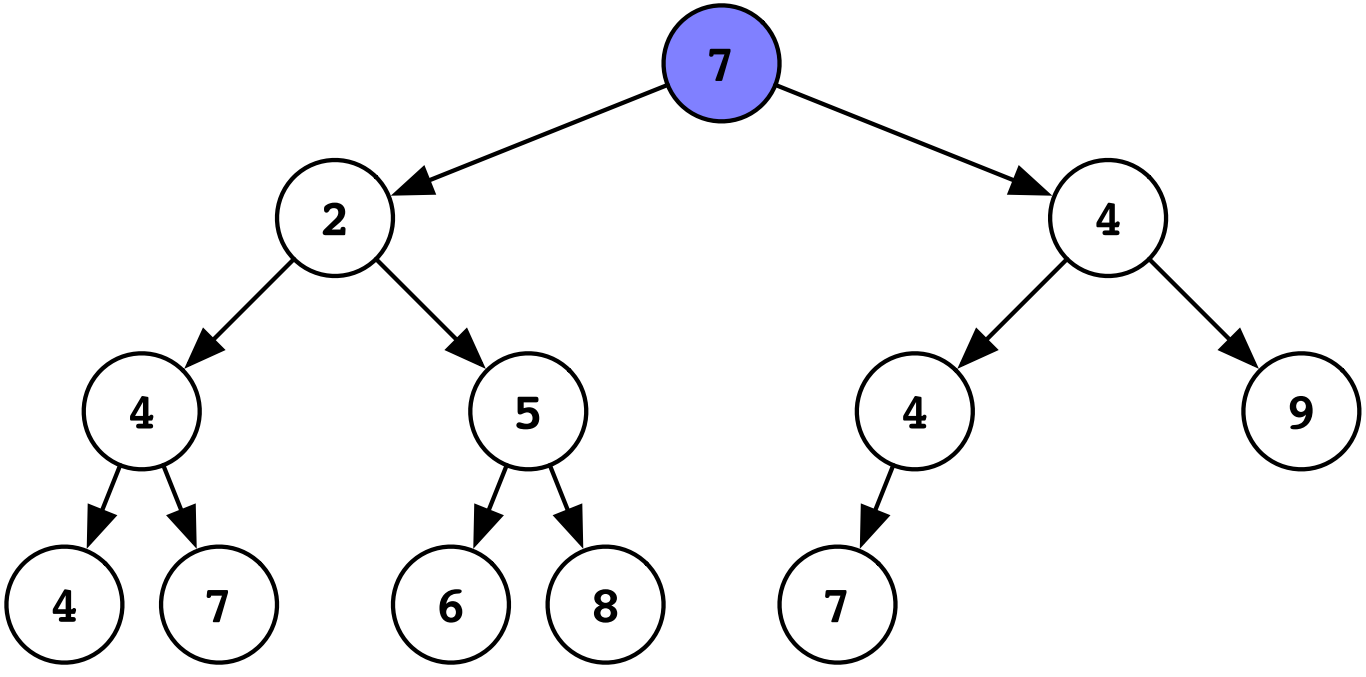
Idea: We can easily rearrange it so it has the right *shape*: a full binary tree except, perhaps, the last level.



Just move the "last" vertex to the root. That is, swap its contents with the root and then pop the back of the list storing the heap.

Popping

Now the heap property is satisfied at every vertex except, perhaps, the root vertex.

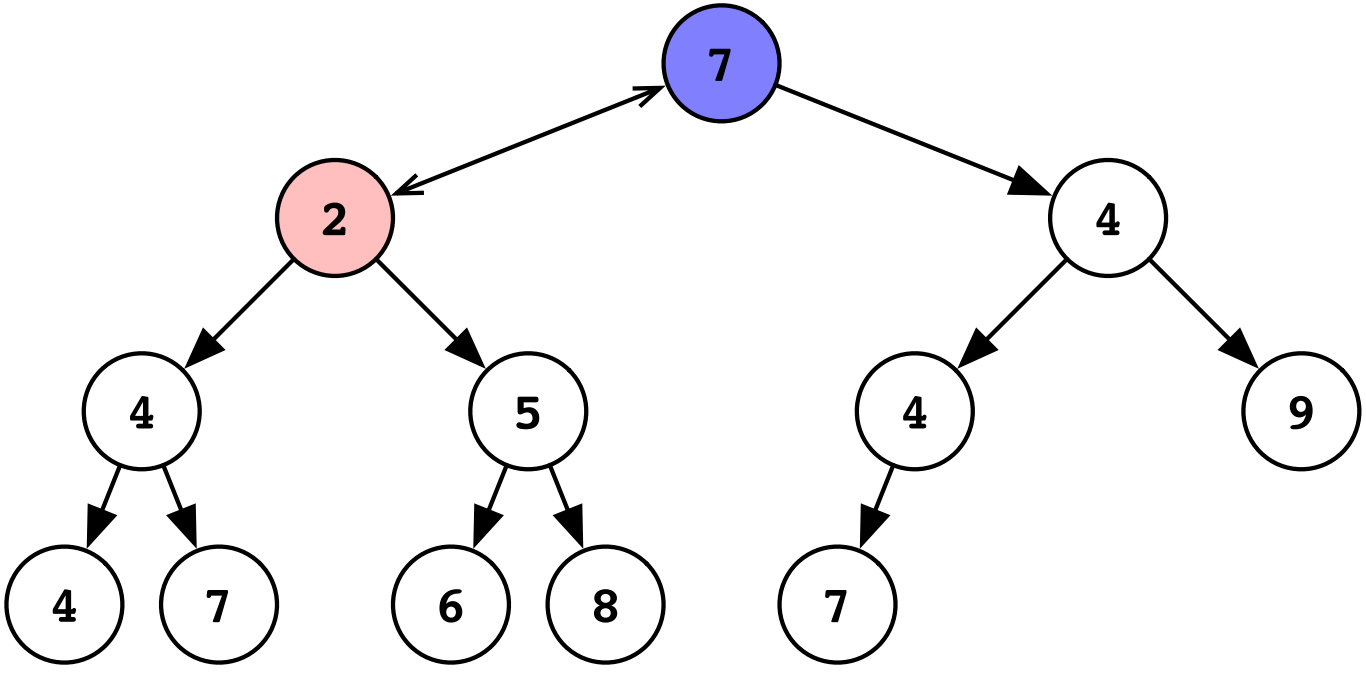


Fix The Heap!

This time, we will move things down the heap.

The Heap Property

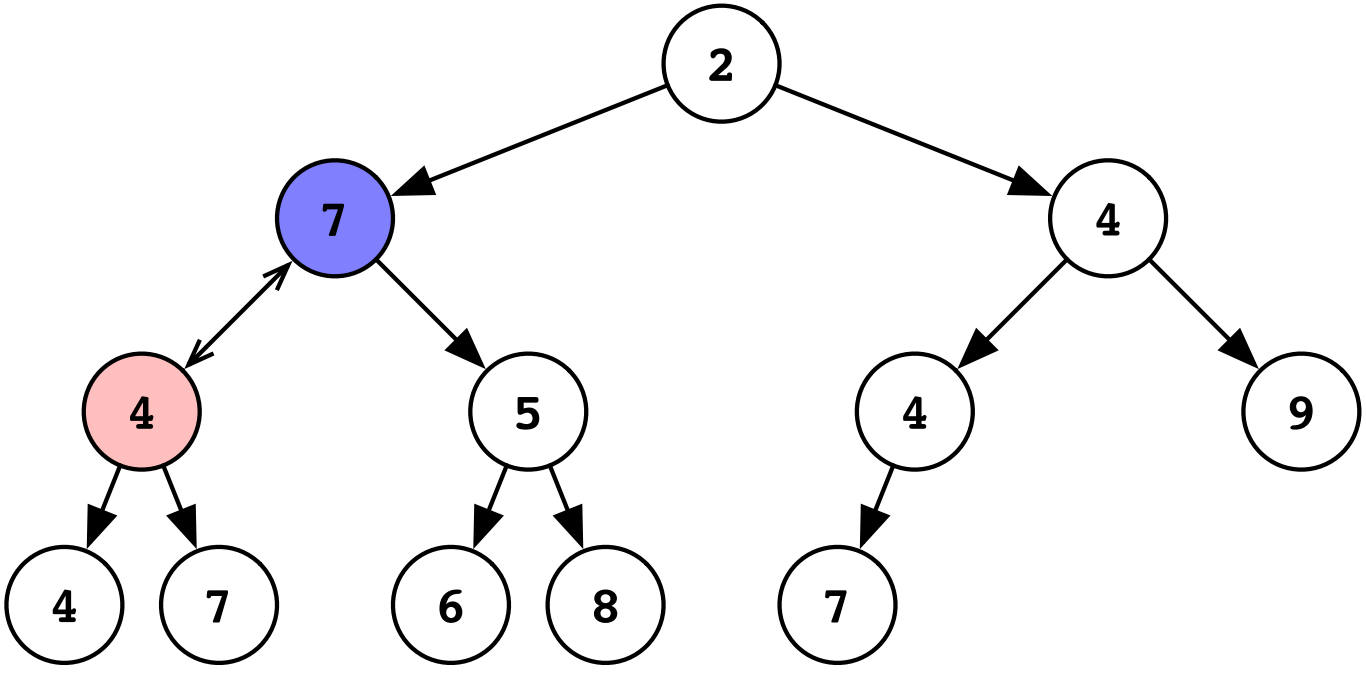
How can we fix the heap property at the root?



If there is a violation, swap with the child having the **smallest** key.
Careful: There may be only one child. Don't index out of the list.

The Heap Property

Picture: After the swap.

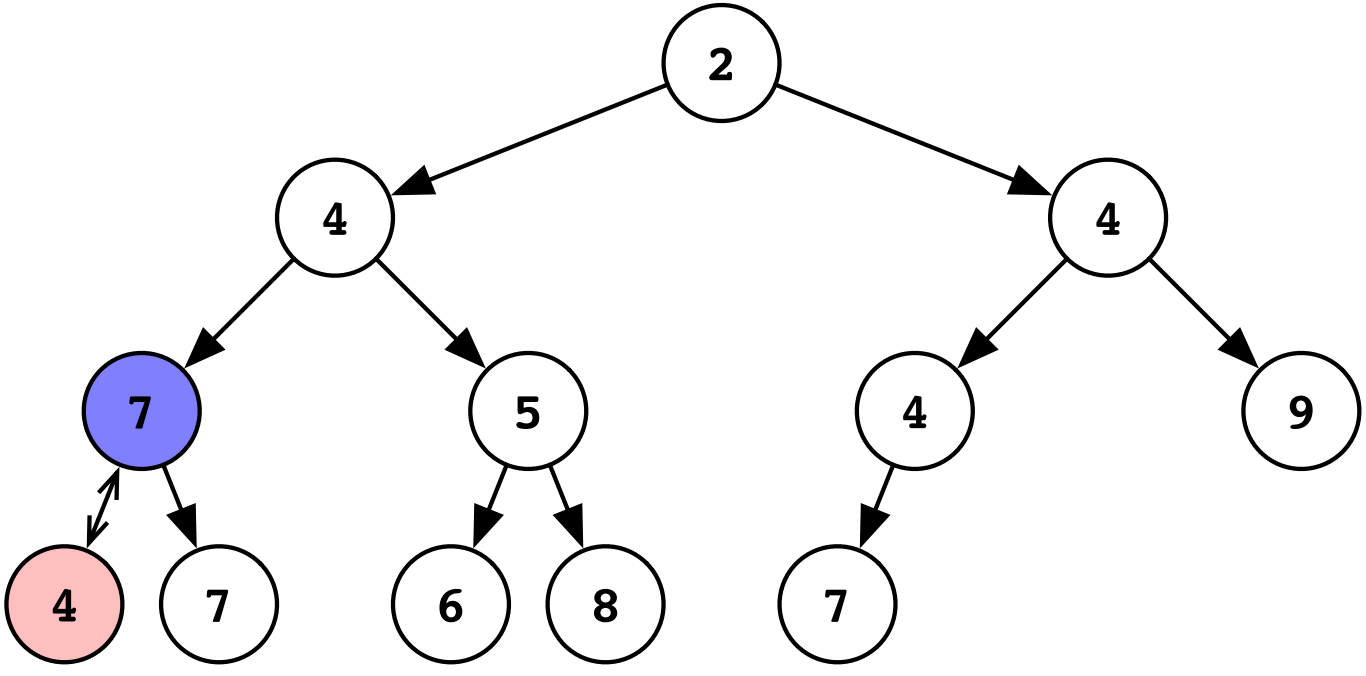


We still have a violation of the heap property with the new position of the item that moved down.

Repeat: swap with the child having the smallest key.

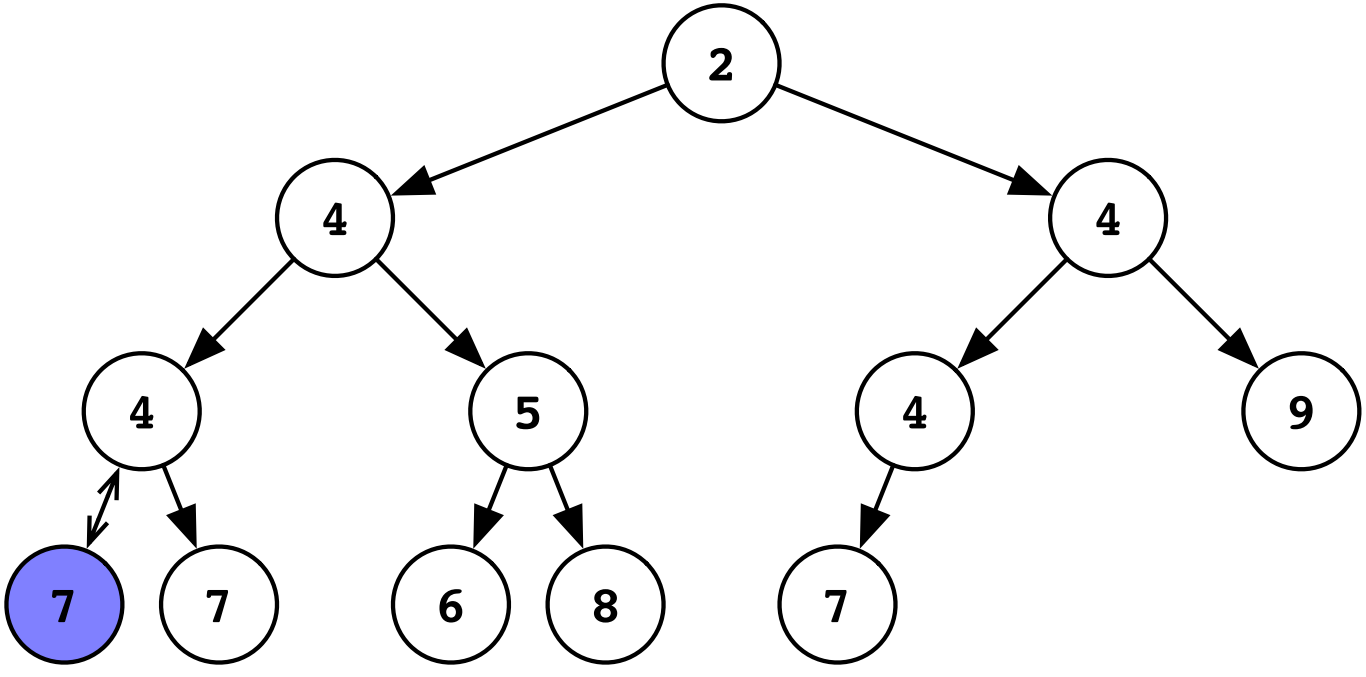
The Heap Property

Picture: Again, after the swap.



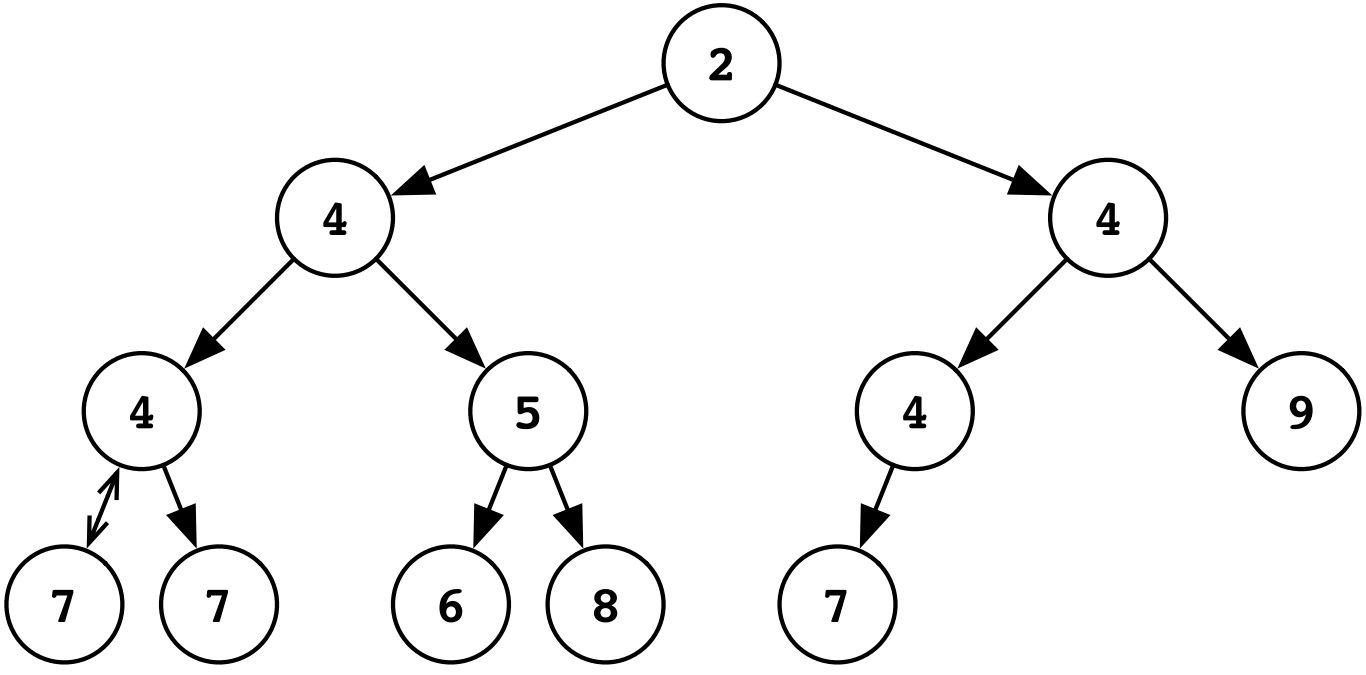
Still a problem at the new location of the item that went down. Swap with the smallest child.

Picture: Yet again, after the swap.



The heap property must hold at this new location for the item that went down. Done!

Picture: The final heap.



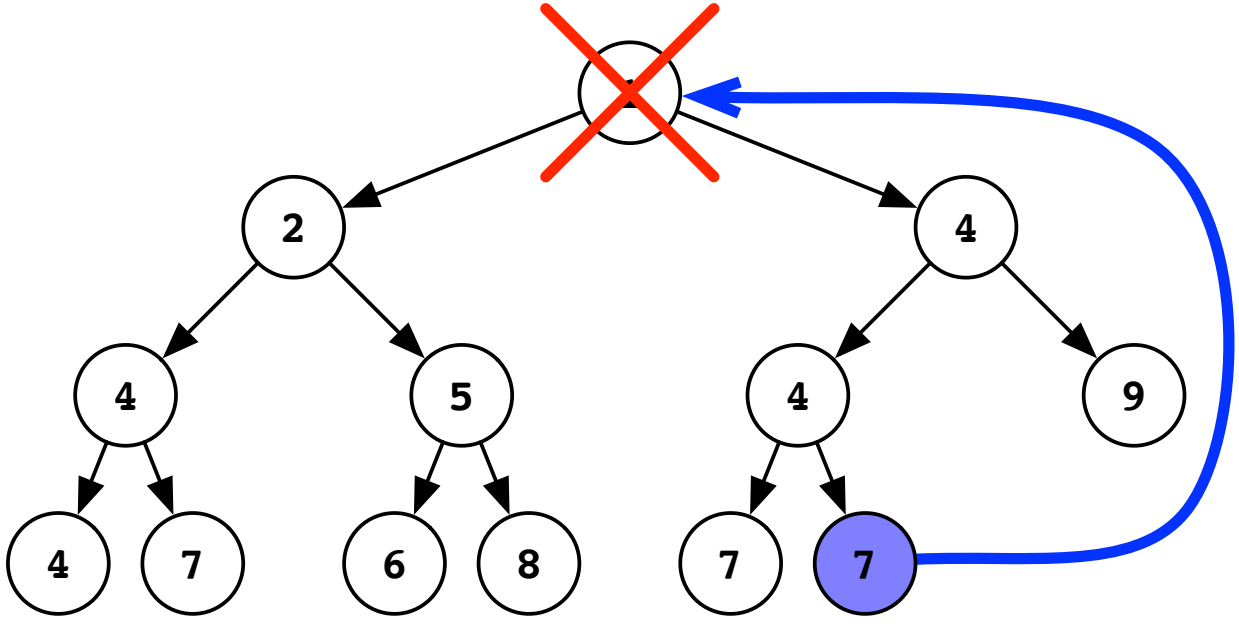
Running Time: $O(\#\text{ layers}) = O(\log n)$.

Verification: Does This Always Work?

Invariant

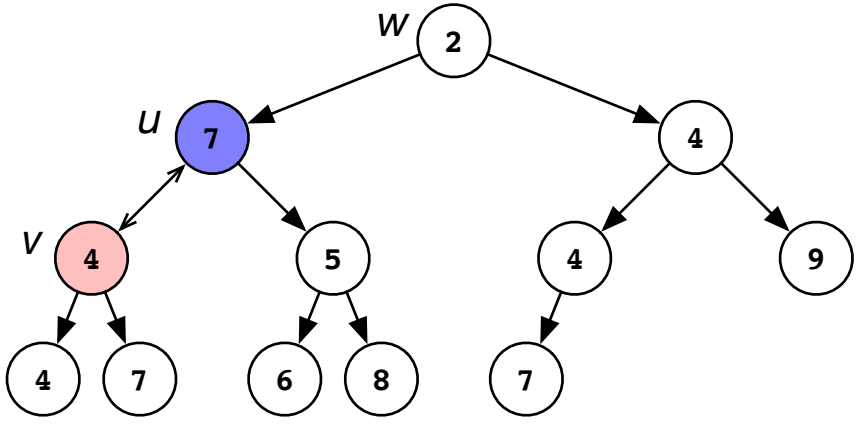
After swapping the item/key at the of the heap to the top, the only vertex that may not have the heap property is the one containing this item/key (the one we are tracking).

Trivially, it is true initially:



Verification: After a Swap

Suppose u is the current vertex with the item/key, v is the smaller-key child, and w is its parent.



- The key of u changes to something that was previously a descendant of w. The heap property continues to hold at w.
- We chose v to ensure the heap property holds at u after the swap.

The only other vertex that had its key or a child's key change is v, the new location of the item we are tracking. **The invariant holds!**

Popping Pseudocode

Suppose the heap is not empty (otherwise `error`!)

Algorithm 2 Popping and returning the minimum-key item.

r_item, *r_key* $\leftarrow$ item/key of the root
swap the items and keys between the root and the last vertex
pop the last item from the list
$u \leftarrow$ root vertex
while the heap property is violated at u **do**
    $v \leftarrow$ child of u with smallest key
    swap the item and key between u and v
    $u \leftarrow v$
return *r_item*, *r_key*

Running Time: $O(\#\text{ layers}) = O(\log n)$.

Other Heap Topics

Heapsort

Think the items to be sorted are keys.

- Insert all keys into the heap (items don't matter, can use `None`).
- Use popmin iteratively until all items are popped out.

Running Time: n insertion, n popmin operations, each taking $O(\log n)$ time. A simple and practical $O(n \log n)$ sorting algorithm!

Building a Heap in $O(n)$ Time

What if all item/key pairs are already in a list? We can turn the list into a heap in $O(n)$ time!

Process items in reverse order (in the list).

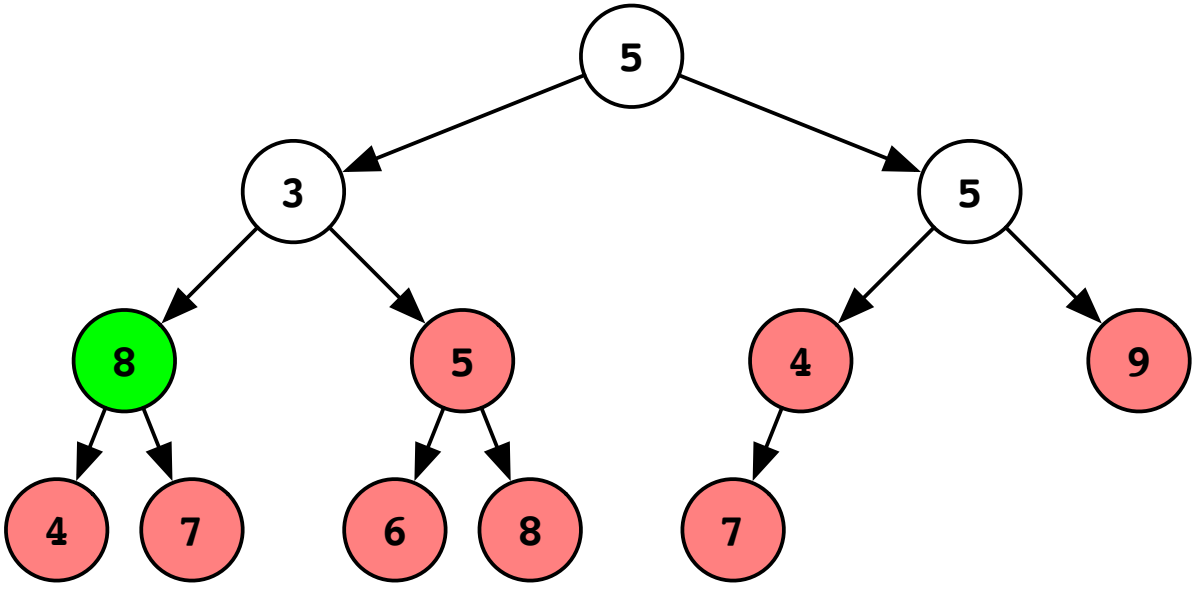
- For each, begin a fix-heap-down operation (like we did with the root in popmin).

Careful checking shows after processing the last i vertices, they all have the heap property.

Building a Heap in $O(n)$ Time

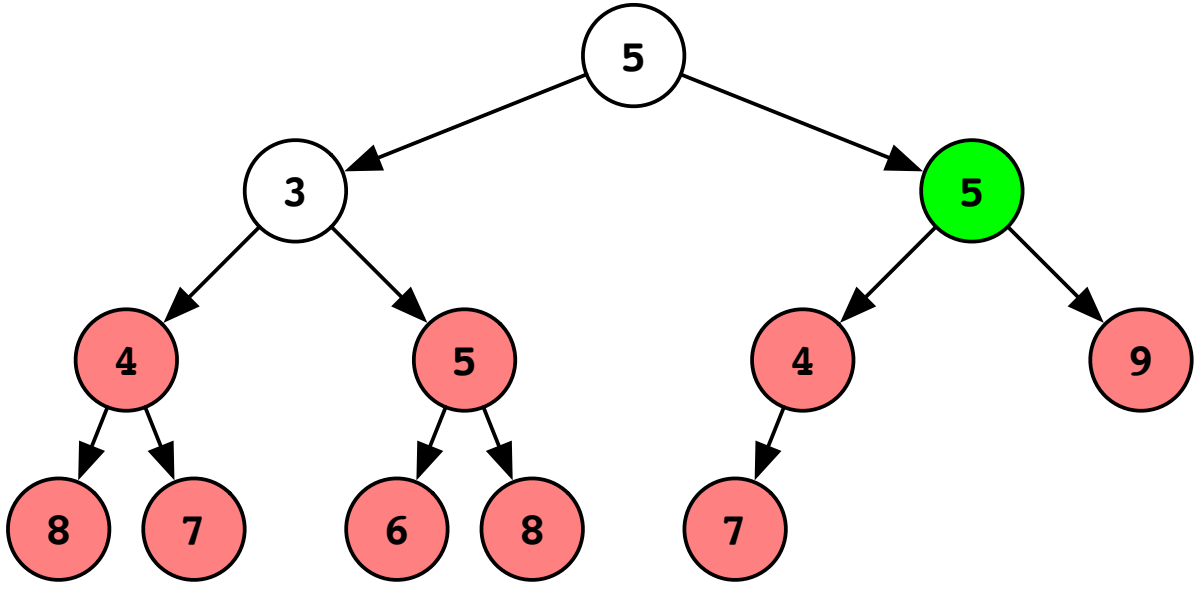
Red vertices have already been processed and have the heap property.

Let's skip ahead in the algorithm until we reach this point. The **green** vertex is the next one to consider.



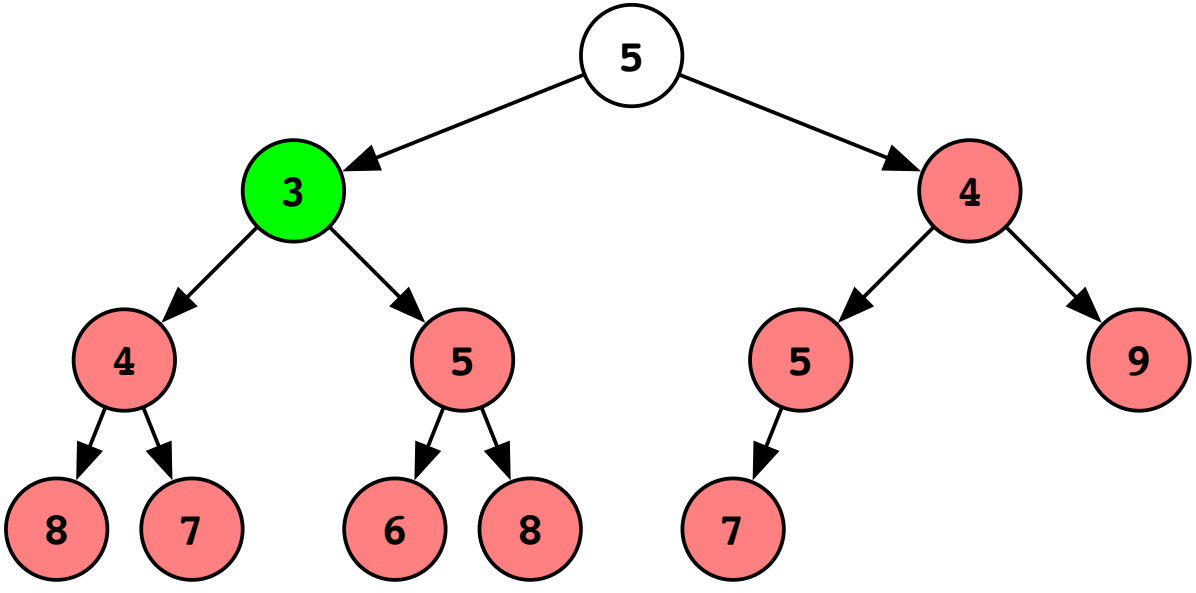
Building a Heap in $O(n)$ Time

Swapping it with the minimum child fixes the heap property. Move on to the next iteration.



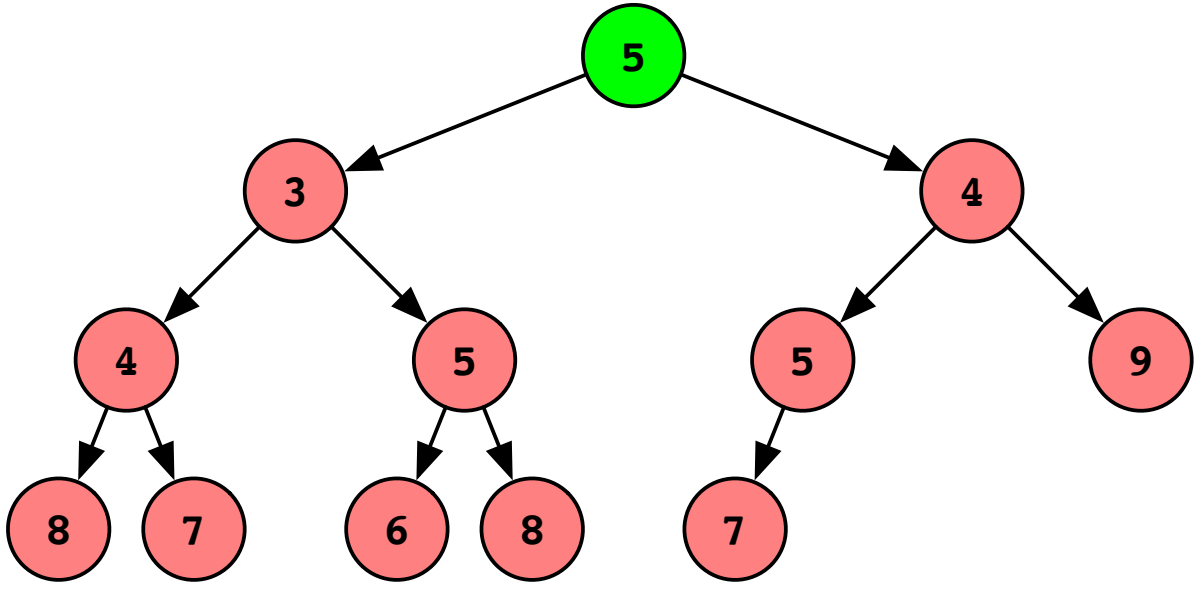
Building a Heap in $O(n)$ Time

Again, swapping it with the minimum child fixes the heap property.
Move on to the next iteration.



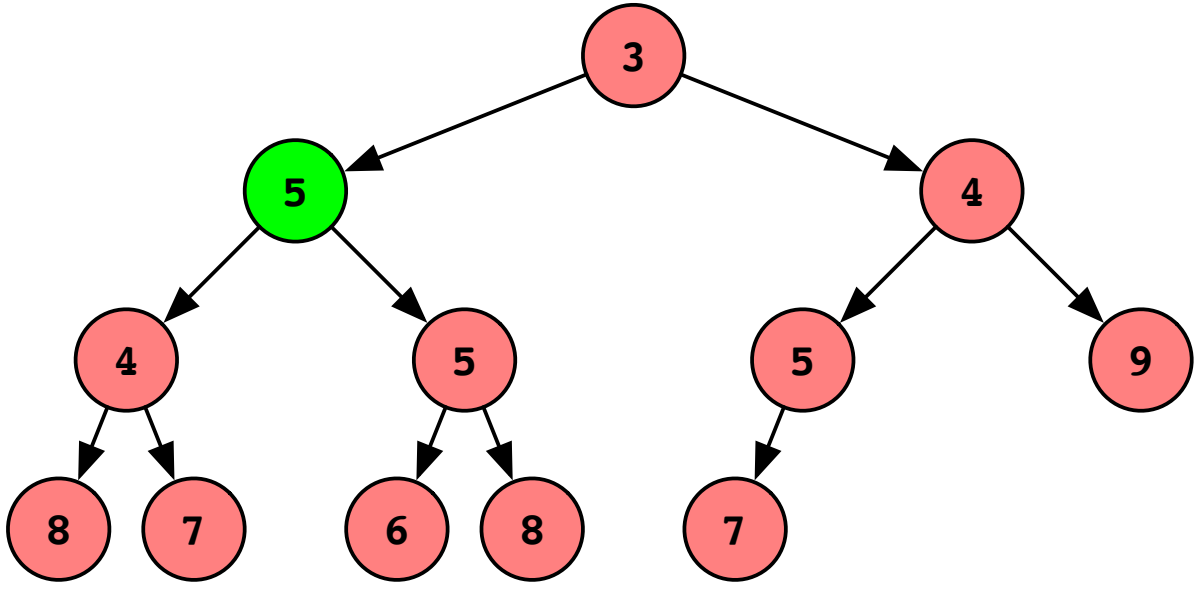
Building a Heap in $O(n)$ Time

The heap property was already satisfied. So move on to consider the root.



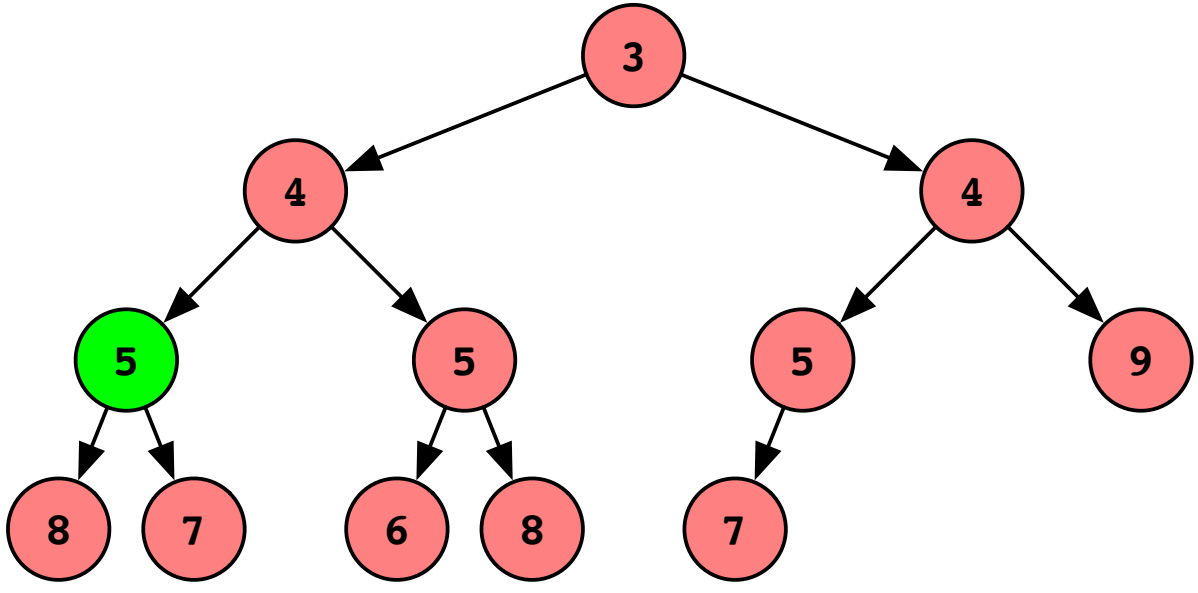
Building a Heap in $O(n)$ Time

We have to fix the heap property. Swap with the minimum-key child and chase it down the tree.



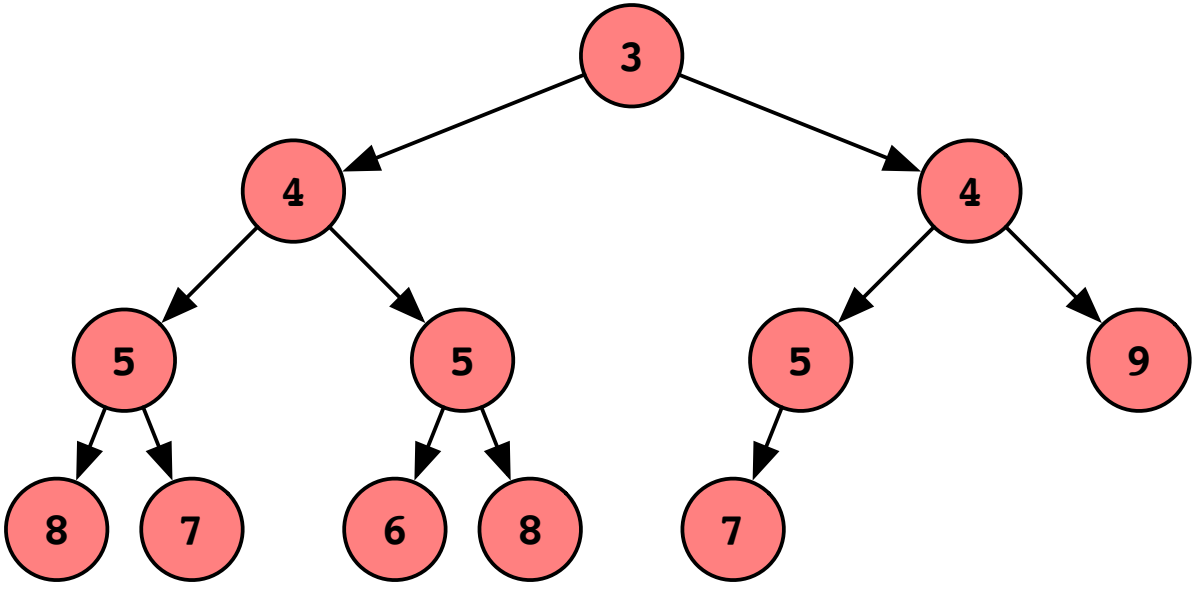
Building a Heap in $O(n)$ Time

We have to fix the heap property. Swap with the minimum-key child and chase it down the tree.



Building a Heap in $O(n)$ Time

Finally done!



Analysis for Building the Heap

Say there are ℓ layers. For each of the at most 2^i items in layer i, the number of iterations for fixing the heap down is $\ell - i$.

Thus, the number of iterations is at most:

$$f(\ell) := \sum_{i=0}^{\ell} 2^i \cdot (\ell - i).$$

How can we determine this value and see it is $O(n)$? With a trick! **Worked out in class**.

$$f(\ell) = 2^{\ell+1} - \ell - 2 \leq 2 \cdot 2^{\ell}.$$

As $\ell \leq \log_2(n+1)$, we have $2^{\ell} \leq (n+1)$ so this is $O(n)$.