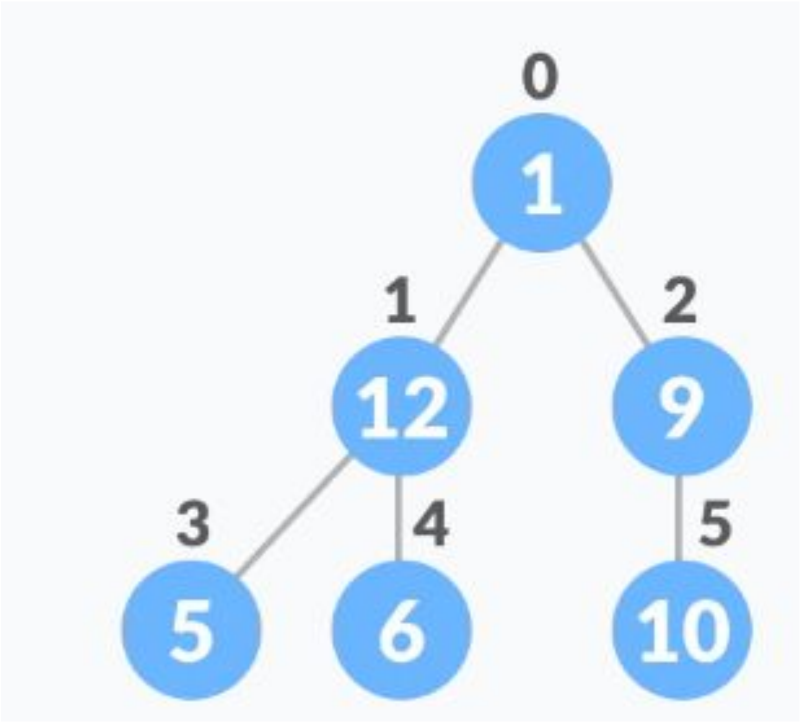
Heapsort

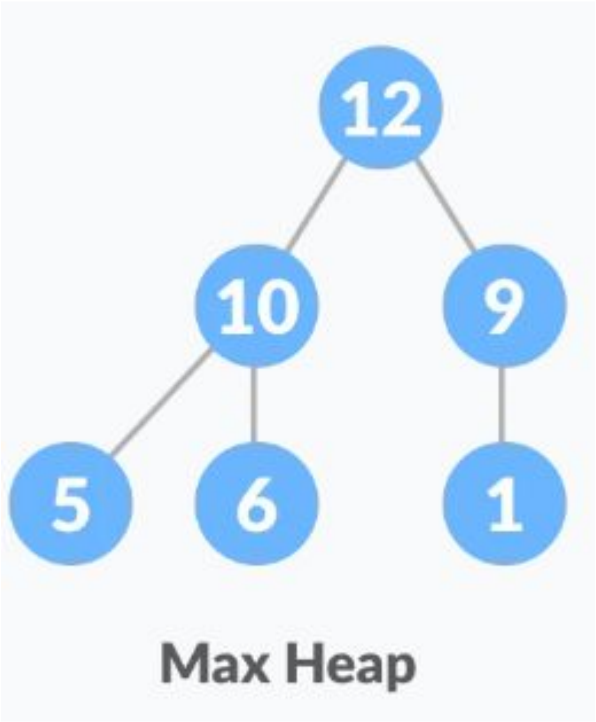
Relationship between Array Indexes and Tree Elements

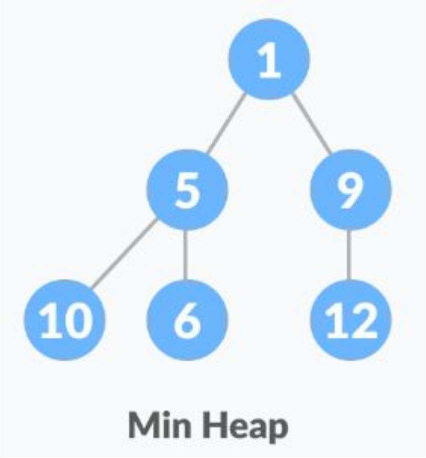
- A complete binary tree has an interesting property that we can use to find the children and parents of any node.
- If the index of any element in the array is i
  - the element in the index 2i+1 will become the left child
  - the element in the index 2i+2 will become the right child
- Also, the parent of any element at index i is given by the lower bound of (i-1)/2.



Heap Data Structure

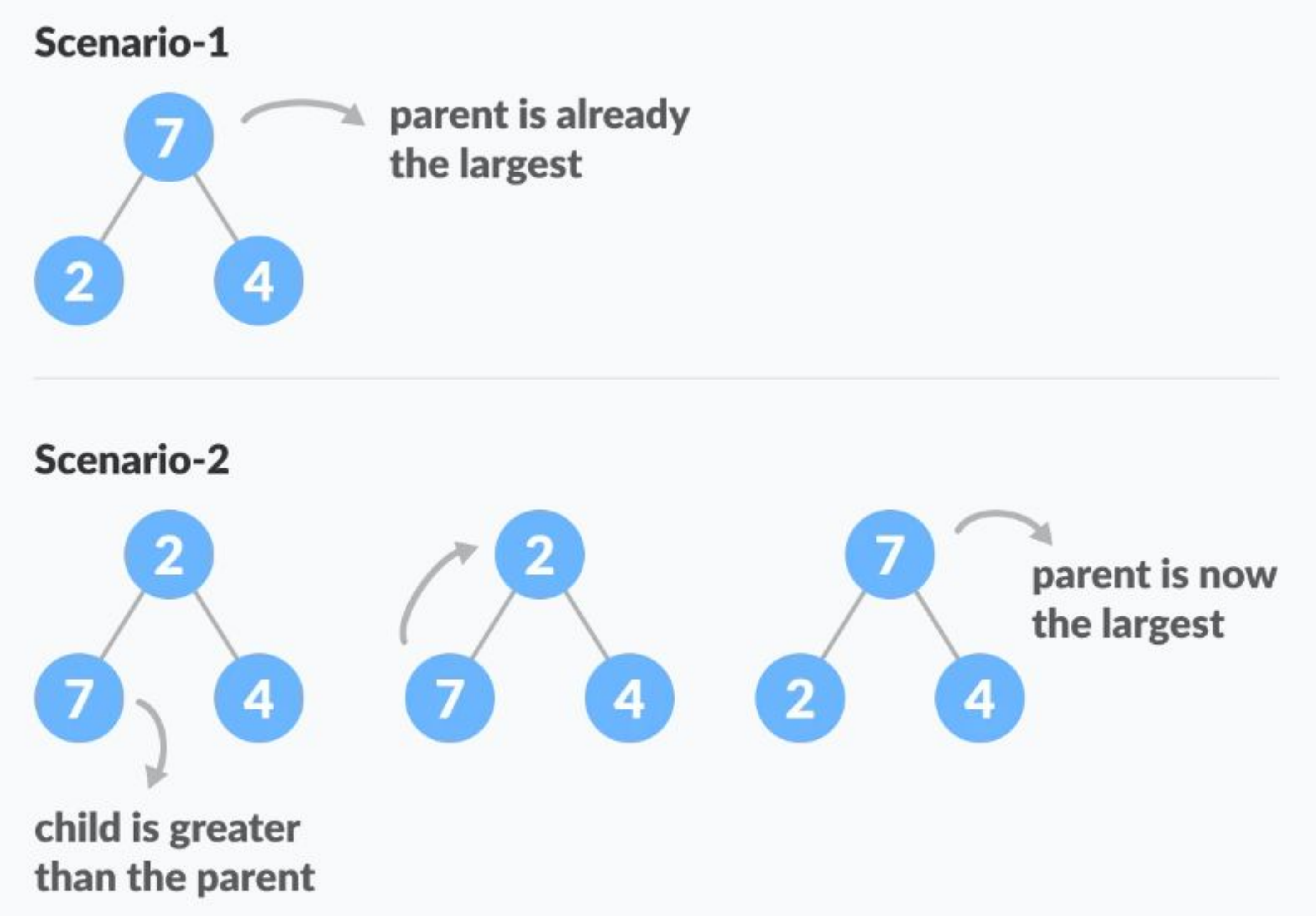
- Heap is a special tree-based data structure
- A binary tree is said to follow a heap data structure if
  - it is a complete binary tree
  - All nodes in the tree follow the property that
    - They are greater than their children : Max-Heap
    - Or all nodes are smaller than their children : Min-Heap

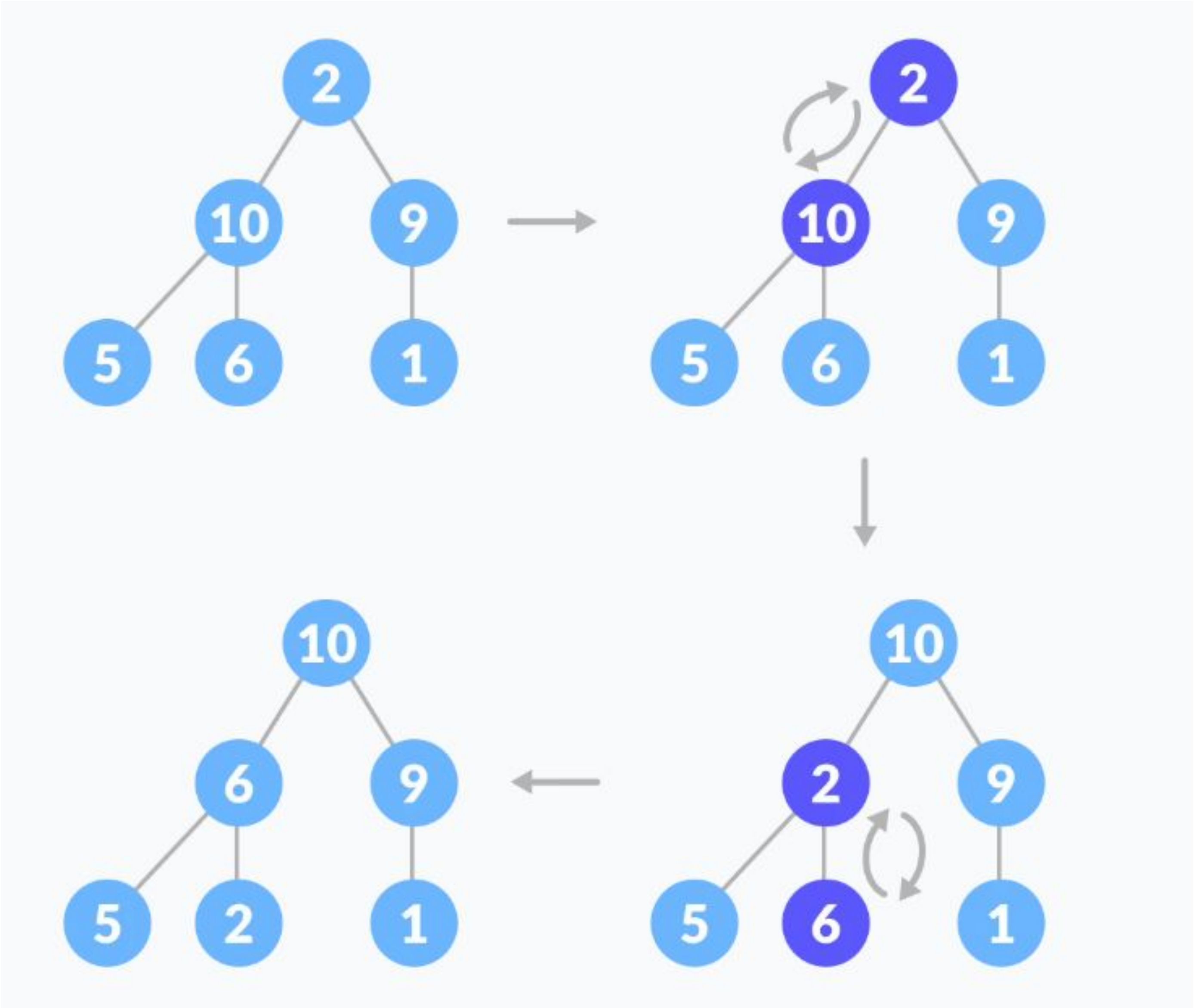




What is **Heapify**

- Heapify is **the process of creating a heap data structure from a binary tree**.




2
10
9
5
6
1
2
10
9
5
6
1
10
6
9
5
2
1
10
2
9
5
6
1

Max-Heapify Operation

Algorithm 1: Max-Heapify Pseudocode

Data: B: input array; s: an index of the node
Result: Heap tree that obeys max-heap property

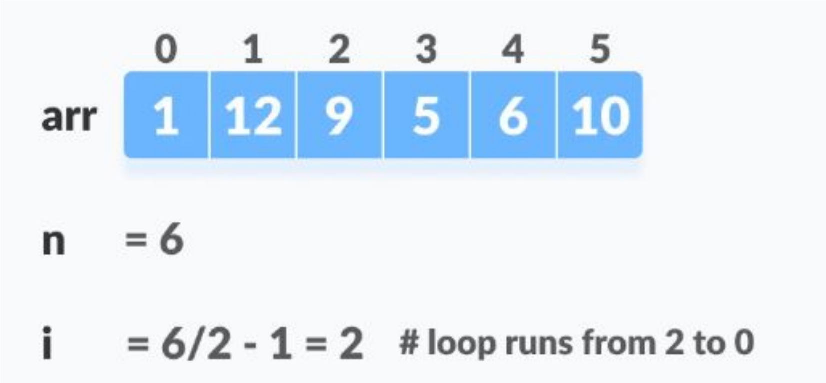
```
Procedure Max-Heapify(B, s)
    left = 2s;
    right = 2s + 1;
    if left ≤ B.length and B[left] > B[s] then
        largest = left;
    else
        largest = s;
    end
    if right ≤ B.length and B[right] > B[largest] then
        largest = right;
    end
    if largest ≠ s then
        swap(B[s], B[largest]);
        Max-Heapify(B, largest);
    end
end
```

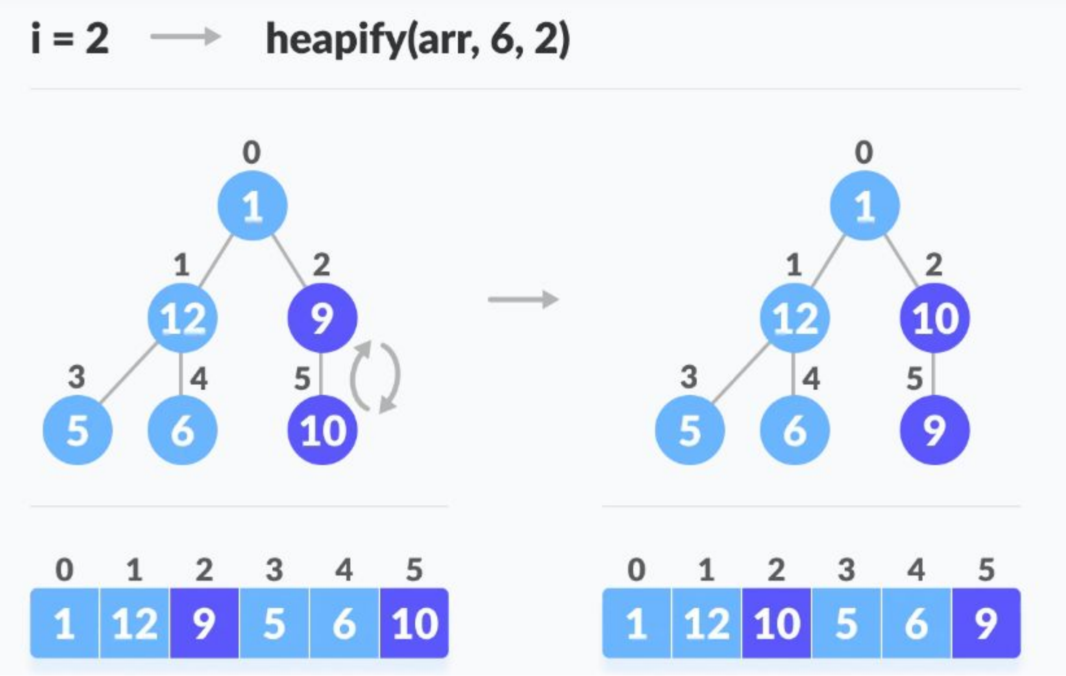
Building **max-heap**

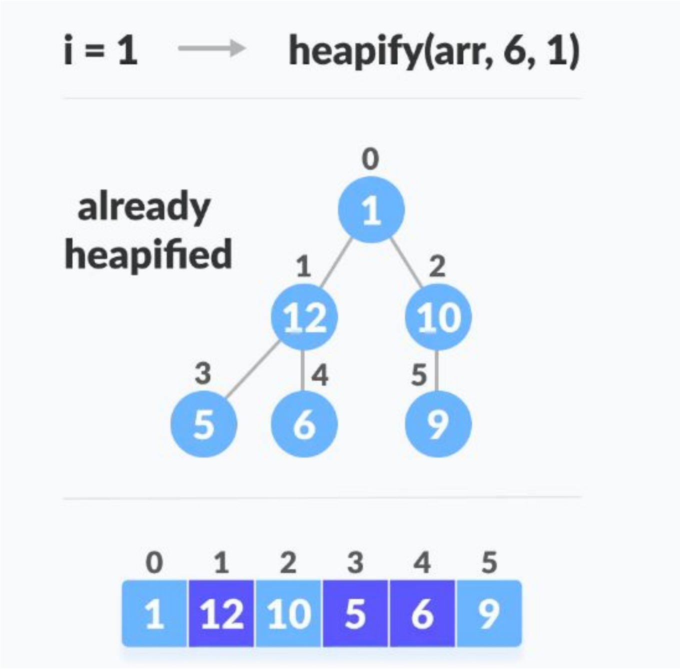
- To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up to the root element.
- Start our algorithm with a node that is at the lowest level of the tree and has children node (n/2 - 1)
- Continue this process and make sure all the subtrees are following the max-heap property

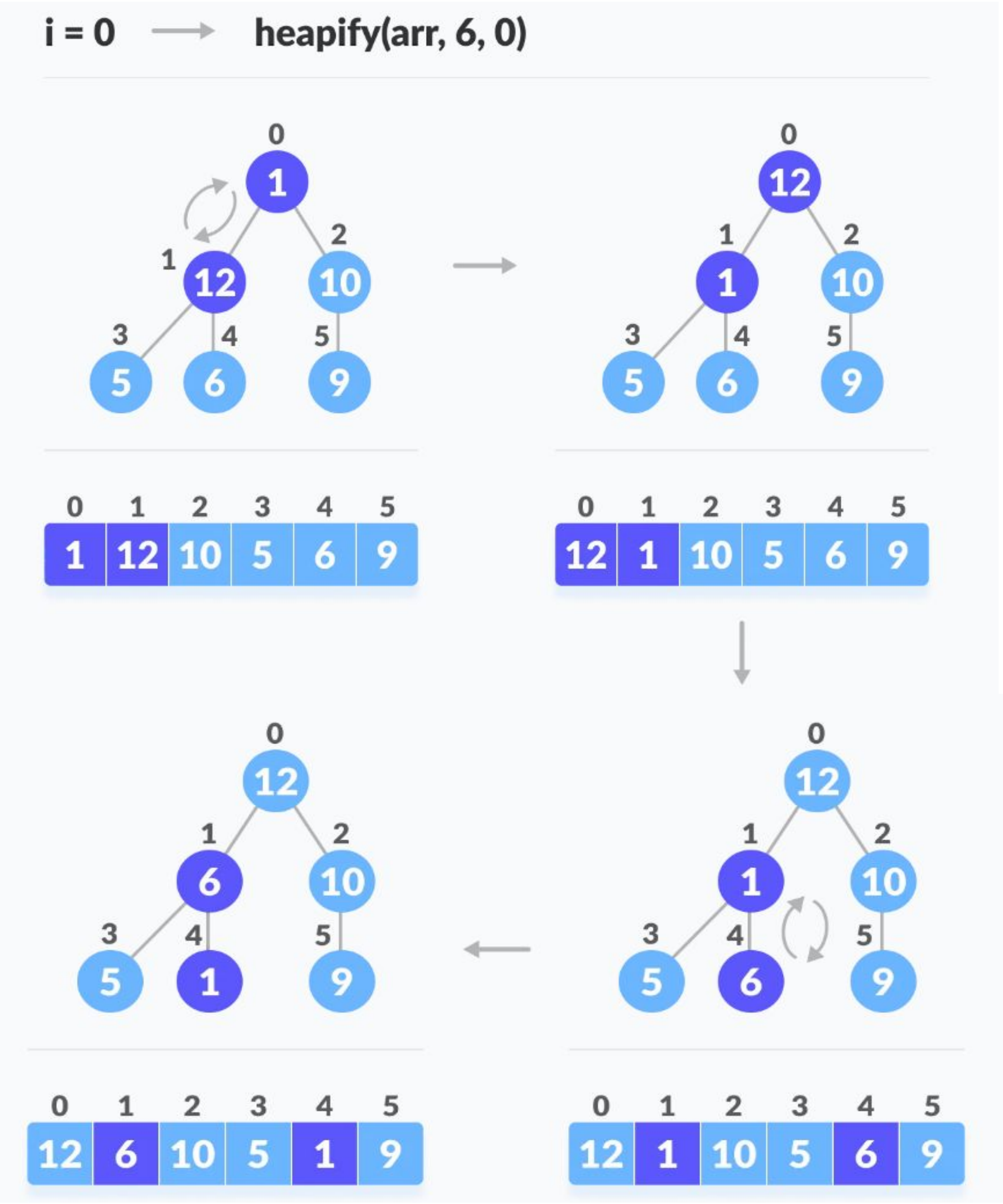
```
// Build heap (rearrange array)
for (int i = n / 2 - 1; i >= 0; i--)
  heapify(arr, n, i);
```

Example


0 1 2 3 4 5
arr 1 12 9 5 6 10
n = 6
i = 6/2 - 1 = 2 # loop runs from 2 to 0


i = 2 → heapify(arr, 6, 2)
0 1 2 3 4 5
1 12 9 5 6 10
0 1 2 3 4 5
1 12 10 5 6 9


i = 1 → heapify(arr, 6, 1)
already heapified
0 1 2 3 4 5
1 12 10 5 6 9



i = 0
heapify(arr, 6, 0)

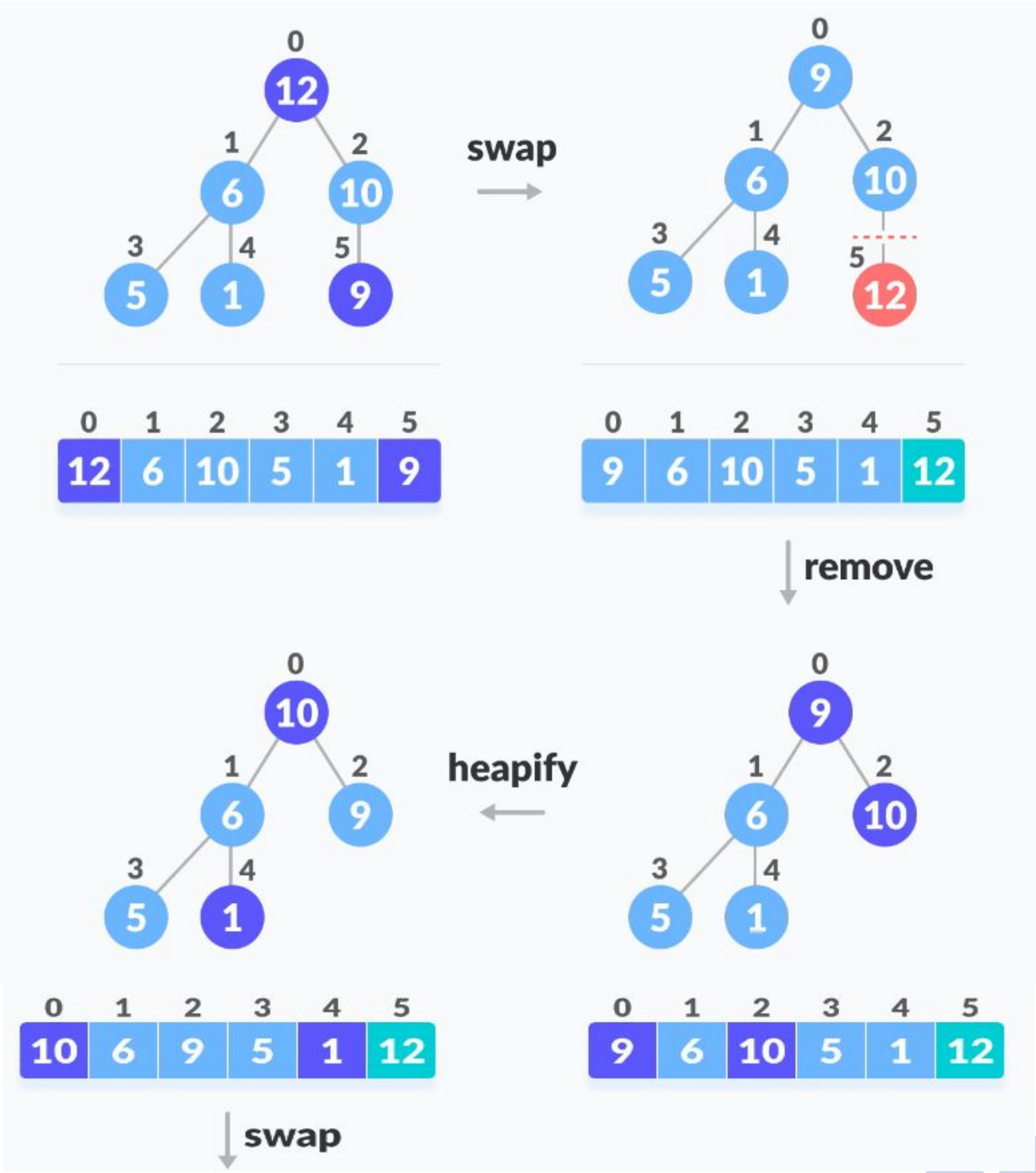
Heap Sort

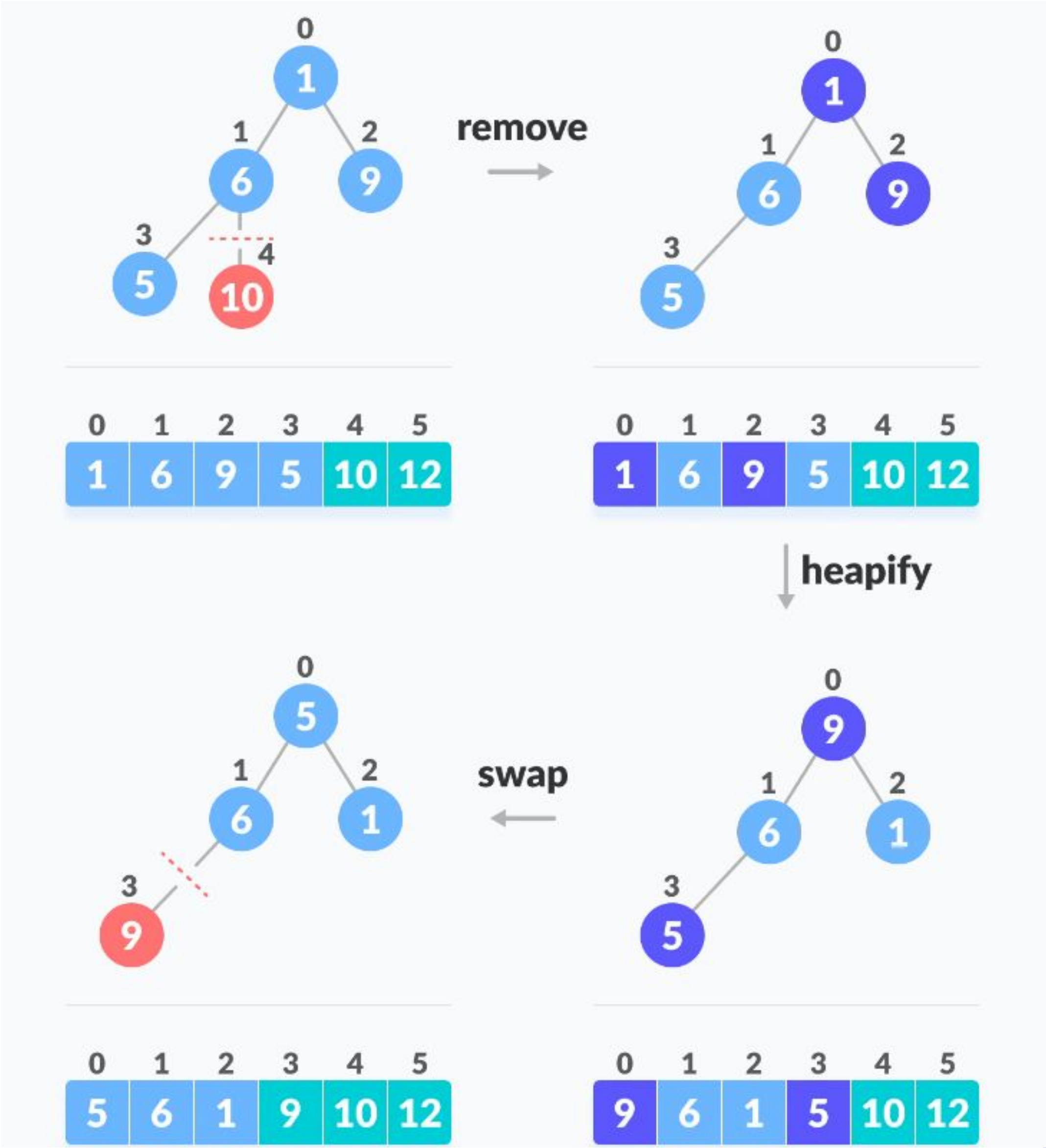
Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.

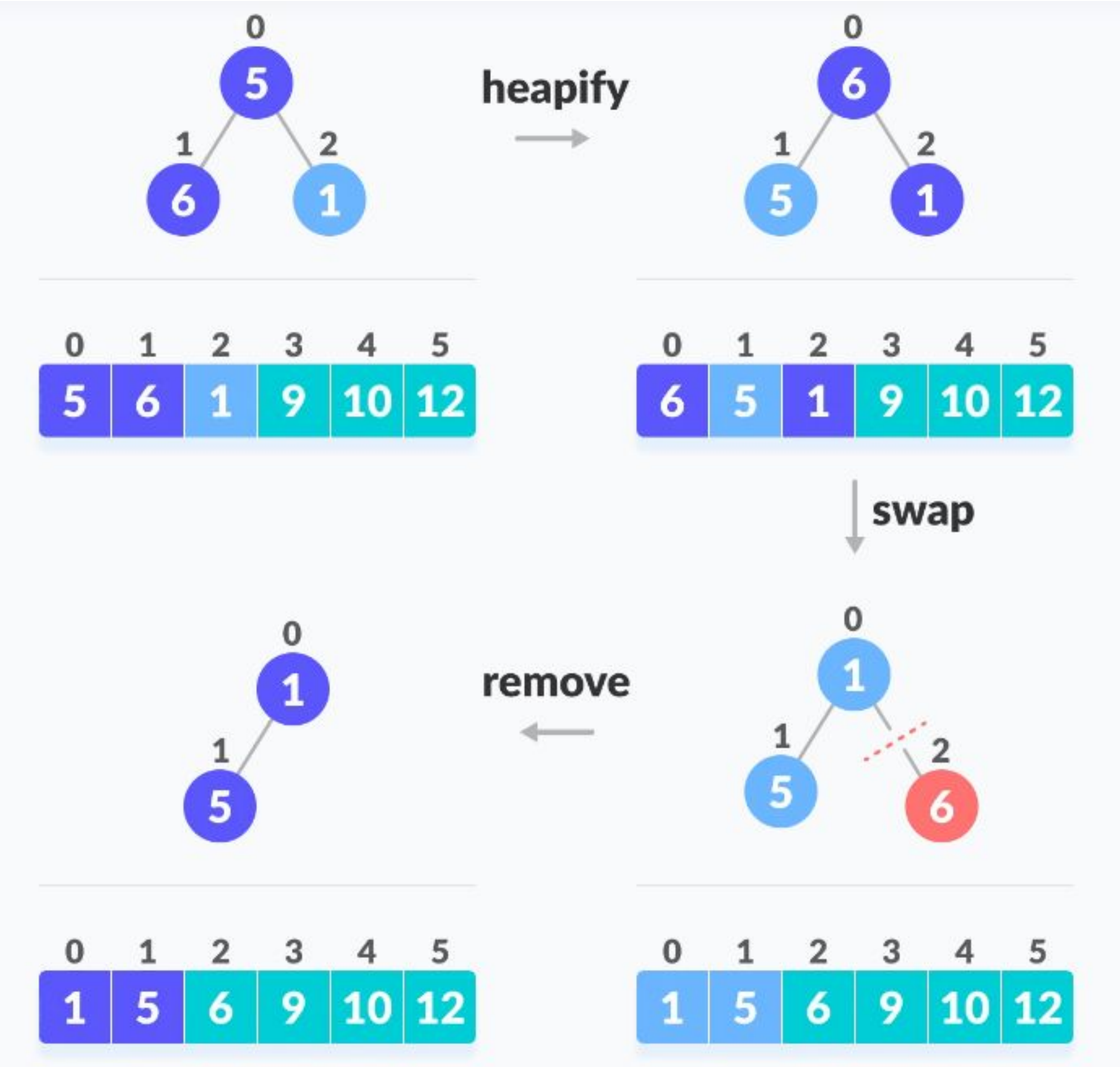
Swap: the root element and the last array element

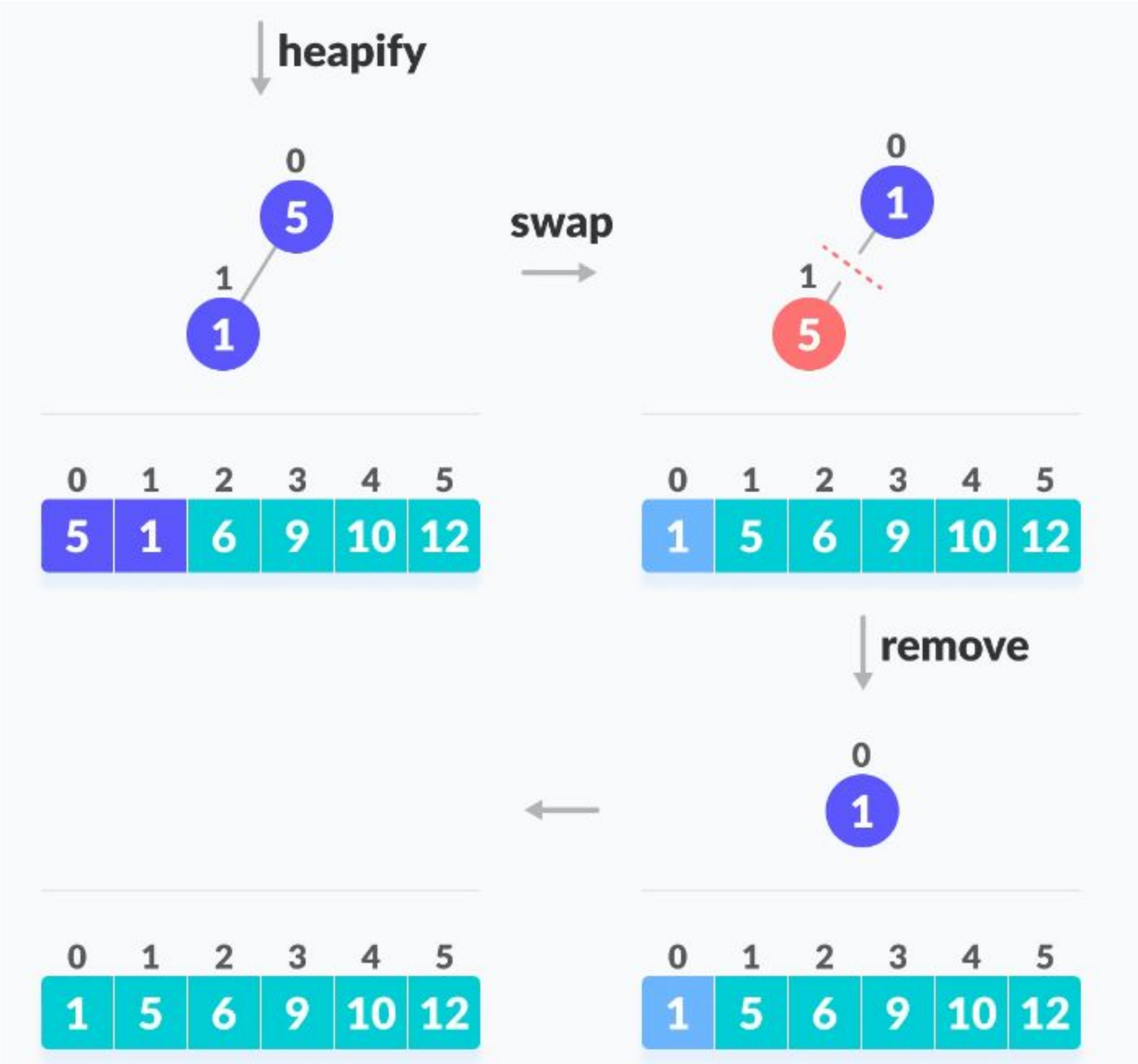
Remove: Reduce the size of the heap by 1.

Heapify: Heapify the root element again so that we have the highest element at root.


swap
remove
heapify
swap


remove
heapify
swap
0 1 2 3 4 5
1 6 9 5 10 12
0 1 2 3 4 5
1 6 9 5 10 12
0 1 2 3 4 5
9 6 1 5 10 12
0 1 2 3 4 5
5 6 1 9 10 12


heapify
swap
remove
0 1 2 3 4 5
5 6 1 9 10 12
6 5 1 9 10 12
1 5 6 9 10 12
1 5 6 9 10 12



heapify
0
5
1
1
swap
0
1
1
5
0 1 2 3 4 5
5 1 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12
remove
0
1
0 1 2 3 4 5
1 5 6 9 10 12
0 1 2 3 4 5
1 5 6 9 10 12

```
// Heap sort
for (int i = n - 1; i >= 0; i--) {
  swap(&arr[0], &arr[i]);

  // Heapify root element to get highest element at root again
  heapify(arr, i, 0);
}
```

Heap Sort Complexity

Time Complexity	
Best	O(nlog n)
Worst	O(nlog n)
Average	O(nlog n)
Space Complexity	O(1)

```
void heapify(int arr[], int n, int i) {
  // Find largest among root, left child and right child
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest])
    largest = left;

  if (right < n && arr[right] > arr[largest])
    largest = right;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
      swap(&arr[i], &arr[largest]);
      heapify(arr, n, largest);
  }
  }
```