UNIT - I

Introduction to Data Structures: Definition and classification of data structures,Abstract data types (ADTs) and their implementation. Overview of time and space complexity analysis.
Searching - Linear search, Binary search, Fibonacci search.
Sorting- Bubble sort, Selection sort, Insertion Sort, Quick Sort and Merge Sort. Time complexity analysis for sorting techniques.

Definition:

- Data Structures is the concept of set of algorithms used to structure the information.
- These algorithms are implemented using C, C++, Java, etc
- Structure the information means store and process data in an efficient manner.
- To store and process data we may use the following operations

  1.create() 6.sorting()
  2.insert() 7.merging()
  3.delete() 8.splitting()
  4.display() 9.traversal()
  5.searching()
- So data structure may contain algorithms, use for different operations implement these algorithms by a programming language
- For example for stack data structure write algorithms for different operations

  1.push,
  2.pop and
  3.display.
- Implement these algorithms in a particular language say C.

Classification of Data Structures:

- Data structures are normally classified into two types.
- They are primitive data structures and non-primitive data structures.

(i)Primitive data structures:

- Primitive data structures are built in types in most programming languages. They are
- Integer: It is whole numbers. i.e. negative values,0,positive values
- Float: It is fractional numbers
- Character: It is character values
- Boolean: it represents true or false.

(ii)Non-primitive data structures:

- These are derived from primitive data structures.
- They are Array, Structure, Union, Files etc
- A Non-primitive data type is further divided into Linear and Non-Linear data structure.

(a)Linear data structures:

- Here the data elements are connected in a sequence manner.
- Examples are Arrays, Linked List, Stacks and Queues.

Array:

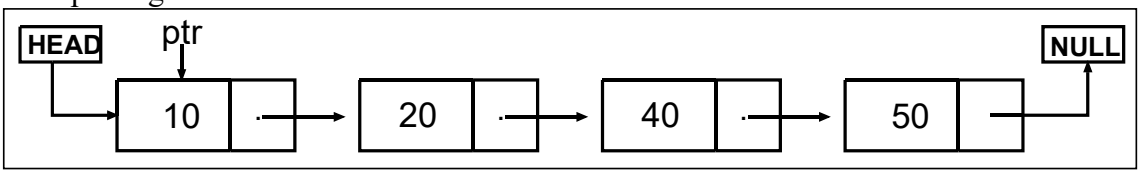
- It is collection of elements of the same type

	0	1	2	3	4
a[5]=	10	20	30	40	50

Linked List:

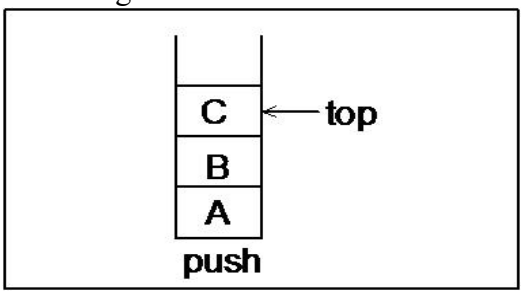
- *linked list* or single linked list is a sequence of elements in which every element has *link* to its next element in the sequence.
- Every element is called as a "node". Every "node" contains two fields, *data* and link. The data is a value or string and link is an address of next node.

- ❖ The first node is called HEAD which is an empty node contains an address of the first node so it link to the first node.
- ❖ The first node link to the second node and so on.
- ❖ The last node does not link to address but link to NULL. Let ptr be a pointer to the linked list. The example is given below



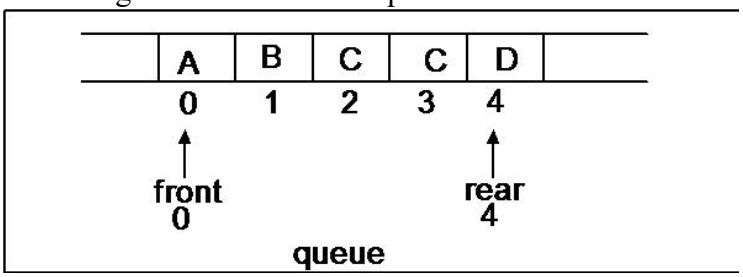
Stack:

- ❖ A stack is a data structure in which additions and deletions are made at the top of the stack. So we can perform two operations on stack.
  1. Adding elements into the stack known as push;
  2. Deleting elements from the stack known as pop



Queue:

- ❖ A queue is a data structure in which additions are made at one end and deletions are made at the other end. We can represent a queue in an array.
- ❖ Here we can perform two operations on queue.
  1. Adding elements into the queue known as insertion at rear
  2. Deleting elements from the queue known as deletion from front

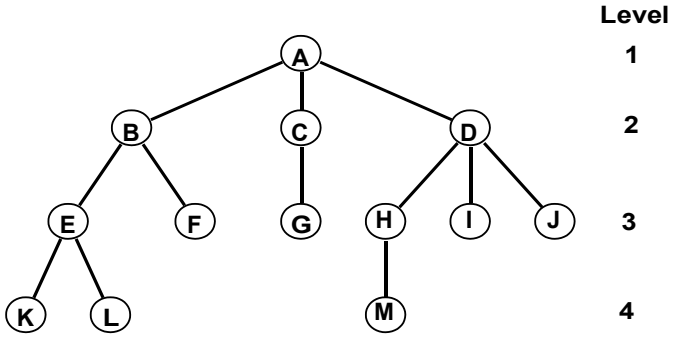


(b)Non-linear data structures:

- ➢ Here data elements are not connected in a sequence manner.
- ➢ Examples are: Trees and Graphs.

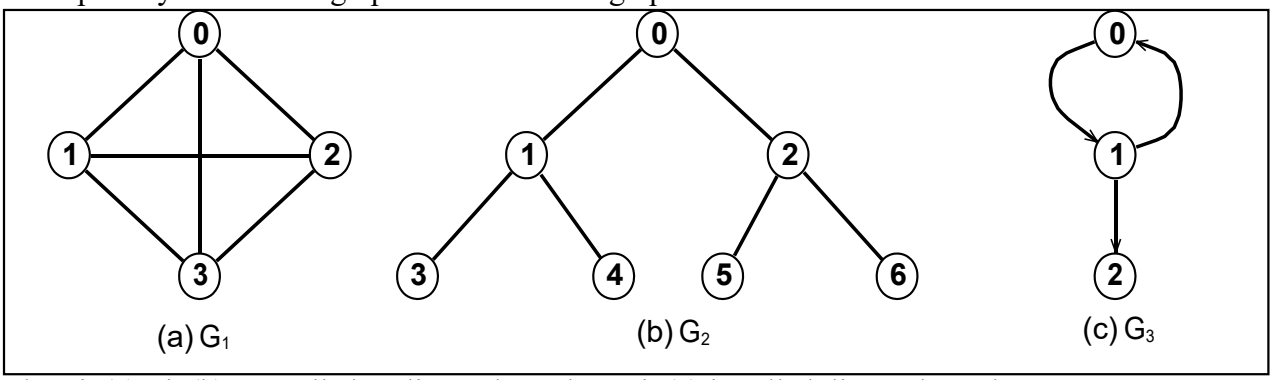
Tree:

- ➢ The tree is defined as a finite set of one or more nodes such that
  1. One node is called a root node and
  2. Remaining nodes partitioned into sub trees of the root.



Graph:

- A graph is a pictorial representation of a set of points or nodes termed as **vertices**, and the links that connect the vertices are called **edges**.
- A Graph(G) consists of two sets V and E where V is called vertices and E is called edges. We also write G = (V,E) to represent a graph.
- A Graph may be directed graph and undirected graph.



- The Fig(a),Fig(b) are called undirected graph & Fig(c) is called directed graph.

Differences between Linear and Non Linear Data Structures:

Linear Data Structure	Non-Linear Data Structure
Every data element is connected to its previous & next one	Every data element is connected with many other
Data is arranged in a sequence manner	Data is not arranged in a sequence manner
Data can be traversed in a single run	Data cannot be traversed in a single run
Ex: Array, Stack, Queue, Linked List	Ex: Tree, Graph
Implementation is easy	Implementation is difficult

Operations on Data Structures:

- The different operations used on data structure are:

1.Create:

- Here we reserve memory for program elements: This can be done by using malloc or calloc function. We can create a data structure with giving different elements.

2.Insert:

- Here we reserve memory for program element. This can be done by using malloc or calloc function. We can insert an element into a data structure.

3.Delete:

- It delete memory space allocated for specified data structure using free().

4.Display:

- It deals with accessing a particular data within a data structure.

5.Searching:

- It finds the data item in the list of data items.
- It also find the location of all elements.

6.Sorting:

- It is the process of arranging all data items in a data structure in a particular order say either in ascending order or in descending order.

7.Merging:

- It is the process of combining the data items of two different sorted list into a single sorted list.

8.Splitting:

- It is the process of partitioning single list to multiple list.

9.Traversal:

- It is the process of visiting each and every node of a list in systematic manner.

Abstract Data Type(ADT):

- An abstract data type (ADT) is a collection of values and a set of operations without specifying its implementation.
- For example in Array ADT set of values are index and item & set of operations are create(),retrieval() and store().
- The purpose of the ADT is to hide the implementation details of a data structure, thus improving software maintenance, reuse and portability.
- The developers of ADT will adapt changing requirements and save time.
- The users of ADT are concerned with the interface, but not the implementation.
- The different ADTs given by

  String ADT, List ADT, Stack (last-in, first-out) ADT,
  Queue (first-in, first-out) ADT
  Binary Search Tree ADT etc

Example:

- A List ADT contains operations known as add element, remove element, etc.
- A List ADT can be represented by an array-based implementation or a linked-list based implementation. In this the linked-list based implementation is so commonly used.
- Similarly, a Binary Search Tree ADT can be represented in different ways with the same operations known as insert, remove, display, etc.

(i)The Array as an Abstract Data Type:

- The Array ADT is a set of values (index, item) and a set of operations known as Array create(), Item Retrieve(), and Array Store().

The Array ADT algorithm is given by
Array ADT is
objects: A set of pairs <index,item> where for each index there is an item.
functions: for all A€Array, i€index, x€item
Array create() - It creates a new empty array
Item Retrieve(A,i) - It returns a value with a particular index, if the index is valid or an error if the index is invalid
Aray Store(A,i,x) - It stores an item

(ii)The Stack as an Abstract Data Type:

ADT stack is
objects: a finite ordered list with zero or more elements
functions: S € Stack, item € Element
Stack create() := create an empty stack
Stack push(S,item) := insert item into top of stack
Element pop(S) :=remove and return the item at the top of the stack

(iii)The Queue as an Abstract Data Type:

ADT queue is
objects: a finite ordered list with zero or more elements
functions: Q € Queue, item € Element Queue
Create() := create an empty stack Queue
addq(Q,item) := insert item into queue
Element deleteq(Q) :=remove and return the item from the queue

(iv)The Binary Tree as an Abstract Data Type:

ADT BinaryTree is
objects: a finite set of nodes
functions: for all bt, bt1, bt2 ϵ BinTree, item ϵElement
BinTree Create() := create an empty BinaryTree
Boolean IsEmpty := if(bt== empty binary tree) return TRUE else return FALSE

BinTree MakeBT(bt1,item,bt2) := return a binary tree whose left subtree is bt1, whose right subtree is bt2, and whose root node contains the data item.
BinTree Lchild(bt) := It return left subtree of bt
Element Data(bt) := It return the data in the root node of bt
BinTree Rchild(bt) := It return the right subtree of bt

(v)The Grapg as an Abstract Data Type:

ADT Graph is
objects: a set of vertices & edges
functions: for all graph ϵ Graph, v, v_1 and v_2 ϵ Vertices
Graph Create() := create an empty Graph
Boolean IsEmpty() := if(graph== empty graph) return TRUE else return FALSE
Graph InsertVertex(graph,v) := return a graph with v inserted.
Graph InsertEdge(graph,v_1,v_2) := return a graph with a new edge (v_1, v_2) is inserted.
Graph DeleteVertex(graph,v) := return a graph in which vertex v is Removed
Graph DeleteEdge(graph,v_1,v_2) := return a graph in which the edge (v_1,v_2) is removed

Time and Space complexity:

- Algorithm: Step by step process of solving a problem is called an algorithm.The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. There are two main complexity measures of the efficiency of an algorithm

(i)Time complexity:

- Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- Time means the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed or some other natural unit related to the amount of real time the algorithm will take.

(ii)Space complexity:

- Space complexity is a function describing the amount of memory an algorithm takes in terms of the amount of input to the algorithm. We often speak of extra memory needed, not counting the memory needed to store the input itself. We can use bytes, but it's easier to use say number of integers used, number of fixed-sized structures etc.
- In the end the function will be independent of the actual number of bytes needed to represent the unit. Space complexity is sometimes ignored because the space used is minimal.
- The complexity of an algorithm is studied with respect to the following 3 cases.

(a)Worst Case Analysis:

- In the worst case analysis, we calculate upper bound on running time of an algorithm.
- We must know the case that causes maximum number of operations to be executed.
- For Linear Search the worst case happens when the element to be searched is not present in the array.

(b)Average Case Analysis:

- In the average case analysis, we calculate average on running time of an algorithm. We must know the average number of operations to be executed.
- In the linear search problem, the average case occurs when x is present at average of its location.

(c)Best Case Analysis:

- In the best case analysis, we calculate lower bound on running time of an algorithm.
- We must know the case that causes minimum number of operations to be executed
- In the linear search problem, the best case occurs when x is present at the first location.

Searching

Searching:

- Searching is the process of finding a given value in a list of values.
- It is the algorithmic process of finding a particular item in a collection of items.

(i) Linear Search;

Definition:

- It starts at the beginning of the list and checks every element of the list.
- i.e. It sequentially checks each element of the list until a match is found or the whole list has been searched. So it is also called sequential search.

Example:

- Let the elements are: 10,6,3,8,9,12,14
- The search element is : 12
- Now it compare 12 with each and every element.
- The 12 is available in 6^{th} place.
- So the searching process is success and element is found

Algorithm:

Step 1: Read elements in array
Step 2: Read the element to search
Step 3: Compare the element to search and each element in array sequentially
Step 4: If match is found then the search success
Step 5: If match is not found upto the end then the search un success

Program:

```
#include <stdio.h>
int main()
{
	int a[100],n,i,s;
	printf("Enter Number of Elements in Array:\n");
	scanf("%d", &n);
	printf("Enter numbers:\n");
	for(i = 0; i < n; i++)
	scanf("%d",&a[i]);
	printf("Enter a number to search in Array:\n");
	scanf("%d", &s);
	for(i = 0; i < n; i++)
	{
		if(s==a[i])
		{
			printf("Number found\n”);
			break;
		}
	}
	if(i== n)
	printf("Number not found\n”);
	return 0;
}
```

(ii) Binary Search;

Definition:

- Binary search is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that are used to solve problems.
- Binary search implement on the records either in ascending or descending order to gain much better performance than linear search.
- Now suppose we have an ascending order record. At the time of search it takes the middle record/element, if the searching element is greater than middle element then the element mush be located in the second part

else it is in the first half. In this way this search algorithm divides the records in the two parts in each iteration and thus called binary search.

Example:

- Let the elements in ascending order are
  2 4 6 8 10 12 15
- Let the element to search 12
- For searching it compare first middle element.
  2 4 6 8 10 12 15
- The middle element is 8 and is not equal to 12. Since 12 is greater than 8 search on right side part of 8. 12 is equal to right side part middle. So element is found.

Algorithm:

Step 1: Read sorted elements in array
Step 2: Read the element to search
Step 3: Compare the element to search and middle element in array. If match is found the search success.
Step 4: If match is not found check the search element with middle element. If search element is greater than the middle element then search on right side of middle element otherwise search on left.
Step 5: This process is repeated for all elements in array. If no match is found upto the end then the search is not success.

Program:

```
#include <stdio.h>
int main()
{
	int i, first, last, middle, n, s, a[100];
	printf("Enter number of elements:\n");
	scanf("%d",&n);
	printf("Enter elements in ascending order:\n");
	for (i = 0; i < n; i++)
	scanf("%d",&a[i]);
	printf("Enter an element to search:\n");
	scanf("%d", &s);
	first = 0;
	last = n - 1;
	middle = (first+last)/2;
	while (first <= last)
	{
		if(s==a[middle])
		{
			printf("Element is found at index: %d",middle);
			break;
		}
		else if(s>a[middle])
		first = middle + 1;
		else if(s<a[middle])
		last = middle - 1;
		middle = (first + last)/2;
	}
	if (first > last)
	printf("Element is not found");
	return 0;
}
```

Sorting

Sorting:

- Sorting is a process of placing a list of elements from the collection of data in some order.
- It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier

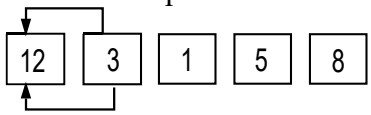
(i) Insertion Sort:

- In this sorting technique first elements are stored in an array.
- The process of sorting starts with second element.
- First the second element is picked and is placed in specified order Next third element is picked and is placed in specified order. Similarly the fourth, fifth, …n^{th} element .is placed in specified order.
- Finally we get the sorting elements.

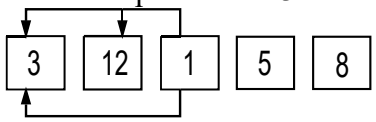
Example:

- Let us consider the elements: 12, 3, 1, 5, 8

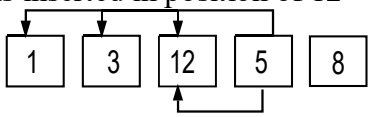
1. Checking second element of array with element before it and inserting it in proper position. In this case 3 is inserted in position of 12



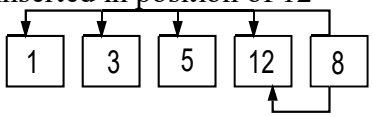
2. Checking third element of array with elements before it and inserting it in proper position. In this case 1 is inserted in position of 3



3. Checking fourth element of array with elements before it and inserting it in proper position. In this case 5 is inserted in position of 12



4. Checking fifth element of array with elements before it and inserting it in proper position. In this case 8 is inserted in position of 12



5. Sorted array in ascending order

Algorithm:

Step 1: Check second element of array with element before it and insert it in proper position.
Step 2: Checking third element of array with element before it and inserting it in proper position.
Step 3: Repeat this till all elements are checked.
Step 4: Stop

Program:

```
#include<stdio.h>
int main()
{
    int n,a[30],key,i,j,temp;
    printf("Enter total elements:\n");
    scanf("%d",&n);
    printf("Enter elements:\n");
    for(i=0;i<n;i++)
    scanf("%d",&a[i]);
    for(i=1;i<n;i++)
    {
        j=i;
```

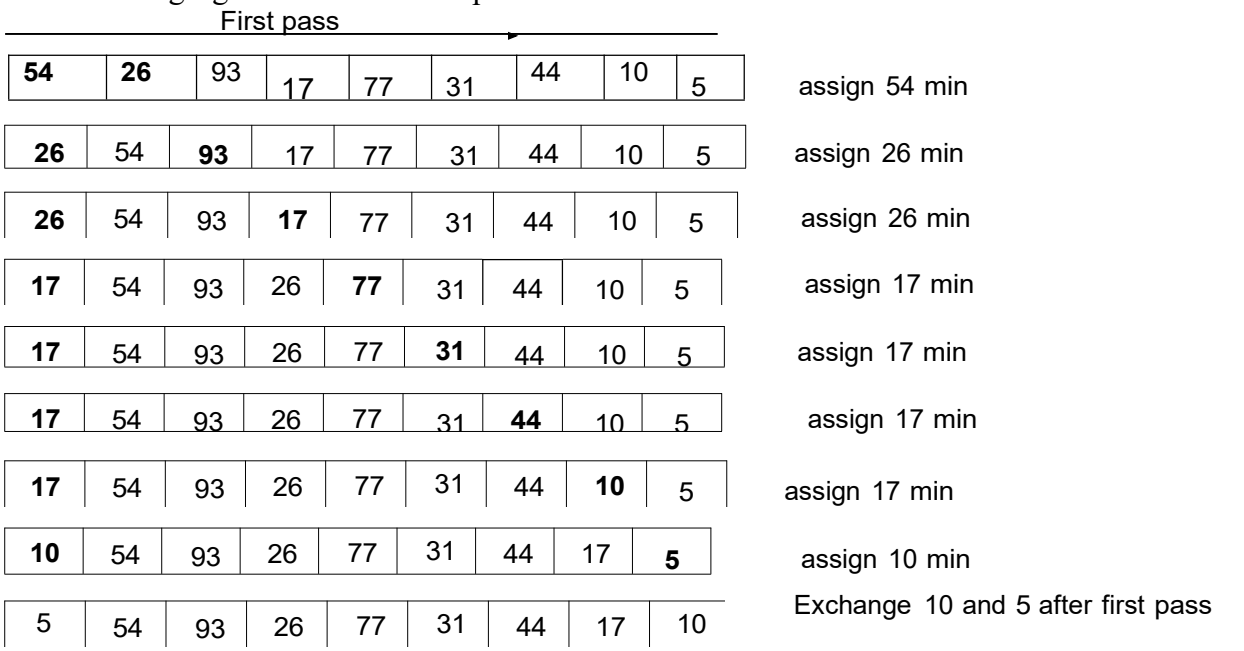
```
            while(j>0 && a[j]<a[j-1])
            {
                temp=a[j];
                a[j]=a[j-1];
                a[j-1]=temp;
                j--;
            }
        }
        printf("After sorting is:\n");
        for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

(ii) Selection Sort:

- Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Example:

- The following figure shows the first pass of a selection sort.



- In first pass the first element is compared with all remaining elements and exchange element if first one is greater than second so that the smallest value is in first place. Leave this element.
- In second pass compare second element to all elements and put the next smallest value, in second place. Leave this element. This process is repeated till all the elements are placed.
- Now we get the sorted elements.

Algorithm:

Step 1 − Set min to the first location.
Step 2 − Search the minimum element in the array.
Step 3 – swap the first location with the minimum value in the array.
Step 4 – assign the second element as min.
Step 5 − Repeat the process until we get a **sorted** array.

Program:

```
#include<stdio.h>
int main()
{
    int n,i,j,temp,a[20],min;
    printf("Enter total elements:\n");
    scanf("%d",&n);
```

```
    printf("Enter elements:\n");
    for(i=0;i<n;i++)
    scanf("%d",&a[i]);
    for(i=0;i<n;i++)
    {
        min=i;
        for(j=i+1;j<n;j++)
        {
            if(a[j] < a[min])
            min=j;
        }
        temp=a[i];
        a[i]=a[min];
        a[min]=temp;
    }
    printf("After sorting is:\n");
    for(i=0;i<n;i++)
    printf(" %d",a[i]);
    return 0;
}
```

(iii) Exchange Sort:

- The exchange sort is almost similar as the bubble sort. The exchange sort compares each element of an array and swap those elements that are not in their proper position,
  just like a bubble sort does. The only difference between the two sorting algorithms is the manner in which they compare the elements..

(i) Bubble Sort:

- Bubble Sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

Example:

- The following figure shows the first pass of a bubble sort. In first pass the first element is compared with second and exchange element if first one is greater than second.
- Similarly second element is compared with third and exchange element if second one is greater than third.
- Repeat this so that at the end of first pass the largest value is in last place. Leave this element.

First pass →

5	**2**	9	1	7	3	4	5	2	Exchange
2	**5**	**9**	1	7	3	4	5	2	No Exchange
2	5	**9**	**1**	7	3	4	5	2	Exchange
2	5	1	**9**	**7**	3	4	5	2	Exchange
2	5	1	7	**9**	**3**	4	5	2	Exchange
2	5	1	7	3	**9**	**4**	5	2	Exchange
2	5	1	7	3	4	**9**	**5**	2	Exchange
2	5	1	7	3	4	5	**9**	**2**	Exchange
2	5	1	7	3	4	5	2	9	9 is placed after first pass

- In second pass compare up to before last place value and put the next largest value, that before last place. Leave this element.

- This process is repeated till all the elements are placed. Now we get the sorted elements..

Algorithm:

Step 1: The first element is compared with second and exchange element if first one is greater than second
Step 2: Similarly second element is compared with third and exchange element if second one is greater than third
Step 3: Repeat this so that at the end the largest value is in last place
Step 4: Likewise sorting is repeated for all elements.

Program:

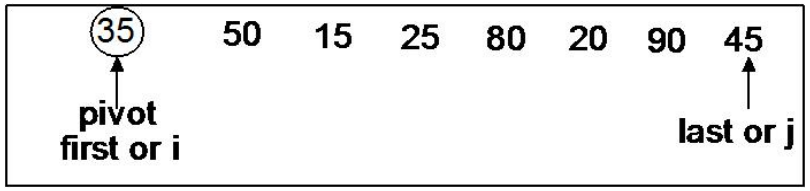
```
#include<stdio.h>
int main()
{
        int n,temp,i,j,a[20];
        printf("Enter total numbers of elements:\n");
        scanf("%d",&n);
        printf("Enter elements:\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        for(i=0;i<n;i++)
        {
              for(j=0;j<n-1;j++)
              {
                    if(a[j]>a[j+1])
                    {
                          temp=a[j];
                          a[j]=a[j+1];
                          a[j+1]=temp;
                    }
              }
        }
        printf("After sorting elements are:\n");
        for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

(ii) Quick Sort:

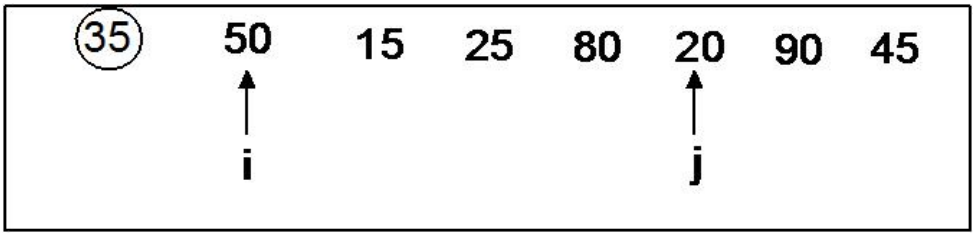
- Quick Sort is also one of the exchange sort.
- In a quick sort we take pivot element, then we place all the smaller elements are on one side of pivot, and greater elements are on other side of pivot.
- After partitioning we have pivot in the final position. After repeatedly partitioning, we get the sorted elements.

Example:

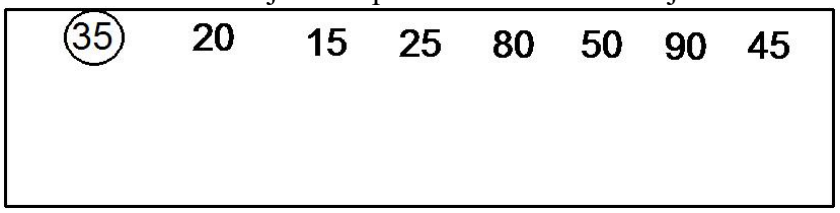
- Let us consider the elements:
  35,50,15,25,80,20,90,45
- Let us consider the first element 35 as pivot or i. The last element 45 as j



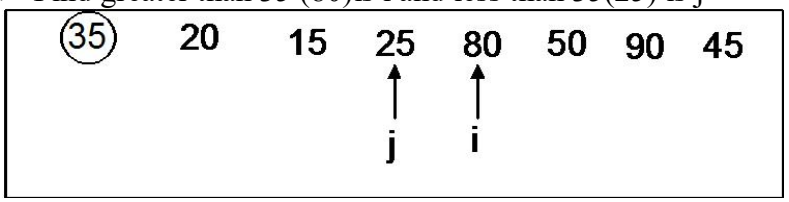
- 50 which is greater than pivot taken as i and the 20 smaller than pivot taken as j



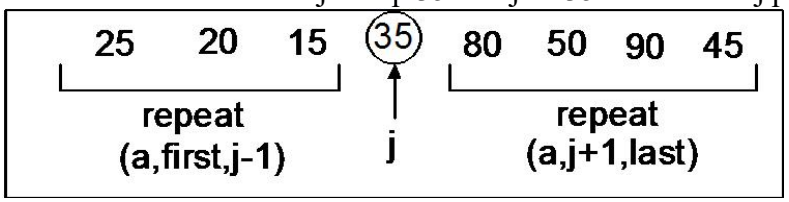
- Now i is less than j so swap the elements in i and j.



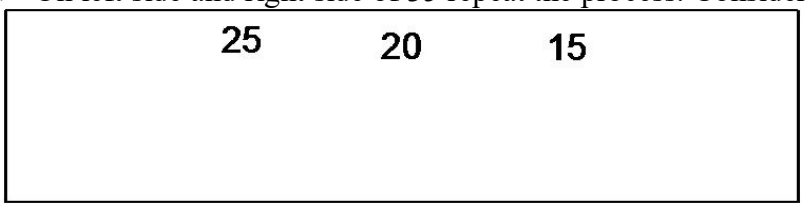
- Find greater than 35 (80)is i and less than 35(25) is j



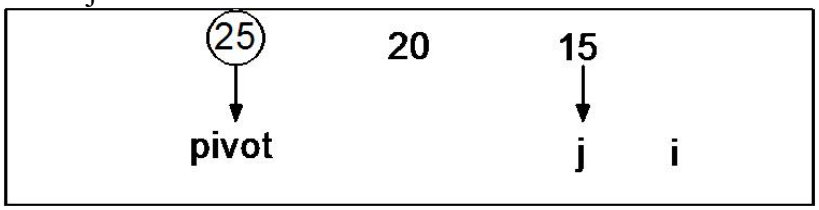
- Now i is not less than j. swap 35 and j so 35 becomes at j place.



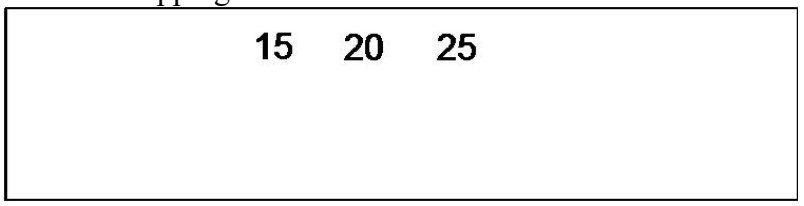
- Now 35 is in correct position.
- On left side and right side of 35 repeat the process. Consider on left side of 35



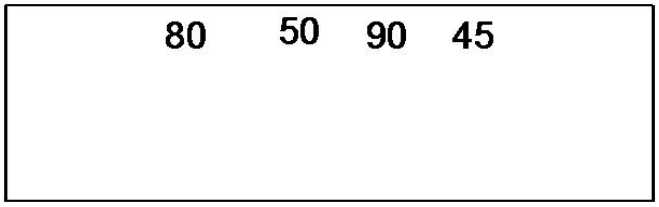
- Let 25 as pivot. The lesser of 25 that is 15 as j and there is no greater. So bring i after j i > j so swap pivot and j



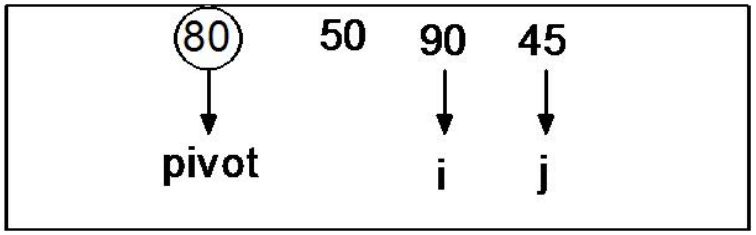
- After swapping



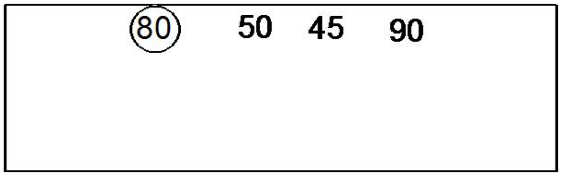
- Now the left part is sorted. Consider right part



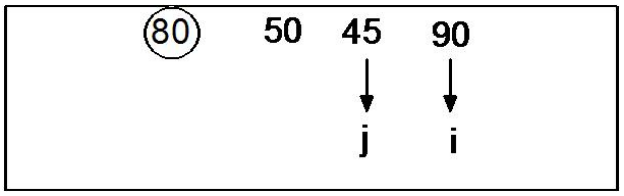
- Here 80 as pivot, Greater than to 80 is i and less than to 80 is j



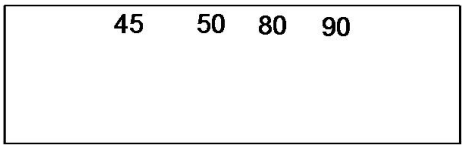
- Here i is less than j so swap i and j elements.



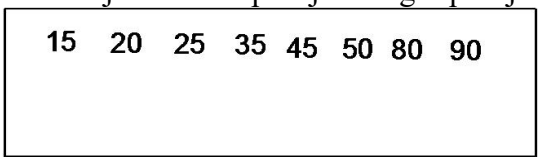
- first find greater to 80 is i and lesser to 80 is j. i > j so swap 80 and j.



- After swapping. The sorting elements are given by



- Now join all left part j and right part j to get the sorted elements



Algorithm:

Step 1: Let the first element taken as pivot
Step 2: Find lesser of pivot say i and greater of pivot say j.
Step 3: If i is less than j then i and j elements are swapped. Repeat step 2
Step 4: Repeat step 3 until i > j
Now swap j and pivot
Step 5: Now the pivot element is final position.
Repeat the above procedure for left and right side of pivot elements until all elements are sorted
Step 6: Stop

Program:

```
#include<stdio.h>
void quicksort(int a[25],int first,int last)
```

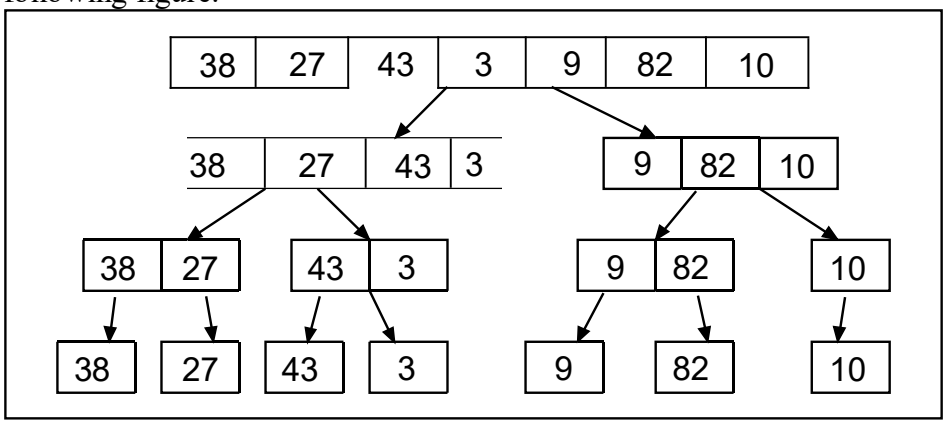
```
{
        int i, j, pivot, temp;
       if(first<last)
        {
                pivot=first;
                i=first;
                j=last;
                while(i<j)
                {
                        while(a[i]<a[pivot]&&i<=last)
                        i++;
                        while(a[j]>a[pivot])
                        j--;
                      if(i<j)
                        {
                                temp=a[i];
                                a[i]=a[j];
                                a[j]=temp;
                        }
                }
                temp=a[pivot];
                a[pivot]=a[j];
                a[j]=temp;
                quicksort(a,first,j-1);
                quicksort(a,j+1,last);
        }
}
int main()
{
        int i, n, a[25];
        printf("Enter total a of elements:\n ");
        scanf("%d",&n);
        printf("Enter elements:\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        quicksort(a,0,n-1);
        printf("The Sorted elements are:\n ");
       for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

(v) Merging or Merge Sort:

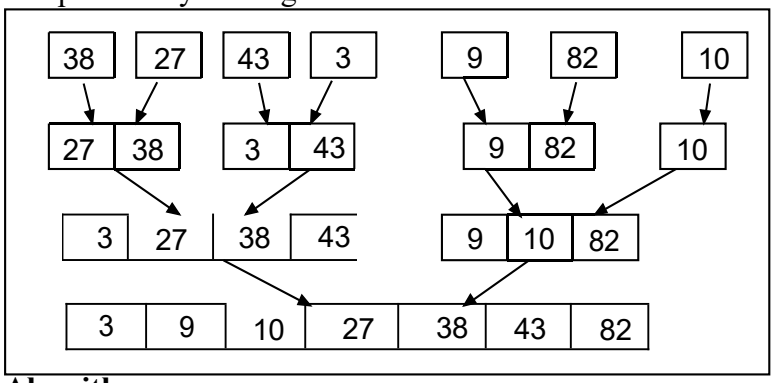
- It divides input array into two halves, calls itself for the two halves and then sorted and merged that two halves.

Example:

- For example consider the array of elements: 38, 27, 43, 3, 9, 82, 10
- Now the array is recursively divided into two halves till the size becomes one which is shown in the following figure.



- Once the size becomes one, the merge process comes into action and starts merging with sorted array till the complete array is merged

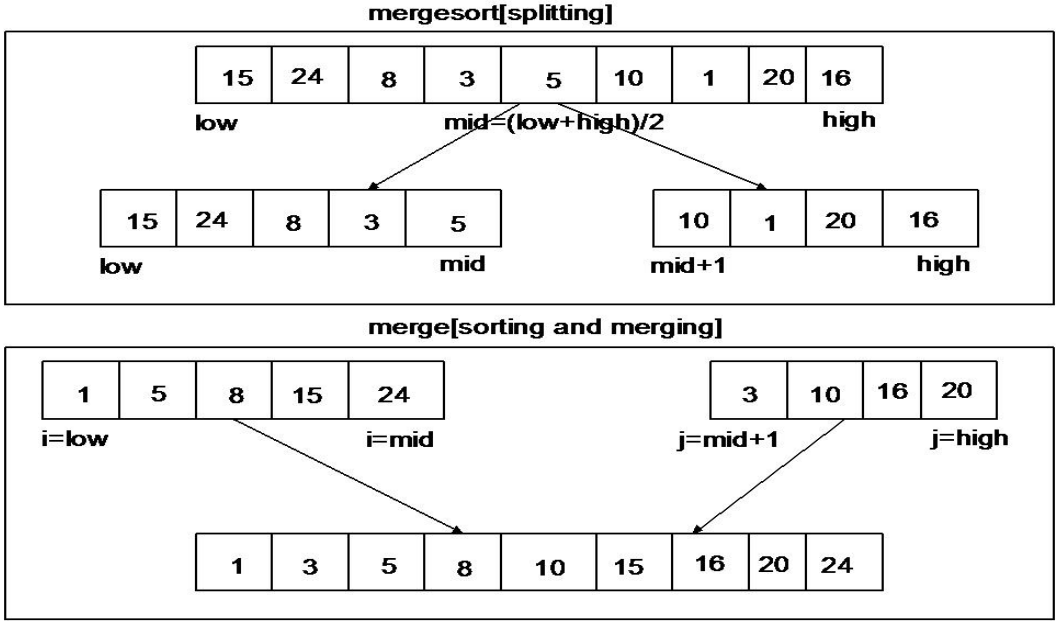


Algorithm:

Step 1 − If it is only one element in the list then it is already sorted.

Step 2 − Divide the list recursively into two halves till the size becomes one.

Step 3 − Once the size becomes 1, the merge process comes into action and starts merging with sorted array till the complete array is merged



Program:

```
#include<stdio.h>
int n,a[30],i,j,k,temp[30];
void merge(int low,int mid,int high)
{
        i=low;
        j=mid+1;
        k=low;
        while((i<=mid) && (j<=high))
        {
                if(a[i]>=a[j])
                temp[k++]=a[j++];
                else
                temp[k++]=a[i++];
        }
        while(i<=mid)
        temp[k++]=a[i++];
        while(j<=high)
        temp[k++]=a[j++];
        for(i=low;i<=high;i++)
        a[i]=temp[i];
}
 void mergesort(int low,int high)
 {
        int mid;
        if(low!=high)
        {
                mid=((low+high)/2);
                mergesort(low,mid);
                mergesort(mid+1,high);
                merge(low,mid,high);
       }
}
int main()
{
        printf("Enter total elements:\n");
        scanf("%d",&n);
        printf("Enter elements:\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        mergesort(0,n-1);
        printf("After sorting is:\n");
        for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

Time Complexity:

- **O(n^2)** means that for every insert, it takes n*n operations. i.e. 1 operation for 1 item, 4 operations for 2 items, 9 operations for 3 items.

Comparison of Sorting Algorithms

Algorithm	Data Structure	Time Complexity		
		Best	Average	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	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)