UNIT - IV

Graphs-Terminology, sequential and linked representation, graph traversals : Depth First Search & Breadth First Search implementation. Spanning trees, Prim's and Kruskal's method.

Searching and Sorting - Linear Search, Binary Search, Insertion Sort, Selection Sort, Merge Sort, Quick sort, Heap Sort.

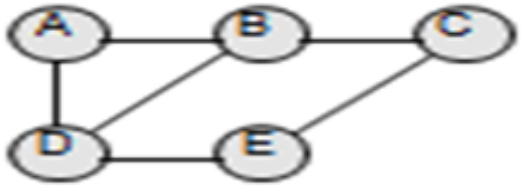
Graph:

- A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.

The figure shows a graph with

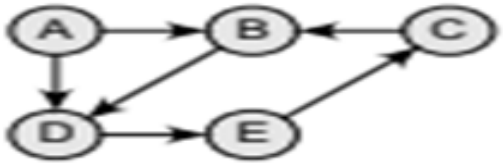
$V(G) = \{A, B, C, D \text{ and } E\}$

and $E(G) = \{(A,B), (B,C), (A,D), (B,D), (D,E), (C,E)\}$.



Undirected Graph

- A graph can be directed or undirected.
- In an **undirected graph**, edges do not have any direction associated with them. That is, if an edge is drawn between nodes then the nodes can be traversed from A to B as well as from B to A.

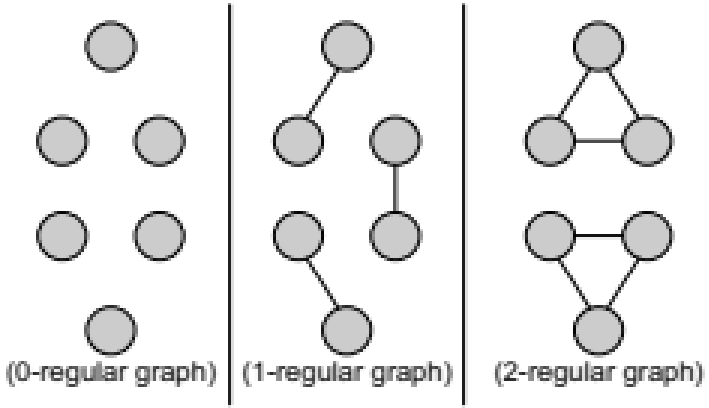


Directed Graph

- In a **directed graph** , edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A.

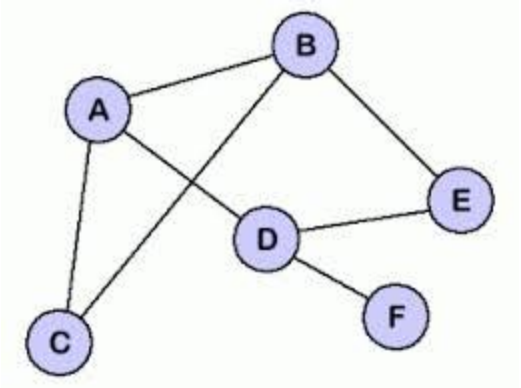
Graph Terminology

- **Degree of a node:** Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.
- **Regular graph :**It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree k is called a k–regular graph or a regular graph of degree k.



Regular graphs

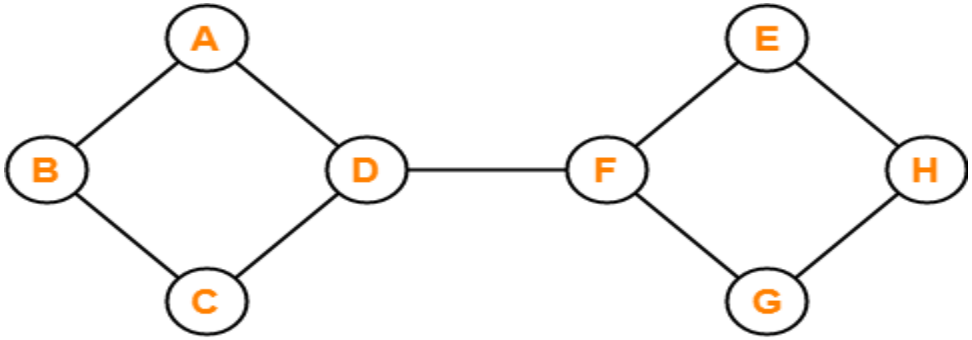
- **Path:** A path P written as $P = \{v_0, v_1, v_2, ..., v_n\}$, of length n from a node u to v is defined as a sequence of (n+1) nodes.



- **Closed path:** A path P is known as a closed path if the edge has the same end-points. That is, if $v_0 = v_n$
- **Simple path** A path P is known as a simple path if all the nodes in the path are distinct with an exception that v_0 may be equal to v_n. If $v_0 = v_n$, then the path is called a closed simple path.
- **Cycle** A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).

Graph Terminology…

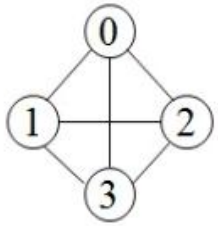
➢**Connected graph:** A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph.



Example of Connected Graph

➢**Complete graph:** A graph G is said to be complete if all its nodes are fully connected.

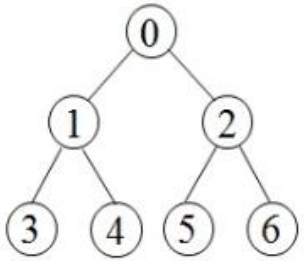
That is, there is a path from one node to every other node in the graph. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.



G_1

complete graph

No.of edges = n(n-1)/2
= 4*3/2
= 12/2
= 6



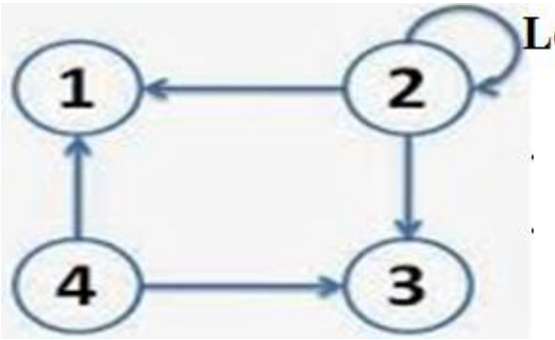
G_2

incomplete graph

No.of edges = n(n-1)/2
= 7*6/2
= 42/2
= 21

Graph Terminology…

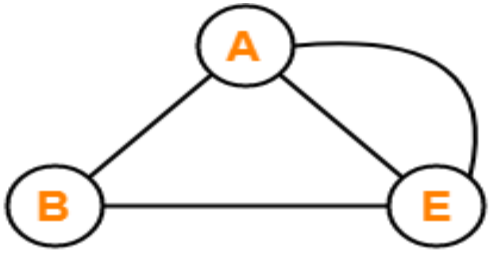
➢**Loop:** An edge that has identical end-points is called a loop. That is, e = (u, u). Example: e=(2,2)



➢**Size of a graph :** The size of a graph is the total number of edges in it.

➢**Multiple edges:** Distinct edges which connect the same end-points are called multiple edges. That is,
e = (u, v) and e' = (u, v) are known as
multiple edges of G.

➢**Multi-graph:** A graph with multiple edges and/or loops is called a multi-graph.



Example of Multi Graph

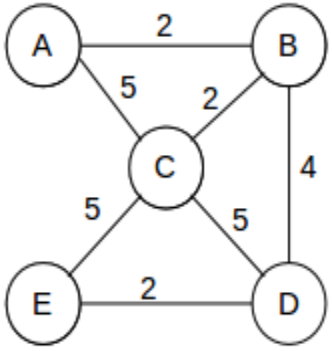
➢**Labelled graph or weighted graph:**

A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length.

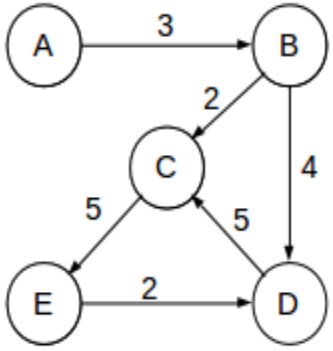
Two types of weighted graphs are:

i) Undirected Weighted Graph

ii) Directed Weighted Graph



Undirected Weighted Graph



Directed Weighted Graph

Graph Terminology…

➢**Out-degree of a node** :The out-degree of a node u, written as outdeg(u), is the number of edges that originate at u.

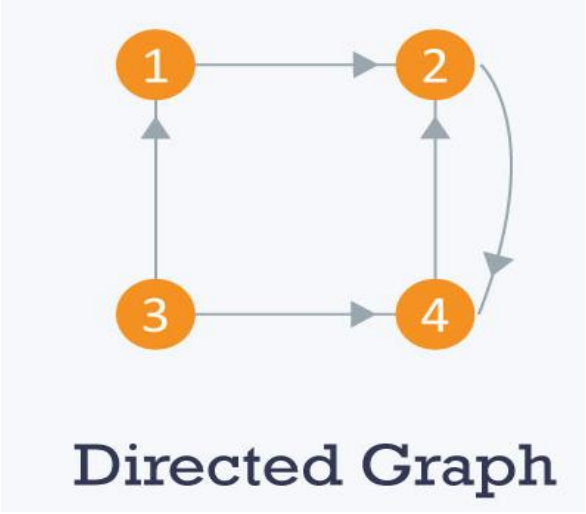
➢**In-degree of a node:** The in-degree of a node u, written as indeg(u), is the number of edges that terminate at u.

➢**Degree of a node** :The degree of a node, written as deg(u), is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg (u).

➢**Directed Graphs :**

A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e.
- v is known as the destination or terminal point of e.
- u is the predecessor of v.
- v is the successor of u.
- Nodes u and v are adjacent to each other.



Graph Representations

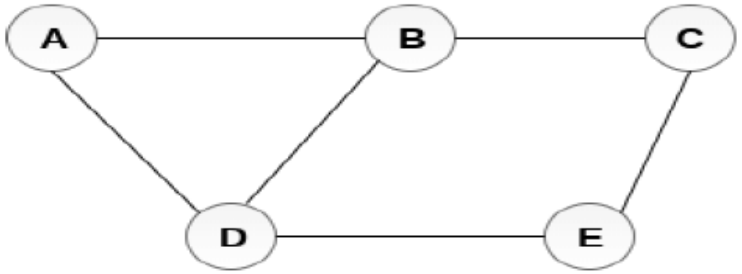
Graphs are represented in two ways:

i) Adjacency Matrix

ii) Adjacency List

Adjacency Matrix Representation:

- Adjacency matrix representation is also called as sequential representation.
- In adjacency matrix, the rows and columns are represented by the graph vertices.
- A graph having n vertices, adjacency matrix will have a dimension n x n.
- An entry Mij in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between Vi and Vj otherwise 0.



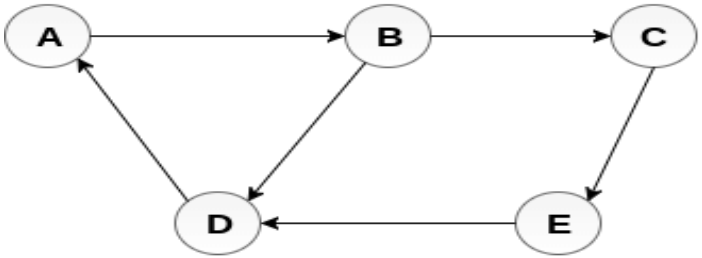
Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

Adjacency Matrix Representations...

In directed graph, an entry Mij will be 1 only when there is an edge directed from Vi to Vj otherwise 0.

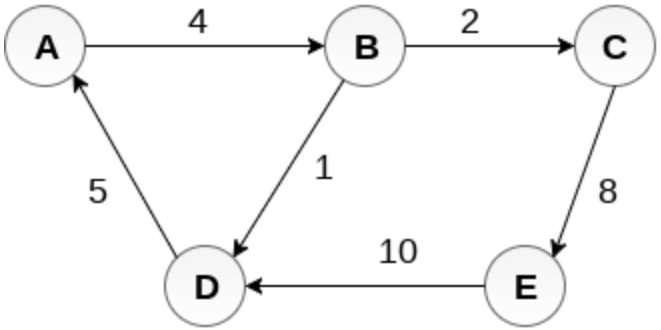


Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

If the graph is a weighted ,then Non- zero entries of the adjacency matrix are represented by the weight of respective edges.



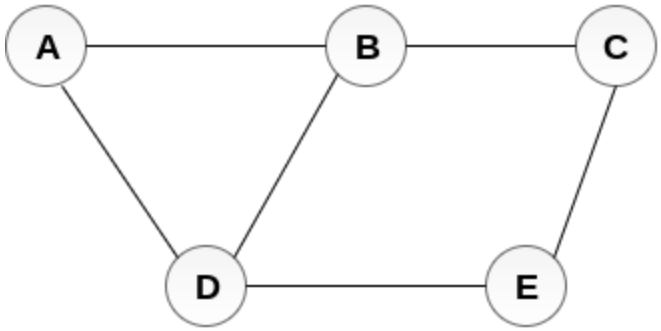
Weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

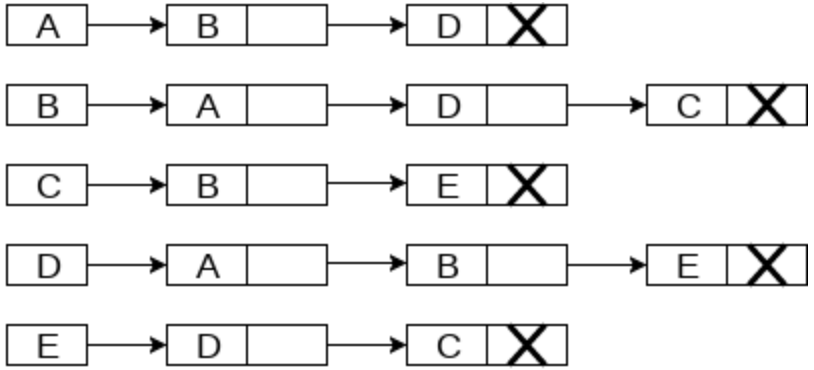
Adjacency Matrix

Adjacency List Representation

- An adjacency list is another way in which graphs can be represented in the computer's memory.
- An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list.
- For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G.
- For an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u.

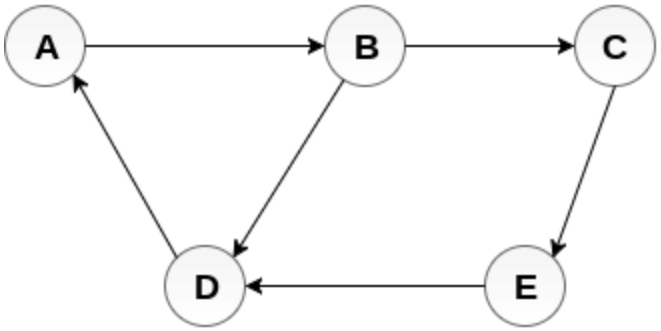


Undirected Graph

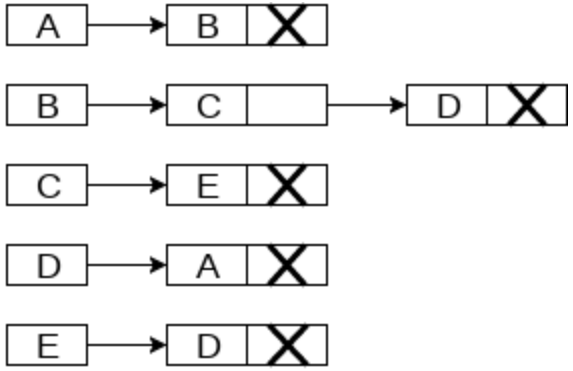


Adjacency List

Adjacency List Representation...

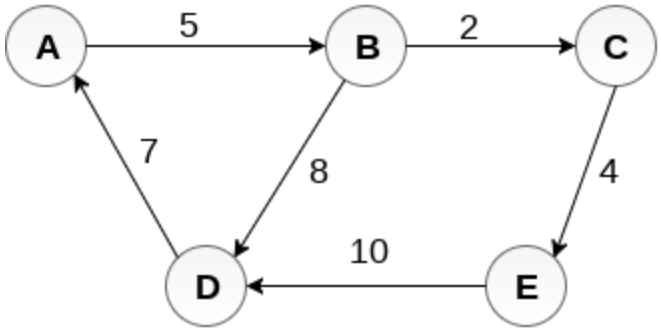


Directed Graph

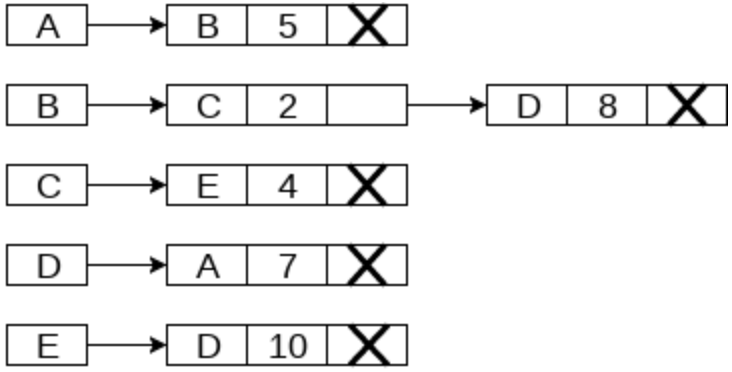


Adjacency List

- Adjacency lists can also be modified to store weighted graphs.
- In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



Weighted Directed Graph



Adjacency List

Graph Traversals

There are two standard methods of graph traversal:

1. Breadth-first search
2. Depth-first search

Value of status and its significance :

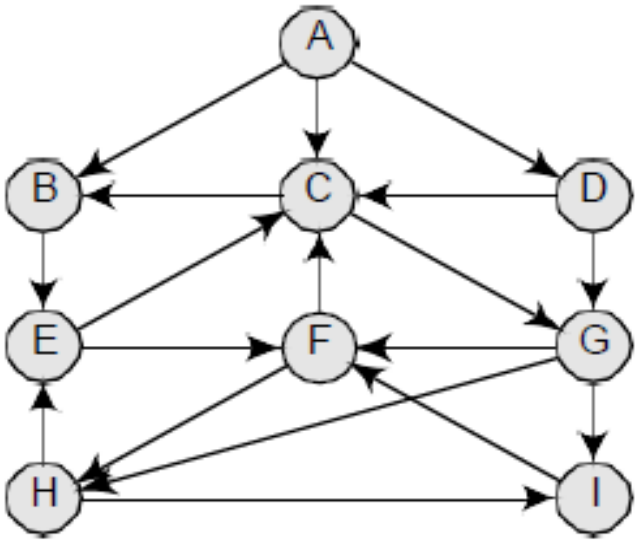
Status	State of the node	Description
1	Ready	The initial state of the node N
2	Waiting	Node N is placed on the queue or stack and waiting to be processed
3	Processed	Node N has been completely processed

Breadth-First Search :

- Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.
- BFS uses the Queue data structure that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

Breadth-First Search…

Example:



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until QUEUE is empty
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT

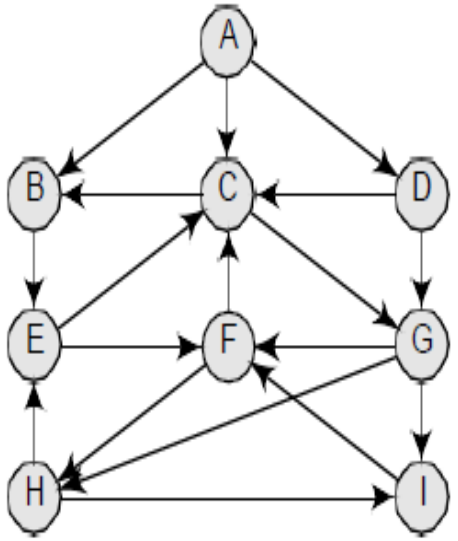
Breadth-First Search…

During the execution of the algorithm, we use two arrays:

- QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge.
- Initially, FRONT = REAR = –1. The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG \0



(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1	QUEUE = A	B	C	D
REAR = 3	ORIG \0	A	A	A

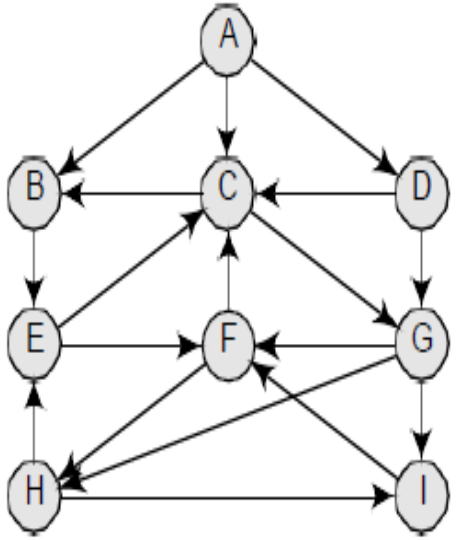
(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2	QUEUE = A	B	C	D	E
REAR = 4	ORIG \0	A	A	A	B

Breadth-First Search…

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG \0	A	A	A	B	C



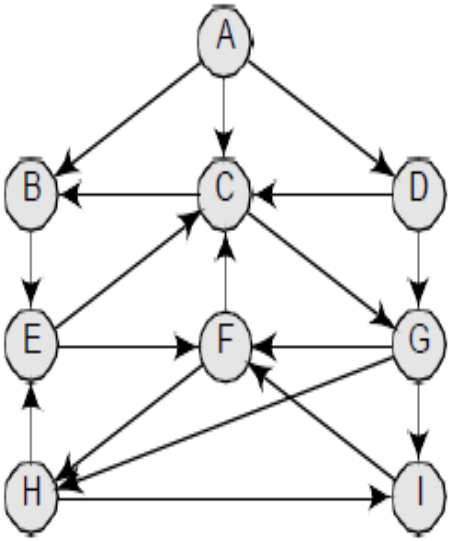
(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours . Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4	QUEUE = A	B	C	D	E	G
REAR = 5	ORIG \0	A	A	A	B	C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

Breadth-First Search…

FRONT = 5	QUEUE = A	B	C	D	E	G	F
REAR = 6	ORIG \0	A	A	A	B	C	E



(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6	QUEUE = A	B	C	D	E	G	F	H	I
REAR = 9	ORIG \0	A	A	A	B	C	E	G	G

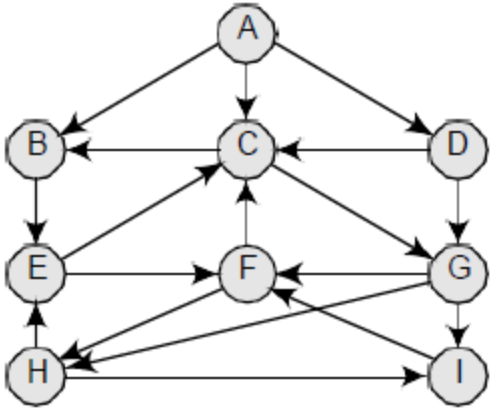
Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as **A -> C -> G -> I.**

Depth-first Search

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered.
- When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- Depth first search uses stack data structure.

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Depth-first Search…



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H **STACK: E, I**

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I **STACK: E, F**

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F **STACK: E, C**

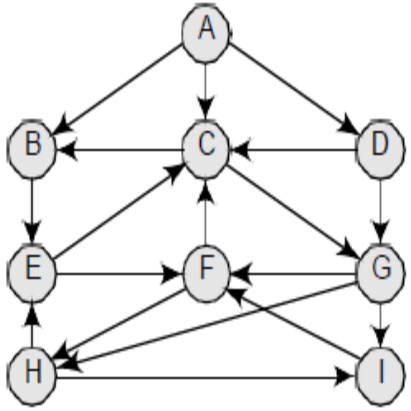
(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C **STACK: E, B, G**

Depth-first Search...

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed.
The STACK now becomes

PRINT: G **STACK: E, B**



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

(g)Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B **STACK: E**

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E **STACK**:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are:

H, I, F, C, G, B, E

These are the nodes which are reachable from the node H.

Applications of BFS &DFS

Applications of Breadth-First Search :

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

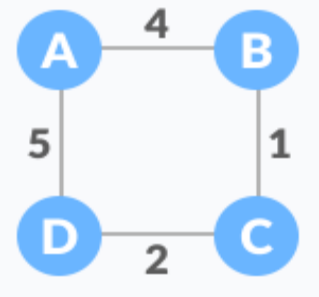
Applications of Depth-First Search Algorithm

Depth-first search is useful for:

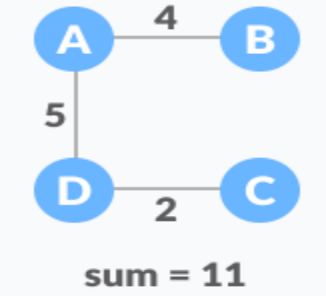
- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

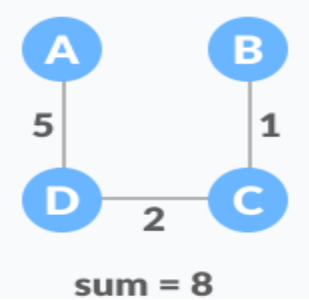
Minimum Spanning Tree

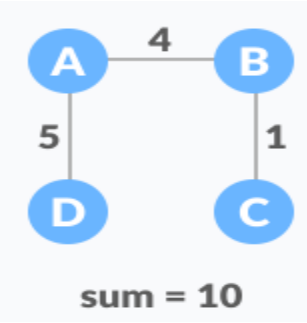
- A **spanning tree** is a subset of Graph G, which has all the vertices covered with minimum possible number of edges.
- It does not have cycles and it cannot be disconnected.
- The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n^{(n-2)}$.

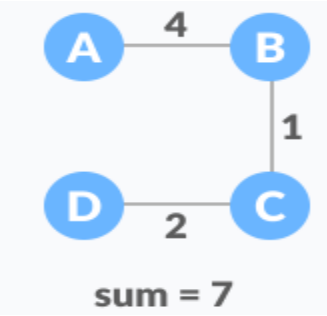


Possible Spanning trees are:





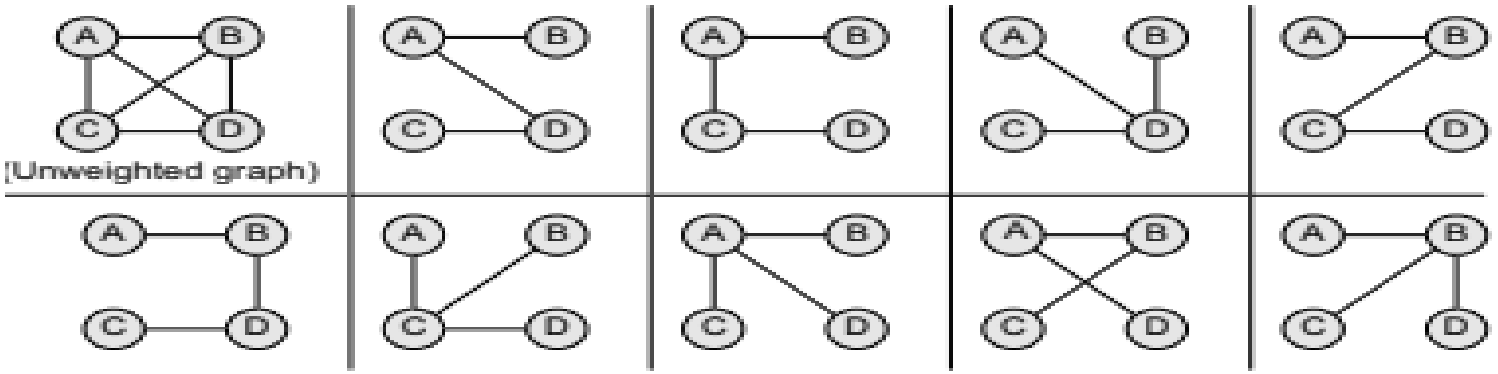




- A **minimum spanning tree** (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning trees.
- In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

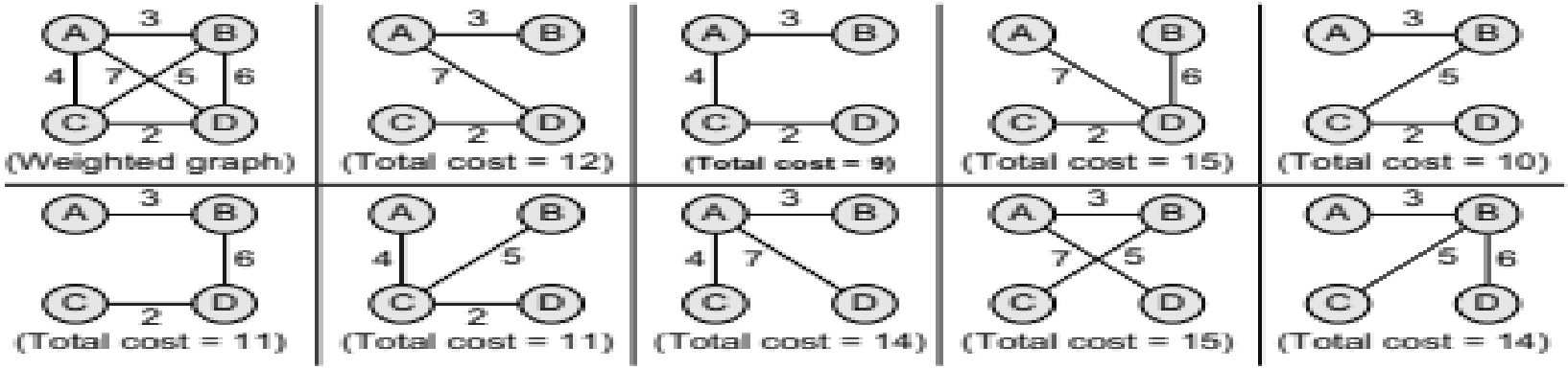
Minimum Spanning Tree...

Consider an **unweighted graph G** given below. From G, we can draw many distinct spanning trees. Eight of them are given here. For an unweighted graph, every spanning tree is a minimum spanning tree.



Unweighted graph and its spanning trees

Consider a **weighted graph G**. From G, we can draw three distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has the minimum weight (cost) associated with it of all the spanning trees given in, the one that is highlighted is called the minimum spanning tree, as it has the lowest cost associated with it.



Weighted graph and its spanning trees

Minimum Spanning Trees...

Applications of MST:

- MSTs are widely used for designing networks. For instance, people separated by varying distances wish to be connected together through a telephone network. A minimum spanning tree is used to determine the least costly paths with no cycles in this network, thereby providing a connection that has the minimum cost involved.
- MSTs are used to find airline routes. While the vertices in the graph denote cities, edges represent the routes between these cities. No doubt, more the distance between the cities, higher will be the amount charged. Therefore, MSTs are used to optimize airline routes by finding the least costly path with no cycles.
- MSTs are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires or telephone lines.
- MSTs are applied in routing algorithms for finding the most efficient path.

Algorithms for finding Minimum Spanning Trees:

i) Kruskal's Algorithm
ii) Prim's Algorithm

Kruskal's Algorithm

- Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph.
- In this algorithm, we use a priority queue in which edges that have minimum weight takes a priority over any other edge in the graph.
- The algorithm aims to find a subset of the edges that forms a tree that includes every vertex, the total weight of all the edges in the tree is minimized.
- When the Kruskal's algorithm terminates, the forest has only one component and forms a minimum spanning tree of the graph.

Kruskal's Algorithm:

Step 1: Create a forest in such a way that each graph is a separate tree.

Step 2: Create a priority queue Q that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY

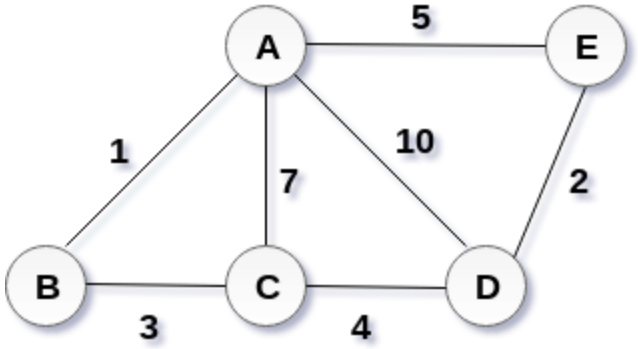
Step 4: Remove an edge from Q

Step 5: IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).
ELSE
Discard the edge

Step 6: END

Kruskal's Algorithm...

Example : Apply Kruskal's algorithm on the graph



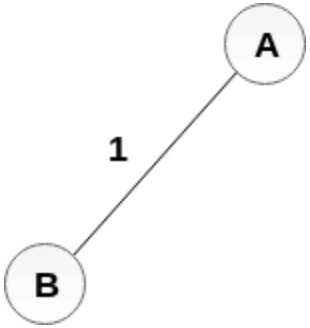
The weight of the edges given as :

Edge	AE	AD	AC	AB	BC	CD	DE
Weight	5	10	7	1	3	4	2

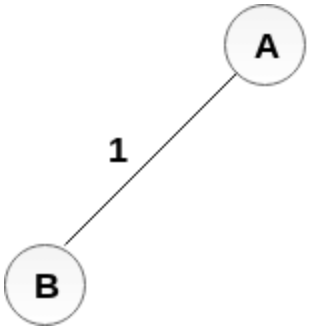
Sort the edges according to their weights:

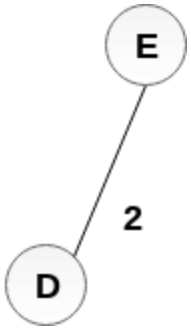
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Add AB to the MST:



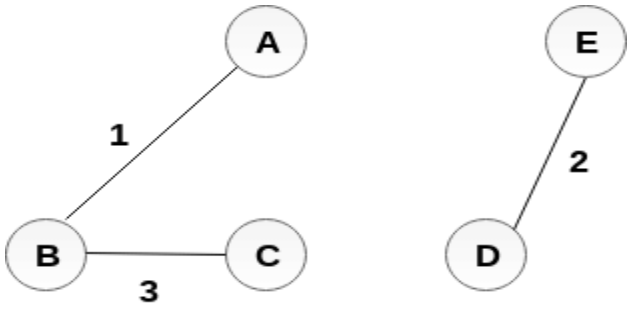
Add DE to MST:



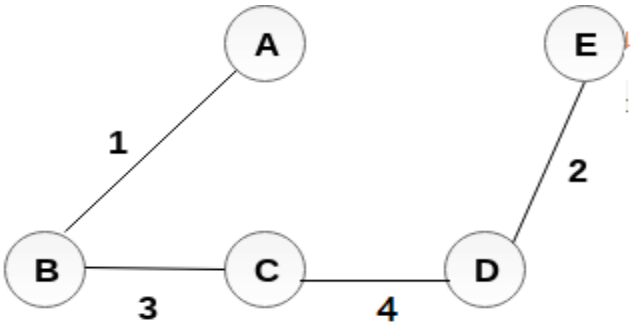


Kruskal's Algorithm...

Add BC to the MST:



Add CD to MST:



The next step is to add AE, but we can't add that as it will cause a cycle.
The next edge to be added is AC, but it can't be added as it will cause a cycle.
The next edge to be added is AD, but it can't be added as it will contain a cycle.

The cost of MST = 1 + 2 + 3 + 4 = 10

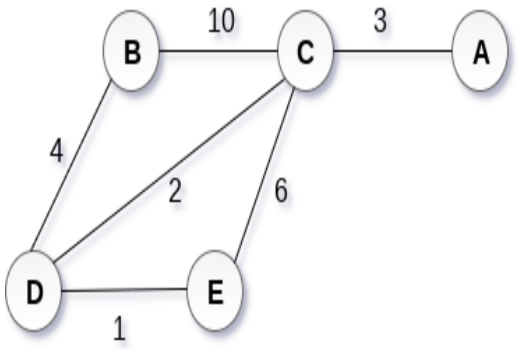
Prim's Algorithm

- Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph.
- This algorithm maintains three sets of vertices which can be given as below:
  1. **Tree vertices :**Vertices that are a part of the minimum spanning tree T.
  2. **Fringe vertices :** Vertices that are currently not a part of T, but are adjacent to some tree vertex.
  3. **Unseen vertices :** Vertices that are neither tree vertices nor fringe vertices fall under this category.
- The steps involved in the Prim's algorithm are:

```
Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
 Step 3:     Select an edge e connecting the tree vertex and
            fringe vertex that has minimum weight
Step 4:     Add the selected edge and the vertex to the
            minimum spanning tree T
        [END OF LOOP]
Step 5: EXIT
```

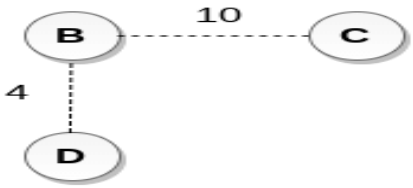
Prim's Algorithm...

Example : Construct a minimum spanning tree of the graph given using Prim's algorithm.

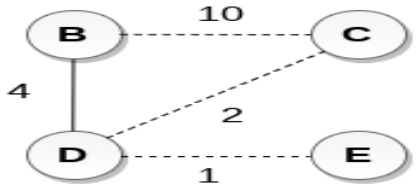


Step 1 : Choose a starting vertex B.
Step 2: Add the vertices that are adjacent to B. the edges that connecting the vertices are shown by dotted lines.
Step 3: Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.
Step 4: Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e. E and A.
Step 5: Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

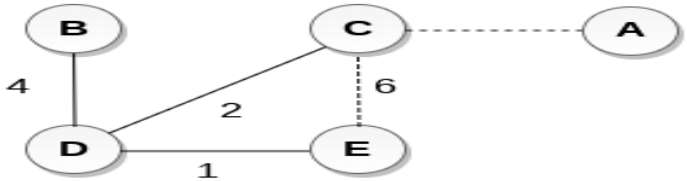
Step 1



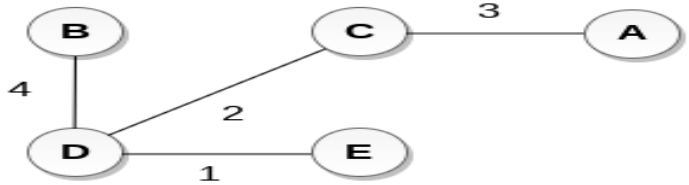
Step 2



Step 3



Step 4



Step 5

cost(MST) = 4 + 2 + 1 + 3 = 10 units.

Searching

- Searching means to find whether a particular value is present in an array or not.
- If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.
- if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
- There are two popular methods for searching the array elements:
  - i) Linear Search
  - ii) Binary Search.

Linear Search:

- Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value.
- It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.
- Linear search is mostly used to search an unordered list of elements.

For example :

int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};

The value to be searched is **VAL = 7**, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence.

Here, POS = 3 (index starting from 0).

Linear Search...

Linear Search Algorithm:

```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:      Repeat Step 4 while I<=N
Step 4:              IF A[I] = VAL
                          SET POS = I
                          PRINT POS
                          Go to Step 6
                     [END OF IF]
                      SET I = I + 1
               [END OF LOOP]
Step 5: IF POS = -1
          PRINT "VALUE IS NOT PRESENT
          IN THE ARRAY"
          [END OF IF]
Step 6: EXIT
```

- The best case of linear search is when VAL is equal to the first element of the array. In this case, only one comparison will be made.
- The worst case will happen when either VAL is not present in the array or it is equal to the last element of the array. In both the cases, n comparisons will have to be made.

Binary Search

- Binary search is a searching algorithm that works efficiently with a sorted list.
- In this algorithm, BEG and END are the beginning and ending positions of the list that we are looking to search for the element. MID is calculated as (BEG + END)/2.
- if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].
  - (a) If VAL < A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID – 1.
  - (b) If VAL > A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID + 1.

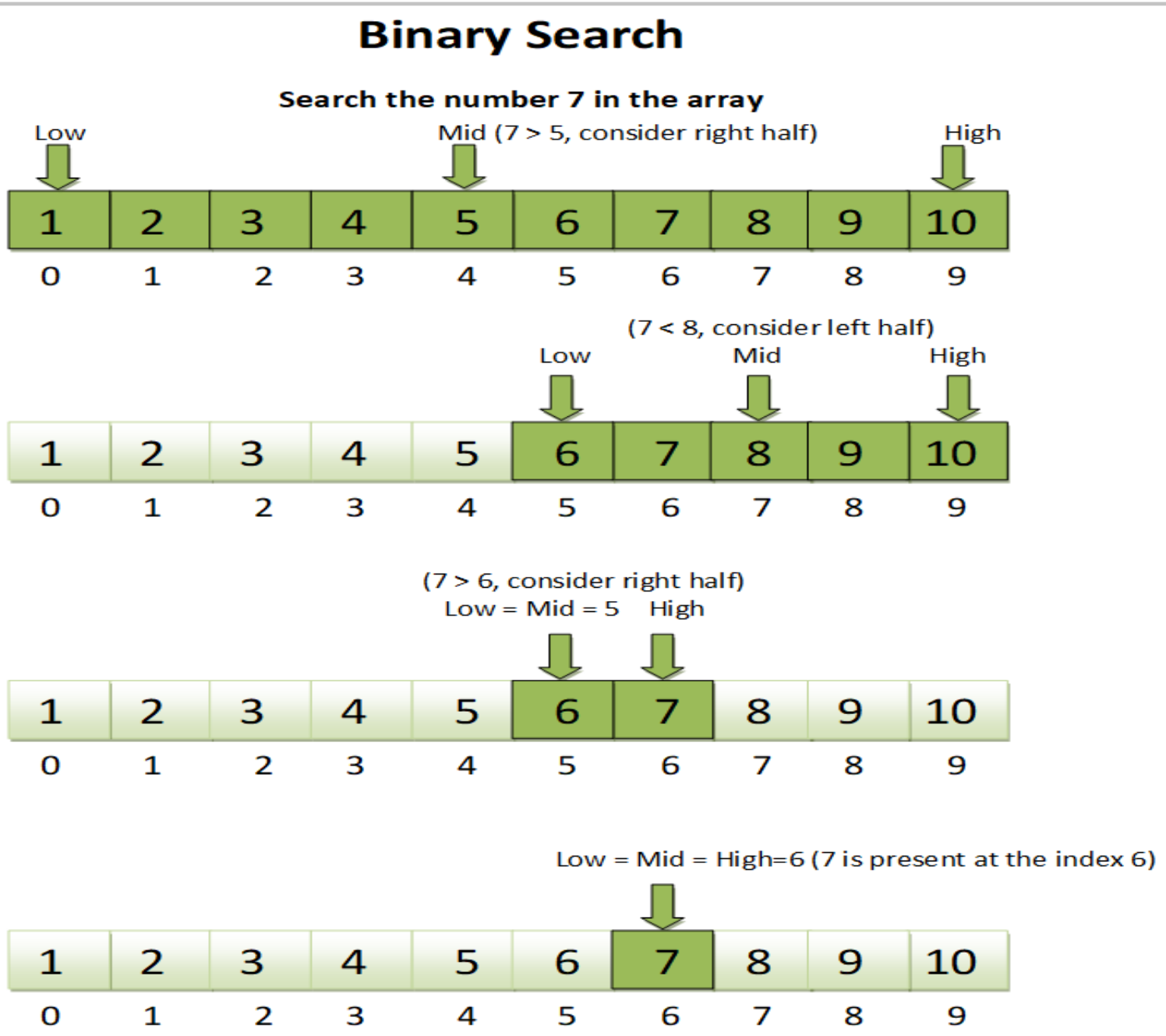
```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:          SET MID = (BEG + END)/2
Step 4:          IF A[MID] =  VAL
                       SET POS = MID
                       PRINT POS
                       Go to Step 6
                 ELSE IF A[MID] > VAL
                       SET END = MID - 1
                 ELSE
                       SET BEG = MID + 1
                 [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

- The algorithm will terminate when A[MID] = VAL. When the algorithm ends, we will set POS = MID. POS is the position at which the value is present in the array.
- Finally, if VAL is not present in the array, then eventually, END will be less than BEG. When this happens, the algorithm will terminate and the search will be unsuccessful.

Binary Search

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:            SET MID = (BEG + END)/2
Step 4:            IF A[MID] =  VAL
                         SET POS = MID
                         PRINT POS
                         Go to Step 6
                   ELSE IF A[MID] > VAL
                         SET END = MID - 1
                   ELSE
                         SET BEG = MID + 1
                   [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Binary Search…



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.
- 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};
- There are two types of sorting:
  1. **Internal sorting :** which deals with sorting the data stored in the computer's memory
  2. **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.

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.
- Here 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 smaller element is placed before the bigger one.
- This process will continue till the list of unsorted elements are over.

Bubble Sort ...

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

Pass1:

(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

After the end of the first pass, the largest element is placed at the highest index of the array.

```
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
```

Bubble Sort...

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

After the end of the second pass, the second largest element is placed at the second highest index of the array.

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

Bubble Sort ...

(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.

After the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array.

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

After the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array.

Bubble Sort ...

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.

After the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array.

Pass 7:

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

Observe that the entire list is sorted now.

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

Insertion Sort

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 (assuming LB = 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 (if LB = 0).
- 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.

```
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
```

Insertion sort...

Example : Sort the elements using Insertion sort

50,10,30,80,70,20,60,40

PASS	SORTED LIST	UNSORTED LIST	TEST
1	50	10 30 80 70 20 60 40	50 > 10
2	10 50	30 80 70 20 60 40	50 > 30
	10 50	30 80 70 20 60 40	10 < 30
3	10 30 50	80 70 20 60 40	50 < 80
4	10 30 50 80	70 20 60 40	80 > 70
	10 30 50 80	70 20 60 40	50 < 70
5	10 30 50 70 80	20 60 40	80 > 20
	10 30 50 70 80	20 60 40	70 > 20
	10 30 50 70 80	20 60 40	50 > 20
	10 30 50 70 80	20 60 40	30 > 20
	10 30 50 70 80	20 60 40	10 < 20
6	10 20 30 50 70 80	60 40	80 > 60
	10 20 30 50 70 80	60 40	70 > 60
	10 20 30 50 70 80	60 40	50 < 60
7	10 20 30 50 60 70 80	40	80 > 40
	10 20 30 50 60 70 80	40	70 > 40
	10 20 30 50 60 70 80	40	60 > 40
	10 20 30 50 60 70 80	40	50 > 40
	10 20 30 50 60 70 80	40	30 < 40
	10 20 30 40 50 60 70 80	<empty>	

Selection sort

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.
- 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.

```
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
```

Selection Sort…

Example : Perform selection sort on the following elements 39,9,81,45,90,27,72,18

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

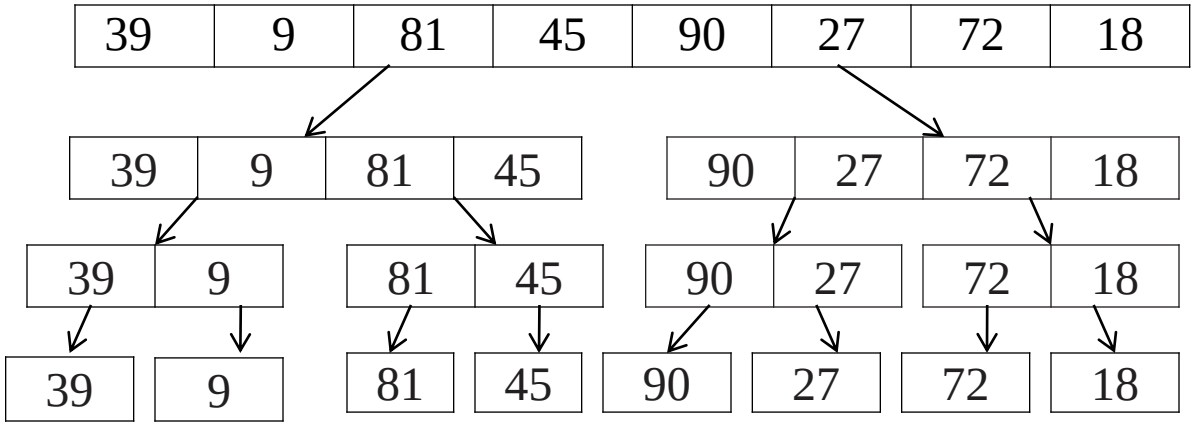
PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Merge Sort

- Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.
- **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 1and A2 , each containing about half of the elements of A.
- **Conquer** means sorting the two sub-arrays recursively using merge sort.
- **Combine** means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

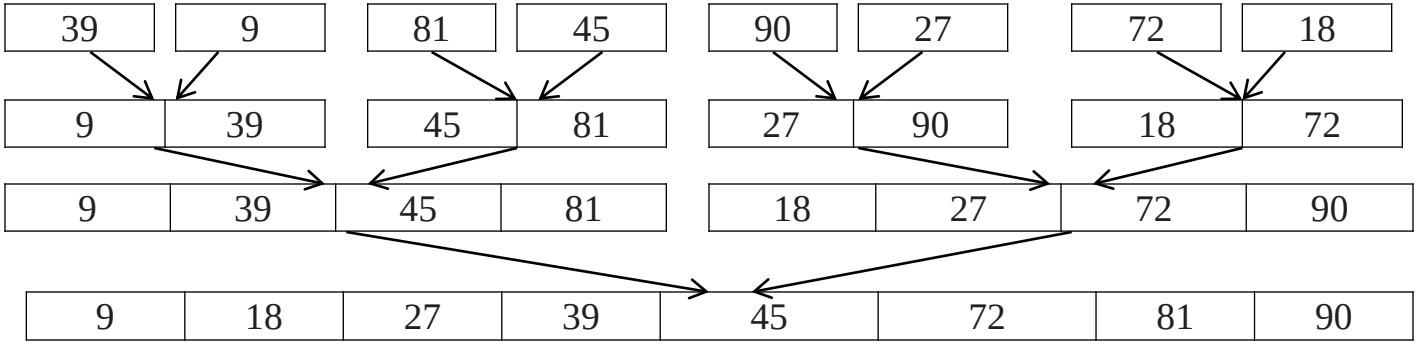
Example: 39,9,81,45,90,27,72,18

- Divide and Conquer the array



Merge Sort…

➢Combine the elements to form the sorted array:



➢The merge sort algorithm recursively divides the list into smaller lists, the merge algorithm conquers the list to sort the elements in individual lists. Finally, the smaller lists are merged to form one list.

Merge Sort Algorithm:

```
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
```

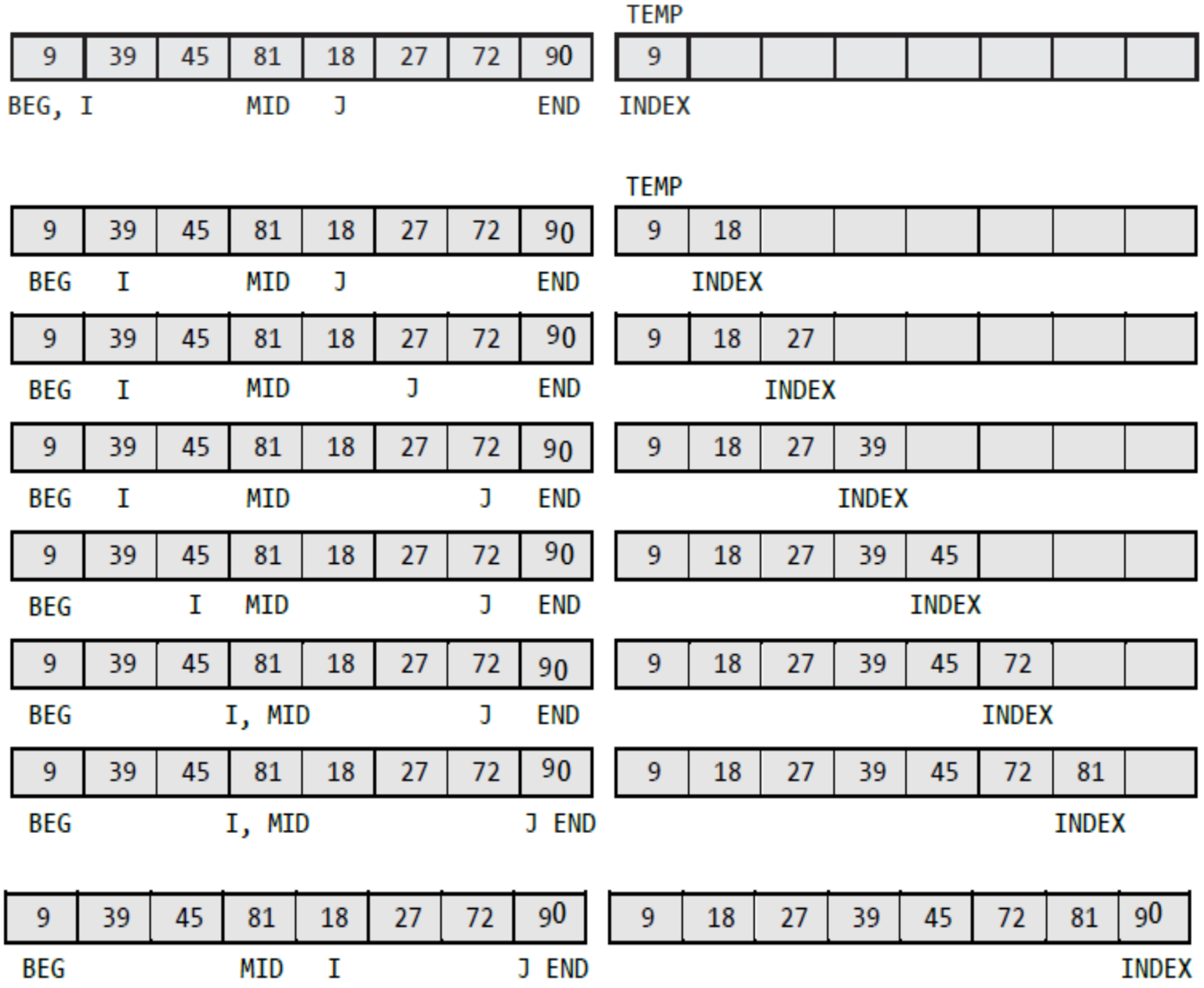
Merge Sort…

Combine the elements to form the sorted array:

```
MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
Step 2: Repeat while (I <= MID) AND (J<=END)
            IF ARR[I] < ARR[J]
                  SET TEMP[INDEX] = ARR[I]
                  SET I = I + 1
            ELSE
                  SET TEMP[INDEX] = ARR[J]
                  SET J = J + 1
            [END OF IF]
            SET INDEX = INDEX + 1
      [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
            IF I > MID
               Repeat while J <= END
                  SET TEMP[INDEX] = ARR[J]
                  SET INDEX = INDEX + 1, SET J = J + 1
               [END OF LOOP]
      [Copy the remaining elements of left sub-array, if any]
            ELSE
               Repeat while I <= MID
                  SET TEMP[INDEX] = ARR[I]
                  SET INDEX = 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 < INDEX
               SET ARR[K] = TEMP[K]
               SET K = K + 1
      [END OF LOOP]
Step 6: END
```

Merge Sort…


TEMP
9 39 45 81 18 27 72 90
9
BEG, I MID J END
INDEX
TEMP
9 39 45 81 18 27 72 90
9 18
BEG I MID J END
INDEX
9 39 45 81 18 27 72 90
9 18 27
BEG I MID J END
INDEX
9 39 45 81 18 27 72 90
9 18 27 39
BEG I MID J END
INDEX
9 39 45 81 18 27 72 90
9 18 27 39 45
BEG I MID J END
INDEX
9 39 45 81 18 27 72 90
9 18 27 39 45 72
BEG I, MID J END
INDEX
9 39 45 81 18 27 72 90
9 18 27 39 45 72 81
BEG I, MID J END
INDEX
9 39 45 81 18 27 72 90
9 18 27 39 45 72 81 90
BEG MID I J END
INDEX

Quick Sort…

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 in 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 right = loc, it means the pivot has been placed in its correct position.

   (b) However, if at any point, we have a[loc] > a[right], then interchange the two values and jump to Step 3.

   (c) Set loc = 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.

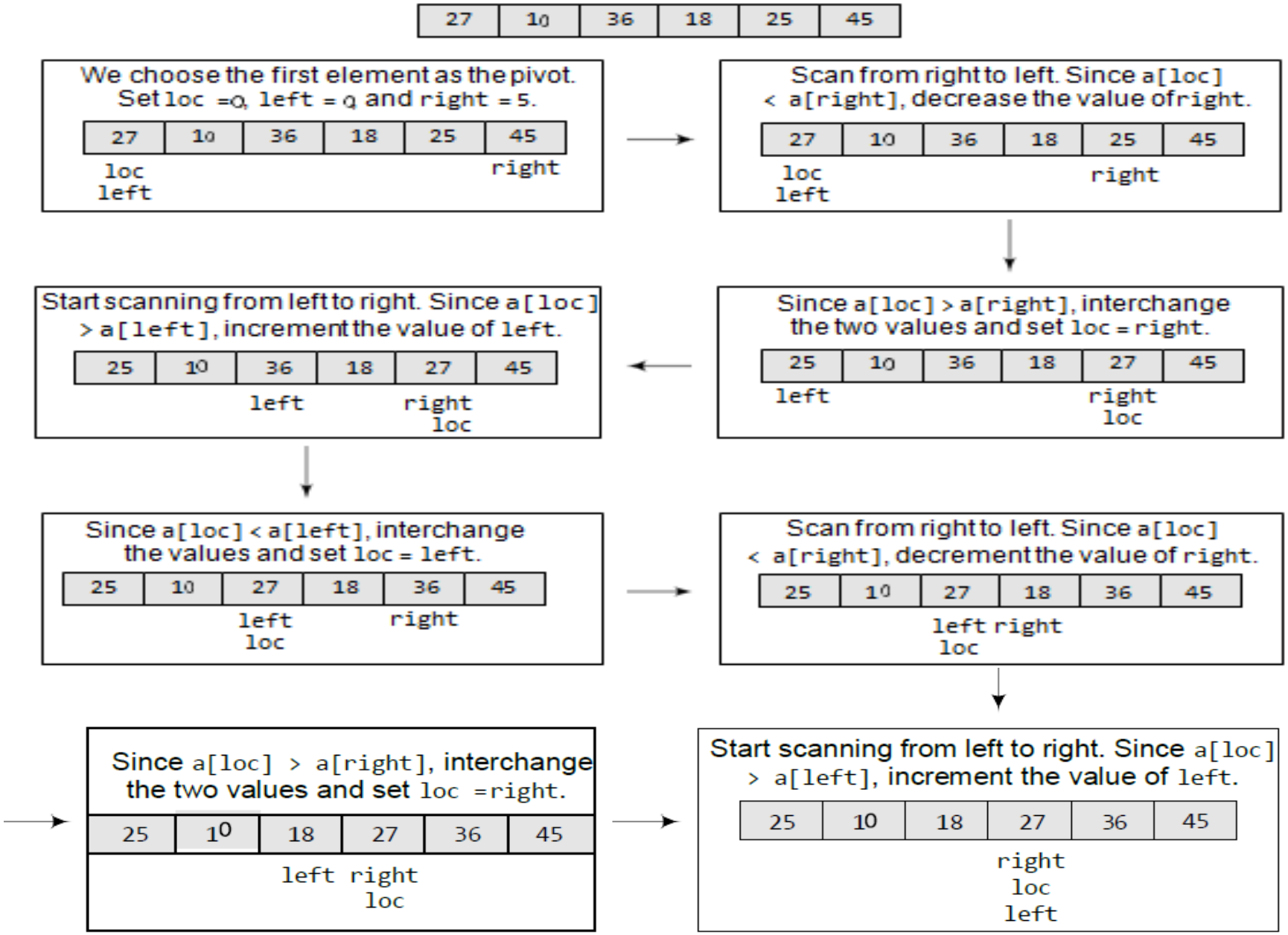
   That is, a[loc] should be greater than a[left].

   (a) 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.

   (b) However, if at any point, we have a[loc] < a[left], then interchange the two values and jump to Step 2.

   (c) Set loc = left.

Quick Sort…


27 10 36 18 25 45
We choose the first element as the pivot. Set loc = 0, left = 0 and right = 5.
27 10 36 18 25 45
loc
left
right
Scan from right to left. Since a[loc] < a[right], decrease the value of right.
27 10 36 18 25 45
loc
left
right
Since a[loc] > a[right], interchange the two values and set loc = right.
25 10 36 18 27 45
left
right
loc
Start scanning from left to right. Since a[loc] > a[left], increment the value of left.
25 10 36 18 27 45
left
right
loc
Since a[loc] < a[left], interchange the values and set loc = left.
25 10 27 18 36 45
left
loc
right
Scan from right to left. Since a[loc] < a[right], decrement the value of right.
25 10 27 18 36 45
left right
loc
Since a[loc] > a[right], interchange the two values and set loc = right.
25 10 18 27 36 45
left right
loc
Start scanning from left to right. Since a[loc] > a[left], increment the value of left.
25 10 18 27 36 45
right
loc
left

Quick Sort…

```
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] <= ARR[RIGHT] AND LOC != 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] >= ARR[LEFT] AND LOC != 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
```