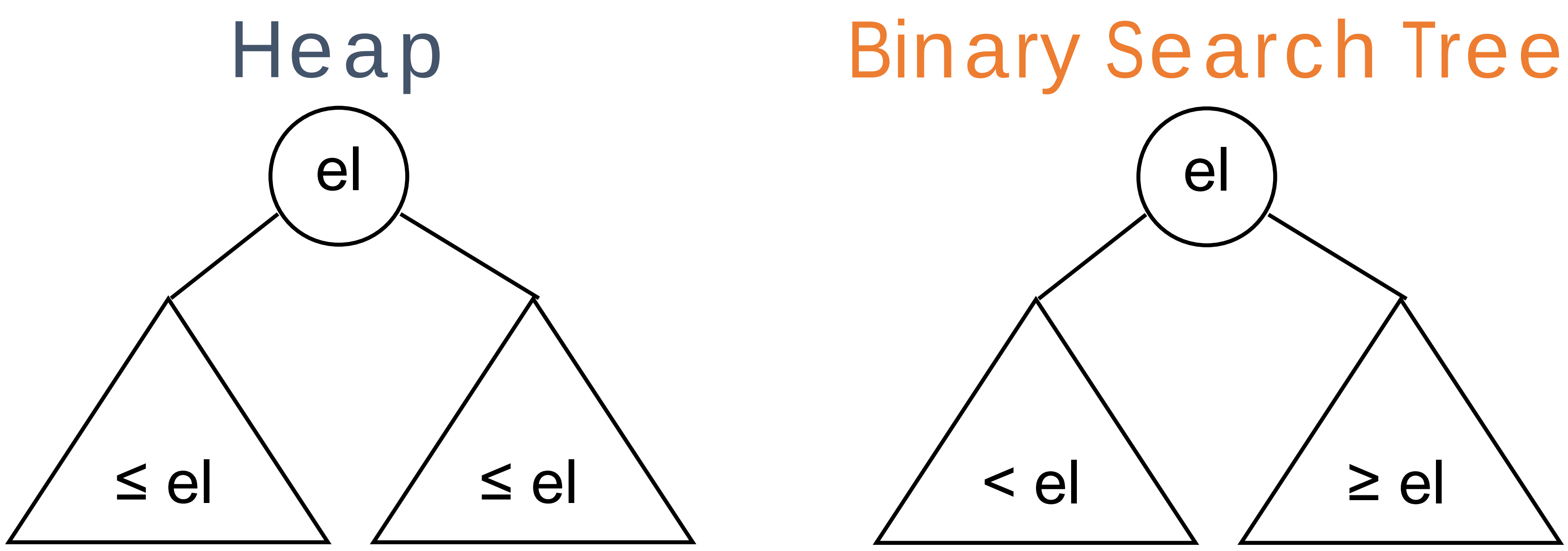
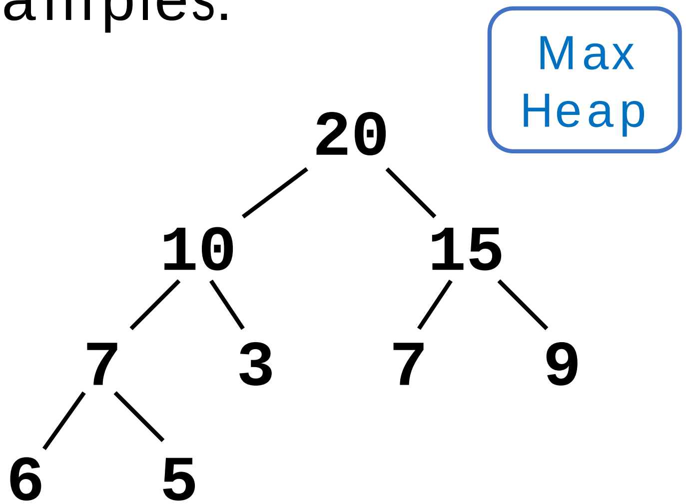
Heaps

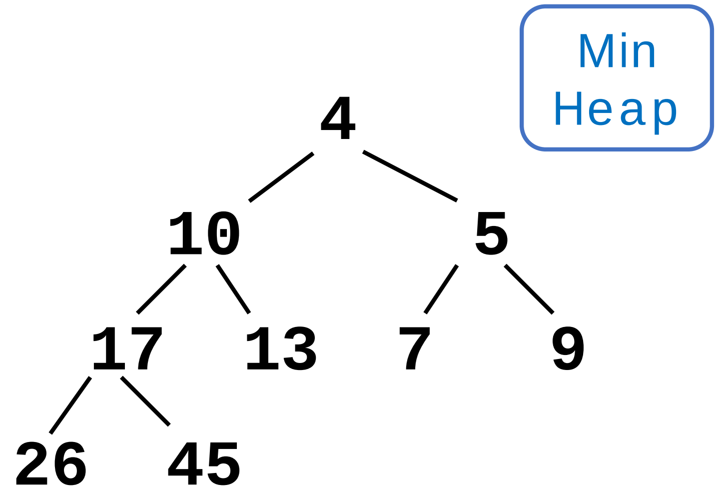
- Another kind of binary tree
- A binary tree is a heap if it satisfies the following two properties:
  1. The value of each node is greater than or equal to the values stored in each of its children
  2. The tree is perfectly balanced and the leaves in the last level are all in the leftmost positions.
- Difference between a BST and a heap:

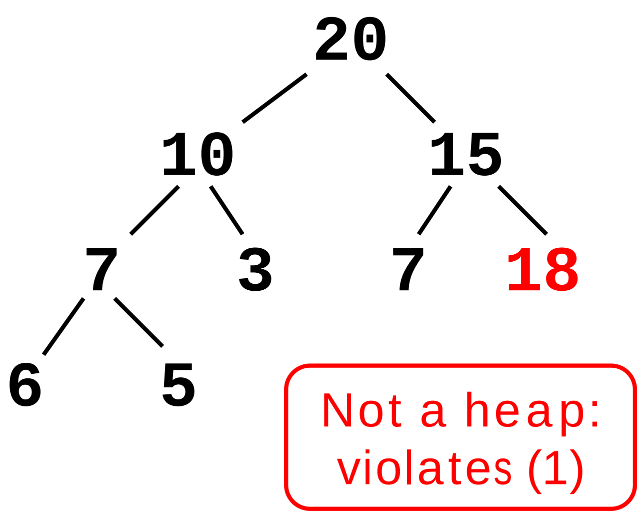


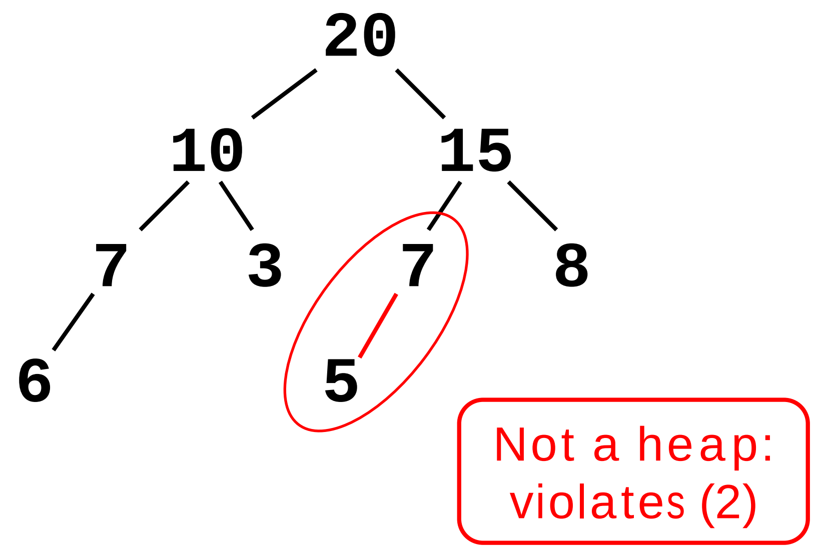
Heaps

- Examples:

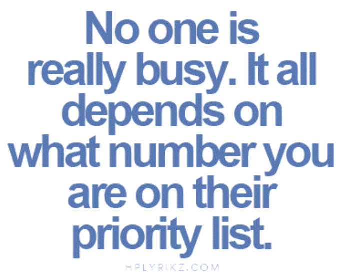




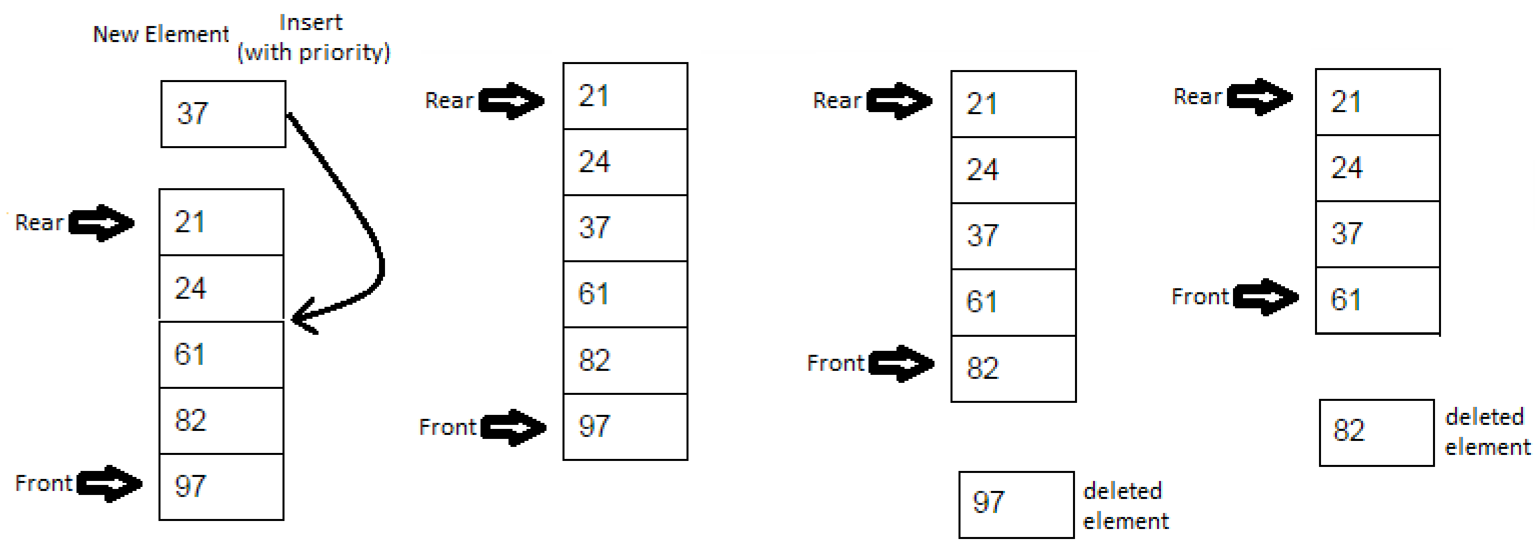




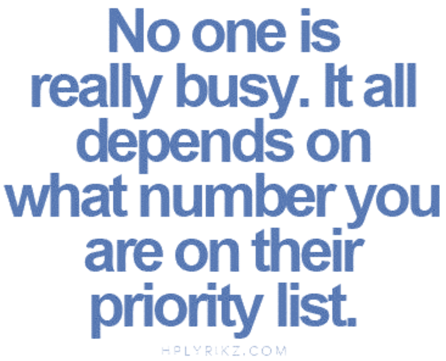
Heaps



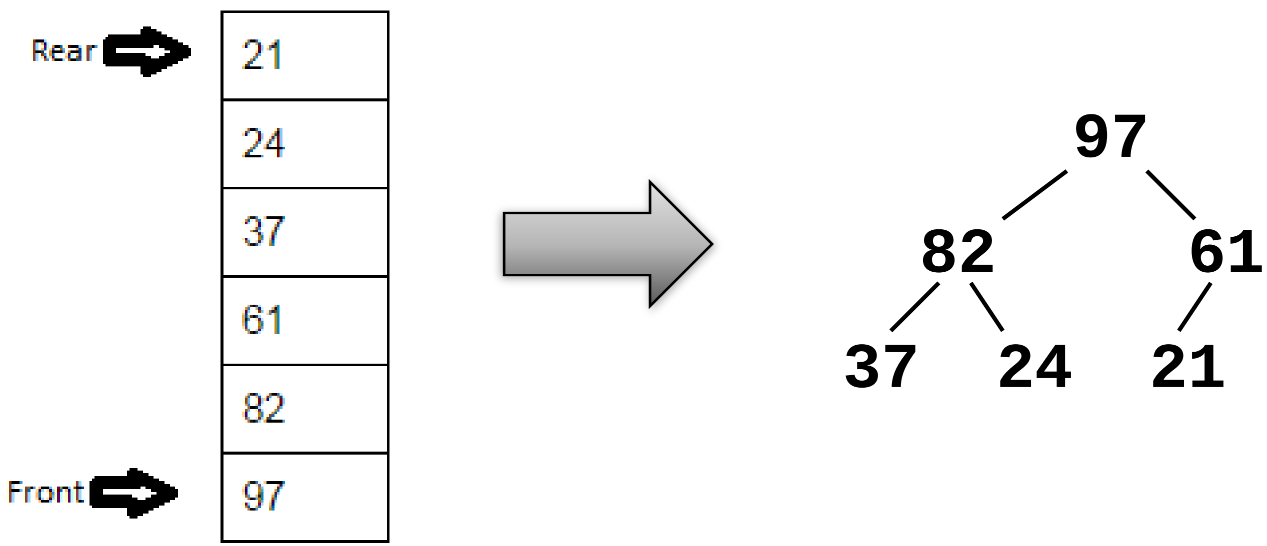
- Why are we interested in heaps?
- Can we perform efficient binary search on heaps?
- What about accessing the smallest/largest element?
  - Max heap provides immediate access to the largest element
  - Min heap provides immediate access to the smallest element
- Can this be useful?
- Consider priority queues:
  - Items are processed on first-come-first-serve basis (FIFO)
  - Every item in the queue has a priority tag
  - Items of higher priority are pushed in front of the items of lower priority
  - Items with the highest priority will always be processed first



Heaps



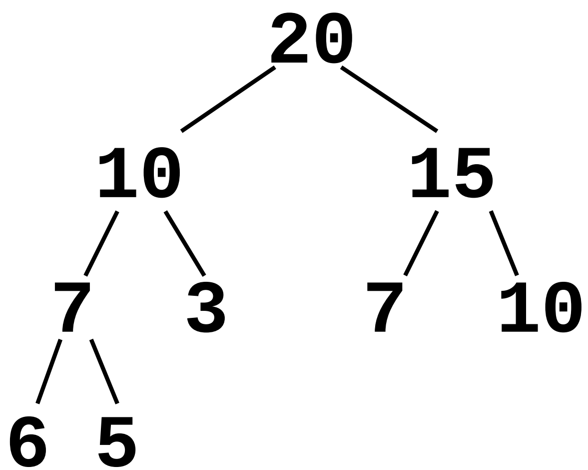
- Heaps make great priority queues:



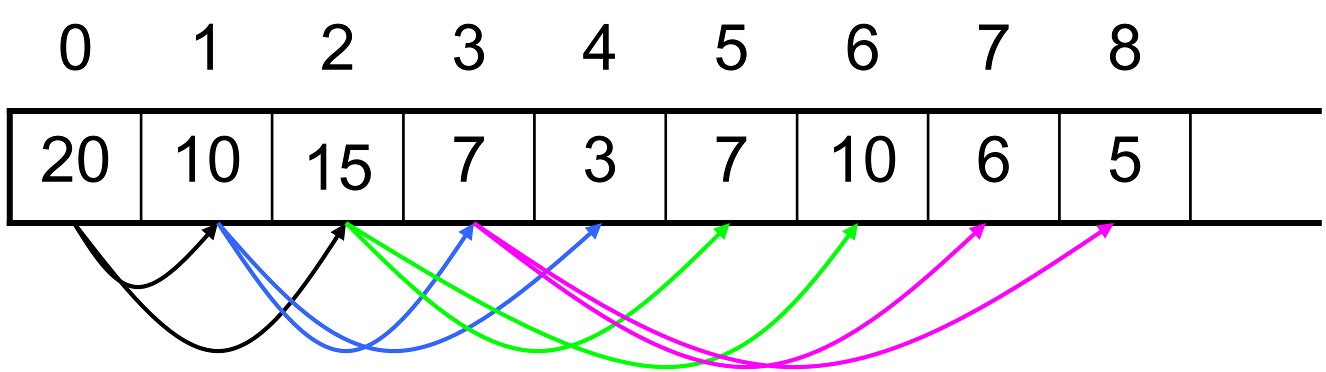
- Root item is the front of the queue
- Tree structure – path to every leaf is *lg n*
- We need enqueue() and dequeue() algorithms!

Heaps

- Heaps are implemented using arrays
- Successive nodes are stored in breadth-first manner

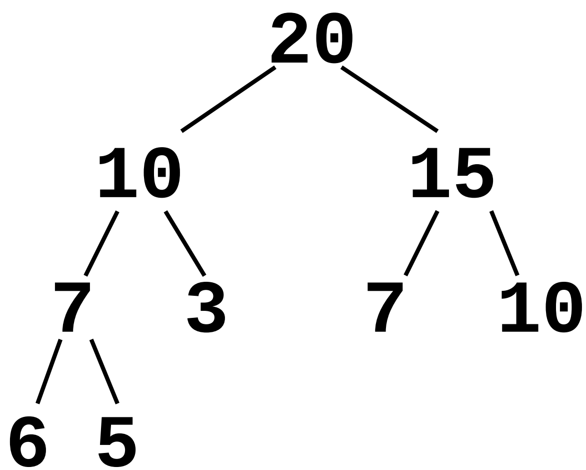


parent	left child	right child
0	1	2
1	3	4
2	5	6
3	7	8
....	....	....
i	$2 \cdot i + 1$	$2 \cdot i + 2$

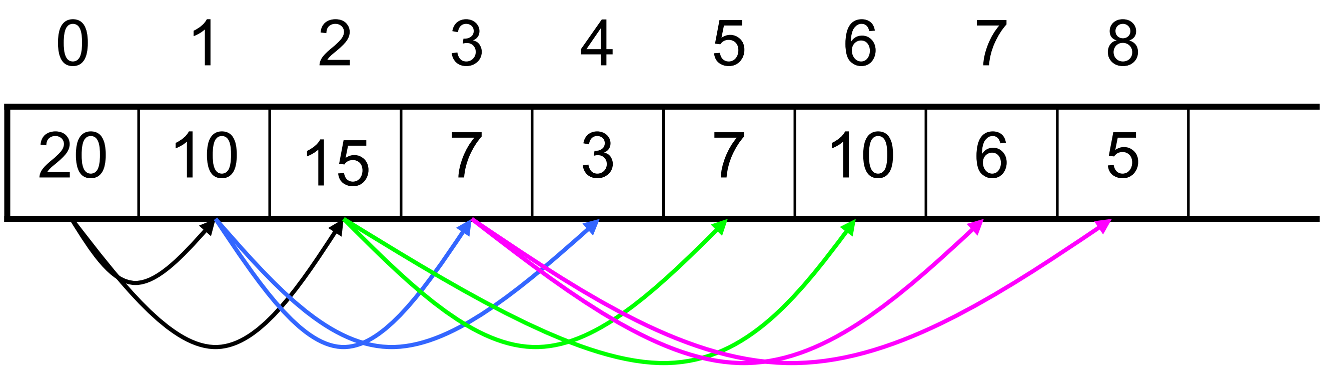


Heaps

- For each index *i*,
  - element *arr*[*i*] has children at *arr*[2*i* + 1] and *arr*[2*i* + 2]
  - and the parent at *arr*[floor((*i* - 1)/2)]



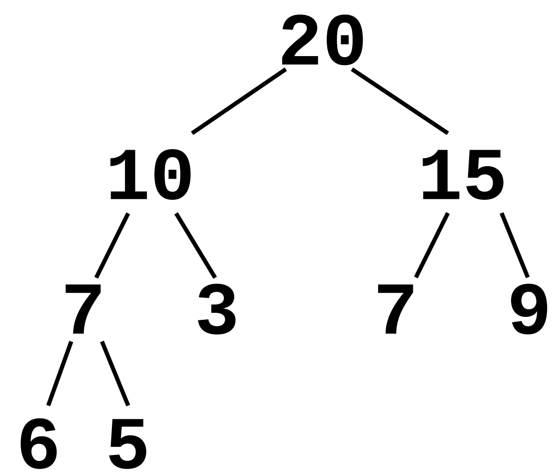
parent	left child	right child
0	1	2
1	3	4
2	5	6
3	7	8
….	….	….
i	$2 \cdot i + 1$	$2 \cdot i + 2$



Heaps as Priority Queues: Enqueuing

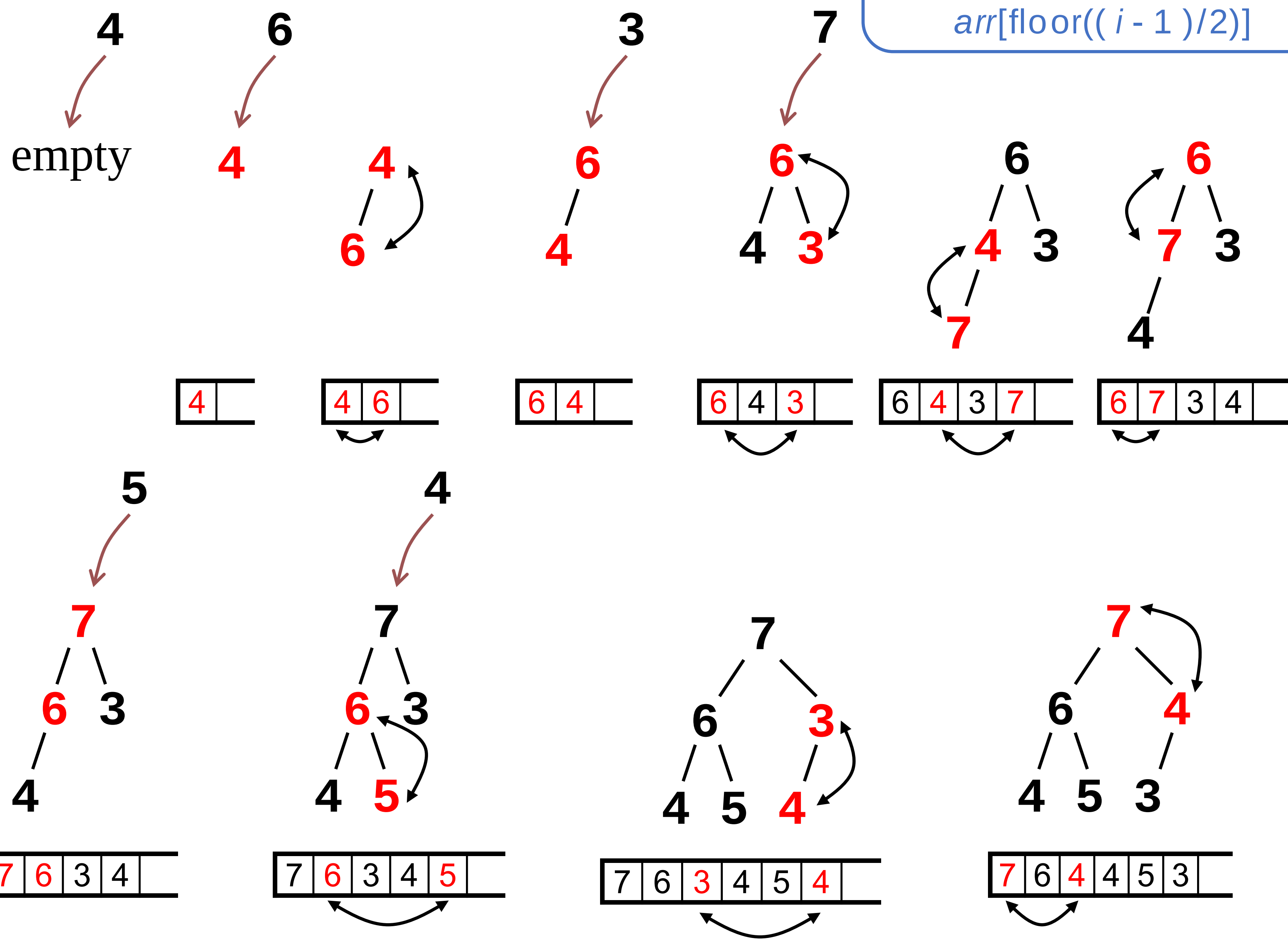
- Enqueue:
  - Add an element
  - Make sure the heap properties are preserved
- What's the best place to add an element?
  - Leftmost leaf position
  - Why? Because property (2) must be preserved
- Enqueue:
  - Add element to the open *leftmost leaf* position
  - Swap the element with its parent while
    element < parent && *element != root*

heapEnqueue(`el`)
  put `el` *at the end of heap*;
  `while` `el` *is not in the root and* `el` > *parent*(`el`)
    swap `el` *with its parent*;



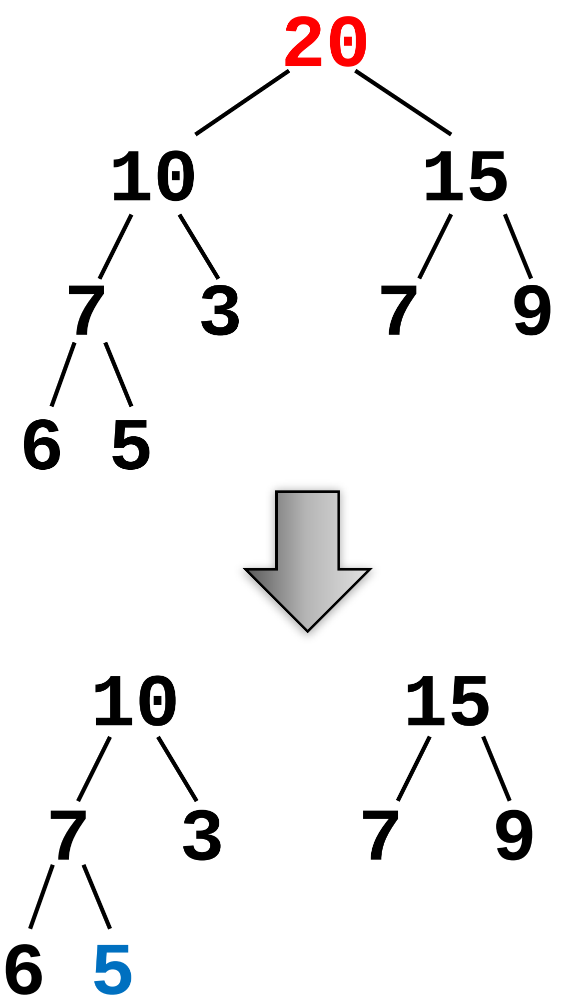
Heaps: Enqueuing

element *arr*[*i*] has children at *arr*[2*i* + 1] and *arr*[2*i* + 2] and parent at *arr*[floor((*i* - 1)/2)]



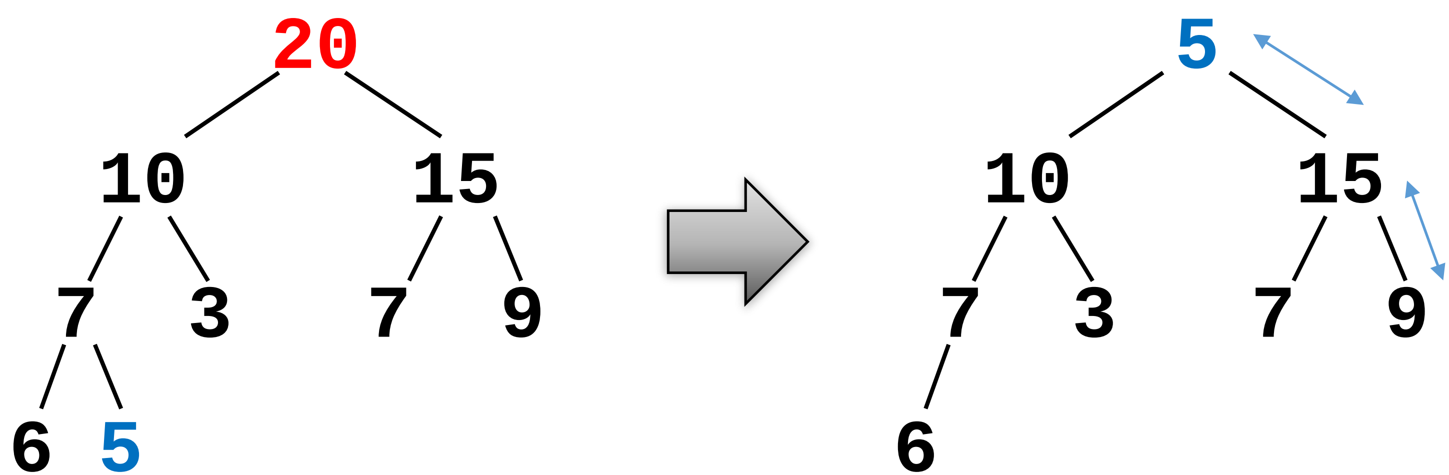
Heaps as Priority Queues: Dequeuing

- Dequeue:
  - Remove root
  - Make sure the heap properties are preserved
- How do we put the two resulting heaps back together?
  - Preserve property (2): take the rightmost leaf, and make it the new root
  - Preserve property (1): swap the p=root with its largest child while (p < largest child)
- Result:
  - Extract root
  - Replace root with the rightmost leaf of the last level
  - Propagate the new root downwards till heap properties are satisfied

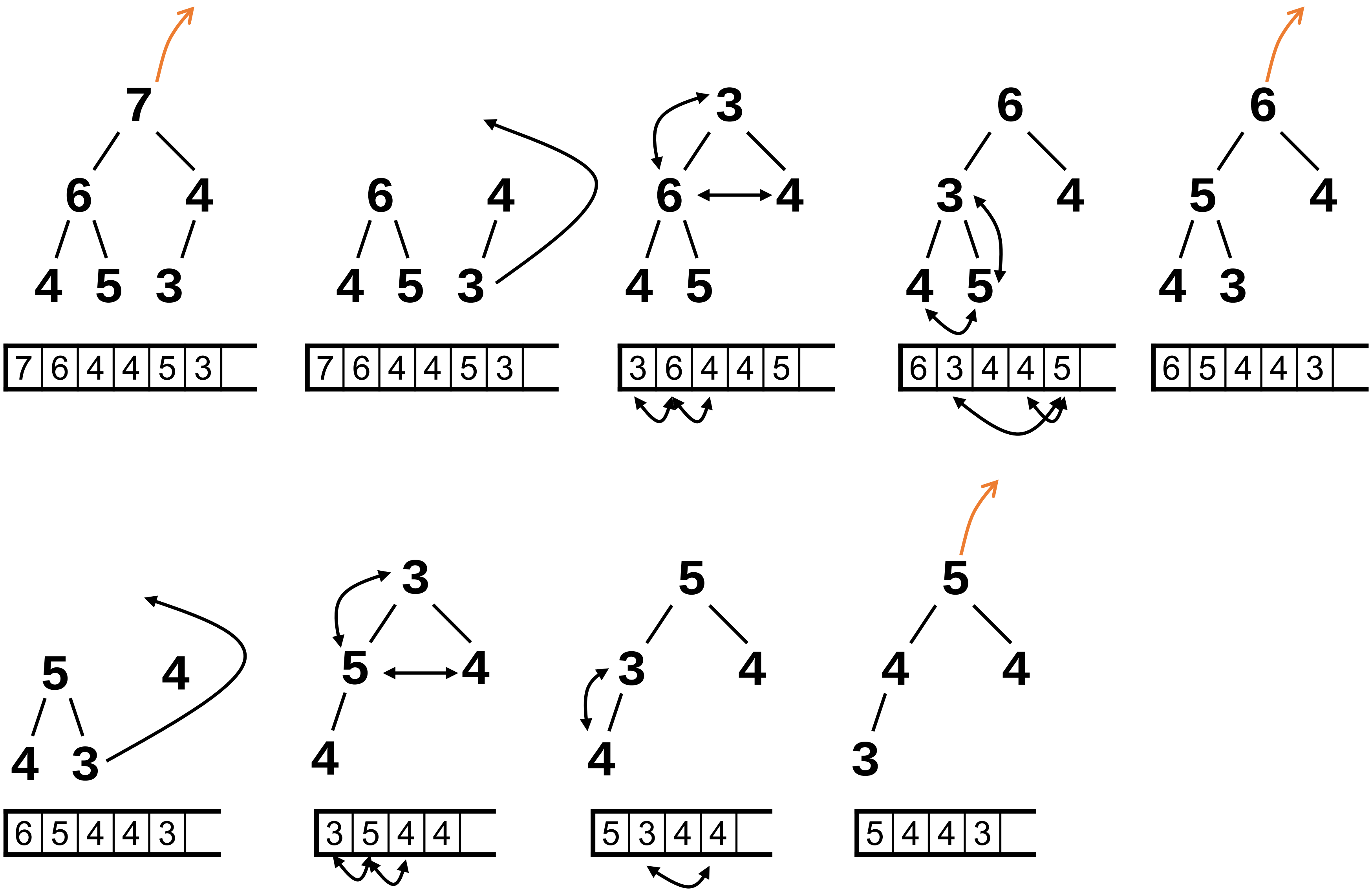


Heaps as Priority Queues: Dequeuing

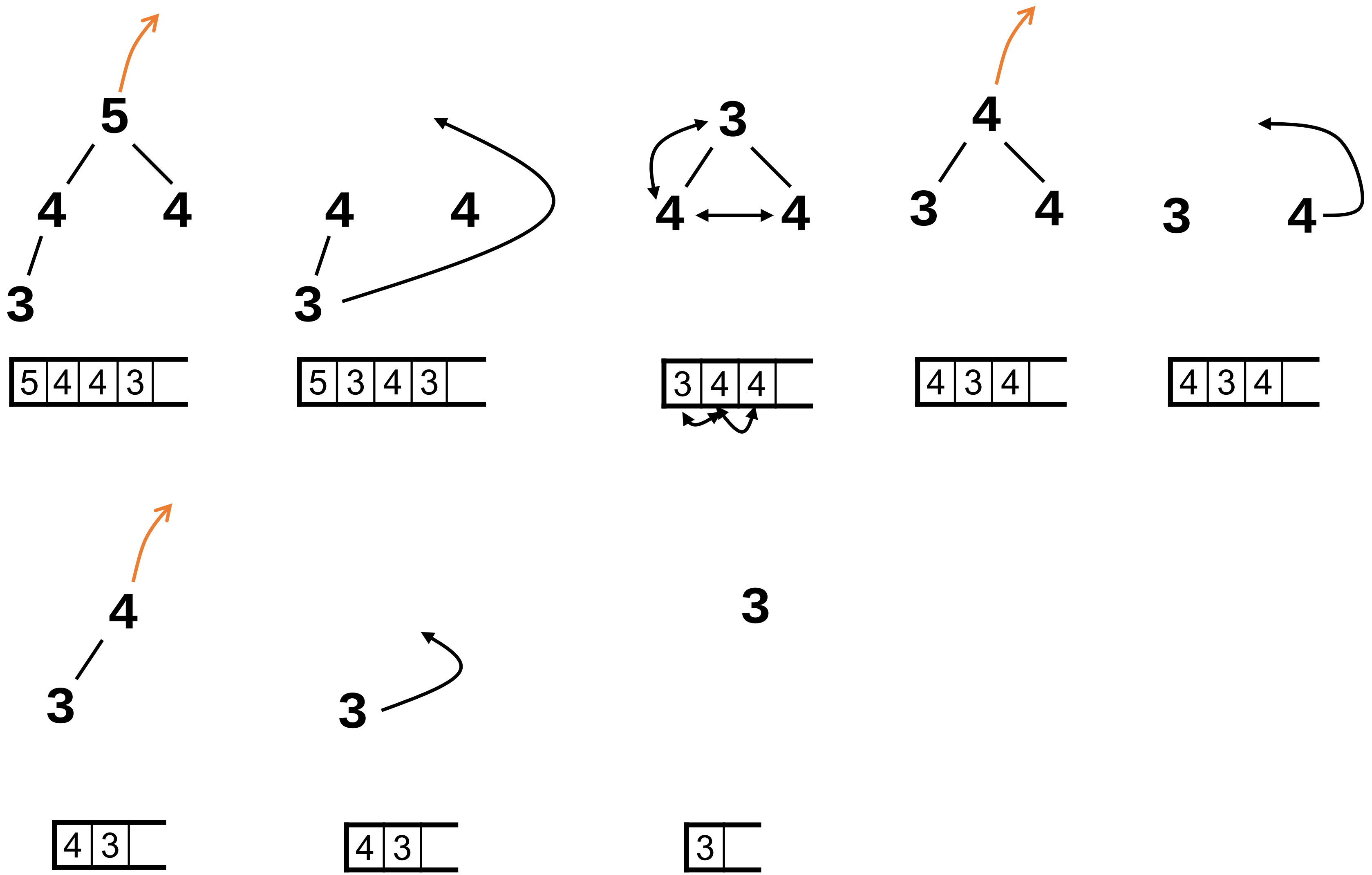
```
heapDequeue()
  extract the element from the root;
  put the element from the last leaf in its place;
  remove the last leaf;
  // both subtrees of the root are heaps
  p = the root;
  while p is not a leaf and p < any of its children
    swap p with the larger child;
```



Heaps as Priority Queues: Dequeuing



Heaps as Priority Queues: Dequeuing



“Heapifying” arrays

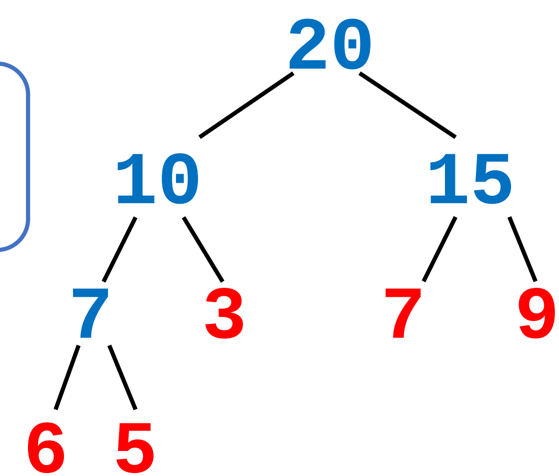
- Given an array of data, how do you convert it into a heap?
- Williams algorithm:
  - Create an empty heap
  - Insert elements from the array to the heap one by one
- Efficiency:
  - To insert n elements: n insertions
  - To insert a single element: maximum of lg n “percolations”
  - Thus: O(n lg n)
- Is there a better way?
- Floyd algorithm:
  - Assume the given array is already a heap
  - Find the last non-leaf node p, set start=p
  - Enforce heap property (1) by moving p down as far as necessary
  - Go to previous non-leaf node
  - Repeat while p >= 0 (move from last non-leaf to first non-leaf)

"Heapifying" arrays

```
FloydAlgorithm(data[])
  i = index of the last nonleaf
  while i >= 0
    p = data[i];
    while p is not a leaf and p < any of its children
      swap p with the larger child;
    i = i--; // index of the previous non-leaf
```

- How do we find the last non-leaf?

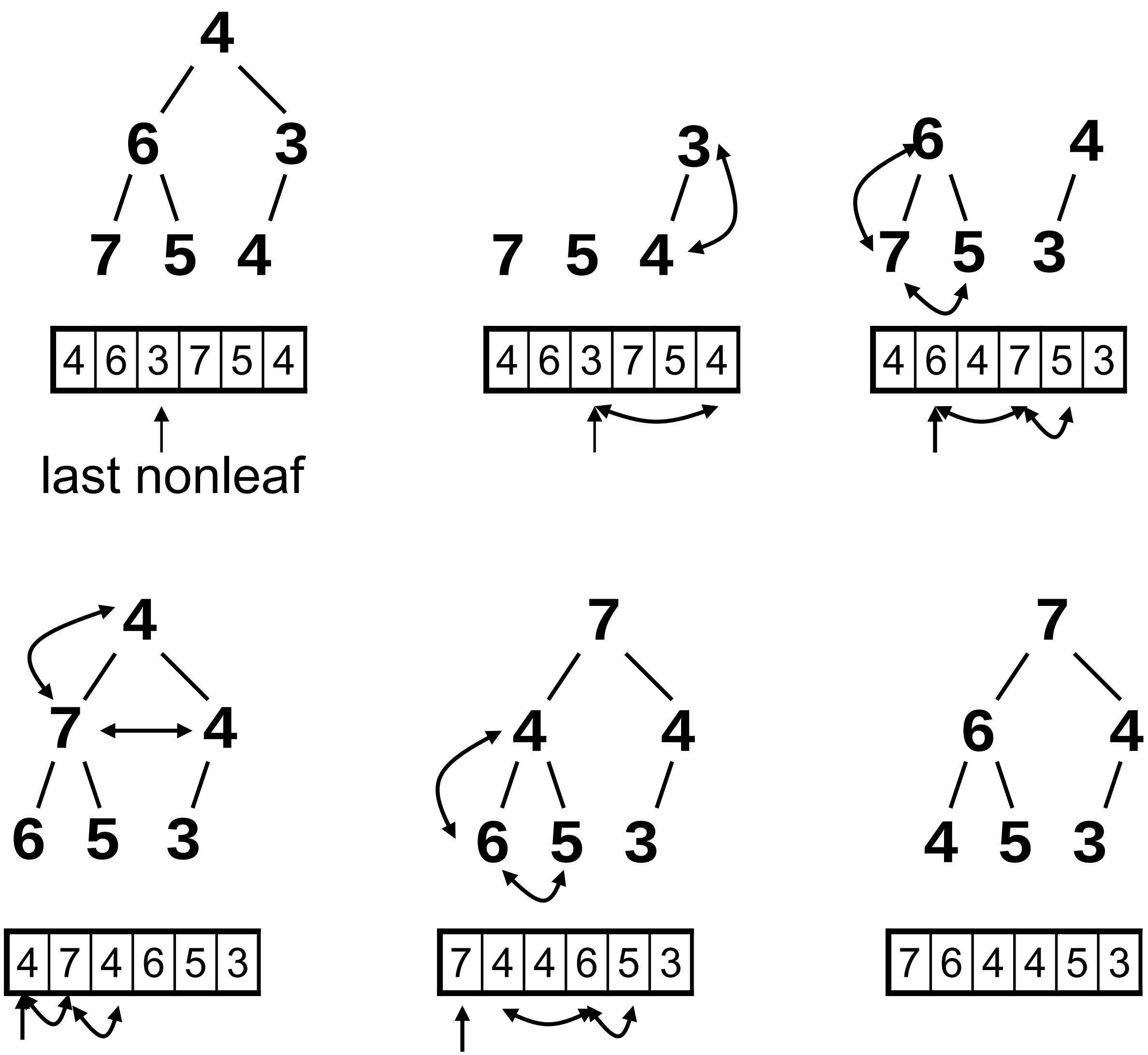
> element *arr*[*i*] has children at *arr*[2*i* + 1] and *arr*[2*i* + 2]
> and parent at *arr*[floor((*i* - 1)/2)]



- Thus, take the index of the last element, and calculate the parent:
  - Index of 5: 8
  - Parent of 5: floor((8 - 1)/2) = 3
  - Value at index 3: 7

20	10	15	7	3	7	9	6	5
0	1	2	3	4	5	6	7	8

“Heapifying” arrays: Floyd’s Algorithm



Efficiency: O(n)

That’s better than O(n lg n)!