**EDA NOTES**

**TOPIC 5.- PRIORITY QUEUE AND HEAP. HEAP SORT**

1.-INTRODUCTION

The Priority Queue (Cola de Prioridad) is a model for a data collection that allows to access to the element of highest priority.

Imagen que contiene captura de pantalla

Descripción generada automáticamente

2.-BINARY HEAP

2.1.-CHARACTERISTICS

Characteristics:

* Implementation based on an array.
* The average cost of **insertar** is a constant. The worst case is logarithmic.
* The average cost of **eliminarMin** is logarithmic. The worst case too.
* The cost of **recuperarMin** is constant.

2.2.-PROPERTIES

**Structural property**: a heap is a complete binary tree.

* Its height is maximum: .
* The cost of the algorithms is in the worst case logarithmic.
* The complete binary trees allow for an array representation.

**Sort property**: in a min heap, the element of the node is always smaller or equal than its children.

2.3.-ARRAY REPRESENTATION OF A COMPLETE BINARY TREE

Data is stored in an **array** following the **traversal by level**. The root is in the position 1. Given the i-th node:

* Position of its left child: 2 \* i (if 2 \* I <= size).
* Position of its right child: 2 \* i + 1 (if 2 \* i + 1 <= size).
* Position of its father: i / 2 (if i != 1).

Every path from the root to a leaf is a sorted sequence. The root is the node with the smallest element. Each subtree of a Heap is also a Heap.

3.-THE CLASS MONTICULOBINARIO (HEAP)

Imagen que contiene captura de pantalla

Descripción generada automáticamente

Steps of the method **insertar**:

1. The new element is inserted in the first available position of the array: *elArray[talla + 1]*.
2. The new element is compared with its predecessors and moved in order to have the sort property accomplished.

Imagen que contiene captura de pantalla

Descripción generada automáticamente

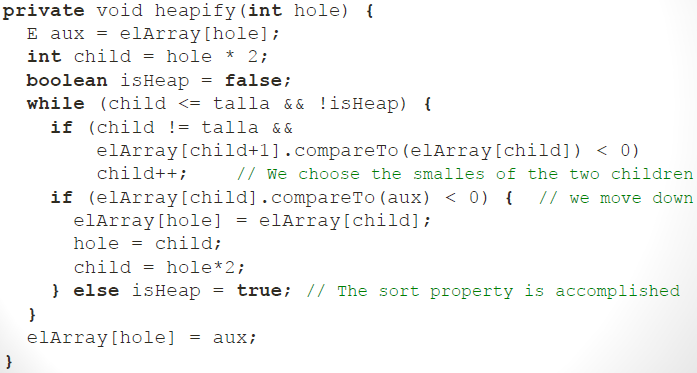
Temporal costs of the method **insertar**:

* Worst case: the complexity is O(log2N) if the added element is the new minimum.
* Best case: when the element to insert is greater than its father (only one comparison).
* It has been empirically proofed that on average, 2.6 comparisons are needed to insert a new element, so it is constant complexity.

Steps of the method **eliminarMin**:

1. The minimum is in the root. The root is substituted by the last element of the Heap.
2. The new root is moved down via its children in order to accomplish the sort property.

The method **heapify** (hundir):

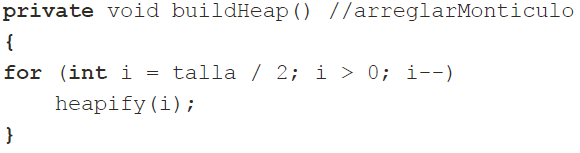


The method **eliminarMin**:

Imagen que contiene captura de pantalla

Descripción generada automáticamente

The method **buildHeap** (arreglarMonticulo): given a complete binary tree it allows to accomplish the sort property. It moves down all nodes in an inverse order than the traversal by levels. It has liner time complexity.

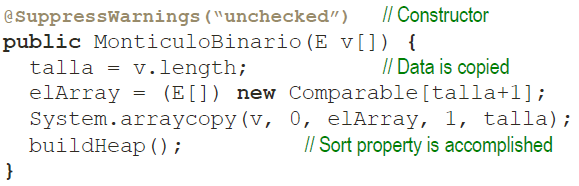


4.- FAST SORTING WITH HEAP SORT

The cost of **HeapSort** is O(N \* log2N). This algorithm is based on the properties of a heap:

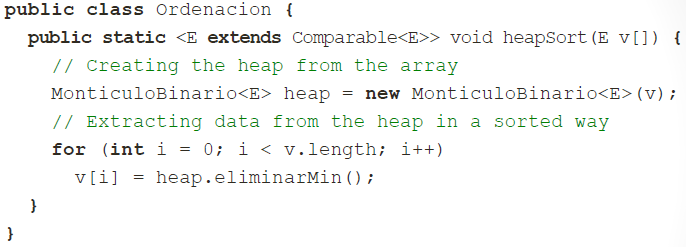
1. All the elements of an array to be sorted are stored in a heap.
2. The smallest element is extracted (root) in an iterative way.

The most efficient way to insert the data of an array in a Heap is with the method **arreglarMonticulo** (buildHeap).



The cost of the constructor is O(N), where N is the size of the array.

Algorithm:



The temporal cost of this algorithm is O(N \* log2N), and for the first k elements, O(N + k \* log2N).

**TOPIC 6.- MF-SETS**

1.-INTRODUCTION

A **disjoint-set data structure** is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. Given a set C previously fixed, a relation R defined in C is a subset of the Cartesian product CxC such that aRb means (a, b) R. A relation is an equivalence relation if the following three properties are accomplished:

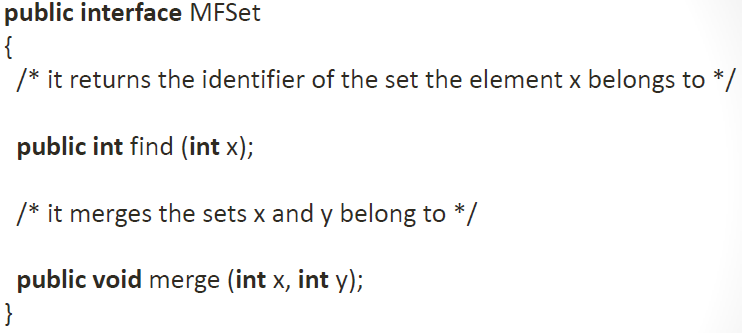
* Reflexive: aRa for each a C.
* Symmetric: aRb if bRa, for each a, b C.
* Transitive: aRb y bRc implies aRc, for each a, b, c C.

A set of elements can be partitioned in equivalence classes from the definition of an equivalence relation.

The problem of Union-Find (Merge-Find) has a search space the set of the possible partitions of a set C. The aim is to develop an efficient data structure in order to group together n different elements in a collection of k disjoint sets. There are two kind of operations:

* **Union (Merge)** of two disjoint sets.
* **Search (Find**) to what set a given element belongs to.

An MF-Set is a data structure of type set in which the elements are organised in disjoint sub-sets and the number of elements is fixed (no elements are added nor deleted).



2.-REPRESENTATION OF MF-SETS

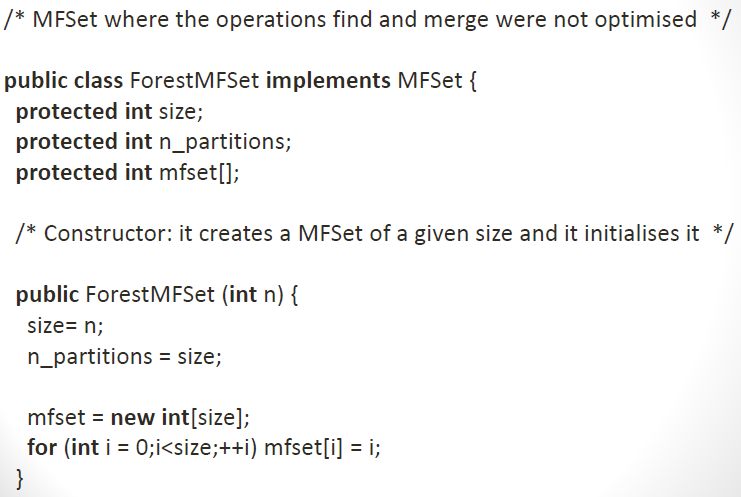
As an array:

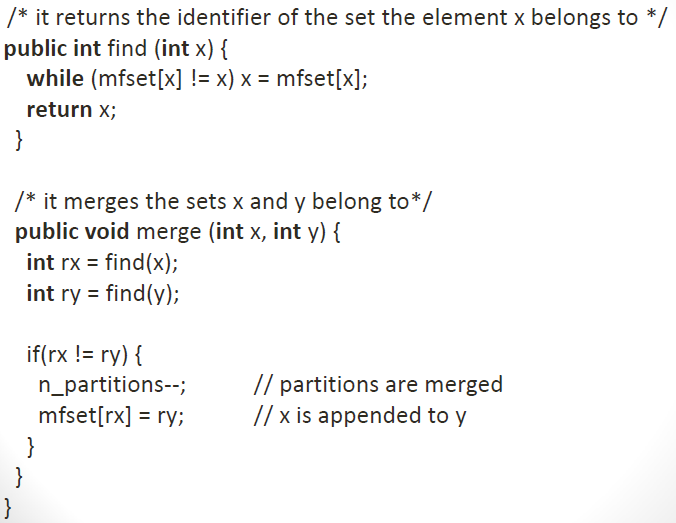
* M is an array of int of size n = |C|.
* M[i] indicates directly the set of class the element i belongs to.
* The complexity of **find(i)** is O(1).
* The complexity of **merge(i, j)** is O(n).

As disjoint-set forests:

* Every **sub-set** is a **tree**:
  + The nodes of the tree are elements of the set.
  + **Every node refers to its father**.
  + **The root of the tree can be used to represent the partition** or class.
* The **MF-Set** is represented as a **collection of trees**, a **forest**.
* The trees are not necessarily binary trees but the representation is easy and only a reference to the father is needed: M = array[n] of int.
  + M[x] is the father of the element x.
  + If M[x] = x, x is the root of the tree of the forest.

Operations with MF-Sets:





A sequence of m operations Merge and Find has a complexity O(m \* n) in the worst case.

3.-IMPROVING EFFICIENCY

Complexity can be improved if the height of the tree is reduced: m operations Merge and Find will have a quasi linear complexity with m:

* **Union by rank**.
* **Path compression**.

With these improvements the complexity of the operations is practically constant.

3.1.-UNION BY RANK

Merge is done in a way that the root of the tree whose height is smaller becomes the child of the tree whose height is higher. If the heights are different, the height after the merge is the height of the higher tree. In case that the merge is with two trees of the same height, the final height is increased by 1. The value of the height of each tree of the forest is kept, for instance, in the array, the value associated to the root is usually a negative number.

If MF-Set[i] < 0, i is the root of the tree and |MF-Set[i]| - 1 is the height of the tree.

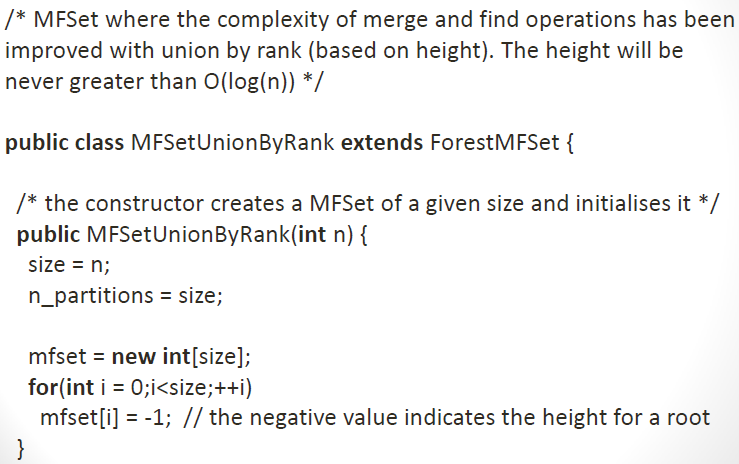


Imagen que contiene captura de pantalla, texto, interior

Descripción generada automáticamente

3.2.-PATH COMPRESSION

The effect of the path compression on Find is that each node of the path from x to the root refers directly to the root of the tree.

Imagen que contiene interior, captura de pantalla

Descripción generada automáticamente

Imagen que contiene captura de pantalla

Descripción generada automáticamente

3.3.- COMBINING STRATEGIES

It may happen that path compression diminishes the height of the tree. In this case, the height stored in the array does not necessarily represent the real height of the tree and it is an upper-bound of the real height O(log n) that in practice is much less due to the path compression.

Imagen que contiene captura de pantalla

Descripción generada automáticamente

**TOPIC 7.- GRAPHS**

1.-INTRODUCTION

**Binary relation** between data of the collection:

* A relation R over a set S is defined as a set of pairs (a, b) / a, b S
* If (a, b) R, it can be written as “a R b” and it denotes that a is related with b.

A **directed graph** (dg) is a pair G = (V, E):

* V is a finite set of **vertices** (or nodes).
* E is a set of directed **edges**, where an edge is an ordered pair of vertices (u, v): u -> v.

A **non directed graph** (ndg) is a pair G = (V, E):

* V is a finite set of **vertices**.
* E is a set of non directed **edges**, where an edge is a pair of non directed vertices (u, v) = (v, u), u != v: u – v.

A **labelled graph** is a graph G = (V, E) where a function is defined f: E -> L, with L is a set whose components are called **labels**.

A **weighted graph** is a labelled graph with real numbers.

Being G = (V, E) a graph, if (u, v) E, we say that the vertex v is adjacent to the vertex u. In a non directed graph the relation is symmetrical.

The **degree of a vertex** in a non directed graph, is the number of its incident edges (or adjacent vertices). The degree of a vertex in a directed graph is the sum of its outdegree and its indegree. The **degree of a graph** is the maximum degree of its vertices.

A **path of length** k from u to u’ in a graph G = (V, E) is a sequence of vertices <v0, v1,…,vk> such that:

* V0 = u and vk = u’.
* ꓯ i : 1…k : (vi-1, vi) E.
* The length k of the path is the number of edges.
* The length of the path with weights is the sum of the weights of the edges of the path.

If there exists a path P from u to u’, we say that u’ is **reachable** from u via P.

A **cycle** is a path <v0, v1,…,vk>:

* Starting and ending in the same vertex (v0 = vk).
* Containing at least an edge.

A path or cycle is **simple** if all its vertices are different. A **loop** is a cycle of length 1. A graph is **acyclic** if it does not contain cycles.

The **connected components** in a non directed graph are the equivalence classes of vertices under the relation “being reachable”. A non directed graph is connected if ꓯ u, v V, v is reachable from u. That is if it has just one connected component.

The **strongly connected components** in a directed graph are the equivalence classes of vertices under the relation of “being mutually reachable”. A directed graph is strongly connected if ꓯ u, v V, v is reachable from u.

2.-REPRESENTATION OF GRAPHS

There exist two forms for representing a graph:

* If the graph is **disperse** (|E| <<< |V|2): **adjacency** **lists**.
* If the graph is **dense** (|E| ≈ |V|2): **adjacency** **matrix**.

2.1.- ADJACENCY MATRIX

A graph G = (V, E) is represented as a matrix of |V|x|V| of elements of type boolean:

* If (u, v) E -> G[u, v] = true, otherwise, false.
* Spatial cost O(|V|2).
* Time for access O(1).

2.2.-ADJACENCY LISTS

A graph G = (V, E) is represented as an **array** of |V| **lists** of vertices:

* G[v], v V, is the list of the adjacent vertices to v.

2.3.-IMPLEMENTATION

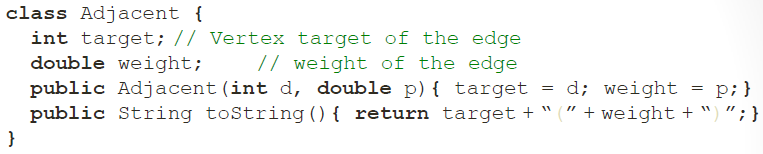


Imagen que contiene captura de pantalla

Descripción generada automáticamente

Imagen que contiene interior, pared, sujetando, mesa

Descripción generada automáticamente

3.-GRAPH TRAVERSALS

**Depth First Search** is like the **PreOrder** traversal (root, left, right), but vertices do not have to be visited twice.

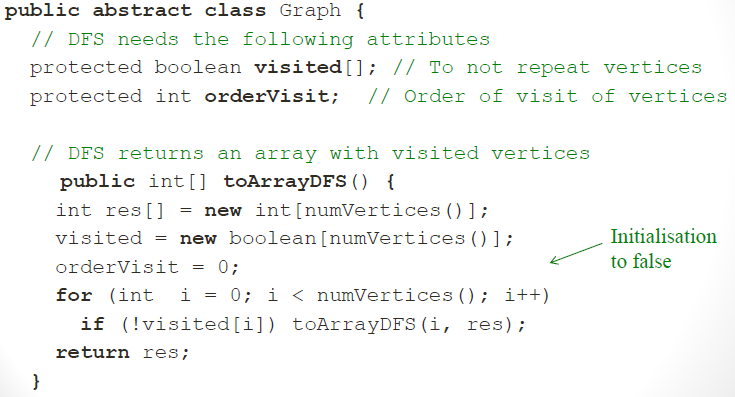


Imagen que contiene captura de pantalla

Descripción generada automáticamente

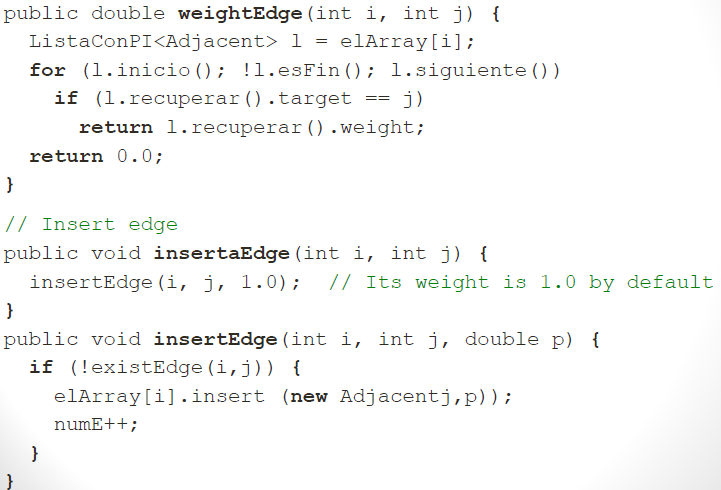
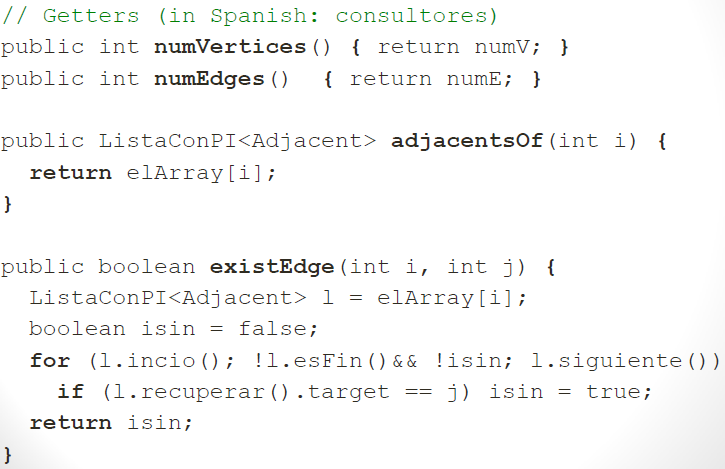
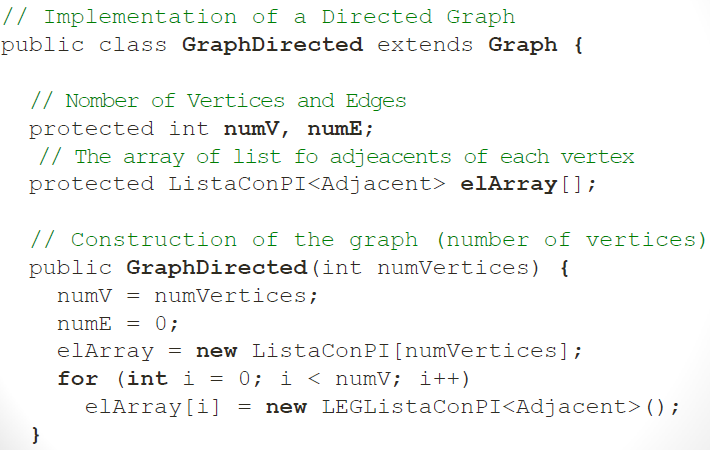
**Breadth First Search** is like the **traversal by levels** of a tree.

Imagen que contiene captura de pantalla, texto

Descripción generada automáticamente

Imagen que contiene texto

Descripción generada automáticamente

4.-IMPLEMENATION

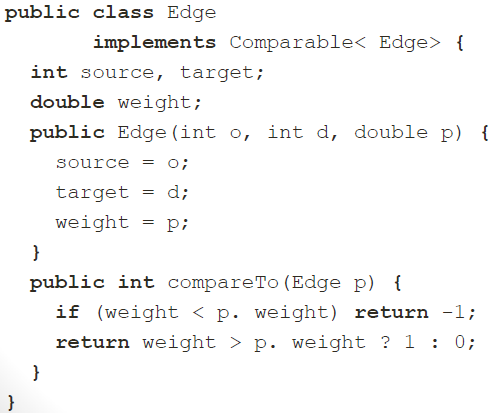
5.-MINIMUM SPANNING TREE (KRUSKAL)

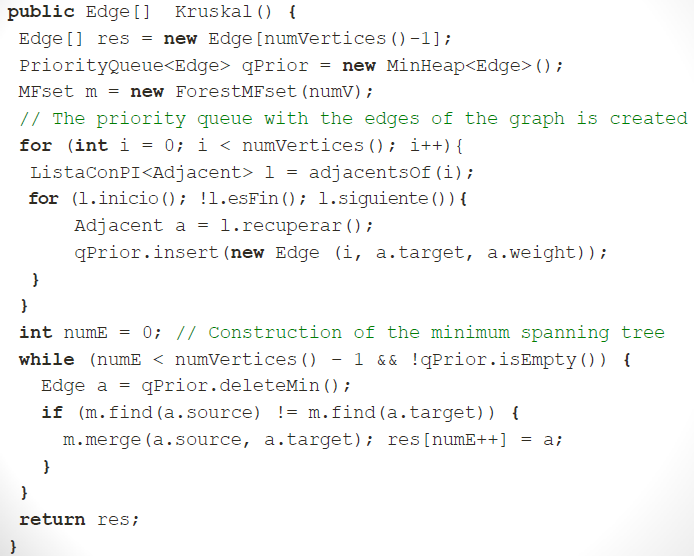
An undirected graph is connected if each pair of vertices is connected via a path. An acyclic undirected and connected graph is a tree. A spanning tree of a graph (V, E) is a tree (V’, E’) such that:

* V’ = V.
* E’ E.

**Kruskal’s algorithm**:

1. Store the edges in a priority queue.
2. Start from a graph without edges.
3. While |E| < |V| - 1 do:
   * Retrieve and delete from the priority queue the edge with the smallest cost.
   * Insert the edge in the graph if not cycles occur.





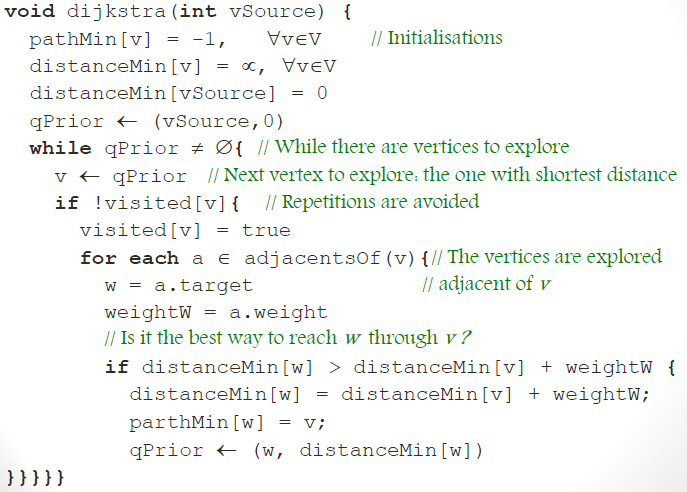
6.-SHORTEST PATH PROBLEM (DIJKSTRA)

The **weight of a path** is the sum of the weights of the edges that it passes through.

**Dijkstra** calculates the shortest paths from a vertex to the rest of vertices. It requires the weights of the edges to be positive. It stores the information in two arrays:

* **distanceMin**: it stores the minimum distance from the source vertex to the rest of vertices.
* **pathMin**: for each vertex it stores the previous vertex in the shortest path from the source vertex.

Pseudocode:



7.-TOPOLOGICAL ORDERS

A **topological sorting** is a linear sorting of a given directed acyclic graph, preserving the original partial sorting.

Imagen que contiene texto, captura de pantalla

Descripción generada automáticamenteImagen que contiene captura de pantalla

Descripción generada automáticamente