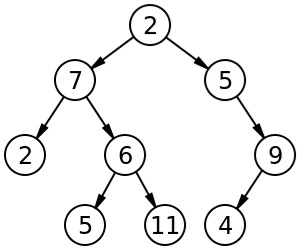
**Introduction to graphs and graph searches**

Here is a **binary tree**. The node shown at the top is the (overall) root.



Each node has a left and right sub-tree (which may be null).

The following class could be used to represent a node in a binary tree:

class Node

{

Object value;

Node left, right;

}

The following code (which you should be familiar with) traverses a binary tree, printing all the values in it:

public void print(Node n)

{

if (n != null) {

System.out.println(n.value);

print(n.left);

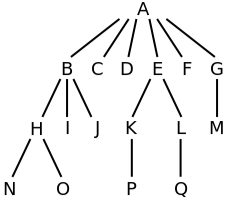
print(n.right);

}

}

Given the binary tree shown previously, the values are printed in the order of: 2, 7, 2, 6, 5, 11, 5, 9, 4

In a *binary* tree, each node has 0, 1, or 2 children. In a ***general tree***, each node can have an arbitrary number of children.



The following class could be used to represent a node in a *general* tree:

class Node

{

Object value;

Set<Node> children;

}

The following code traverses a *general* tree, printing all the values in it:

public void print(Node n)

{

if (n != null) {

System.out.println(n.value);

for (Node child : n.children)

print(child);

}

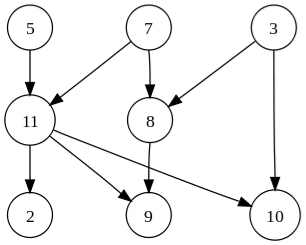
}

For the general tree shown previously, the values are printed: A, B, H, N, O, I, J, C, D, E, K, P, L, Q, F, G, M. (Assume that the for loop iterates over children in alphabetical order.)

print is using a ***depth-first traversal***, meaning it goes "down" as far as possible before returning to the previous node. With binary trees, we called this an in-order traversal. We can also use a stack to implement a depth-first traversal without recursion.

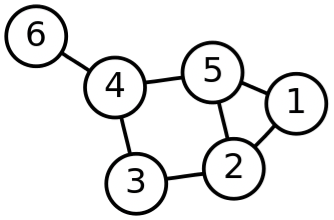
If we use a queue instead of a stack (implicit or explicit), the code will perform a ***breadth-first traversal****.* With binary trees, we called this a *level-order traversal*.

A ***graph*** is basically an unrestricted tree (with no overall root), where nodes can be connected to any number of other nodes.



A node in a graph is usually called a ***vertex***. The linkages between nodes are called ***edges***. In the graph shown above, values are associated with the nodes. In some graphs, values may also be associated with the edges.

The graph shown above is *a* ***directed graph***. Each edge has a direction. (5's child is 11, but 5 is not one of 11's children.) An ***undirected graph*** is shown below:

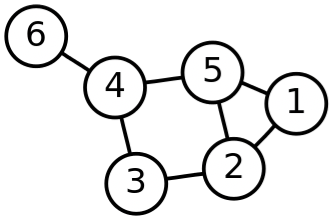


In this graph, the edges are bi-directional. (6's child is 4, and 6 is also one of 4's children).

Here are 3 kinds of data that graphs might be used to represent:

1. A map, where nodes are cities and edges are highways (or airplane flights), OR a map, where nodes are intersections and edges are road segments
2. A computer network, where nodes are routers and edges are wires, OR the web, where nodes are web pages and edges are hyperlinks
3. A social network, where nodes are people and edges are friendships,  
   OR a food web, where nodes are animals and edges indicate who eats whom

Consider the following graph:



Given the following traversals of the graph above, label each as either depth-first search, breadth-first search, or neither (solutions in white at the end of each line, download this doc and change font color to view):

* + 4, 6, 5, 3, 2, 1 BFS
  + 1, 2, 5, 4, 6, 3 DFS
  + 3, 2, 5, 4, 1, 6 Neither

What goes wrong if we use our most recent print code on a graph like the one shown above? Do a mental traversal of this graph.

An infinite loop occurs! Without some way of keeping track of which nodes have already been visited, this code will never terminate. With binary trees this wasn’t an issue, as both left and right links only ever lead down the tree (never back to a previous node).

See the "Traversing a graph" folder for more info (when ready) on graph traversals.