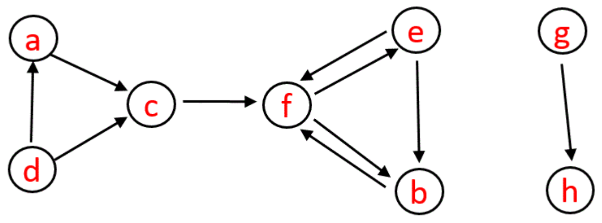
Graphs

You are familiar with data structures such as arrays and lists (linear), and trees (non-linear). We next consider a more general non-linear data structure, known as a graph. Here is an example.



Like in many previous data structures, a graph contains a set nodes. Each node has a reference to other nodes. For graphs, a reference from one node to another is called an *edge*.

In a linked list, there are references from one to the "next" and/or "previous" node. In a (rooted) tree, there are references to children nodes or siblings or parent nodes. In a graph, there is no unique notion of "next" or "prev" as in a list, and there is no unique notion of child or parent. Every node can potentially reference every other node. We will discuss data structures for graphs below.

Graphs have been studied and used for many years, and some of the basic results go back a few hundred years, to mathematicians like Euler. Mathematically, a graph consists of a set V called "vertices" and a set of edges $E \subseteq V \times V$, that is, the edges E in a graph is some set of *ordered* pairs of vertices.

When the ordering of the vertices in an edge is important, we say that we have a *directed graph*. One can also define graphs in which the edges do not have "arrows", that is, each edge is a pair of vertices but we don't care about the order. This is called an *undirected* graph, and in this case we would draw the edges as line segments with no arrows.

Examples of graphs include transportation networks. For example, V might be a set of airports and E might be direct flights between airports. Or V might be a set of html documents and E might be defined by URLs (links) between documents.

Data Structures for Graphs

Unlike linked lists, Java does not come with a class for graphs. The reason is probably that there are many ways to define graph data structures and some might work better than others, depending on what you want to use them for. Here I will present one graph data structure, which we will use for Assignment 4.

We will have a label (key) for each of the vertices, and we will use the key to access the vertices by making a hash map. Let each vertex have a generic type `T` associated with it. For example, if the vertex were an airport, then we might have information about the airport.

```
  class  Graph<T>{
     HashMap< String, Vertex<T> >    vertexMap;
           :                                        //  other stuff
}
```

There are two general ways to represent the edges, known as adjacency lists and adjacency matrix.

Adjacency List

For an adjacency list representation, for each vertex $v \in V$, we represent a list of vertices w such that $(v, w) \in E$. For example, here is the adjacency lists for the graph on the previous page.

```
a  -  c
b  -  f
c  -  f
d  -  a,c
e  -  b,f
f  -  b,e
g  -  h
h  -
```

In our Java implementation in Assignment 4, we will use the following:

```
class Vertex<T> {
    LinkedList<Edge<T>>  adjList;
    String               key;           // convenient but not necessary
    boolean              visited;
    T                    element;
}

class Edge<T> {
    Vertex<T>            endVertex;
    double               weight;
      :
}
```

Notice that each `Vertex` object has several fields. The most important for now is the adjacency list for that vertex which is a linked list of `Edge` objects. This might be surprising – you might have expected it to be a list of other `Vertex` objects. The main reason we define the adjacency list this way is that it is more convenient for associating other information with the edge, such as a weight in this example. If the adjacency list were defined by a list of `Vertex` objects instead, then in order to access the weight information, one would need another data structure that lists all the edges and the weights associated with those edges. Using the present scheme, this extra data structure isn't necessary. (The penalty one pays is that it is a bit confusing to represent the edge implicitly by its start vertex and explicitly by its end vertex.)

Using the classes above, how many Java objects are there for the example graph on the previous page? There are eight vertices and hence eight `Vertex` objects. There is one `HashMap` for identifying the keys with the vertices. (The HashMap itself has an underlying array and linked list structure which we ignore here.) Each `Vertex` object has an adjacency list, which is a `LinkedList<Edge>` object. There are also ten `Edge` objects, and these are referenced by the nodes in the `LinkedList` (and these linked list nodes are also objects, but I ignore them here).

Adjacency Matrix

A different data structure for representing the edges in a graph is an *adjacency matrix* which is a $|V| \times |V|$ array of booleans, where $|V|$ is the number of elements in set V i.e. the number of vertices. The value 1 at entry $(v1, v2)$ in the array indicates that $(v1, v2)$ is an edge, that is, $(v1, v2) \in E$, and the value 0 indicates that $(v1, v2) \notin E$.

The adjacency matrix for the graph from earlier is shown below.

```
  abcdefgh
a 00100000
b 00000100
c 00000100
d 10100000
e 01000100
f 01001000
g 00000001
h 00000000
```

Note that in this example, the diagonals are all 0's, meaning that there are no edges of the form (v, v). But graphs can have such edges. An example is given in the slides.

Terminology

Finally, here is some basic graph terminology that you should become familiar with, and that is heavily used in discussing properties of graphs. Please see the slides for examples.

- *outgoing edges from v* - the set of edges of the form $(v, w) \in E$ where $w \in V$
- *incoming edges to v* - the set of edges of the form $(w, v) \in E$ where $w \in V$
- *in-degree of v* - the number incoming edges to v
- *out-degree of v* - the number outgoing edges from v
- *path* - a sequence of verticies $(v_1, \ldots, v_m)$ where $(v_i, v_{i+1}) \in E$ for each i.

  Note that the definition of path is similar to the definition of path that we saw for trees.
- *cycle* - a path such that the first vertex is the same as the last vertex

  If there were an edge (v, v), then this would be considered as a cycle. Such edges are certainly allowed in graphs and indeed are quite common.

At the end of the lecture, I mentioned a few definitions and important graph problems that you will see in subsequent courses. Details omitted in these lecture notes.