Graphs: Many problems can be represented using graphs. In fact, all of the data structures we have discussed (lists, stacks, queues, trees, etc.) can be represented as graphs.

Basic Definitions:

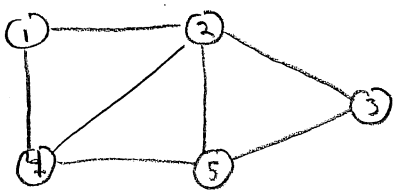
$G = (V, E)$, where V is a set of vertices (or nodes)
and E is a set of edges between vertices

An edge (u, v) specifies a connection between vertex u and vertex v.

A path is a sequence of vertices traversed by following edges between vertices. A path from a vertex u to a vertex v is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$ in which $u = v_0$ and $v = v_k$ and for all $i = 1$ to k, $(v_{i-1}, v_i) \in E$.

Undirected graph contains edges that specify connections between vertices without any predecessor (source) / successor (sink) relationship. An edge (u, v) is equivalent to (v, u) in and undirected graph.

Example:



$G = (V, E)$

$V = \{1, 2, 3, 4, 5\}$

$E = \{(1,2), (1,4), (2,3), (2,4), (2,5), (3,5), (4,5)\}$

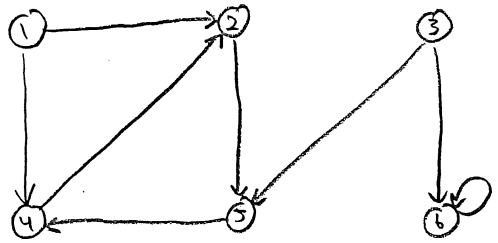
It could also represent this graph as the following

$G = (V, E)$

$V = \{(2,1), (1,4), (3,2), (4,2), (2,5), (5,3), (5,4)\}$

Directed Graph: consists of edge that specify a directed connection from a source vertex to a sink vertex.

Example:



$G = (V, E)$

$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{(1,2), (1,4), (2,5), (3,5), (3,6), (4,2), (5,4), (6,6)\}$

Edge Weights: Graphs can also be weighted in which a weight can be associated with each edge.

Graph Representation: Adjacency List Representation ⇒ Graph can be represented as a list of vertices where each vertex contains a list of vertices to which the vertex connects.

Vertex:

```
struct Vertex_ {
    int id;
    int distance;
    int color
    // we may need other data elements
    EdgeList edges;
    struct Vertex_ *next;
    struct Vertex_ *prev;
};
```

- id → id of node can be int, char *, etc.
- distance → will be used in traversal and shortest path algorithms
- color → many algorithms use colors to track status of nodes
- edges → List of edges (see below)
- next, prev → will be used in list of vertices (VertexList)

Edge:

```
struct Edge_ {
    Vertex *adjVertex;
    int weight;
    struct Edge_ *next;
    struct Edge_ *prev;
};
```

- adjVertex → pointer to adjacent Vertex
- weight → weight of edge
- next, prev → will be used in list of edges (EdgeList)

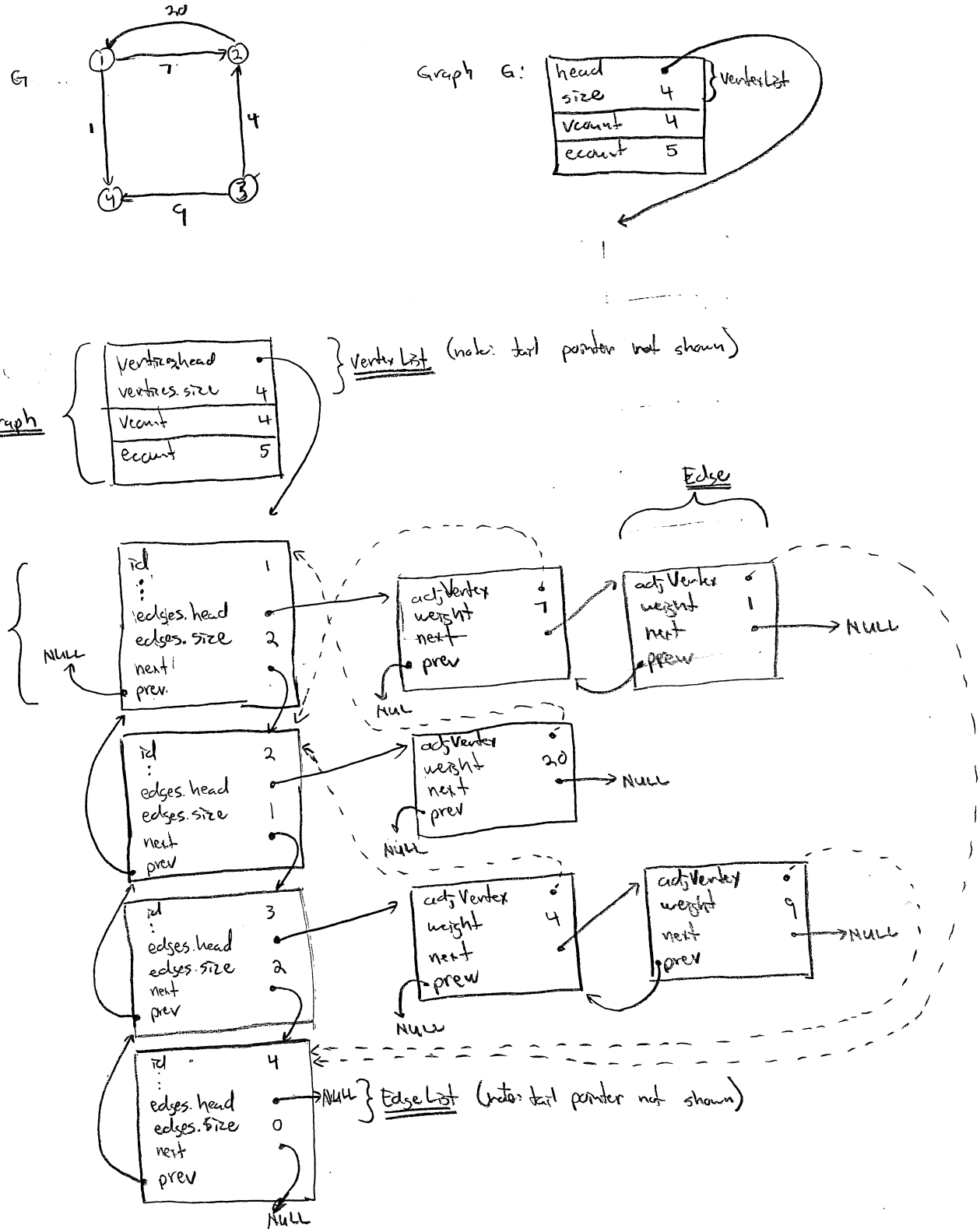
Graph:

```
typedef struct Graph_ {
    VertexList vertices;
    int vcount;
    int ecount;
} Graph;
```

* Must include the following typedefs in shared header file (e.g. global.h) in order to avoid circular dependencies.

```
typedef struct Vertex_ Vertex;
typedef struct Edge_ Edge;
```

Example: Graph representation of directed weighted graph.



Basic Graph Traversal Algorithms:

Breadth-First Search: Given a graph $G = (V, E)$ and a start vertex s, compute distance to all vertices $v \in V - \{s\}$ from s

* assuming unweighted graph

```
BFS(Graph *g, Vertex *s):
① for each vertex u ∈ g->vertices, u != s
    a) u->color = white          ——> white indicates a vertex has not been found or visited
    b) u->distance = ∞
② s->color = gray                ——> gray indicates a vertex has been found but not visited
③ s->distance = 0
④ initialize Queue Q             ——> Q is a stack holding all vertices found so far but not visited.
⑤ push(Q, s)                         Note: we need a stack pointing to the vertices, so we can use a special
⑥ while Q->size != 0                 stack of reuse the EdgeList
    a) u = pop(Q)
    b) for each edge e ∈ u->edges
        I) v = e->adjVertex
        II) if v->color == white
            i) v->color = gray
            ii) v->distance = u->distance + 1
            iii) push(Q, v)
    c) u->color = black          ——> black indicates a node has been visited
```

Depth-First Search: Given graph $G = (V, E)$, compute discovery and finishing time for each vertex $v \in G$.

* Does not require start vertex, but could be modified to do so
* Assuming unweighted graph

```
DFS(Graph * g):
① for each vertex u ∈ g->vertices
      u->color = white
② time = 0
③ for each vertex u ∈ g->vertices
   a) if u->color == white
      I> DFS-Visit(g, u)
```

```
DFS-Visit(Graph * g, Vertex * u):
① u->color = gray
② time++
③ u->distance = time ——————> discovery time
④ for each edge e ∈ u->edges
   a) v = e->adjVertex
   b) if v->color == white
      I> DFS-Visit(g, v)
⑤ u->color = black
⑥ time++
⑦ u->finish = time ——————> finish time
```

Note: Can be used to create a forest of depth-first trees. by maintaining predecessor point in each vertex

* Depth-first starting from a single vertex s can be implemented as:

```
① for each vertex u ∈ g->vertices
   a) u->color = white
   b) u->distance = ∞
② time = 0
③ DFS-Visit(g, s)
```

Shortest Path Algorithms:

- Many variants of shortest path algorithms exist (single destination, single-source, all pairs)
  - Consider only single-source algorithms for now

Dijkstra's Single-Source Shortest Path: Given a graph $G = (V, E)$ and a start vertex $s \in V$, compute the shortest path from s to each vertex $v \in V$

* Assumes G is a weighted graph, such that for each edge (u, v), a weight $w(u, v)$ is specified. All edge weights must be non-negative

```
DijkstraShortestPath(Graph *g, Vertex *s):
① for each vertex v ∈ g->vertices
    a) v->distance = ∞
    b) v->pred = NULL
② s->distance = 0
③ initialize list Q
④ for all vertices v ∈ g->vertices
    a) list_insert(Q, v)
⑤ while Q->size != 0
    a) u = extract_min(Q)
    b) for each edge e ∈ u->edges
         v = e->adjVertex
         Relax(u, e)
```

- pred is pointer to previous vertex in shortest path
- Q is a list of vertices found so far but not visited sorted by vertices' distance. (often referred to as a priority queue or heap)
- returns pointer to vertex in Q with smallest distance
  - simple implementation can use unsorted list that will be searched each time to find vertex with minimum distance

```
Relax(Vertex *u, Edge *e)
① v = e->adjVertex
② if v->distance > u->distance + e->weight
    a) v->distance = u->distance + e->weight
    b) v->pred = u
```

Runtime Complexity: $O(V^2)$ using list to implement the Q

* Can be improved to $O((V+E)\lg V)$ and even $O(V \lg V + E)$ using different data structures for the minimum priority queue.

Bellman-Ford Shortest Path: Single-source shortest path that works on graphs with negative edge weights. Can also detect if a negative-weight cycle exists

Given a graph $G = (V, E)$ and a start vertex $s \in V$, compute the shortest path from s to each vertex $v \in V$. Returns TRUE if no cycles exist (i.e. graph is acyclic), FALSE otherwise.

```
bool
BellmanFordShortestPath (Graph *g, Vertex v s):
① for each vertex v ∈ g→vertices
    a) v→ distance = ∞
    b) v→ pred = NULL
② s→ distance = 0
③ for i = 0 to g→vertices→size - 1          ]——→ repeats process |V| times
    a) for each vertex u ∈ g→vertices       ]——→ Equivalent to: for all edges (u,v) ∈ G.E
        I) for each edge e ∈ u→edges
            ii) v = e→adjVertex
            iii) Relax(u, e)
④ for each vertex u ∈ g→vertices
    a) for each edge e ∈ u→edges
        I) v = e→adjVertex
        II) if v→distance > u→distance + e→weight, return FALSE ]— condition that indicates a negative weight cycle exists.
⑤ return TRUE
```

Runtime Complexity: $O(VE)$