Graphs

A graph is a non-linear data structure that can be looked at as a collection of `vertices` (or `nodes`) potentially connected by line segments named `edges`.

Here is some common terminology used when working with Graphs:

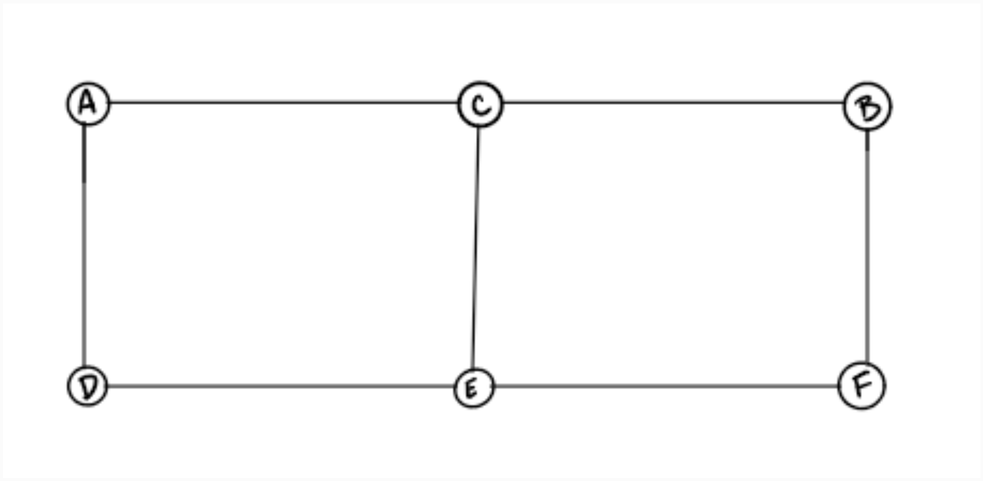
1. *Vertex* - A vertex, also called a "node", is a data object that can have zero or more adjacent vertices.
2. *Edge* - An edge is a connection between two nodes.
3. *Neighbor* - The neighbors of a node are its adjacent nodes, i.e., are connected via an edge.
4. *Degree* - The degree of a vertex is the number of edges connected to that vertex.

Directed vs Undirected

Undirected Graphs

An `Undirected Graph` is a graph where each edge is undirected or bi-directional. This means that the undirected graph does not move in any direction.

For example, in the graph below, Node `C` is connected to `Node A`, `Node E` and `Node B`. There are no "directions" given to point to specific vertices. The connection is bi-directional.



The undirected graph we are looking at has 6 vertices and 7 undirected edges.

Vertices/Nodes = {a,b,c,d,e,f}

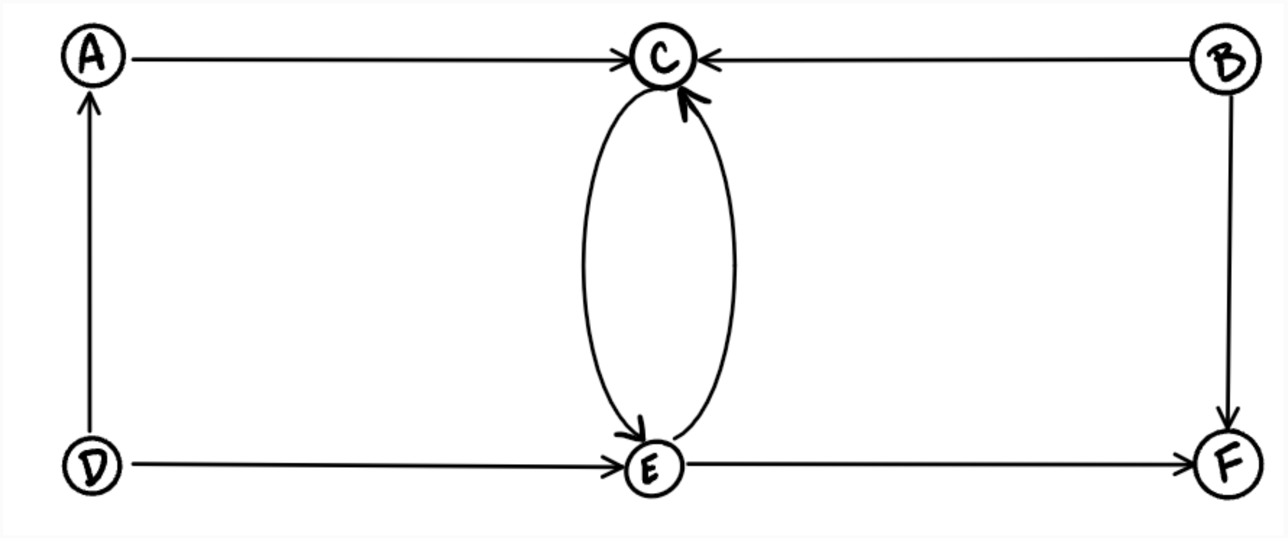
Edges = {(a,c),(a,d),(b,c),(b,f),(c,e),(d,e),(e,f)}

Directed Graphs (Digraph)

A `Directed Graph` also called a `Digraph` is a graph where every edge is directed.

Unlike an undirected graph, a `Digraph` has direction. Each node is directed at another node with a specific requirement of what node should be referenced next.

Compare the visual below with the undirected graph above. Can you see the difference? The `Digraph` has arrows pointing to specific nodes.



The directed graph above has six vertices and eight directed edges

Vertices = {a,b,c,d,e,f}

Edges = {(a,c),(b,c),(b,f),(c,e),(d,a),(d,e)(e,c)(e,f)}

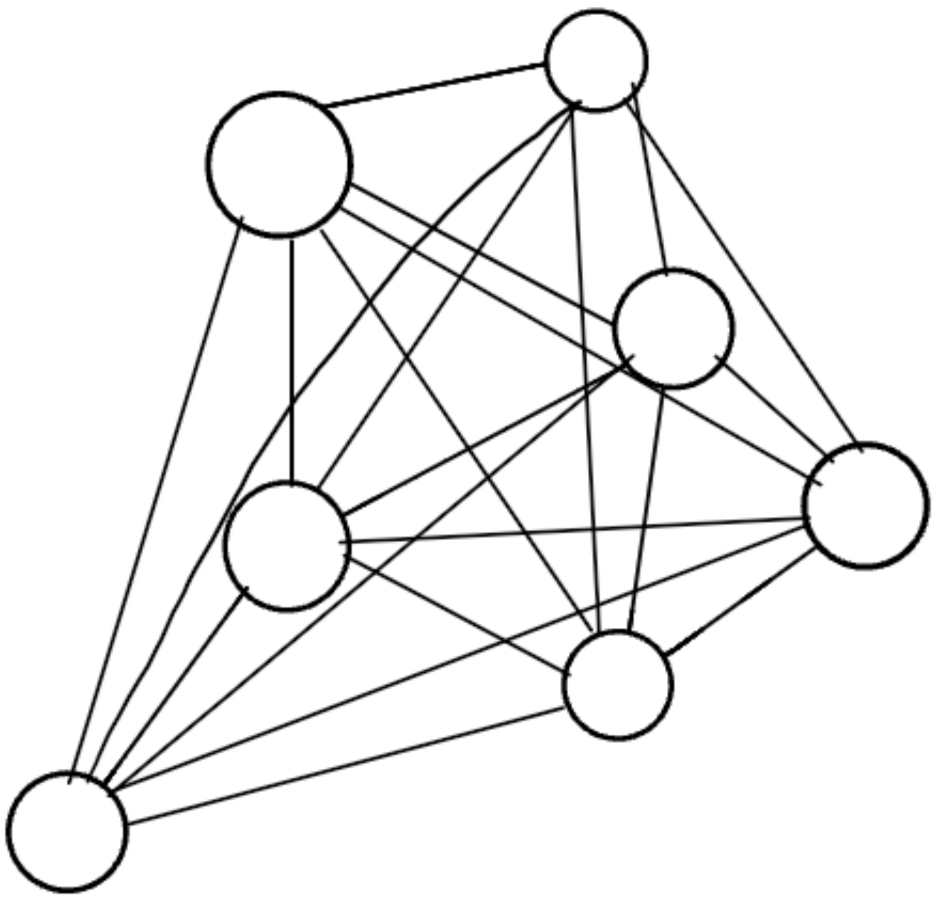
Complete vs Connected vs Disconnected

There are many different types of graphs. This depends on how connected the graphs are to other node/vertices.

The three different types are completed, connected, and disconnected.

Complete Graphs

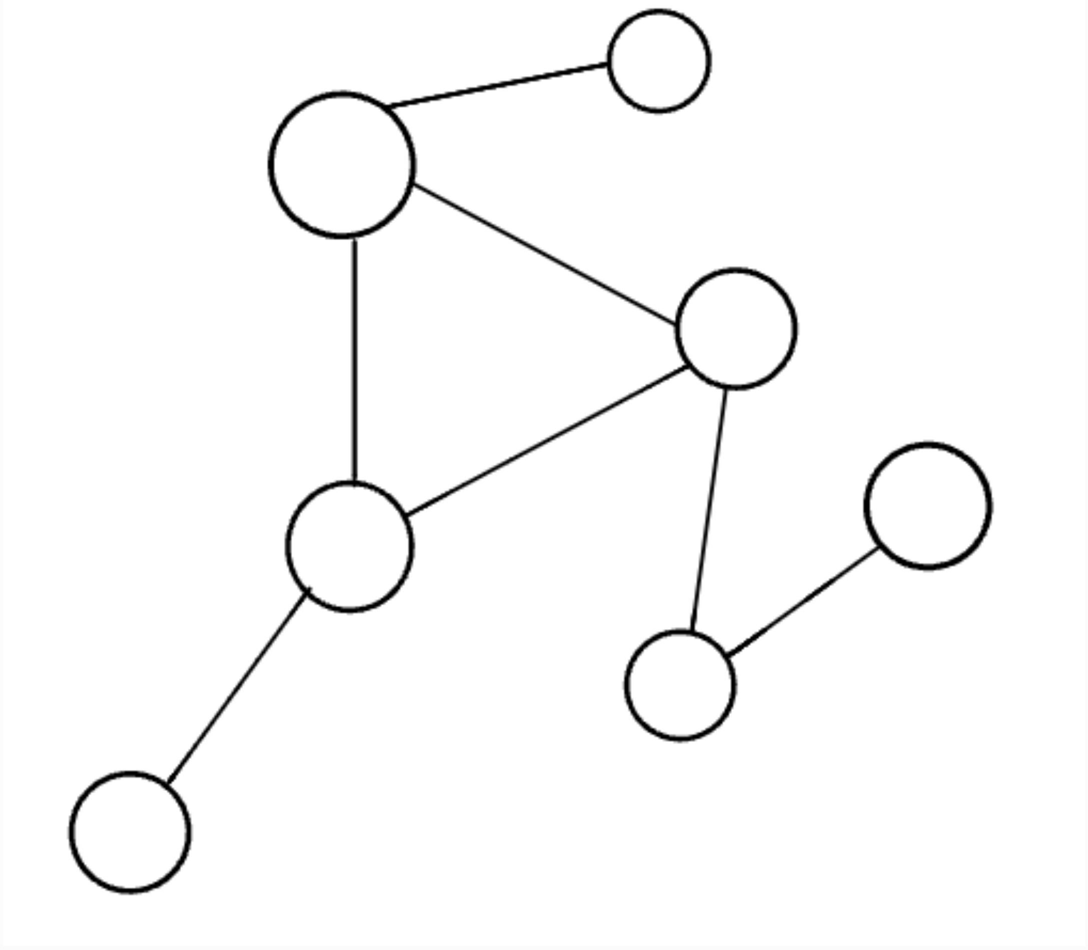
A complete graph is when all nodes are connected to all other nodes.



Take a close look at each of the vertices in the graph above. Do you notice that each vertex is actually connected to every other node on the graph? That is what makes it a complete graph.

Connected

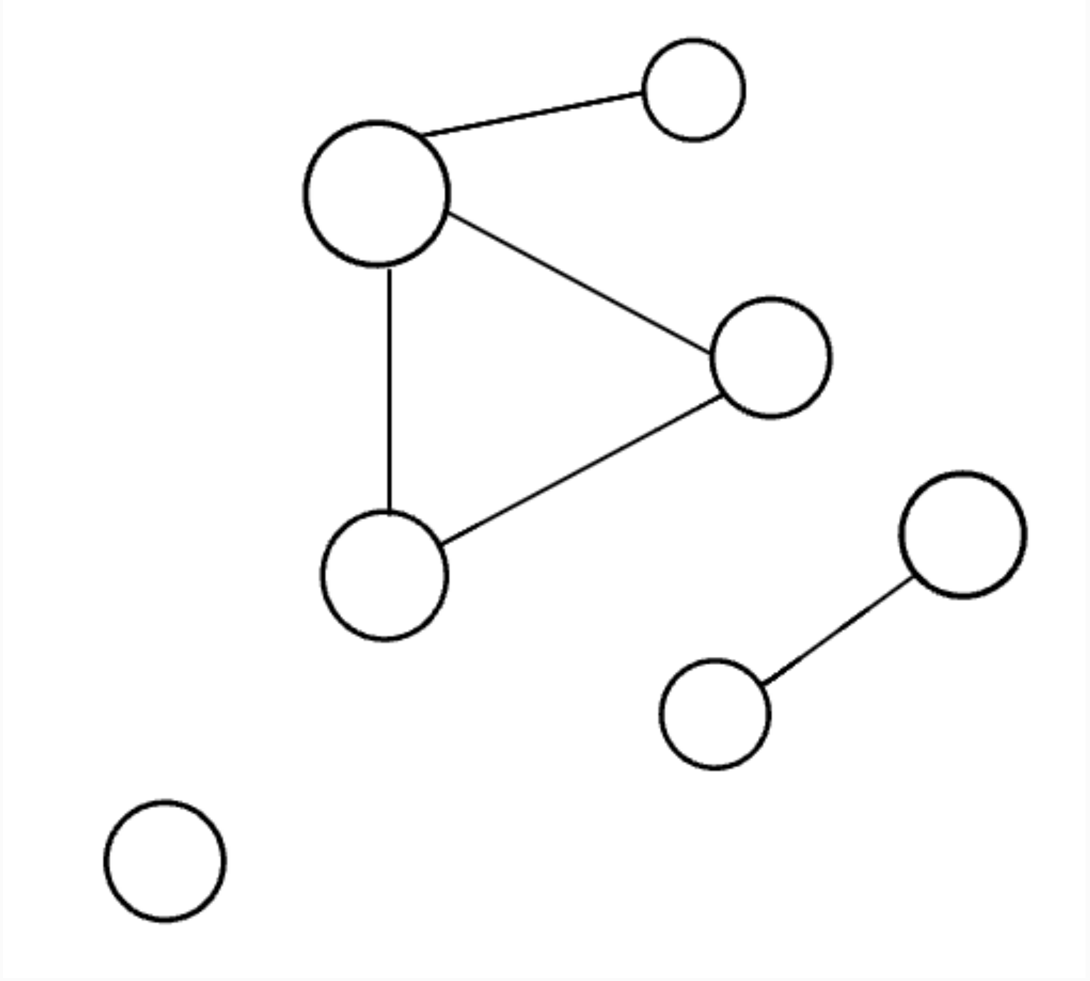
A connected graph is graph that has all of `vertices` / `nodes` have at least one edge.



In the visual above, this looks a lot more than what you are used to seeing. If you look closely at the different vertices of the graph, you will see that each node is connected to at least one other node or vertices. A `Tree` is a form of a connected graph. We will talk more about that in a bit.

Disconnected

A disconnected graph is a graph where some vertices may not have edges.



In the above visual, the disconnected graph shows that some nodes may not always be connected to other nodes or vertices of the graph. It is complelty possible to have standalone nodes or edges (also known as islands) in a graph data structure.

Acyclic vs Cyclic

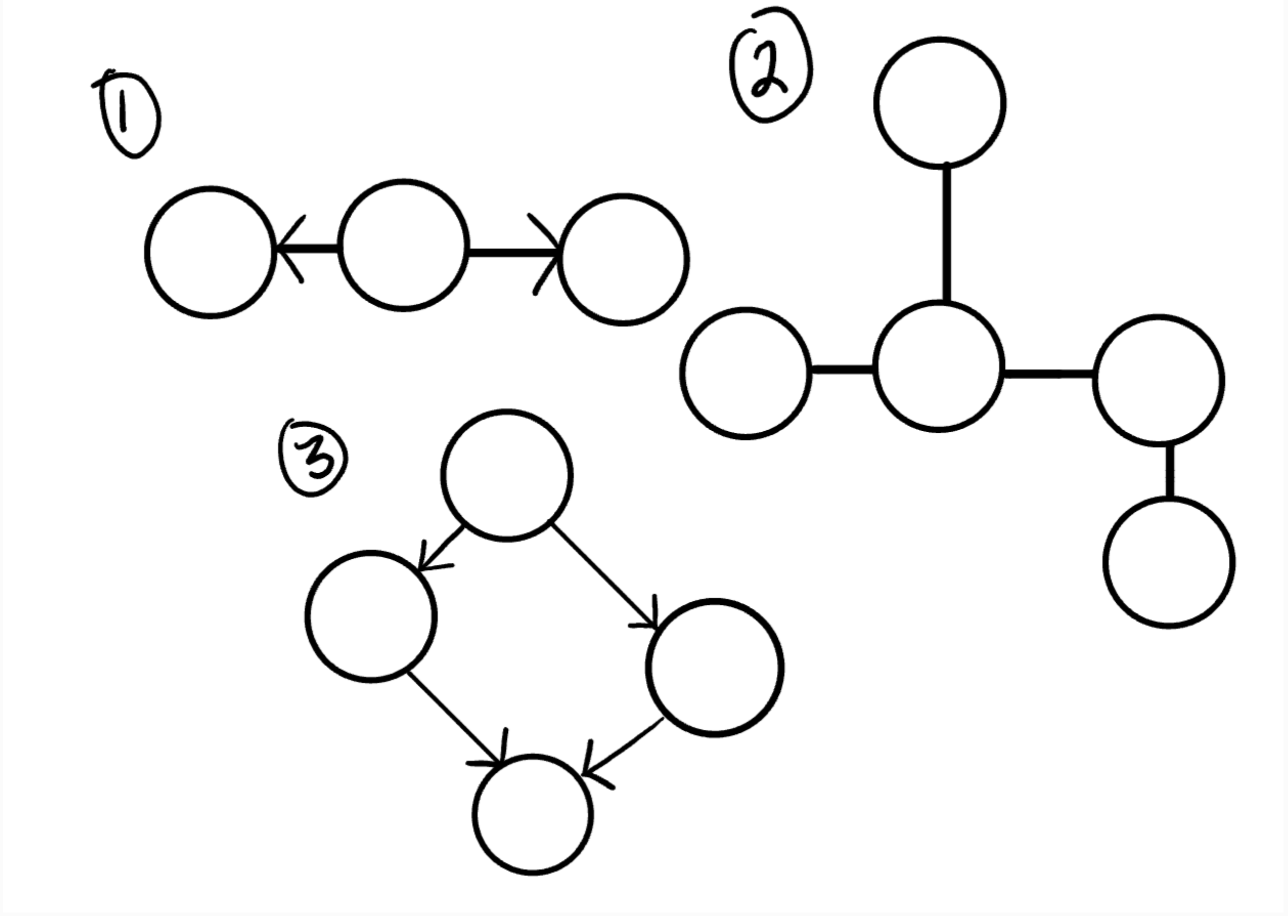
In addition to undirected and directed graphs, we also have acyclic and cyclic graphs.

Acyclic Graph

An acyclic graph is a directed graph without cycles.

A cycle is when a node can be traversed through and potentially end up back at itself.

Here is an example of 3 acyclic graphs:



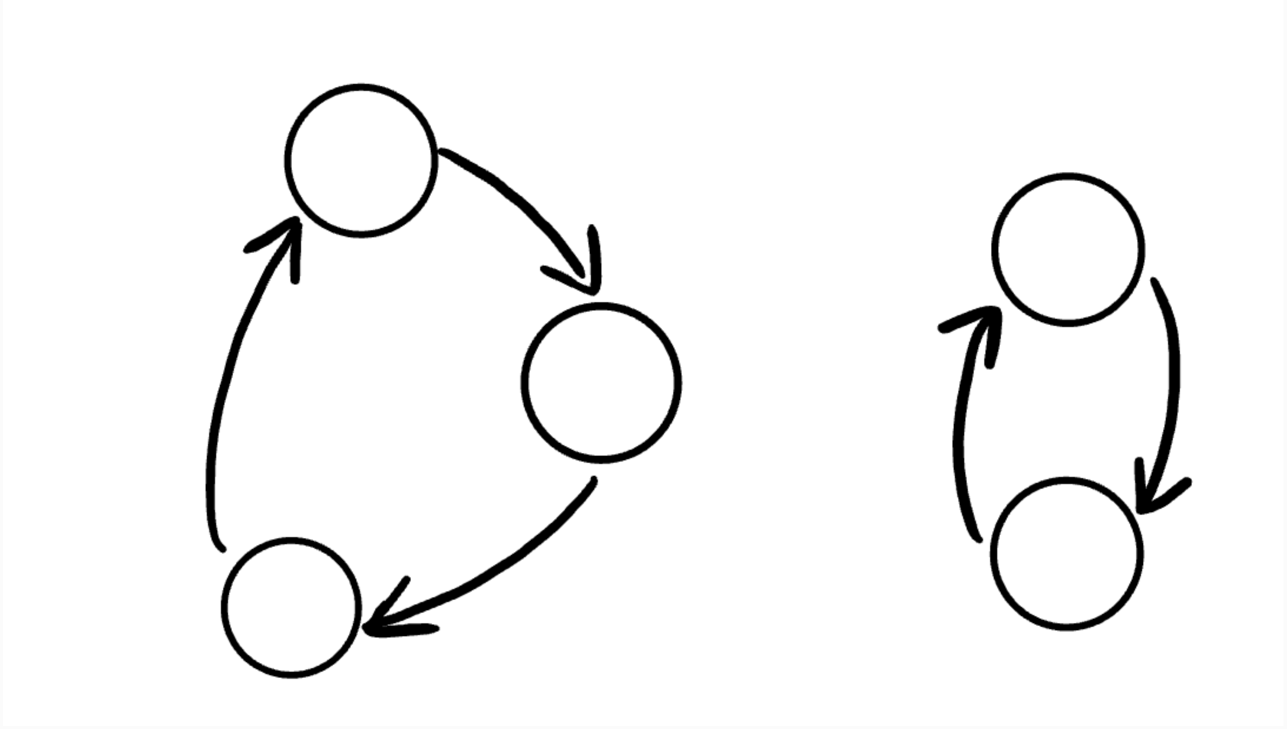
A directed acyclic graph is also called a DAG. This can also be represented as what we know as a **tree**.

Cyclic Graphs

A Cyclic graph is a graph that has cycles.

A cycle is defined as a path of a positive length that starts and ends at the same vertex.

Here is an example of a two different cyclic graph:

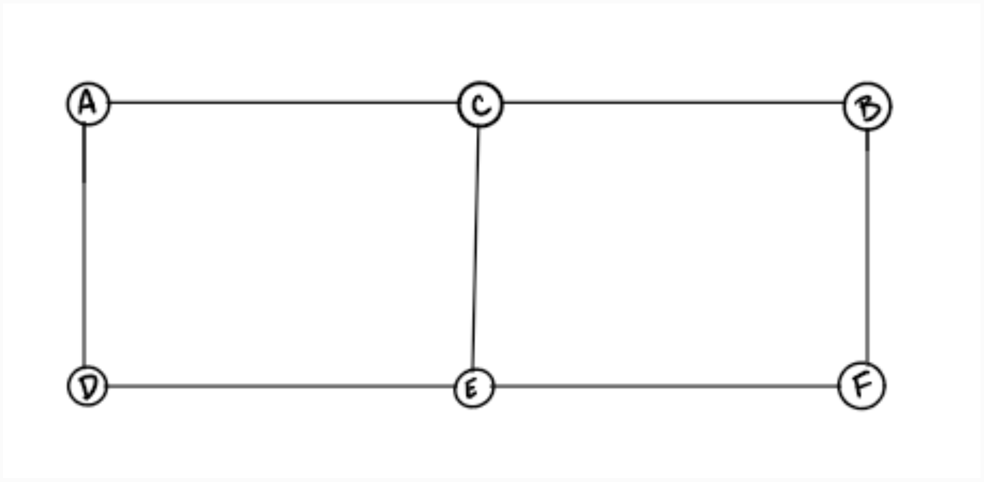


Graph Representation

We represent graphs through:

1. Adjacency Matrix
2. Adjacency List

We will represent the following graph as both an Adjacency Matrix and an Adjacency List:



Adjacency Matrix

An Adjacency matrix is represented through a 2-dimensional array. If there are *n* vertices, then we are looking at an *n x n* Boolean matrix

Each Row and column represents each vertex of the data structure. The elements of both the column and the row must add up to 1 if there is an edge that connects the two, or zero if there isn't a connection.

This is what an adjacency matrix looks like:

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

A few things to note from the above:

1. Looking at the graph we are representing, you can see that `Vertex A` connects to both `Vertex D` and `Vertex C`. To show this, we place a `1` in the position of `(a,c)` and `(a,d)`.
2. We follow this same pattern for the other vertex's and where they are connected.
3. If there is not an edge/connection between the vertex's, we represent this by placing a `0` in the appropriate point of the matrix.

a **sparse** graph is when there are very few connections. a **dense** graph is when there are many connections

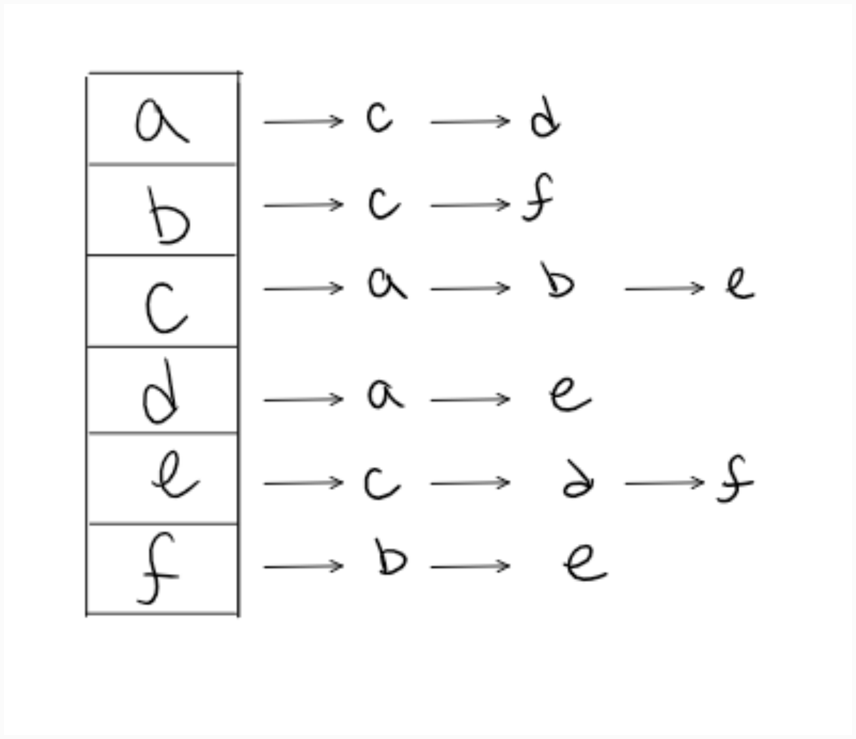
Within an adjacency matrix, an undirected graph will always be symmetric. This is not the case for a directed graph.

Adjacency List

An adjacency list is the most common way to represent graphs. An adjacency list is a collection of linked lists or array that lists all of the other vertices that are connected.

Adjacency lists make it easy to view if one vertices connects to another.

This is what an Adjacency List looks like:



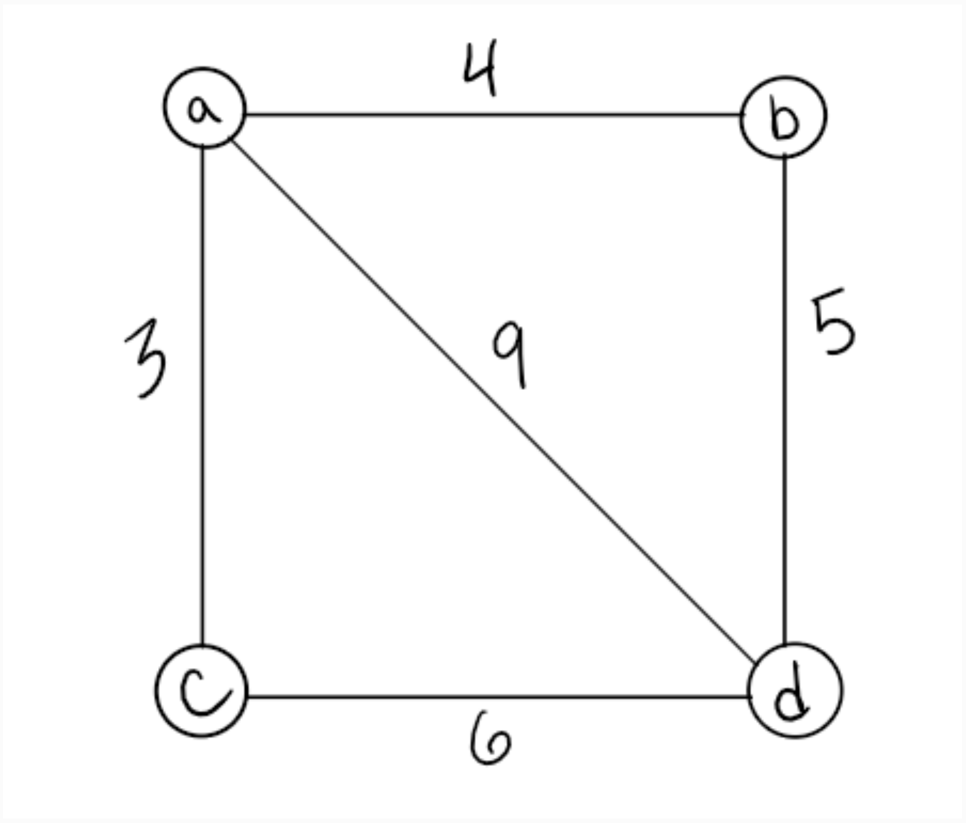
Looking at the original graph that we are representing, we can see that `Vertex A` has an edge to both `Vertex C` and `Vertex D`. As a result, we will place both `Vertex C` and `Vertex D` in the adjacency list. Just from observation, we can see that we will only place the vertices that are connected in the list. If there is no connection between the vertices, they are not listed.

Thinking about how we will implement this in code? Well, let's look at what the visual is telling us.

1. We can visually see that we are working with a collection of some sort. The visual is depicting a *Linked List*, but you could easily make it an *array of arrays* if you'd like.
2. Each index or node (depending on the data structure you choose to represent the adjacency list) will be a vertex within the graph.
3. Every time you ***add an edge***, you will find the appropriate vertices in the data structure and add it to the appropriate location.

Weighted Graphs

A weighted graph is a graph with numbers assigned to its edges. These numbers are called weights. This is what a weighted graph looks like:



When representing a weighted graph in a matrix, you set the element in the 2D array to represent the actual weight between the two paths. If there is not a connection between the two vertices, you can put a `0`, although it is known for some people to put the infinity sign instead.

Using the graph from above, here is an example of what a `weight matrix` would look like:

$$\begin{array}{c|cccc|} & a & b & c & d \\ \hline a & 0 & 4 & 3 & 9 \\ b & 4 & 0 & 0 & 5 \\ c & 3 & 0 & 0 & 6 \\ d & 9 & 5 & 6 & 0 \end{array}$$

Within adjacency lists, you must include *both* the weight and the name of the adjacent vertex.

Here is an example of what this may look like:

$$\begin{tikzcd}
\boxed{a} \arrow[r] & b,4 \arrow[r] & d,9 \arrow[r] & c,3 \\
\boxed{b} \arrow[r] & a,4 \arrow[r] & d,5 & \\
\boxed{c} \arrow[r] & a,3 \arrow[r] & d,6 & \\
\boxed{d} \arrow[r] & a,9 \arrow[r] & b,5 \arrow[r] & c,6
\end{tikzcd}$$

Do you notice the differences and similarities of a weighted adjacency list vs an unweighted? A great way to represent the {vertices, weight} connection is through some sort of key/value pair data structure.

Traversals

You will be required to traverse through a graph. The traversals itself are like those of trees. Below is a breakdown of how you would traverse a graph.

Breadth First

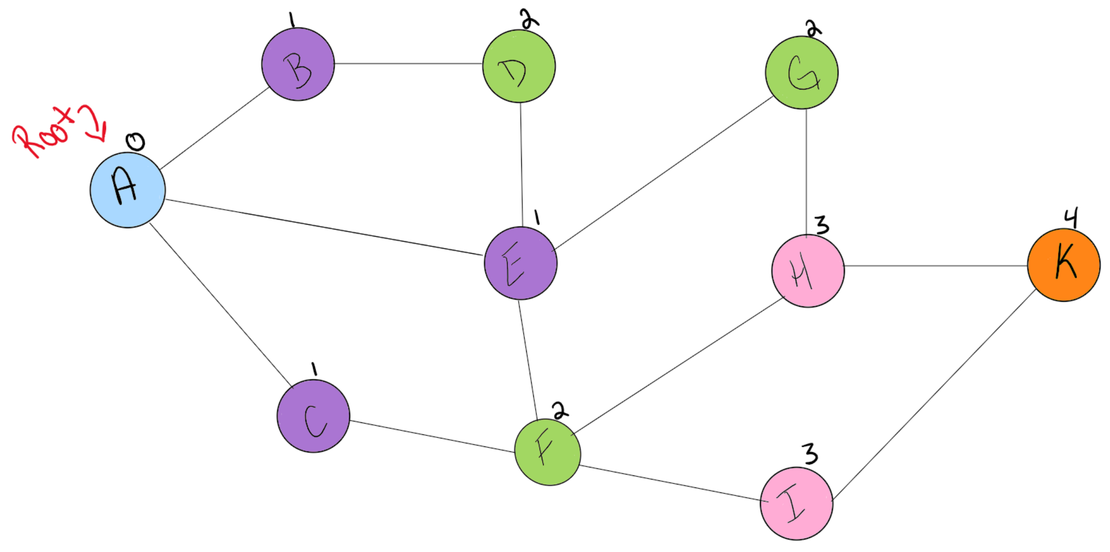
In a breadth first traversal, you are starting at a specific vertex/node. This node must be specified when calling the `BreadthFirst()` method. The breadth-first traversal of a graph is like that of a tree, with the exception that graphs can have cycles. When a graph has cycles and we are trying to traverse, this leaves the possibility to be in an infinite loop….this is bad. To prevent such behavior, we need to have some sort of flag that specifies if we have already visited that vertices. Upon each visit, we just set the "visited" flag from `false` to `true`.

As a refresher of what breadth first actually means here it is: Breadth first traversal is when you visit all the nodes that are closest to the root as possible. From there you traverse outwards, level by level, until you have visited all the vertices/nodes.

Here is what the algorithm breadth first traversal looks like:

1. `Enqueue` the declared start node into the Queue.
2. Create a loop that will run while the node still has nodes present.
3. `Dequeue` the first node from the queue
4. if the `Dequeue`'d node has unvisited child nodes, mark the unvisited children as visited and re-insert them back into the queue.

Let's look at a visual for a breadth first:



The visual above shows the levels in which the nodes will be added to the queue. You can see that since the root node is `A`, it will look the nodes that are only 1 away from the root. This is `C`, `E`, & `B`.

Next it will look at the nodes that are 2 away from the root, this is `F`, `G`, & `D`. It will follow this pattern until it reaches the end of the graph and all nodes have been visited.

Look at the code below to take a closer look at what is actually happening. This is the code for a breadth first traversal:

```
ALGORITHM BreadthFirst(vertex)
    DECLARE nodes <-- new List()
    DECLARE breadth <-- new Queue()
    breadth.Enqueue(vertex)

    while (breadth is not empty)
        DECLARE front <-- breadth.Dequeue()
        nodes.Add(front)

        for each child in front.Children
            if(child is not visited)
                child.Visited <-- true
                breadth.Enqueue(child)
    return nodes;
```

Here is a breakdown of what is going on:

1. We have declared that our starting node (or root) is going to be `Node A`.
2. The first thing we want to do is `Enqueue` the root
3. Next, we enter a while loop. We want this loop to keep running until there are no more nodes in our queue.
4. Once we are in the while loop, we want to `Dequeue` the front node and then check to see if it has any children.
5. if there are children of the node we are currently looking at, we want to mark each one as "visited". By marking each child node as visited, this will help us know that we have already seen that node before, and won't accidently push us into an infinite loop if the graph was cyclic. In addition to marking each child node as visited, we want to place any of its children that have not yet been visited into the queue.
6. The process will complete until the queue is empty.
7. Once the while loop breaks, we can then return the `order` list. This order list will contain, in order, of all the nodes that we traversed too.

A few things to note about breadth-first traversals:

1. If there are any disconnected nodes (such as islands) with the graph data structure, they will not be traversed.
2. Breadth-first only outputs the nodes that are connected in some relation to the root that you pass in.

Depth First

In a depth first traversal, we approach it a bit different than the way we do when working with a depth first traversal of a tree. Similar to how the breadth-first uses a queue, we are going to use a Stack for our depth-first traversal.

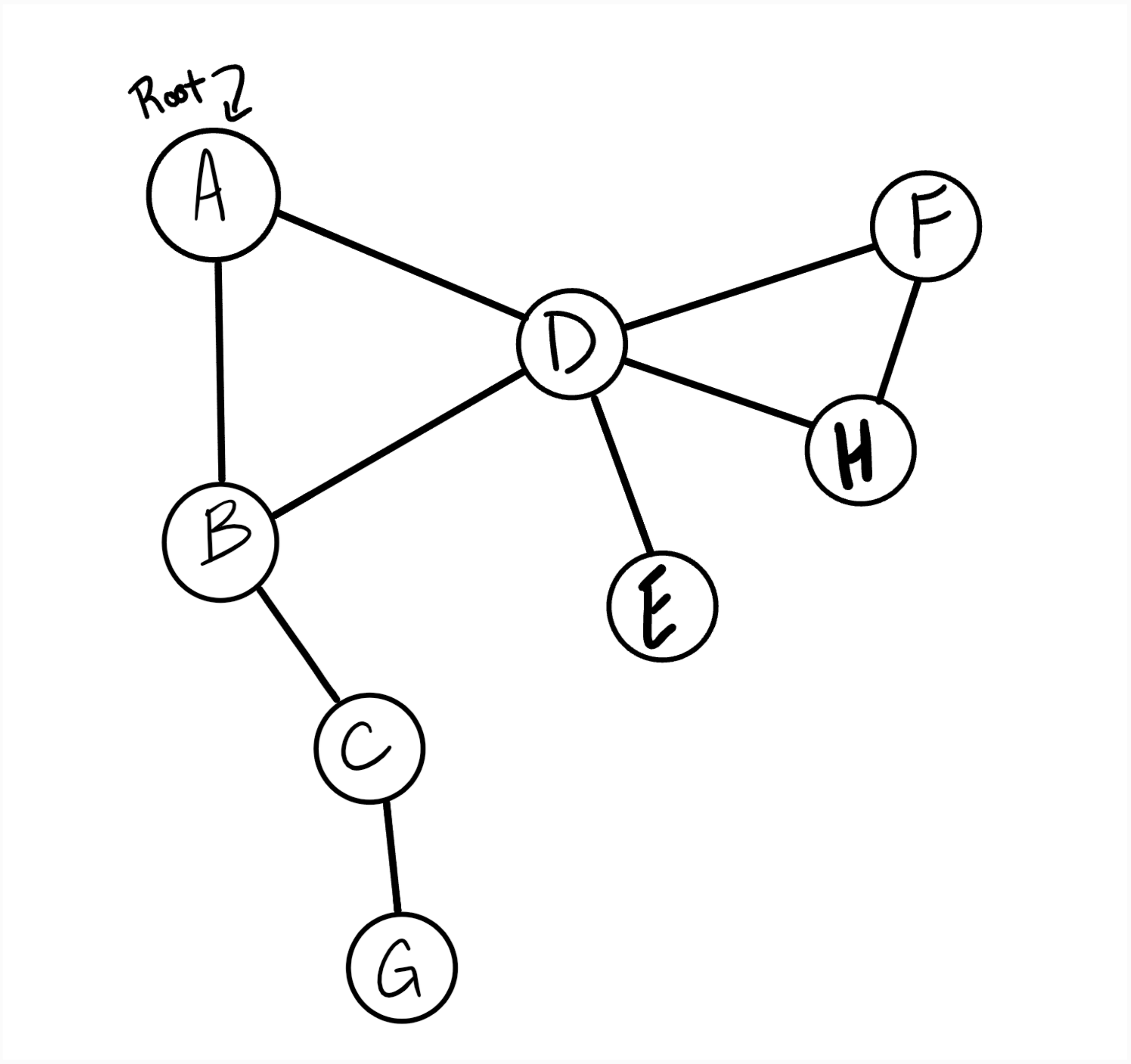
The algorithm for a depth first traversal is as follows:

1. `Push` the root node into the stack
2. Start a while loop while the stack is not empty
3. `Peek` at the top node in the stack
4. If the top node has unvisited children, mark the top node as visited, and then `Push` any unvisited children back into the stack.
5. If the top node does not have any unvisited children, `Pop` that node off the stack

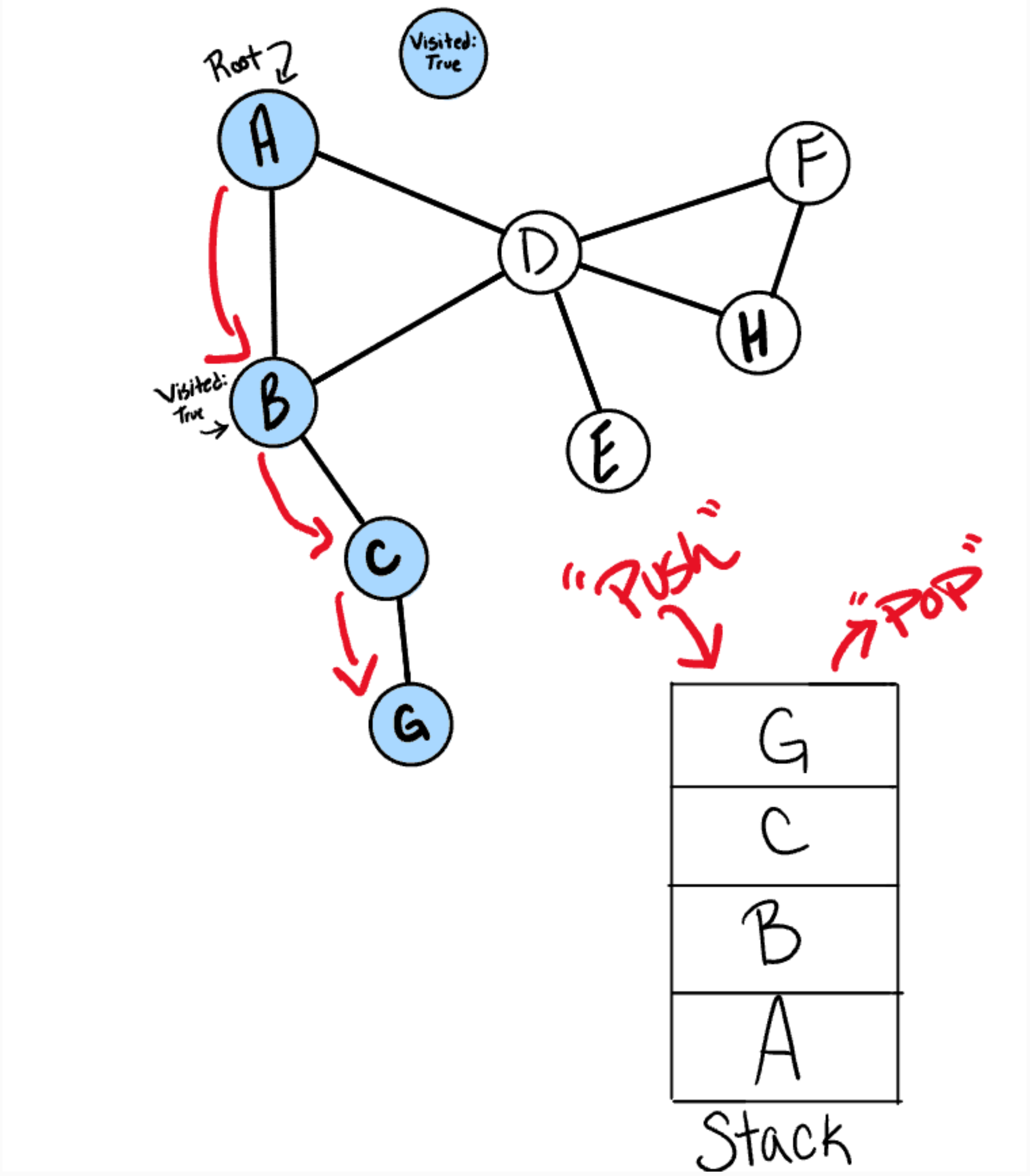
6. repeat until the stack is empty.

Let's look at the visual for a depth-first traversal.

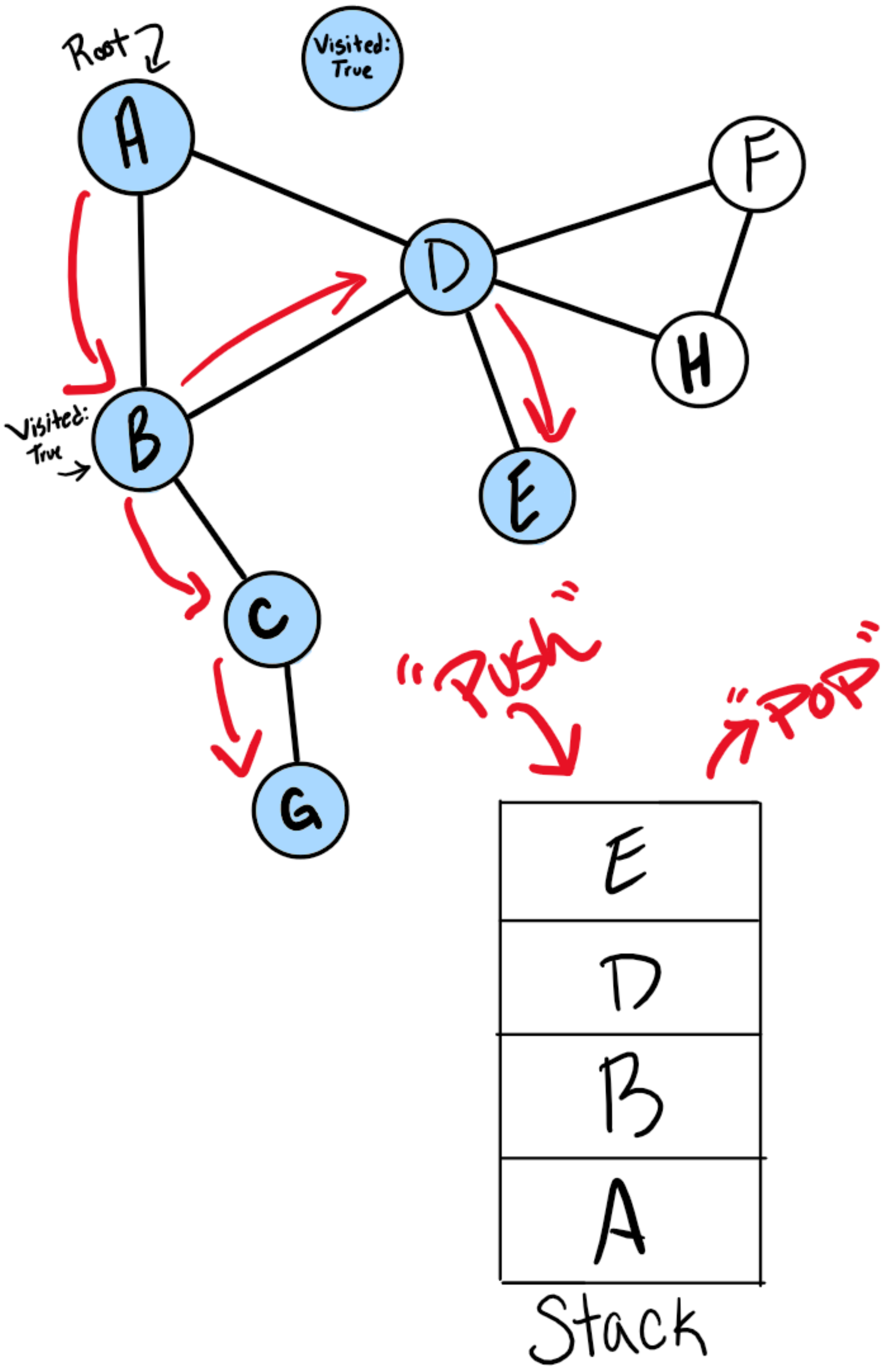
We will conduct a depth-first traversal on this graph:



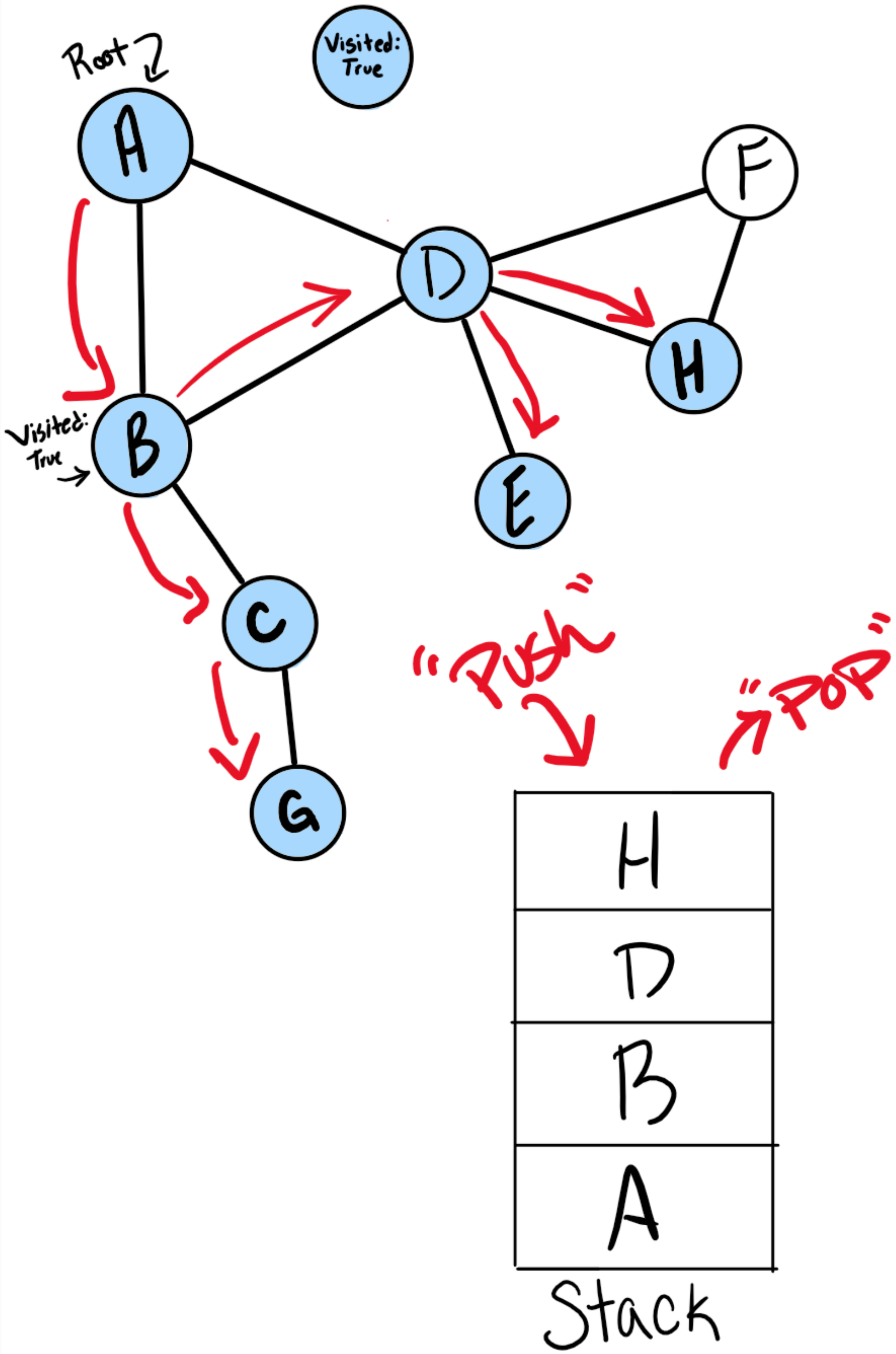
1. The first thing that we are going to do is look at the root. Following the algorithm we described above, we want to look at all of `Node A`'s children (`Node B` and `Node D`). Since `Node B` is first, we notice that it has not yet been visited, it then gets `Pushed` into the stack.
2. Before looking at the rest of `Node A`'s children, we will look at the children of `Node B`. and start visiting children that haven't yet been visited.
3. `Node C` then gets `Pushed` into the stack.
4. By the time we hit `Node G`, This is the state of our data structure:



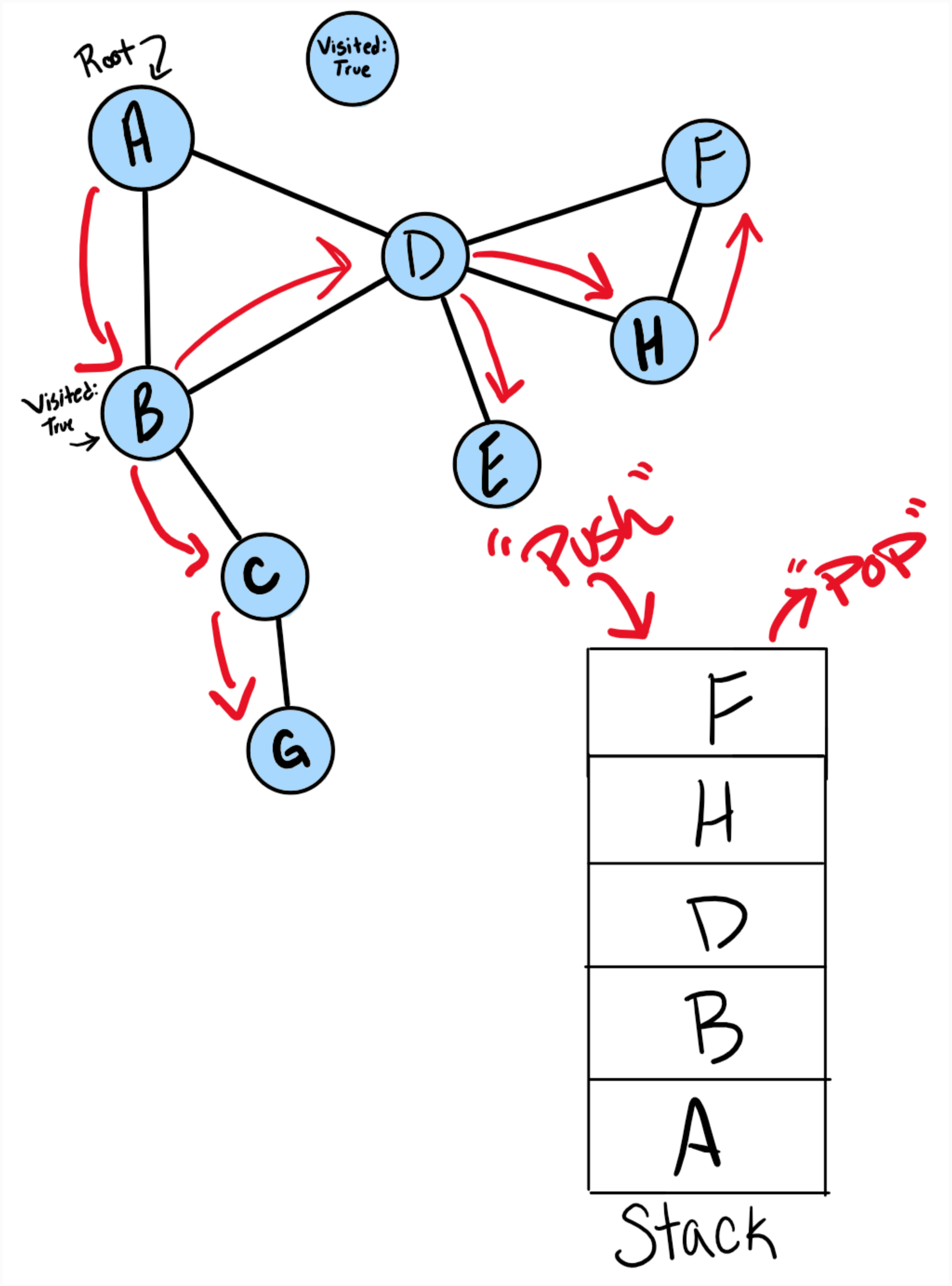
1. Moving back up the graph, we slowly `Pop` off the stack until we reach a child node that has not yet been visited. This just so happens to be `Node D`, since it is connected to `Node B`.
2. The algorithm keeps running and once we hit `NodeE`, this is the state of our program:


Root
Visited: True
A
B
C
D
E
F
G
H
Visited: True
"Push"
"Pop"
E
D
B
A
Stack

1. Notice how `Node D` has 3 children. We already visited `Node E`, so we move onto `Node H`.


Root
Visited: True
A
B
Visited: True
C
G
D
F
H
E
"Push"
"Pop"
H
D
B
A
Stack

1. Finally, we can see that `Node H` has a child, so we can wrap up the traversal by visiting `Node F`.



1. `Node F` gets popped off the stack since it has no children.

2. All three of the children of `Node D` have been visited so it also gets `Pop` ‘d off the stack
3. Finally, `Node A` gets popped off the stack to complete the algorithm.

Real World Uses of Graphs

Graphs are extremely popular when it comes to it’s uses. Here are just a few examples of graphs in use:

1. GPS and Mapping
2. Driving Directions
3. Social Networks
4. Airline Traffic
5. Netflix uses graphs for suggestions of products

There is a lot to graphs, and a lot you can do with them. Let this be document be used as an introduction to graphs. There is a lot more to them then described, so start exploring, and have fun with them!