Graphs

1. Graphs
   - directed and undirected graphs
   - weighted graphs
   - adjacency matrices
2. Graph Representations
   - abstract data type
   - adjacency list
   - adjacency matrix
3. Graph Implementations
   - adjacency matrix
   - adjacency list

MCS 360 Lecture 38
Introduction to Data Structures
Jan Verschelde, 27 November 2017

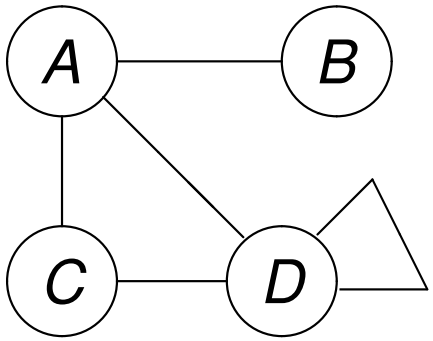
graphs, terminology and definition

- A node in a graph is called a *vertex*.
- A connection between two vertices is an *edge*.

A *graph* G is defined by a tuple (V, E):

- V is the set of vertices, and
- E is the set of edges.

An example:



$$V = \{A, B, C, D\}$$

$$E = \{(A, B), (A, C), (A, D), (C, D), (D, D)\}$$

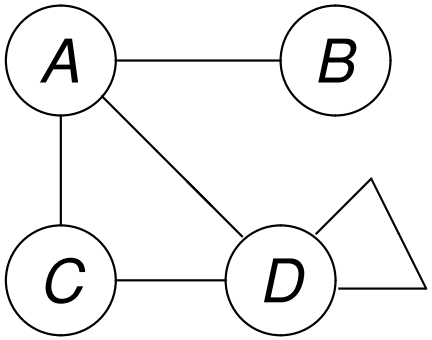
Two vertices are *adjacent* if there is an edge between them.
A *path* is a sequence of successively adjacent vertices.
For example, a path from B to D is B, A, D.

Graphs

1. Graphs
   - directed and undirected graphs
   - weighted graphs
   - adjacency matrices
2. Graph Representations
   - abstract data type
   - adjacency list
   - adjacency matrix
3. Graph Implementations
   - adjacency matrix
   - adjacency list

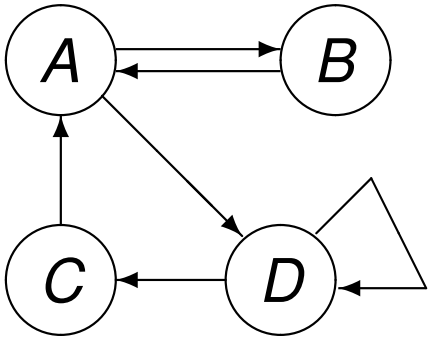
directed and undirected graphs

In an *undirected* graph (which is the default), the edge (A, B) means: there is an edge from A to B and there is an edge from B to A.



$$V = \{A, B, C, D\}$$

$$E = \{(A, B), (A, C), (A, D), (C, D), (D, D)\}$$



$$V = \{A, B, C, D\}$$

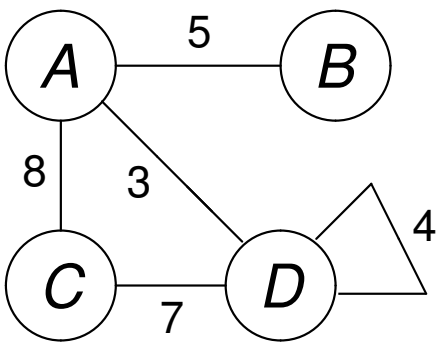
$$E = \{(A, B), (B, A), (C, A), (A, D), (D, C), (D, D)\}$$

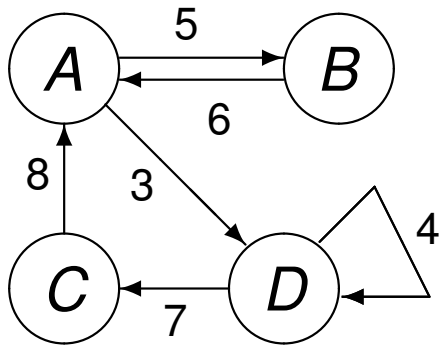
In a *directed* graph or *digraph*, (A, B) means there is an edge from A to B and (B, A) means there is an edge from B to A.

Graphs

weighted graphs

In an *weighted* graph, a value (a weight) is associated to every edge.





Edges in a weighted graphs are triplets (u, v, w),
with w the weight of the edge which connects u to v.

A *cycle* is a path where the first vertex equals the last one.
For example: A, D, C, A is a cycle.

A graph without cycles can be viewed as a tree.

The *degree* of a vertex is the number of edges that contain that vertex.

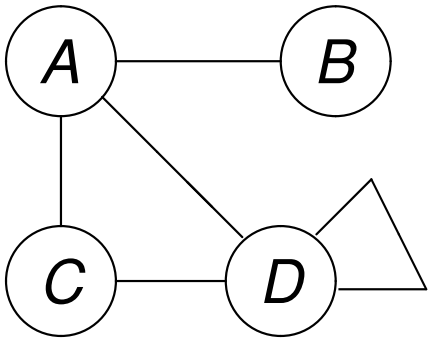
Graphs

1. Graphs
   - directed and undirected graphs
   - weighted graphs
   - adjacency matrices
2. Graph Representations
   - abstract data type
   - adjacency list
   - adjacency matrix
3. Graph Implementations
   - adjacency matrix
   - adjacency list

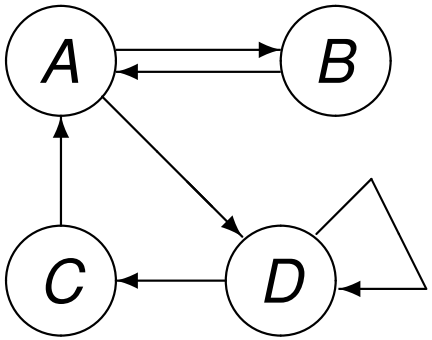
adjacency matrices

For a graph $G = (V, E)$, the *adjacency matrix* M

- has as many rows and columns as the number of vertices in V;
- for all $v_i, v_j \in V$: if $(v_i, v_j) \notin E$: $M_{i,j} = 0$;
- for all $v_i, v_j \in V$: if $(v_i, v_j) \in E$: $M_{i,j} = 1$.



	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	1	0	0	1
D	1	0	1	1

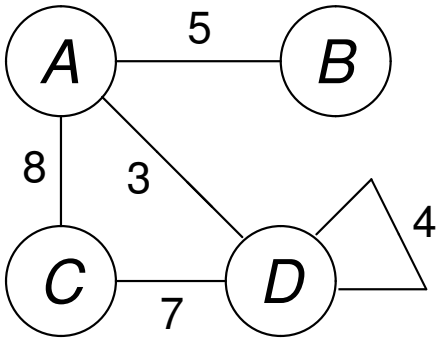


	A	B	C	D
A	0	1	0	1
B	1	0	0	0
C	1	0	0	0
D	0	0	1	1

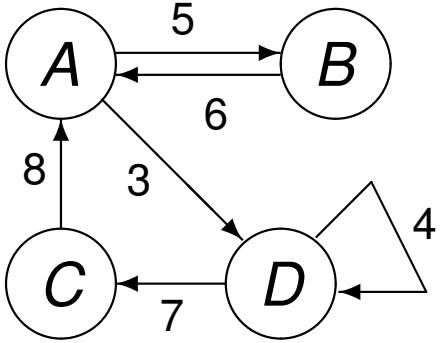
adjacency matrices for weighted graphs

For a weighted graph $G = (V, E)$, the *adjacency matrix* M

- has as many rows and columns as the number of vertices in V;
- for all $v_i, v_j \in V$: if $(v_i, v_j) \notin E$: $M_{i,j} = 0$;
- for all $v_i, v_j \in V$: if $(v_i, v_j) \in E$: $M_{i,j} = w$, w is the weight of (v_i, v_j).



$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 5 & 8 & 3 \\ 5 & 0 & 0 & 0 \\ 8 & 0 & 0 & 7 \\ 3 & 0 & 7 & 4 \end{bmatrix} \end{array}$$



$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 5 & 0 & 3 \\ 6 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 \\ 0 & 0 & 7 & 4 \end{bmatrix} \end{array}$$

Graphs

1. Graphs
   - directed and undirected graphs
   - weighted graphs
   - adjacency matrices
2. Graph Representations
   - abstract data type
   - adjacency list
   - adjacency matrix
3. Graph Implementations
   - adjacency matrix
   - adjacency list

graph representations

A graph $G = (V, E)$ can be one of the four types:

1. undirected: the edge (u, v) goes from u to v and from v to u;
2. directed: the edge (u, v) goes only from u to v, *not* from v to u;
3. weighted undirected: every edge is a triplet (u, v, w), w is the weight and (u, v) goes from u to v and from v to u;
4. weighted directed: every edge is a triplet (u, v, w), w is the weight and (u, v) goes only from u to v, *not* from v to u.

A graph $G = (V, E)$ is a data structure:

- The set of vertices is finite: $V = \{v_0, v_1, \ldots, v_{n-1}\}$.
  The vertices store values, the data are stored in V.
- The set of edges if finite: $E = \{e_0, e_1, \ldots, e_{m-1}\}$.
  In a weighted graph, the edges have weights.

operations on graphs

On a graph $G = (V, E)$, we want the following operations:

- Get the number $|V|$ of vertices in V.
- Get the number $|E|$ of edges in E.
- For an iterator i or index i, get the i-th vertex in V.
- For an iterator i or index i, get the i-th edge in E.
- Given $u, v \in V$, is $(u, v) \in E$?
- Add a vertex to V.
- Given $u, v \in V$, add (u, v) to E.

For a weighted graph, an additional operation is the following:

- Given $u, v \in V$, get the weight of the edge from u to v.

abstract data type

$V = \{v_0, v_1, \ldots, v_{n-1}\}$ is represented by the sequence $0, 1, \ldots, n-1$.

```
abstract <typename T> graph;
/* A graph has vertices 0, 1, .. to store data of
   type T and edges which connect the vertices. */

abstract int number_of_vertices ( graph g );
postcondition: number_of_vertices(g) = |vertices of g|;

abstract int number_of_edges ( graph g );
postcondition: number_of_edges(g) = |edges of g|;

abstract T data_element ( graph g, size_t i );
precondition: 0 <= i < number_of_vertices(g);
postcondition: data_element(g, i) is the data at i;
```

abstract data type continued

```
abstract (i, j) get_edge ( graph g, size_t k );
precondition: 0 <= k < number_of_edges(g);
postcondition: get_edge(g, k) is the k-th edge (i, j);

abstract bool is_edge ( graph g, size_t i, size_t j );
postcondition: is_edge(g, i, j) is true if (i, j) is
  an edge of g, otherwise is_edge(g, i, j) is false;

abstract void add_vertex ( graph g );
postcondition: for the graph g on input, there is
  a vertex with index == number_of_vertices(g);

abstract void add_edge ( graph g, size_t i, size_t j );
precondition: 0 <= i, j < number_of_vertices(g);
postcondition: is_edge(g, i, j) is true;
```

abstract data type for weighted graph

The ADT for a weighted graph starts differently:

```
abstract <typename T> weighted graph;
/* A weighted graph has vertices 0, 1, .. to store
   data of type T and edges which connect vertices.
   Every edge has a weight of type double. */
```

and there is an additional operation:

```
abstract double get_weight
  ( graph g, size t i, size_t j );
precondition: 0 <= i, j < number_of_vertices(g)
  and is_edge(g, i, j) is true;
postcondition: get_weight(g, i, j) is the weight
  of the edge (i, j);
```

A more general ADT for a weighted graph would leave the type of the weight as defined by a template.

Graphs

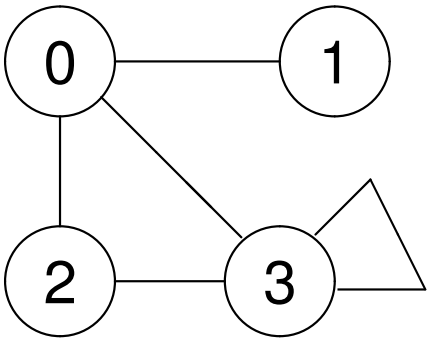
1. Graphs
   - directed and undirected graphs
   - weighted graphs
   - adjacency matrices
2. Graph Representations
   - abstract data type
   - adjacency list
   - adjacency matrix
3. Graph Implementations
   - adjacency matrix
   - adjacency list

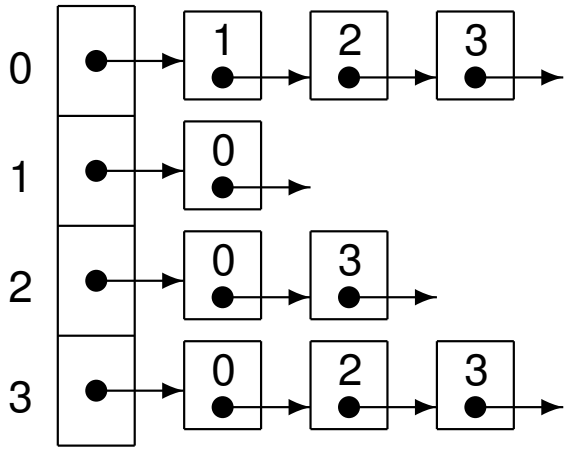
adjacency list representation

An adjacency list representation

- uses an array of lists, one list per vertex;
- the list *i* stores the vertices adjacent to vertex *i*.

An example:





The vertices in the list need not to be ordered.

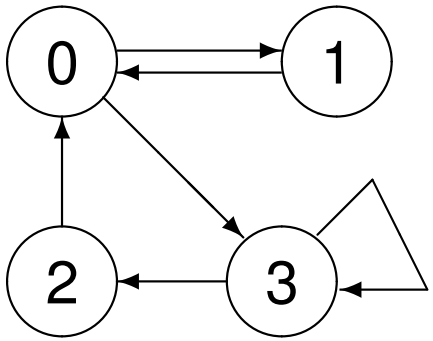
The number of elements in list *i* equals the degree of vertex *i*.

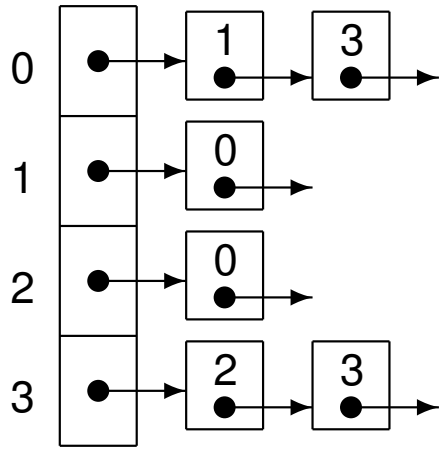
adjacency list representation for a directed graph

An adjacency list representation

- uses an array of lists, one list per vertex;
- the list i stores the vertices j for all edges (i, j).

An example:





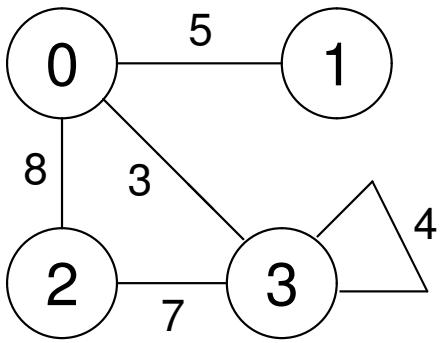
The vertices in the list need not to be ordered.

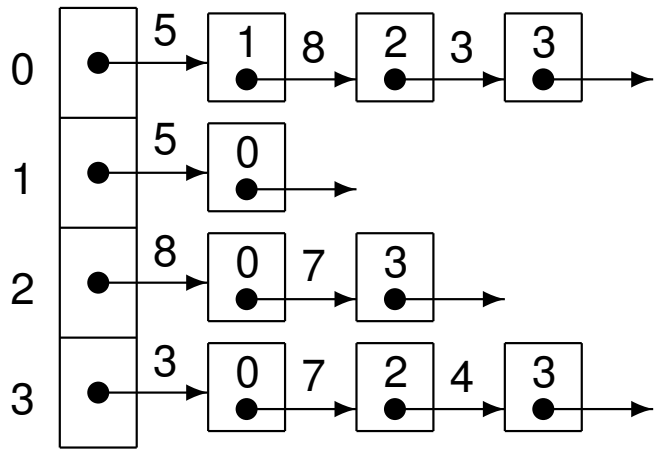
adjacency list representation for a weighted graph

An adjacency list representation

- uses an array of lists, one list per vertex;
- the list *i* stores the vertices adjacent to vertex *i*;
- the weight of an edge is stored with the pointer to the next vertex.

An example:





The vertices in the list need not to be ordered.
Exercise: draw the adjacency list for a weighted digraph.

Graphs

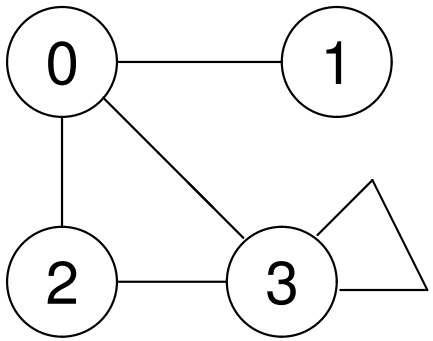
1. Graphs
   - directed and undirected graphs
   - weighted graphs
   - adjacency matrices
2. Graph Representations
   - abstract data type
   - adjacency list
   - adjacency matrix
3. Graph Implementations
   - adjacency matrix
   - adjacency list

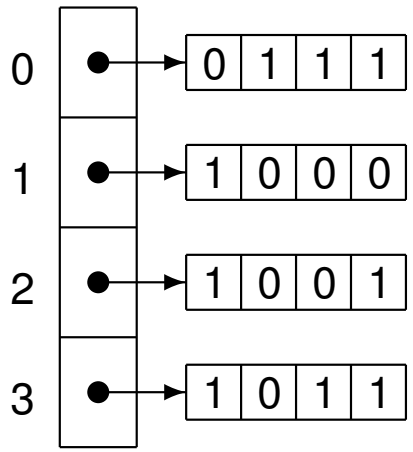
adjacency matrix representation

An adjacency matrix representation

- uses an array of arrays, one array per vertex;
- the *i*-th array stores a boolean at its *j*-th position:
  - the *j*-th boolean in array *i* is false if no edge from *i* to *j*;
  - the *j*-th boolean in array *i* is true if an edge from *i* to *j*.

An example:





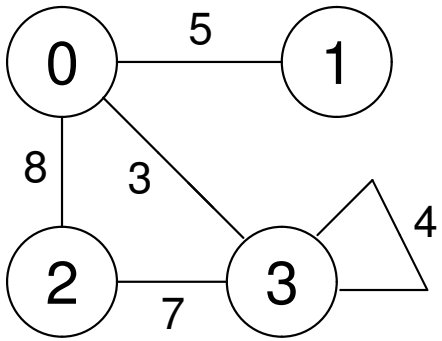
Exercise: draw the adjacency matrix for a digraph.

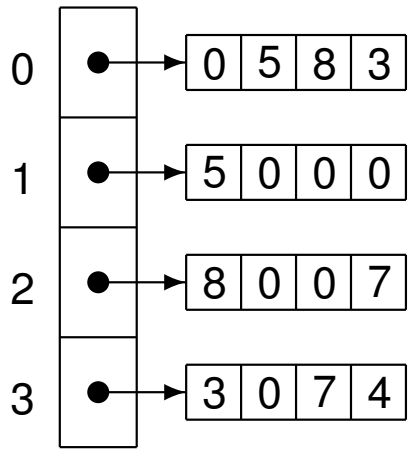
adjacency matrix representation for a weighted graph

An adjacency matrix representation

- uses an array of arrays, one array per vertex;
- the *i*-th array stores the weight at its *j*-th position:
  - the *j*-th number in array *i* is zero if no edge from *i* to *j*;
  - the *j*-th number in array *i* is the weight if an edge from *i* to *j*.

An example:





Exercise: draw the adjacency matrix for a weighted digraph.

Graphs

1. Graphs
   - directed and undirected graphs
   - weighted graphs
   - adjacency matrices
2. Graph Representations
   - abstract data type
   - adjacency list
   - adjacency matrix
3. Graph Implementations
   - adjacency matrix
   - adjacency list

graph as adjacency matrix

For the array of arrays, we use the STL `vector` class.

An adjacency matrix is of type `vector< vector<bool> >`.

For a weighted graph, we use `vector< vector<double> >`.

We have four types:

1. undirected, unweighted
2. directed, unweighted
3. undirected, weighted
4. directed, weighted

We generate random matrices to test the data structures.

prototypes for directed, unweighted graphs

```
void allocate_boolean_matrix
  ( vector< vector<bool> >& mat, size_t dim );
// Allocates space for an adjacency matrix
// of dimension dim.

void random_directed_graph
  ( vector< vector<bool> >& mat );
// Fills up the adjacency matrix for a random
// directed graph.

void write_boolean_matrix
  ( vector< vector<bool> >& mat );
// Writes the matrix to screen.
```

allocation of the matrix

```
void allocate_boolean_matrix
  ( vector< vector<bool> >& mat, size_t dim )
{
   for(size_t i=0; i<dim; i++)
   {
      vector<bool> row;

      for(size_t j=0; j<dim; j++)
         row.push_back(0);

      mat.push_back(row);
   }
}
```

generating a random graph

```
void random_directed_graph
  ( vector< vector<bool> >& mat )
{
   const size_t dim = mat.size();

   for(size_t i=0; i<dim; i++)
      for(size_t j=0; j<dim; j++)
         mat[i][j] = (rand() % 2);
}
```

writing the matrix

```
void write_boolean_matrix
  ( vector< vector<bool> >& mat )
{
   const size_t dim = mat.size();

   for(size_t i=0; i<dim; i++)
   {
      for(size_t j=0; j<dim; j++)
         cout << " " << mat[i][j];
      cout << endl;
   }
}
```

a random symmetric matrix

In a symmetrix matrix, the lower half equals the upper half.

```
void random_undirected_graph
  ( vector< vector<bool> >& mat )
{
   const size_t dim = mat.size();

   for(size_t i=0; i<dim; i++)
   {
      for(size_t j=0; j<i; j++) // copy lower half
         mat[i][j] = mat[j][i];

      for(size_t j=i; j<dim; j++) // generate upper
         mat[i][j] = (rand() % 2);
   }
}
```

weights in the interval $[0.1, 9.9]$.

```
void random_directed_weighted_graph
  ( vector< vector<double> >& mat )
{
   const size_t dim = mat.size();
   for(size_t i=0; i<dim; i++)
      for(size_t j=0; j<dim; j++)
      {
         size_t rnd = rand() % 2;  // edge or not
         if(rnd == 0)
            mat[i][j] = 0.0; // no edge => 0 weight
         else
         {
            rnd = 1 + rand() % 99; // random weight
            mat[i][j] = ((double) rnd)/10.0;
         }
      }
}
```

Graphs

1. Graphs
   - directed and undirected graphs
   - weighted graphs
   - adjacency matrices
2. Graph Representations
   - abstract data type
   - adjacency list
   - adjacency matrix
3. Graph Implementations
   - adjacency matrix
   - adjacency list

graph as adjacency list

We use the STL `vector` and `list` classes.

The array of lists is then of type `vector< list<size_t> >`.

For a weighted graph,
we use `vector< list< pair<size_t, double> > >`.

Memory allocation happens in two stages:

1. The vector is allocated immediately after its declaration.
2. Space in the list is allocated for each new edge.

prototypes for directed, unweighted graphs

```
void allocate_adjacency_list
 ( vector< list<size_t> >& lst, size_t dim );
// Allocates space for an adjacency list
// of dimension dim.

void random_directed_graph
 ( vector< list<size_t> >& lst );
// Fills up the adjacency list for a random
// directed graph.

void write_adjacency_list
 ( vector< list<size_t> >& lst );
// Writes the adjacency list to screen.
```

allocating space

```
void allocate_adjacency_list
  ( vector< list<size_t> >& lst, size_t dim )
{
   for(size_t i=0; i<dim; i++)
   {
      list<size_t> row;
      lst.push_back(row);
   }
}
```

generating a directed, unweighted graph

```
void random_directed_graph
 ( vector< list<size_t> >& lst )
{
   const size_t dim = lst.size();

   for(size_t i=0; i<dim; i++)
   {
      for(size_t j=0; j<dim; j++)
      {
         size_t rnd = rand() % 2;
         if(rnd != 0) lst[i].push_back(j);
      }
   }
}
```

writing the adjacency list

```
void write_adjacency_list
  ( vector< list<size_t> >& lst )
{
   const size_t dim = lst.size();

   for(size_t i=0; i<dim; i++)
   {
      cout << i;

      for(list<size_t>::const_iterator it=lst[i].begin();
          it != lst[i].end(); it++)
         cout << " -> " << *it;

      cout << endl;
   }
}
```

summary + exercises

Graphs representations are adjacency lists or adjacency matrices.

Exercises:

1. For the Graph ADT, write a class definition where the private data attribute is an adjacency matrix.
2. Give the implementation of the class of the previous exercise.
3. For the Graph ADT, write a class definition where the private data attribute is an adjacency list.
4. Give the implementation of the class of the previous exercise.