Our first non-linear data structure!

A graph $G$ consists of two sets $G = \{V, E\}$
- A set of $V$ vertices, or nodes
- A set of $E$ edges, relationships between nodes

A subgraph $G'$ consists of a subset of the vertices and edges of $G$

Adjacent vertices are two vertices joined by an edge
- A **path** between two vertices
  - A sequence of edges that begins at the first vertex and ends at the other vertex
- A **simple path**
  - A path that passes through a vertex at most once
- A **cycle**
  - A path that begins and ends at the same vertex
- A **simple cycle**
  - A cycle that does not pass through a vertex more than once

- A **complete** graph
  - A graph that has an edge between each pair of distinct vertices
- How many edges does a complete graph with \( n \) nodes have?

- A **connected** graph
  - A graph that has a path between each pair of distinct vertices

- A **disconnected** graph
  - A graph that has at least one pair of vertices without a path between them

- **Directed** graph
  - Each edge is a directed edge, or an arc, or link
  - May have two arcs between a given pair of vertices, one in each direction
  - Vertex \( y \) is adjacent to vertex \( x \) **iff** (if and only if) there is a directed edge from \( x \) to \( y \)
- Directed path
  - A sequence of directed edges between two vertices

- **Directed Acyclic Graph (DAG)**
  - Directed graph with no cycles

**How few arcs can you remove to make the graph a DAG?**
- **Weighted graph**
  - A graph whose edges have weights
  - Weight is the “cost” or “magnitude” of the relationship represented by the edge

```
public interface Graph<T>   // partial

public boolean isEmpty() // returns true iff a graph is empty
public int n() // returns the number of vertices in a graph
public int m() // returns the number of edges in a graph

public void addVertex(T v)   // Insert a vertex in a graph
public void removeVertex(T v)
   // Deletes a vertex from a graph along with any edges between the vertex and other vertices

public void addEdge(T v1, T v2)
   // Insert an edge between two given vertices in a graph

public void removeEdge(T v1, T v2)
   // Deletes the edge between two given vertices in a graph

public T findVertex(String key)
   // Retrieves and returns the vertex that contains a given search key

public boolean isEdge(T v1, T v2)
   // returns true iff an edge exists between two given vertices

public LinkedList<T> getNeighbors(T v)   // FOR UNDIRECTED GRAPH
   // Retrieves and returns a list of the vertices adjacent to vertex v
```

- **Adjacency matrix for a graph with** $n$ **vertices numbered 0, 1, ..., $n - 1**
- **A boolean** $n \times n$ **array** *Arcs* **such that** $Arcs[i][j] =$
  - 1 (true) iff there is an arc from vertex $i$ to vertex $j$
  - 0 (false) iff there is no arc from vertex $i$ to vertex $j$

What property does the matrix of an undirected graph have?
public class AdjMatGraph<T> implements Graph<T> {
    private final int NOT_FOUND = -1;
    private final int DEFAULT_CAPACITY = 10;

    private int n; // number of vertices in the graph
    private boolean[][] arcs; // adjacency matrix of arcs
    private T[] vertices; // values of vertices

    public AdjMatGraph(){
        n = 0;
        this.arcs = new boolean[DEFAULT_CAPACITY][DEFAULT_CAPACITY];
        this.vertices = (T[])(new Object[DEFAULT_CAPACITY]);
    }
}

- Which representation supports better these two frequent operations on graphs?
  - isEdge(v, w)
    Determine whether there is an edge from vertex v to vertex w
  - getNeighbors(v)
    Return list of all vertices linked to from a given vertex v

- An adjacency list for a graph with n vertices numbered 0, 1, ..., n – 1
  - Consists of n linked lists
  - The i<sup>th</sup> linked list has a list entry for vertex j
    iff the graph contains an arc from vertex i to vertex j

- Adjacency matrix for a weighted graph with n vertices numbered 0, 1, ..., n – 1
  - An n x n array matrix EdgeW such that
    EdgeW[i][j] =
    - The weight of the arc from vertex i to vertex j
    - ∞ iff there is no edge from vertex i to vertex j
• Adjacency list for a weighted undirected graph
  - Each list entry contains the edge label and weight
  - Treats each edge as if it were two arcs in opposite directions

• Multigraph
  - Not a graph
  - Allows multiple edges between vertices
  - Multiple edges indicate multiple relations between vertices

Tree: A Special Graph

A tree is a connected graph in which there is exactly one simple path connecting any two nodes

How many edges does a tree with $n$ nodes have?

• yEd: A great and simple graph visualization
  - You can create any graph by clicking (for vertices) and clicking-and-dragging (for edges)
  - Lots of graph formats supported. Use .tgf
  - TGF format: a text file listing lines of:
    - vertexID vertexName (for vertices)
    - # vertexID pairs (for arcs)
  - Once you upload a file, choose Layout > Circular to see it laid out nicely.
A graph-traversal algorithm
- Visits all the vertices that it can reach starting at some vertex
- Visits all vertices of the graph *iff* the graph is connected (effectively computing Connected Components)
- Must not loop forever, if a graph contains a cycle
- Must never visit a vertex more than once

Connected component (for undirected graphs) =
- The subset of vertices visited during a traversal that begins at a given vertex

Strongly connected component (for directed graphs) =
- The subset of vertices visited during a traversal that begins at any of its members

High Planes Airline Company (HPAir)

Problem
- For each customer request, indicate whether a sequence of HPAir flights exists from the origin city to the destination city

The flight map for HPAir is a directed graph
- Arc between vertices means
  - There is a flight between cities
- Directed path means
  - There is a sequence of flight connections

The solution performs an exhaustive search
- Beginning at the origin city, tries every possible sequence of flights until either
  - Finds a sequence that gets to the destination city
  - Determines that no such sequence exists

The ADT Stack is useful in organizing an exhaustive search
- It helps you remember how you got to the current point

Backtracking can be used to recover from a wrong choice of a city
DFS(originCity): Searching the Flight Map

\[ \text{stk} = \text{new Stack}\langle E \rangle(); \]
\[ \text{stk}.\text{push}(\text{originCity}); \]
\[ \text{while} \ (\text{a sequence of flights from} \ \text{originCity} \ \text{to} \ \text{destinCity} \ \text{has not been found}) \ { \}
\[ \hspace{1em} \text{if} \ (\text{you cannot go anywhere from the city on top of stack}) \]
\[ \hspace{2em} \text{stk}.\text{pop}(); \ // \text{backtrack} \]
\[ \text{else} \ \text{select a neighbor,} \ \text{anotherCity}, \ \text{from the city on top of stack;} \]
\[ \hspace{2em} \text{stk}.\text{push}(\text{anotherCity}); \]
\[ \}\]

... and remember where you've been

\[ \text{stk} = \text{new Stack}\langle E \rangle(); \ \text{Clear Marks}; \]
\[ \text{stk}.\text{push}(\text{originCity}); \]
\[ \text{Mark(} \text{originCity} \text{)} \ \text{as visited}; \]
\[ \text{while} \ (\text{a sequence of flights from} \ \text{originCity} \ \text{to} \ \text{destinCity} \ \text{has not been found}) \ { \}
\[ \hspace{1em} \text{if} \ (\text{you cannot find an unvisited city from the city on top of stack}) \]
\[ \hspace{2em} \text{stk}.\text{pop}(); \ // \text{backtrack} \]
\[ \text{else} \ \text{select an unvisited neighbor,} \ \text{anotherCity}, \ \text{from the city on top of stack;} \]
\[ \hspace{2em} \text{stk}.\text{push}(\text{anotherCity}); \]
\[ \text{Mark(} \text{anotherCity} \text{)} \ \text{as visited}; \]

Depth-First-Search Example: From P→Z

List visited (marked)

P R X W S T Y Z

Stack stk

Would DFS(oC) work for undirected graphs?

\[ \text{stk} = \text{new Stack}\langle E \rangle(); \ \text{Clear Marks}; \]
\[ \text{stk}.\text{push}(\text{originCity}); \]
\[ \text{Mark(} \text{originCity} \text{)} \ \text{as visited}; \]
\[ \text{while} \ (\text{a sequence of flights from} \ \text{originCity} \ \text{to} \ \text{destinCity} \ \text{has not been found}) \ { \}
\[ \hspace{1em} \text{if} \ (\text{you cannot find an unvisited city from the city on top of stack}) \]
\[ \hspace{2em} \text{stk}.\text{pop}(); \ // \text{backtrack} \]
\[ \text{else} \ \text{select an unvisited neighbor,} \ \text{anotherCity}, \ \text{from the city on top of stack;} \]
\[ \hspace{2em} \text{stk}.\text{push}(\text{anotherCity}); \]
\[ \text{Mark(} \text{anotherCity} \text{)} \ \text{as visited}; \]
**Labyrinth: Umberto Eco advises Theseus**

“To find the way out of a labyrinth there is only one means. At every new junction, never seen before, the path we have taken will be marked with three signs. If … you see that the junction has already been visited, you will make only one mark on the path you have taken. If all the apertures have already been marked, then you must retrace your steps. But if one or two apertures of the junction are still without signs, you will choose any one, making two signs on it. Proceeding through an aperture that bears only one sign, you will make two more, so that now the aperture bears three. All the parts of the labyrinth must have been visited if, arriving at a junction, you never take a passage with three signs, unless none of the other passages is now without signs.”

**Mazes as Graphs**

**Testing for Connectivity using DFS(oC)**

Connected: An undirected graph for which there is a path from any node to any other node

Is this graph connected?

Connected component: A connected sub-graph

Can we use DFS to find all connected components?