

INTRODUCTION TO SORTING

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that $A[0] < A[1] < A[2] < \dots < A[N]$

For example, if we have an array that is declared and initialized as

```
int A [] = {21, 34, 11, 9, 1, 0, 22};
```

Then the sorted array (ascending order) can be given as:

```
A [] = {0, 1, 9, 11, 21, 22, 34};
```

A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order.

Following are some of the examples of sorting in real-life scenarios --

- ❖ **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- ❖ **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

For example, suppose we have a record of employee. It has following data:

Employee No.
Employee Name
Employee Salary
Department Name

Here, employee no. can be taken as key for sorting the records in ascending or descending order. Now, we have to search an Employee with employee no. 116, so we don't require to search the complete record, simply we can search between the Employees with employee no. 100 to 120.

Sorting Techniques:

The two main criteria to judge which algorithm is better than the other have been Sorting technique depends on the situation. It depends on two parameters

1. Execution time of program that means time taken for execution of program
2. Space that means space taken by the program

Sorting techniques are differentiated by their efficiency and space requirements.

There are two types of sorting:

- ❖ **Internal sorting** which deals with sorting the data stored in the computer's memory.
- ❖ **External sorting** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory.

Practical Considerations for Internal Sorting:

When analyzing the performance of different sorting algorithms, the practical considerations would be the following:

- Number of sort key comparisons that will be performed
- Number of times the records in the list will be moved
- Best case performance
- Worst case performance
- Average case performance
- Stability of the sorting algorithm where stability means that equivalent elements or records retain their relative positions even after sorting is done

Different Sorting Algorithms:

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covered:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Shell Sort
5. Quick Sort
6. Merge Sort
7. Heap Sort

1. BUBBLE SORT:

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list.

Note: The end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Technique:

The basic methodology of the working of bubble sort is given as follows:

- (a) In Pass 1, A [0] and A [1] are compared, then A [1] is compared with A [2], A [2] is compared with A [3], and so on. Finally, A[N-2] is compared with A[N-1]. Pass 1 involves n-1 comparison and places the biggest element at the highest index of the array.
- (b) In Pass 2, A [0] and A [1] are compared, then A [1] is compared with A [2], A[2] is compared with A[3], and so on. Finally, A[N-3] is compared with A[N-2]. Pass 2 involves n-2 comparisons and places the second biggest element at the second highest index of the array.
- (c) In Pass 3, A [0] and A [1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N-4] is compared with A[N-3]. Pass 3 involves n-3 comparisons and places the third biggest element at the third highest index of the array.
- (d) In Pass n-1, A[0] and A[1] are compared so that $A[0] < A[1]$. After this step, all the elements of the array are arranged in ascending order.

Example: let us consider an array A[] that has the following elements:

A [] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

- (a) Compare 30 and 52. Since $30 < 52$, no swapping is done.
- (b) Compare 52 and 29. Since $52 > 29$, swapping is done.
30, **29**, **52**, 87, 63, 27, 19, 54
- (c) Compare 52 and 87. Since $52 < 87$, no swapping is done.
- (d) Compare 87 and 63. Since $87 > 63$, swapping is done.
30, 29, 52, **63**, **87**, 27, 19, 54
- (e) Compare 87 and 27. Since $87 > 27$, swapping is done.
30, 29, 52, 63, **27**, **87**, 19, 54
- (f) Compare 87 and 19. Since $87 > 19$, swapping is done.
30, 29, 52, 63, 27, **19**, **87**, 54
- (g) Compare 87 and 54. Since $87 > 54$, swapping is done.
30, 29, 52, 63, 27, 19, **54**, **87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:

- (a) Compare 30 and 29. Since $30 > 29$, swapping is done.
29, 30, 52, 63, 27, 19, 54, 87
- (b) Compare 30 and 52. Since $30 < 52$, no swapping is done.
- (c) Compare 52 and 63. Since $52 < 63$, no swapping is done.
- (d) Compare 63 and 27. Since $63 > 27$, swapping is done.
29, 30, 52, 27, 63, 19, 54, 87
- (e) Compare 63 and 19. Since $63 > 19$, swapping is done.

29, 30, 52, 27, 19, 63, 54, 87

- (f) Compare 63 and 54. Since $63 > 54$, swapping is done.
29, 30, 52, 27, 19, 54, 63, 87

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Pass 3:

- (a) Compare 29 and 30. Since $29 < 30$, no swapping is done.
- (b) Compare 30 and 52. Since $30 < 52$, no swapping is done.
- (c) Compare 52 and 27. Since $52 > 27$, swapping is done.
29, 30, 27, 52, 19, 54, 63, 87
- (d) Compare 52 and 19. Since $52 > 19$, swapping is done.
29, 30, 27, 19, 52, 54, 63, 87
- (e) Compare 52 and 54. Since $52 < 54$, no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

Pass 4:

- (a) Compare 29 and 30. Since $29 < 30$, no swapping is done.
- (b) Compare 30 and 27. Since $30 > 27$, swapping is done.
29, 27, 30, 19, 52, 54, 63, 87
- (c) Compare 30 and 19. Since $30 > 19$, swapping is done.
29, 27, 19, 30, 52, 54, 63, 87
- (d) Compare 30 and 52. Since $30 < 52$, no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:

- (a) Compare 29 and 27. Since $29 > 27$, swapping is done.
27, 29, 19, 30, 52, 54, 63, 87
- (b) Compare 29 and 19. Since $29 > 19$, swapping is done.
27, 19, 29, 30, 52, 54, 63, 87
- (c) Compare 29 and 30. Since $29 < 30$, no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

Pass 6:

- (a) Compare 27 and 19. Since $27 > 19$, swapping is done.
19, 27, 29, 30, 52, 54, 63, 87
- (b) Compare 27 and 29. Since $27 < 29$, no swapping is done.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

Pass 7:

- (a) Compare 19 and 27. Since $19 < 27$, no swapping is done.

Observe that the entire list is sorted now.

Algorithm for bubble sort:**BUBBLE_SORT(A, N)**

Step 1: Repeat Step 2 For $i = 0$ to $N-1$

Step 2: Repeat For $j = 0$ to $N - i$

Step 3: IF $A[j] > A[j + 1]$
SWAP $A[j]$ and $A[j+1]$

[END OF INNER LOOP]

[END OF OUTER LOOP]

Step 4: EXIT

Complexity of Bubble Sort:

Worst Case Time Complexity [Big-O] $O(n^2)$

Best Case Time Complexity [Big-omega] $\Omega(n)$, it is when the list is already sorted

Average Time Complexity [Big-theta] $\Theta(n^2)$

Programming Example

Q1. Write a program to enter n numbers in an array. Display the array with elements being sorted in ascending order.

Source Code:

```
#include <stdio.h>
#include <conio.h>
int main()
{
    int i, n, temp, j, arr[10];
    clrscr();
    printf("\n Enter the number of elements in the array : ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for(i=0; i<n; i++)
    {
        scanf("%d", &arr[i]);
    }
    for(i=0; i<n; i++)
    {
        for(j=0; j<n-i-1; j++)
        {
            if(arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    printf("\n The array sorted in ascending order is :\n");
    for(i=0; i<n; i++)
        printf("%d\t", arr[i]);
    getch();
    return 0;
}
```

Output:

```
Enter the number of elements in the array : 10
Enter the elements : 8   9   6   7   5   4   2   3   1   10
The array sorted in ascending order is :
1   2   3   4   5   6   7   8   9   10
```

2. INSERTION SORT:

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

The main idea behind insertion sort is that it inserts each item into its proper place in the final list.

Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

Technique: Insertion sort works as follows:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1.
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Example: Consider an array of integers given below. We will sort the values in the array using insertion sort

39	9	45	63	18	81	108	54	72	36
----	---	----	----	----	----	-----	----	----	----

Solution:

39	9	45	63	18	81	108	54	72	36
----	---	----	----	----	----	-----	----	----	----

39	9	45	63	18	81	108	54	72	36
----	---	----	----	----	----	-----	----	----	----

4[0] is the only element in sorted list

9	39	45	63	18	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 1)

9	39	45	63	18	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 2)

9	39	45	63	18	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 3)

9	18	39	45	63	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 4)

9	18	39	45	63	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 5)

9	18	39	45	63	81	108	54	72	36
---	----	----	----	----	----	-----	----	----	----

(Pass 6)

9	18	39	45	54	63	81	108	72	36
---	----	----	----	----	----	----	-----	----	----

(Pass 7)

9	18	39	45	54	63	72	81	108	36
---	----	----	----	----	----	----	----	-----	----

(Pass 8)

9	18	36	39	45	54	63	72	81	108
---	----	----	----	----	----	----	----	----	-----

(Pass 9)

☐ Sorted ☐ Unsorted

Initially, A [0] is the only element in the sorted set. In Pass 1, A [1] will be placed either before or after A [0], so that the array A is sorted. In Pass 2, A [2] will be placed either before A [0], in between A [0] and A [1], or after A [1]. In Pass 3, A [3] will be placed in its proper place. In Pass N-1, A[N-1] will be placed in its proper place to keep the array sorted.

Algorithm for insertion sort:

INSERTION-SORT (ARR, N)

```

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:   SET TEMP = ARR[K]
Step 3:   SET J = K - 1
Step 4:   Repeat while TEMP <= ARR[J]
           SET ARR[J + 1] = ARR[J]
           SET J = J - 1
           [END OF INNER LOOP]
Step 5:   SET ARR[J + 1] = TEMP
           [END OF LOOP]
Step 6: EXIT

```

Complexity of the Insertion Sort Algorithm:

Worst Case Time Complexity [Big-O]: $O(n^2)$.

Best Case Time Complexity [Big-omega]: $\Omega(n)$.

Average Time Complexity [Big-theta]: $\Theta(n^2)$.

Advantages:

- ❖ The advantages of this sorting algorithm are as follows:
- ❖ It is easy to implement and efficient to use on small sets of data.
- ❖ It can be efficiently implemented on data sets that are already substantially sorted.
- ❖ It performs better than algorithms like selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort.
- ❖ it requires less memory space.

Programming Example

Q2. Write a program to sort an array using insertion sort algorithm.

Source Code:

```

#include <stdio.h>
#include <conio.h>
#define size 5
void insertion_sort(int arr[], int n);
void main()
{
    int arr[size], i, n;
    printf("\n Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("\n Enter the elements of the array: ");
    for(i=0; i<n; i++)
    {
        scanf("%d", &arr[i]);
    }
    insertion_sort(arr, n);
    printf("\n The sorted array is: \n");
    for(i=0; i<n; i++)
    printf(" %d\t", arr[i]);
    getch();
}
void insertion_sort(int arr[], int n)
{
    int i, j, temp;
    for(i=1; i<n; i++)
    {
        temp = arr[i];
        j = i-1;
        while((temp < arr[j]) && (j>=0))
        {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = temp;
    }
}

```

Output

```

Enter the number of elements in the array : 5
Enter the elements of the array : 500 1 50 23 76
The sorted array is :
1    23    20    76    500    6    7    8    9    10

```

3. SELECTION SORT:

Selection sort is conceptually the simplest sorting algorithm. Selection sort is a simple sorting algorithm which finds the smallest element in the array and exchanges it with the element in the first position. Then finds the second smallest element and exchanges it with the element in the second position and continues until the entire array is sorted.

It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place

Technique: Consider an array ARR with N elements. Selection sort works as follows

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position POS of the smallest value in the array and then swap $ARR[POS]$ and $ARR[0]$. Thus, $ARR[0]$ is sorted.
- In Pass 2, find the position POS of the smallest value in sub-array of $N-1$ elements. Swap $ARR[POS]$ with $ARR[1]$. Now, $ARR[0]$ and $ARR[1]$ is sorted
- In Pass $N-1$, find the position POS of the smaller of the elements $ARR[N-2]$ and $ARR[N-1]$. Swap $ARR[POS]$ and $ARR[N-2]$ so that $ARR[0]$, $ARR[1]$, ..., $ARR[N-1]$ is sorted.

Example: Sort the array given below using selection sort.

		39	9	81	45	90	27	72	18
PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Algorithm for Selection Sort:

SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = $ARR[K]$
 Step 2: [INITIALIZE] SET POS = K
 Step 3: Repeat for $J = K+1$ to $N-1$
 IF $SMALL > ARR[J]$
 SET SMALL = $ARR[J]$
 SET POS = J
 [END OF IF]
 [END OF LOOP]
 Step 4: RETURN POS

SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for $K = 1$ to $N-1$
 Step 2: CALL **SMALLEST**(ARR, K, N, POS)
 Step 3: SWAP $A[K]$ with $ARR[POS]$
 [END OF LOOP]
 Step 4: EXIT

Advantages of Selection Sort:

- It is simple and easy to implement.
- It can be used for small data sets
- It is 60 per cent more efficient than bubble sort

Complexity of Selection Sort:

Worst Case Time Complexity [Big-O]: $O(n^2)$

Best Case Time Complexity [Big-omega]: $\Omega(n^2)$

Average Time Complexity [Big-theta]: $\Theta(n^2)$

Programming Example:

Q3. Write a program to sort an array using selection sort algorithm.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
int smallest (int arr [], int k, int n);
void selection_sort (int arr [], int n);
void main ()
{
    int arr [10], i, n;

    printf ("\n Enter the number of elements in the array: ");

    scanf ("%d", &n);

    printf ("\n Enter the elements of the array: ");

    for (i=0; i<n; i++)
    {
        scanf ("%d", &arr[i]);
    }

    selection_sort (arr, n);

    printf ("\n The sorted array is: \n");

    for (i=0; i<n; i++)

        printf (" %d\t", arr[i]);
}
```

```

int smallest (int arr [], int k, int n)
{
    int pos = k, small=arr[k], i;
    for (i=k+1; i<n; i++)
    {
        if(arr[i]< small)
        {
            small = arr[i];
            pos = i;
        }
    }
    return pos;
}

void selection_sort (int arr [], int n)
{
    int k, pos, temp;
    for (k=0; k<n; k++)
    {
        pos = smallest (arr, k, n);
        temp = arr[k];
        arr[k] = arr[pos];
        arr[pos] = temp;
    }
}

```

Output:

Enter the number of elements in the array: 5

Enter the elements of the array: 8

6

12

50

2

The sorted array is: 2 6 8 12 50.

4. SHELL SORT:

Shell sort, invented by Donald Shell in 1959, is a sorting algorithm that is a generalization of insertion sort. While discussing insertion sort, we have observed two things:

- First, insertion sort works well when the input data is 'almost sorted'.
- Second, insertion sort is quite inefficient to use as it moves the values just one position at a time.

Shell sort is considered an improvement over insertion sort as it compares elements separated by a gap of several positions. This enables the element to take bigger steps towards its expected position.

In Shell sort, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes. However, the last step of shell sort is a plain insertion sort. But by the time we reach the last step, the elements are already 'almost sorted', and hence it provides good performance.

Technique:

To visualize the way in which shell sort works, perform the following steps:

- Step 1. Arrange the elements of the array in the form of a table and sort the columns (using insertion sort).
- Step 2. Repeat Step 1, each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.

Example: Sort the elements given below using shell sort.

63, 19, 7, 90, 81, 36, 54, 45, 72, 27, 22, 9, 41, 59, 33

Solution

Arrange the elements of the array in the form of a table and sort the columns.

Result:

63 19 7 90 81 36 54 45
72 27 22 9 41 59 33

63 19 7 9 41 36 33 45
72 27 22 90 81 59 54

The elements of the array can be given as:

63, 19, 7, 9, 41, 36, 33, 45, 72, 27, 22, 90, 81, 59, 54

Repeat Step 1 with smaller number of long columns.

Result:

63 19 7 9 41
36 33 45 72 27
22 90 81 59 54

22 19 7 9 27
36 33 45 59 41
63 90 81 72 54

The elements of the array can be given as:

22, 19, 7, 9, 27, 36, 33, 45, 59, 41, 63, 90, 81, 72, 54

Repeat Step 1 with smaller number of long columns.

Result:

22 19 7
9 27 36
33 45 59
41 63 90
81 72 54

9 19 7
22 27 36
33 45 54
41 63 59
81 72 90

The elements of the array can be given as:

9, 19, 7, 22, 27, 36, 33, 45, 54, 41, 63, 59, 81, 72, 90

Finally, arrange the elements of the array in a single column and sort the column.

Result:

9	7
19	9
7	19
22	22
27	27
36	33
33	36
45	41
54	45
41	54
63	59
59	63
81	72
72	81
90	90

Finally, the elements of the array can be given as:

7, 9, 19, 22, 27, 33, 36, 41, 45, 54, 59, 63, 72, 81, 90

Algorithm for shell sort:

Shell_Sort (Arr, n)

Step 1: SET FLAG = 1, GAP_SIZE = N

Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP_SIZE > 1

Step 3: SET FLAG = 0

Step 4: SET GAP_SIZE = (GAP_SIZE + 1) / 2

Step 5: Repeat Step 6 for I = 0 to I < (N - GAP_SIZE)

Step 6: IF Arr [I + GAP_SIZE] < Arr[I]
 SWAP Arr [I + GAP_SIZE], Arr[I]
 SET FLAG = 1

Step 7: END

Programming Example

Q4. Write a program to implement shell sort algorithm.

Source Code:

```
#include<stdio.h>
void main()
{
    int arr[10]={-1};
    int i, j, n, flag = 1, gap_size, temp;
    printf("\n Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("\n Enter %d numbers: ",n); // n was added
    for(i=0;i<n;i++)
        scanf("%d", &arr[i]);
    gap_size = n;
    while(flag == 1 || gap_size > 1)
    {
        flag = 0;
        gap_size = (gap_size + 1) / 2;
        for(i=0; i< (n - gap_size); i++)
        {
            if( arr[i+gap_size] < arr[i])
            {
                temp = arr[i+gap_size];
                arr[i+gap_size] = arr[i];
                arr[i] = temp;
                flag = 0;
            }
        }
    }
    printf("\n The sorted array is: \n");
    for(i=0;i<n;i++){
        printf(" %d\t", arr[i]);
    }
}
```

Output:

Enter the number of elements in the array: 5

Enter the elements of the array: 8

6

12

50

2

The sorted array is: 2 6 8 12 50.

5. QUICK SORT:

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 uses **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*". Quick sort is also known as partition exchange sort.

The quick sort algorithm works as follows:

1. Select an element pivot from the array elements
2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the partition operation.
3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Like merge sort, the base case of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After each iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.

Thus, the main task is to find the pivot element, which will partition the array into two halves.

Technique:

Quick sort works as follows:

1. Set the index of the first element in the array to *loc* and *left* variables. Also, set the index of the last element of the array to the *right* variable.

That is, $loc = 0$, $left = 0$, and $right = n - 1$ (where n is the number of elements in the array)

- 2 Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable loc.

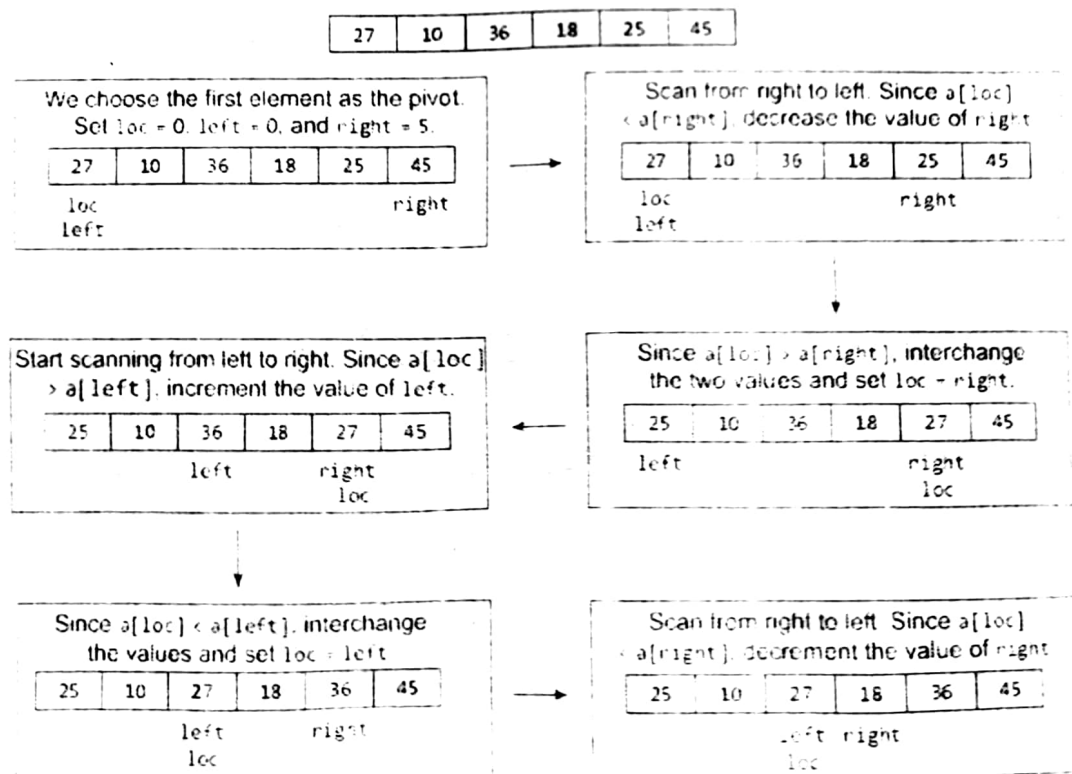
That is, `a[loc]` should be less than `a[right]`.

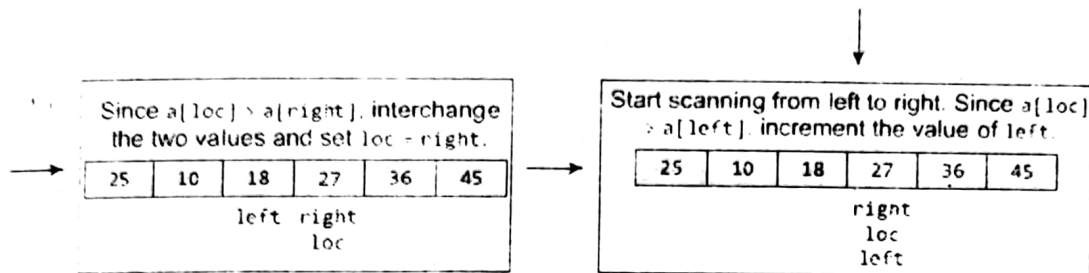
- a) If that is the case, then simply continue comparing until right becomes equal to loc. Once $\text{right} = \text{loc}$, it means the pivot has been placed in its correct position.
 - b) However, if at any point, we have $a[\text{loc}] > a[\text{right}]$, then interchange the two values and jump to Step 3.
 - c) Set $\text{loc} = \text{right}$.
3. Start from the element pointed by left and scan the array from left to right, comparing each element on the way with the element pointed by loc.

That is, $a[\text{loc}]$ should be greater than $a[\text{left}]$.

- If that is the case, then simply continue comparing until `left` becomes equal to `loc`. Once `left = loc`, it means the pivot has been placed in its correct position.
- However, if at any point, we have $a[loc] < a[left]$, then interchange the two values and jump to Step 2.
- Set `loc = left`.

Example: Sort the elements given in the following array using quick sort algorithm





Now $left = loc$, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

Algorithm for Quick Sort:

PARTITION (ARR, BEG, END)

Step 1: [INITIALIZE] SET $LEFT = BEG$, $RIGHT = END$, $LOC = BEG$, $FLAG = 0$

Step 2: Repeat Steps 3 to 6 while $FLAG = 0$

Step 3: Repeat while $ARR[LOC] \leq ARR[RIGHT]$ AND $LOC \neq RIGHT$
 SET $RIGHT = RIGHT - 1$

 [END OF LOOP]

Step 4: IF $LOC = RIGHT$
 SET $FLAG = 1$
 ELSE IF $ARR[LOC] > ARR[RIGHT]$
 SWAP $ARR[LOC]$ with $ARR[RIGHT]$
 SET $LOC = RIGHT$
 [END OF IF]

Step 5: IF $FLAG = 0$
 Repeat while $ARR[LOC] \geq ARR[LEFT]$ AND $LOC \neq LEFT$
 SET $LEFT = LEFT + 1$
 [END OF LOOP]

Step 6: IF $LOC = LEFT$
 SET $FLAG = 1$
 ELSE IF $ARR[LOC] < ARR[LEFT]$
 SWAP $ARR[LOC]$ with $ARR[LEFT]$
 SET $LOC = LEFT$
 [END OF IF]

 [END OF IF]

Step 7: [END OF LOOP]

Step 8: END

QUICK_SORT (ARR, BEG, END)

```

Step 1: IF (BEG < END)
        CALL PARTITION (ARR, BEG, END, LOC)
        CALL QUICKSORT(ARR, BEG, LOC - 1)
        CALL QUICKSORT(ARR, LOC + 1, END)
    [END OF IF]
Step 2: END

```

Complexity of Quick Sort:

Worst Case Time Complexity [Big-O]: $O(n^2)$

Best Case Time Complexity [Big-omega]: $\Omega(n \log n)$

Average Time Complexity [Big-theta]: $\Theta(n \log n)$

Pros and Cons of Quick Sort:

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. Quick Sort is complex and massively recursive.

Programming Example

Q5. Write a program to implement quick sort algorithm.

Source Code:

```

#include <stdio.h>
#include <conio.h>
#define size 100
int partition(int a[], int beg, int end);
void quick_sort(int a[], int beg, int end);
void main()
{
    int arr[size], i, n;
    printf("\n Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("\n Enter the elements of the array: ");
    for(i=0; i<n; i++)
    {
        scanf("%d", &arr[i]);
    }
    quick_sort(arr, 0, n-1);
    printf("\n The sorted array is: \n");
    for(i=0; i<n; i++)
    printf(" %d\t", arr[i]);
    getch();
}

```

```

int partition(int a[], int beg, int end)
{
    int left, right, temp, loc, flag;
    loc = left = beg;
    right = end;
    flag = 0;
    while(flag != 1)
    {
        while((a[loc] <= a[right]) && (loc!=right))
            right--;

        if(loc==right)
            flag = 1;
        else if(a[loc]>a[right])
        {
            temp = a[loc];
            a[loc] = a[right];
            a[right] = temp;
            loc = right;
        }
        if(flag!=1)
        {
            while((a[loc] >= a[left]) && (loc!=left))
                left++;
            if(loc==left)
                flag = 1;
            else if(a[loc] < a[left])
            {
                temp = a[loc];
                a[loc] = a[left];
                a[left] = temp;
                loc = left;
            }
        }
    }
    return loc;
}

void quick_sort(int a[], int beg, int end)
{
    int loc;
    if(beg<end)
    {
        loc = partition(a, beg, end);
        quick_sort(a, beg, loc-1);
        quick_sort(a, loc+1, end);
    }
}

```

Output:

Enter the number of elements in the array: 5

Enter the elements of the array: 8

6

12

50

2

The sorted array is: 2 6 8 12 50.

6. MERGE SORT:

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.

Divide Divide means partitioning the n -element array to be sorted into two sub-arrays of $n/2$ elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A_1 and A_2 , each containing about half of the elements of A .

Conquer: Conquer means sorting the two sub-arrays recursively using merge sort.

Combine: Combine means merging the two sorted sub-arrays of size $n/2$ to produce the sorted array of n elements.

Merge sort algorithm focuses on two main concepts to improve its performance (running time)

- A smaller list takes fewer steps and thus less time to sort than a large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted lists rather than creating it using two unsorted lists

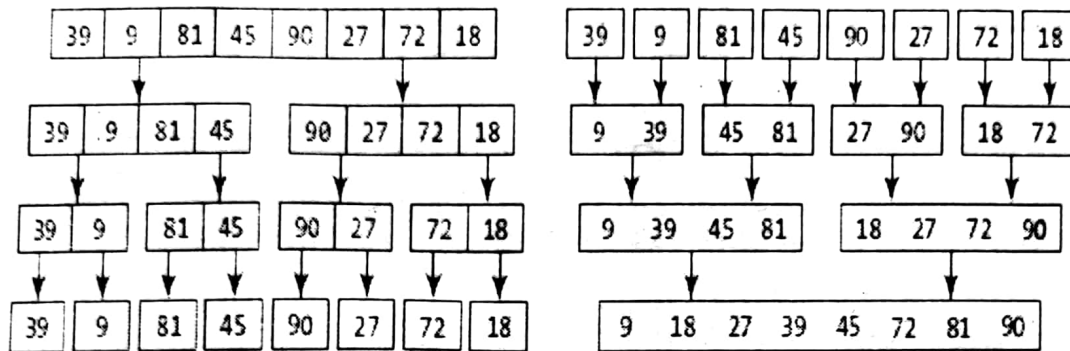
The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array
- Merge the two sub-arrays to form a single sorted list.

Example: Sort the array given below using merge sort.

39	9	81	45	90	27	72	18
----	---	----	----	----	----	----	----

Solution:

**Algorithm for merge sort:**

MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET $I = \text{BEG}$, $J = \text{MID} + 1$, $\text{INDEX} = 0$

Step 2: Repeat while $(I \leq \text{MID})$ AND $(J \leq \text{END})$

 IF $\text{ARR}[I] < \text{ARR}[J]$

 SET $\text{TEMP}[\text{INDEX}] = \text{ARR}[I]$

 SET $I = I + 1$

 ELSE

 SET $\text{TEMP}[\text{INDEX}] = \text{ARR}[J]$

 SET $J = J + 1$

 [END OF IF]

 SET $\text{INDEX} = \text{INDEX} + 1$

 [END OF LOOP]

Step 3: [Copy the remaining elements of right sub-array, if any]

 IF $I > \text{MID}$

 Repeat while $J \leq \text{END}$

 SET $\text{TEMP}[\text{INDEX}] = \text{ARR}[J]$

 SET $\text{INDEX} = \text{INDEX} + 1$, SET $J = J + 1$

 [END OF LOOP]

 [Copy the remaining elements of left sub-array, if any]

 ELSE

 Repeat while $I \leq \text{MID}$

 SET $\text{TEMP}[\text{INDEX}] = \text{ARR}[I]$

 SET $\text{INDEX} = \text{INDEX} + 1$, SET $I = I + 1$

 [END OF LOOP]

 [END OF IF]

Step 4: [Copy the contents of TEMP back to ARR] SET $K=0$

Step 5: Repeat while $K < \text{INDEX}$

 SET $\text{ARR}[K] = \text{TEMP}[K]$

 SET $K = K + 1$

 [END OF LOOP]

Step 6: END

MERGE_SORT(ARR, BEG, END)

Step 1: IF BEG < END

 SET MID = (BEG + END)/2

 CALL MERGE_SORT (ARR, BEG, MID)

 CALL MERGE_SORT (ARR, MID + 1, END)

 MERGE (ARR, BEG, MID, END)

 [END OF IF]

Step 2: END

Complexity of Merge Sort:

Worst Case Time Complexity [Big-O]: $O(n \log n)$

Best Case Time Complexity [Big-omega]: $\Omega(n \log n)$

Average Time Complexity [Big-theta]: $\Theta(n \log n)$

Programming Example

Q6. Write a program to implement merge sort.

Source Code:

```
#include <stdio.h>
#include <conio.h>
#define size 100

void merge(int a[], int, int, int);
void merge_sort(int a[], int, int);
void main()
{
    int arr[size], i, n;
    printf("\n Enter the number of elements in the array : ");
    scanf("%d", &n);
    printf("\n Enter the elements of the array: ");
    for(i=0; i<n; i++)
    {
        scanf("%d", &arr[i]);
    }
    merge_sort(arr, 0, n-1);
    printf("\n The sorted array is: \n");
    for(i=0; i<n; i++)
        printf(" %d\t", arr[i]);
    getch();
}
```

```

void merge(int arr[], int beg, int mid, int end)
{
    int i=beg, j=mid+1, index=beg, temp[size], k;
    while((i<=mid) && (j<=end))
    {
        if(arr[i] < arr[j])
        {
            temp[index] = arr[i];
            i++;
        }
        else
        {
            temp[index] = arr[j];
            j++;
        }
        index++;
    }
    if(i>mid)
    {
        while(j<=end)
        {
            temp[index] = arr[j];
            j++;
            index++;
        }
    }
    else
    {
        while(i<=mid)
        {
            temp[index] = arr[i];
            i++;
            index++;
        }
    }
    for(k=beg;k<index;k++)
        arr[k] = temp[k];
}

void merge_sort(int arr[], int beg, int end)
{
    int mid;
    if(beg<end)
    {
        mid = (beg+end)/2;
        merge_sort(arr, beg, mid);
        merge_sort(arr, mid+1, end);
        merge(arr, beg, mid, end);
    }
}

```

Output:

Enter the number of elements in the array: 5

Enter the elements of the array: 8

6

12

50

2

The sorted array is: 2 6 8 12 50.

7. HEAP SORT

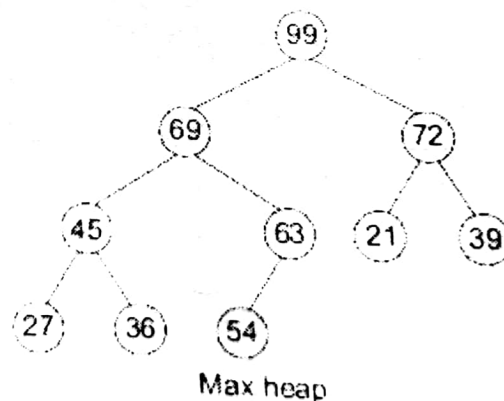
Before we discussing about heap sort, we have discussed what is binary heaps, how to build a heap H from an array, how to insert a new element in an already existing heap, and how to delete an element from H.

What is binary heap?

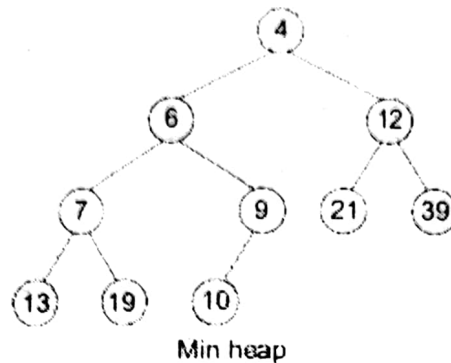
A binary heap is a complete binary tree in which every node satisfies the heap property which states that:

If B is a child of A, then $\text{key}(A) \geq \text{key}(B)$

This implies that elements at every node will be either greater than or equal to the element at its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a **max-heap**.



Alternatively, elements at every node will be either less than or equal to the element at its left and right child. Thus, the root has the lowest key value. Such a heap is called a **min-heap**.



The properties of binary heaps are given as follows:

- Since a heap is defined as a complete binary tree, all its elements can be stored sequentially in an array. It follows the same rules as that of a complete binary tree. That is, if an element is at position i in the array, then its left child is stored at position $2i$ and its right child at position $2i+1$. Conversely, an element at position i has its parent stored at position $i/2$.
- Being a complete binary tree, all the levels of the tree except the last level are completely filled.
- Heaps (also known as partially ordered trees) are a very popular data structure for implementing priority queues.
- The height of a binary tree is given as $\log_2 n$, where n is the number of elements.

A binary heap is a useful data structure in which elements can be added randomly but only the element with the highest value is removed in case of max heap and lowest value in case of min heap. A binary heap is more space efficient and simpler.

Inserting a New Element in a Binary Heap

Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well.

To do this, compare the new value with its parent to check if they are in the correct order. If they are, then the procedure halts, else the new value and its parent's value are swapped and Step 2 is repeated.

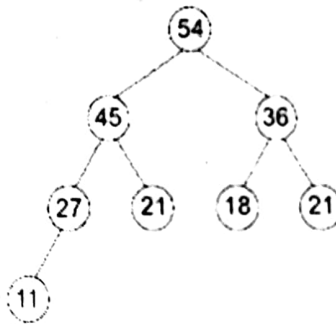
Algorithm to insert an element in a max heap:

```

Step 1: [Add the new value and set its POS]
        SET  $N = N + 1$ ,  $POS = N$ 
Step 2: SET  $HEAP[N] = VAL$ 
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while  $POS > 1$ 
Step 4:   SET  $PAR = POS/2$ 
Step 5:   IF  $HEAP[POS] \leq HEAP[PAR]$ ,
        then Goto Step 6.
        ELSE
            SWAP  $HEAP[POS]$ ,  $HEAP[PAR]$ 
             $POS = PAR$ 
        [END OF IF]
    [END OF LOOP]
Step 6: RETURN

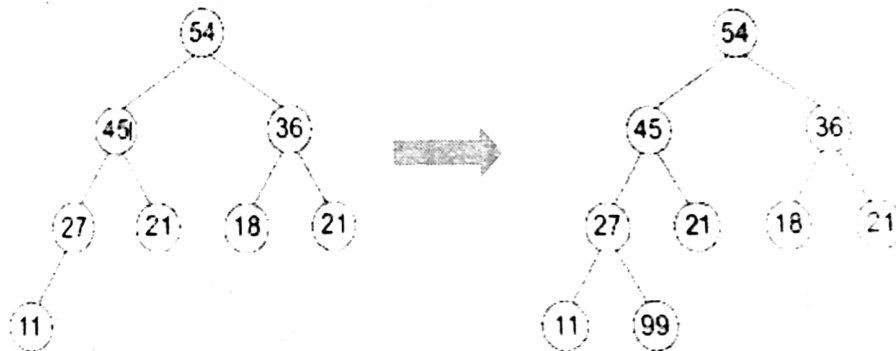
```

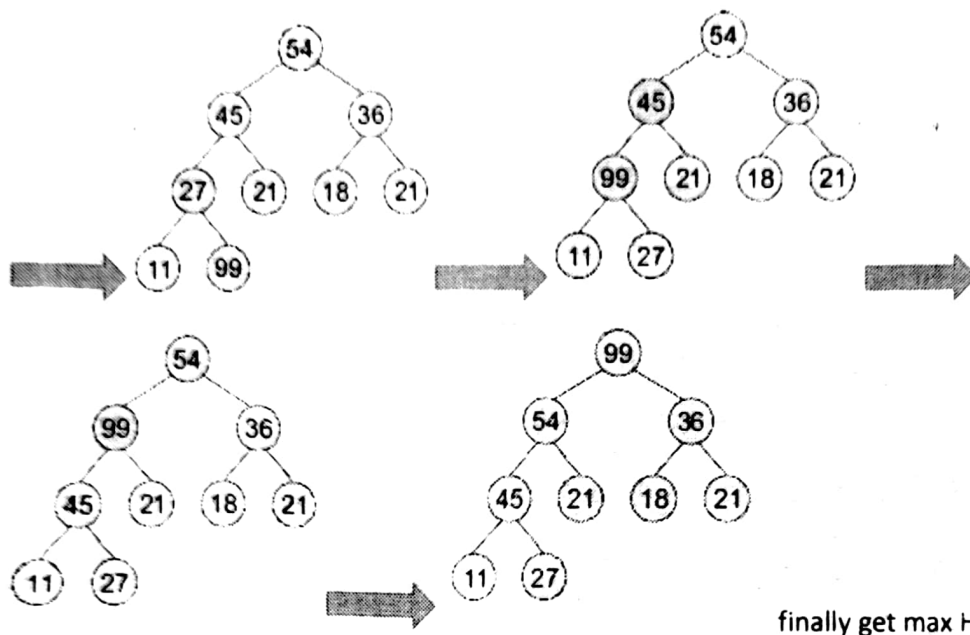
Example: Consider the max heap given below in figure and insert 99 in it.



Solution:

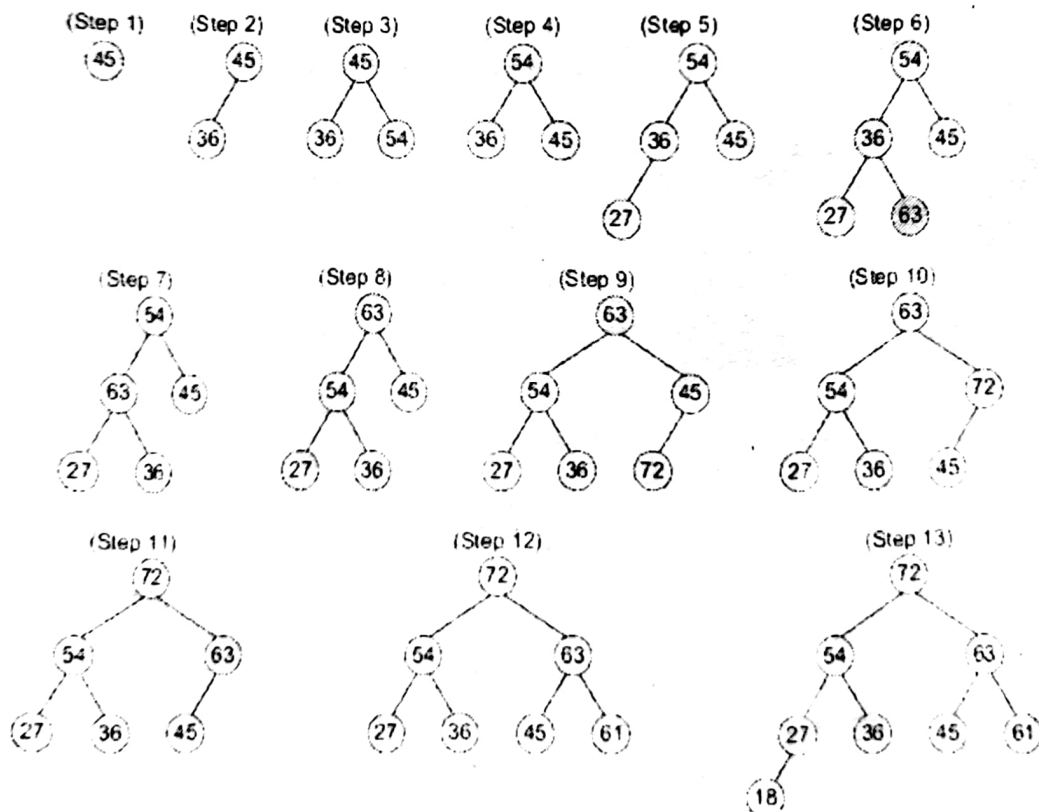
First step the insert new element 99 in existing heap so that the heap is a complete binary tree and then heapify the binary heap.





Example: Build a max heap from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18.

Solution:



```

    }
    Heap[j/2]val;
}

```

8. RADIX SORT

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort.

Observe that words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.

During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the *n*th pass, where *n* is the length of the name with maximum number of letters.

After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the names from the second bucket, and so on.

When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, ..., 9) and the number of passes will depend on the length of the number having maximum number of digits.

Algorithm for radix sort:

Algorithm for RadixSort (ARR, N)

```

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:     SET I = 0 and INITIALIZE buckets
Step 6:     Repeat Steps 7 to 9 while I < N-1
Step 7:         SET DIGIT = digit at PASSth place in A[I]
Step 8:         Add A[I] to the bucket numbered DIGIT
Step 9:         INCREMENT bucket count for bucket numbered DIGIT
                [END OF LOOP]
Step 10:     Collect the numbers in the bucket
                [END OF LOOP]
Step 11: END

```

Example 8. : Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

Solution:

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

UNIT 6 SORTING

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as 123, 345, 472, 555, 567, 654, 808, 911, and 924.

Complexity of Radix Sort:

To calculate the complexity of radix sort algorithm, assume that there are n numbers that have to be sorted and k is the number of digits in the largest number. In this case, the radix sort algorithm is called a total of k times. The inner loop is executed n times. Hence, the entire radix sort algorithm takes $O(kn)$ time to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in $O(n)$ asymptotic time.

Programming Example:

Q.) Write a program to implement radix sort algorithm.

```

#include <stdio.h>
#include <conio.h>
#define size 10
int largest(int arr[], int n);
void radix_sort(int arr[], int n);
void main()
{
    int arr[size], i, n;
    printf("\n Enter the number of elements in the array: ");
    scanf("%d", &n);
    printf("\n Enter the elements of the array: ");
    for(i=0; i<n; i++)
    {
        scanf("%d", &arr[i]);
    }
    radix_sort(arr, n);
    printf("\n The sorted array is: \n");
    for(i=0; i<n; i++)
    printf(" %d\t", arr[i]);
    getch();
}

int largest(int arr[], int n)
{
    int large=arr[0], i;
    for(i=1; i<n; i++)
    {
        if(arr[i]>large)
            large = arr[i];
    }
    return large;
}

```

UNIT_6_SORTING

```

void radix_sort(int arr[], int n)
{
    int bucket[size][size], bucket_count[size];
    int i, j, k, remainder, NOP=0, divisor=1, large, pass;
    large = largest(arr, n);
    while(large>0)
    {
        NOP++;
        large/=size;
    }
    for(pass=0;pass<NOP;pass++) // Initialize the buckets
    {
        for(i=0;i<size;i++)
            bucket_count[i]=0;
        for(i=0;i<n;i++)
        {
            // sort the numbers according to the digit at passth place
            remainder = (arr[i]/divisor)%size;
            bucket[remainder][bucket_count[remainder]] = arr[i];
            bucket_count[remainder] += 1;
        }
        // collect the numbers after PASS pass
        i=0;
        for(k=0;k<size;k++)
        {
            for(j=0;j<bucket_count[k];j++)
            {
                arr[i] = bucket[k][j];
                i++;
            }
        }
        divisor *= size;
    }
}

```

Deleting an Element from a Binary Heap:

Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:

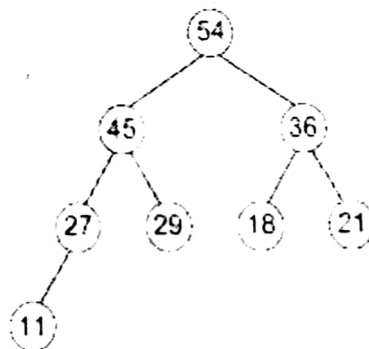
1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.
2. Delete the last node.
3. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

Algorithm to delete the root element from a max heap

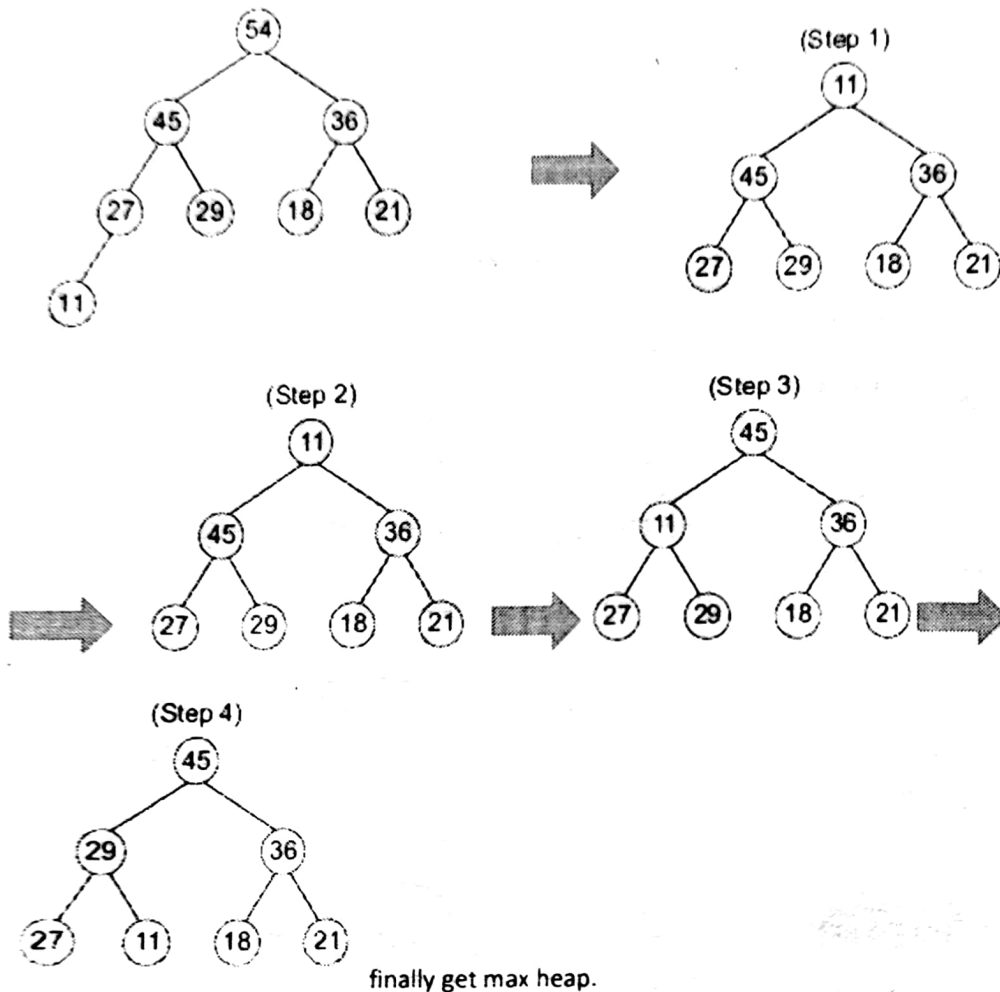
```

Step 1: [Remove the last node from the heap]
        SET LAST = HEAP[N], SET N = N - 1
Step 2: [Initialization]
        SET PTR = 1, LEFT = 2, RIGHT = 3
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT ≤ N
Step 5: IF HEAP[PTR] ≥ HEAP[LEFT] AND
        HEAP[PTR] ≥ HEAP[RIGHT]
        Go to Step 8
        [END OF IF]
Step 6: IF HEAP[RIGHT] ≤ HEAP[LEFT]
        SWAP HEAP[PTR], HEAP[LEFT]
        SET PTR = LEFT
    ELSE
        SWAP HEAP[PTR], HEAP[RIGHT]
        SET PTR = RIGHT
    [END OF IF]
Step 7: SET LEFT = 2 * PTR and RIGHT = LEFT + 1
        [END OF LOOP]
Step 8: RETURN
  
```

Example: Consider the max heap H given below in the figure and delete the root node's value.



Solution:



Applications of Binary Heaps

Binary heaps are mainly applied for:

1. Sorting an array using heapsort algorithm
2. Implementing priority queues.

What is Heap Sort?

Heap sort is one of the sorting algorithms used to arrange a list of elements in an order. Heap sort 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.

Given an array ARR with n elements, the heap sort algorithm can be used to sort ARR in two phases:

In Step 1, build a heap H using the elements of ARR.

In Step 2, repeatedly delete the root element of the heap formed in Step 1.

In a max heap, we know that the largest value in H is always present at the root node. So in Step 2, when the root element is deleted, we are actually collecting the elements of ARR in decreasing order.

The algorithm of heap Sort:

HEAPSORT(ARR, N)

Step 1: [Build Heap H]

Repeat for I = 0 to N-1

CALL Insert_Heap(ARR, N, ARR[I])

[END OF LOOP]

Step 2: (Repeatedly delete the root element)

Repeat while N > 0

CALL Delete_Heap(ARR, N, VAL)

SET N = N - 1

[END OF LOOP]

Step 3: END

Complexity of Heap Sort:

Worst Case Time Complexity [Big-O]: $O(n \log n)$

Best Case Time Complexity [Big-omega]: $\Omega(n \log n)$

Average Time Complexity [Big-theta]: $\Theta(n \log n)$

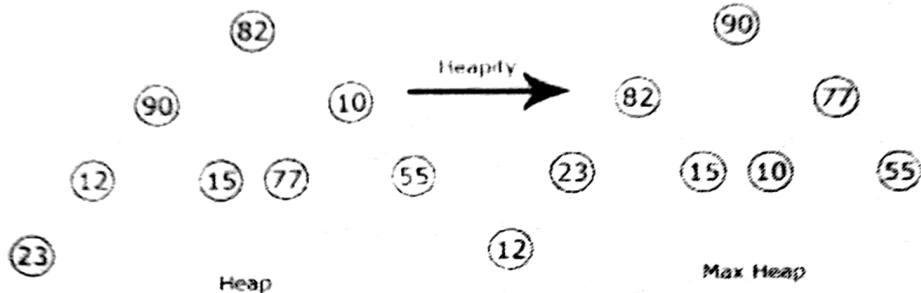
Example:

UNIT 6 SORTING

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

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

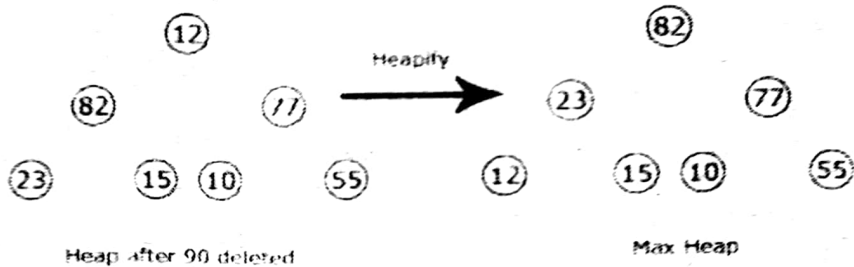
Step 1 Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

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

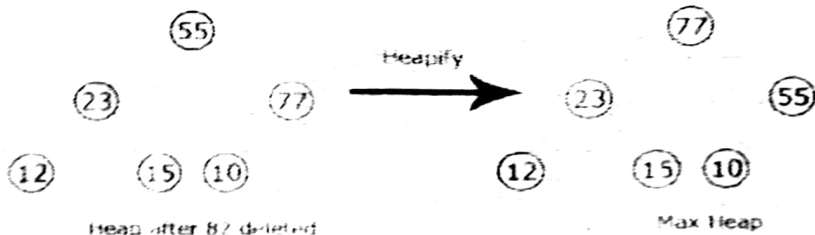
Step 2 Delete root (90) from the Max Heap. To delete root node it needs to be swapped with last node (12). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

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

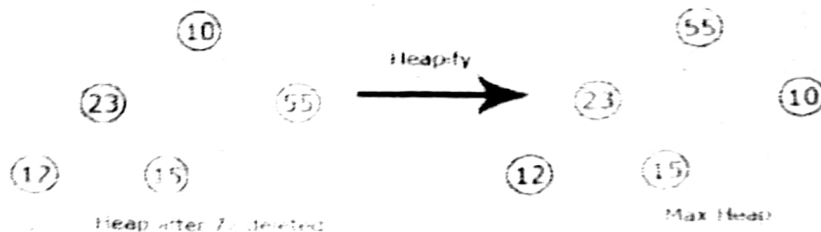
Step 3 Delete root (82) from the Max Heap. To delete root node it needs to be swapped with last node (55). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

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

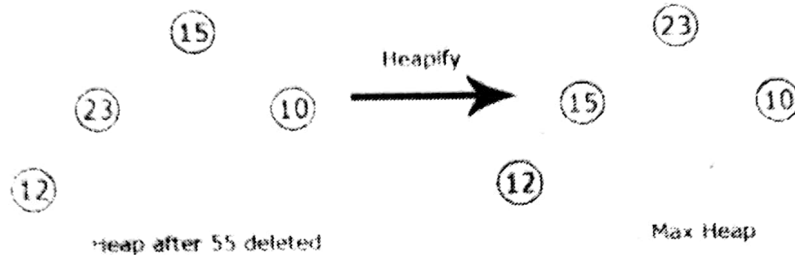
Step 4 Delete root (77) from the Max Heap. To delete root node it needs to be swapped with last node (10). After delete tree needs to be heapify to make it Max Heap.



list of numbers after 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**

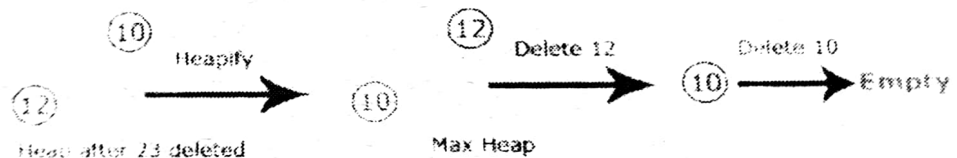
Step 6 Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



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 & 10 from the Max Heap.

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

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

Programming Example

Q6. Write a program to implement Heap Sort.

Source Code:

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
```

```

void RestoreHeapUp(int *,int);
void RestoreHeapDown(int*,int,int);
int main()
{
    int Heap[MAX],n,i,j;
    clrscr();
    printf("\n Enter the number of elements : ");
    scanf("%d",&n);
    printf("\n Enter the elements : ");
    for(i=1;i<=n;i++)
    {
        scanf("%d",&Heap[i]);
        RestoreHeapUp(Heap, i); // Heapify
    }
    // Delete the root element and heapify the heap
    j=n;
    for(i=1;i<=j;i++)
    {
        int temp;
        temp=Heap[1];
        Heap[1]= Heap[n];
        Heap[n]=temp;
        n = n-1; // The element Heap[n] is supposed to be deleted
        RestoreHeapDown(Heap,1,n); // Heapify
    }
    n=j;
    printf("\n The sorted elements are: ");
    for(i=1;i<=n;i++)
        printf("%4d",Heap[i]);
    return 0;
}

void RestoreHeapUp(int *Heap,int index)
{
    int val = Heap[index];
    while( (index>1) && (Heap[index/2] < val) )// Check parent's value
    {
        Heap[index]=Heap[index/2];
        index /= 2;
    }
    Heap[index]=val;
}

void RestoreHeapDown(int *Heap,int index,int n)
{
    int val = Heap[index];
    int j=index*2;
    while(j<=n)
    {
        if( (j<n) && (Heap[j] < Heap[j+1]))// Check sibling's value
            j++;
        if(Heap[j] < Heap[j/2]) // Check parent's value
            break;
        Heap[j/2]=Heap[j];
        j=j*2;
    }
    Heap[index]=val;
}

```

