UNIT-II

Based on the organizing method of data structure, data structures are divided into two types.

- **Linear Data Structures**
- **Non - Linear Data Structures**

2.1 LINEAR DATA STRUCTURE

If a data structure organizes the data in sequential order, then that data structure is called a Linear Data Structure.

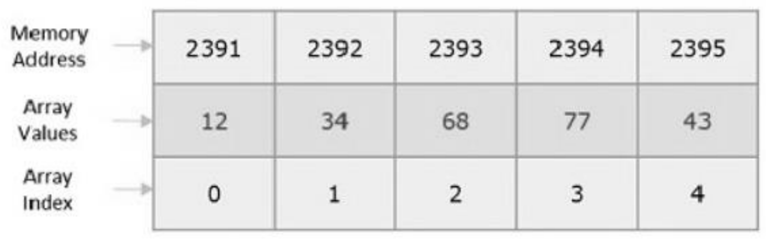
Example

1. Arrays
2. Linked List
3. Stack
4. Queue

2.2 WHAT ARE ARRAYS IN DATA STRUCTURES?

An array is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations. Arrays work on an index system starting from 0 to (n-1), where n is the size of the array.

a) Array declaration: An "array declaration" names the array and specifies the type of its



Memory Address →	2391	2392	2393	2394	2395
Array Values →	12	34	68	77	43
Array Index →	0	1	2	3	4

elements. It can also define the number of elements in the array. Above mentioned example is an array with char datatype and size of the array is 5.

Syntax: Array_datatypeArray_Name [Array_Size];

Eg: int a[5];

b) Array initialization:

Arrays are initialized in four different methods.

Method 1:

int a[5] = {2,4,6,1,3};

Method 2:

int a[] = {2,4,6,1,3};

Method 3:

int n;

scanf("%d",&n);

```
int arr[n];
for(int i=0;i<5;i++)
{
scanf("%d",&arr[i]);
}
```

Method 4

```
int arr[5]; arr[0]=1;
arr[1]=2;
arr[2]=3;
arr[3]=4;
arr[4]=5;
```

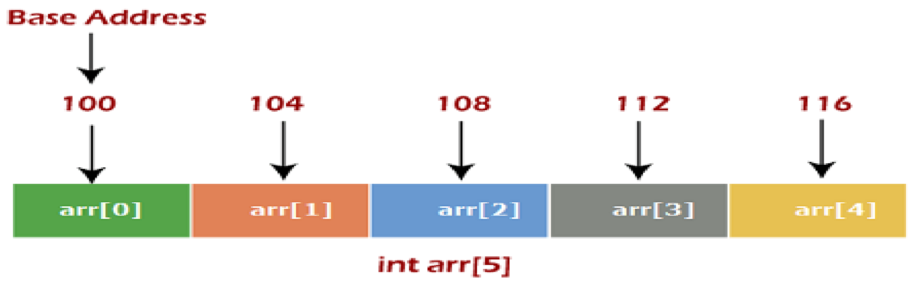
The number n of elements is called the length or size of the array. The length or the numbers of elements of the array can be obtained from the index set by the formula

$$\boxed{n=UB-LB+1}$$

When LB = 0,
Length = UB + 1
When LB = 1,
Length = UB
Where, UB is the largest index called the Upper Bound
LB is the smallest index, called the Lower Bound

2.3 REPRESENTATION OF LINEAR ARRAYS IN MEMORY

The elements of LA are stored in successive memory cells. The computer does not keep track of the address of every element of LA, but needs to keep track only the address of the first element of LA and is called the base address of LA.



LOC (LA [K]) = address of the element LA [K] of the array LA
Using the base address of LA, the computer calculates the address of any element of LA by the formula

$$\boxed{LOC(LA[K]) = Base(LA) + w(K - \text{lower bound})}$$

Where, w is the number of words/bytes per memory cell for the array LA.
Example:
BA=2000
Int a[10];
LOC(a[k])=BA+w(k-LB)
LOC(a[5])=2000+2(5-0) = 2000+10 = 2010

2.4 ARRAY OPERATIONS

1. TRAVERSING/TRAVERSAL

Let A be a collection of data elements stored in the memory of the computer. Suppose if the contents of each element of array A needs to be printed or to count the numbers of elements of A with a given property can be accomplished by Traversing.

Traversing is an accessing and processing each element in the array exactly once.

Algorithm 1: (Traversing a Linear Array)

Here UB means maximum size of an arrayor the number of elements in an array

Traversal in an array is a process of visiting each element once.

```
1. Start
2. Read an Array of certain size and datatype.
3. Initialize another variable 'i' with 0.
4. Print the ith value in the array and increment i.
5. Repeat Step 4 until the end of the array is reached.
6. End
```

Program

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int A[100],K=0,UB;
        printf("Enter the Array size less than 100: ");
        scanf("%d",&UB);
        printf("Enter the elements in array: \n");
        for(K=0;K<UB;K++)
        {
                scanf("%d",&A[K]);
        }
        printf("The Traverse of array is:\n");
        for(K=0;K<UB;K++)
        {
                printf("%d\n",A[K]);
        }
        getch();
        return 0;
}
```

2. INSERTING

Let A be a collection of data elements stored in the memory of the computer. Inserting refers to the operation of adding another element to the collection A.

Inserting an element at the "end" of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element.

Inserting an element in the middle of the array, then on average, half of the elements must be moved downwards to new locations to accommodate the new element and keep the order of the other elements.

Algorithm:

```
INSERT (LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that K ≤ N.
This algorithm inserts an element ITEM into the Kt h position in LA.
1. [Initialize counter] set J:= N
2. Repeat step 3 and 4 while J ≥ K
3. [Move Jt h element downward] Set LA [J+1] := LA[J]
4. [Decrease counter] set J:= J – 1
[End of step 2 loop]
5. [Insert element] set LA[K]:= ITEM
6. [Reset N] set N:= N+1
7. Exit
```

Program

Insert At the Beginning:

```
#include<stdio.h>
int main()
{
        int array[10], n,i, item;
        printf("Enter the size of array: ");
        scanf("%d", &n);
        printf("\nEnter Elements in array: ");
        for(i=0;i<n;i++)
        scanf("%d", &array[i]);
        printf("\n enter the element at the beginning");
        scanf("%d", &item);
        n++;
        for(i=n; i>1; i--)
        {
                array[i-1]=array[i-2];
        }
        array[0]=item;
        printf("resultant array element");
         for(i=0;i<n;i++)
        printf("\n%d", array[i]);
        return 0;
}
```

Insert At the End:

```
#include<stdio.h>
int main()
{
        int array[10], i, values;
        printf("Enter 5 Array Elements: ");
        for(i=0; i<5; i++)
        scanf("%d", &array[i]);
        printf("\nEnter Element to Insert: ");
        scanf("%d", &values);
        array[i] = values;
        printf("\nThe New Array is:\n");
        for(i=0; i<6; i++)
        printf("%d ", array[i]);
        return 0;
}
```

InsertAt a Specific Position:

```
#include<stdio.h>
int main()
{
        int array[100], pos, size, val;
        printf("Enter size of the array:");
        scanf("%d", &size);
        printf("\nEnter %d elements\n", size);
        for (int i = 0; i< size; i++)
        scanf("%d", &array[i]);
        printf("Enter the insertion location\n");
        scanf("%d", &pos);
        printf("Enter the value to insert\n");
        scanf("%d", &val);
        for (int i = size - 1; i>= pos - 1; i--)
        {
                array[i+1] = array[i];
        }
        array[pos-1] = val;
        printf("Resultant array is\n");
        for (int i = 0; i<= size; i++)
        printf("%d\n", array[i]);
        return 0;
}
```

3. DELETING

Deleting refers to the operation of removing one element to the collection A.
Deleting an element at the "end" of the linear array can be easily done with difficulties.
If element at the middle of the array needs to be deleted, then each subsequent elements be moved one location upward to fill up the array.

Algorithm

DELETE (LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that K ≤ N. this algorithm deletes the Kth element from LA

```
1. Set ITEM:= LA[K]
2. Repeat for J = K to N – 1
[Move J + 1 element upward] set LA[J]:= LA[J+1]
[End of loop]
3. [Reset the number N of elements in LA] set N:= N – 1
4. Exit
```

Program

Delete from the Beginning:

```
#include<stdio.h>
int main()
{
        int n,array[10];
        printf("enter the size of an array");
        scanf("%d" ,&n);
        printf("enter elements in an array");
        for(int i=0;i<n;i++)
        scanf("%d", &array[i]); n--;
        for(int i=0;i<n;i++)
        array[i]=array[i+1];
        printf("\nAfter deletion ");
        for(int i=0;i<n;i++)
        printf("\n%d" , array[i]);
        return 0;
}
```

Delete from the End:

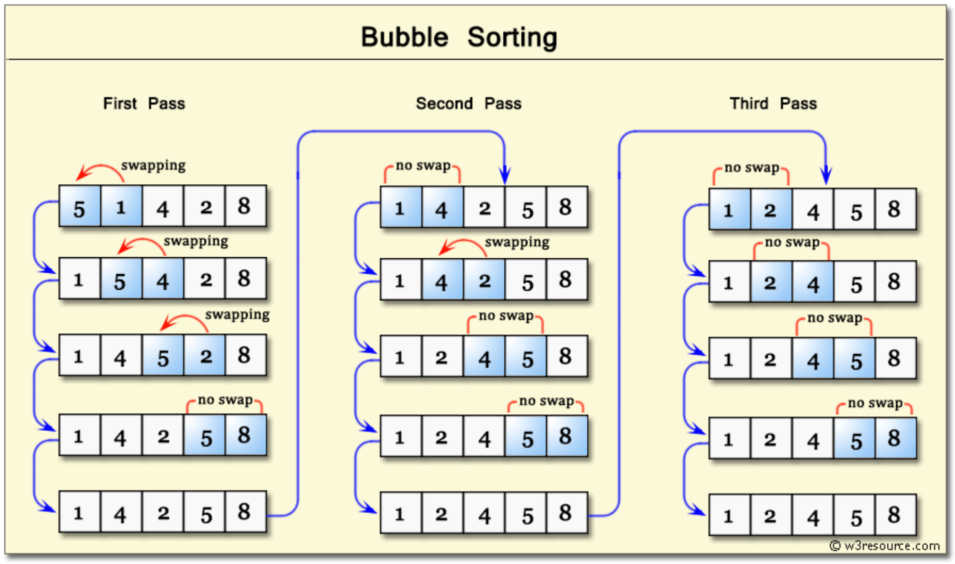
```
#include<stdio.h>
int main()
{
        int n,array[10];
        printf("enter the size of an array");
        scanf("%d" ,&n);
        printf("enter elements in an array");
        for(int i=0;i<n;i++)
        scanf("%d", &array[i]);
        printf("\nafter deletion array elements are");
        for(int i=0;i<n-1;i++)
        printf("\n%d" , array[i]);
        return 0;
}
```

2.5 SORTING TECHNIQUES

Sorting is the processing of arranging the data in ascending and descending order. There are several types of sorting in data structures namely – bubble sort, insertion sort, selection sort, bucket sort, heap sort, quick sort, radix sort etc.

1. BUBBLE SORT

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

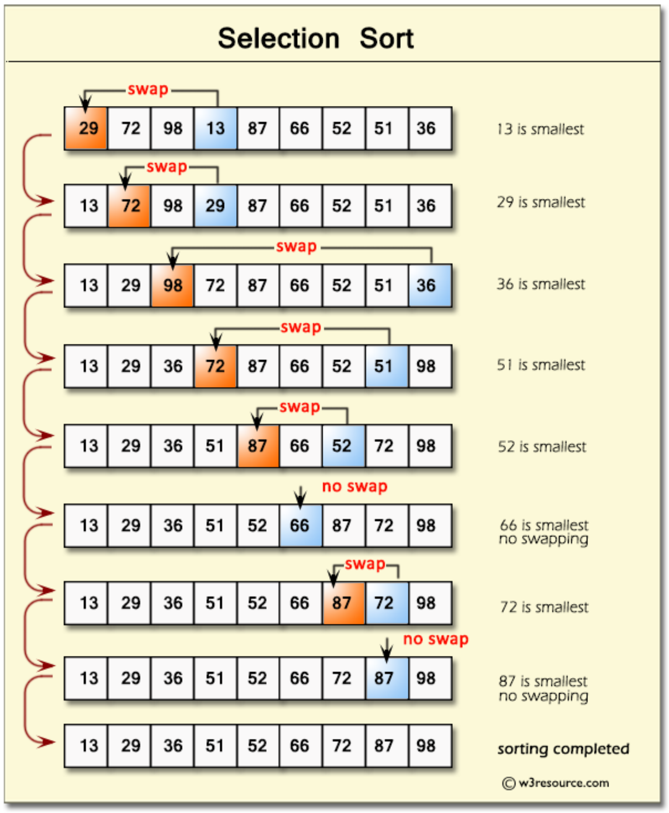


Program:

```
#include <stdio.h>
int main()
{
 int array[100], n, c, d, temp;
printf("Enter number of elements\n");
scanf("%d", &n);
printf("Enter %d integers\n", n);
 for (c = 0; c < n; c++)
scanf("%d", &array[c]);
 for (c = 0 ; c < n - 1; c++)
 {
  for (d = 0 ; d < n - c - 1; d++)
  {
      if (array[d] > array[d+1])
      {
        temp       = array[d];
        array[d]   = array[d+1];
        array[d+1] = temp;
      }
    }
  }
printf("Sorted list in ascending order:\n");
  for (c = 0; c < n; c++)
printf("%d\t", array[c]);
  return 0;
}
```

2. SELECTION SORT

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.This process is repeated for the remaining unsorted portion of the list until the entire list is sorted.

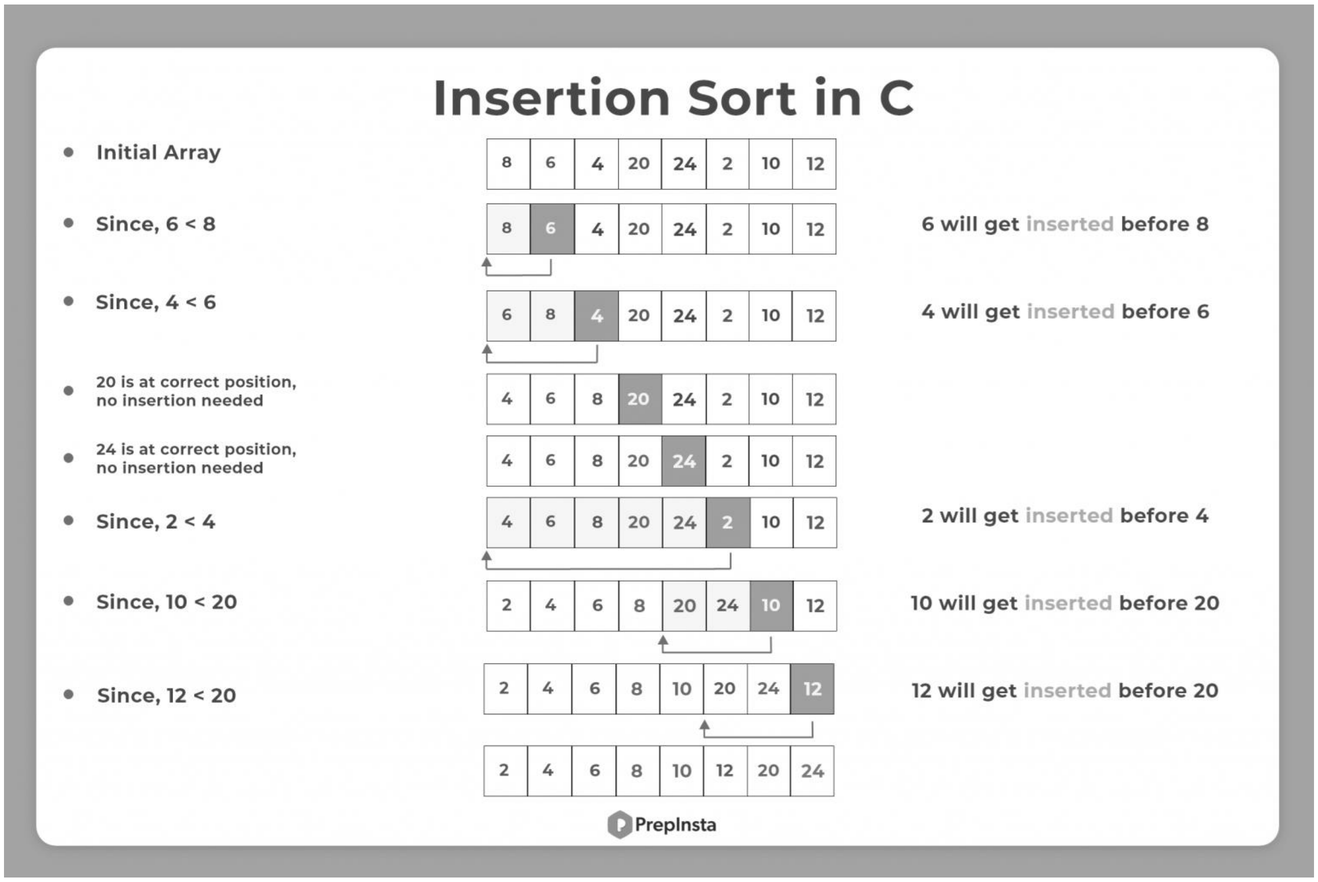


Program:

```
#include <stdio.h>
int main()
{
int a[100], n, i, j, position, temp;
printf("Enter number of elements");
scanf("%d", &n);
printf("Enter %d Numbers", n);
for (i = 0; i< n; i++)
scanf("%d", &a[i]);
for(i = 0; i< n - 1; i++)
{
position=i;
for(j = i + 1; j < n; j++)
{
if(a[position] > a[j])
position=j;
}
if(position != i)
{
temp=a[i];
a[i]=a[position];
a[position]=temp;
}
}
printf("Sorted Array:\n");
for(i = 0; i< n; i++)
printf("%d\t", a[i]);
return 0;
}
```

3. <u>INSERTION SORT</u>

Insertion sort is one of the simplest algorithm with simple implementation. Basically, Insertion sort is efficient for small data values. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list. Then the values from the unsorted parts are picked and placed at the correct position in the sorted part.


Insertion Sort in C
Initial Array
Since, 6 < 8
6 will get inserted before 8
Since, 4 < 6
4 will get inserted before 6
20 is at correct position, no insertion needed
24 is at correct position, no insertion needed
Since, 2 < 4
2 will get inserted before 4
Since, 10 < 20
10 will get inserted before 20
Since, 12 < 20
12 will get inserted before 20
8 6 4 20 24 2 10 12
6 8 4 20 24 2 10 12
4 6 8 20 24 2 10 12
2 4 6 8 20 24 10 12
2 4 6 8 10 20 24 12
2 4 6 8 10 12 20 24
PrepInsta

Program:

```
#include<stdio.h>
int main()
{
  int i, j, count, temp, number[25];
printf(Enter the number of elements: ");
scanf("%d",&count);
printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
scanf("%d",&number[i]);
  for(i=1;i<count;i++)
{
    temp=number[i];
    j=i-1;
    while((temp<number[j])&&(j>=0))
{
      number[j+1]=number[j];
      j=j-1;
    }
    number[j+1]=temp;
  }
printf("Sorted elements: \n");
```

```
  for(i=0;i<count;i++)
printf(" %d\t",number[i]);
  return 0;
}
```

4. MERGE SORT

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly.Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement.

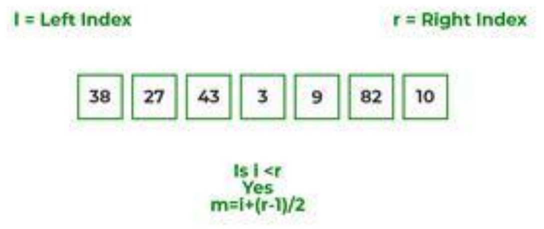
The Working Process of Merge Sort:

If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Illustration:

To know the functioning of merge sort lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

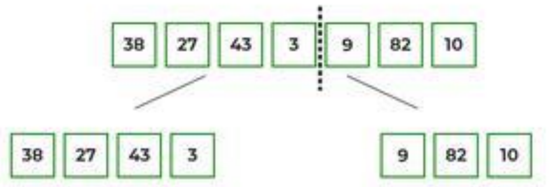
- At first, check if the left index of array is less than the right index, if yes then calculate its mid point.



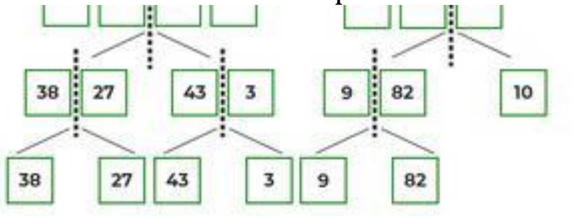
- Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.
- Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

l = Left Index
r = Right Index
38
27
43
3
9
82
10

- Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.

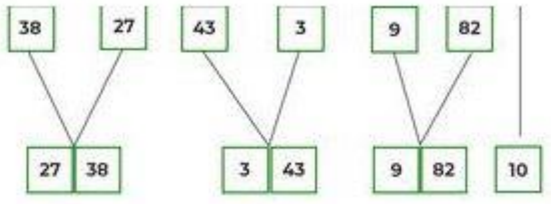


- Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.

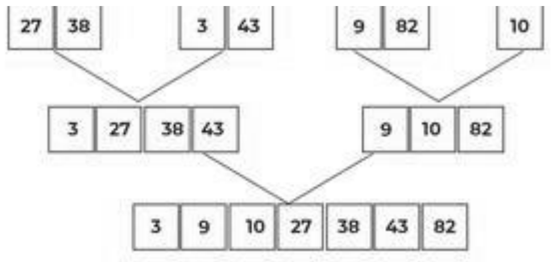


After dividing the array into smallest units merging starts, based on comparison of elements.

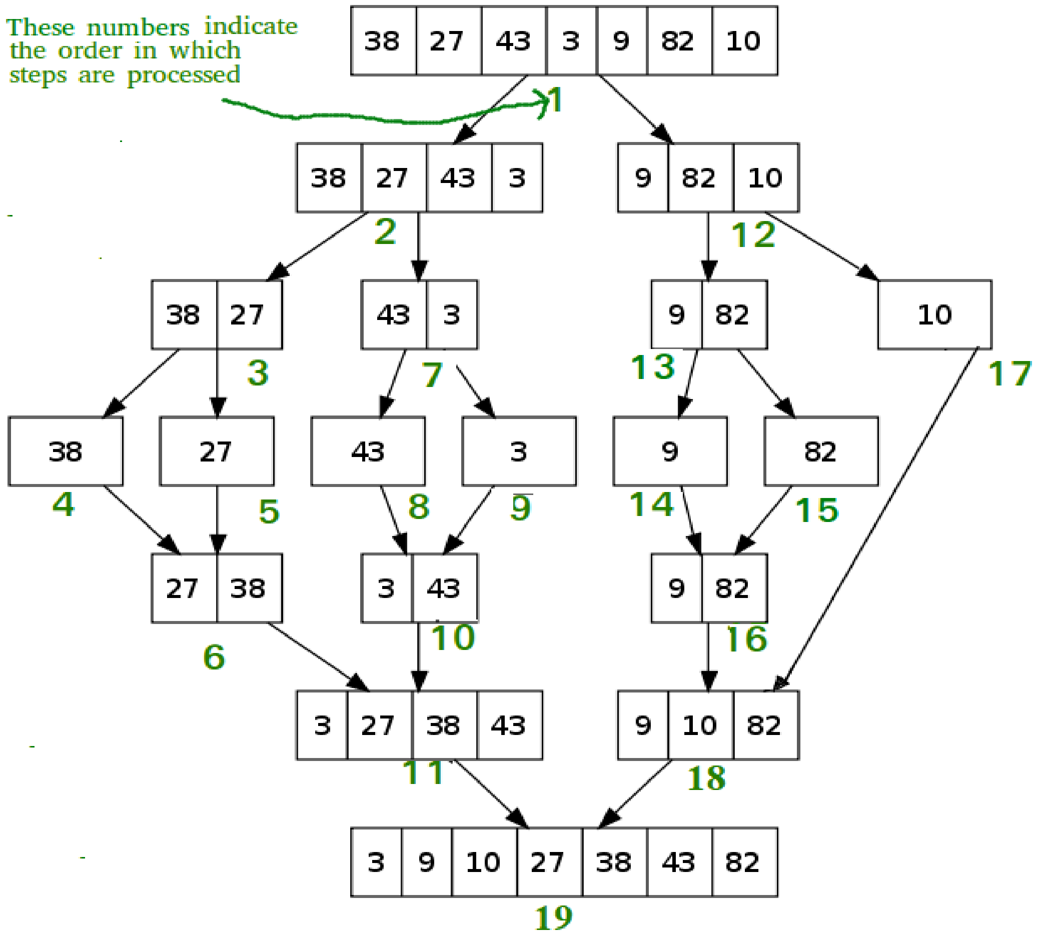
- After dividing the array into smallest units, start merging the elements again based on comparison of size of elements
- Firstly, compare the element for each list and then combine them into another list in a sorted manner.



After the final merging, the list looks like this:



If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.
The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.



5. <u>QUICK SORT</u>

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

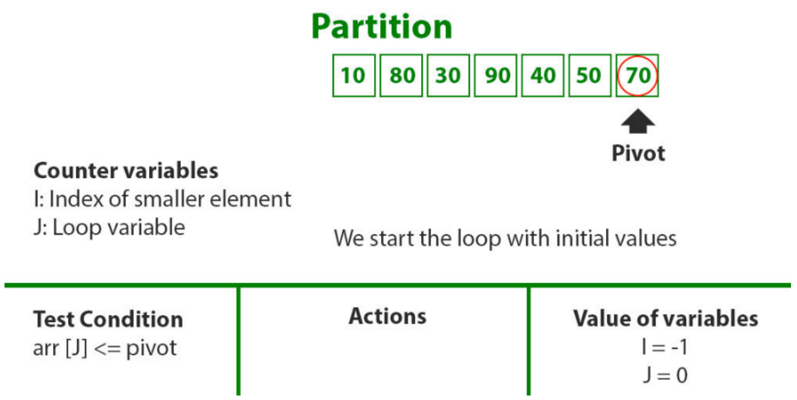
The Working Process of Merge Sort:

The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

<u>Illustration:</u>

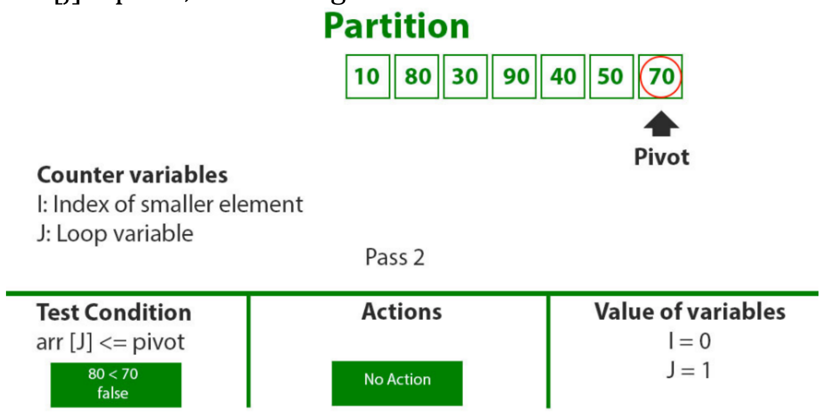
Consider: arr[] = {10, 80, 30, 90, 40, 50, 70}

- Indexes: 0 1 2 3 4 5 6
- low = 0, high = 6, pivot = arr[h] = 70
- Initialize index of smaller element, **i = -1**



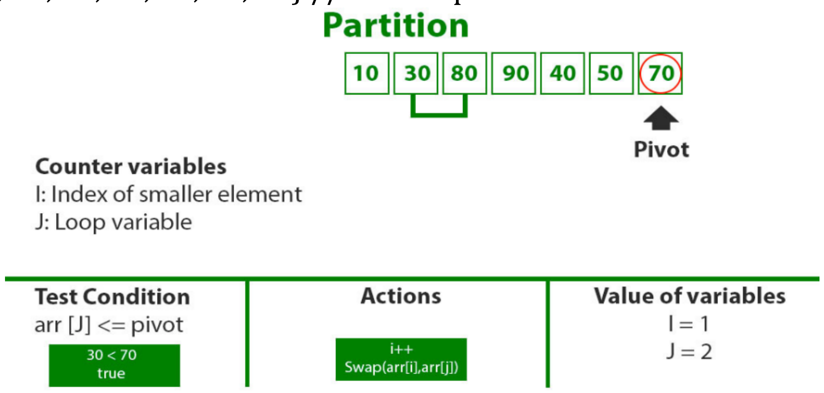
Compare pivot with first element

- Traverse elements from j = low to high-1
    - **j = 0**: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
    - **i = 0**
- arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same
- **j = 1**: Since arr[j] > pivot, do nothing



Compare pivot with arr[1]

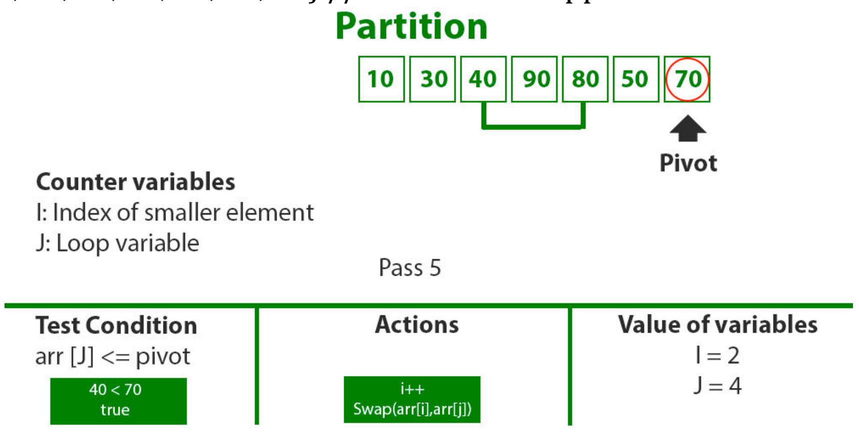
- **j = 2** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
- **i = 1**
- arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30



Compare pivot with arr[2]

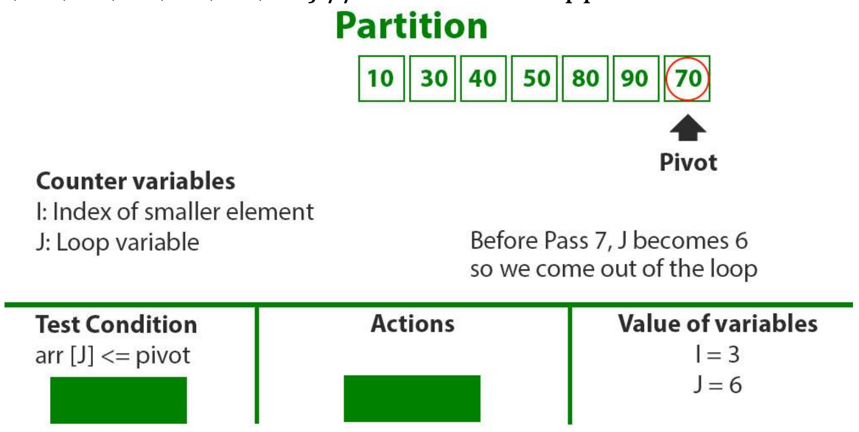
- **j = 3** : Since arr[j] > pivot, do nothing // No change in i and arr[]
- **j = 4** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
- **i = 2**

- arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped



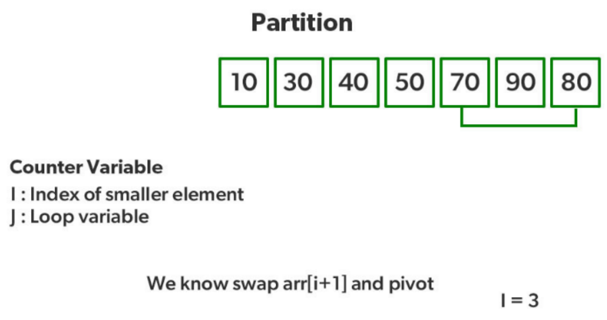
Compare pivot with arr[4]

- **j = 5** : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
- **i = 3**
- arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped



Compare pivot with arr[6]

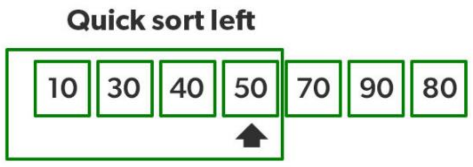
- We come out of loop because j is now equal to high-1.
- **Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)**
- arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped



Swap arr[i+1] with pivot

- Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

- Since quick sort is a recursive function, we call the partition function again at left and right partitions



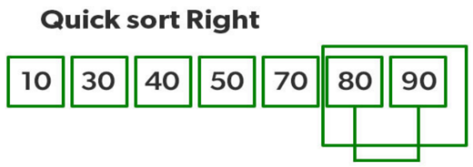
Since quick sort is a recursion function, wecall the Partition function again

First 50 is the pivot.

As it is already at its correct position we call the quicksort function again on the left part.

Recursively sort the left side of pivot

- Again call function at right part and swap 80 and 90

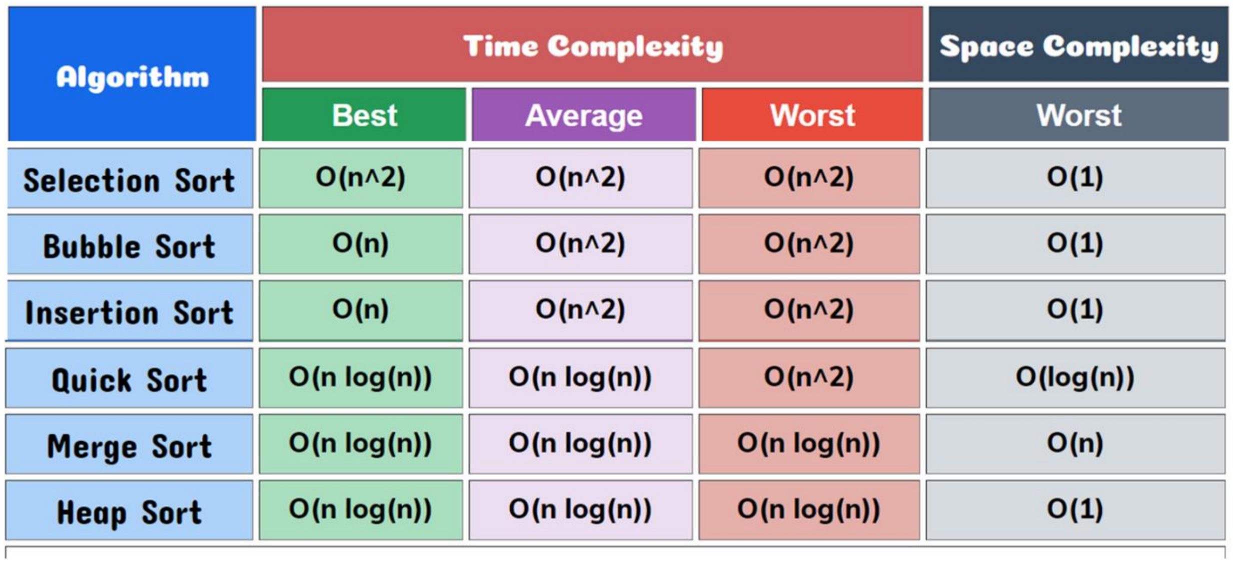


80 is the Pivot

80 and 90 are swapped to bring pivot to correct position

Recursively sort the right side of pivot.

Time & Space Complexity of Different Sorting Methods:



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Quick Sort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)

2.6 SEARCHING TECHNIQUES

The method of searching for a specific value in an array is known as searching. There are two ways we can search in an array, they are:
Linear search and Binary search

1. LINEAR SEARCH

Linear search in C to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends.

Program:

```
#include <stdio.h>
int main()
{
        int array[100], search, c, n;
        printf("Enter number of elements in array\n");
        scanf("%d", &n);
        printf("Enter %d integer(s)\n", n);
        for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
        printf("Enter a number to search\n");
        scanf("%d", &search);
        for (c = 0; c < n; c++)
        {
                if (array[c] == search)
                {
                        printf("%d is present at location %d\n", search, c+1);
                        break;
                }
         }
        if (c == n)
        printf("%d isn't present in the array\n", search);
        return 0;
}
```

2. BINARY SEARCH

The logic behind the binary search is that there is a key. This key holds the value to be searched. The highest and the lowest value are added and divided by 2. The lowest and highest values are the first and last elements in the array.

The mid value is then compared with the key. If mid is equal to the key, then we get the output directly. Else if the key is greater then mid then the mid+1 becomes the lowest value and the process is repeated on the shortened array. Else if the key value is less then mid, mid-1 becomes the highest value and the process is repeated on the shortened array. If it is not found anywhere, an error message is displayed.

Program:

```
#include <stdio.h>
int main()
```

```
{
        int i, low, high, mid, n, key, array[100];
        printf("Enter number of elements\n");
        scanf("%d",&n);
        printf("Enter %d integers\n", n);
        for(i = 0; i< n; i++)
        scanf("%d",&array[i]);
        printf("Enter value to search\n");
        scanf("%d", &key);
        low = 0;
        high = n - 1;
        mid = (low+high)/2;
        while (low <= high)
        {
                if(array[mid] < key)
                low = mid + 1;
                else if (array[mid] == key)
                {
                        printf("%d found at location %d\n", key, mid+1);
                        break;
                }
                else
                high = mid - 1;
                mid = (low + high)/2;
        }
        if(low > high)
        printf("Not found! %d isn't present in the list\n", key);
        return 0;
}
```