Lab 11

Task 01:

Give implementation of the Adjacency List of the Graph.

```
class GraphNode:
    def __init__(self, vertex=0, next_node=None):
        self.vertex = vertex  # Vertex identifier
        self.next = next_node  # Pointer to the next node (for adjacency
list)


class Graph:
    MAX = 10  # Maximum number of vertices

    def __init__(self):
        self.headnodes = [None] * self.MAX  # Array of head nodes for
each vertex
        self.n = 0  # Number of vertices in the graph
        self.visited = [False] * self.MAX  # Visited flag for each
vertex (used in traversal)

    def initialize_visited(self):
        self.visited = [False] * self.MAX  # Reset visited status for
all vertices

    def addVertex(self, vertex):  # Add a vertex to the graph

        implement this method

    def removeVertex(self, vertex):  # Remove a vertex and its
associated edges from the graph
        implement this method

    def addEdge(self, vertex1, vertex2):  # Add an edge between two
vertices
        implement this method

    def removeEdge(self, vertex1, vertex2):  # Remove an edge between
two vertices
        implement this method

    def vertexExists(self, vertex):  # Check if a vertex exists in the
graph
        return self.headnodes[vertex] is not None

    def printGraph(self):  # Print the graph (adjacency list
representation)
        for i in range(self.MAX):
            if self.headnodes[i] is not None:
                print(f"Vertex {i}:", end="")
                curr = self.headnodes[i].next
                while curr is not None:
                    print(f" {curr.vertex}", end=", ")
                    curr = curr.next
                print()

    def dfs(self, vertex):  # Perform DFS starting from a given vertex
        implement this method
    def bfs(self, vertex):  # Perform BFS starting from a given vertex
        implement this method
```

1. **def addVertex(self, vertex)**

This method is used to insert a vertex node in the graph.

2. **def removeVertex(self, vertex): # Remove a vertex and its associated edges from the graph**

This method is used to remove a vertex from graph and its associated edges.

3. **def addEdge(self, vertex1, vertex2): # Add an edge between two vertices**

This method is used to insert a edge from vertex1 to vertex2 in the graph.

4. **def removeEdge(self, vertex1, vertex2): # Remove an edge between two vertices**

This method is used to remove a edge from vertex1 to vertex2 in the graph.

5. **def dfs(self, vertex): # Perform DFS starting from a given vertex**

This method is used to DFS traversal of the graph. And print the order.

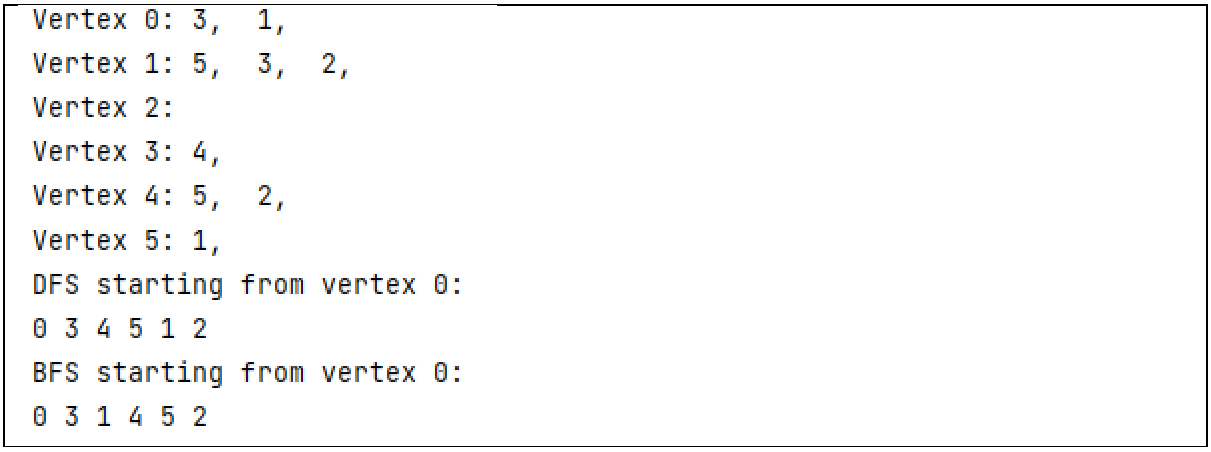
6. **def bfs(self, vertex): # Perform BFS starting from a given vertex**

This method is used to BFS traversal of the graph. And print the order.

Driver program:

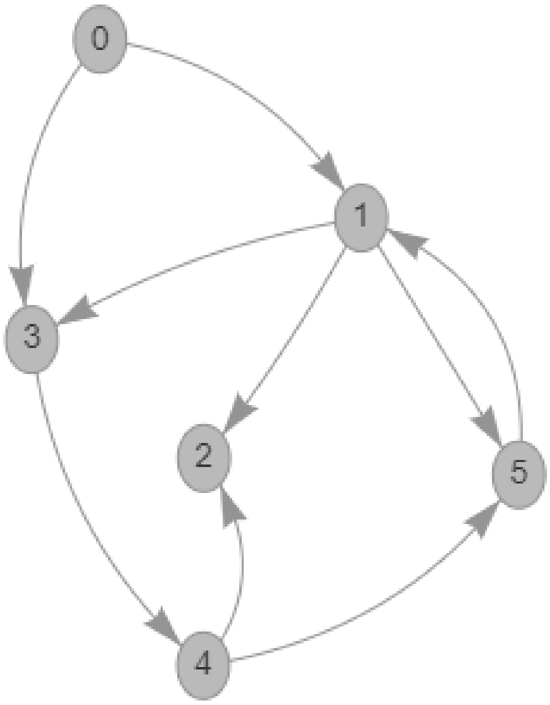
```
# Create a graph instance
g = Graph()

# Add vertices
for i in range(6):
    g.addVertex(i)

# Add edges
g.addEdge(0, 1)
g.addEdge(0, 3)
g.addEdge(1, 2)
g.addEdge(1, 3)
g.addEdge(1, 5)
g.addEdge(3, 4)
g.addEdge(4, 2)
g.addEdge(4, 5)
g.addEdge(5, 1)
# Print the graph
g.printGraph()

# Perform DFS and BFS traversals
print("DFS starting from vertex 0:")
g.dfs(0)

g.initialize_visited()

print("BFS starting from vertex 0:")
g.bfs(0)
```

The output of the following program is:



```
Vertex 0: 3,  1,
Vertex 1: 5,  3,  2,
Vertex 2:
Vertex 3: 4,
Vertex 4: 5,  2,
Vertex 5: 1,
DFS starting from vertex 0:
0 3 4 5 1 2
BFS starting from vertex 0:
0 3 1 4 5 2
```

Here is the graph which is used in the driver code

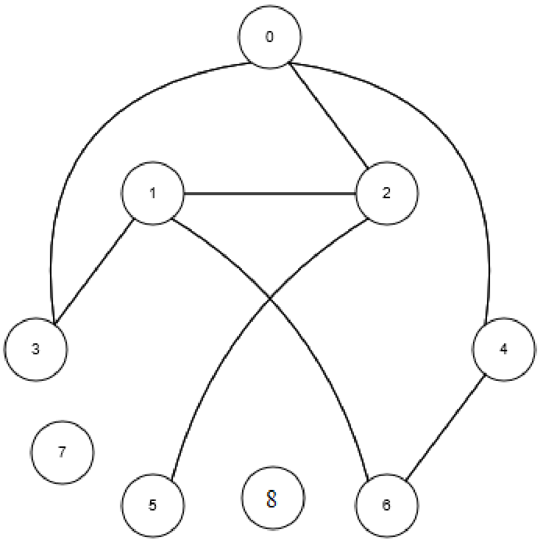


Task 02:

1. Create adjacency list for **undirected**, unweighted graph. The representation of the undirected graph has number of vertices. Number of edges, followed by bi-directional edges pair (the pair in edges are both source and destination)

Input:

```
9
8
2 0
0 3
5 2
4 6
4 0
1 3
2 1
6 1
```



Output:

```
0 -> 2 3 4
1 -> 3 2 6
2 -> 0 5 1
3 -> 0 1
4 -> 6 0
5 -> 2
6 -> 4 1
7 -> x
8  -> x
```

2. Create adjacency list for **directed**, unweighted graph. The representation of the directed graph has number of vertices. Number of edges, followed by edges pair (edge is from **source to destination**)

Input:

```
8
11
0 3
5 7
7 5
4 6
4 2
2 5
1 5
1 6
5 6
6 5
4 7
```

Output:

```
0 -> 3
1 -> 5 6
2 -> 5
3 -> x
4 -> 2 6 7
5 -> 6 7
6 -> 5
7 -> 5
```