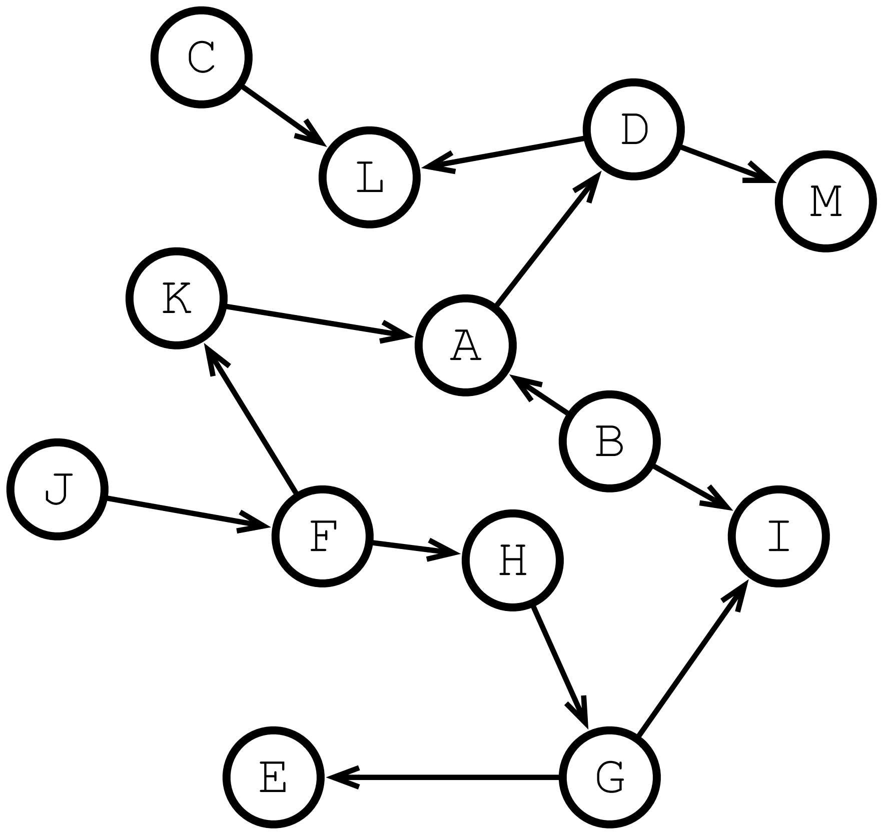
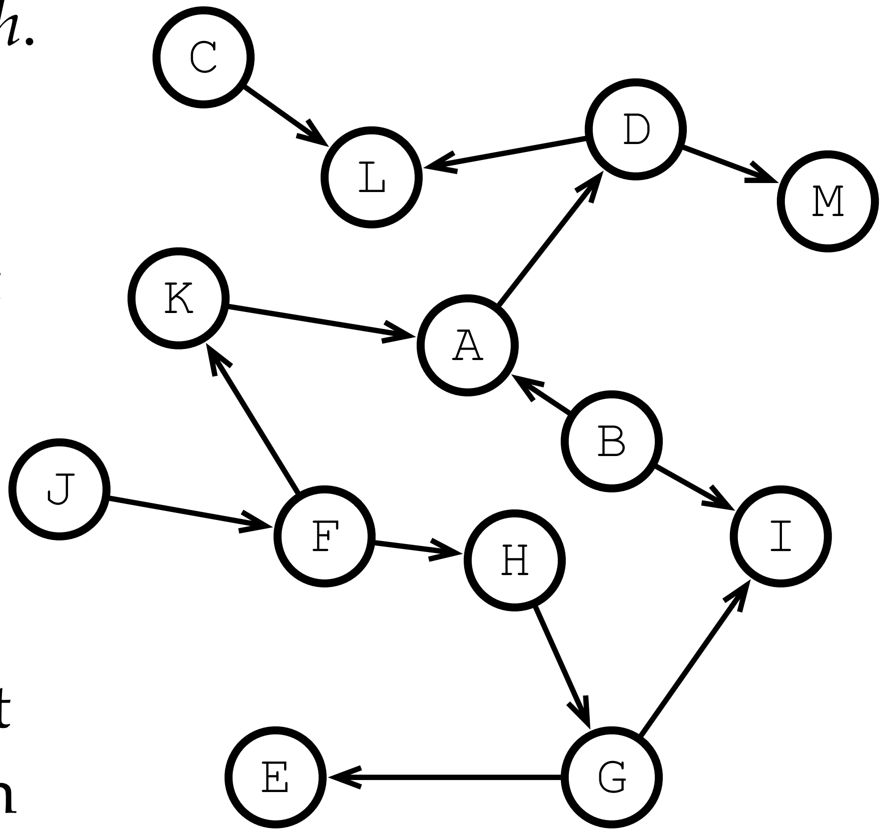
Depth First Search
Breadth First Search



Recall what depth-first and breadth-first search are:

They are about *exploring a graph*. They can tell us a lot of useful information about the graph's topology. Are there cycles? Is it bipartite? What's the distance from nodes A to B? Can we even discover B from A?



Basically everything you might want to do with graph data can be done by using some combination of BFS and DFS.

You can modify both DFS and BFS for your purposes, obviously. There are *many* variations of these algorithms, but the basic structure looks the same.

A DFS requires a ***stack*** data structure, which is the last-in, first-out structure we talked about way back when.

A BFS requires a ***queue*** data structure, which is the first-in, first out structure.

We can use the C++ vector template class as either a stack or a queue, depending on the specific member functions we use.

In this assignment there are three classes for you to implement. I have already defined their structure in graph.hpp, and you need to edit graph.cpp to fill in the missing functions. The three classes are:

Graph

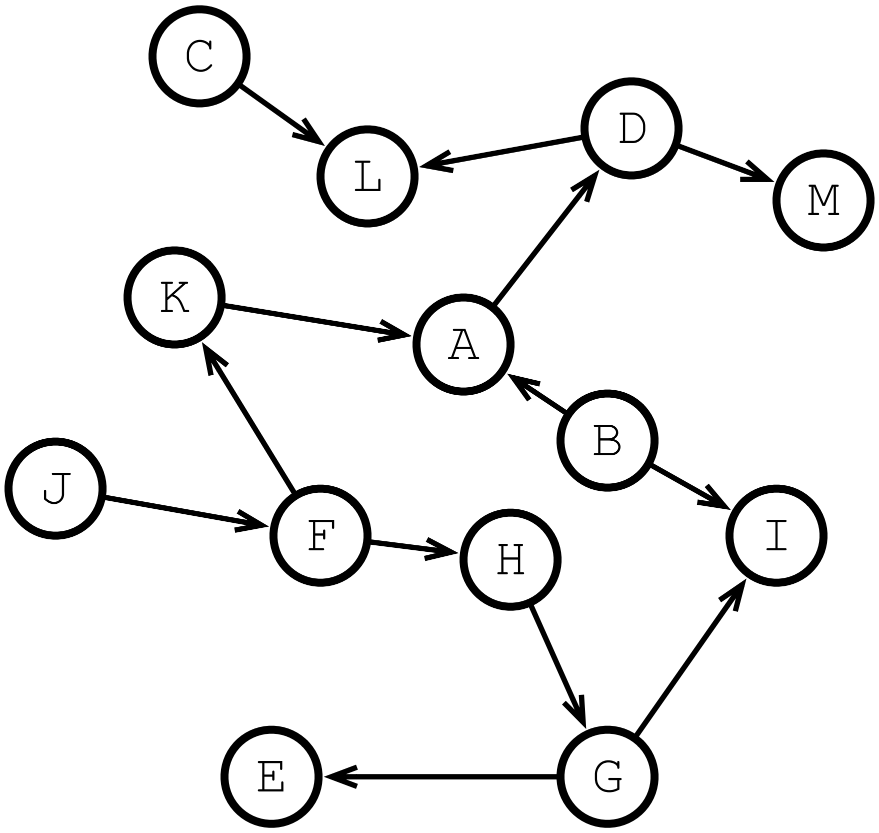
Holds Node and Edge objects that define the data and data structure. Gives access to the DFS and BFS operations.

Node

Holds data (in first assignment this is a string). DFS sets discovery and finish times. BFS sets discovery rank.

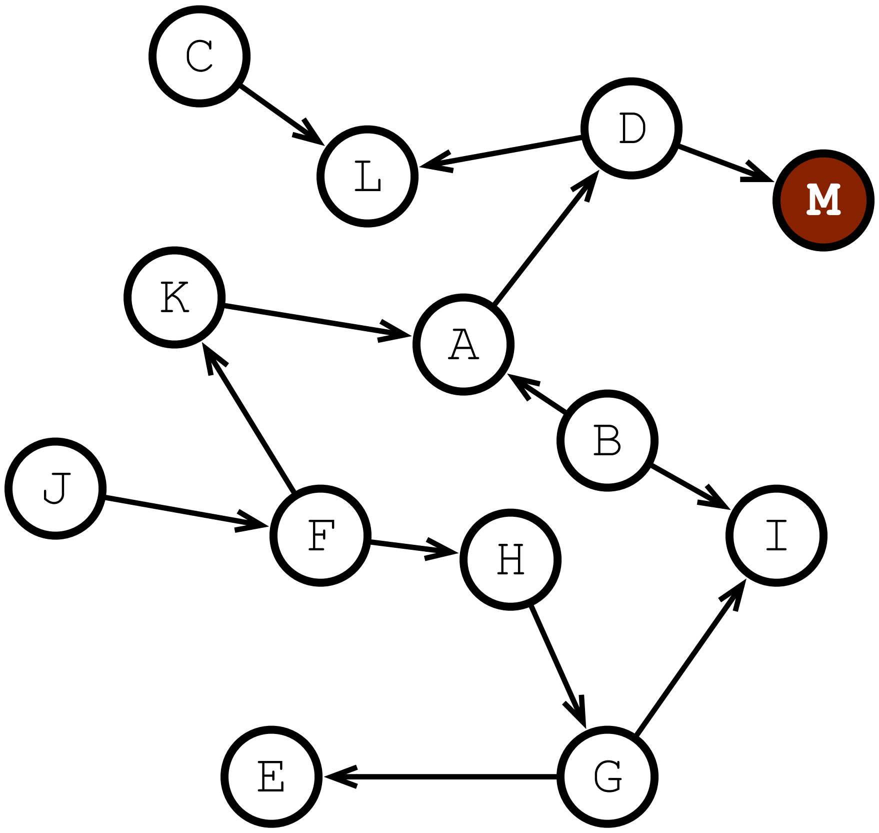
Edge

A relation between two Nodes. Doing a DFS sets edge *type*. May have a *weight* indicating cost of traversing it (not used in 1st assign).

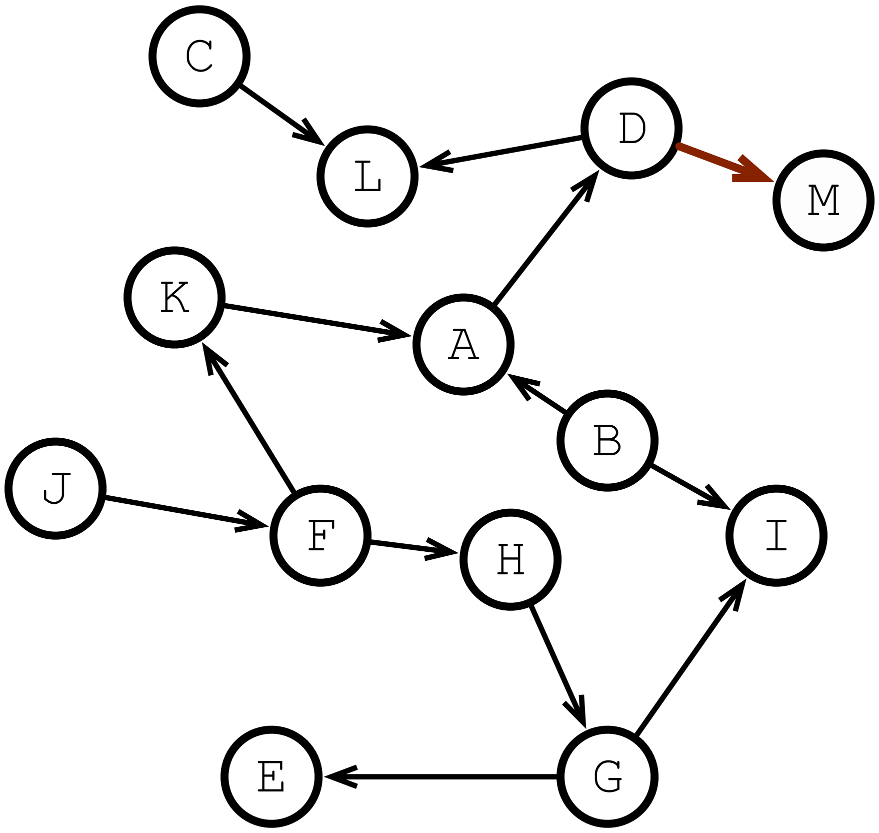


The whole thing here is a graph. There is not necessarily a 'start' node, or one that is 'first', unless our application semantics dictate that.

In general we could choose to begin a graph exploration at any of these locations.



A single node is indicated with one of the circles. They contain data (a string in our case), and the DFS and BFS operations assign additional useful information (discovery time or rank, color).



An Edge is indicated with an arrow. It holds references to the Node objects it joins.

Edges can be directed (have a distinct start and end, indicated with arrowheads), or undirected (make no distinction between start/end, shown without arrowheads).

Importantly, our Edge objects don't know if they are directed or not. That information is actually a property of the graph. This was an implementation decision I made since I wanted all edges in the graph to be the same, and they can share a common value.

```
#define WHITE 1
#define GRAY 2
#define BLACK 3

#define UNDISCOVERED_EDGE 9
#define TREE_EDGE 10
#define BACK_EDGE 11
#define FORWARD_EDGE 12
#define CROSS_EDGE 13
```

Nodes are *colored* according to their discovery state.

White = Not (yet) discovered
Gray = Discovered but not (yet) finished
Black = Finished

```
#define WHITE 1
#define GRAY 2
#define BLACK 3

#define UNDISCOVERED_EDGE 9
#define TREE_EDGE 10
#define BACK_EDGE 11
#define FORWARD_EDGE 12
#define CROSS_EDGE 13
```

Edges are given *types* according to how they were discovered.

We'll talk about these later.

```
class Graph {
private:
  bool directed;

  vector<Node*> nodes;
  vector<Edge*> edges;

  vector<Edge*> search_edges;
  vector<Node*> search_nodes;

  int clock;
public:
  // will show these next
}
```

'directed' tells us if the Edges have distinct start/end nodes (like parent/child) or if they make no such distinction (like friends in a social network).

```
class Graph {
private:
  bool directed;

  vector<Node*> nodes;
  vector<Edge*> edges;

  vector<Edge*> search_edges;
  vector<Node*> search_nodes;

  int clock;
public:
  // will show these next
}
```

'nodes' and 'edges' contain information about the graph's data (in 'nodes') and the structure of the data (in 'edges').

Notice that the vectors contain *pointers* to Node and Edge objects.

```
class Graph {
private:
  bool directed;

  vector<Node*> nodes;
  vector<Edge*> edges;

  vector<Edge*> search_edges;
  vector<Node*> search_nodes;

  int clock;
public:
  // will show these next
}
```

'search_edges' & 'search_nodes' contain information about an ongoing or most recently completed BFS or DFS. The one for edges is not necessary for this assignment, but some algorithms can use it.

The one for nodes is used as your queue or stack for BFS and DFS respectively.

Notice that the vectors contain *pointers* to Node and Edge objects.

```
class Graph {
private:
  bool directed;

  vector<Node*> nodes;
  vector<Edge*> edges;

  vector<Edge*> search_edges;
  vector<Node*> search_nodes;

  int clock;
public:
  // will show these next
}
```

The 'clock' variable should be used to keep track of the current time-step in a DFS.

Constructor and Destructor

```
class Graph {
private:
  ...
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
}
```

```
class Graph {
private:
  ...
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
}
```

get node and edge vectors. Used to query state of graph after a search. Node and Edge discovery information should be kept around after a search, so traversing these vectors gives us the results of the search.

```
class Graph {
private:
  ...
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
}
```

Housekeeping functions to add or remove data and structure.

```
class Graph {
private:
  ...
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
}
```

Accessor and mutator functions that query and set the edges to directed or undirected. Again: this is a property of the graph, not of individual edges, though other implementations might do it differently.

```
class Graph {
private:
  ...
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
}
```

Supplies you with a set of edges that are related to this node. For undirected graphs, this includes any edge that touches the node; for directed graphs it only includes edges *leaving* the node (i.e. the node is the edge's start node).

```
class Graph {
private:
  ...
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
}
```

Convenience method that lets you toss a Graph object into an output stream and get some useful output.

Notice this is a 'friend' function, so it is not defined in the Graph namespace. We're just saying to the compiler that it is cool with us if this function uses the Graph's private member variables.

```
class Graph {
private:
   ...
public:
   ...
   void clear();
   void tick(string message);
   void dfs(Node& start);
   void dfs(Node& start, Node& target);
   void bfs(Node& start);
   void bfs(Node& start, Node& target);
}
```

Resets all nodes to have WHITE color, with -1 discovery and finish times and rank. Resets all edges to type UNDISCOVERED_EDGE. Resets the clock to 0.

The testing code will call this when necessary. **Don't** call this yourself (e.g. when starting a new search).

```
class Graph {
private:
  ...
public:
  ...
  void clear();
  void tick(string message);
  void dfs(Node& start);
  void dfs(Node& start, Node& target);
  void bfs(Node& start);
  void bfs(Node& start, Node& target);
}
```

OPTIONAL debugging method. Use this after every time you increment the clock. The clock should increment every time a Node color changes to GRAY or BLACK. You might pass in a string to describe what just happened. If you are having trouble, consider using this function as a place to track your algorithm's progress. Print out your graph on each tick. If you want to get very fancy, you can output your graph in 'dot' format. This is the same format we used in the B-Tree assignment for debugging.

This function is 100% optional, though.

```
class Graph {
private:
  ...
public:
  ...
  void clear();
  void tick(string message);
  void dfs(Node& start);
  void dfs(Node& start, Node& target);
  void bfs(Node& start);
  void bfs(Node& start, Node& target);
}
```

Run a depth-first search from the indicated start node, and explores all reachable nodes. This ignores unreachable nodes. When this function returns, all explored nodes are colored BLACK, all unreachable nodes are WHITE. All explored nodes have correct discovery/exit time information. All edges have correct edge types (unfollowed edges should remain UNDISCOVERED).

For a DFS, mark nodes GRAY when we first discover them, and BLACK when we exit (finish) them.

```
class Graph {
private:
  ...
public:
  ...
  void clear();
  void tick(string message);
  void dfs(Node& start);
  void dfs(Node& start, Node& target);
  void bfs(Node& start);
  void bfs(Node& start, Node& target);
}                   ↑
```

Run a breadth-first search starting from the indicated node. This sets the 'rank' value on all nodes to something appropriate: -1 for unreachable nodes, 0 for the start node, 1 for nodes that are one edge from the start node, and so forth.

For a BFS, mark nodes GRAY when they are enqueued, and BLACK when they are dequeued.

```
class Node {
private:
  string data;
  int color;
  int discovery_time;
  int completion_time;
  int rank;
  Node* predecessor;
public:
  Node(string s);
  ~Node();
  string getData();
  void setData(string s);
  void setRank(int rank);
  friend std::ostream &operator
     << (std::ostream& out, Node node);
  ...
}
```

This is the part of the Node class that is done for you. You don't need to worry about setting the 'data' variable, but you are responsible for setting (or clearing) the other variables.

```
class Node {
private:
  ...
public:
  ...
  void clear();

  void setColor(int search_color, int time);

  void getDiscoveryInformation
    (int& color, int& disco_time,
    int& finish_time, int& bfs_rank);

  bool isAncestor(Node& other);
  void setPredecessor(Node& other);

}
```

Implement these functions. 'clear' resets the private variables to a default state. 'setColor' sets the node's discovery state and (for DFS) sets the time.

```
class Node {
private:
  ...
public:
  ...
  void clear();

  void setColor(int search_color, int time);

  void getDiscoveryInformation
    (int& color, int& disco_time,
    int& finish_time, int& bfs_rank);

  bool isAncestor(Node& other);
  void setPredecessor(Node& other);

}
```

The 'getDiscoveryInformation' uses four pass-by-reference variables that you should set to the appropriate values. While this function technically *returns* void, it is used to access four state variables at one time.

```
class Node {
private:
  ...
public:
  ...
  void clear();

  void setColor(int search_color, int time);

  void getDiscoveryInformation
    (int& color, int& disco_time,
     int& finish_time, int& bfs_rank);

  bool isAncestor(Node& other);
  void setPredecessor(Node& other);
}
```

These functions give us information about the *spanning tree*. This is the tree formed by the particular order we discover nodes in a depth-first search. A predecessor is the node we were on when we discovered a node. An ancestor is a predecessor, or any of the predecessor's predecessors.

```
class Edge {
private:
  Node* a;
  Node* b;
  int type;
public:
  Edge(Node& n1, Node& n2);
  ~Edge();
  int getType();
  Node* getStart();
  Node* getEnd();
  friend std::ostream &operator
    << (std::ostream& out, Edge edge);

  void setType(int edge_type);
};
```

The Edge class is mostly done for you. All you need to implement is the 'setType' function (and that should be one line long).

Breadth-First Search Pseudocode

```
bfs(node):
  reset queue Q
  add node to Q
  mark node gray
  while Q has elements:
    n = Q.dequeue()
    mark n black
    E = edges related to n
    for all edges e in E:
      a = other end of e
      if a is white:
        add a to Q
        mark a gray
```

Depth-First Search Pseudocode (recursive version)

```
dfs(node):
  mark node gray
  E = edges related to node
  for all edges e in E:
    a = other end of e
    if a is white:
      dfs(a) ; recurse!
  mark node black
```

Depth-First Search Pseudocode (iterative version)

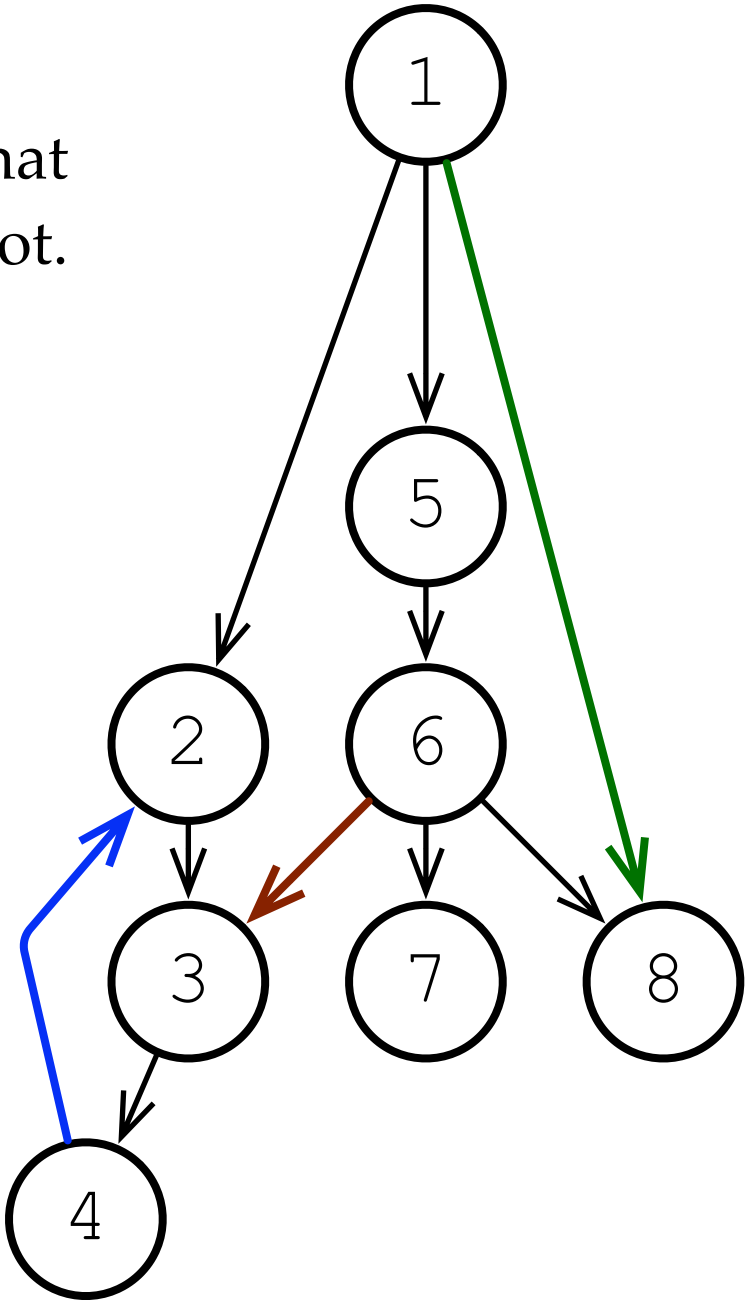
```
dfs(node):
  ;; Stack S must be cleared before
  ;; this is called.
   mark node gray
  S.push(node)
  while S has elements:
    n = S.peek() ;; doesn't remove
    E = edges related to n
    pushed = false
    for all edges e in E {
      if e is already classified, continue to next edge
      a = other end of e
      if a is white:
        mark a gray
        S.push(a)
        pushed = true
        break out of for-loop
    }
    if pushed is false:
        mark n black
        S.pop() ;; removes n
```

What about edge types? We can make a *spanning tree* which describes the order that we discover nodes starting from some root.

```
Black arrows = Tree Edges
Green arrows = Forward Edges
Red arrows   = Cross Edges
Blue arrows  = Back Edges
```

This graph was shamelessly stolen from Wikipedia.

You'll have to augment the pseudocode given earlier to incorporate edge types.



Undiscovered nodes are **white**. When we mark a node **gray**, that's when we discovered it. When we mark a node **black**, that's when we finished it.

When we're on a node (like node 5) and we discover a white node (like node 6), we set the white node's predecessor to the one we're on. So, 5 is 6's predecessor. These relations are always associated with tree edges (the black arrows).

Ancestors are any node that can be found by following the predecessor chain. So, node 1 is everybody's ancestor; 2 is an ancestor to 4, but 4 is not an ancestor to 2.