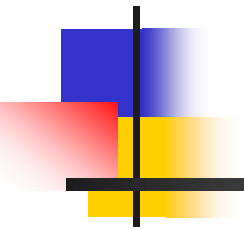
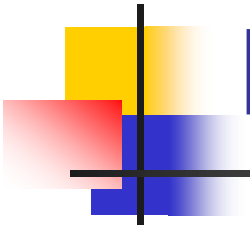


CS 2213
Advanced Programming
Ch 16 – Graphs (Networks)
Definitions and Graph Representations

Turgay Korkmaz

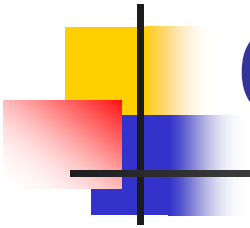
Office: SB 4.01.13
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

Thanks to Eric S. Roberts, the author of our textbook, for providing some slides/figures/programs. I also used some materials from other textbooks (specifically, The Algorithm Design Manual by **Steven Skiena** and his slides at http://www.cs.sunysb.edu/skiena. So I also thank to **Steven Skiena** for presentations on the Internet).



Disclaimer

- You are highly recommended to read the chapter 16 from our textbook, which provides a general interface-based design for graph abstraction using several other data structures (e.g., list, set) covered in previous chapters.
- Instead of this general abstract design, we will **directly implement concrete graph structures** and operations. So even though the key concepts are the same, the implementation details will be different than the textbook.

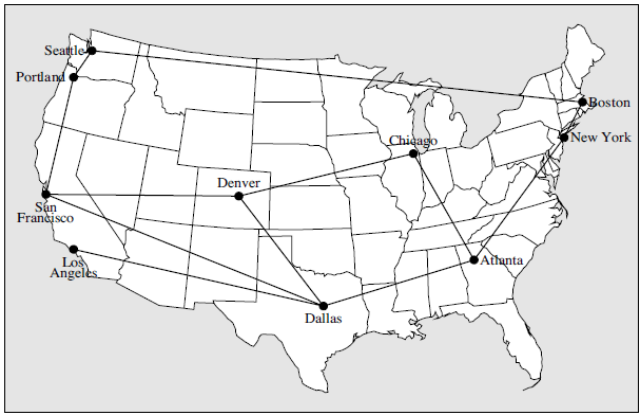


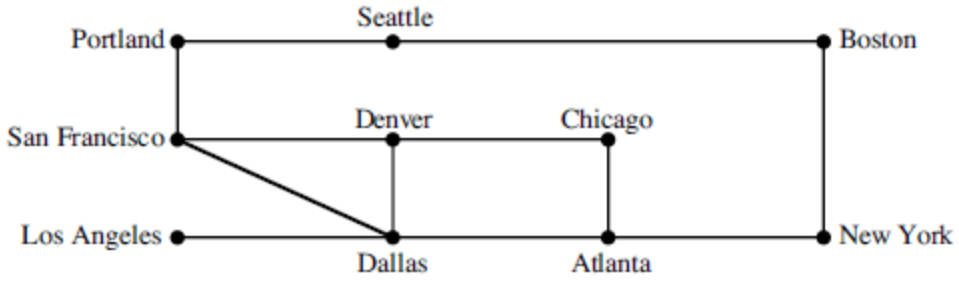
Objectives

- To appreciate the **conceptual** structure of a graph and its **applications**
- To learn basic graph theory **terminology/notation**
- To learn underlying representations for graphs (**adjacency matrix**, **adjacency list**)
- To understand and be able to apply basic graph algorithms (topological sort, depth-first search, breath-first search, Dijkstra's shortest path etc.)
- To provide knowledge and skills to students so that they can comfortably use graph structures and algorithms (or develop new ones) in their research or workplace.

Structure of a graph

A **graph** (or network) is a mathematical object consisting of **nodes** (also called vertices) and **edges** (also called arcs, links).





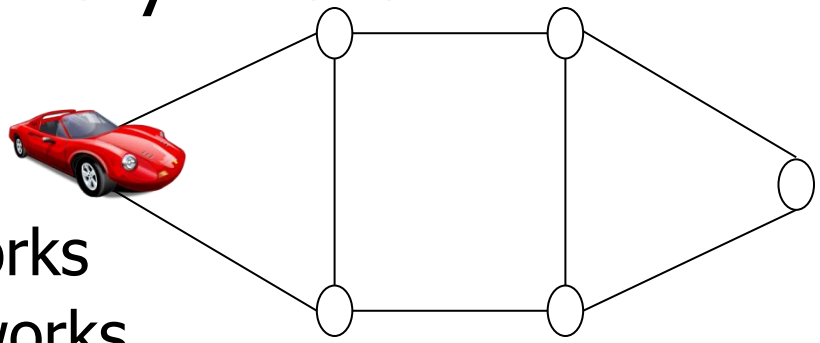
The airline graph represented by $G(V,E)$ consists of the following sets.

V = { Atlanta, Boston, Chicago, Dallas, Denver, Los Angeles,
New York, Portland, San Francisco, Seattle }

E = { Atlanta↔Chicago, Atlanta ↔ Dallas, Atlanta ↔ New York,
Boston ↔ New York, Boston ↔ Seattle, Chicago ↔ Denver,
Dallas ↔ Denver, Dallas ↔ Los Angeles, Dallas ↔ San Francisco,
Denver ↔ San Francisco, Portland ↔ San Francisco, Portland ↔ Seattle }

Applications of graphs

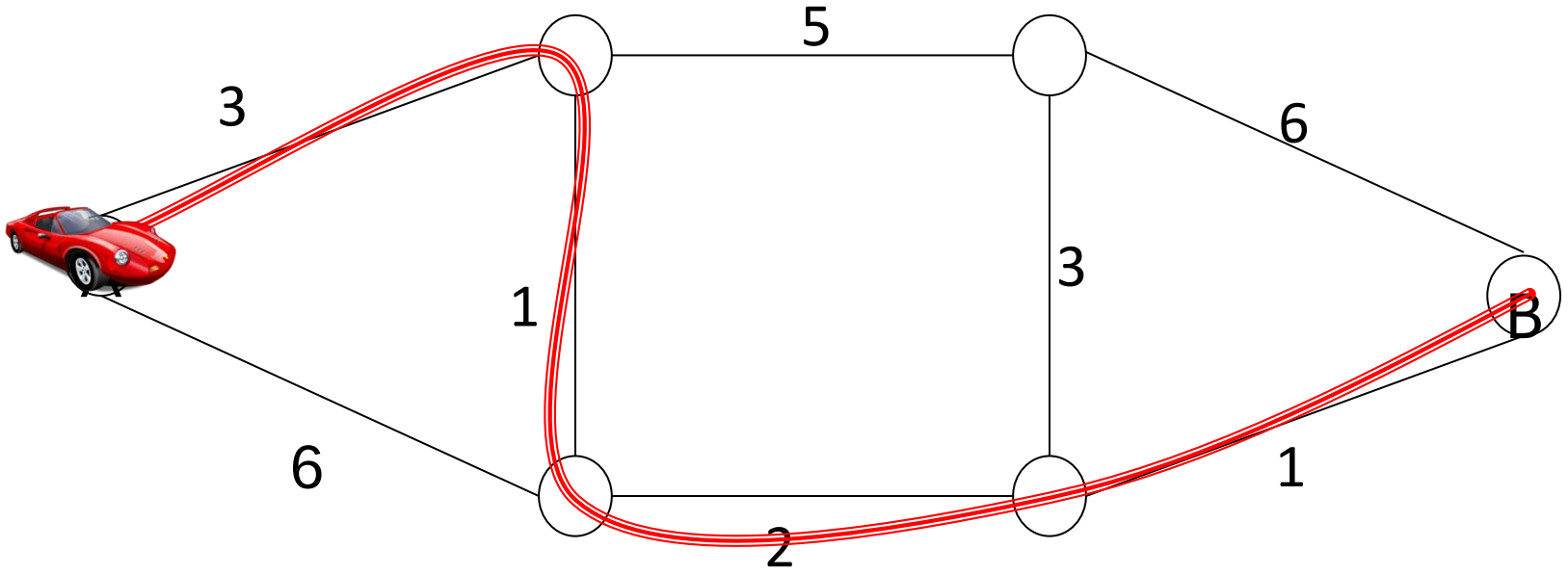
- Graphs (networks) are everywhere!
  - **Physical Networks**
    - Transportation networks
    - Power transmission networks
    - Telephone/computer networks



  - What are the common things in all these problems?
    - We move some **entity** {car, people, electricity, voice, data} from one **node** {intersection, city, generator, telephone, computer} to another through underlying **links** {roads, power lines, communication links}.
  - But, we want to move entities from one point to another in an **efficient** way (what do we mean?)

Application: Shortest path

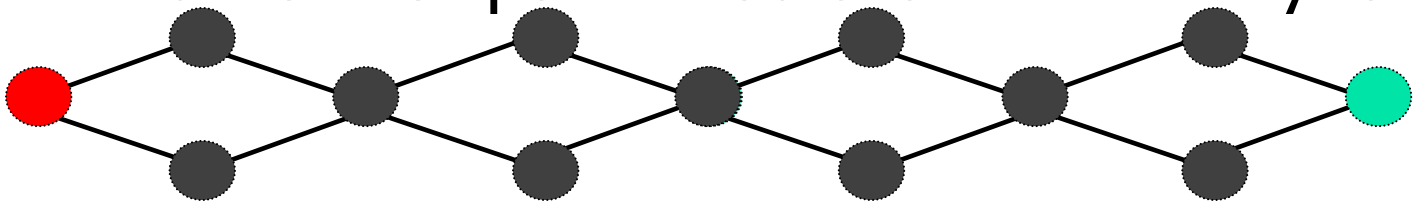
- Assume each link has cost (e.g., distance).



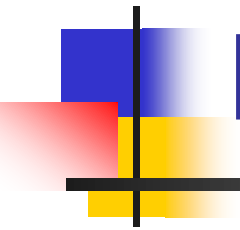
- What is the best way to get from A to B as cheaply as possible? How can we find it?

Finding Shortest path

- Enumerate all possible solutions
  - Measure the "cost" of each alternative solution
  - Select the one with min cost
- From the perspective of pure mathematics, the above problem is trivial to solve! BUT
  - The number of paths could be extremely large



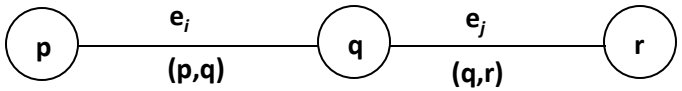
  - 2^n possible solutions, what happens when n=100
- So, we need **efficient** graph algorithms.



Basic Graph Notation

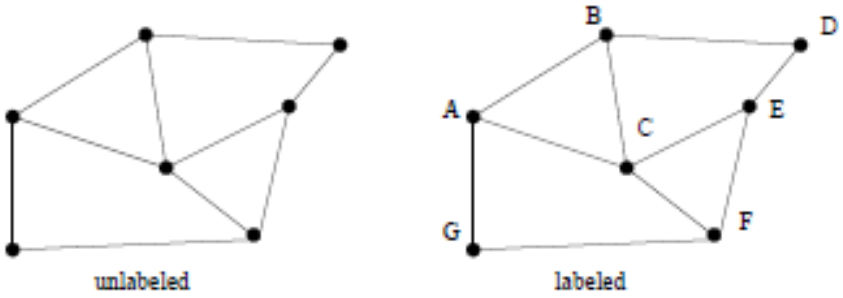
Basic Graph Notation

- **Network or Graph**
  - G = (N, A) or G(V, E)
    - **V = { v_1, v_2, …, v_n }** node/vertex set
    - **E = { e_1, e_2,…, e_m }** arc/link/edge set
    - Link/edge connecting v_p to v_q is also denoted by *(p,q)*
  - **Incidence and adjacency**:
    - e_i and e_j are adjacent links
    - v_p and v_q are adjacent nodes
    - e_i and e_j are incident to node v_q



Basic Graph Notation (cont'd)

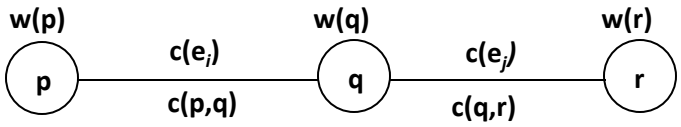
- **Labeled vs. Unlabeled Graphs**
  - Each vertex is assigned a unique name or identifier to distinguish it from other vertices.

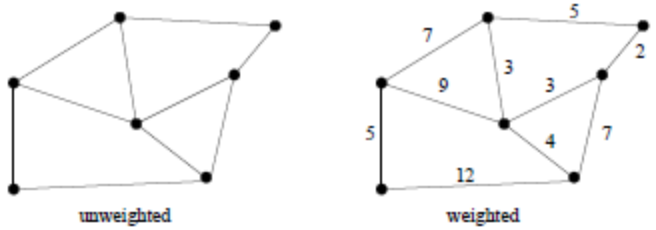


  - An important graph problem is *isomorphism testing,* determining whether the topological structure of two graphs are in fact identical if we ignore any labels.

Basic Graph Notation (cont'd)

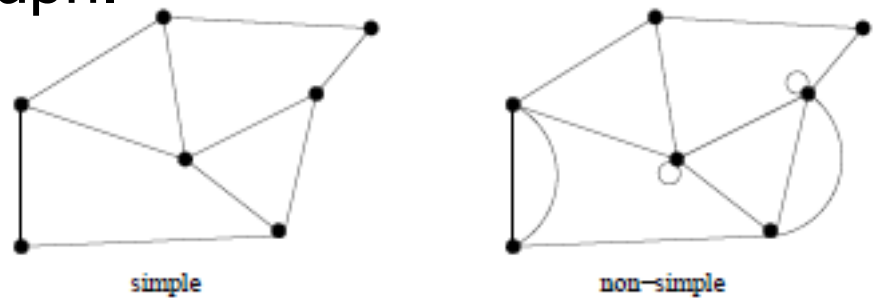
- **Weighted vs. Unweighted Graphs**
  - In *unweighted* graphs, there is no cost distinction between various edges and vertices.
  - In *weighted* graphs, we can associate weights/labels/attributes with nodes or links,
    - for node *p:* **w(*p*),** or w_p
    - *for link (p,q)*: ***c(p,q)*** or c_{pq}





Basic Graph Notation (cont'd)

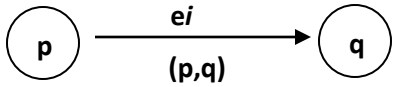
- **Simple vs. Non-simple Graphs**
  - A ***self-loop*** *is an edge (x,x) involving only one* node
  - An edge (x,y) is a ***multi-edge*** if it occurs more than once in the graph.



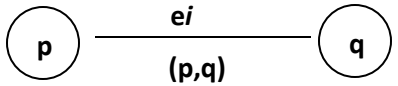
- Any graph which avoids these structures (loops, multi edges) is called ***simple***.

Basic Graph Notation (cont'd)

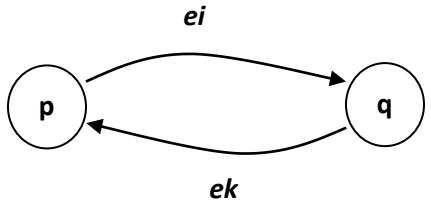
- **Directed and Undirected graphs**:
  - Directed network:
    - Flow is in one direction, e.g., from v_p to v_q



  - Undirected network:
    - Flow can be in any direction with the same attributes.

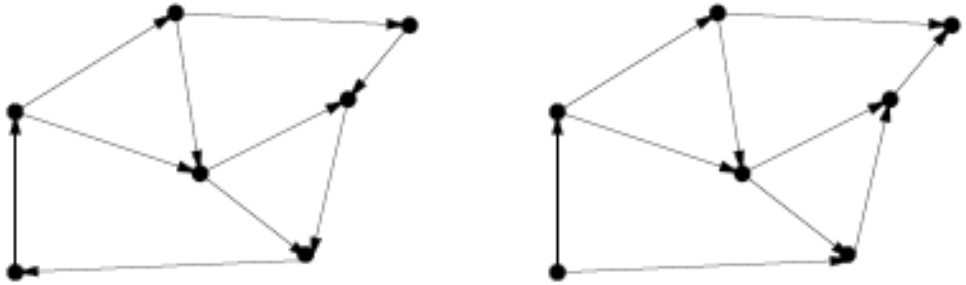


  - If link attributes are different, we can use directed network model



Basic Graph Notation (cont'd)

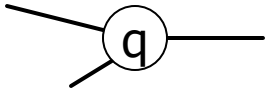
- An ***acyclic*** graph does not contain any cycles.



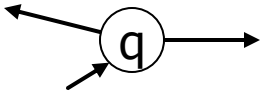
- **D**irected **A**cyclic **G**raphs are called *DAGs.*
  - They arise naturally in scheduling problems, where a directed edge (x, y) indicates that x must occur before y.

Basic Graph Notation (cont'd)

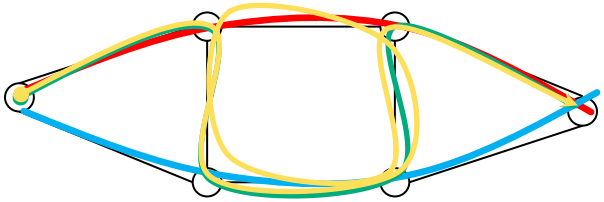
- **Degree**: the number of incident to a node
  - deg(v_q) is 3



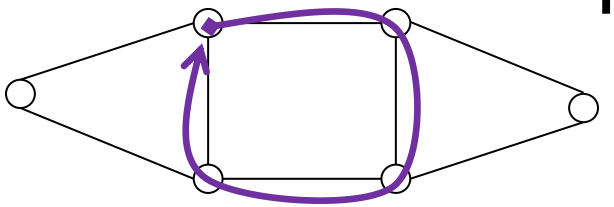
  - If Directed, in-degree, out-degree



- **Path**: a **distinct** sequence of nodes
  - Undirected vs. directed

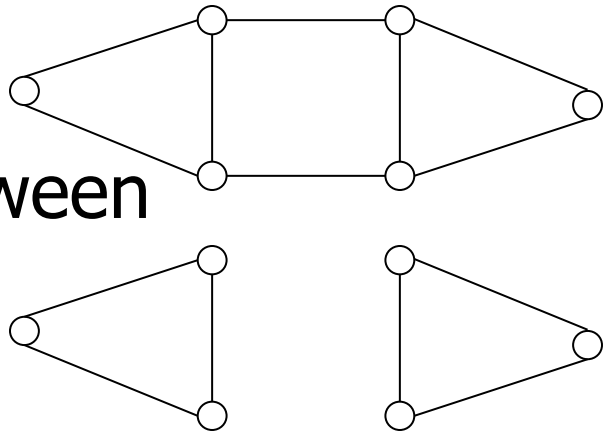


- **Walk**: a sequence of nodes
- **Cycle**: a distinct sequence of nodes except the first and last node

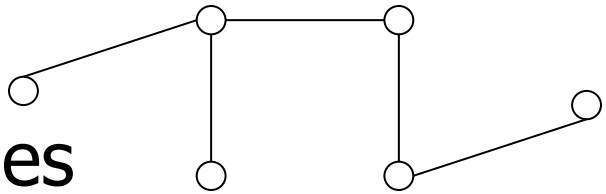


Basic Graph Notation (cont'd)

- **Connected graph**
  - There is at least one path between every pair of nodes.
  - Connected components
  - Incase of directed graphs
    - Strongly connected
    - Weakly connected

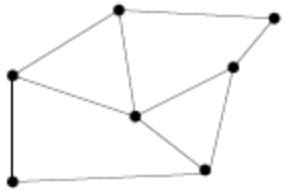


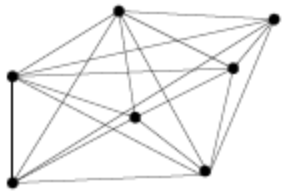
- **Tree**
  - A connected graph with no cycles

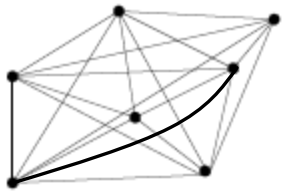


Basic Graph Notation (cont'd)

- **Sparse vs. Dense Graphs**
  - Graphs are ***sparse*** when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
  - Graphs are ***dense*** when the number of edges increases
  - ***Complete*** graph: there is an edge between every pair of nodes
  - Dense graphs have a **quadratic** number of edges while sparse graphs are **linear in size**.

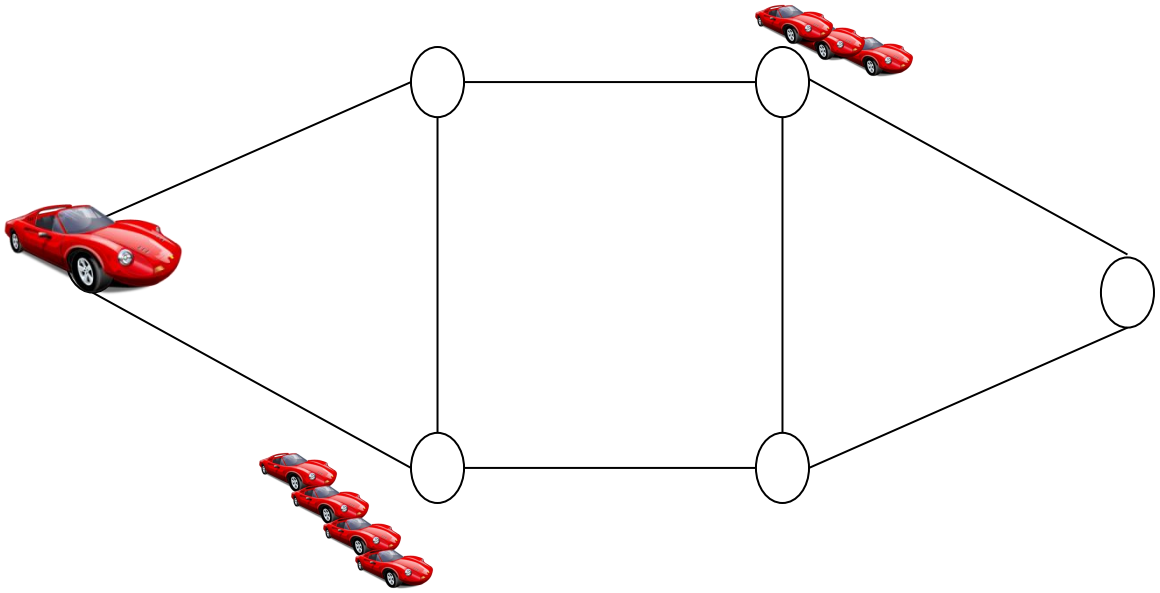

sparse


dense

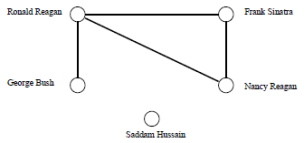

Complete

Basic Graph Notation (cont'd)

- **Some important nodes**
  - **Source node**: (or starting point) a node where flow originates
  - **Destination/Sink**: (or termination point) a node where flow ends
  - **Supply node**: a node where supply for flow is available
  - **Demand node**: a node where non-zero demand for flow exists



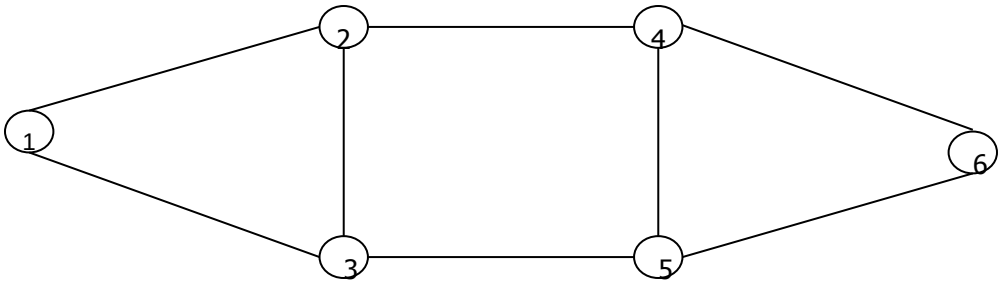
Example: The Friendship Graph



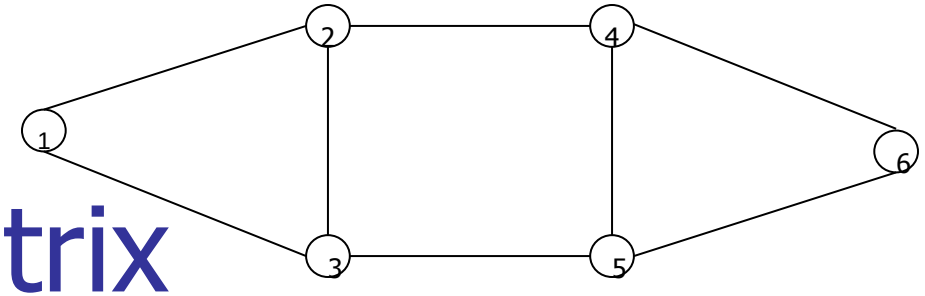
- Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.
- What questions might we ask about the friendship graph?
  - If I am your friend, does that mean you are my friend? (Undirected)
  - Am I my own friend? (self-loop, simple)
  - Am I linked by some chain of friends to the President? (path)
  - How close is my link to the President? (shortest path)
  - Is there a path of friends between any two people? (strongly connected)
  - Who has the most friends? (degree)
  - Is there a clique? a **clique** in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge.

Data Structures for Graphs

Suppose the graph G = (V, E) contains |V|=n vertices and |E|=m edges.



How can we represent this graph in C?



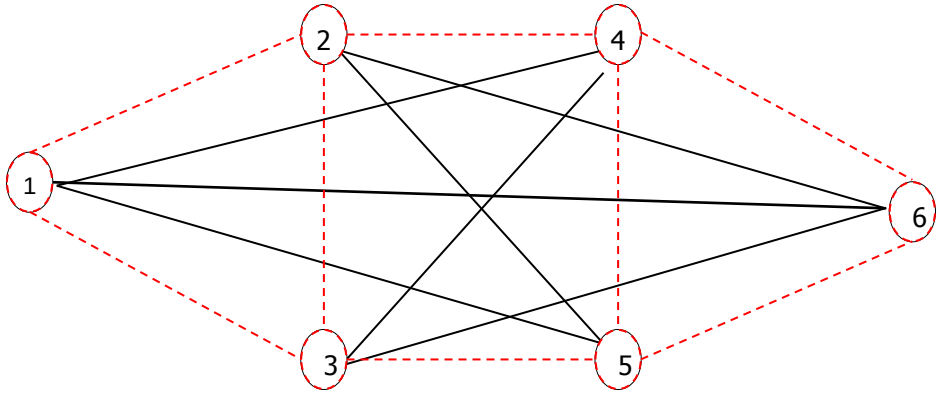
Adjacency Matrix

- We can represent G(V, E) using an n x n matrix M, where element M[i][j] will be
  - 1 if (i,j) is an edge of G, and
  - 0 if it isn't.
- Is this a good representation
  - for dense graphs
  - for sparse graphs,
- Can we save space,
  - if the graph is undirected?
  - if the graph is sparse?

	0	1	2	3	4	5	6
0							
1			1	1			
2		1		1	1		
3		1	1			1	
4			1			1	1
5				1	1		1
6					1	1	

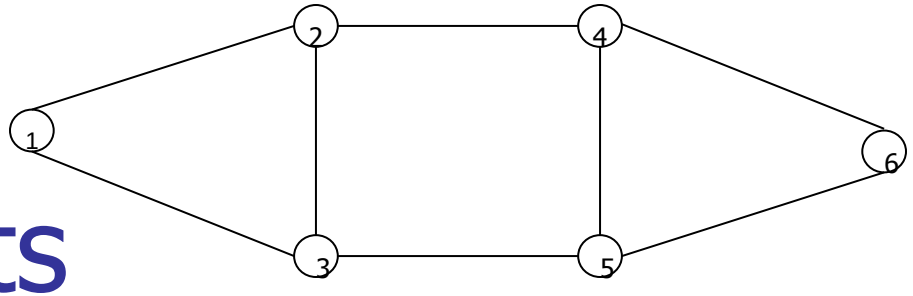
Each element M[i][j] might be a structure to represent other attributes of link (i,j)

Exercise: Adjacency Matrix for the complement graph



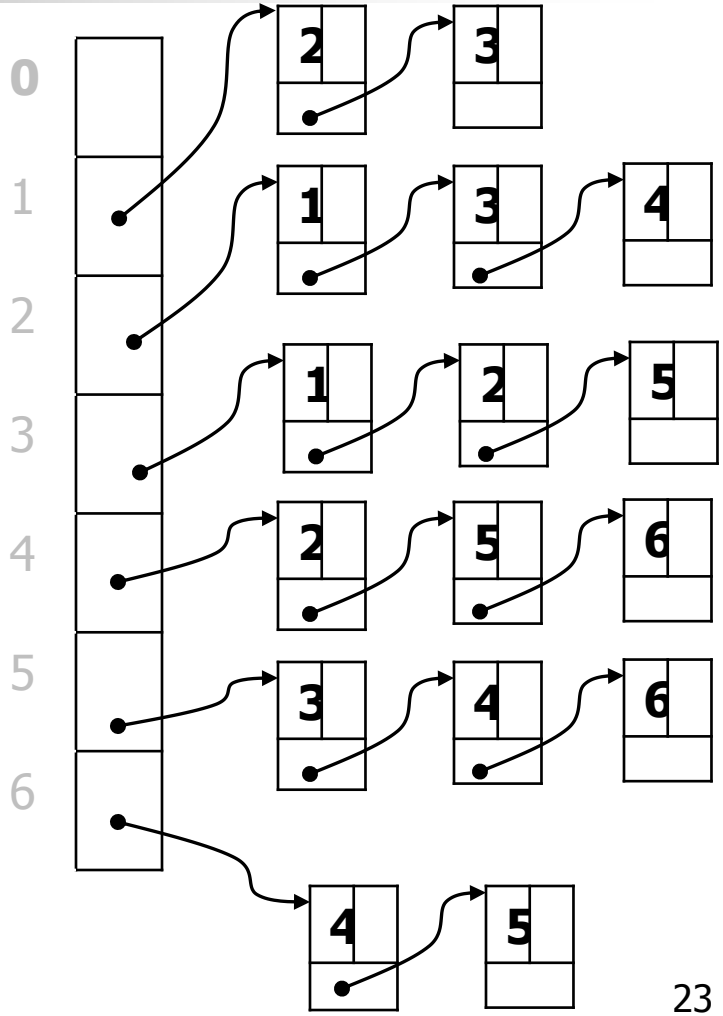
Modify this matrix

	0	1	2	3	4	5	6
0							
1			1	1			
2		1		1	1		
3		1	1			1	
4			1			1	1
5				1	1		1
6					1	1	

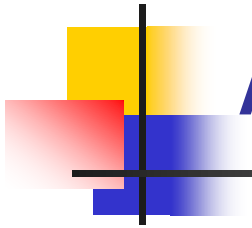


Adjacency Lists

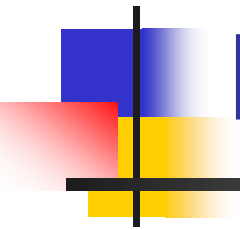
- We can represent G(V, E) using an n$\times$1 array of **pointers**, where the ith element points to a **linked list** of the edges incident on vertex *i*.
- Is this a good representation
  - for dense graphs
  - for sparse graphs



Adjacency Lists vs. Adjacency Matrices



- Which representation would be better for the followings, also give complexity:
  - Test if (x, y) exists? Matrices O(1)
  - Find vertex degree? Lists O(degree)
  - Memory usage for sparse graphs? lists (m+ n) vs. (n^2)
  - Memory usage for dense graphs? matrices O(n^2)
  - Edge insertion or deletion? matrices O(1)
  - Faster to traverse the graph? lists (m+ n) vs. (n^2)
- Better for most problems? lists



Adjacency List Representation in C

Concrete Graph Structure

```
#define MAXV 100

typedef  struct  edgenode {
   int y;
   int w; //weight
   struct  edgenode  *next;
} edgenodeT;

typedef struct {
   edgenodeT *edges[MAXV+1];
   int degree[MAXV+1];
   int nvertices;
   int nedges;
   bool directed;
} graphT;
```

We will assume that we will have at most MAXV many nods in our graphs; but , we can dynamically change graph size. How?

adjacency info x - - - - > y
link weight/cost if any
pointer to next link

- The edges field is an array of pointers to list of edges originating from each node x.
- The degree field counts the number of meaningful entries for the given vertex.
- An undirected edge (x, y) appears twice in any adjacency-based graph structure, once as y in x's list, and once as x in y's list.

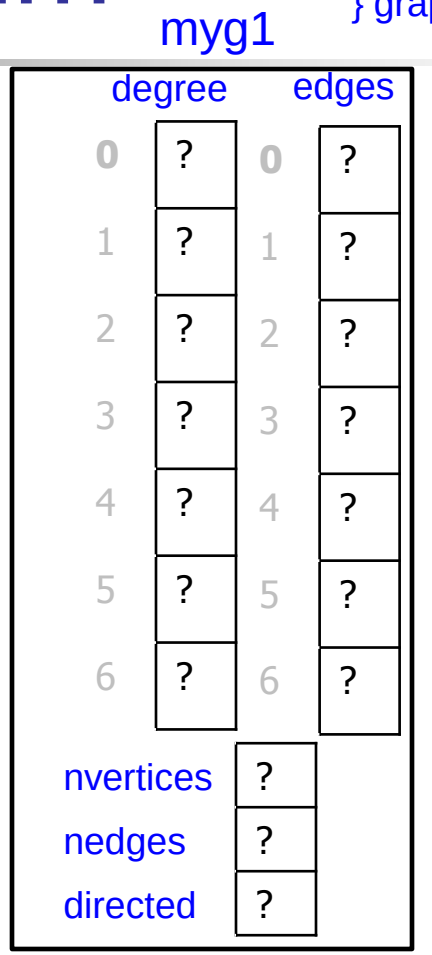
Creating a graph in a client program

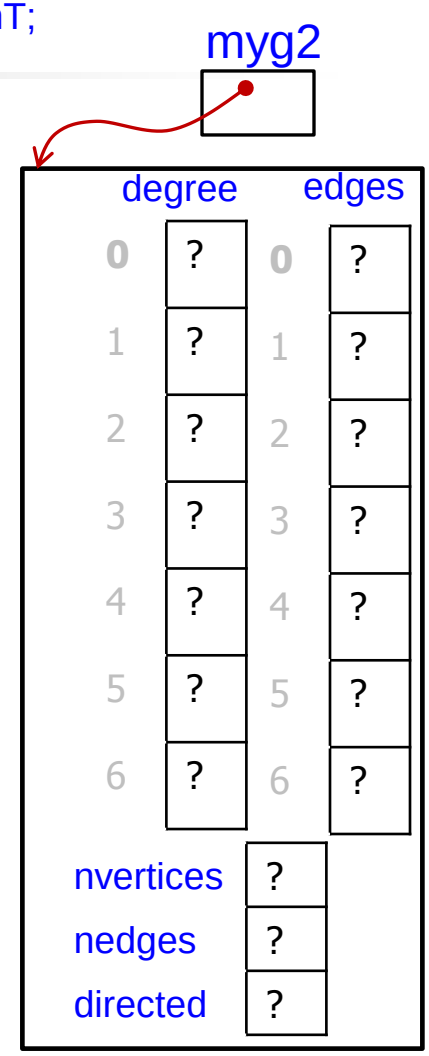
```
typedef struct {
   edgenodeT *edges[MAXV+1];
   int degree[MAXV+1];
   int nvertices;
   int nedges;
   bool directed;
} graphT;
```

```
main() // Assume that MAXV is 6
{
  graphT   myg1, *myg2;

  myg2 = (graphT *) malloc(sizeof(graphT));
  if(myg2==NULL) {
     printf("no memory");
     exit(-1);
  }     // myg2 = New(graphT *);
  initialize_graph(&myg1, TRUE);

  initialize_graph(myg2, FALSE);
}
```



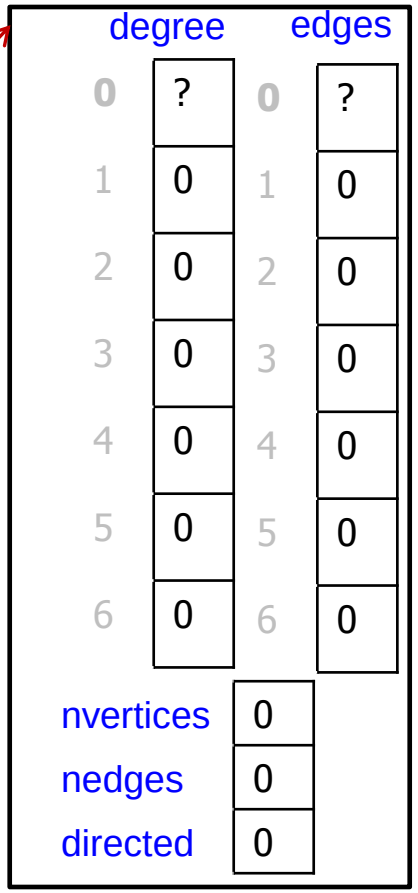


Initializing a Graph

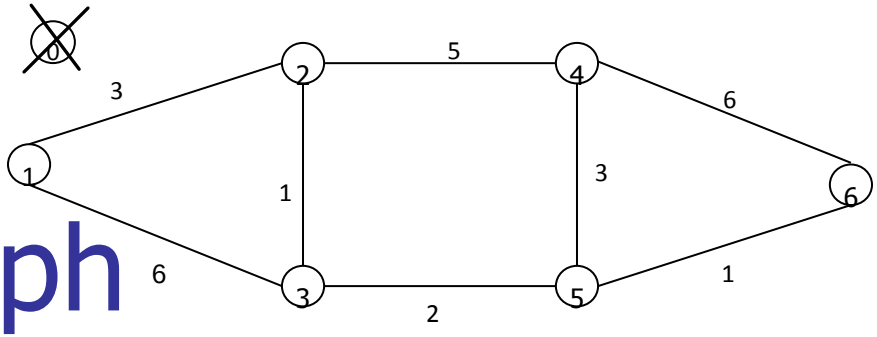
```
typedef struct {
   edgenodeT *edges[MAXV+1];
   int degree[MAXV+1];
   int nvertices;
   int nedges;
   bool directed;
} graphT;
```

```
initialize_graph(graphT *g, bool directed)
{
   int i;
   g->nvertices = 0;
   g->nedges = 0;
   g->directed = directed;

   for (i=1; i<=MAXV; i++) {
      g->edges[i] = NULL;
     g->degree[i] = 0;
   }
}
```

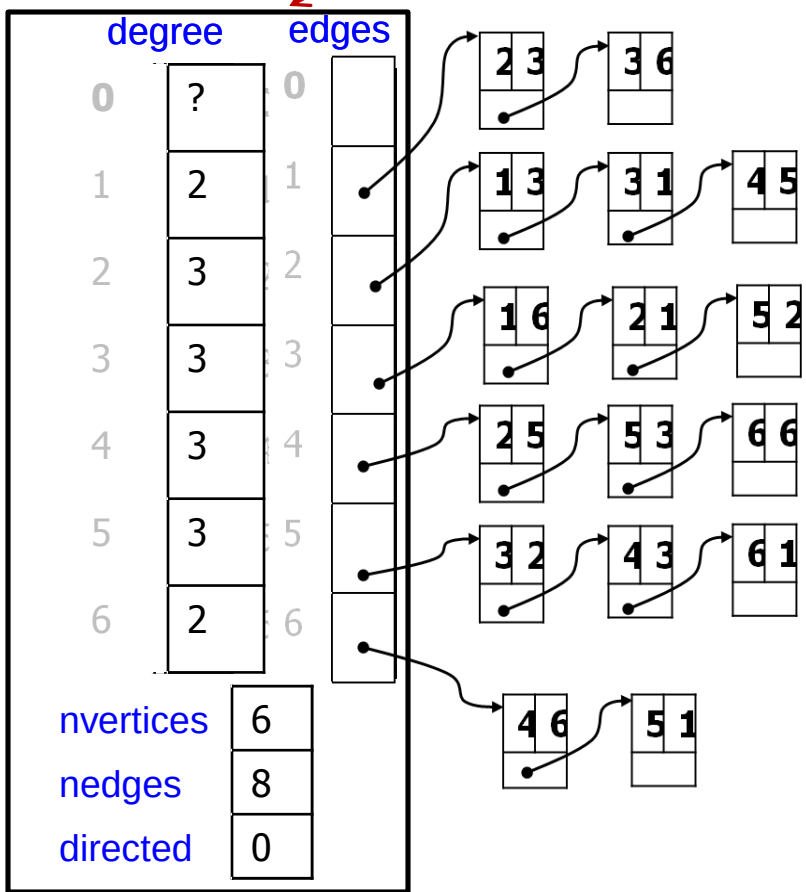


Reading a Graph



A typical graph format consists of an initial line featuring the number of vertices and edges in the graph, followed by a listing of the edges at one vertex pair per line and weight of that edge.

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




```
read_graph(graphT *g, bool directed)
{
   int i;
   int n,m;
   int x, y, w;

   scanf("%d %d", &n, &m);
   g->nvertices=n; g->nedges=m;
   for (i=1; i<=m; i++) {
      scanf("%d %d %d", &x, &y, &w);
      insert _edge(g, x, y, w, directed);
   }
}
```

FALSE

Inserting an edge (suppose we have new edge 1,5 cost is 7)

```
typedef struct edgenode {
   int y;
   int w;
   struct edgenode *next;
} edgenodeT;
```

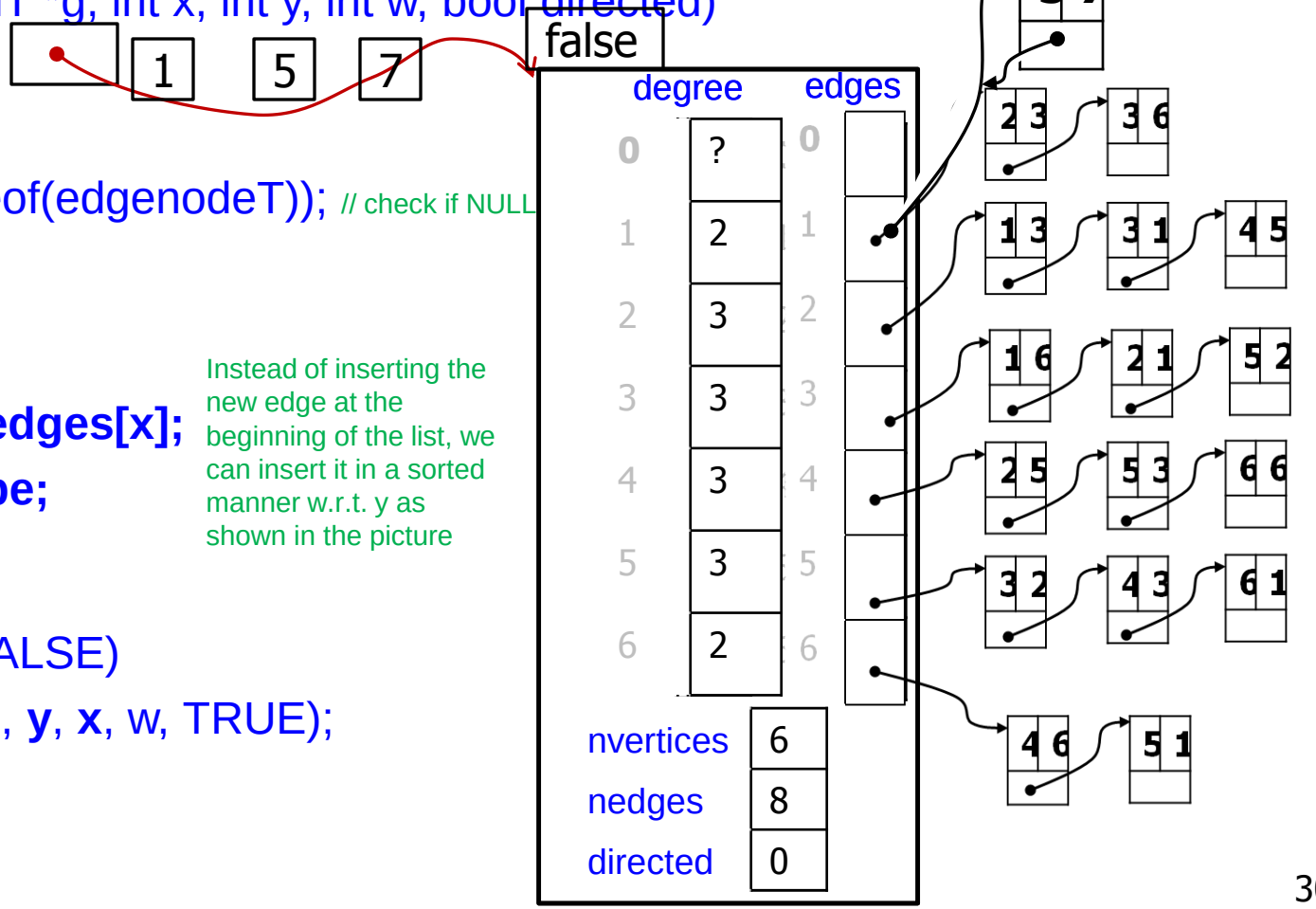
```
typedef struct {
   edgenodeT
      *edges[MAXV+1];
   int degree[MAXV+1];
   int nvertices;
   int nedges;
   bool directed;
} graphT;
```

```
insert_edge(graphT *g, int x, int y, int w, bool directed)
{
   edgenodeT *pe;
   pe = malloc(sizeof(edgenodeT)); // check if NULL
   pe->w = w;
   pe->y = y;

   pe->next = g->edges[x];
   g->edges[x] = pe;

   g->degree[x]++;
   if (directed == FALSE)
      insert_edge(g, y, x, w, TRUE);
}
```

Instead of inserting the new edge at the beginning of the list, we can insert it in a sorted manner w.r.t. y as shown in the picture

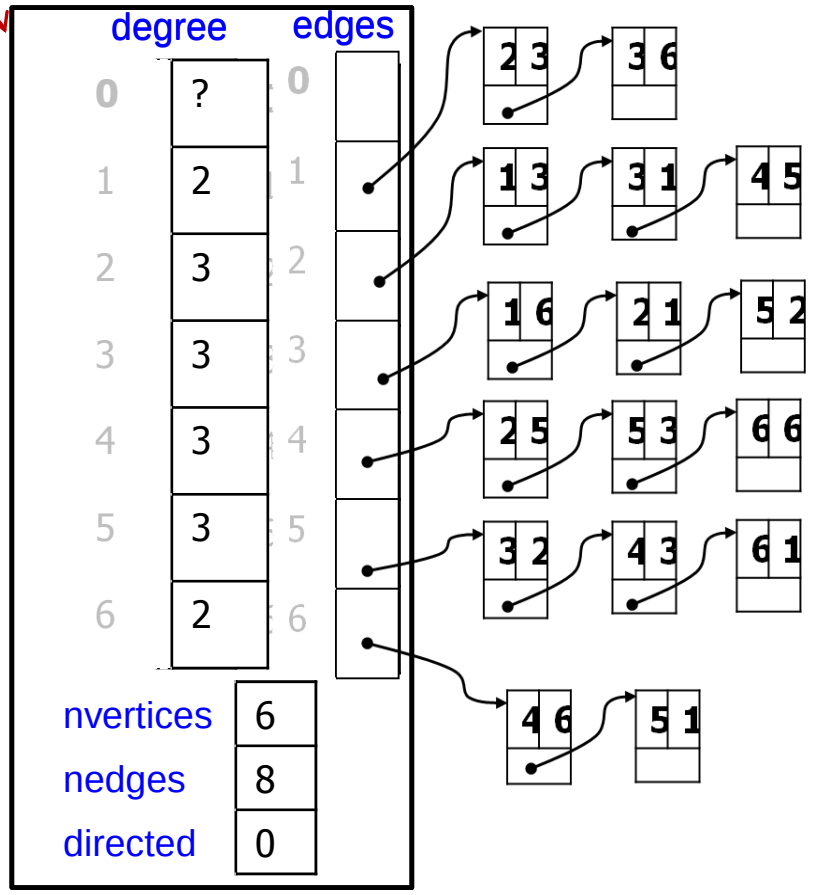


Print Graph

```
typedef struct edgenode {
   int y;
   int w;
   struct edgenode *next;
} edgenodeT;
```

```
typedef struct {
   edgenodeT
      *edges[MAXV+1];
   int degree[MAXV+1];
   int nvertices;
   int nedges;
   bool directed;
} graphT;
```

```
print_graph(graphT *g)
{
   edgenodeT *pe;
   int i;
   for(i=1; i<=g->nvertices; i++) {
      printf("Node %d: ", i);
      pe = g->edges[i];
      while(pe){
         printf(" %d %d,", pe->y, pe->w);
         pe = pe->next;
      }
      printf("\n");
   }
}
```

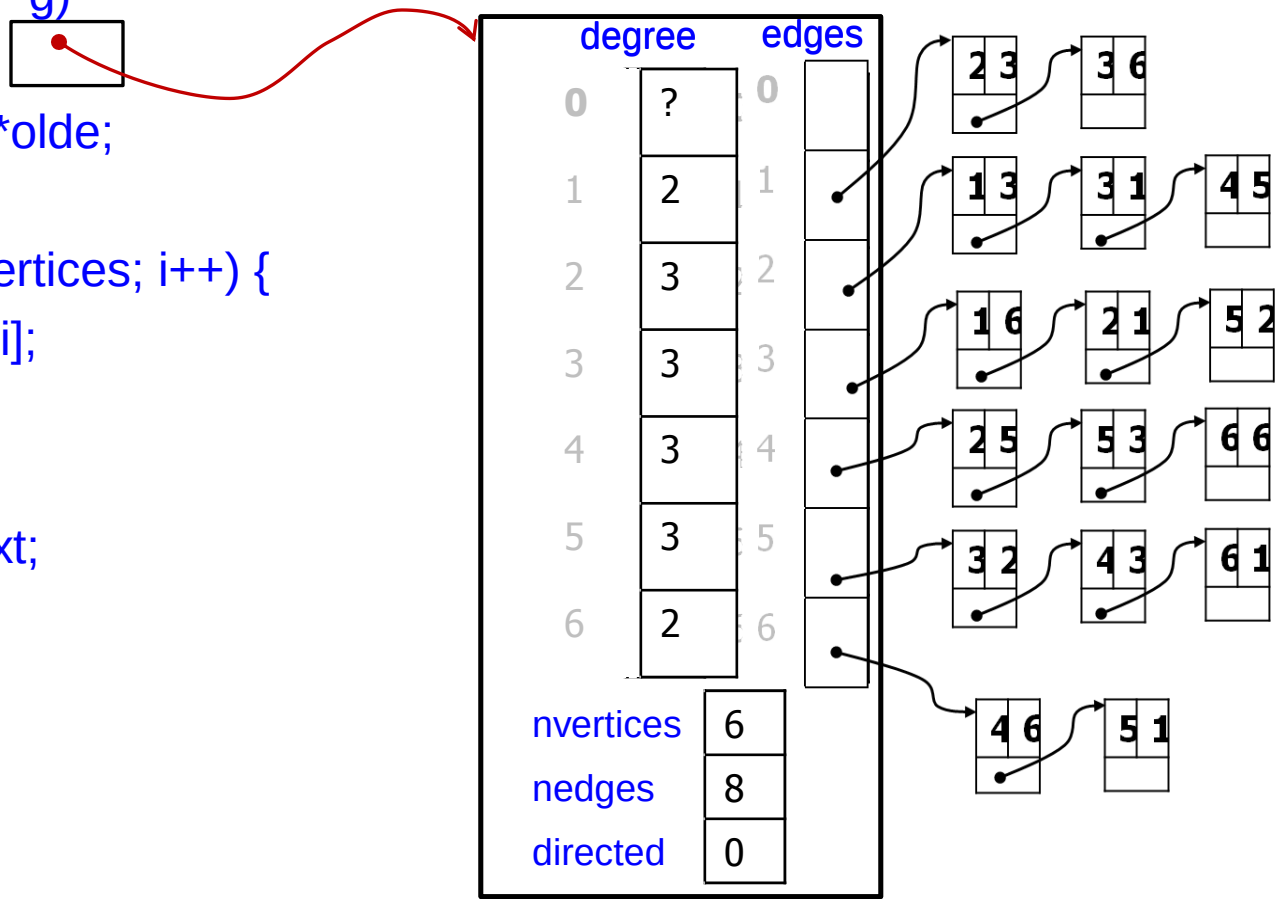


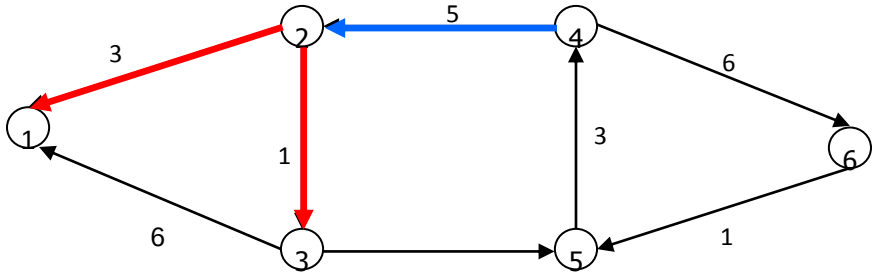
Free Graph

```
typedef struct edgenode {
   int y;
   int w;
   struct edgenode *next;
} edgenodeT;
```

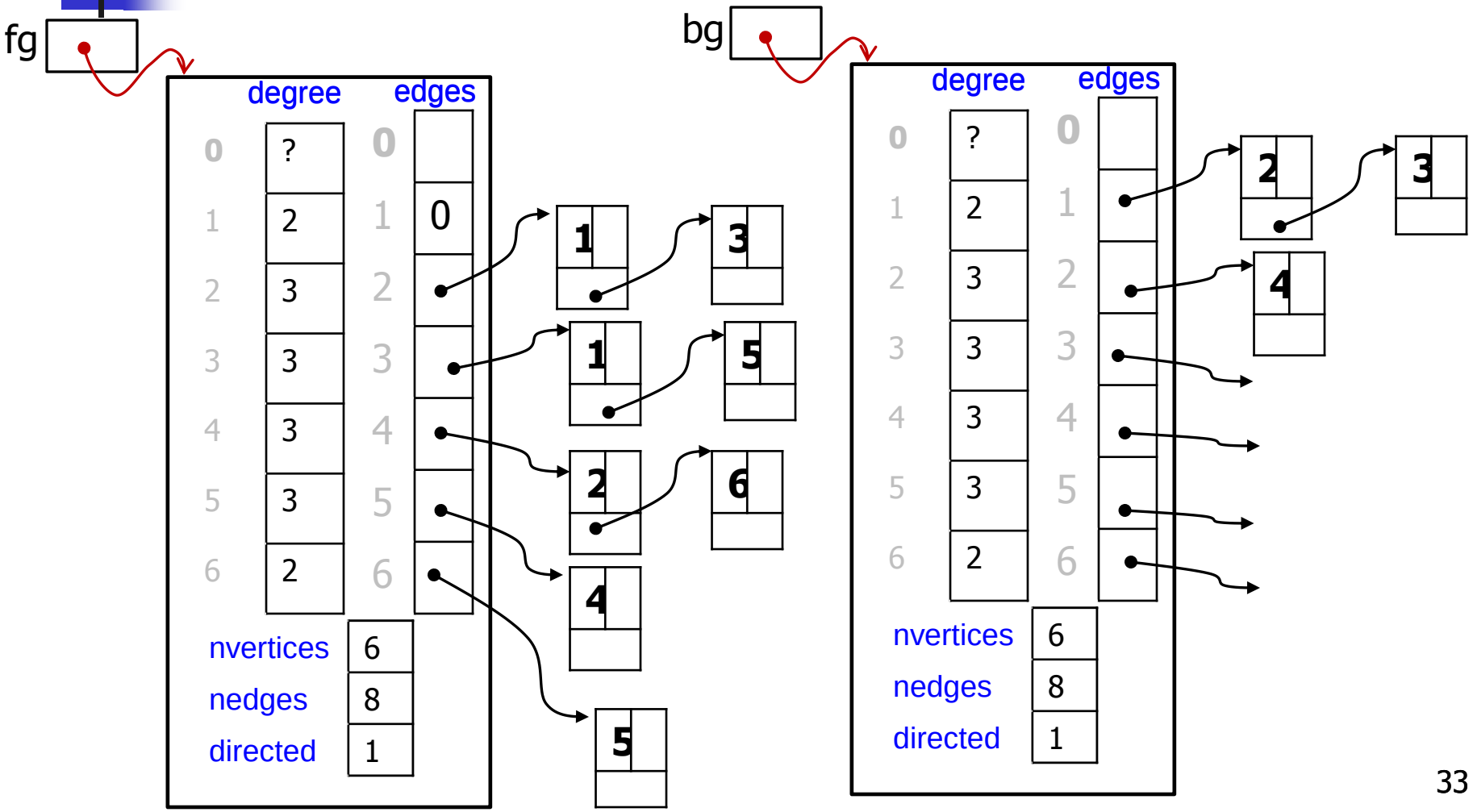
```
typedef struct {
   edgenodeT
      *edges[MAXV+1];
   int degree[MAXV+1];
   int nvertices;
   int nedges;
   bool directed;
} graphT;
```

```
free_graph(graphT *g)
{
   edgenodeT *pe, *olde;
   int i;
   for(i=1; i<=g->nvertices; i++) {
      pe = g->edges[i];
      while(pe){
         olde = pe;
         pe = pe->next;
         free(olde);
      }
   }
   free(g);
}
```





Forward-star
vs.
Backward-star



Exercise: Graph copy

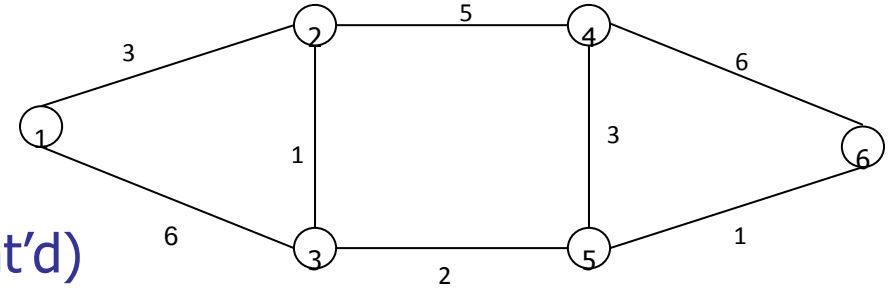
- Write a function that creates a new copy of a given graph g using adjacency list or adjacency matrix

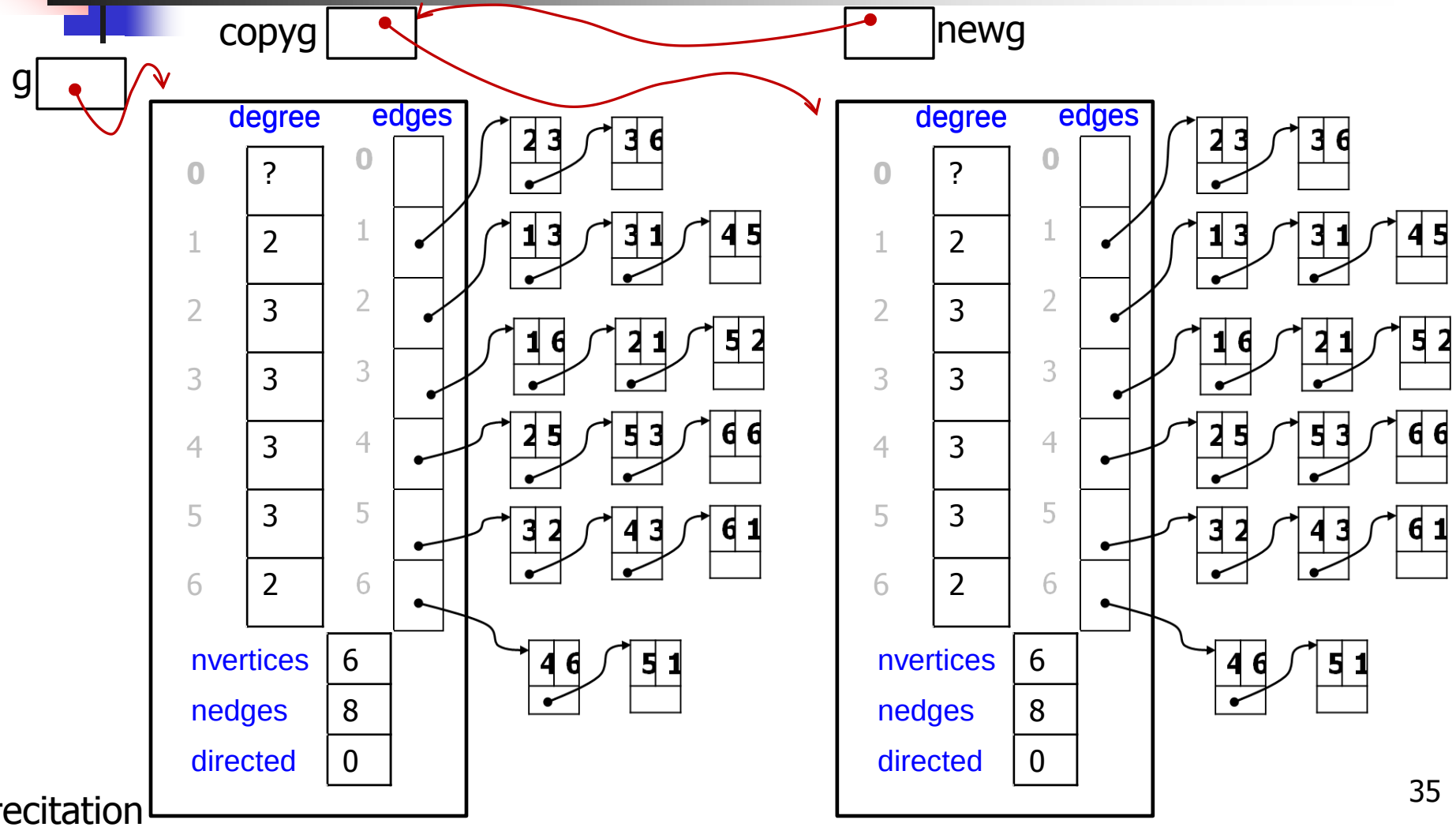
```
void copy_graph_list(graphT *g, graphT **newg)
{
...
}
```

Why ** ?

```
graph *copy_graph_list(graphT *g)
{ ....
}
```

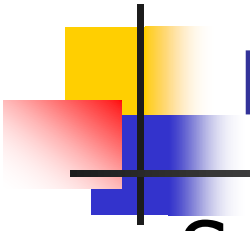
Exercise: Graph copy (cont'd)



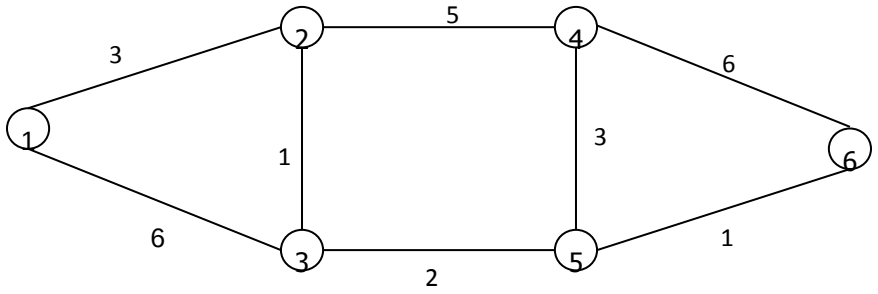


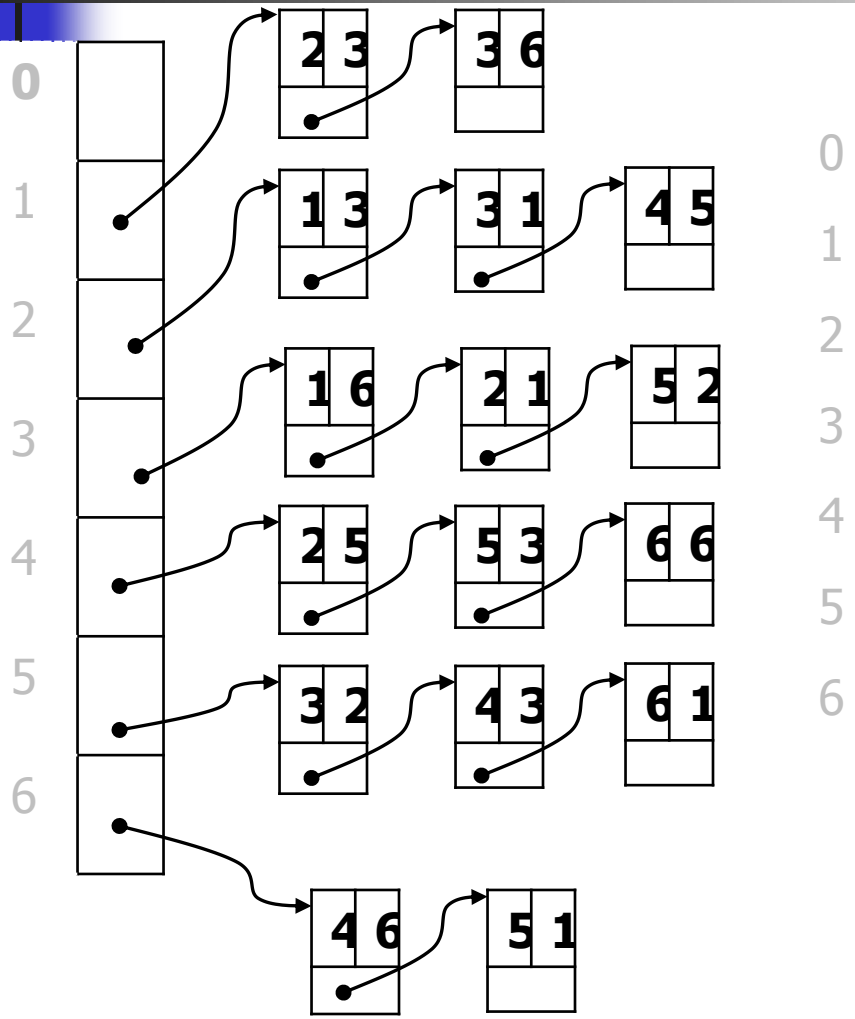
MORE EXERCISES

Exercise: Convert graph representation



- Suppose you are given a graph g, which is represented using
  - adjacency list or
  - adjacency matrix,
- Write a function to convert it to a new graph newg using
  - adjacency matrix or
  - adjacency list, respectively…
- See next slide for an example…





	0	1	2	3	4	5	6
0							
1			1,3	1,6			
2		1,3		1,1	1,5		
3		1,6	1,1			1,2	
4			1,5			1,3	1,6
5				1,2	1,3		1,1
6					1,6	1,1	

Each element M[i][j] might be a structure to represent other attributes of link (i,j)

```
typedef struct { int valid; int w;} linkT;
linkT M[MAXV+1] [MAXV+1];
```

convert_list_to_matrix
Modify print_graph

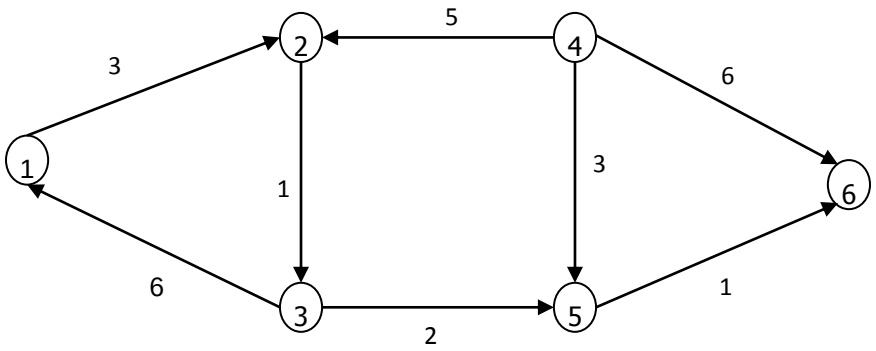
```
print_graph(graphT *g)
{
   edgenodeT *pe;
   int i;
   for(i=1; i<=g->nvertices; i++) {
      printf("Node %d: ", i);
      pe = g->edges[i];
      while(pe){
         printf(" %d", pe->y);
         pe = pe->next;
      }
      printf("\n");
   }
}
```

convert_list_to_matrix(graphT *g, linkT M[] [MAXV+1])

```
for(j=1; j<=g->nvertices; j++)
    M[i][j].valid = 0;
```

```
M[i][pe->y].valid = 1;
M[i][pe->y].w = pe->w;
```

Find in-degree and out-degree for each node



	0	1	2	3	4	5	6
0							
1			1,3				
2				1,1			
3		1,6				1,2	
4			1,5			1,3	1,6
5							1,1
6							