(No specific Questions given for Unit 3)

Basic Definiton:

1.Circular Queue

- **Definition: A circular queue connects the last position back to the first, forming a circle.**
- **Advantage: Solves the problem of wasted space in linear queues.**

Operations:

- **Enqueue (Insert)**
- **Dequeue (Delete)**
- **Traverse**

2. Double Ended Queue (Deque)

- **Definition: A linear structure where insertions and deletions can be performed at both ends.**

Types:

1. **Input-restricted deque: Insertion only at one end, deletion at both ends.**
2. **Output-restricted deque: Deletion only at one end, insertion at both ends.**

Operations:

- **InsertFront(x)**
- **InsertRear(x)**
- **DeleteFront()**
- **DeleteRear()**

Q3. Priority Queue

- **Definition: A queue where each element has a priority, and higher priority elements are dequeued before lower priority ones.**

- **Applications: Job scheduling, CPU scheduling, Dijkstra's algorithm.**

Types:

1. **Ascending (lowest priority dequeued first)**
2. **Descending (highest priority dequeued first)**

(Actual Answers starting here):

5. Algorithm for traversing a singly linked list (with diagram)

A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.

Algorithm:

Step 1: [INITIALIZE] SET PTR = START

Step 2: Repeat Steps 3 and 4 while PTR != NULL

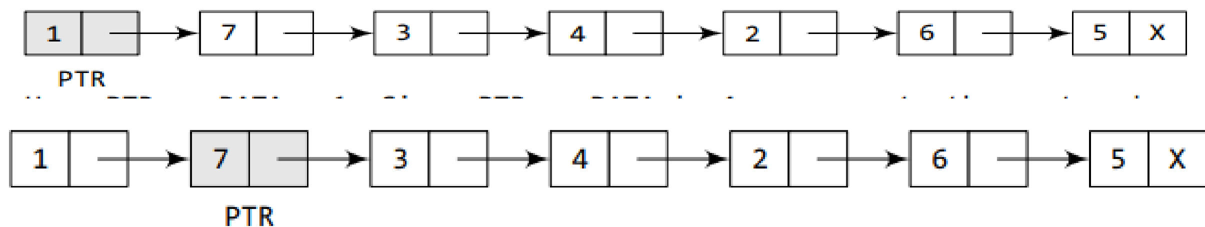
Step 3: Apply Process to PTR DATA

Step 4: SET PTR = PTR NEXT

[END OF LOOP]

Step 5: EXIT

Diagram:



… (Keep Incrementing pointer until the end of the linked list is reached)

6. Algorithm for inserting a new node in a singly linked list (with diagram)(beginning ,end ,before a node ,after a node)

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

Figure 6.13 Algorithm to insert a new node at the beginning

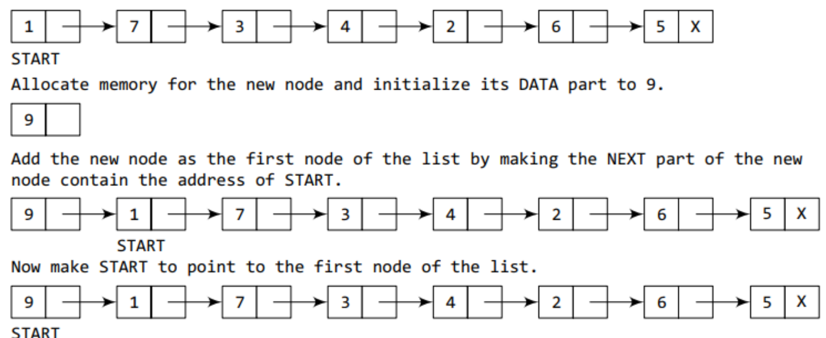
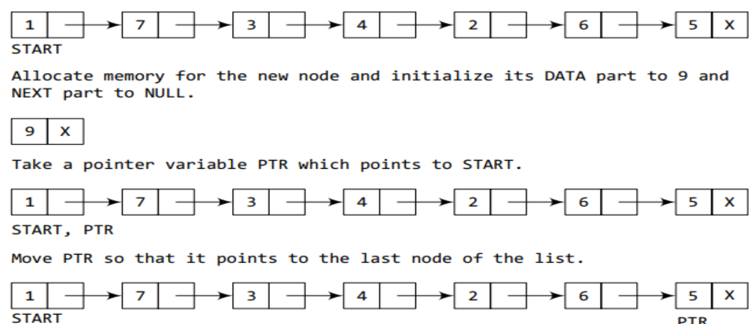


Figure 6.12 Inserting an element at the beginning of a linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
```



```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
         != NUM
Step 8:      SET PREPTR = PTR
Step 9:      SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Figure 6.16 Algorithm to insert a new node after a node that has value NUM

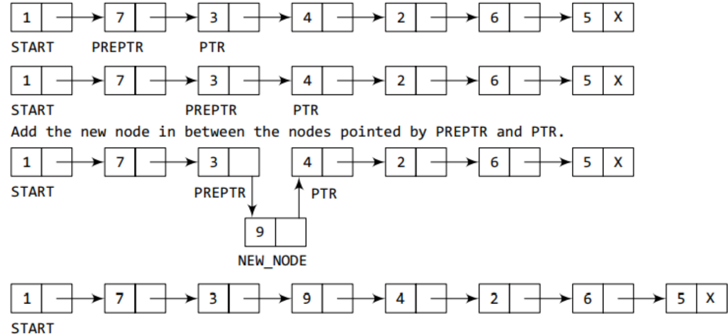


Figure 6.17 Inserting an element after a given node in a linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR -> DATA != NUM
Step 8:      SET PREPTR = PTR
Step 9:      SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Figure 6.18 Algorithm to insert a new node before a node that has value NUM

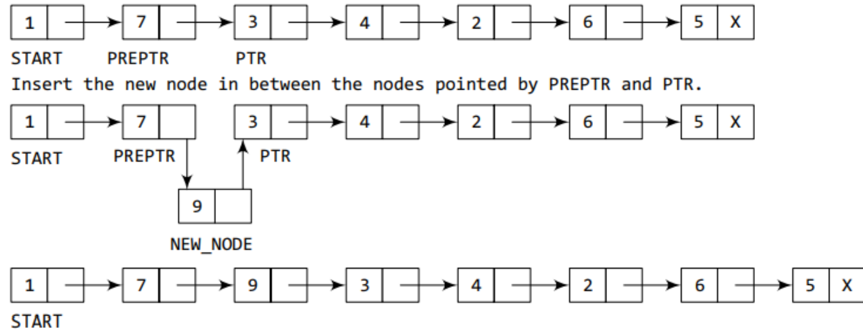


Figure 6.19 Inserting an element before a given node in a linked list

7. Algorithm for deleting a node in a singly linked list (with diagram)(beginning ,end)

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Figure 6.21 Algorithm to delete the first node

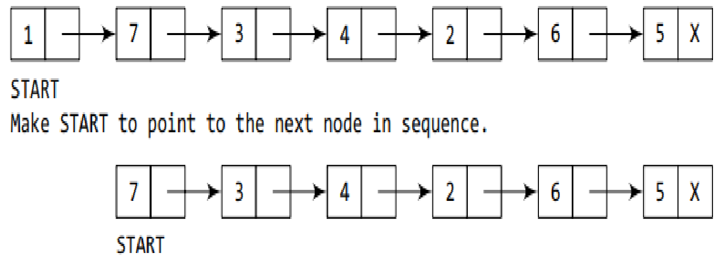


Figure 6.20 Deleting the first node of a linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Figure 6.23 Algorithm to delete the last node

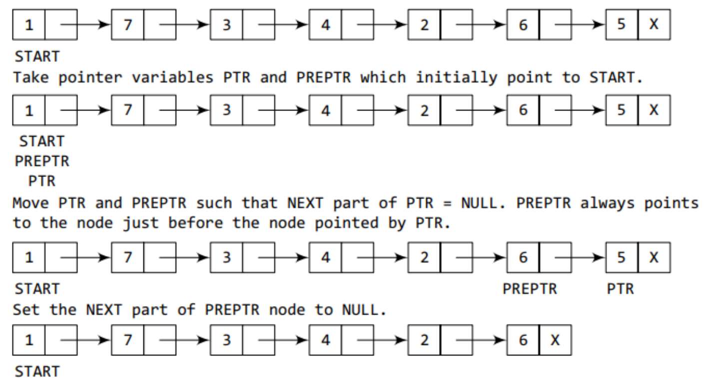


Figure 6.22 Deleting the last node of a linked list

8. Header linked list and its classifications with memory representation.

A header linked list is a special type of linked list which contains a header node at the beginning of the list. So, in a header linked list, START will not point to the first node of the list but START will contain the address of the header node. As in other linked lists, if START = NULL, then this denotes an empty header linked list. The following are the two variants of a header linked list:

- Grounded header linked list which stores NULL in the next field of the last node.
- Circular header linked list which stores the address of the header node in the next field of the last node. Here, the header node will denote the end of the list.

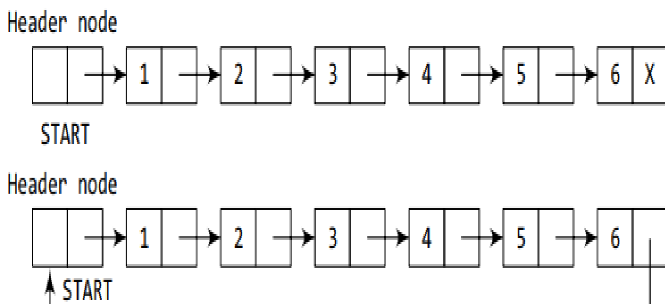


Figure 6.65 Header linked list

START: 5 → 5

	DATA	NEXT
1	H	3
2		
3	E	6
4		
5	1234	1
6	L	7
7	L	9
8		
9	0	-1

Figure 6.66 Memory representation of a header linked list

9. Doubly Linked List and its memory Representation.

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node.

In C, the structure of a doubly linked list can be given as,

```
struct node {
struct node *prev;
int data;
struct node *next;
};
```

The PREV field of the first node and the NEXT field of the last node will contain NULL. The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

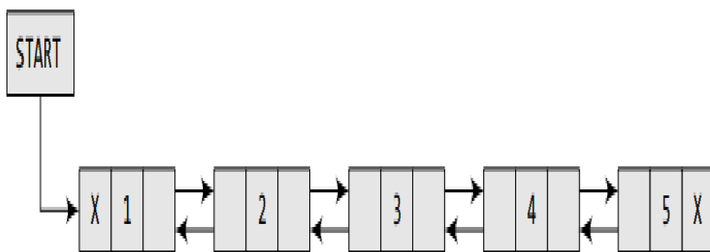


Figure 6.37 Doubly linked list

START: 1

	DATA	PREV	NEXT
1	H	–1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	–1

Figure 6.38 Memory representation of a doubly linked list

11. Properties of tree and terms related to tree

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root. The following are the properties of a tree:
Root node:
The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.
Sub-trees:
If the root node R is not NULL, then the trees T1 , T2 , and T3 are called the sub-trees of R.
Leaf node:
A node that has no children is called the leaf node or the terminal node. Path A sequence of consecutive edges is called a path.

Ancestor node:
An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors. In the given tree, nodes A, C, and G are the ancestors of node K.
Descendant node:
A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the given figure, nodes C, G, J, and K are the descendants of node A.
Level number:
Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.
Degree:
Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.
In-degree:
In-degree of a node is the number of edges arriving at that node.
Out-degree:
Out-degree of a node is the number of edges leaving that node.

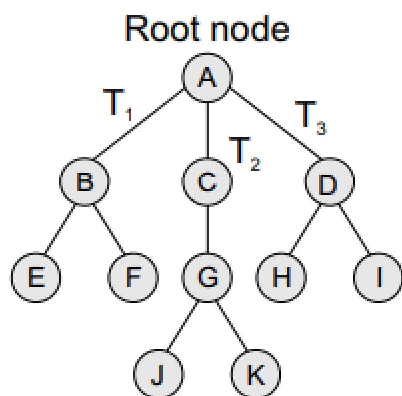


Figure 9.1 Tree

Trees are of following 6 types:

1. General trees
2. Forests
3. Binary trees
4. Binary search trees
5. Expression trees
6. Tournament trees

12. Properties of Binary tree

A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children. A node that has zero children is called a leaf node or a terminal node. Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty. A complete binary tree is a binary tree that satisfies two properties. First, in a complete binary tree, every level, except possibly the last, is completely filled. Second, all nodes appear as far left as possible.

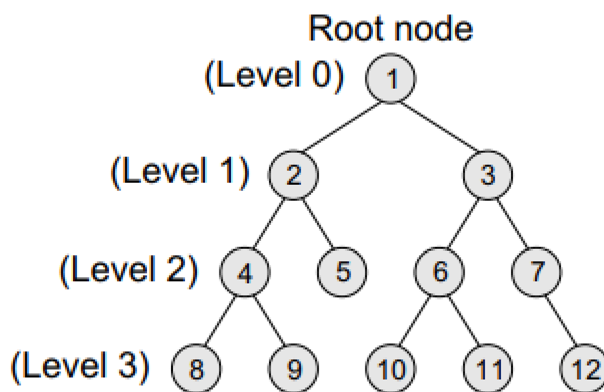


Figure 9.4 Levels in binary tree

(If asked for 8 marks this can be added)

Terminology

Parent If N is any node in T that has *left successor* S_1 and *right successor* S_2, then N is called the *parent* of S_1 and S_2. Correspondingly, S_1 and S_2 are called the left child and the right child of N. Every node other than the root node has a parent.

Level number Every node in the binary tree is assigned a *level number* (refer Fig. 9.4). The root node is defined to be at level 0. The left and the right child of the root node have a level number 1. Similarly, every node is at one level higher than its parents. So all child nodes are defined to have level number as `parent's level number + 1`.

Degree of a node It is equal to the number of children that a node has. The degree of a leaf node is zero. For example, in the tree, degree of node 4 is 2, degree of node 5 is zero and degree of node 7 is 1.

Sibling All nodes that are at the same level and share the same parent are called *siblings* (brothers). For example, nodes 2 and 3; nodes 4 and 5; nodes 6 and 7; nodes 8 and 9; and nodes 10 and 11 are siblings.

Leaf node A node that has no children is called a leaf node or a terminal node. The leaf nodes in the tree are: 8, 9, 5, 10, 11, and 12.

Similar binary trees Two binary trees T and T′ are said to be similar if both these trees have the same structure. Figure 9.5 shows two *similar binary trees*.

Copies Two binary trees T and T′ are said to be *copies* if they have similar structure and if they have same content at the corresponding nodes. Figure 9.6 shows that T′ is a copy of T.

Edge It is the line connecting a node N to any of its successors. A binary tree of n nodes has exactly `n - 1` edges because every node except the root node is connected to its parent via an edge.

Path A sequence of consecutive edges. For example, in Fig. 9.4, the path from the root node to the node 8 is given as: 1, 2, 4, and 8.

Depth The *depth* of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.

Height of a tree It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1.

13. Algorithm for tree traversal (post order pre order in order)

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           Write TREE -> DATA
Step 3:           PREORDER(TREE -> LEFT)
Step 4:           PREORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 9.16 Algorithm for pre-order traversal

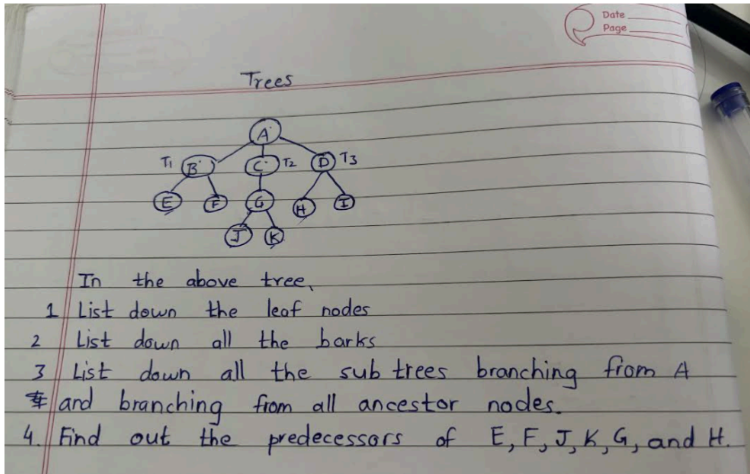
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           INORDER(TREE -> LEFT)
Step 3:           Write TREE -> DATA
Step 4:           INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

Figure 9.17 Algorithm for in-order traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:          POSTORDER(TREE -> LEFT)
Step 3:          POSTORDER(TREE -> RIGHT)
Step 4:          Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```

Figure 9.18 Algorithm for post-order traversal

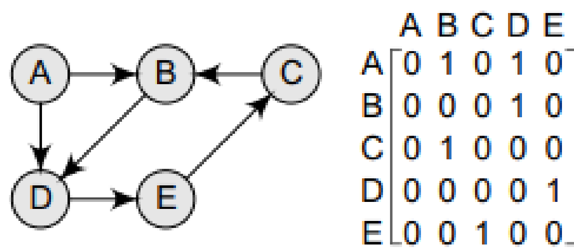
14.Numerical problem on tree traversal (Sample Question)



15. Sequential representation of graph

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

For any graph G having n nodes, the adjacency matrix will have the dimension of n X n. In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry a i,j in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, a i,j will be set to zero.



(a) Directed graph

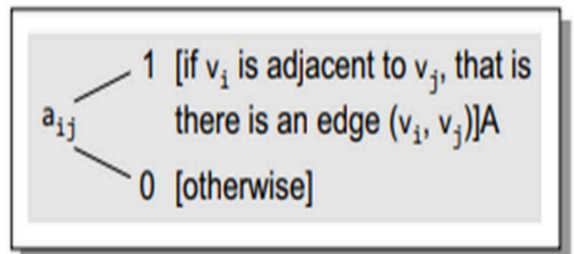


Figure 13.13 Adjacency matrix entry

16. Linked list representation of graph

An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

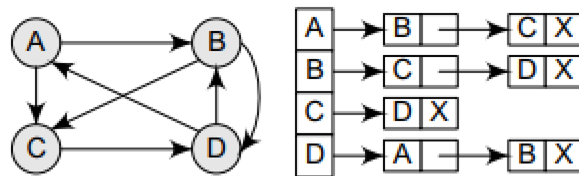
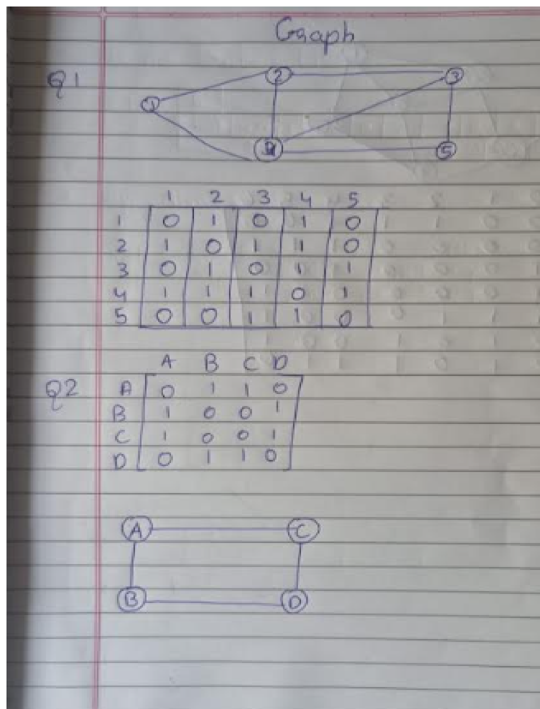


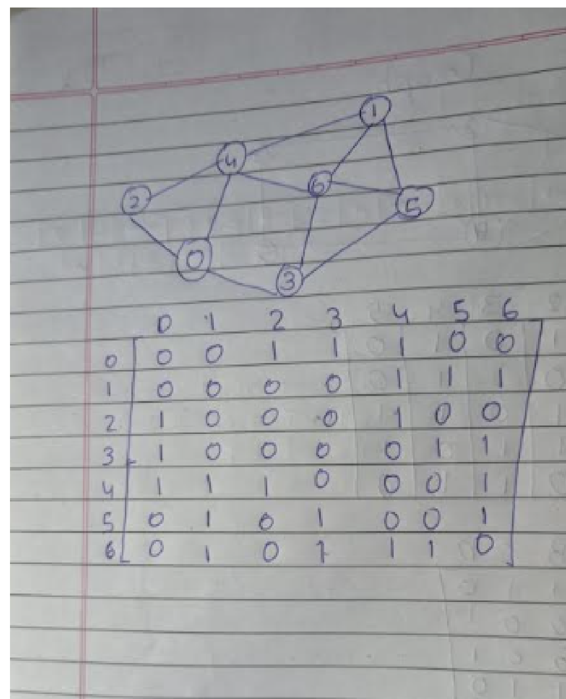
Figure 13.17 Graph G and its adjacency list

Advantages (If asked for more marks)

It is easy to follow and clearly shows the adjacent nodes of a particular node. It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice. Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

17. Numerical problem on Sequential representation of graph





18. Numerical problem on Linked list representation of graph (To review)**

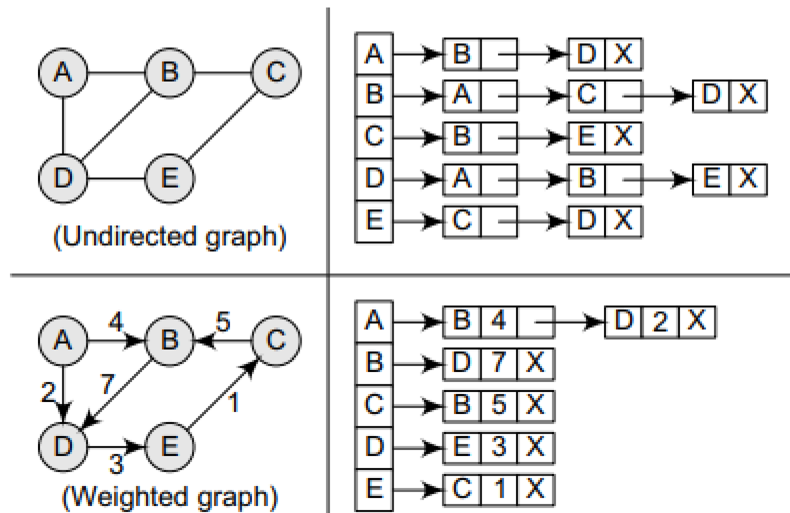


Figure 13.18 Adjacency list for an undirected graph and a weighted graph

19.BFS of a graph

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal. That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

```
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
```

Figure 13.20 Algorithm for breadth-first search

20 DFS of a graph

The depth-first search algorithm progresses by expanding the starting node 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. In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

```
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
```

Figure 13.22 Algorithm for depth-first search

Q21-Q26 Not for PT2