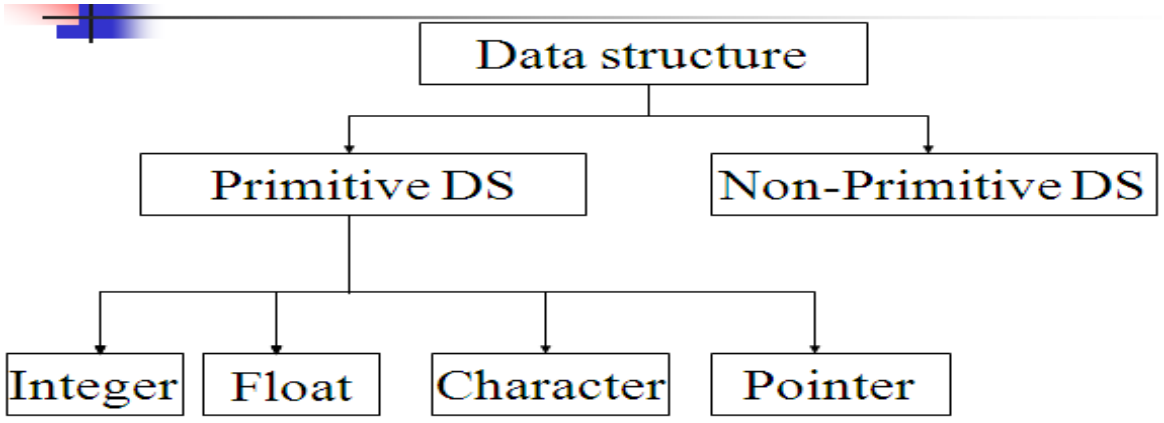
UNIT - I

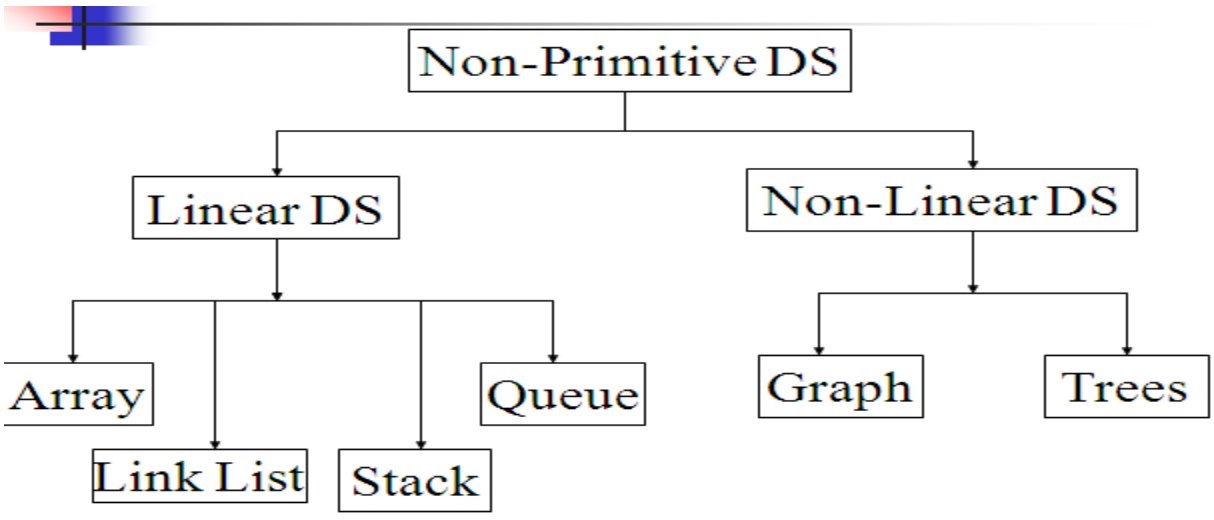
Introduction to Data Structures

- Data structure is collection of data elements and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.
- In other words we can store and retrieve data easily depends on the user application

Classification of Data Structure

- Data structure are normally divided into two broad categories:
    - Primitive Data Structure
    - Non-Primitive Data Structure





Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
- Linked List, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

Data Structure Operations

The most commonly used operation on data structure are broadly categorized into following types:

- Traversing
- Searching
- Inserting
- Deleting
- Sorting
- Merging

Traversing- It is used to access each data item exactly once so that it can be processed.

Searching- It is used to find out the location of the data item if it exists in the given collection of data items.

Inserting- It is used to add a new data item in the given collection of data items.

Deleting- It is used to delete an existing data item from the given collection of data items.

Sorting- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

Merging- It is used to combine the data items of two sorted files into single file in the sorted form.

Abstract Data Types

- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called "abstract" because it gives an implementation independent view.
- The process of providing only the essentials and hiding the details is known as abstraction. For example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it.

Stack ADT

A Stack contains elements of same type arranged in sequential order.

- All operations takes place at a single end that is top of the stack and following operations can be performed:
  push() – Insert an element at one end of the stack called top.
  pop () – Remove and return the element at the top of the stack, if it is not empty.

Queue ADT

A Queue contains elements of same type arranged in sequential order.

- Operations takes place at both ends, insertion is done at end and deletion is done at front. Following operations can be performed:
  enqueue() – Insert an element at the end of the queue.
  dequeue() – Remove and return the first element of queue, if the queue is not empty.

What is an algorithm?

An algorithm is a step by step procedure to solve a problem. In normal language, algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Different Approaches to Design an Algorithm

- Divide and conquer
- Greedy Method
- Dynamic Programming
- Backtracking
- Branch & Bound

Divide and Conquer Method

The divide and conquer approach is applied in the following algorithms

- Binary search
- Quick sort
- Merge sort

Dynamic Programming

Recursive algorithm for Fibonacci Series is an example of dynamic programming.

Greedy Method

A greedy algorithm works recursively creating a group of objects from the smallest possible component parts. Recursion is a procedure to solve a problem in which the solution to a specific problem is dependent on the solution of the smaller instance of that problem.

Branch and Bound

The purpose of a branch and bound search is to maintain the lowest-cost path to a target. Once a solution is found, it can keep improving the solution. Branch and bound search is implemented in depth-bounded search and depth–first search.

Backtracking

Can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

Recursive Algorithm

A **recursive algorithm** is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.

In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Factorial

Factorial is an important mathematical function. A recursive definition of factorial is as follows.

$$n! = \begin{cases} 1 & \text{if } n \leq 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

Linear Search Algorithm

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear Search Algorithm (Sequential Search Algorithm)

Linear search algorithm finds a given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing search element with the first element in the list. If both are matched then result is element found otherwise search element is compared with next element in the list. Repeat the same until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

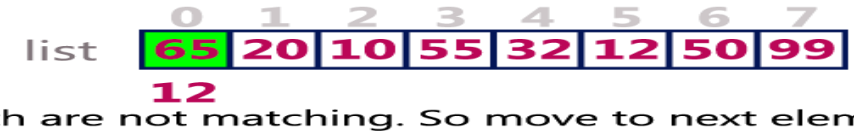
Linear search is implemented using following steps...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the first element in the list.
- **Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then compare search element with the next element in the list.
- **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

search element 12

Step 1:

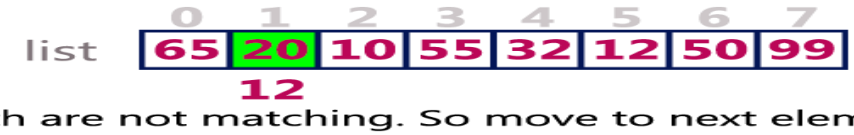
search element (12) is compared with first element (65)



Both are not matching. So move to next element

Step 2:

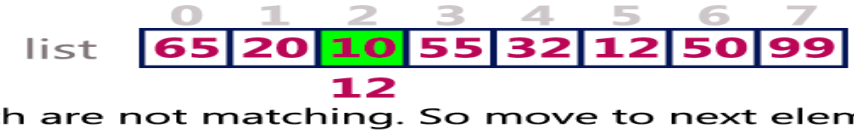
search element (12) is compared with next element (20)



Both are not matching. So move to next element

Step 3:

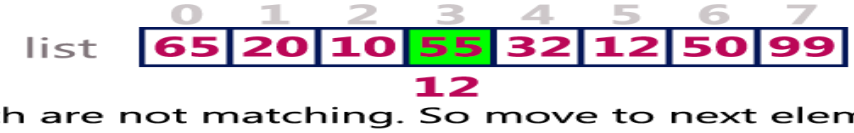
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Linear Search Program

//AIM: write a program to search the given element by using linear search and binary search

```
#include<stdio.h>
#include<conio.h>
void main()
 {
 int i,j,a[100],key,n,flag=0;
 clrscr();
 printf("Number of elements in the array\n");
 scanf("%d",&n);
 printf("\n Enter %d elements\n",n);
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 printf("Element to be searched\n");
 scanf("%d",&key);
  for(i=0;i<n;i++)
   {
    if(key==a[i])
     {
       flag=1;
       break;
      }
    }
    if(flag==1)
    printf("key  value found at %d index",i);
    else
    printf("key value not found");
    getch();
}
```

Binary Search Algorithm

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Binary Search Algorithm

Binary search algorithm finds a given element in a list of elements with **O(log n)** time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only sorted list of elements. That means, binary search is used only with list of elements that are already arraged in an order. The binary search can not be used for list of elements arranged in random order. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is " Element not found in the list".

Binary search is implemented using following steps...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Find the middle element in the sorted list.
- **Step 3 -** Compare the search element with the middle element in the sorted list.
- **Step 4 -** If both are matched, then display "Given element is found!!!" and terminate the function.
- **Step 5 -** If both are not matched, then check whether the search element is smaller or larger than the middle element.
- **Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7 -** If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8 -** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9 -** If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Example

Consider the following list of elements and the element to be searched...

Step 1:

search element (12) is compared with middle element (50)

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

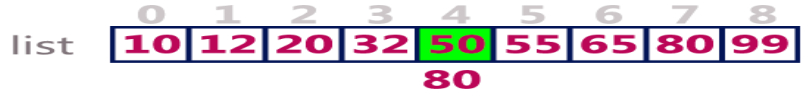
Step 2:

search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

Step 1:

search element (80) is compared with middle element (50)



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

Binary Search Program

//AIM: write a program to search the given element by using binary search

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[100],i,key,n,flag=0,high,mid,low=0;
clrscr();
printf("enter the size array:");
scanf("%d",&n);
printf("enter an element of array:");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("enter the key value:");
scanf("%d",&key);
high=n-1;
while(low<=high)
{
mid=(low+high)/2;
if(key==a[mid])
{
flag=1;
break;
}
else if(key<a[mid])
high=mid-1;
else if(key>a[mid])
low=mid+1;
}
if(flag==0)
printf("key value notfound:");
else
printf("key value found at %d index:",mid);
getch();
}
```

Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

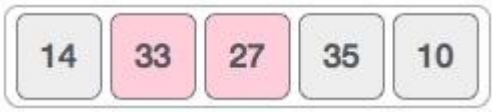
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

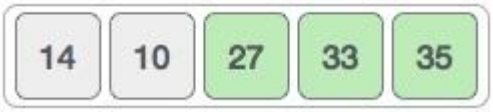
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

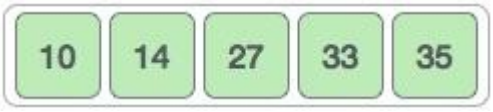
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Bubble Sort Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int a[100], n, i, j, s;
  clrscr();
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
    for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
    for (i= 0 ; i < ( n - 1 ); i++)
     {
       for (j = 0 ; j < n - i- 1; j++)
            {
              if (a[j] > a[j+1])
               {
                 s= a[j];
                 a[j]   = a[j+1];
                 a[j+1] = s;
               }
            }
     }

     printf("Sorted list in ascending order:\n");
     for ( i = 0 ; i < n ; i++ )
     printf("%d\n", a[i]);
     getch();
}
```

Selection Sort Algorithm

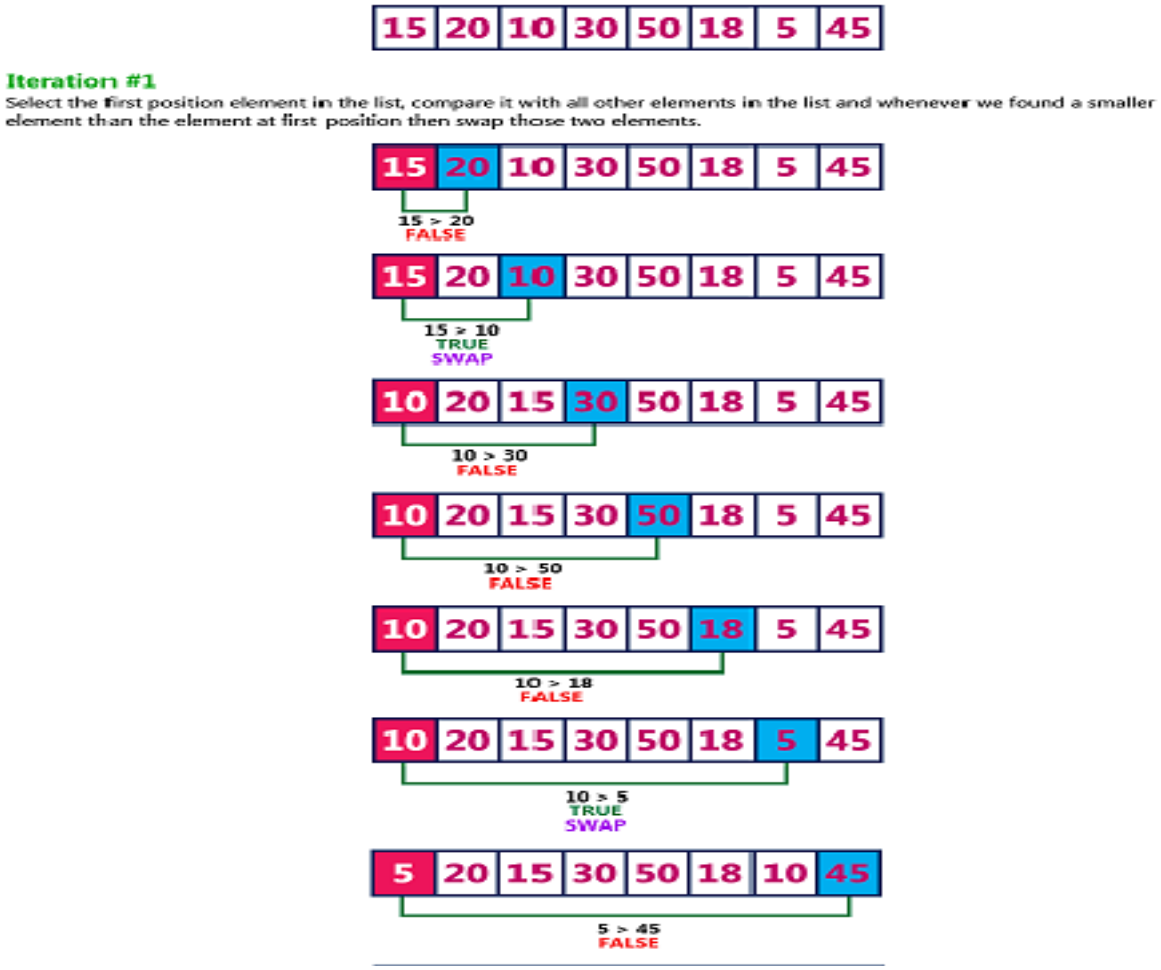
Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with smallest element in the sorted order. Next we select the element at second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using following steps...

- **Step 1 -** Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparison, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Example



Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration

Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration

Selection sort program

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,temp,n,a[20];
clrscr();
printf("Enter the number of elements:");
scanf("%d",&n);
printf("\nEnter the elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(i=0;i<n;i++)
for(j=i+1;j<n;j++)
if(a[i]>a[j])
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
printf("\nElements after sorting:");
for(i=0;i<n;i++)
printf("\n%d",a[i]);
getch();
}
```

Insertion Sort Algorithm

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

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

Step by Step Process

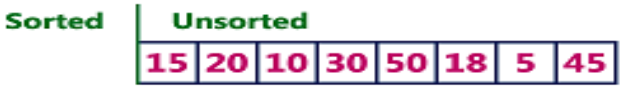
The insertion sort algorithm is performed using following steps...

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

EXAMPLE

Consider the following unsorted list of elements...

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



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

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

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

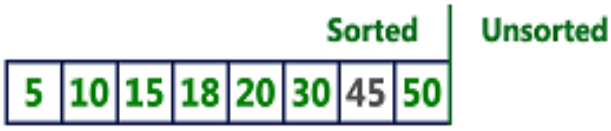
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

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

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

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

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



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

Insertion sort program

```
#include<stdio.h>
void main()
{
 int i,j,t,a[10],n,p=0;
clrscr();
printf("enter the range of array:");
scanf("%d",&n);
printf("enter elements into array:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(i=1;i<n;i++)
{
t=a[i];
for(p=i;p>0 && a[p-1]>t;p--)
a[p]=a[p-1];
a[p]=t;
}
printf("the sorted order is:");
for(i=0;i<n;i++)
printf("\t%d",a[i]);
getch();
}
```

Quick Sort

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.
There are many different versions of quick Sort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quick Sort is partition().

Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

All this should be done in linear time.

Illustration of partition() :

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**

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 // No change in i and arr[]
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

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

Quick sort program:

```
#include<stdio.h>
#include<conio.h>
void quicksort(int[],int,int);
void main()
{
 int x[20],i,n;
 clrscr();
 printf("Enter size of list:");
 scanf("%d",&n);
 printf("Enter %d elements:",n);
 for(i=0;i<n;i++)
 scanf("%d",&x[i]);
 quicksort(x,0,n);
 printf("Sorted List:");
 for(i=0;i<n;i++)
 printf("%d\t",x[i]);
 getch();
}
void quicksort(int x[20],int first,int last)
{
 int pivot,i,j,t;
 if(first<last)
 {
  pivot=first;
  i=first;
  j=last;
  while(i<j)
  {
   while(x[i]<=x[pivot] && i<last)
   i++;
   while(x[j]>x[pivot])
   j--;
   if(i<j)
   {
    t=x[i];
    x[i]=x[j];
    x[j]=t;
   }
  }
  t=x[pivot];
  x[pivot]=x[j];
  x[j]=t;
  quicksort(x,first,j-1);
  quicksort(x,j+1,last);
 }}
```

Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
The merge() function is used for merging two halves.
The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

   middle m = (l+r)/2

2. Call mergeSort for first half:

   Call mergeSort(arr, l, m)

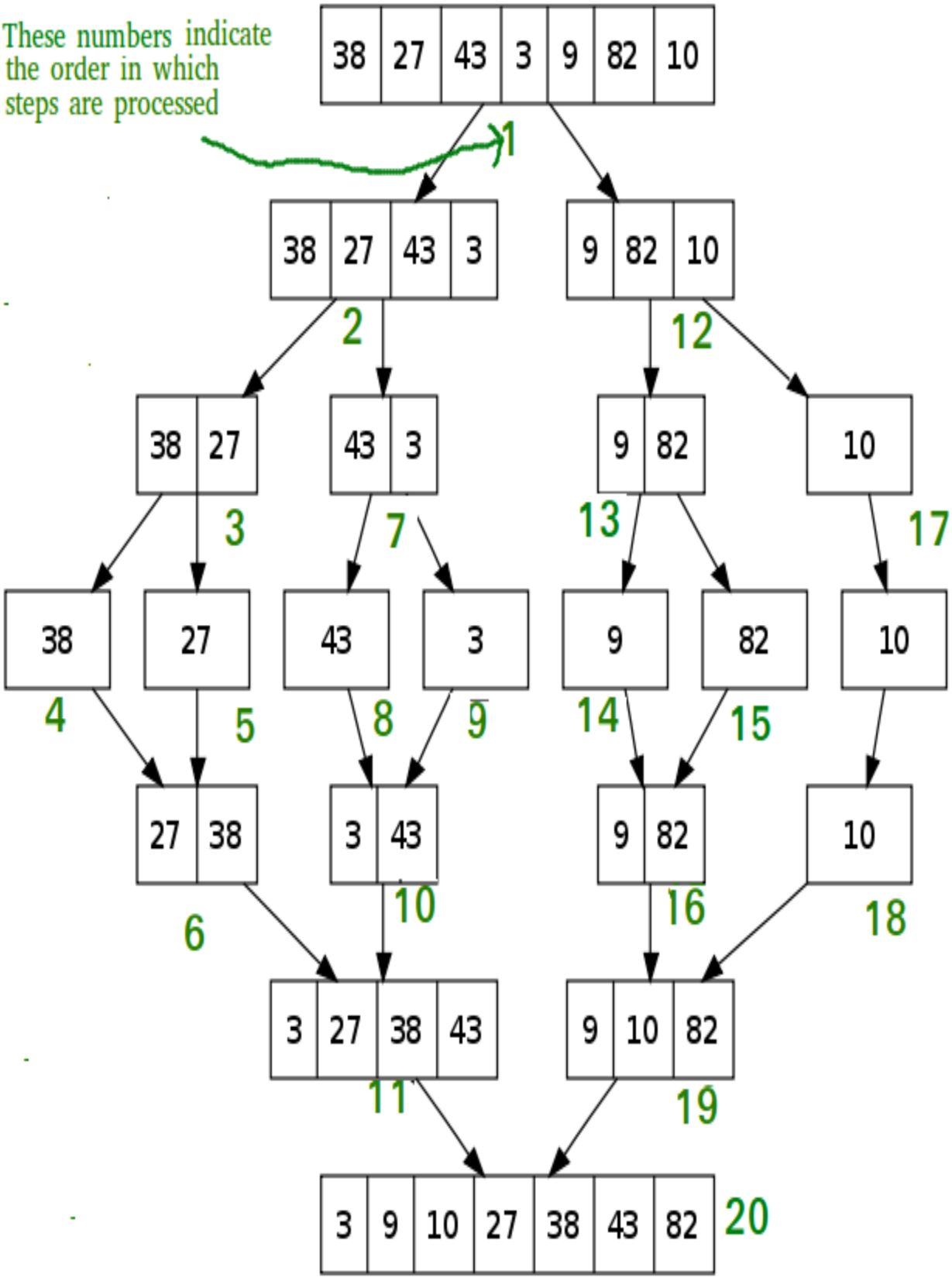
3. Call mergeSort for second half:

   Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

   Call merge(arr, l, m, r)

Example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Merge Sort program:

```
#include <stdio.h>
#include <stdlib.h>
void merge(int a[],int,int,int);
void mergesort(int a[],int,int);
void main()
{
 int a[20],i,n;
 clrscr();
 printf("Enter size of List:");
 scanf("%d",&n);
 printf("\nEnter %d elements: \n",n);
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 mergesort(a,0,n-1);
 printf("Sorted List:");
 for(i=0;i<n;i++)
 printf("%d\t",a[i]);
 getch();
}
void mergesort(int a[],int beg,int end)
{
 int mid;
 if(beg<end)
 {
  mid=(beg+end)/2;
  mergesort(a, beg, mid);
  mergesort(a, mid + 1, end);
  merge(a,beg,mid,end);
 }
}
void merge(int a[],int beg,int mid,int end)
{
 int i=beg,j = mid+1,index=beg,temp[20],k;
 while((i<=mid)&&(j<=end))
            {
  if(a[i]<a[j])
  {
   temp[index]=a[i];
   i++;
  }
  else
  {
   temp[index]=a[j];
   j++;
  }
```

```
 index++;
          }
if(i>mid)
{
 while(j<=end)
 {
  temp[index]=a[j];
  j++;
  index++;
 }
}
else
{
 while(i<=mid)
 {
  temp[index]=a[i];
  i++;
  index++;
 }
}
for(k=beg;k<index;k++)
a[k]=temp[k];
}
```

Comparison of Sorting Algorithms

Algorithm	**Data Structure**	**Time Complexity**		
		Best	**Average**	**Worst**
Quick sort	Array	O(n log(n))	O(n log(n))	O(n^2)
Merge sort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)