Hashing with Separate Chaining Implementation – data members

```
public class SCHashTable<T extends KeyedItem>
implements HashTableInterface<T>
{
  private List<T>[] table;

  private int h(long key) // hash function
  // return index
  {
    return (int)(key % table.length); // typecast to int
  }

  public SCHashTable(int size)
  // recommended size: prime number roughly twice bigger
  //                   than the expected number of elements
  {
    table = new List[size];
    // initialize the lists
    for (int i=0; i<size; i++)
      table[i] = new List<T>();
  }
```

Implementation – insertion

```
public void insert(T item)
 {
   int index = h(item.getKey());
   List<T> L = table[index];
   // insert item to L
   L.add(1,item); // if linked list is used,
                  // insertion will be efficient
 }
```

Implementation – search

```
private int findIndex(List<T> L, long key)
 // search for item with key 'key' in L
 // return -1 if the item with key 'key' was not found in L
 {
   // search of item with key = 'key'
   for (int i=1; i<=L.size(); i++)
     if (L.get(i).getKey() ==  key)
       return i;

   return -1; // not found
 }

 public T find(long key)
 {
   int index = h(key);
   List<T> L = table[index];
   int list_index = findIndex(L,key);
   if (index>=0)
     return L.get(list_index);
   else
     return null; // not found
 }
```

Implementation – deletion

```
public T delete(long key)
 {
   int index = h(key);
   List<T> L = table[index];
   int list_index = findIndex(L,key);
   if (index>=0) {
     T item = L.get(list_index);
     L.remove(list_index);
     return item;
   } else
     return null; // not found
 }
```

Hashing – comparison of different methods

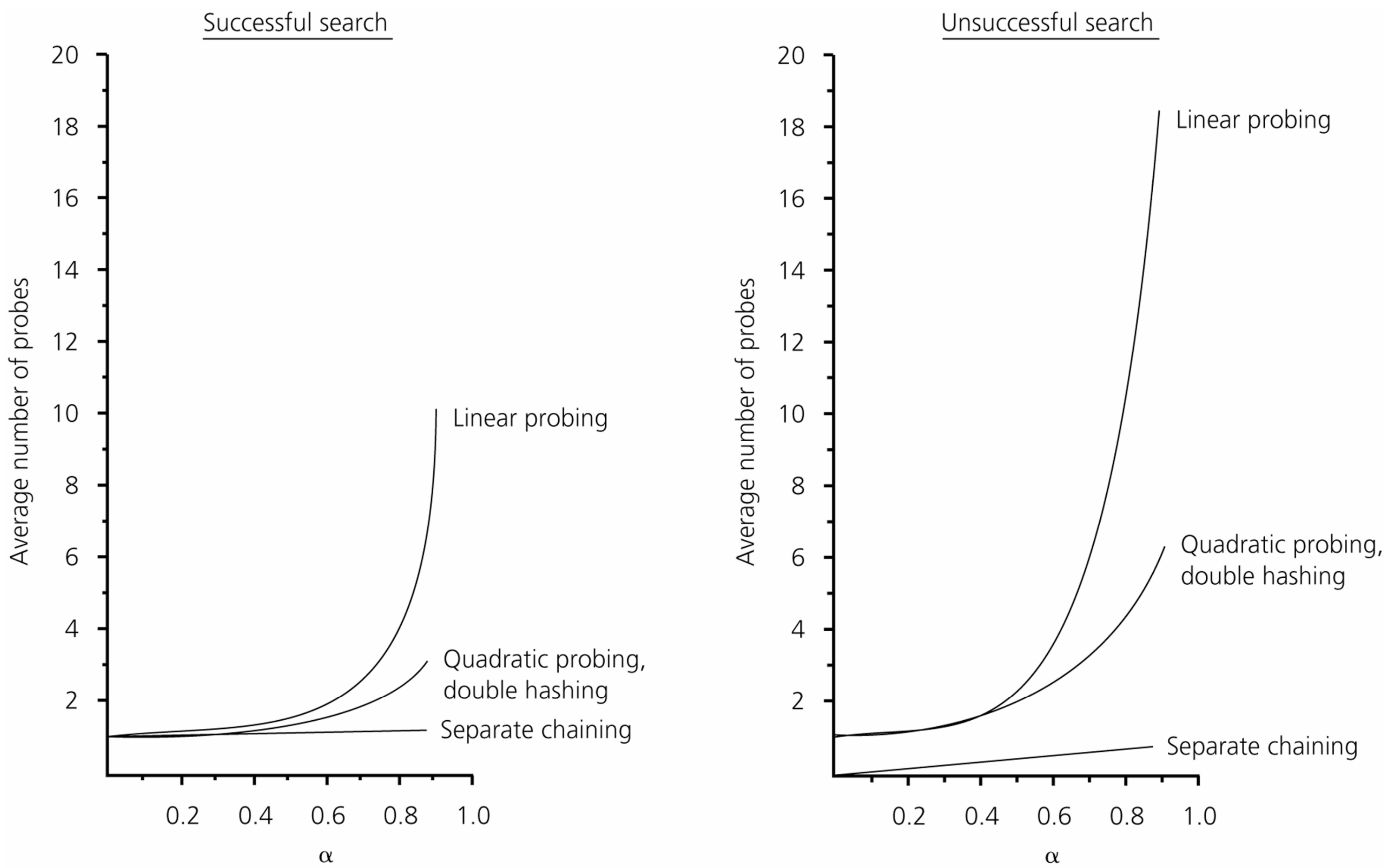


Figure: The relative efficiency of four collision-resolution methods

Comparing hash tables and balanced BSTs

- With good hash function and load kept low, hash tables perform insertions, deletions and search in O(1) time on average, while balanced BSTs in O(log n) time.
- However, there are some tasks (order related) for which, hash tables are not suitable:
  - traversing elements in sorted order: O(N+n.log n) vs. O(n)
  - finding minimum or maximum element: O(N) vs. O(1)
  - range query: finding elements with keys in an interval [a,b]: O(N) vs. O(log n + s), s is the size of output
- Depending on what kind of operations you will need to perform on the data and whether you need guaranteed performance on each query, you should choose which implementation to use.

CMPT 225

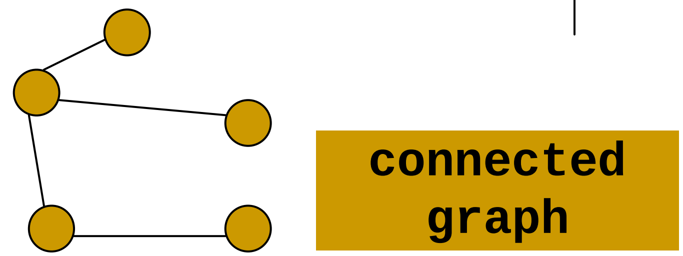
Graphs

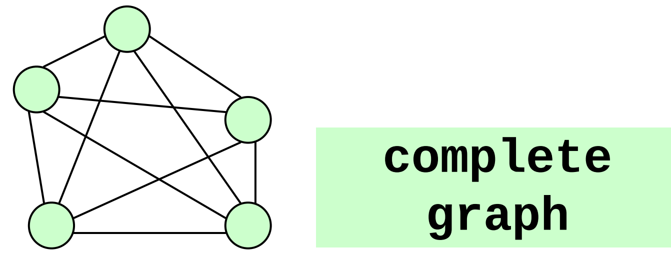
Graph Terminology

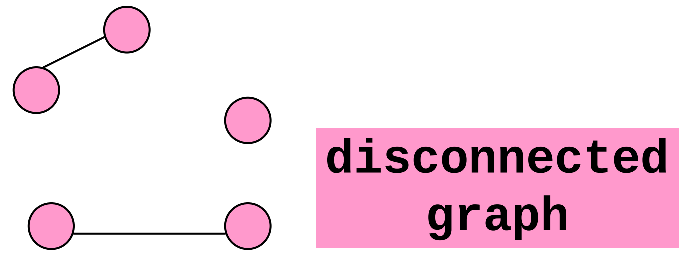
- A graph consists of two sets
  - A set V of vertices (or nodes) and
  - A set E of edges that connect vertices
  - |V| is the size of V, |E| the size of E
- A **path** (walk) between two vertices is a sequence of edges that begins at one vertex and ends at the other
  - A **simple path** (path) is one that does not pass through the same vertex more than once
  - A cycle is a path that begins and ends at the same vertex

Connected Graphs

- A connected graph is one where every pair of distinct vertices has a *path* between them
- A complete graph is one where every pair of vertices has an *edge* between them
- A graph cannot have multiple edges between the same pair of vertices
- A graph cannot have loops [a loop = an edge from and to the same vertex]

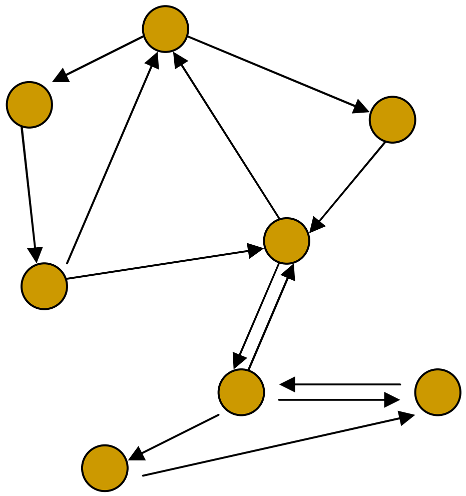






Directed Graphs

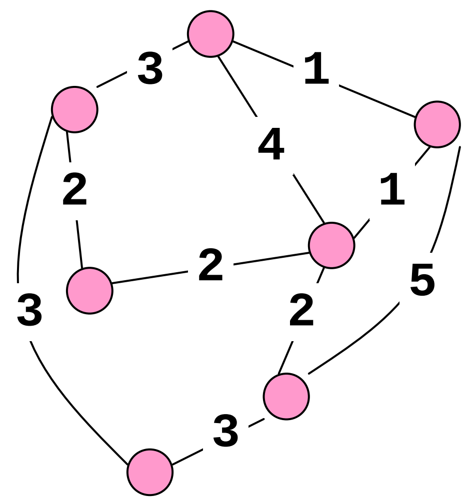
- In a directed graph (or digraph) each edge has a direction and is called a directed edge
- A directed edge can only be traveled in one direction
- A pair of vertices in a digraph can have two edges between them, one in each direction



directed graph

Weighted Graphs

- In a weighted graph each edge is assigned a weight
  - Edges are labeled with their weights
- Each edge's weight represents the cost to travel along that edge
  - The cost could be distance, time, money or some other measure
  - The cost depends on the underlying problem

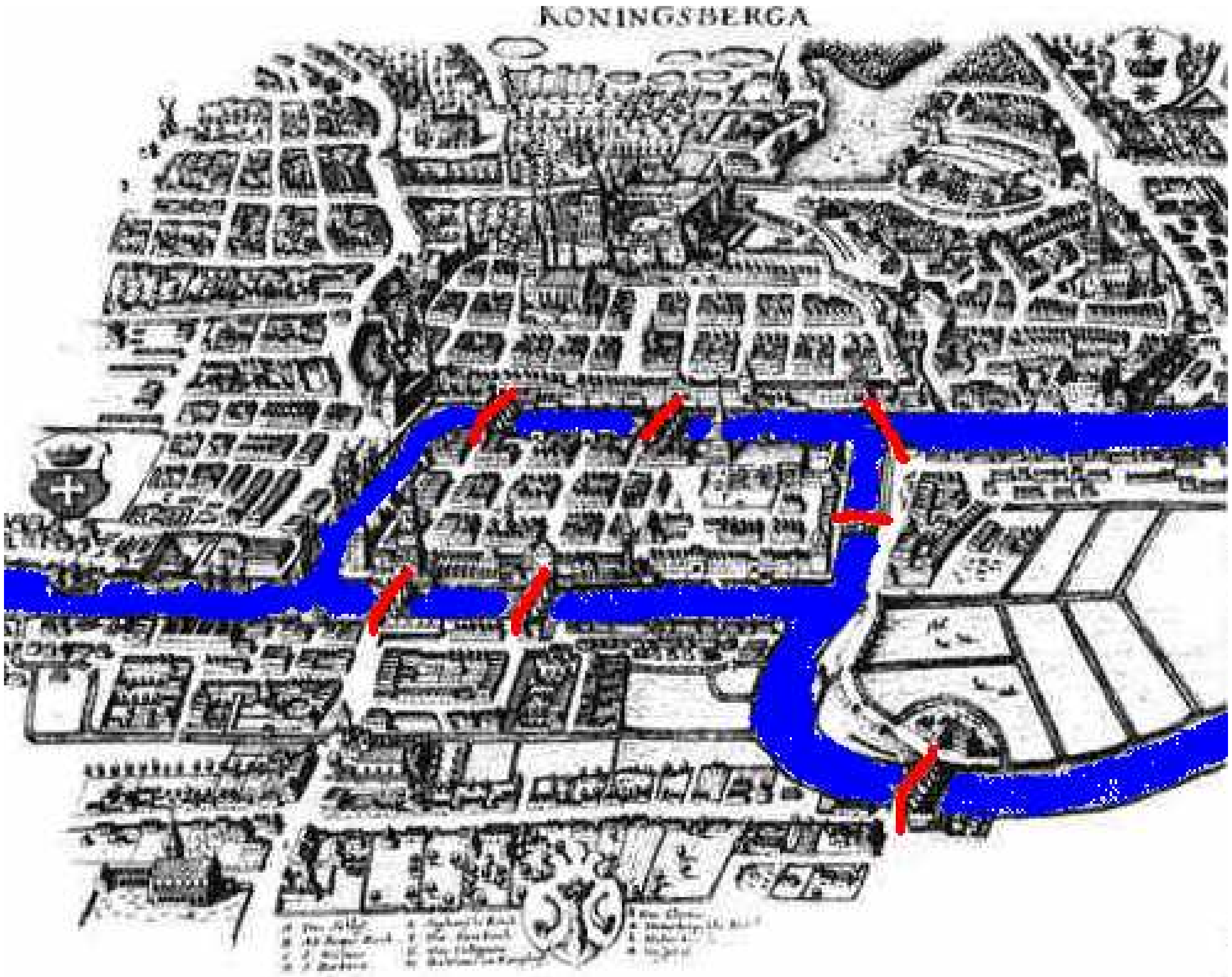


weighted graph

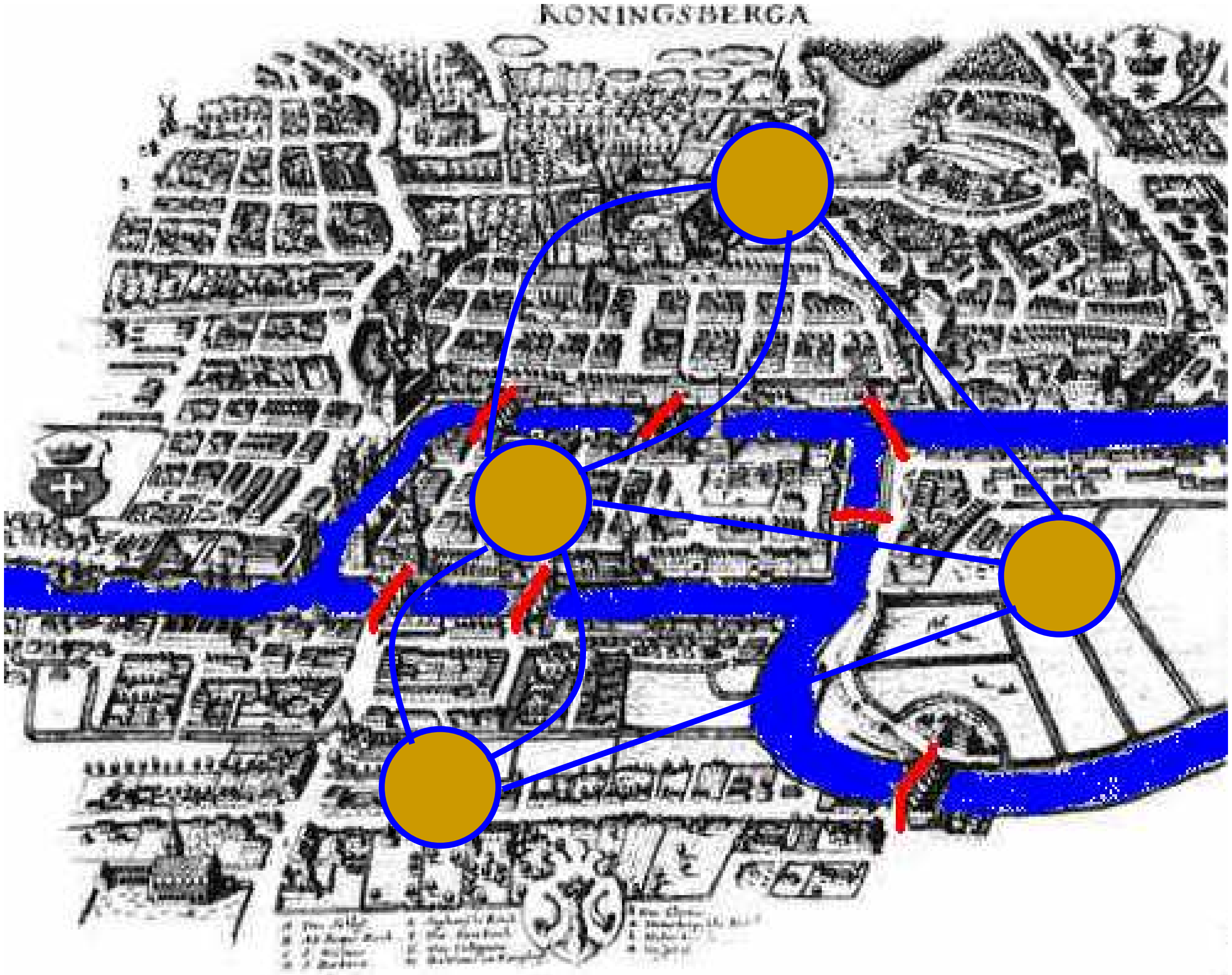
Graph Theory and Euler

- The Swiss mathematician Leonhard Euler invented graph theory in the 1700's
  - One problem he solved (in 1736) was the Konigsberg bridge problem
- Konigsberg was a city in Eastern Prussia which had seven bridges in its centre
  - Konigsberg was renamed Kalinigrad when East Prussia was divided between Poland and Russia in 1945
  - The inhabitants of Konigsberg liked to take walks and see if it was possible to cross each bridge once and return to where they started
  - Euler proved that it was impossible to do this, as part of this proof he represented the problem as a graph

Konigsberg

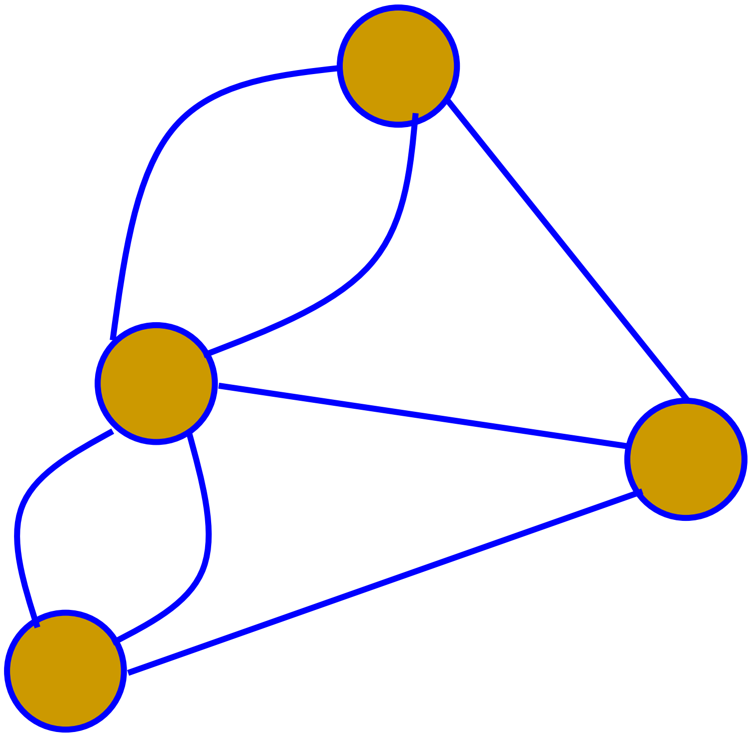


Konigsberg Graph



Multigraphs

- The Konigsberg graph is an example of a multigraph
- A multigraph has multiple edges between the same pair of vertices
- In this case the edges represent bridges



Graph Uses

- Graphs are used as representations of many different types of problems
  - Network configuration
  - Airline flight booking
  - Pathfinding algorithms
  - Database dependencies
  - Task scheduling
  - Critical path analysis
  - Garbage collection in Java
  - etc.

Basic Graph Operations

- Create an empty graph
- Test to see if a graph is empty
- Determine the number of vertices in a graph
- Determine the number of edges in a graph
- Determine if an edge exists between two vertices
  - and in a weighted graph determine its weight
- Insert a vertex
  - each vertex is assumed to have a distinct search key
- Delete a vertex, and its associated edges
- Delete an edge
- Return a vertex with a given key

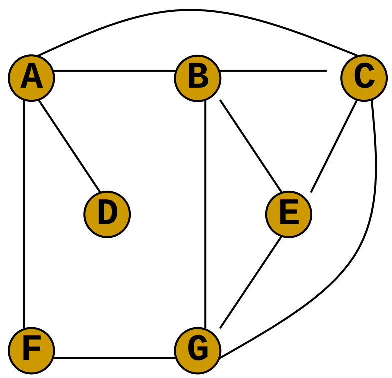
Graph Implementation

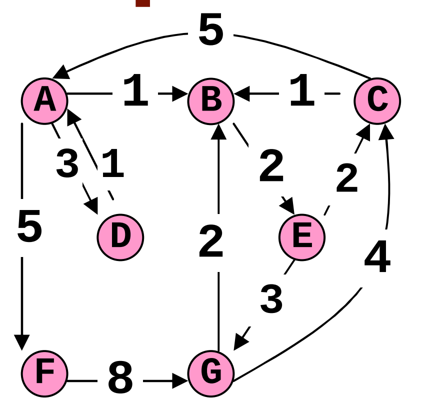
- There are two common implementation of graphs
  - Both implementations require to map a vertex (key) to an integer 0..|V|-1. For simplicity, we will assume that vertices are integers 0..|V|-1 and cannot be added or deleted.
  - The implementations record the set of edges differently
- **Adjacency matrices** provide fast lookup of individual edges but waste space for sparse graphs
- **Adjacency lists** are more space efficient for sparse graphs and find all the vertices adjacent to a given vertex efficiently

Adjacency Matrix

- The edges are recorded in an |V| * |V| matrix
- In an unweighted graph entries in matrix[*i*][*j*] are
  - 1 when there is an edge between vertices *i* and *j* or
  - 0 when there is no edge between vertices *i* and *j*
- In a weighted graph entries in matrix[*i*][*j*] are either
  - the edge weight if there is an edge between vertices *i* and *j* or
  - infinity when there is no edge between vertices *i* and *j*
- Looking up an edge requires O(1) time
- Finding all vertices adjacent to a given vertex requires O(|V|) time
- The matrix requires $|V|^2$ space

Adjacency Matrix Examples





	A	B	C	D	E	F	G
A	0	1	1	1	0	1	0
B	1	0	1	0	1	0	1
C	1	1	0	0	1	0	1
D	1	0	0	0	0	0	0
E	0	1	1	0	0	0	1
F	1	0	0	0	0	0	1
G	0	1	1	0	1	1	0

	A	B	C	D	E	F	G
A	∞	1	∞	3	∞	5	∞
B	∞	∞	∞	∞	2	∞	∞
C	5	1	∞	∞	∞	∞	∞
D	1	∞	∞	∞	∞	∞	∞
E	∞	∞	2	∞	∞	∞	3
F	∞	∞	∞	∞	∞	∞	8
G	∞	2	4	∞	∞	∞	∞

Implementation with adjacency matrix

```
class Graph {
  // simple graph (no multiple edges); undirected; unweighted
  private int numVertices;
  private int numEdges;

  private boolean[][] adjMatrix;

  public Graph(int n) {
    numVertices = n;
    numEdges = 0;
    adjMatrix = new boolean[n][n];
  } // end constructor

  public int getNumVertices() {
    return numVertices;
  } // end getNumVertices

  public int getNumEdges() {
    return numEdges;
  } // end getNumEdges

  public boolean isEdge(int v, int w) {
    return adjMatrix[v][w];
  } // end isEdge
```

```java
  public void addEdge(int v, int w) {
    if (!isEdge(v,w)) {
      adjMatrix[v][w] = true;
      adjMatrix[w][v] = true;
      numEdges++;
    }
  } // end addEdge

  public void removeEdge(int v, int w) {
    if (isEdge(v,w)) {
      adjMatrix[v][w] = false;
      adgMatrix[w][v] = false;
      numEdges--;
    }
  } // end removeEdge

  public int nextAdjacent(int v,int w)
  // if w<0, return the first adjacent vertex
  // otherwise, return next one after w
  // if none exist, return -1
  {
    for (int i=w+1; i<numVertices; i++)
      if (isEdge(v,i))
        return i;
    return -1;
  }
} // end Graph
```