# ***Graphs***
## ***DFS, BFS, Topological Sorting***

**DFS :**

**Iterative Approach :**

```
// visited is a global array
void DFS_iterartive(struct graph* G, int visited[], int start) {
    int stack[G->V]; // stack of size G->V (G->V represents the no of
vertices in G)
    int top = -1;
    int i;
    visited[start] = 1;
    stack[++top] = start;
    struct ListNode *p = NULL;
    while(top != -1) {
        start = stack[top--];// but better to use another new
variable
        cout << start;
        while(p) {
            i = p->vertex;
            if(visited[i] == 0) {
                stack[++top] = i;
                visited[i] = 1;
            }
            p = p->next;
        }
    }
}
```

**Recursive Approach :**

```
void DFS_recursive(struct graph* G, int visited[], int start) {
    int i;
    struct ListNode *p = NULL;
    visited[start] = 1;
    cout << start;
    p = G->adjList[start];
    while(p) {
        i = p->vertex;
        if(visited[i] == 0) {
            DFS_recursive(G, visited, i);
        }
        p = p->next;
    }
}
```

**BFS :**

```
void BFS(sruct Graph* graph, int start) {
    strcut Queue* q = createQueue();
    enQueue(q, start);
    while(!isEmpty(q)) {
        printQueue(q);
        int current = deQueue(q);
        graph->visited[current] = 1;
        struct ListNode* temp = graph->adjLists[current];
        while(temp) {
            int adj = temp->vertex;
            if(graph->visited[adj] == 0) {
                graph->visited[adj] = 1;
                enQueue(q, adj);
            }
            temp= temp->next;
        }
    }
}
```

**Topological Sorting :**

```
void topologicalSort(struct graph* G) {
    int topsort[G->V], indeg[G->V]; // arrays with (size = no of
vertices)
    int i;
    for (i = 0; i < G->V; i++) {
        indeg[i] = findIndegree(G, i);
        if (indeg[i] == 0) {
            enqueue(i);
        }
    }
    int j = 0;
    int del_node;
    while (!isEmpty(Queue)) {
        del_node = dequeue();
        topsort[j] = del_node;
        j++;
        for (i = 0; i < G->V; i++) {
            if (G->adjMatrix[del_node][i] == 1) {
                G->adjMatrix[del_node][i] = 0;
                indeg[i]--;
                if (indeg[i] == 0) {
                    enqueue(i);
```

```
                }
            }
        }
    }
    for (i = 0; i < j; i++) {
        cout << topsort[i];
    }
}
```

## ***Shortest Path Algorithms***

**Shortest path in unweighted graph :**

```
void UnweightedShortestPath(struct Grap *G, int s) {
    struct Queue *Q = createQueue();
    int v, w;
    enQueue(Q, s);
    for(int i = 0; i < G->V; i++) {
        Distance[i] = -1;
    }
    Distance[s] = 0;
    while(!isEmpty(Q)) {
        v = deQueue(Q);
        for each w adjacent to v
            if(Distance[w] == -1) {
                Distance[w] = Distance[v] + 1;
                Path[w] = v;
                enQueue(Q, w);
            }
    }
    deleteQueue(Q);
}
```

**Dijktra's Algorithm (Shortest path in weighted graph without negative edge weights) :**

```
void Dijkstra(struct Grap *G, int s) {
    struct PriorityQueue *PQ = createPriorityQueue();
    int v, w;
    enQueue(PQ, s);
    for(int i = 0; i < G->V; i++) {
        Distance[i] = -1;
    }
    Distance[s] = 0;
    while(!isEmpty(PQ)) {
        v = deleteMin(PQ);
        for each w adjacent to v {
```

```
            Compute new distance d = Distance[v] + weight[v][w];
            if(Distance[w] == -1) {
                Distance[w] = new distance d;
                Insert w in the priority queue with priority d
                Path[w] = v;
            }
            if(Distance[w] > new distance d) {
                Distance[w] = new distance d;
                Update priority of vertex w to be d;
                Path[w] = v;
            }
        }
    }
}
```

**Bellman Ford (Weighted graphs with negative weights) , (also to check existence of negative cycles) :**

```
void BellmanFord(struct graph *G, int s) {
    strcut Queue *Q = createQueue();
    int v, w;
    enQueue(Q, s);
    Distance[s] = 0;  // assume the distance is filled with INT_MAX
    while(!isEmpty(Q)) {
        v = deQueue(Q);
        for all w adjacent to v {
            Compute new distance d = Distance[v] + weight[v][w];
            if(old distance to w > new distance d) {
                Distance[v] = ( distance to v ) + weight[v][w];
                Path[w] = v;
                if(w is there in queue) {
                    enQueue(Q, w);
                }
            }
        }
    }
}
```

## ***Minimal Spanning Tree***

**Prim's algorithm :**

```
void Prims(struct graph *G, int s) {
    struct PriorityQueue *PQ = createPriorityQueue();
    int v, w;
    enQueue(PQ, s);
    Distance[s] = 0;
    while(!isEmpty(PQ)) {
```

```
        v = deleteMin(PQ);
        for all adjacent vertices w of v {
            Compute new distance d = Distance[v] + weight[v][w];
            if(Distance[w] == -1) {
                Distance[w] = weight[v][w];
                Insert w in the priority queue wuth priority d
                Path[w] = v;
            }
            if(Distance[w] > new distance d) {
                Distance[w] = weight[v][w];
                Update priority of vertex w to be d;
                Path[w] = v;
            }
        }
    }
}
```

**Kruskal’s algorithm :**

```
void Kruskals(struct graph *G) {
    S = phi;
    for(int v = 0; v < G->V; v++) {
        MakeSet(v);
    }
    Sort edges of E by increasing weights weightsfor each edhe (u, v)
in E {
        if(FIND (u) != FIND (v)) {
            S = S U {(u, v)};
            UNION (u, v);
        }
    }
    return S;
}
```