

# Programación III

# Índice

1 Principios básicos .....	4
1.1 Clases .....	4
1.1.1 Casteo .....	4
1.1.2 Constructores .....	4
1.1.3 Static .....	4
1.2 Herencia y Abstracción .....	5
2 Tipos de datos abstractos (TDA) .....	5
2.1 Listas .....	5
2.1.1 Iterador .....	5
2.2 Pila .....	6
2.3 Cola .....	6
3 Arboles .....	6
3.1 Arboles binarios .....	6
3.1.1 Propiedades: .....	6
3.1.2 Recorridos .....	6
3.2 Arboles de expresión .....	6
3.2.1 Recorridos .....	7
3.3 Arboles generales .....	7
3.3.1 Recorridos .....	8
4 Cola de prioridad (HEAP) .....	8
4.1 Implementaciones .....	8
4.2 Heap .....	8
4.2.1 Propiedad estructural .....	8
4.2.2 Propiedad de orden .....	8
4.2.3 Restauracion .....	9
5 Tiempo de ejecución .....	9
5.1 Algoritmo constante .....	9
5.2 Algoritmos iterativos .....	9
5.2.1 For .....	9
5.2.2 While .....	9
5.3 Algoritmos recursivos .....	10
5.4 Big Oh .....	10
6 Grafos .....	10
6.1 Terminologia .....	10
6.2 Representacion .....	12
6.2.1 Matriz de adyacencias .....	12
6.2.2 Lista de adyacencias .....	12
6.3 Recorridos .....	12
6.3.1 DFS (Depth First Search) .....	12
6.3.2 BFS (Breath First Search) .....	13
6.3.3 Bosque de expansion DFS .....	13
6.3.4 Aplicaciones del DFS .....	14
6.3.5 Algoritmo de Kosajaru .....	14
6.4 Caminos minimos .....	14
6.4.1 Grafos sin peso .....	14
6.4.2 Grafos con pesos positivos (Dijkstra) .....	15

6.4.3 Grafos con pesos positivos y negativos .....	15
6.4.4 Grafos aciclicos .....	15
6.4.5 Algoritmo de Floyd .....	15
6.4.6 Resumen <b>Importante</b> .....	15
6.5 Ordenacion topologica .....	16
6.5.1 Version 1 .....	16
6.5.2 Version 2 .....	16
6.5.3 Version 3 .....	17

# 1 Principios básicos

## 1.1 Clases

Una clase java es un bloque de código que modela a un objeto, definiendo sus atributos y comportamientos a través de variables y métodos de instancia. Estas variables pueden ser:

- Primitivas: tipo de dato simple, inicializado en cero, nulo o false.
- Wrapper: tipo de dato objeto, inicializado en nulo e immutable.

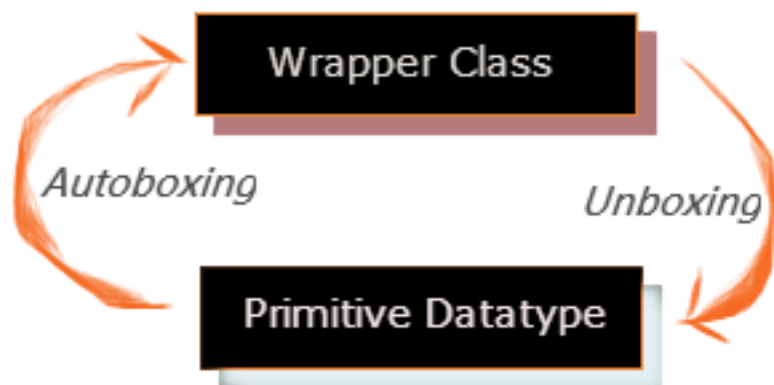
**Importante:** Las variables locales a un modulo, deben inicializarse antes de su uso.

**Importante:** los tipos de datos wrapper 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.

### 1.1.1 Casteo

Tipos de datos genéricos: permiten generalizar clases y/o métodos para cualquier tipo de dato wrapper. sirven también para detectar errores de tipos de datos en ejecución.

Autoboxing: cambiar una variable de primitivo a wrapper. Unboxing: cambiar una variable de wrapper a primitivo.



### 1.1.2 Constructores

Un constructor, es un bloque de código dentro de una clase, que se encarga de declarar el estado inicial por defecto para un objeto. La sobrecarga de métodos y constructores permite al objeto tener distintos métodos y constructores definidos, con el mismo nombre pero distintos parámetros.

**Importante:** los parámetros en java son siempre por valor.

### 1.1.3 Static

La palabra clave static declara atributos (variables) y métodos asociados con la clase en lugar de asociarlos a cada una de las instancias de la clase. Las variables de clase son compartidas por todas las instancias de la clase. En el caso de los métodos de clase se utilizan cuando se necesita algún comportamiento que no depende de una instancia particular. En ambos casos se debe anteponer la palabra clave static al tipo de dato de la variable o de retorno del método.

## 1.2 Herencia y Abstracción

El objetivo de definir una clase abstracta es lograr una interface de comportamiento común para los objetos de las subclases (de la clase abstracta). No se pueden crear instancias a partir de una clase abstracta. Se espera que una clase abstracta sea extendida por clases que implementen todos sus métodos abstractos. Un método abstracto no tiene cuerpo! Se debe anteponer la palabra clave `abstract` al tipo de datos de retorno del método.

Características:

- Si dentro de una clase se declara un método abstracto, la clase debe declararse abstracta.
- Las clases abstractas pueden tener métodos concretos y métodos abstractos.
- Las clases concretas no pueden tener métodos abstractos, solo métodos concretos.



## 2 Tipos de datos abstractos (TDA)

### 2.1 Listas

Una lista es una secuencia lineal de elementos que pueden manipularse libremente, respetando las operaciones dadas por el lenguaje. En java se pueden usar dos implementaciones provistas por la api:

- `ArrayList`: declara a la lista como un arreglo de  $n$  posiciones
- `LinkedList`: declara a la lista como un puntero a un nodo que tiene la dirección del siguiente.

Entre otras diferencias, las mas importantes se resumen en:

ArrayList	LinkedList
Los elementos se almacenan en un arreglo dinámico. 	Los elementos se almacenan en una lista doblemente enlazada. 
Permite el acceso aleatorio ya que los arreglos se basan en índices. Eso significa que acceder a cualquier elemento siempre lleva un tiempo constante $O(1)$ .	Eso significa que acceder a cualquier elemento siempre lleva un tiempo lineal $O(n)$ .
Verificar si existe un elemento específico en la lista dada se ejecuta en tiempo lineal $O(n)$ .	Verificar si existe un elemento específico en la lista dada se ejecuta en tiempo lineal $O(n)$ .
Agregar/Borrar elementos en/de un índice específico, en el peor de los casos, es de $O(n)$ .	Agregar/Borrar elementos en/de un lugar específico, en el peor de los casos, es de $O(n)$ . Suele ser más rápida porque nunca se necesita cambiar el tamaño de la estructura.
Esta clase es más útil cuando la aplicación requiere acceso a datos y su tamaño no varía demasiado.	Esta clase es más útil cuando se conoce que la aplicación requiere manipulación de datos (muchas inserciones y borrados).

Se ve que, al usar `LinkedList`, el tiempo de acceso es  $O(1)$  para operaciones de agregar, eliminar, mientras que al usar `ArrayList`, la búsqueda es de  $O(1)$ . Además, `LinkedList` consume algo más de memoria por tener almacenado la dirección del nodo siguiente.

#### 2.1.1 Iterador

Es un patrón de diseño de comportamiento que permite el recorrido secuencial por una estructura de datos sin exponer sus detalles internos.

```

List<Integer> lista = new ArrayList<Integer>();
Iterator<Integer> it = lista.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}

```

## 2.2 Pila

Una pila es una secuencia lineal de elementos actualizada en un extremo llamado tope de pila, usando la politica LIFO (last in - first out).

## 2.3 Cola

Una cola es una secuencia lineal de elementos actualizada en sus extremos, siguiendo una politica FIFO (first in - first out).

# 3 Arboles

## 3.1 Arboles binarios

Un arbol binario es una coleccion de nodos, que pueden tener a lo sumo dos hijos. En caso de no tener hijos, el nodo se denomina hoja.

### 3.1.1 Propiedades:

- **Profundidad:** es la longitud del camino desde la raiz hasta un nodo. Por ejemplo, la raiz tiene profundidad cero.
- **Altura:** es la longitud del camino mas largo desde un nodo hasta una hoja.
- **Grado:** es la cantidad de hijos maxima que tiene el arbol, o un nodo. Para arbol binario, el maximo de grado es 2.
- **Arbol lleno:** todos los nodos son de grado 2 y la cantidad de nodos es

$$2^{h+1} - 1$$

- **Arbol completo:** si es lleno de altura h-1, la cantidad de nodos esta entre

$$[2^h; 2^{h+1} - 1]$$

### 3.1.2 Recorridos

- Preorder: se procesa primero la raiz y despues el hijo izquierdo y derecho.
- In order: se procesa primero el hijo izquierdo, luego la raiz y luego hijo derecho.
- Post order: se procesa primer el hijo izquierdo, luego el hijo derecho y luego la raiz.
- Por niveles: se encola la raiz y sus hijos, y se va desencolando y agregando. (buena practica poner un null cada vez que termino de procesar un nivel).

## 3.2 Arboles de expresi3n

Los 3rboles de expresi3n representan el c3digo en una estructura de datos en forma de 3rbol donde cada nodo es una expresi3n

### 3.2.1 Recorridos

- Postfija: Este proceso es posible ya que la expresión posfija está organizada en una forma en la que los operandos aparecen antes de los operadores. Esto nos permite construir el árbol de expresión utilizando una pila, donde se apilan operandos hasta que se encuentre un nodo operador que tome los dos últimos nodos de la pila como hijos.

#### Estrategia I:

**convertir(expr\_posfija)**

crear una Pila vacía

**mientras** (existe un carácter) **hacer**

tomo un carácter de la expresión

**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 con ese operador

        desapilo y lo agrego como hijo derecho de R

        desapilo y lo agrego como hijo izquierdo de R

    apilo R.

**desapilar** ❑ árbol de expresión posfija final



- Prefija: Este proceso es posible ya que la expresión prefija está organizada en una forma en la que los operadores siempre aparecen antes de los operandos. Cuando se llega a las hojas, la recursión retorna y permite ir armando el árbol desde abajo hacia arriba.

#### Estrategia II:

**convertir(expr\_prefija)**

tomo primer carácter de la expresión

creo un nodo R con ese operador

**si** el carácter es un operador

    ❑ agrego como hijo izquierdo de R(convertir(expr\_prefija sin 1 carácter()))

    ❑ agrego como hijo derecho de R(convertir(expr\_prefija sin 1 carácter()))

**//es un operando**

devuelvo el nodo R



### 3.3 Arboles generales

Para un árbol general, se sigue respetando la definición de árbol binario, a diferencia de que cada nodo puede tener uno o más hijos.

- Árbol lleno: Sea T un árbol lleno de grado k y altura h, la cantidad de nodos N es

$$\frac{k^{h+1}-1}{k-1}$$

- Árbol completo: Sea T un árbol completo de grado k y altura h, la cantidad de nodos N varía entre

$$\left[ \frac{k^h + k - 2}{k - 1}; \frac{k^{h+1} - 1}{k - 1} \right]$$

### 3.3.1 Recorridos

- 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.

## 4 Cola de prioridad (HEAP)

### 4.1 Implementaciones

Una cola de prioridad es una estructura que nos permite hacer inserciones y eliminaciones en una secuencia de datos, con el agregado de que estas operaciones mantendrán cierta organización y orden para establecer algunas prioridades. Usualmente se utiliza para buscar el mínimo (prioridad cero), recuperarlo, ejecutarlo y eliminarlo, así, se van atendiendo las prioridades mediante un orden.

- Lista ordenada:
  - Insert tiene  $O(n)$  operaciones, ya que solo necesita buscar la posición para insertar.
  - DeleteMin tiene  $O(1)$  operaciones, ya que elimina el 1er dato (mínimo).
- Lista no ordenada:
  - Insert tiene  $O(1)$  operaciones, ya que inserta al inicio o final.
  - DeleteMin tiene  $O(n)$  operaciones, ya que necesita buscar el dato.
- Árbol binario de búsqueda:
  - Para las dos implementaciones tiene en promedio  $O(\log n)$  operaciones.
  - La altura  $h$ , es de  $O(\log n)$  y lo importante de esto, es que las dos operaciones recorren la altura del árbol.

### 4.2 Heap

Una heap es un árbol binario completo de altura  $h$ , por lo que podrá ser almacenado en un arreglo de dimensión  $N$ . Se pueden insertar y eliminar elementos libremente, pero se deben respetar las siguientes propiedades:

#### 4.2.1 Propiedad estructural

Para buscar al padre o los hijos de un elemento del arreglo, tengo que ver su posición ( $i$ ) y hacer alguna operación.

- Subárbol izquierdo:  $2*i$
- Subárbol derecho:  $2*i + 1$
- Padre:  $[i/2]$

**Importante:** el primer elemento del arreglo está en la posición 1, por convención.

Ejemplo: si tengo al elemento de la posición 4 ( $i=4$ ), su hijo izquierdo estará en la posición 8 ( $2*i$ ), su hijo derecho estará en la posición 9 ( $2*i + 1$ ) y su padre estará en la posición 2 ( $i/2$ ).

#### 4.2.2 Propiedad de orden

El orden de los elementos dependerá del tipo de heap que estemos tratando.

- MinHeap: El elemento mínimo está al inicio del arreglo, por consecuencia, el dato almacenado es menor o igual al de sus hijos.
- MaxHeap: Propiedad inversa.



### 4.2.3 Restauracion

Luego de realizadas las operaciones de insercion y eliminacion, se debe volver a la propiedad de orden inicial de la heap.

- **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 un valor menor.
- **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 dónde insertarlo.

Ya que ambos algoritmos recorren la altura de la heap, tienen  $O(\log n)$  operaciones de intercambio.

La ventaja de usar una heap para ordenar vectores, es que ahorramos espacio en memoria mientras mantenemos la misma cantidad de operaciones  $O(n \log(n))$ .

## 5 Tiempo de ejecución

Para una expresion  $T(n) = O(f(n))$  se dice que  $T(n)$  tiene un orden de  $f(n)$ , que representa una cota superior de la funcion  $T(n)$ . La tasa de crecimiento de  $T(n)$  sera menor o igual a la de  $f(n)$ , y al ser una cota superior, si no se esta seguro de su valor exacto se puede aproximar.

Por ejemplo, es comun decir que si tengo  $T(n) = cte + \log(n) = O(n)$ , ya que me representa una cota superior de la funcion.

### 5.1 Algoritmo constante

Un algoritmo tendra  $O(1)$  siempre que no contenga un ciclo, un llamado a una funcion, o recursividad. Por ejemplo aca entran los condicionales, asignaciones, operaciones basicas y tambien los ciclos que no dependan de la variable de entrada.

### 5.2 Algoritmos iterativos

#### 5.2.1 For

Usualmente, para el caso del for, el calculo de tiempo dependera tanto de la cantidad de datos que se ingresen como de la cantidad de for anidados que se tengan. Para un caso de  $n$  datos y dos for se tiene:

$$T(n) = cte1 + \sum \sum cte2 = cte1 + n*n*cte2 \Rightarrow O(n^2)$$

En el caso de que el for se incremente multiplicando una constante, será  $O(\log n)$ .

#### 5.2.2 While

Para el caso de while es un poco mas complejo, porque no se sabe la cantidad de iteraciones que el algoritmo tendrá, por lo que se opta decir que tiene orden:

$$T(n) = cte1 + cte2 * \log(n) \Rightarrow O(\log n)$$

Por ejemplo, si se tiene un for anidado con un while, el tiempo sera de  $O(n * \log(n))$

## 5.3 Algoritmos recursivos

En la recursividad, se debe ir evaluando el tiempo de cada llamada a la funcion, usando  $i$  como parametro de la cantidad de llamados. Una vez llegada a una expresion, se analiza el caso base para descartar  $i$  y dar el orden final.

## 5.4 Big Oh

Se dice que  $T(n)$  es de  $O(f(n))$  si cumple con que:  $T(n) \leq c * f(n), n \geq n_0$

Tabla de tiempos:

Notation	Type	Examples	Description
$O(1)$	Constant	Hash table access	Remains constant regardless of the size of the data set
$O(\log n)$	Logarithmic	Binary search of a sorted table	Increases by a constant. If $n$ doubles, the time to perform increases by a constant, smaller than $n$ amount
$O(<n)$	Sublinear	Search using parallel processing	Performs at less than linear and more than logarithmic levels
$O(n)$	Linear	Finding an item in an unsorted list	Increases in proportion to $n$ . If $n$ doubles, the time to perform doubles
$O(n \log(n))$	$n \log(n)$	Quicksort, Merge Sort	Increases at a multiple of a constant
$O(n^2)$	Quadratic	Bubble sort	Increases in proportion to the product of $n*n$
$O(c^n)$	Exponential	Travelling salesman problem solved using dynamic programming	Increases based on the exponent $n$ of a constant $c$
$O(n!)$	Factorial	Travelling salesman problem solved using brute force	Increases in proportion to the product of all numbers included (e.g., $1*2*3*4...$ )

## 6 Grafos

### 6.1 Terminologia

Un grafo es una estructura de datos no lineal que consiste en un conjunto finito de vertices ( $V$ ) y un conjunto finito de aristas ( $E$ ). Existen dos tipos de grafos

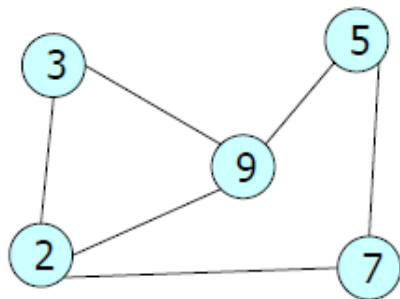
- **Dirigidos:** la relacion ( $V,E$ ) es no simetrica. Se representan con aristas dirigidas que van de vertice a vertice. El grado de un vertice dirigido se da mediante dos formas:
  - indegree: es el numero de arcos que inciden en él.
  - outdegree: es el numero de arcos que salen de él.
- **No dirigido:** la relacion ( $V,E$ ) es simetrica. La direccion de las aristas no esta dada. El grado de un vertice es la cantidad de adyacentes.

El grado de un grafo es el grado maximo de alguno de sus vertices.

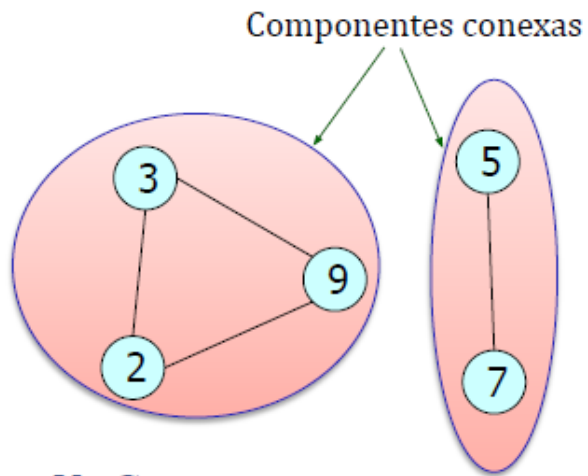
Se dice que dos vertices son adyacentes si existe una arista ( $E$ ) que los une, esto se representa como:  $(V,U) \in E$

Tambien, se puede definir a un grafo como ciclico o aciclico, que basicamente nos indica si tenemos un posible loop dentro. A diferencia del bucle, que es cuando tenemos un loop pero no podemos salir de este.

Un grafo no dirigido es conexo si existe un camino entre cada par. En este pueden existir componentes conexas aisladas. Claramente, un árbol se puede ver como un grafo conexo acíclico.

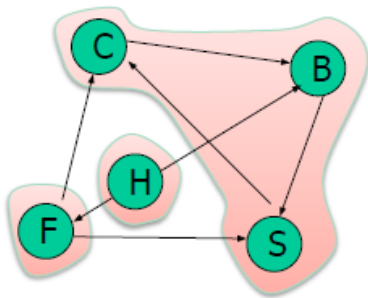


**Conexo**

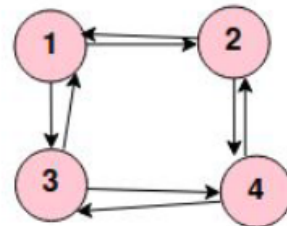


**No Conexa**

Por otro lado, un grafo dirigido se denomina fuertemente conexo si existe un camino para cualquier par de vertices. Si en cambio, el grafo es dirigido pero no tiene caminos entre todos sus vertices se le llama debilmente conexo.



**No Fuertemente Conexa**

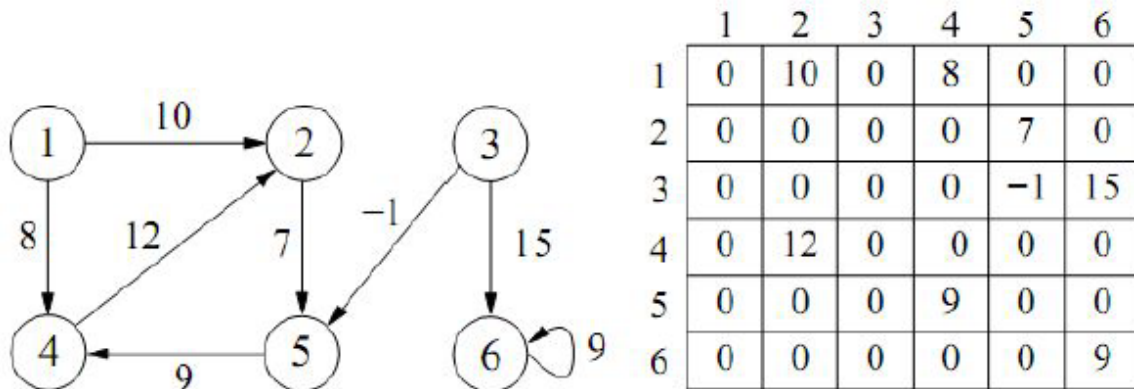


**Fuertemente Conexa**

## 6.2 Representacion

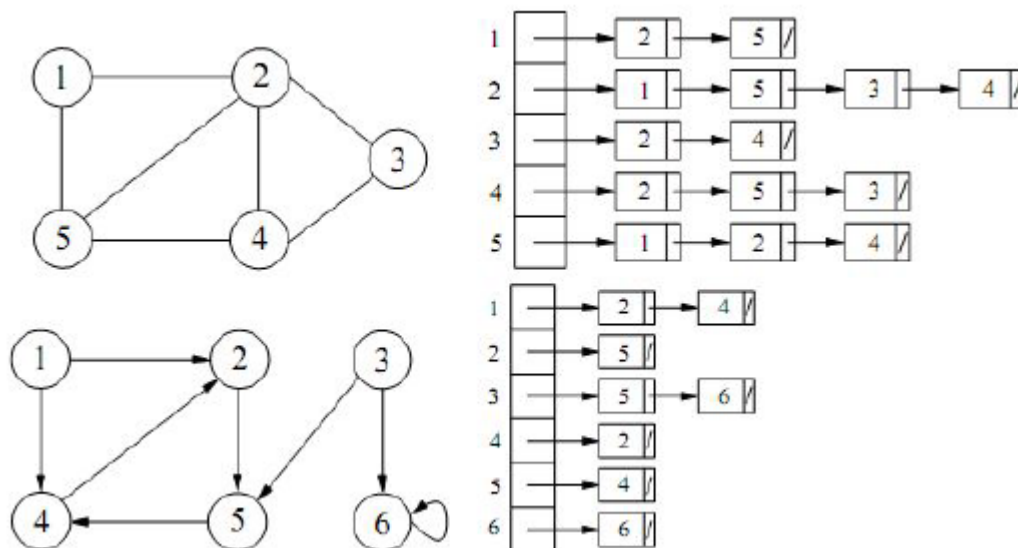
Se puede representar a un grafo mediante una matriz de adyacencias o mediante una lista de adyacencias.

### 6.2.1 Matriz de adyacencias



Costo de tiempo:  $T(|V|, |E|)$ . Costo espacial:  $O(|V|^2)$ . Acceso a una arista  $O(1)$ .

### 6.2.2 Lista de adyacencias



Si el grafo es dirigido, la suma de las longitudes de adyacencia es  $|E|$ , mientras que si no es dirigido será  $2 * |E|$ . Costo espacial  $O(|V| + |E|)$ . La desventaja, es que el acceso a una arista ya no es de  $O(1)$ , ya que hay que recorrer la lista.

## 6.3 Recorridos

### 6.3.1 DFS (Depth First Search)

Es un algoritmo de búsqueda utilizado para recorrer todos los nodos de un grafo o árbol de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto.

Estrategia generalizacion del preorden:

1. Partir de un vértice determinado  $v$ .
2. Cuando se visita un nuevo vértice, explorar cada camino que salga de él.

3. Hasta que no se haya finalizado de explorar uno de los caminos no se comienza con el siguiente
4. Un camino deja de explorarse cuando se llega a un vértice ya visitado -o sin adyacentes-.
5. 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.

El tiempo del dfs es el tiempo que tarda en recorrer cada una de las aristas, ya que una vez que visita deja la marca así no vuelve a pasar por ella, además hay que agregarle el tiempo que visita cada vertice, entonces:  $O(|V| + |E|)$ .

### 6.3.2 BFS (Breath First Search)

El algoritmo de búsqueda por amplitud, básicamente se encarga de recorrer el grafo «por niveles».

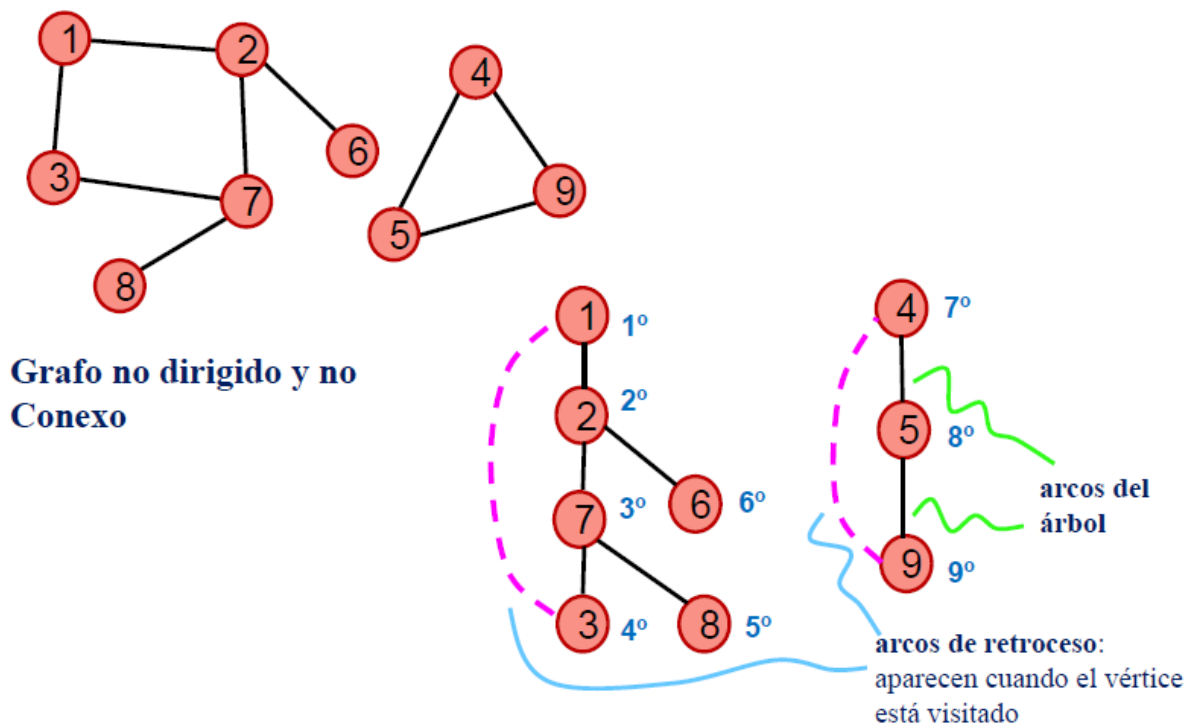
Estrategia generalizada por niveles:

1. Partir de algún vértice  $u$ , visitar  $u$  y, después, visitar cada uno de los vértices adyacentes a  $u$ .
2. Repetir el proceso para cada nodo adyacente a  $u$ , siguiendo el orden en que fueron visitados.
3. 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  $O(|V| + |E|)$

### 6.3.3 Bosque de expansion DFS

El recorrido de un grafo no es único, depende del nodo inicial y del orden de la visita de los adyacentes. Este recorrido puede ser visto como un árbol en profundidad asociado a un grafo. Si aparecen muchos árboles, se le llama bosque de expansion.



### 6.3.4 Aplicaciones del DFS

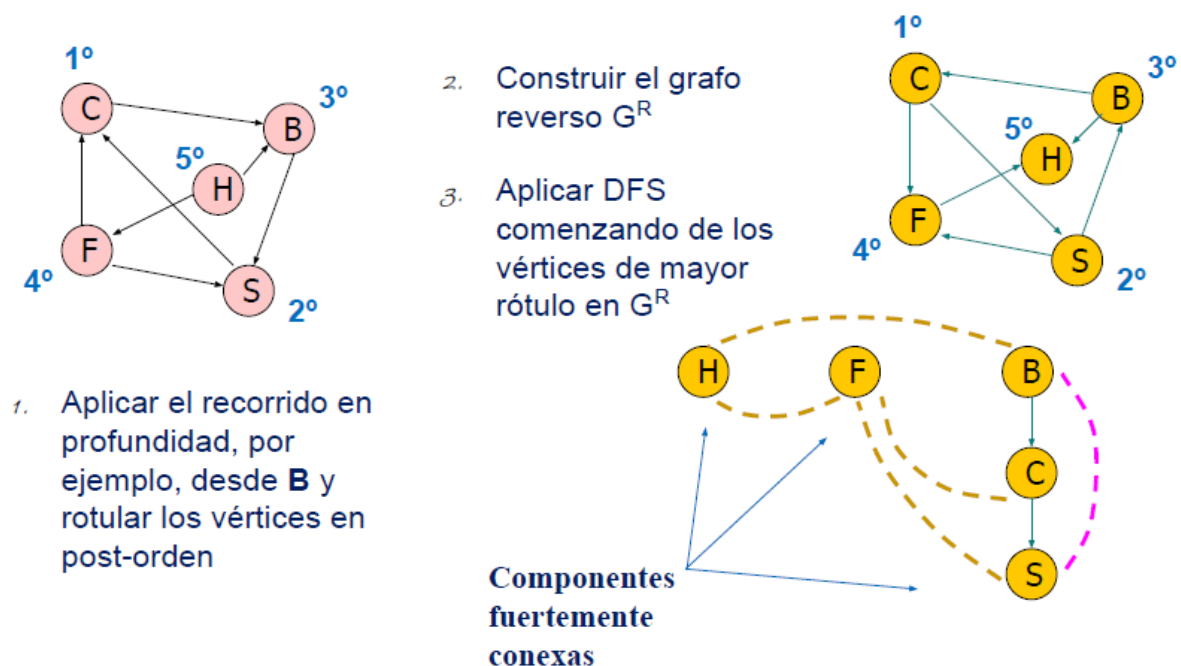
Si el grafo es fuertemente conexo, existirá un recorrido que permita salir desde cualquier vertice y visitarlos todos.

### 6.3.5 Algoritmo de Kosajaru

El algoritmo de Kosajaru sirve para encontrar componentes fuertemente conexas dentro de un arbol dirigido. Pasos:

1. Aplicar DFS( $G$ ) rotulando los vértices de  $G$  en post-orden (apilar).
2. Construir el grafo reverso de  $G$ , es decir  $G^r$  (invertir los arcos).
3. Aplicar DFS ( $G^r$ ) comenzando por los vértices de mayor rótulo (tope de la pila).
4. Cada árbol de expansión resultante del paso 3 es una componente fuertemente conexa.

Si resulta en un unico arbol, entonces el grafo es fuertemente conexo. El orden del algoritmo es  $O(|V| + |E|)$ , ya que se recorren dos DFS y todas las aristas una vez para crear el grafo reverso.



## 6.4 Caminos minimos

Los algoritmos calculan los caminos mínimos desde un vértice origen  $s$  a todos los restantes vértices del grafo.

### 6.4.1 Grafos sin peso

Este recorrido esta basado en BFS, ya que es un recorrido por niveles.

1. Avanzar por niveles a partir del origen, asignando distancias según se avanza (se utiliza una cola).
2. Inicialmente, es  $D_w = \infty$ . Al inspeccionar  $w$  se reduce al valor correcto  $D_w = D_v + 1$ .
3. Desde cada  $v$  visitamos a todos los nodos adyacentes a  $v$ .

### 6.4.2 Grafos con pesos positivos (Dijkstra)

Pasos:

1. Dado un vértice origen  $s$  elegir el vértice  $v$  que esté a la menor distancia de  $s$  dentro de los vértices no procesados.
2. Marcar  $v$  como procesado.
3. Actualizar la distancia de  $w$  adyacente a  $v$ .

Para cada vértice  $v$  mantiene la siguiente información:

- $D_v$  distancia mínima desde el origen (inicialmente para todos los vértices excepto el origen con valor 0).
- $P_v$  vértice por donde paso para llegar.
- Conocido dato booleano que me indica si está procesado (inicialmente todos en 0).

En caso de almacenar las distancias en un vector, el costo del algoritmo es  $O(|V|^2)$ , mientras que si almacenamos las distancias en una heap, el costo se reduce a  $O(|E| \log(|V|))$

### 6.4.3 Grafos con pesos positivos y negativos

Pasos

1. Encolar el vértice origen  $s$ .
2. Procesar la cola.
3. Desencolar un vértice.
4. Actualizar la distancia de los adyacentes  $D_w$  siguiendo el mismo criterio de Dijkstra.
5. Si  $w$  no está en la cola, encolarlo.

El costo total del algoritmo es  $O(|V| |E|)$ .

### 6.4.4 Grafos aciclicos

El costo total del algoritmo es  $O(|V| + |E|)$ .

### 6.4.5 Algoritmo de Floyd

El costo total el algoritmo es  $O(|V|^3)$ .

### 6.4.6 Resumen Importante

<b>Grafos</b>	<b>BFS <math>O(V+E)</math></b>	<b>Dijkstra <math>O(E \log V)</math></b>	<b>Algoritmo modificado (encola vértices) <math>O(V \cdot E)</math></b>	<b>Optimización de Dijkstra (sort top) <math>O(V+E)</math></b>
<b>No pesados</b>	Óptimo	Correcto	Malo	Incorrecto si tiene ciclos
<b>Pesados</b>	Incorrecto	Óptimo	Malo	Incorrecto si tiene ciclos
<b>Pesos negativos</b>	Incorrecto	Incorrecto	Óptimo	Incorrecto si tiene ciclos
<b>Grafos pesados aciclicos</b>	Incorrecto	Correcto	Malo	Óptimo

Correcto  $\rightarrow$  adecuado pero no es el mejor

Malo  $\rightarrow$  una solución muy lenta

## 6.5 Ordenacion topologica

Una ordenacion topologica es como una ordenacion de los vertices a lo largo de una linea horizontal, con los arcos de izquierda a derecha. Este recorrido no es unico, ya que depende del orden visitado de las aristas. Esta ordenacion se usa para establecer prioridades o organizar tareas de pre-requisitos.

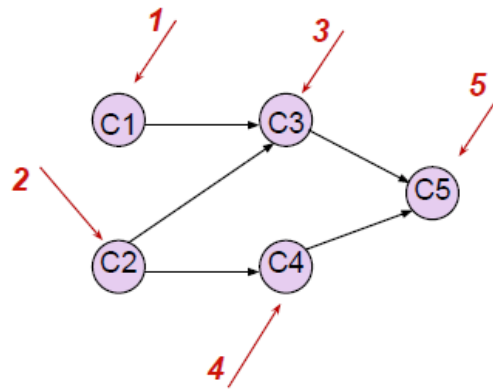
### 6.5.1 Version 1

En esta versión el algoritmo utiliza un arreglo en el que se almacenan los grados de entradas de los vértices y en cada paso se toma de allí un vértice con  $\text{grado\_in} = 0$  y se lo procesa. Continuar con el procedimiento hasta visitar todos los nodos.

→ *Tomando vértice con  $\text{grado\_in} = 0$  del vector  $\text{Grado\_in}$*

Grado\_in

C1	C2	C3	C4	C5
0	0	2	1	2
0	0	1	1	2
0	0	0	0	2
0	0	0	0	1
0	0	0	0	0



**Sort Topológico :**

**C1 C2 C3 C4 C5**

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

El tiempo total del algoritmo es  $O(|V|^2 + |E|)$

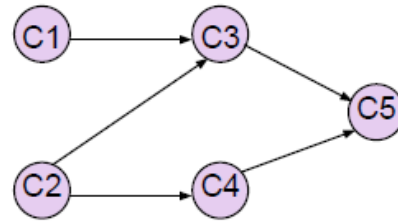
### 6.5.2 Version 2

En esta versión el algoritmo utiliza un arreglo  $\text{Grado\_in}$  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. El tiempo total del algoritmo es  $O(|V| + |E|)$



Grado\_in

C1	C2	C3	C4	C5
0	0	2	1	2
0	0	1	0	2
0	0	1	0	1
0	0	0	0	1
0	0	0	0	0



Pila **P** : **C1** – **C2**

: C1 // C1 – **C4**

: C1 // C1

: // **C3**

: // **C5**

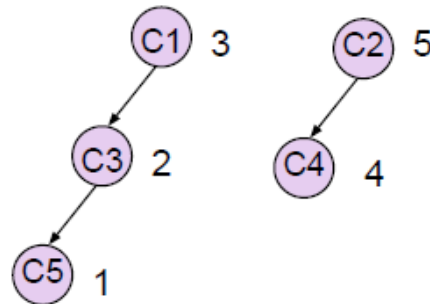
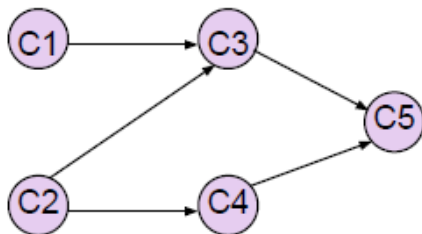
**Sort Topológico :**

**C2 C4 C1 C3 C5**

### 6.5.3 Version 3

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



*Dado un grafo dirigido acíclico, se aplica DFS a partir de un vértice cualquiera, por ejemplo C1*

**Ordenación Topológica: C2 C4 C1 C3 C5**

b) apilando los vértices: se colocan en una pila P, luego se listan empezando por el tope.

*Dado un grafo dirigido acíclico*

- 1. Se aplica DFS a partir de un vértice cualquiera, por ejemplo C1, y se apilan los vértice en post-orden.*
- 2. Listo los vértices a medida que se desapilan.*

