NextGen Academy
Towards fulfilling a million dreams

Data Structures Using C

Module V: Sorting & Hashing

June 20, 2023

Contents

This unit covers

- Sorting Algorithms Overview: Introduces Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, and Heap Sort, outlining their individual mechanisms and characteristics.
- Performance Analysis: Examines and compares the efficiency metrics (time and space complexities) of the sorting techniques, considering their best, worst, and average-case scenarios.
- Hashing Fundamentals: Discusses Hash Functions and their variants, emphasizing their role in creating Hash Tables for efficient data storage and retrieval.
- Hash Table Construction: Details the process of constructing Hash Tables, highlighting the principles and methodologies involved in organizing data using hashing.
- Collision Resolution Techniques: Explores various methods to handle collisions within Hash Tables, including strategies like Universal Addressing and Open Hashing, ensuring efficient data storage and access.
- Practical Application: Illustrates real-world applications where sorting algorithms and hashing techniques are employed, demonstrating their significance in solving various computational problems.

1 Sorting Algorithms and their Analysis:

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

For Example: The below list of characters is sorted in increasing order of their ASCII values. That is, the character with a lesser ASCII value will be placed first than the character with a higher ASCII value.

Sorting Algorithms:

SN	Sorting Algorithms	Description
1	Bubble Sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly.
2	Bucket Sort	Bucket sort is also known as bin sort. It works by distributing the element into the array also called buckets. In this sorting algorithms, Buckets are sorted individually by using different sorting algorithm.
3	Comb Sort	Comb Sort is the advanced form of Bubble Sort. Bubble Sort compares all the adjacent values while comb sort removes all the turtle values or small values near the end of the list.
4	Counting Sort	It is a sorting technique based on the keys i.e. objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of objects and stores its key values. New array is formed by adding previous key elements and assigning to objects.
5	Heap Sort	In the heap sort, Min heap or max heap is maintained from the array elements deending upon the choice and the elements are sorted by deleting the root element of the heap.
6	Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge.
7	Merge Sort	Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array.
8	Quick Sort	Quick sort is the most optimized sort algorithms which performs sorting in O(n log n) comparisons. Like Merge sort, quick sort also work by using divide and conquer approach.
9	Radix Sort	In Radix sort, the sorting is done as we do sort the names according to their alphabetical order. It is the linear sorting algorithm used for Inegers.

SN	Sorting Algorithms	Description
10	Selection Sort	Selection sort finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and place it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time O(n2) which is worst than insertion sort.
11	Shell Sort	Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

Time and Space Complexity Comparison Table :

Sorting Algorithm	Time Complexity		
	Best Case	Average Case	Worst Case
Bubble Sort	(N)	(N2)	O(N2)
Selection Sort	(N2)	(N2)	O(N2)
Insertion Sort	(N)	(N2)	O(N2)
Merge Sort	(N log N)	(N log N)	O(N log N)
Heap Sort	(N log N)	(N log N)	O(N log N)
Quick Sort	(N log N)	(N log N)	O(N2)
Radix Sort	(N k)	(N k)	O(N k)
Count Sort	(N + k)	(N + k)	O(N + k)
Bucket Sort	(N + k)	(N + k)	O(N2)

Sort stability, Efficiency, Passes Comparison Table :

Sorting algorithm	Efficiency	Passes	Sort stability
Bubble sort	0(n2)	n-1	stable
Selection sort	0(n2)	n-1	stable
Insertion sort	0(n)	n-1	stable
Best case	0(n2)	n-1	
Worst case			
Quick sort	0(n log n)	log n	unstable
Best case	0(n2)	n-1	
Worst case			
Merge sort	0(n log n)	log n	stable
Shell sort	0(n)	log n	unstable
Best case	0(n2)	log n	
Worst case			

Sorting algorithm	Efficiency	Passes	Sort stability
Radix sort	0(n)	No. of digits in the largest number	stable

1.1 Selection Sort

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

The average and worst-case complexity of selection sort is $\mathbf{O(n^2)}$, where **n** is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

Algorithm

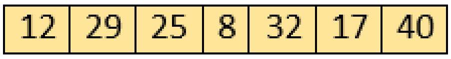
```
SELECTION SORT(arr, n)


Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
Step 2: CALL SMALLEST(arr, i, n, pos)
Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]
Step 4: EXIT


SMALLEST (arr, i, n, pos)
Step 1: [INITIALIZE] SET SMALL = arr[i]
Step 2: [INITIALIZE] SET pos = i
Step 3: Repeat for j = i+1 to n
```

```
if (SMALL > arr[j])
    SET SMALL = arr[j]
SET pos = j
[END OF if]
[END OF LOOP]
Step 4: RETURN pos
```

1.1.1 Working of Selection sort Algorithm

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

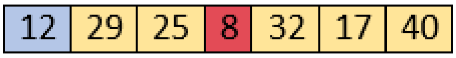
Let the elements of array are -



12	29	25	8	32	17	40

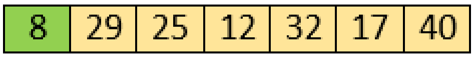
Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.



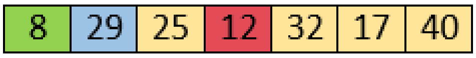
12	29	25	8	32	17	40

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



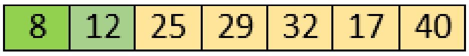
8	29	25	12	32	17	40

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



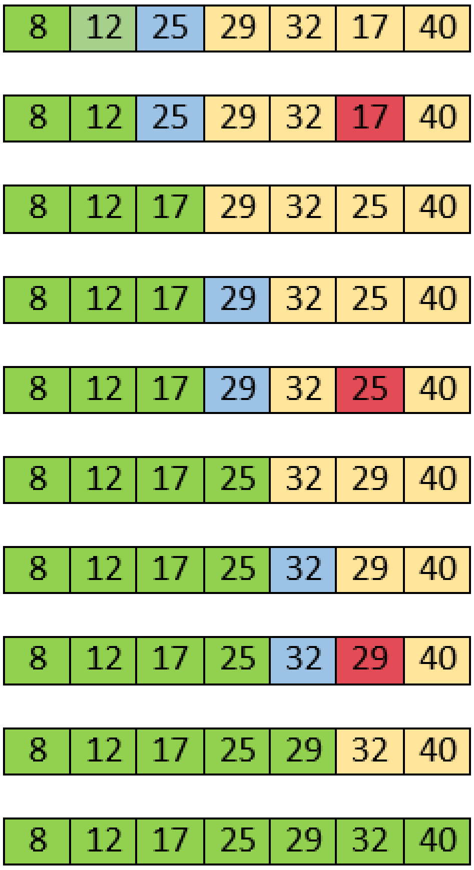
8	29	25	12	32	17	40

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



8	12	25	29	32	17	40

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



8	12	25	29	32	17	40
8	12	25	29	32	17	40
8	12	17	29	32	25	40
8	12	17	29	32	25	40
8	12	17	29	32	25	40
8	12	17	25	32	29	40
8	12	17	25	32	29	40
8	12	17	25	32	29	40
8	12	17	25	29	32	40
8	12	17	25	29	32	40

Now, the array is completely sorted.

1.1.2 Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

Time Complexity

Case	**Time Complexity**
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is $\mathbf{O(n^2)}$.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is $\mathbf{O(n^2)}$.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is $\mathbf{O(n^2)}$.

Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

1.1.3 Implementation of selection sort

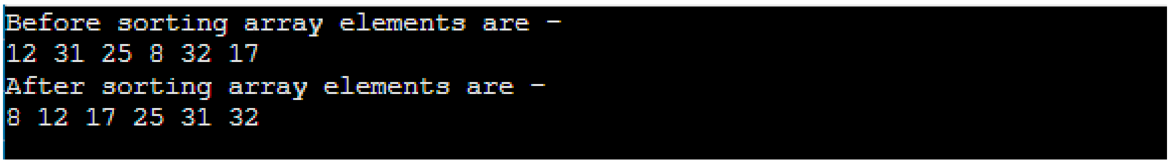
Now, let's see the programs of selection sort in different programming languages.

Program: Write a program to implement selection sort in C language.

```
#include <stdio.h>

void selection(int arr[], int n)
{
    int i, j, small;

    for (i = 0; i < n-1; i++)    // One by one move boundary of unsorted subarray
    {
        small = i; //minimum element in unsorted array

        for (j = i+1; j < n; j++)
        if (arr[j] < arr[small])
            small = j;
// Swap the minimum element with the first element
    int temp = arr[small];
    arr[small] = arr[i];
    arr[i] = temp;
    }
}

void printArr(int a[], int n) /* function to print the array */
```

```
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}


int main()
{
    int a[] = { 12, 31, 25, 8, 32, 17 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    selection(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

OUTPUT:



```
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

1.2 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 $\mathbf{O(n^2)}$, where **n** is a number of items.

Bubble short is majorly used where -

- complexity does not matter
- simple and shortcode is preferred

Algorithm

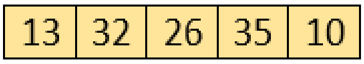
In the algorithm given below, suppose **arr** is an array of **n** elements. The assumed **swap** function in the algorithm will swap the values of given array elements.

```
begin BubbleSort(arr)
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
  return arr
end BubbleSort
```

1.2.1 Working of Bubble sort Algorithm

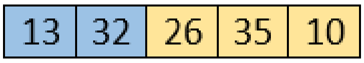
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is $\mathbf{O(n^2)}$.

Let the elements of array are -

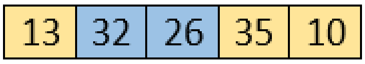


First Pass

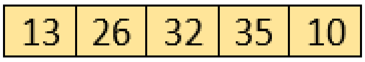
Sorting will start from the initial two elements. Let compare them to check which is greater.



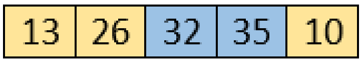
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



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



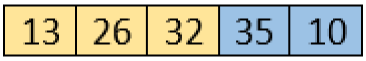
Now, compare 32 and 35.



13	26	32	35	10

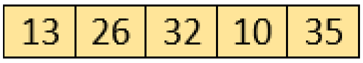
Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.



13	26	32	35	10

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

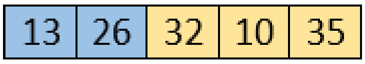


13	26	32	10	35

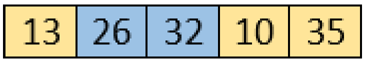
Now, move to the second iteration.

Second Pass

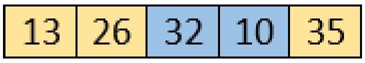
The same process will be followed for second iteration.



13	26	32	10	35

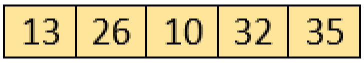


13	26	32	10	35

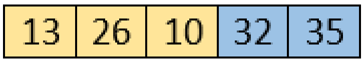


13	26	32	10	35

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



13	26	10	32	35

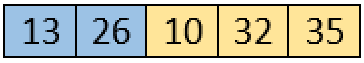


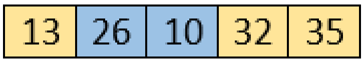
13	26	10	32	35

Now, move to the third iteration.

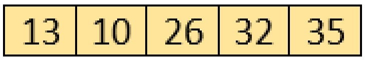
Third Pass

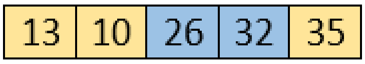
The same process will be followed for third iteration.

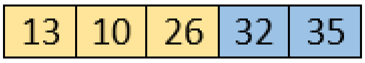




Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



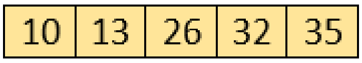




Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -



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

1.2.2 Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

Time Complexity

Case	**Time Complexity**
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is $\mathbf{O(n^2)}$**.**

- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is $\mathbf{O(n^2)}$**.**

Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.
- The space complexity of optimized bubble sort is O(2). It is because two extra variables are required in optimized bubble sort.

1.2.3 Optimized Bubble sort Algorithm

In the bubble sort algorithm, comparisons are made even when the array is already sorted. Because of that, the execution time increases.

To solve it, we can use an extra variable ***swapped.*** It is set to **true** if swapping requires; otherwise, it is set to **false.**

It will be helpful, as suppose after an iteration, if there is no swapping required, the value of variable **swapped** will be **false.** It means that the elements are already sorted, and no further iterations are required.

This method will reduce the execution time and also optimizes the bubble sort.

Algorithm for optimized bubble sort

```
bubbleSort(array)
n = length(array)
repeat
  swapped = false
  for i = 1 to n - 1
         if array[i - 1] > array[i], then
         swap(array[i - 1], array[i])
         swapped = true
         end if
   end for
   n = n - 1
 until not swapped
end bubbleSort
```

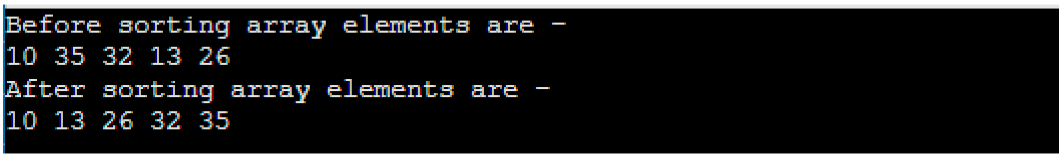
1.2.4 Implementation of Bubble sort

Now, let's see the programs of Bubble sort in different programming languages.

Program: Write a program to implement bubble sort in C language.

```
#include<stdio.h>
 void print(int a[], int n) //function to print array elements
    {
    int i;
    for(i = 0; i < n; i++)
    {
        printf("%d ",a[i]);
    }
    }
 void bubble(int a[], int n) // function to implement bubble sort
 {
   int i, j, temp;
   for(i = 0; i < n; i++)
    {
      for(j = i+1; j < n; j++)
        {
            if(a[j] < a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
 }
void main ()
{
    int i, j,temp;
    int a[5] = { 10, 35, 32, 13, 26};
    int n = sizeof(a)/sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    print(a, n);
    bubble(a, n);
    printf("\nAfter sorting array elements are - \n");
    print(a, n);
}
```

Output



```
Before sorting array elements are -
10 35 32 13 26
After sorting array elements are -
10 13 26 32 35
```

1.3 Insertion sort

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $\mathbf{O(n^2)}$, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Insertion sort has various advantages such as -

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

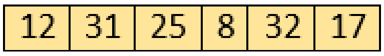
Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

1.3.1 Working of Insertion sort Algorithm

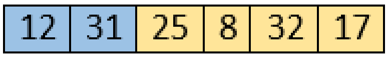
let's take an unsorted array. It will be easier to understand the insertion sort via an example.

Let the elements of array are -



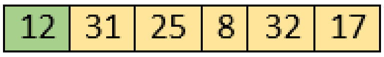
12	31	25	8	32	17

Initially, the first two elements are compared in insertion sort.



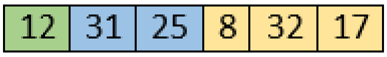
12	31	25	8	32	17

Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

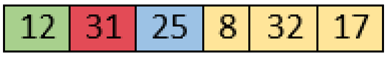


12	31	25	8	32	17

Now, move to the next two elements and compare them.



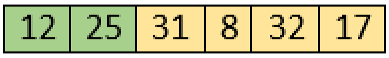
12	31	25	8	32	17



12	31	25	8	32	17

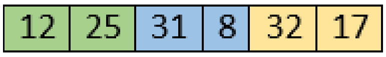
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

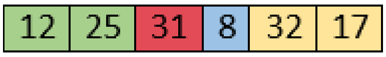


12	25	31	8	32	17

Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

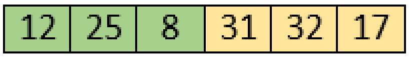


12	25	31	8	32	17



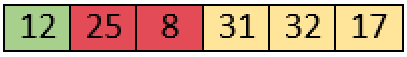
12	25	31	8	32	17

Both 31 and 8 are not sorted. So, swap them.



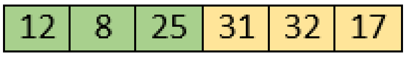
12	25	8	31	32	17

After swapping, elements 25 and 8 are unsorted.



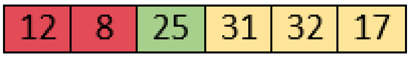
12	25	8	31	32	17

So, swap them.



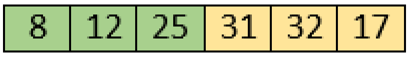
12	8	25	31	32	17

Now, elements 12 and 8 are unsorted.



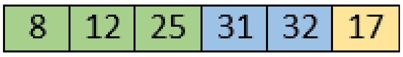
12	8	25	31	32	17

So, swap them too.



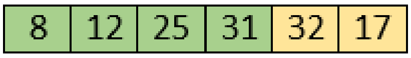
8	12	25	31	32	17

Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



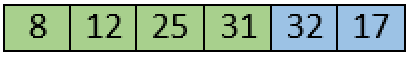
8	12	25	31	32	17

Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.



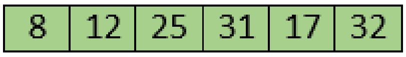
8	12	25	31	32	17

Move to the next elements that are 32 and 17.

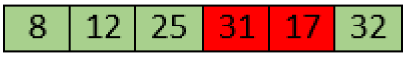


8	12	25	31	32	17

17 is smaller than 32. So, swap them.

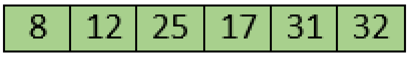


8	12	25	31	17	32

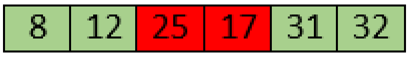


8	12	25	31	17	32

Swapping makes 31 and 17 unsorted. So, swap them too.

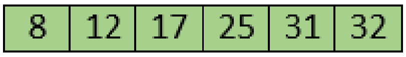


8	12	25	17	31	32



8	12	25	17	31	32

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



8	12	17	25	31	32

Now, the array is completely sorted.

1.3.2 Insertion sort complexity

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

Time Complexity

Case	**Time Complexity**
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is $\mathbf{O(n)}$.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is $\mathbf{O(n^2)}$.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is $\mathbf{O(n^2)}$.

Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

1.3.3 Implementation of insertion sort

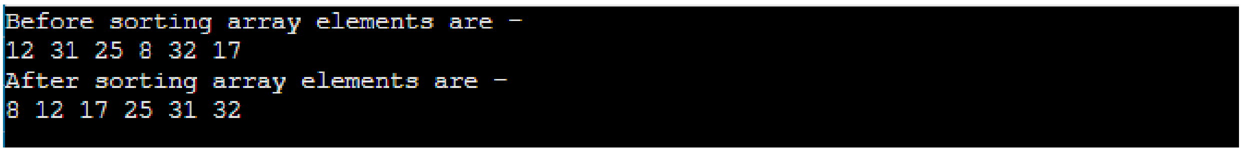
Now, let's see the programs of insertion sort in different programming languages.

Program: Write a program to implement insertion sort in C language.

```
#include <stdio.h>

void insert(int a[], int n) /* function to sort an aay with insertion sort */
{
    int i, j, temp;
    for (i = 1; i < n; i++) {
        temp = a[i];
        j = i - 1;

         while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one position ahead from their current position*/
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = temp;
    }
}

void printArr(int a[], int n) /* function to print the array */
{
    int i;
    for (i = 0; i < n; i++)
```

```
        printf("%d ", a[i]);
    }


    int main()
    {
        int a[] = { 12, 31, 25, 8, 32, 17 };
        int n = sizeof(a) / sizeof(a[0]);
        printf("Before sorting array elements are - \n");
        printArr(a, n);
        insert(a, n);
        printf("\nAfter sorting array elements are - \n");
        printArr(a, n);


        return 0;
    }
```

Output:



```
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

1.4 Quick sort

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

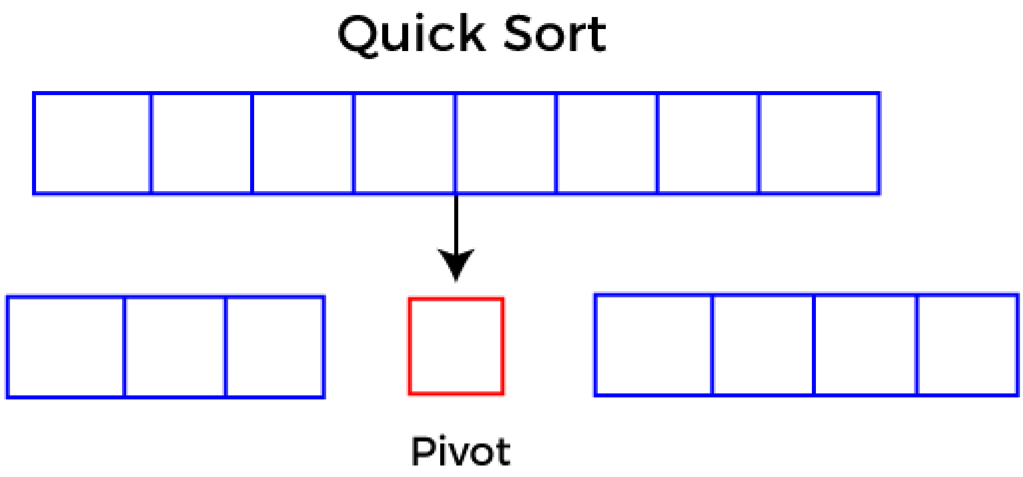
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



1.4.1 Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm:

```
QUICKSORT (array A, start, end)
{
    if (start < end)
    {
        p = partition(A, start, end)
        QUICKSORT (A, start, p - 1)
        QUICKSORT (A, p + 1, end)
    }
}
```

Partition Algorithm:

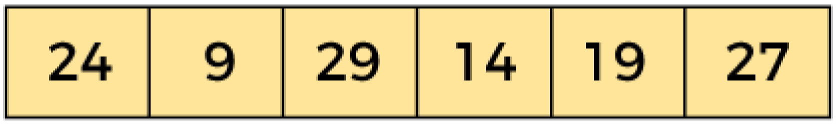
The partition algorithm rearranges the sub-arrays in a place.

```
PARTITION (array A, start, end)
{
 pivot ? A[end]
 i ? start-1
 for j ? start to end -1 {
     do if (A[j] < pivot) {
         then i ? i + 1
         swap A[i] with A[j]
 }}
 swap A[i+1] with A[end]
 return i+1
}
```

1.4.2 Working of Quick Sort Algorithm

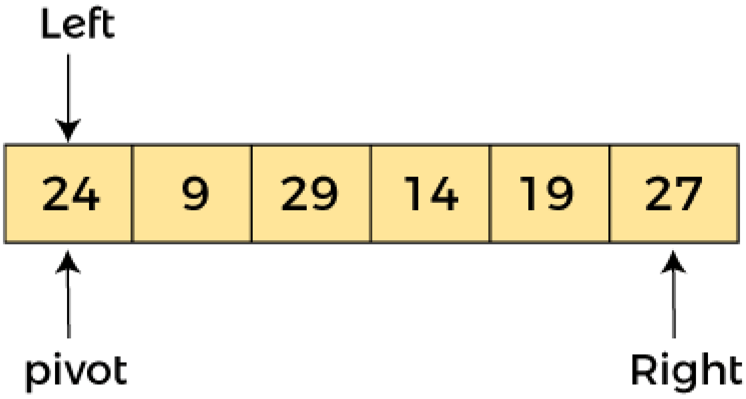
let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

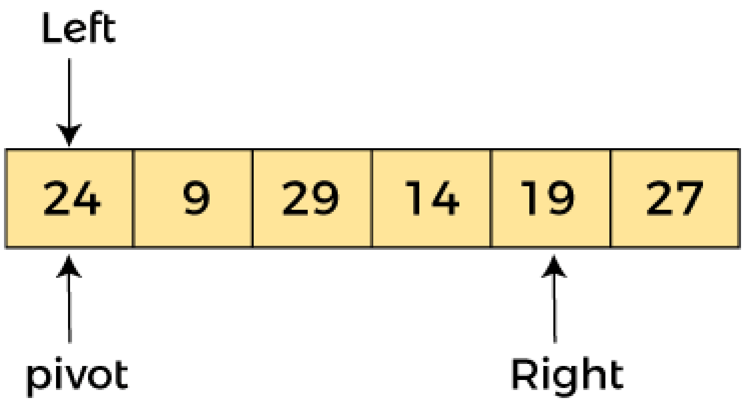


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

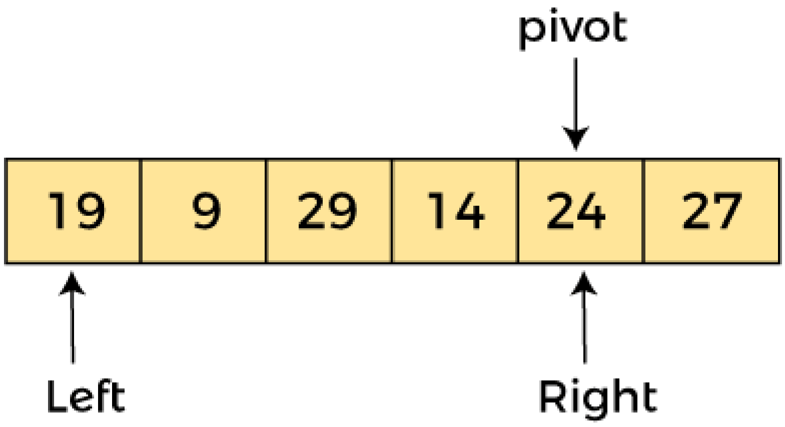


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



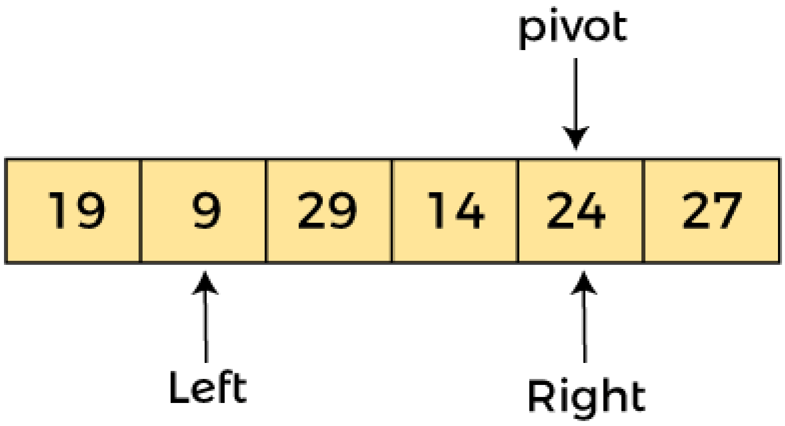
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

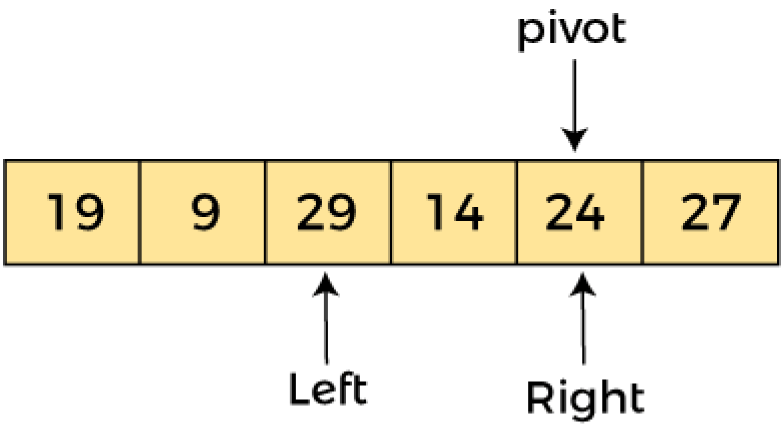


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

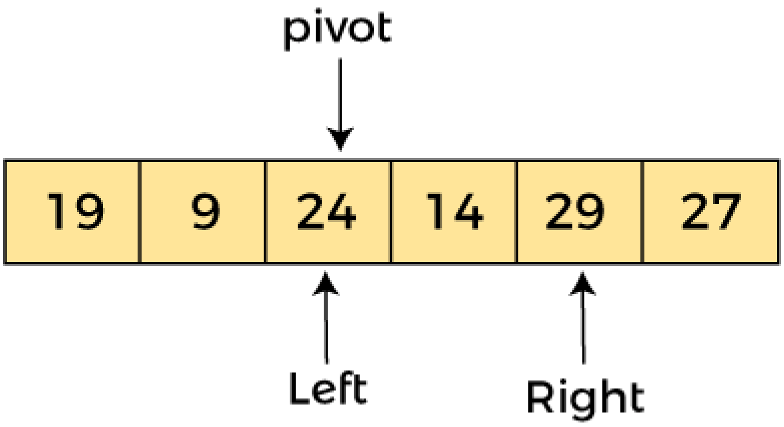
As a[pivot] > a[left], so algorithm moves one position to right as -



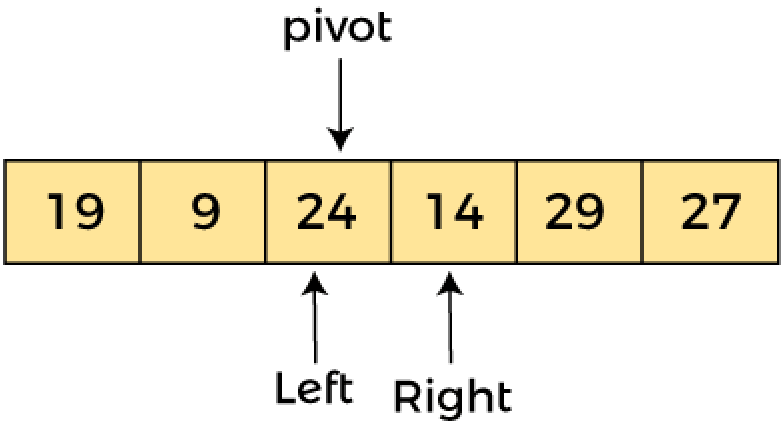
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



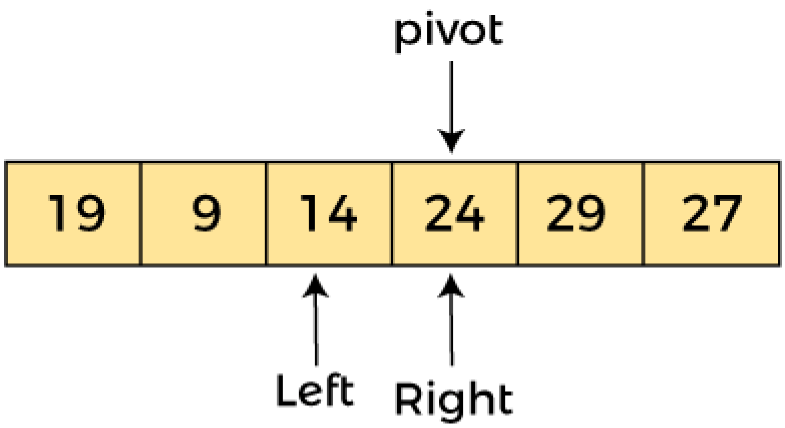
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



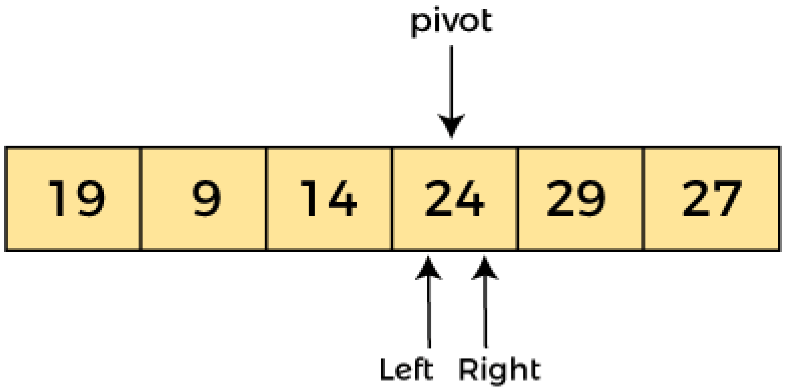
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



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. -



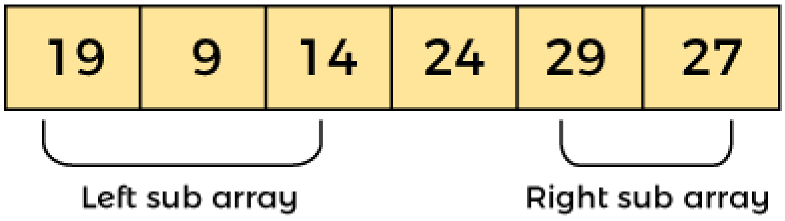
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



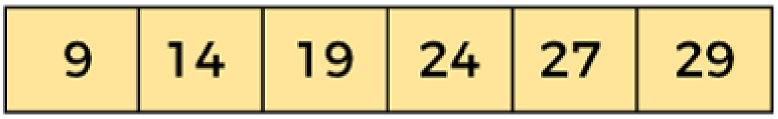
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

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-arrays. After sorting gets done, the array will be -



1.4.3 Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

Space Complexity

Space Complexity	O(n*logn)
Stable	NO

- The space complexity of quicksort is O(n*logn).

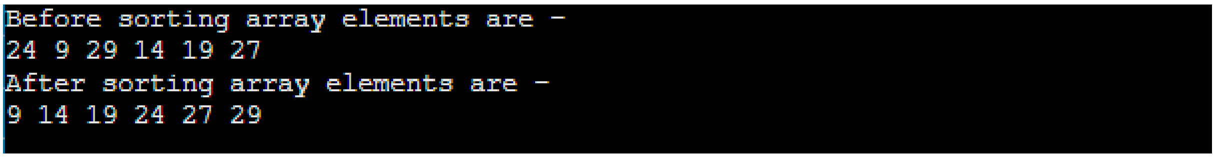
1.4.4 Implementation of quicksort

Now, let's see the programs of quicksort in different programming languages.

Program: Write a program to implement quicksort in C language.

```
#include <stdio.h>
/* function that consider last element as pivot,
place the pivot at its exact position, and place
smaller elements to left of pivot and greater
elements to right of pivot. */
int partition (int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {
        // If current element is smaller than the pivot
        if (a[j] < pivot)
        {
            i++; // increment index of smaller element
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
    return (i + 1);
}

/* function to implement quick sort */
void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end = Ending index */
{
    if (start < end)
    {
        int p = partition(a, start, end); //p is the partitioning index
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}


/* function to print an array */
void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}
int main()
{
    int a[] = { 24, 9, 29, 14, 19, 27 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    quick(a, 0, n - 1);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);

    return 0;
}
```

Output:



```
Before sorting array elements are -
24 9 29 14 19 27
After sorting array elements are -
9 14 19 24 27 29
```

1.5 Merge sort

we will discuss the merge sort Algorithm. Merge sort is the sorting technique that follows the divide and conquer approach. This article will be very helpful and interesting to students as they might face merge sort as a question in their examinations. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

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 algorithm. 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 is merged into the four-element lists, and so on until we get the sorted list.

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

```
MERGE_SORT(arr, beg, end)


if beg < end

set mid = (beg + end)/2

MERGE_SORT(arr, beg, mid)

MERGE_SORT(arr, mid + 1, end)

MERGE (arr, beg, mid, end)

end of if


END MERGE_SORT
```

The important part of the merge sort is the **MERGE** function. This function performs the merging of two sorted sub-arrays that are **A[beg. . . mid]** and **A[mid+1. . . end]**, to build one sorted array **A[beg. . . end]**. So, the inputs of the **MERGE** function are **A[], beg, mid,** and **end**.

The implementation of the MERGE function is given as follows -

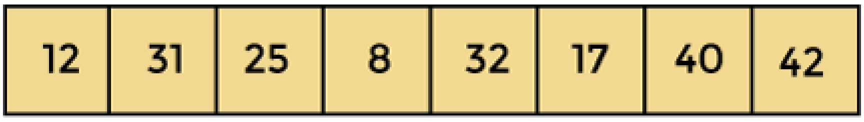
```
/* Function to merge the subarrays of a[] */

void merge(int a[], int beg, int mid, int end)

{

    int i, j, k;

    int n1 = mid - beg + 1;

    int n2 = end - mid;
    int LeftArray[n1], RightArray[n2]; //temporary arrays


    /* copy data to temp arrays */
    for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];


    i = 0, /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */


    while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }
```

```
        while (j<n2)
        {
            a[k] = RightArray[j];
            j++;
            k++;
        }
    }
```

1.5.1 Working of Merge sort Algorithm

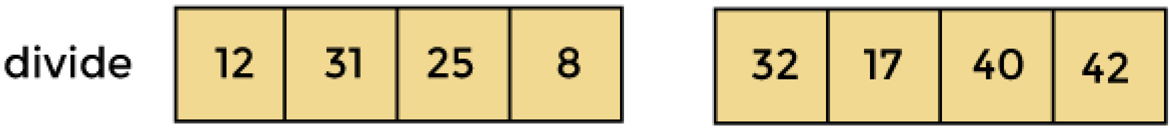
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

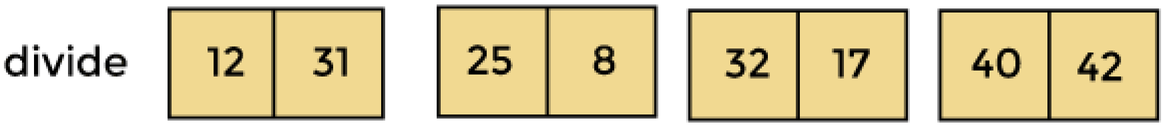


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

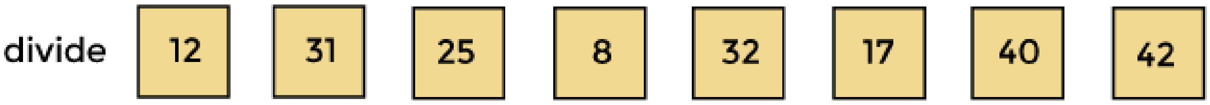
As there are eight elements in the given array, so it is divided into two arrays of size 4.



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



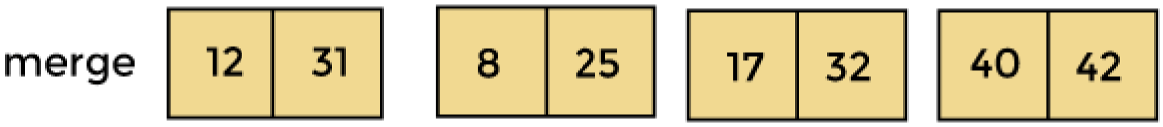
Now, again divide these arrays to get the atomic value that cannot be further divided.



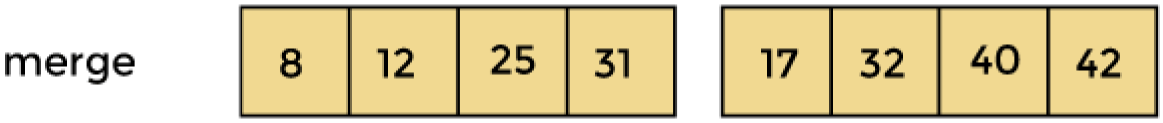
Now, combine them in the same manner they were broken.

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

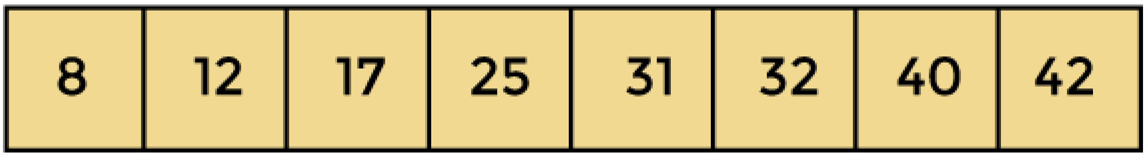
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



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



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

1.5.2 Merge sort complexity

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

Space Complexity

Space Complexity	O(n)
Stable	YES

- The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

1.5.3 Implementation of merge sort

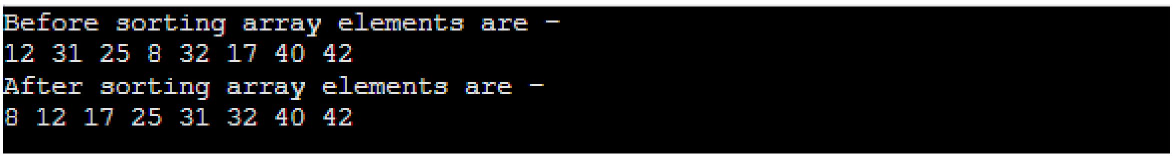
Now, let's see the programs of merge sort in different programming languages.

Program: Write a program to implement merge sort in C language.

```
#include <stdio.h>


/* Function to merge the subarrays of a[] */
void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;

    int LeftArray[n1], RightArray[n2]; //temporary arrays

    /* copy data to temp arrays */
    for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];

    i = 0; /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */

    while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int beg, int end)
```

```
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}

/* Function to print the array */
void printArray(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

int main()
{
    int a[] = { 12, 31, 25, 8, 32, 17, 40, 42 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArray(a, n);
    mergeSort(a, 0, n - 1);
    printf("After sorting array elements are - \n");
    printArray(a, n);
    return 0;
}
```

Output:



```
Before sorting array elements are -
12 31 25 8 32 17 40 42
After sorting array elements are -
8 12 17 25 31 32 40 42
```

1.6 Heap Sort

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

Before knowing more about the heap sort, let's first see a brief description of Heap.

1.6.1 What is a heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

What is heap sort?

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list. Heapsort is the in-place sorting algorithm.

Algorithm

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
    swap arr[1] with arr[i]
        heap_size[arr] = heap_size[arr] ? 1
        MaxHeapify(arr,1)
End
```

BuildMaxHeap(arr)

```
BuildMaxHeap(arr)
   heap_size(arr) = length(arr)
   for i = length(arr)/2 to 1
MaxHeapify(arr,i)
End
```

MaxHeapify(arr,i)

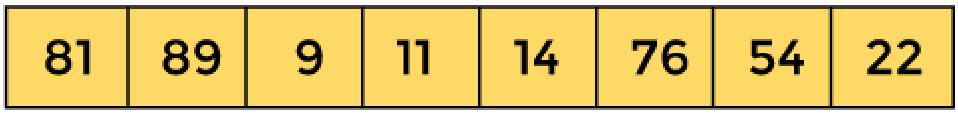
```
MaxHeapify(arr,i)
L = left(i)
R = right(i)
if L ? heap_size[arr] and arr[L] > arr[i]
largest = L
else
largest = i
if R ? heap_size[arr] and arr[R] > arr[largest]
largest = R
if largest != i
swap arr[i] with arr[largest]
MaxHeapify(arr,largest)
End
```

1.6.2 Working of Heap sort Algorithm

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

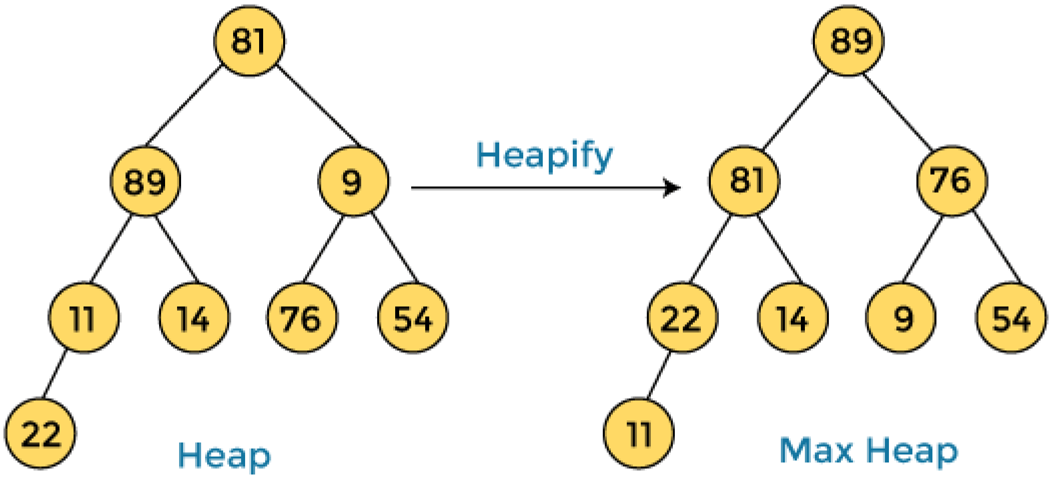
- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

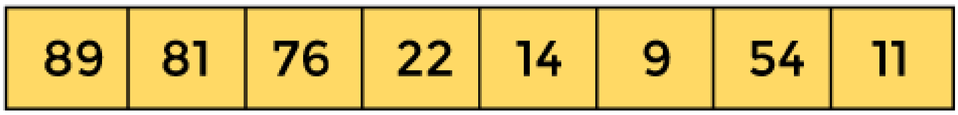


81	89	9	11	14	76	54	22

First, we have to construct a heap from the given array and convert it into max heap.

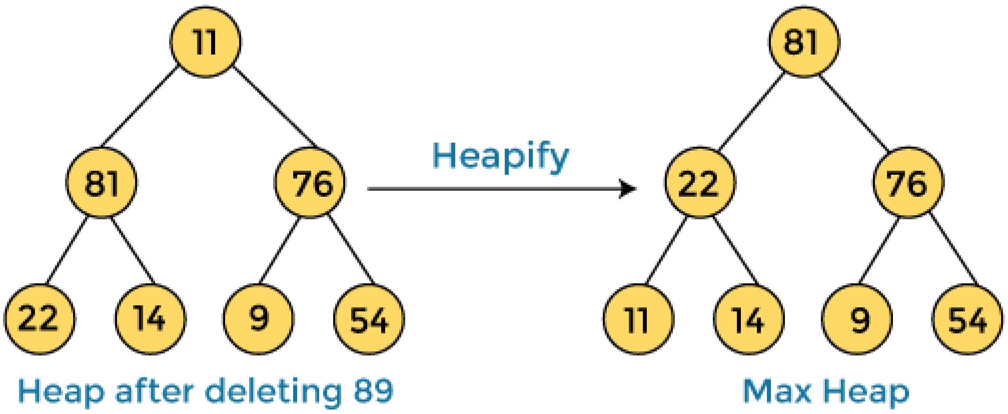


After converting the given heap into max heap, the array elements are -

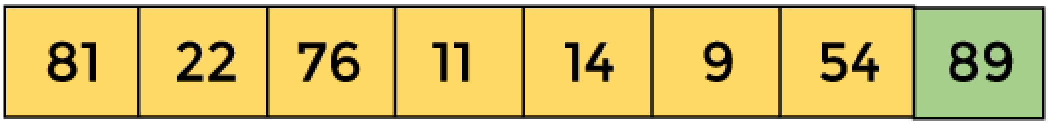


89	81	76	22	14	9	54	11

Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

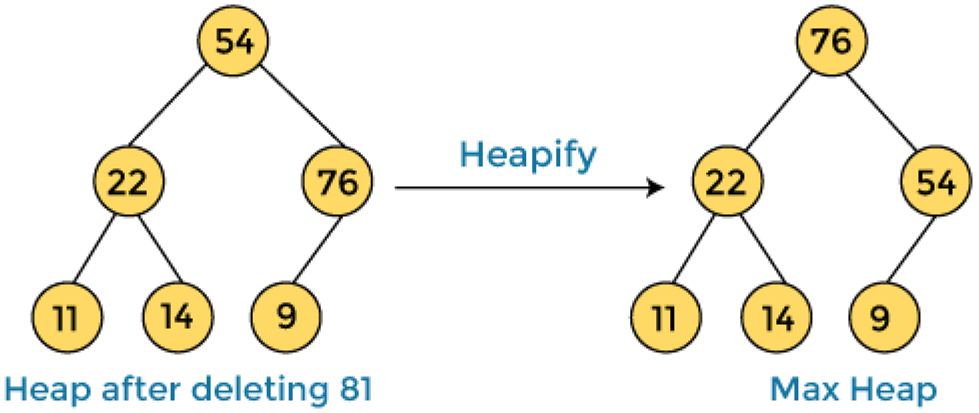


After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

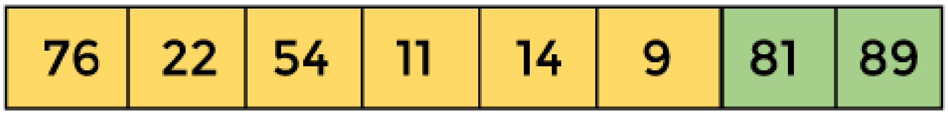


81	22	76	11	14	9	54	89

In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.

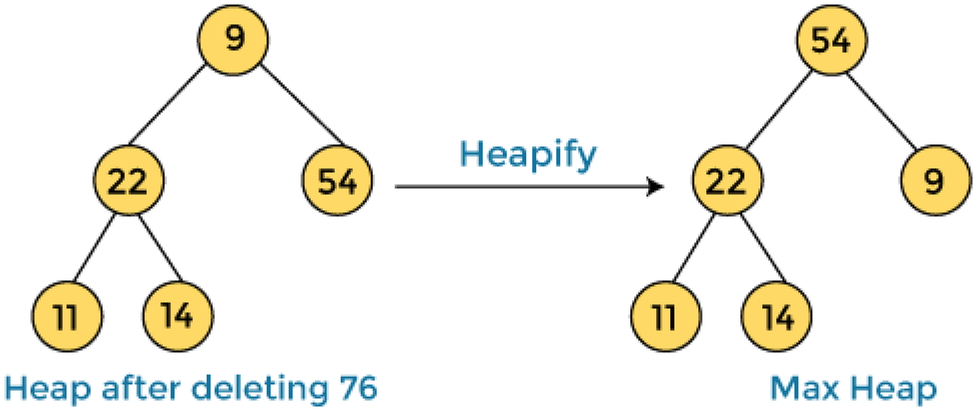


After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

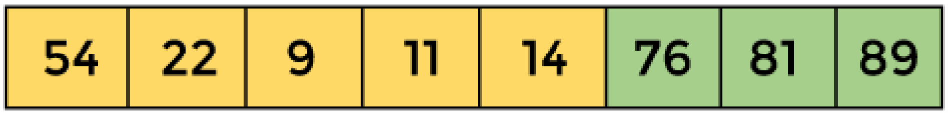


76	22	54	11	14	9	81	89

In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

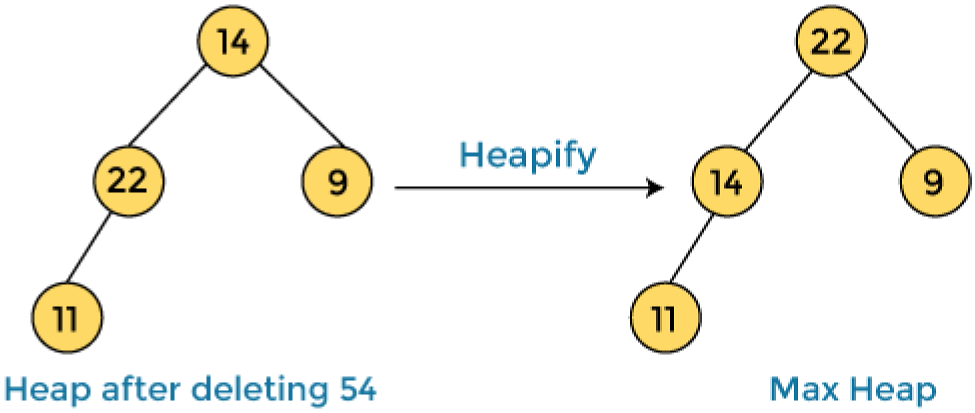


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -



54	22	9	11	14	76	81	89

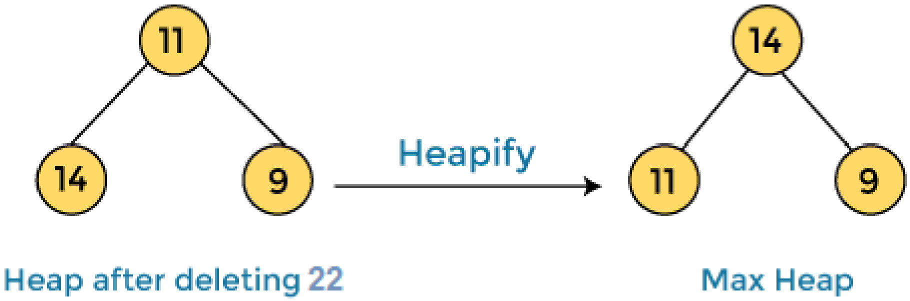
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

22	14	9	11	54	76	81	89

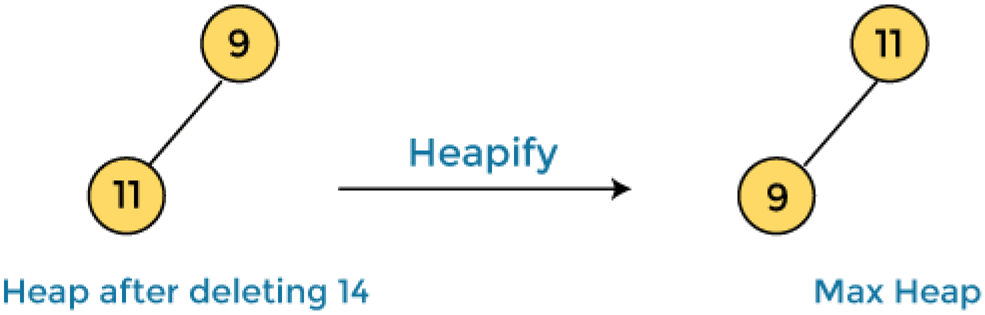
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



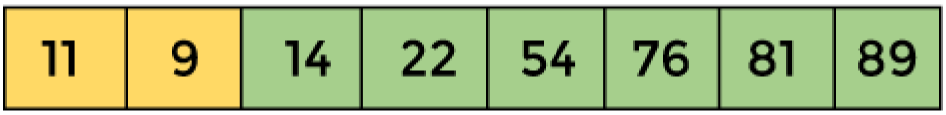
After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

14	11	9	22	54	76	81	89

In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

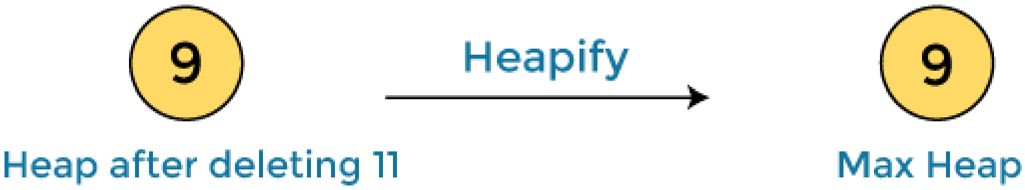


After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

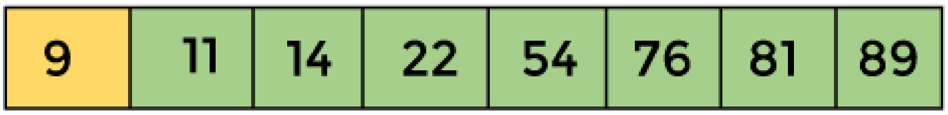


11	9	14	22	54	76	81	89

In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **11** with **9,** the elements of array are -

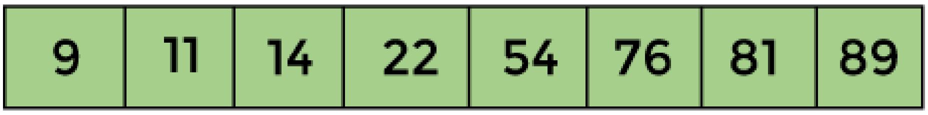


9	11	14	22	54	76	81	89

Now, heap has only one element left. After deleting it, heap will be empty.

9
Remove 9
Empty

After completion of sorting, the array elements are -



9	11	14	22	54	76	81	89

Now, the array is completely sorted.

1.6.3 Heap sort complexity

Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1.6.4 Time Complexity

Case	Time Complexity
Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

1.6.5 Space Complexity

Space Complexity	O(1)
Stable	N0

- The space complexity of Heap sort is O(1).

1.6.6 Implementation of Heapsort

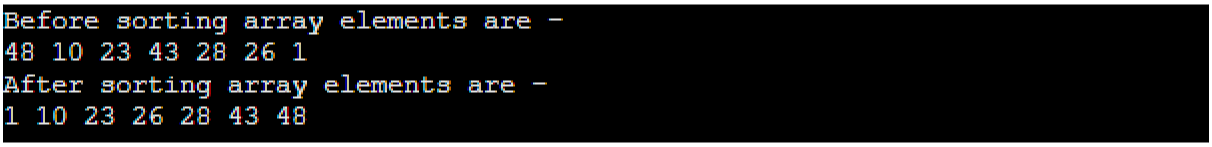
Now, let’s see the programs of Heap sort in different programming languages.

Program: Write a program to implement heap sort in C language.

```
#include <stdio.h>
/* function to heapify a subtree. Here ’i’ is the
index of root node in array a[], and ’n’ is the size of heap. */
void heapify(int a[], int n, int i)
{
   int largest = i; // Initialize largest as root
   int left = 2 * i + 1; // left child
   int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
        // swap a[0] with a[i]
        int temp = a[0];
        a[0] = a[i];
        a[i] = temp;

        heapify(a, i, 0);
    }
}
/* function to print the array elements */
void printArr(int arr[], int n)
```

```
{
    for (int i = 0; i < n; ++i)
    {
        printf("%d", arr[i]);
        printf(" ");
    }


}
int main()
{
    int a[] = {48, 10, 23, 43, 28, 26, 1};
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    heapSort(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

Output



```
Before sorting array elements are -
48 10 23 43 28 26 1
After sorting array elements are -
1 10 23 26 28 43 48
```

2 Hashing:

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping. Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1). Till now, we read the two techniques for searching, i.e., linear search and binary search. The worst time complexity in linear search is O(n), and O(logn) in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time.

For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

Hash(key)= index;

When we pass the key in the hash function, then it gives the index.

Hash(john) = 3;

The above example adds the john at the index 3.

Drawback of Hash function

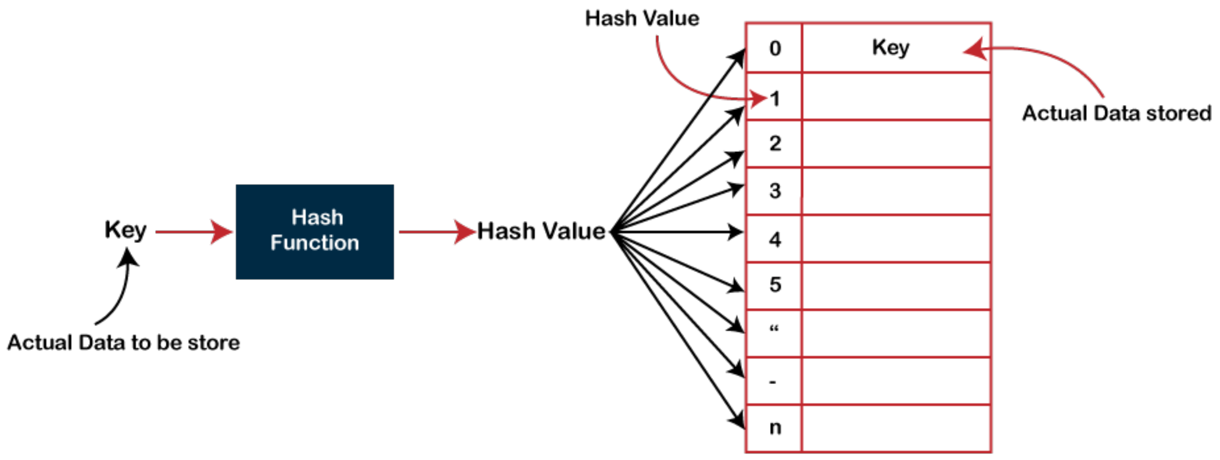
A Hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.

2.1 Hash table Construction and Addressing:

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

Index = hash(key)



There are three ways of calculating the hash function:

- **Division method**
- **Folding method**
- **Mid square method**

In the division method, the hash function can be defined as:

$\mathbf{h(k_i) = k_i \ \% \ m;}$

where **m** is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

$h(6) = 6\%10 = 6$

The index is 6 at which the value is stored.

2.2 Collision

When the two different values have the same value, then the problem occurs between the two values, known as a collision. In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

$h(26) = 26\%10 = 6$

Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem. To resolve these collisions, we have some techniques known as collision techniques.

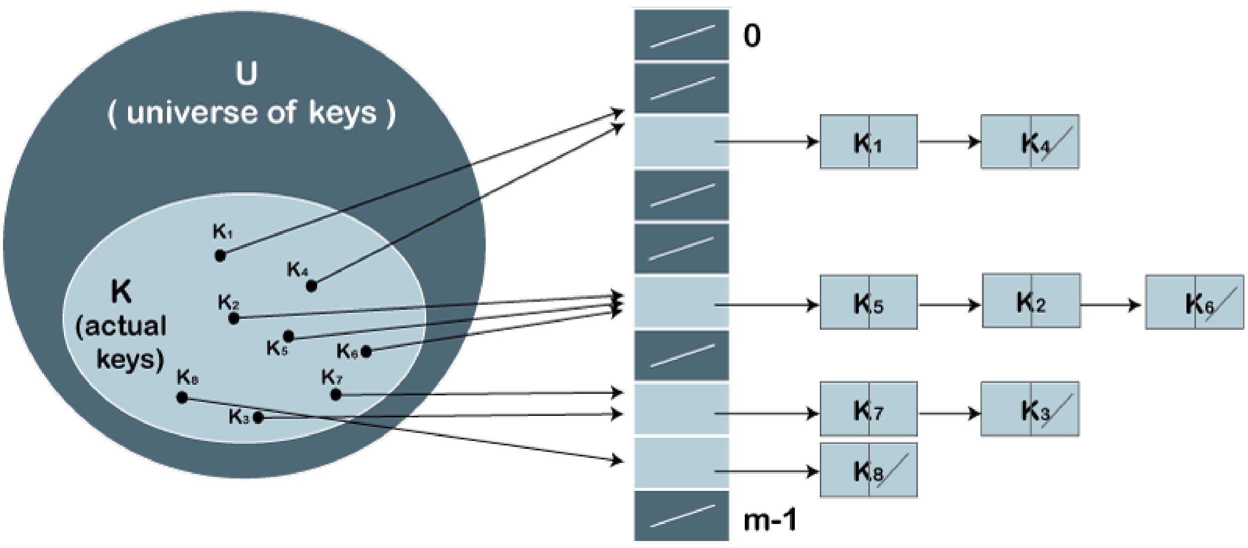
The following are the collision techniques:

- Open Hashing: It is also known as closed addressing.
- Closed Hashing: It is also known as open addressing.

2.3 Open Hashing

In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.

Collision Resolution by Chaining



Let's first understand the chaining to resolve the collision.

Suppose we have a list of key values

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

In this case, we cannot directly use h(k) = k_i/m as h(k) = 2k+3

- The index of key value 3 is:

index = h(3) = (2(3)+3)%10 = 9

The value 3 would be stored at the index 9.

- The index of key value 2 is:

index = h(2) = (2(2)+3)%10 = 7

The value 2 would be stored at the index 7.

- The index of key value 9 is:

index = h(9) = (2(9)+3)%10 = 1

The value 9 would be stored at the index 1.

- The index of key value 6 is:

index = h(6) = (2(6)+3)%10 = 5

The value 6 would be stored at the index 5.

- The index of key value 11 is:

index = h(11) = (2(11)+3)%10 = 5

The value 11 would be stored at the index 5. Now, we have two values (6, 11) stored at the same index, i.e., 5. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 11 to this list. After the creation of the new list, the newly created list will be linked to the list having value 6.

- The index of key value 13 is:

index = h(13) = (2(13)+3)%10 = 9

The value 13 would be stored at index 9. Now, we have two values (3, 13) stored at the same index, i.e., 9. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 13 to this list. After the creation of the new list, the newly created list will be linked to the list having value 3.

- The index of key value 7 is:

index = h(7) = (2(7)+3)%10 = 7

The value 7 would be stored at index 7. Now, we have two values (2, 7) stored at the same index, i.e., 7. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 7 to this list. After the creation of the new list, the newly created list will be linked to the list having value 2.

- The index of key value 12 is:

index = h(12) = (2(12)+3)%10 = 7

According to the above calculation, the value 12 must be stored at index 7, but the value 2 exists at index 7. So, we will create a new list and add 12 to the list. The newly created list will be linked to the list having a value 7.

The calculated index value associated with each key value is shown in the below table:

key	Location(u)
3	**((2*3)+3)%10 = 9**
2	**((2*2)+3)%10 = 7**
9	**((2*9)+3)%10 = 1**
6	**((2*6)+3)%10 = 5**
11	**((2*11)+3)%10 = 5**
13	**((2*13)+3)%10 = 9**
7	**((2*7)+3)%10 = 7**
12	**((2*12)+3)%10 = 7**

2.4 Types of Hash functions

There are many hash functions that use numeric or alphanumeric keys. discussing different hash functions:

1. **Division Method.**
2. **Mid Square Method.**
3. **Folding Method.**
4. **Multiplication Method.**

1. Division Method:

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:

h(K) = k mod M

Here,

***k** is the key value, and*

***M** is the size of the hash table.*

It is best suited that **M** is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

Example:

$k = 12345$

$M = 95$

$h(12345) = 12345 \bmod 95$

$= 90$

$k = 1276$

$M = 11$

$h(1276) = 1276 \bmod 11$

$= 0$

Pros:

1. This method is quite good for any value of M.
2. The division method is very fast since it requires only a single division operation.

Cons:

1. This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
2. Sometimes extra care should be taken to choose the value of M.

2. Mid Square Method:

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

1. Square the value of the key k i.e. k^2
2. Extract the middle **r** digits as the hash value.

Formula:

$\boldsymbol{h(K) = h(k\ x\ k)}$

Here,

k *is the key value.*

The value of **r** can be decided based on the size of the table.

Example:

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

$k = 60$

$k\ x\ k = 60\ x\ 60$

$\quad = 3600$

$h(60) = 60$

The hash value obtained is 60

Pros:

1. The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
2. The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

Cons:

1. The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.
2. Another disadvantage is that there will be collisions but we can try to reduce collisions.

3. Digit Folding Method:

This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,....,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.
2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula:

$\boldsymbol{k = k1,\ k2,\ k3,\ k4,\ .....,\ kn}$

$\boldsymbol{s = k1 + k2 + k3 + k4 + ... + kn}$

$\boldsymbol{h(K) = s}$

Here,

$\boldsymbol{s}$ *is obtained by adding the parts of the key* $\boldsymbol{k}$

Example:

$k = 12345$

$k1 = 12,\ k2 = 34,\ k3 = 5$

$s = k1 + k2 + k3$

$\quad = 12 + 34 + 5$

$\quad = 51$

$h(K) = 51$

Note:

The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have a lesser number of digits.

4. Multiplication Method

This method involves the following steps:

1. Choose a constant value A such that $0 < A < 1$.
2. Multiply the key value with A.
3. Extract the fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M.
5. The resulting hash value is obtained by taking the floor of the result obtained in step 4.

Formula:

$$h(K) = \text{floor}\ (M\ (kA \bmod 1))$$

Here,

M *is the size of the hash table.*

k *is the key value.*

A *is a constant value.*

Example:

$k = 12345$

$A = 0.357840$

$M = 100$

$$\begin{aligned} h(12345) &= \text{floor}[\ 100\ (12345*0.357840 \bmod 1)] \\ &= \text{floor}[\ 100\ (4417.5348 \bmod 1)\ ] \\ &= \text{floor}[\ 100\ (0.5348)\ ] \\ &= \text{floor}[\ 53.48\ ] \\ &= 53 \end{aligned}$$

Pros:

The advantage of the multiplication method is that it can work with any value between 0 and 1, although there are some values that tend to give better results than the rest.

Cons:

The multiplication method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast.

2.4.1 Commonly used hash functions:

Hash functions are widely used in computer science and cryptography for a variety of purposes, including data integrity, digital signatures, password storage, and more.

There are many types of hash functions, each with its own strengths and weaknesses. Here are a few of the most common types:

1. **SHA (Secure Hash Algorithm)**: SHA is a family of cryptographic hash functions designed by the National Security Agency (NSA) in the United States. The most widely used SHA algorithms are SHA-1, SHA-2, and SHA-3. Here's a brief overview of each:

- **SHA-1**: SHA-1 is a 160-bit hash function that was widely used for digital signatures and other applications. However, it is no longer considered secure due to known vulnerabilities.
- **SHA-2**: SHA-2 is a family of hash functions that includes SHA-224, SHA-256, SHA-384, and SHA-512. These functions produce hash values of 224, 256, 384, and 512 bits, respectively. SHA-2 is widely used in security protocols such as SSL/TLS and is considered secure.
- **SHA-3**: SHA-3 is the latest member of the SHA family and was selected as the winner of the NIST hash function competition in 2012. It is designed to be faster and more secure than SHA-2 and produces hash values of 224, 256, 384, and 512 bits.

2. **CRC (Cyclic Redundancy Check)**: CRC is a non-cryptographic hash function used primarily for error detection in data transmission. It is fast and efficient but is not suitable for security purposes. The basic idea behind CRC is to append a fixed-length check value, or checksum, to the end of a message. This checksum is calculated based on the contents of the message using a mathematical algorithm, and is then transmitted along with the message.

When the message is received, the receiver can recalculate the checksum using the same algorithm, and compare it with the checksum transmitted with the message. If the two checksums match, the receiver can be reasonably certain that the message was not corrupted during transmission.

The specific algorithm used for CRC depends on the application and the desired level of error detection. Some common CRC algorithms include CRC-16, CRC-32, and CRC-CCITT.

3. **MurmurHash**: MurmurHash is a fast and efficient non-cryptographic hash function designed for use in hash tables and other data structures. It is not suitable for security purposes as it is vulnerable to collision attacks.

4. **BLAKE2**: BLAKE2 is a cryptographic hash function designed to be fast and secure. It is an improvement over the popular SHA-3 algorithm and is widely used in applications that require high-speed hashing, such as cryptocurrency mining.

BLAKE2 is available in two versions: BLAKE2b and BLAKE2s. BLAKE2b is optimized for 64-bit platforms and produces hash values of up to 512 bits, while BLAKE2s is optimized for 8- to 32-bit platforms and produces hash values of up to 256 bits.

5. **Argon2**: Argon2 is a memory-hard password hashing function designed to be resistant to brute-force attacks. It is widely used for password storage and is recommended by the Password Hashing Competition. The main goal of Argon2 is to make it difficult for attackers to crack passwords by using techniques such as brute force attacks or dictionary attacks. It achieves this by using a computationally-intensive algorithm that makes it difficult for attackers to perform large numbers of password guesses in a short amount of time.

Argon2 has several key features that make it a strong choice for password hashing and key derivation:

- Resistance to parallel attacks: Argon2 is designed to be resistant to parallel attacks, meaning that it is difficult for attackers to use multiple processing units, such as GPUs or ASICs, to speed up password cracking.
- Memory-hardness: Argon2 is designed to be memory-hard, meaning that it requires a large amount of memory to compute the hash function. This makes it more difficult for attackers to use specialized hardware to crack passwords.
- Customizable: Argon2 is highly customizable, and allows users to adjust parameters such as the memory usage, the number of iterations, and the output length to meet their specific security requirements.

Resistance to side-channel attacks: Argon2 is designed to be resistant to side-channel attacks, such as timing attacks or power analysis attacks, that could be used to extract information about the password being hashed.

6. MD5 (Message Digest 5): MD5 is a widely-used cryptographic hash function that produces a 128-bit hash value. It is fast and efficient but is no longer recommended for security purposes due to known vulnerabilities. The basic idea behind MD5 is to take an input message of any length, and produce a fixed-length output, known as the hash value or message digest. This hash value is unique to the input message, and is generated using a mathematical algorithm that involves a series of logical operations, such as bitwise operations, modular arithmetic, and logical functions.

MD5 is widely used in a variety of applications, including digital signatures, password storage, and data integrity checks. However, it has been shown to have weaknesses that make it vulnerable to attacks. In particular, it is possible to generate two different messages with the same MD5 hash value, a vulnerability known as a collision attack.

There are many other types of hash functions, each with its own unique features and applications. The choice of hash function depends on the specific requirements of the application, such as speed, security, and memory usage.