➢ Insertion Sort Algorithm: -

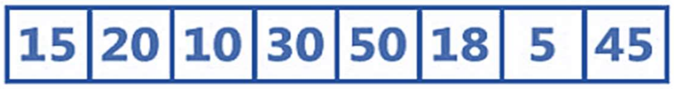
Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Steps for Insertion Sort: -

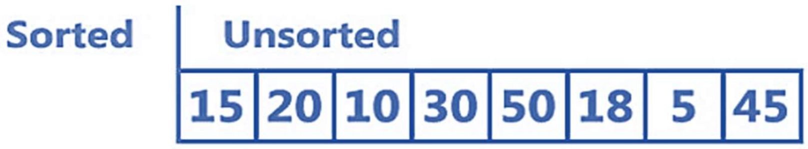
- **Step 1 -** Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- **Step 2**: Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- **Step 3**: Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Example: -

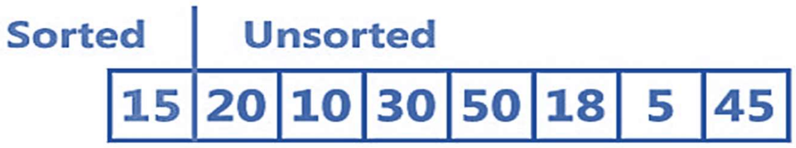
Consider the following unsorted list elements



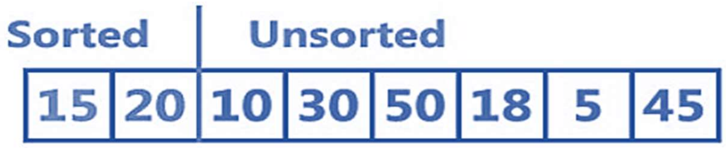
Assume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below



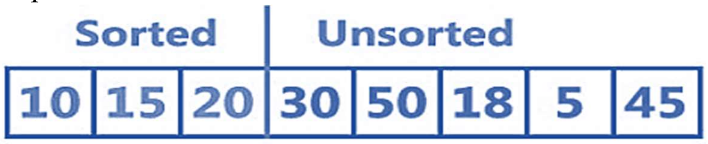
Move the first element 15 from unsorted portion to sorted portion of the list



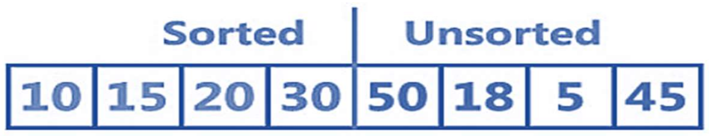
To move element 20 from unsorted to sorted portion, compare 20 with 15 and insert it at correct position



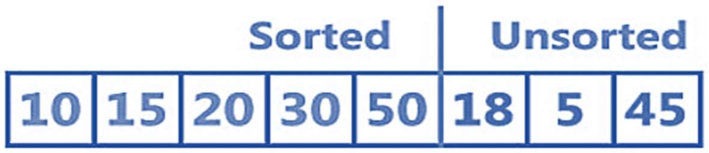
To move element 10 from unsorted to sorted portion, compare 10 with 20 and it is smaller so swap then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list



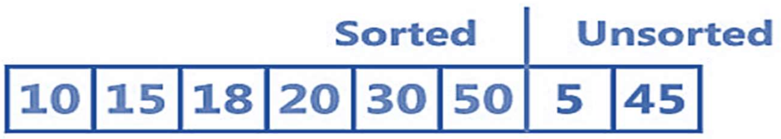
To move element 30 from unsorted to sorted portion, compare 30 with 20, 15 and 10 and it is larger than all these so 30 is directly at last position in sorted portion of the list



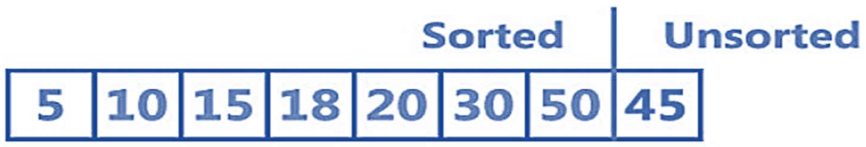
To move element 50 from unsorted to sorted portion, compare 50 with 30, 20, 15 and 10 And it is larger then all these so 50 is directly at last position in sorted portion of the list



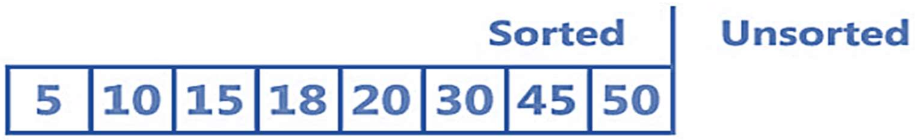
To move element 18 from unsorted to sorted portion, compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion



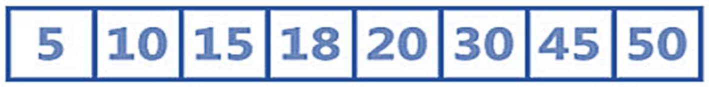
To move element 5 from unsorted to sorted portion, compare 5 with 50, 30, 20, 18, 15 and 10 Since 5 is smaller than all these elements move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list



To move element 45 from unsorted to sorted portion Compare 45 with 50 and 30 Since 45 is larger than 30 move 50 one position to the right in the list and insert 45 after 30 in the sorted list



Unsorted portion of the list has become empty. So, we stop the process and the final sorted list of elements as follows



Complexity of the Insertion Sort Algorithm

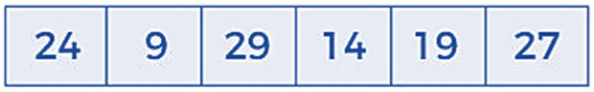
- Worst Case: $O(n^2)$
- Best Case: $\Omega(n)$
- Average Case: $\Theta(n^2)$

➢ Quick Sort Algorithm: -

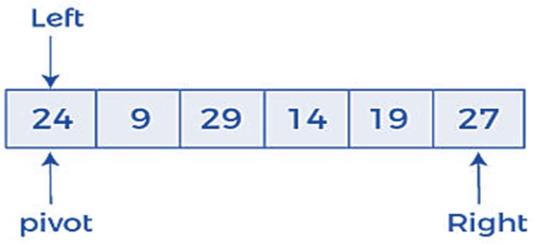
Quick sort is a fast-sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by C. A. R. Hoare.
The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use divide and conquer strategy. In quick sort, the partition of the list is performed based on the element called pivot. Here pivot element is one of the elements in the list.
The list is divided into two partitions such that "all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot".

Working of Quick Sort Algorithm: -
To understand the working of quick sort, let's take an unsorted list. It will make the concept clearer and more understandable.

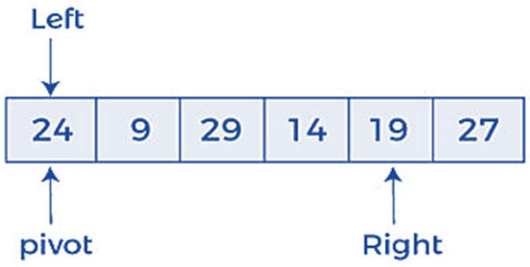
Let the elements of list are



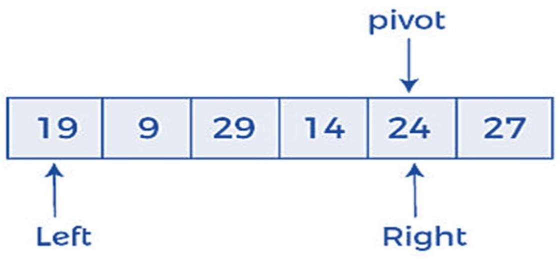
In the given list, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.
Since, pivot is at left, so algorithm starts from right and move towards left.



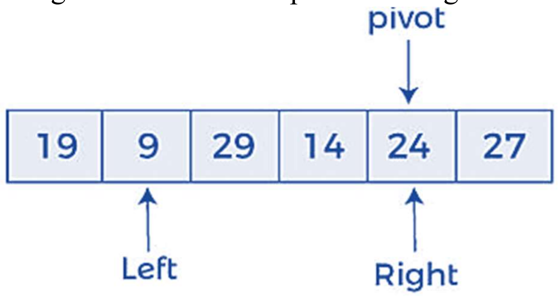
Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e.,



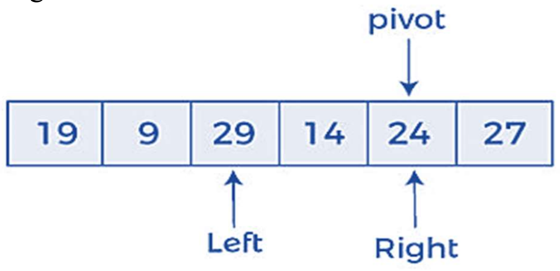
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.
Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as –



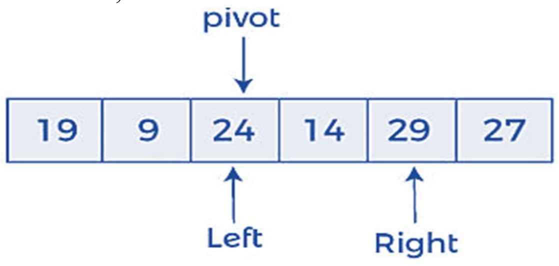
Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.
As a[pivot] > a[left], so algorithm moves one position to right as –



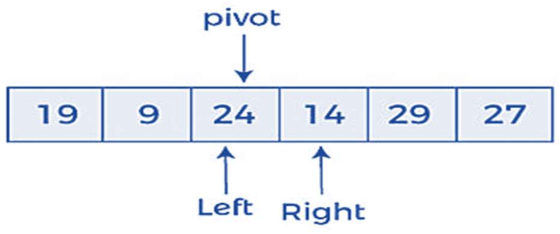
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as –



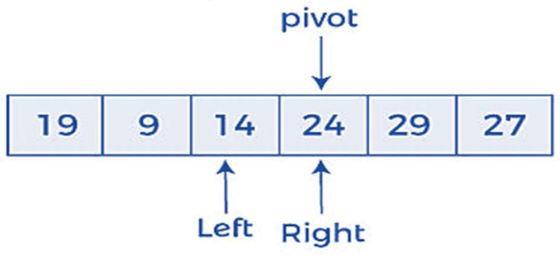
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. –



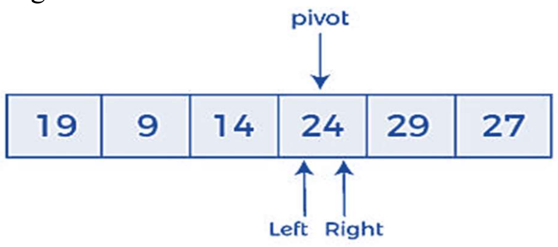
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as –



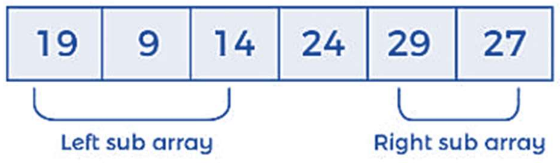
Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. –



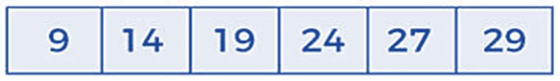
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.
Element 24, which is the pivot element is placed at its exact position.
Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-list. After sorting gets done, the list will be –



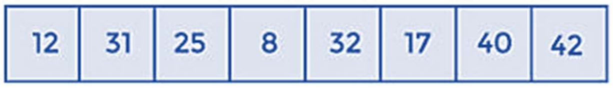
➢ Merge Sort: -

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithms. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.
The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs are merged into the four-element lists, and so on until we get the sorted list.

Working of Merge sort Algorithm: -

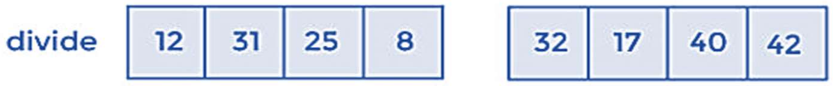
To understand the working of the merge sort algorithm, let's take an unsorted list. It will be easier to understand the merge sort via an example.

Let the elements of list are –

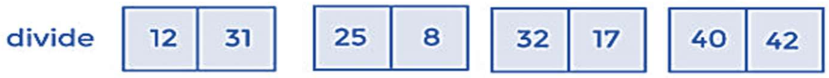


According to the merge sort, first divide the given list into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

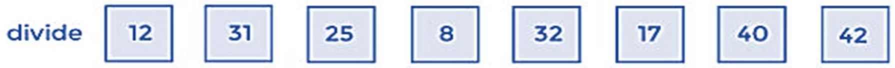
As there are eight elements in the given list, so it is divided into two list of size 4.



Now, again divide these two list into halves. As they are of size 4, so divide them into new list of size 2.



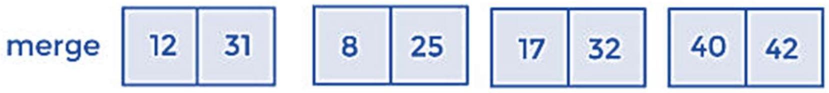
Now, again divide these lists to get the atomic value that cannot be further divided.



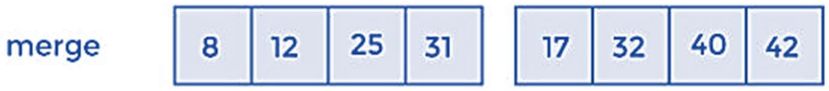
Now, combine them in the same manner they were broken.

In combining, first compare the element of each list and then combine them into another list in sorted order.

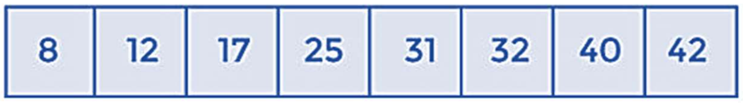
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the list with two data values and merge them into a list of found values in sorted order.



Now, there is a final merging of the list. After the final merging of above list, the list will look like –



Now, the list is completely sorted.

➢ Heap Sort Algorithm: -

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called Heap Tree. In this sorting algorithm, we use Max Heap to arrange list of elements in Descending order and Min Heap to arrange list elements in Ascending order.

Steps for Heap Sort: -

The Heap sort algorithm to arrange a list of elements in descending order is performed using following steps...

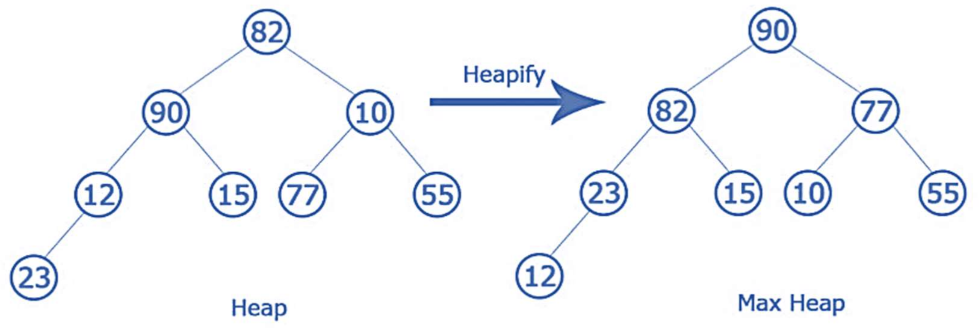
- **Step 1** - Construct a Binary Tree with given list of Elements.
- **Step 2** - Transform the Binary Tree into Max Heap.
- **Step 3** - Delete the root element from Max Heap using Heapify method.
- **Step 4** - Put the deleted element into the Sorted list.
- **Step 5** - Repeat the same until Max Heap becomes empty.
- **Step 6** - Display the sorted list.

Example: -

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

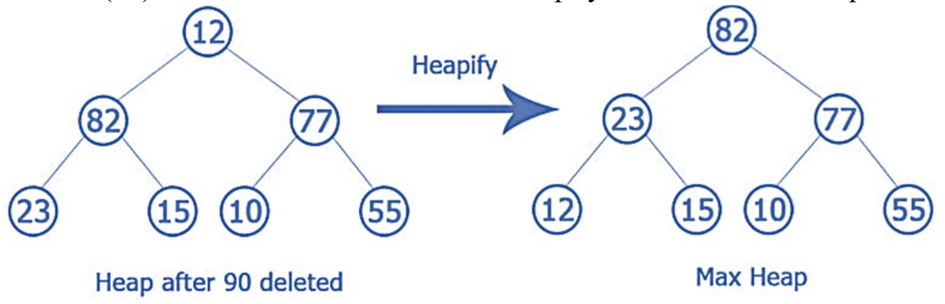
Step 1: Construct a Heap with given list of unsorted numbers and convert to Max Heap



List of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

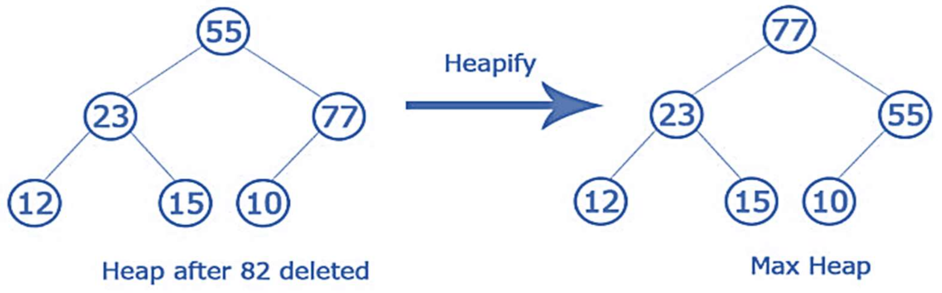
Step 2: Delete root (20) from the Max Heap. To delete root node, it needs to be swapped with last node (12). After delete tree needs to be heapify to make it Max Heap



List of numbers after swapping 90 with 20

12, 82, 77, 23, 15, 10, 55, **90**

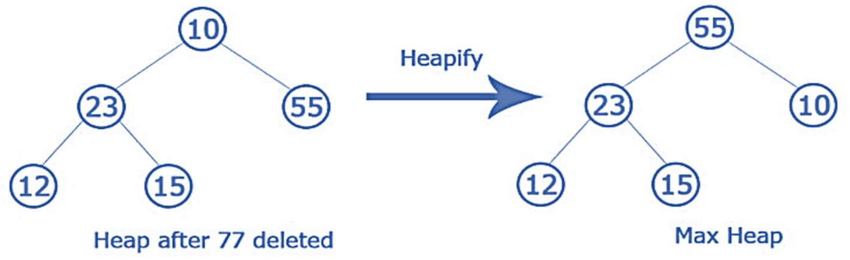
Step 3: Delete root (82) from the Max Heap. To delete root node it needs to be swapped with last node (55) After delete tree needs to be heapify to make it Max Heap



List of numbers after swapping 82 with 55

12, 55, 77, 23, 15, 10, **82, 90**

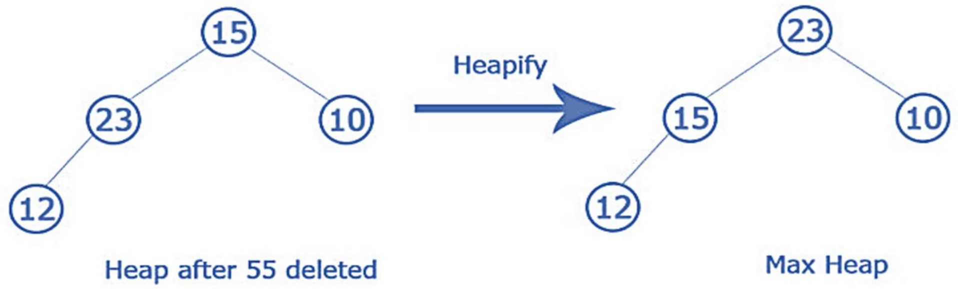
Step 4: Delete root (77) from the Max Heap. To delete root, it needs to be swapped with last node (10). After delete tree needs to be heapify to make it Max Heap



List of numbers after swapping 77 with 10

12, 55, 10, 23, 15, **77, 82, 90**

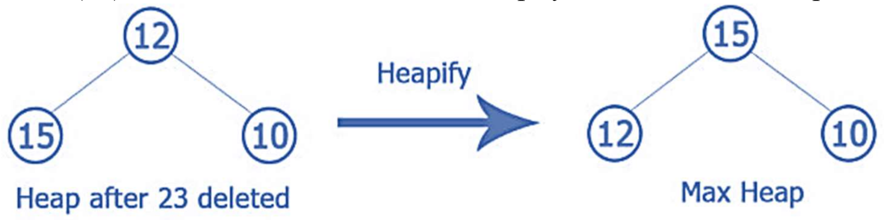
Step 5: Delete root (55) from the Max Heap. To delete root, node it needs to be swapped with last node (15) After delete tree needs to be heapify to make it Max Heap



List of numbers after swapping 55 with 15

12, 15, 10, 23, **55, 77, 82, 90**

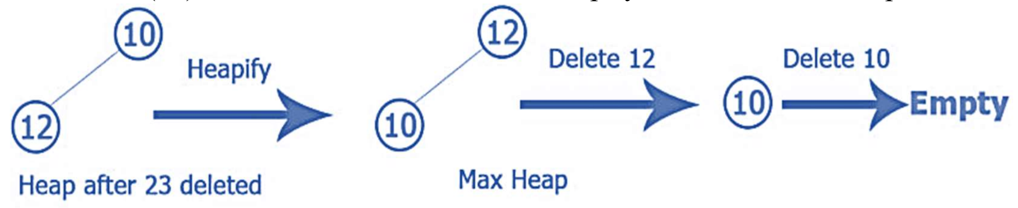
Step 6: Delete root (23) from the Max Heap. To delete root, node it needs to be swapped with last node (12) After delete tree needs to be heapify to make it Max Heap



List of numbers after swapping 23 with 12

12, 15, 10, 23, 55, 77, 82, 90

Step 7: Delete root (15) from the Max Heap. To delete root, node it needs to be swapped with last node (10) After delete tree needs to be heapify to make it Max Heap



List of numbers after deleting 15, 12 and 10 from the Max Heap

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list gets sorted in Ascending order

➢ Bubble Sort: -

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.
Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is O(n2), where n is a number of items.

Bubble short is majorly used where –

- complexity does not matter
- simple and short code is preferred

Working of Bubble sort: -

To understand the working of bubble sort algorithm, let's take an unsorted list. We are taking a short and accurate list, as we know the complexity of bubble sort is O(n2).
Let the elements of list are

13	32	26	35	10

First Pass: -

Sorting will start from the initial two elements. Let compare them to check which is greater.

13	32	26	35	10

Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.

13	32	26	35	10

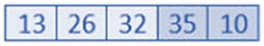
Here, 26 is smaller than 36. So, swapping is required. After swapping new list will look like –

13	26	32	35	10

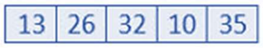
Now, compare 32 and 35.

13	26	32	35	10

Here, 35 is greater than 32. So, there is no swapping required as they are already sorted. Now, the comparison will be in between 35 and 10.

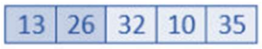


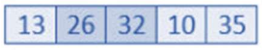
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the list. After first pass, the list will be –

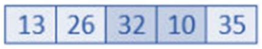


Now, move to the second iteration.

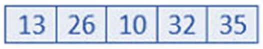
Second Pass: -
The same process will be followed for second iteration.

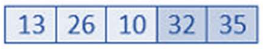






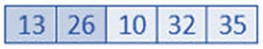
Here, 10 is smaller than 32. So, swapping is required. After swapping, the list will be

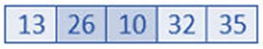




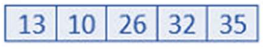
Now, move to the third iteration.

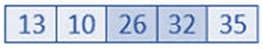
Third Pass: -
The same process will be followed for third iteration.

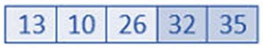




Here, 10 is smaller than 26. So, swapping is required. After swapping, the list will be –

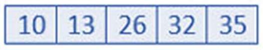






Now, move to the fourth iteration

Fourth Pass: -
Similarly, after the fourth iteration, the list will be –

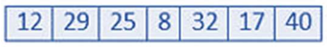


Hence, there is no swapping required, so the list is completely sorted.

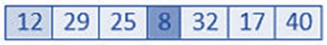
➢ Selection Sort: -

In selection sort, the smallest value among the unsorted elements of the list is selected in every pass and inserted to its appropriate position into the list. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the list is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the list is empty, and unsorted part is the given list. Sorted part is placed at the left, while the unsorted part is placed at the right.
In selection sort, the first smallest element is selected from the unsorted list and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the list is entirely sorted.

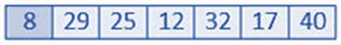
Working of Selection sort Algorithm: -
To understand the working of the Selection sort algorithm, let's take an unsorted list. It will be easier to understand the Selection sort via an example.
Let the elements of list are –



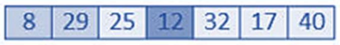
Now, for the first position in the sorted list, the entire list is to be scanned sequentially. At present, 12 is stored at the first position, after searching the entire list, it is found that 8 is the smallest value.



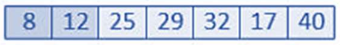
So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted list.



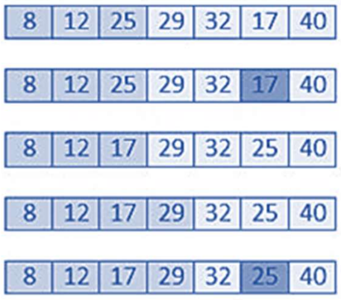
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted list. After scanning, we find that 12 is the second lowest element in the list that should be appeared at second position.

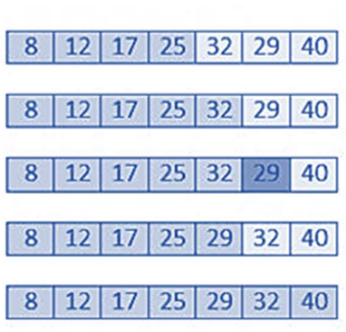


Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted list. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



The same process is applied to the rest of the list elements. Now, we are showing a pictorial representation of the entire sorting process.





Now, the list is completely sorted.