DATA STRUCTURES AND ALGORITHMS

Dr Samabia Tehsin

BS (AI)

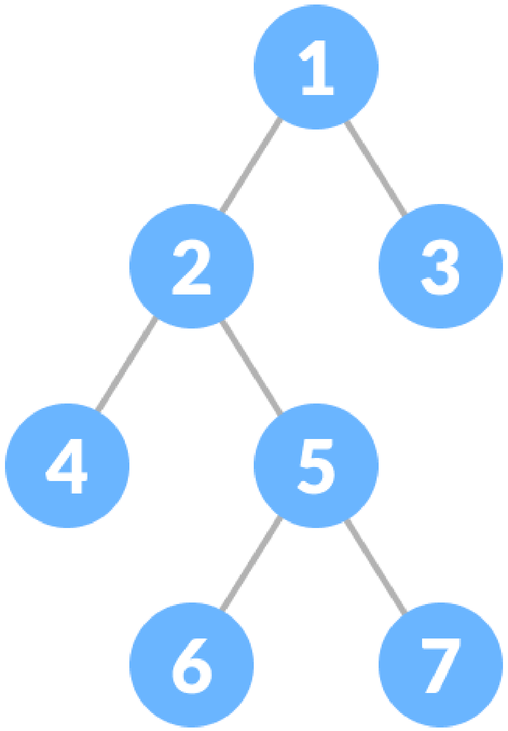
Heap Sort

- Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.
- Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

Types of Binary Tree

Full Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



Types of Binary Tree

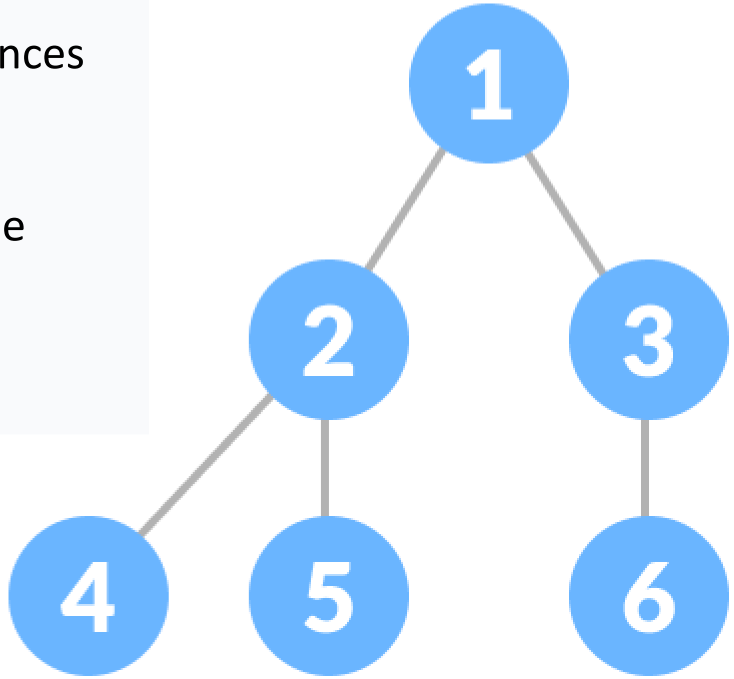
Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two major differences

1.Every level must be completely filled

2.All the leaf elements must lean towards the left.

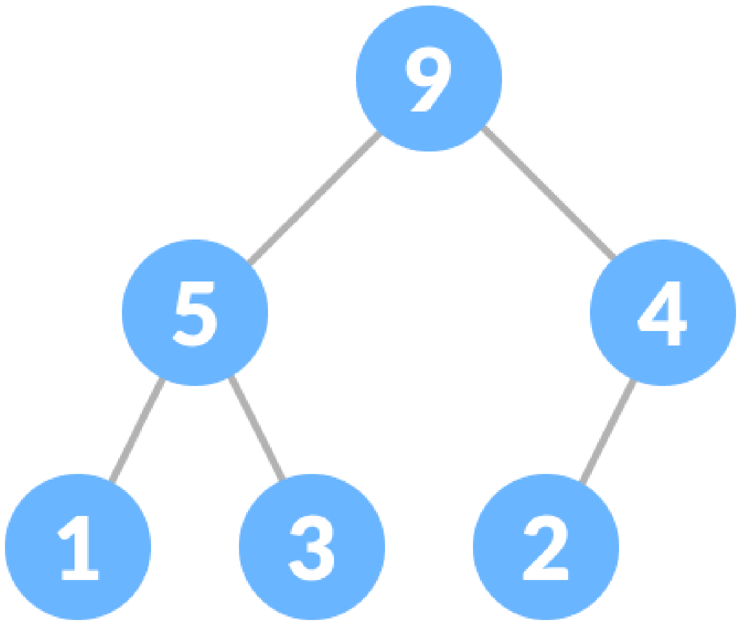
3.The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



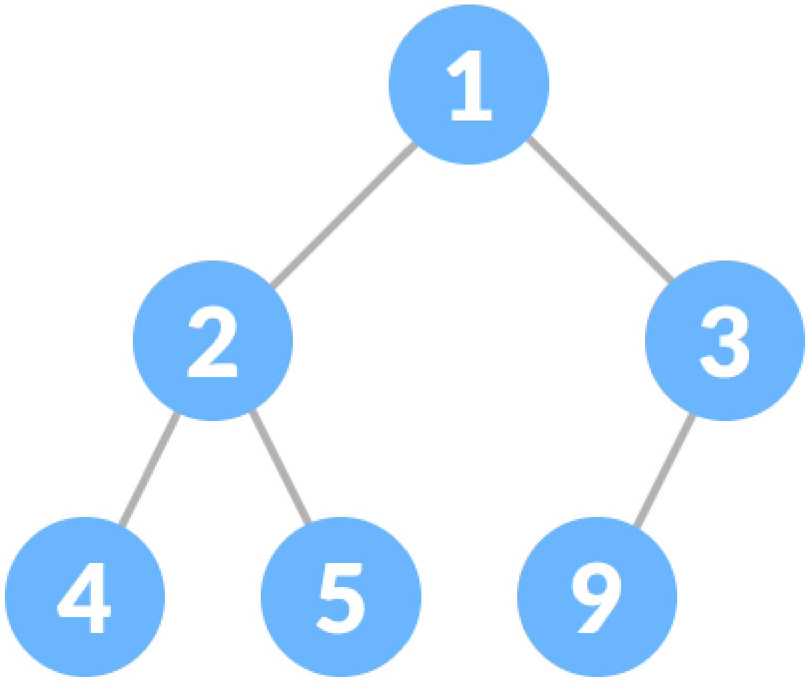
Heap Data Structure

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.



Max-heap

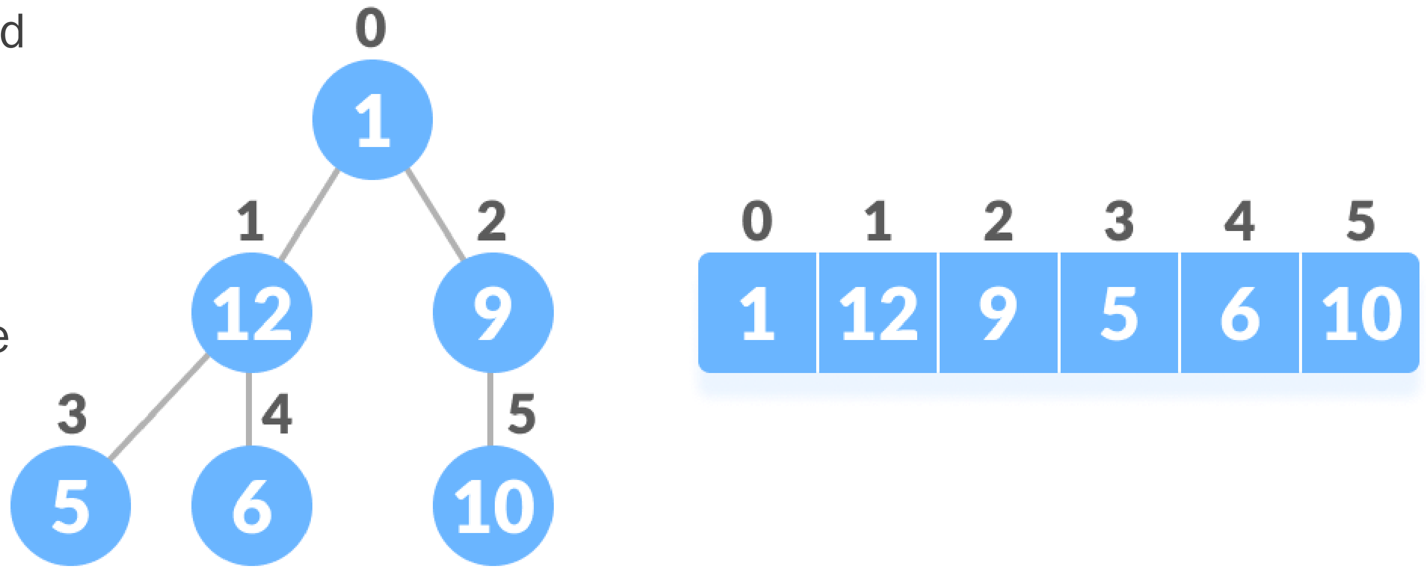


Min-heap

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 and element in 2i+2 index will become the right child. Also, the parent of any element at index i is given by the lower bound of (i-1)/2.



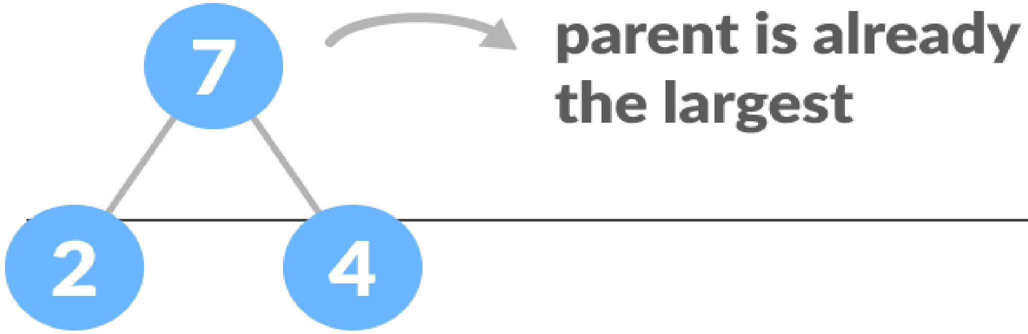
How to "heapify" a tree

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap.

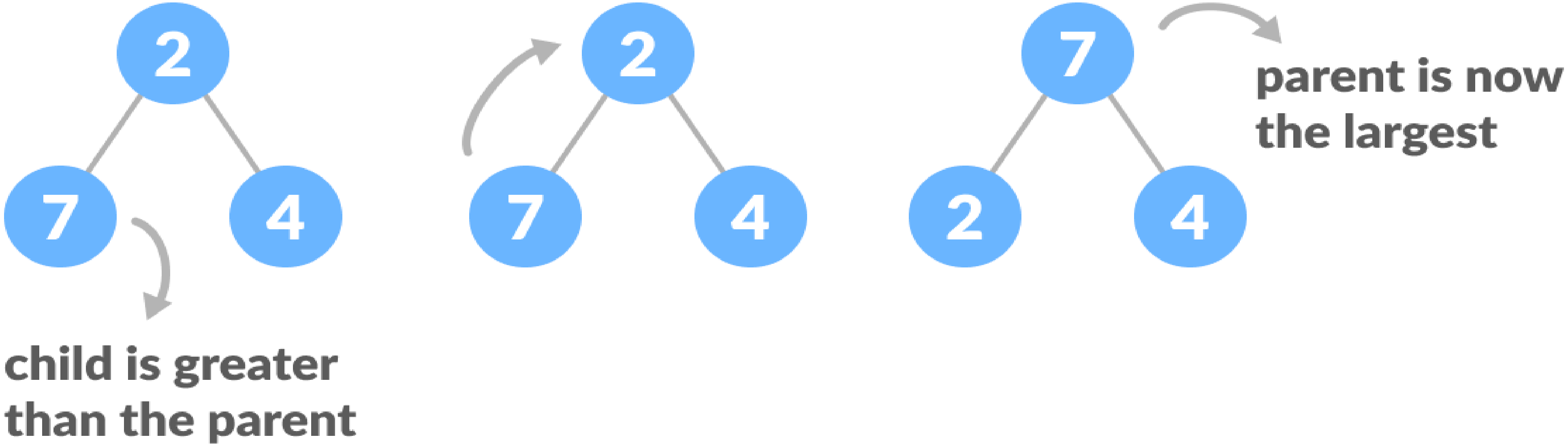
```
heapify(array)
    Root = array[0]
    Largest = largest( array[0] , array [2*0 + 1], array[2*0+2])
    if(Root != Largest)
        Swap(Root, Largest)
```

Heapify: base cases

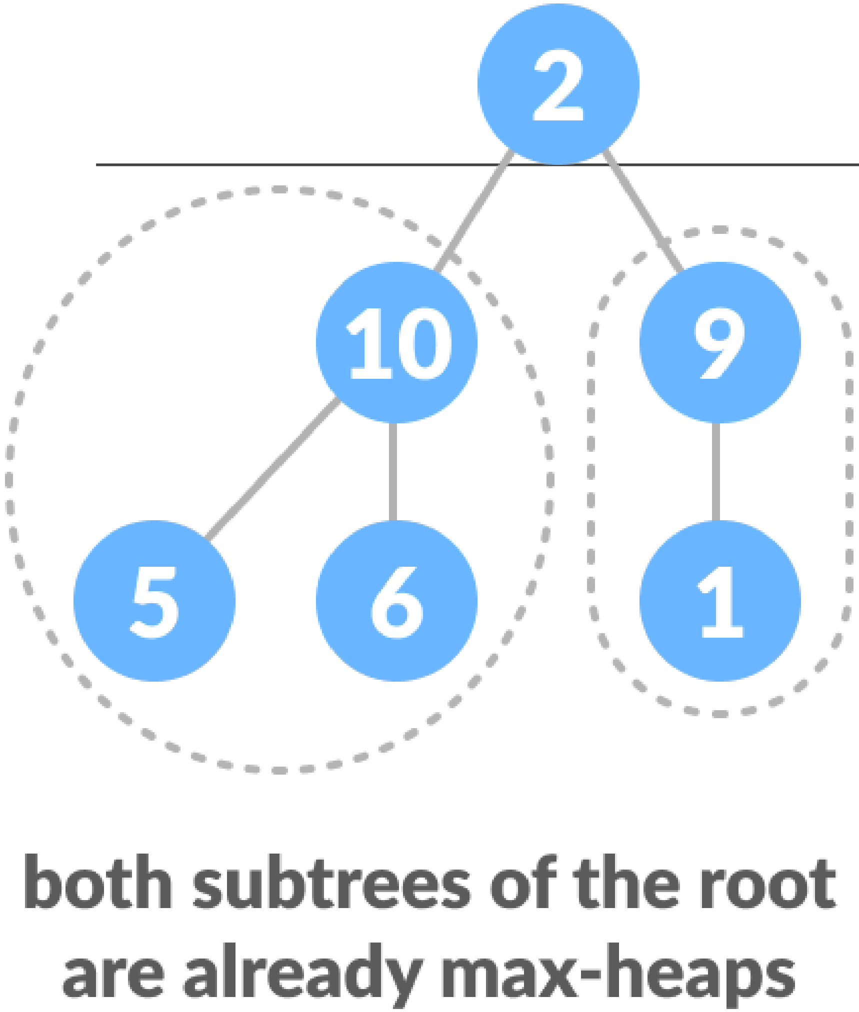
Scenario-1



Scenario-2



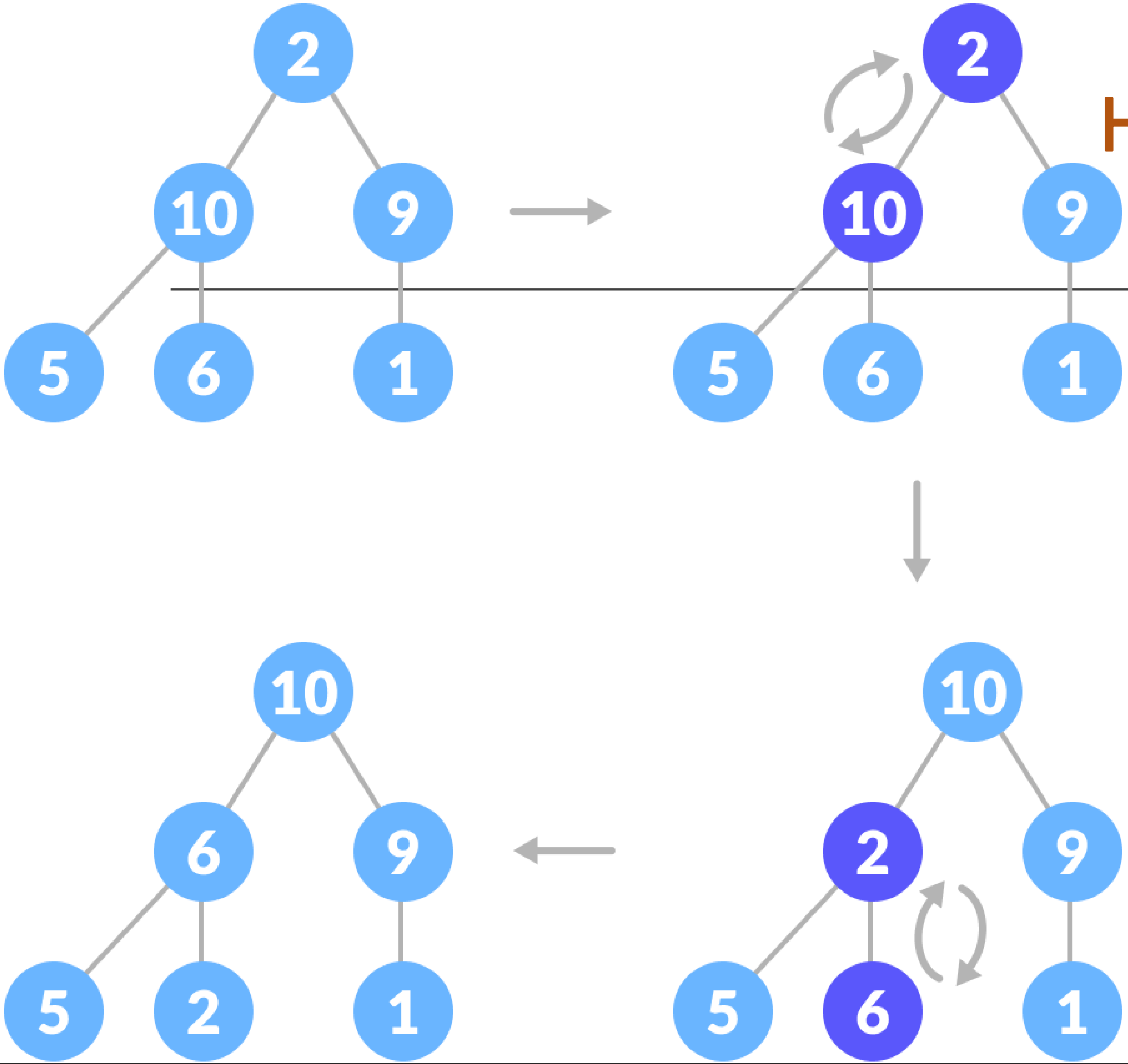
Heapify: Recursive cases



The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.

Heapify: Recursive cases



The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.

```c
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);
   }
  }
```

Build max-heap

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

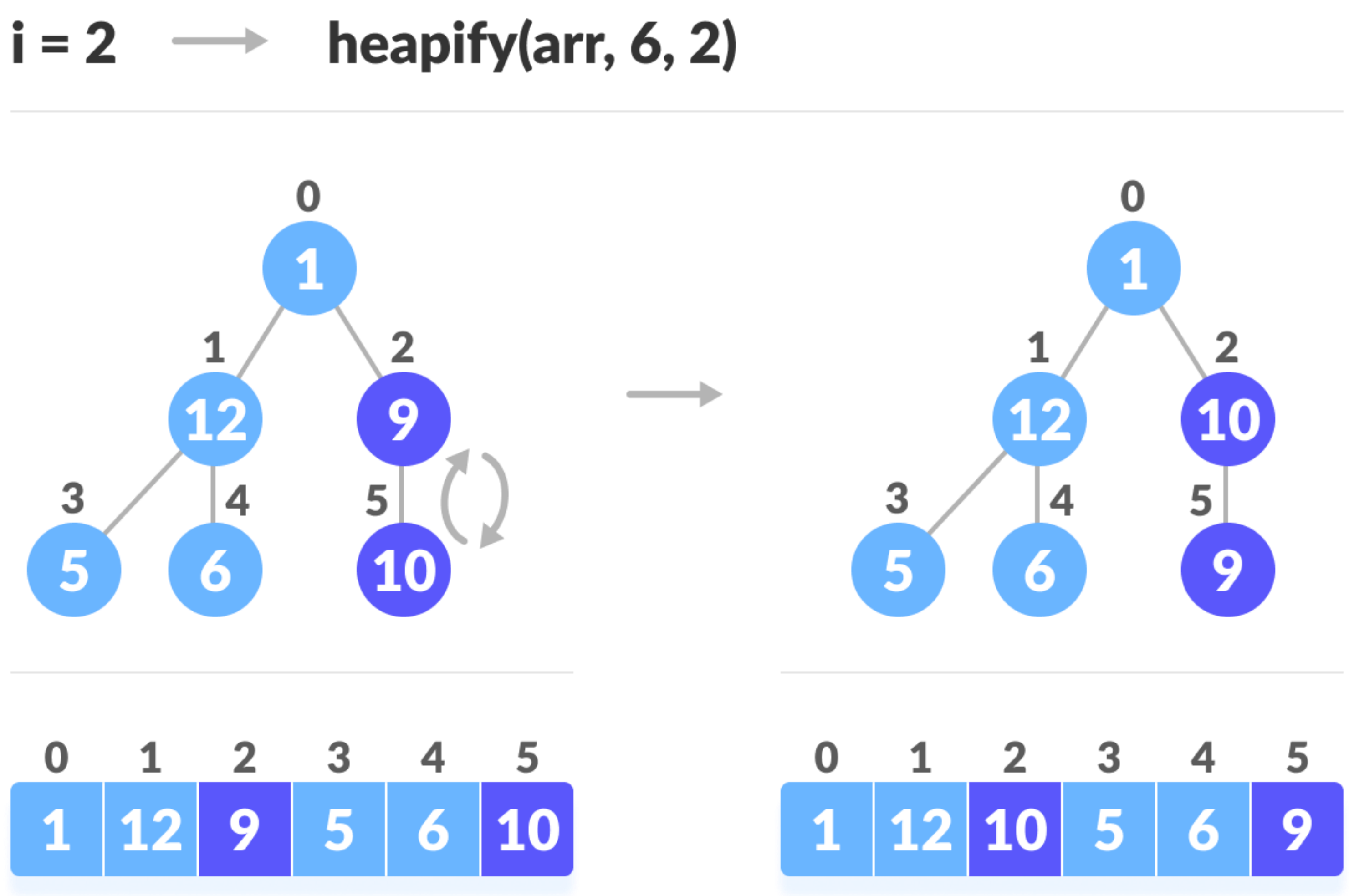
Build max-heap

	0	1	2	3	4	5
arr	1	12	9	5	6	10

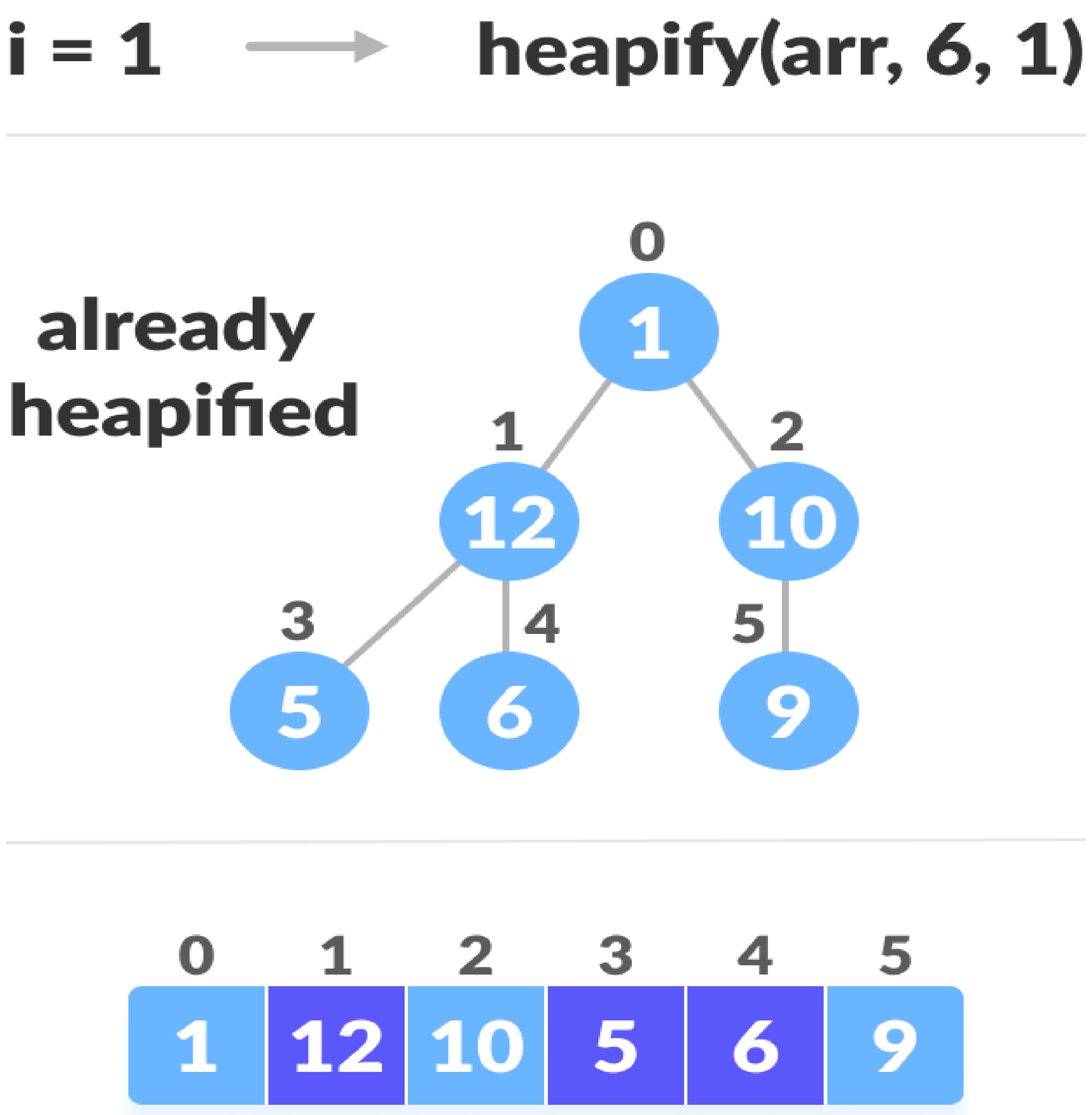
n = 6

i = 6/2 - 1 = 2 # loop runs from 2 to 0

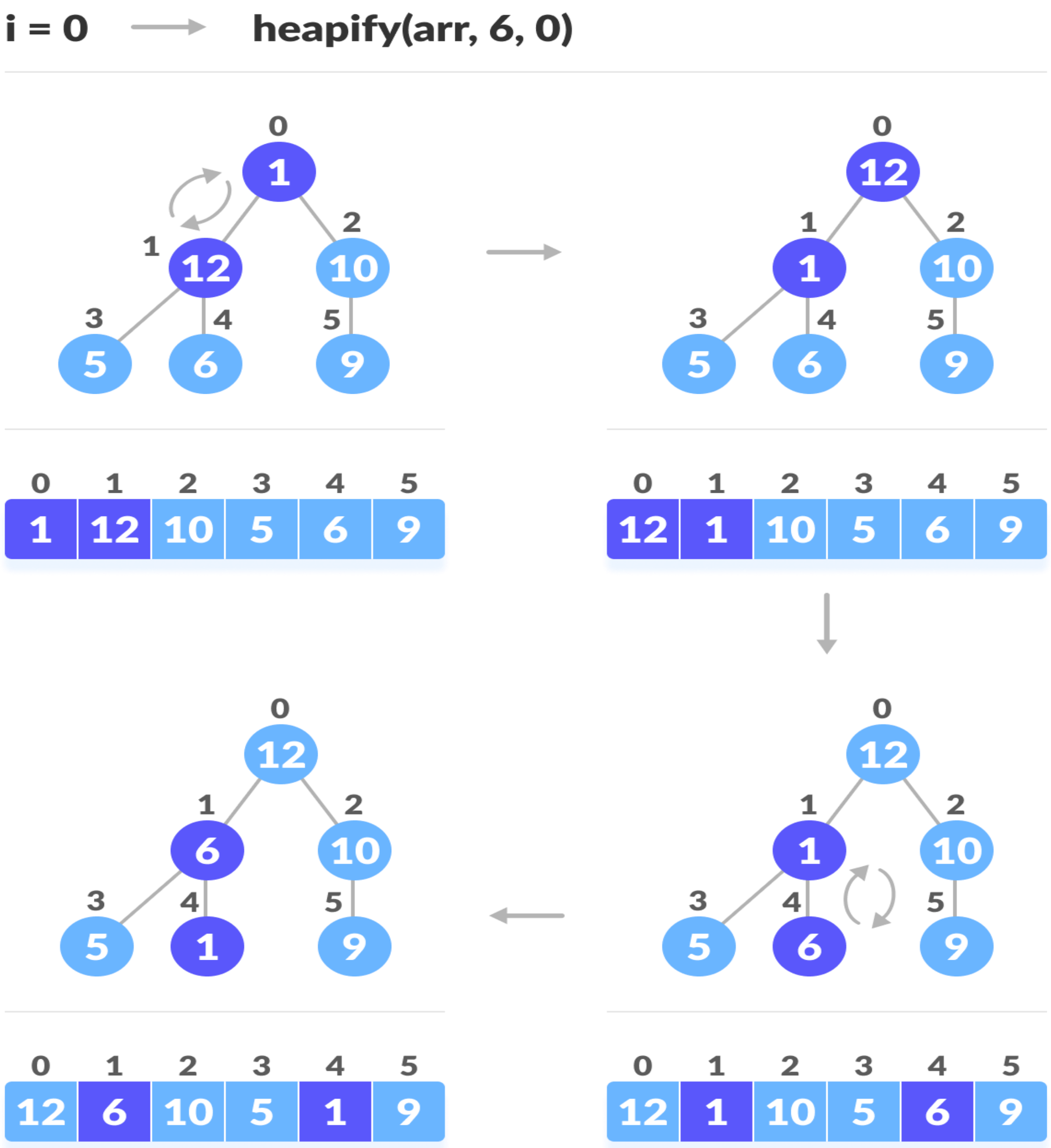
Build max-heap



Build max-heap



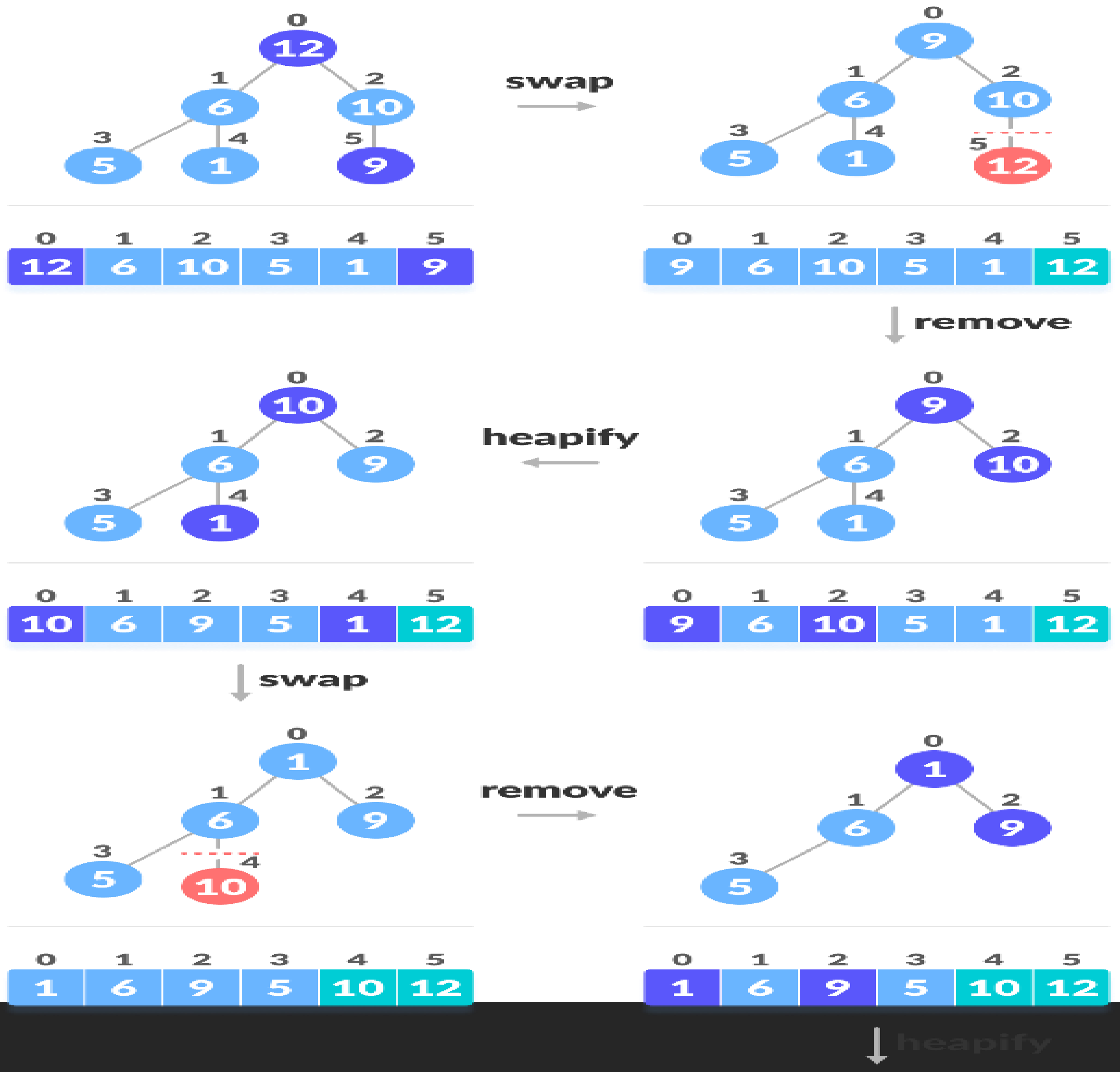
Build max-heap



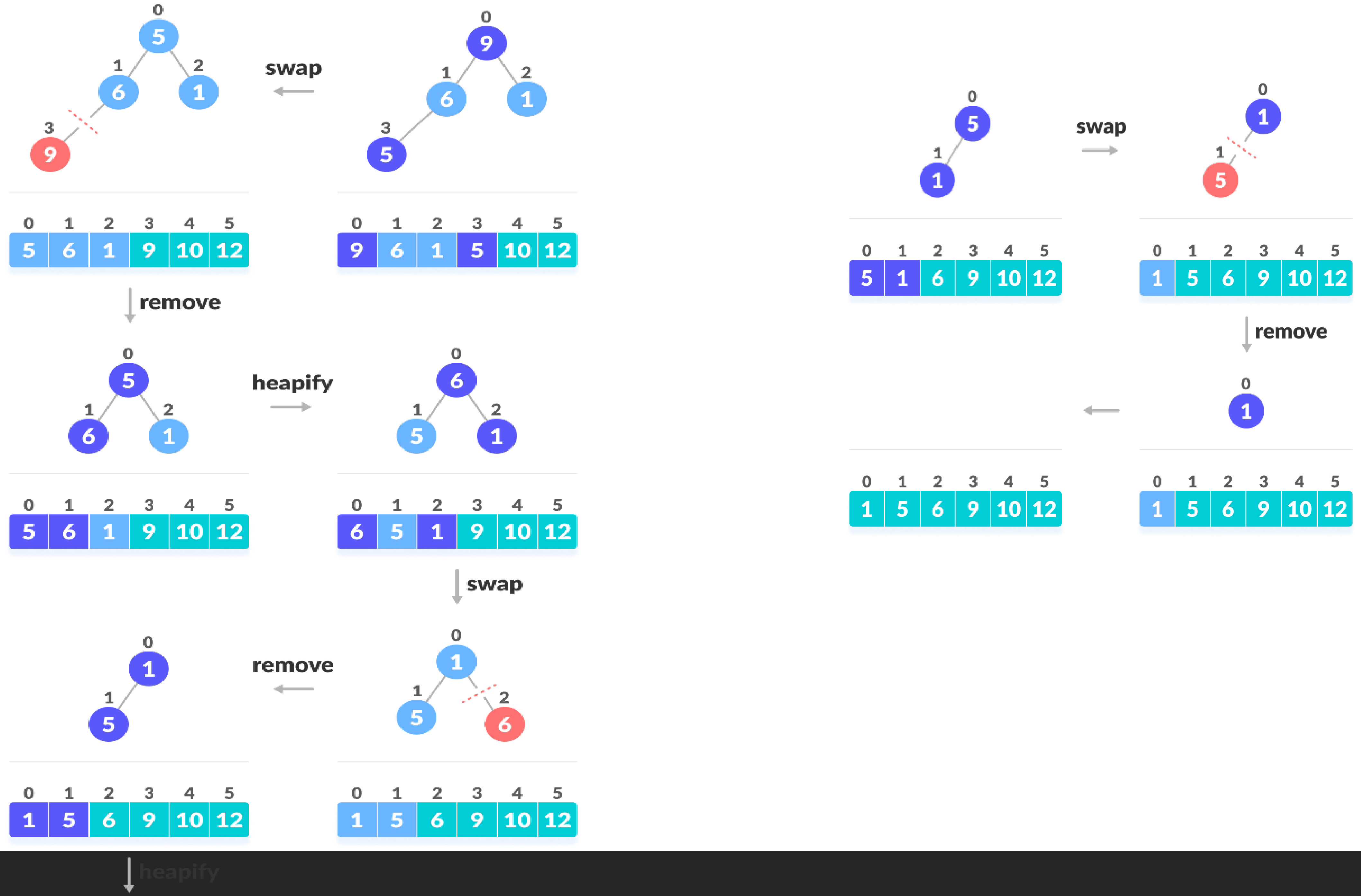
Working of Heap Sort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.

Working of Heap Sort



Working of Heap Sort



Code for Heap Sort

```
// 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); }
```

Code for Heap Sort

```
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);
  }
}
```

```
// main function to do heap sort
  void heapSort(int arr[], int n) {
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);

    // 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)

	Heap	Quick	Merge
Best	O(nlog n)	O(n*log n)	O(nlog n)
Worst	O(nlog n)	$O(n^2)$	O(nlog n)
Average	O(nlog n)	O(n*log n)	O(nlog n)