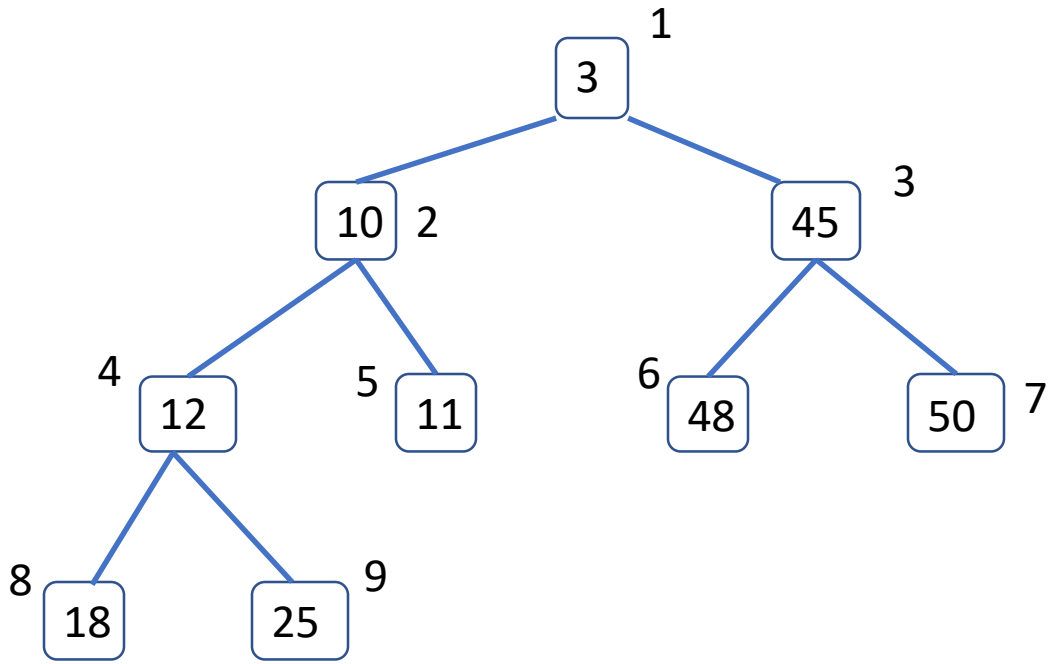
Priority Queue Algorithms

We implement priority queues in Binary Heaps stored in arrays. The Heap Property is that the value at each node is less than or equal to the values of its children.

A heap with n values will be stored in indices 1 through n of the array.

If a value is at index j of the array, its parent is at index j/2 (integer division; ignore any remainder. E.g. 7/2 is 3) and its children are at indices 2*j and 2*j+1.

Here is a typical heap and its array



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	3	**10**	**45**	**12**	**11**	**48**	**50**	**18**	**25**	

The peek method returns the smallest value in the priority queue, which is always at the root. If the queue is empty throw an exception; otherwise just return the data at index 1 of the array.

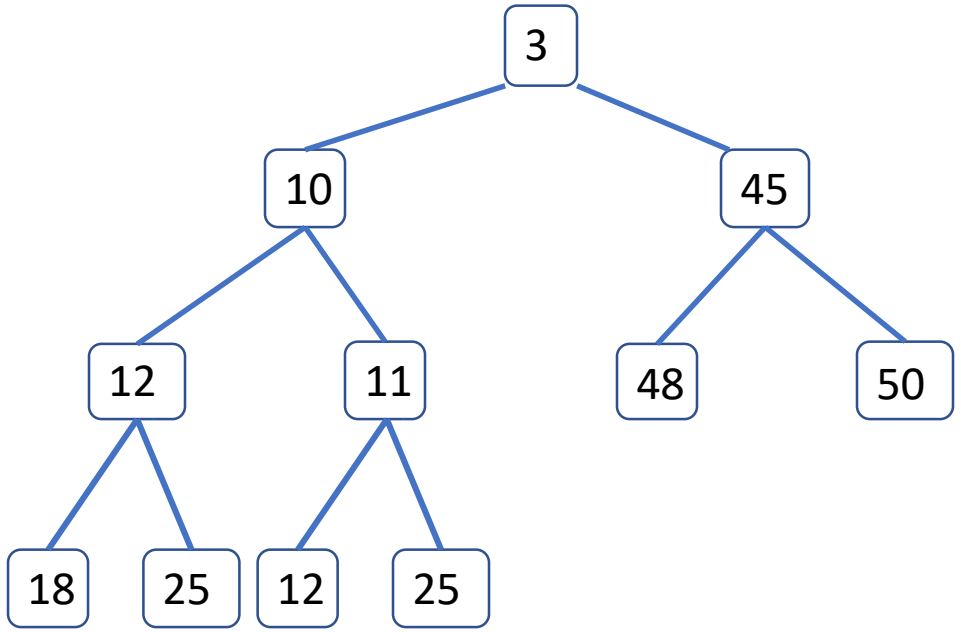
The offer(x) method of the priority queue inserts x into the heap and then adjusts it so that every node again satisfies the heap property.

We do this in two steps:

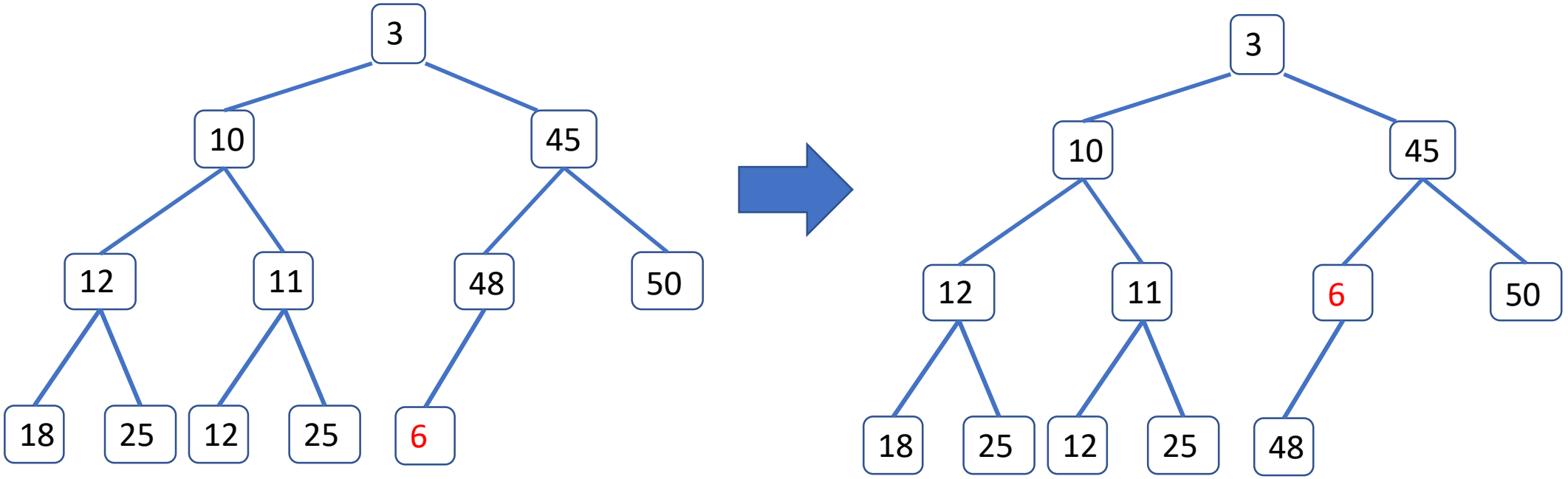
a) Add x to the array in the only location we can: if the heap has n elements they must be at indices 1…n and the new element goes at index n+1 and we increase the *size* counter of the heap.
b) "Percolate" the new element up to its proper location. Let variable *hole* be the index of the new element: initially n+1. Keep comparing the value at index *hole* with the value of its parent, which is at *hole/2*. Flip these two elements if they are in the wrong order and change *hole* to *hole/2*. Stop when the value at *hole* Is at least as large as its parent.

The only tricky thing about percolateUp( ) is making sure you stop if *hole* gets to 1. You can either test for this in your loop or make it occur naturally by copying the inserted data value into index 0 of the array.

So do it: add value 6 to this heap:

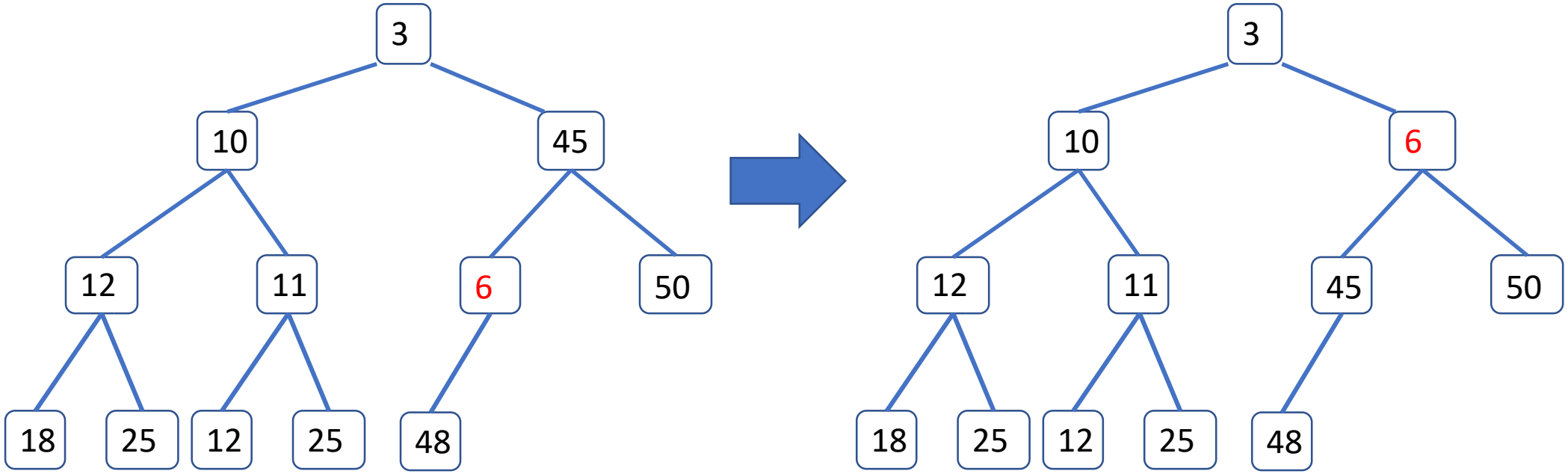


Answer:



Continued next slide

Answer:

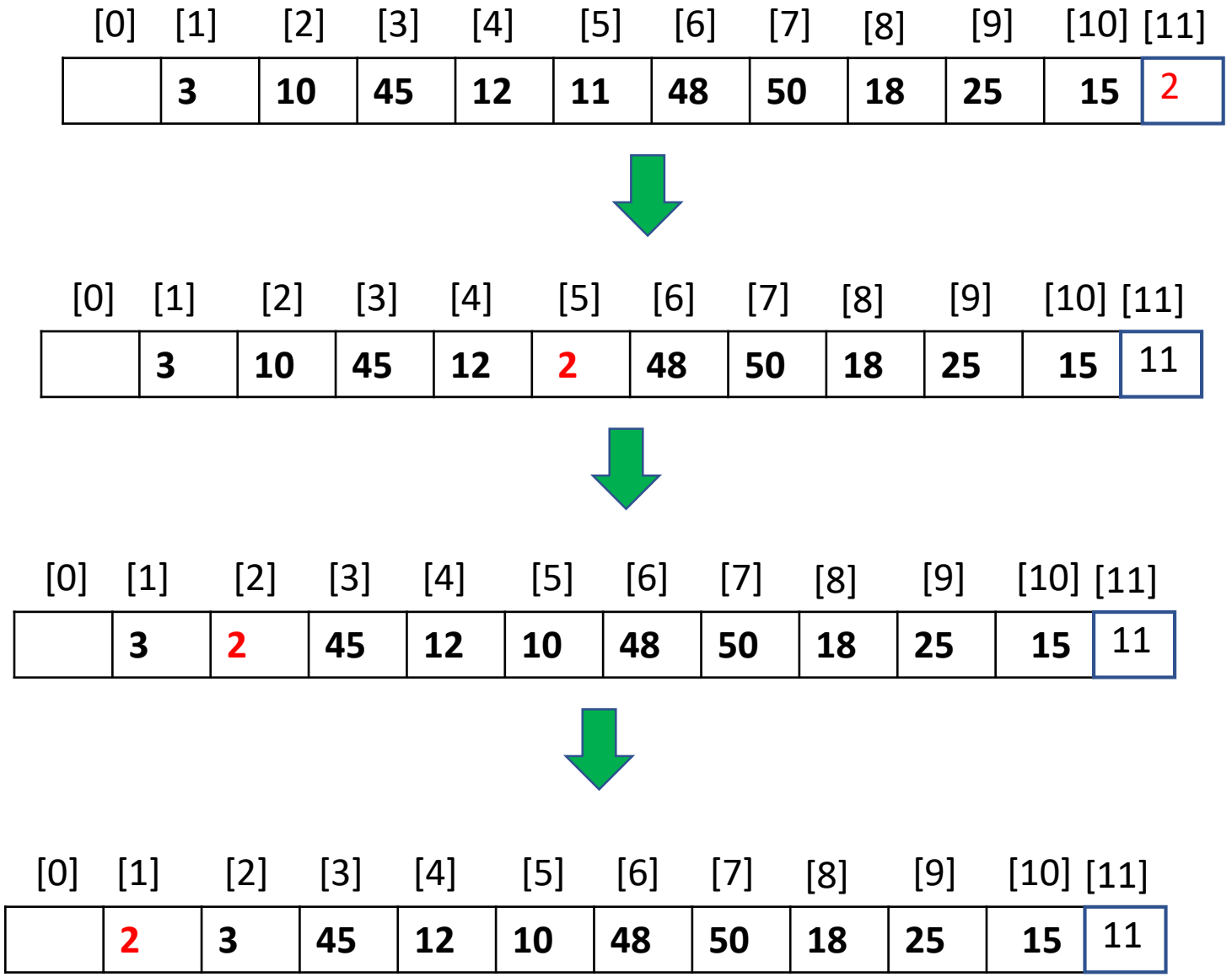


6 is greater than its parent value, so we are done.

Let’s do another one, this time working with the array. Insert 2 into this heap:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
	3	**10**	**45**	**12**	**11**	**48**	**50**	**18**	**25**	**15**	

Solution:

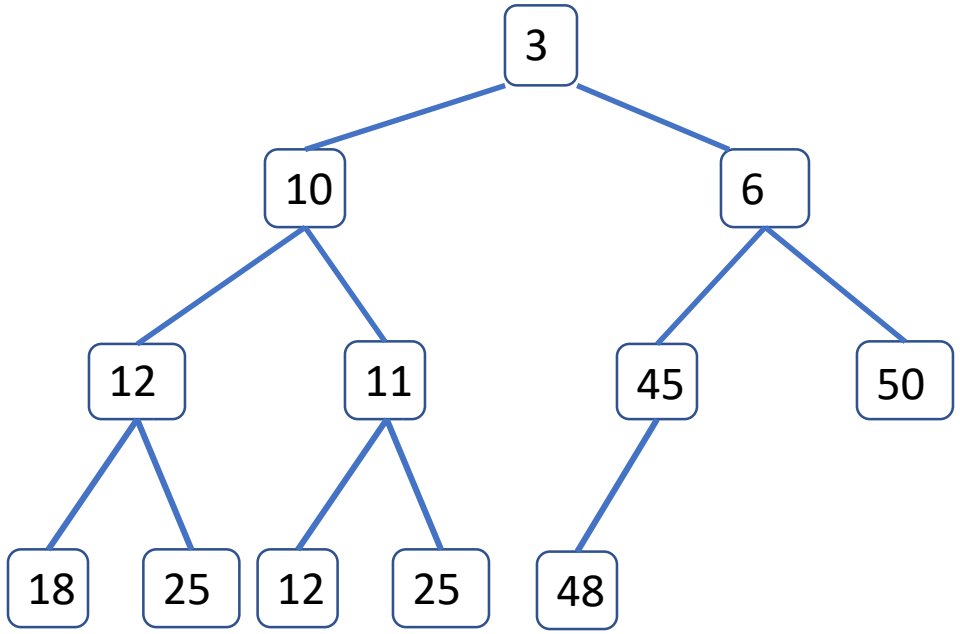

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
3 10 45 12 11 48 50 18 25 15 2
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
3 10 45 12 2 48 50 18 25 15 11
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
3 2 45 12 10 48 50 18 25 15 11
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]
2 3 45 12 10 48 50 18 25 15 11

The poll( ) operation removes and returns the smallest value in the priority queue. The smallest value, of course, is at the root. If our queue contains n values they are at indexes 1…n of the heap. The only node of this tree that we can remove is the one at index n. So we save the element at index 1, move the data at index n to index 1, and decrement the *size* of the heap, reducing the size to n-1. Then we percolate down the value at the root until all of the nodes satisfy the heap property.

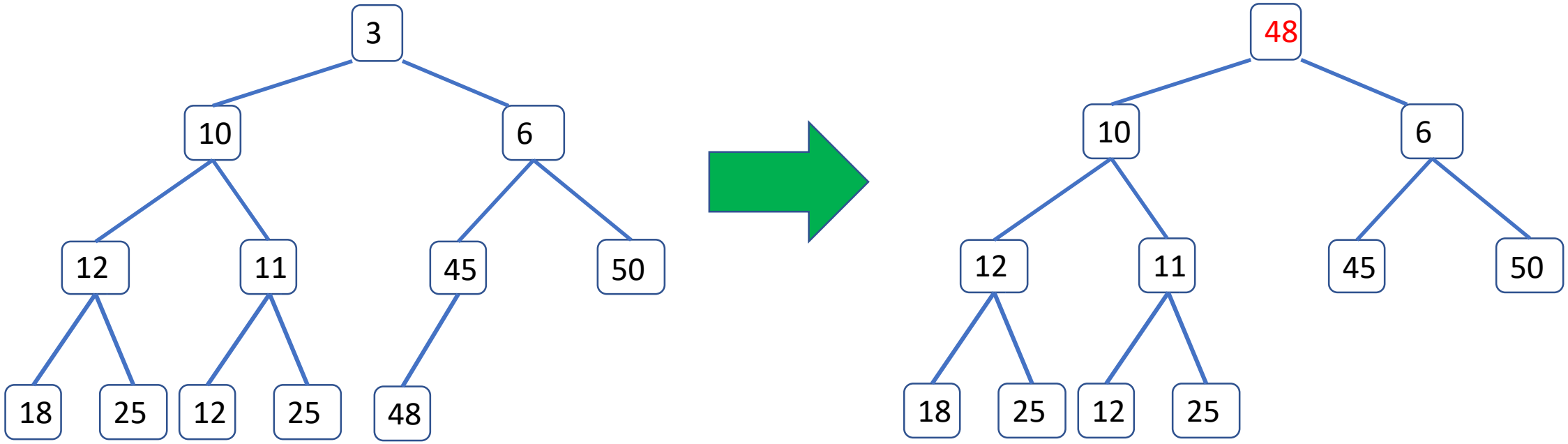
(Continued next slide)

To percolate down, again we make *hole* be the index of the element that is out of place, which is initially index 1. If hole has two children (i.e., 2*hole+1 <= size, which is the same as 2*hole<size); if the value at hole is greater than the smaller of its two children we flip them and make hole be the index of the smaller child, then repeat the test. If the hole has one child (i.e., 2*hole==size) we compare the value of the hole to its child and flip them if the value at the hole is larger. If the hole has no children (2*hole > size) we are done.

So do it: remove the smallest value from this heap:

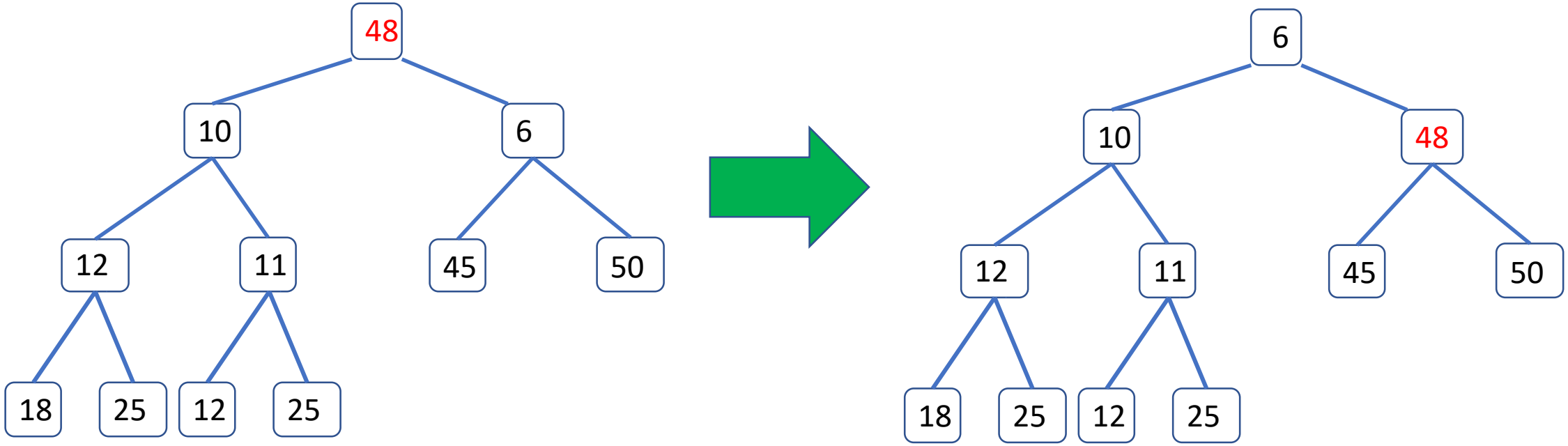


Solution:



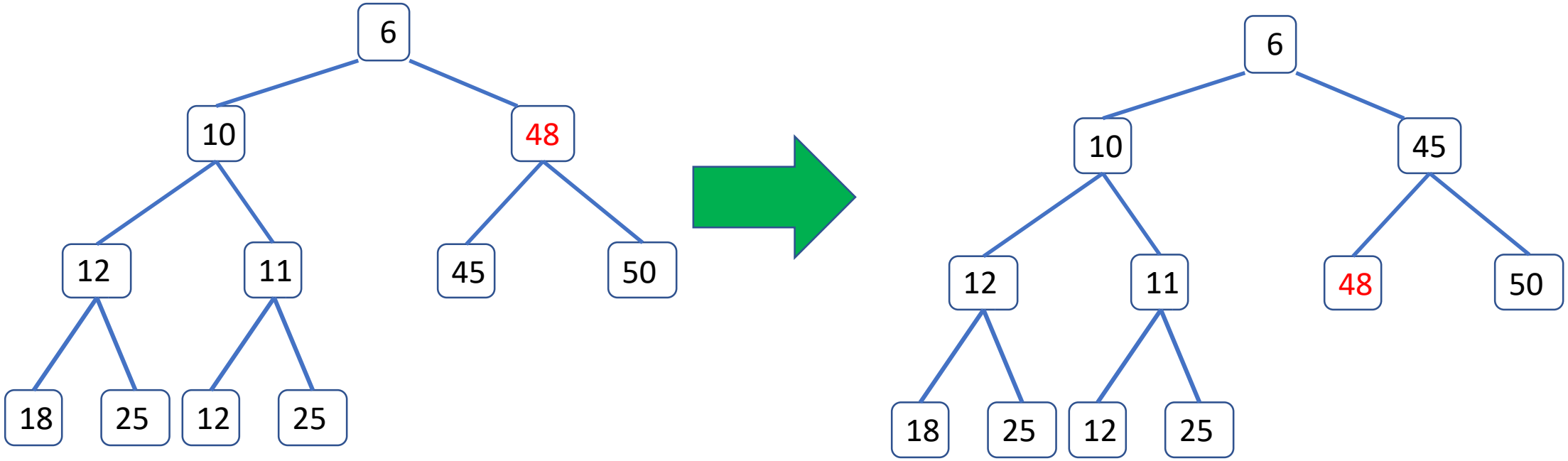
Continued

Solution:



Continued

Solution:

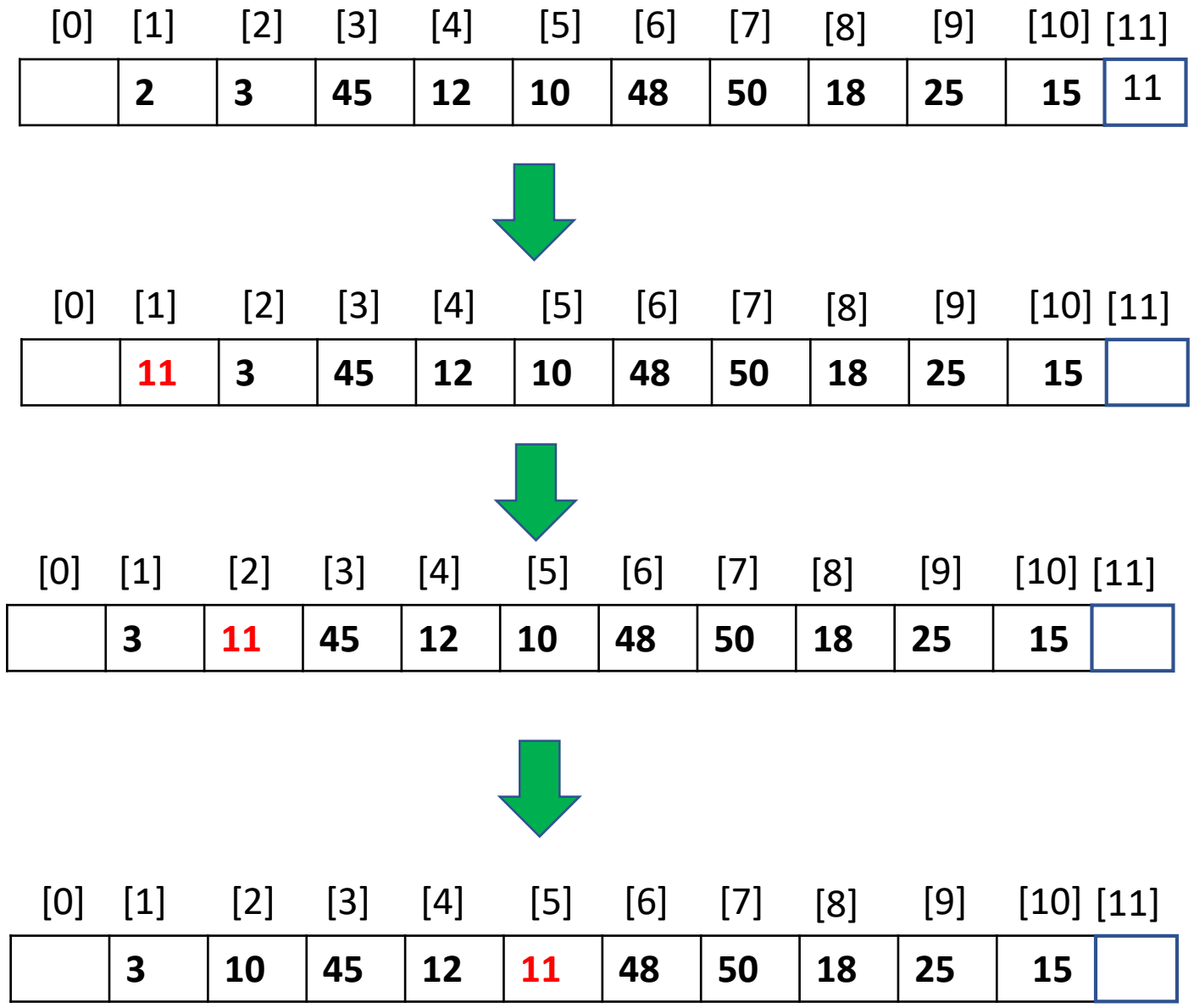


The hole has no children so we are done.

Now do it with an array. Remove the root from this heap:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
	2	**3**	**45**	**12**	**10**	**48**	**50**	**18**	**25**	**15**	11

Solution:



The hole (index 5) only has one child (index 10) and the hole value is smaller, so we are done.