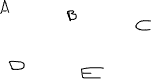
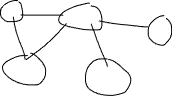
Graphs

* In the beginning, there were LISTS.
* A LIST always has one successor & one predecessor item.
* If we relax the definition of a list to have multiple successor elements, we end up with TREES.
* In a tree, a node can have multiple children (successor elements), but only one parent (predecessor element).
* Suppose we relax that constraint --- you can have multiple parents and multiple children.

We get a ADT called a GRAPH.

* Draw a sample graph. 🡪 this one
* Now each node can be connected to any  
  other number of other nodes.



* Graphs are useful for modeling any sort of set of relationships between pairs of items.
  + e.g., imagine these ABCDE are cities, and the lines between them represent roads.
  + or maybe ABCDE are airports, and the lines represent flights.
  + or maybe ABCDE are people, and the lines represent friendships on a social networking site.
* So there are two concepts in a GRAPH, that we care about: the CIRCLES, and the LINES connecting them.
* The circles, depending on which textbook you use, are called either Vertices or Nodes. (Vertices is the mathematical term, whereas Nodes is the term you typically see in a programming context, because they are like nodes in a LL or tree).
* The lines between them are called Edges (or sometimes Arcs).
* In a graph ADT, these are the only two things that matters:
  + you have a collection of Vertices [these vertices might be represented by numbers or strings or something else], (stored in some data structure),
  + and a collection of Edges, which are just pairs of Vertices, (stored in some data structure).
* Draw graph above:
  + So the abstract representation of the graph above would be:
  + Vertices = {A, B, C, D, E}
  + Edges = {(A, B), (A, D), (B, D), (B, E), (B, C)}
* The visual representation of a graph [where the nodes are located on the page] is not usually part of its data structure, so you can draw the graph above in a gazillion different ways.
* In lists and trees, the links (edges/arcs) always had a direction (predecessor/succ), parent/child. In graphs, sometimes the edges have an associated direction, and sometimes they don’t, depends on the relationship we’re trying to model.
* DIRECTED vs UNDIRECTED.
  + For instance, if we're using a graph to model twitter, if the vertices are people and the edges represent who follows who. But following is not a reciprocal relationship, so you would represent this with a directed graph, so for every edge, you'd know who the follower was and the followee.
  + In a directed graph, you may have one arrow in each direction between two nodes (2 people on twitter who follow each other).
  + Other examples: city with one-way streets, course pre-requisites.
  + On the other hand, some situations the directionality is irrelevant.
    - Facebook friendship.
    - Streets where you can always drive in both directions.
* Strange situations that sometimes arise but we won't really discuss here:
  + Self loops are OK as well. Parallel edges are OK.
* Graphs can also be **weighted** or **unweighted**. A weighted graph means every edge has a value (usually a number) associated with that edge called its weight.
* Whether a graph is weighted or not has nothing to do with whether its directed or not, so you can have four different kinds of graphs.
  + Weights usually give information about the relationship between the vertices that are connected by that edge.  
    A picture containing shape

    Description automatically generated
  + A picture containing shape

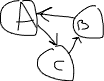
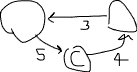
    Description automatically generated
  + Driving times in hours.
  + But weights can also be things like prices, travel times.
* Terminology:
  + A vertex is **adjacent** to another vertex if there is an edge to it from the other vertex.
  + e.g., StL is adjacent to LR, Memphis, and Nashville.
  + Adjacency definition is important when we're talking about directed graphs.
  + A 🡪 B B is adjacent to A, but A is not adjacent to B.
  + A path is a sequence of vertices where each vertex is adjacent to its predecessor.
  + e.g., Path: Memphis, Nashville, StL, LR.  
    also: Memphis, DC, Dallas, Chicago, Memphis, Chicago.
  + So yes, a path may have repeated vertices.
  + A **simple path** has no repeated vertices, except possibly the first and last vertices may be the same.
  + A **cycle** is a path where the start & ending vertices are the same.
  + an undirected graph is **connected** if there is a path from every vertex to every other vertex
  + a directed graph with this property is called **strongly connected**

**Graph representations**.

To discuss graphs at the implementation level, we often will need to refer to the vertices and edges in the graph. So we will call the set of vertices in a graph "V" and the set of edges "E".



[Draw simple graph:]



So if I ask you "In this graph, what is V?" you'd say {A, B, C} and if I ask you, "What is E?" you'd say it's the set {(A, B), (B, A), (A, C), (C, B)}

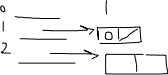
Adjacency matrix representation

* We number the vertices 0, 1, 2, 3...
* Create a square matrix of |V| rows and |V| columns.
* Unweighted: If there is an edge from vertex A to vertex B, then we put a 1 in the matrix at location matrix[A][B]. (where "A" and "B" are the numbers of the vertices).
* Weighted: Put the weight at location [A][B].
* Draw for graph above.

Adj list representation:

* A few different ways to do it.
* [Most books: ] One way is to again number the vertices 0, 1, 2, 3 and create a an ArrayList, of LinkedLists. Each index in the ArrayList corresponds to a vertex, and each linkedlist holds a list of vertices that can be reached from the vertex in the ArrayList.
  + Draw picture.

ddd



ddd

ddd

d

* Now this is a nice representation, but one drawback is if a user wants to know how far it is from A to B, first they have to know that A is vertex "0" and B is vertex "1". And for that we need some other data structure to store the correspondence between the vertex letters and the vertex numbers.
* What I see more often in the real world is not using this extra step of numbering the vertices. If the user is already using numbers, fine. If they are using letters, fine. If they are using strings (like names of cities), fine.
* What we'll do is instead of using an arraylist to store the linked lists, we'll use a map, where the key is whatever data type the user is using to represent the vertices.
  + e.g., if they're using letters, like above, we'll have a Map(keys=char [or string], value=Linkedlist)
  + Redraw above pictures.
* In the real world, there are many other types of representations as well. Most are based on one of the above representations.
  + example: Maybe we need to store multiple pieces of info for each edge, like if the vertices are cities, maybe we need to store the distance between them **and** the time it takes to drive it. Or maybe we need to store the speed limit on the road between them. So we might have multiple pieces of information in these linked lists.
  + Another common variation is to switch out the LL for something faster, like a BST or a Hashtable. This is useful in very dense graphs, where each node has many connections to other nodes.
* If time: Graph API
  + Add vertex
  + Add edge
  + Test if graph contains vertex
  + Test if graph contains edge
  + Remove vertex
  + Remove edge
  + Get weight of an edge.
  + Get all vertices connected to another vertex.
* Real world example 1 – Dijkstra project
* nodes are letters of the alphabet
* integer distances between edges.
* Solution: make edge class that holds two strings, plus a int weight.
* Then make Map<String, Set<Edge>> edges.
* Real world example:
* Streets – with street names and speed limits.
* I know latitude longitude coordinates of each intersection, plus a unique ID.
* Solution:
  + Make Intersection class {holds unique ID, lat, long}
  + Make Street class {holds starting unique ID/Intersection, ending unique ID/Intersection, street name, speed limit}
  + Make Map<Intersection, Set<Street>> edges WITH Map<unique ID, Intersection>  
    OR Map<uniqueID, Set<Street>>
* **List** ADT
  + Maintains a collection of items in a specific sequential order, usually where the user may explicitly control the order. Each item in the list has an immediate successor item (except the last item), and an immediate predecessor item (except the first item).
  + Possible implementations:
    - An array (like ArrayList/RArrayList), where usually the order of the items is controlled through an integer index. The user can choose to insert/add/remove items at any index.
    - A linked list, including singly-linked lists and doubly-linked lists, where the user may insert/add/remove items anywhere within the list.
* **SortedList** ADT
  + Maintains a collection of items in sorted order. The user does not have control over where an item goes in the list; the list itself ensures it is always kept sorted.
  + Possible implementations:
    - An array, where the programmer enforces that the array is kept in sorted order.
    - A linked list (singly/doubly), where the programmer enforces that the list is kept in sorted order.
* **Stack** ADT
  + Maintains a collection of items in a specific sequential order. The "push" operation adds an item to the collection, and the "pop" operation removes the most recently-added item that has not yet been removed.
  + Possible implementations:
    - Arrays or linked lists, usually where pushes and pops all happen at one end of the array or list.
* **Queue** ADT
  + Maintains a collection of items in a specific sequential order. The "enqueue" operation adds an item to one end of the collection, and the "dequeue" operation removes an item from the other end.
  + Possible implementations:
    - Arrays or linked lists, usually where enqueues happens at one end and dequeues happen at the other. Circularly-linked lists are also possible.
* **Set** ADT
  + Maintains a collection of items in no specific order. The **Set** ADT serves as an abstraction of a finite set in mathematics. Items may be added to the set, removed from the set, and one may test whether an item is a member of the set or not. Sets are often optimized to make the test-for-membership operation very fast.
  + Possible implementations:
    - Usually a binary search tree or a hash table, though a regular array or linked list is possible (albeit slow).
* **Map** ADT (also called a Dictionary, Symbol table, or Associative Array)
  + Maintains a collection of key-value pairs in no specific order. The Map ADT allows one to quickly lookup a value based on its key. New key-value pairs may be added to the map, and a pair may be removed given its key.
  + Possible implementations:
    - Usually a binary search tree or a hash table, though a regular array or linked list is possible (albeit slow). In the BST or hash table implementations, the keys of the Map serve as the main piece of information that is used in less-than/greater-than comparisons in a BST, or to compute a hash function in a hash table. The value part of the key-value pair is carried along as extra information, but does not determine the pair's position in a BST or a hash table.
* **Priority Queue** ADT
  + **implementations: Heap (less efficient=array, linked lists)**
* **Graph ADT**
  + **implementations: adjacency list, adjacency matrix, and variations.**