Q.1) What is Stack? Explain Stack operation.

- ✓ Stack is an ordered list of the same type of elements.
- ✓ It is a linear list where all insertions and deletions are permitted only at one end of the list.
- ✓ Stack is a LIFO (Last In First Out) structure.
- ✓ In a stack, when an element is added, it goes to the top of the stack.

Definition

"Stack is a collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle"

➢ **There are four basic operations performed in a Stack.**

1. **Push()** function is used to add or insert new elements into the stack.
2. **Pop()** function is used to delete or remove an element from the stack.
3. **Peep()** function is used to find the i^{th} element from stack.
4. **Change()** function is used to change the i^{th} element from stack.

- ✓ When a stack is completely full, it is said to be **Overflow state** and if stack is completely empty, it is said to be **Underflow state**.
- ✓ Stack allows operations at **one end only**.
- ✓ Stack behaves like a real life stack, for example, in a real life, we can remove a plate or dish from the top of the stack only or while playing a deck of cards, we can place or remove a card from top of the stack only.
- ✓ Similarly, here also, we can only access the top element of a stack.
- ✓ According to its LIFO structure, the element which is inserted last, is accessed first.

Implementation of Stack

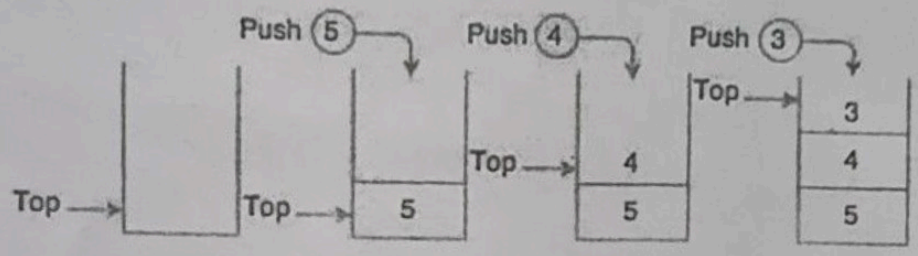


Fig. Insertion of Elements in a Stack

- ✓ The above diagram represents a stack insertion operation.
- ✓ In a stack, inserting and deleting of elements are performed at a single position which is known as, **Top**.
- ✓ Insertion operation can be performed using Push() function.
- ✓ New element is added at top of the stack and removed from top of the stack, as shown in the diagram below:

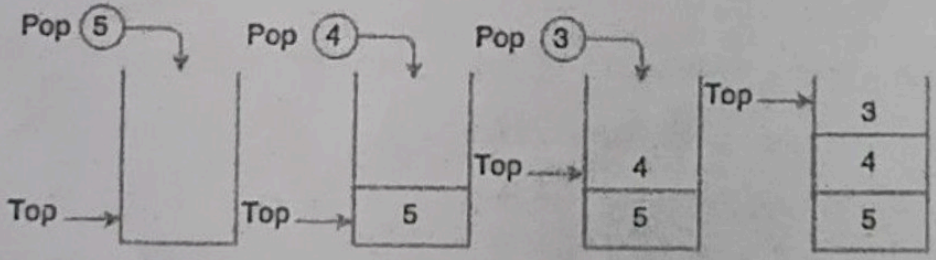


Fig. Deletion of Elements in a Stack

- ✓ An element is removed from top of the stack.
- ✓ Delete operation is based on LIFO principle. This operation is performed using a Pop() function.
- ✓ It means that the insertion and deletion operations are performed at one end i.e at Top.

Q.2) Write an algorithm of push, pop ,peep, change.

❖ ***Procedure : PUSH (S, TOP, X)***

✓ This procedure inserts an element x to the top of a stack which is represented by a vector S containing N elements with a pointer TOP denoting the top element in the stack.

1. [Check for stack overflow]
   If TOP ≥ N
   Then write ('STACK OVERFLOW')
   Return
2. [Increment TOP]
   TOP ←TOP + 1
3. [Insert Element]
   S[TOP] ←X
4. [Finished] Return

❖ ***Function : POP (S, TOP)***

✓ This function removes the top element from a stack which is represented by a vector S and returns this element.

✓ TOP is a pointer to the top element of the stack.

1. [Check for underflow of stack]
   If TOP = 0
   Then Write ('STACK UNDERFLOW ON POP')
   Take action in response to
   underflow Return
2. [Decrement Pointer]
   TOP ← TOP – 1
3. [Return former top
   element of stack] Return
   (S[TOP + 1])

❖ *Function : PEEP (S, TOP, I)*

✓ This function returns the value of the i^{th} element from the TOP of the stack which is represented by a vector S containing N elements.
✓ The element is not deleted by this function.

1. [Check for stack Underflow]

   If TOP - I +1 ≤ 0

   Then Write ('STACK UNDERFLOW ON PEEP')

   Take action in response to

   Underflow Exit

2. [Return I^{th} element from top of the stack]

   Return (S[TOP – I + 1])

❖ *PROCEDURE : CHANGE (S, TOP, X, I)*

✓ This procedure changes the value of the I^{th} element from the top of the stack to the value containing in X.
✓ Stack is represented by a vector S containing N elements.

1. [Check for stack Underflow]

   If TOP – I + 1 ≤ 0

   Then Write ('STACK UNDERFLOW ON CHANGE')

   Return

2. [Change I^{th} element from top of the stack]

   S[TOP – I + 1] ← X

3. [Finished]

   Return

Q.3) Convert the following expression infix to prefix and postfix.

Expression: A+B*C+D

Prefix : A+B*C+D

Step 1 : A+*BC+D

Step 2: +A*BC+D

Step 3: ++A*BCD

Postfix: A+B*C+D

Step 1 : A+BC*+D

Step 2: ABC*++D

Step 3: ABC*+D+

Q.3) Convert the following expression infix to postfix using stack.

X: A + (B*C –(D / E↑F) * G) * H

SLNo	Symbol Scanned	Stack	Expression Y
1	A	(	A
2	+	(+	A
3	(	(+(	A
4	B	(+(	A B
5	*	(+(*	A B
6	C	(+(*	A B C
7	-	(+(-	A B C *
8	(	(+(-(	A B C *
9	D	(+(-(	A B C * D
10	/	(+(-(/	A B C * D
11	E	(+(-(/	A B C * D E
12	↑	(+(-(/↑	A B C * D E
13	F	(+(-(/↑	A B C * D E F
14	)	(+(-	A B C * D E F ↑ /
15	*	(+(-*	A B C * D E F ↑ /
16	G	(+(-*	A B C * D E F ↑ / G
17	)	(+	A B C * D E F ↑ / G * -
18	*	(+*	A B C * D E F ↑ / G * -
19	H	(+ *	A B C * D E F ↑ / G * - H
20	)		A B C * D E F ↑ / G * - H * +

Q.4) Difference between Stack and Queue.

Stack	Queue
A Linear List Which allows insertion or deletion of an element at one end only is called as Stack	A Linear List Which allows insertion at one end and deletion at another end is called as Queue
Stacks follow Last In First Out (LIFO) order.	Queues following First In First Out (FIFO) order.
In stacks only one pointer is used	In queues, two different pointers are used.
Stack used TOP pointer for insertion and deletion	Queue used REAR for insertion and FRONT for deletion
In stacks, the last inserted object is first to come out.	In queues, the object inserted first is first deleted.
To check if a stack is empty, following condition is used: **TOP == 0**	To check if a queue is empty, following condition is used: **FRONT == 0**
A Stack cannot be divided into sub sections	A Queue can be divided into sub sections: Double Ended Queue, Simple Queue, Priority Queue and Circular Queue.
Example of stack is arranging plates in one above one.	Example is ordinary queue in provisional store.

Q.5) Write an algorithm of Simple queue insert and delete operation.

Procedure: QINSERT_REAR (Q, F, R, N,Y)

✓ Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This procedure inserts Y at rear end of Queue.

1. [Overflow]
   IF R >= N
   Then write ('OVERFLOW')
   Return
2. [Increment REAR pointer]
   R = R + 1
3. [Insert element]
   Q[R] = Y
4. [Is front pointer properly set]
   IF F=0
   Then
   F=1 Return

Function: QDELETE_FRONT (Q, F, R)

- ✓ Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This function deleted and element from front end of the Queue.
- ✓

1. [Underflow]
   IF F= 0
   Then write ('UNDERFLOW')
   Return(0) (0 denotes an empty Queue)
2. [Decrement element]
   Y = Q[F]
3. [Queue empty?]
   IF F=R
   Then F= R= 0
   Else F=F+1 (increment front pointer)
4. [Return element]
   Return (Y)

Q.6) Write an algorithm of Circular Queue insert and delete

Procedure: CQINSERT (F, R, Q, N, Y)

- ✓ Given F and R pointers to the front and rear elements of a circular queue respectively.
- ✓ Circular queue Q consisting of N elements.
- ✓ This procedure inserts Y at rear end of Circular queue.

1. **[Reset Rear Pointer]**
   If R = N
   Then R ← 1
   Else R ← R + 1
2. **[Overflow]**
   If F = R
   Then Write ('Overflow')
   Return
3. **[Insert element]**
   Q[R] ← Y
4. **[Is front pointer properly set?]**
   If F = 0
   Then
   F ← 1
   Return

Function CQDELETE (F, R, Q, N)

- ✓ Given F and R pointers to the front and rear elements of a Circular queue respectively.
- ✓ Circular Queue Q consisting of N elements.
- ✓ This function deleted and element from front end of the Circular Queue.

✓ Y is temporary pointer variable.

1 [Underflow?]
If F = 0
Then
Write ('UNDERFLOW')
Return (0)

2 [Delete Element]
Y ← Q[F]

3 [Queue Empty?]
If F = R
Then F ← R ← 0
Return (Y)

4 [Increment front pointer]
If F = N
Then F ← 1
Else F ← + 1
Return (Y)

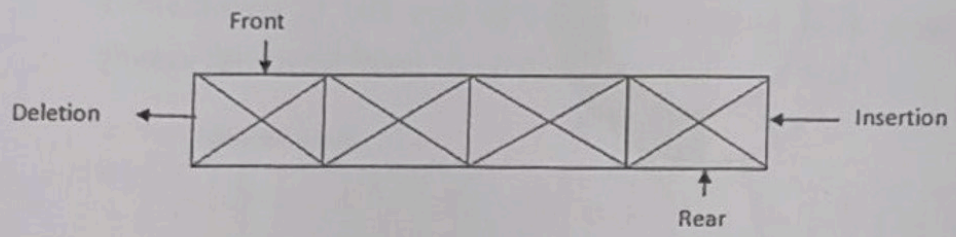
Q.7) What is Queue? Explain types of Queue with example.

✓ There are four types of queue in data structure.

(i) Simple Queue

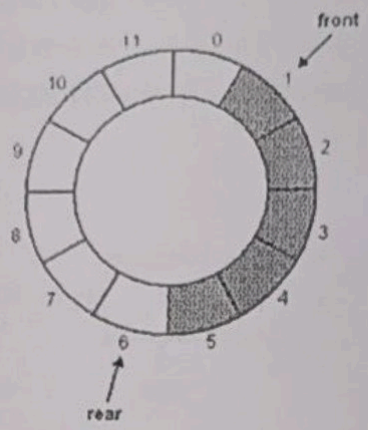
- A linear list which permits deletion to be performed at one end of the list and insertion at the other end is called queue.
- The information in such a list is processed FIFO (first in first out) of FCFS (first come first served) pattern.
- Front is the end of queue from that deletion is to be performed.
- Rear is the end of queue at which new element is to be inserted.
- The process to add an element into queue is called **Enqueue**

- The process of removal of an element from queue is called **Dequeue**.
- The familiar and traditional example of a queue is Checkout line at Supermarket Cash Register where the first person in line is usually the first to be checked out.



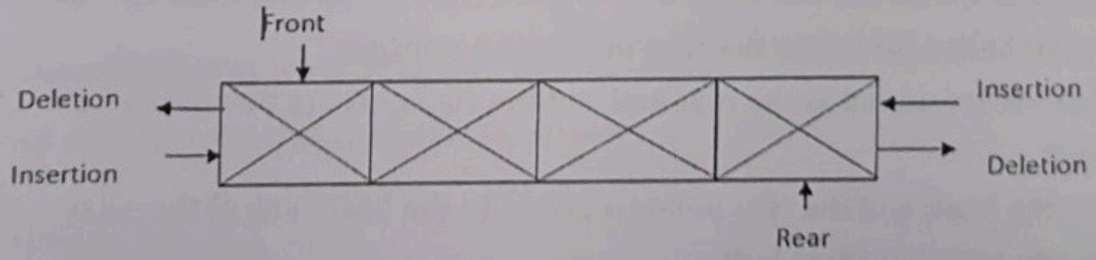
(ii) Circular Queue

- A more suitable method of representing simple queue which prevents an excessive use of memory is to arrange the elements Q[1], Q[2]….,Q[n] in a circular fashion with Q[1] following Q[n], this is called circular queue
- In a standard queue data structure re-buffering problem occurs for each **dequeue** operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue
- Circular queue is a linear data structure. It follows FIFO principle.
- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as "Ring buffer".



(i) Dequeue

- A dequeue (double ended queue) is a linear list in which insertion and deletion are performed from the either end of the structure.
- There are two variations of Dqueue
  - Input restricted dqueue- allows insertion at only one end
  - Output restricted dqueue- allows deletion from only one end
- Such a structure can be represented by following fig.



(i) Priority Queue

- A queue in which we are able to insert remove items from any position based on some property (such as priority of the task to be processed) is

often referred as priority queue.

- Below fig. represent a priority queue of jobs waiting to use a computer.
- Priorities of 1, 2, 3 have been attached with jobs of real time, online and batch respectively. Therefore if a job is initiated with priority i,it is inserted immediately at the end of list of other jobs with priorities i. Here jobs are always removed from the front of queue

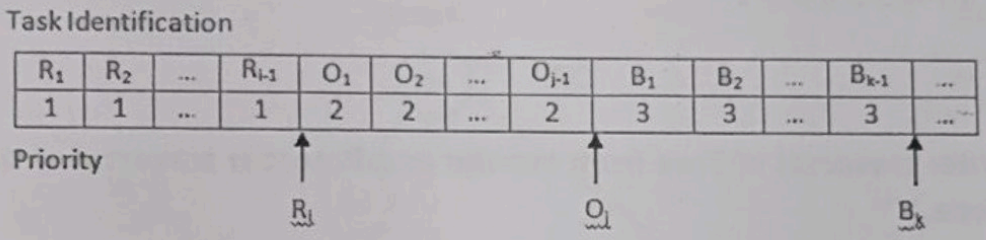


Fig (a) : Priority Queue viewed as a single queue with insertion allowed at any position.

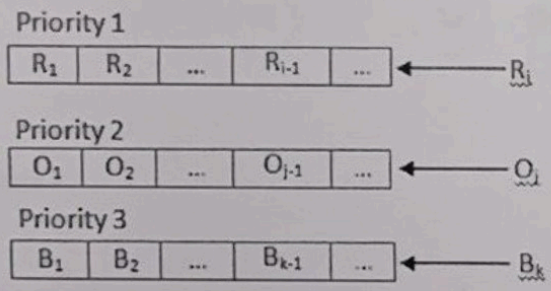
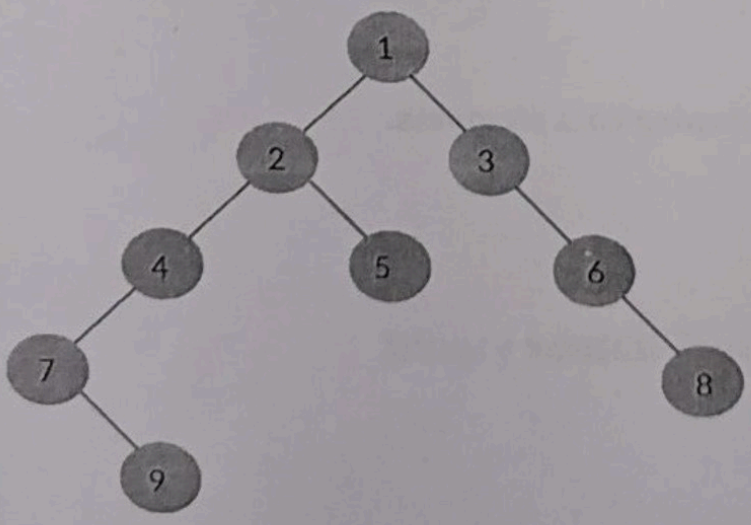


Fig (b) : Priority Queue viewed as a Viewed as a set of queue

Q.8) Using tree find out inorder, prorder, postorder (7)



Inorder Traversal: 7 9 4 2 5 1 3 6 8

Preorder Traversal: 1 2 4 7 9 5 3 6 8

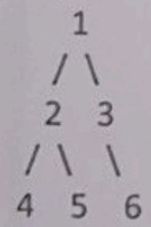
Postorder Traversal: 9 7 4 5 2 8 6 3 1

OR

Q.8) Generate Postorder traversal of Tree from Inorder and Preorder traversal of tree without generating Tree.

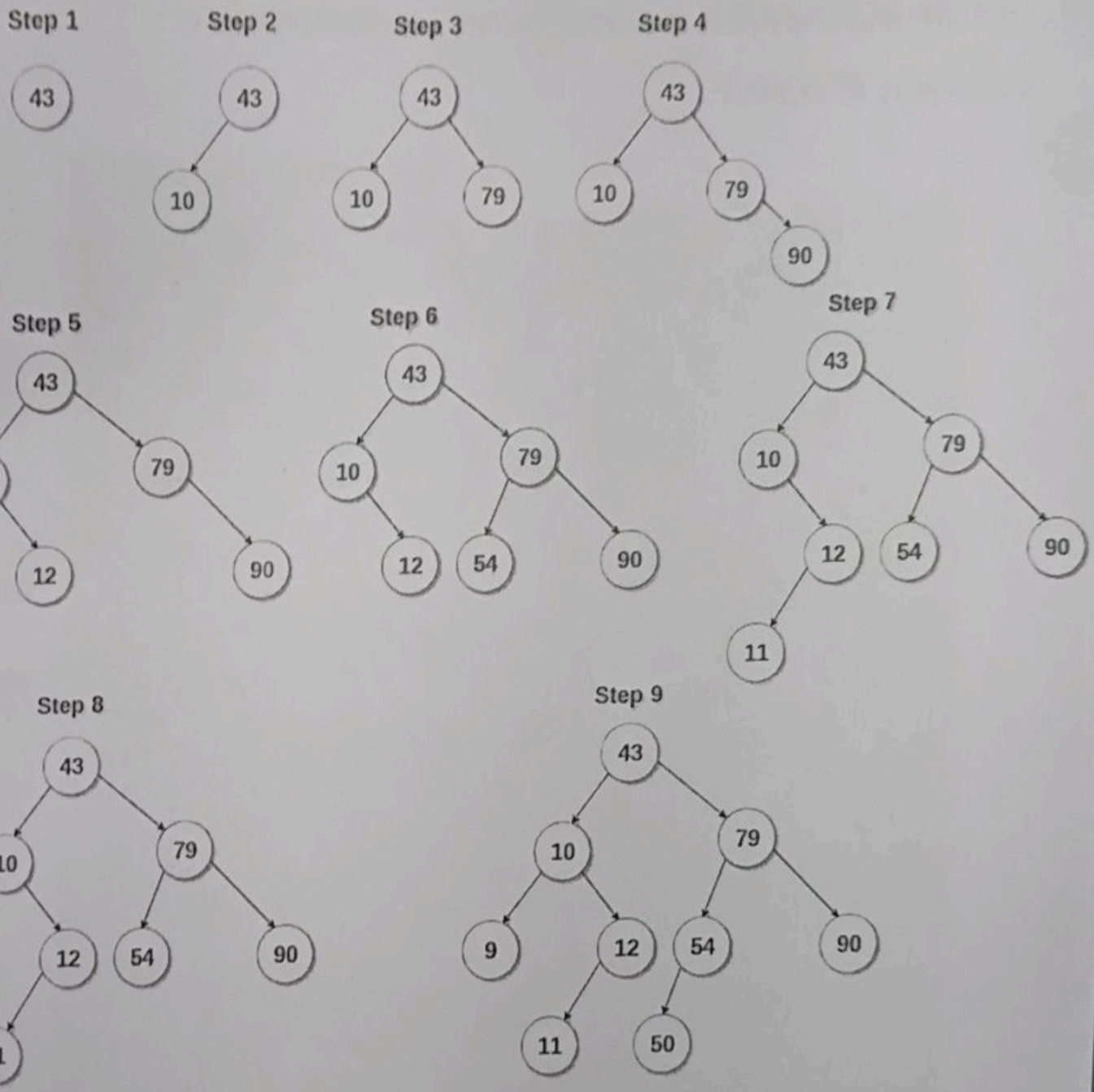
Input:
In-order traversal inorder {4, 2, 5, 1, 3, 6}
Pre-order traversal preorder {1, 2, 4, 5, 3, 6}

Output:



Post-order traversal is {4, 5, 2, 6, 3, 1}

Q.9) Create the binary search tree using the following data elements.



Binary search Tree Creation

Q.10) Create the Heap tree / Heap Sort using the following data elements.

29,21,16,4,41,49,32,24,74

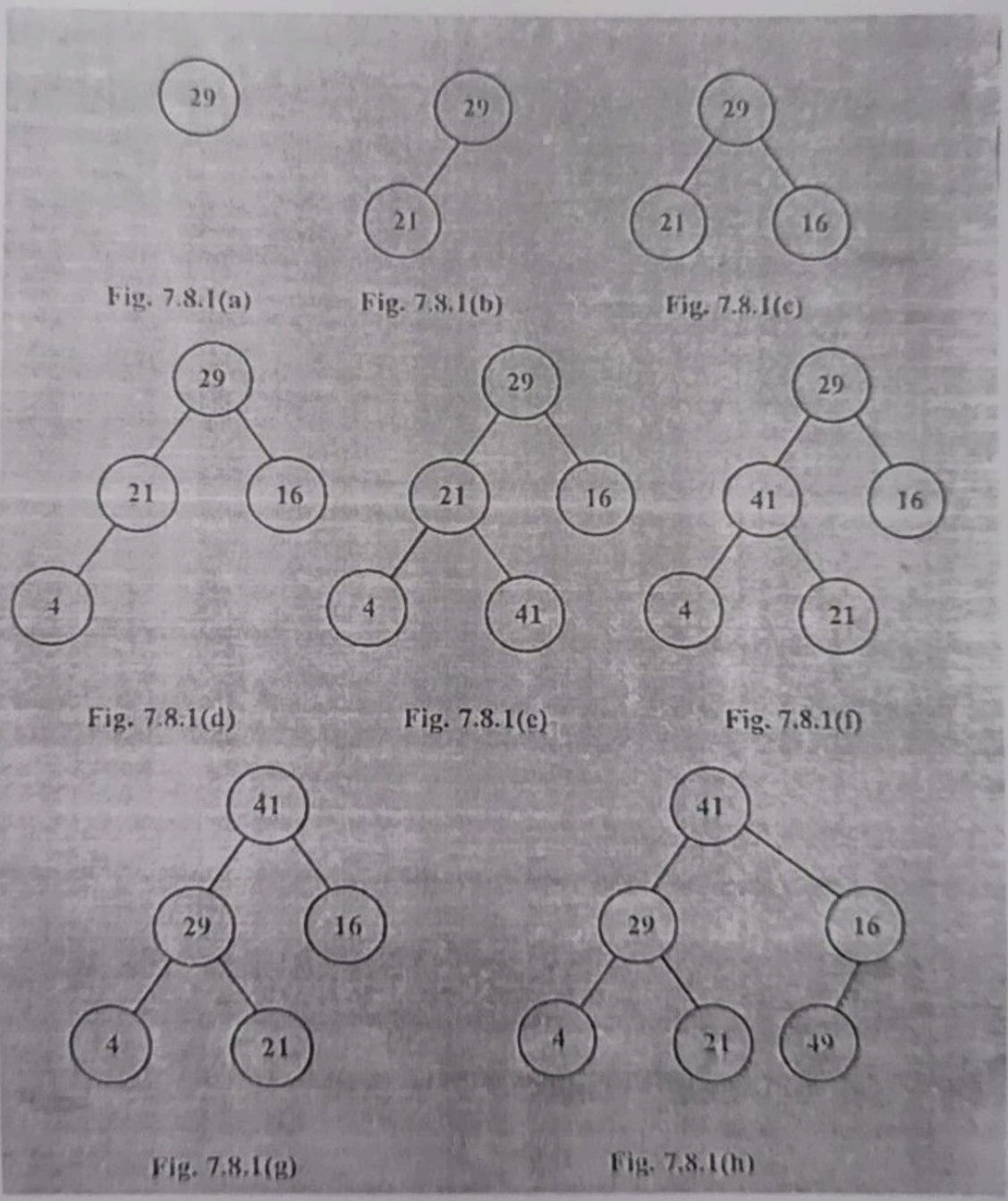


Fig. 7.8.1(a) Fig. 7.8.1(b) Fig. 7.8.1(c)

Fig. 7.8.1(d) Fig. 7.8.1(e) Fig. 7.8.1(f)

Fig. 7.8.1(g) Fig. 7.8.1(h)

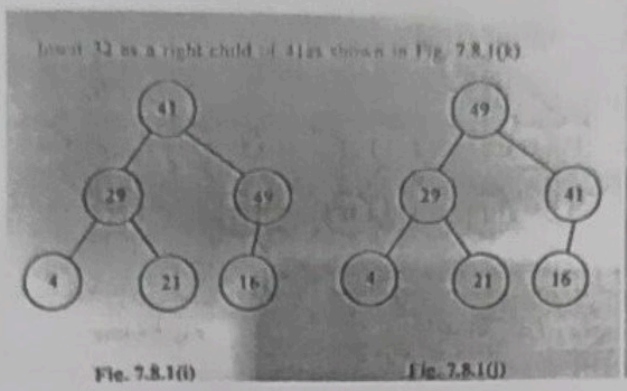


Fig. 7.8.1(i) Fig. 7.8.1(j)

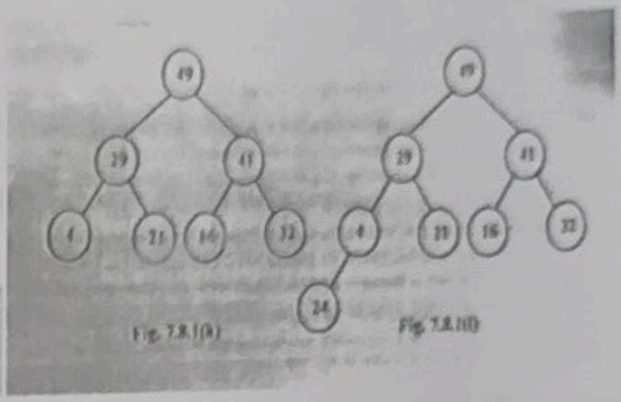


Fig. 7.8.1(k) Fig. 7.8.1(l)

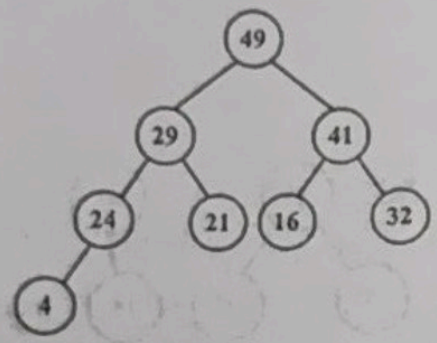


Fig. 7.8.1(m)

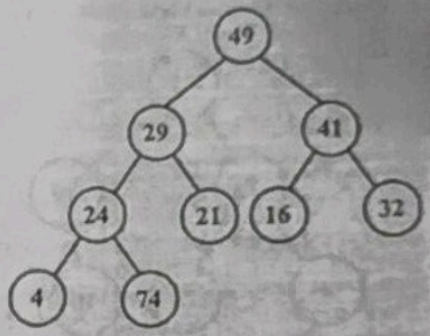


Fig. 7.8.1(n)

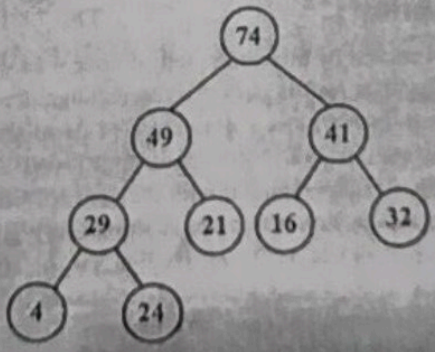
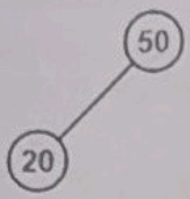


Fig. 7.8.1(o) Heap Created

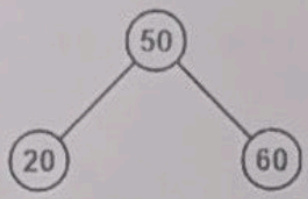
Q.11) Create the AVL tree using the following data elements.

50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

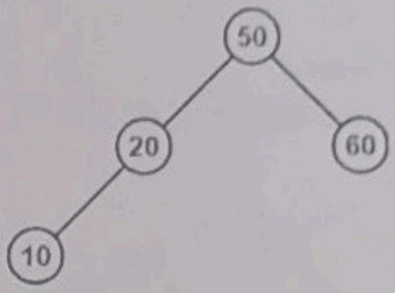
Tree is Balanced



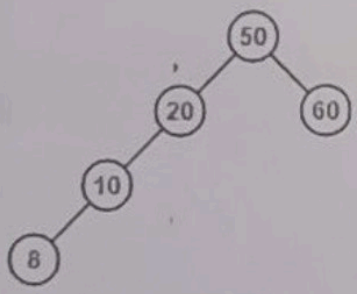
Tree is Balanced



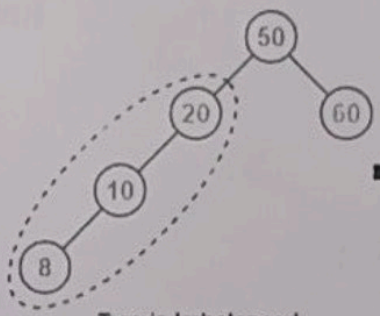
Tree is Balanced



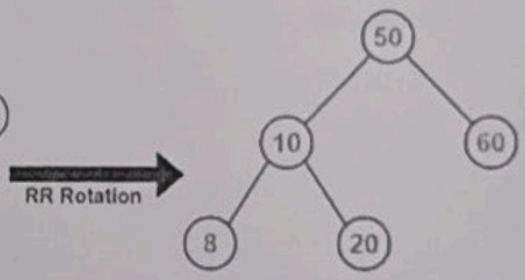
Tree is Balanced



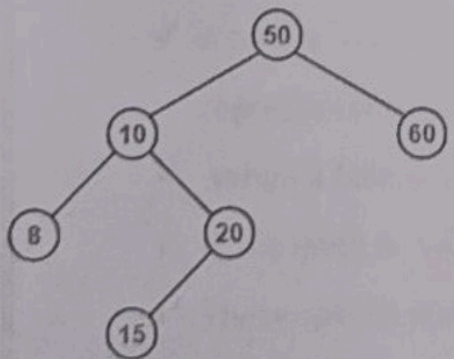
Tree is Imbalanced



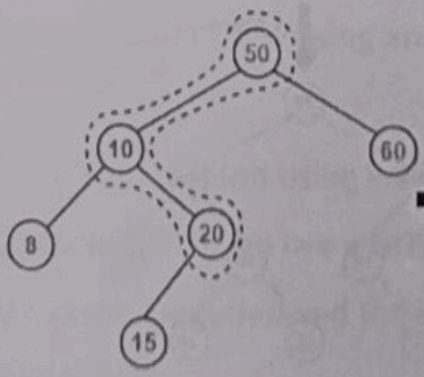
Tree is Imbalanced



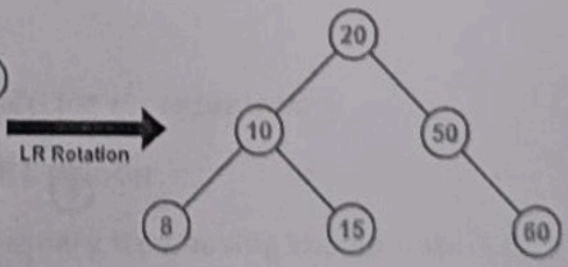
Tree is Balanced



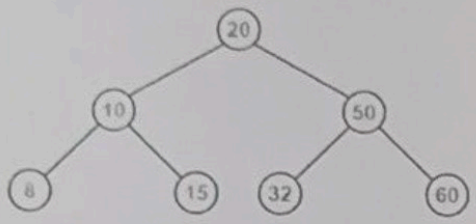
Tree is Imbalanced



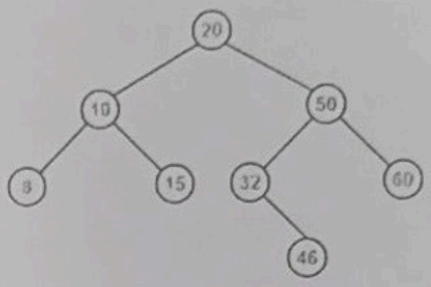
Tree is Imbalanced



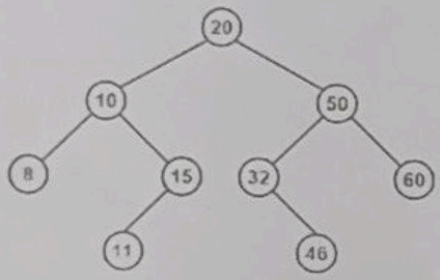
Tree is Balanced



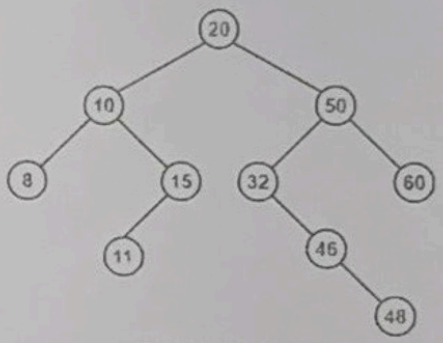
Tree is Balanced



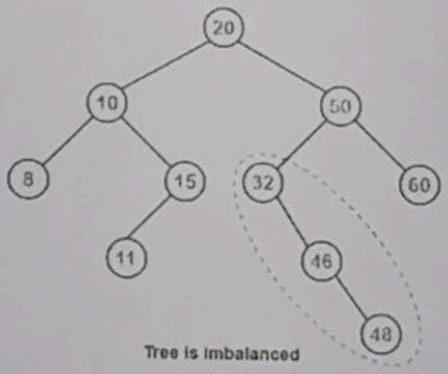
Tree is Balanced



Tree is Balanced

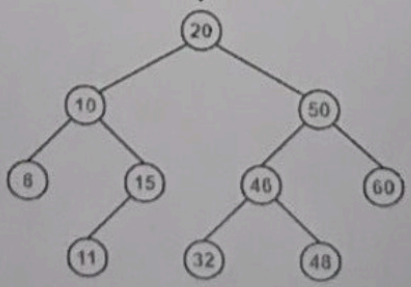


Tree is Imbalanced



Tree is Imbalanced

LL Rotation



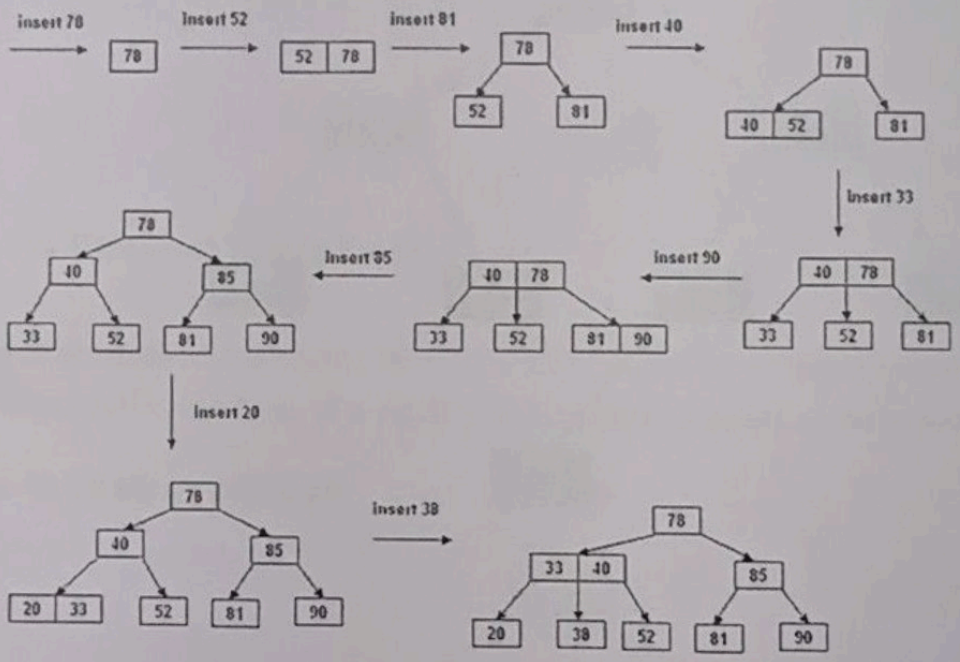
Tree is Balanced

Q.12) Create the B tree using the following data elements.

78,52,81,40,33,90,85,20,38

Insertion in B-Trees

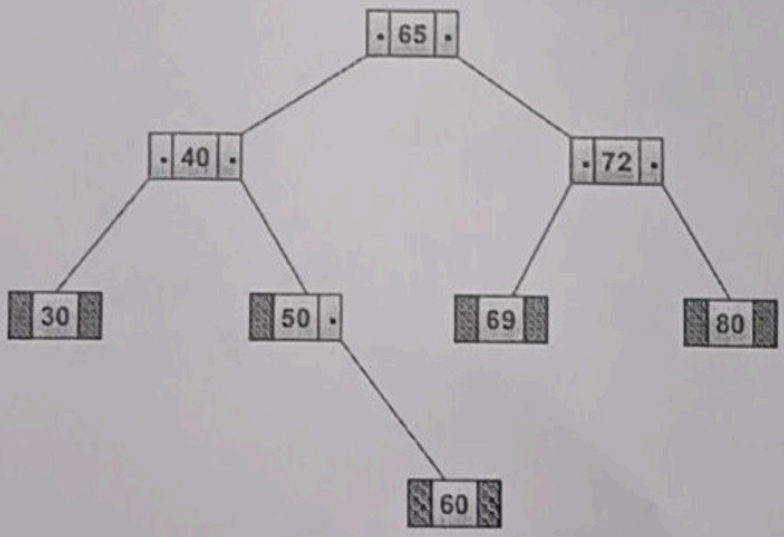
- **Insertion in a B-tree of odd order**
- Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



Q.13) **Write a short note Threaded binary tree with example.**

- ✓ A binary tree can be represented by using array representation or linked list representation.
- ✓ When a binary tree is represented using linked list representation.
- ✓ If any node is not having a child we use a NULL pointer.
- ✓ These special pointers are threaded and the binary tree having such pointers is called a threaded binary tree.

- ✓ Thread in a binary tree is represented by a dotted line.
- ✓ Consider the following binary search tree.
- ✓ Most of the nodes in this tree hold a NULL value in their left or right child fields.
- ✓ In this case, it would be good if these NULL fields are utilized for some other useful purpose

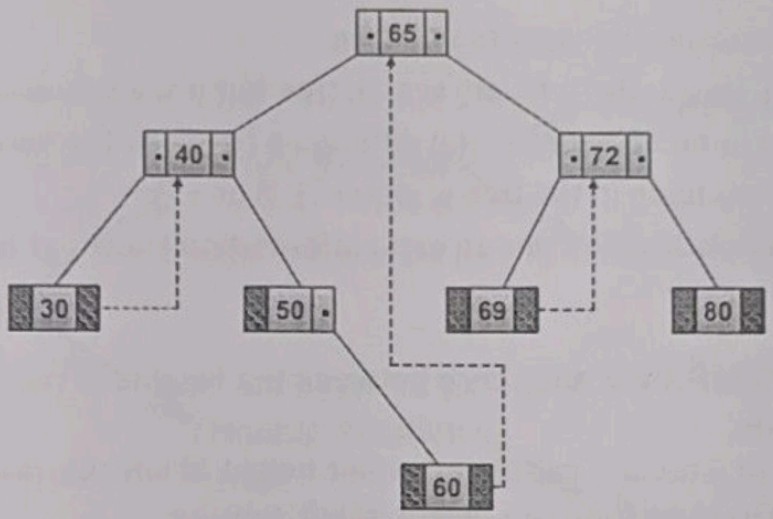


- ✓ In the threaded binary tree when there is only one thread is used then it is called as **one way threaded tree** and when both the threads are used then it is called the **two way threaded tree.**

❖ **One Way Threaded Binary Trees**

- ✓ The empty left child field of a node can be used to point to its inorder predecessor.
- ✓ Similarly, the empty right child field of a node can be used to point to its in-order successor.
- ✓ Such a type of binary tree is known as a one way threaded binary tree.

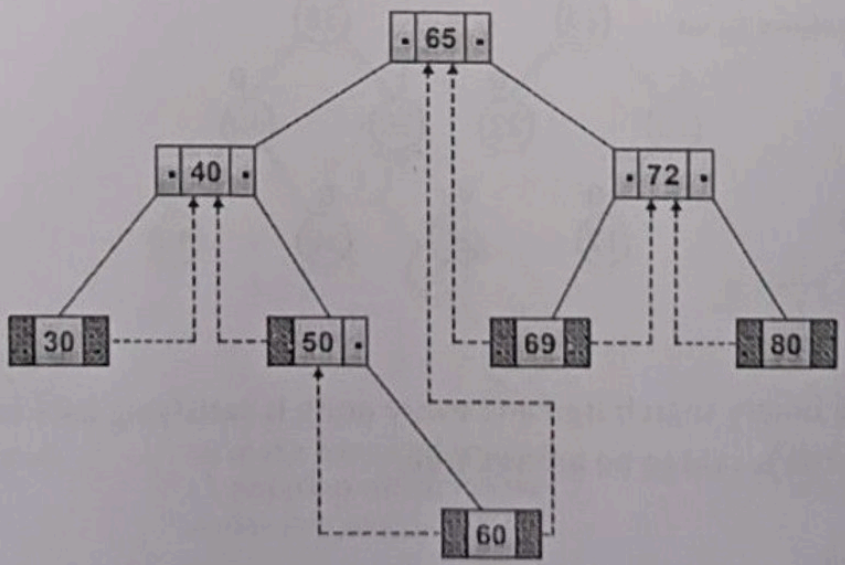
In-order :- 30 40 50 60 65 69 72 80



❖ **Two way Threaded Binary Trees:**

- ✓ A field that holds the address of its inorder successor or predecessor is known as thread.
- ✓ The empty left child field of a node can be used to point to its inorder predecessor.

Inorder :- 30 40 50 60 65 69 72 80

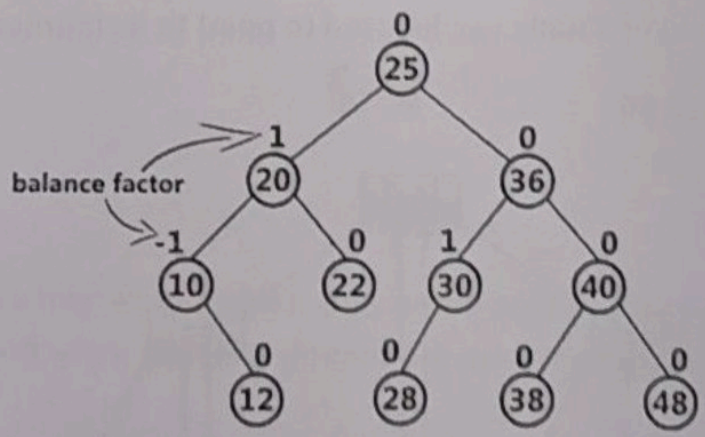


Q.14) Explain AVL Tree(Height Balance Tree) with example.

- ✓ AVL tree is a height-balanced binary search tree.
- ✓ That means, an AVL tree is also a binary search tree but it is a balanced tree.
- ✓ A binary tree is said to be balanced if, the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.
- ✓ In an AVL tree, every node maintains an extra information known as balance factor.

- ✓ Balance factor of a node is the difference between the heights of the left and right subtrees of that node.
- ✓ The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree.

Balance factor = heightOfLeftSubtree – heightOfRightSubtree

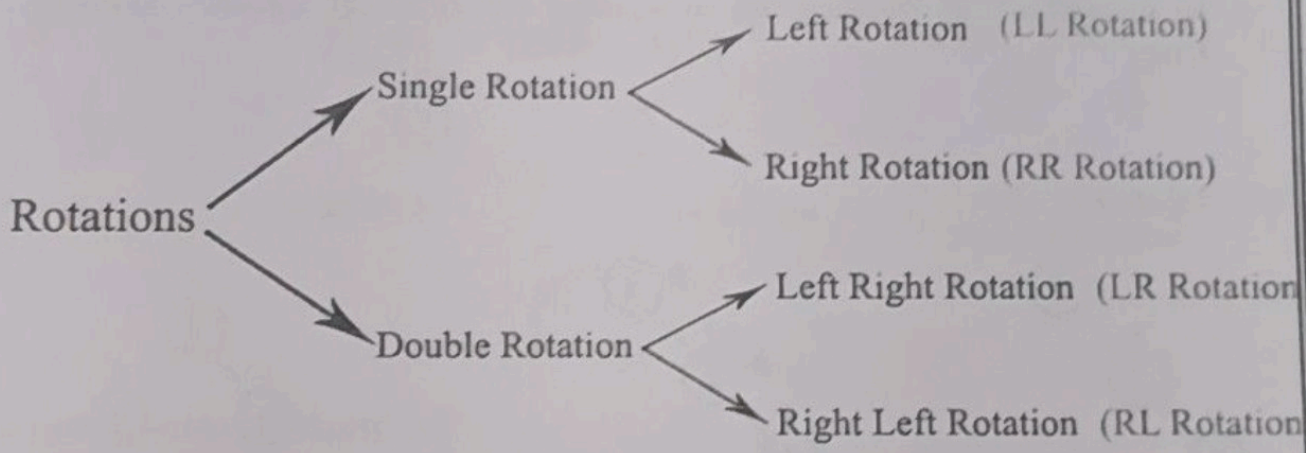


- ✓ The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an **AVL tree.**

❖ **AVL Tree Rotations**

- ✓ Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.

✓ There are four rotations and they are classified into two types.

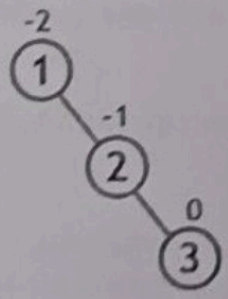


1) Single Left Rotation (LL Rotation)

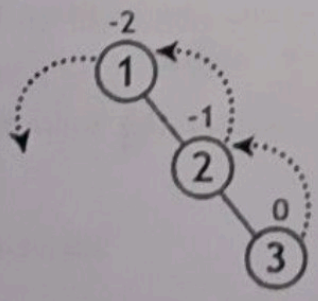
✓ In LL Rotation, every node moves one position to left from the current position.

✓ To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

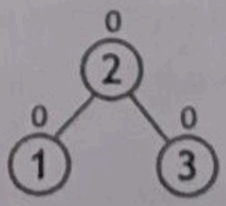
insert 1, 2 and 3



Tree is imbalanced



To make balanced we use LL Rotation which moves nodes one position to left

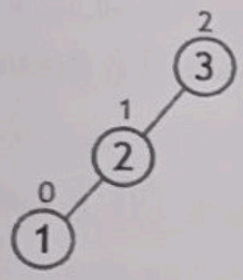


After LL Rotation Tree is Balanced

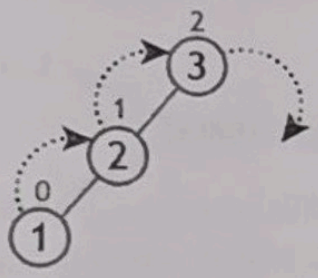
2) Single Right Rotation (RR Rotation)

- ✓ In RR Rotation, every node moves one position to right from the current position.
- ✓ To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

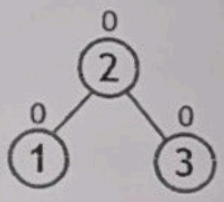
insert 3, 2 and 1



Tree is imbalanced
because node 3 has balance factor 2



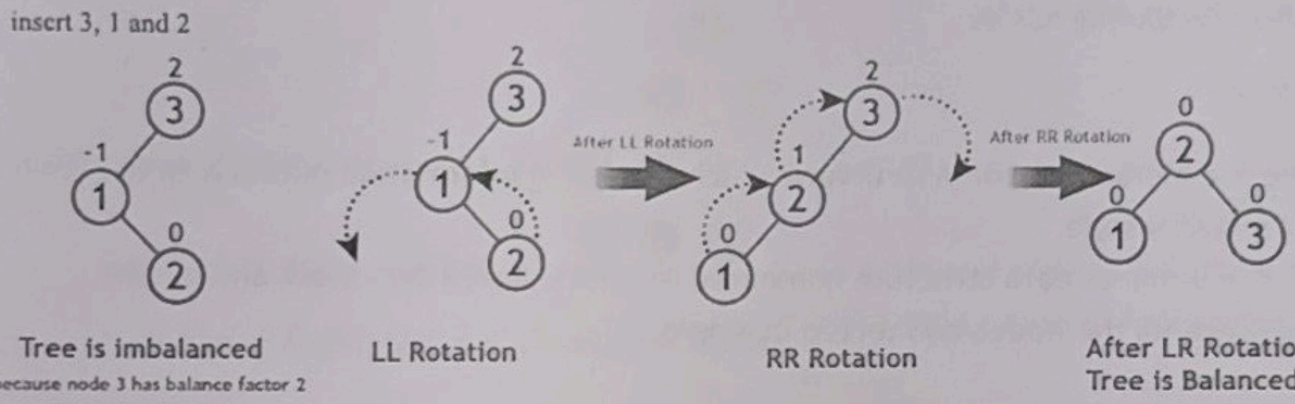
To make balanced we use RR Rotation which moves nodes one position to right



After RR Rotation Tree is Balanced

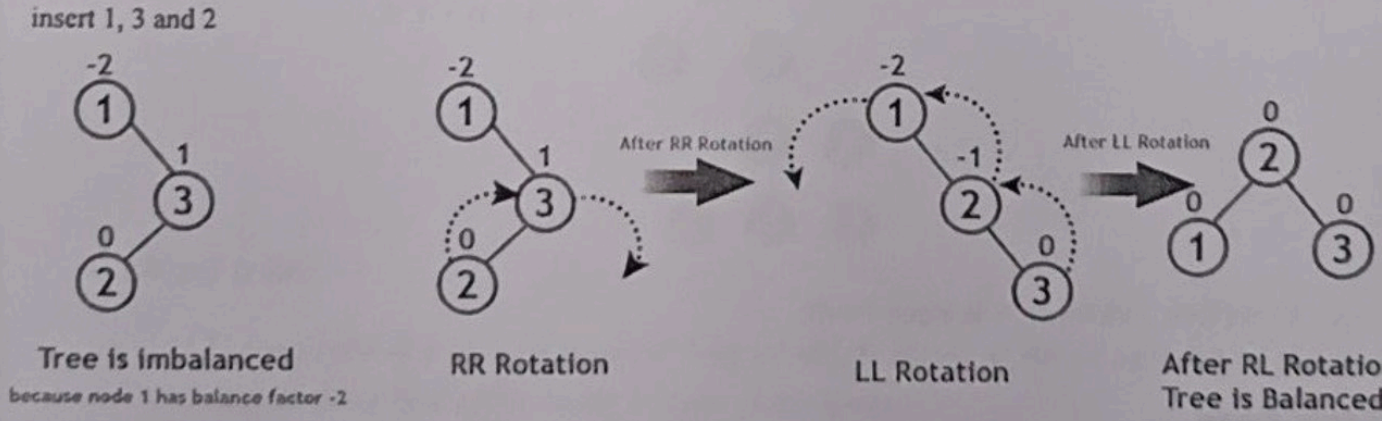
3) Left Right Rotation (LR Rotation)

- ✓ The LR Rotation is a sequence of single left rotation followed by a single right rotation.
- ✓ In LR Rotation, at first, every node moves one position to the left and one position to right from the current position.
- ✓ To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



4) Right Left Rotation (RL Rotation)

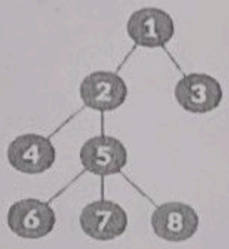
- ✓ The RL Rotation is sequence of single right rotation followed by single left rotation.
- ✓ In RL Rotation, at first every node moves one position to right and one position to left from the current position.
- ✓ To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Q.15) Define following terms

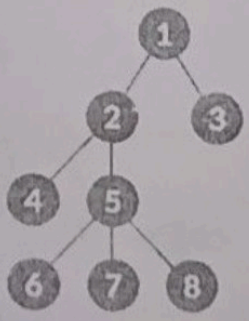
1) Tree:

- ✓ Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- ✓ It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- ✓ It represents the nodes connected by edges.



2) Root Node:

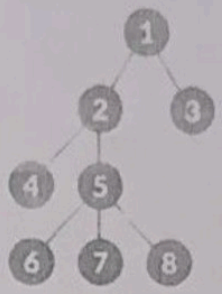
- ✓ The root node is the topmost node in the tree hierarchy.
- ✓ In other words, the root node is the one which doesn't have any parent.



- ✓ In Above Example 1 is root node.

3) Leaf Node

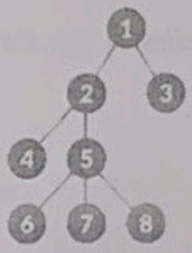
- ✓ The node which does not have any child node is called the leaf node.
- ✓ Leaf node is the bottom most node of the tree.



- ✓ In Above Example 6,7 ,8 all are Leaf Node

4) Path:

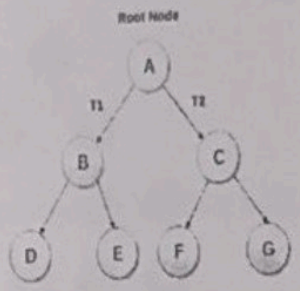
- ✓ Path refers to the sequence of nodes along the edges of a tree.



- ✓ In Above Example 1-2-4 is a path
  1-2-5-7 is a path

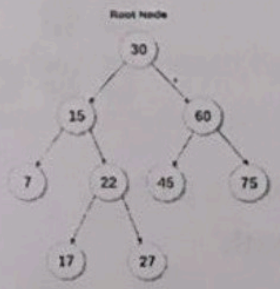
5) Binary tree:

- ✓ A binary tree is a special type of tree in which every node or vertex has either no child node or one child node or two child nodes.



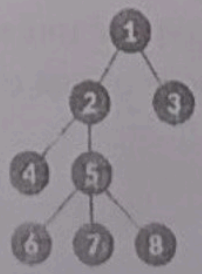
6) Binary Search tree:

- ✓ Binary Search tree in which the nodes are arranged in a specific order.
- ✓ This is also called ordered binary tree.
- ✓ In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
- ✓ Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.



7) Degree of a node:

- ✓ In a tree data structure the total number of child of a node is called as degree



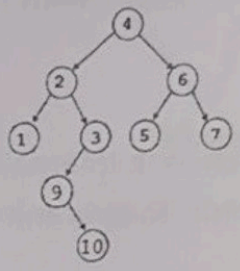
Degree of 1 is 2

Degree of 3 is o
Degree of 5 is 3

8) Level of a node:

- ✓ In a tree data structure the root node is said to be level 0 and the children of root node are at Level 1 and so.

Find Level/Height of Node



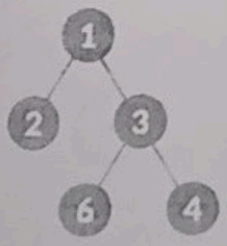

```
Input: Node 5
Output: Node 5 is at level 2.

Input: Node 10
Output: Node 10 is at level 4.

Input: Node 4
Output: Node 4 is at level 0.
```

9) Full Binary tree:

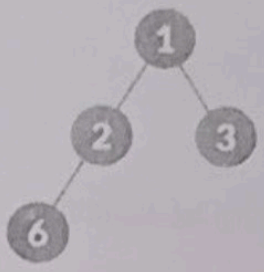
- ✓ A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.



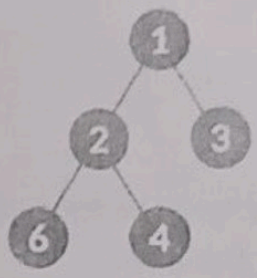
✓ Full Binary Tree

10) Complete Binary tree:

- ✓ A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



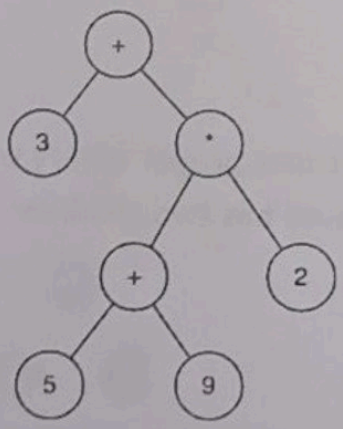
✗ Full Binary Tree
✓ Complete Binary Tree



✓ Full Binary Tree
✓ Complete Binary Tree

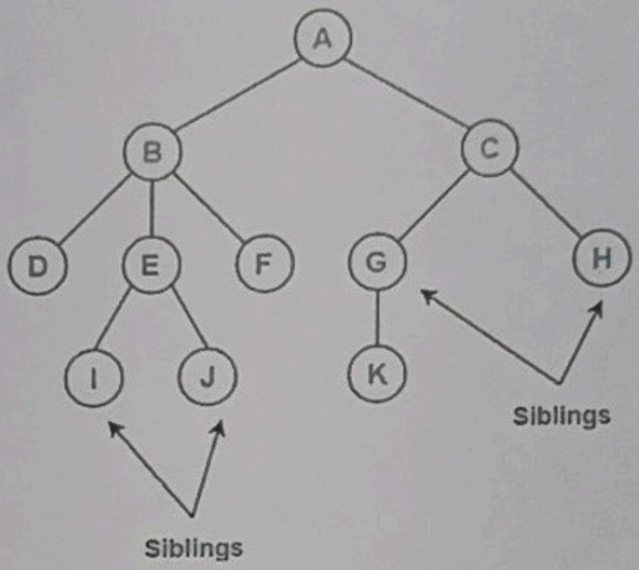
11) Expression Tree:

- ✓ Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



12) Sibling:

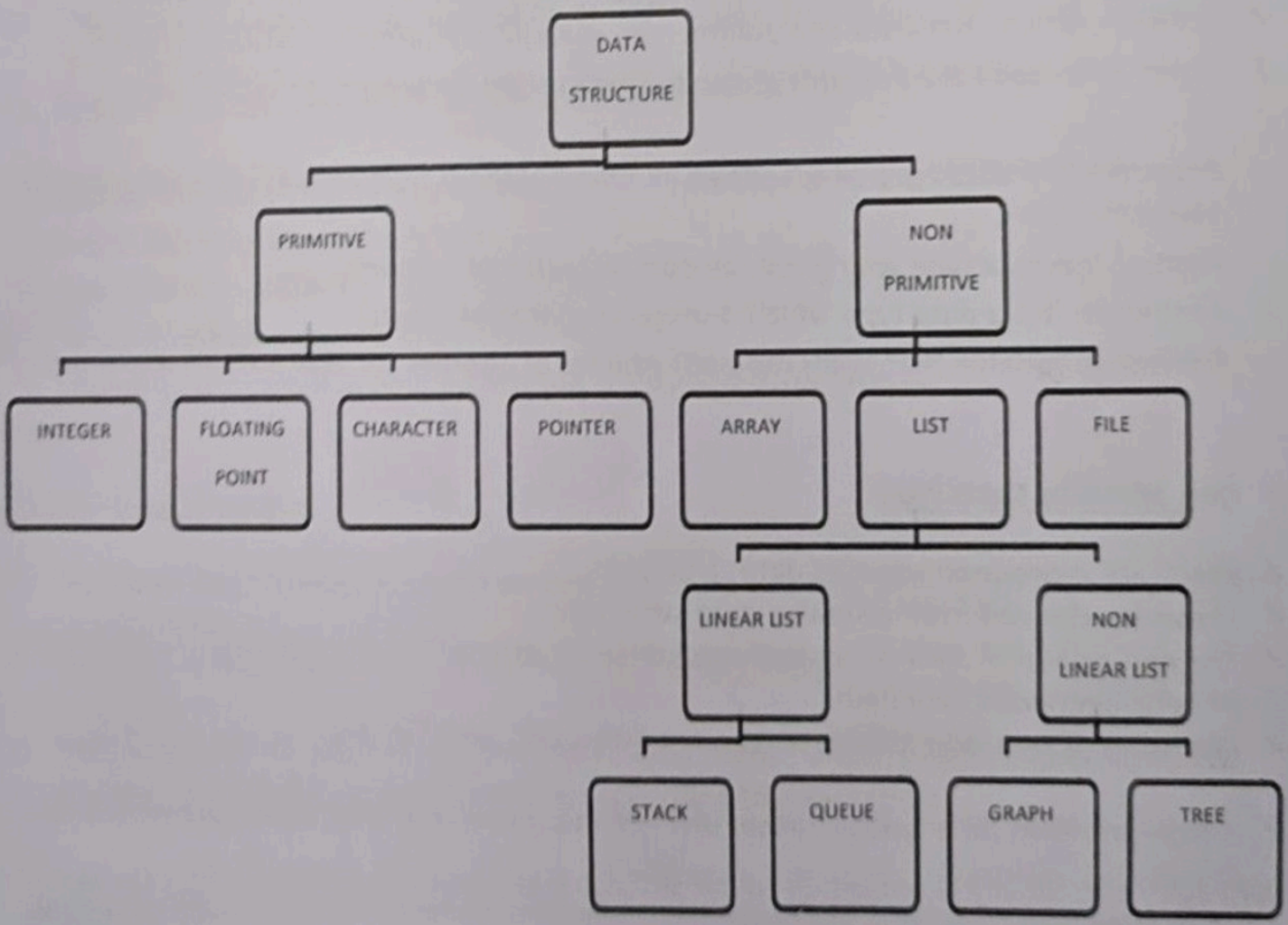
- ✓ Two nodes are said to be **siblings** if they are present at the same level, and their parents are same.


A
B
C
D
E
F
G
H
I
J
K
Siblings
Siblings

Q.16) what is Data Structure? Explain classification of Data Structure.

- ✓ Data Structure is a way to store and organize data so that it can be used efficiently.
- ✓ Data Structure such as Array, Pointer, Structure, Linked List, Stack, Queue, Graph, Searching, Sorting, Programs, etc.

Classification of Data Structure



- ✓ Data Structures are normally classified into two broad categories

1. Primitive Data Structure
2. Non-primitive data Structure

➢ Primitive Data Structure

- ✓ Primitive data structures are basic structures and are directly operated upon by machine instructions.
- ✓ Primitive data structures have different representations on different computers.
- ✓ Integers, floats, character and pointers are examples of primitive data structures.
- ✓ These data types are available in most programming languages as built in type.

- **Integer**: It is a data type which allows all values without fraction part. We can use it for whole numbers.
- **Float**: It is a data type which use for storing fractional numbers.
- **Character**: It is a data type which is used for character values.
- **Pointer**: A variable that holds memory address of another variable are called pointer.

➢ Non primitive Data Type

- ✓ These are more sophisticated data structures.
- ✓ These are derived from primitive data structures.
- ✓ The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- ✓ Examples of Non-primitive data type are Array, List, and File etc.

- ✓ A Non-primitive data type is further divided into Linear and Non-Linear data structure

- **Array:** An array is a fixed-size sequenced collection of elements of the same data type.
- **List:** An ordered set containing variable number of elements is called as Lists.
- **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

➢ **Linear data structures**

- ✓ A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- ✓ Examples of Linear Data Structure are Stack and Queue.
- ✓ **Stack:** Stack is a data structure in which insertion and deletion operations are performed at one end only.
  - The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation.
  - Stack is also called as Last in First out (LIFO) data structure.
- ✓ **Queue:** The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
  - End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
  - Queue is also called as First in First out (FIFO) data structure.

Nonlinear data structures

- ✓ Nonlinear data structures are those data structure in which data items are not arranged in a sequence.
- ✓ Examples of Non-linear Data Structure are Tree and Graph.
- ✓ **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.
- ✓ Tree consists of nodes connected by edge, the node represented by circle and edge lives connecting to circle.
- ✓ **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.

Q.17) Explain Sparce Matrix with example.

What is Sparse Matrix?

- ✓ In computer programming, a matrix can be defined with a 2-dimensional array.
- ✓ Any array with 'm' columns and 'n' rows represents a mXn matrix.
- ✓ There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.
- ✓ Sparse matrix is a matrix which contains very few non-zero elements.
- ✓ When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements.
- ✓ In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

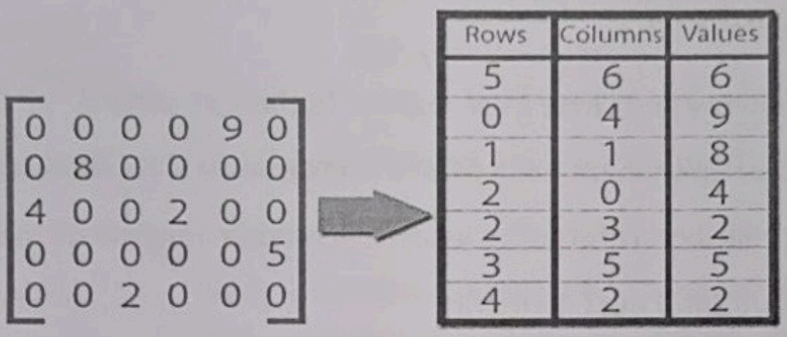
➢ **Sparse Matrix Representations**

- ✓ A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation
2. Linked Representation

Method 1 : Triplet Representation (Array Representation)

- ✓ In this representation, we consider only non-zero values along with their row and column index values.
- ✓ In this representation, the 0^{th} row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.
- ✓ For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...



0	0	0	0	9	0
0	8	0	0	0	0
4	0	0	2	0	0
0	0	0	0	0	5
0	0	2	0	0	0

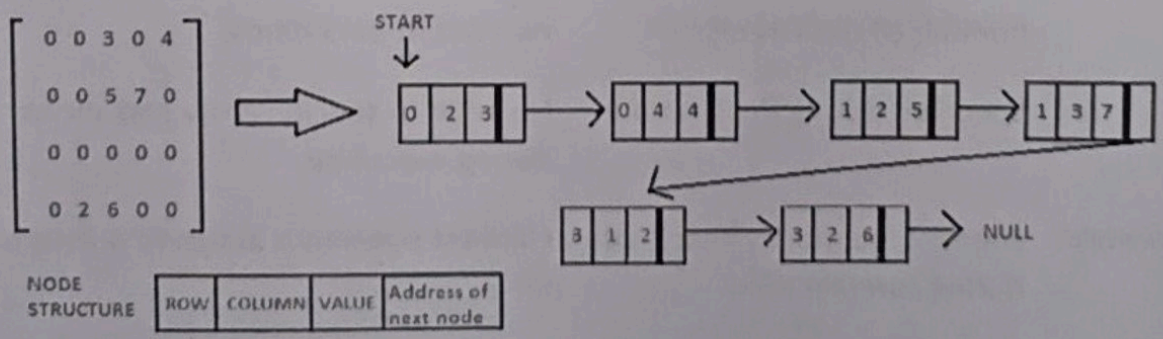
Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

- ✓ In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6.
- ✓ We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values.
- ✓ The second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix.
- ✓ In the same way, the remaining non-zero values also follow a similar pattern.

Method 2: Using Linked Lists

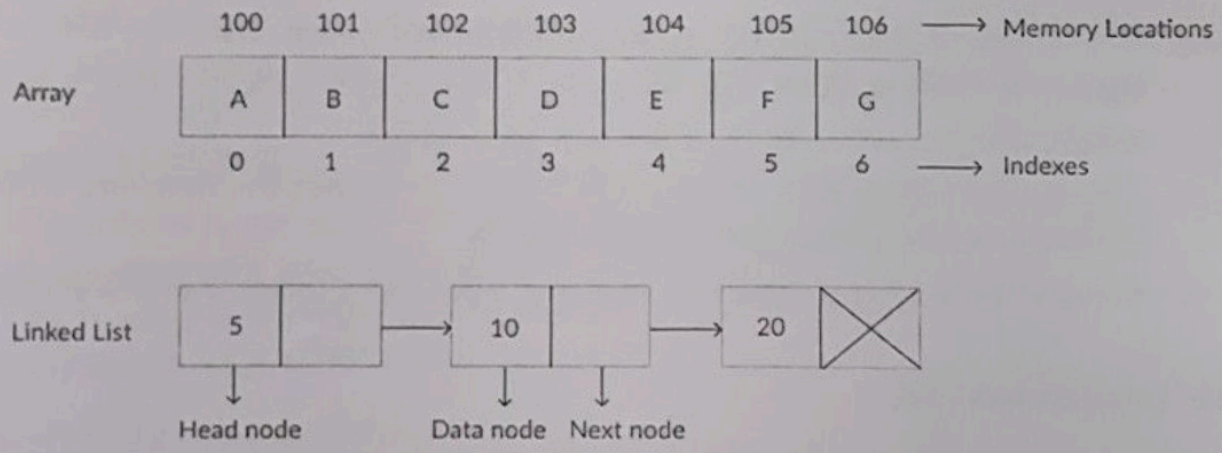
In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



Q.18) Difference between Array and linked list.

- ✓ The major difference between **Array** and **Linked list** regards to their structure.
- ✓ Arrays are **index based** data structure where each element associated with an index.
- ✓ On the other hand, Linked list relies on **references** where each node consists of the data and the references to the previous and next element.

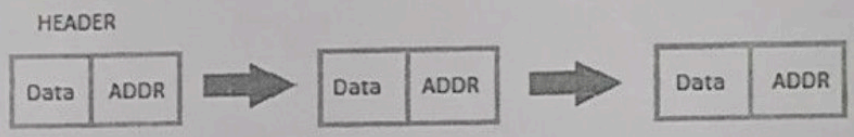


BASIS FOR COMPARISON	ARRAY	LINKED LIST
Basic	It is a consistent set of a fixed number of data items.	It is an ordered set comprising a variable number of data items.
Size	Specified during declaration.	No need to specify; grow and shrink during execution.
Storage Allocation	Element location is allocated during compile time.	Element position is assigned during run time.

BASIS FOR COMPARISON	ARRAY	LINKED LIST
Order of the elements	Stored consecutively	Stored randomly
Accessing the element	Direct or randomly accessed, i.e., Specify the array index or subscript.	Sequentially accessed, i.e., Traverse starting from the first node in the list by the pointer.
Insertion and deletion of element	Slow relatively as shifting is required.	Easier, fast and efficient.
Searching	Binary search and linear search	linear search
Memory required	less	More
Memory Utilization	Ineffective	Efficient
Types	It can be single dimensional, two dimensional or multi-dimensional.	It can be singly, doubly or circular linked list.
Extra Space	Extra space is not required.	To link nodes of list, pointers are used which require extra Space.

Q.19) what is linked list? Explain types of linked list with diagram.

- ✓ Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.
- ✓ Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.
- ✓ Linked Lists are used to create trees and graphs.



❖ **Advantages of Linked Lists**

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

❖ **Disadvantages of Linked Lists**

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Types of Linked Lists

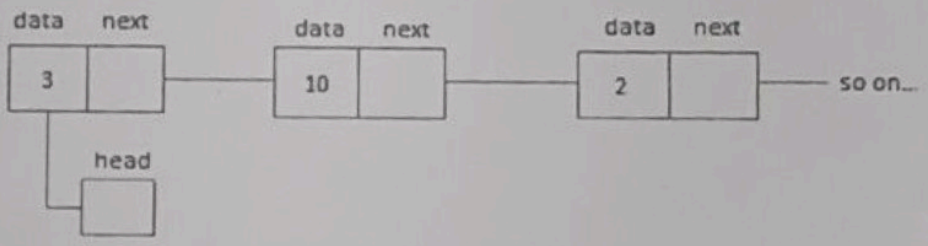
- ✓ There are 3 different implementations of Linked List available, they are:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

❖ Singly Linked List

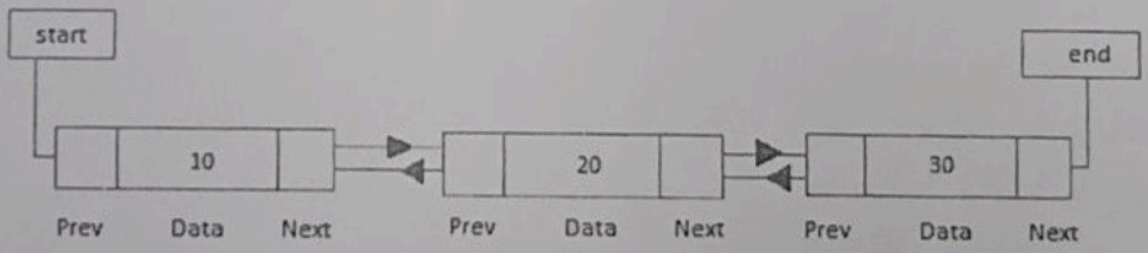
✓ Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. next, which points to the next node in the sequence of nodes.

✓ The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



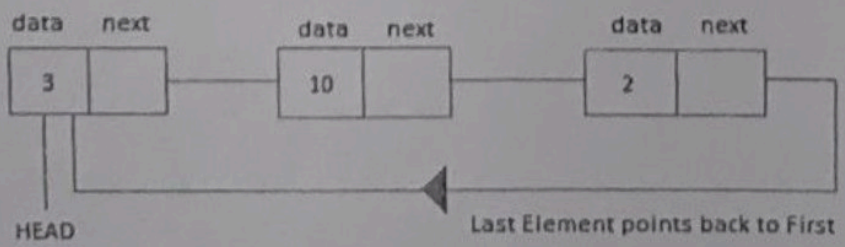
❖ Doubly Linked List

✓ In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.



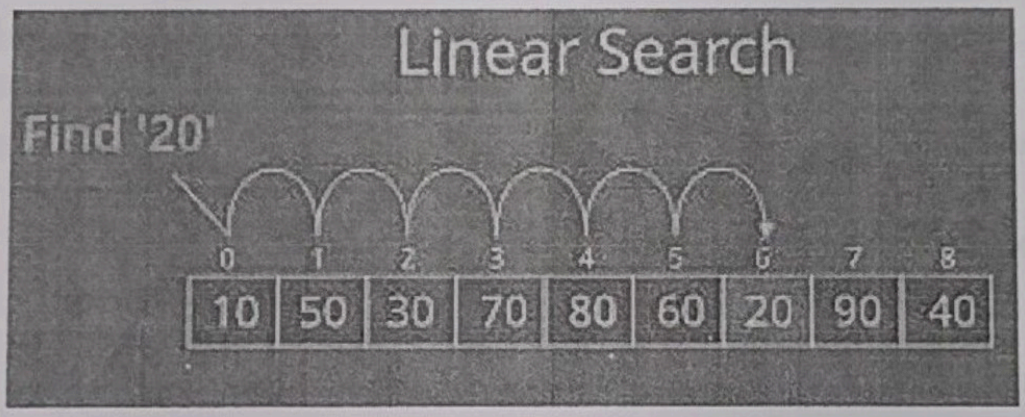
❖ Circular Linked List

✓ In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.



Q.20) write an algorithm of sequential search.

- ✓ Linear search is a very simple search algorithm.
- ✓ In this type of search, a sequential search is made over all items one by one.
- ✓ Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



❖ **Algorithm of Linear Search:**

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Q.21) write an algorithm of binary search.

- ✓ Binary search is a fast search algorithm.
- ✓ This search algorithm works on the principle of divide and conquer.
- ✓ Binary search looks for a particular item by comparing the middle most item of the collection.
- ✓ If a match occurs, then the index of item is returned.
- ✓ If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item.

❖ **Function BINARY_SEARCH(K,N,X)**

K=Array
N=Size of the array
X= Element to be searched
LOW, HIGH, MIDDLE= limits of search

1) **[Initialize]**
LOW←1
HIGH←N

2) **[Perform Search]**
Repeat through step 4 while LOW<= HIGH

3) **[obtain midpoint]**
MIDDLE← FLOOR(LOW+HIGH)/2

4) **[compare]**
If X<K[MIIDDLE]
Then
HIGH←MIDDLE-1
Else

```
If X>K[MIDDLE]
Then
        LOW←MIDDLE+1
Else
        Write("Successful Search")
        Return(MIDDLE)
```

5) **[Unsuccessful Search]**

```
Write("Unsuccessful Search")
Return(0)
```

Q.22) Sort the following data using bubble sort technique.

6, 5, 3, 1, 8, 7, 2, 4

n = 8

0	1	2	3	4	5	6	7	
6	5	3	1	8	7	2	4	j = 1
5	6	3	1	8	7	2	4	j = 2
5	3	6	1	8	7	2	4	j = 3
5	3	1	6	8	7	2	4	j = 4
5	3	1	6	8	7	2	4	j = 5
5	3	1	6	7	8	2	4	j = 6
5	3	1	6	7	2	8	4	j = 7

n = 7

0	1	2	3	4	5	6	7	
5	3	1	6	7	2	4	8	j = 1
3	5	1	6	7	2	4	8	j = 2
3	1	5	6	7	2	4	8	j = 3
3	1	5	6	7	2	4	8	j = 4
3	1	5	6	7	2	4	8	j = 5
3	1	5	6	2	7	4	8	j = 6

n = 6

0	1	2	3	4	5	6	7	
3	1	5	6	2	4	7	8	j = 1
1	3	5	6	2	4	7	8	j = 2
1	3	5	6	2	4	7	8	j = 3
1	3	5	6	2	4	7	8	j = 4
1	3	5	2	6	4	7	8	j = 5

n = 5

0	1	2	3	4	5	6	7	
1	3	5	2	4	6	7	8	j = 1
1	3	5	2	4	6	7	8	j = 2
1	3	5	2	4	6	7	8	j = 3
1	3	2	5	4	6	7	8	j = 4

n = 4

0	1	2	3	4	5	6	7	
1	3	2	4	5	6	7	8	j = 1
1	3	2	4	5	6	7	8	j = 2
1	2	3	4	5	6	7	8	j = 3

n = 3

0	1	2	3	4	5	6	7	
1	2	3	4	5	6	7	8	j = 1
1	2	3	4	5	6	7	8	j = 2

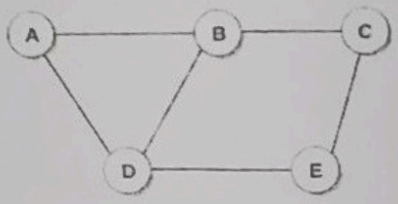
n = 2

0	1	2	3	4	5	6	7	
1	2	3	4	5	6	7	8	j = 1

Q.23) Defines the following terms (Graph)

- **Graph:**

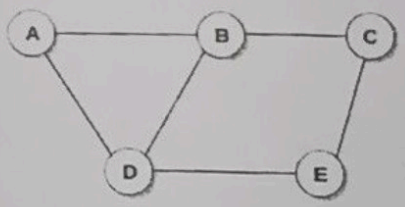
A graph is a common data structure that consists of a finite set of nodes (or vertices) and a set of edges connecting them.



❖ **Adjacent Nodes**

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbors' or adjacent nodes.

In below graph A and B are adjacent nodes.

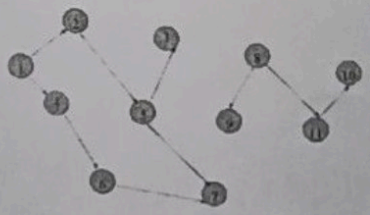


❖

❖ **Indgree of vertex:**

Indegree of vertex V is the number of edges which are coming into the **vertex** V.

In-Degree Of A Vertex

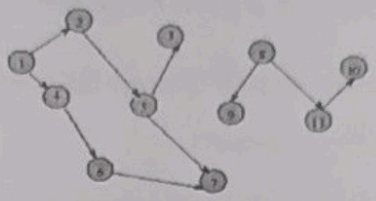


in-degree is number of incoming edges
indegree(2) = 1, indegree(8) = 0

❖ **Outdegree of vertex:**

Outdegree of vertex V is the number of edges which are going out from the **vertex** V.

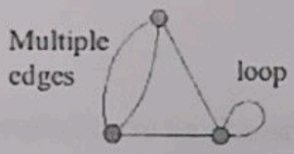
Out-Degree of a Vertex



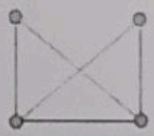
- **Out-degree** of vertex *i* is the number of edges incident from *i* (i.e., the number of outgoing edges).
- e.g., outdegree(2) = 1, outdegree(8) = 2

❖ Simple graph

A graph has no loops or multiple edges.



It is not simple.

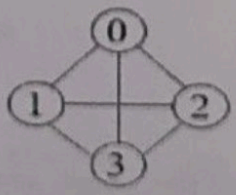


It is a simple graph.

❖ Complete Graph

A complete graph is the one in which every node is connected with all other nodes.

A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.



G_1

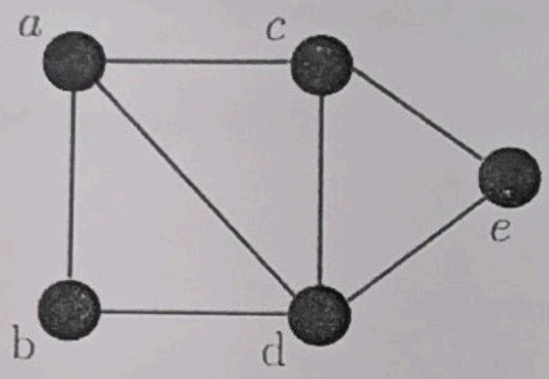
complete graph

❖ **Connected Graph**

A graph is said to be connected if there is a path between every pair of vertex.

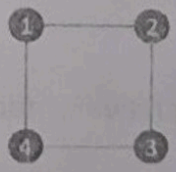
From every vertex to any other vertex, there should be some path to traverse. That is called the connected graph.

A graph with multiple disconnected vertices and edges is said to be **disconnected.**



❖ **Undirected:**

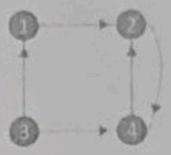
- An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.



Undirected Graph

❖ **Directed:**

- A directed graph is a graph in which all the edges are uni-directional i.e. the edges point in a single direction.

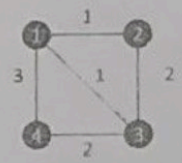


Directed Graph

❖ **Weighted graph:**

In a weighted graph, each edge is assigned a weight or cost.

Consider a graph of 4 nodes as in the diagram below. As you can see each edge has a weight/cost assigned to it.

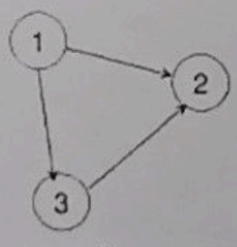


Weighted Graph

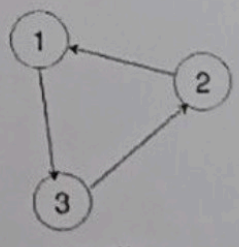
❖ **Cyclic:**

A graph is cyclic if the graph comprises a path that starts from a vertex and ends at the same vertex.

That path is called a cycle. An **acyclic graph** is a graph that has no cycle.



acyclic



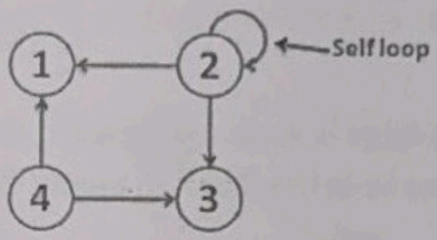
cyclic

❖ **Parallel Edges:**

- If two vertices are connected with more than one edge than such edges are called parallel edges that is many roots but one destination.

❖ **Loop(Sling):**

An edge of a graph which join a vertex to itself is called loop or a self-loop.



Graph with self loop

❖ **Isolated Vertex**

A vertex with degree zero is called an isolated vertex.

Example

Q.24) Explain different representation of graph in detail.

- ✓ In graph theory, a graph representation is a technique to store graph into the memory of computer.
- ✓ To represent a graph, we just need the set of vertices, and for each vertex the neighbors of the vertex (vertices which is directly connected to it by an edge). If it is a weighted graph, then the weight will be associated with each edge.

- ✓ There are different ways to optimally represent a graph, depending on the density of its edges, type of operations to be performed and ease of use.

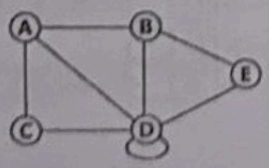
- ✓ **Graph data structure is represented using following representations...**

1. **Adjacency Matrix**
2. **Adjacency List**

❖ **Adjacency Matrix**

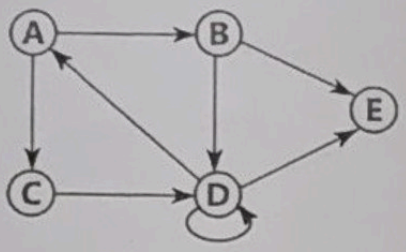
- ✓ In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices.
- ✓ That means a graph with 4 vertices is represented using a matrix of size 4X4.
- ✓ In this matrix, both rows and columns represent vertices.
- ✓ This matrix is filled with either 1 or 0.
- ✓ Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

- ✓ For example, consider the following undirected graph representation...



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

Directed graph representation...



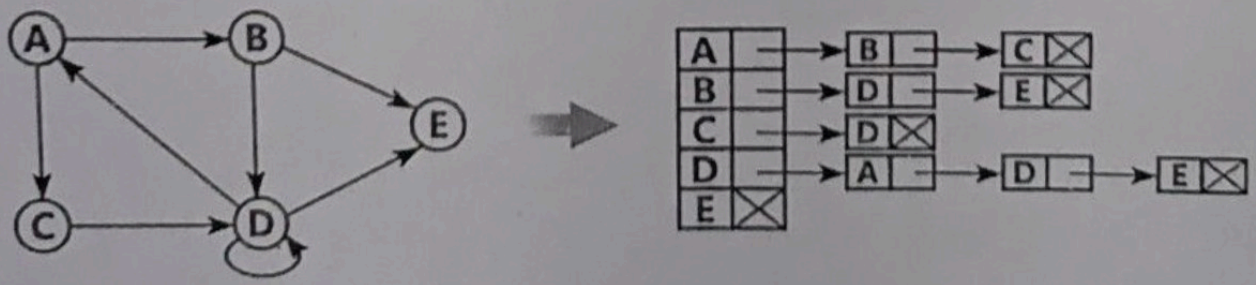
	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

❖ **Adjacency List**

- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a **singly linked list** of neighbors of v.

Example

Let's see the following directed graph representation implemented using linked list:

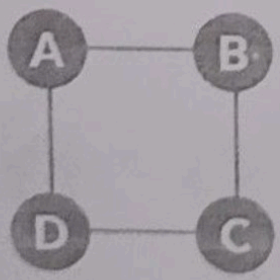


Q.25) What is Spanning Tree? Explain minimum spanning tree with example.

- ✓ A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges.
- ✓ If a vertex is missed, then it is not a spanning tree.
- ✓ The edges may or may not have weights assigned to them.
- ✓ The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n^{(n-2)}$.
- ✓ If we have n = 4, the maximum number of possible spanning trees is equal to $4^{4-2} = 16$. Thus, 16 spanning trees can be formed from a complete graph with 4 vertices.

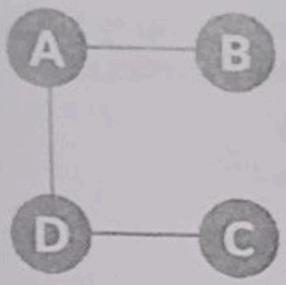
❖ **Example of a Spanning Tree**

- ✓ Let's understand the spanning tree with examples below:
- ✓ Let the original graph be:

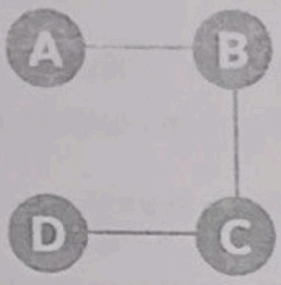


Normal graph

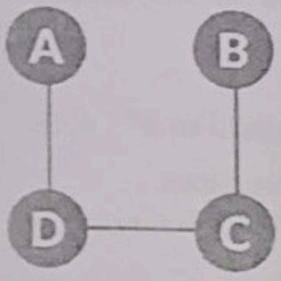
- ✓ Some of the possible spanning trees that can be created from the above graph are:



A spanning tree



A spanning tree



Minimum Spanning Tree

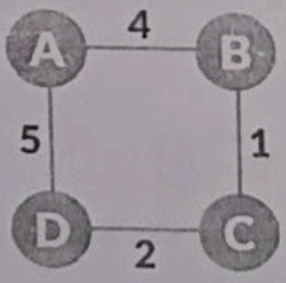
A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

Example of a Spanning Tree

Let's understand the above definition with the help of the example below.

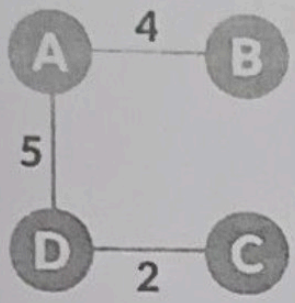
The initial graph is:

The initial graph is:



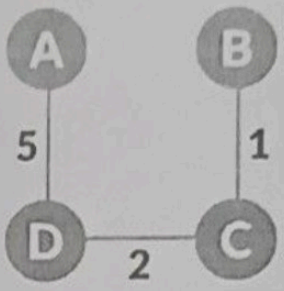
Weighted graph

The possible spanning trees from the above graph are:



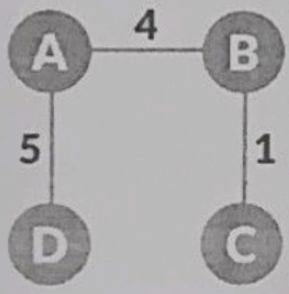
sum = 11

Minimum spanning tree - 1



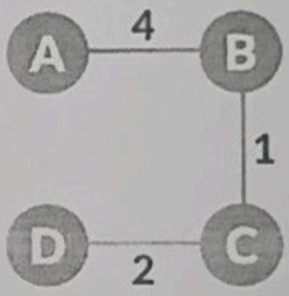
sum = 8

Minimum spanning tree - 2



sum = 10

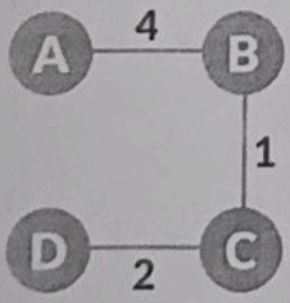
Minimum spanning tree - 3



sum = 7

Minimum spanning tree - 4

The minimum spanning tree from the above spanning trees is:



sum = 7

Minimum spanning tree

Q.26) Explain Prims's algorithm with example.

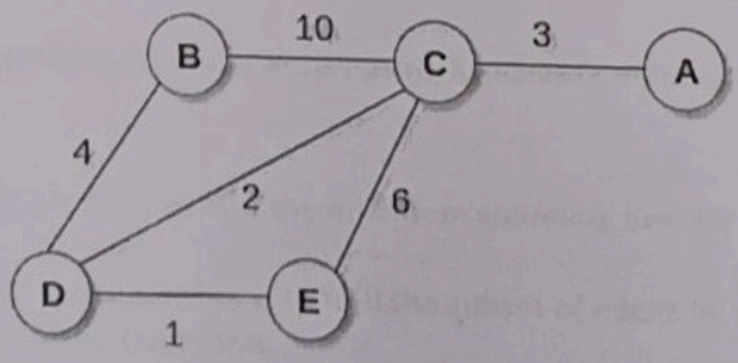
- ✓ Prim's Algorithm is used to find the minimum spanning tree from a graph.
- ✓ Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
- ✓
- ✓ Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step.
- ✓ The edges with the minimal weights causing no cycles in the graph got selected.
- ✓ The algorithm is given as follows.

Algorithm

- **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

Example :

- ✓ Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.



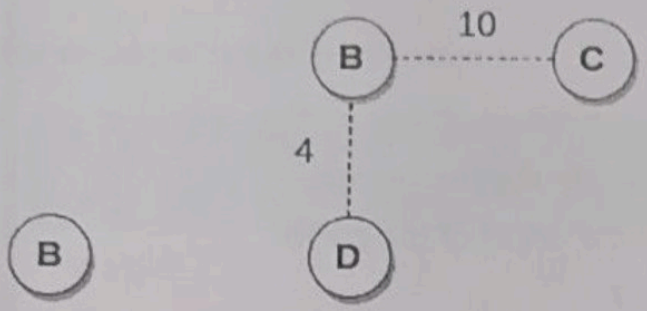
Solution

- **Step 1 :** Choose a starting vertex B.
- **Step 2:** Add the vertices that are adjacent to A. 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 3:** 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 4:** Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

✓ The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.

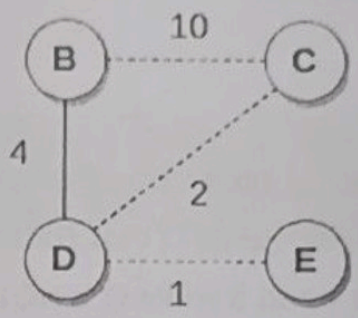
The cost of MST will be calculated as;

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

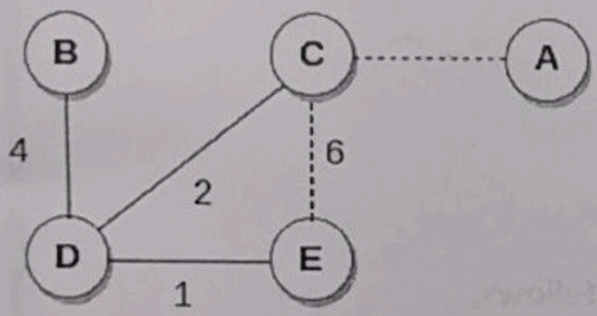


Step 1

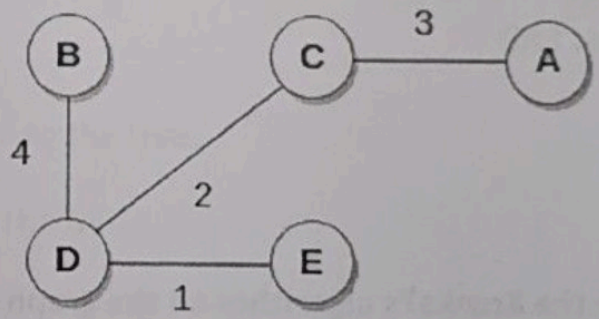
Step 2



Step 3



Step 4



Step 5

Q.27) Find minimum spanning tree of a graph using Krushkal's algorithm with proper example.

- ✓ Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph.
- ✓ The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph.
- ✓ Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

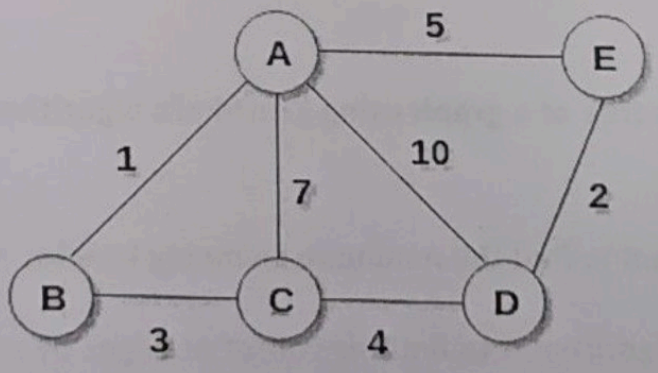
The Kruskal's algorithm is given as follows.

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

Example :

- **Apply the Kruskal's algorithm on the graph given as follows.**



Solution:

the weight of the edges given as :

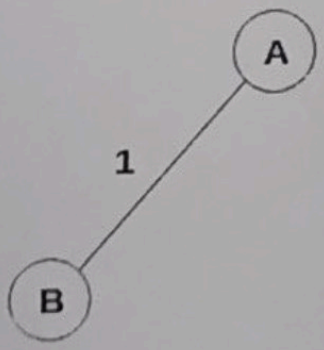
Edge	AE	AD	AC	AB	BC	CD	DE
Weight	5	10	7	1	3	4	2

Sort the edges according to their weights.

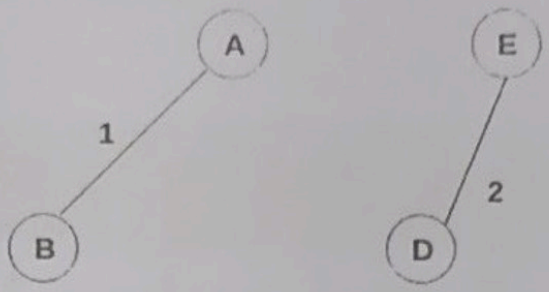
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Start constructing the tree;

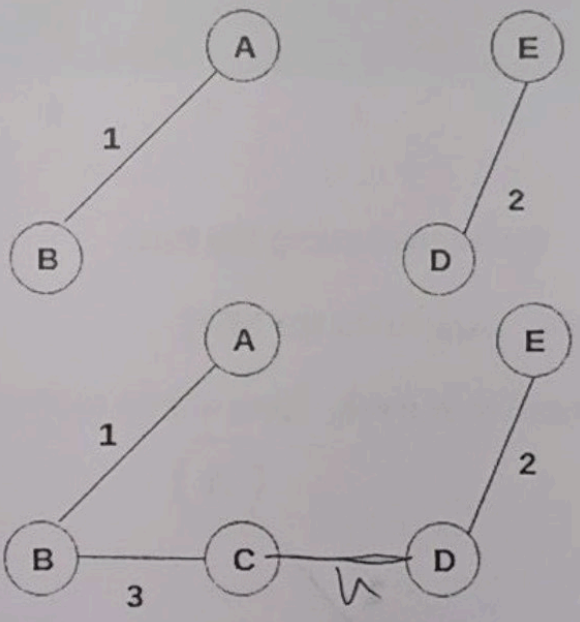
Add AB to the MST;



Add DE to the MST;



Add BC to the 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.

Hence, the final MST is the one which is shown in the step 4.

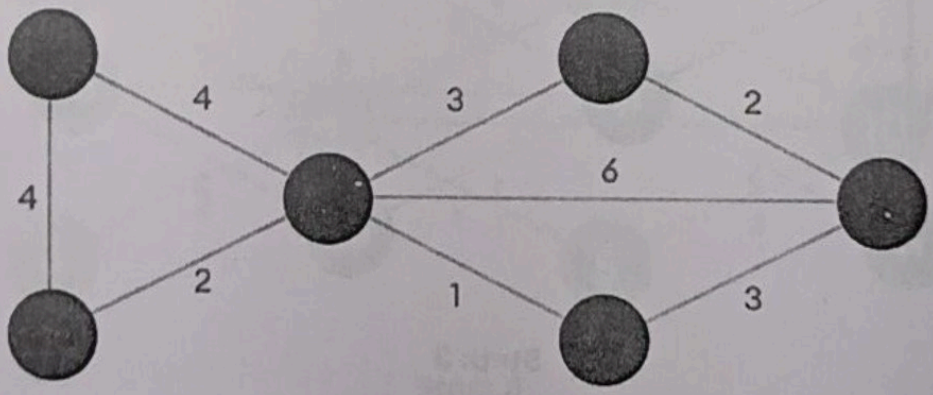
the cost of MST = 1 + 2 + 3 + 4 = 10.

Q.28) Explain Dijkstra's algorithm with proper example.

- ✓ Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.
- ✓ It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.
- ✓ Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex.
- ✓ Then we visit each node and its neighbors to find the shortest subpath to those neighbors.
- ✓ The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

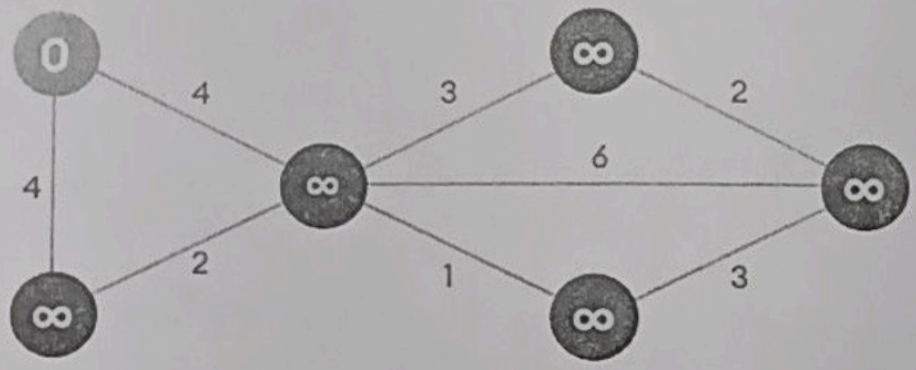
Example of Dijkstra's algorithm

- ✓ It is easier to start with an example and then think about the algorithm.



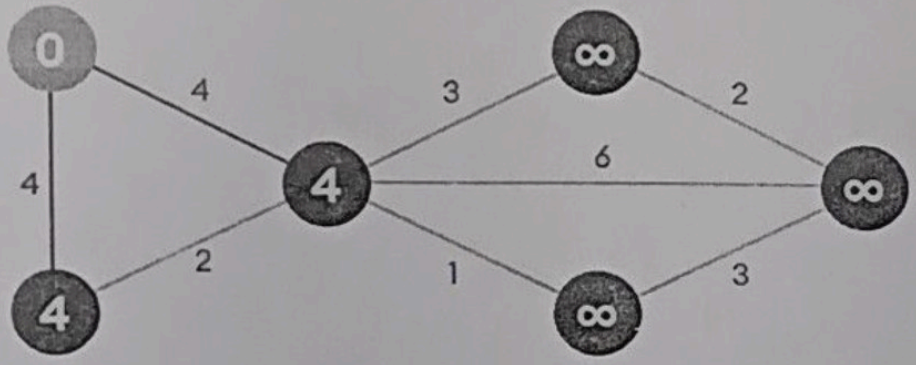
Step: 1

Start with a weighted graph



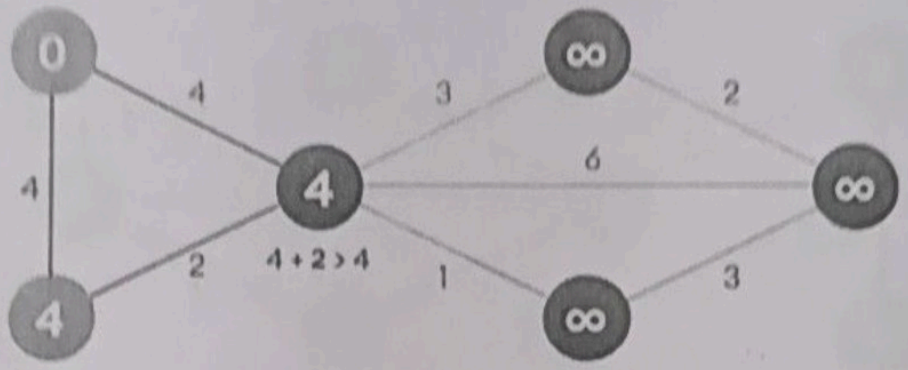
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



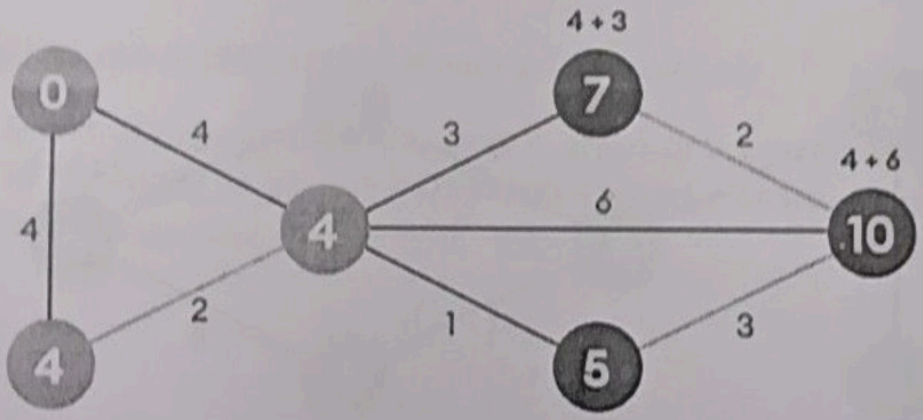
Step: 3

Go to each vertex and update its path length



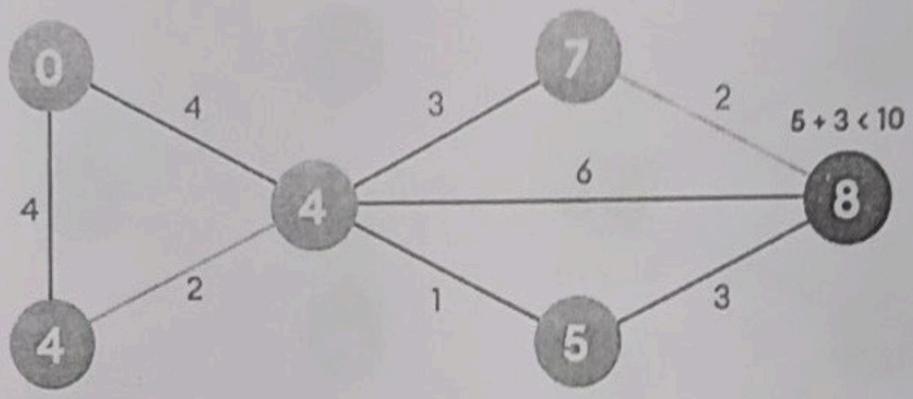
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



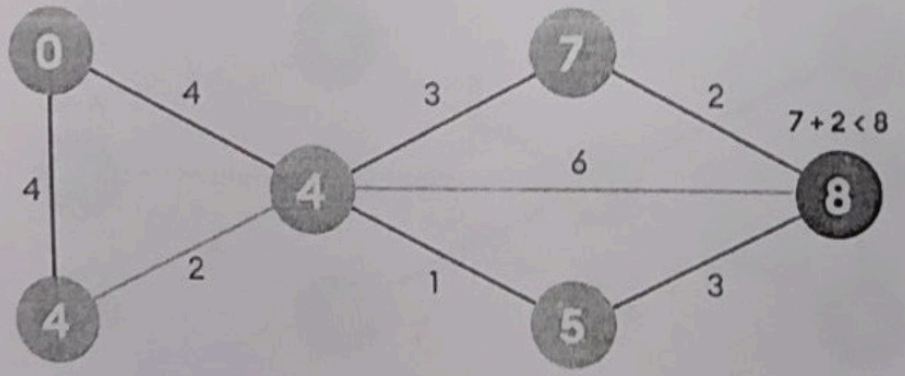
Step: 5

Avoid updating path lengths of already visited vertices



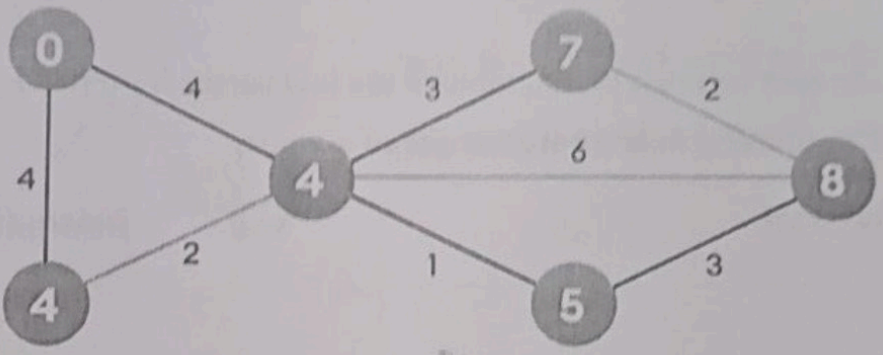
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice

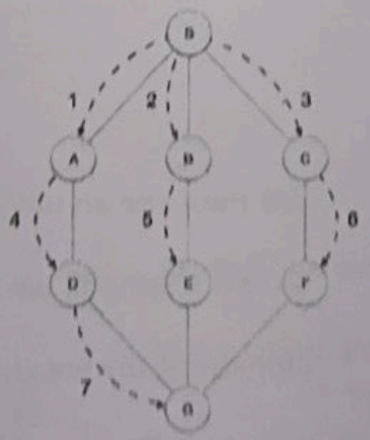


Stop: 8

Repeat until all the vertices have been visited

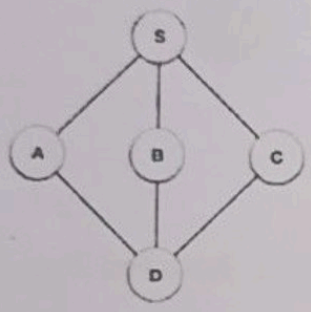
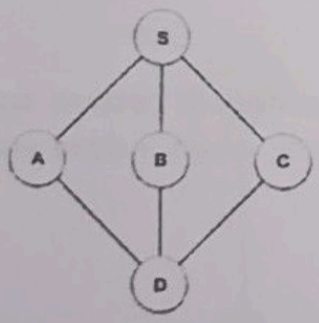
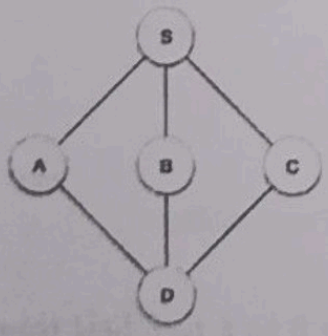
Q.29) Explain Breath first search (BFS) traversal with proper example.

- ✓ Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

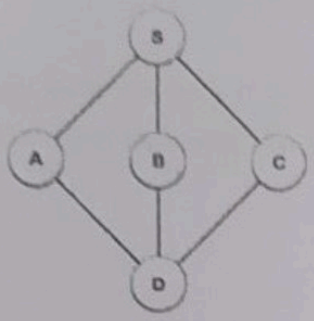


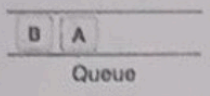
- ✓ As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D.
- ✓ It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1	 Queue	Initialize the queue.
2	 Queue	We start from visiting **S** (starting node), and mark it as visited.
3	 A Queue	We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

4

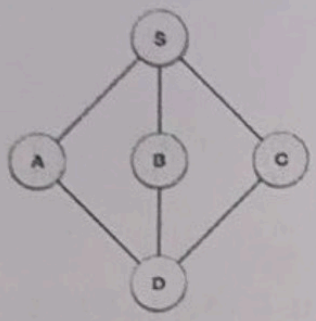


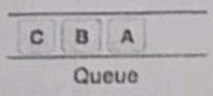


Next, the unvisited adjacent node from **S** is **B**.

We mark it as visited and enqueue it.

5

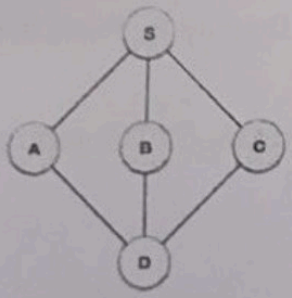


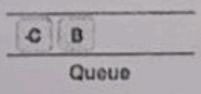


Next, the unvisited adjacent node from **S** is **C**.

We mark it as visited and enqueue it.

6

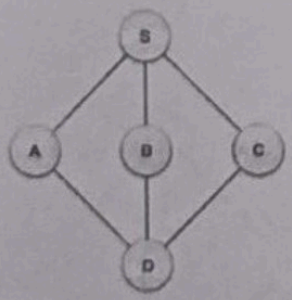


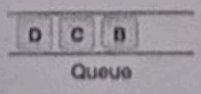


Now, **S** is left with no unvisited adjacent nodes.

So, we dequeue and find **A**.

7





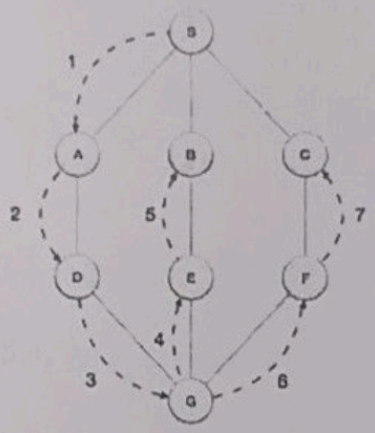
From **A** we have **D** as unvisited adjacent node.

We mark it as visited and enqueue it.

- ✓ At this stage, we are left with no unmarked (unvisited) nodes.
- ✓ But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

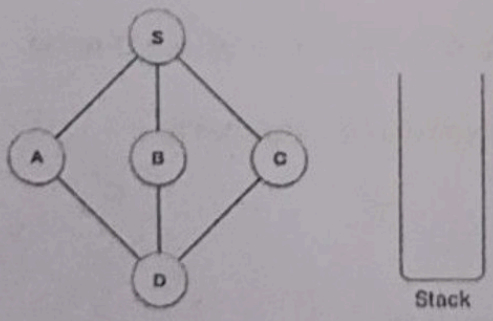
Q.30) Explain Depth first search (DFS) traversal with proper example.

✓ Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

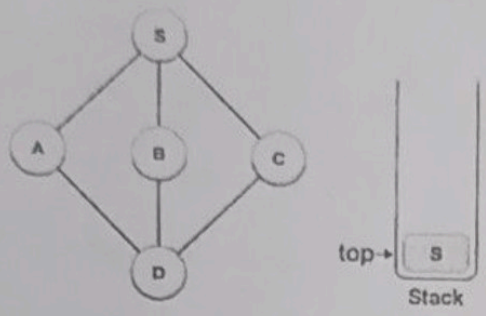


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.

2



Mark **S** as visited and put it onto the stack.

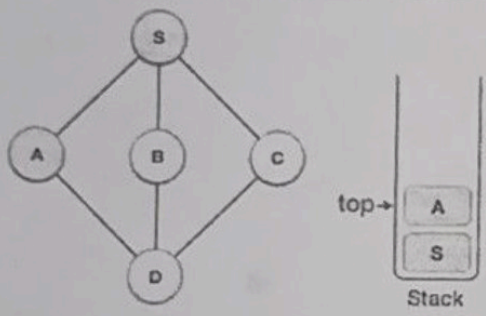
Explore any unvisited adjacent node from **S**.

We have three nodes and we can pick any of them.

For this example, we shall take the node in an

Alphabetical order.

3



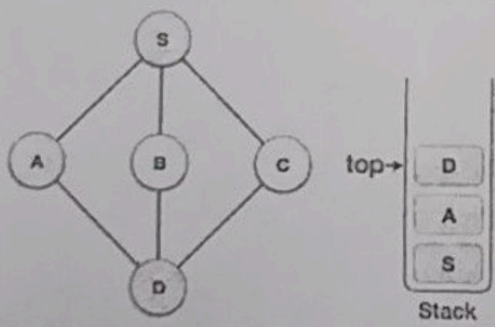
Mark **A** as visited and put it onto the stack.

Explore any unvisited adjacent node from A.

Both **S** and **D** are adjacent to **A** but we are concerned

for unvisited nodes only.

4



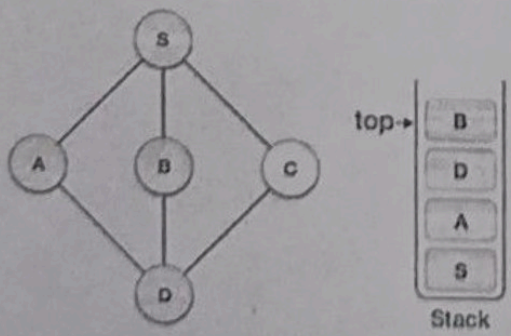
Visit **D** and mark it as visited and put onto the stack.

Here, we have **B** and **C** nodes, which are adjacent to **D**

and both are unvisited. However, we shall again

Choose in an alphabetical order.

5

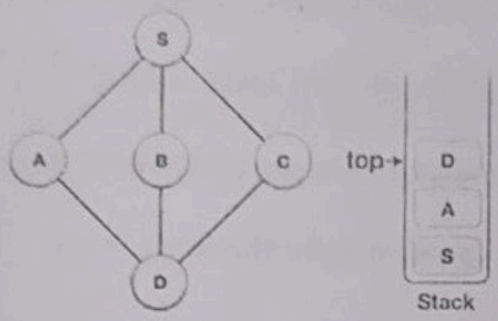


We choose **B**, mark it as visited and put onto the

stack. Here **B** does not have any unvisited adjacent

node. So, we pop **B** from the stack.

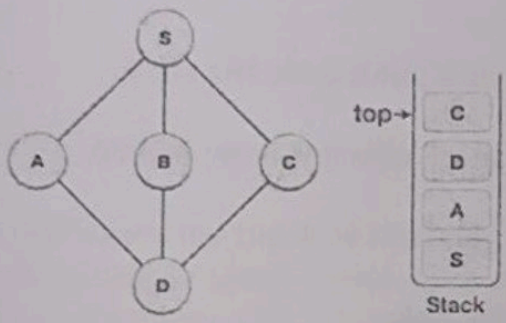
6



We check the stack top for return to the previous node and check if it has any unvisited nodes.

Here, we find **D** to be on the top of the stack.

7



Only unvisited adjacent node is from **D** is **C** now.

So we visit **C**, mark it as visited and put it onto the stack.

- ✓ As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node.
- ✓ In this case, there's none and we keep popping until the stack is empty.