# 05 Graph (1)

College of Computer Science, CQU

# Outline

- Basic Concept
- Graph ADT
- Graph Representation
- Adjacency Matrix
- Adjacency List

# Basic Concept

- Graphs are a formalism useful for representing relationships between things
- A **graph G** consists of a set of **vertices** and a set of connections linking pairs of vertices. These pairs of vertices are called **edges.**
- A **graph** **G** is represented as **G = (V, E)**
  - **V** is a set of **vertices**: $\{v_1, \ldots, v_n\}$
  - **E** is a set of **edges**: $\{e_1, \ldots, e_m\}$ where each $e_i$ connects two vertices $(v_{i1}, v_{i2})$
- Operations include:
  - iterating over vertices
  - iterating over edges
  - iterating over vertices adjacent to a specific vertex
  - asking whether an edge exists connects two vertices

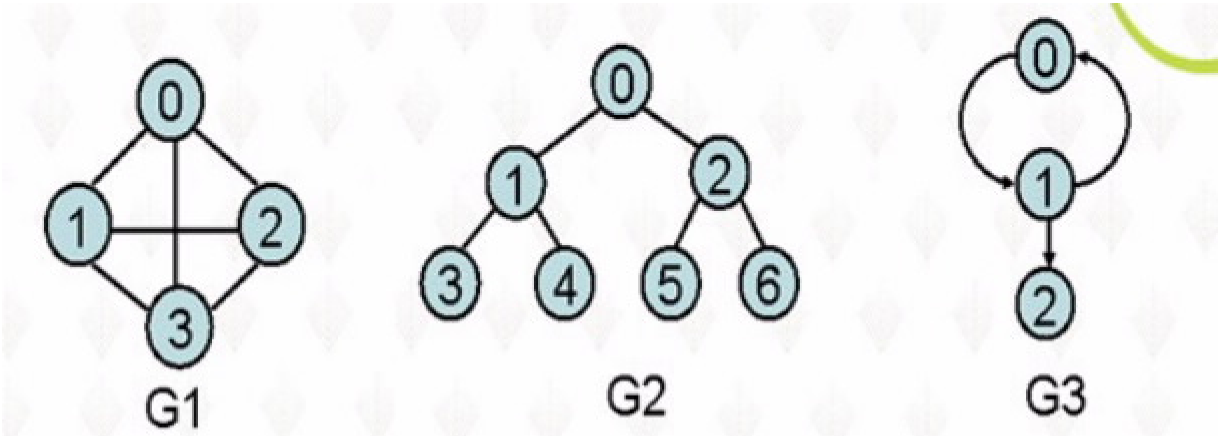
# Basic Concept

- If each $<v_i,v_j>$ in the E is undirected, that is $<v_i,v_j>$ is same as $<v_j,v_i>$, G is called an undirected graph. In undirected graph, the edge $<v_i,v_j>$ can be written as is $(v_i,v_j)$ .

- If each $<v_i,v_j>$ in the E is directed, G is called a directed graph（ digraph ）. In directed graph, the edge $<v_i,v_j>$ is also called arcs.

- Complete graph: a graph that has the maximum number of edges
  - Undirected graph (n vertices)----n(n-1)/2
  - Directed graph (n vertices)-------n(n-1)

# Example




$V(G1) = \{0, 1, 2, 3\}$ $E(G1) = \{ (0,1), (0,2), (0,3), (1,2), (1,3), (2,3) \}$

$V(G2) = \{0, 1, 2, 3, 4, 5, 6\}$ $E(G2) = \{ (0,1), (0,2), (1,3), (1,4), (2,5), (2,6) \}$

$V(G3) = \{0, 1, 2\}$ $E(G3) = \{ \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 2 \rangle \}$

- G1 and G2 are undirected graphs, and G3 is a directed graph.
- G2 is a tree -→tree is a special case of graphs

# Basic Concept

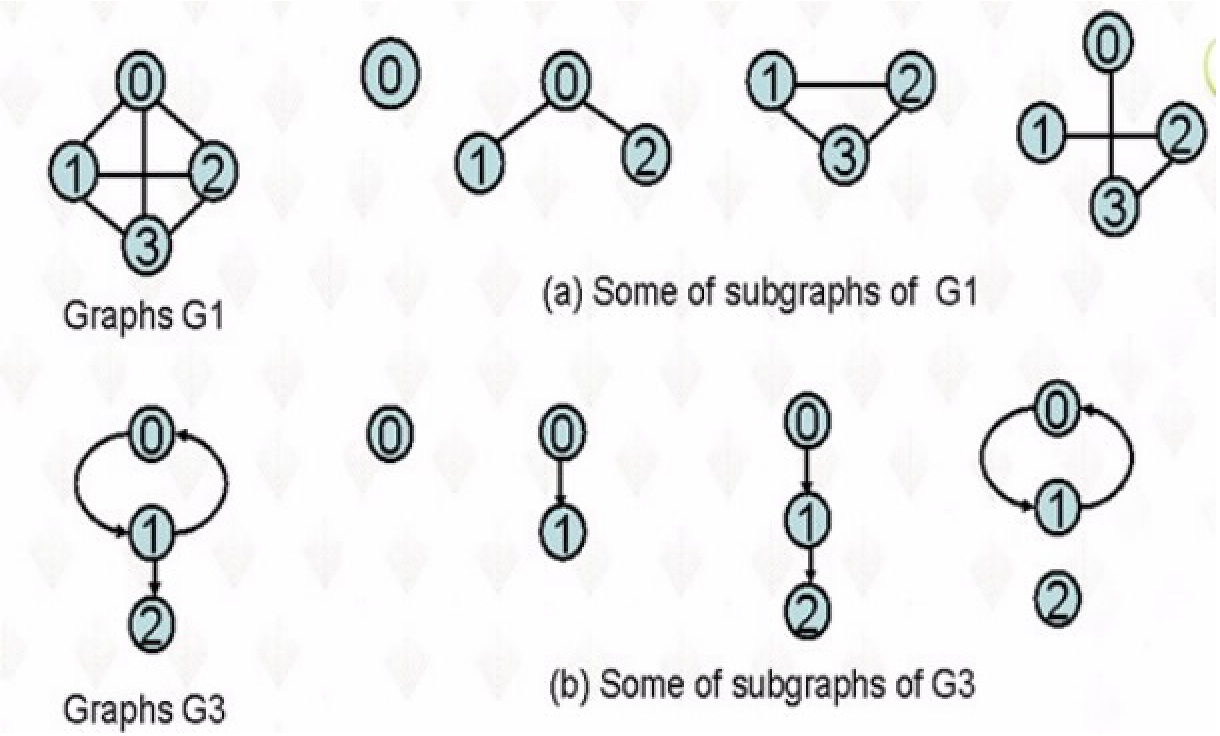
- $(v_i,v_j)$ : vertices $v_i$ and $v_j$ are adjacent (相邻的)

  edge $(v_i,v_j)$ is incident on (相关联) $v_i$ and $v_j$

- $<v_i,v_j>$ vertex $v_i$ is adjacent to vertex $v_j$ , vertex $v_j$ is adjacent from vertex $v_i$ . edge $<v_i,v_j>$ is incident on (相关联) $v_i$ and $v_j$

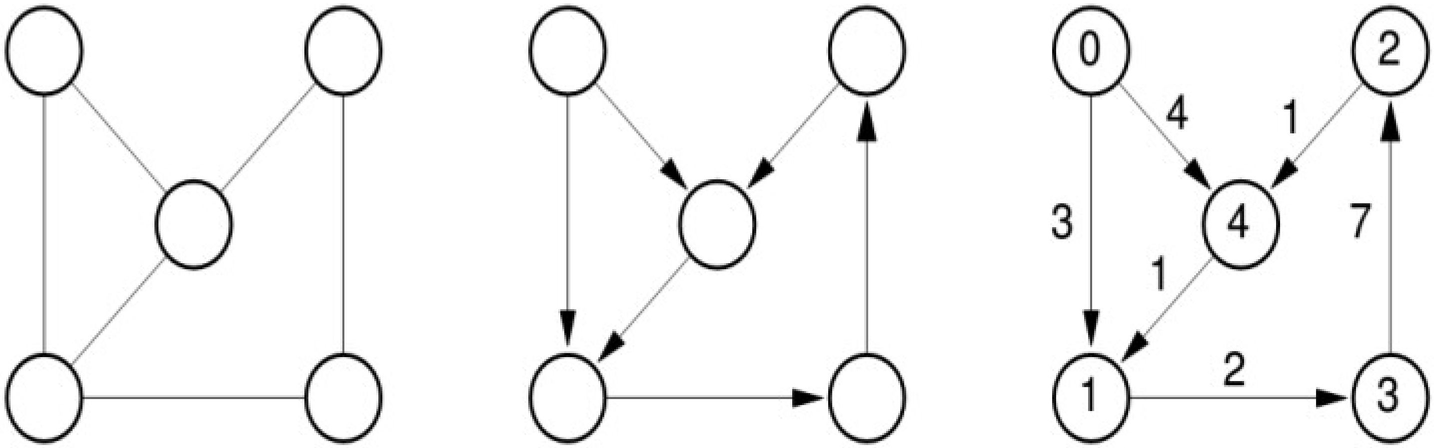
- Weighted graph: graphs whose each edge has a weight.
- Subgraph: Assume there are two graphs G=(V,E) and G′=(V′,E′). If V′ ≤V,and E′ ≤E, G′ is called subgraph of G.

# Example



Graphs G1
(a) Some of subgraphs of G1
Graphs G3
(b) Some of subgraphs of G3

# Example




(a) (b) (c)

# Graph Applications

- Modeling connectivity in computer and communications networks.
- Representing a map as a set of locations with distances between locations; used to compute shortest routes between locations.
- Modeling flow capacities in transportation networks.
- Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving.
- Modeling computer algorithms, showing transitions from one program state to another.
- Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.

# Graph ADT

```
// Graph abstract class. This ADT assumes that the number
// of vertices is fixed when the graph is created.
class Graph {
private:
    void operator =(const Graph&) {} // Protect assignment
    Graph(const Graph&) {} // Protect copy constructor
public:
    Graph() {} // Default constructor
    virtual ~Graph() {} // Base destructor

    // Initialize a graph of n vertices
    virtual void Init(int n) =0;

    // Return: the number of vertices and edges
    virtual int n() =0;
    virtual int e() =0;
```

# Graph ADT

```
// Return v's first neighbor
virtual int first(int v) =0;

// Return v's next neighbor
virtual int next(int v, int w) =0;

// Set the weight for an edge
// i, j: The vertices
// wgt: Edge weight
virtual void setEdge(int v1, int v2, int wght) =0;

// Delete an edge
// i, j: The vertices
virtual void delEdge(int v1, int v2) =0;
```

# Graph ADT

```
// Determine if an edge is in the graph
// i, j: The vertices
// Return: true if edge i,j has non-zero weight
virtual bool isEdge(int i, int j) =0;

// Return an edge's weight
// i, j: The vertices
// Return: The weight of edge i,j, or zero
virtual int weight(int v1, int v2) =0;

// Get and Set the mark value for a vertex
// v: The vertex
// val: The value to set
virtual int getMark(int v) =0;
virtual void setMark(int v, int val) =0;
};
```

# Graph Representations

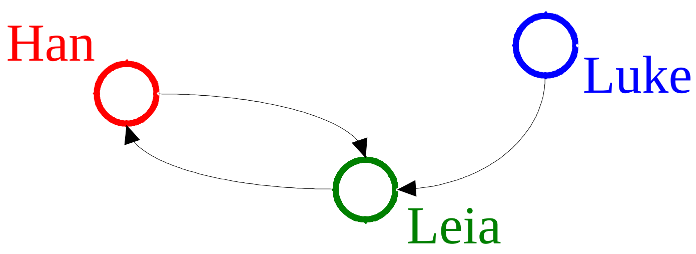
- List of vertices + list of edges

- 2-D matrix of vertices (marking edges in the cells)
  "adjacency matrix"

- List of vertices each with a list of adjacent vertices
  "adjacency list"

# Adjacency Matrix

- A $|V| \times |V|$ array in which an element $(u, v)$ is true if and only if there is an edge from **u** to **v**

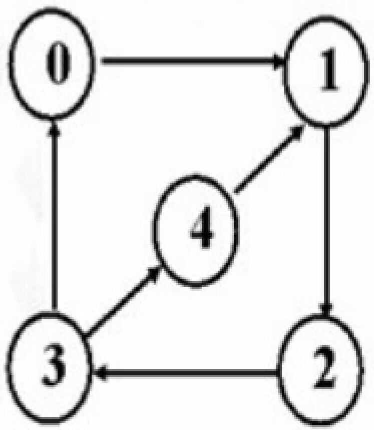



| | Han | Luke | Leia |
|---|---|---|---|
| Han | | | |
| Luke | | | |
| Leia | | | |

# Adjacency Matrix

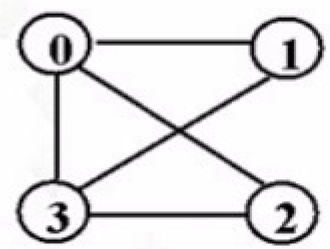
- A graph may be represented with a two-dimensional array. If G has n=|v| vertices, let M be an n×n matrix whose entries are defined by:

$$M_{ij} = \begin{cases} 1 & \text{if } \langle i,j \rangle \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$$




$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

# Adjacency Matrix



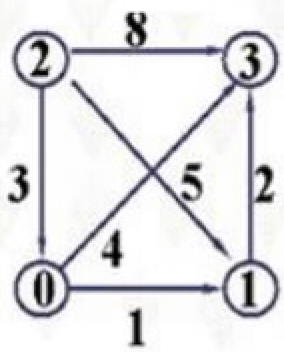

$$M = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

**Worst case:**

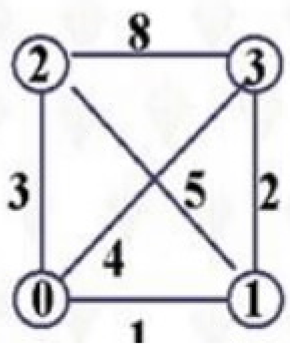
- O(1): to determine existence of a specific edge
- $O(|V|^2)$ : storage cost
- O(|V|) : for finding all vertices accessible from a specific vertex
- O(1): to add or delete an edge
- Not  easy to add or delete a vertex; better for static graph structure
- Symmetric (对称)：matrix for undirected graph; so half if redundant then.

# Adjacency Matrix

$$M_{ij} = \begin{cases} w_{ij} & \text{if } \langle i,j\rangle \in E,\ w_{ij} \text{ is the weight with } \langle i,j\rangle \\ 0 & \text{otherwise} \end{cases}$$




$$M = \begin{bmatrix} 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 2 \\ 3 & 5 & 0 & 8 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$




$$M = \begin{bmatrix} 0 & 1 & 3 & 4 \\ 1 & 0 & 5 & 2 \\ 3 & 5 & 0 & 8 \\ 4 & 2 & 8 & 0 \end{bmatrix}$$

# Adjacency Matrix Implementation

```
// Implementation for the adjacency matrix representation
class Graphm : public Graph {
private:
  int numVertex, numEdge; // Store number of vertices, edges
  int **matrix;           // Pointer to adjacency matrix
  int *mark;              // Pointer to mark array
public:
  Graphm(int numVert)     // Constructor
    { Init(numVert); }

  ~Graphm() {        // Destructor
    delete [] mark; // Return dynamically allocated memory
    for (int i=0; i<numVertex; i++)
      delete [] matrix[i];
    delete [] matrix;
  }
```

# Adjacency Matrix Implementation

```
void Init(int n) { // Initialize the graph
  int i;
  numVertex = n;
  numEdge = 0;
  mark = new int[n];       // Initialize mark array
  for (i=0; i<numVertex; i++)
    mark[i] = UNVISITED;
  matrix = (int**) new int*[numVertex]; // Make matrix
  for (i=0; i<numVertex; i++)
    matrix[i] = new int[numVertex];
  for (i=0; i< numVertex; i++) // Initialize to 0 weights
    for (int j=0; j<numVertex; j++)
      matrix[i][j] = 0;
}
```

# Adjacency Matrix Implementation

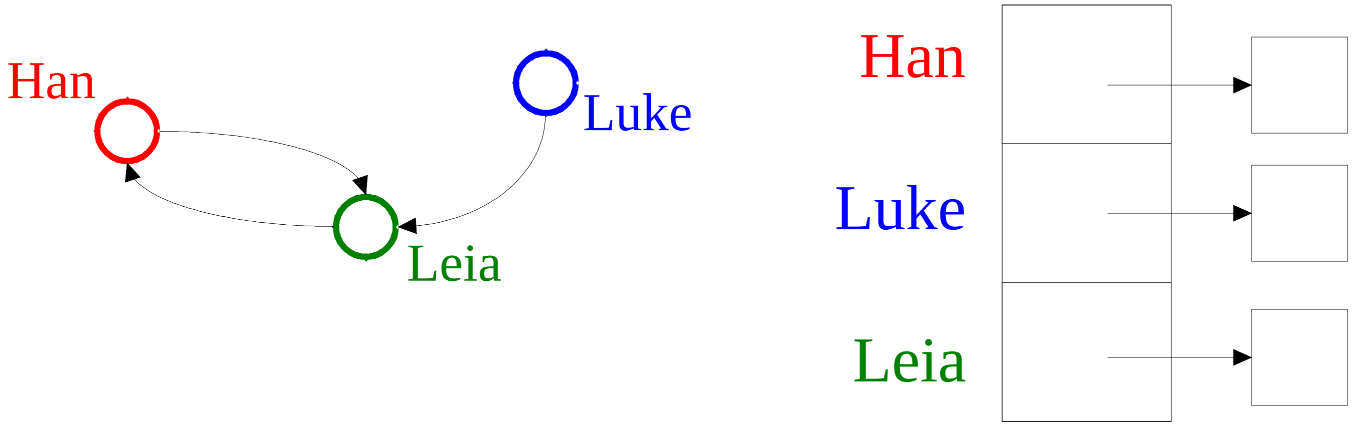
```
int n() { return numVertex; } // Number of vertices
int e() { return numEdge; }   // Number of edges

// Return first neighbor of "v"
int first(int v) {
  for (int i=0; i<numVertex; i++)
    if (matrix[v][i] != 0) return i;
  return numVertex;           // Return n if none
}

// Return v's next neighbor after w
int next(int v, int w) {
  for(int i=w+1; i<numVertex; i++)
    if (matrix[v][i] != 0)
      return i;
  return numVertex;           // Return n if none
}
```

# Adjacency Matrix Implementation

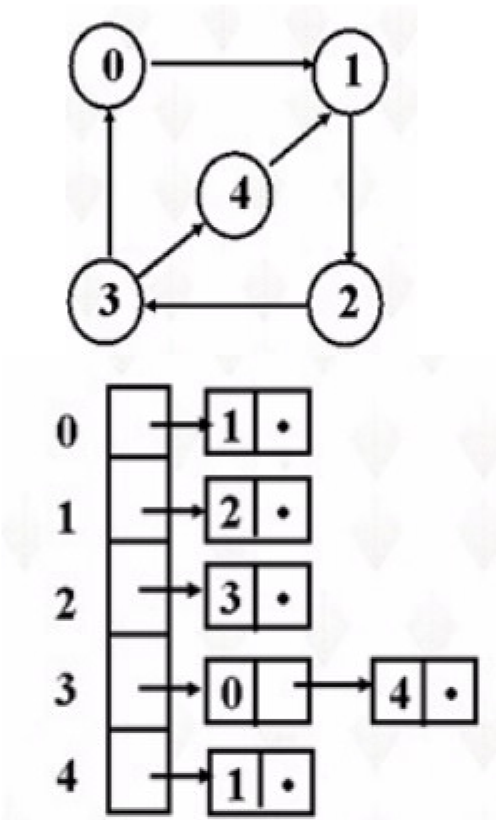
```
// Set edge (v1, v2) to "wt"
void setEdge(int v1, int v2, int wt) {
  Assert(wt>0, "Illegal weight value");
  if (matrix[v1][v2] == 0) numEdge++;
  matrix[v1][v2] = wt;
}

void delEdge(int v1, int v2) { // Delete edge (v1, v2)
  if (matrix[v1][v2] != 0) numEdge--;
  matrix[v1][v2] = 0;
}

bool isEdge(int i, int j) // Is (i, j) an edge?
{ return matrix[i][j] != 0; }

int weight(int v1, int v2) { return matrix[v1][v2]; }
int getMark(int v) { return mark[v]; }
void setMark(int v, int val) { mark[v] = val; }
};
```

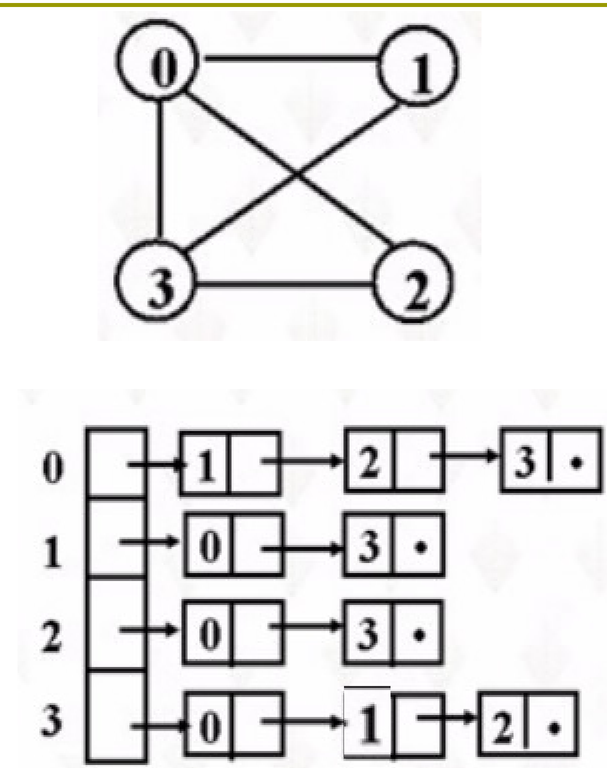
# Adjacency List

- A **|V|**-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices (or edges)

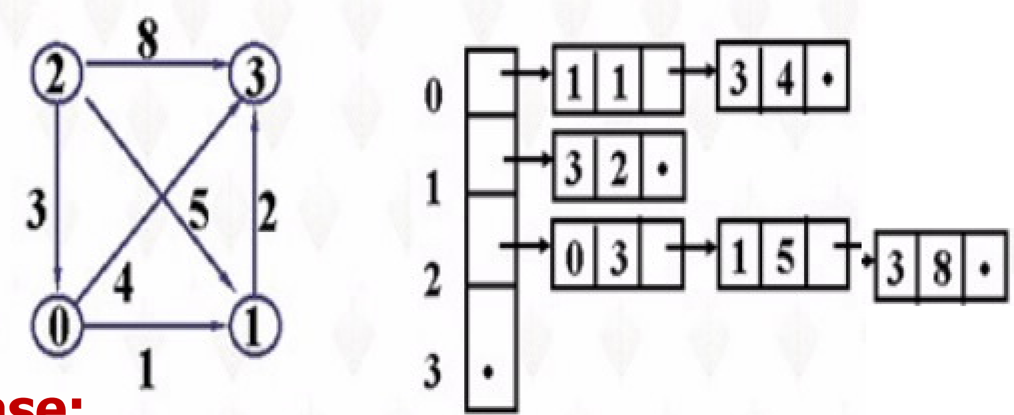


# Adjacency List





# Adjacency List




**Worst case:**

- O(|V|): to determine existence of a specific edge
- O(|V|+|E|) : storage cost
- O(|V|) : for finding all neighbors of a specific vertex
- O(|V|): to add or delete an edge
- Still not  easy to add or delete a vertex; however, we can use a linked list in place of the arrya

# Adjacency List Implementation

```
// Edge class for Adjacency List graph representation
class Edge {
  int vert, wt;
public:
  Edge() { vert = -1; wt = -1; }
  Edge(int v, int w) { vert = v; wt = w; }
  int vertex() { return vert; }
  int weight() { return wt; }
};
```

# Adjacency List Implementation

```
class Graphl : public Graph {
private:
  List<Edge>** vertex;          // List headers
  int numVertex, numEdge;       // Number of vertices, edges
  int *mark;                    // Pointer to mark array
public:
  Graphl(int numVert)
    { Init(numVert); }

  ~Graphl() {        // Destructor
    delete [] mark; // Return dynamically allocated memory
    for (int i=0; i<numVertex; i++) delete [] vertex[i];
    delete [] vertex;
  }
```

# Adjacency List Implementation

```
void Init(int n) {
  int i;
  numVertex = n;
  numEdge = 0;
  mark = new int[n];  // Initialize mark array
  for (i=0; i<numVertex; i++) mark[i] = UNVISITED;
  // Create and initialize adjacency lists
  vertex = (List<Edge>**) new List<Edge>*[numVertex];
  for (i=0; i<numVertex; i++)
    vertex[i] = new LList<Edge>();
}

int n() { return numVertex; } // Number of vertices
int e() { return numEdge; }   // Number of edges
```

# Adjacency List Implementation

```
int first(int v) { // Return first neighbor of "v"
  if (vertex[v]->length() == 0)
    return numVertex;        // No neighbor
  vertex[v]->moveToStart();
  Edge it = vertex[v]->getValue();
  return it.vertex();
}
```

# Adjacency List Implementation

```
// Get v's next neighbor after w
int next(int v, int w) {
  Edge it;
  if (isEdge(v, w)) {
    if ((vertex[v]->currPos()+1) < vertex[v]->length()) {
      vertex[v]->next();
      it = vertex[v]->getValue();
      return it.vertex();
    }
  }
  return n(); // No neighbor
}
```

# Adjacency List Implementation

```
// Set edge (i, j) to "weight"
void setEdge(int i, int j, int weight) {
  Assert(weight>0, "May not set weight to 0");
  Edge currEdge(j, weight);
  if (isEdge(i, j)) { // Edge already exists in graph
    vertex[i]->remove();
    vertex[i]->insert(currEdge);
  }
  else { // Keep neighbors sorted by vertex index
    numEdge++;
    for (vertex[i]->moveToStart();
         vertex[i]->currPos() < vertex[i]->length();
         vertex[i]->next()) {
      Edge temp = vertex[i]->getValue();
      if (temp.vertex() > j) break;
    }
    vertex[i]->insert(currEdge);
  }
}
```

# Adjacency List Implementation

```
void delEdge(int i, int j) {  // Delete edge (i, j)
  if (isEdge(i,j)) {
    vertex[i]->remove();
    numEdge--;
  }
}

bool isEdge(int i, int j) { // Is (i,j) an edge?
  Edge it;
  for (vertex[i]->moveToStart();
       vertex[i]->currPos() < vertex[i]->length();
       vertex[i]->next()) {              // Check whole list
    Edge temp = vertex[i]->getValue();
    if (temp.vertex() == j) return true;
  }
  return false;
}
```

# Adjacency List Implementation

```
  int weight(int i, int j) { // Return weight of (i, j)
    Edge curr;
    if (isEdge(i, j)) {
      curr = vertex[i]->getValue();
      return curr.weight();
    }
    else return 0;
  }

  int getMark(int v) { return mark[v]; }
  void setMark(int v, int val) { mark[v] = val; }
};
```

# Knowledge Points

- Chapter 11, pp.381-392

# Homework

- P410, 11.3

-End-