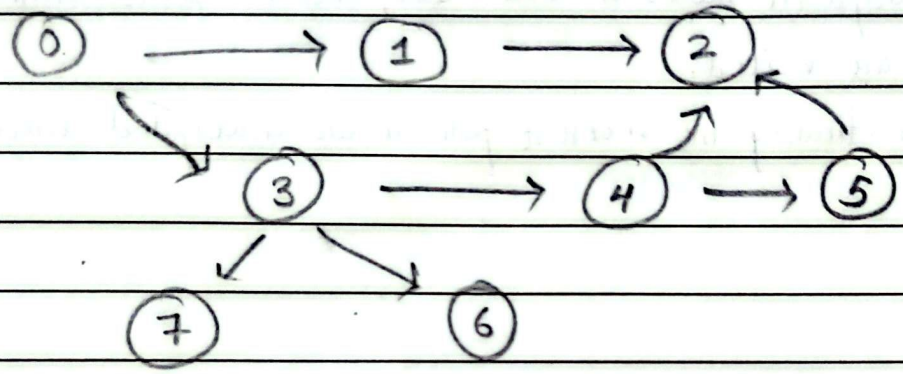
Graphs

Date:

An ADJACENCY MATRIX is a square grid (matrix) used to represent a graph, where rows and columns correspond to the graph's vertices, and the entry at (i, j) shows if there is an edge between vertex i and vertex j (usually 1 == edge || 0 == NO edge).

⇒ Considering the following graph :-



Populating the ADJACENCY MATRIX :-

	0	1	2	3	4	5	6	7
0	0	1	0	1	0	0	0	0
1	0	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	1	1
4	0	0	1	0	0	1	0	0
5	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

- The number of vertices (# of nodes) equals the size of the square matrix.

⇒ The value at entry (i, j) is 1 when there is a connection of the node at the i^{th} row, TO a node at the j^{th} column.

eg. (1) → (2)

DRAWBACK : Space Complexity : $O(n^2)$ } grows exponentially large in real-world scenarios, making it impractical.

Graph Traversal Techniques :-

i- Depth-first search (DFS) : implemented using **stacks** (iterative approach)

ii- Breadth-first search (BFS) : implemented using **queues** (iterative approach)

BREADTH-FIRST SEARCH

It is a graph traversal algorithm that starts from a source node and explores the graph level by level. First, it visits all nodes directly adjacent to the source. Then, it moves on to visit the adjacent nodes of those nodes, and this process continues until all reachable nodes are visited.

- standard algorithm for finding the shortest path in an unweighted directed graph.

Pseudocode :-

```
procedure BFS (Vertex s) :
    initialize VisitedSet VS
    initialize Queue Q
    initialize List L
    Q.enqueue (s)
    VS.add (s)
    while Q.size > 0
        v ← Q.dequeue()
        L.add (v)
        for all w adjacent to v :
            if w NOT in VS :
                Q.enqueue(w)
                VS.add (w)
            end if
        end for
    end while
end procedure
```

Date: _____

An adjacency list is a way to represent a graph data structure.
↳ For each vertex, store a list of its neighbors.

There are three (3) different implementations of an adjacency list :

i- Array of dynamic arrays (basic)

ii- Struct-based implementation (tracks size automatically)

iii- Linked list (most flexible)

The class structure of Graph is the same across the struct-based implementation and the linked list implementation of an adjacency list.

⇒ class Graph {	class Graph {
private :	private :
AdjListNode* adjList ;	EdgeNode** adjList ;
int numVertices ;	int numVertices ;
bool IsDirected ;	bool IsDirected ;
Struct-based implementation	Linked list implementation

- When adding an edge in the graph, the function signature remains the same across all implementations, and a necessary error validation is added.

```
void addEdge(int src, int dest) {
    if (src < 0 || src >= numVertices || dest < 0 || dest >= numVertices) {
        cout << "Invalid vertex!" << endl;
        return;
    }
// for a linked list implementation :
    // Add edge from src to dest (insert at beginning)
    EdgeNode* newNode = new EdgeNode(dest);
    newNode->next = adjList[src];
    adjList[src] = newNode;
```

Date:

```
    // If undirected, add reverse edge :
    if (!IsDirected) {
        newNode = new EdgeNode(src);
        newNode -> next = adjList [dest];
        adjList [dest] = newNode;
    }
}
```

Date:

DEPTH-FIRST SEARCH

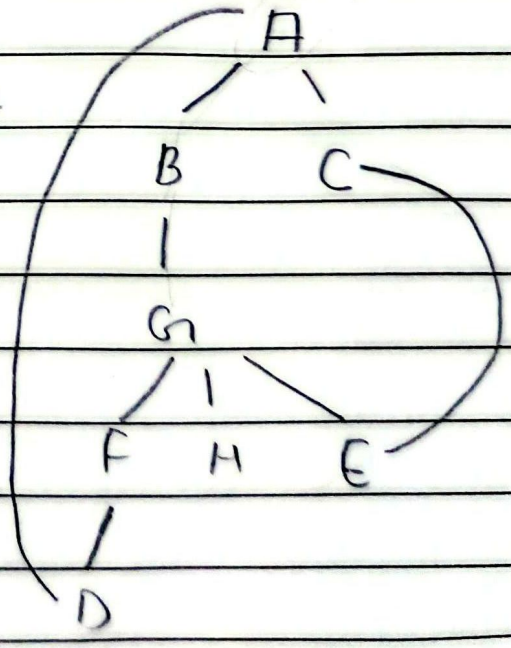
It is a graph traversal method that starts from a source vertex and explores each path completely before backtracking and exploring other paths.

Pseudocode :-

```
procedure DFS (vertex s, set VS, List L):
    initialize stack k
    k.add(s)
    VS.add(s)
    while (k != 0) : // while k is not empty
        v ← k.pop()
        L.add(v)
        for all w adjacent to v :
```

- DFS always prefers depth over backtracking.

⇒ In DFS, backtracking occurs only when a vertex has no unvisited vertices. (all vertices are visited)
If E is connected to C and C is unvisited, DFS will always move to C before returning to G.



A B G E C F D H

↓

alphabetical order traversal