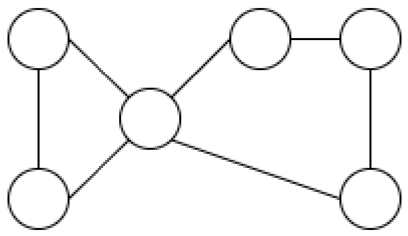
ADT: Graph



{inv: f tal que $f(G_1) = f(G_2)$ siempre que G_1 y G_2 sean isomorfos ^ $G = (V, E)$ }

Primitive operations:

	Input	Output
• Graph()	< >	-
• addVertex()	<data>	-
• addEdge()	<source, destination>	-
• removeVertex()	<data>	-
• removeEdge()	<source, destination>	-
• dfs()	<data>	tree
• bfs()	<data>	tree
• dijkstra()	<source>	pair <>, distance
• flodyWarshall()	< >	pair <>, distance
• prim()	< >	pair <>, distance
• kruskal()	< >	tree

• **Graph isomorphism** is a bijection of the vertices of one graph onto another, such that the adjacency of the vertices is preserved.

• A graph G = (V, E) is an ordered pair in which V is a non-empty set of vertices and E is a set of edges.

Graph(boolean)

" initializes an empty Graph object with the ability to store vertices that may or may not be directed, depending on the value passed as an argument to the constructor. "

{ pre: none}

{post: the constructor initializes correctly }

addVertex(T data)

" Add a new vertex to the graph only if a vertex with the same data doesn't already exist "

{ pre: the "data" parameter must be an object of type T.}

{post: a new vertex is created with the data provided and added to the graph. }

addEdge(T source, T destination)

" Adds an edge between two existing vertices in the graph. "

{ pre: the "source" & "destination" parameter must already exists.}

{post: adds an edge between them, proves the neighbor relationship between the vertices. If the graph is not directed, the edge is added in both directions to ensure that the neighbor relationship is bidirectional.}

removeVertex(value)

" Delate vertex from de graph. "

{ pre: the value must exist.}

{post:, it will be removed from the "adjacencyList" adjacency map, along with any edges associated with it. }

removeEdge(source, destination)

" Delate an edge from de graph. "

{ pre: source & destination must exist.}

{post: modified tree}

dfs(source, destination)

" Visits all vertices reachable from the start vertex and returns an ordered list of vertices visited in the DFS traversal. "

{ pre: must be an existing vertex in the graph, that is, it must be present in the adjacency map.}

{post: the result list that will contain all the vertices visited in the DFS traversal in the order they were found.}

bfs(vertex)

" Starts from a given start vertex and returns an ordered list of vertices visited during the BFS traversal "

{ pre: parameter must exist.}

{post:, it returns a list of vertices visited during the BFS traversal. The list represents the order in which the vertices were encountered during the traversal.}

dijkstra(source)

" find the shortest path from a source vertex to all other vertices in an undirected or directed weighted graph with nonnegative weights "

{ pre: parameter must exist ^ graph must be correctly represented}

{post: Pair<ArrayList<Vertex<V>>, ArrayList<Integer>> "

floydWarshall()

"finds the shortest paths between all pairs of vertices in a directed weighted graph."

{ pre: Graph must be intialize.}

{post: minimum distances between all pairs of vertices and the parent matrix corresponding to the shortest paths found in the graph.. }

prim()

" Prim's algorithm for finding the minimum spanning tree of an undirected weighted graph. "

{ pre: Graph must be intialize.}

{post: pair of objects representing the results of Prim's algorithm. }

kruskal()

" find the minimum spanning tree of an undirected weighted graph. "

{ pre: Graph must be intialize.}

{post: minimum spanning tree represented as a list of edges. }