7: Other trees

2-3 trees

So you might have noticed a couple of things about binary search trees in the last chapter, both of which have to do with the binary search tree's apparent quick searching ($log_2 n$). First, the search time is only as low as $O(log_2 n)$ if the binary search tree is balanced. If our tree turns out to be very one sided, then our search time is closer to $O(n)$ than it is to $O(log_2 n)$. Secondly, if the search time is $O(log_2 n)$, then maybe we can improve it to $O(log_3 n)$ just by increasing the number of children that a node in the tree is allowed to have...who knows...Maybe even improve it even further than that.

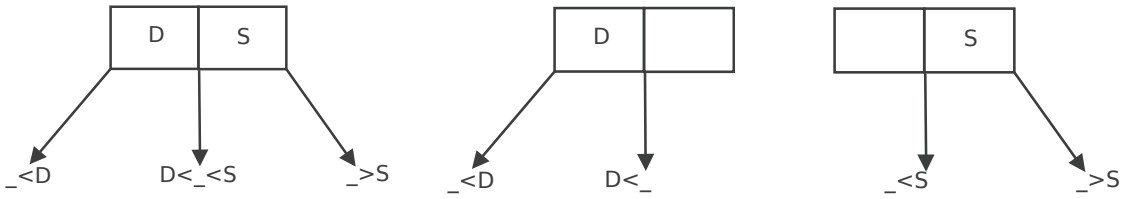
Well there is a data structure that allows that: **2-3 trees** i.e. it allows for a balanced tree where each node is allowed up to three children and improves search time from binary search trees. It turns out that the more perfect you want your life to be i.e. more balanced, with quicker searches, then the more complicated it gets. As a result, we are just going to discuss this data structure at a high level and see how its insertions work out and leave it at that...for now. (dun_dun_duuuun!!!...)

So what is a 2-3 tree?

So far we've said that a 2-3 tree is a balanced tree where each node has up to three children. Well there are a few more rules.

With a binary tree, we have two children per node, and we only have to make one comparison (and decision) in order to determine which subtree to traverse to. Well if we're going to have three subtrees, then we need two comparisons, which in turn means that we need two pieces of data.

A node of a 2-3 tree holds **up to 2 pieces of data**, and its three children represent values **less than**, **in between**, and **greater than** those values.



Insertions are **always** done at the leaves, all leaves are at the same level (hence the balanced part), and all data is stored in sorted order within the nodes.

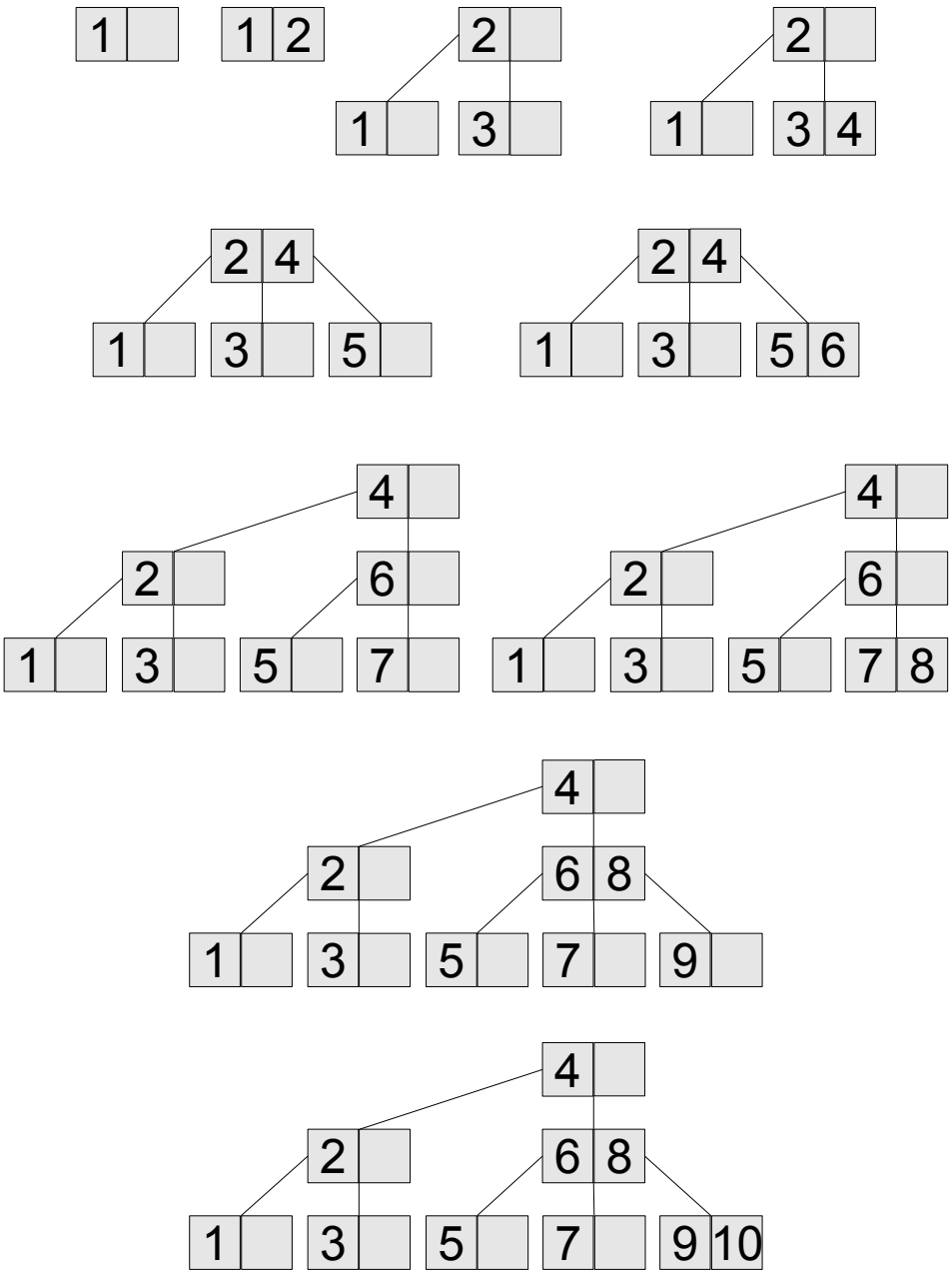
Insertions!

So here are a few rules to remember.

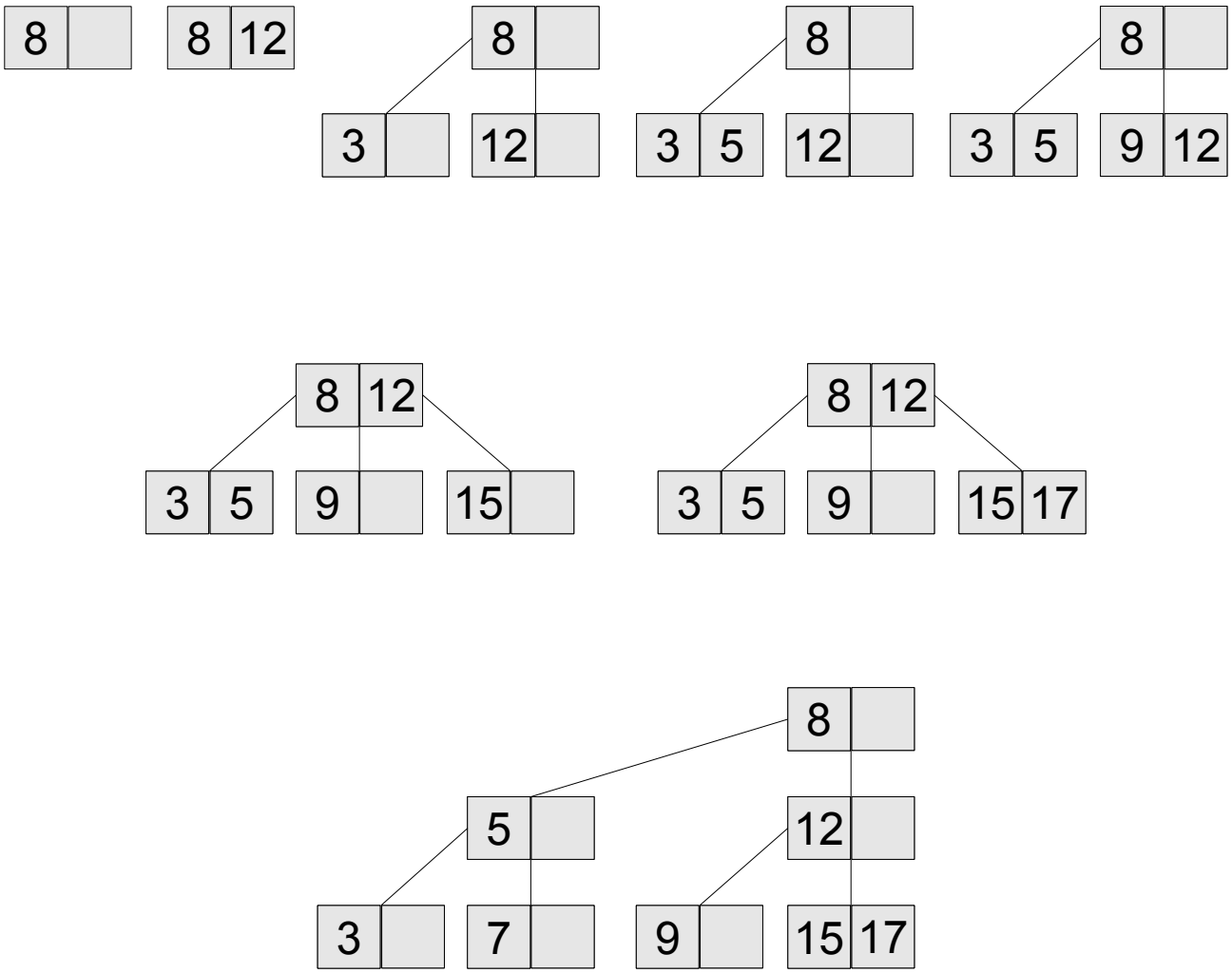
- At the end of the insertion, you should still have a valid 2-3 tree i.e. all leaves at the bottom (or top...depending on how you look at it) of the tree, maximum of two pieces of information in each node, maximum of three subtrees in their proper positions.

- If there is space in an appropriate leaf node (i.e. it only has 1 piece of data) then just insert it there.
- If you have one piece of information in the node, then you can only have 2 subtrees, and if you have 2 pieces of information, then you can have 3 subtrees.
- If you are forced to temporarily push a third piece of data into a node, that node splits up into two nodes with the two extreme values in them, and the middle value is promoted to the parent node.

Let's try inserting the sequence *1, 2, 3, 4, 5, 6, 7, 8, 9, 10.*



Let's try another sequence to drive this point home. *8, 12,3, 5, 9, 15, 17, 7*



General trees

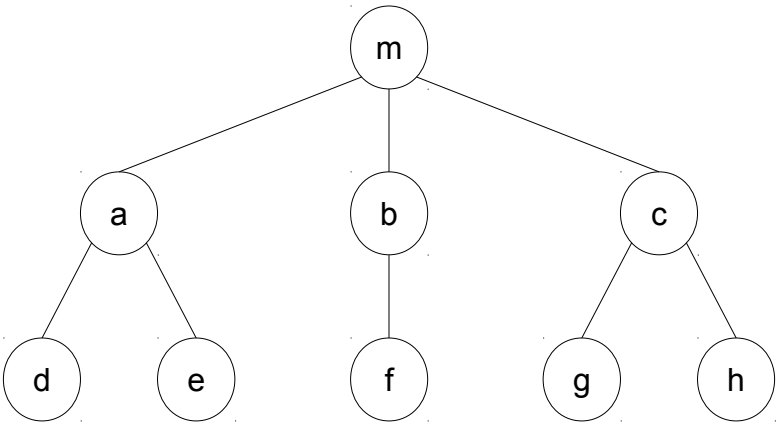
In general, trees consist of nodes and branches without any restrictions to the number of children that each parent is allowed to have.

There are cases in which we will have to convert an arbitrary tree into a binary tree and there are some rules that can help us come up with a standard binary tree representation of an arbitrary tree.

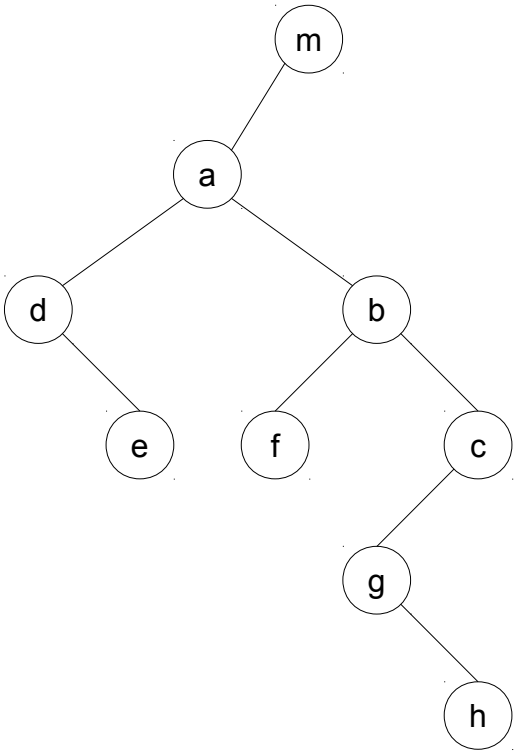
These rules are simple

- root stays the same
- left-most child remains as the left child
- remaining children are cascaded as the right children of the left child

Using the rules above we can convert this tree:



into this:



Heaps

Heaps are another kind of tree in which the keys of the nodes matter. In a max heap, the key of a node is either greater than or equal to the keys of its children. Heaps can also be binary (binary heaps) and it is this kind that we are going to be discussing.

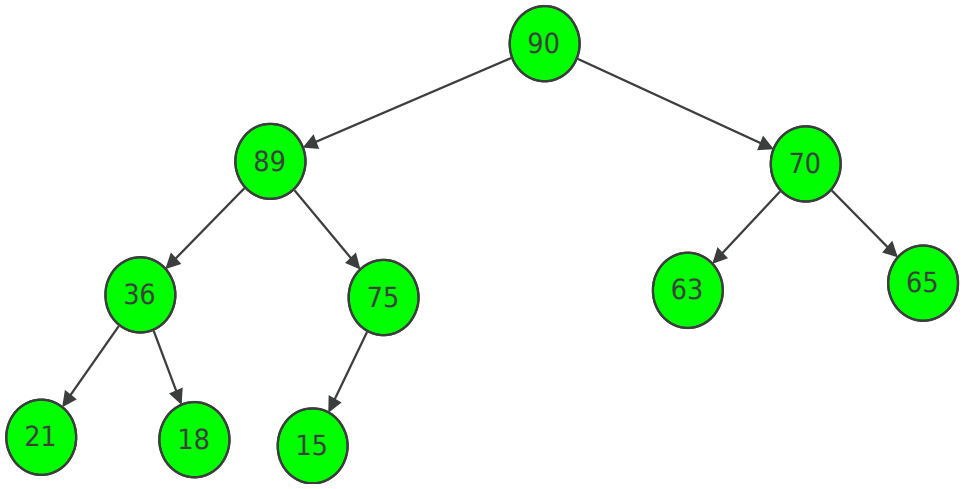
A natural consequence of the key arrangement in a max heap is that the node with the largest key

(or the maximum value) is always the root.

Another feature of a heap is that it is always balanced. This means that the leaves are always at the bottom two levels. Additionally, insertion occurs from left to right at the leaves.

Below is an example of a max heap. You'll notice two properties (that we've already alluded to) .

- Every non leaf has exactly two children (hence the balanced shape). There are cases where you will have only one node on the bottom level having one child.
- Every node has a value at least as large as its children



Another difference from the binary search tree is that there is no order between siblings i.e. right child is not necessarily larger than the left child. Sometimes a larger node will be on a lower level than a smaller node. This can happen if the larger node is not a direct descendant of the smaller node.

A consequence of all these properties is that every node is the root of its own sub-heap.

Representation

It should be obvious that a heap can be represented using the data and two link struct that we used for binary trees. However, the shape of a heap is so regular that it can also be represented as an array. By regular, we mean that every 5 node heap will have the same shape, every 10 node heap will have the same shape, and so on and so forth.

The array below is a representation of the heap above.

90	89	70	36	75	63	65	21	18	15

If the array begins at index *1*, then a node is at position *n*, and its children are at *2n* (left) and *2n+1* (right).

The parents of any node are at *n/2* (with the remainder discarded).

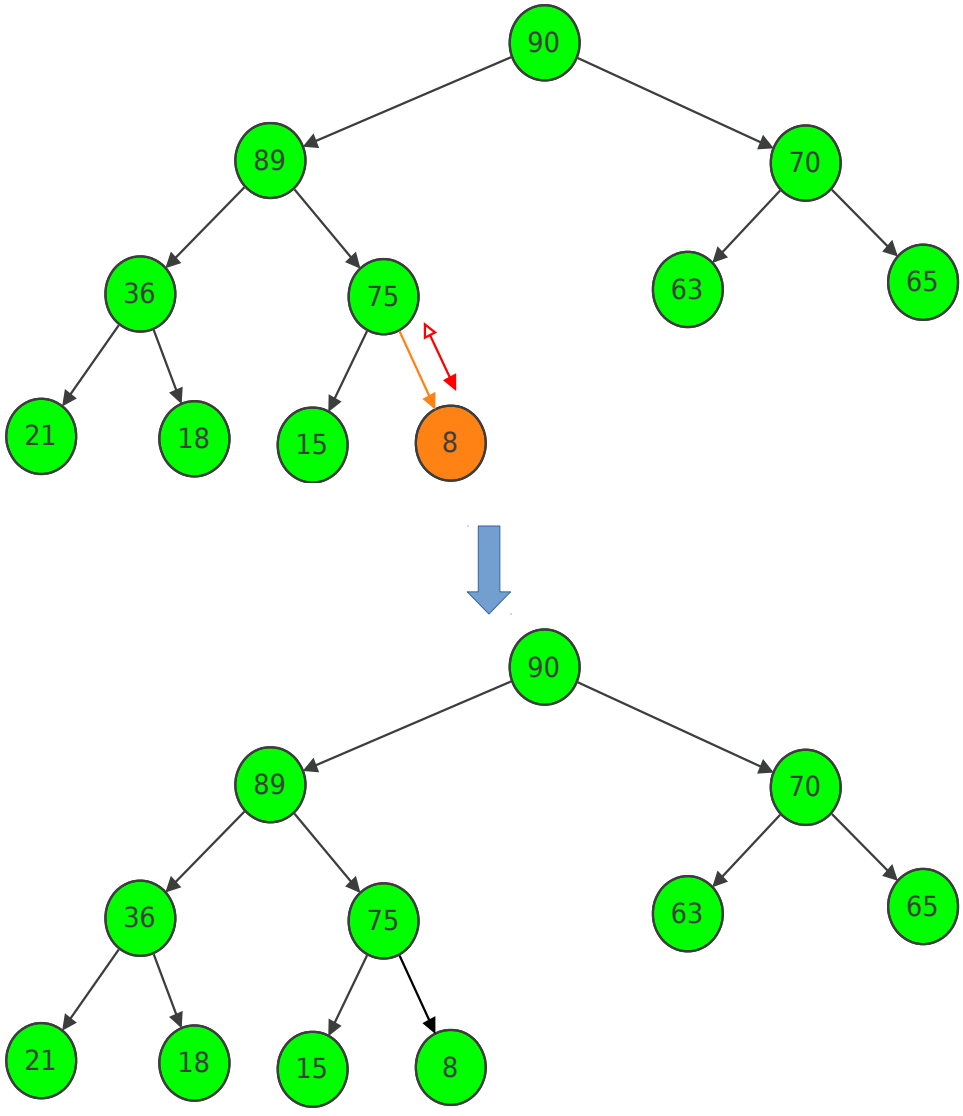
Finding max node

This is the easiest operation in a max heap. The maximum element is in the root. *O(1)*.

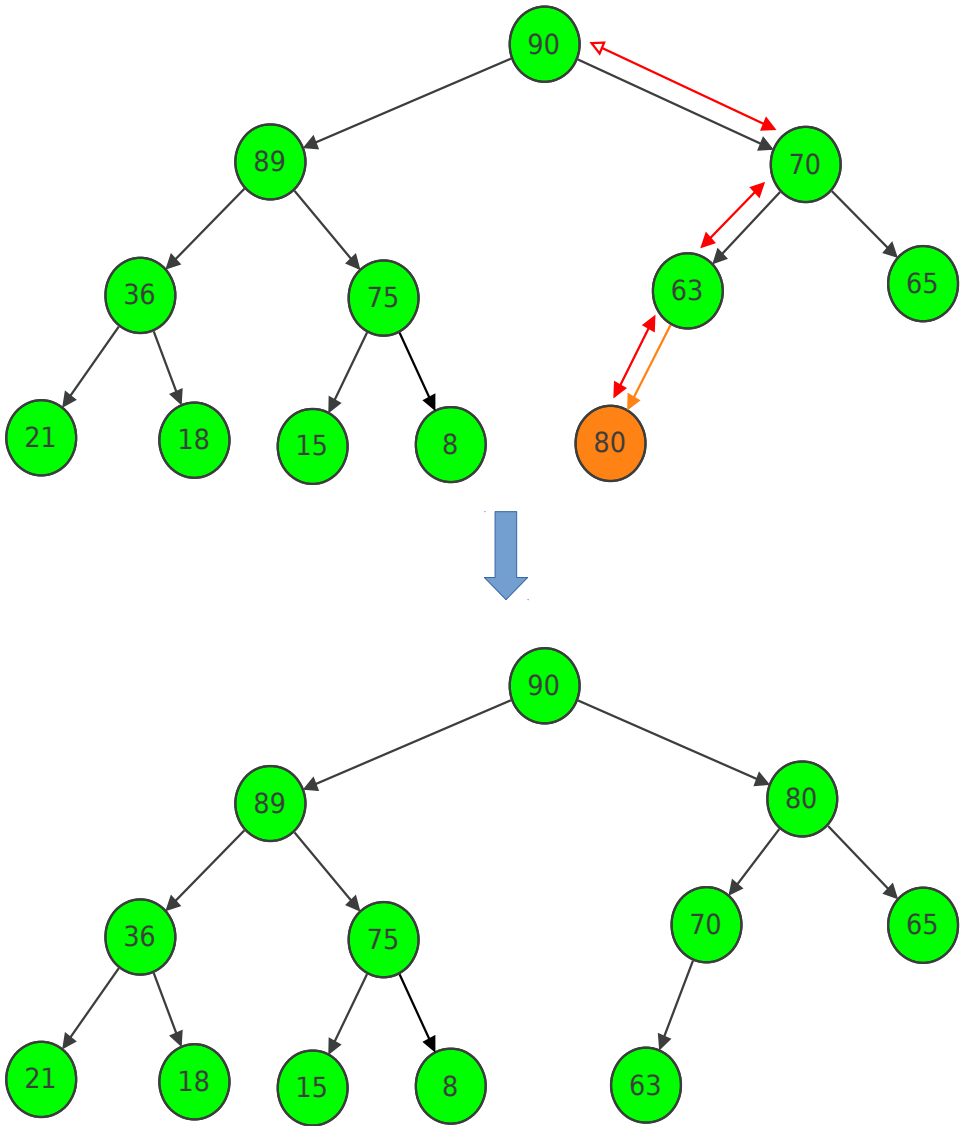
Insertion

This is also another relatively easy operation. All insertions happen at the lowest level from left to right. The inserted node then "sifts up" the tree by swapping it with its parent if the parent is less than this new node.

Consider the tree above. Suppose we wanted to insert an *8* to the heap above. Its simply a matter of including it in the next leaf position. After comparing it with its new parent, we decide whether to promote it to parent (and swap it with that parent). 8 is less than 75 so no swaps are necessary and insertion is complete.

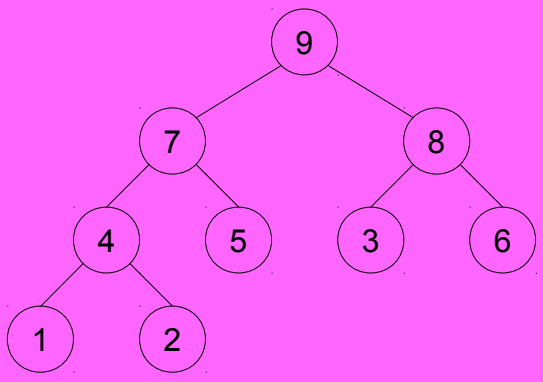


That was an easy insertion. What about if we wanted to insert *80* into the heap above?



Let's see if you can come up with the heap after inserting the following values in the order in which they are given.

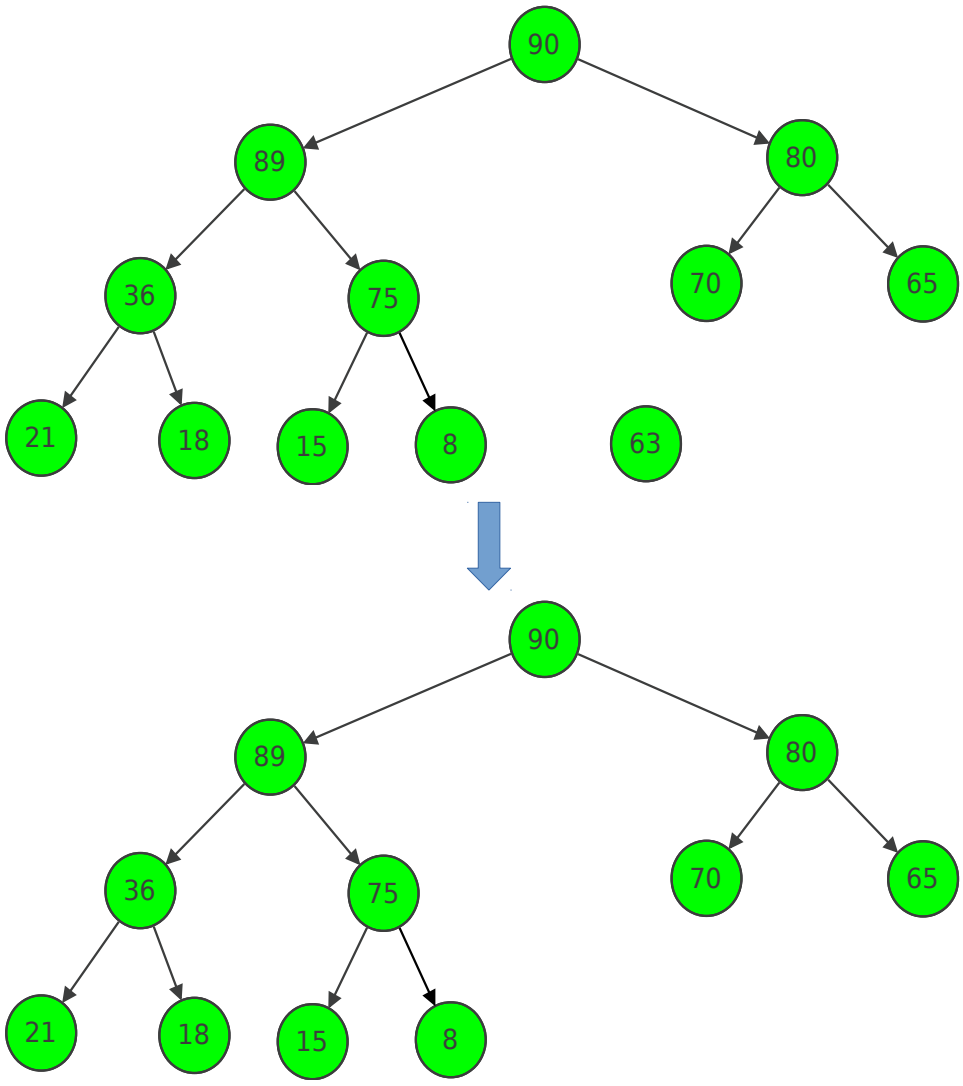
1, 3, 5, 7, 9, 8, 6, 4, 2.



Deletion

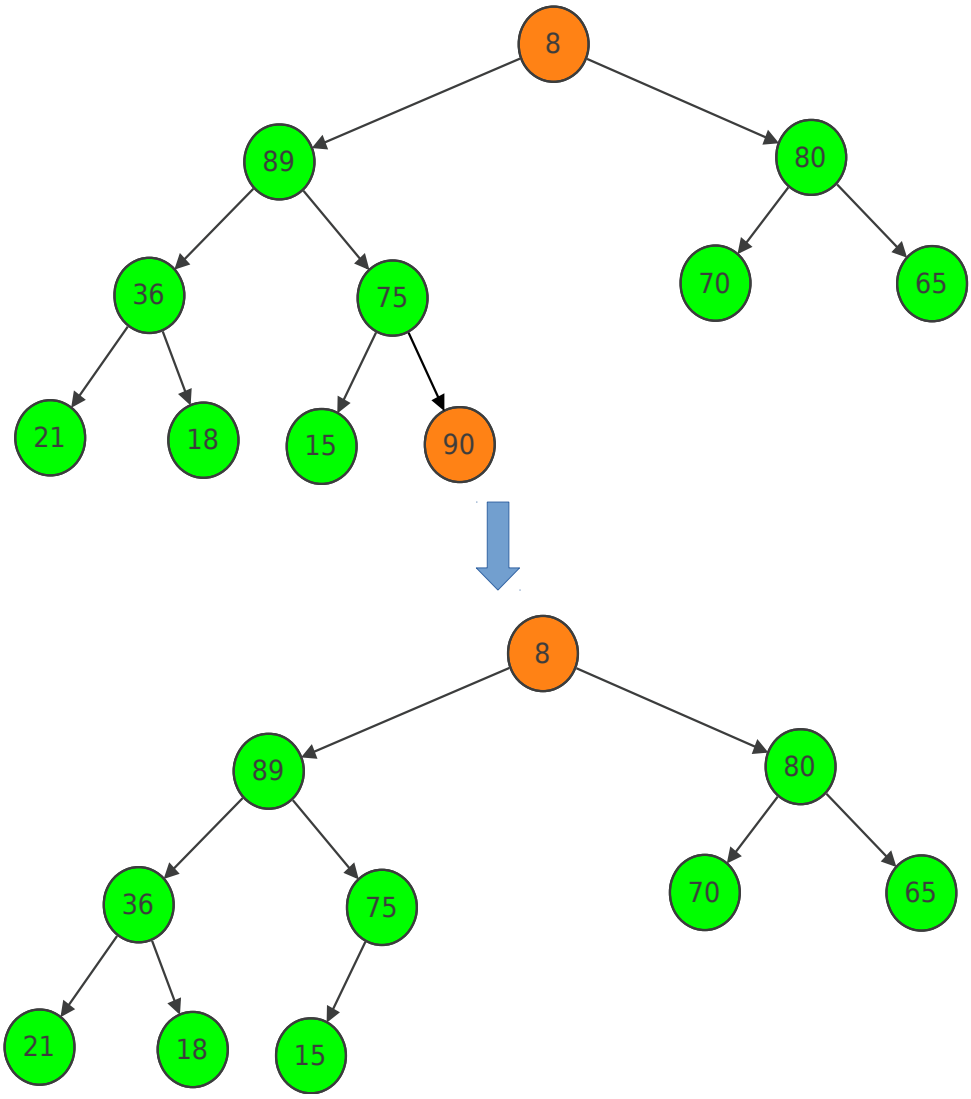
One of the main reasons for using a max heap is that it allows us to keep track of the maximum element very easily. In fact, max heaps are normally used to implement priority queues. That way, the item with the highest priority will always be at the root. Anyway, as a result, the most common deletion that a max heap goes through is deleting the root.

Before we discuss deleting a root, let's just delete the last leaf of the heap.



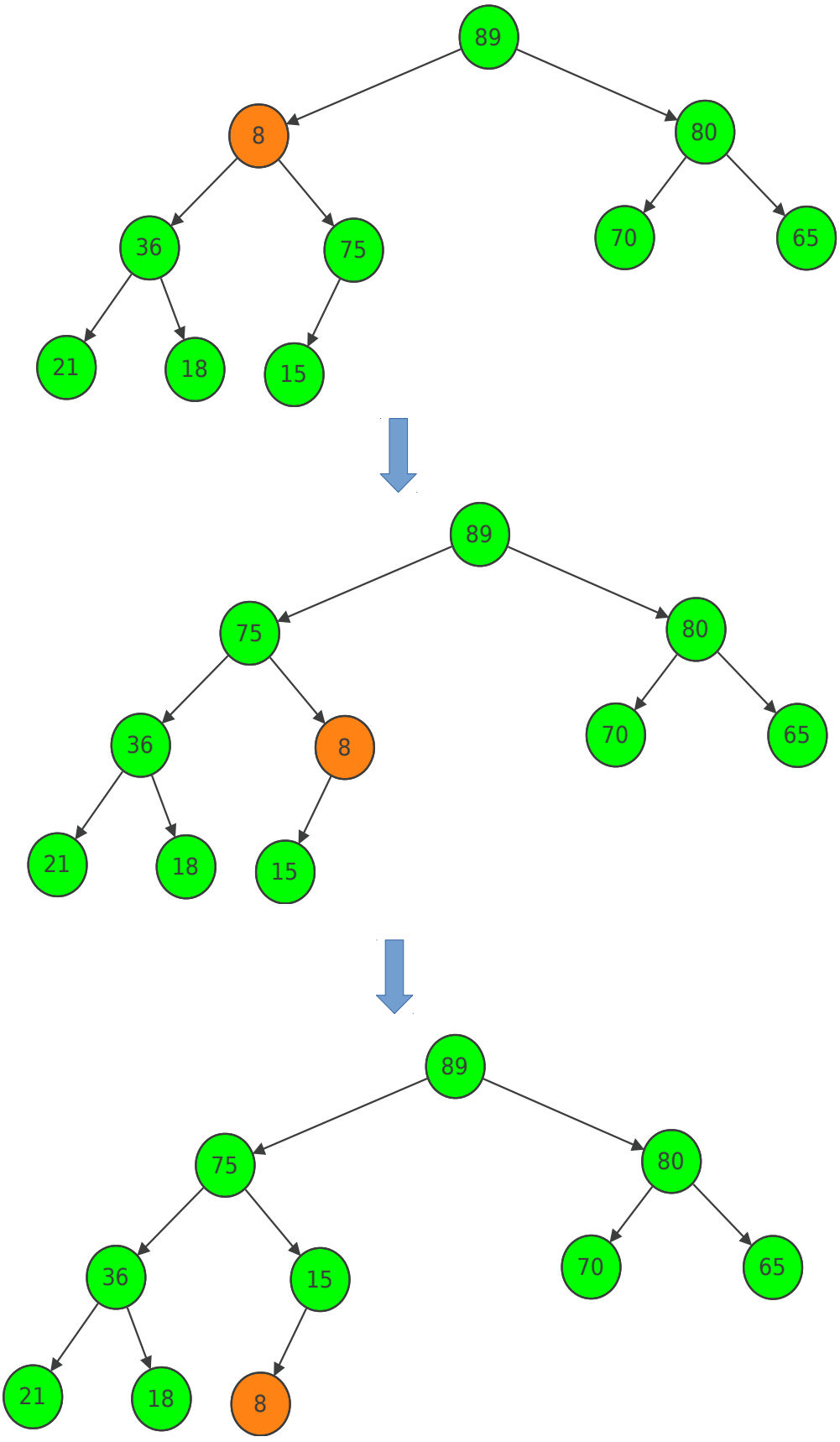
Deleting the last leaf is just a matter of removing it. (or the link to it). You'll notice that the heap still retains all the properties that make it a heap when this happens and therefore we can say we've successfully deleted that node.

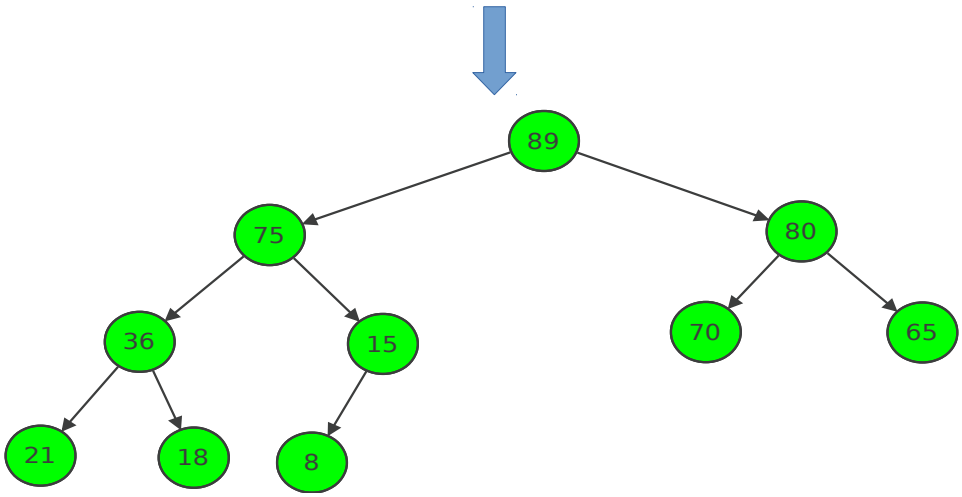
So what about deleting the root? Well we start by swapping the root with the last leaf, then deleting that last leaf (which we've just seen is a very easy process).



Notice that other than our new root, the left and right sub-heaps are still valid heaps.

The next part is a process that is referred to as **max-heapify** (yes...computer scientists sometimes run out of names) or re-heap in which we'll move *8* (the out of place root) down to its proper position. This involves comparing that node to its two children and swapping it with the larger child until it gets to a position where it is larger than its children.


89
8
80
36
75
70
65
21
18
15
89
75
80
36
8
70
65
21
18
15
89
75
80
36
15
70
65
21
18
8



Now let's see if you can delete the 9 from the heap you created in the last activity.