SWEN 601
Software Construction

•••

Graphs, & Breadth-First Search

Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
   a. *The project **may** already contain some code!*
2. Create a `session` package. This is where you will write your solutions to today's activities.
3. Create a `homework` package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

Do not submit code that ***does not compile***. Comment it out if necessary.

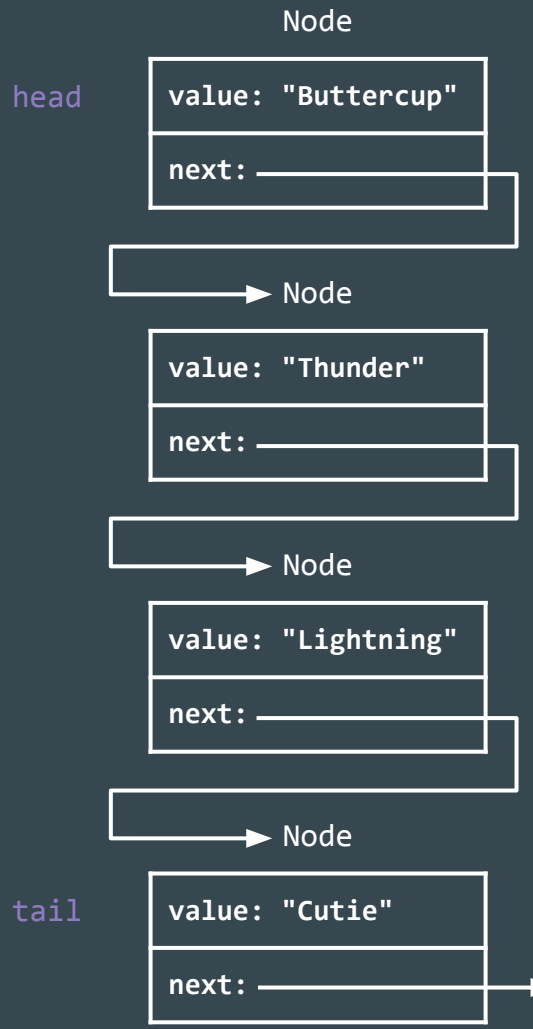
Next Two Weeks

WEEK 11	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #16		Quiz #17		
LECTURE			Graphs & BFS		DFS & Dijkstra's Algorithm		
HOMEWORK	Hwk 16 Due (**11:30PM**)		*Hwk 17 Assigned*		*Hwk 18 Assigned*	Hwk 17 Due (**11:30pm**)	

WEEK 12	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #18				
LECTURE			Dijkstra's Algorithm		Practicum 3		
HOMEWORK	Hwk 18 Due (**11:30pm**)		*Hwk 19 Assigned*				

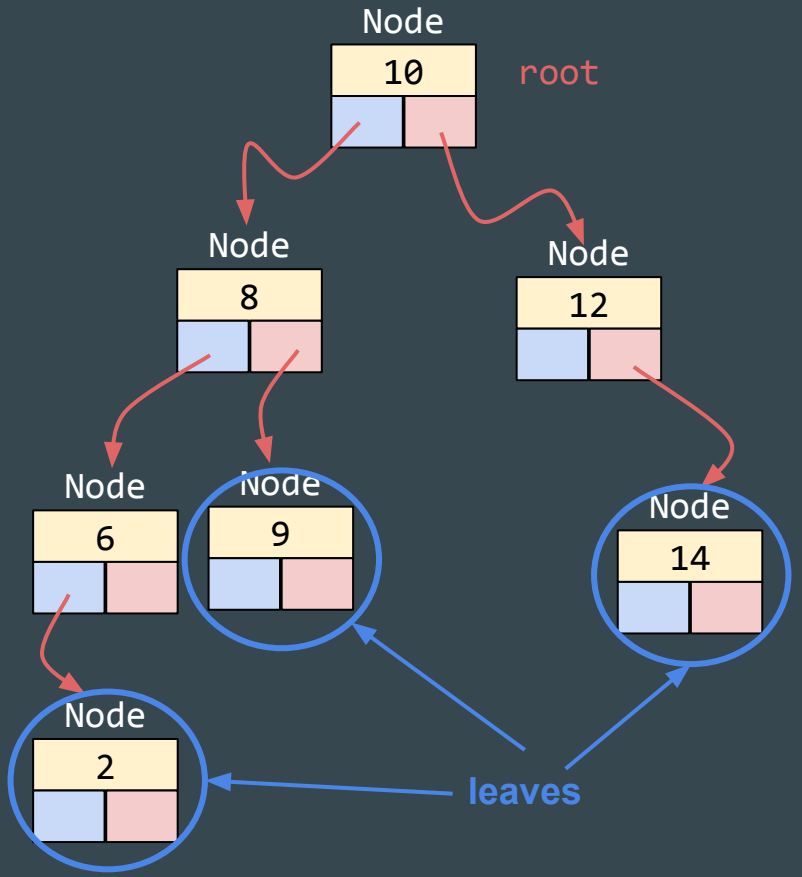
Node-Based Structures

- So far this semester, we have spent a lot of time talking about ***nodes***.
- As you know, a Node comprises two parts:
  - A ***value*** of some generic type.
  - A ***reference*** to the next Node(s) in the sequence.
- We have discussed many simple ***node-based structures*** including ***lists***, ***stacks***, and ***queues***.
- A ***linked list***, for example, is one of the following:
  - The empty list (`size=0`, `head=null`, `tail=null`).
  - A non-empty list:
    - The `head` refers to the first node.
    - The `tail` refers to the last node.
    - The `size` is the number of nodes in the list.



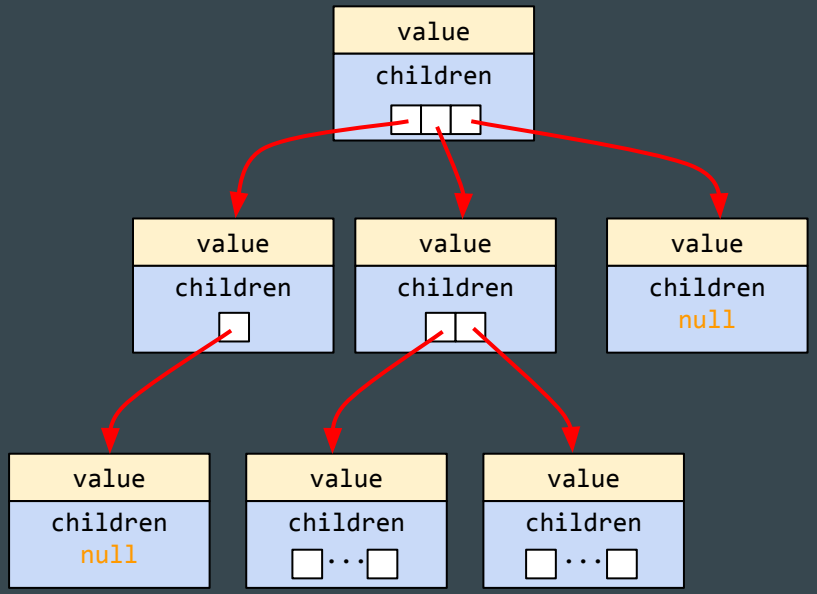
Binary Trees

- You should remember that a ***binary tree*** is a slightly more complex node-based structure.
- A binary tree is one of the following:
  - The ***empty tree*** (`null`).
  - The ***non-empty tree*** comprising at least one ***binary node***. Each binary node contains:
    - A ***value*** of some generic type.
    - A ***left subtree*** that may be empty.
    - A ***right subtree*** that may be empty.
- A ***binary search tree*** is a special Binary Tree.
  - The left subtree of each BinaryNode contains only values that ***come before*** the node's value.
  - The right subtree of each BinaryNode contains only values that ***come after*** the node's value.
- The ***root*** of the tree is the one node that does not have a parent.
- Nodes without children are called ***leaves***.



N-Ary Trees

- Each Node in a Binary Tree has exactly ***two*** subtrees (***left*** & ***right***).
  - Either or both tree may be empty.
- An N-ary Tree may have ***any number*** of subtrees.
  - This means that an N-ary Tree must use ***another*** data structure (e.g. a ***list***) to store its subtrees.
- An N-ary Tree may be one of following:
  - The empty tree (no nodes).
  - At least one Node with:
    - A ***value*** of some generic type.
    - ***Zero or more*** children, each of which is the root of a subtree.
- As with binary trees, nodes that do not have children are ***leaves***.

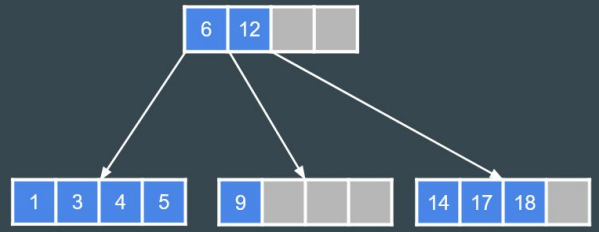


All of the node-based structures that we have discussed this semester are N-ary trees.

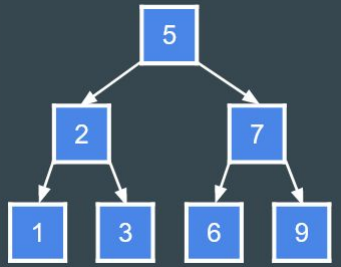
Lists are N-ary trees with 1 child, and binary trees are N-ary trees with two children.

B-Trees

- A ***B-Tree*** is an N-ary tree that is very similar to a ***binary search tree***. The major differences are:
  - Each node may have ***more than one value***, which we call ***keys***.
  - Each node may have ***more than two children***.
- The ***order*** of a tree determines the ***maximum number of children*** that a node can have.
  - In a ***binary tree*** the order is 2.
  - In a B-Tree, the degree can be ***greater than 2***.
  - We use ***m*** to represent the order of a tree.
- Each node in a B-Tree may have ***up to m-1 keys***.
  - We us ***k*** to represent the number of keys in a node.
  - The keys are ***sorted***.
  - Each node may have up to ***k+1*** children.
  - Like a binary search tree, keys are added to children ***from left to right*** based on their value relative to the parent.



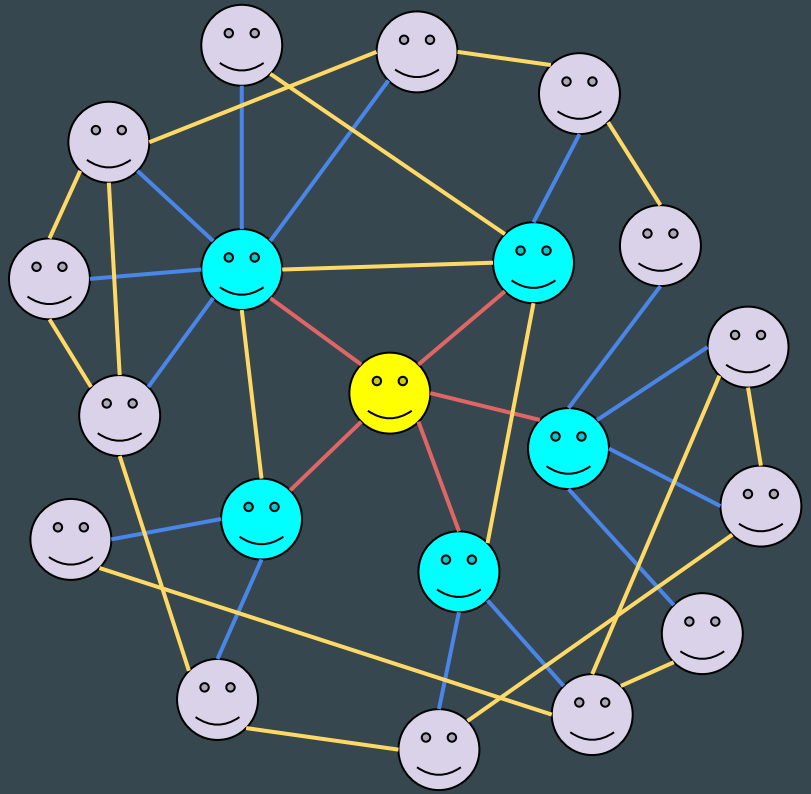
B-Trees will be the most complex data structure that we have talked about so far this semester.



Fun Fact: a binary search tree ***is*** a B-Tree with ***m=2***, and a ***k=1***.

Problem: A Social Network

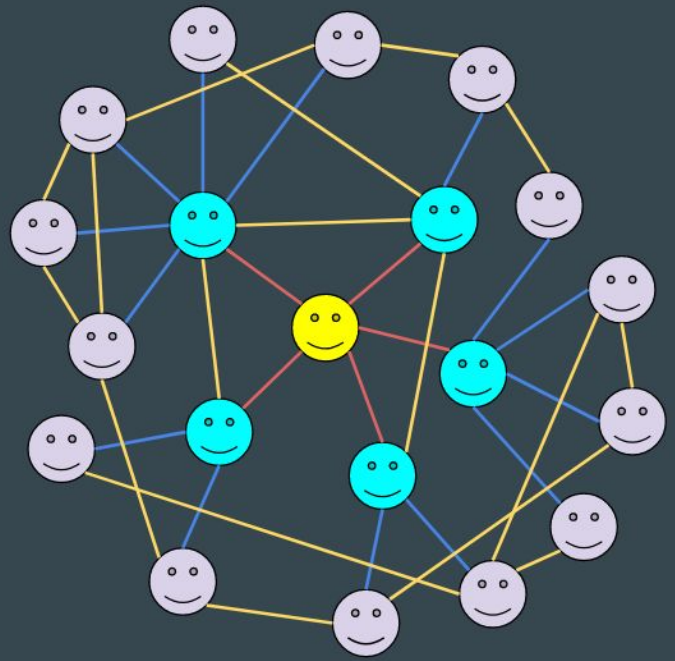
- Consider your account on the social network of your choice.
  - There is you...
  - ... and your followers...
  - ... and their followers.
- What data structure would you use to represent this?
  - Perhaps an N-ary tree?
- Now consider that some of your followers follow each other.
- Will the N-ary tree be sufficient?
  - No; in a tree, children are not connected to each other.



We will need a new kind of data structure to represent the social network: a ***graph***.

Graphs

- Like lists and trees, a ***graph*** is a linked-Node structure.
- A graph may be:
  - The ***empty graph*** (no nodes)
  - At least ***one node*** with:
    - A ***value*** of some generic type.
    - A ***list*** of nodes to which it is ***connected***.
- In this way, a Graph may seem very similar to an N-ary Tree, but the same rules do not apply: ***any*** node in a Graph may be connected to ***any other*** node.
- Graphs also use special terminology.
  - Each Node is referred to as a ***vertex***.
  - Connections between ***vertices*** are called ***edges***.
  - Two vertices connected by an edge are called ***neighbors***.



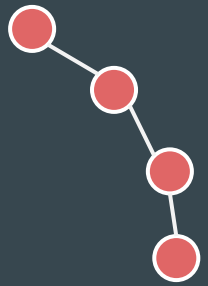
Let's practice identifying whether or not a data structure is a Graph.

Activity: Graph Identification

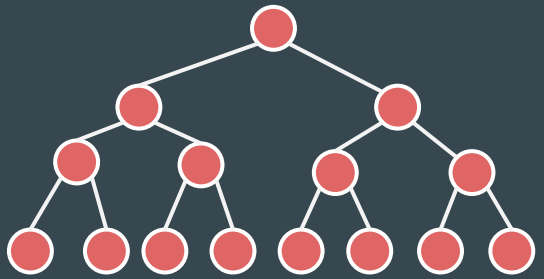
Look at the diagrams below. Vertices are represented as circles, and edges are lines. Which of these represents a graph?

Yes. A graph may only include a single vertex.

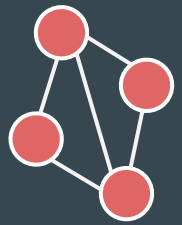
Yes. A graph may include only a single edge.



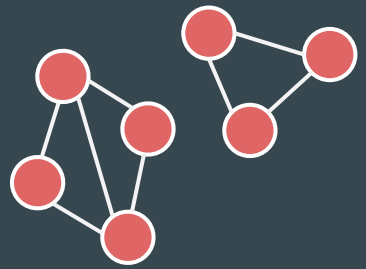
Yes. A linked list is a kind of graph.



Yes. A binary tree is a kind of graph.



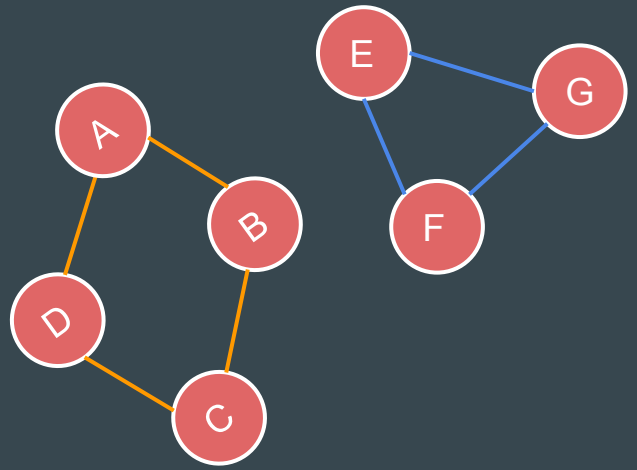
Yes. This graph includes 4 vertices and 5 edges.



Yes. There does not have to be a path from any two vertices.

Connected Components

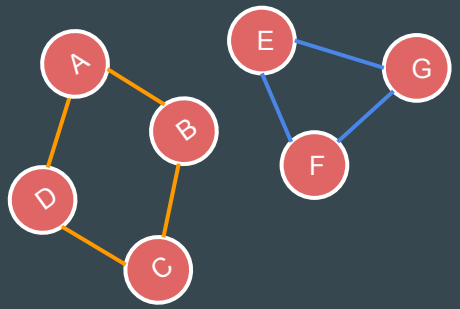
- An edge may be identified by the two vertices that it connects.
  - For example, the edge connecting vertex A to vertex B may be identified as E_{AB}.
- A path is a series of edges that connect two vertices together.
  - For example, the the path from A to C comprises the edges E_{AB} and E_{BC}, or perhaps E_{AD} and E_{DC}.
- If a path exists between two vertices in a graph, they are said to be part of the same ***connected component***.
- A graph may include ***zero or more*** connected components.
  - In the example to the right, there is ***no path*** from A to F, and so they are ***not*** part of the same connected component.



An algorithm that determines whether or not a path exists between two nodes in a graph is called a ***search***.

We will discuss *three* such algorithms this semester.

Activity: A `Graph` Interface I



Write a `Graph` interface. Remember that:

- It should be ***generic***; include a type parameter in the class declaration.
- You should be able to `add` a value of the generic type to the graph.
- You should be able to `connect` two values together as neighbors.
  - In fact, you should be able to easily connect one value to any number of neighbors...
- You should be able to get the `size` of the graph (i.e. the number of vertices).

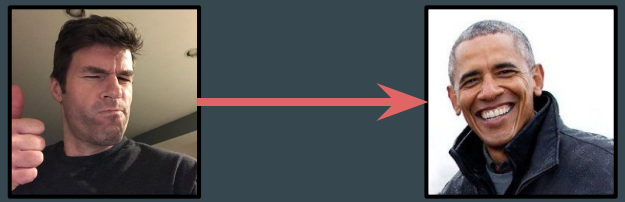
Directed vs. Undirected Graphs

Consider two friends on Facebook.

One friend sends the other a friend request. Once accepted, they can ***both*** see each other's timelines.

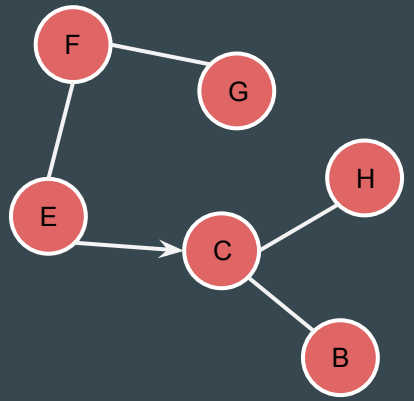
If we think of the friends as vertices in a graph, the edge between them is ***undirected***; it works in both directions.

Now consider Twitter.



One user may choose to follow another so that they can see that person's tweets.

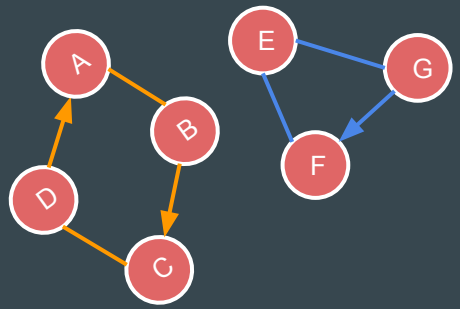
However, the second user does not see the first person's tweets. The edge between the two users is ***directed***; it only works in one direction.



Directed edges are represented in a graph by an ***arrow*** on one end or the other of the edge.

In the example above, $\mathbf{E}_{\mathbf{EC}}$ is directed; the edge can be traversed from E to C but not back. The other edges in the graph are undirected.

Activity: A `Graph` Interface II

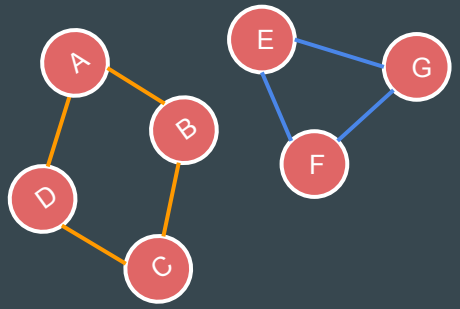


The vertices within a `Graph` may be connected with edges that are directed ***or*** undirected.

- Rename your existing connect method(s) to indicate that the connection will be ***directed***.
- Add two new methods to establish ***undirected*** connections.

Graphs as Adjacency Lists

- There are a number of different ways to represent a graph. One is referred to as an ***adjacency list***.
  - Each vertex keeps track of the other vertices to which it is connected in a ***data structure***, e.g. a ***list***.
- An adjacency list can be diagrammed using a simple table.
  - The first column lists all of the vertices in the graph.
  - The second column lists the vertices to which each vertex is connected (its ***neighbors***).



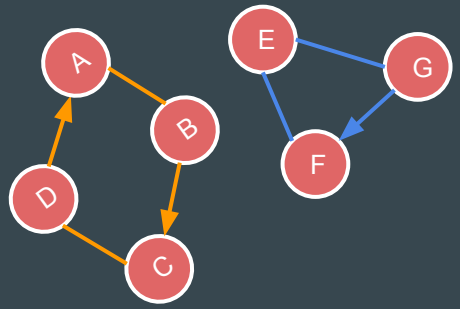
Vertex	Adjacency List
A	B, D
B	A, C
C	B, D
D	A, C
E	F, G
F	E, G
G	E, F

Activity: A `Vertex` Class

We will be creating an adjacency list representation of a graph. Begin by creating a class to represent a single `Vertex`. Remember that:

- Each `Vertex` must contain a value of some generic type.
- Each `Vertex` must also have an adjacency list of neighbors. What is the best data structure to use?
  - Hint: you can't connect a `Vertex` to the same neighbor more than once.
- How will you add neighbors?

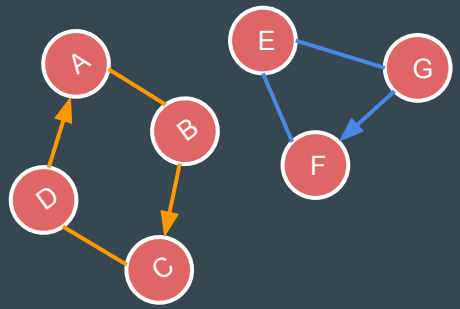
Activity: An `AdjacencyGraph` Class



Create a new class named `AdjacencyGraph` that implements your `Graph` interface.

- What data structure will you use to store the vertices?
  - Each value will be unique.
  - It is important that, given a value, you can quickly determine whether its vertex is already in the graph.
  - You will also need to quickly fetch the vertex for a given value (e.g. so that you can connect it to its neighbors).
- Implement the various methods of your class using your chosen data structure.
  - Each should be very simple and comprise fewer than 5 lines of code.

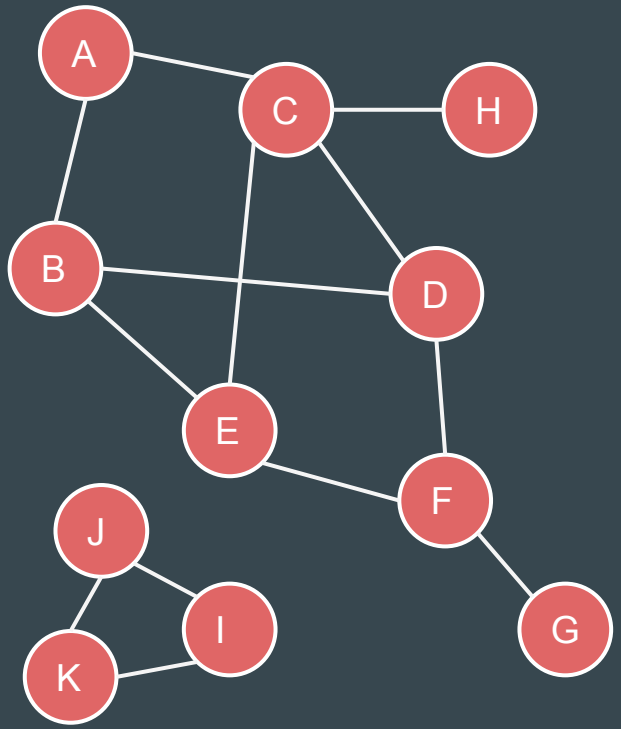
Activity: Making a Graph



Add a main method to your `AdjacencyGraph` class.

- Make a new, empty graph.
- Build the graph depicted to the right.
  - Note that some of the edges are directed, while others are undirected.

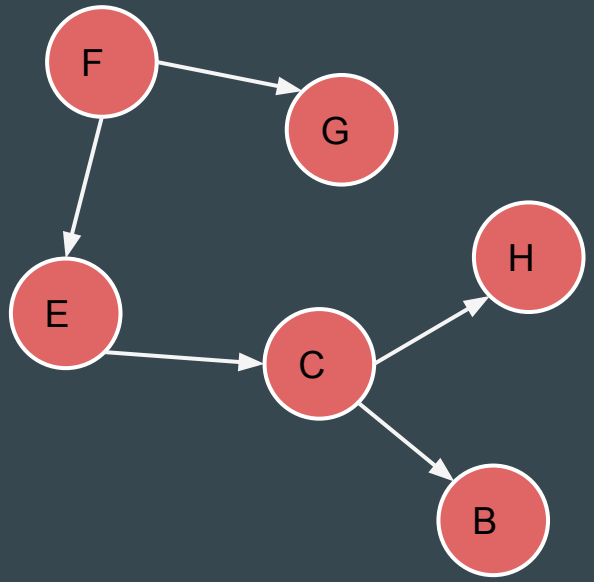
A Simple Example



- Is there a ***path*** from A to G in this graph?
- Of course there is! You can see that there are actually ***multiple*** possible paths from A to G. For example:
  - E_{AC}, E_{CD}, E_{DF}, E_{FG}
  - E_{AB}, E_{BE}, E_{EC}, E_{CD}, E_{DF}, E_{FG}
- You can also state with certainty that there is no path from A to K.
- This is easy when the graph is ***small*** and you (a ***human***) have the ability to look at the ***entire graph*** all at once.
- But how would an ***algorithm*** find a path?

Breadth-First Search (BFS)

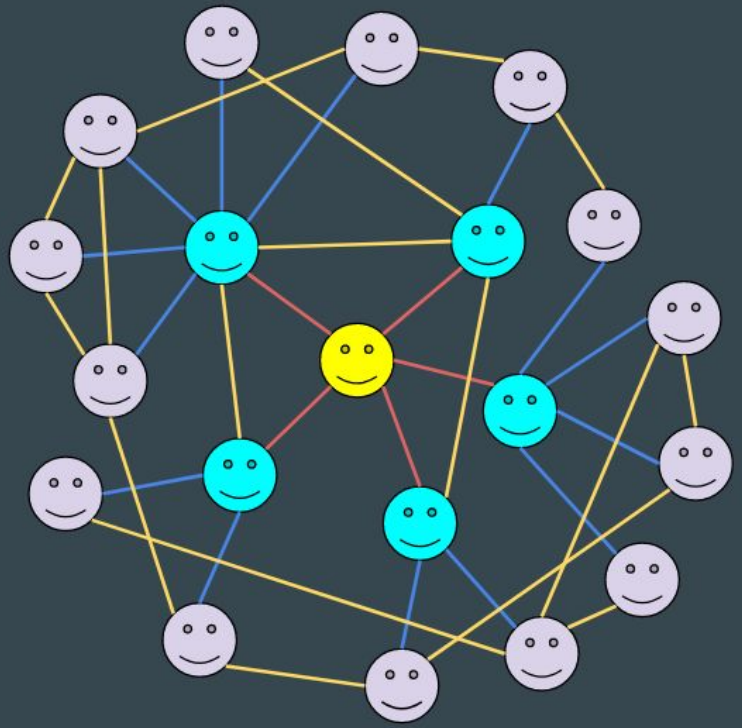
- Given some starting vertex S and ending vertex E, we'd like to determine if a path exists between the two.
- Let's try this:
  - Create a ***queue*** and add S to it.
  - As long as the queue is not ***empty***:
    - ***Dequeue*** the next vertex.
    - If it is E, you found a path! Return `true`.
    - If it is not E, add its neighbors to the queue.
  - If the queue is ***empty***, there is no path. Return `false`.
- Let's try it to find a path from F to B in the graph to the right using the queue to the right.



Vertex	F	E	G	C	B	H
Next	^	^	^	^	^	

Cycles

- Imagine that you are traversing a graph, following edges from one vertex to the next.
- As you continue along your path, you eventually arrive ***back*** at the node from which you ***started***.
- Such a path is called a ***cycle***, and will cause BFS to visit the same nodes ***many times over***.
  - In some cases, BFS may get ***stuck*** in a cycle ***forever***.
- We will need to modify our BFS algorithm so that it keeps track of vertices that were seen *previously*.
  - Only ***new*** vertices are added to the queue.
- What is the best data structure to quickly determine whether or not we have seen a vertex previously?

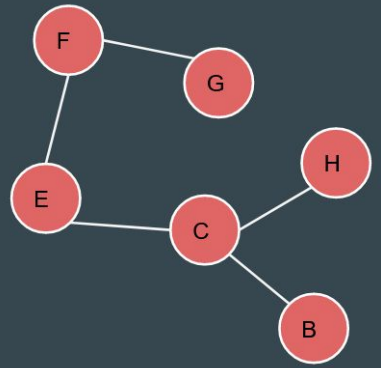


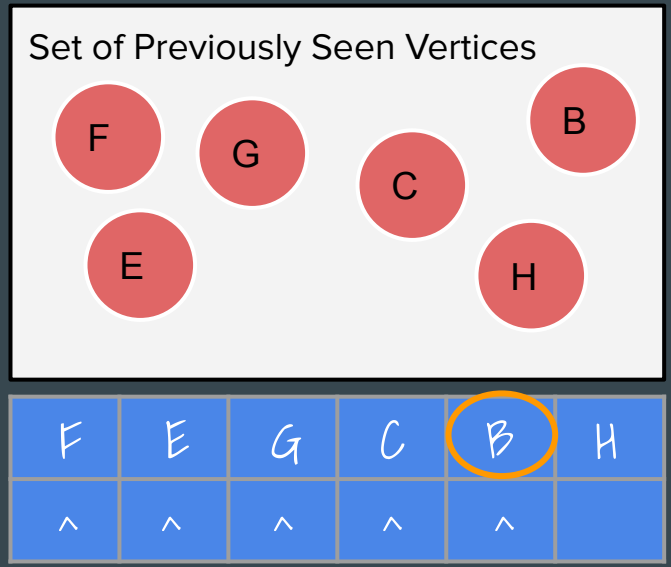
Not all graphs contain cycles. Those that do are ***cyclic***. Those that do not are ***acyclic***.

NEW AND IMPROVED

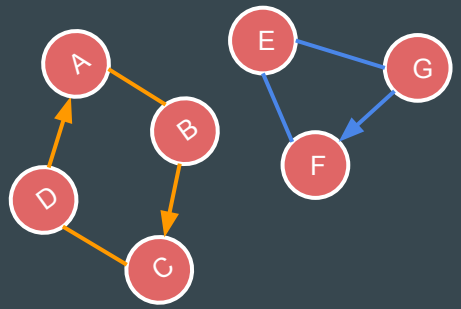
Breadth First Search (BFS)

- Given some starting vertex S and ending vertex E, we'd like to determine if a path exists between the two.
- Let's try this:
  - Create a queue and a ***set*** and add S to ***both***.
  - As long as the queue is not empty:
    - Dequeue the next vertex.
    - If it is E, you found a path! Return `true`.
    - If it is not E, for each of E's neighbors:
      - If it is not in the ***set***, add it to ***both*** the ***set*** and the queue.
  - If the queue is empty, there is no path. Return `false`.
- Let's try it to find a path from F to B in the graph to the right using the queue ***and*** the set.





Activity: Implement BFS



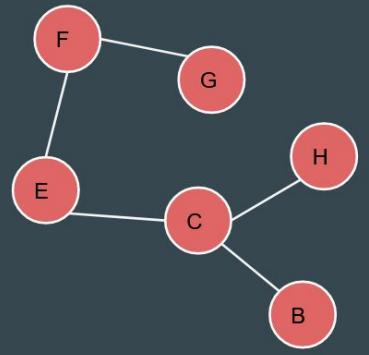
Modify your `Graph` interface to include a new method:

- `boolean breadthFirstSearch(T start, T end)`

Implement the method in your `AdjacencyGraph` class so that it implements BFS to determine whether or not a path exists between the two values in the graph. It should return `true` if the path exists, and `false` otherwise.

Building a Path with BFS

- BFS determines whether a path ***exists*** but not what the path ***actually is***.
  - It's useful to know that we can get from ROC to LAX, but we can't book the trip if we don't know through which airports we must fly.
- So let's modify BFS once again, this time so that it keeps track of ***all*** of the vertices in the ***path*** between S and E.
- Instead of keeping previously seen nodes in a set, let's keep them in a ***map***.
- When we dequeue a vertex ***V***, for each of its neighbors ***N*** that is not already a ***key*** in the ***map***, add ***N*** (the ***key***) and ***V*** (the ***value***) to the ***map***.
  - Each vertex N keeps track of is ***predecessor***, the vertex V through which N was ***initially found***.

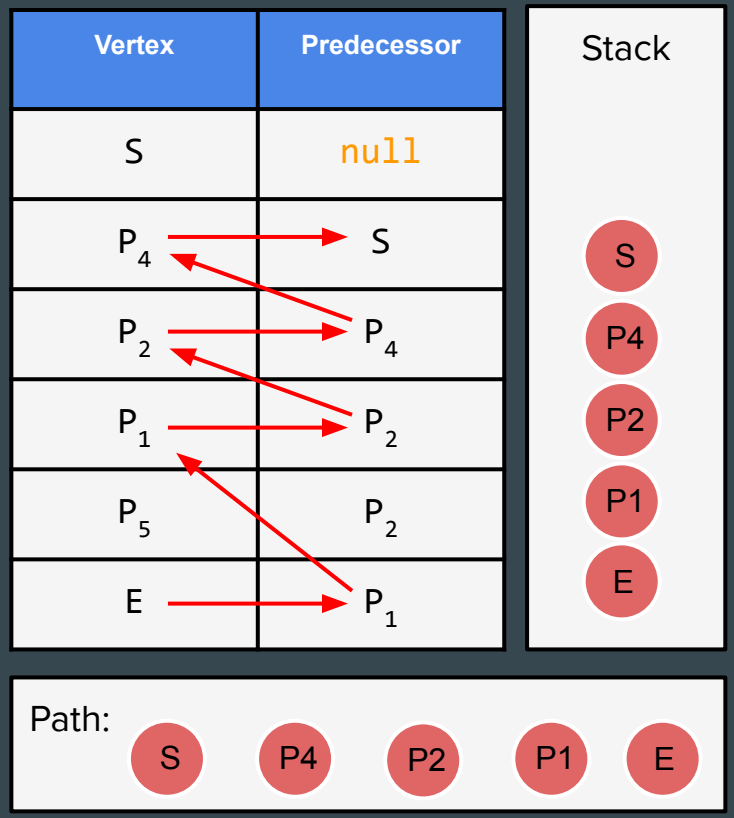


Vertex	Predecessor
F	null
E	F
G	F
C	E
B	C
H	C

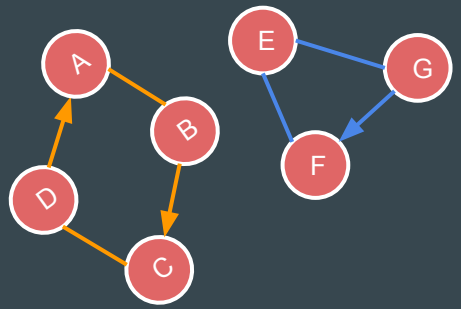
Vertex	F	E	G	C	B	H
Next	^	^	^	^	^	^

Making a Path

- So now that we have a map of each vertex and its predecessor, how do we know which vertices are along the path from S to E?
- We start at the end of the path, E, and use it as a key to look up its predecessor, P1, and push it onto a ***stack***.
- We then use P1 to look up its predecessor, P2, and add it to the stack.
- And so on, all the way back to S.
- We then build the path by popping all of the vertices off of the stack.



Activity: Implement BFS Path



Modify your `Graph` interface to include a new method:

- `List<T> breadthFirstPath(T start, T end)`

Implement the method in your `AdjacencyGraph` class so that it implements BFS to return the path of values from the start to the end, if it exists, or `null` otherwise.

- The path should be a list of values from start to end.
- The algorithm uses a stack to build the path in reverse. Is there a better way?