Graphs
Graph Representations

Graph Representation by Adjacency Matrix :

Declaration :
```
struct Graph {
    int V;
    int E;
    int **adjMatrix;
}
```

Implementation for Undirected using Adj Matrix :
```
#define MAX_VERTICES 50
#define MAX_DEGREE 50

struct graph {
    int V;               // Number of vertices
    int E;               // Number of edges
    int **adjMatrix;     // Adjacency matrix
};

struct edge {
    int source;          // Source vertex
    int destination;     // Destination vertex
};

// Initialize random seed
void rand_init(void) {
    time_t t;
    srand((unsigned)time(&t));
}

// Function to create a new graph
graph* createGraph(int vertices) {
    graph* g = new graph;
    g->V = vertices;
    g->E = 0;

    // Allocate memory for the adjacency matrix
    g->adjMatrix = new int*[vertices];
    for (int i = 0; i < vertices; ++i) {
        g->adjMatrix[i] = new int[vertices]();
    }

    return g;
}
```

```
// Function to insert an edge into the graph
void insertEdge(graph* g, int u, int v) {
    if (u >= 0 && u < g->V && v >= 0 && v < g->V) {
        if (g->adjMatrix[u][v] == 0) {
            g->adjMatrix[u][v] = 1;
            g->adjMatrix[v][u] = 1; // Since it's undirected
            g->E++;
        }
    }
}

// Function to remove an edge from the graph
void removeEdge(graph* g, int u, int v) {
    if (u >= 0 && u < g->V && v >= 0 && v < g->V) {
        if (g->adjMatrix[u][v] == 1) {
            g->adjMatrix[u][v] = 0;
            g->adjMatrix[v][u] = 0; // Since it's undirected
            g->E--;
        }
    }
}

// Function to display the adjacency matrix
void displayGraph(graph* g) {
    cout << "Adjacency Matrix:" << endl;
    for (int i = 0; i < g->V; ++i) {
        for (int j = 0; j < g->V; ++j) {
            cout << g->adjMatrix[i][j] << " ";
        }
        cout << endl;
    }
}

// Function to display all edges in the graph
void displayEdges(graph* g) {
    cout << "Edges in the Graph:" << endl;
    for (int i = 0; i < g->V; ++i) {
        for (int j = i + 1; j < g->V; ++j) { // Avoid duplicate edges
            if (g->adjMatrix[i][j] == 1) {
                cout << "(" << i << ", " << j << ")" << endl;
            }
        }
    }
}
```

```
// Function to free allocated memory and destroy the graph
void destroyGraph(graph* g) {
    for (int i = 0; i < g->V; ++i) {
        delete[] g->adjMatrix[i];
    }
    delete[] g->adjMatrix;
    delete g;
}

// Function to create a new edge
edge newEdge(int u, int v) {
    edge e;
    e.source = u;
    e.destination = v;
    return e;
}

// Function to create a random graph
void randomGraph(graph* g, int numEdges) {
    rand_init();
    for (int i = 0; i < numEdges; ++i) {
        int u = rand() % g->V;
        int v = rand() % g->V;
        if (u != v) {
            insertEdge(g, u, v);
        }
    }
}
```

Graph Representation by Adjacency List:

Declaration :
```
struct Graph {
    int V;
    int E;
    int *Adj;
}
```

Implementation for Undirected using Adj Matrix :
```
#define MAX_VERTICES 50
#define MAX_DEGREE 50

struct graph {
    int V;                  // Number of vertices
    int E;                  // Number of edges
    vector<list<int>> Adj;  // Adjacency list
```

```
};

struct edge {
    int source;             // Source vertex
    int destination;        // Destination vertex
};

// Initialize random seed
void rand_init(void) {
    time_t t;
    srand((unsigned)time(&t));
}

// Function to create a new graph
graph* createGraph(int vertices) {
    graph* g = new graph;
    g->V = vertices;
    g->E = 0;
    g->Adj.resize(vertices);  // Resize the adjacency list for
vertices
    return g;
}

// Function to insert an edge into the graph
void insertEdge(graph* g, int u, int v) {
    if (u >= 0 && u < g->V && v >= 0 && v < g->V) {
        g->Adj[u].push_back(v);
        g->Adj[v].push_back(u); // Since it's undirected
        g->E++;
    }
}

// Function to remove an edge from the graph
void removeEdge(graph* g, int u, int v) {
    if (u >= 0 && u < g->V && v >= 0 && v < g->V) {
        g->Adj[u].remove(v);
        g->Adj[v].remove(u); // Since it's undirected
        g->E--;
    }
}

// Function to display the adjacency list
void displayGraph(graph* g) {
    cout << "Adjacency List:" << endl;
    for (int i = 0; i < g->V; ++i) {
```

```
        cout << i << ": ";
        for (int neighbor : g->Adj[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }
}

// Function to display all edges in the graph
void displayEdges(graph* g) {
    cout << "Edges in the Graph:" << endl;
    for (int i = 0; i < g->V; ++i) {
        for (int neighbor : g->Adj[i]) {
            if (i < neighbor) { // To avoid duplicate edges
                cout << "(" << i << ", " << neighbor << ")" << endl;
            }
        }
    }
}

// Function to free the graph's resources
void destroyGraph(graph* g) {
    delete g;
}

// Function to create a new edge
edge newEdge(int u, int v) {
    edge e;
    e.source = u;
    e.destination = v;
    return e;
}

// Function to create a random graph
void randomGraph(graph* g, int numEdges) {
    rand_init();
    for (int i = 0; i < numEdges; ++i) {
        int u = rand() % g->V;
        int v = rand() % g->V;
        if (u != v) {
            insertEdge(g, u, v);
        }
    }
}
```

Graph Representation by Adjacency List:

Declaration :

```
struct graph {
    int V;                  // Number of vertices
    int E;                  // Number of edges
    vector<set<int>> Adj;   // Adjacency set
};
```

Implementation for Undirected using Adj Matrix :

```
#define MAX_VERTICES 50
#define MA_DEGREE 50

struct graph {
    int V;                  // Number of vertices
    int E;                  // Number of edges
    vector<set<int>> Adj;   // Adjacency set
};

struct edge {
    int source;             // Source vertex
    int destination;        // Destination vertex
};

// Initialize random seed
void rand_init(void) {
    time_t t;
    srand((unsigned)time(&t));
}

// Function to create a new graph
graph* createGraph(int vertices) {
    graph* g = new graph;
    g->V = vertices;
    g->E = 0;
    g->Adj.resize(vertices); // Resize the adjacency set for vertices
    return g;
}

// Function to insert an edge into the graph
void insertEdge(graph* g, int u, int v) {
    if (u >= 0 && u < g->V && v >= 0 && v < g->V && u != v) {
        // Insert edge into the adjacency sets
        if (g->Adj[u].find(v) == g->Adj[u].end()) {
            g->Adj[u].insert(v);
```

```
            g->Adj[v].insert(u); // Since it's undirected
            g->E++;
        }
    }
}

// Function to remove an edge from the graph
void removeEdge(graph* g, int u, int v) {
    if (u >= 0 && u < g->V && v >= 0 && v < g->V) {
        if (g->Adj[u].find(v) != g->Adj[u].end()) {
            g->Adj[u].erase(v);
            g->Adj[v].erase(u); // Since it's undirected
            g->E--;
        }
    }
}

// Function to display the adjacency set
void displayGraph(graph* g) {
    cout << "Adjacency Set:" << endl;
    for (int i = 0; i < g->V; ++i) {
        cout << i << ": ";
        for (int neighbor : g->Adj[i]) {
            cout << neighbor << " ";
        }
        cout << endl;
    }
}

// Function to display all edges in the graph
void displayEdges(graph* g) {
    cout << "Edges in the Graph:" << endl;
    for (int i = 0; i < g->V; ++i) {
        for (int neighbor : g->Adj[i]) {
            if (i < neighbor) { // To avoid duplicate edges
                cout << "(" << i << ", " << neighbor << ")" << endl;
            }
        }
    }
}

// Function to free the graph's resources
void destroyGraph(graph* g) {
    delete g; // Adjacency sets are managed by STL; no explicit
cleanup needed
```

```
}

// Function to create a new edge
edge newEdge(int u, int v) {
    edge e;
    e.source = u;
    e.destination = v;
    return e;
}

// Function to create a random graph
void randomGraph(graph* g, int numEdges) {
    rand_init();
    for (int i = 0; i < numEdges; ++i) {
        int u = rand() % g->V;
        int v = rand() % g->V;
        if (u != v) {
            insertEdge(g, u, v);
        }
    }
}
```