Week 06

Things to Note ...

- Congratulations on finishing your first assignment!

In This Lecture ...

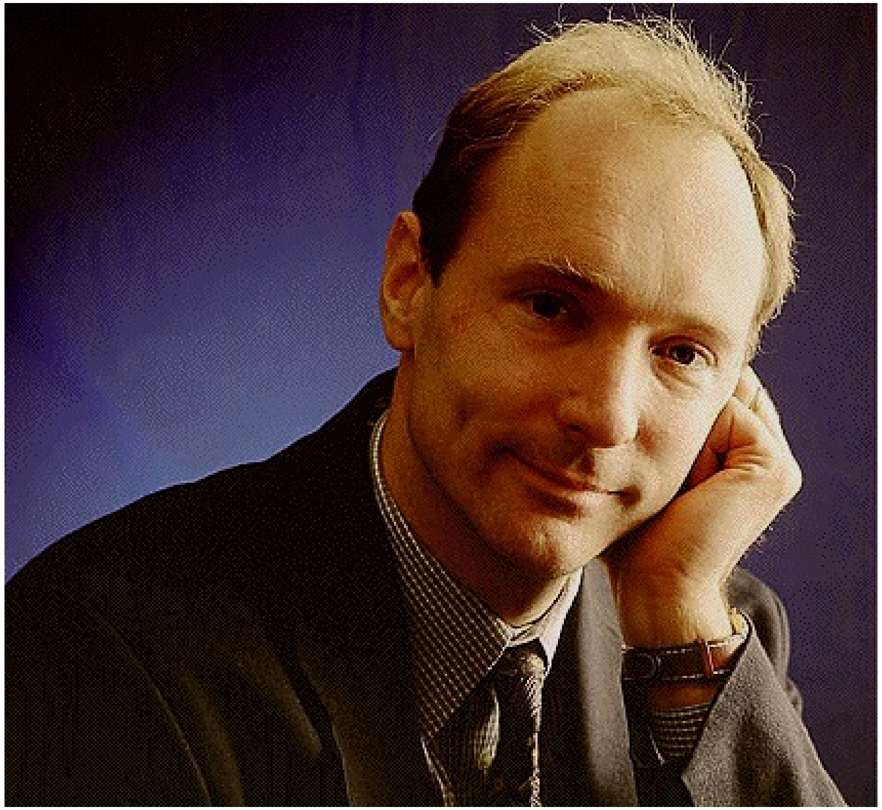
- Graph data structures (Slides, [S] 17.1-17.5)

Coming Up ...

- Graph algorithms (Slides, [S] Ch.18)

Nerds You Should Know

The next in a series on famous computer scientists …



What he invented affects your life every single day …

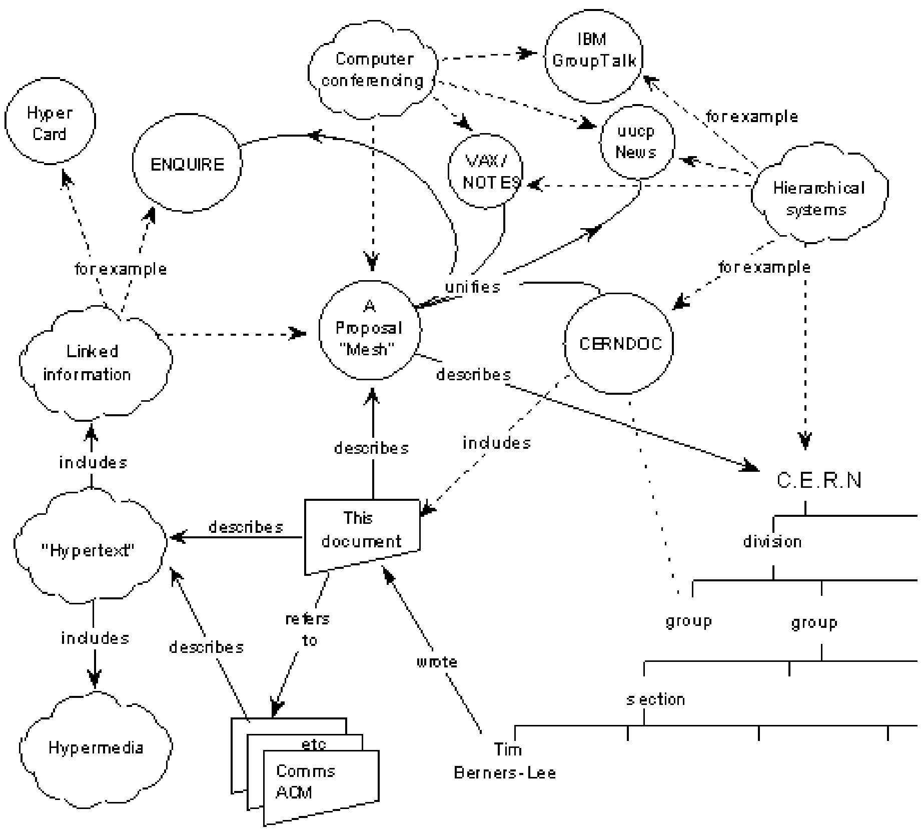
Nerds You Should Know (cont)

Sir Tim Berners-Lee

- Oxford CS Graduate (1976)
- Software engineer at CERN
- Founder/Director of W3C at MIT (1994)
- Inventing the Web ...
  - distributed hypertext
  - linking heterogeneous documents
  - universal naming scheme (URL)
  - transfer protocol (http)
  - later apologised for initial pair of slashes ('`//`') in a web address
  - also thinks he should have defined web addresses the other way round (`au.edu.unsw.cse`)
- Winner of the Turing Award in 2016

Nerds You Should Know (cont)

Tim Berners-Lee's original diagram of the "Web"



(from his proposal document, 1989)

Graph Definitions

Graphs

Many applications require

- a collection of items (i.e. a set)
- relationships/connections between items

Examples:

- maps: items are cities, connections are roads
- web: items are pages, connections are hyperlinks

Collection types you're familiar with

- lists … linear sequence of items (week 3, COMP9021)
- trees … branched hierachy of items (COMP9021)

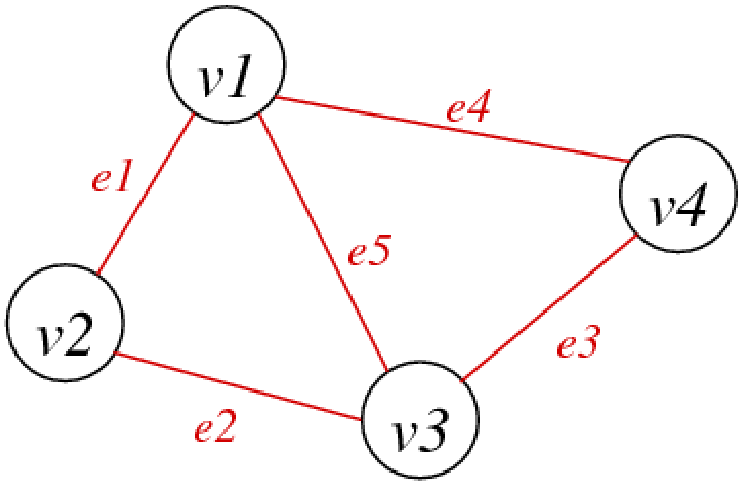
Graphs are more general … allow arbitrary connections

Graphs (cont)

A graph $G = (V,E)$

- V is a set of vertices
- E is a set of edges (subset of $V \times V$)

Example:



$V = \{v1, v2, v3, v4\}$

$E = \{e1, e2, e3, e4, e5\}$

Graphs (cont)

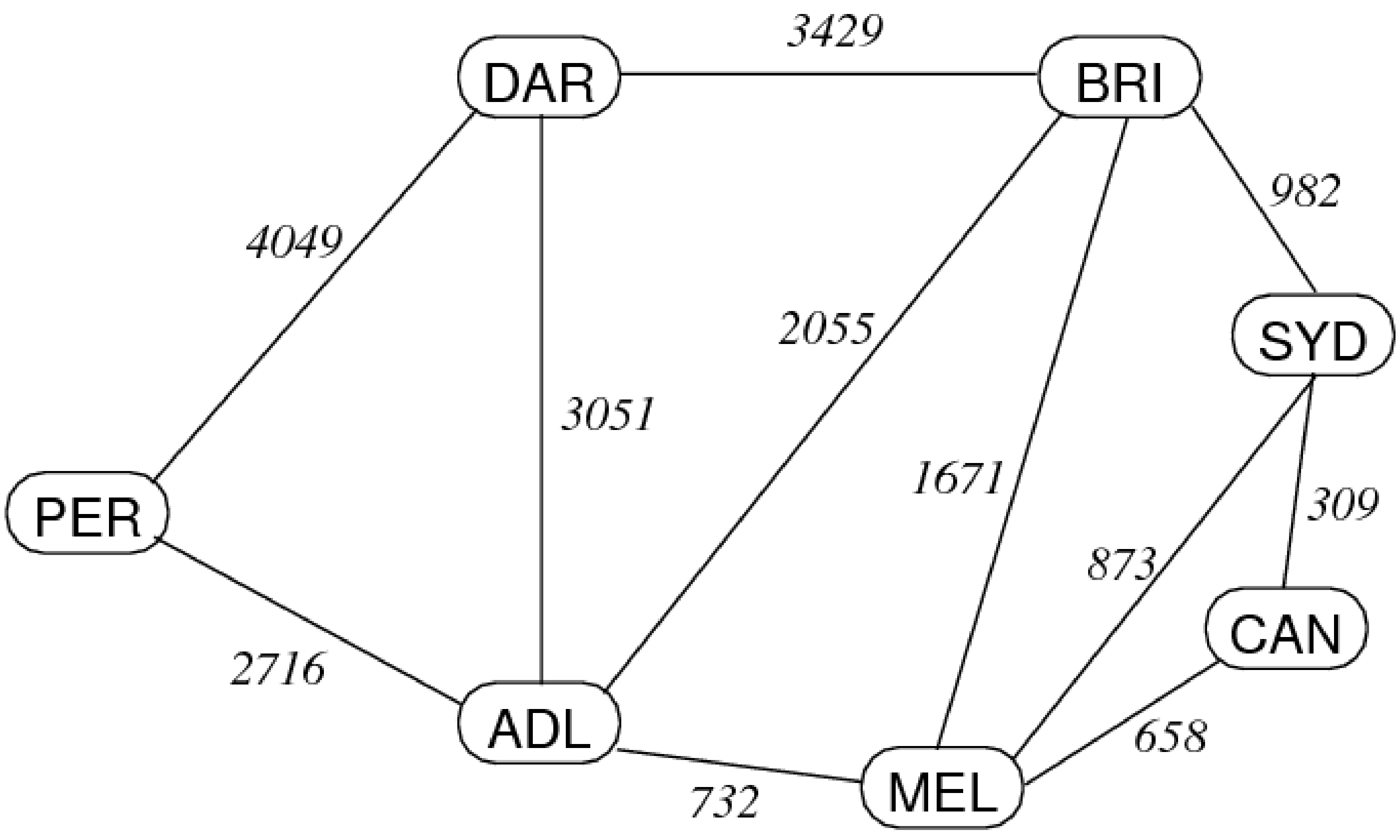
A real example: Australian road distances

Distance	Adelaide	Brisbane	Canberra	Darwin	Melbourne	Perth	Sydney
Adelaide	-	2055	1390	3051	732	2716	1605
Brisbane	2055	-	1291	3429	1671	4771	982
Canberra	1390	1291	-	4441	658	4106	309
Darwin	3051	3429	4441	-	3783	4049	4411
Melbourne	732	1671	658	3783	-	3448	873
Perth	2716	4771	4106	4049	3448	-	3972
Sydney	1605	982	309	4411	873	3972	-

Notes: vertices are cities, edges are distance between cities, symmetric

Graphs (cont)

Alternative representation of above:



Graphs (cont)

Questions we might ask about a graph:

- is there a way to get from item A to item B?
- what is the best way to get from A to B?
- which items are connected?

Graph algorithms are generally more complex than tree/list ones:

- no implicit order of items
- graphs may contain cycles
- concrete representation is less obvious
- algorithm complexity depends on connection complexity

Properties of Graphs

Terminology: $|V|$ and $|E|$ (cardinality) normally written just as V and E.

A graph with V vertices has at most $V(V-1)/2$ edges.

The ratio $E:V$ can vary considerably.

- if E is closer to V^2, the graph is dense
- if E is closer to V, the graph is sparse
    - Example: web pages and hyperlinks

Knowing whether a graph is sparse or dense is important

- may affect choice of data structures to represent graph
- may affect choice of algorithms to process graph

Exercise #1: Number of Edges

The edges in a graph represent pairs of connected vertices. A graph with V has V^2 such pairs.

Consider $V = \{1,2,3,4,5\}$ with all possible pairs:

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

Why do we say that the maximum #edges is $V(V-1)/2$?

… because

- *(v,w)* and *(w,v)* denote the same edge (in an undirected graph)
- we do not consider loops *(v,v)*

Graph Terminology

For an edge *e* that connects vertices *v* and *w*

- *v* and *w* are adjacent (neighbours)
- *e* is incident on both *v* and *w*

Degree of a vertex *v*

- number of edges incident on *e*

Synonyms:

- vertex = node, edge = arc = link (Note: some people use arc for *directed* edges)

Graph Terminology (cont)

Path: a sequence of vertices where

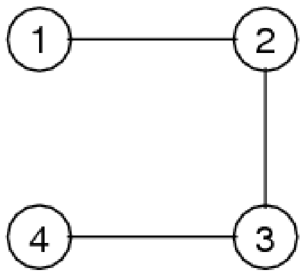
- each vertex has an edge to its predecessor

Cycle: a path where

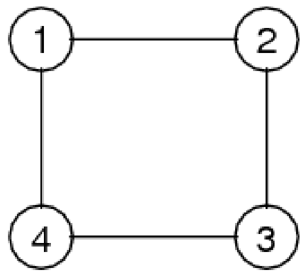
- last vertex in path is same as first vertex in path

Length of path or cycle:

- #edges



Path: 1–2, 2–3, 3–4



Cycle: 1–2, 2–3, 3–4, 4–1

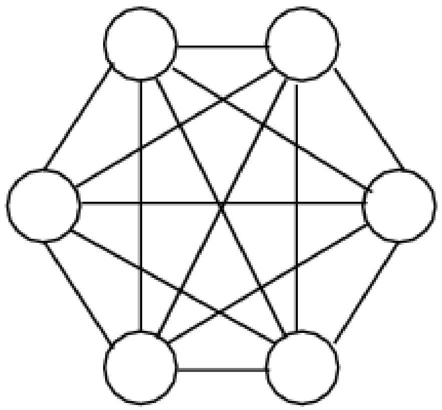
Graph Terminology (cont)

Connected graph

- there is a *path* from each vertex to every other vertex
- if a graph is not connected, it has ≥2 connected components

Complete graph K_V

- there is an *edge* from each vertex to every other vertex
- in a complete graph, $E = V(V-1)/2$



Complete Graph

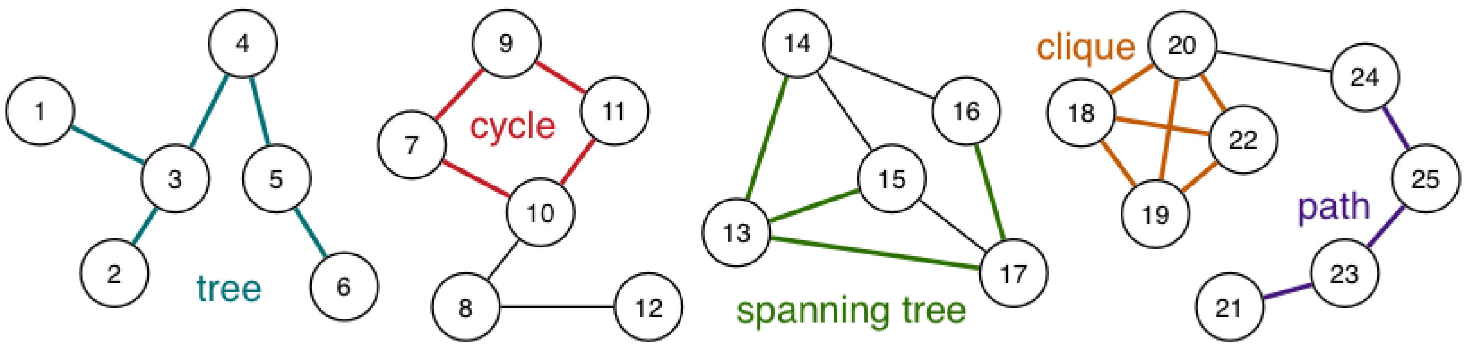
Graph Terminology (cont)

Tree: connected (sub)graph with no cycles

Spanning tree: tree containing all vertices

Clique: complete subgraph

Consider the following single graph:



This graph has 25 vertices, 32 edges, and 4 connected components

Note: The entire graph has no spanning tree; what is shown in green is a spanning tree of the third connected component

Graph Terminology (cont)

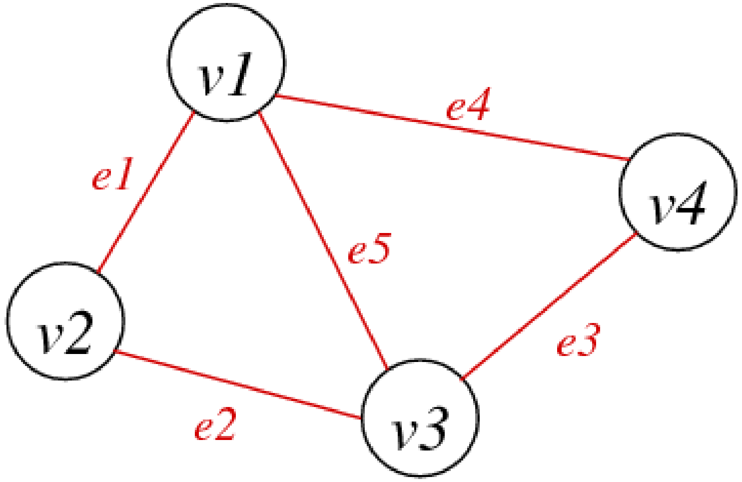
A spanning tree of connected graph $G = (V,E)$

- is a subgraph of G containing all of V
- and is a single tree (connected, no cycles)

A spanning forest of non-connected graph $G = (V,E)$

- is a subgraph of G containing all of V
- and is a set of trees (not connected, no cycles),
  - with one tree for each *connected component*

Exercise #2: Graph Terminology



$V = \{v1, v2, v3, v4\}$

$E = \{e1, e2, e3, e4, e5\}$

1. How many edges to remove to obtain a spanning tree?
2. How many different spanning trees?

1. 2
2. $\frac{5 \cdot 4}{2} - 2 = 8$ spanning trees (no spanning tree if we remove *{e1,e2}* or *{e3,e4}*)

Graph Terminology (cont)

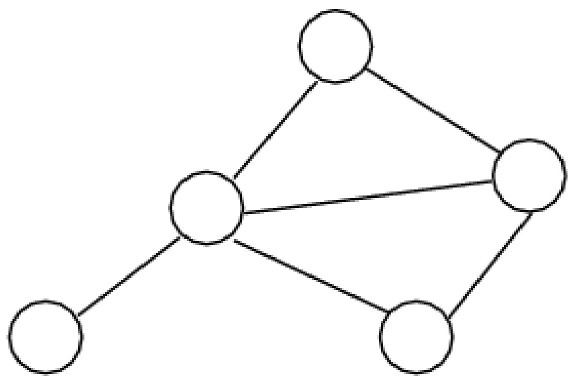
Undirected graph

- *edge(u,v)* = *edge(v,u)*,   no self-loops   (i.e. no *edge(v,v)*)

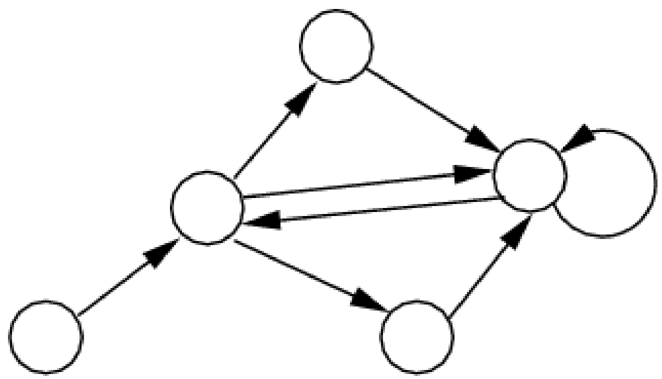
Directed graph

- *edge(u,v)* ≠ *edge(v,u)*,   can have self-loops   (i.e. *edge(v,v)*)

Examples:



Undirected graph



Directed graph

Graph Terminology (cont)

Other types of graphs …

Weighted graph

- each edge has an associated value (weight)
- e.g. road map (weights on edges are distances between cities)

Multi-graph

- allow multiple edges between two vertices
- e.g. function call graph (**`f()`** calls **`g()`** in several places)

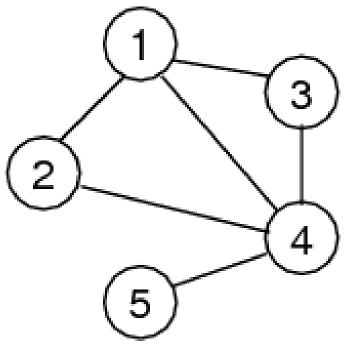
Graph Data Structures

Graph Representations

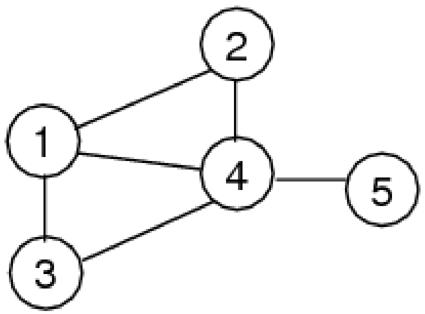
Defining graphs:

- need some way of identifying vertices
- could give diagram showing edges and vertices
- could give a list of edges

E.g. four representations of the same graph:



(a)



(b)

1–2 1–3 1–4
2–4
3–4
4–5

(c)

1–3
2–1 2–4
4–1 4–3
5–4

(d)

Graph Representations (cont)

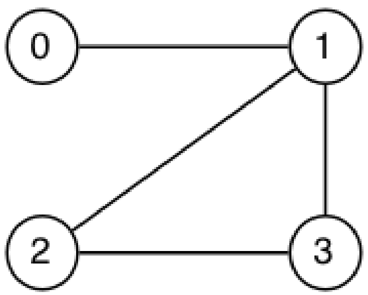
We will discuss three different graph data structures:

1. Array of edges
2. Adjacency matrix
3. Adjacency list

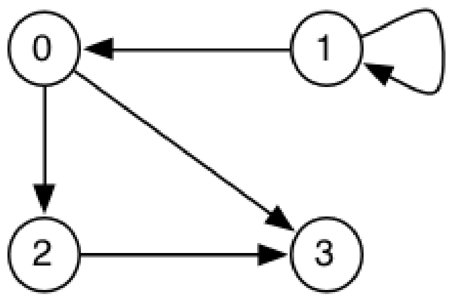
Array-of-edges Representation

Edges are represented as an array of **`Edge`** values (= pairs of vertices)

- space efficient representation
- adding and deleting edges is slightly complex
- undirected: order of vertices in an **`Edge`** doesn't matter
- directed: order of vertices in an **`Edge`** encodes direction



[(0,1), (1,2), (1,3), (2,3)]



[(1,0), (1,1), (0.2), (0,3), (2,3)]

For simplicity, we always assume vertices to be numbered **`0..V-1`**

Array-of-edges Representation (cont)

Graph initialisation

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V    // #vertices (numbered 0..V-1)
|  g.nE = 0    // #edges
|  allocate enough memory for g.edges[]
|  return g
```

Array-of-edges Representation (cont)

Edge insertion

```
insertEdge(g,(v,w)):
|   Input graph g, edge (v,w)
|
|   i=0
|   while i<g.nE ∧ (v,w)≠g.edges[i] do
|       i=i+1
|   end while
|   if i=g.nE then          // (v,w) not found
|       g.edges[i]=(v,w)
|       g.nE=g.nE+1
|   end if
```

Array-of-edges Representation (cont)

Edge removal

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  i=0
|  while i<g.nE ∧ (v,w)≠g.edges[i] do
|     i=i+1
|  end while
|  if i<g.nE then                        // (v,w) found
|     g.edges[i]=g.edges[g.nE-1]  // replace by last edge in array
|     g.nE=g.nE-1
|  end if
```

Cost Analysis

Storage cost: *O(E)*

Cost of operations:

- initialisation: *O(1)*
- insert edge: *O(E)* (assuming edge array has space)
- delete edge: *O(E)* (need to find edge in edge array)

If array is full on insert

- allocate space for a bigger array, copy edges across ⇒ still *O(E)*

If we maintain edges in order

- use binary search to find edge ⇒ *O(log E)*

Exercise #3: Array-of-edges Representation

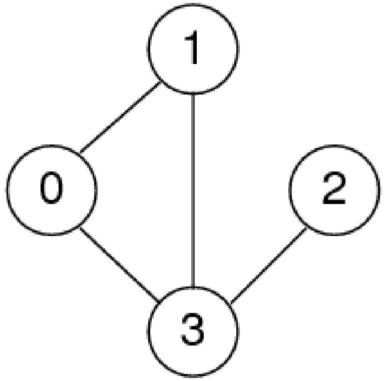
Assuming an array-of-edges representation …

Write an algorithm to output all edges of the graph

```
show(g):
    Input graph g

    for all i=0 to g.nE-1 do
        print g.edges[i]
    end for
```

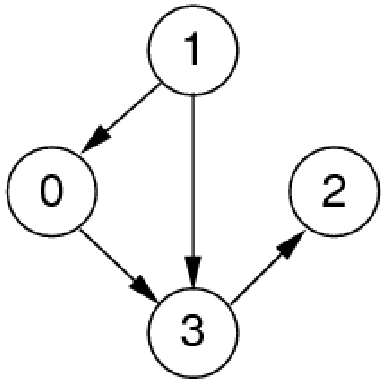
Adjacency Matrix Representation

Edges represented by a $V \times V$ matrix



Undirected graph

A	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0



Directed graph

A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

Adjacency Matrix Representation (cont)

Advantages

- easily implemented as 2-dimensional array
- can represent graphs, digraphs and weighted graphs
  - graphs: symmetric boolean matrix
  - digraphs: non-symmetric boolean matrix
  - weighted: non-symmetric matrix of weight values

Disadvantages:

- if few edges (sparse) ⇒ memory-inefficient

Adjacency Matrix Representation (cont)

Graph initialisation

```
newGraph(V):
|   Input   number of nodes V
|   Output  new empty graph
|
|   g.nV = V    // #vertices (numbered 0..V-1)
|   g.nE = 0    // #edges
|   allocate memory for g.edges[][]
|   for all i,j=0..V-1 do
|      g.edges[i][j]=0    // false
|   end for
|   return g
```

Adjacency Matrix Representation (cont)

Edge insertion

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]=0 then   // (v,w) not in graph
|     g.edges[v][w]=1        // set to true
|     g.edges[w][v]=1
|     g.nE=g.nE+1
|  end if
```

Adjacency Matrix Representation (cont)

Edge removal

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]≠0 then   // (v,w) in graph
|     g.edges[v][w]=0        // set to false
|     g.edges[w][v]=0
|     g.nE=g.nE-1
|  end if
```

Exercise #4: Show Graph

Assuming an adjacency matrix representation …

Write an algorithm to output all edges of the graph (no duplicates!)

Adjacency Matrix Representation (cont)

```
show(g):
|  Input graph g
|
|  for all i=0 to g.nV-1 do
|  |  for all j=i+1 to g.nV-1 do
|  |     if g.edges[i][j]≠0 then
|  |        print i"—"j
|  |     end if
|  |  end for
|  end for
```

Exercise #5:

Analyse storage cost and time complexity of adjacency matrix representation

Storage cost: $O(V^2)$

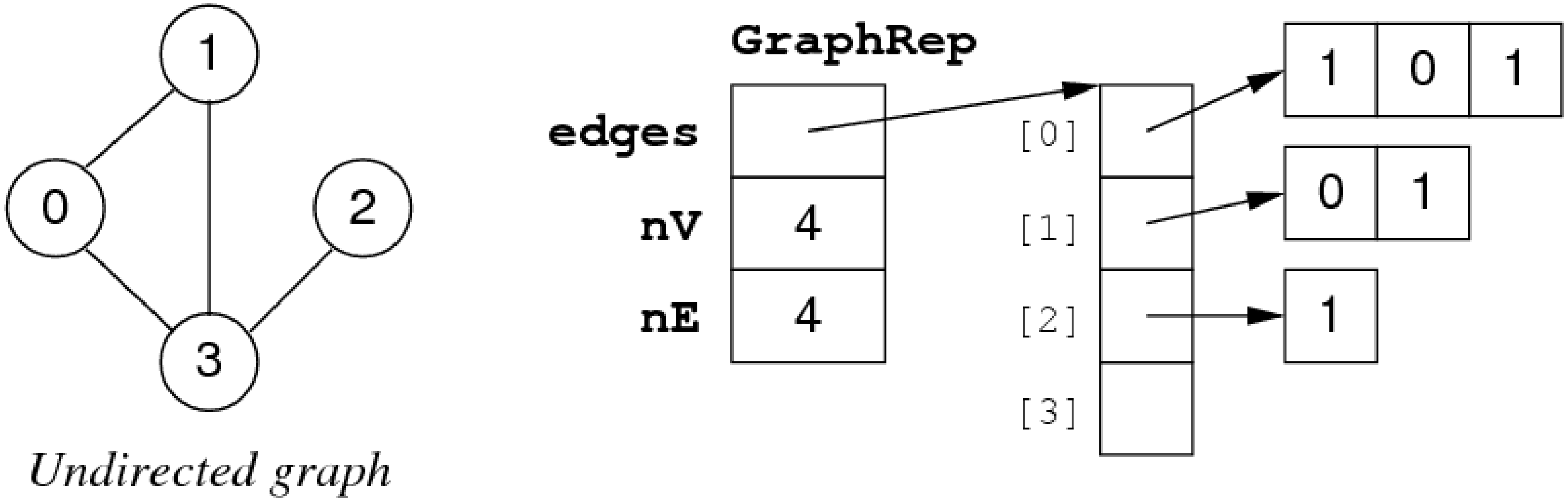
If the graph is sparse, most storage is wasted.

Cost of operations:

- initialisation: $O(V^2)$ (initialise $V \times V$ matrix)
- insert edge: $O(1)$ (set two cells in matrix)
- delete edge: $O(1)$ (unset two cells in matrix)

Adjacency Matrix Representation (cont)

A storage optimisation: store only top-right part of matrix.

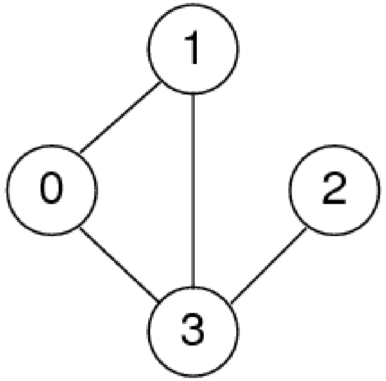


New storage cost: *V-1* int ptrs + *V(V+1)/2* ints (but still $O(V^2)$)

Requires us to always use edges *(v,w)* such that *v* < *w*.

Adjacency List Representation

For each vertex, store linked list of adjacent vertices:



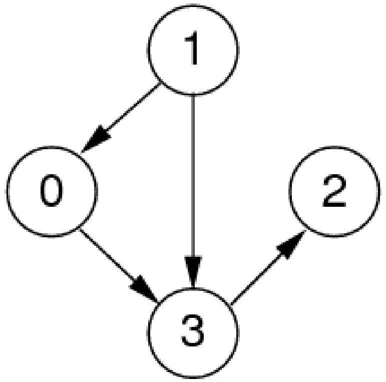
Undirected graph

A[0] = <1, 3>

A[1] = <0, 3>

A[2] = <3>

A[3] = <0, 1, 2>



Directed graph

A[0] = <3>

A[1] = <0, 3>

A[2] = < >

A[3] = <2>

Adjacency List Representation (cont)

Advantages

- relatively easy to implement in languages like C
- can represent graphs and digraphs
- memory efficient if *E:V* relatively small

Disadvantages:

- one graph has many possible representations
  (unless lists are ordered by same criterion e.g. ascending)

Adjacency List Representation (cont)

Graph initialisation

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V    // #vertices (numbered 0..V-1)
|  g.nE = 0    // #edges
|  allocate memory for g.edges[]
|  for all i=0..V-1 do
|     g.edges[i]=NULL    // empty list
|  end for
|  return g
```

Adjacency List Representation (cont)

Edge insertion:

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if ¬inLL(g.edges[v],w) then   // (v,w) not in graph
|     insertLL(g.edges[v],w)
|     insertLL(g.edges[w],v)
|     g.nE=g.nE+1
|  end if
```

Adjacency List Representation (cont)

Edge removal:

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if inLL(g.edges[v],w) then   // (v,w) in graph
|     deleteLL(g.edges[v],w)
|     deleteLL(g.edges[w],v)
|     g.nE=g.nE-1
|  end if
```

Exercise #6:

Analyse storage cost and time complexity of adjacency list representation

Storage cost: *O(V+E)*

Cost of operations:

- initialisation: *O(V)* (initialise *V* lists)
- insert edge: *O(1)* (insert one vertex into list)
- delete edge: *O(E)* (need to find vertex in list)

If vertex lists are sorted

- insert requires search of list ⇒ *O(E)*
- delete always requires a search, regardless of list order

Comparison of Graph Representations

	array of edges	adjacency matrix	adjacency list
space usage	E	V^2	$V+E$
initialise	1	V^2	V
insert edge	E	1	1
remove edge	E	1	E

Other operations:

	array of edges	adjacency matrix	adjacency list
disconnected(v)?	E	V	1
isPath(x,y)?	$E \cdot \log V$	V^2	$V+E$
copy graph	E	V^2	$V+E$
destroy graph	1	V	$V+E$

Graph Abstract Data Type

Graph ADT

Data:

- set of edges, set of vertices

Operations:

- building: create graph, add edge
- deleting: remove edge, drop whole graph
- scanning: check if graph contains a given edge

Things to note:

- set of vertices is fixed when graph initialised
- we treat vertices as **`int`**s, but could be arbitrary **`Item`**s

Graph ADT (cont)

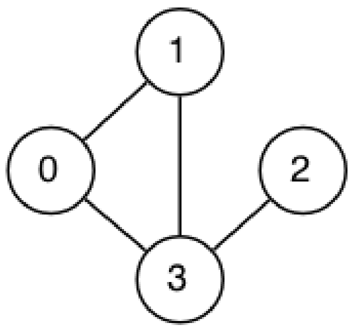
Graph ADT interface **`graph.h`**

```
// graph representation is hidden
typedef struct GraphRep *Graph;

// vertices denoted by integers 0..N-1
typedef int Vertex;

// edges are pairs of vertices (end-points)
typedef struct Edge { Vertex v; Vertex w; } Edge;

// operations on graphs
Graph newGraph(int V);                   // new graph with V vertices
void  insertEdge(Graph, Edge);
void  removeEdge(Graph, Edge);
bool  adjacent(Graph, Vertex, Vertex); /* is there an edge
                                          between two vertices */
void  freeGraph(Graph);
```

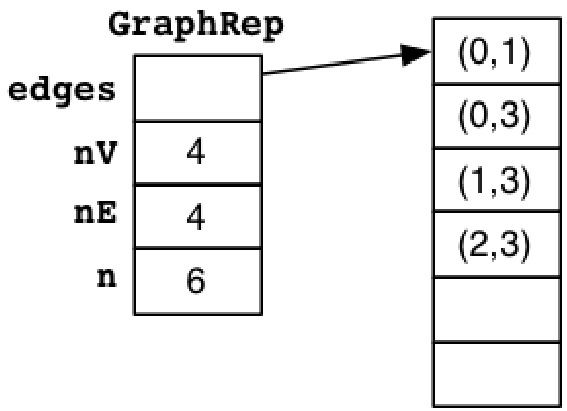
Graph ADT (Array of Edges)

Implementation of **`GraphRep`** (array-of-edges representation)

```
typedef struct GraphRep {
   Edge *edges; // array of edges
   int   nV;     // #vertices (numbered 0..nV-1)
   int   nE;     // #edges
   int   n;      // size of edge array
} GraphRep;
```



Undirected graph



Graph ADT (Array of Edges) (cont)

Implementation of graph initialisation (array-of-edges representation)

```
Graph newGraph(int V) {
   assert(V >= 0);
   Graph g = malloc(sizeof(GraphRep));    assert(g != NULL);

   g->nV = V; g->nE = 0;
   // allocate enough memory for edges
   g->n = Enough;
   g->edges = malloc(g->n*sizeof(Edge));  assert(g->edges != NULL);

   return g;
}
```

How much is enough? … No more than *V(V-1)/2* … Much less in practice (sparse graph)

Graph ADT (Array of Edges) (cont)

Implementation of edge insertion/removal (array-of-edges representation)

```
// check if two edges are equal
bool eq(Edge e1, Edge e2) {
   return ( (e1.v == e2.v && e1.w == e2.w)
            || (e1.v == e2.w && e1.w == e2.v) );
}

void insertEdge(Graph g, Edge e) {
   // ensure that g exists and array of edges isn't full
   assert(g != NULL && g->nE < g->n);
   int i = 0;
   while (i < g->nE && !eq(e,g->edges[i]))
      i++;
   if (i == g->nE)                         // edge e not found
      g->edges[g->nE++] = e;
}

void removeEdge(Graph g, Edge e) {
   assert(g != NULL);                      // ensure that g exists
   int i = 0;
   while (i < g->nE && !eq(e,g->edges[i]))
      i++;
   if (i < g->nE)                          // edge e found
      g->edges[i] = g->edges[--g->nE];
}
```

Exercise #7: Checking Neighbours (i)

Assuming an array-of-edges representation …

Implement a function to check whether two vertices are directly connected by an edge

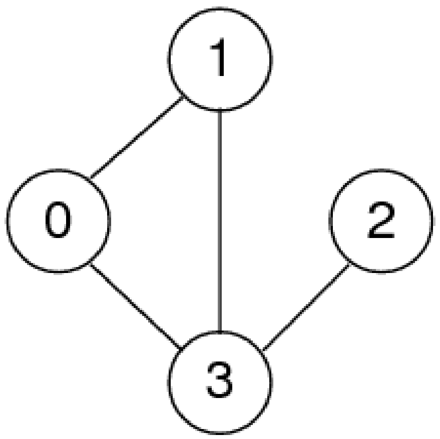
```
bool adjacent(Graph g, Vertex x, Vertex y) { … }
```

```c
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL);
   Edge e;
   e.v = x; e.w = y;
   int i = 0;
   while (i < g->nE) {
      if (eq(e,g->edges[i]))   // edge found
         return true;
      i++;
   }
   return false;               // edge not found
}
```

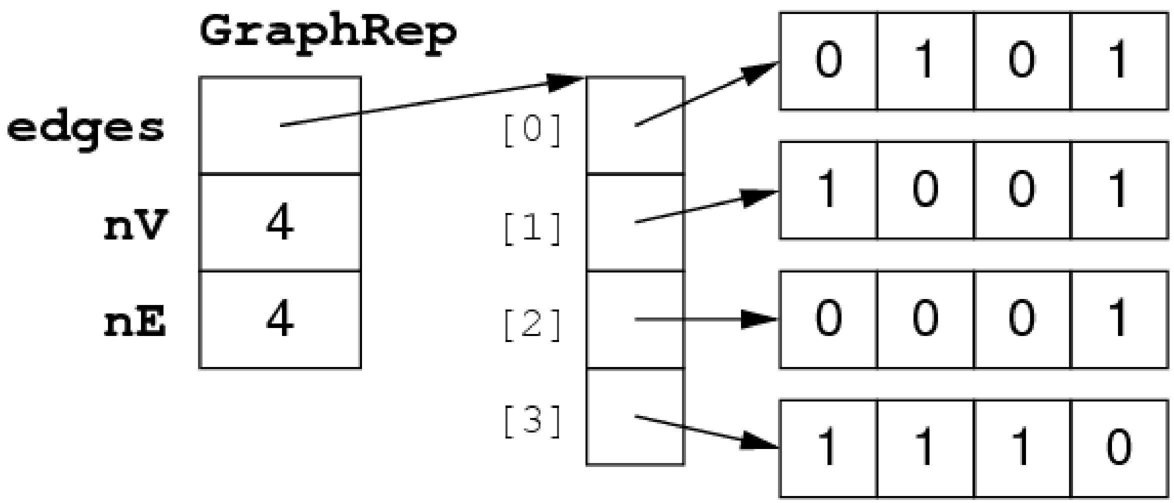
Graph ADT (Adjacency Matrix)

Implementation of **`GraphRep`** (adjacency-matrix representation)

```
typedef struct GraphRep {
   int   **edges; // adjacency matrix
   int     nV;    // #vertices
   int     nE;    // #edges
} GraphRep;
```



Undirected graph



Graph ADT (Adjacency Matrix) (cont)

Implementation of graph initialisation (adjacency-matrix representation)

```
Graph newGraph(int V) {
   assert(V >= 0);
   int i;

   Graph g = malloc(sizeof(GraphRep));       assert(g != NULL);
   g->nV = V;  g->nE = 0;

   // allocate memory for each row
   g->edges = malloc(V * sizeof(int *));     assert(g->edges != NULL);
   // allocate memory for each column and initialise with 0
   for (i = 0; i < V; i++) {
      g->edges[i] = calloc(V, sizeof(int)); assert(g->edges[i] != NULL);
   }
   return g;
}
```

standard library function **`calloc(size_t nelems, size_t nbytes)`**

- allocates a memory block of size **`nelems*nbytes`**
- and sets all bytes in that block to zero

Graph ADT (Adjacency Matrix) (cont)

Implementation of edge insertion/removal (adjacency-matrix representation)

```
// check if vertex is valid in a graph
bool validV(Graph g, Vertex v) {
   return (g != NULL && v >= 0 && v < g->nV);
}

void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));

   if (!g->edges[e.v][e.w]) {  // edge e not in graph
      g->edges[e.v][e.w] = 1;
      g->edges[e.w][e.v] = 1;
      g->nE++;
   }
}

void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));

   if (g->edges[e.v][e.w]) {   // edge e in graph
      g->edges[e.v][e.w] = 0;
      g->edges[e.w][e.v] = 0;
      g->nE--;
   }
}
```

Exercise #8: Checking Neighbours (ii)

Assuming an adjacency-matrix representation …

Implement a function to check whether two vertices are directly connected by an edge

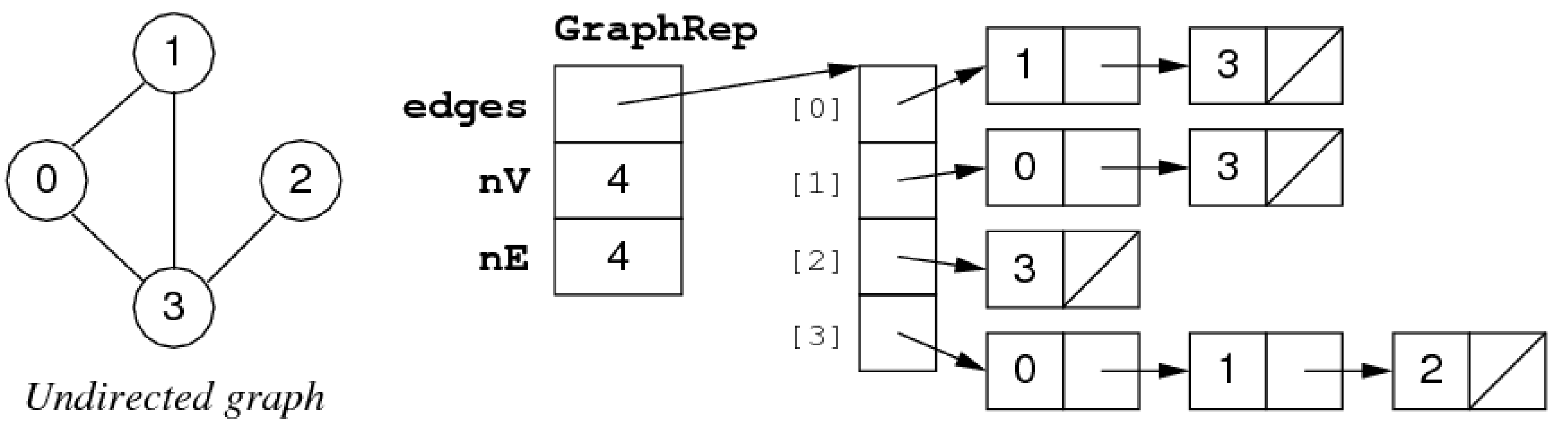
```
bool adjacent(Graph g, Vertex x, Vertex y) { … }
```

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));

   return (g->edges[x][y] != 0);
}
```

Graph ADT (Adjacency List)

Implementation of **GraphRep** (adjacency-list representation)

```
typedef struct GraphRep {
   Node **edges;  // array of lists
   int    nV;     // #vertices
   int    nE;     // #edges
} GraphRep;

typedef struct Node {
   Vertex       v;
   struct Node *next;
} Node;
```



Graph ADT (Adjacency List) (cont)

Implementation of graph initialisation (adjacency-list representation)

```
Graph newGraph(int V) {
   assert(V >= 0);
   int i;

   Graph g = malloc(sizeof(GraphRep));      assert(g != NULL);
   g->nV = V;  g->nE = 0;

   // allocate memory for array of lists
   g->edges = malloc(V * sizeof(Node *));  assert(g->edges != NULL);
   for (i = 0; i < V; i++)
      g->edges[i] = NULL;

   return g;
}
```

Graph ADT (Adjacency List) (cont)

Implementation of edge insertion/removal (adjacency-list representation)

```
void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));

   if (!inLL(g->edges[e.v], e.w)) {     // edge e not in graph
      g->edges[e.v] = insertLL(g->edges[e.v], e.w);
      g->edges[e.w] = insertLL(g->edges[e.w], e.v);
      g->nE++;
   }
}

void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));

   if (inLL(g->edges[e.v], e.w)) {      // edge e in graph
      g->edges[e.v] = deleteLL(g->edges[e.v], e.w);
      g->edges[e.w] = deleteLL(g->edges[e.w], e.v);
      g->nE--;
   }
}
```

`inLL`, **`insertLL`**, **`deleteLL`** are standard linked list operations (as discussed in week 3)

Exercise #9: Checking Neighbours (iii)

Assuming an adjacency list representation …

Implement a function to check whether two vertices are directly connected by an edge

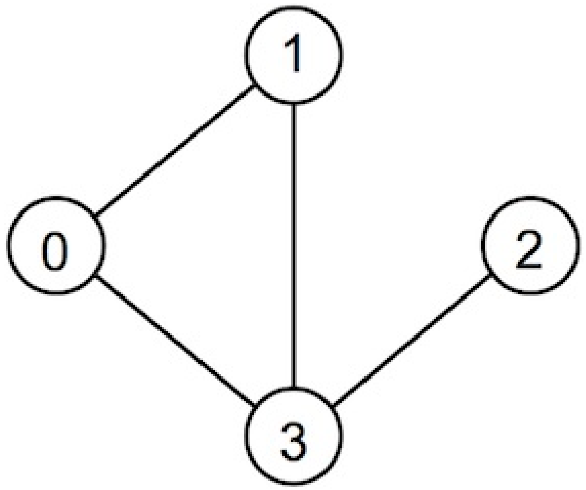
```
bool adjacent(Graph g, Vertex x, Vertex y) { … }
```

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x));

   return inLL(g->edges[x], y);
}
```

Exercise #10: Graph ADT Client

Write a program that uses the graph ADT to

- build the graph depicted below
- print all the nodes that are incident to vertex **`1`** in ascending order



```c
#include <stdio.h>
#include "Graph.h"

#define NODES 4
#define NODE_OF_INTEREST 1

int main(void) {
   Graph g = newGraph(NODES);

   Edge e;
   e.v = 0; e.w = 1; insertEdge(g,e);
   e.v = 0; e.w = 3; insertEdge(g,e);
   e.v = 1; e.w = 3; insertEdge(g,e);
   e.v = 3; e.w = 2; insertEdge(g,e);

   int v;
   for (v = 0; v < NODES; v++) {
      if (adjacent(g, v, NODE_OF_INTEREST))
         printf("%d\n", v);
   }

   freeGraph(g);
   return 0;
}
```

Summary

- Graph terminology
  - vertices, edges, vertex degree, connected graph, tree
  - path, cycle, clique, spanning tree, spanning forest
- Graph representations
  - array of edges
  - adjacency matrix
  - adjacency lists

- Suggested reading:
  - Sedgewick, Ch.17.1-17.5