GCIS-124
Software Development & Problem Solving

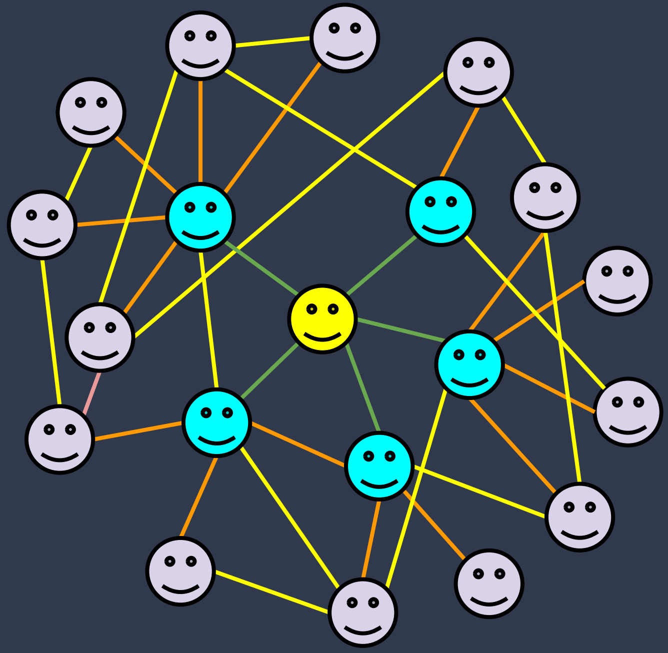
7: Data Structures III

RIT | Golisano College of Computing and Information Sciences

SUN	MON (2/26)	TUE	WED (2/28)	THU	FRI (3/1)	SAT
Unit 6: Data Structures II					Unit 7: Data Structures III	
			Assignment 6.1 Due (start of class)		Unit 6 Mini-Practicum Assignment 6.2 Due (start of class)	

SUN	MON (3/4)	TUE	WED (3/6)	THU	FRI (3/8)	SAT
	Unit 7: Data Structures III				Midterm	
			Assignment 7.1 Due (start of class)		Midterm Exam 2 (Units 4-6) Written Practical	

Graphs



A ***social network*** is a kind of graph that includes people who are connected based on whether or not they follow each other on the network.

- In this unit we will learn about a new kind of data structure: ***graphs***.
  - A graph comprises ***vertices***, each of which holds some kind of data.
  - Two ***vertices*** may be connected by an ***edge***. If so, they are said to be ***neighbors***.
  - Graphs can be used to model complex data sets representing entities with lots of connections between them including computer networks, airports, highway systems, and so on.
- We will be exploring several different aspects of graphs, including:
  - Graph Terminology
  - The Graph ADT
  - Implementing a Graph using Adjacency Lists
  - Breadth-First Search & Path
  - Depth-First Search & Path
- Today we will specifically be focusing on ***graph terminology*** and implementing a graph using ***adjacency lists***.

Review: Abstract Data Types

- An ***abstract data type*** (***ADT***) ***defines*** the behavior of a data structure from the perspective of its user, but ***does not*** provide any implementation details.
- So far in this course we have explored many different ADTs, including:
  - Stacks
  - Queues
  - Lists
  - Binary Trees/Binary Search Trees
  - Heaps
  - Maps (Dictionaries)
  - Sets
- We have also discussed implementing several of these abstract data types in more than one way, for example:
  - ***Node-based*** vs. ***array-based*** queues and lists.
  - Maps that use ***chaining*** vs. ***open addressing***.
- We have also been introduced to the ***Java Collections Framework*** (***JCF***), including:
  - Java's implementations of many data structures.
  - `Iterable` & `Iterator`
  - `Comparable` & `Comparator`
  - `Collection` & `Collections`

By now we all understand that, not only can each abstract data type be implemented in ***more than one way***...

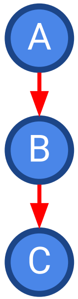
...but that each implementation has different strengths and weaknesses under different circumstances, e.g. ***lists***...

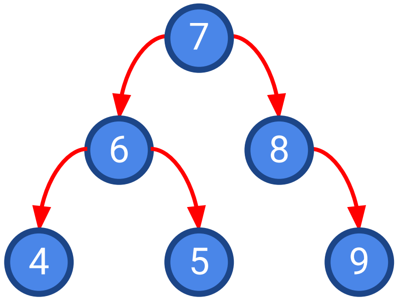
If you...	- Always Append at the End - Never Insert or remove in the middle - Need Random Access - Know the maximum number of elements	- Always Insert or Remove at head and tail - Don't need random access - Don't know the maximum number of elements
...then choose:	Array List	Linked List

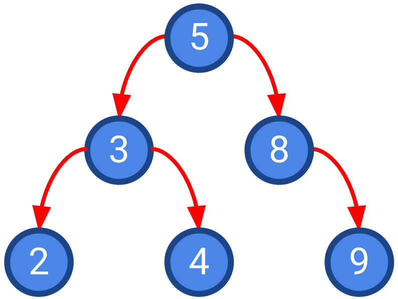
Understanding which operations are needed ***the most frequently*** will help you choose the most efficient implementation to solve a specific problem.

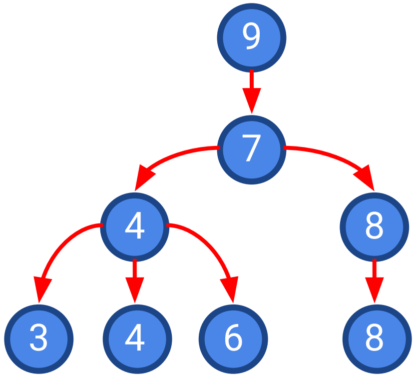
7.1 Identifying Data Structures

Lots of different data structures can be created by connecting nodes together. We have discussed several of them over the course of the past two units. Examine each picture below and indicate the type of data structure that is being represented. Be as specific as possible with the information given.



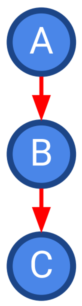




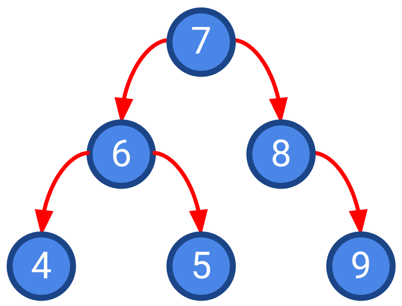


7.1 Identifying Data Structures

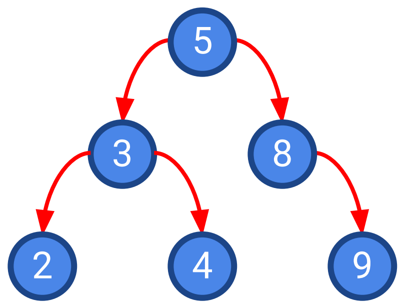
Lots of different data structures can be created by connecting nodes together. We have discussed several of them over the course of the past two units. Examine each picture below and indicate the type of data structure that is being represented. Be as specific as possible with the information given.



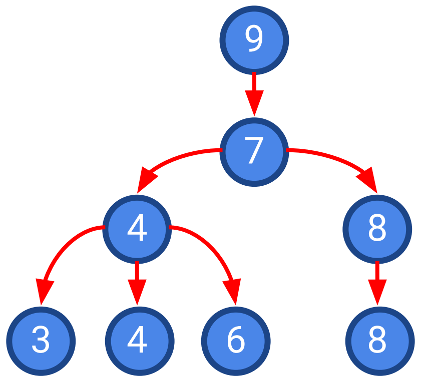
LINKED SEQUENCE
(LIST, STACK, QUEUE?)



BINARY TREE
(5 < 6)

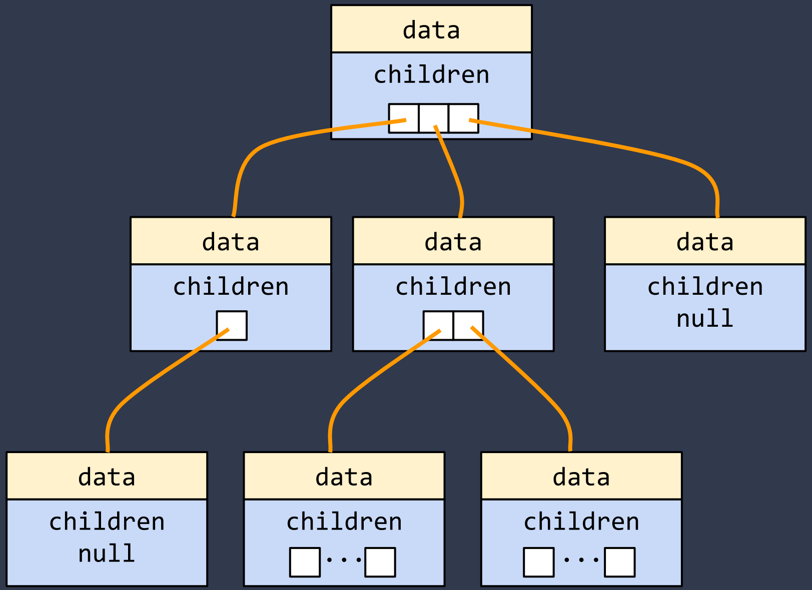


BINARY SEARCH TREE



N-ARY TREE

Node-Based Structures



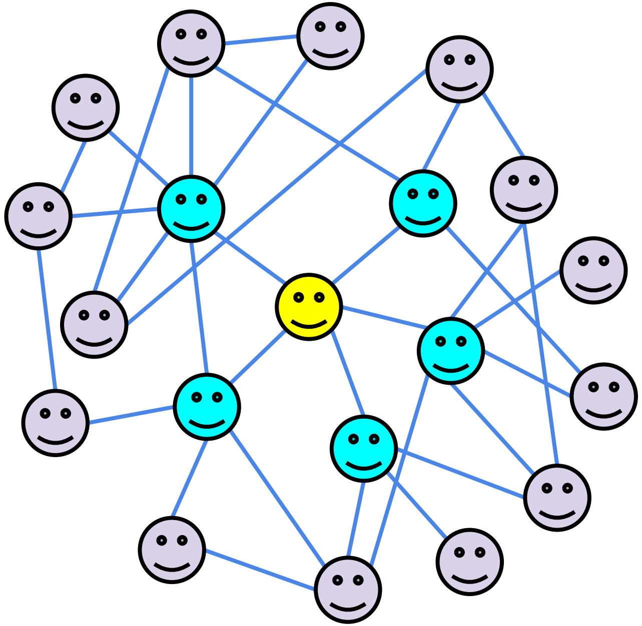
Because the nodes in an ***N-ary Tree*** may contain ***zero or more*** children, children must be stored in a data structure like a ***list***.

All of the other node-based data structures that we have examined are simply ***specialized versions*** of an N-ary tree.

- So far in this course we have spent a lot of time talking about ***nodes***.
- As you recall, a node comprises two parts:
  - A ***value*** of some generic type.
  - A reference to the ***next*** node(s).
- We have discussed many simple ***node-based*** implementations of ADTs including ***stacks***, ***queues***, ***lists***, and ***binary trees***.
- A ***linked list***, for example comprises a ***head*** node and a ***tail*** node, either of which may be `null` (if the list is empty).
- The nodes in a ***binary tree*** are slightly more complex.
  - Each node has a ***value***, a ***left child***, and a ***right child***, either or both of which may be null.
- A ***binary search tree*** (***BST***) is a special kind of binary tree with some extra rules:
  - The values in a non-empty ***left subtree*** are always ***less than*** the value in the node.
  - The values in a non-empty ***right subtree*** are always ***greater than*** the value in the node.
- Of course not all trees limit nodes to exactly two children; the nodes in an ***N-ary tree*** may contain ***zero or more*** children.

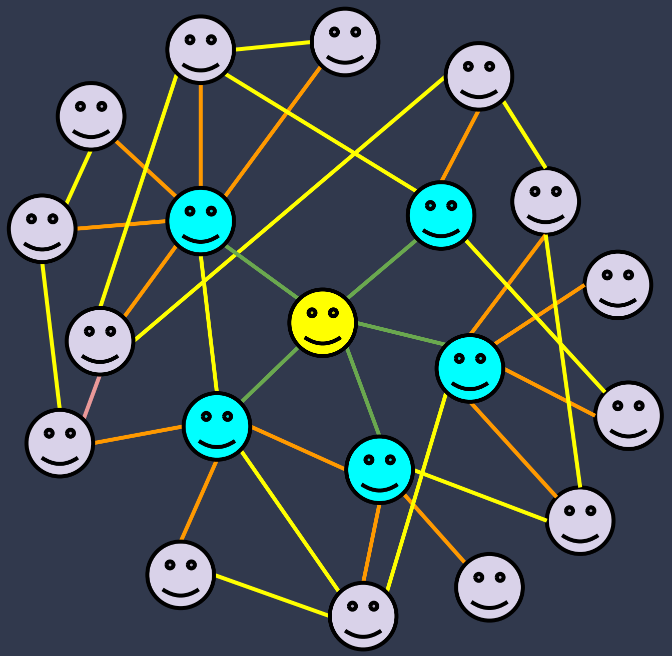
A Social Network

- Consider your account on the ***social network*** of your choice.
  - There is ***you***...
  - ...and ***your followers***...
  - ...and ***their followers***.
- What kind of data structure would you use to represent this system?
  - How about an ***N-ary Tree***?
- Now consider that some of your followers ***follow each other***.
- Will an N-ary Tree be sufficient?
  - ***No!***
  - In a tree of any kind, children cannot be connected to ***each other***.
  - Using an N-ary Tree would mean that no two of your followers could ever follow each other.



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

Graphs

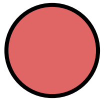


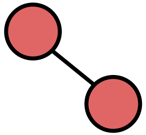
A ***social network*** is a great example of a graph. Each ***vertex*** represents a ***person*** in the network. The ***edges*** indicate whether or not the people ***follow*** each other.

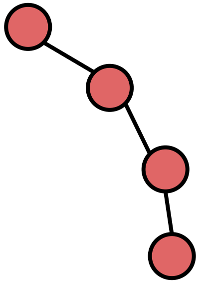
- Like lists and trees, a ***graph*** is conceptually a ***linked-node structure***.
  - The ***empty*** graph, containing no nodes.
  - A graph containing ***at least one*** node with:
    - A ***value*** of some type.
    - A ***list*** of nodes to which is is ***connected***.
- In this way, a graph may seem very similar to an ***N-ary Tree***, but the same rules do not apply.
  - There is no "***parent/child***" relationship between nodes in a graph.
  - ***Any*** node in the graph may be connected to ***any other*** node.
- Graphs also use special terminology.
  - Each node in the graph is referred to as a ***vertex***.
    - The plural of vertex is ***vertices***.
  - The connection between two vertices are called ***edges***.
  - Two vertices that are connected by an edge are called ***neighbors***.
- Graphs are an ideal data structure for representing real world systems with many-to-many connections between entities, e.g.
  - Computer networks
  - Cities connected by highways
  - Airports connected by airline routes
  - etc.

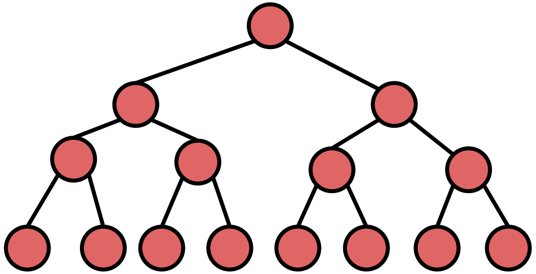
7.2 Graph Identification

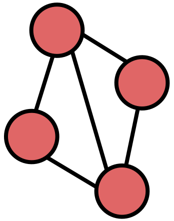
Graphs may come in any one of many possible configurations. Look at the diagrams below. Vertices are represented as circles, and edges are lines. Which of the diagrams below represents a graph?

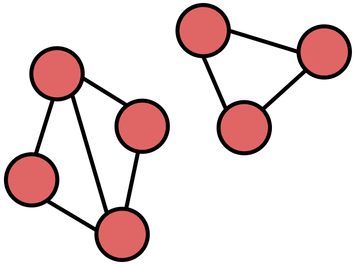






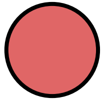




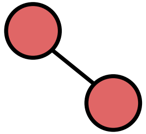


7.2 Graph Identification

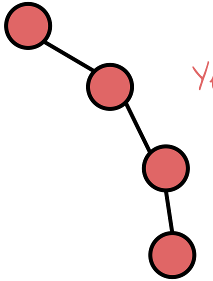
Graphs may come in any one of many possible configurations. Look at the diagrams below. Vertices are represented as circles, and edges are lines. Which of the diagrams below represents a graph?



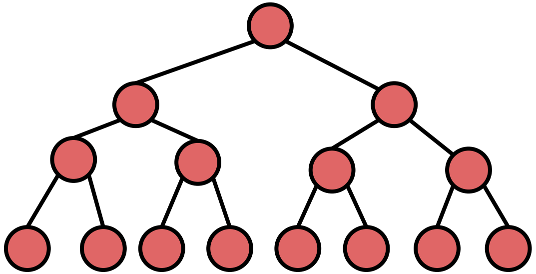
Yes. A graph may only include a single vertex.



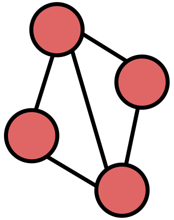
Yes. A graph may include only a single edge.



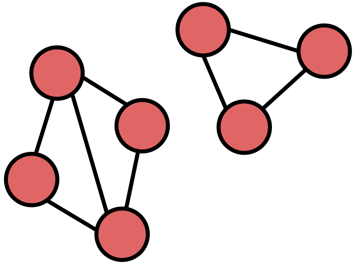
Yes. A linked list is a kind of graph.



Yes. A binary tree is a kind of graph.



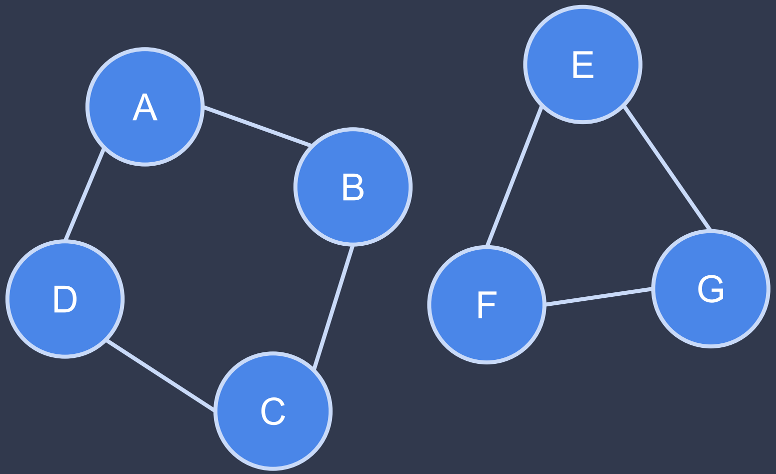
Yes. This graph includes 4 vertices and 5 edges.



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

Connected Components

- An ***edge*** may be uniquely 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 connects two vertices together.
  - For example, one possible path from ***vertex A*** to ***vertex C*** in the graph depicted to the right comprises the edges E_{AB} and E_{BC}.
- If a path ***exists*** between two vertices, they are said to be part of the same ***connected component*** within a graph.
- A graph may include ***any number*** of connected components.
  - In the graph depicted to the right, ***vertex D*** and ***vertex B*** are clearly part of the same connected component, because there is ***at least one path*** that connects the two.
  - On the other hand, ***vertex A*** and ***vertex F*** are clearly ***not*** part of the same connected component because there is ***no*** path connecting the two.

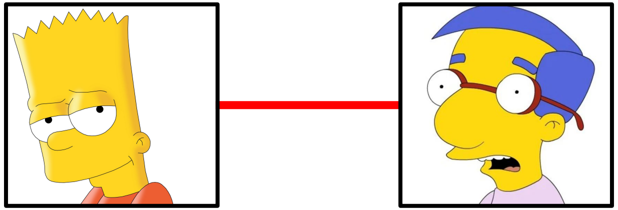


An algorithm that determines whether or not a path exists between two vertices in a graph is called a ***search***.

We will explore ***three*** such algorithms throughout this course.

Directed vs Undirected Edges

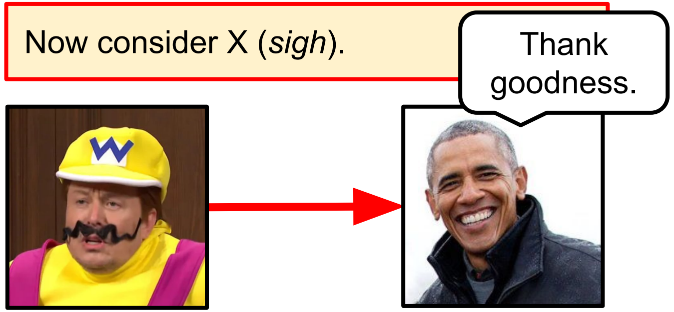
Consider two friends on Facebook.



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

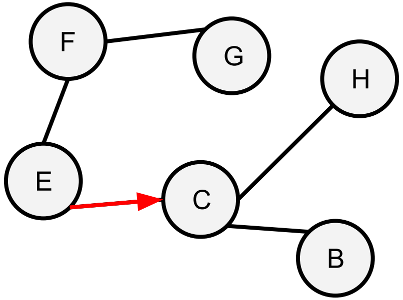
If we think of the friends as vertices in a graph, the edge between them is ***undirected***; it works in ***both*** directions.

Now consider X (*sigh*).



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 user's tweets. The edge between the two users is ***directed***; it only works in ***one*** direction.

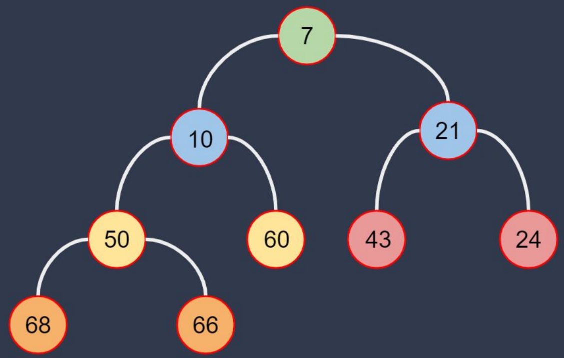


Directed edges are represented in a graph by an ***arrow*** on one end or the other of the edge.

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

The Graph ADT

We've seen many data structures that are easier to visualize ***conceptually*** using diagrams comprising ***nodes***...



...but by now we also understand that those same data structures may be implemented without using node ***objects***. Graphs are no exception!

7	10	21	50	60	43	24	68	66

- The graph abstract data type defines the behavior that a graph must include, without specifying any implementation details.
- A graph should provide at least the following behavior:
  - ***add(E value)*** - adds a value to the graph.
  - ***contains(E value)*** - returns `true` if the value is present in the graph, and `false` otherwise.
  - ***size*** - returns the number of values in the graph.
  - ***connectDirected(E a, E b)*** - creates a ***directed*** connection between the specified two values in the graph.
  - ***connectUndirected(E a, E b)*** - creates an ***undirected*** connection between the specified two values in the graph.
  - ***connected(Ea, Eb)*** - returns `true` if the two values are connected in the graph, and `false` otherwise.
- Note that the described behavior does not include any references to ***vertices*** - that is because there is more than one way to implement a graph!
  - And the user of the graph ***does not need to know*** (or ***care***) about implementation details!

7.3 The Graph Abstract Data Type

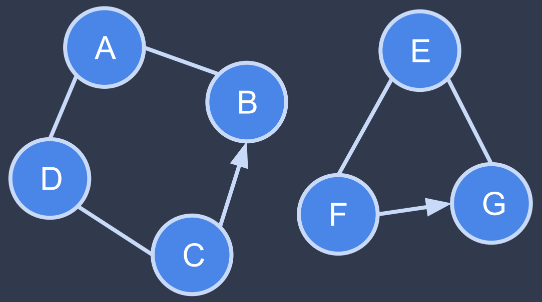
Like all of the other abstract data types we have implemented this semester, we will represent the *Graph* ADT using a Java interface to define (but not implement) it's behavior. The Graph interface should be *generic* so that the vertices can store any kind of value.

```
<<interface>>
Graph<E>
---------------------------------------
---------------------------------------
+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connectDirected(a: E, b: E)
+ connectUndirected(a: E, b: E)
+ connected(a: E, b: E): boolean
```

- Unless otherwise instructed, use the provided package named "`graphs`" for all of the code that you write in this unit.
- Create a new Java named "`Graph`".
  - Use the UML diagram to the left as a guide when implementing your `Graph` interface.
- Verify that your interface doesn't contain any syntax errors!
  - If you're not sure how to fix a problem in your code, raise your hand!

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 some ***data structure***.
  - In our graph implementation, no two vertices can be connected with more than one edge. What is the best data structure to efficiently keep track of ***unique*** neighbors?
- An adjacency list can be diagrammed using a simple table.
  - The ***first column*** lists ***each vertex*** in the graph on its own row.
  - The ***second column*** lists the vertices to which that vertex is connected (it's ***neighbors***).
- Assuming that vertex A is connected to vertex B:
  - If the edge is ***directed***, then B will appear in A's adjacency list, but not the other way around.
  - If the edge is ***undirected***, then each vertex will appear in the other's adjacency list.
- Let's take a look at an example using the graph depicted to the right.

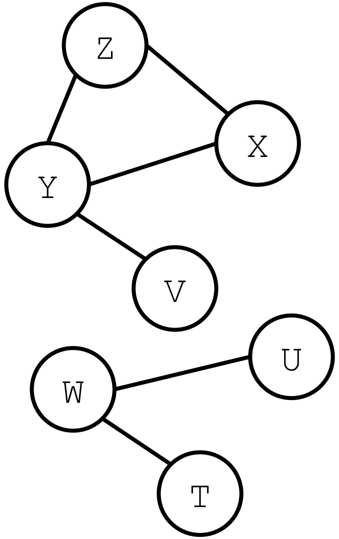


Vertex	Adjacency List
A	B, D
B	A
C	B, D
D	A, C
E	F, G
F	E, G
G	E

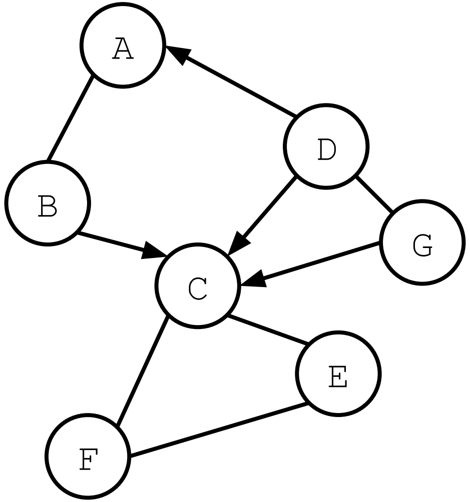
7.4 Adjacency Lists

One possible implementation of a graph involves each vertex keeping track of its neighbors in an *adjacency list*. An adjacency list can be represented using a table. Given the graphs depicted below, fill in the adjacency list for each vertex.

Vertex	Adjacency List



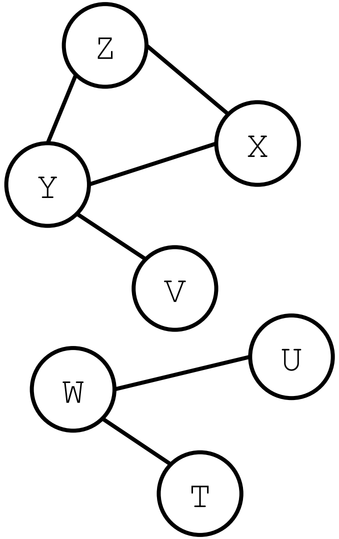
Vertex	Adjacency List



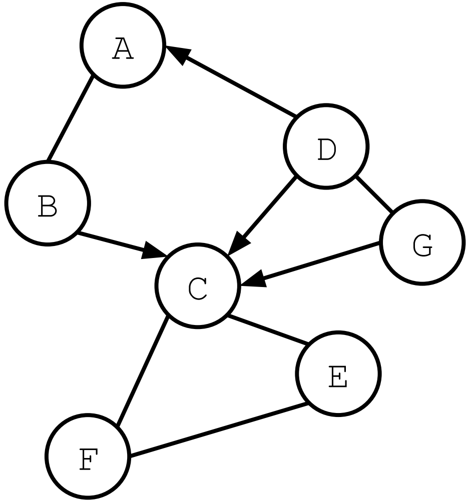
7.4 Adjacency Lists

One possible implementation of a graph involves each vertex keeping track of its neighbors in an *adjacency list*. An adjacency list can be represented using a table. Given the graphs depicted below, fill in the adjacency list for each vertex.

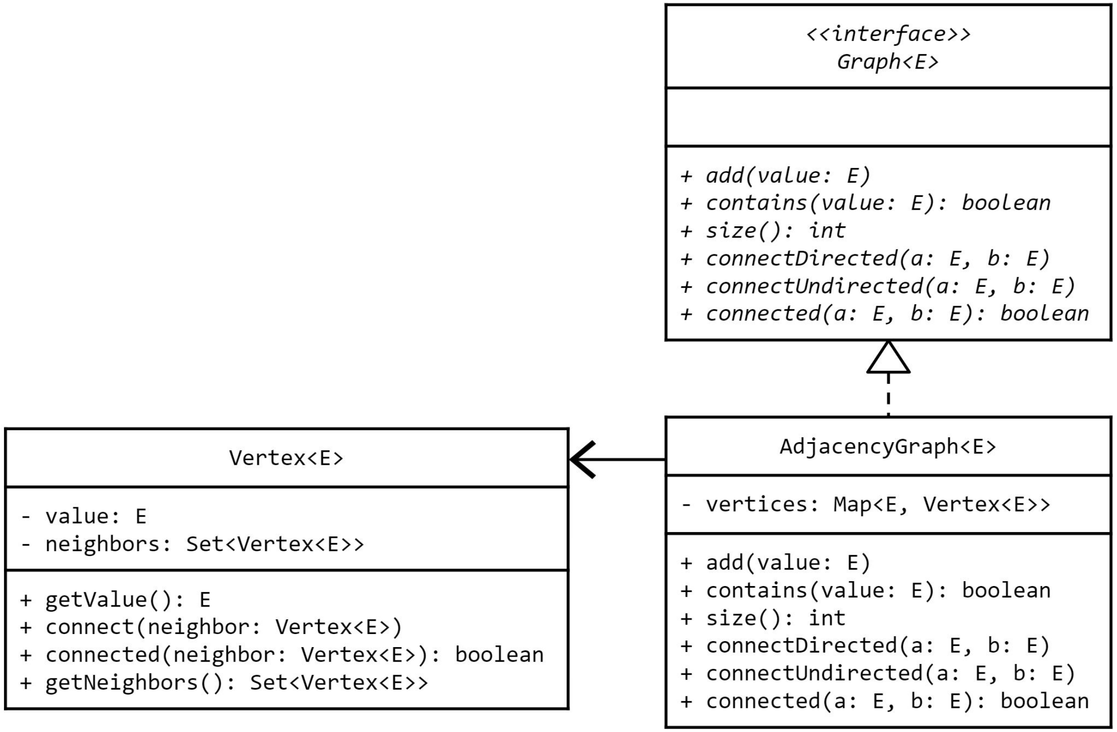
Vertex	Adjacency List
T	W
U	W
V	Y
W	T, U
X	Y, Z
Y	V, X, Z
Z	X, Y



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



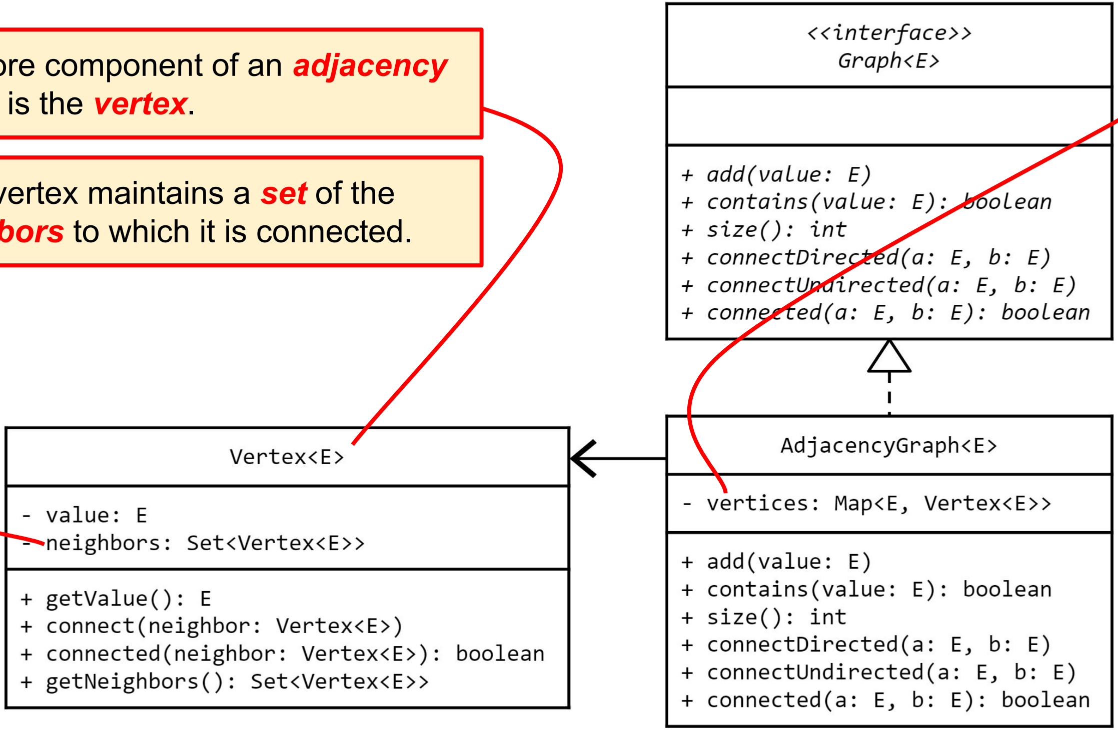
An Adjacency List Implementation



An Adjacency List Implementation

The core component of an ***adjacency graph*** is the ***vertex***.

Each vertex maintains a ***set*** of the ***neighbors*** to which it is connected.



The ***graph*** itself, maintains a ***map*** wherein the ***entries*** are ***value/vertex*** pairs.

When a value is added to the graph, a new vertex is created and the two are added to the map as the ***key/value***.

Using a ***hash map*** provides ***constant time put*** and ***get*** when adding or retrieving vertices by value.

The fact that vertices are used at all is ***completely hidden*** from the user behind the `Graph` abstraction.

7.5 A Vertex Class

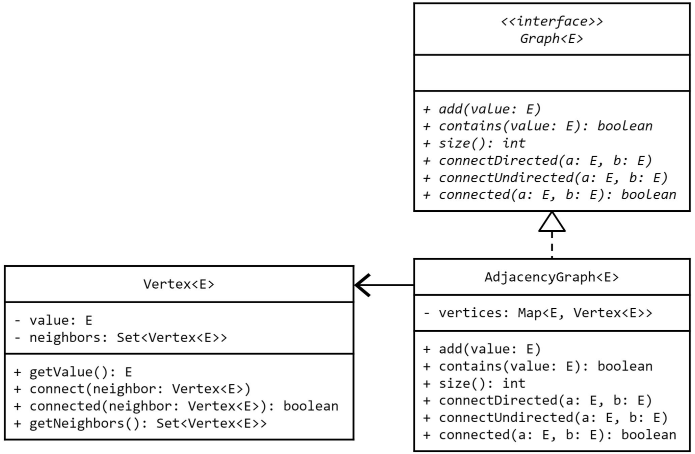
The fundamental building block of an *adjacency list* implementation of a Graph is the *Vertex*. Like the nodes used in other node-based structures that we have implemented, a vertex holds a value and keeps track of the other vertices to which it is connected. Let's create a generic Vertex class now.

```
Vertex<E>
--------------------------------------------
- value: E
- neighbors: Set<Vertex<E>>
--------------------------------------------
+ getValue(): E
+ connect(neighbor: Vertex<E>)
+ connected(neighbor: Vertex<E>): boolean
+ getNeighbors(): Set<Vertex<E>>
```

- Create a new Java class named "`Vertex`".
  - Use the UML to the left to guide your implementation of your `Vertex` class.
  - Remember, `Set` is a generic data structure, and so a ***type*** must be specified for its ***type parameter***. In this case, `neighbors` is a set of ***vertices*** of the ***same type***, i.e. `Vertex<E>`.
- ***Rename*** the provided `VertexTest` and use it to ***test*** your `Vertex` class.
  - Be careful to run the individual test in order to avoid lots of failures.

7.6 Implementing An Adjacency Graph

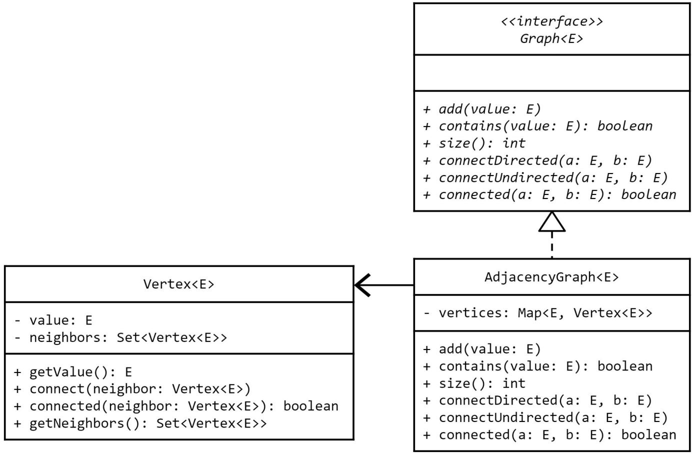
Now that we have a working Vertex that keeps track of its neighbors, we can implement the Adjacency-List Graph. It will store vertices in a `HashMap`, using each vertex's value as the key. Most of the essential functions will be implemented by storing and retrieving vertices from the map.



- Create a new Java class named "`AdjacencyGraph`" that implements the `Graph` interface.
  - Use the UML to the left as a guide in implementing your adjacency graph.
  - For now focus on:
    - ***Fields***
    - ***Constructors***
    - The `add`, `contains`, and `size` methods.
  - Stub out any remaining methods.
- ***Rename*** the provided `AdjacencyGraphTest` and use it to ***test*** your implementation.
  - Some tests will ***fail*** because you have only implemented some of the methods in the `Graph` interface.

7.7 Finish the Adjacency Graph

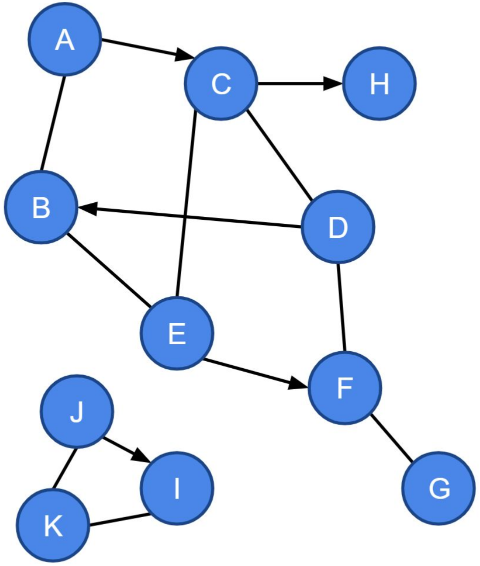
We have only partially implemented the Adjacency-List Graph. Let's complete it by implementing the remaining methods inherited from the Graph interface. When it is completed, all of the tests in the `AdjacencyGraphTest` JUnit test should pass.



- Open the `AdjacencyGraph` class and complete the implementation of your adjacency graph.
  - Implement the `connectDirected` method.
  - Implement the `connectUndirected` method.
  - Implement the `connected` method, i.e. return `true` if `a` is connected to `b`. ***Do not*** check to see if `b` is connected to `a`.
    - ***Hint***: Use the `connected` method in the `Vertex` class.
- ***Test*** your class using the provided `AdjacencyGraphTest`.
  - All tests should now ***pass***.

7.8 Building a Graph

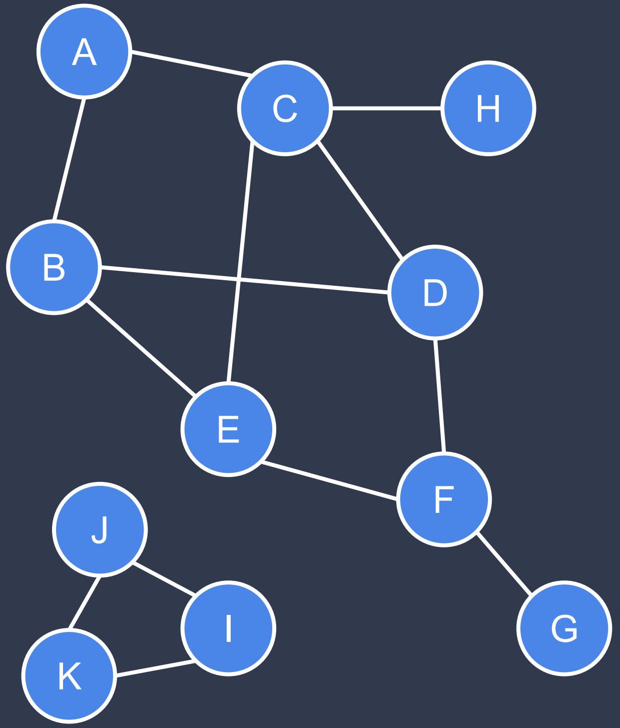
We will spend the remainder of this unit implementing several algorithms that work with the data stored inside of a graph. We'll need to build a graph so that we can test the algorithms to see if they are working as expected.



- Create a new Java class named "`Graphs`" and define a ***static*** method called `makeGraph`.
  - Create an ***empty*** `AdjacencyGraph` ***of*** `String` ***values***.
  - Write the necessary code to ***build*** the graph depicted to the left.
  - ***Return*** the graph.
- ***Test*** your code with the provided `GraphsTest`.

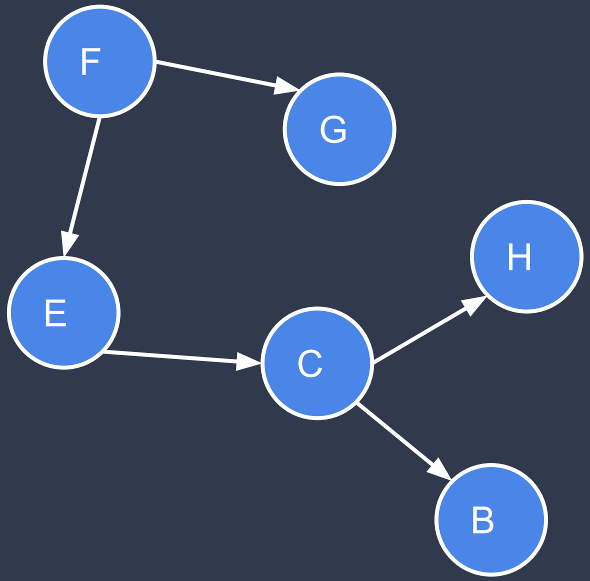
- A ***path*** is any series of edges that connects two vertices within a graph.
    - If a path ***exists*** between two vertices, then they are in the same ***connected component***.
      If no path exists, then they are in ***different*** connected components.
- A path is represented by listing the vertices along the path, e.g. the path from ***A to D*** in the example graph to the right might be ***A C D*** or ***A B D***.
- Is there a path from ***A to G*** in the example graph to the right?
- Of course there is! You can see that there are actually multiple possible paths, e.g.
    - ***A C D F G***
    - ***A B E C D F G***
- 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 have the ability to look at the ***entire graph*** all at once.
- But how would a ***computer*** determine whether or not a path exists?

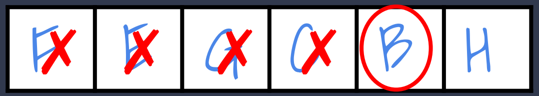
A Simple Example



Throughout this course we will examine ***three*** different path finding ***algorithms***.

Breadth-First Search (BFS)



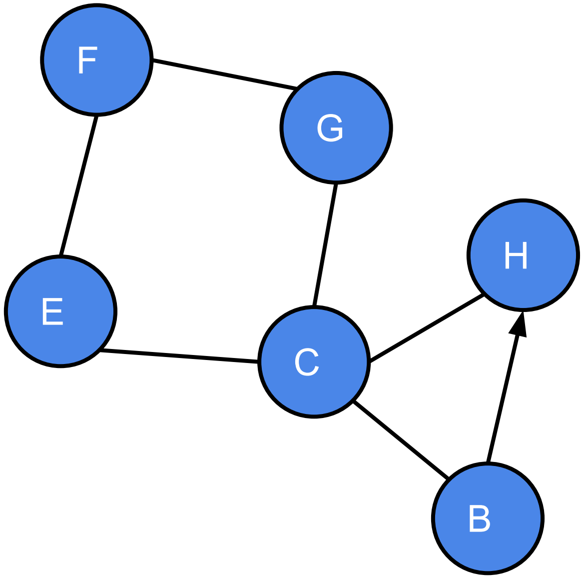
Queue 

In this example, vertex ***B*** is found ***before*** the queue is empty, and so we know that a path exists!

- Given some ***starting*** vertex ***S*** and an ***ending*** vertex ***E***, we'd like to determine if a ***path exists*** between ***S and E***.
- One algorithm for doing this is called a ***breadth-first search*** (***BFS***) and it works like this:
  - Create a ***queue*** and add ***S*** to the queue.
  - As long as the queue is ***not empty***:
    - ***Dequeue*** the next vertex. Let's call it ***V***.
    - If ***V*** is ***E***, you found a path to ***E***! ***Return*** `true`!
    - If ***V is not E***, then add each of ***V's neighbors*** to the queue.
  - If the queue is ***empty***, then there is no path from ***S to E***. If there were, one of ***E***'s neighbors would have added ***E*** to the queue. ***Return*** `false`.
- Let's try finding a path from ***F to B*** in the example graph to the right using ***BFS***.
  - Begin by adding ***F*** to a ***queue***.
  - We'll ***cross out*** each vertex as it is removed from the queue and ***add its neighbors*** to the queue.
  - The search ends when ***B*** is found, or the queue is empty.

7.9 Breadth-First Search "On Paper"

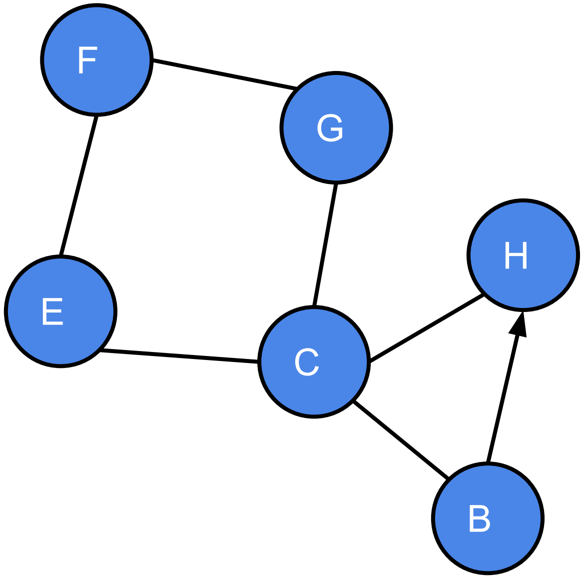
Before trying to implement an algorithm in code it can be useful to run through the steps "on paper" so that you can see how it works. Use the graph depicted below to run each Breadth-First Search (BFS). Your solution should always visit neighbors in ***alphabetical order***.



G to C												
F to C												
B to G												

7.9 Breadth-First Search "On Paper"

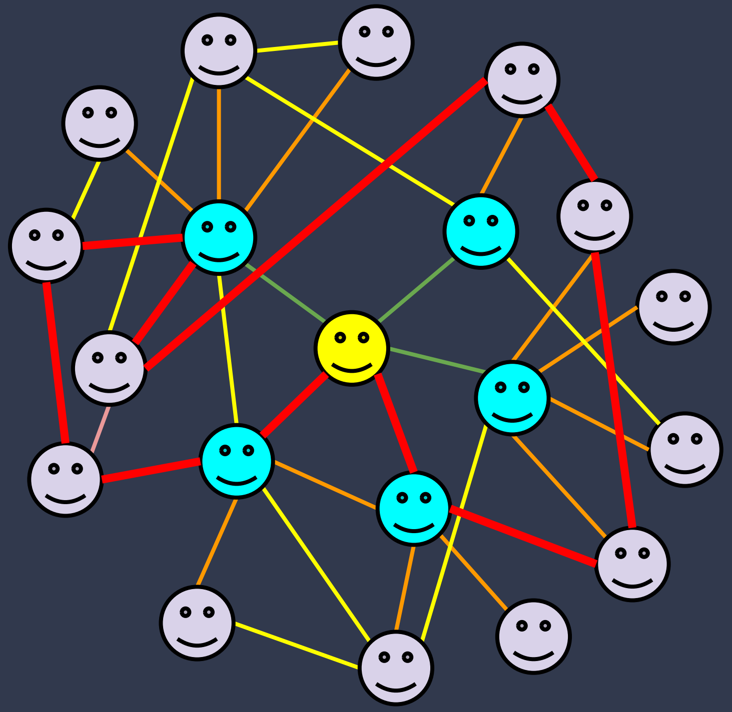
Before trying to implement an algorithm in code it can be useful to run through the steps "on paper" so that you can see how it works. Use the graph depicted below to run each Breadth-First Search (BFS). Your solution should always visit neighbors in ***alphabetical order***.



G to C	~~G~~	C	F									
F to C	~~F~~	~~E~~	~~G~~	C	F	C	F					
B to G	~~B~~	~~C~~	~~H~~	~~B~~	~~E~~	G	H	C	C	H	C	F

Cycles

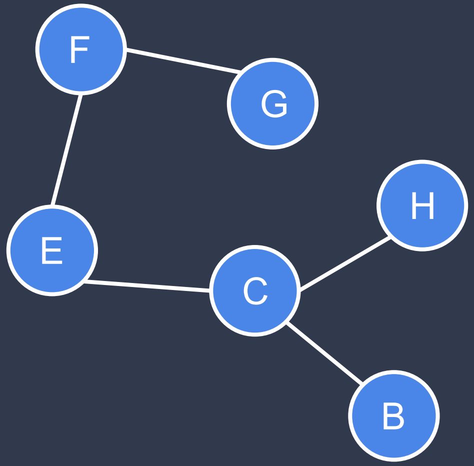
- You may have noticed something ***a little odd*** when running your BFS searches on the graph in the previous activity.
  - Namely, you kept seeing the ***same*** nodes ***again and again***.
- Imagine that you are traversing a graph, following edges from one vertex to the next.
- As you continue along your path, you eventually end up ***back*** at the node from which you ***started***.
- Such a path is called a ***cycle***, and will cause a search algorithm like ***BFS*** to visit the same nodes ***many times over***.
  - In some cases a search might even get stuck in the same cycle ***forever***.
- We will need to modify our BFS algorithm so that it keeps track of the vertices that were seen ***previously***.
  - Only ***new*** vertices are added to the queue.



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

NEW AND IMPROVED

Breadth-First Search (BFS)



Set

F H G C E B

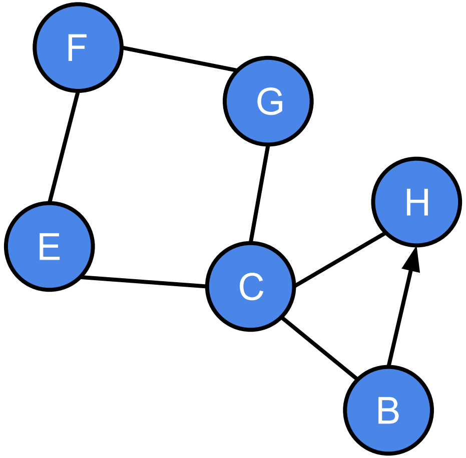
F to B

~~F~~	~~E~~	~~G~~	~~C~~	B	H

- Given some starting vertex ***S*** and ending vertex ***E***, we'd like to determine if a path ***exists*** between the two.
- But we ***don't*** want to get ***stuck in a cycle*** and waste a lot of time visiting the same vertices over and over again.
- We need a data structure to keep track of ***previously visited*** vertices.
  - We need to ***efficiently*** determine if the data structure already ***contains*** a vertex.
  - If so, we can skip it.
  - If not, we need to ***efficiently*** add the vertex to the set so that we don't visit it again.
- What data structure provides efficient ***add*** and ***get*** functionality for ***unique*** elements?
  - How about a `HashSet`?
- We'll modify the BFS algorithm to:
  - Create an empty ***set*** as well as a queue.
  - Add ***S*** to ***both*** the set and the queue.
  - Each time we dequeue a vertex ***V*** from the queue, we will make sure that each of its neighbors is ***not already in the set*** before adding it to ***both***.
- These changes should avoid visiting the same vertex more than once!

7.10 Breadth-First Search Redux

As it is currently written, our BFS algorithm may visit the same vertices over and over again if the graph that it is searching contains ***cycles***. Let's modify the algorithm to use a ***set*** to keep track of the vertices that it has visited before so that it does not visit them more than once.



G to C

Set

F to C

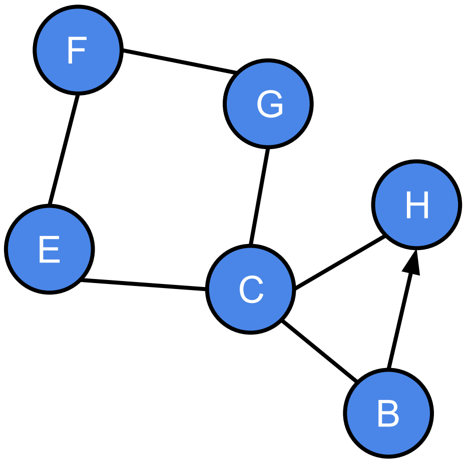
Set

B to G

Set

7.10 Breadth-First Search Redux

As it is currently written, our BFS algorithm may visit the same vertices over and over again if the graph that it is searching contains ***cycles***. Let's modify the algorithm to use a ***set*** to keep track of the vertices that it has visited before so that it does not visit them more than once.



G to C

~~G~~	C	F				

Set

G C F

F to C

~~F~~	~~E~~	~~G~~	C			

Set

G C F E

B to G

~~B~~	~~C~~	~~H~~	~~E~~	G	F	

Set

E H B C G

7.11 Adding BFS to the `Graph` Interface

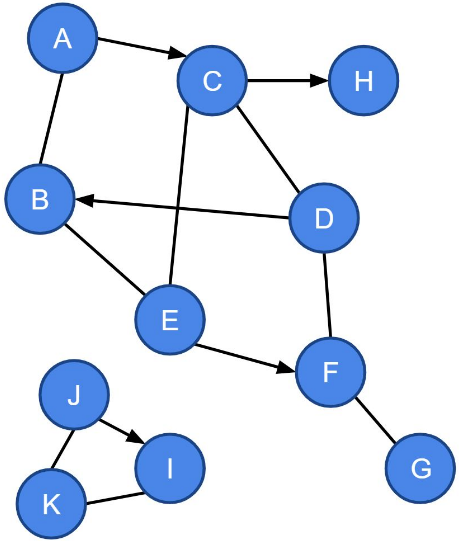
A Breadth-First Search may be run on any graph, and so it should be part of the Graph interface. Java's default method feature can be used to to add the new method to the interface without immediately breaking any existing implementations.

```
<<interface>>
Graph<E>
---
---
+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connectDirected(a: E, b: E)
+ connectUndirected(a: E, b: E)
+ connected(a: E, b: E): boolean
+ bfSearch(start: E, end: E): boolean
```

- Open the `Graph` interface and define a new `default` method named `bfSearch` that declares ***parameters*** for `start` and `end` values and ***returns*** a `boolean`.
  - Use the provided UML as a guide for your method signature.
  - Throw an `UnsupportedOperationException` from this default implementation.
- ***Run*** the provided `BFSearchTest` JUnit test.
  - The tests should ***run***, but will ***fail*** when the exception is thrown.

7.12 Begin Implementing BFS

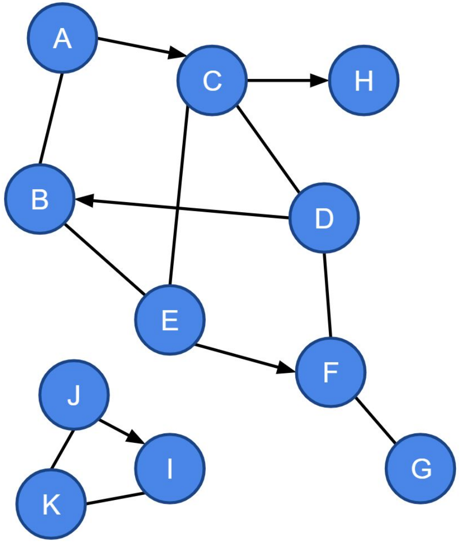
The Breadth-First Search algorithm is fairly straightforward, but it still requires a significant amount of code to implement fully. Let's begin implementing the algorithm in the `AdjacencyGraph` now by creating all of the data structures that we will need.



- Open the `AdjacencyGraph` class and ***override*** the default implementation of `bfSearch` inherited from the `Graph` interface.
- For now, focus only on the setup for the algorithm.
  - Get the vertices corresponding to the `start` and `end` values from the graph's map of `vertices`.
  - Create the ***queue*** (`java.util.LinkedList`) and ***set*** (`java.util.HashSet`).
  - ***Add*** the starting vertex to ***both***.
  - Return `false` for now.
- ***Test*** your partially implemented algorithm using the provided `BFSearchTest`.
  - ***Some*** of the tests should now ***pass***.

7.13 Complete the BFS Algorithm

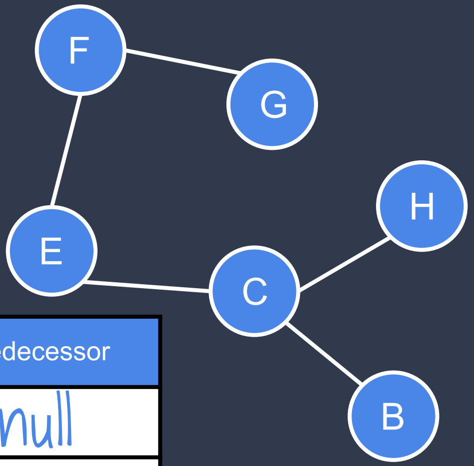
Now that the data structures are squared away, we can begin implementing the main BFS loop in the `AdjacencyGraph` class. This loop will continue visiting vertices until the end vertex is found, or the queue is empty, whichever happens first.



- Open the `AdjacencyGraph` class and navigate to the `bfSearch` method.
- Implement the ***main BFS loop*** according to the following algorithm:
  - As long as the queue is ***not empty***:
    - ***Dequeue*** the next vertex and call it ***V***.
    - If ***V*** is ***E*** return `true`.
    - Otherwise, for each of ***V***'s neighbors ***N***:
      - If the ***N*** is ***not*** already in the set:
        - ***Add N*** to ***both*** the set ***and*** the queue
  - If the queue is ***empty***, return `false`.
- ***Test*** your completed algorithm using the provided `BFSearchTest`.
  - ***All*** of the tests should now pass.

- The ***breadth-first search*** algorithm that we implemented determines whether or not a path ***exists*** between two vertices in the graph, but it doesn't tell us 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 to fly!
- Recall that, for each neighbor ***N*** of vertex ***V***:
    - If the ***N*** is not already in the set, this is the ***first time*** that we have seen ***N***.
    - This means that ***V*** is ***N***'s ***predecessor*** along the path (if it exists) - ***V*** is the vertex through which we ***first discovered N***.
- We will need to make a ***small*** change to the BFS algorithm so that it not only keeps track of every vertex, but also its ***predecessor***.
    - We will use a ***map*** instead of a ***set*** where the ***key*** is the ***vertex*** and the ***value*** is its ***predecessor***.
    - In other words, for each neighbor ***N*** of vertex ***V***, if ***N*** is not already in the map, we will put the ***N:V*** pair into the map.
- This will allow us to reverse engineer vertices along the path (if it exists) by working ***backwards*** through the map.

Building a path



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

F to B

~~F~~	~~E~~	~~G~~	~~C~~	B	H

Constructing a Path

- So now that we've filled a map with each ***vertex and its predecessor*** along the path, how do we know exactly which vertices actually form the path between ***F and B***?
- Using the map, we start at the ***end*** of the path (***B***) and work our way ***backwards*** to the beginning (***F***).
    - We use each ***vertex*** along the path to look up its ***predecessor***.
    - Each time we retrieve a vertex from the map, we insert it at the ***front*** of the path.
    - We continue until the predecessor is `null`.
- What data structure provides the features we need the most efficiently?
    - It needs to be ***iterable*** and/or offer ***access by index***.
    - Insert at the ***front*** needs to be efficient.
- How about a `LinkedList`?

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

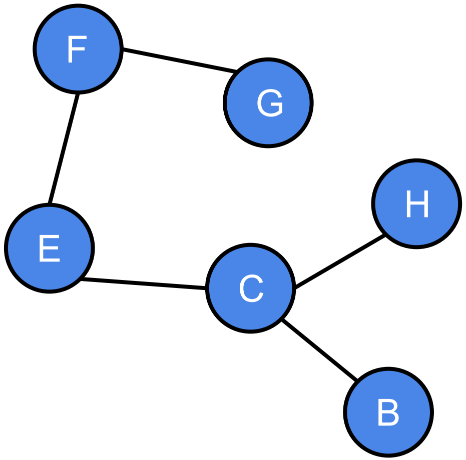
STOP HERE

START HERE

Path							
	F	E	C	B			

7.14 Breadth-First Path

Building a path using BFS is a little more complicated than simply *finding* one. Once again, it can be a useful activity to run through the steps "on paper" before trying to implement the algorithm Try it now using the searches below. Don't forget to visit neighbors in ***alphabetical order***.



Vertex	Predecessor

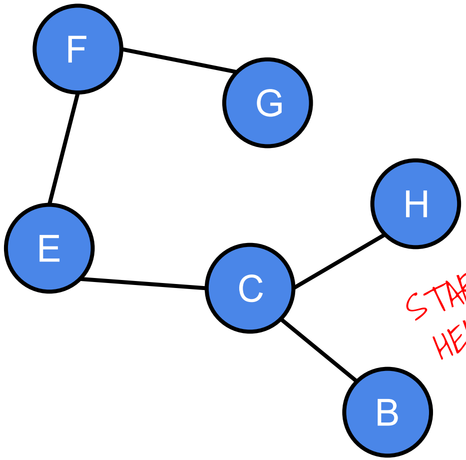
Vertex	Predecessor

G to C							
Path							

F to H							
Path							

7.14 Breadth-First Path

Building a path using BFS is a little more complicated than simply *finding* one. Once again, it can be a useful activity to run through the steps "on paper" before trying to implement the algorithm Try it now using the searches below. Don't forget to visit neighbors in ***alphabetical order***.



Vertex	Predecessor
G	null
F	G
E	F
C	E

STOP HERE

START HERE

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

STOP HERE

START HERE

G to C	~~G~~	~~F~~	~~E~~	C			
Path	G	F	E	C			

F to H	~~F~~	~~E~~	~~G~~	~~C~~	~~B~~	H	
Path	F	E	C	H			

7.15 Another Default Method in `Graph`

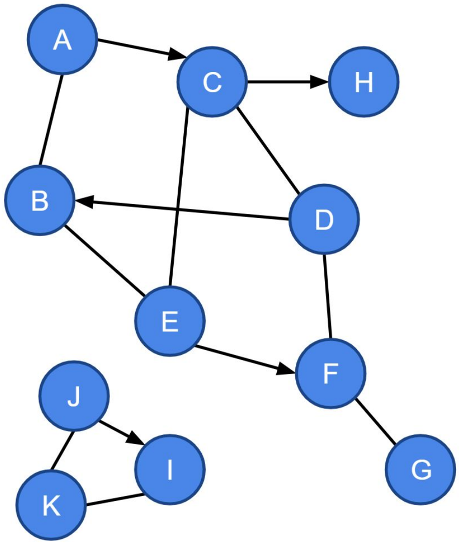
The current Breadth-First Search method returns a boolean indicating whether or not a path was found. Let's add another default method to the `Graph` interface that uses BFS to find and return the values along the path (if it exists).

```
<<interface>>
Graph<E>
-----------------------------------------
-----------------------------------------
+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connectDirected(a: E, b: E)
+ connectUndirected(a: E, b: E)
+ connected(a: E, b: E): boolean
+ bfSearch(start: E, end: E): boolean
+ bfPath(start: E, end: E): List<E>
```

- Open the `Graph` interface and define a new `default` method named `bfPath` that declares ***parameters*** for `start` and `end` values and ***returns*** a `java.util.List<E>`.
  - Use the provided UML as a guide for your method signature.
  - Throw an `UnsupportedOperationException` from this default implementation.
- ***Run*** the provided `BFPathTest` JUnit test.
  - The tests should ***run***, but will ***fail*** when the exception is thrown.

7.16 Begin Implementing BFS-Path

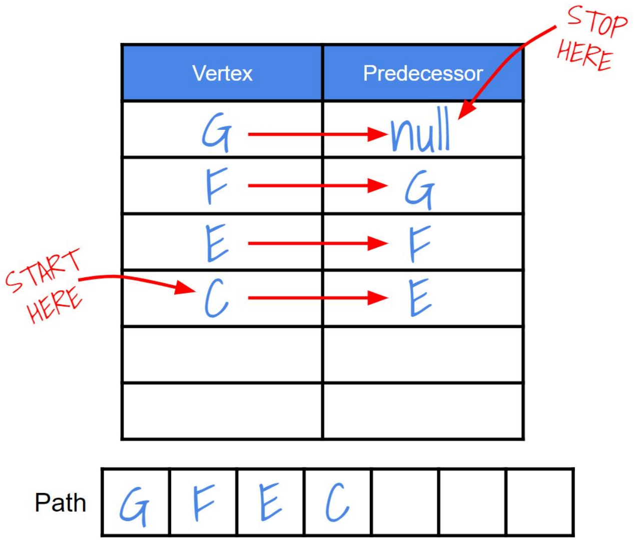
The setup for BFS-Path is very similar to the existing BFS implementation. The only major difference is that we will be using a ***map*** to keep track of each visited vertex and its predecessor (instead of a set). Begin implementing the algorithm now by focusing on creating the data structures that you will need.



- Open the `AdjacencyGraph` class and ***override*** the default implementation of `bfPath` inherited from the `Graph` interface.
- Focus on the setup for the algorithm.
  - Get the vertices corresponding to the `start` and `end` values from the graph's map of `vertices`.
  - Create the ***queue*** (`java.util.LinkedList`) and add the starting vertex.
  - Create a ***predecessor map*** (`java.util.HashMap`) and add the starting vertex as the key with a `null` predecessor.
  - Return `null` for now.
- ***Test*** your partially implemented algorithm using the provided `BFPathTest`.
  - ***Some*** of the tests should now ***pass***.

7.17 A Path-Making Helper Function

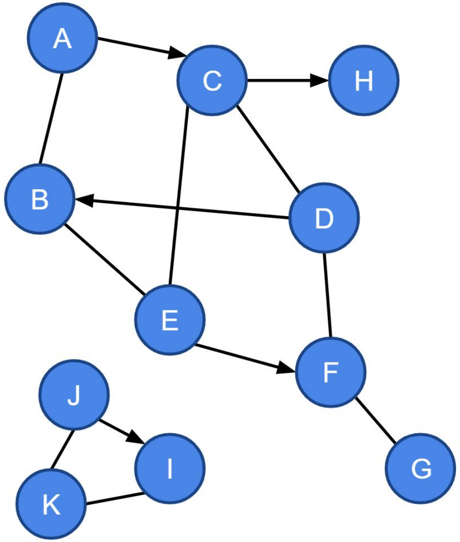
The most complicated part of the BFS-Path algorithm is using the map of predecessors to build the path ***in reverse***. Implement a helper function that adds values to the path by inserting them at index 0 in a list. Which list implementation should you use?



- Open the `AdjacencyGraph` class and define a `private` method named "`makePath`" that declares parameters for a `predecessors` map and the `end` vertex and returns a `List<E>`.
- Use the map to build a path according to the following algorithm:
  - If the map ***does not contain*** the `end` vertex, ***return*** `null`.
    - ***Hint***: use the `containsKey(end)` method on the map.
  - Otherwise:
    - Make an empty list for the ***path***.
    - Set the ***current*** vertex to `end`.
    - As long as current ***is not*** `null`:
      - ***Add*** the current vertex's ***value*** to the ***front*** of the path.
      - Use current to ***retrieve*** its predecessor from the map.
      - ***Set*** current to the predecessor.
    - ***Return*** the path.

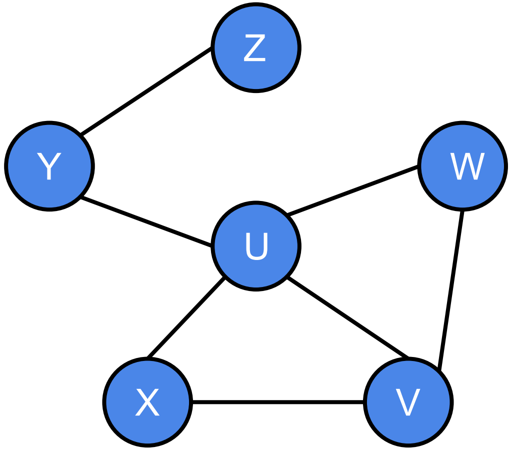
7.18 The Finishing Touches

Now that the hardest part of the BFS-Path algorithm has been implemented, finishing the algorithm should be fairly straightforward. Let's put all of the pieces together and then test the algorithm!



- Open the `AdjacencyGraph` class and navigate to the `bfPath` method.
- Implement the ***main BF Path loop*** according to the following algorithm:
  - As long as the queue is ***not empty***:
    - ***Dequeue*** the next vertex and call it ***V***.
    - If ***V*** is ***E***, ***break***.
    - Otherwise, for each of ***V***'s neighbors ***N***:
      - If ***N*** is ***not*** already in the map:
        - ***Add N*** to the queue
        - ***Put N*** (key) and ***V*** (value) in the predecessor map.
  - ***Call*** your `makePath` helper function and return the result.
- ***Test*** your completed algorithm using the provided `BFPathTest`.
  - ***All*** of the tests should now pass.

7.19 Breadth-First Path "On Paper"



Vertex	Predecessor

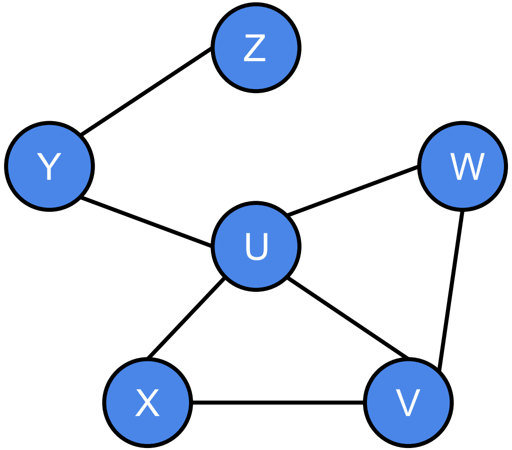
Vertex	Predecessor

Z to X							
Path							

W to Z							
Path							

7.19 Breadth-First Path "On Paper"

It is important that you be able to demonstrate an understanding of how the BFS-Path algorithm works on your exams, quizzes (and technical interviews!). Practice "on paper" now.



STOP HERE

Vertex	Predecessor
Z	null
Y	Z
U	Y
V	U
W	U
X	U

START HERE

STOP HERE

Vertex	Predecessor
W	null
U	W
V	W
X	U
Y	U
Z	Y

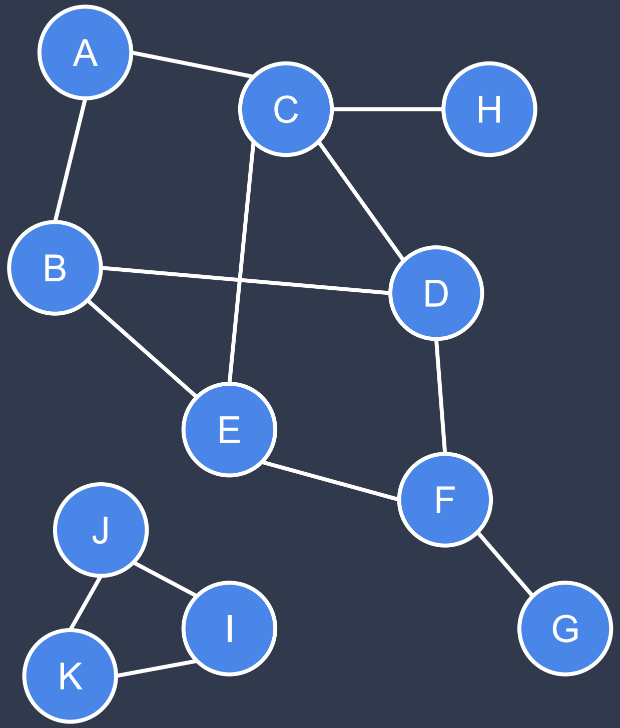
START HERE

Z to X	~~Z~~	~~Y~~	~~U~~	~~V~~	~~W~~	X	
Path	Z	Y	U	X			

W to Z	~~W~~	~~U~~	~~V~~	~~X~~	~~Y~~	Z	
Path	W	U	Y	Z			

- A ***search*** is an algorithm that determines whether or not a path ***exists*** between two vertices in a graph.
- ***Breadth-First Search*** (***BFS***) is one such algorithm.
  - BFS uses a ***queue*** to search vertices in order based on ***distance*** from the starting vertex.
  - All of the vertices that are ***one*** edge away are searched ***first***, then those that are ***two*** edges away, and so on.
- A ***Depth-First Search*** (***DFS***) is an alternative to BFS.
  - Rather than explore all of the nodes that are ***one*** edge away first, DFS explores deeper and deeper into the graph.
  - If it arrives at a vertex that does not have any ***unexplored*** neighbors, it ***backtracks*** along its path only as far as necessary to find a vertex with at least one unexplored neighbor.
- A DFS can be implemented using a ***stack*** rather than a queue, or through ***recursion***.

Depth-First Search
(DFS)

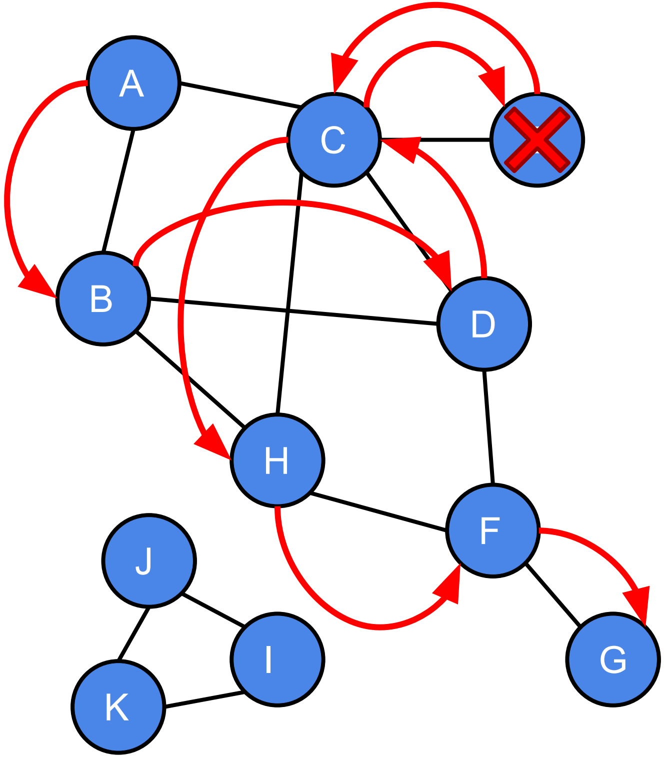


Let's practice ***depth-first search*** on this example graph to find a path from ***A to G***.

DFS Illustrated Example

As with BFS, when practicing ***depth-first search*** we will always visit neighbors in ***alphabetical order***.

Rather than search ***all*** of the neighbors of vertex ***A***, DFS will explore ***deeper*** into the graph.



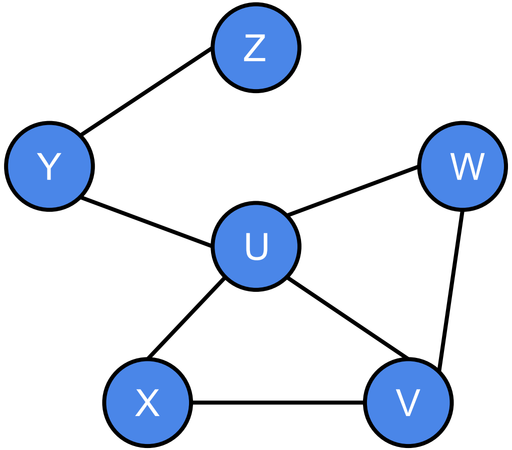
If the algorithm arrives at a node with ***no unexplored neighbors***, it will ***backtrack*** to the last node with at least ***one*** unexplored neighbor.

Like BFS, DFS also uses a ***set*** to keep track of ***previously visited vertices*** so that we avoid getting trapped in cycles.

The search will continue until ***all*** vertices have been explored, or until the ***end*** vertex is found.

7.20 Depth-First Search

Once again, practicing a new algorithm "on paper" can help gain a better understanding before trying to implement it. Practice using Depth-First Search to perform searches beginning at each vertex listed below below. Remember to always search neighbors in ***alphabetical order***.



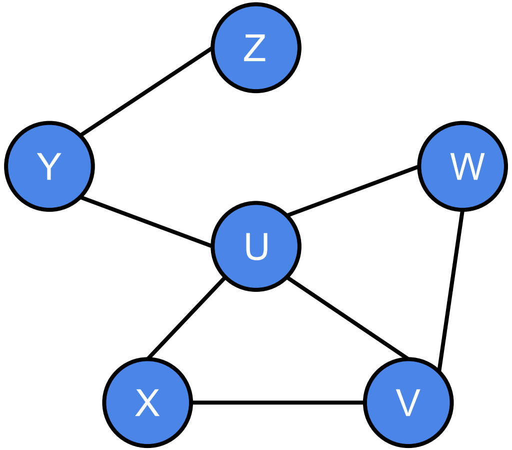
Z | | | | | | | | Set

W | | | | | | | | Set

Y | | | | | | | | Set

7.20 Depth-First Search

Once again, practicing a new algorithm "on paper" can help gain a better understanding before trying to implement it. Practice using Depth-First Search to perform searches beginning at each vertex listed below below. Remember to always search neighbors in ***alphabetical order***.



Z							
	Z	Y	U	V	W	X	

Set: Z W U Y V X

W							
	W	U	V	X	Y	Z	

Set: W V Y U Z X

Y							
	Y	U	V	W	X	Z	

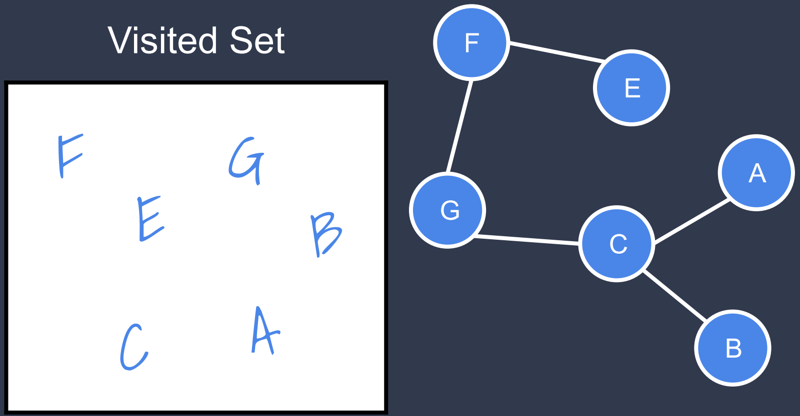
Set: U V X Z W Y

The DFS Algorithm

- Like BFS, a ***depth-first search*** can be used to determine whether or not a path ***exists*** between a start vertex ***S*** and an end vertex ***E***.
- We will be implementing a ***recursive*** implementation of DFS in two parts.
- The first part is a ***helper function*** that ***visits*** a vertex.
  - It will declare parameters for a vertex ***V*** and a set of previously ***visited*** vertices.
  - For each neighbor ***N*** of ***V*** that is not already in the ***visited*** set:
    - ***Add N*** to the ***visited*** set.
    - ***Visit N*** (make a ***recursive*** call).
- The second part is the ***main DFS method***. It will simply:
  - ***Create*** the ***visited*** set.
  - ***Add S*** to the ***visited*** set.
  - ***Call*** the helper function with ***S*** and the ***visited*** set.
  - When the function returns, return `true` if ***E*** is in the set, and `false` otherwise.

Let's look at an example of performing a ***depth-first search*** on the example graph below to find a path from ***F to B***.

Visited Set



A DFS wil search ***all*** of the vertices that are ***reachable*** from the starting vertex. If the ***end*** vertex is in the set after the search is complete, then a path exists!

7.21 Adding a Default DFS Method to `Graph`

As with the BFS and BFS-Path algorithms, we'll need to add a new method to the `Graph` interface so that we can attempt a Depth-First Search on any `Graph` implementation. We should provide a default implementation so as to avoid breaking our existing implementation.

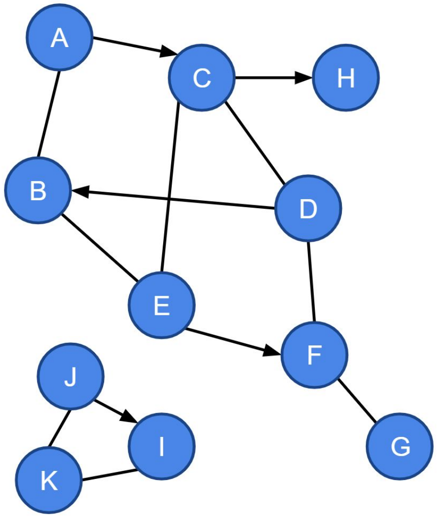
```
<<interface>>
Graph<E>
---------------------------------------

---------------------------------------
+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connectDirected(a: E, b: E)
+ connectUndirected(a: E, b: E)
+ connected(a: E, b: E): boolean
+ bfSearch(start: E, end: E): boolean
+ bfPath(start: E, end: E): List<E>
+ dfSearch(start: E, end: E): boolean
```

- Open the `Graph` interface and define a new `default` method named `dfSearch` that declares ***parameters*** for `start` and `end` values and ***returns*** a `boolean`.
  - Use the provided UML as a guide for your method signature.
  - Throw an `UnsupportedOperationException` from this default implementation.
- ***Run*** the provided `DFSearchTest` JUnit test.
  - The tests should ***run***, but will ***fail*** when the exception is thrown.

7.22 A Recursive DFS Helper Method

Depth-First Search can be implemented using either iteration or recursion. Let's write a recursive implementation beginning with a private helper method that will do most of the work.

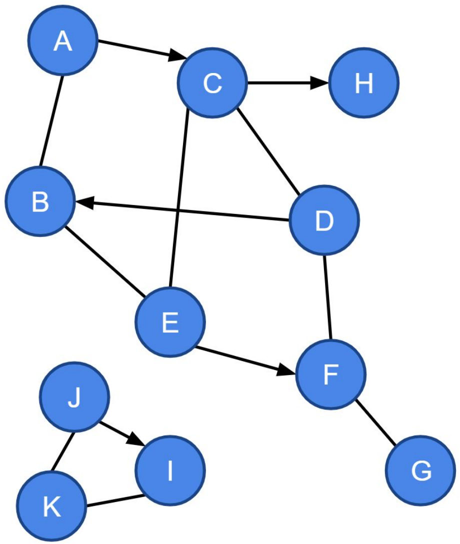


Instead of searching ***all*** of the immediate neighbors of a vertex, DFS picks ***one*** neighbor, and then ***one*** of its neighbors, and so on.

- Open the `AdjacencyGraph` class and define a `private` method named "`visitDFS`" that declares parameters for a `vertex` and a set of previously `visited` vertices.
  - For each of the `vertex`'s neighbors ***N*** that is not already in the `visited` set:
    - ***Add N*** to the `visited` set.
    - Make a ***recursive call*** with ***N*** and the `visited` set.

7.23 Put the Finishing Touches on DFS

The recursive `visitDFS` helper method does most of the work. The main DFS method will only need to create the set of visited vertices and return the final result.



DFS searches ***every*** vertex that is ***reachable*** from the start vertex. If a path exists to the end vertex, it will be in the set of ***visited*** vertices.

- Open the `AdjacencyGraph` class and ***override*** the default implementation of `dfSearch` inherited from the `Graph` interface.
- Implement the main DFS algorithm as follows:
  - Get the vertices corresponding to the `start` and `end` values from the graph's map of `vertices`.
  - Create the ***set*** of `visited` vertices.
  - ***Add*** the start vertex to the `visited` set.
  - ***Call*** the `visitDFS` helper method.
  - ***Return*** `true` if the end vertex is in the `visited` set, and `false` otherwise.
- ***Test*** your algorithm using the provided `DFSearchTest`.

DFS Path Building

The DFS path algorithm does ***not*** use a map to keep track of predecessors.

Instead, vertex ***V*** makes a ***recursive call*** for each of is neighbors ***N***. If the recursive returns `null`, that means that there ***is no path*** to ***E*** that includes ***N***.

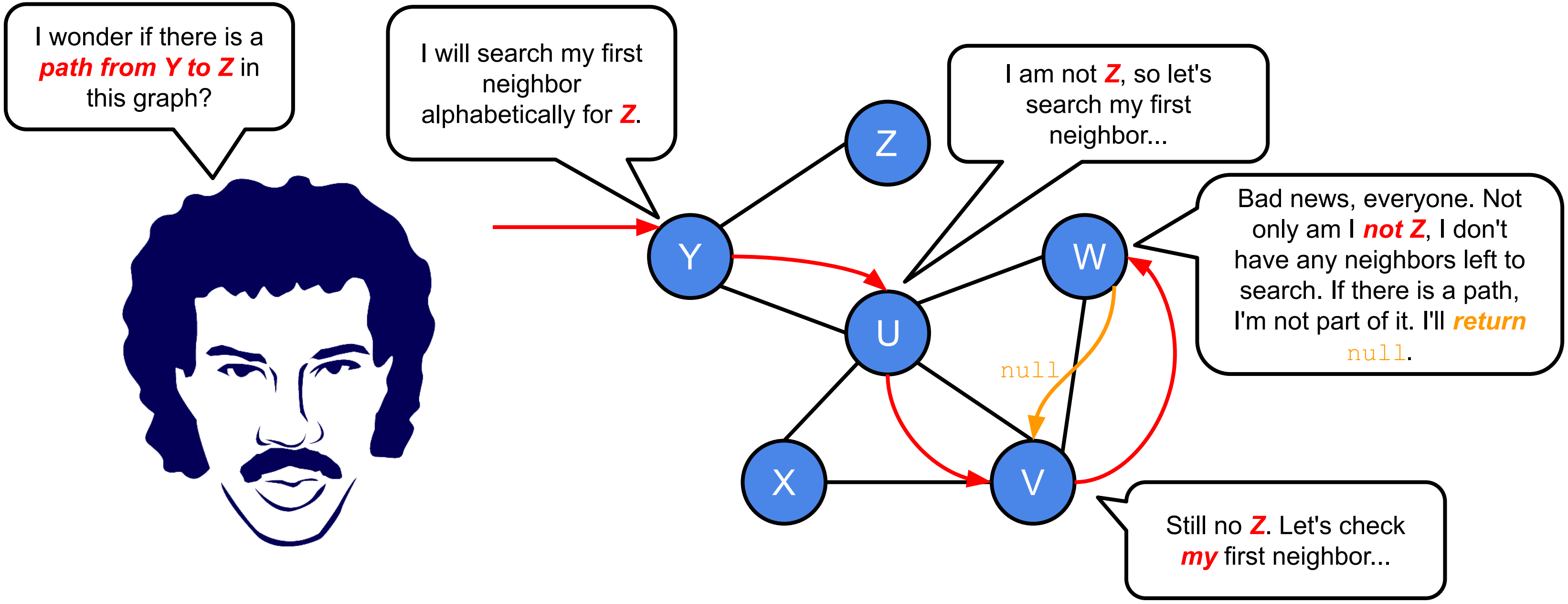
If the recursive call returns a ***non-`null` path***, e.g. a ***list*** with at least one other vertex in it, that means that a path to ***E exists***, and it includes ***N***.

Being that ***V*** is the vertex through which ***N*** was first discovered, then ***V*** is also on the path, and so ***V*** is ***added to the path*** before it is returned.

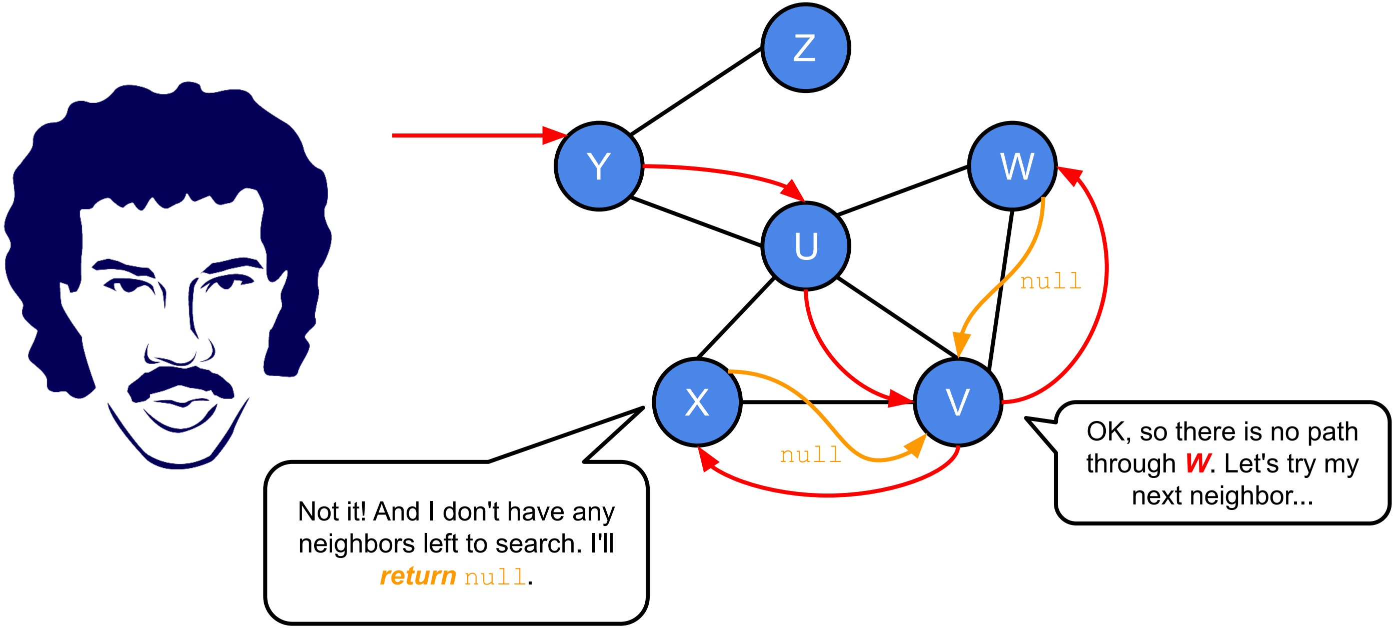
If ***none*** of ***V's*** neighbors is on the path to ***E***, then that means that ***V*** is not either, and so `null` is returned (no path).

- As with our initial version of BFS, the current version of DFS tells us whether or not a path ***exists***, but not which vertices are included along the path.
    - Also like BFS, we will need to modify the current algorithm to construct a path.
- The ***DFS path*** algorithm declares parameters for a vertex ***V***, the end vertex ***E***, and a set of previously ***visited*** vertices.
    - If ***V is E***, we found a path! Return the path containing ***only E***.
    - Otherise, for each neighbor ***N*** of ***V*** that is not in the ***visited*** set:
        - ***Add N*** to the ***visited*** set.
        - Make a ***recursive call*** to the DFS path function with ***N***, ***E***, and ***visited***.
        - If the recursive call returns a ***path***, add ***V*** to the front of the path and ***return it***.
    - If ***none*** of the neighbors returns a path, ***return*** `null`.
- Once again, the ***path*** will be built ***in reverse*** starting with the end value and working backwards.

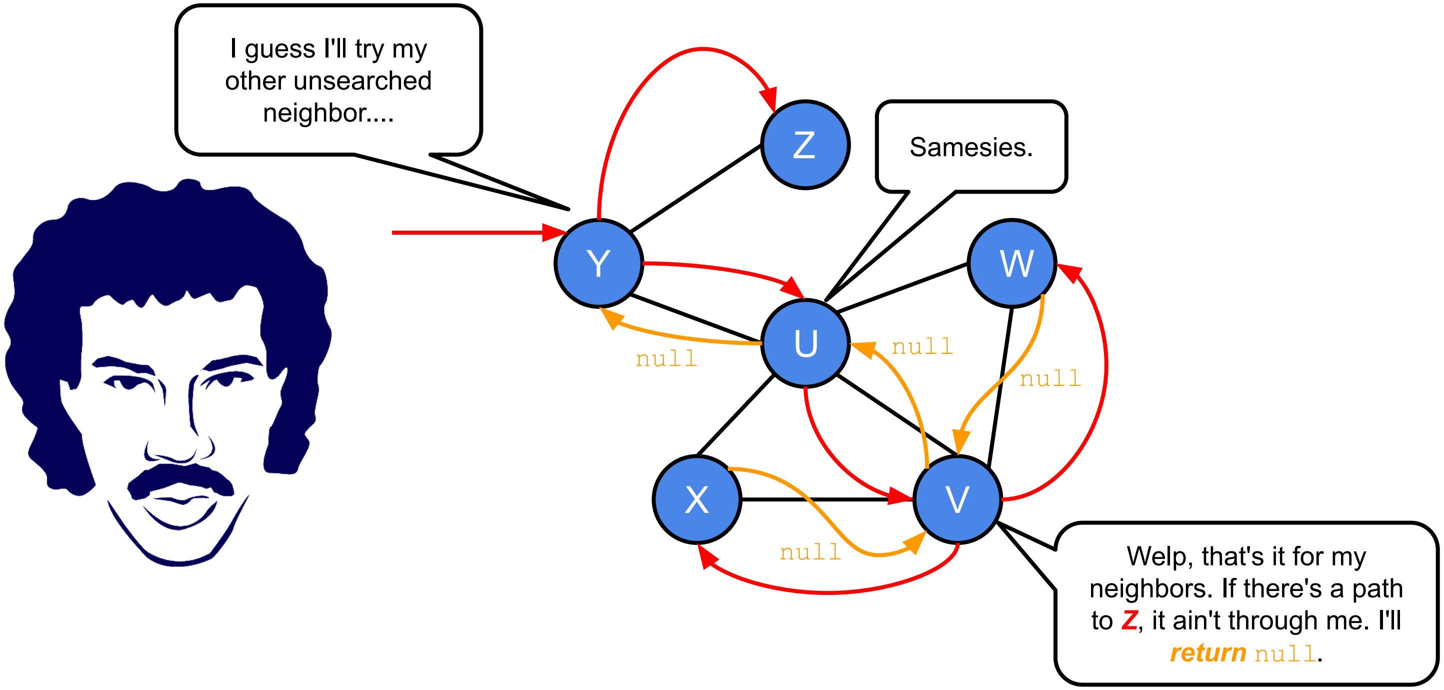
DFS Path Illustrated Example: Y to Z



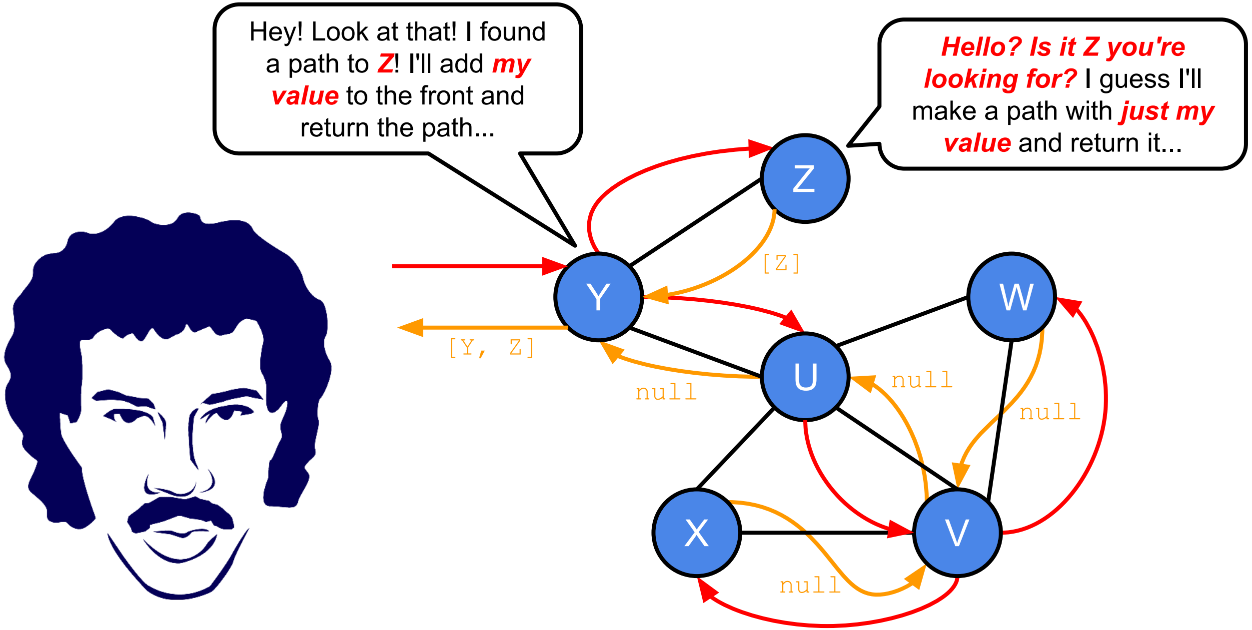
DFS Path Illustrated Example: Y to Z



DFS Path Illustrated Example: Y to Z



DFS Path Illustrated Example: Y to Z



7.24 One More Update to Graph

Our current DFS implementation will return true if a path *exists*, but like the initial BFS implementation it doesn't tell us what the path *is*. Let's add a default method to the Graph interface that we can override to return the values along a DFS-path (if it exists).

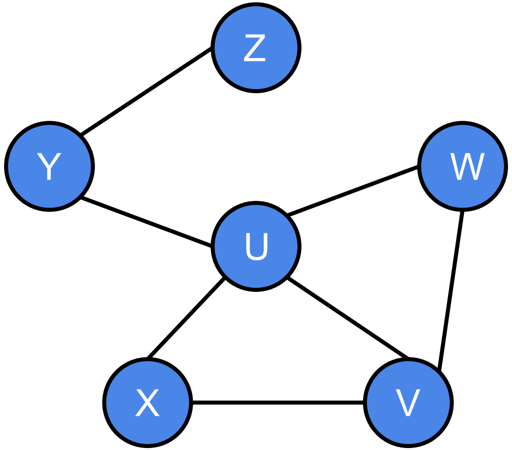
```
<<interface>>
Graph<E>

+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connectDirected(a: E, b: E)
+ connectUndirected(a: E, b: E)
+ connected(a: E, b: E): boolean
+ bfSearch(start: E, end: E): boolean
+ bfPath(start: E, end: E): List<E>
+ dfSearch(start: E, end: E): boolean
+ dfPath(start: E, end: E): List<E>
```

- Open the Graph interface and define a new default method named dfPath that declares ***parameters*** for start and end values and ***returns*** a java.util.List<E>.
    - Use the provided UML as a guide for your method signature.
    - Throw an UnsupportedOperationException from this default implementation.
- ***Run*** the provided DFPathTest JUnit test.
    - The tests should ***run***, but will ***fail*** when the exception is thrown.

7.25 Beginning Depth-First Path

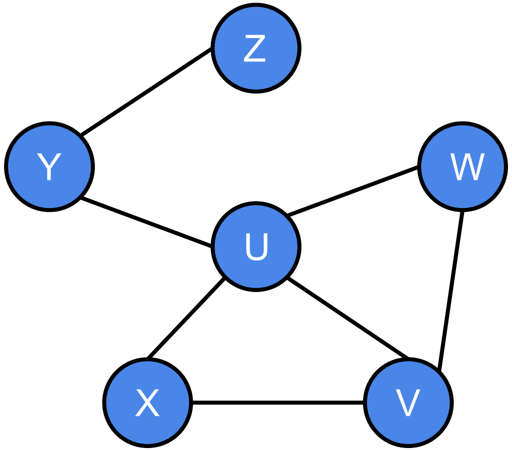
As with the numerous algorithms that we have implemented so far in this unit, we will begin implementing the DFS-Path algorithm by retrieving the start and end vertices and creating the necessary data structures.



- Open the `AdjacencyGraph` class and ***override*** the default implementation of `dfPath` inherited from the `Graph` interface.
- Focus on the setup for the algorithm.
  - Get the vertices corresponding to the `start` and `end` values from the graph's map of `vertices`.
  - Create the ***set*** of previously ***visited*** vertices (`java.util.HashSet`) and add the starting vertex.
    - ***Hint***: copy/paste your code from `dfSearch`.
- For now, just ***return*** `null`.
- ***Test*** your partially implemented algorithm using the provided `DFPathTest`.
  - ***Some*** of the tests should now ***pass***.

7.26 Complete Depth-First Path

Once again, we will be using recursion to implement the DFS-Path algorithm. As with our implementation of DFS, most of the work will be done by a helper function. Let's complete the algorithm now!



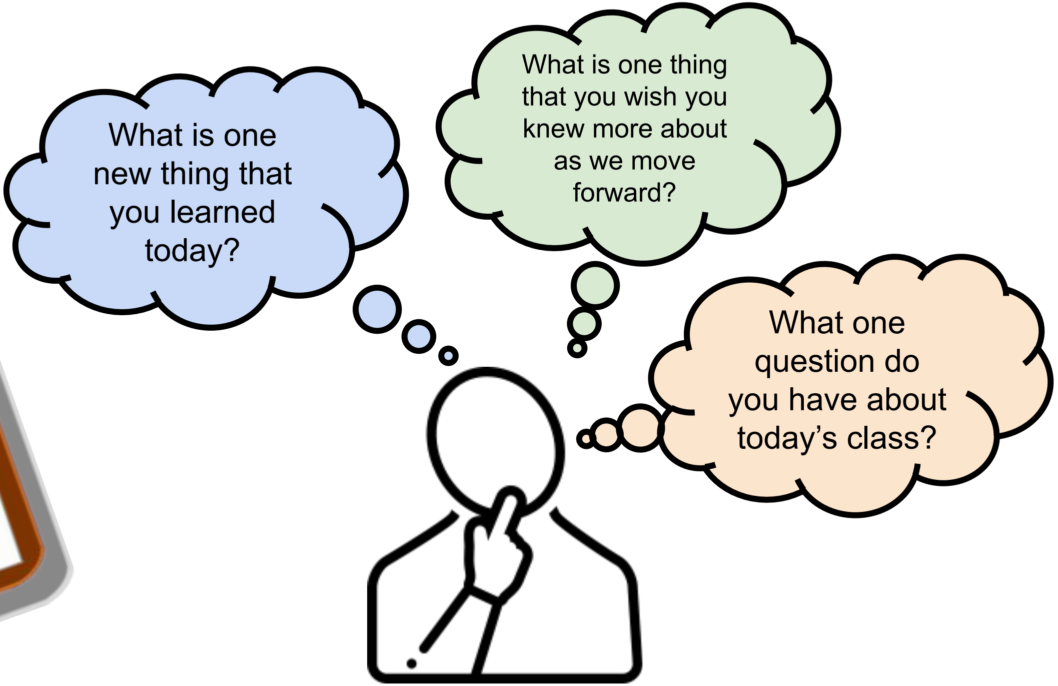
If a path is found through one of ***V***'s neighbors, ***V*** is added to the path before it is returned.

- Like the DFS algorithm, most of the work done by DFS-Path is handled by a helper function. Open `AdjacencyGraph` and define a new method called "`visitDFPath`" that declares parameters for vertices ***V*** & ***E***, and a set of previously ***visited*** vertices and returns a java.util.List<E>.
  - If ***V* is *E***, return a new `LinkedList` containing ***only E's value***.
  - Otherwise, for each neighbor ***N*** of ***V*** that is not already in the ***visited*** set:
    - Add ***N*** to the ***visited*** set.
    - Make a ***recursive call*** with the neighbor ***N***, ***E***, and the ***visited*** set.
    - If the recursive call returns a ***non-*** `null` ***path***:
      - Add ***V's value*** to the ***front*** of the path and return it.
  - If ***none*** of ***V***'s neighbors returns a path, then ***V*** is not on the path. Return `null`.
- ***Update*** `dfPath` to call `visitDFPath` and ***return the result***.
- ***Test*** your algorithm using the provided `DFPathTest`.
  - ***All*** of the tests should now ***pass***.

BFS vs. DFS

	Breadth-First Search (BFS)	Depth-First Search (DFS)
Order of Visits	ALL NEIGHBORS FIRST	ONE NEIGHBOR, THEN NEIGHBOR-OF-NEIGHBOR, ETC.
Data Structures Used	QUEUE, MAP	STACK (RECURSION)
Guarantees Shortest Path	YES (FEWEST EDGES)	NO
Time Complexity	$O(V + E)$	$O(V + E)$

Summary & Reflection



Please answer the questions above in your notes for today.