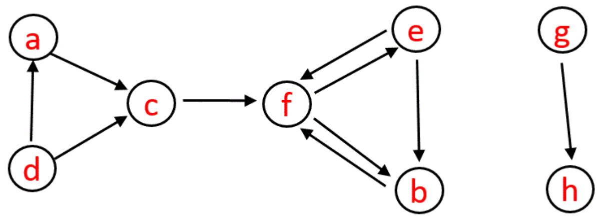
Graphs

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 $E \subseteq V \times V$. The edges E in a graph is a 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. There are many other examples. V might be a set of html documents and E would be the URLs (links) between documents. Another example is that V might be a set of objects in a running Java program, and E would be a set of references, namely when an object has a field (reference variable) that references another object. In a future lecture, I will discuss the graph of html documents and examine how google search works. I will also discuss the graph of objects referencing each other and examine how garbage collection works.

Terminology

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.

- *weighted graph* – a graph that has a number (weight) associated with each edge; for example, in a flight network where the vertices are airports, the weight might be the time it takes to fly between two airports
- *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 vertices $(v_1, \ldots, v_m)$ where $(v_i, v_{i+1}) \in E$ for each i. The length of a path in a graph is the number of edges in the path (not the number of vertices). The definition of path is essentially the same for graphs as it was 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 called loops.
- *directed acyclic graph* (DAG) – a directed graph that has no cycles. Such graphs are used to capture dependencies between objects or events. For example, the graph of prerequisite relationships in McGill courses is directed and acyclic.

In the lecture itself, I briefly discussed a few important graph problems that you will see in subsequent courses.

Adjacency List

A graph is a generalization of a tree. Each node in a tree has a list of children. Similarly, each graph vertex v has a list of other vertices w that are adjacent to it. We call this an *adjacency list*, namely 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  -
```

I have represented vertices in alphabetic order, but this is not necessary.

Data structures for graphs

Java does not have a `Graph` class since there are too many different types of graphs and no one size fits all. So one needs to implement one's own `Graph` class. A very basic `Graph` class might be as simple as this:

```
class Graph<T>{

   class Vertex<T> {
      LinkedList<Vertex<T>>  adjList;
      T                      element;
   }
}
```

However, it is common to have other vertex attributes and also to have attributes for the edges, in particular, edge weights. So a more common graph would be like this:

```
class Graph<T>{

   class Vertex<T> {
       LinkedList<Edge<T>>  adjList;
       T                    element;
   }

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

Now the adjacency list for a vertex is a list of `Edge` objects and each edge is represented only by the end vertex of the edge. The start vertex of each edge does not need to be represented explicitly because it is the vertex that has the edge in its adjacency list.

An important difference between rooted trees and graphs is that rooted trees have a special node (the root) where methods between. For graphs, we may wish to access any node. For this, we need a map.

We will have a label (key) for each of the vertices, and we will use the key to access the vertices by using a hash map. The key might be a string, for example.

```
class  Graph<T>{
     HashMap< String, Vertex<T> >    vertexMap;
        :
         //  Vertex and Edge inner classes as above
}
```

Using the classes above, how many Java objects are there for the example graph on page 1? 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.

$$\begin{array}{c|cccccccc}
 & a & b & c & d & e & f & g & h \\
\hline
a & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
b & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
c & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
d & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
e & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
f & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
g & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
h & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

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 (called *loops*). An example is given in the slides.

I finished the lecture by comparing adjacency lists and adjacency matrices. When would you use one rather than another. See the Exercises.