Java - El Lenguaje y la Plataforma

**Java abarca dos aspectos**

* Un Lenguaje de Programación orientado a objetos independientemente de la plataforma.
* Una plataforma: Los procesadores (CPU) requieren un sistema operativo, tal como Linux, Windows o Solaris para ejecutar programas, grabar archivos, etc. La combinación de procesador y sistema operativo se llama **plataforma de ejecución.**
* La Plataforma java aísla al programa del S.O. y del hardware sobre el que se está ejecutando. Provee “independencia” de la plataforma.
* La MVJ es una máquina de software que emula una maquina real. Es el corazón de la plataforma java. También provee un estándar.
* Los programas se escriben en **lenguajes de alto nivel**, como Java. Un programa escrito en lenguaje en un lenguaje de alto nivel, no puede ejecutarse directamente en la computadora, necesita a ser traducido a lenguaje de máquina. Esta traducción pueda realizarla un programa llamado **compilador.**
* El **lenguaje de maquina** o **código binario** consiste de instrucciones muy simples que la CPU de la computadora ejecuta directamente. Cada tipo de procesador tiene su propio lenguaje de máquina. Cada código binario es específico para cada plataforma.
* Una alternativa al compilador es un intérprete. Un intérprete es un programa que traduce y ejecuta un programa escrito en un lenguaje de alto nivel, instrucción por instrucción en el momento que se ejecuta (a diferencia del compilador que traduce el programa como un todo y genera un ejecutable). Los programas escritos en Java se **compilas** e **interpretan.**
* En Java, el código fuente es escrito en archivos con texto plano con extensión **.java**. Esos archivos son posteriormente compilados en archivos con extensión **.class** (puede ejecutarse en los distintos sistemas operativos) por el compilador java (**javac.exe**). Un archivo con extensión **.class** no contiene código nativo/especificado para un procesador determinado, sino que contiene **bytcodes** (el lenguaje de la máquina virtual de java). El **java.exe** es un programa que viene con la plataforma java, que permite ejecutar los **bytcodes**, es el intérprete de java.

**Ediciones de Java**

* **JSE (Java Estándar Edition)**: está diseñada para programar y ejecutar applets y aplicaciones de escritorio JAVA. Está compuesta por el JRE y el JDK.
* **JEE (Java Entreprise Edition)**: está diseñada para programar y ejecutar aplicaciones empresariales, caracterizadas por ser multiusuario y distribuidas. El procesamiento de estas aplicaciones se realiza en un servidor.
* **JME (Java Micro Edition)**: está diseñada para programar y ejecutar aplicaciones para dispositivos con recursos de computo limitados. Tienen poca memoria, pantallas muy chicas y muchas veces conexiones de red intermitente.

**La Plataforma Estándar de Java**

Java provee una plataforma de software para **desarrollar** programas y otra para **ejecutarlos**.

* **Plataforma de Desarrollo:** El Java Development Kit es la plataforma básica para desarrollo de programas java. Actualmente, el nombre oficial es Java SE o JSE. Incluye compilador, un intérprete, un depurador, etc. Que se usan desde la línea de comandos.
* **Plataforma de Ejecución:** Llamada JRE, provee todas las componentes necesarias para ejecutar programas escritos para JSE.

Programación Orientada a Objetos – Fundamentos

**El proceso de abstracción**

U modelo mental es una visión simplificada de como las cosas funcionan y cómo podemos interactuar con ella. La atracción es uno de los mecanismos que utilizamos para combatir la complejidad. La orientación a objetos, maneja la complejidad de los problemas del mundo real, abstrayendo su conocimiento y encapsulándolo en objetos.

**Definición de un Programa Orientado a Objetos**

Esta organizado como una comunidad de agentes interactuando, llamados objetos. Cada objeto cumple un rol. Cada objeto provee un servicio o ejecuta una acción. Es una entidad que provee información y operaciones relacionadas, que tiene sentido agrupar (**encapsulamiento**). Cuando un programa se está ejecutando, los objetos necesarios se van creando en la memoria HEAP

**¿Cuál es el objetivo buscado por la programación orientada a objetos?**

Organizar los datos del programa y el procesamiento asociado a ellos, en entidades coherentes, llamado objetos. Cada objeto abstrae un dato del programa y lo que puede hacerse sobre él.

**Ocultamiento de Información**

Los objetos son capsulas opacas, con una interfaz publica y una representación privada. Este concepto se conoce como ocultamiento de información. Permite eliminar de la vista cierta información propia del objeto, logrando mayor nivel de abstracción y facilitando los cambios del código. Se complementa con el encapsulamiento-

**Mensajes y Métodos**

Una acción es iniciada cuando un objeto, el emisor, envía un mensaje a un agente responsable de la acción, el receptor. El mensaje representa el requerimiento y es acompañado por información adicional (argumentos) necesarios para cumplir el requerimiento. El receptor en respuesta al mensaje ejecutara un conjunto de acciones o método para satisfacer el requerimiento.

**Mensajes y Métodos vs Llamadas a Procedimientos**

* Cuando se llama a un procedimiento NO hay un receptor.
* La interpretación del mensaje es determinada por el receptor y podría variar para diferentes receptores (polimorfismo).

**Clases e Instancias**

Una clase es un molde a partir de la cual se crean instancias con las mismas características y comportamiento. Todos los objetos son instancias de una clase.

Un objeto o instancia es una entidad de software que combina es estado/datos y comportamiento/métodos.

El estado de un objeto es todo lo que conoce de si mismo y, el comportamiento es todo lo que el objeto puede hacer.

**Herencia**

Las clases pueden ser organizadas en jerarquías de herencia donde, las clases subclases heredaran estado y comportamiento de las clases que se encuentran más arriba de la jerarquía, llamadas superclases. Las subclases pueden agregar nuevas variables y métodos y pueden cambiar el comportamiento de los métodos heredados.

**Variables de instancia**

Constituyen el estado de un objeto. Normalmente, las variables de instancia se declaran private, lo que significa que se puede acceder a ellas solo desde la clase donde se definen. La declaración de una variable incluye: un identificador, un tipo de dato primitivo o referencial, un especificador de acceso (opcional).

**Métodos de instancia**

Definen las operaciones que puedan realizar los objetos de un tipo de clase. Un método es un bloque de código. La declaración de un método especifica: un nombre, una lista opcional de argumentos, un tipo de retorno, un modificador de acceso (opcional).

**Tipos de datos en Java**

* **Tipos primitivos:** mantienen valores simples y NO objetos. Existen 8 tipos de datos primitivos: byte, short, int, long, float, double, char, boolean. Se inicializan con valores por defecto.
* **Tipos de una clase:** las variables que referencian a un objeto son llamadas variables referenciadas y contienen la ubicación de objetos en memoria. Se inicializan con el valor por defecto null.

**Clases wrappers**

Java proporciona clases wrappers para manipular a los datos primitivos como objetos. Los datos primitivos están envueltos en un objeto que se crea entorno a ellos, encapsulas a un único valor primitivo.

**¿Cómo se crean instancias de una clase?**

La creación e inicialización de un objeto involucra los siguientes pasos:

Se aloca espacio para la variable. Se aloca espacio para el objeto en la **HEAP** y se inicializan los atributos con valores por defecto. Se inicializan explícitamente los atributos del objeto. Se ejecuta el constructor. Se asigna la referencia del nuevo objeto a la variable.

**Variables de instancias**

Son creadas afuera del método, son creadas cuando el objeto es construido usando el new (). Estas variables existen, mientras exista el objeto.

**Variables locales**

Son creadas adentro de un método. Deben inicializarse antes de ser usadas. Los parámetros de los métodos también son variables locales. Son creadas cuando el método comienza a ejecutar y son destruidas cuando el método finaliza su ejecución.

**La palabra clave static**

Declara variables y métodos asociados con la clase en lugar de asociados a cada una de las instancias de la clase. Se debe anteponer la palabra static al tipo de dato de la variable o del retorno del método.

**Arreglos**

Un arreglo es un objeto que hace referencia a un conjunto de valores primitivos u objetos, a través de una única variable.

Permiten manipular un conjunto de valores del mismo tipo de datos usando un único nombre. Los datos se guardan en posiciones contiguas. La dimensión no puede cambiar.

Cada arreglo mantiene una propiedad length con el tamaño del mismo. El tamaño va de 0 a length-1.

Tiene dos recorridos: el for tradicional y el for-each (for (int elto: a) y se lee: para cada elemento elto de tipo int, en el arreglo a)

**Pasaje** **de** **Parámetros - Tipo de dato primitivo**

En Java los parámetros se pasan por valor (copia del parámetro actual).

Existen los parámetros formales que son los

parámetros en definición del método y los parámetros actuales/reales que son los parámetros en la invocación al método. Cuando el método finaliza, esas copias se descartan y los valores de las variables en el código principal son los mismos que antes.

Las clases wrapper y las instancias de String son inmutables, esto significa que su valor no puede cambiar, de manera que cuando se modifica el valor a una variable de estos tipos, se crea una nueva instancia con el nuevo valor y se le asigna a la variable.

**Pasaje** **de** **Parámetros - Tipo de dato referencial**

Podemos pasar como parámetro una referencia a un objeto y con ella cambiar el estado del objeto, pero no podemos cambiar a que objeto se refiere.

**Herencia y clases abstractas**

El término herencia se refiere al hecho de que una clase hereda los atributos (variables) y el comportamiento (métodos) de otra clase.

La herencia toma una clase existente y construye una versión especializada => reusabilidad de código.

Un método sobrescribe a otro método cuando se define en una subclase y coincide el nombre, tipo de retorno y lista de argumentos con un método ya definido en una superclase.

Es posible crear una clase nueva basada en una existente, agregándole características adicionales y modificándole el comportamiento a la superclase.

Tratar a una referencia de la clase derivada como una referencia de la clase base, se denomina upcasting. Con el upcasting, la conversión es hacia arriba en la jerarquía de herencia y se pierde el tipo específico del objeto. Para recuperar el tipo del objeto, se debe mover hacia abajo en la jerarquía y a esto se lo llama downcasting.

La clase Object es la raíz de todas las clases JAVA y está ubicada en el paquete java.lang. Cuando se declara una clase sin usar la palabra clave extends el compilador JAVA implícitamente agrega el código extends Object a la declaración de la clase.

El método public boolean equals(Object obj){}, compara la igualdad de 2 objetos. La versión original del método equals(), devuelve true si las dos referencias son iguales, es decir si apuntan al mismo objeto en memoria. Es equivalente a usar el operador ==. El método public String toString(){}, retorna la dirección del objeto como un String.

La intención del método equals(Object o) es comparar el contenido de dos objetos y la del toString() es producir una representación textual, concisa, legible y expresiva del contenido del objeto.

El objetivo de definir una clase abstracta es lograr una interface de comportamiento común para los objetos de las subclases (de la clase abstracta). Se espera que una clase abstracta sea extendida por clases que implementen todos sus métodos abstractos. En una clase abstracta pueden existir métodos concretos y métodos abstractos**. ¿Qué es un método abstracto?** Es un método que NO tiene implementación. Se define el nombre, el tipo de retorno, la lista de argumentos, termina con “;” y se antepone la palabra clave abstract. NO tiene código. **¿Para qué sirve un método sin código?** Para definir un comportamiento común para todos los objetos de las subclases concretas

Para declarar una clase abstracta se antepone la palabra clave abstract a la palabra clave class. Una clase abstracta es una clase que solamente puede ser extendida, no puede ser instanciada.

**Listas**

Una Lista es una estructura de datos en donde los objetos están ubicados en forma secuencial. A diferencia de la Pila y la Cola, en una Lista se puede “agregar” y “eliminar” en cualquier posición.

Puede estar implementada a través de: una estructura estática (arreglo) o una estructura dinámica (usando nodos enlazados).

Puede estar ordenada o no.

Algunas operaciones:

* elemento(int pos): retorna el elemento de la posición indicada
* incluye (Integer elem): retorna true si elem está en la lista, false en caso contrario
* agregarInicio (Integer elem): agrega al inicio de la lista
* agregarFinal(Integer elem): agrega al final de la lista
* agregarEn(Integer elem, int pos): agrega el elemento elem en la posición pos
* eliminarEn(int pos): elimina el elemento de la posición pos
* eliminar(Integer elem): elimina, si existe, el elemento elem
* esVacia(): retorna true si la lista está vacía, false en caso contrario
* tamanio(): retorna la cantidad de elementos de la lista
* comenzar(): se prepara para iterar los elementos de la lista
* proximo(): retorna el elemento y avanza al próximo elemento de la lista.
* fin(): determina si llegó o no al final de la lista, retorna true si no hay más elementos, false en caso contrario

Independientemente de la estructura de datos usada para implementar la lista, ambas responden al mismo conjunto de operaciones.

**Analizamos la implementación de Listas con elementos de tipo Integer y con elementos de tipo Object:**

Con Un Tipo de Dato: Ventajas: el compilador chequea el tipo de dato que se inserta. No se necesita hacer uso del casting. Desventajas: si se quisiera tener una estructura para cada tipo de datos, se debería definir una clase para cada tipo. Por ejemplo: ListaDeEnteros, ListaDeAlumnos, etc.

Con Object: Ventajas: Se logra una estructura genérica. Desventajas: El compilador pierde la oportunidad de realizar chequeos y se debe hacer uso de casting.

Usando tipos genéricos, es posible definir estructuras dónde la especificación del tipo de objeto a guardar se posterga hasta el momento de la instanciación. Para especificar el uso de genéricos, se utiliza <tipo>.

**Estructuras de Datos**

Una estructura de datos es una forma de almacenar y organizar los datos con el fin de facilitar el acceso y las modificaciones.

Estudiar formas inteligentes de organizar la información, información, de forma tal de obtener algoritmos eficientes.

Estructuras de datos: Listas, Pilas, Colas, Árboles Binarios, Árboles Generales, Heaps, Grafos

Algoritmos: Insertar, Borrar, Buscar, Caminos mínimos, Ordenación

Los programas reciben, reciben, procesan y devuelven datos, Necesitan organizar los datos de acuerdo al problema que vamos a resolver -🡪 Las estructuras de datos son formas de organización de los datos

Un programa depende fundamentalmente de la organización de los datos: cómo se organizan los datos está relacionado con: Implementación de algunas operaciones, La velocidad del programa, La memoria usada

**¿Qué es un algoritmo?** Es una secuencia de pasos que resuelven un problema. Es independiente del lenguaje de programación. La elección de un algoritmo y de la estructura de datos para resolver un problema son interdependientes

Caso: Buscar un elemento en un arreglo

Si el arreglo está desordenado -> Búsqueda secuencial

Si el arreglo está ordenado -> Búsqueda binaria: Comparo la clave con la entrada del centro. Si es menor, voy hacia la izquierda. Si es mayor, voy hacia la derecha. Si es igual, la encontré

**¿Cómo medir el tiempo?** En forma empírica: Se realiza a posteriori (Correr el programa para varios tamaños de la entrada y medir el tiempo.). En forma teórica: Se realiza a priori

Factores independientes del sistema: Algoritmo y Entrada de datos

Factores dependientes del sistema: Hardware: CPU, memoria, Software: lenguaje, compilador, etc, Sistema: SO, red, otras aplicaciones

Tiempo total: Suma del costo x frecuencia de todas las operaciones. Es necesario analizar el algoritmo para determinar el conjunto de operaciones. Costo depende de la máquina, del compilador, del lenguaje. Frecuencia depende del algoritmo y de la entrada

**Recursión**

Un objeto es recursivo cuando forma parte de sí mismo, es decir puede definirse en términos de sí mismo.

En programación, la recursividad es la propiedad que tienen los Algoritmos de llamarse a sí mismos para resolver un problema.

Soluciones recursivas:

* División sucesiva del problema original en problemas más pequeños del mismo tipo
* Se van resolviendo estos problemas más sencillos
* Con las soluciones de éstos se construyen las soluciones de los problemas más complejos.

**Constructores clase 4.1**

* Los constructores son códigos similares a los métodos que permiten definir el estado inicial de un objeto en el momento de su creación. La diferencia con los métodos es:
* Deben si o si tener el mismo nombre de la clase
* La regla de que los métodos deben comenzar en minúscula no se aplica para los constructores
* No retornan un valor
* Son invocados automáticamente
* Utilizando un constructor la inicialización está garantizada al momento de la creación ya que java siempre llama a un constructor al momento de crear el objeto.

**Constructor default (sin argumentos)**

Un constructor default es usado para crear un objeto básico, si la clase no tiene constructores el compilador inserta automáticamente uno default al momento de hacer el new nombre (). Esto no sucede si la clase tiene al menos 1 constructor, tenga este argumentos o no.

**Constructor con argumentos**

Son usados para inicializar elementos de la clase, sus valores especificados pueden ser invocados dentro del constructor tanto con this. Como sin él. Si el constructor es el único de la clase, el compilador no permitirá crear un objeto de otra forma que no sea usando el mismo constructor.

**Sobrecarga de constructores**

Cuando se requieren crear más de un solo constructor dentro de la clase

La sobrecarga de métodos permite que el mismo nombre sea usado en distintos tipos, el compilador se guía utilizando los tipos y argumentos para ver cuál utilizar.

**Uso del this() y this**

This() se utiliza cuando adentro de una clase hay más de un constructor, se pueden invocar entre ellos para evitar duplicar el código, el this(\*nombre\*) hace una invocación a otro constructor de la misma clase que coincida con la lista de argumentos. SIEMPRE DEBE ESTAR EN LA PRIMERA LINEA.

This se utiliza dentro del constructor para hacer referencia a una variable declarada en la misma clase

Si se llama a un constructor en el main con dos clases instanciadas iguales, el compilador agrega una referencia al objeto que está siendo manipulado para no confundirse entre objetos idénticos.

**Cadena de invocación de constructores**

Cuando tenemos clases y superclases, en cada constructor de una clase derivada debe existir una llamada a un constructor de la superclase

**El compilador Java, automáticamente invoca en cada constructor de una clase derivada, al constructor nulo de su clase base, si no se invocó ninguno explícitamente**.

Si un constructor no invoca a ningún constructor de la clase base, el compilador inserta la invocación al constructor nulo (super ()).

Si un constructor invoca explícitamente a un constructor de la superclase, debe hacerlo en la primera línea de dicho constructor.

**Super() y Super**

**Super ():**

invoca a un constructor de la superclase y debe estar si o si en la primera línea de código del constructor. Java garantiza la correcta creación de objetos ya que los constructores siempre invocan a los constructores de la superclase.

**Super:**

Todos los métodos de instancia disponen de la variable super (además de this), la cual contiene una referencia al objeto padre. La palabra clave super, permite invocar desde la subclase un método de la superclase.

**Arboles Binarios clase 5**

Un árbol binario es una colección de nodos, tal que:

* puede estar vacía
* puede estar formada por un nodo distinguido R, llamado raíz y dos sub-árboles T1 y T2, donde la raíz de cada subárbol Ti está conectado a R por medio de una arista
* Cada nodo puede tener a lo sumo dos nodos hijos.
* Cuando un nodo no tiene ningún hijo se denomina hoja.
* Los nodos que tienen el mismo nodo padre se denominan hermanos.

**Recorrido en arboles binarios**

**Preorden:**

Se procesa primero la raíz y luego sus hijos, izquierdo y derecho.

**Inorden:**

Se procesa el hijo izquierdo, luego la raíz y último el hijo derecho

**Postorden:**

Se procesan primero los hijos, izquierdo y derecho, y luego la raíz

**Por niveles:**

Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

* Denominaremos un árbol binario lleno cuando un árbol binario de altura h si cada nodo interno tiene grado 2 y todas las hojas están en el mismo nivel(h).
* Cantidad de nodos está dada por: Sea T un árbol **binario lleno** de altura h, la cantidad de nodos. N es (2 h+1 – 1)
* Árbol **binario completo:** Dado un árbol binario T de altura h, diremos que T es completo si es lleno de altura h-1 y el nivel h se completa de izquierda a derecha.
* Cantidad de nodos en un árbol **binario completo**: Sea T un árbol binario completo de altura h, la cantidad de nodos N varía entre (2 h) y (2 h+1 – 1).
* Profundidad: de ni es la longitud del único camino desde la raíz hasta ni (la raíz tiene profundidad cero).
* Grado: de ni es el número de hijos del nodo ni.
* Altura: de ni es la longitud del camino más largo de ni hasta una hoja

**Árbol de Expresión**

Un árbol de expresión es un árbol binario asociado a una expresión aritmética

donde:

* Nodos internos representan operadores
* Nodos externos (hojas) representan operando

**Aplicaciones**

En compiladores para analizar, optimizar y traducir programas

Evaluar expresiones algebraicas o lógicas (No se necesita el uso de paréntesis)

Traducir expresiones a notación sufija, prefija e infija

**Construcción de un árbol de expresión a partir de una expresión postfija:**

**Algoritmo:**

Tomo un carácter de la expresión mientras (existe carácter) hacer

si es un operando ►creo un nodo y lo apilo.

Si es un operador (lo tomo como la raíz de los dos últimos nodos creados) ► creo un nodo R - desapilo y lo agrego como hijo derecho de R - desapilo y lo agrego como hijo izquierdo de R - apilo R - Tomo otro carácter - fin

**Construcción de un árbol de expresión a partir de una expresión prefija**

**Algoritmo:**

ArbolExpresión (A: ArbolBin, exp: string)

si exp nulo ► nada

si es un operador ► creo un nodo raíz R - ArbolExpresión (subArbIzq de R, exp (sin 1 carácter) ) - ArbolExpresión ( subArbDer de R, exp (sin 1 carácter) ) si es un operando creo un nodo (hoja)

**Convertir una expresión infija en árbol de expresión**

se debe convertir la expresión infija en postfija (i) y a partir de ésta, construir el árbol de expresión (ii).

(i) Estrategia del Algoritmo para convertir exp. infija en postfija:

a) si es un operando ► se coloca en la salida.

b) si es un operador ► se maneja una pila según la prioridad del operador en relación al tope de la pila

operador con ► prioridad que el tope ► se apila operador con <= prioridad que el tope ► se desapila elemento colocándolo en la salida. Se vuelve a comparar el operador con el tope de la pila

c) si es un “(“ , “)” ► “(“ se apila“)” se desapila todo hasta el “(“, incluido éste

d) cuando se llega al final de la expresión, se desapilan todos los elementos llevándolos a la salida, hasta que la pila quede vacía.

**Árbol de Frecuencia Natural**

Un árbol de frecuencia natural (AFN) es un árbol binario asociado con ocurrencias, donde para todos los nodos que no son hojas, su valor es la suma del valor de sus dos hijos.

**Arboles Generales clase 7**

Un árbol binario es una colección de nodos, tal que:

* puede estar vacía
* puede estar formada por un nodo distinguido R, llamado raíz y **un conjunto de árboles** TK, K > 0 (subárboles), donde la raíz de cada subárbol Ti está conectado a R por medio de una arista.
* **Grado:** de ni es el número de hijos del nodo ni. El grado del árbol es el grado del nodo con mayor grado.
* **Profundidad:** de ni es la longitud del único camino desde la raíz hasta ni (la raíz tiene profundidad cero). La raíz tiene profundidad o nivel cero.
* **Altura:** de ni es la longitud del camino más largo de ni hasta una hoja. Las hojas tienen altura cero. La altura de un árbol es la altura del nodo raíz.

**Árbol General Lleno**

Dado un árbol **T** de grado **k** y altura **h**, diremos que **T** es lleno si cada nodo interno tiene grado **K** y todas las hojas están en el mismo nivel (**h**).

Es decir, recursivamente, T es lleno si: T es un nodo simple (árbol lleno de altura 0) o T es de altura h y todos sus sub-árboles son llenos de altura h-1.

**Cantidad de nodos en un árbol lleno:** Sea T un árbol lleno de grado k y altura h, la cantidad de nodos N es (k h+1-1) / (k - 1)

**Árbol General Completo:**

Dado un árbol T de grado k y altura h, diremos que T es completo si es lleno de altura h-1 y el nivel h se completa de izquierda a derecho.

Cantidad de nodos en un árbol completo: Sea T un árbol completo de grado k y altura h, la cantidad de nodos N varía entre (kh + k - 2) / (k-1) y (kh+1- 1) /(k-1)

**Representación: Lista de hijos**

La lista de hijos, puede estar implementada a través de: Arreglos (Desventaja: espacio ocupado) y Listas dinámicas (Mayor flexibilidad en el uso).

**Recorrido en arboles generales**

**Preorden:**

Se procesa primero la raíz y luego los hijos

**Inorden:**

Se procesa el primer hijo, luego la raíz y por último los restantes hijos

**Postorden:**

Se procesan primero los hijos y luego la raíz

**Por niveles:**

Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

Cola de prioridad

Una cola de prioridad es una estructura de datos que permite al menos dos operaciones:

Insert (inserta un elemento en la estructura) y DeleteMin (encuentra, recupera y elimina el elemento mínimo).

Implementación

* Lista ordenada: Insert tiene O(N) operaciones y DeleteMin tiene O(1) operaciones
* Lista no ordenada: Insert tiene O(1) operaciones y DeleteMin O(N) operaciones
* Árbol Binario de Búsqueda: Insert y DeleteMin tienen en promedio O(log N) operaciones

**Heap Binaria**

**Una heap es un árbol binario completo**

El número de nodos n de un árbol binario completo de altura h, satisface: 2h ≤ n ≤ (2h+1-1)

Demostración:

Si el árbol es lleno, n= 2h+1-1.

Si no, el árbol es lleno en la altura h-1 y tiene por lo menos un nodo en el nivel h.

La altura h del árbol es de **O(log n)**

**Propiedad estructural**

Dado que un árbol binario completo es una estructura de datos regular, puede almacenarse en un arreglo, tal que:

* La raíz este almacenado en la posición 1.
* Para un elemento que está en la posición i:
* El hijo izquierdo está en la posición 2\*ï.
* El hijo derecho está en la posición 2\*i+1
* El padre está en la posición i/2

**Propiedades de orden**

* **MinHeap:** El elemento mínimo está almacenado en la raíz. El dato almacenado en cada nodo es menor o igual al de sus hijos.
* **MaxHeap:** Se usa la propiedad inversa.

**Implementación del Heap**

**Una heap H consta de:**

* Un arreglo que contiene los datos.
* Un valor que me indica el número de elementos almacenados.

**Ventaja:**

* No se necesita usar punteros.
* Fácil implementación de las operaciones.

**Operación Insert**

El dato se inserta como último ítem en la heap (la propiedad de la heap puede ser violada).

Se debe hacer un filtrado hacia arriba para restaurar la propiedad de orden

**Insert: Filtrado hacia arriba (Percolate Up)**

* El filtrado hacia arriba restaura la propiedad de orden intercambiando k a lo largo del camino hacia arriba desde el lugar de inserción
* El filtrado termina cuando la clave k alcanza la raíz o un nodo cuyo padre tiene una clave menor.
* Ya que el algoritmo recorre la altura de la heap, tiene O (log n) intercambios.

**Operación: DeleteMin**

* Guardo el dato de la raíz.
* Elimino el último elemento y lo almaceno en la raíz.
* Se debe hacer un filtrado hacia abajo para restaurar la propiedad de orden.

**DeletMin: Filtrado hacia abajo (Percolate Down)**

* Es similar al filtrado hacia arriba.
* El filtrado hacia abajo restaura la propiedad de orden intercambiando el dato de la raíz hacia abajo a lo largo del camino que contiene los hijos mínimos.
* El filtrado termina cuando se encuentra el lugar correcto donde insertarlo.
* Ya que el algoritmo recorre la altura de la heap, tiene O(log n) operaciones de intercambio.

**Otras operaciones**

* DecreaseKey (x, Δ, H): Decrementa la clave que está en la posición x de la heap H, en una cantidad Δ.
* IncreaseKey (x, Δ, H): Incrementa la clave que está en la posición x de la heap H, en una cantidad Δ.
* DeletKey (x): Elimina la clave que está en la posición x. Puede realizarse DecreaseKey (x, infinito, H) o DeleteMin(H).

**¿Cómo construir una heap a partir de una lista de elementos?**

Para construir una heap a partir de una lista de n elementos:

* Se puede insertar los elementos de a uno, se realizan (n log n) operaciones en total.
* Se puede usar un algoritmo de orden lineal, es decir, proporcional a los n elementos (BuildHeap): Insertar los elementos desordenados en un árbol binario completo. Filtrar hacia abajo cada uno de los elementos.

**Algoritmo BuilHeap**

* Para filtrar: se elige el menor de los hijos. Se compara el menor de los hijos con el padre.
* Se empieza filtrando desde el elemento que está en la posición (tamaño/2): se filtra los nodos que tienen hijos. El resto de los nodos son hojas.

**Cantidad de operaciones requeridas**

* En el filtrado de cada nodo recorremos su altura.
* Para acotar la cantidad de operaciones (tiempo de ejecución) del algoritmo BuildHeap, debemos calcular la suma de las alturas de todos los nodos.
* **Teorema:** En un árbol binario lleno de altura h que contiene 2h+1-1 nodos, la suma de las alturas de los nodos es: 2h+1-1-(h+1)

**Demostración:**

Un árbol tiene 2i nodos de altura h – i

S = 2i (h-i)

∑ h i=0 S = h + 2 (h-1) + 4 (h-2) + 8 (h-3) + ………2h-1 (1)

S = h + 2 (h-1) + 4 (h-2) + 8 (h-3) + …………… + 2h-1 (1) (A)

2S = 2h + 4 (h-1) + 8 (h-2) + 16 (h-3) + ………. + 2h (1) (B)

2S = 2h + 4 (h-1) + 8 (h-2) + 16 (h-3) + ………. + 2 (1) (B)

Restando las dos igualdades (B) – (A)

S = -h + 2 (h-(h-1)) + 4((h-1)-(h-2)) + 8 ((h-2)-(h-3)) + …+ 2h-1 (2-1)+ 2h

S = -h + 2 + 4 + 8 + 16 + ………+ 2h-1 + 2h

S + 1 = -h + 1 + 2 + 4 + 8 + 16 + ………+ 2h-1 + 2h

S + 1 = -h + (2h+1 S + 1 = -h + (2 – 1) h+1 – 1)

S = (2h+1 – 1) – (h + 1).

* Un árbol binario completo no es un árbol binario lleno, pero el resultado obtenido es una cota superior de la suma de las alturas de los nodos en un árbol binario completo
* Un árbol binario completo tiene entre 2 h y 2 h+1 - 1 nodos, el teorema implica que esta suma es de O(n) donde n es el número de nodos.
* Este resultado muestra que la operación BuildHeap es lineal.

**Operaciones de vectores usando Heap**

Dado un conjunto de n elementos y se los quiere ordenar en forma creciente, existen dos alternativas:

a) Algoritmo que usa una heap y requiere una cantidad aproximada de (n log n) operaciones.

* Construir una MinHeap, realizar n DeleteMin operaciones e ir guardando los elementos extraídos en otro arreglo.
* Desventaja: **requiere el doble de espacio**

b) Algoritmo Heap0Sort que requiere una cantidad aproximada de (n log n) operaciones, pero menos espacio.

* Construir una **MaxHeap** con los elementos que se desean ordenar, intercambiar el último elemento con el primero, decrementar el tamaño de la heap y filtrar hacia abajo. Usa sólo el espacio de almacenamiento de la heap.

**Interfaces (clase 6.1)**

**¿Qué son?**

* Una interface java es una colección de definiciones de métodos sin implementación/cuerpo y de declaraciones de variables de clases constantes, agrupados bajo un nombre.
* Las interfaces proporcionan un mecanismo para que una clase defina comportamiento (métodos) de un tipo de datos diferentes al de sus superclases.
* Una interface establece que debe hacer la clase que la implementa, sin especificar el cómo.

**¿Cómo se define una interface?**

package nomPaquete;

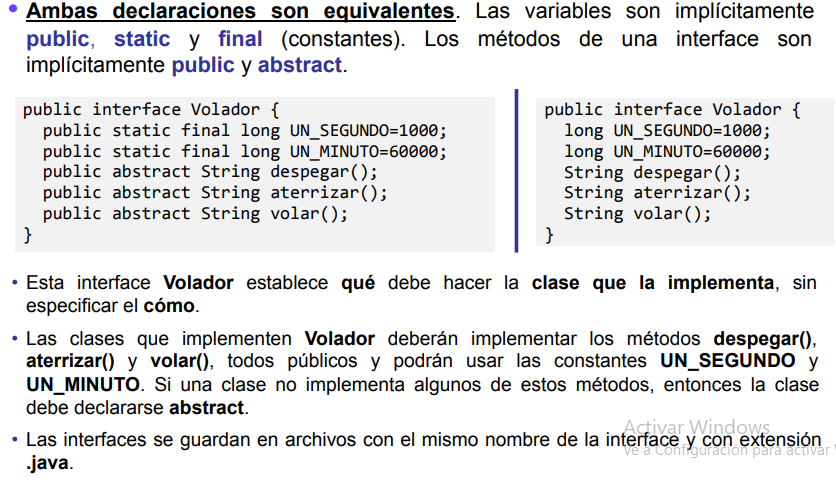
public interface UnaInter extends SuperInter1 , SuperInter2 , ... {

Declaración de métodos: implícitamente public y abstract

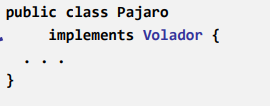
Declaración de constantes: implícitamente public, static y final

}

* El especificador de acceso public, establece que la interface puede ser usada por cualquier clase o interface de cualquier paquete. Si se omite el especificador de acceso, la interface solamente podría ser usada por las clases e interfaces contenidas en el mismo paquete que la interface declarada.
* Una interface puede extender múltiples interfaces. Hay herencia múltiple de interfaces.
* Una interface hereda todas las constantes y métodos de sus SuperInterfaces. Declaración de métodos: implícitamente public y abstract Declaración de constantes: implícitamente public, static y final



* **Para especificar que una clase implementa una interface se usa la palabra clave implements**



* Una clase que implementa una interface, hereda las constantes y debe implementar cada uno de los métodos declarados en la interface.
* Cuando una clase implementa una interface se establece como un contrato entre la interface y la clase que la implementa.
* El compilador hace cumplir este contrato asegurándose de que todos los métodos declarados en la interface se implementen en la clase.

**Interfaces y Herencia múltiple**

* El mecanismo que permite crear una clase derivada de varias clases bases, se llama herencia múltiple. Como cada clase tiene una implementación propia, la combinación puede genera ambigüedades.
* Java, NO soporta herencia múltiple, pero provee interfaces para lograr un comportamiento “similar”. Como las interfaces no tienen implementación, NO causa problemas combinarlas, entonces: **una clase puede heredar de una única clase e implementar tantas interfaces como quiera.**

**Interfaces y upcasting**

Tratar a una referencia de la clase derivada como una referencia de la clase base, se denomina upcasting.

El principal objetivo de las interfaces es permitir el “upcasting” a otros tipos, además del upcasting al tipo base.

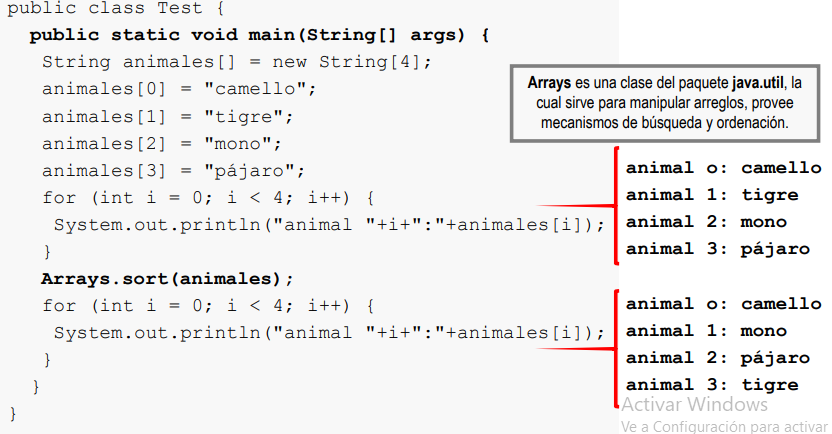
**Interfaces vs. Clases Abstractas**

* Las interfaces son completamente abstractas, no tienen ninguna implementación.
* Con interfaces no hay herencia de métodos, con clases abstractas sí.
* No es posible crear instancias de clases abstractas ni de interfaces.
* Una clase puede extender solo una clase abstracta, pero puede implementar múltiples interfaces.

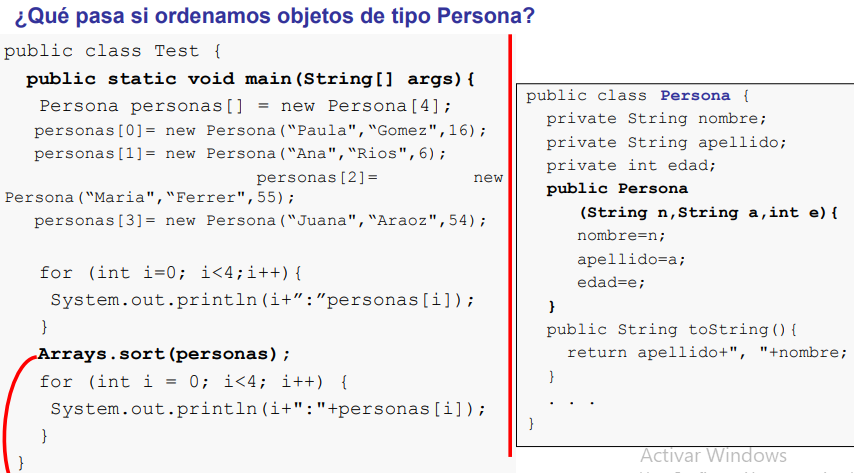
**¿Usar interfaces o clases abstractas?**

* Si es posible crear una clase base con métodos sin implementación y sin variables de instancia, es preferible usar interfaces.
* Si estamos forzados a tener implementación o definir atributos, entonces usamos clases abstractas.
* Java no soporta herencia múltiple de clases, por lo tanto, si se quiere que una clase sea además del tipo de su superclase de otro tipo diferente, entonces es necesario usar interfaces.

**Ordenando objetos**



Después de invocar al método sort(), el arreglo quedo ordenado alfabéticamente. Esto es porque los objetos de tipo **String** son **comparables.**

****

Al invocar al metodo **sort()** y pasar el arreglo personas, da un error porque los objetos PERSONAS **no son comparables.**

**La interface Comparable<T>**

Si una clase implementa la interface java.lang.Comparable, hace sus instancias comparables. Esta interface tiene solo un método, compareTo(), el cual determina como comparar dos instancias de una misma clase. El método es el siguiente:

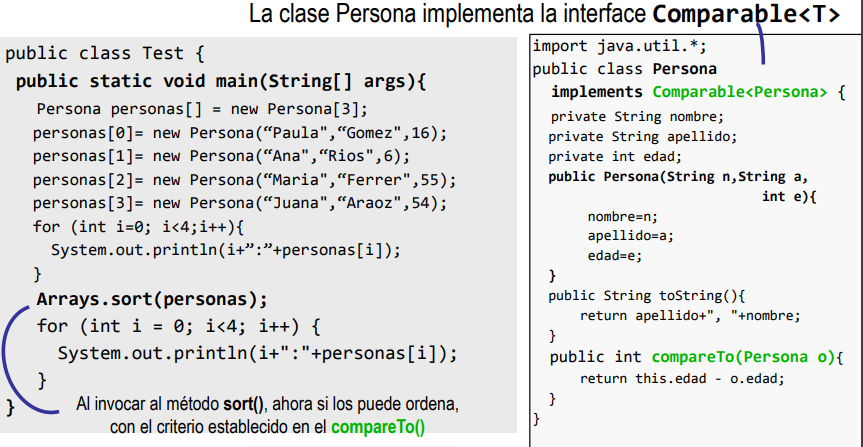
public interface Comparable<T>{

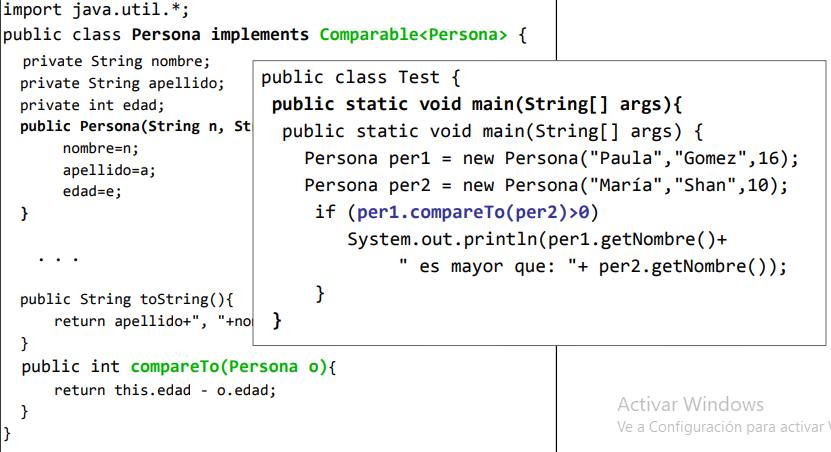
Public int compareTo(T o)

}

**Este método retorna:**

* **=0 :** si el objeto receptor es igual al pasado en el argumento.
* **>0 :** si el objeto receptor es mayor que el pasado como parámetro.
* **<0 :** si el objeto receptor es menor que el pasado como parámetro.

****



**Análisis de Algoritmo – Clase 12**

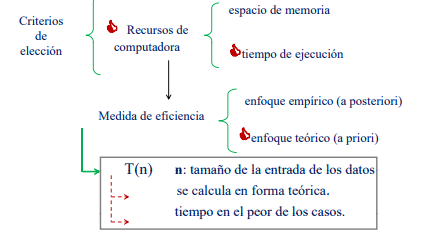
Nos permite comparar algoritmos en forma independiente de una plataforma en particular.

Mide la eficiencia de un algoritmo, dependiendo del tamaño de la entrada.

Pasos a seguir:

* Caracterizar los datos de entrada del algoritmo.
* Identificar las operaciones abstractas, sobre las que se basa el algoritmo.
* Realizar un análisis matemático, para encontrar los valores de las cantidades del punto anterior.

**Introducción al concepto T(n)**



Se toman en cuenta los recursos de la computadora(como medida de eficiencia),el tiempo de ejecución y un enfoque teorico.

T(n) siendo n el tamaño de la entrada de los datos. Se calcula teóricamente y el peor de los casos

Debemos enfocarnos en cuán rápido crece una función T(n) respecto al tamaño de la entrada. A esto lo llamamos la tasa o velocidad de crecimiento del tiempo de ejecución.

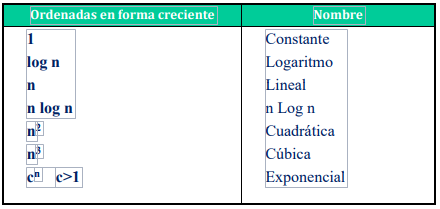
Por ejemplo, supongamos que un algoritmo, que corre con una entrada de tamaño n, tarda 6n2+100n+300 instrucciones de máquina. El término 6n2 se vuelve más grande que el resto de los términos, 100n+300 una vez que n se hace suficientemente grande, 20 en este caso.

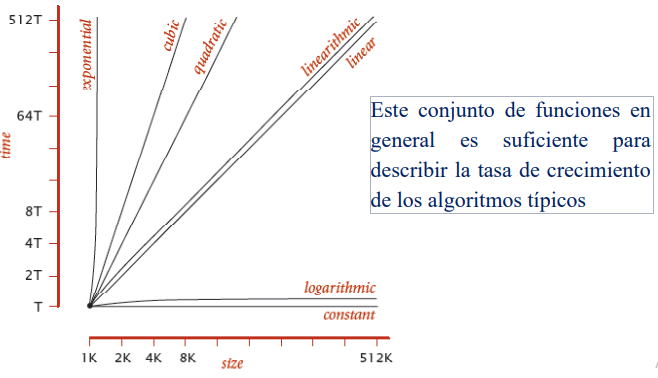
Al descartar los términos menos significativos y los coeficientes constantes, podemos enfocarnos en la parte importante del tiempo de ejecución de un algoritmo, si taza o velocidad de crecimiento, sin involucrarnos en detalles que complican nuestro entendimiento.

Cuando descartamos los coeficientes constantes y los términos menos significativo, usamos notación asintótica.

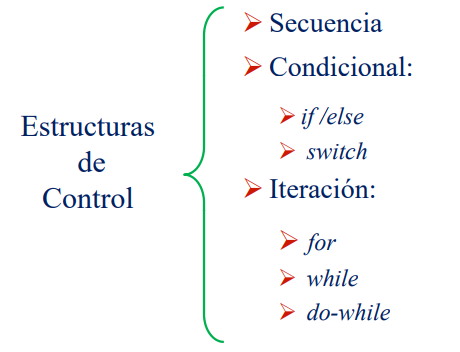
El T(n) siempre evalúa el peor de los casos.

**Algunas funciones**





**Calculo del Tiempo de Ejecución**



Los cálculos de cada uno los hice en mi cuaderno

Condicional

If/Else

Siempre se evalua la condición analizada en el if como 1 unidad, mas la cantidad de condiciones dentro (en el caso de if else, se evaluara el peor caso)

Switch

Se evalua la condición y el peor caso de todas las opciones disponibles

Iteracion

For

Evalua desde A a B la cantidad de condiciones, se utiliza sumatoria para definir la cantidad de tiempo-repeticiones del for

*int sum = 0;//tiempo const*

*int [] a = new int [n];//tiempo const*

*for (int i =1; i<= n ; i++ )*

*sum += a[i];*

el T(n)=c0(tiempo constante al principio)+ la sumatoria desde 1 hasta n de:

( la condición dentro del for)

Como no es conocido el tiempo, podemos afirmar que seria = T(n): C0+n\*C1=O(n)

While

A priori no se sabe el tiempo de el mismo while, entonces se va a ir analizando el desarrollo del mismo

*int x= 0;*

*int i = 1;*

*while ( i <= n)* {

*x = x + 1;*

*i = i + 2;*

Podemos decir, que i tiene un desarrollo dado cualquier K)i=1+2(k-1)

el T(n)=C0+ (la sumatoria desde i=1 hasta (n+1)/2 de) C1, por lo tanto quedaría

c0+c1/2\*(n+1) que seria igual a O(n) ya que no se conoce la cantidad de cada valor.

Otro ejemplo:

*int x= 1;*

*while ( x < n)*

*x = 2 \*x;*

Su T(n)=c0+c2\*log(n)🡺(ya que si n es potencia de 2 realiza log(n) iteraciones, y si no lo es , realiza log(n)+1 iteraciones)

Que esto da como resultado O(log(n))

**Calculo de tiempo de ejecución en algoritmos recursivos**

Cuando el algoritomo es recursivo, se pueden identificar dos casos:

El caso base y el caso recursivo

Dado

/\*\*

Calcula el Factorial.

\*/

public static int factorial( int n ) {

if (n == 1)

return 1;

else return n \* factorial( n -1 );

}

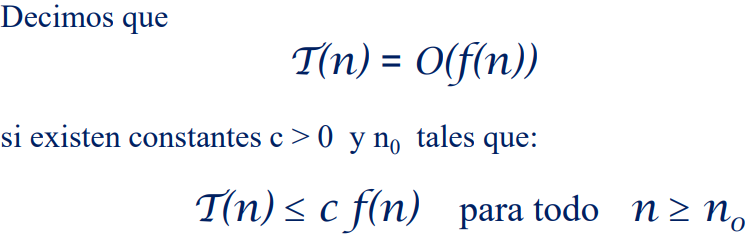
Funcion de recurrencia

C1 *n = 1*

*T(n) =*

C2 *+ T(n -1) n >1*

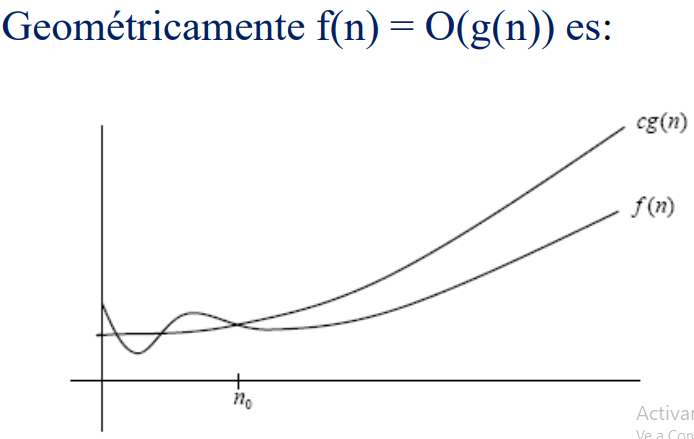
**Análisis de algoritmo –Definición de Big-Oh – Clase 13**



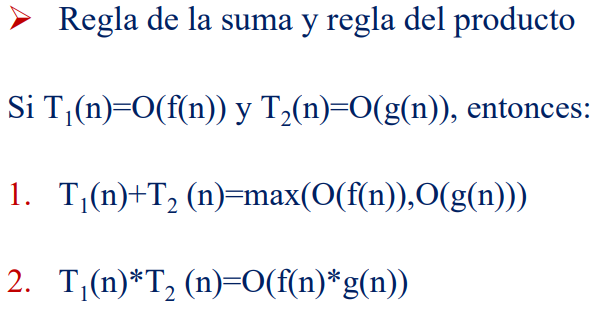
Se lee: T(n) es de orden de f(n).

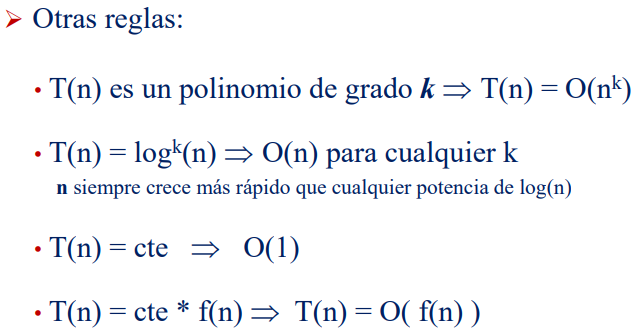
f(n)representa una cota superior de T(n).

La tasa de crecimiento de T(n) es menor o igual a la de f(n).



Como la notación O grande solamente da una cota asintótica superior, y no da una cota asintóticamente ajustada, podemos hacer declaraciones que en primera instancia parecen incorrectas, pero que son técnicamente correctas.





**Tiempo de ejecución de los recorridos de Arboles Binarios y Generales**

**Árbol binario**

Preorden • Se procesa primero la raíz y luego sus hijos, izquierdo y derecho.

Considerando un árbol binario lleno y altura h

c n = 1

T(n) =

c + 2 T( (n – 1) / 2 ) n >1

T(n) es O(n)

Por niveles • Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

T(n)=cte1 + la sumatoria desde i=1 hasta n de(cte2)

=cte1+n\*cte2

T(n) es O(n)

Inorden • Se procesa el hijo izquierdo, luego la raíz y último el hijo derecho Postorden • Se procesan primero los hijos, izquierdo y derecho, y luego la raíz

¿?????? No mandaron de este. Revisar clase

**Árbol general**

Preorden • Se procesa primero la raíz y luego los hijos

Considerando un árbol lleno de grado k y altura h

c n = 1

T(n) =

c + k T( (n – 1) / k ) n >1

T(n) es O(n)

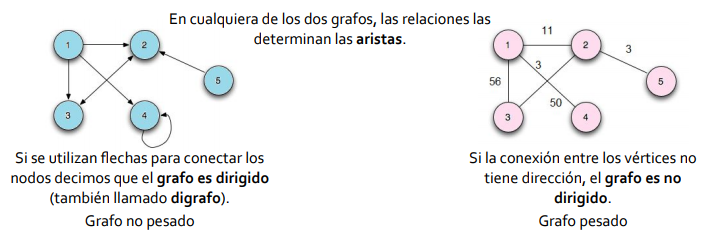
Por niveles • Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

T(n)=cte1+la sumatoria desde i=1 hasta n()

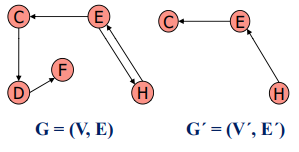
**Grafos - Clase 14**

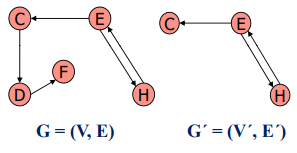
**Terminologia**

* **Grafo:** modelo para representar relaciones entre elementos de un conjunto.
* **Grafo: (V, E),** V es un conjunto de vértices o nodos, con una relación entre ellos; E es un conjunto d pares (**u, v**), donde u y v pertenecen a V, llamados aristas o arcos. Por lo general, los vértices son nodos de procesamiento o estructuras que contienen algún tipo de información mientras que las aristas determinan las relaciones entre nodos. Las aristas también pueden tener algún tipo de información asociada (distancia, peso, etc.) en cuyo caso es un grafo pesado.



* **Grafo dirigido:** la relación sobre V no es simétrica. Arista=par ordenado (u, v).
* **Grafo no dirigido:** la relación sobre V es simétrica. Arista=par no ordenado {u, v}, donde u y v pertenecen a V, y **u** es distinto de **v**.
* **v** es **adyacente** a **u** si existe una arista (u, v) que pertenecen a E.
* es un grafo no dirigido, (u, v) pertenece a E índice en los nodos u, v.
* es un grafo dirigido, (u, v) pertenece a E índice en v y parte de u.
* **En grafos no dirigidos:** el grado de un nodo es el número de arcos que inciden en él.
* **En grafos dirigidos:** existen el grado de salida (**grado\_out**, es el número de arcos que parten de él) y el grado de entrada (**grado\_in**, es el número de arcos que inciden de él). El grado de vértice será la suma de los grados de entrada y de salida.
* **Grado de un grafo**: máximo grado de sus vértices Camino desde u Є V a v Є V : secuencia v1, v2, ... , vk tal que u=v1, v=v k, y (vi−1,vi) Є E, para i = 2, ... ,k.
* **Camino** desde **u** perteneciente a V a **v** perteneciente a V: es la secuencia de v1, v2, …, vk tal que **u=v1**, **v=vk**, y (vi-1, vi) perteneciente a E, para i= 2, …, k.
* **Longitud de un camino:** número de arcos del camino.
* **Camino simple:** camino en el que todos sus vértices, excepto, tal vez, el primero y el ultimo, son distintos.
* Ciclo: camino desde v1, v2, ... , vk tal que v1=vk. El ciclo es simple si el camino es simple.
* **Ciclo:** camino desde v1, v2, ... , vk tal que v1=vk
* **Bucle:** ciclo de longitud 1.
* **Grafo aciclico:** grafo sin ciclos.
* Dado un grafo G=(V, E), se dice que G´=(V´ , E´) es un **subgrafo** de G, si V’ V y E’  E.



* Un **subgrafo inducido** por V’  V : G’ = (V’,E’) tal que E’ = {(u,v)  E | u,v  V’}.
* Un **grafo ponderado**, pesado o con costos: cada arco o arista tiene asociado asociado un valor o etiqueta etiqueta.
* Un **grafo no dirigido es conexo** si hay un camino entre cada par de vértices.
* Un **bosque** es un grafo sin ciclos.
* Un **árbol libre** es un bosque conexo.
* Un árbol es un **árbol libre** en el que un nodo se ha designado como raíz.

**Propiedades**

Sea G un grafo no dirigido con n vértices y m arcos, entonces

Σ deg(v) = 2\*m v Є G

Siempre: m≤ (n\*(n-1))/2

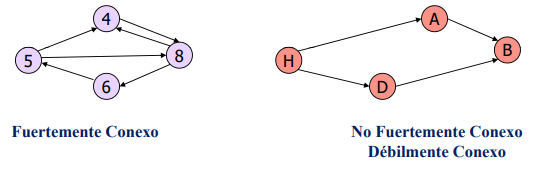
Si G conexo: m≥n-1

Si G árbol: m=n-1

Si G bosque: m≤n-1

**Conectividad en Grafos dirigidos**

* v es alcanzable desde u, si existe un camino de u a v.
* Un grafo dirigido se denomina fuertemente conexo si existe un camino desde cualquier vértice a cualquier otro vértice

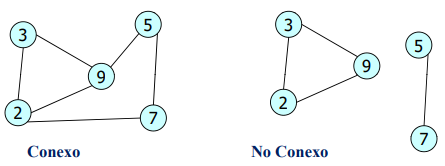


* Si un grafo dirigido no es fuertemente conexo, pero el grafo subyacente (sin sentido en los arcos) es conexo, el grafo es débilmente conexo.

**Componentes conexas**

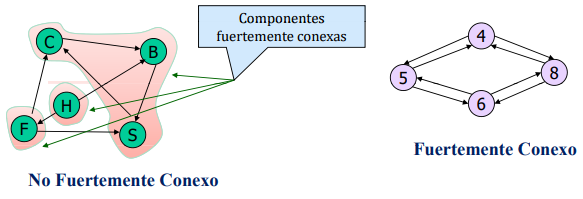
En un grafo no dirigido, una componente conexa es un subgrafo conexo tal que no existe otra componente componente conexa que lo contenga contenga.

Es un subgrafo conexo maximal.

Un grafo no dirigido es no conexo si está formado por varias componentes conexas.

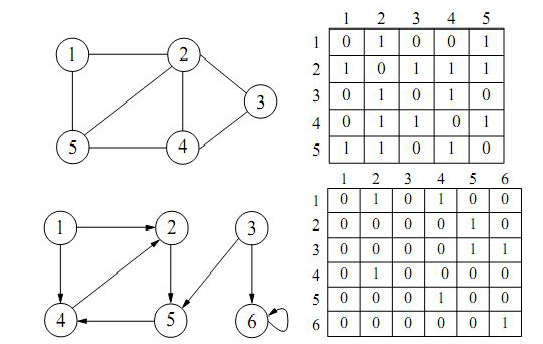
**Componentes fuertemente conexas**

En un grafo dirigido, una componente fuertemente conexa, es el máximo subgrafo fuertemente conexo.

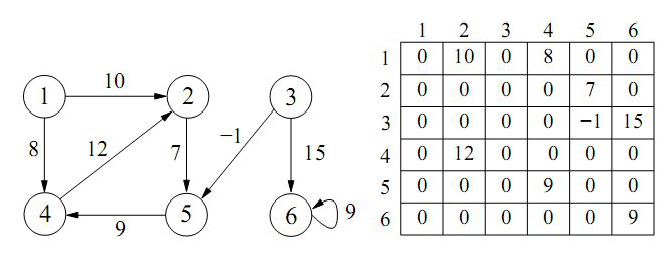
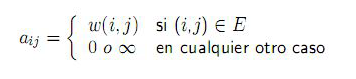
Un grafo dirigido es no fuertemente conexo si está formado por varias componentes fuertemente conexas.

**Representaciones: Matriz de Adyacencias**

* G = (V, E): matriz A de dimensión |V |×|V |.
* Valor aij de la matriz:
* Costo espacial: O (|V|2)
* Representación es útil para grafos con número de vértices pequeño, o grafos densos (|E|≈|V|×|V|)
* Comprobar si una arista (u,v) pertenece a E → consultar posición A(u,v)
* Costo de tiempo T(|V|,|E|) = O(1)

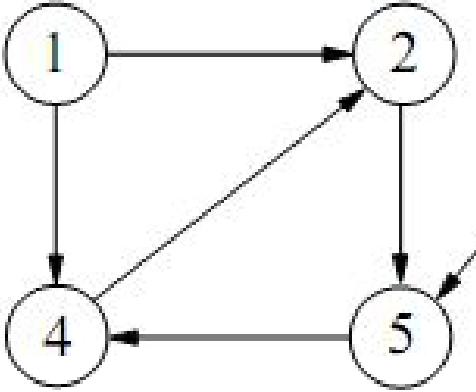
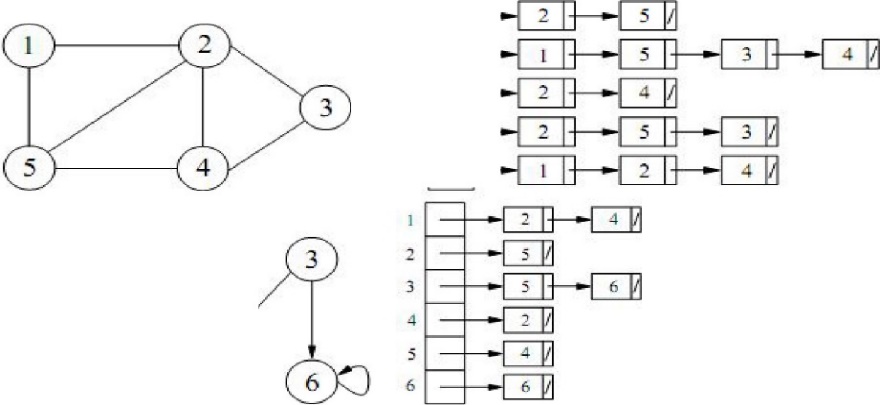


* Representación aplicada a Grafos pesados
* El peso de (i,j) se almacena en A (i, j)



**Representaciones: Lista de Adyacencias**

* G = (V,E): vector de tamaño |V|.
* Posicion i : puntero a una lista enlazada de elementos.
* Los elementos de la lista son los vértices adyacentes a i



*2*

3

4

5

* Si G es dirigido, la suma de las longitudes de las listas deadyacencia será |E|.
* Si G es no dirigido, la suma de las longitudes de las listas deadyacencia será 2|E|.
* Costo espacial, sea dirigido o no: O (|V|+|E|) .
* Representación apropiada para grafos con |E| menor que |V|2.
* Desventaja: si se quiere comprobar si una arista (u,v)pertenece a E🡺 buscar v en la lista de adyacencia de u.
* Costo temporal T (|V|,|E|) será O (Grado G) perteneciente a O (|V|).
* Representación aplicada a Grafos pesados
* El peso de (u,v) se almacena en el nodo de v de la lista de adyacencia de u.

**Recorridos**

**En profundidad: DFS (Depth First Seach)**

Generalización del recorrido **preorden** de un árbol.

**Estrategia:**

. Partir de un vértice determinado v.

. Cuando se visita un nuevo vértice, explorar cada camino que salga de él.

. Hasta que no se haya finalizado de explorar uno de los caminos no se comienza con el siguiente.

. Un camino deja de explorarse cuando se llega a un vértice ya visitado.

. Si existían vértices no alcanzables desde v el recorrido queda incompleto entonces, se debe seleccionar algún vértice como nuevo vértice de partida, y repetir el proceso.

Esquema recursivo: dado G = (V , E)

1. Marcar todos los vértices como no visitados.

2. Elegir vértice u como punto de partida.

3. Marcar u como visitado.

4. Para todo v adyacente a u,(u,v) Є E, si v no ha sido visitado, repetir recursivamente (3) y (4) para v.

• Finalizar cuando se hayan visitado todos los nodos alcanzables desde u.

• Si desde u no fueran alcanzables todos los nodos del grafo: volver a (2), elegir un nuevo vértice de partida v no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

**Tiempo de ejecución**

.G (V , E) se representa mediante listas de adyacencia.

. El método dfs(v) se aplica únicamente sobre vértices no visitados → sólo una vez sobre cada vértice.

. Dfs(v) depende del número de vértices adyacentes que tenga (longitud de la lista de adyacencia). → el tiempo de todas las llamadas a dfs(v): O(|E|)

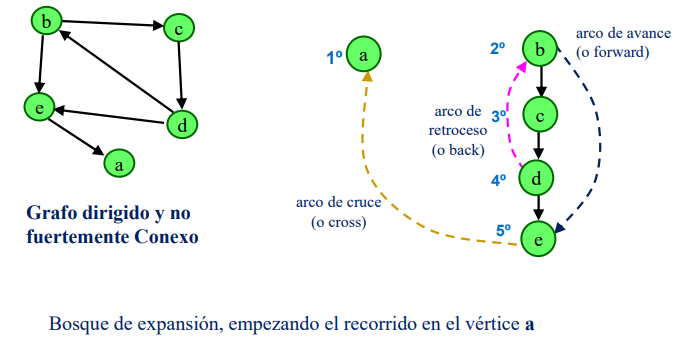
. Añadir el tiempo asociado al bucle de main\_dfs(grafo): O (|V|)). 🡺 Tiempo del recorrido en profundidad es O(|V|+|E|) .

**Bosque de expansión del DFS**

El recorrido no es único: depende del nodo inicial y del orden de visita de los adyacentes.

•El orden de visita de unos nodos a partir de otros puede ser visto como un árbol: árbol de expansión (o abarcador) en profundidad asociado al grafo.

•Si aparecen varios árboles: bosque de expansión (o abarcador) en profundidad.

**

**Clasificación de los arcos de un grafo dirigido en el bosque de expansión de un DFS.**

**•Arcos tree (del árbol):** son los arcos en el bosque depth-first-search, arcos que conducen a vértices no visitados durante la búsqueda.

**• Arcos forward**: son los arcos u🡺v que no están en el bosque, donde v es descendiente, pero no es hijo en el árbol.

•**Arcos backward:** son los arcos u🡺v, donde v es antecesor en el árbol. Un arco de un vértice a si mismo es considerado un arco back.

•**Arcos cross:** son todos los otros arcos u🡺v, donde v no es ni antecesor ni descendiente de u. Son arcos que pueden ir entre vértices del mismo árbol o entre vértices de diferentes árboles en el bosque depth-first-search

**Recorrido en amplitud: BFS**

Generalización del recorrido **por niveles** de un árbol.

**Estrategia:**

. Partir de algún vértice u, visitar u y, después, visitar cada uno de los vértices adyacentes a u.

. Repetir el proceso para cada nodo adyacente a u, siguiendo el orden en que fueron visitados.

Esquema iterativo: dado G = (V , E)

1. Encolar el vértice origen u.

2. Marcar el vértice u como visitado.

3. Procesar la cola.

4. Desencolar u de la cola

5. para todo adyacente a u,(u,v) Є E,

6. si v no ha sido visitado

7. encolar y visitar v

• Si desde u no fueran alcanzables todos los nodos del grafo: volver a (1), elegir un nuevo vértice de partida no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices

Costo T(|V|,|E|) es de O(|V|+|E|)

**Agenda - Ordenación topológica** (clase 16)

**Definición**

La ordenación topológica es una permutación:

v1, v2, v3, ..., v|V| de los vértices, tal que si (vi,vj) Є E,

vi ≠ vj, entonces vi precede a vj en la permutación.

* La ordenación no es posible si G es cíclico.
* La ordenación topológica no es única.
* Una ordenación topológica es como una ordenación de los vértices a lo largo de una línea horizontal, con los arcos de izquierda a derecha.

Grafos dirigidos Acíclicos (DAG)

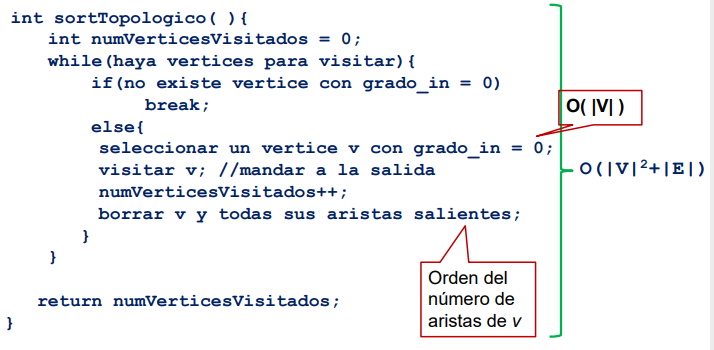
**Algoritmos**

* Con complejidad O( |V|2): Implementación con Arreglo (versión 1)

En esta versión el algoritmo utiliza un arreglo Grado\_in en el que se almacenan los grados de entradas de los vértices y en cada paso se toma de allí un vértice con grado\_in = 0.

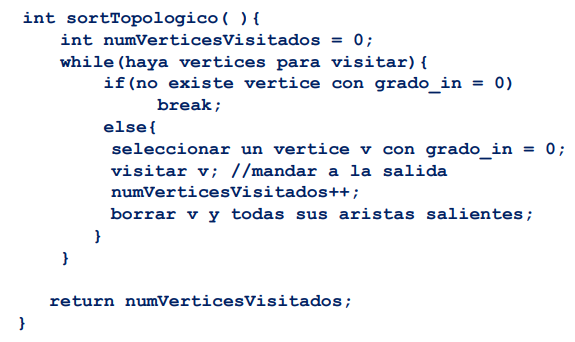
**Pasos generales:**

1. Seleccionar un vértice v con grado de entrada cero
2. Visitar v
3. “Eliminar” v, junto con sus aristas salientes
4. Repetir el paso 1 hasta seleccionar todos los vértices



* Con complejidad O( |V| + |A| )
* Implementación con Pila o Cola (versión 2)

En esta versión el algoritmo utiliza un arreglo Grado\_in en el que se almacenan los grados de entradas de los vértices y una pila P (o una cola Q) en donde se almacenan los vértices con grados de entrada igual a cero.



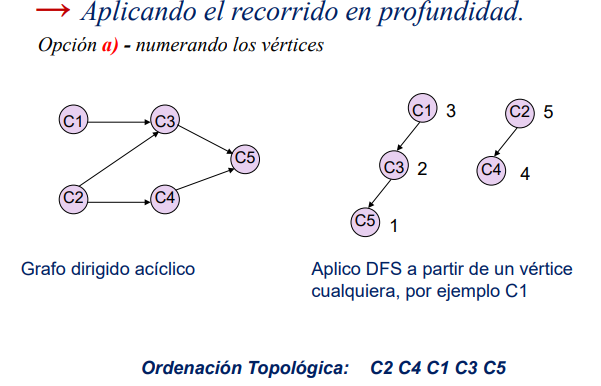
* DFS (versión 3)

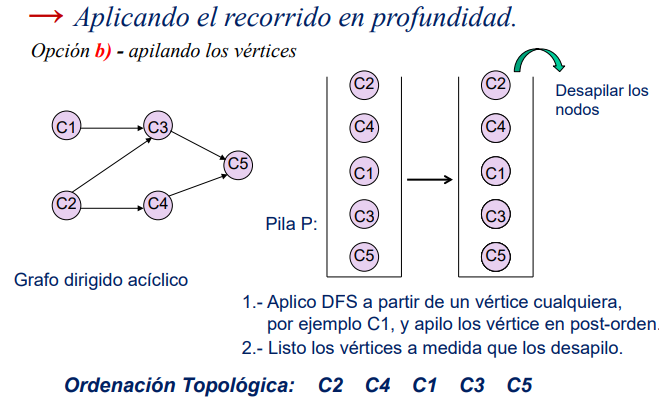
En esta versión se aplica el recorrido en profundidad.

 Se realiza un recorrido DFS, marcando cada vértice en post-orden, es decir, una vez visitados todos los vértices a partir de uno dado, el marcado de los vértices en post- orden puede implementarse según una de las sig. opciones:

a) numerándolos antes de retroceder en el recorrido; luego se listan los vértices según sus números de post- orden de mayor a menor.

b) colocándolos en una pila P, luego se listan empezando por el tope.



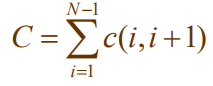


**Grafos – Caminos de Costos Mínimos**

**Definición**

Sea G=(V,A) un grafo dirigido y pesado, el costo c(i,j) está asociado a la arista v(i,j).

Dado un camino: v1, v2, v3,…….vN. El costo del camino es:



Este valor también se llama longitud del camino pesado. La longitud del camino no pesado es la cantidad de aristas.

El camino de costo mínimo desde un vértice vi a otro vértice v es aquel en que la suma de los costos de las aristas es mínima. Esto significa que la sumatoria es mínima.

***Algoritmos para el cálculo del camino mínimo desde el origen hacia todos sus vértices:***

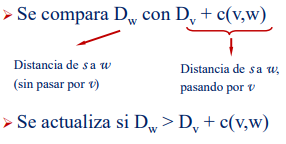
* **Grafos sin peso**

Para cada vértice v se mantiene la siguiente información:

* Dv: distancia mínima desde el origen s (incialmente ∞ para todos los vértices excepto el origen con valor 0)
* Pv: vértice por donde paso para llegar.
* Conocido: dato booleano que me indica si está procesando (inicialmente todos en 0).
* Usa la estrategia de recorrido en amplitud BFS. Los pasos son: Avanzar por niveles a partir del origen, asignando distancias según se avanza (se utiliza una cola). Inicialmente, es Dw=∞. Al inspeccionar w se reduce al valor correcto Dw = Dv + 1- Desde cada vi visitamos a todos los nodos adyacentes a v
* **Grafos con pesos positivos - Algoritmo de Dijkstra (dos implementaciones)**
* Pasos: Dado un vértice origen s, elegir el vértice v que esté a la menor distancia de s, dentro de los vértices no procesados. Marcar v como procesado. Actualizar la distancia de w adyacente a v.

Para cada vértice v mantiene la siguiente información:

* Dv: distancia mínima desde el origen s (incialmente ∞ para todos los vértices excepto el origen con valor 0)
* Pv: vértice por donde aso para llegar.
* Conocido: dato booleano que me indica si esta procesado (inicialmente todos en 0)
* La actualización de la distancia de los adyacentes w se realiza con el siguiente criterio:



**Tiempo de ejecución**:

* Si almacenamos las distancias en un vector, tendremos que:

 El bucle para de la línea (4) se ejecuta para todos los vértices ► |V| iteraciones

 La operación vérticeDesconocidoMenorDist -línea (5)- es O(|V|) y dado que se realiza |V| veces ► el costo total de vérticeDesconocidoMenorDist es O(|V|2)

 El bucle para de la línea (7) se ejecuta para los vértices adyacentes de cada vértice. El número total de iteraciones será la cantidad de aristas del grafo. ► |E| iteraciones

 El costo total del algoritmo es (|V|2 + |E| ) es O(|V|2)

**Optimización:** la operación vérticeDesconocidoMenorDist es más eficiente si almacenamos las distancias en una heap.

 La operación vérticeDesconocidoMenorDist -línea (5)- es O(log|V|) y dado que se realiza |V| veces ► el costo total de vérticeDesconocidoMenorDist es O(|V| log |V|)

 El bucle para de la línea (7) que se ejecuta para los vértices adyacentes de cada vértice, también supone modificar la prioridad (distancia) y reorganizar la heap luego de la línea (10). Cada iteración es O(log|V|) ► realiza |E| iteraciones, O(|E| log|V|)

 El costo total del algoritmo es (|V| log|V|+ |E| log|V|) es O(|E| log|V|)

**Variante para evitar modificar y reorganizar la heap:**

la actualización de la heap luego de la línea (10) se puede resolver insertando el vértice w y su nuevo valor Dw cada vez que éste se modifica.

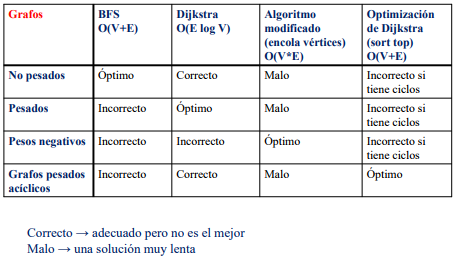
 El tamaño de la heap puede crecer hasta |E|. Dado que |E| ≤ |V|2, log |E| ≤ 2 log |V|, el costo total del algoritmo no varía

 El costo total del algoritmo es O(|E| log|V|)

* **Grafos con pesos positivos y negativos**
* Estrategia: Encolar los vértices. Si el grafico tiene aristas negativas, el algoritmo de Dijkstra puede dar un resultado erróneo.
* Pasos: Encolar el vértice de origen s. Procesar la cola: Desencolar un vértice. Actualizar la distancia de los adyacentes Dw siguiendo el mismo criterio de Dijkstra. Si w no esta en la cola, encorarlo.
* El costo toal del algoritmo es O(|V| |E|)
* **Grafos dirigidos aciclicos**

Estrategia: Orden Topológico.

* Optimización del algoritmo de Dijkstra
* La selección de cada vértice se realiza siguiendo el orden topológico
* Esta estrategia funciona correctamente, dado que al seleccionar un vértice v, no se va a encontrar una distancia dv menor, porque ya se procesaron todos los caminos que llegan a él
* El costo total del algoritmo es O(|V| + |E|)



***Algoritmos para el cálculo del camino mínimo entre todos los pares de vertice:***

