- Directed: connections between nodes have a directions
- Weighted: connections between nodes have attached value
- Adjacency Matrix
    - Space: n^2 elements for n vertices
- Adjacency List
    - Space: Number of edges [2*(number of edges) if undirected] + number of vertices
- Breadth First Search (BFS)
    - Iterative
    - Stored in Queue
    - Runs in O(|V|+|E|)
    - Shortest path (unweight)
    - Testing bipartiteness
    - Tree Traversal
        - Level-order
- Depth First Search (DFS)
    - Recursive or Iterative
    - Stored in Stack
    - Runs in O(|V|+|E|)
    - Topological sorting
    - Strongly Connected Components
    - Tree Traversal
        - In-order, Pre-order, Post-order
- Inorder Traversal (left-current-right)
    - Start at root, go all the way down on the left, then work up going to down the right side of any node already visited all the way to the bottom.
    - Entire left side will be done before the right side is started

```
public void inorderTraversal(TreeNode root) {
	if (root != null) {
		inorderTraversal(root.left);
		System.out.print(root.data + " ");
		inorderTraversal(root.right);
	}
}
```

Preorder Traversal (current-left-right)

- Start at root again, except we append things to the list as we go down the tree. Inorder appends things to list when we get back to it, this appends when first discovered

```
public void preorderTraversal(TreeNode root) {
	if (root != null) {
		System.out.print(root.data + " ");
		preorderTraversal(root.left);
		preorderTraversal(root.right);
	}
}
```

Postorder Traversal (left-right-current)

- Root goes in last as all children of a node are recorded before the node itself is recorded.

```
public void postorderTraversal(TreeNode root) {
	if (root != null) {
		postorderTraversal(root.left);
		postorderTraversal(root.right);
		System.out.print(root.data + " ");
	}
}
```

Level-Order Traversal

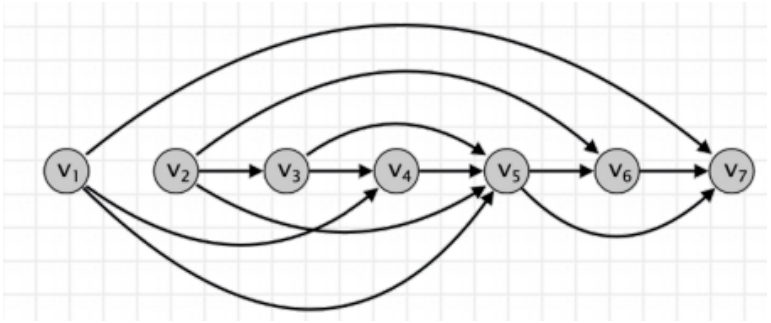
- Used by BFS, visit everything in a level before continuing

```
public void levelorderTraversal(TreeNode root) {
	if (root == null) {
		return;
	}
	Queue<TreeNode> queue = new LinkedList<>();
	queue.add(root);
	while (!queue.isEmpty()) {
		TreeNode node = queue.remove();
		System.out.print(node.data + " ");

		if (node.left != null) {
			queue.add(node.left);
		}
		if (node.right != null) {
			queue.add(node.right);
		}
	}
}
```

Definitions

- $\pi$(v) means parent of v
- Strongly Connected Components:
  - Only in directed graphs, things are considered strongly connected if you can reach u from v and v from u. (mutually reachable).
- Strongly Connected Graph:
  - Means that every every vertex can be reached from every other vertex.
- Cycle: When a loop can be made that starts and ends at the same node without revisiting any other node in the graph.
- DAG: Directed Acyclic Graph, there are no directed cycles
- Topological order:
  - An ordering of its nodes as v1, v2, ..., vn such that for every edge (vi, vj) we have i < j




  - Can be used in scenarios where one task must occur before another. (course prereqs, job pipeline)
  - If graph G has a topological ordering, then it also is a DAG and vice versa.

BFS: Testing Bipartiteness

- An undirected graph G = (V,E) can be called bipartite if the nodes on the graph can be colored with 2 colors in such a way that no nodes of the same color directly connect to one another.
- If there is an odd-length cycle in the graph, it cannot be bipartite
- One of the two following is true for determining bipartiteness
  - No edge of G joins two nodes of the same layer, and G is bipartite.
  - An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).
- We can modify the BFS algorithm to color each neighbor with the opposite color when it explores a node.
- If a neighbor has already been colored (i.e., visited), and has the same color, then return false.
- If the BFS can traverse the entire graph and color all nodes, then return true.

How to Obtain Topological-sort:

- Call DFS to compute finishing times for each vertex v.
- As each vertex is finished, insert it onto the front of a linked list
- Return the linked list of vertices
- Can also be done using BFSstarting from a node with no entering edge (no edge goes into this node, it is like a base root for the rest of the graph)
- Both of these run in O(m+n)

Minimum Spanning Tree (MST)

- 2 algos to find this, Kruskal and Prim (both are greedy)
- A cut is when we make a partition of a graph
- A cut respects a set of edges if there is no edge in the set that crosses the cut
- A light edge is one that has the minimum weight of all edges that cross a given cut cut
- Kruskal's
  - Add a safe edge to the tree that is the lightest edge connecting two distinct components (one of them must not already be in the tree)
- MSTs are always acyclic
- Prim's
  - Add a safe edge to the tree by selecting the least-weight edge connecting the tree to a vertex not already in the tree.
- Both run in O(|E|log|V|)
- Data Structure used
  - Kruskal: Disjoint-Set (Union-Find)
  - Prim: Priority Queue (Binary Min-Heap)