Sorting

Heapsort

Heapsort

Heapsort is a time- and space-efficient sorting algorithm

- its best-, average-, and worst-case complexity are all O(n log n)
- it has O(1) space complexity (better than either recursive algorithm we saw... why?)

Heapsort works by first transforming the data into a heap

- a heap is a tree-based data structure that makes finding the largest (or smallest) value a constant-time operation
- in order to understand heapsort, it's important to understand exactly what a heap is

Heaps

A heap is:

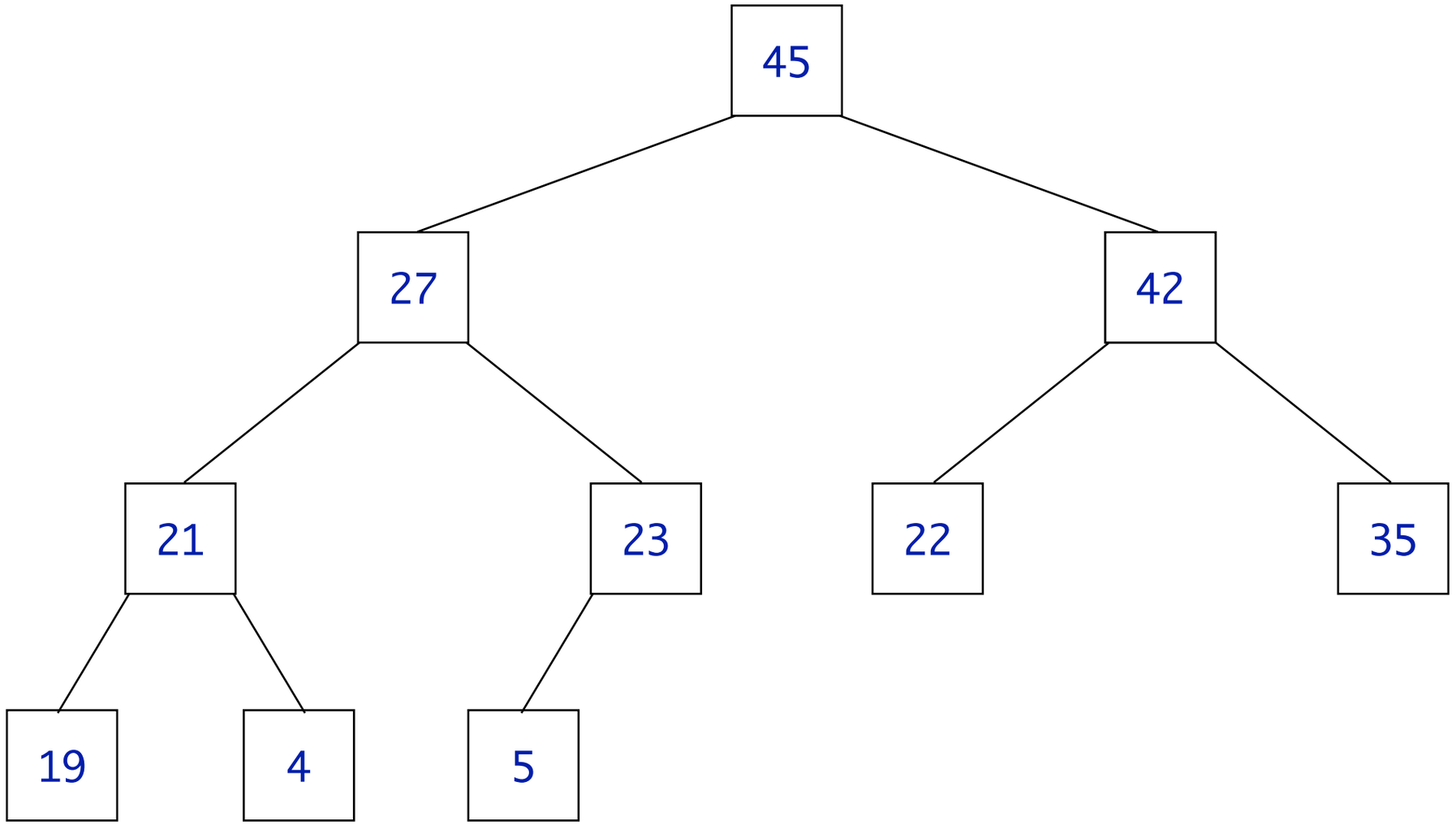
- a binary tree where values are organized using a strict weak ordering (see below)
- the value contained by a node is always larger than the values in the node's children (for a max heap)
- the tree is a complete binary tree (every level except the deepest hast as many nodes as possible, and all nodes at the bottom level are as far left as possible)

Strict weak ordering (common-sense orderings):

- `x < x is never true (irreflexivity)`
- `if x != y, then x < y and y < x cannot both be true (asymmetric)`
- `if x < y and y < z, then x < z (transitivity)`

Heaps

Example heap:



Heaps

Heaps are <u>complete</u> binary trees

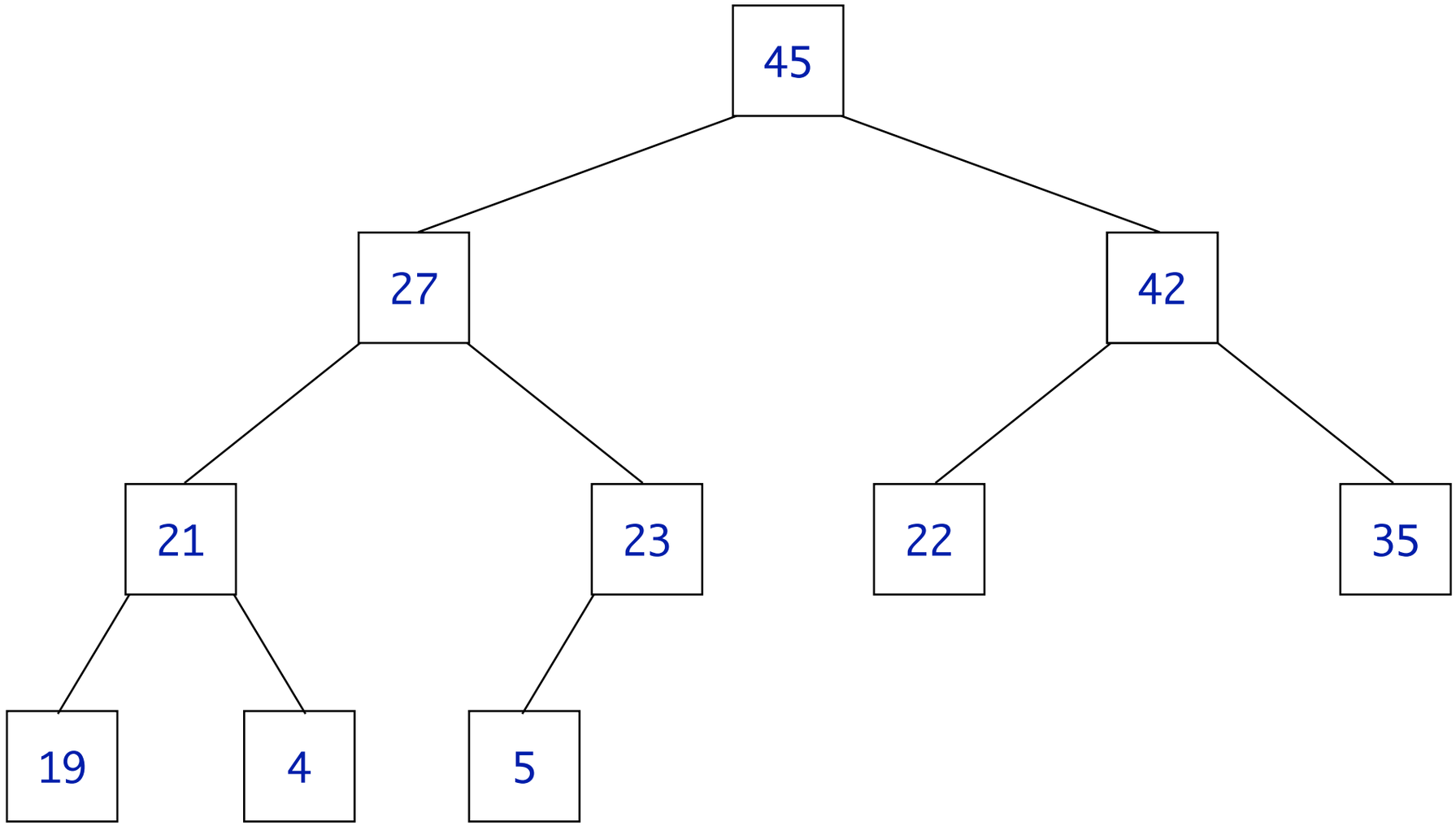
- all levels but the deepest contain the maximum possible
- nodes in the deepest level are as far left as possible

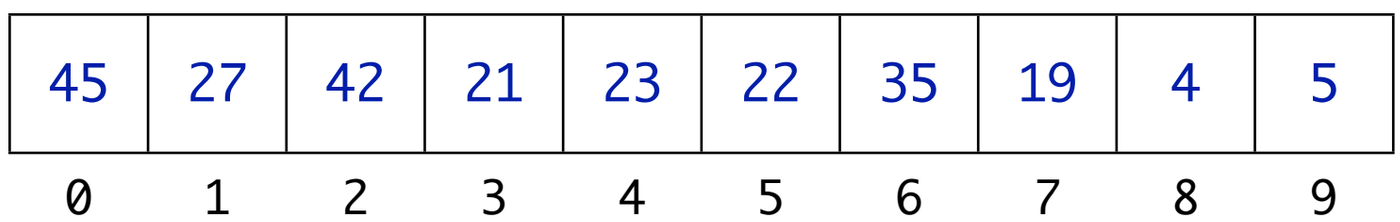
We've seen an efficient array representation for complete trees:

- data at the root of the tree is stored at index `0`
- the left child of the element at index `i` is stored at index `2i+1`
- the right child of the element at index `i` is stored at index `2i+2`
- the parent of the element at index `i` is stored at index `(i-1)/2` (using int division)

Heaps

Example heap:





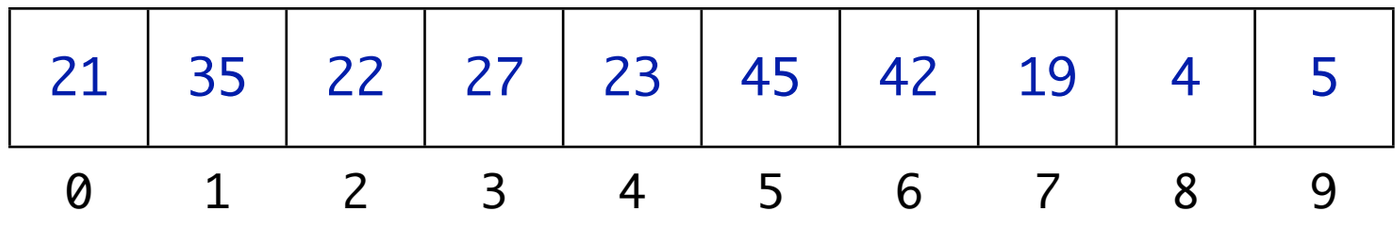
Heapsort

The first step in heapsort is to make the array into a heap (surprise!)

- the value contained in a node is always larger than that of either of its children
- it must be treated as a complete binary tree (easy, given our representation)

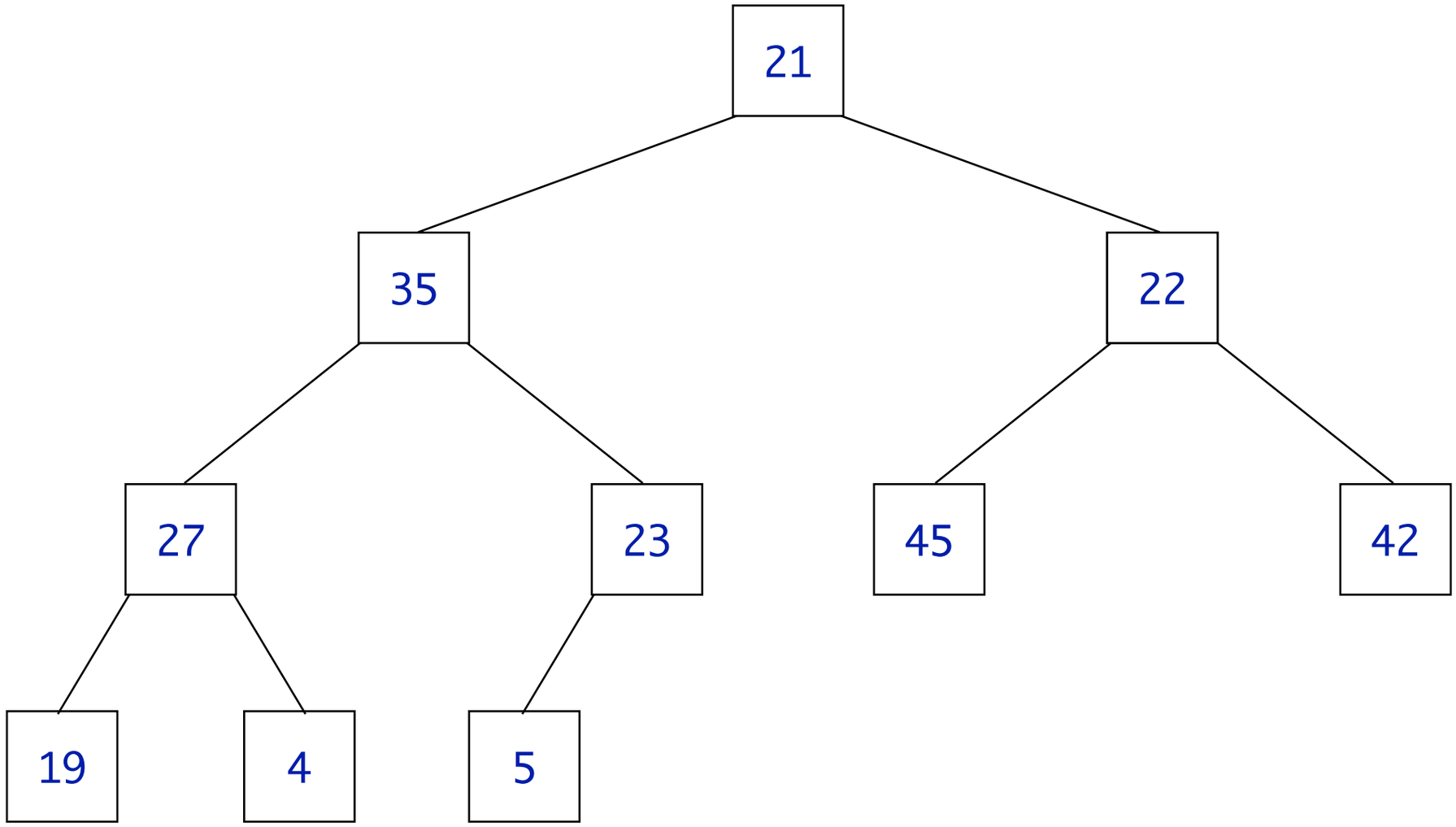
Consider the following array:

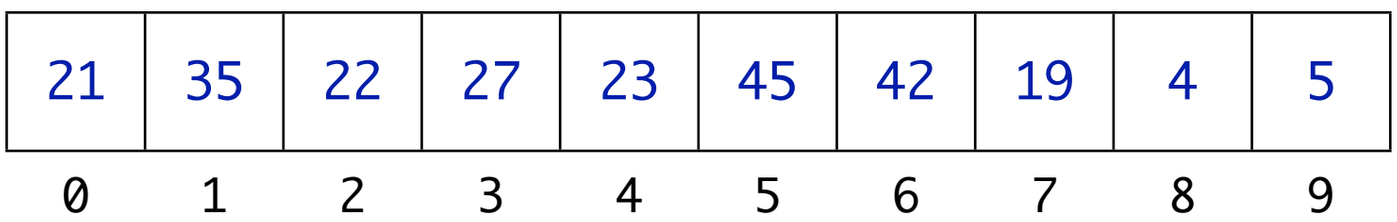
- it's not yet a heap, but—like we've seen—it can be treated as a complete binary tree
- we need to heapify the tree so that each node is larger than its children



Heapsort

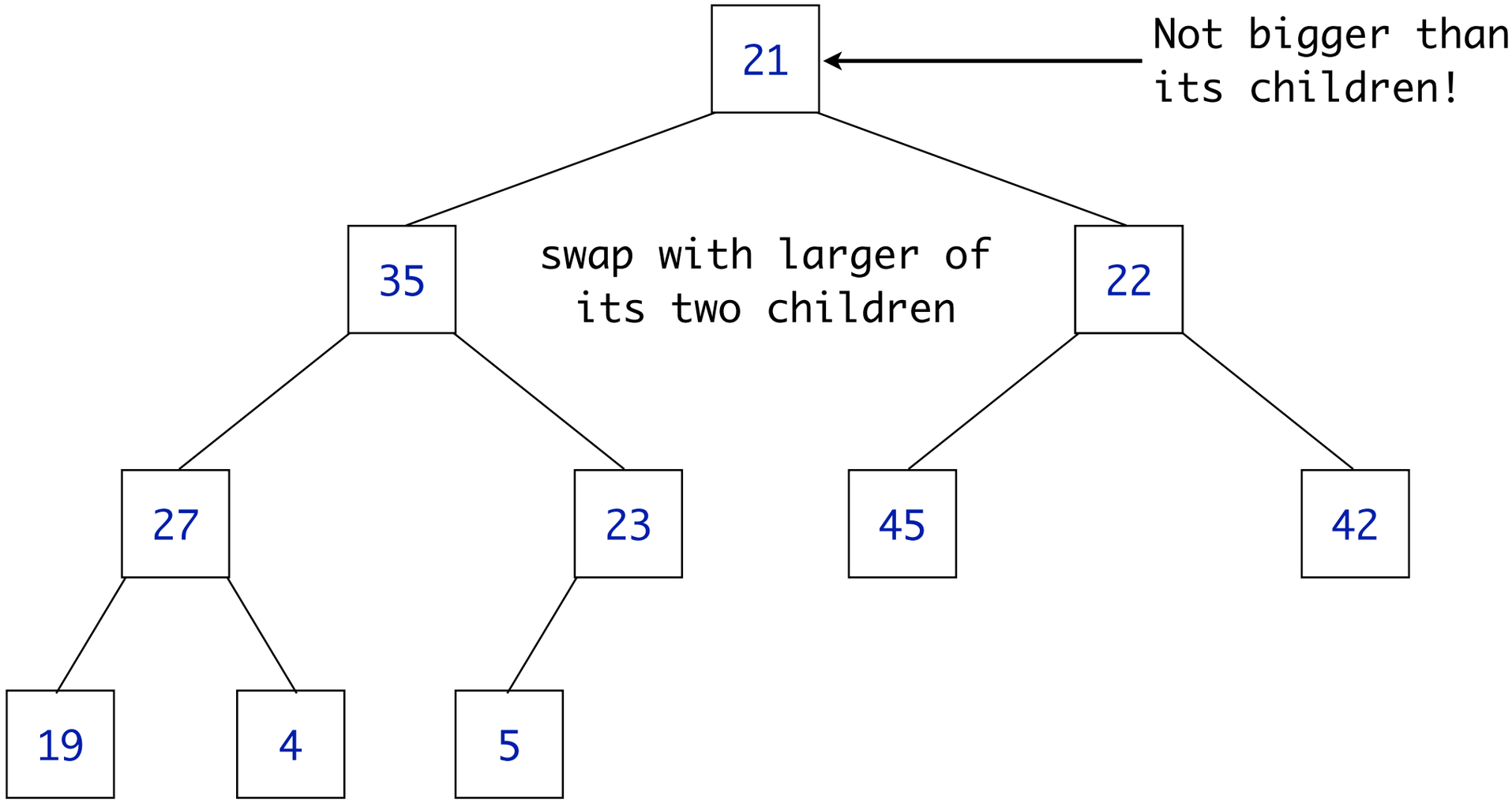
The process of making a heap is called "heapifying"—clever, huh?

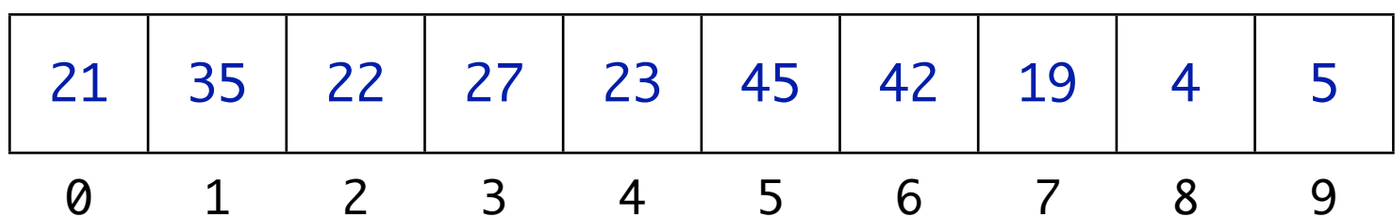




Heapsort

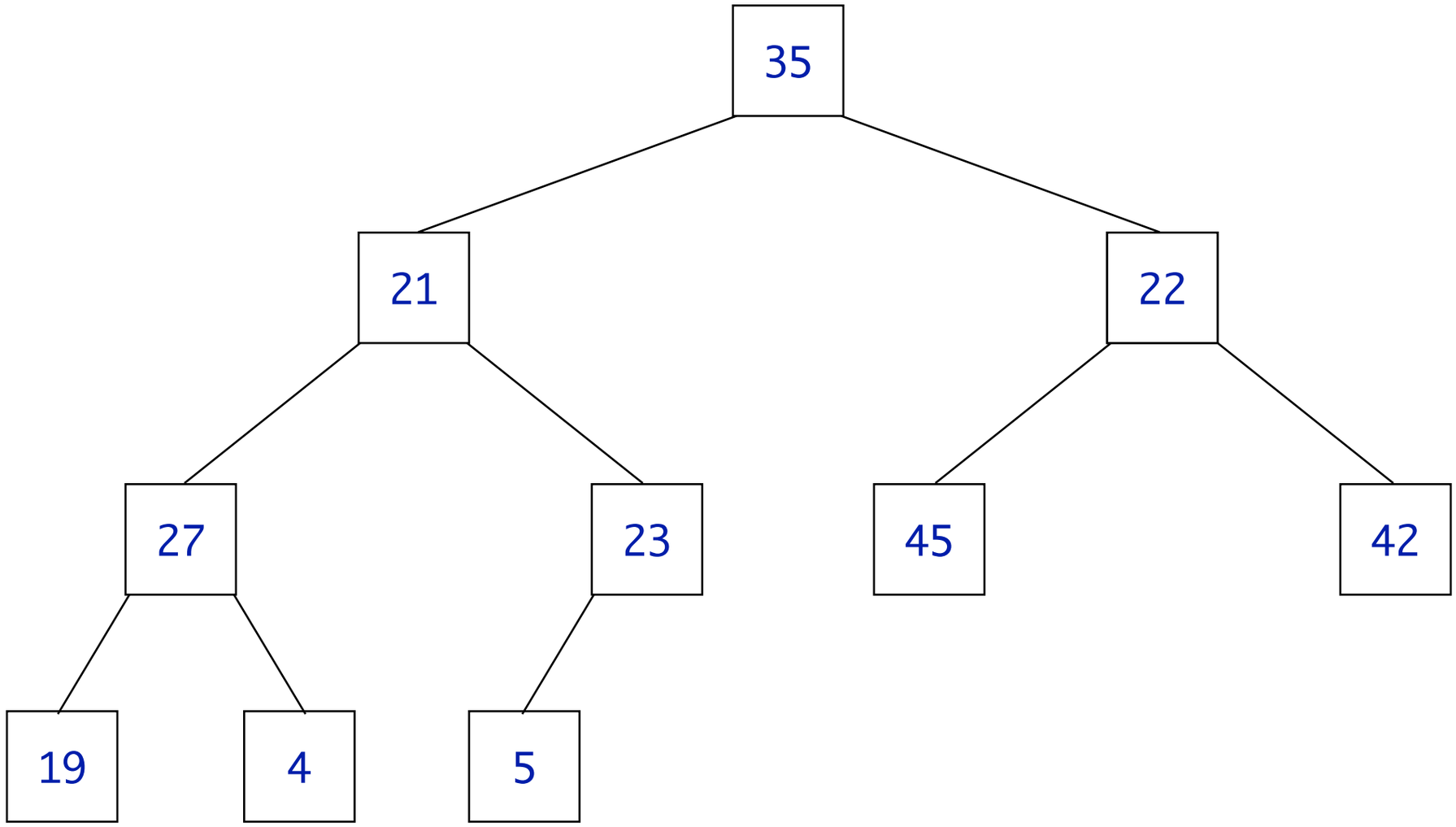
"Heapify" the entire tree:

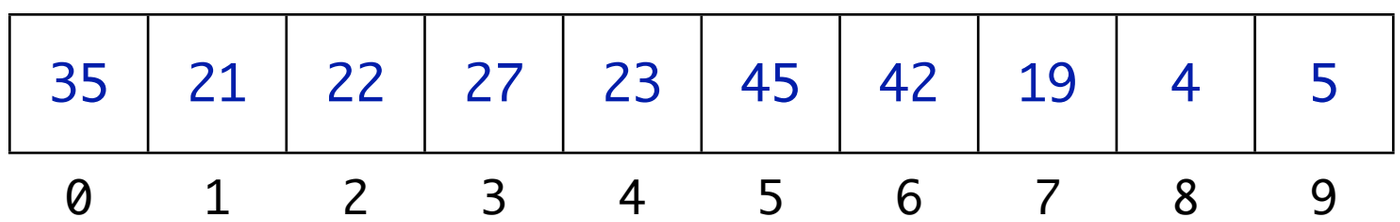




Heapsort

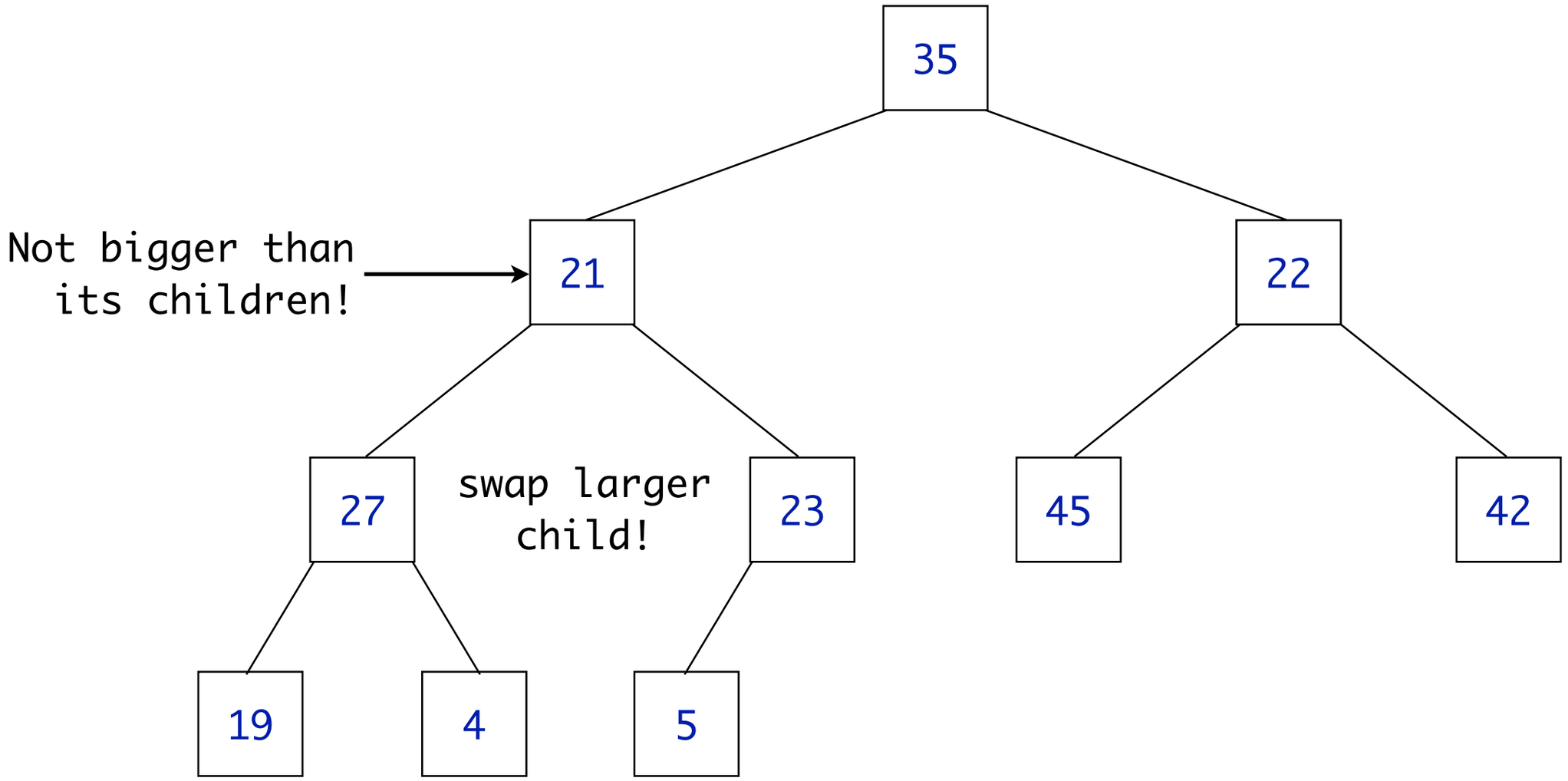
"Heapify" the entire tree:

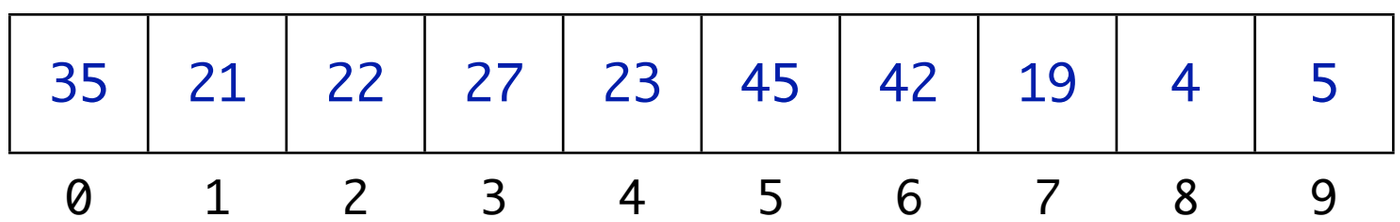




Heapsort

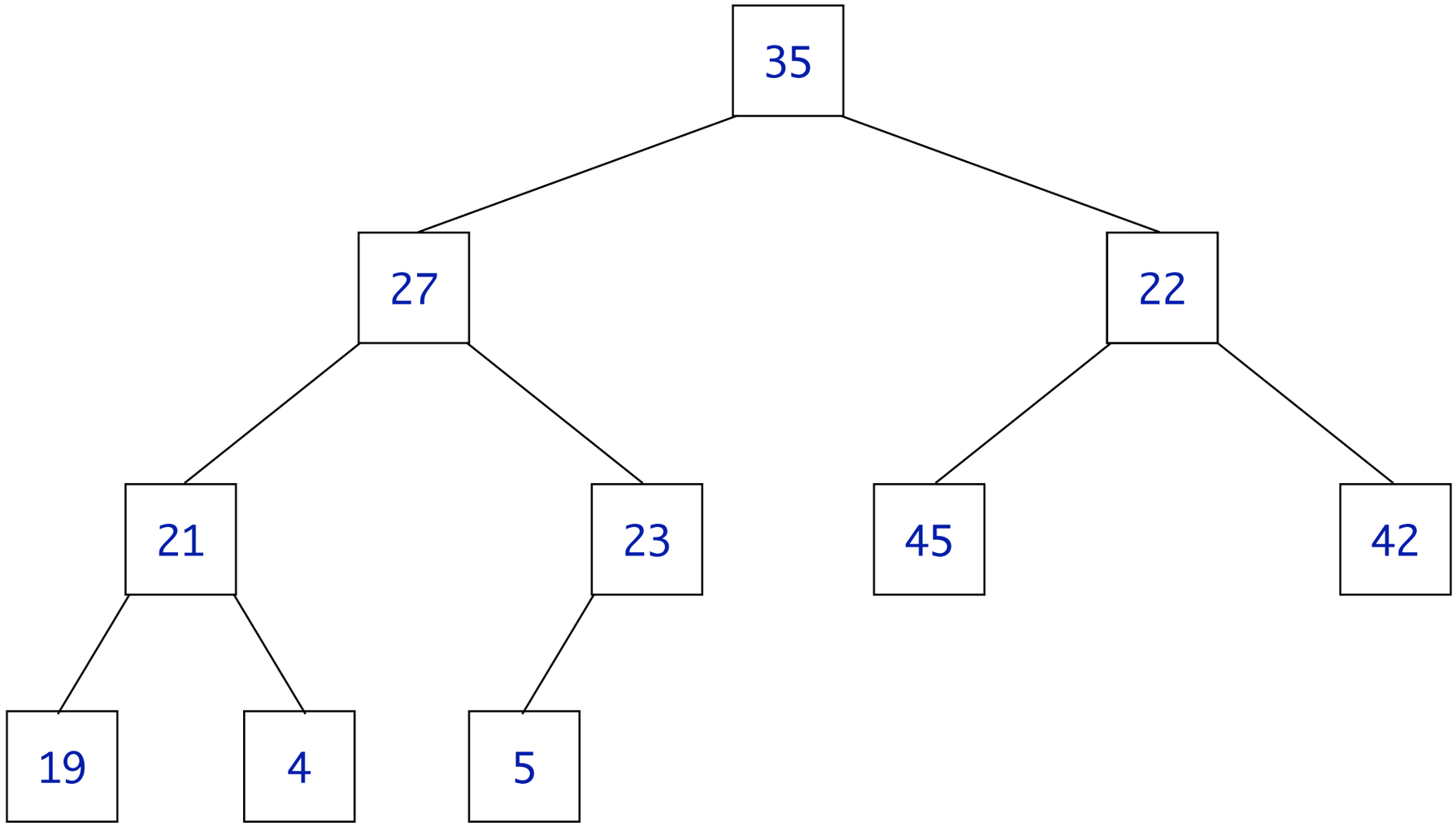
Recursively "heapify" the left subtree:

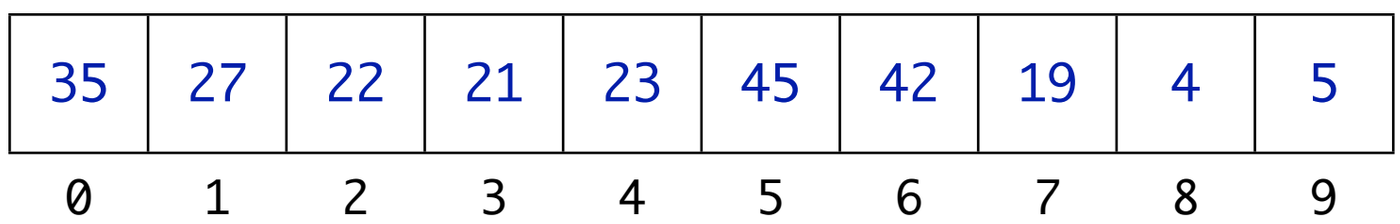




Heapsort

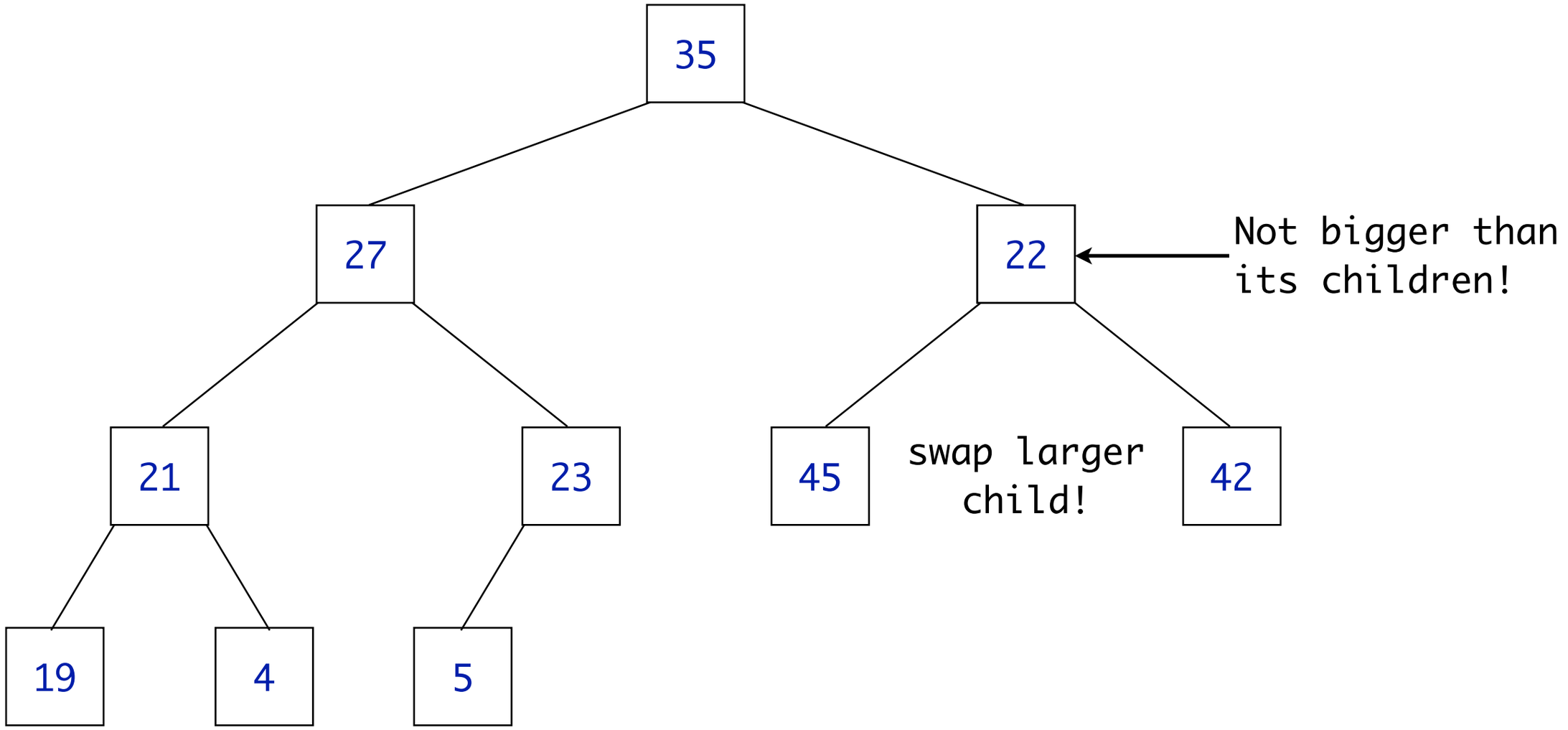
Left subtree is now good!

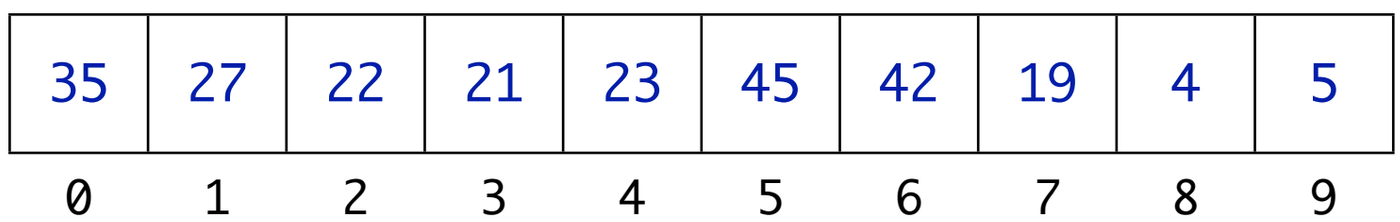




Heapsort

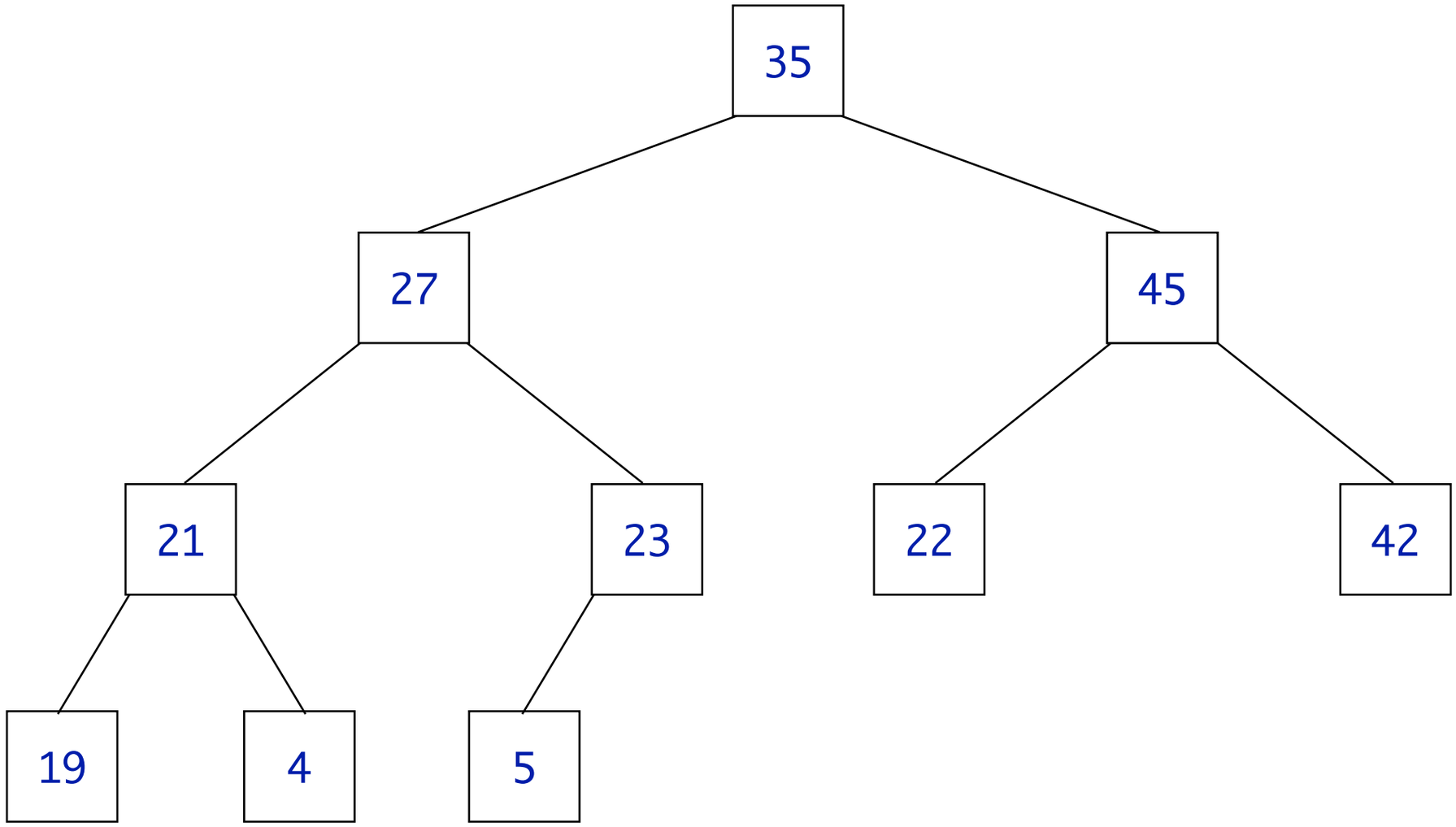
Recursively "heapify" the right subtree:

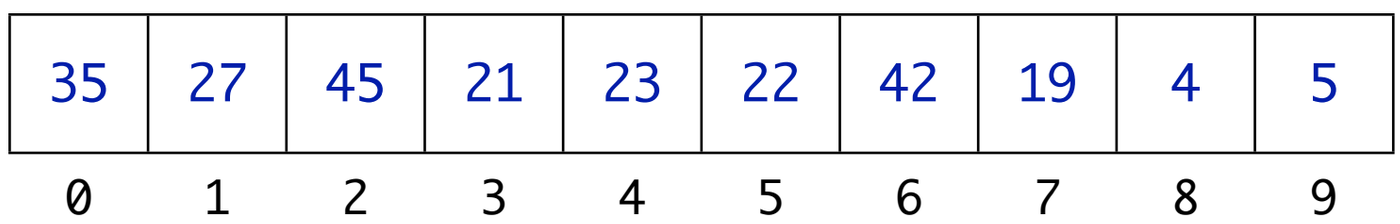




Heapsort

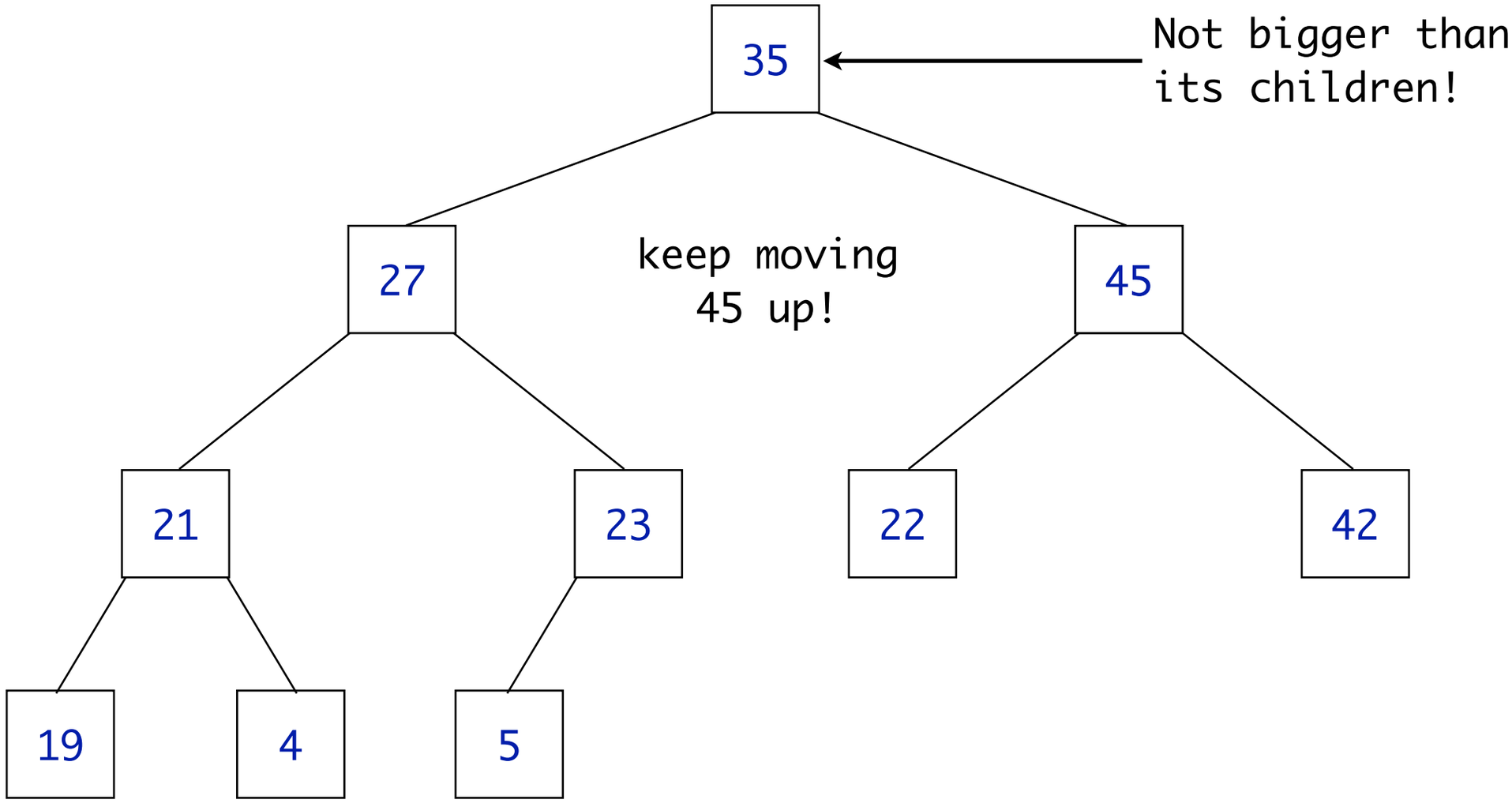
Recursively "heapify" the right subtree:

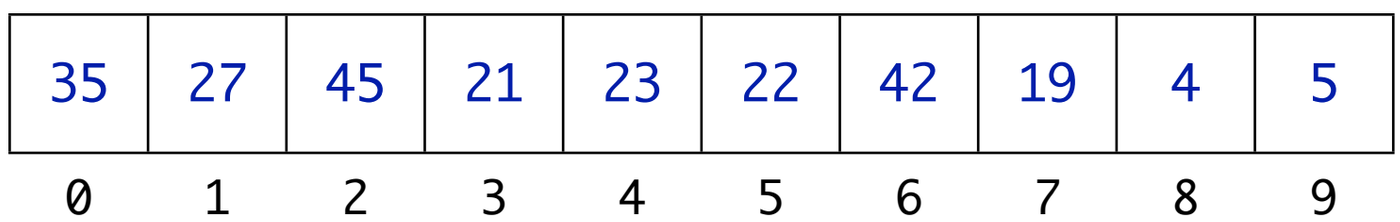




Heapsort

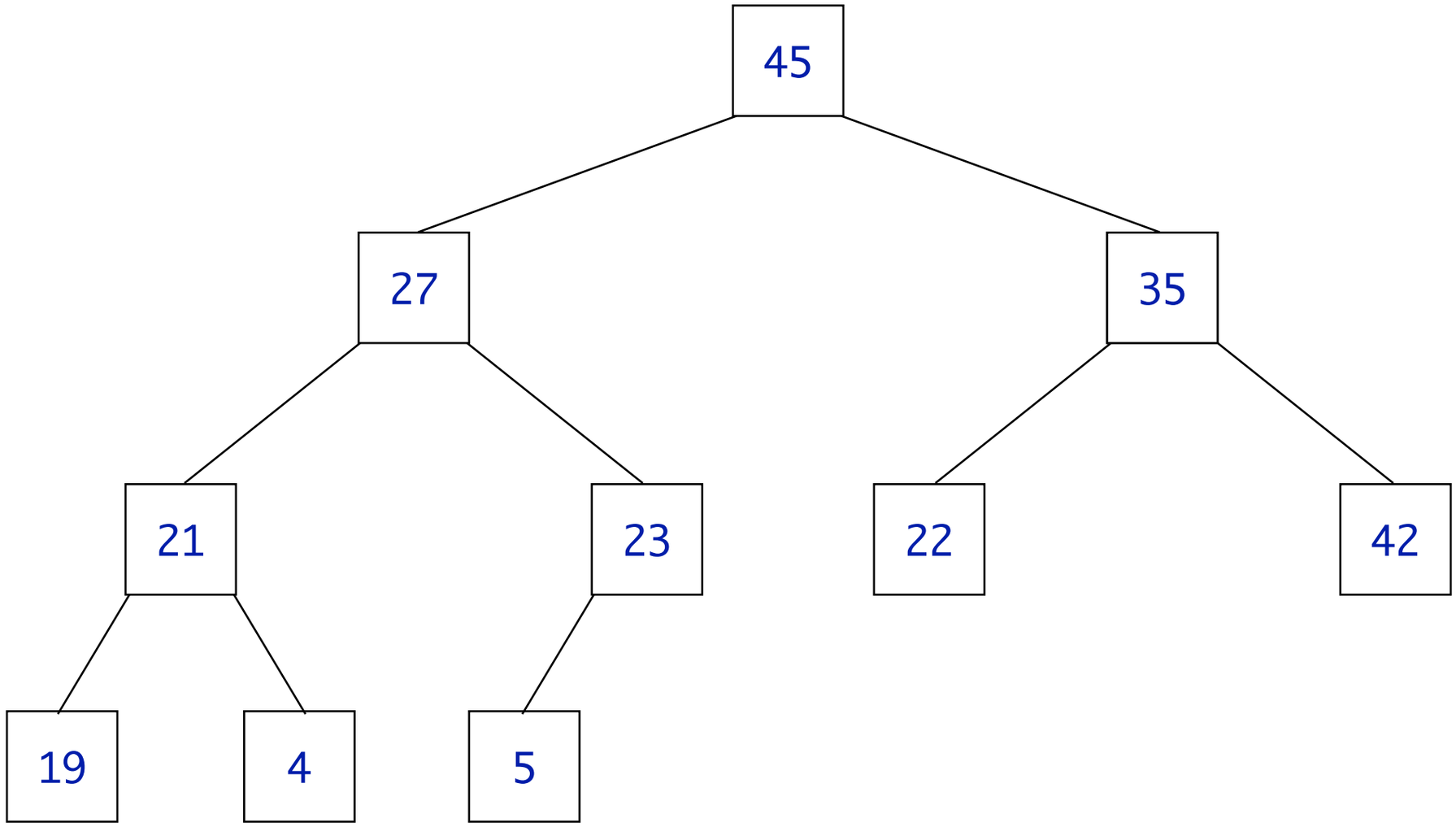
Recursively "heapify" the right subtree:

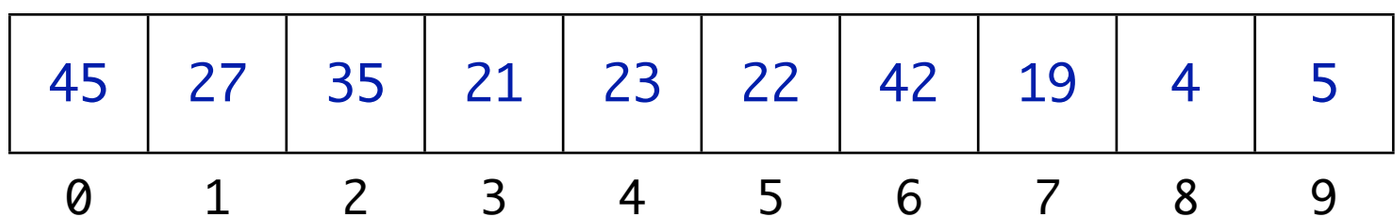




Heapsort

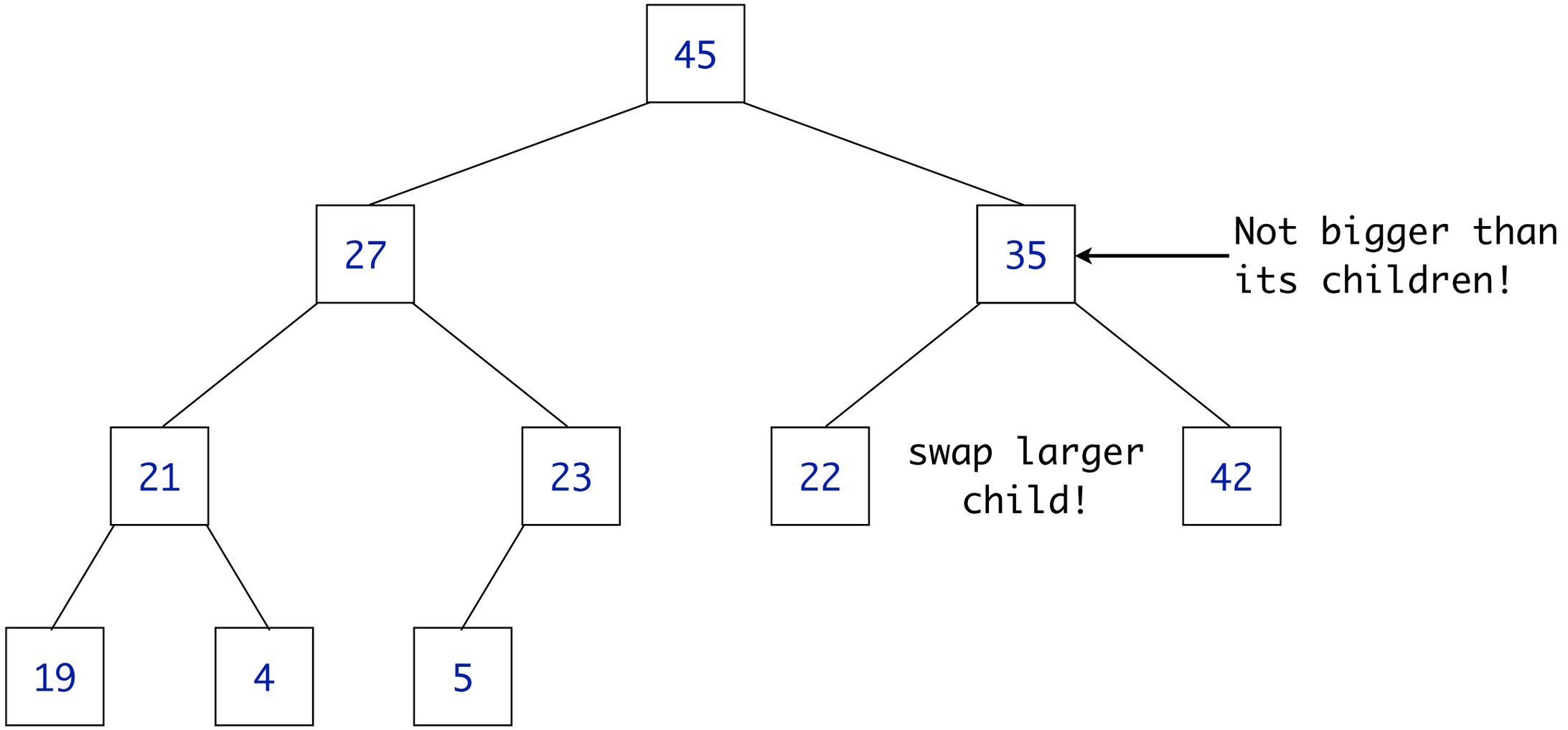
Recursively "heapify" the right subtree:

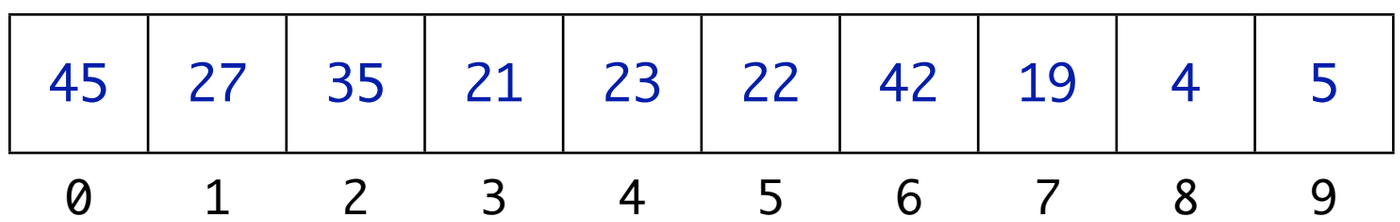




Heapsort

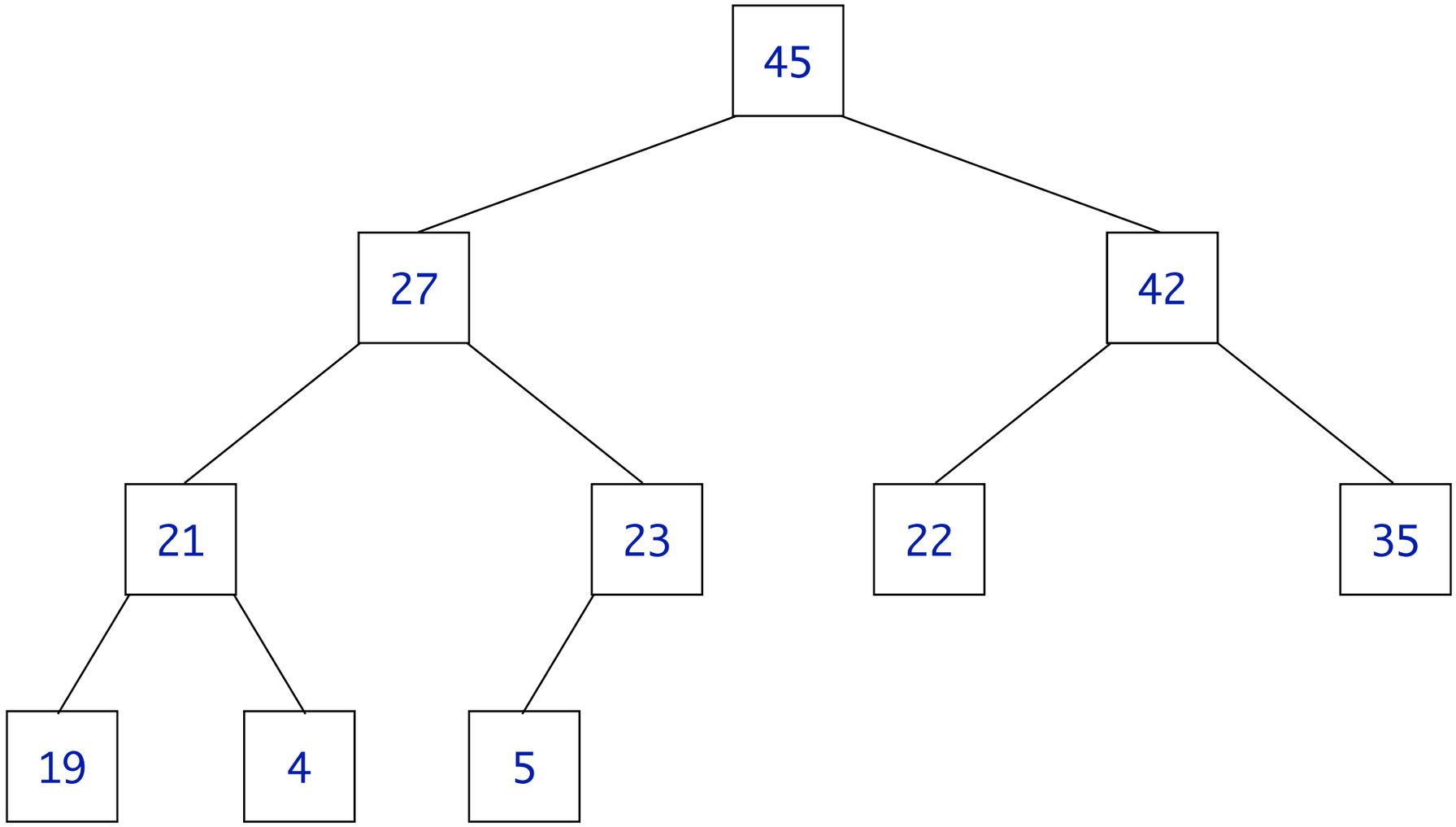
Recursively "heapify" the right subtree:





Heapsort

Right subtree is now good!



45	27	42	21	23	22	35	19	4	5
0	1	2	3	4	5	6	7	8	9

Heapsort

The first step in heapsort is to make the array into a heap (surprise!)

- the value contained in a node is always larger than that of either of its children
- it must be treated as a complete binary tree (easy, given our representation)

Consider the following array:

- now that it is a heap, we're ready to sort it...
- we know that the largest value is on the top, so we can put it in its correct place at the end of the array (via swapping)

45	27	42	21	23	22	35	19	4	5
0	1	2	3	4	5	6	7	8	9

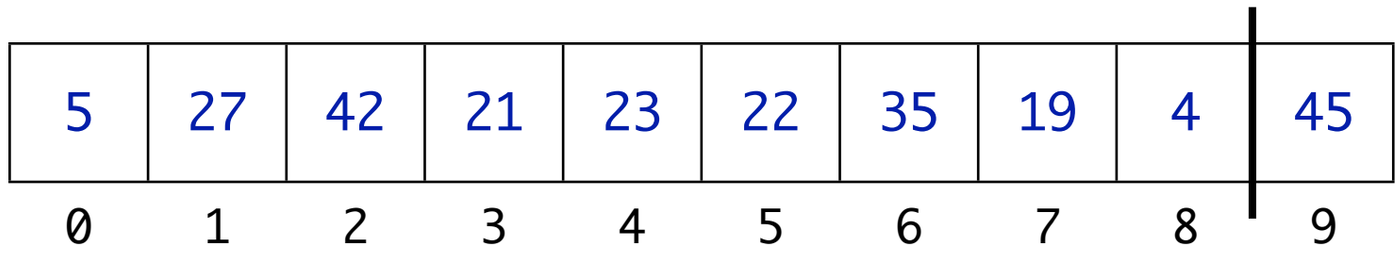
Heapsort

The first step in heapsort is to make the array into a heap (surprise!)

- the value contained in a node is always larger than that of either of its children
- it must be treated as a complete binary tree (easy, given our representation)

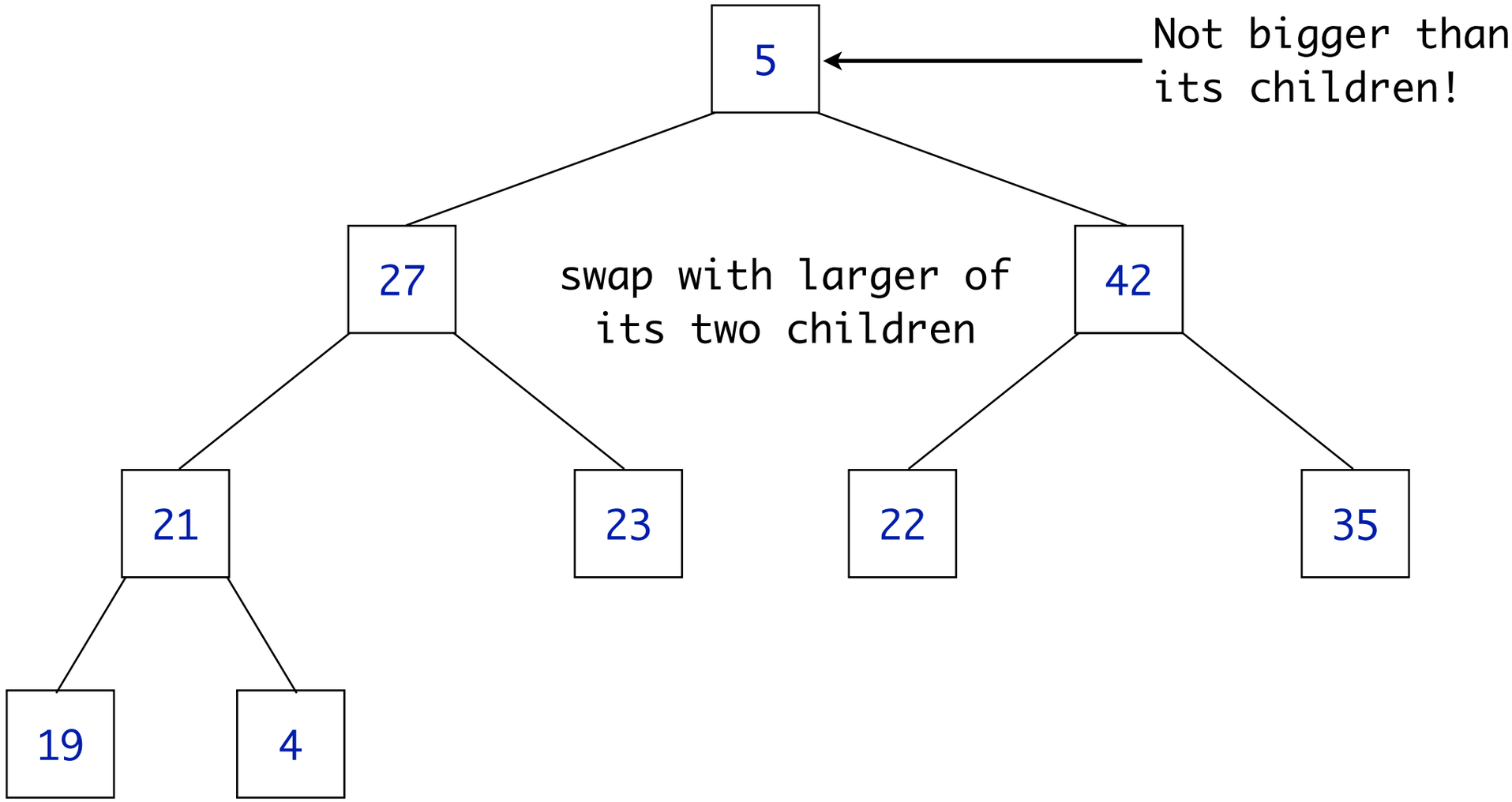
Consider the following array:

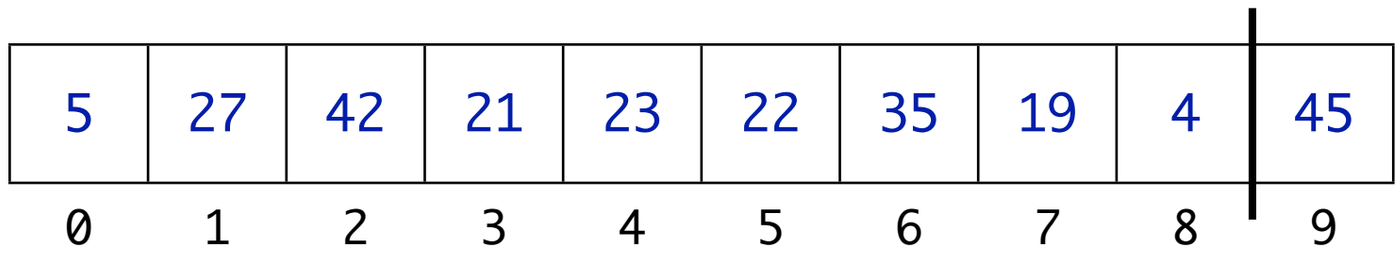
- now that it is a heap, we're ready to sort it...
- we know that the largest value is on the top, so we can put it in its correct place at the end of the array (via swapping)
- we now have a sorted array of size 1 (at the end)
- the unsorted array is down to 9 elements, and it's NEARLY A HEAP already!



Heapsort

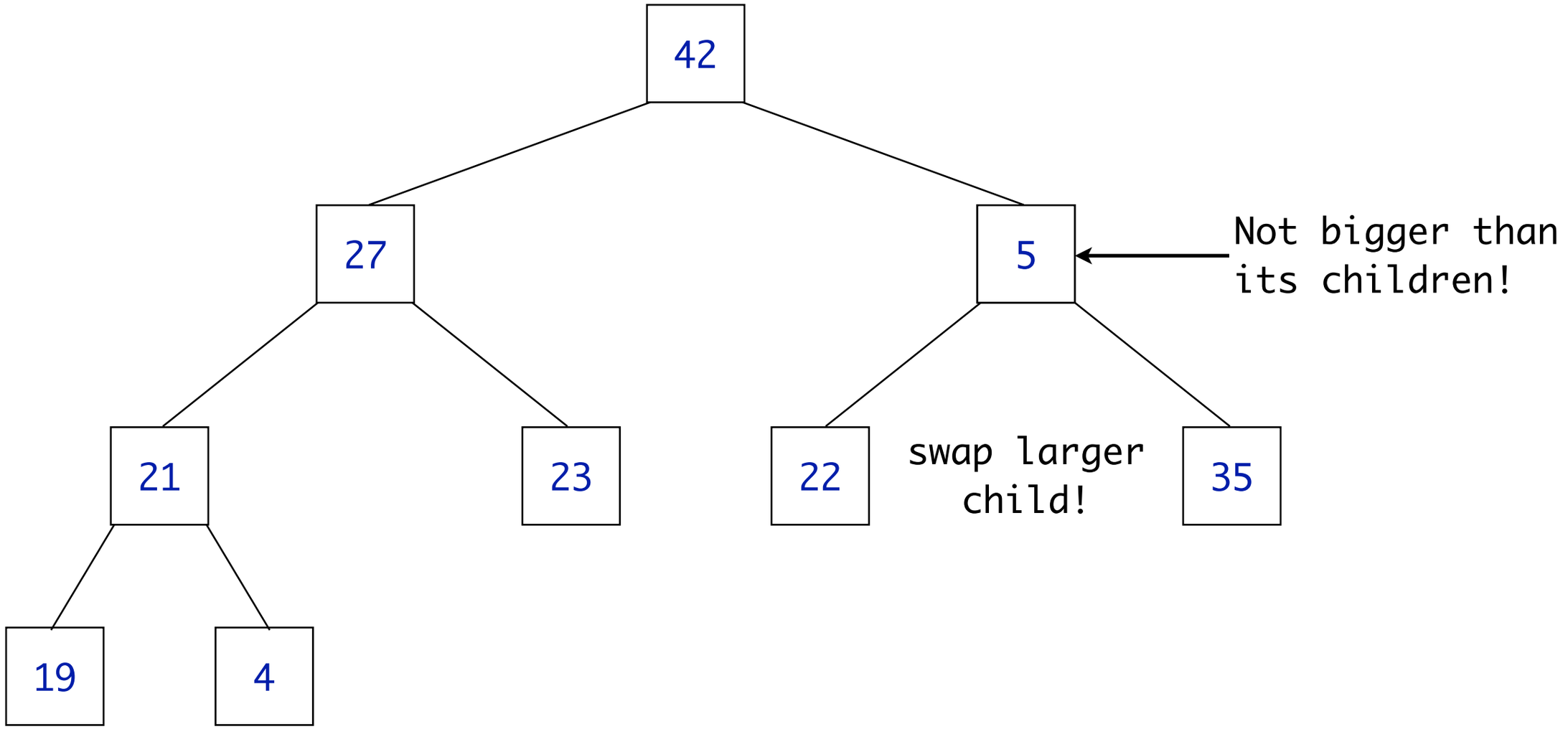
Nine-element unsorted array... Nearly a heap already!

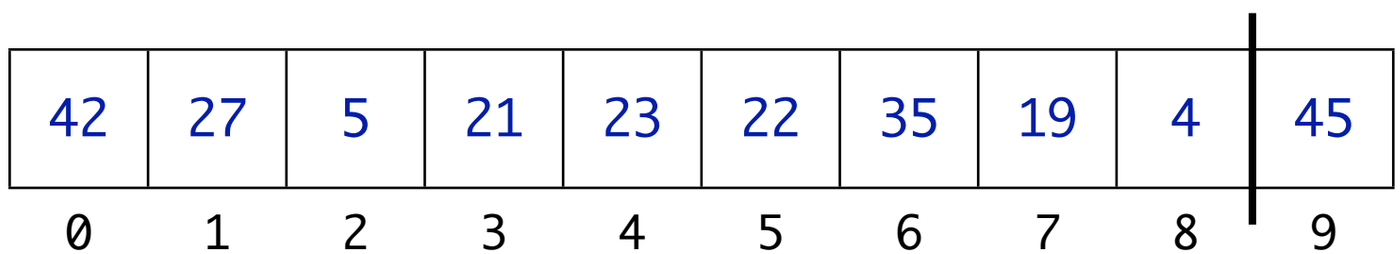




Heapsort

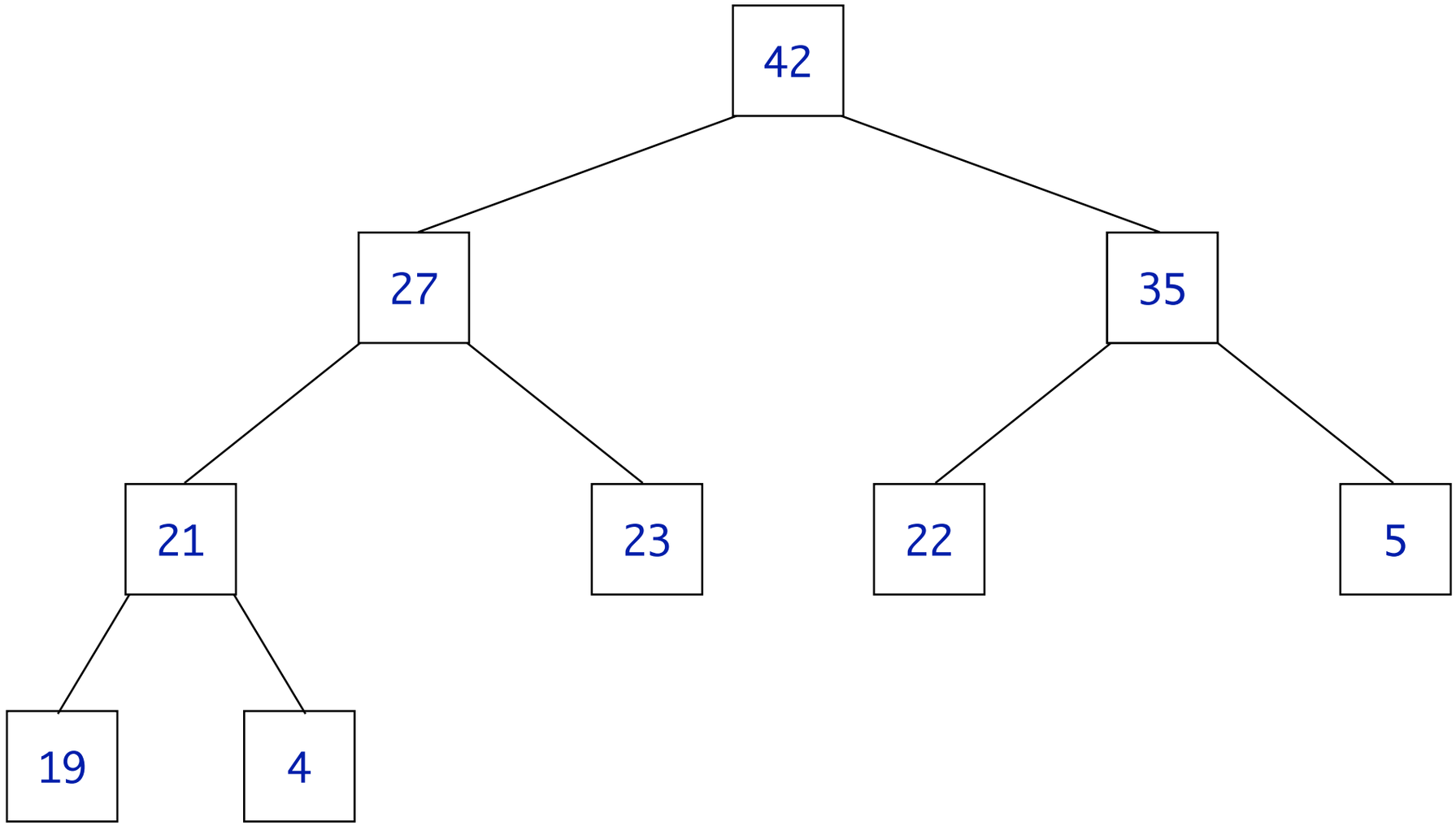
Nine-element unsorted array... Nearly a heap already!

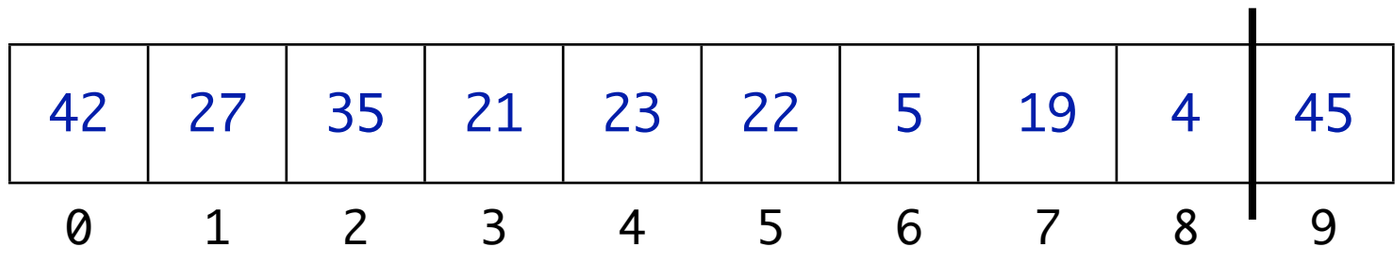




Heapsort

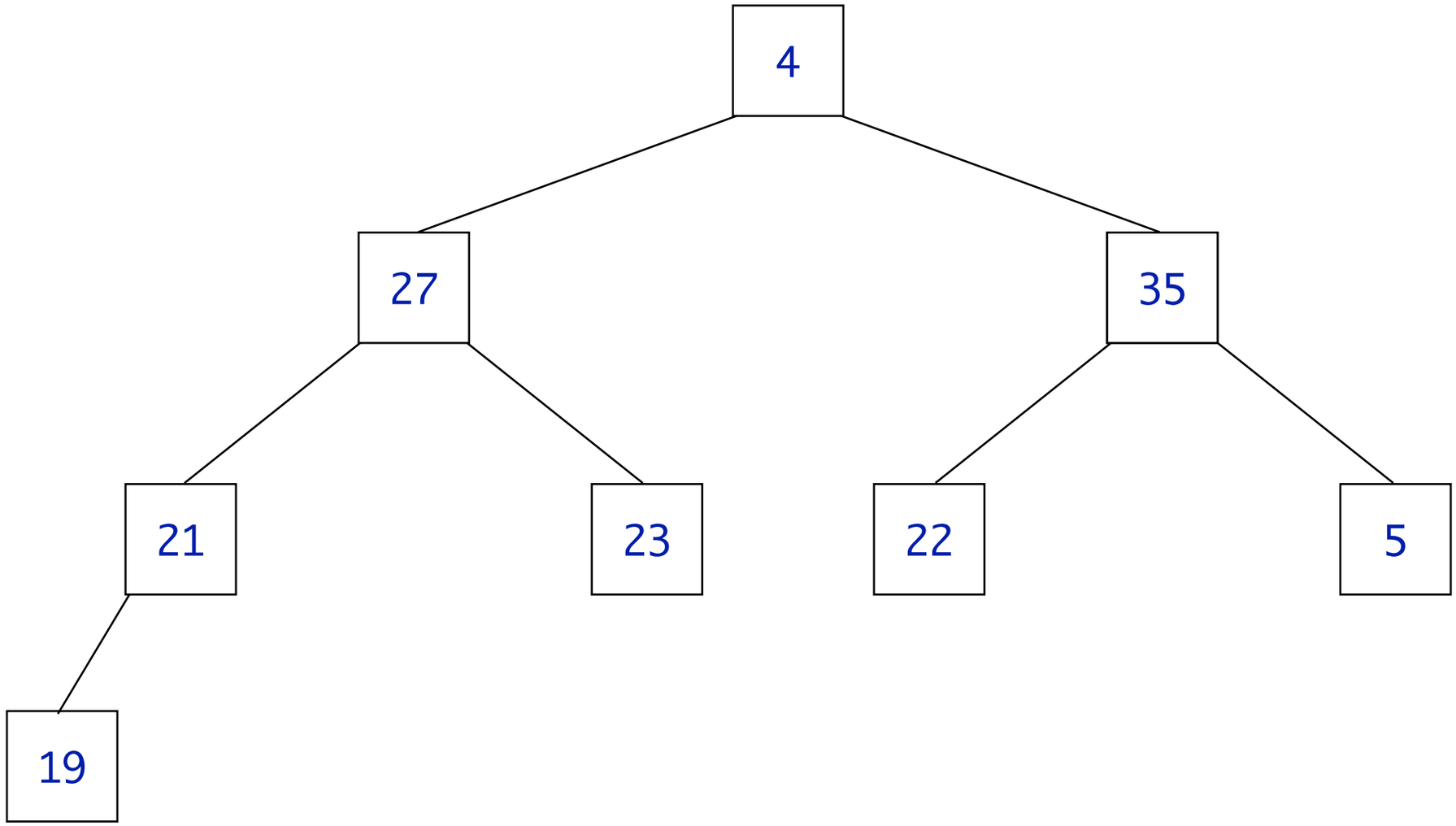
That was easy!

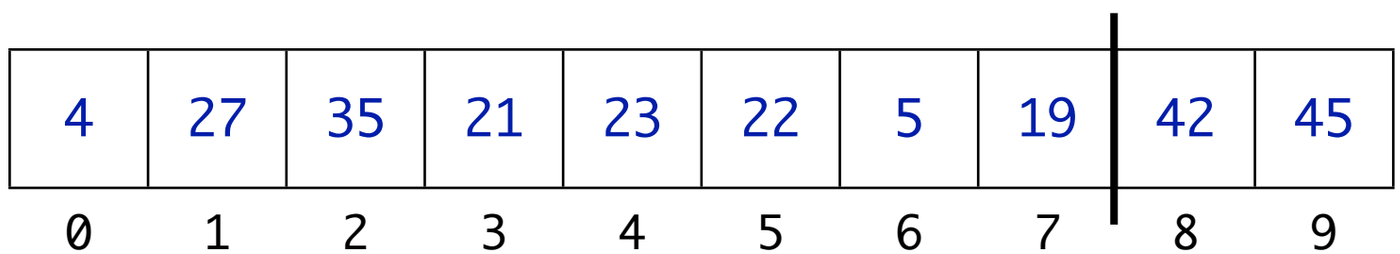




Heapsort

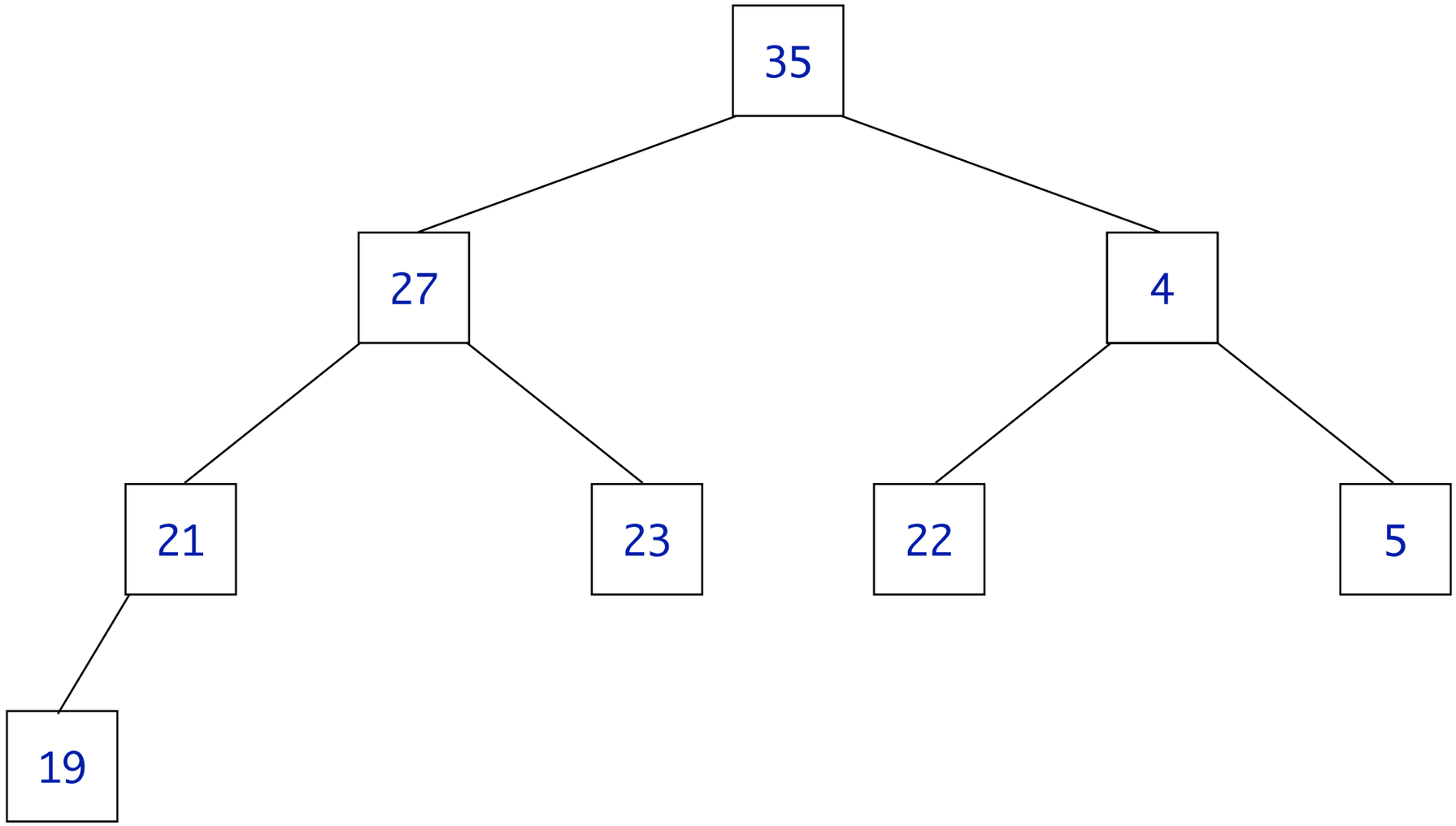
Take largest value and put it in its correct spot...

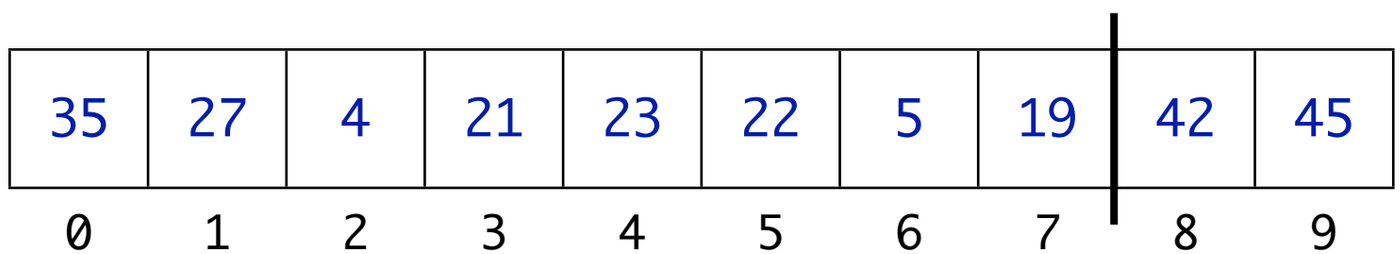




Heapsort

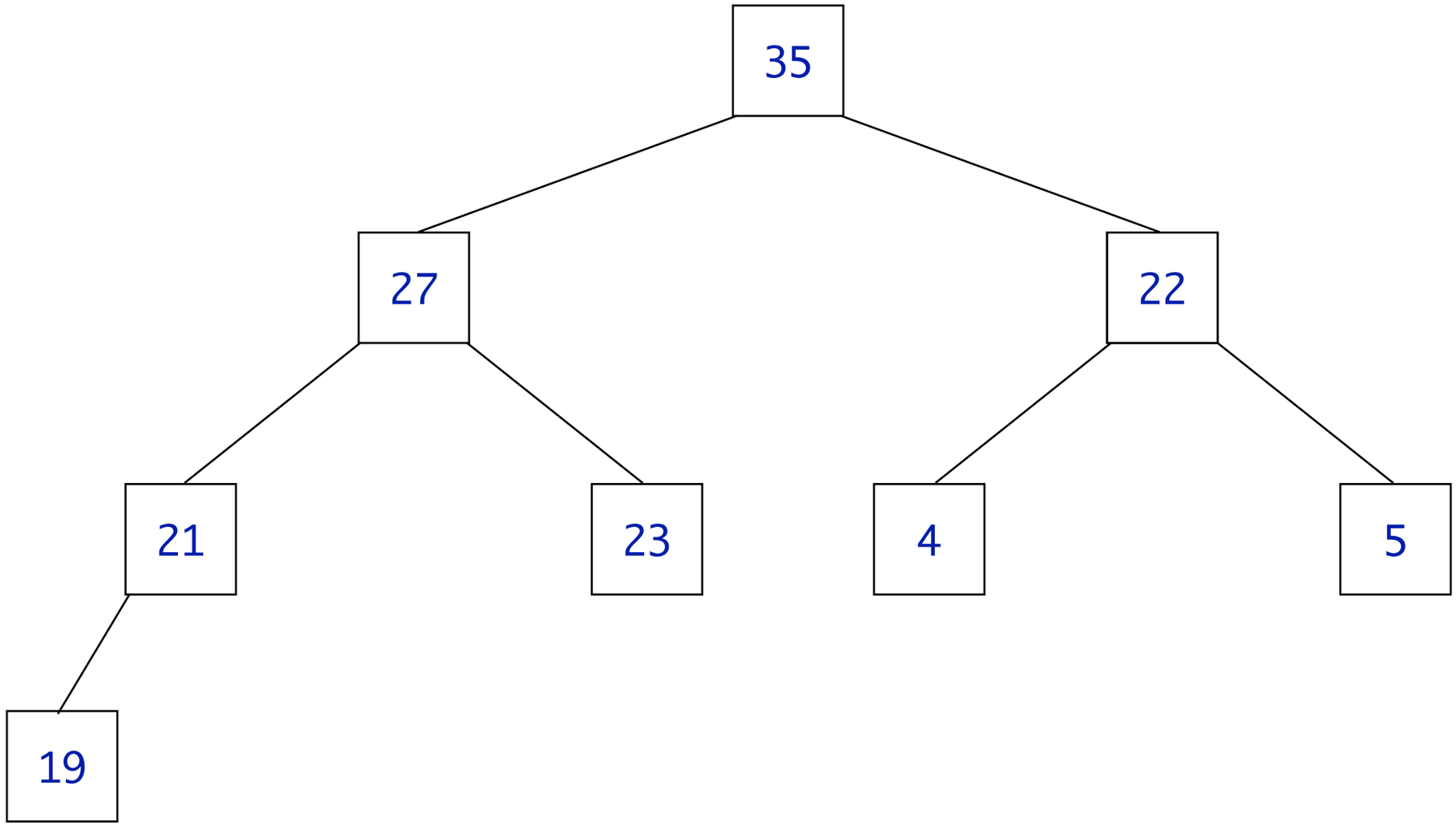
Reheapify downward!

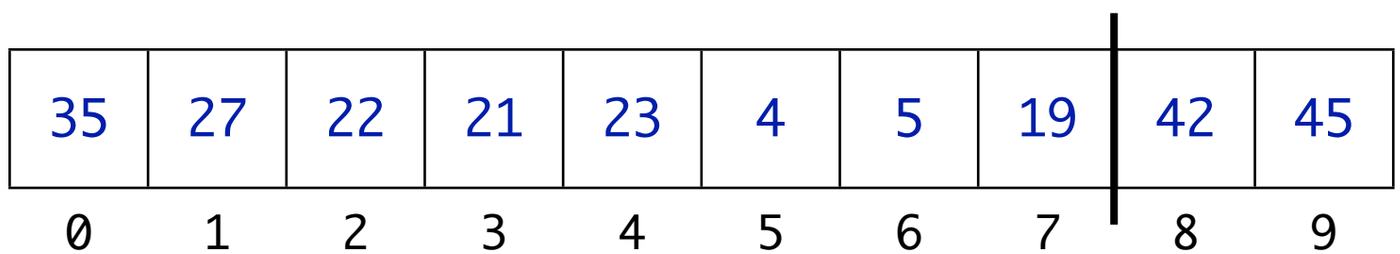




Heapsort

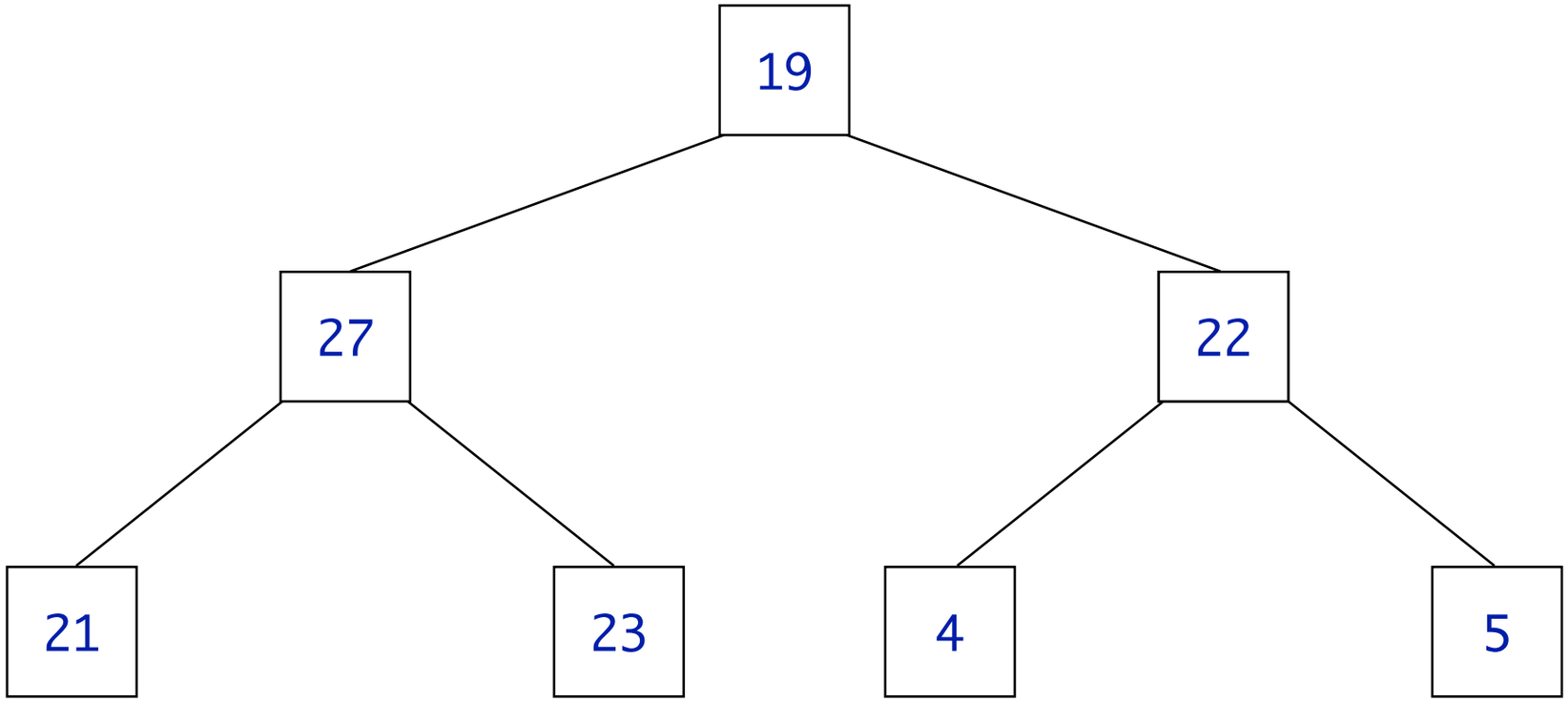
Reheapify downward!

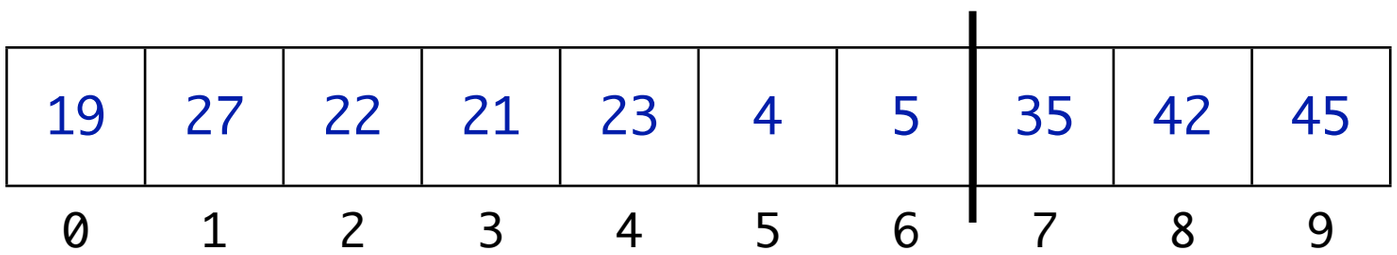




Heapsort

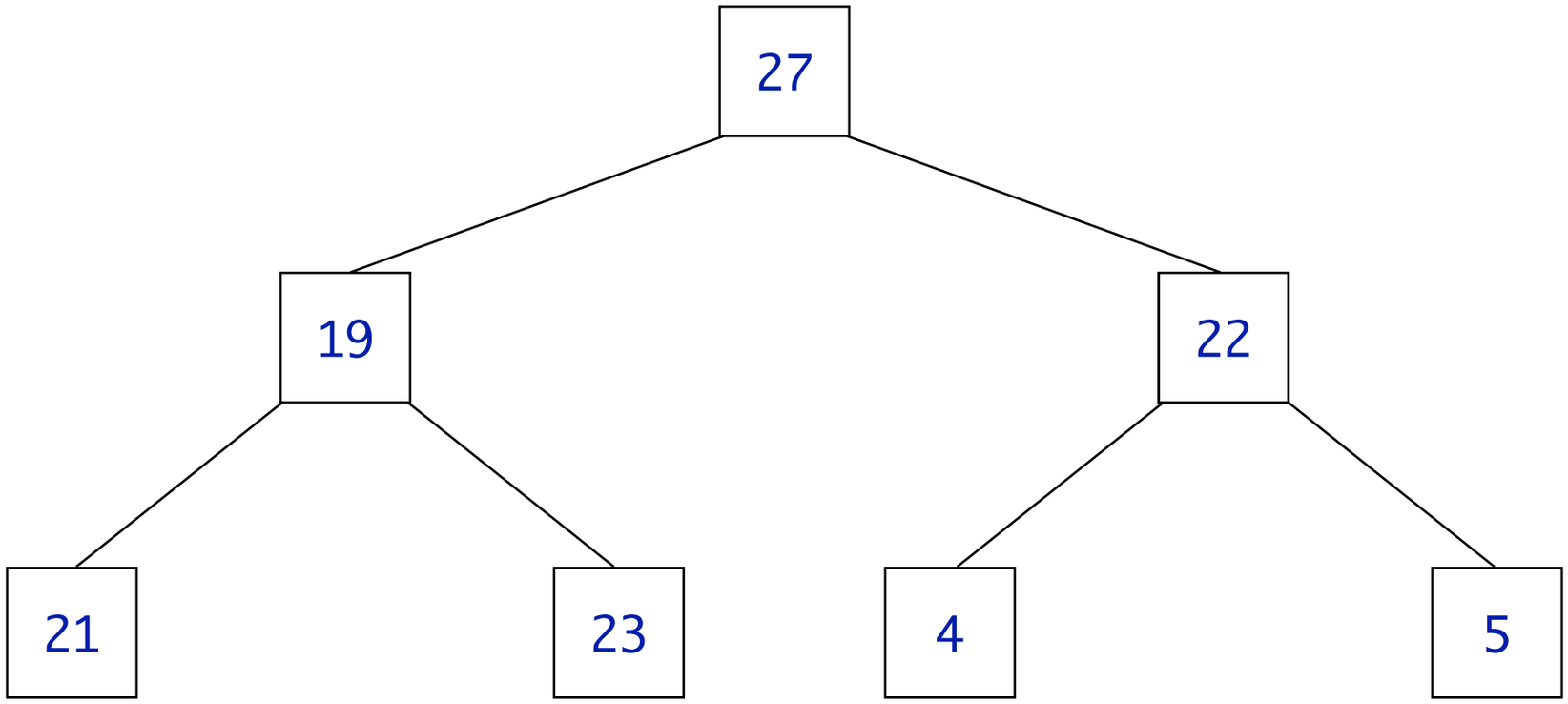
Take largest value and put it in its correct spot...

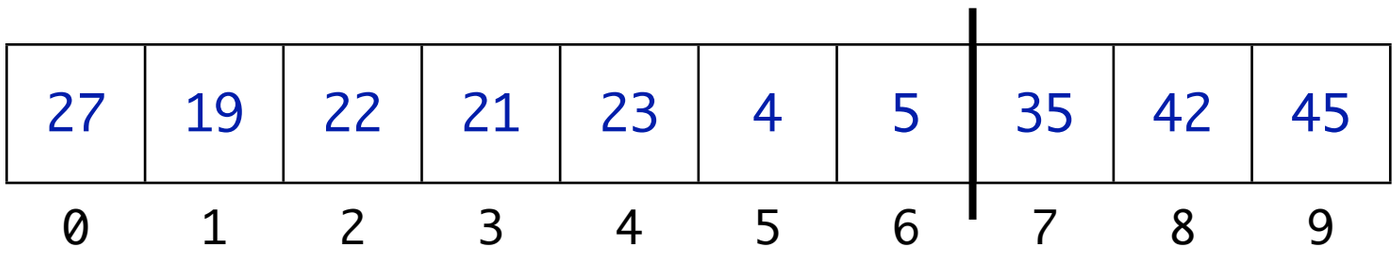




Heapsort

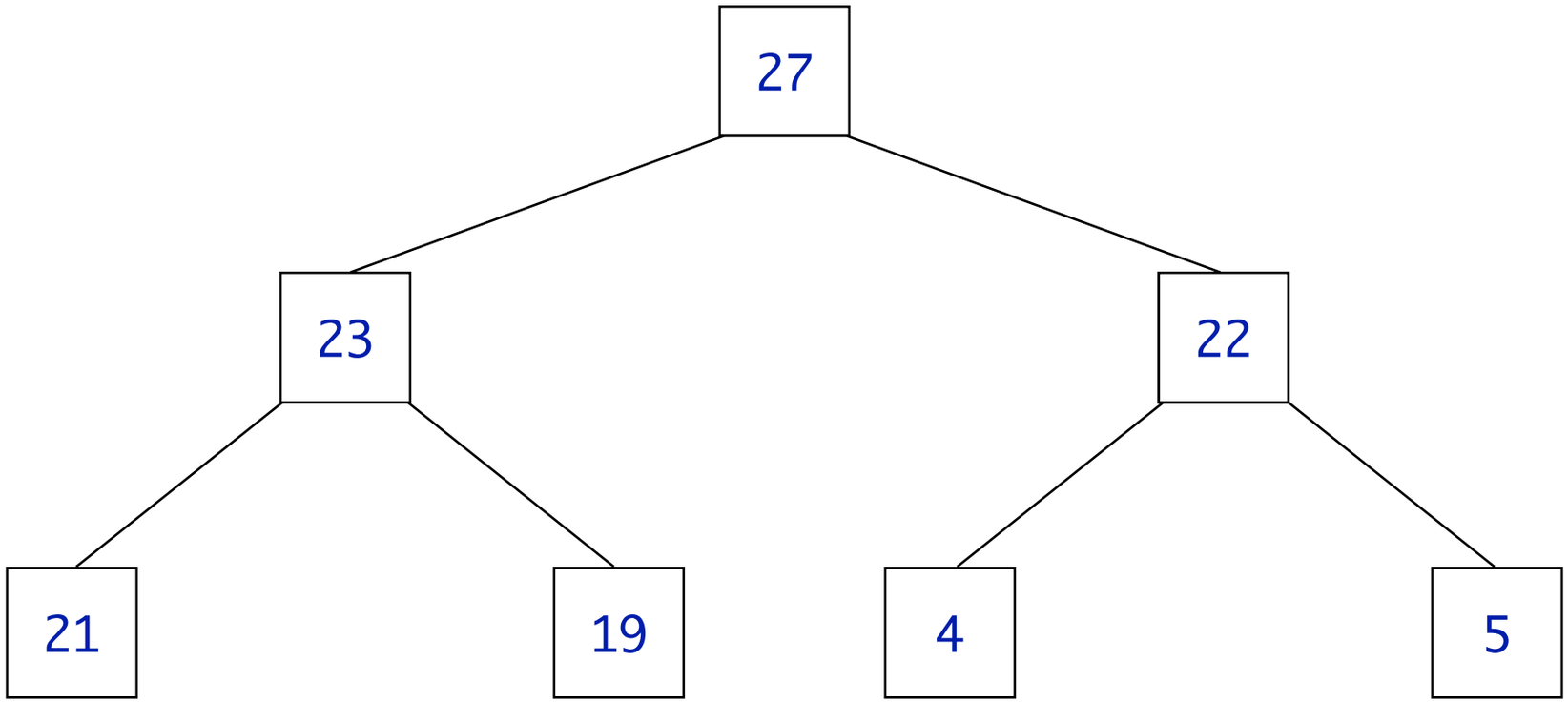
Reheapify downward!

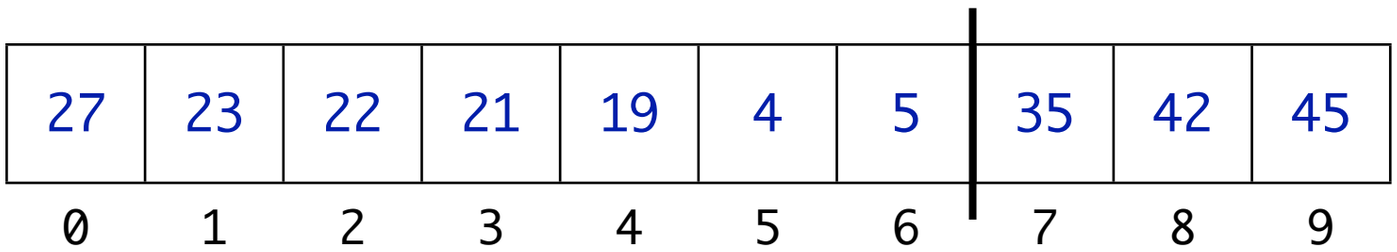




Heapsort

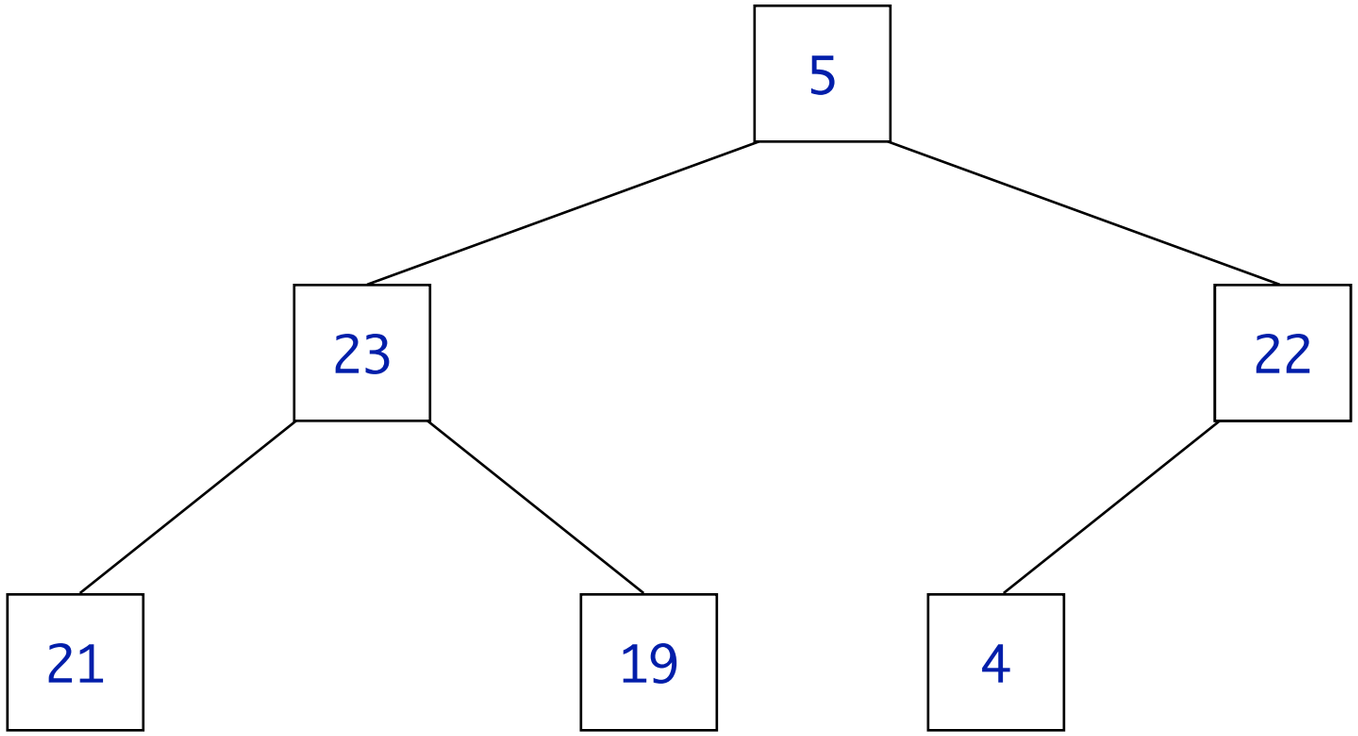
Reheapify downward!

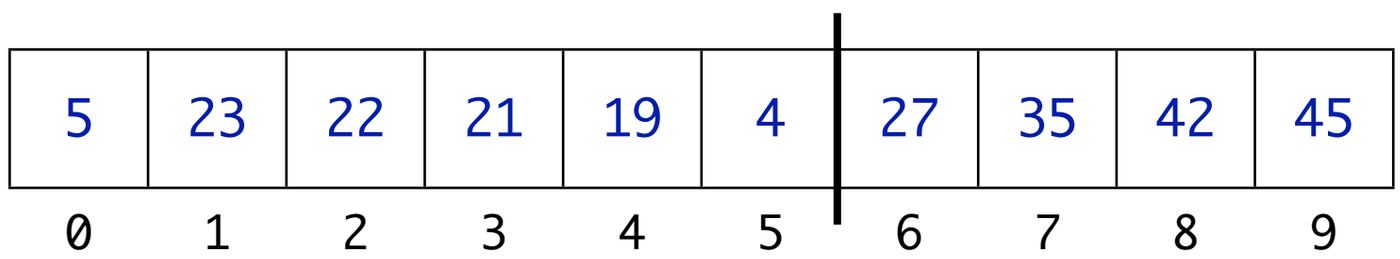




Heapsort

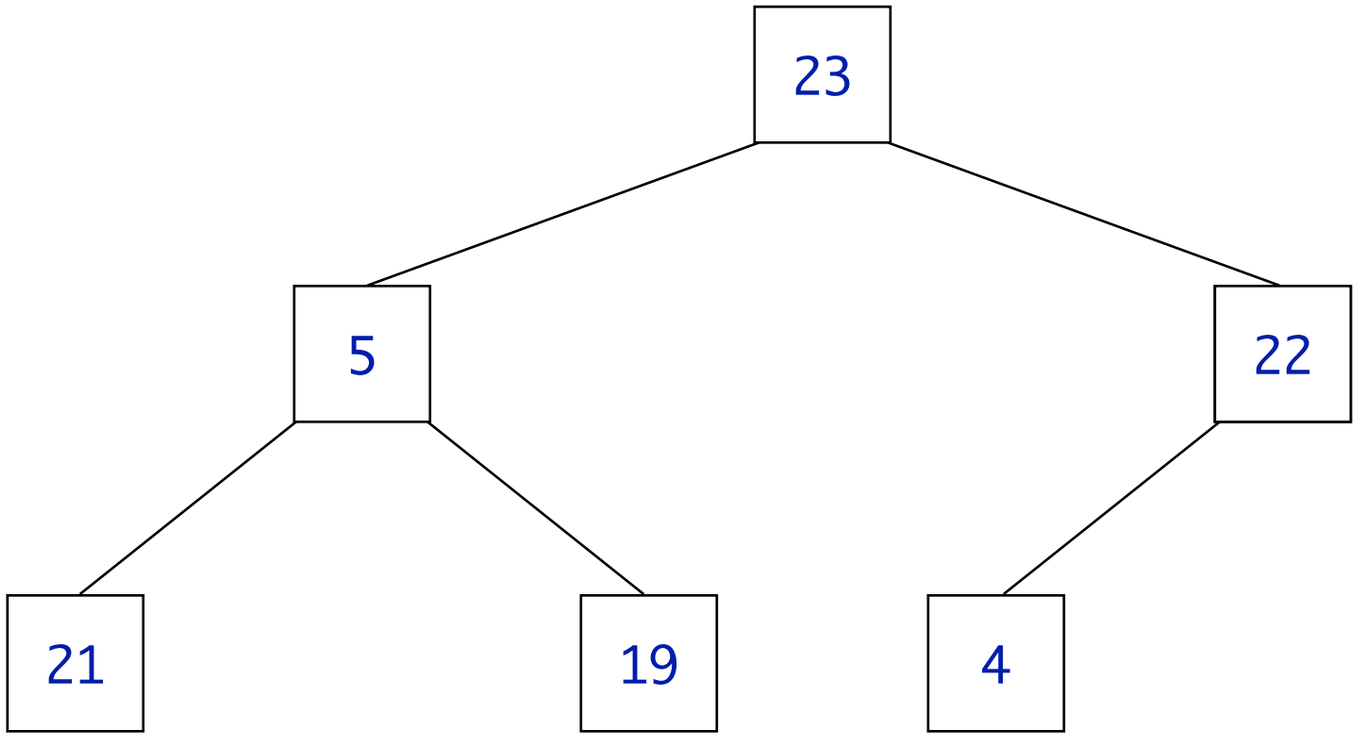
Take largest value and put it in its correct spot...

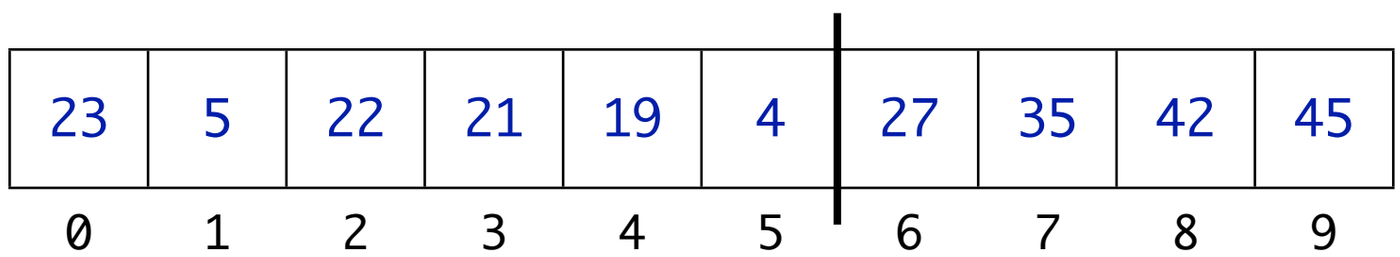




Heapsort

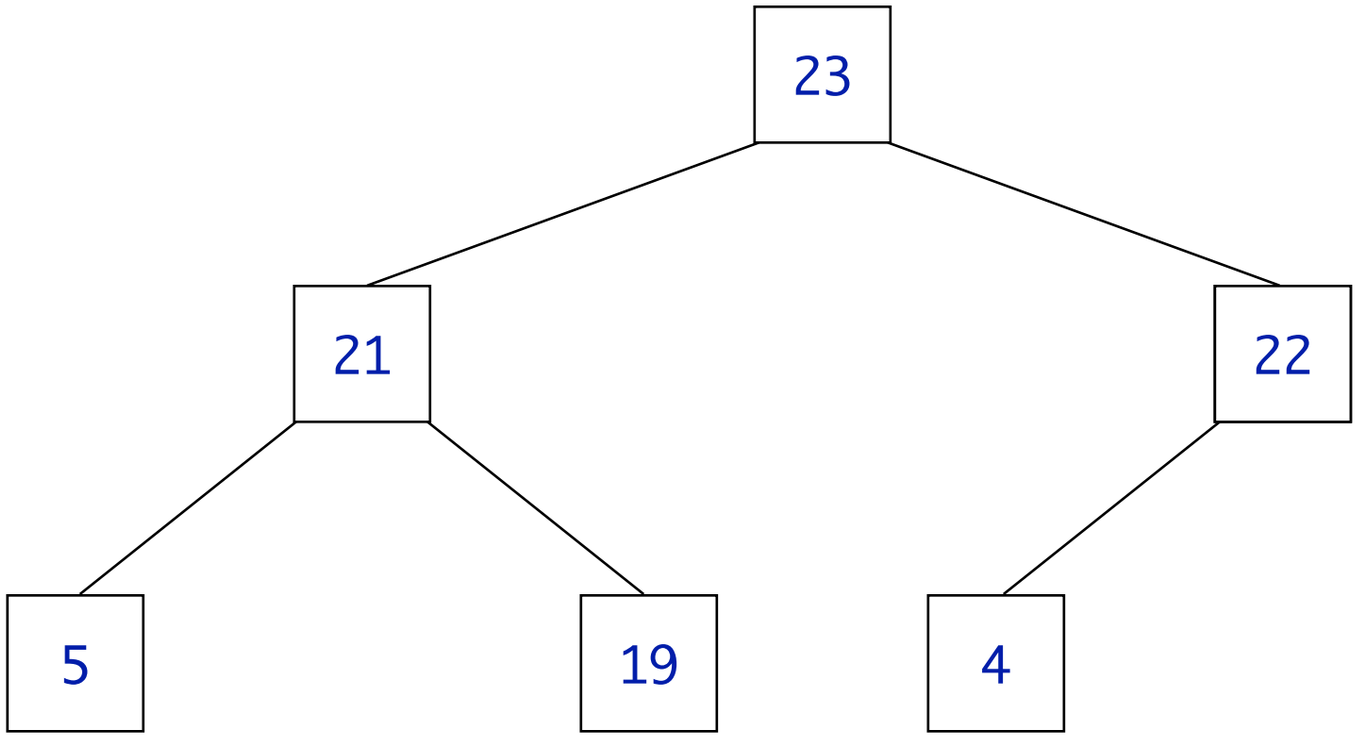
Reheapify downward!

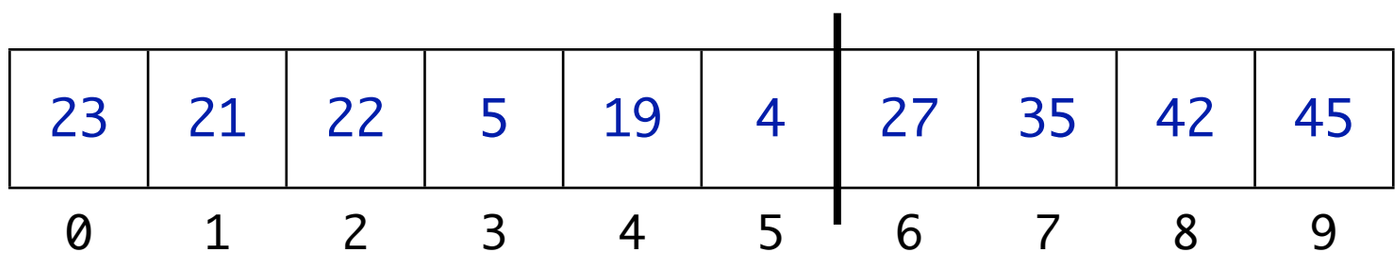




Heapsort

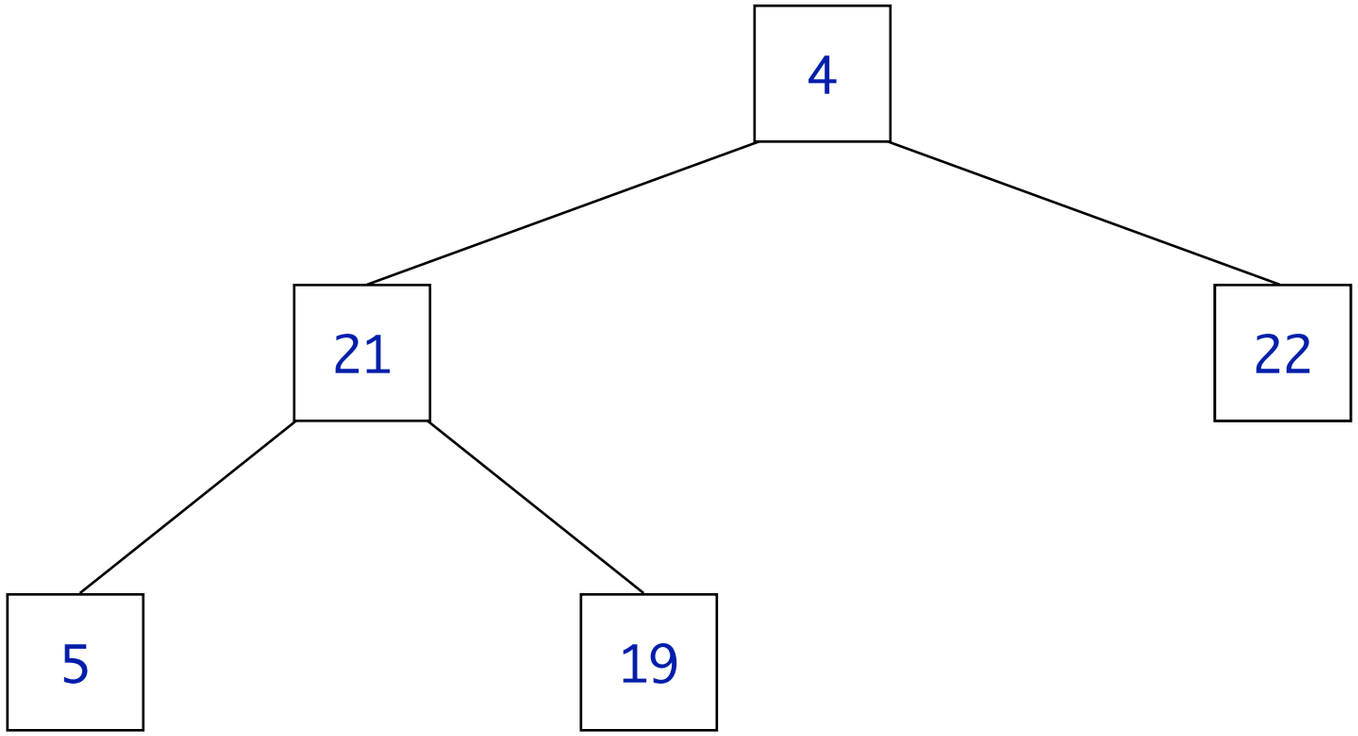
Reheapify downward!

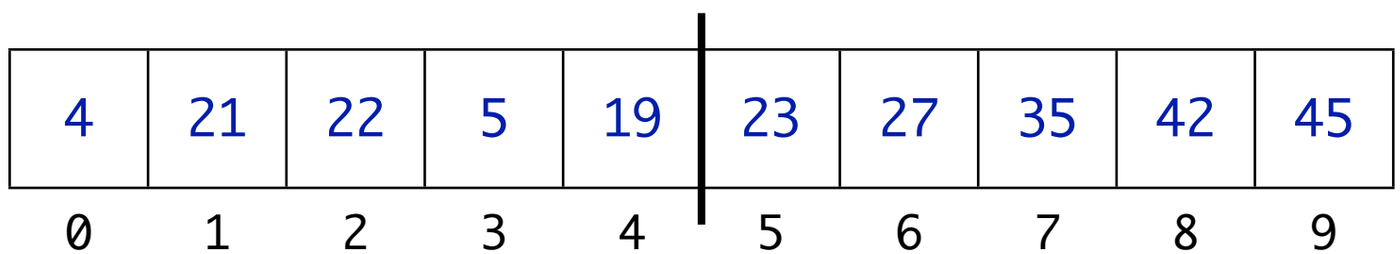




Heapsort

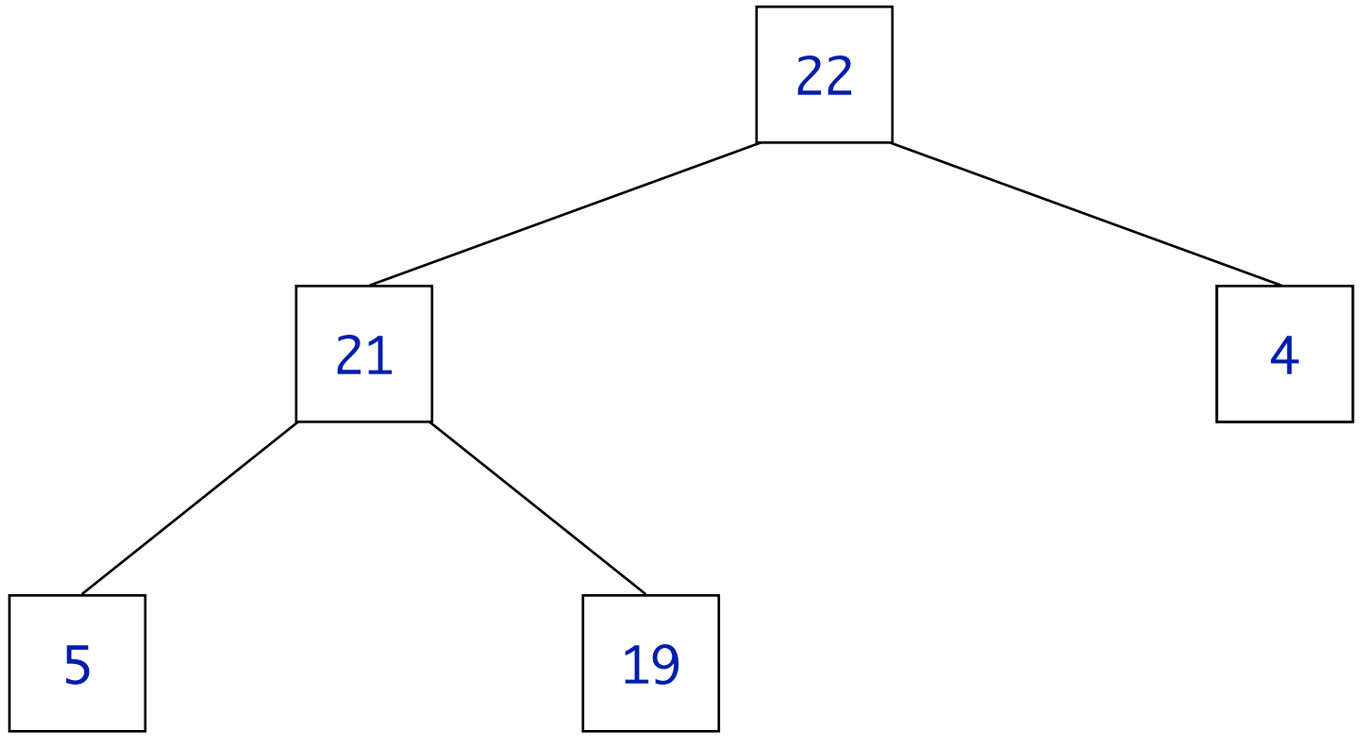
Take largest value and put it in its correct spot...

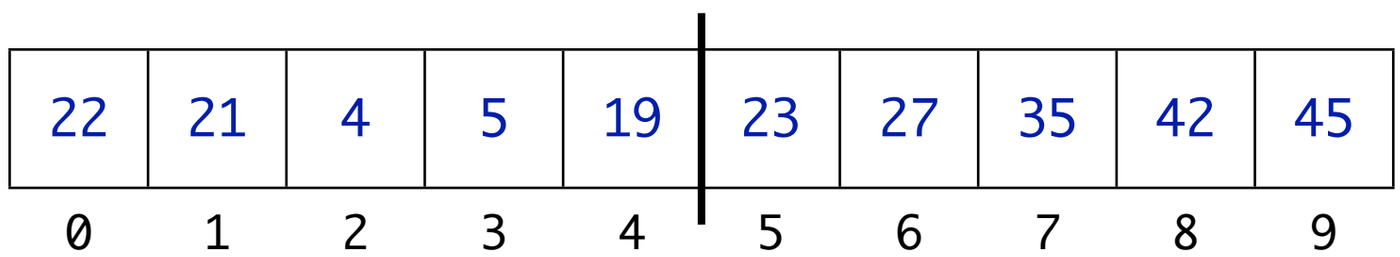




Heapsort

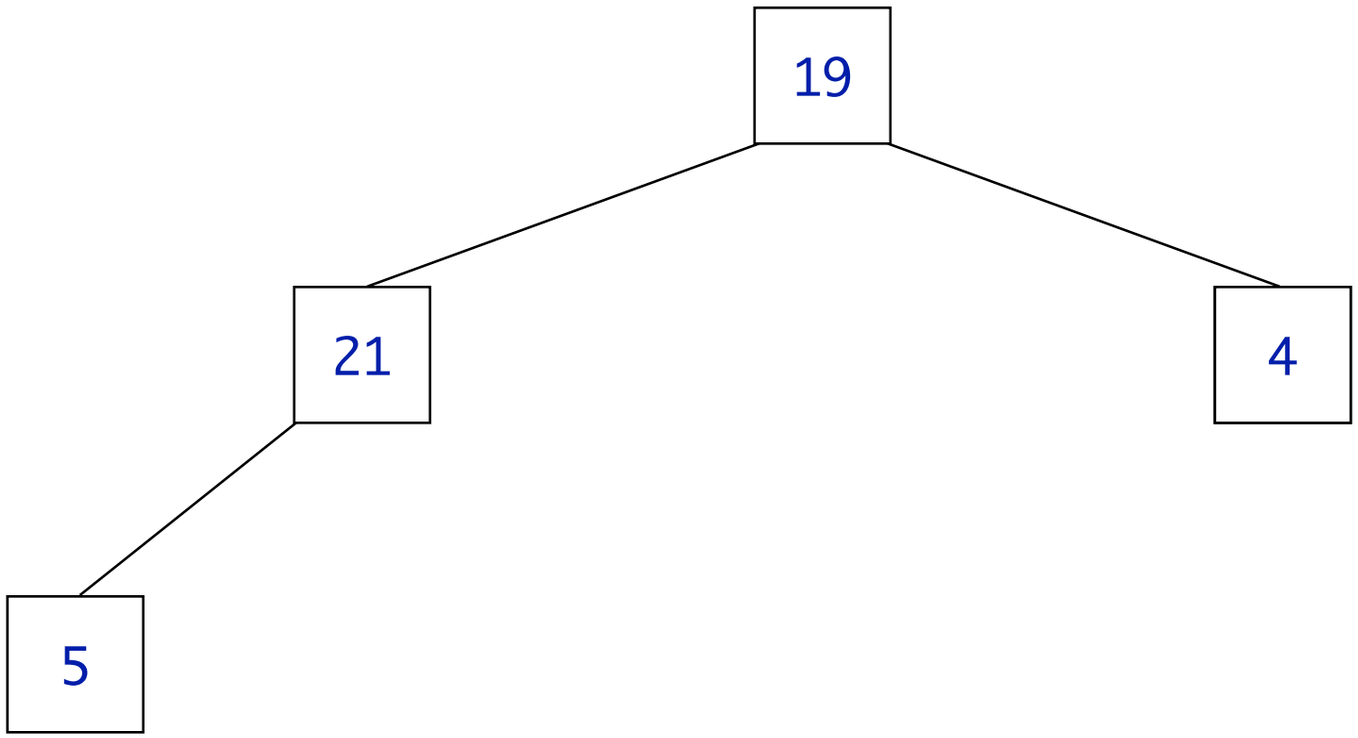
Reheapify downward!

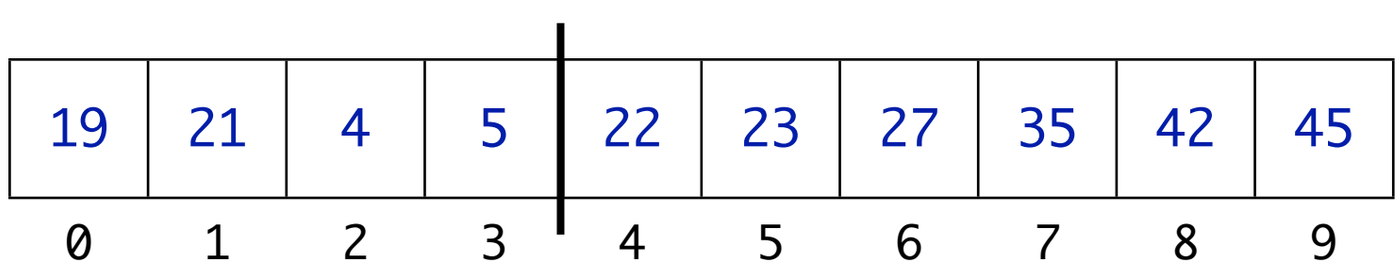




Heapsort

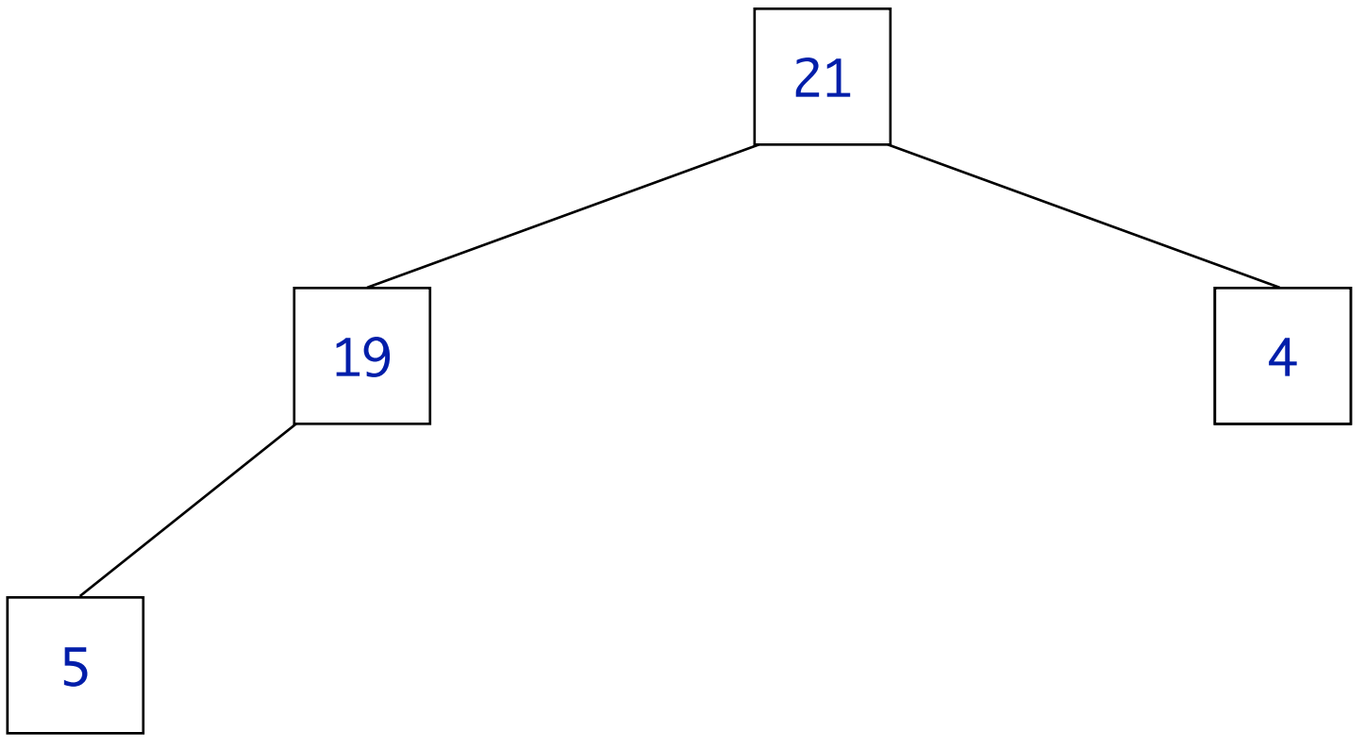
Take largest value and put it in its correct spot...

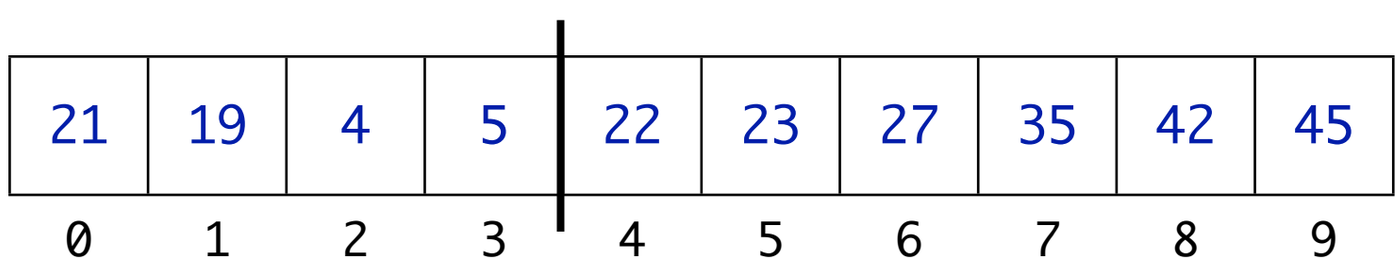




Heapsort

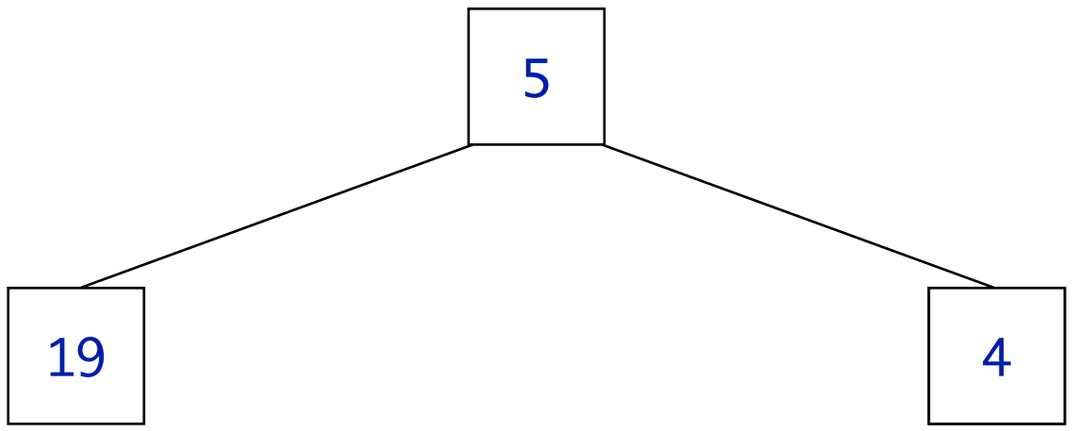
Reheapify downward!

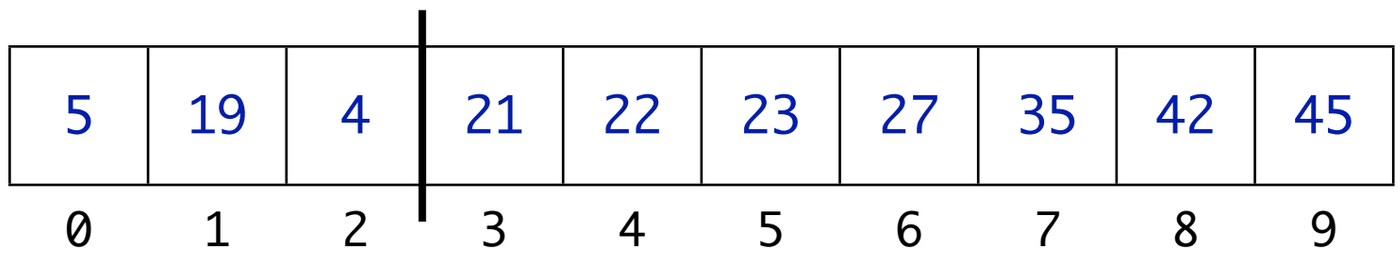




Heapsort

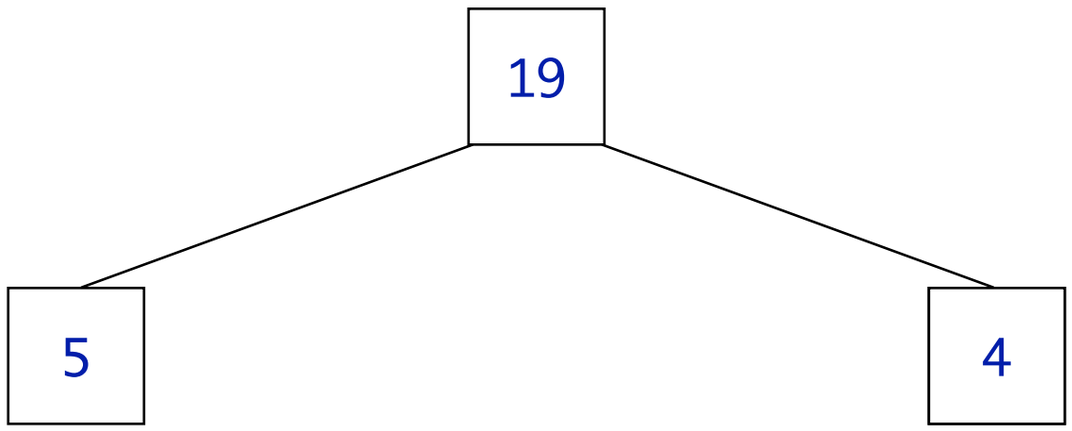
Take largest value and put it in its correct spot...

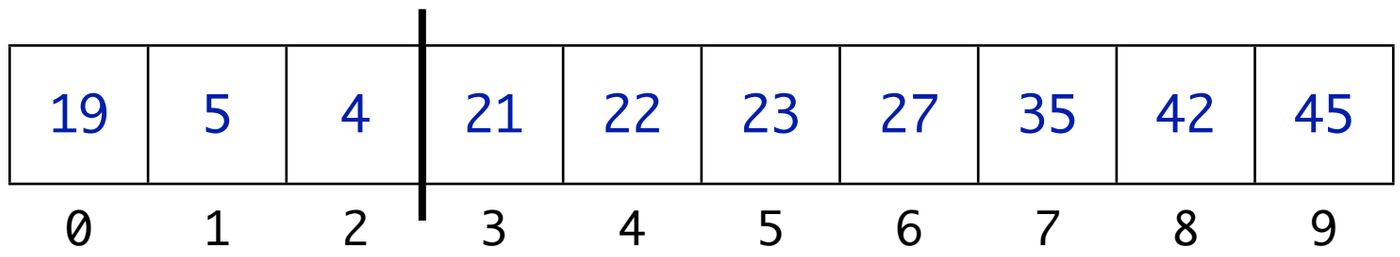




Heapsort

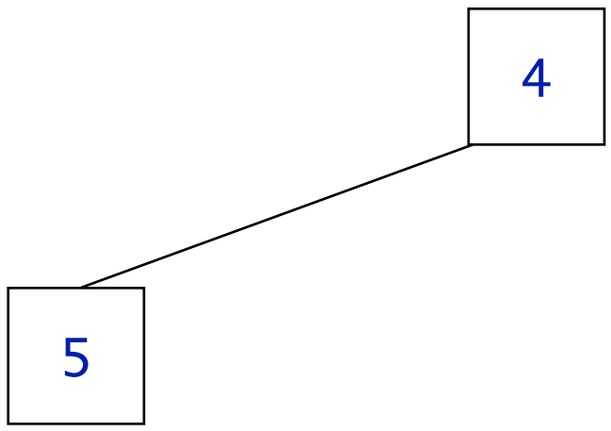
Reheapify downward!

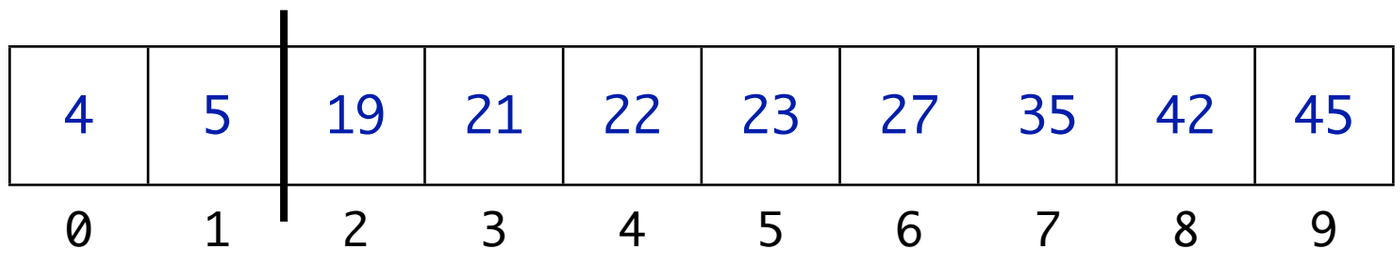




Heapsort

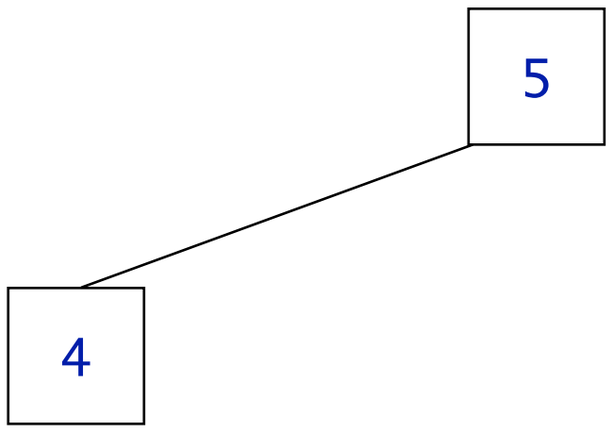
Take largest value and put it in its correct spot...

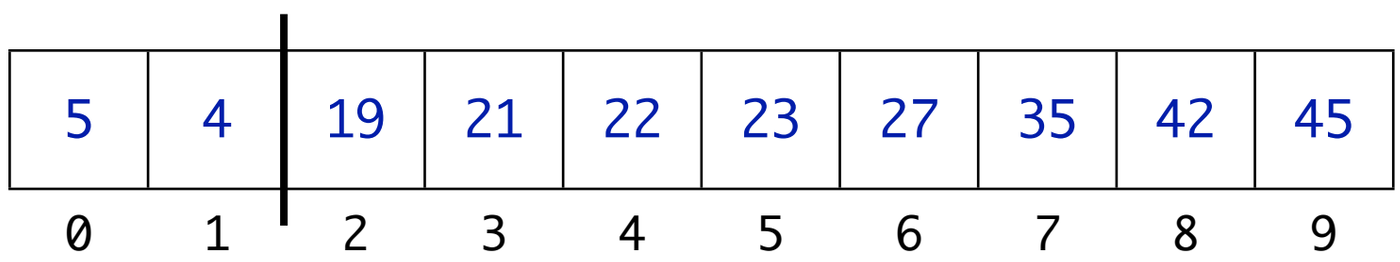




Heapsort

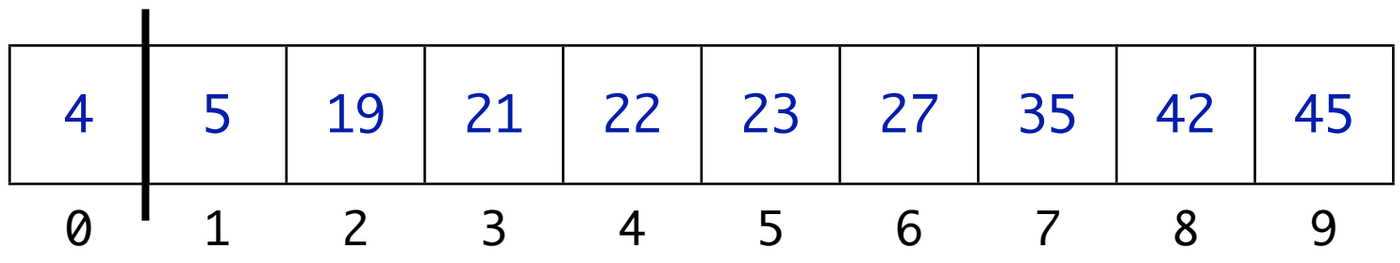
Reheapify downward!





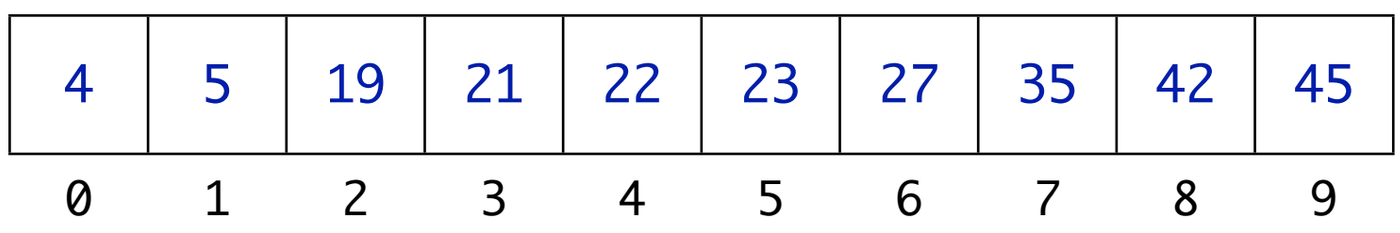
Heapsort

Take largest value and put it in its correct spot...



Heapsort

One value left... All done!



Heapsort

An implementation:

```
template <typename Item, typename SizeType>
void heapsort(Item data[], SizeType size) {
    make_heap(data, size);
    while (size-- > 1) {
        swap(data[0], data[size]);
        reheapify_down(data, size);
    }
}
```

Heapsort

To make the heap:

```
template <typename Item, typename SizeType>
void make_heap(Item data[], SizeType size) {
    for (SizeType i = 1; i < size; i++) {
        SizeType k = i;
        while (k != 0 && data[k] > data[parent(k)]) {
            swap(data[k], data[parent(k)]);
            k = parent(k);
        }
    }
```