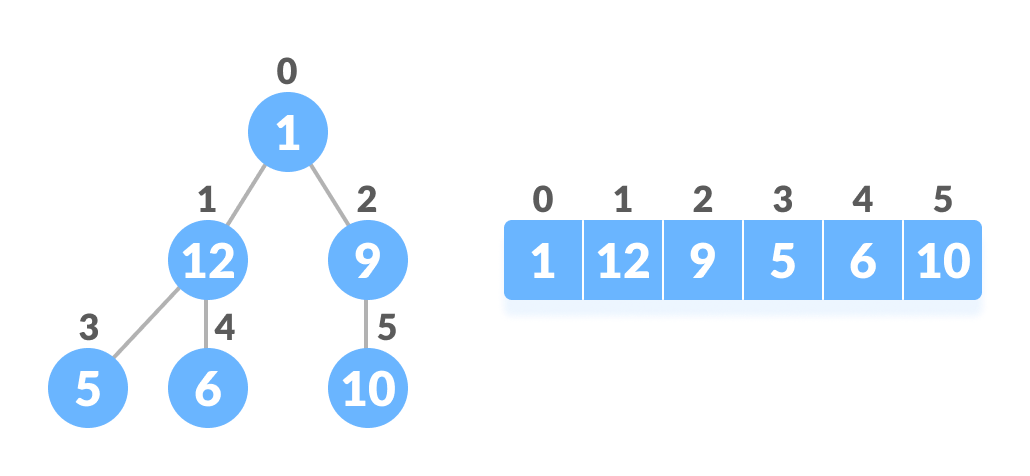
# HeapSort

Heap sort involves building a Heap data structure from the given array and then utilizing the Heap to sort the array.

**Relationship between Array Indexes and Tree Elements**

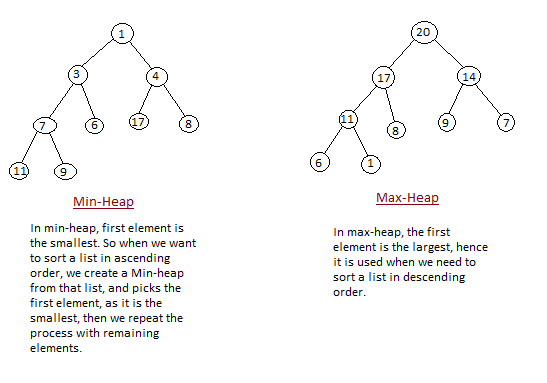
if the index of any element in the array is i, the element in the index 2i+1 will become the left child and element in 2i+2 index will become the right child. Also, the parent of any element at index i is given by the lower bound of (i-1)/2.



**What is a Heap ?**

Heap is a special tree-based data structure, that satisfies the following special heap properties:

1. Heap data structure is always a almost Complete Binary Tree, which means all levels of the tree are filled from left to right.
2. Heap Property: All nodes are either greater than or equal to or less than or equal to each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap(root node contains maximum element)**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap(root node contains the minimum element)**.



**Heap Sort Algorithm :**

Sorting can be in ascending or descending order. Either Max heap or min heap logic can be taken depending on the need.

1. Build a max/min heap using Heapify() from the input data.
2. At this point, the largest/smallest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.

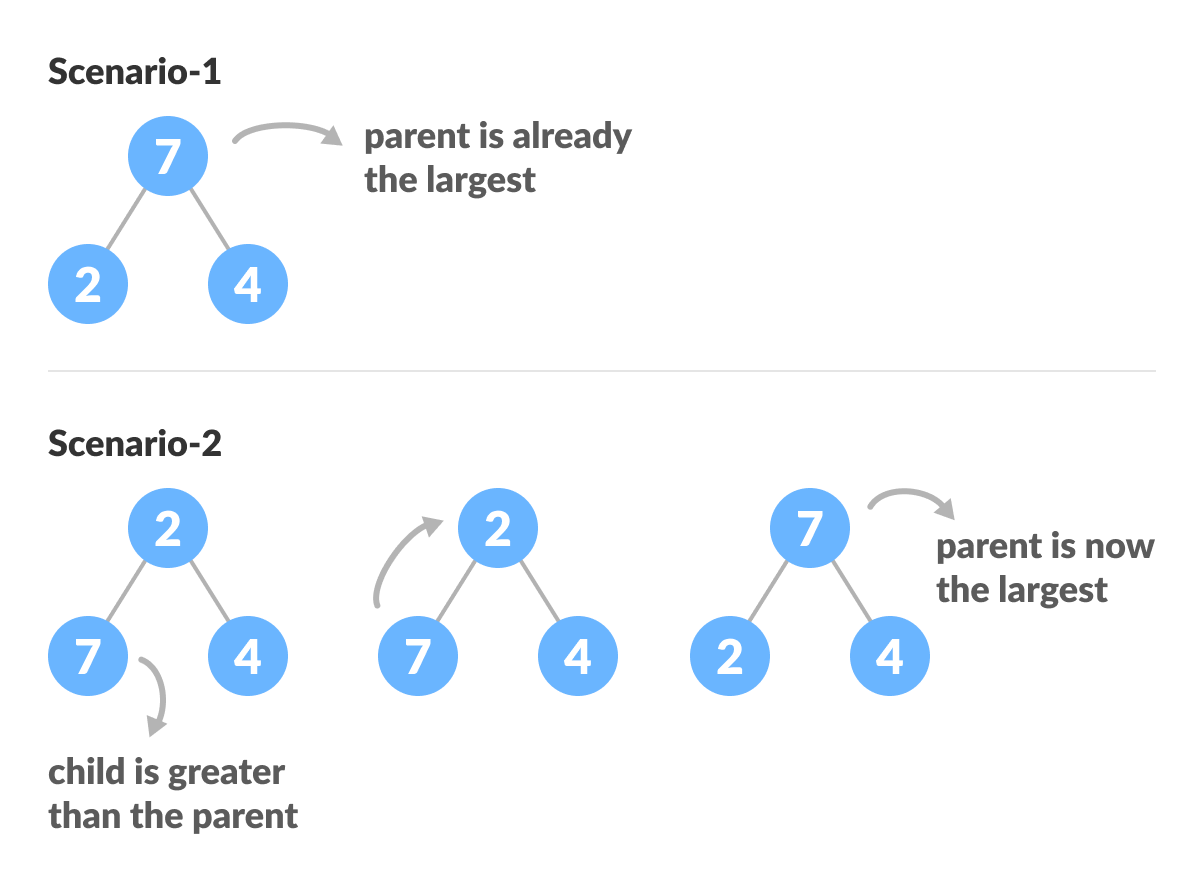
Heap sort uses two core procedure

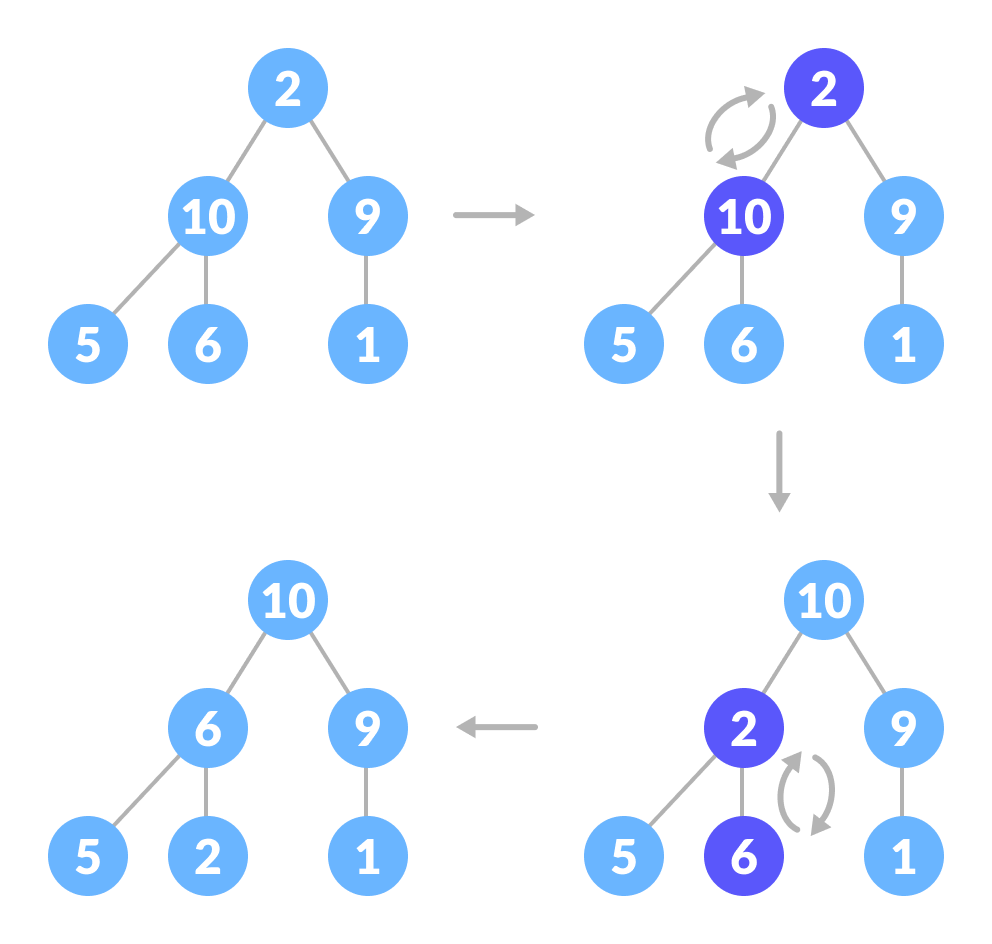
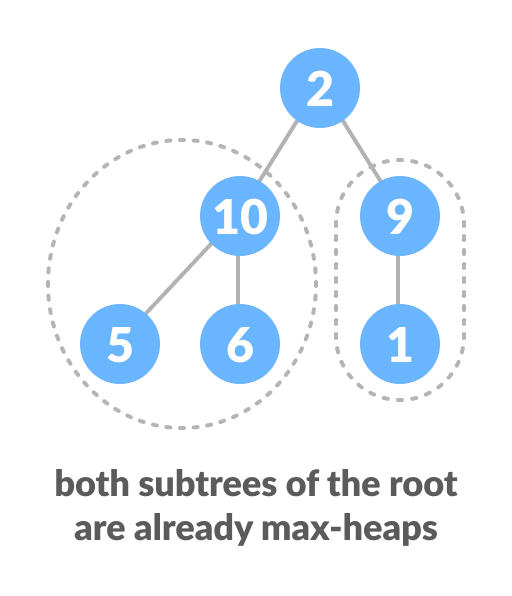
1. Heapify: To restore Heap propoerty
2. BuildHeap: To construct max or min heap from the data

Max heap is used in ascending order sorting and Mean heap is used in descending order sorting

**Heapify**

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapify must be performed in the bottom up order. Leaf nodes are 1 element heap as no it has no child node.





void max\_heapify(int arr[], int n, int i)

{

int largest = i ,t;

int l = (2 \* i) + 1; /\* left \*/

int r = (2 \* i) + 2; /\* right \*/

/\* If left child is larger than root \*/

if (l < n && arr[l] > arr[largest])

largest = l;

/\* If right child is larger than largest so far\*/

else if (r < n && arr[r] > arr[largest])

largest = r;

/\* If largest is not root \*/

if (largest != i)

{

t = arr[i];

arr[i] = arr[largest];

arr[largest] = t

/\* Recursively heapify the sub-tree \*/

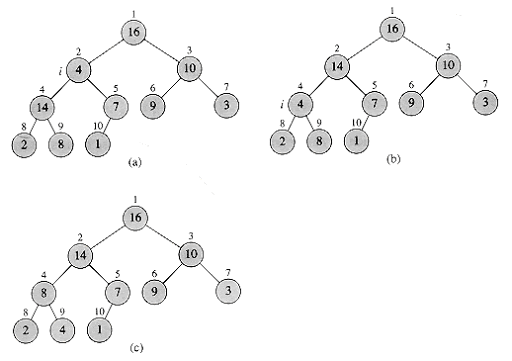
max\_heapify(arr, n, largest);

}

}

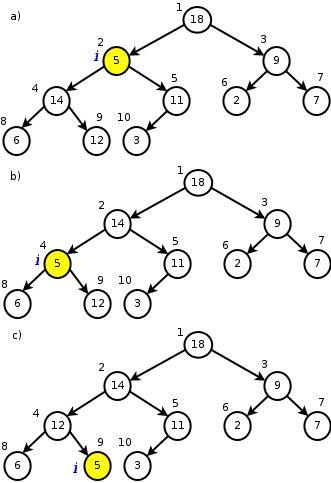
Ex:1 input[16,4,10,14,7,9,3,2,8,1]

output[16,14,10,8,7,9,3,2,4,1]



1. Main ->Heapify(arr,1, 10)
2. Heapify(arr,1, 10) -> Heapify(arr,4, 10)
3. Heapify(arr,1, 10) -> Heapify(arr,4, 10) -> Heapify(arr,9, 10)

Ex-2



**Build Heap**

We can use the procedure HEAPIFY in a bottom-up manner to convert an array A[1 . . n], where n = length[A], into a heap. Since the elements in the subarray A[(n/2 + l) . . n] are all leaves of the tree, each is a 1-element heap to begin with. The procedure BUILD-HEAP goes through the remaining nodes of the tree and runs HEAPIFY on each one. The order in which the nodes are processed guarantees that the subtrees rooted at children of a node i are heaps before HEAPIFY is run at that node. Heapify start from the last non-leaf node and proceed toward root. The index of the last non-leaf node is **n/2-1**.

BUILD-HEAP(A)

1 heap-size[A] length[A]

2 for i length[A]/2-1 downto 0

3 do HEAPIFY(A, i)

BUILD-HEAP(arr,n)

{

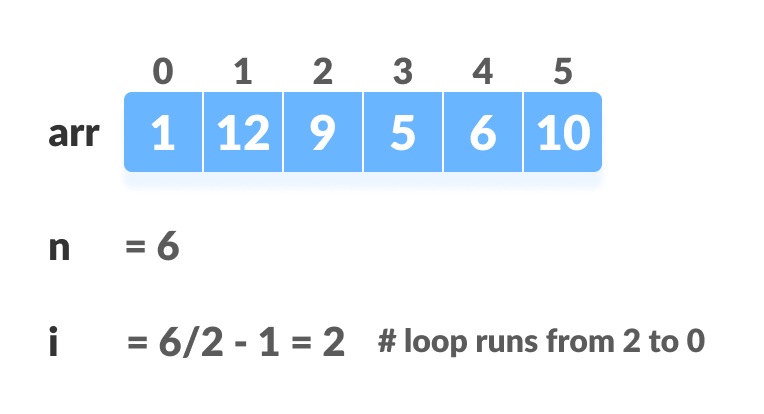
for (int i = n / 2 - 1; i >= 0; i--)

{

heapify(arr, n, i);

}

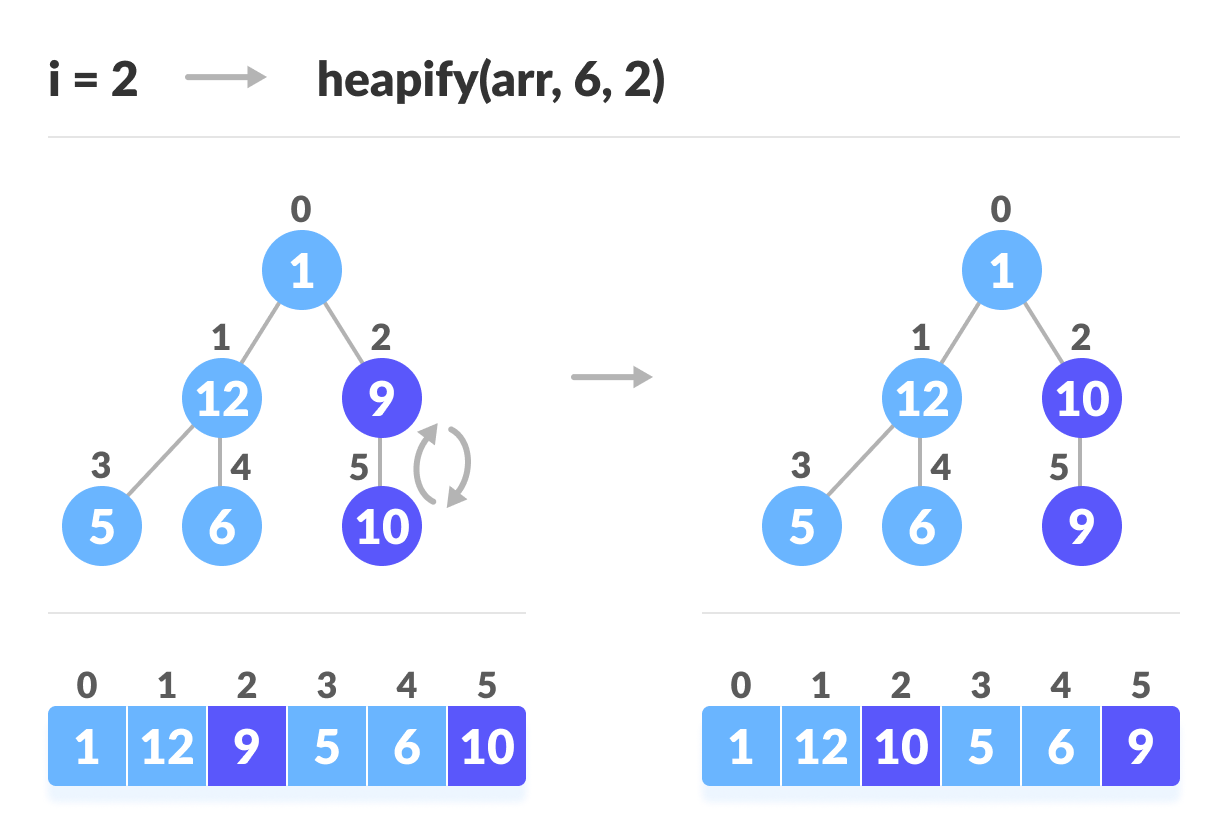
}

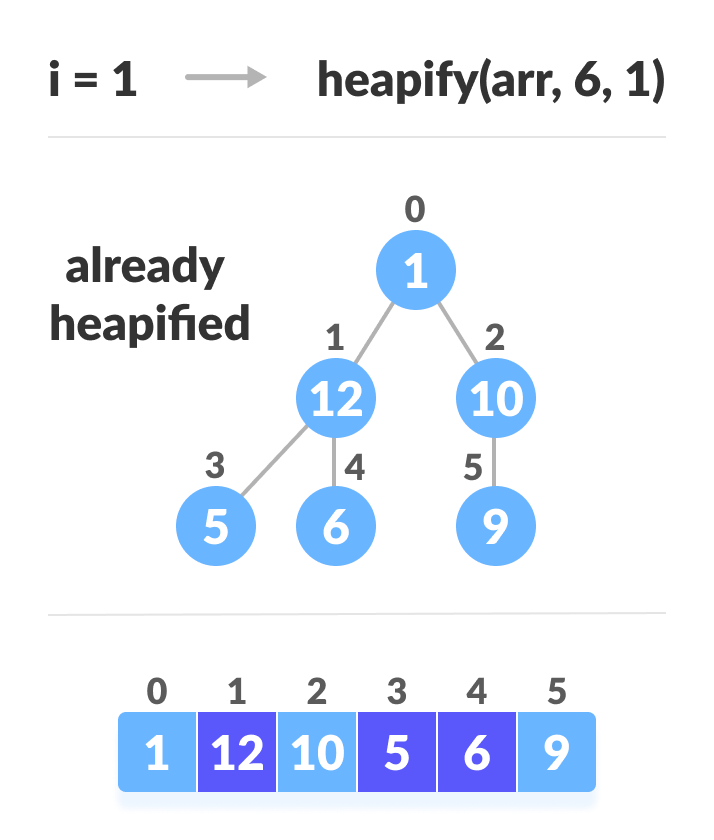


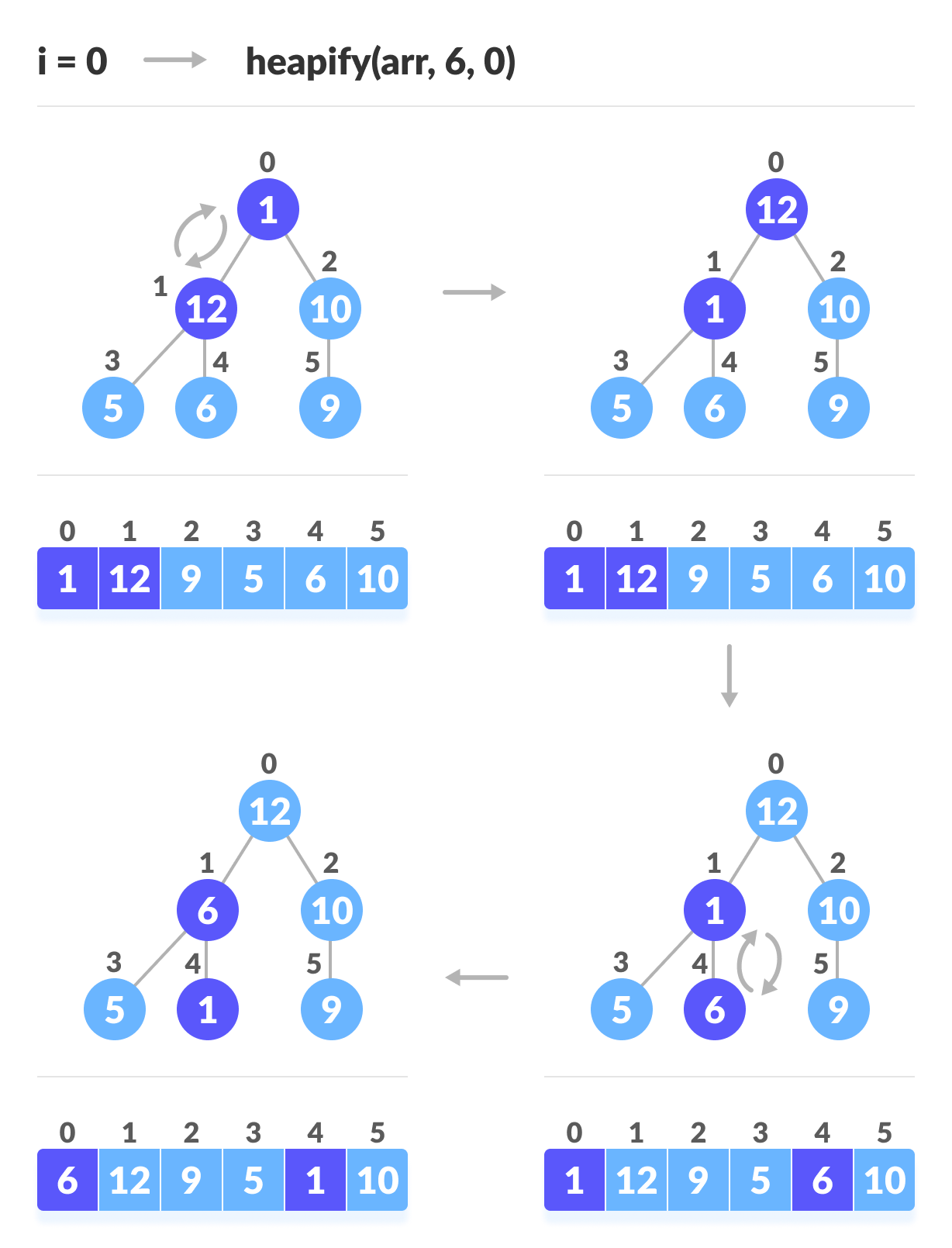
// Build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

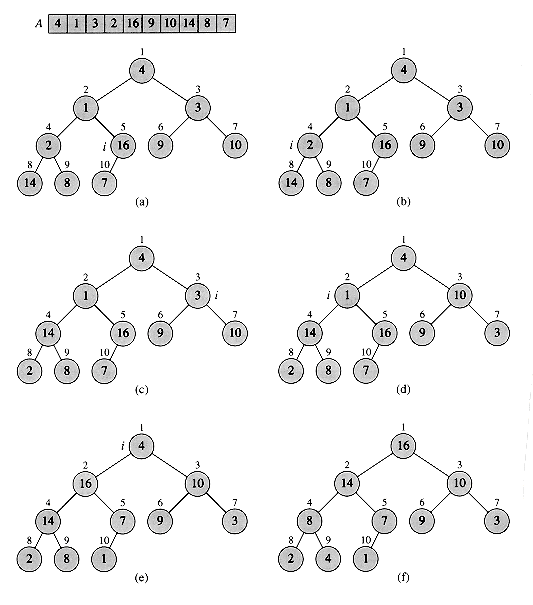
heapify(arr, n, i);







Example



Array represtion of f[ 16, 14,10, 8, 7, 9 ,3,2,4,1]

**Heap Sort**

The heapsort algorithm starts by using BUILD-HEAP to build a heap on the input array A[1 . . n], where n = length[A]. Since the maximum element of the array is stored at the root A[1], it can be put into its correct final position by exchanging it with A[n-1]. If we now "discard" node n from the heap (by decrementing heap-size[A]), we observe that A[0 . . (n - 2)] can easily be made into a heap. The children of the root remain heaps, but the new root element may violate the heap property . All that is needed to restore the heap property, however, is one call to HEAPIFY(A, 0), which leaves a heap in A[0 . . (n - 2)]. The heapsort algorithm then repeats this process for the heap of size n - 1 down to a heap of size 2.

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.

HEAPSORT(A)

1 BUILD-HEAP(A)

2 for i length[A]-1 downto 0

3 do exchange A[0] A[i]

4 heap-size[A] heap-size[A] -1

5 HEAPIFY(A, 1)

void heap\_sort(int arr[], int n)

{

int t;

/\* Build heap (rearrange elements) \*/

BUILD-HEAP(arr,n);

/\* One by one extract an element from heap \*/

for (int i=n-1; i>=0; i--)

{

/\* Move current root to end \*/

t = arr[0];

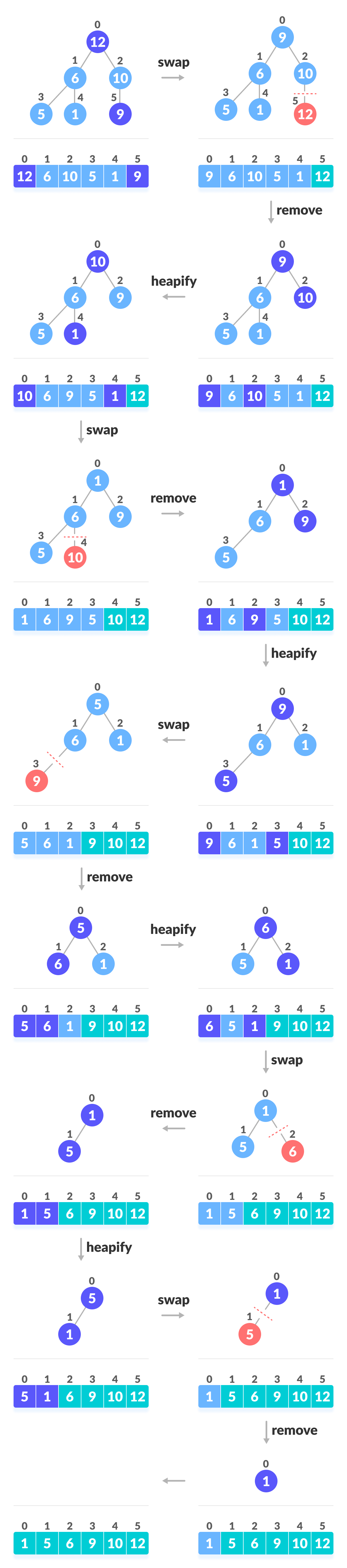
arr[0] = arr[i];

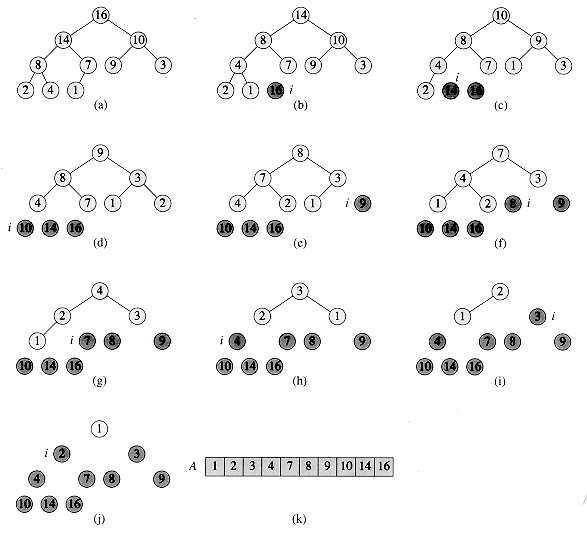
arr[i] = t;

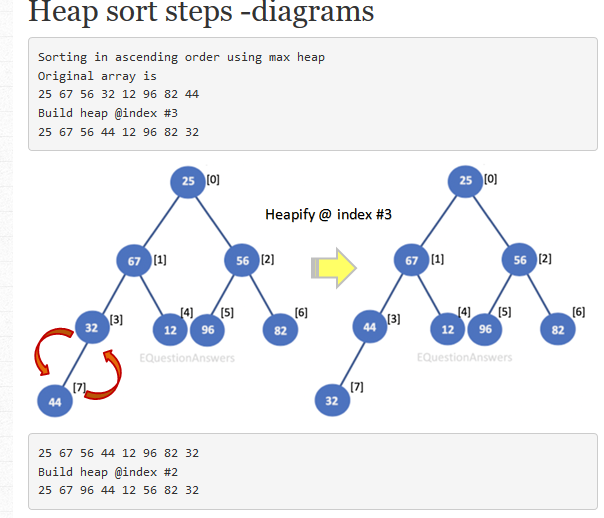
heapify(arr, i, 0);

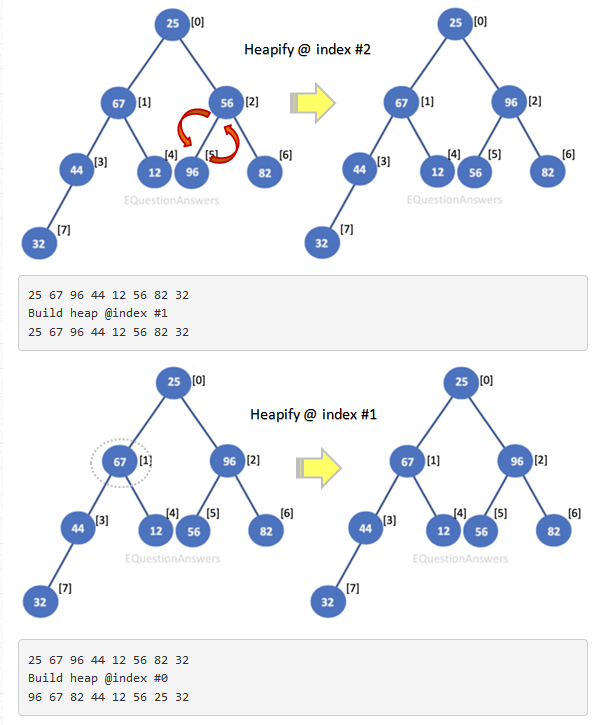
}

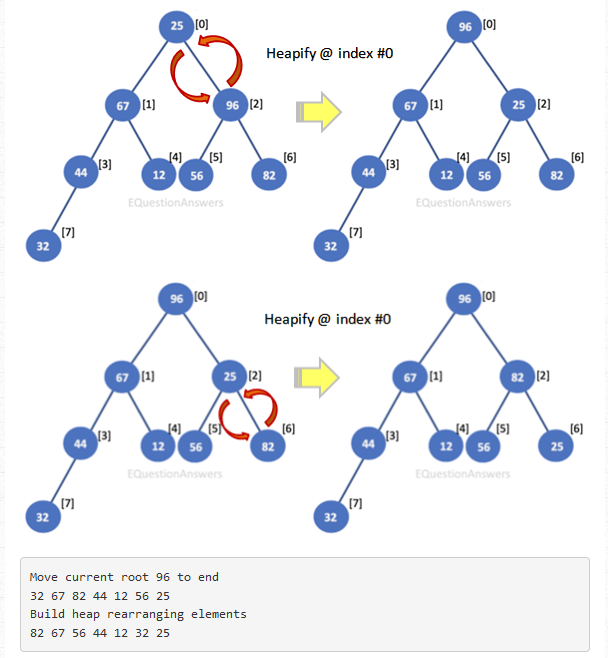
}

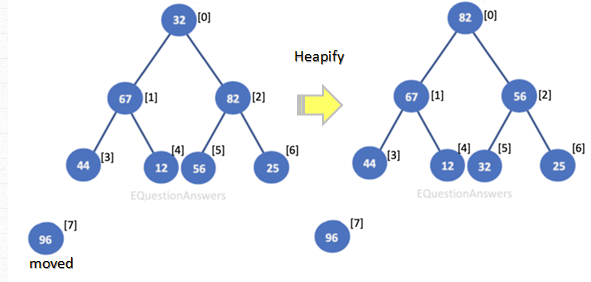


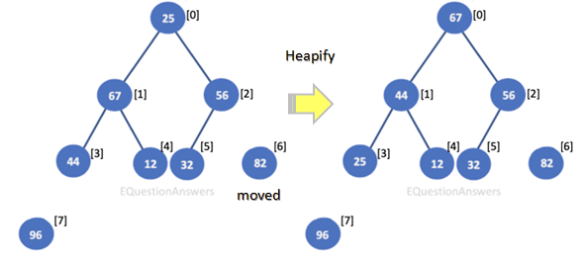


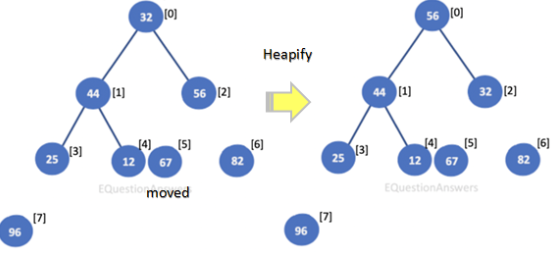


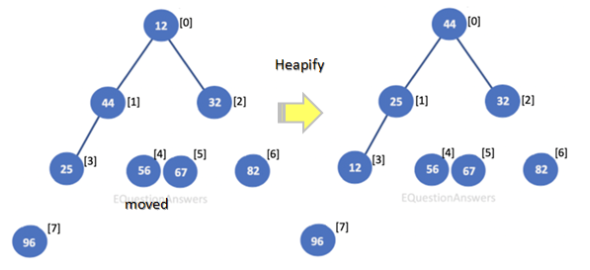


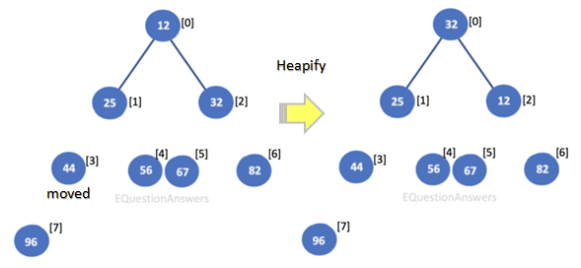


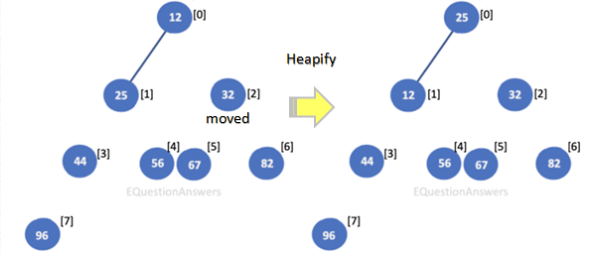


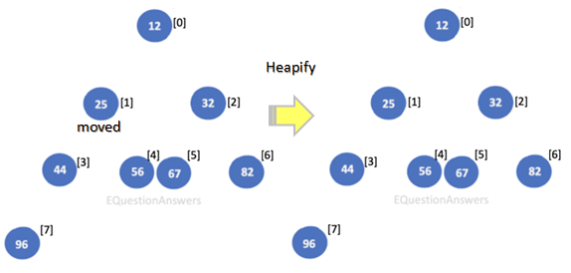


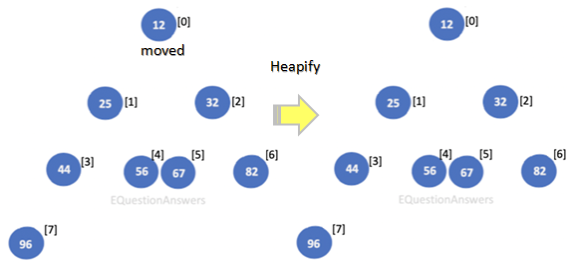












**Worst case time O(nlogn)**

**Best case time O(n)**

**Average case time O(nlogn)**