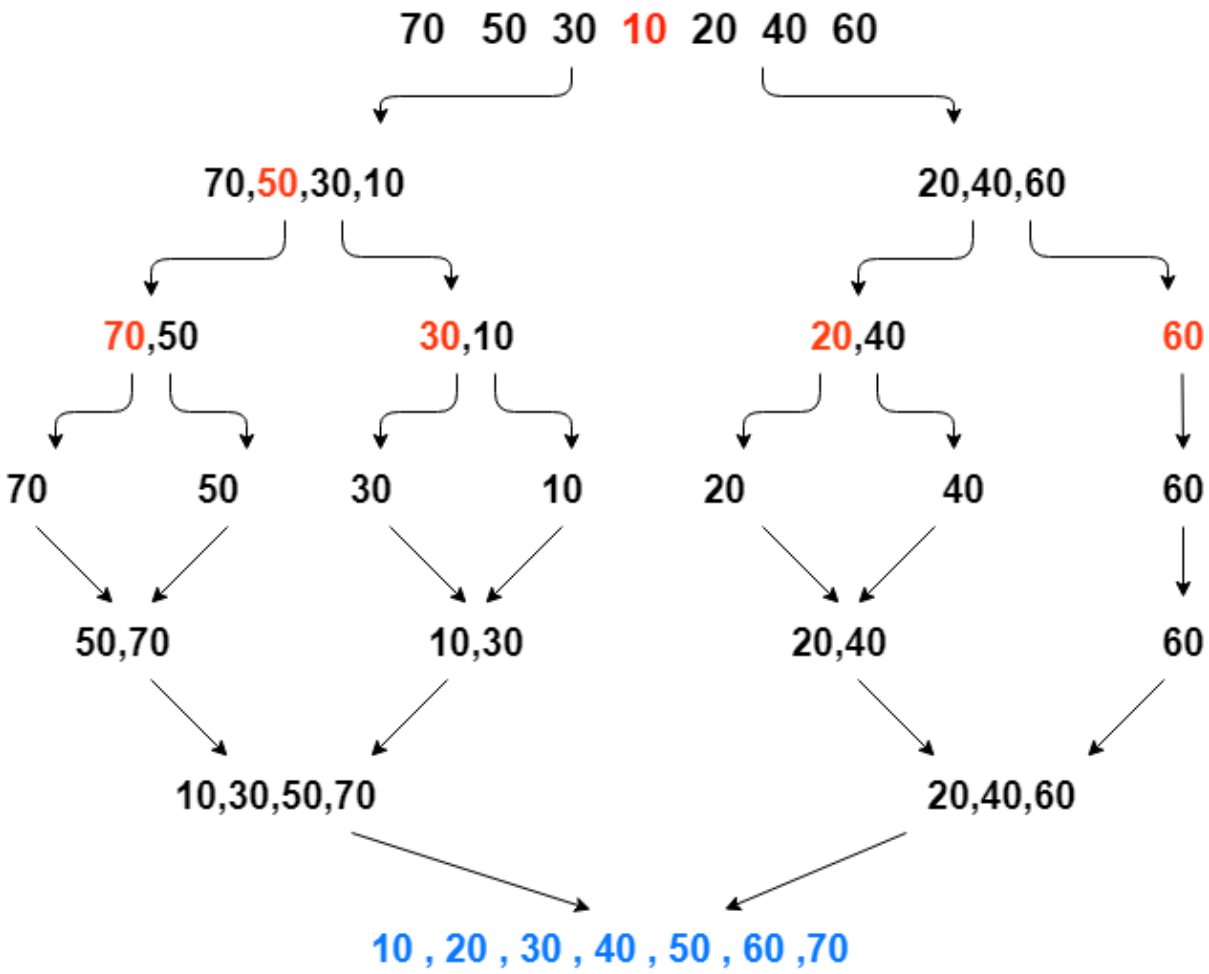
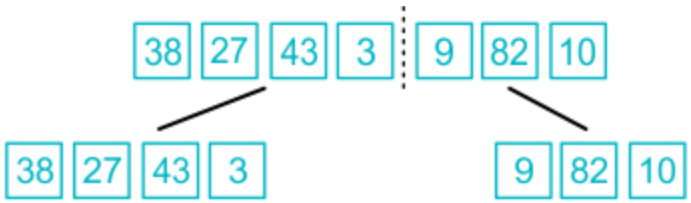
Merge sort:


70 50 30 10 20 40 60
70,50,30,10
20,40,60
70,50
30,10
20,40
60
70
50
30
10
20
40
60
50,70
10,30
20,40
60
10,30,50,70
20,40,60
10 , 20 , 30 , 40 , 50 , 60 ,70

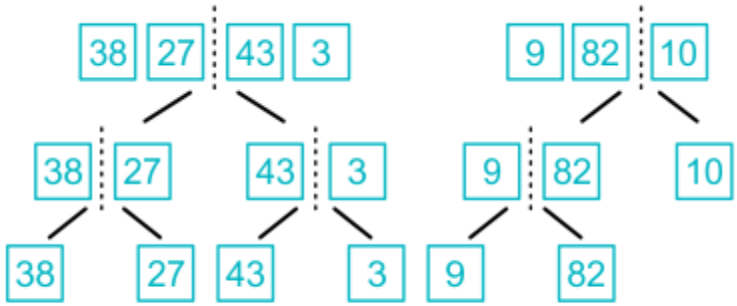
Consider the following array, arr[ ]={38,27,43,3,9,82,10} to understand how merge sort works in the divide and conquer approach.

Step 1: First determine whether the array's left index is lower than its right index; if so, get the array's midpoint.

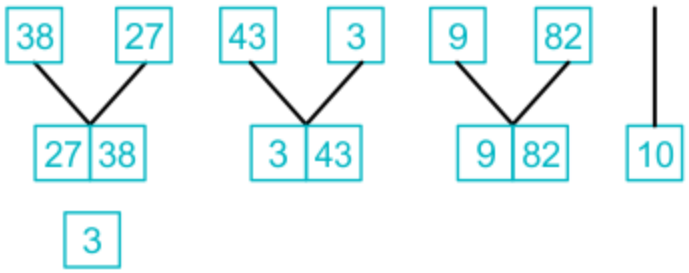
Step 2: Now, as we already know, until the atomic values are attained, merge sort divides the entire array into equal halves iteratively first. Here, we can see that a 7-item array is split into two arrays with sizes of 4 and 3, respectively.



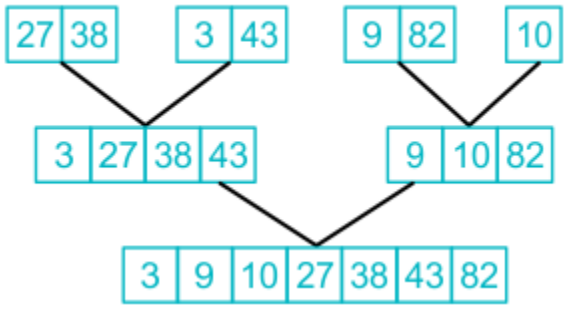
Step 3: Now, keep dividing these two arrays into smaller and smaller parts until the array's atomic units are reached and more division is impossible.



Step 4: Start merging the items once more based on the comparison of element sizes after dividing the array into the smallest units. In order to combine the elements from two lists into one, you must first compare each list's elements

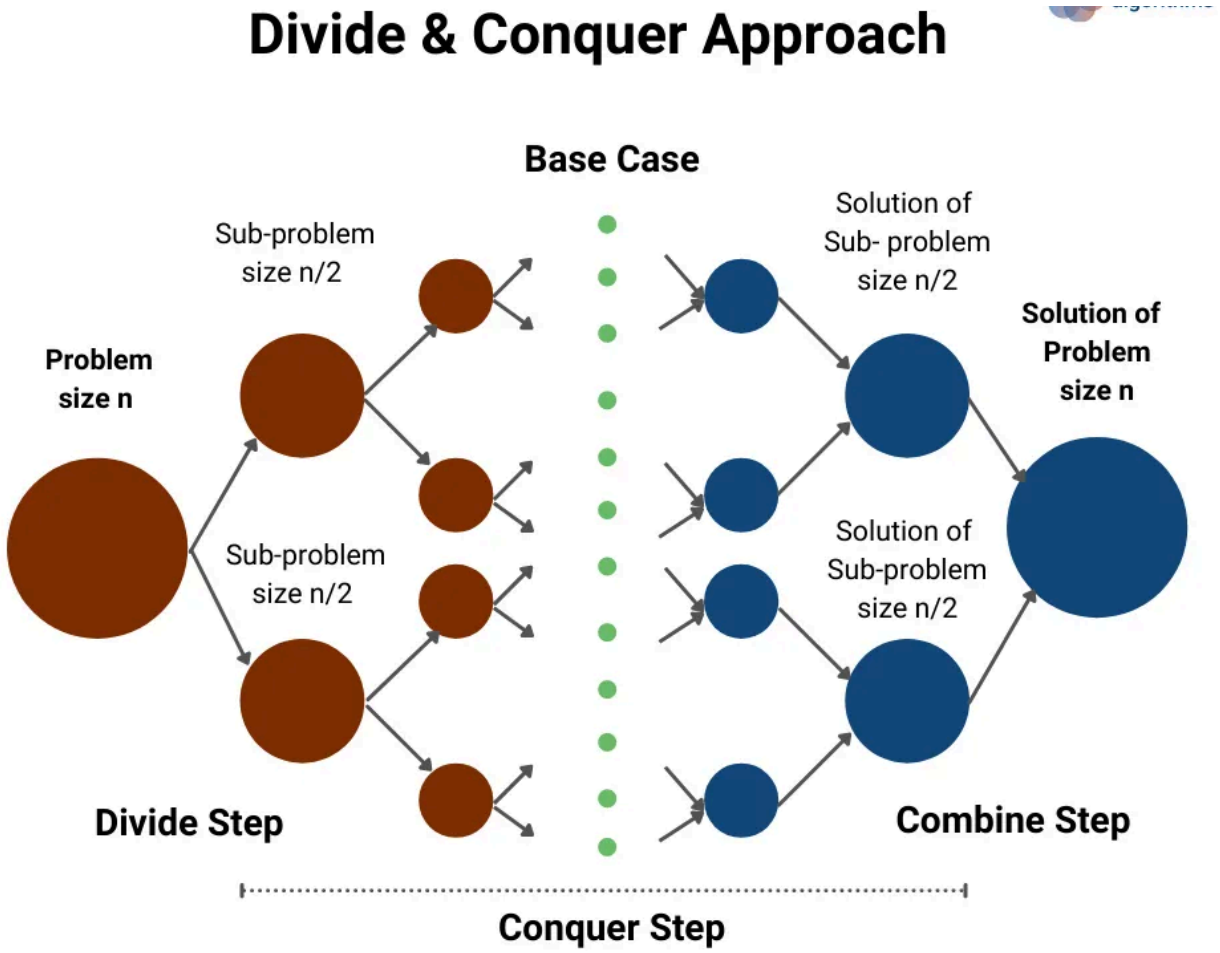


Step 5: Following the final merging, the list appears as follows:



Pseudo code and Recurrence relations:

```
1. void mergeSort(int arr[], int l, int r)
2. {
3.     if (l < r) {
4.         int m = l + (r - l) / 2;
5.         mergeSort(arr, l, m);
6.         mergeSort(arr, m + 1, r);
7.         merge(arr, l, m, r);
8.     }
9. }
```



Using master's theorem case 2 the formulated recurrence relation can be solved for merge sort using divide and conquer algorithms.

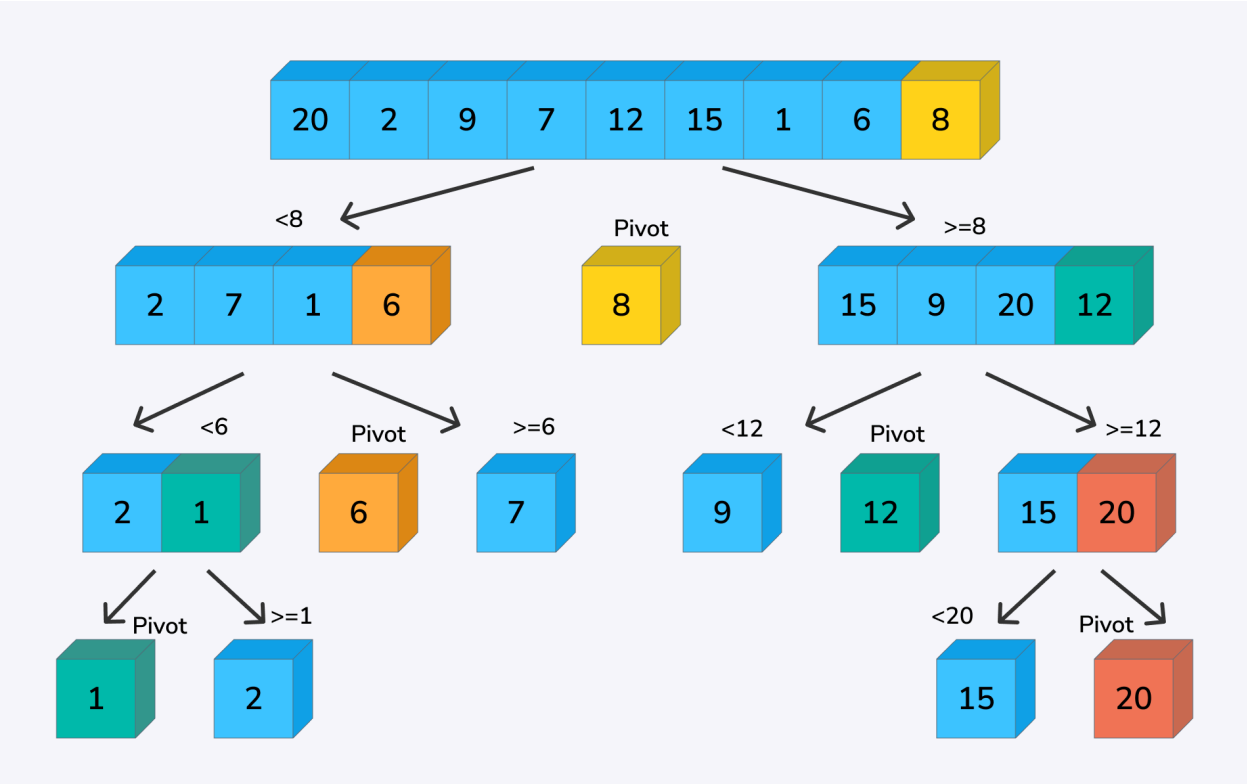
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- $a = 2; b = 2 \rightarrow n^{\log_b a} = n$
- $f(n) = n$

$$Case2: f(n) = \theta(n)$$

$$T(n) = \theta(nlgn)$$

Quick sort:



Working on Quicksort Algorithm

1. Select the Pivot Element

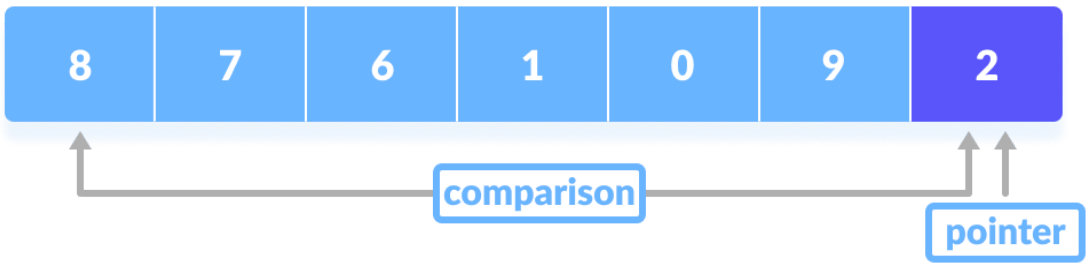
There are different variations of quicksort where the pivot element is selected from different positions. Here, we will select the array's rightmost element as the pivot element.

2. Rearrange the Array

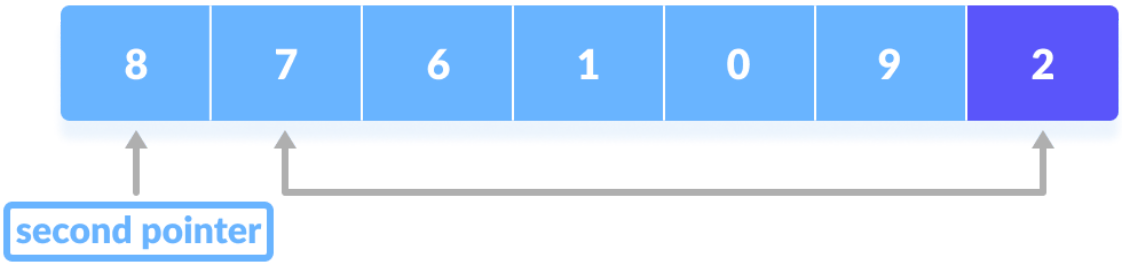
Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

Here's how we rearrange the array:

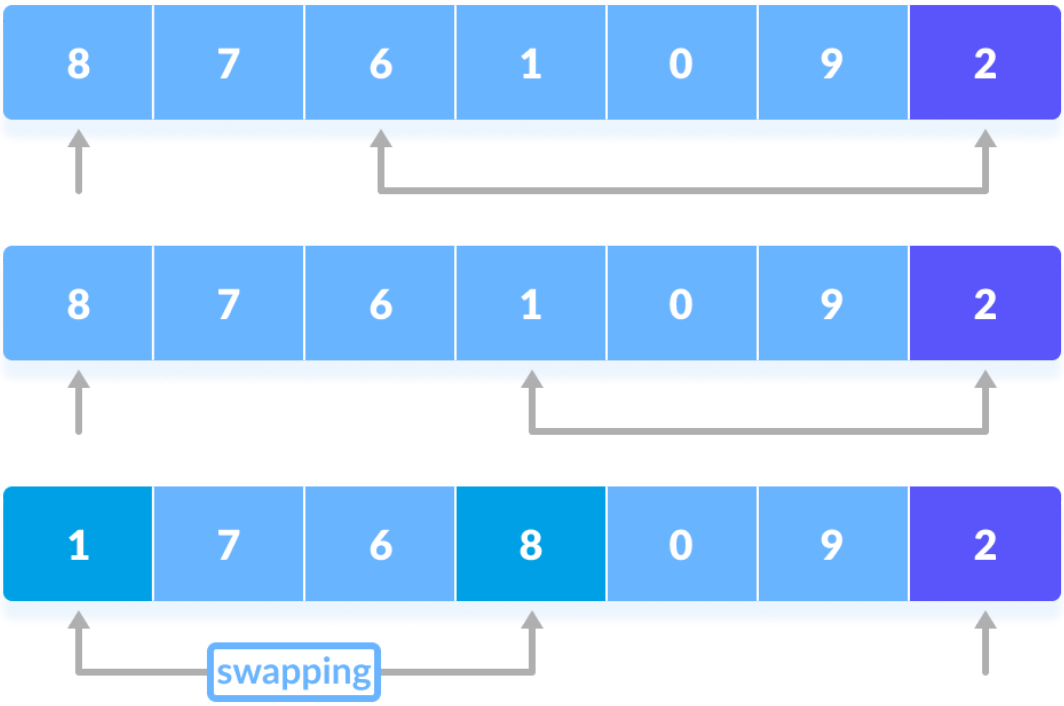
1. A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



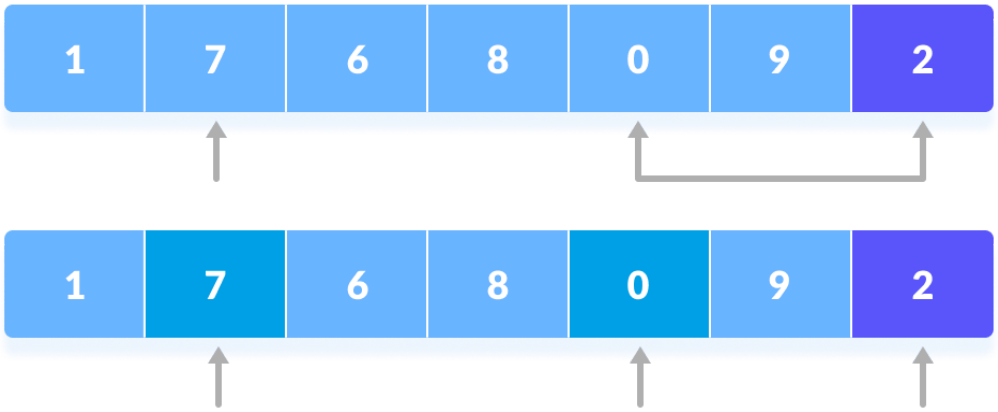
2. If the element is greater than the pivot element, a second pointer is set for that element.



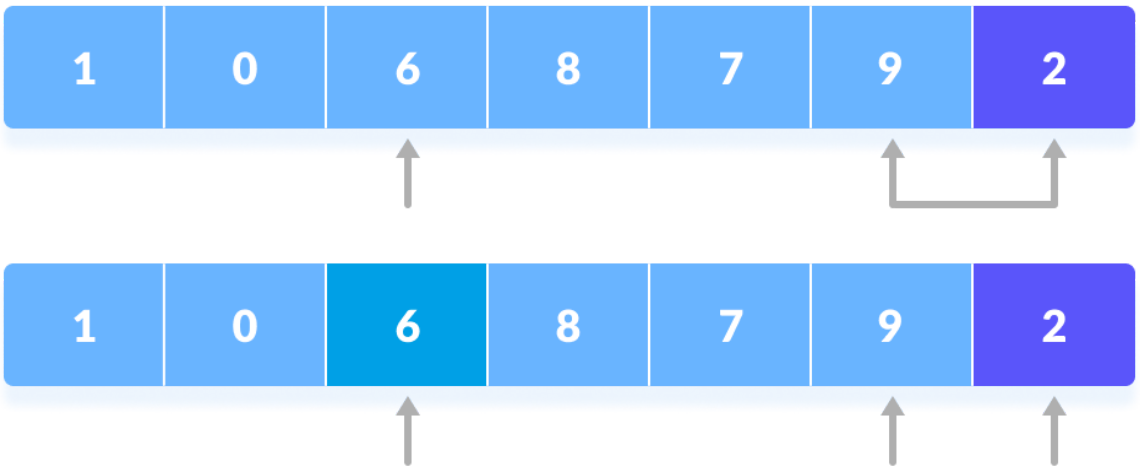
3. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



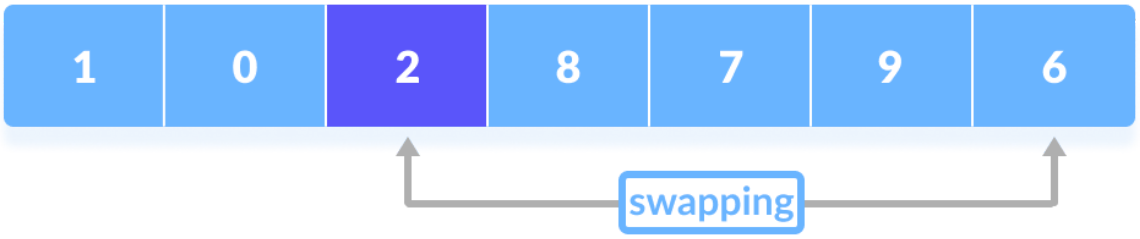
4. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



5. The process goes on until the second last element is reached.

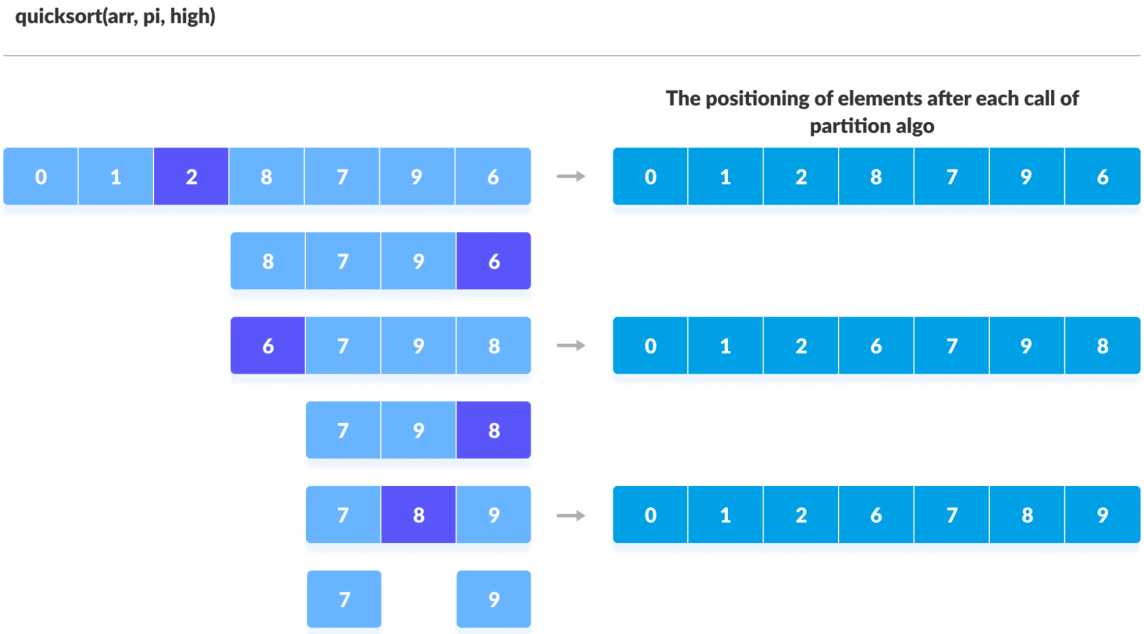


6. Finally, the pivot element is swapped with the second pointer

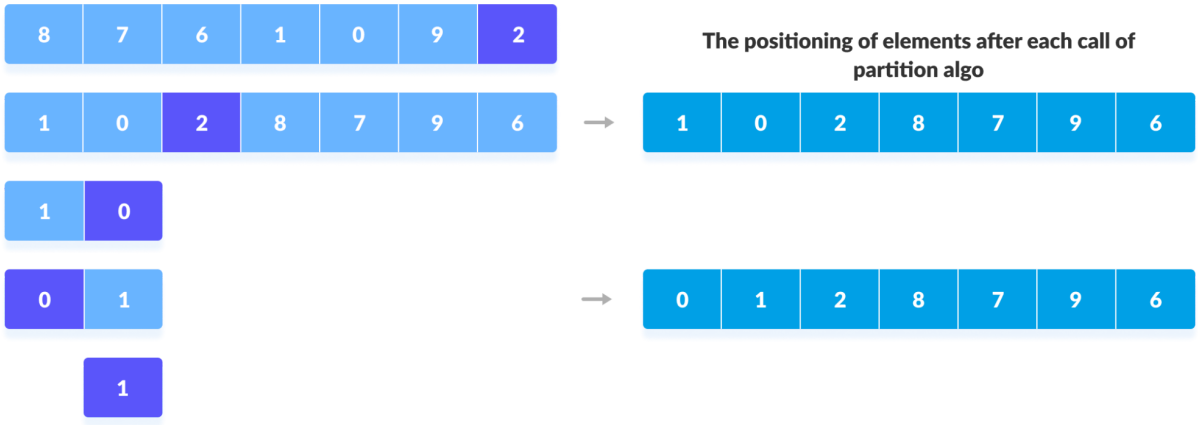


3. Divide Subarrays

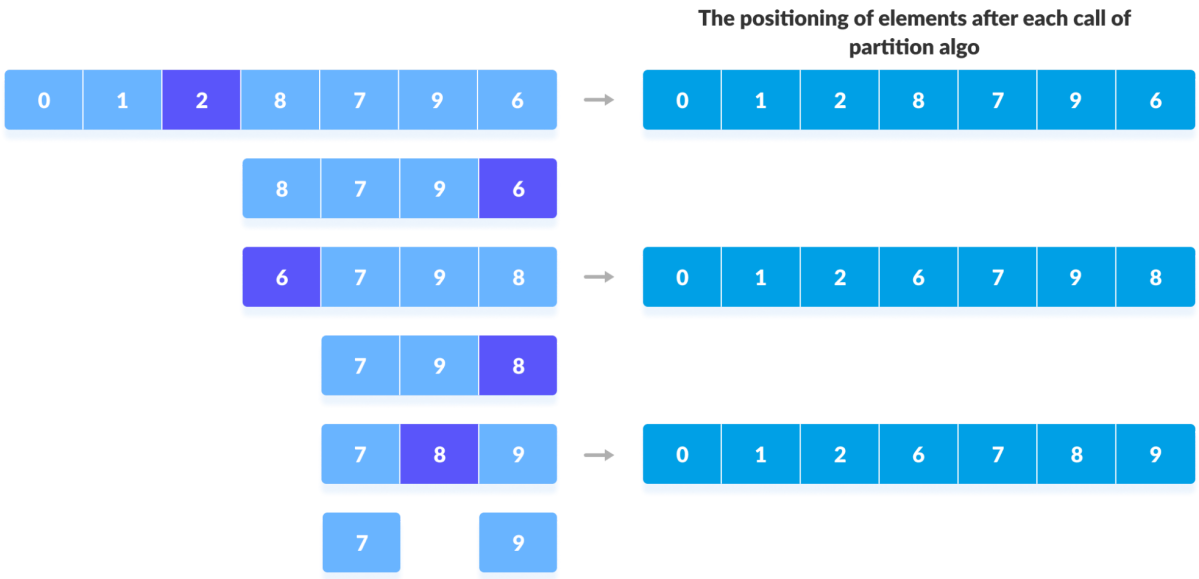
Pivot elements are again chosen for the left and the right sub-parts separately. And, **step 2** is repeated.



quicksort(arr, low, pi-1)



quicksort(arr, pi+1, high)



Pseudo code:

```
1. quickSortUtil( arr[], low, high) {
2.     if (low >= high) {
3.         return; }
4.     pivot = partition (arr, low, high);
5.     quickSortUtil(arr, low, pivot - 1);
6.     quickSortUtil(arr, pivot + 1, high);
7. }
```

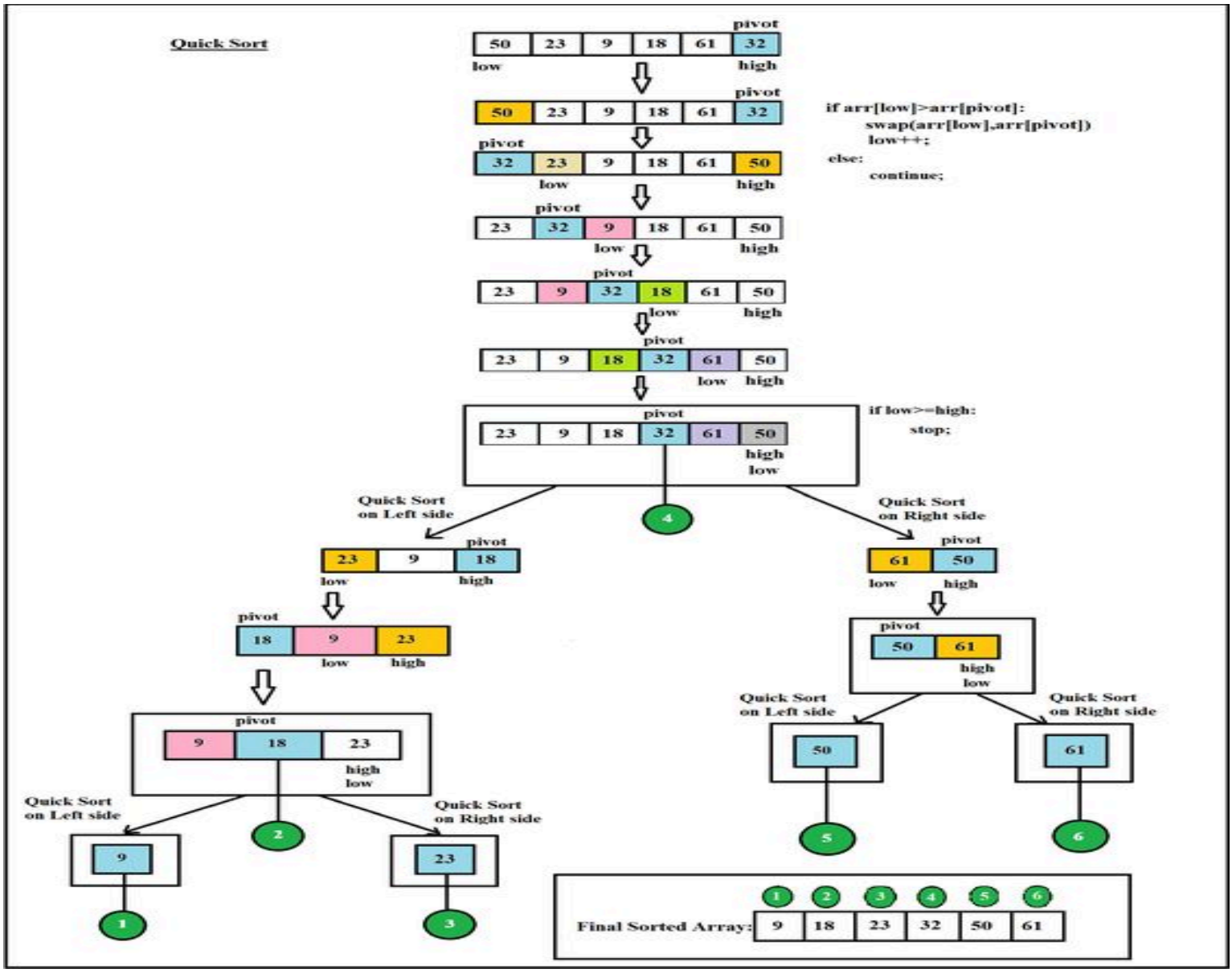
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- $a = 2; b = 2 \rightarrow n^{\log_b a} = n$
- $f(n) = n$

$$Case2: f(n) = \theta(n)$$

$$T(n) = \theta(nlgn)$$

Consider the following array: 50, 23, 9, 18, 61, 32



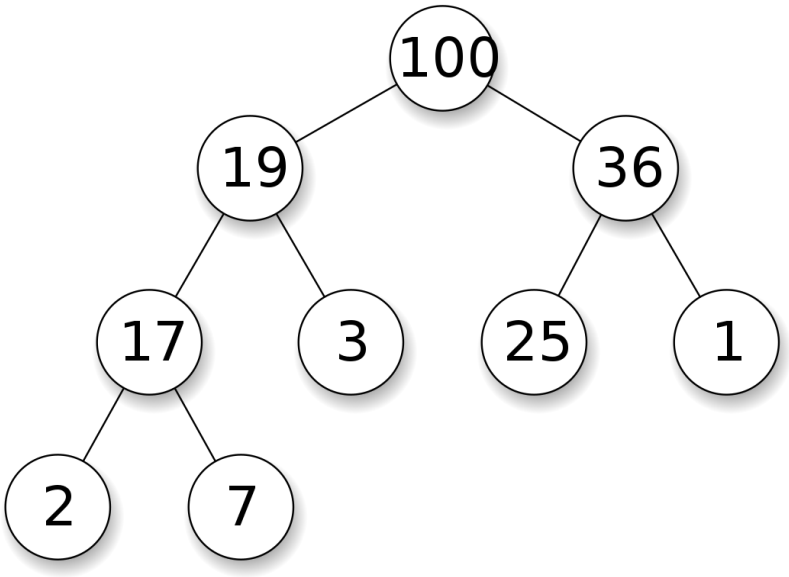
Heap sort:

A heap is a binary tree that is sorted by either of the following rules:

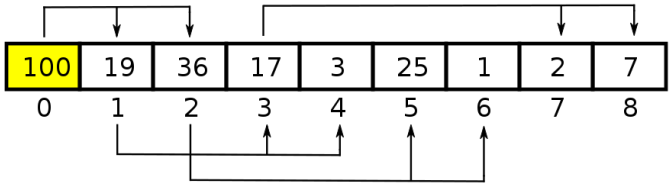
1. Node is **greater** than or equal to it's children
2. Node is **less** than or equal to it's children

Arrays can be used to represent a **binary tree**. This is done by looking at the array in a sequential order like this.

Tree representation



Array representation



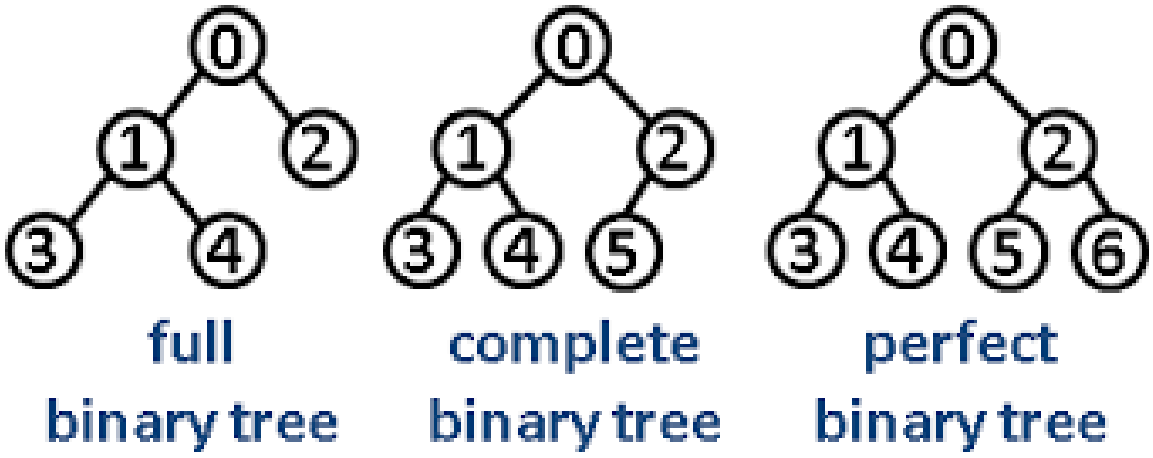
Understanding the relationships between the elements in the array and a binary tree from that array.

1. The root of the tree: A[i]
2. Left child: A[i]=A[2***i***]
3. Right child: A[i] =A[2***i***+1]
4. Parent: a[i]= A[***i***/2] //floor value
5. Heap size [A]<= Length[A]

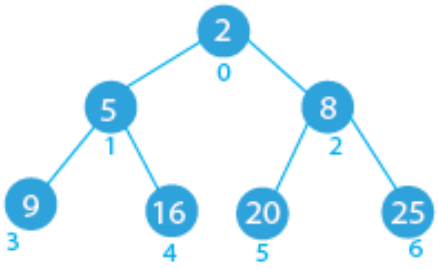
Trace a binary tree from the given array:

1. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
2. [1, 2, 3, 4, 5, 6, 7, -, -, 10, 11]
3. [1, 2, 3, 4, 5, 6, 7, 10, 11]

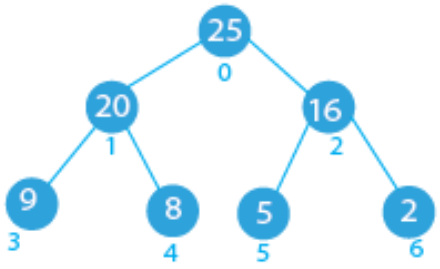
Complete and Full Binary Tree:



Max and Min heap:



Min Heap



Max Heap

Max-heaps (largest element at root), have the max-heap property:

- for all nodes i, excluding the root:

$$A[PARENT(i)] \geq A[i]$$

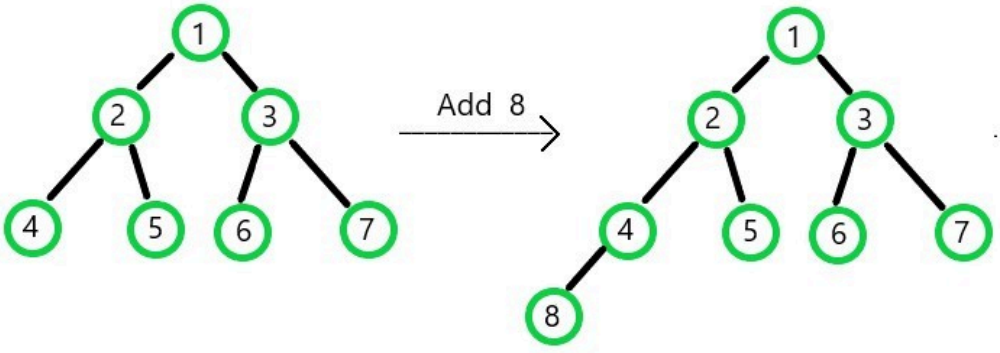
Min-heaps (smallest element at root), have the min-heap property:

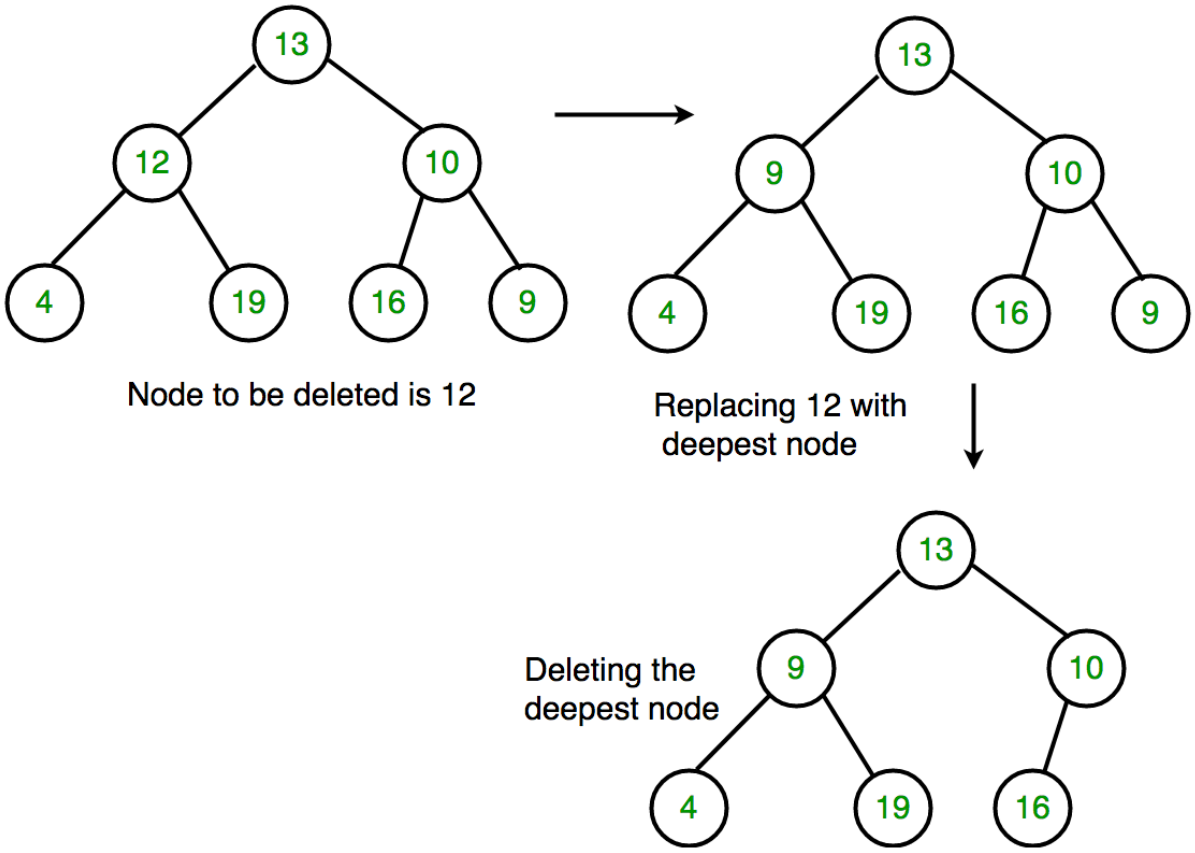
- for all nodes i, excluding the root:

$$A[PARENT(i)] \leq A[i]$$

Adding/Deleting Nodes:

New nodes are always inserted at the bottom level (left to right) and nodes are removed from the bottom level (right to left).





Operations on Heaps:
Maintain/Restore the max-heap property

MAX-HEAPIFY

Create a max-heap from an unordered array

BUILD-MAX-HEAP

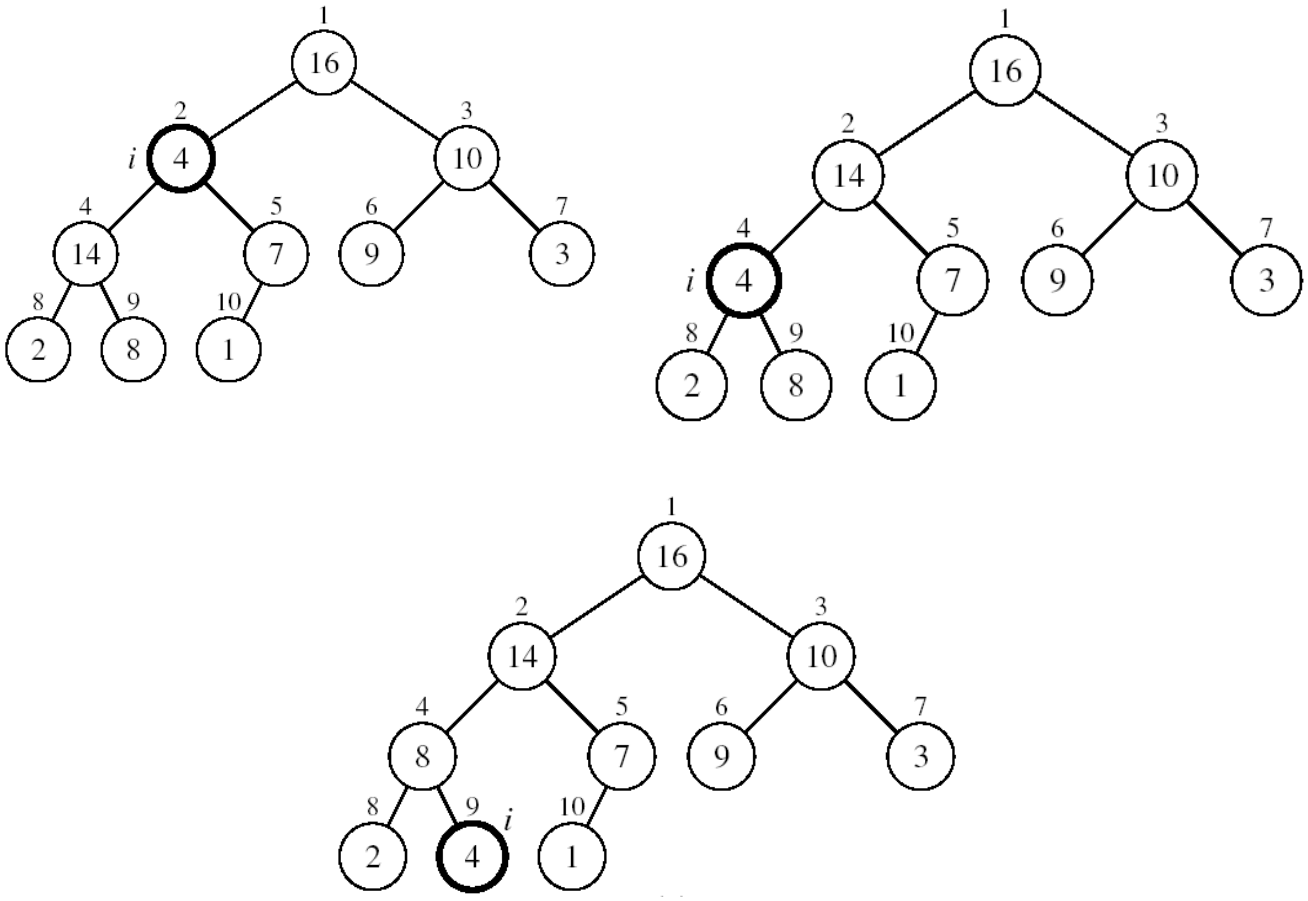
Sort an array in place

HEAPSORT

Maintaining the Heap Property

Suppose a node is smaller than a child and the Left and Right subtrees of i are max-heaps. To eliminate the violation:

- Exchange with a larger child
- Move down the tree
- Continue until the node is not smaller than the children



Heapify: Time Complexity Analysis:

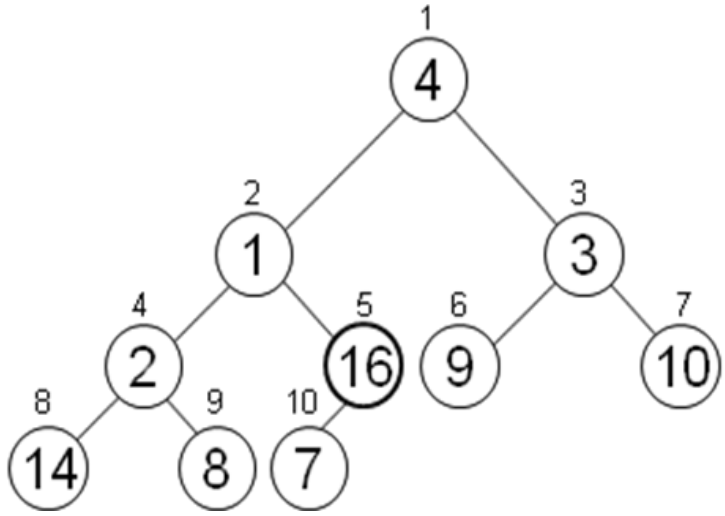
In the worst case, Max-Heapify is called recursively h times, where h is the height of the heap and since each call to the heapify takes constant time. Time complexity = O(h) = O(logn)

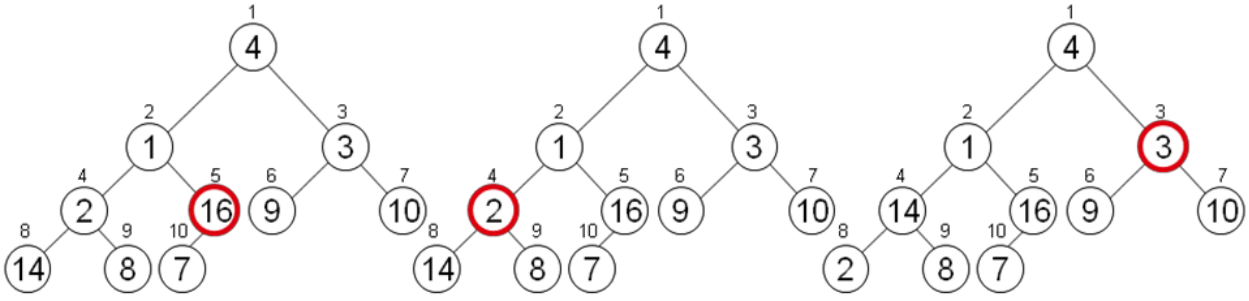
Building a Heap:

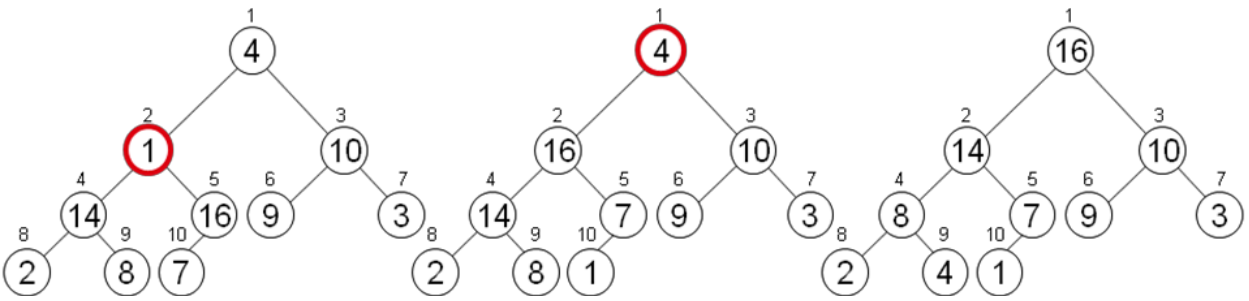
To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

Consider the following examples

- Given data sequence{4,1,3,2,16,9,10,14,8,7}
- Here we need to construct a binary tree first and
- We need to carry out heapify operations on every non-leaf node to build the Max-heap.







Algorithms:

```
1. Build-Max-Heap(A){
2. n = length[A]
3. for (i = floor(n/2);i>=1; i- -){
4. MAX-HEAPIFY(A, i, n)
5. }
6. }
```

Time Complexity:

Running time: Loop executes O(n) times and the complexity of Heapify is O(logn), therefore complexity of Build-Max-Heap is O(nlogn).

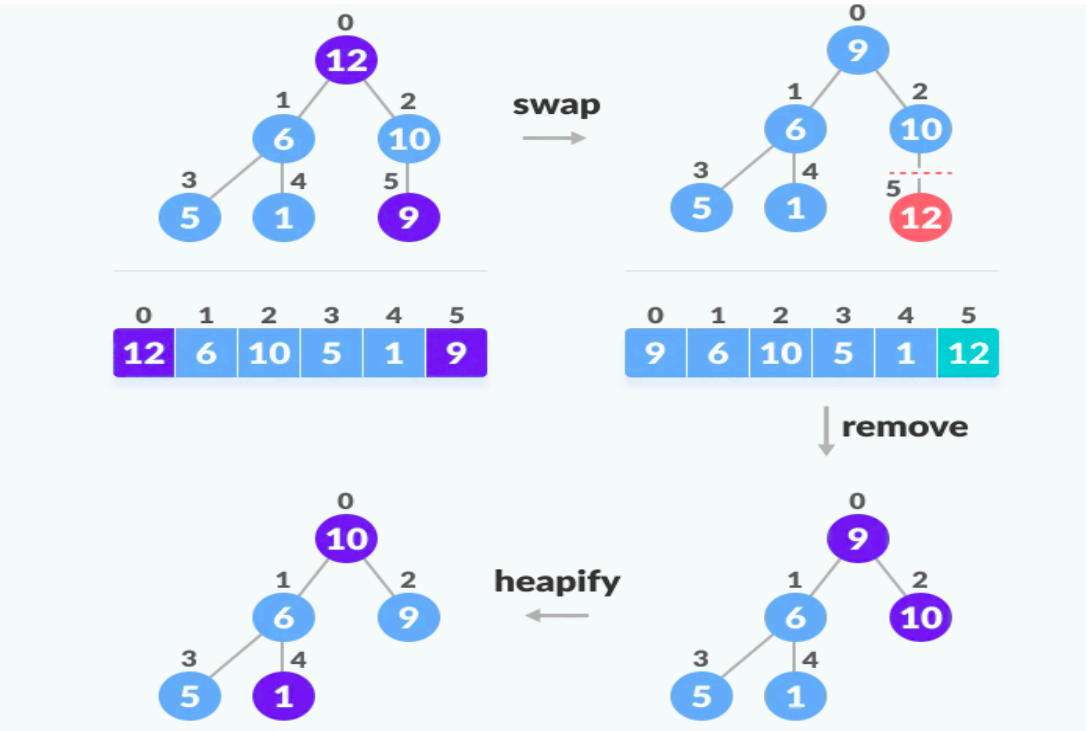
Heap sort:

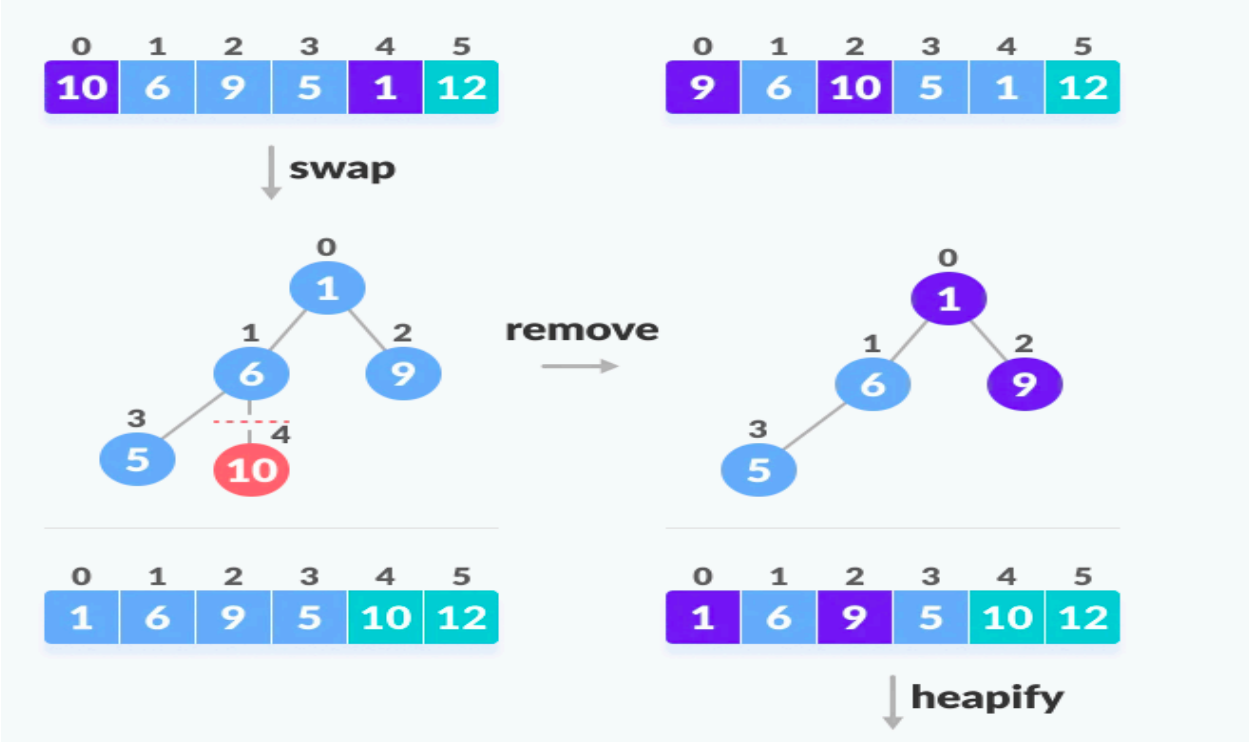
- Heap sort is a comparison-based sorting technique based on the Binary Heap data structure.
- It is similar to selection sort where we first find the minimum element and place the minimum element at the beginning.
- We repeat the same process for the remaining elements.
- Steps involved to sort n elements:
    - Build a max-heap from the array

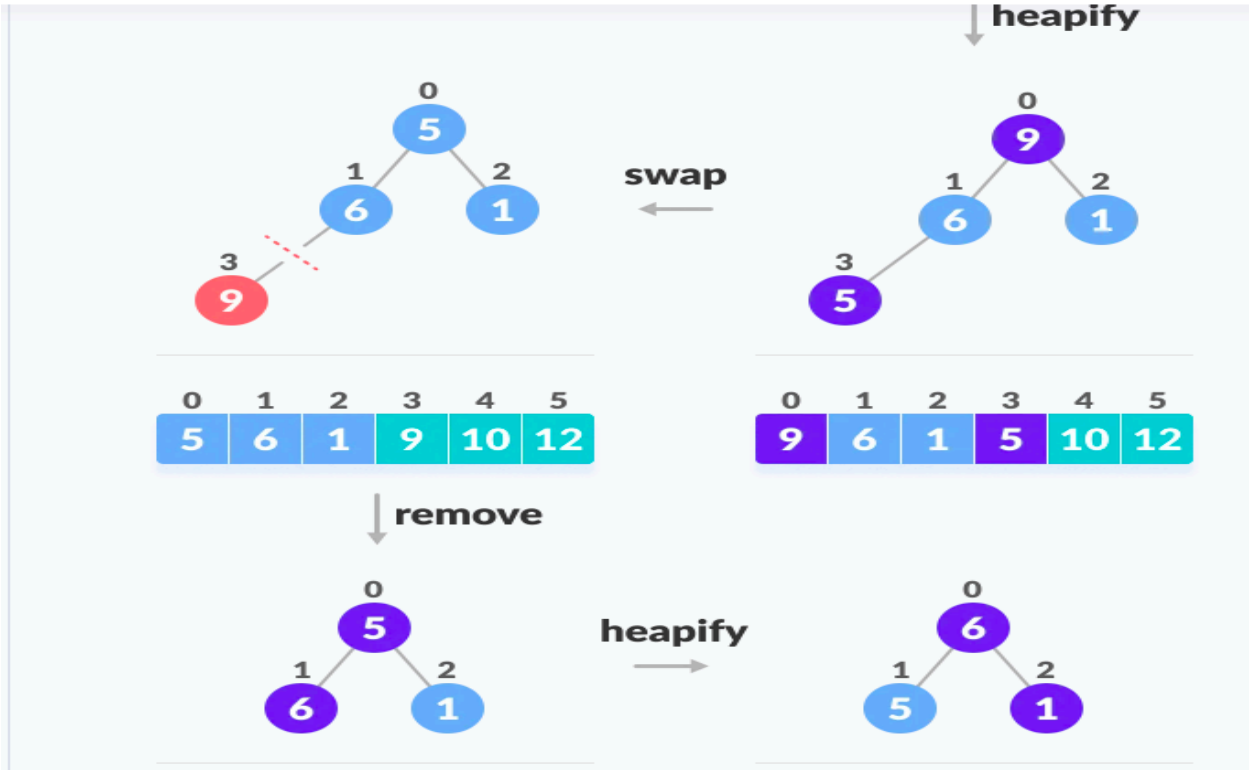
- Swap the root (the maximum element) with the last element in the array
- “Discard” this last node by decreasing the heap size
- Call Max-Heapify on the new root
- Repeat this process until only one node remains

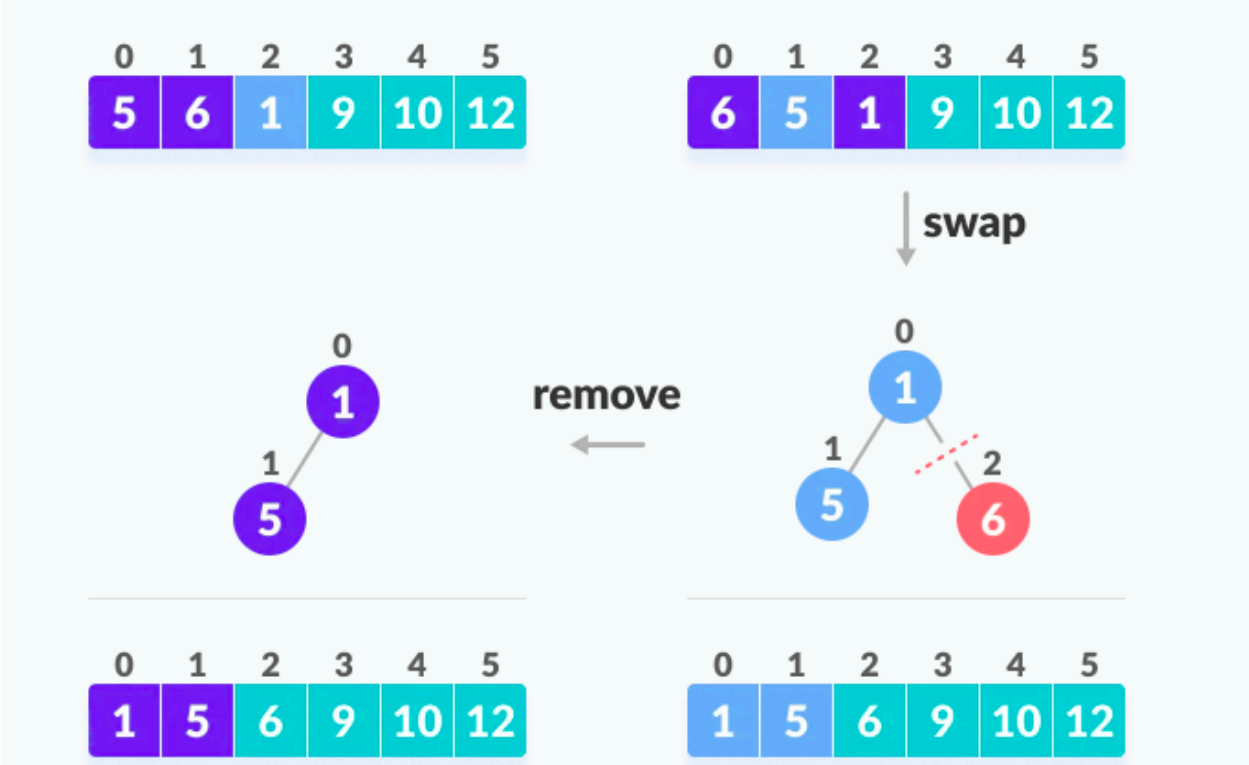
Working of Heap Sort

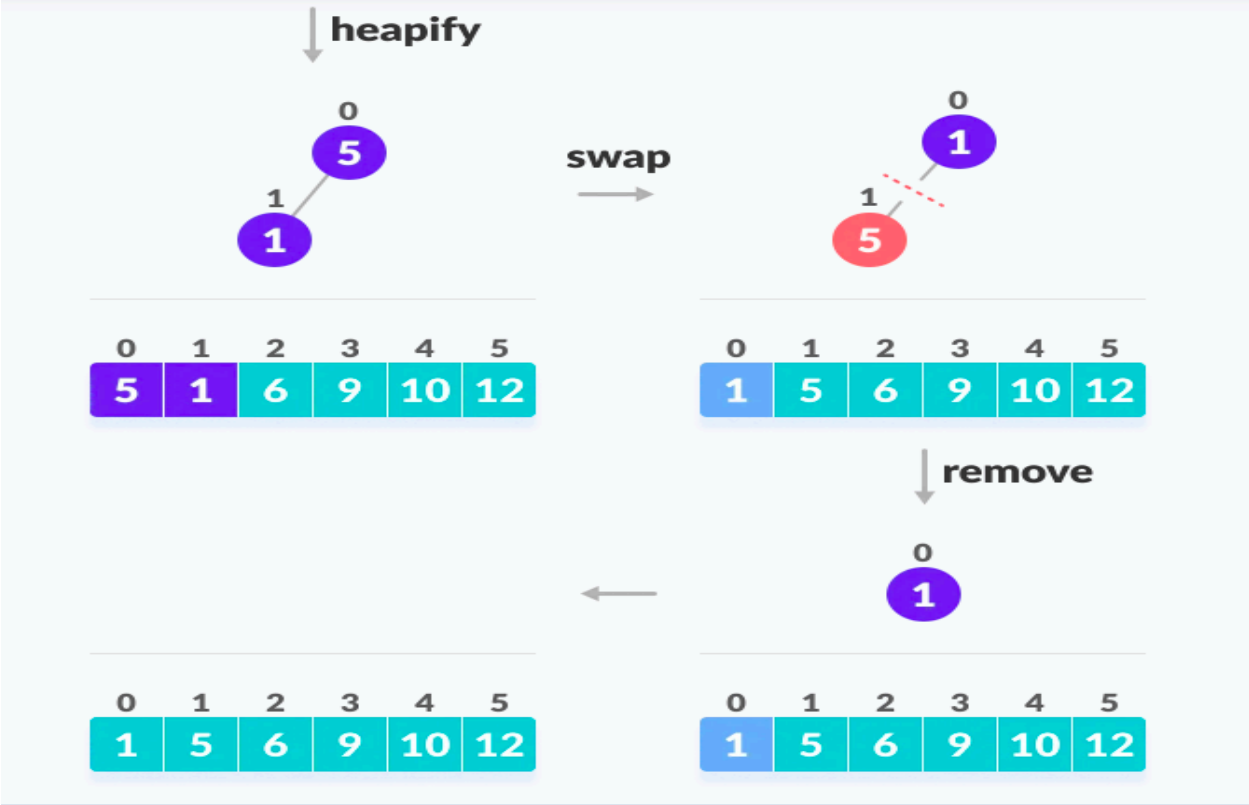
1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** 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.




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


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


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

- Sort the following elements using heap sort [1,4,2,7,3]
  - First, construct the binary tree and construct the heap of the recently constructed binary tree.
  - Then apply heap sort.

Analysis:

- Analysis:
  - Building heap takes O(n)
  - Loop executes n times
  - Heapify operations take O(logn)
  - So, total T(n)=O(nlogn)

```
HeapSort(A)
{
        BuildHeap(A); //into max heap
        n = length[A];
        for(i = n ; i >= 2; i--)
        {
                swap(A[1],A[n]);
                n = n-1;
                Heapify(A,1);
        }
}
```