

Estructuras de Datos.

Grado en Informática, Ingeniería del Software y Computación

ETSI Informática

Universidad de Málaga

## 6. Grafos

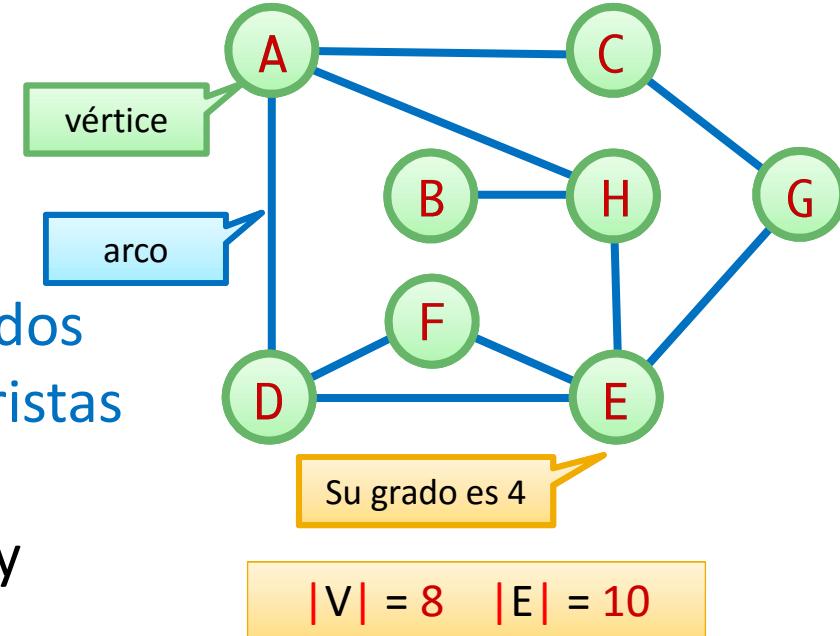
@ José E. Gallardo, Francisco Gutiérrez, Pablo López  
Dpto. Lenguajes y Ciencias de la Computación  
Universidad de Málaga

# Índice

- Grafos
  - Terminología
  - Representación en Haskell (12)
- Recorrido de grafos
  - Recorrido en profundidad (Depth First Traversal) (17). Propiedades (38). Implementación (52).
  - Recorrido en amplitud (Breadth First Traversal) (113). Propiedades (125). Implementación (126).
  - DFT vs BFT. Aplicaciones (162), grafos bipartitos (172)
    - El problema de los caníbales (174)
- Grafos dirigidos (179). Representación en Haskell
- El problema de la jarras (186)
- Representación clásica de los grafos (192)
- Grafos en Java (203)
- Componentes conexas (229)
- Ordenación topológica (233)
- Ciclos y ordenación topológica (274)
- Resumen. Problemas clásicos sobre grafos (278)

# Grafos

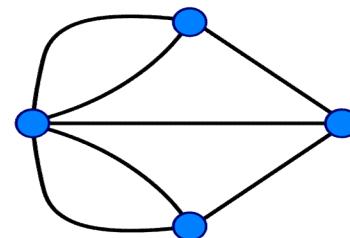
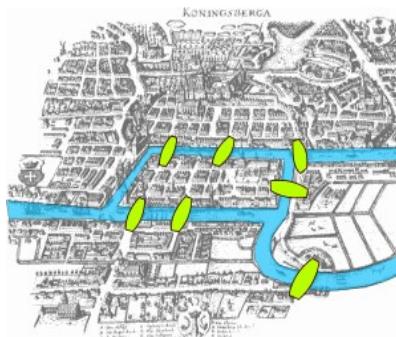
- **Grafo:** Conjunto de objetos, algunos de ellos conectados dos a dos.
- Los objetos se llaman **vértices** o **nodos**
- Las conexiones se llaman **arcos** o **aristas**
- $|V|$  denota el número de vértices, y  $|E|$  el número de aristas
- Si existe la arista  $v-v'$ , los vértices  $v$  y  $v'$  se dicen **adyacentes**;  $v'$  se dirá un **sucesor** de  $v$ , y recíprocamente,  $v$  es un sucesor de  $v'$
- **Grado** del vértice  $v$ : es el número de aristas incidentes, o sea, el número de vértices adyacentes a  $v$ .



# Grafos (II<sub>a</sub>) Origen de esta teoría

El problema de los puentes de Königsberg  
Leonhard Euler (1707 - 1783) resuelve el problema en 1736.

Königsberg  
(Kalinkingrado) en  
1736

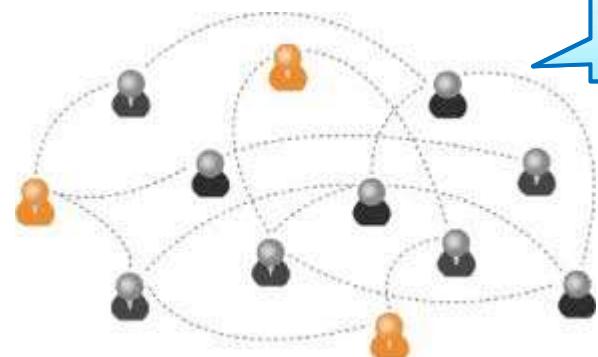


abstracción  
de Euler: una  
arista para  
cada puente

- Cada vértice  $v$  intermedio de un recorrido de Euler debe tener grado par (si entramos en  $v$ , salimos por una arista distinta). Luego el inicial y el final son los únicos que pueden tener grado impar.
- Luego: es necesario que “el número de vértices de grado impar sea o bien 0 (circuito) o bien 2 (camino no cerrado)” (el problema de los puentes no tiene solución) (*prueba con otros dibujos*).
- La demostración de que esta condición es suficiente es de Carl Hierholer (1873) (Ejercicio 6.5).

# Grafos (II<sub>b</sub>)

- Los grafos constituyen una potente abstracción para representar la solución de muchos problemas
- Estudiaremos algoritmos sobre grafos para resolver estos problemas

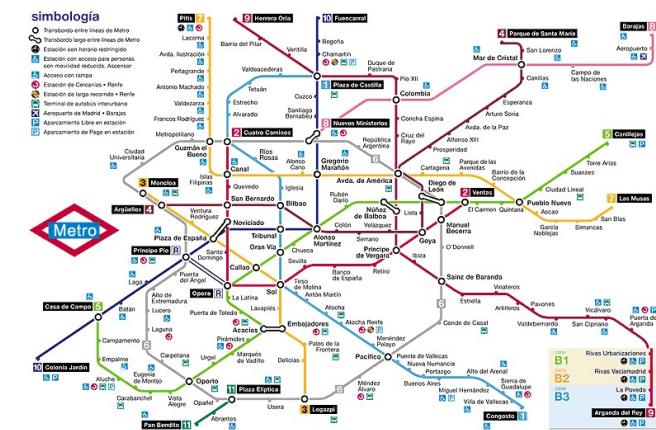


Redes Sociales



El cerebro

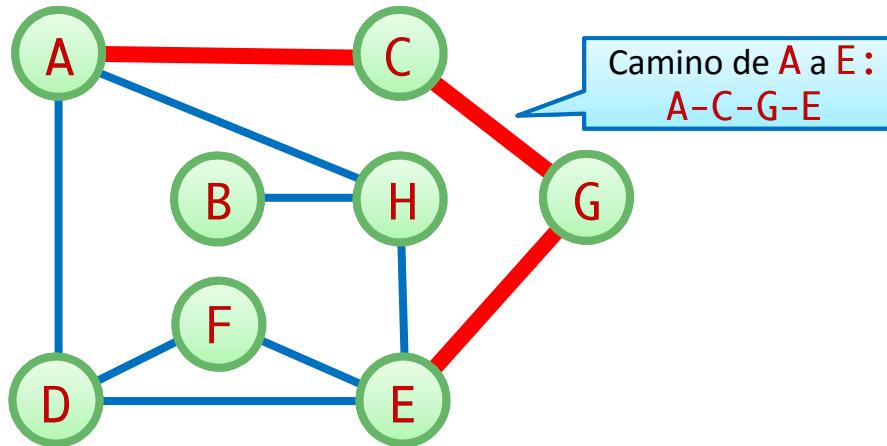
Internet en  
España



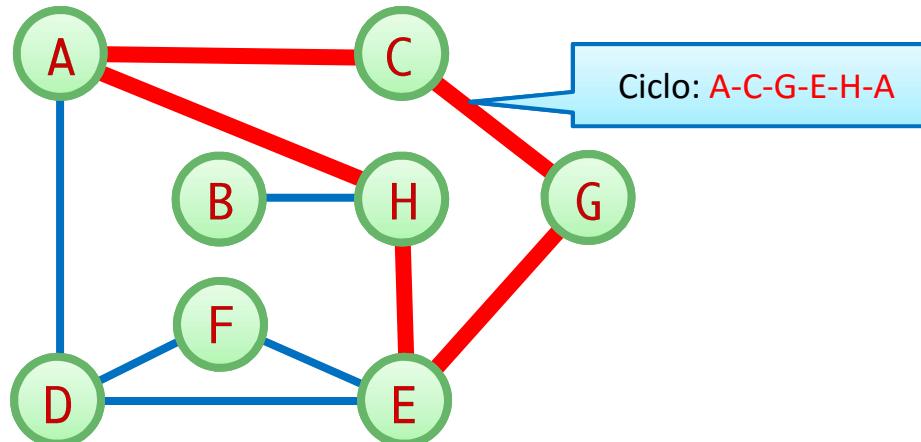
El metro de  
Madrid

# Grafos (III)

- Camino (path): secuencia de vértices conectados por aristas

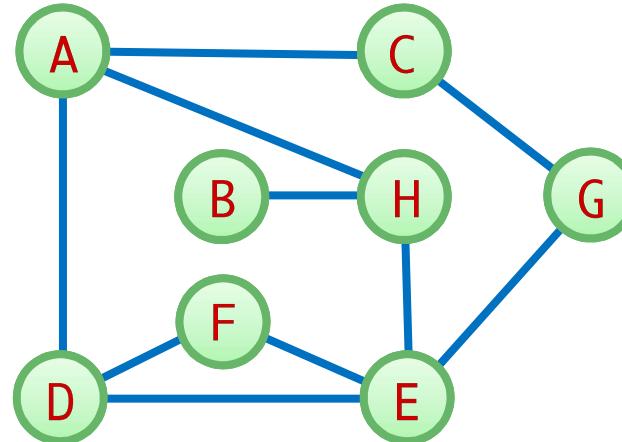


- Ciclo (cycle): camino cerrado donde todas las aristas recorridas son distintas

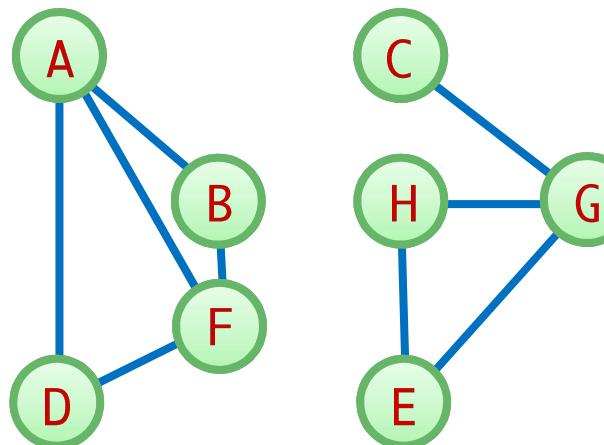


# Grafos (IV)

- Grafo **conexo**: dos vértices cualesquiera están conectados con un camino

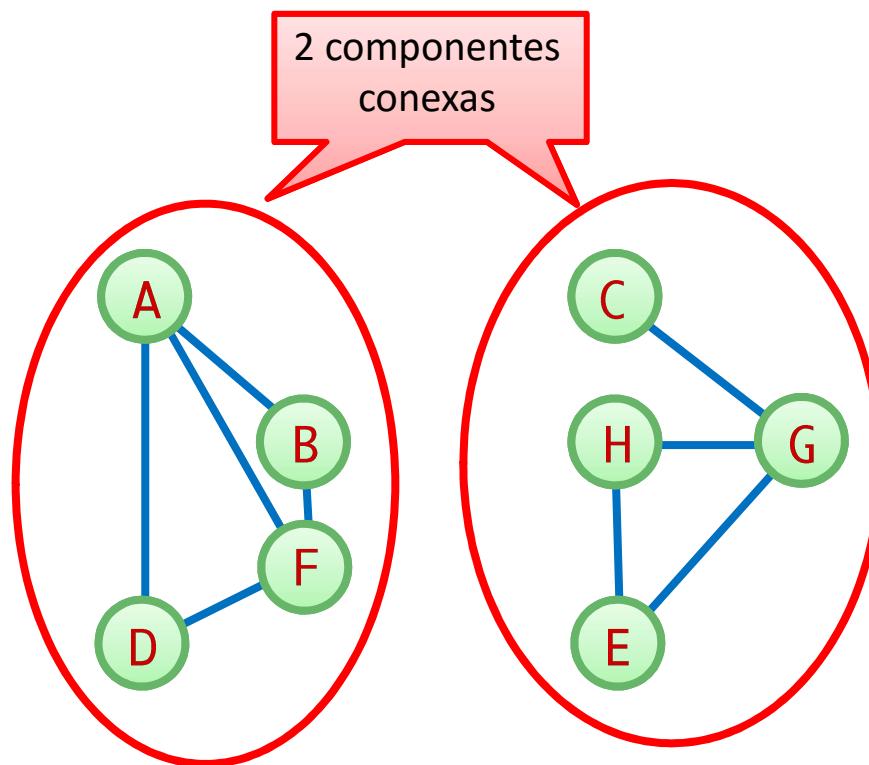


- Grafo **no conexo**



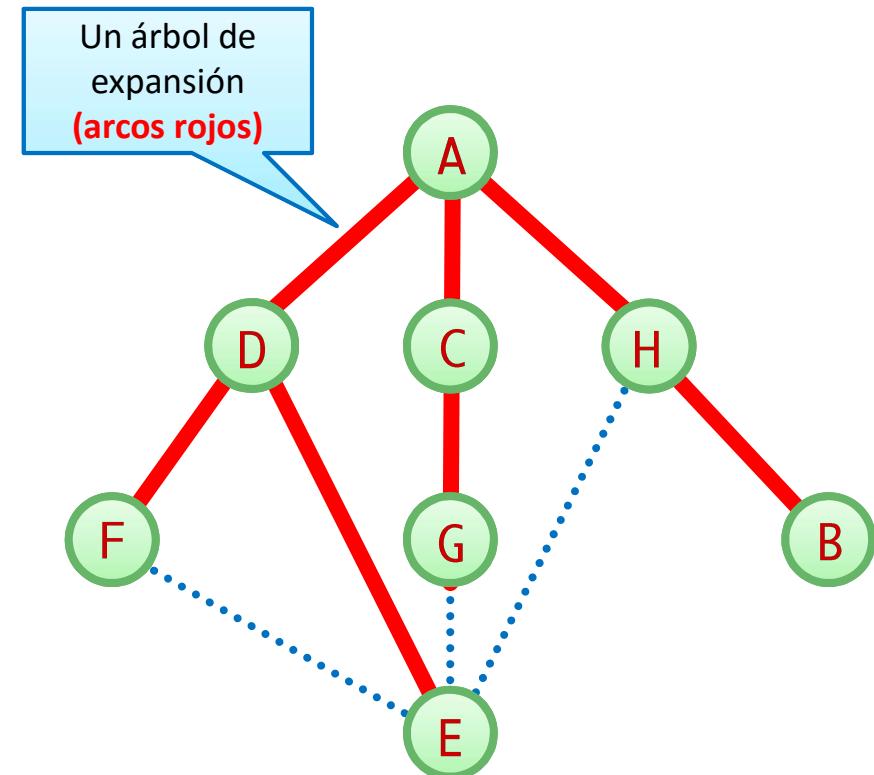
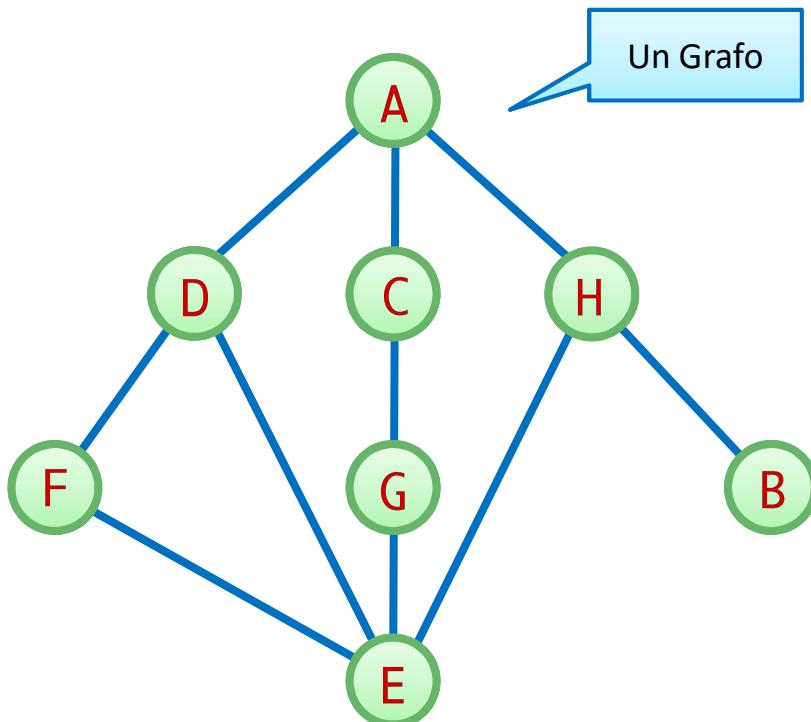
# Grafos (V)

- Componentes conexas: subgrafos maximales conexos



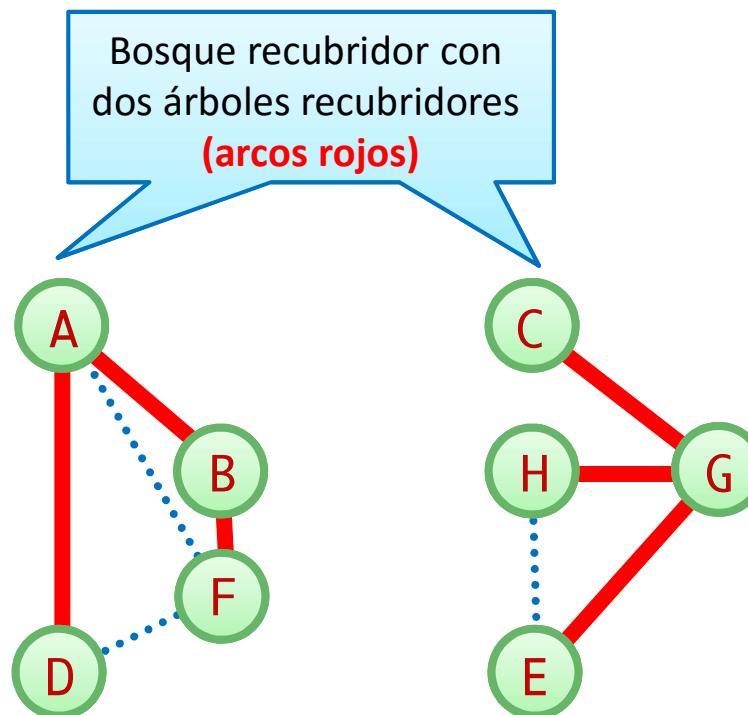
# Grafos (VI)

- Un **Árbol (Tree)** es un grafo conexo acíclico
- Un **Árbol recubridor o de expansión (Spanning Tree)** de un grafo conexo es un árbol subgrafo que contiene todos los vértices del grafo



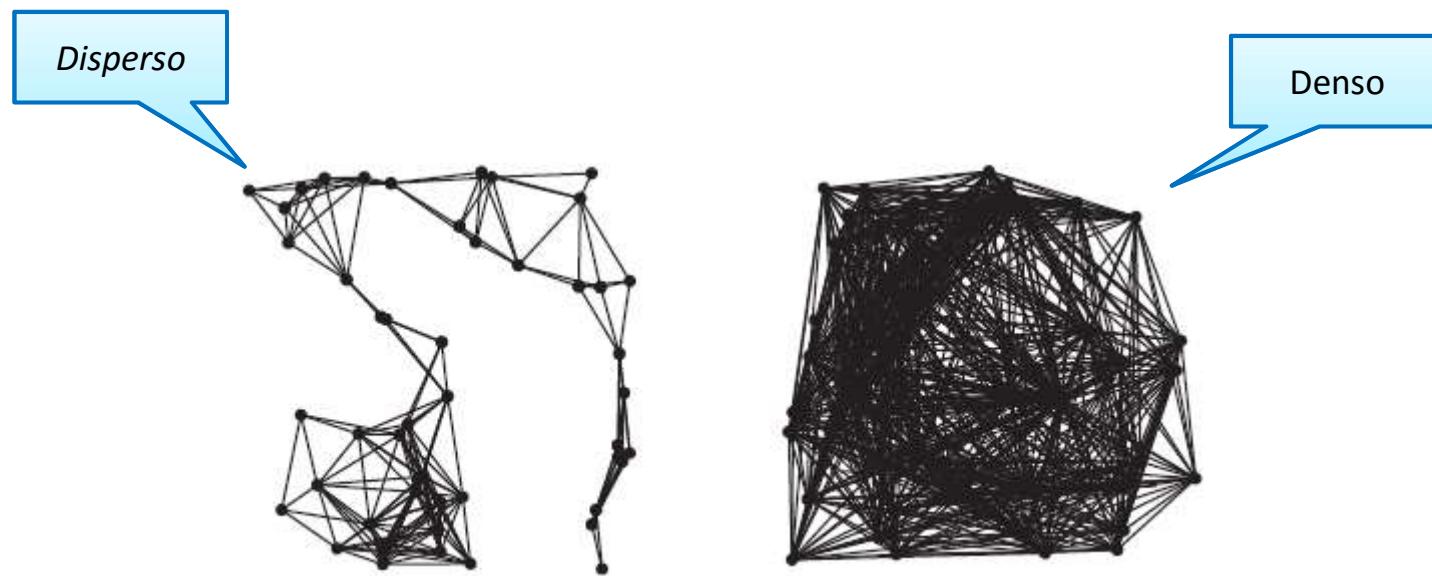
# Grafos (VII)

- Bosque recubridor (Spanning forest): árboles recubridores de cada componente conexa



# Grafos (VIII)

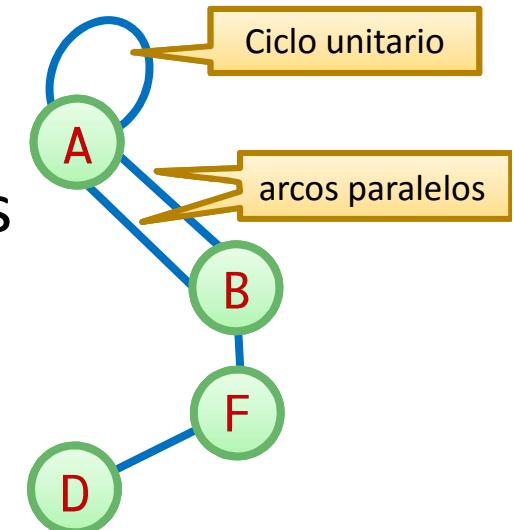
- **Densidad**: proporción de aristas con respecto a vértices
- **Grafo disperso (sparse graph)**: baja densidad
- **Grafo denso**: alta densidad de aristas



- Un grafo se dice *disperso* si  $|E| < |V| \log |V|$

# Grafos (IX)

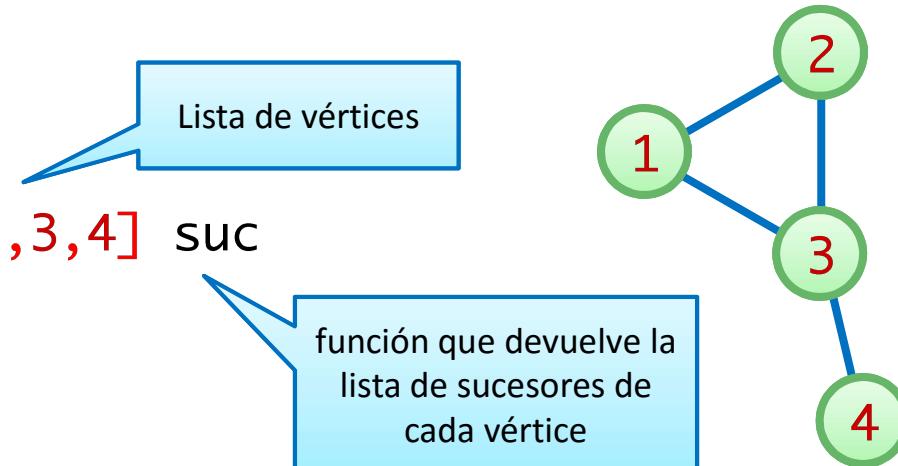
- Anomalías:
  - Ciclo unidad (Self loop): arista de un vértice a él mismo
  - Aristas paralelas: aristas que conectan los mismos vértices
- Multigrafo: pueden existir aristas paralelas
- Grafo simple: sin aristas paralelas ni ciclos unidad (solo estudiaremos éstos).



# Construcción de Grafos en Haskell

- A través de vértices y una función para los sucesores:

```
import Graph  
  
g1 :: Graph Int  
g1 = mkGraphSuc [1,2,3,4] suc  
where  
    suc 1 = [2,3]  
    suc 2 = [1,3]  
    suc 3 = [1,2,4]  
    suc 4 = [3]
```



- A través de vértices y aristas:

```
g1' :: Graph Int  
g1' = mkGraphEdges [1,2,3,4] [(1,2), (1,3), (2,3), (3,4)]
```



# Representación de Grafos en Haskell

```
module DataStructures.Graph.Graph
( Graph
, Edge
, Path
, mkGraphSuc
, mkGraphEdges
, successors
, vertices
, edges
, degree
) where
```

# Representación de Grafos en Haskell (II)

```
data Graph a = G [a] (a -> [a])
```

la implementación contiene la lista de vértices y la función *sucesores*

```
mkGraphSuc :: [a] -> (a -> [a]) -> Graph a  
mkGraphSuc vs suc = G vs suc
```

Trivial:  $\text{mkGraphSuc} = \text{G}$

```
type Edge a = (a,a)
```

Prelude> nub [1,7,3,1,1,2,3]  
[1,7,3,2]

```
mkGraphEdges :: (Eq a) => [a] -> [Edge a] -> Graph a  
mkGraphEdges vs es = G vs suc
```

where

```
suc v = nub ( [ y | (x,y) <- es, x==v ]
```

++

```
 [ x | (x,y) <- es, y==v ] )
```

define la función sucesor conocidas las aristas

# Representación de Grafos en Haskell (III)

-- un camino es una lista de vértices

```
type Path a = [a]
```

-- sucesores de un vértice v

```
successors :: Graph a -> a -> [a]
```

```
successors (G vs suc) v = suc v
```

-- Todos los vértices

```
vertices :: Graph a -> [a]
```

```
vertices (G vs suc) = vs
```

-- grado de un vértice

```
degree :: Graph a -> a -> Int
```

```
degree g v = length (successors g v)
```

# Recorridos de Grafos

- Un **recorrido de un grafo (graph traversal)** es un proceso o método que visita cada vértice
- Son la clave de otros algoritmos más sofisticados
- Recorridos más usuales:
  - Recorrido en profundidad (Depth First Traversal)
  - Recorrido en amplitud o anchura (Breadth First Traversal)

# Recorrido de un Grafo en Profundidad

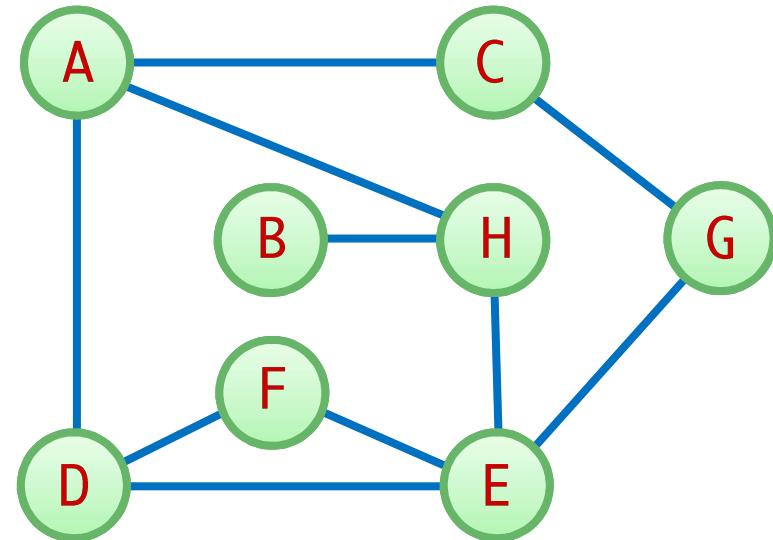
- Algoritmo para un recorrido en profundidad (depth first traversal, o dft) comenzando desde un vértice  $v$

Algoritmo Recursivo

- $\text{DFT}(v)$ 
  - Visitar  $v$  (*anotar, colecciónar, numerar ...*)
  - Para cada vértice  $w$  sucesor de  $v$  que no ha sido visitado
    - $\text{DFT}(w)$

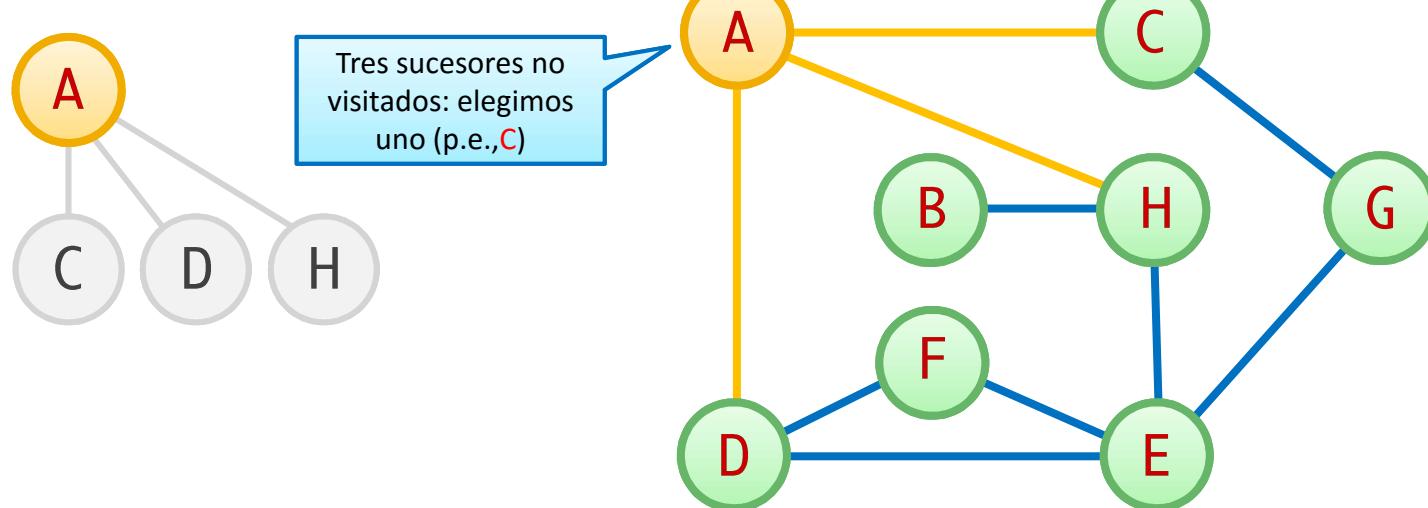
# Recorrido de un Grafo en Profundidad (II<sub>1</sub>)

- DFT comenzando en A



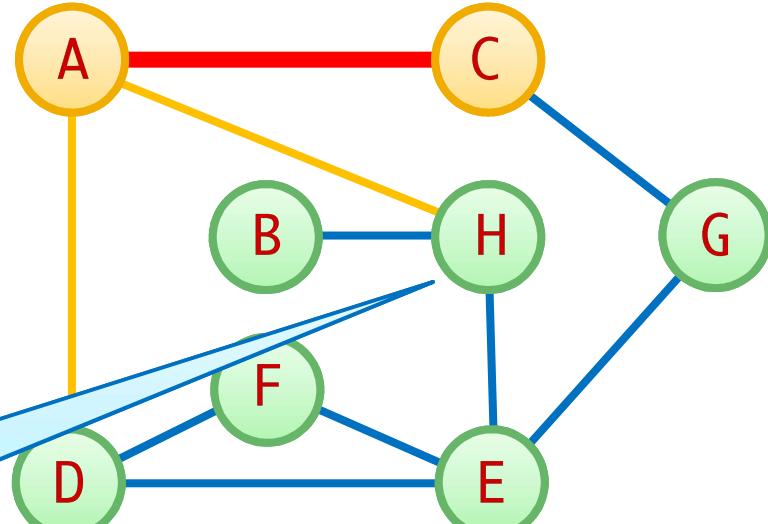
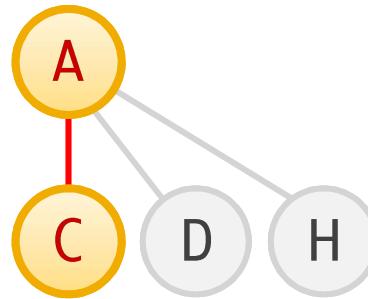
# Recorrido de un Grafo en Profundidad (II<sub>2</sub>)

- DFT comenzando en A



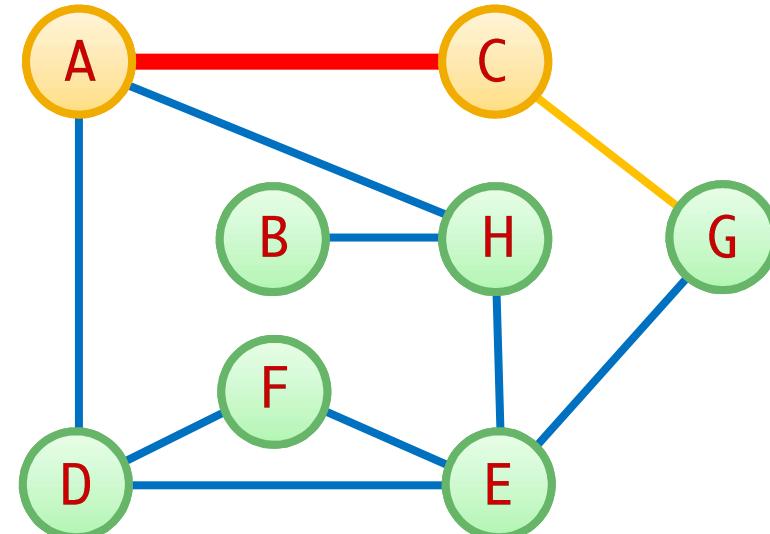
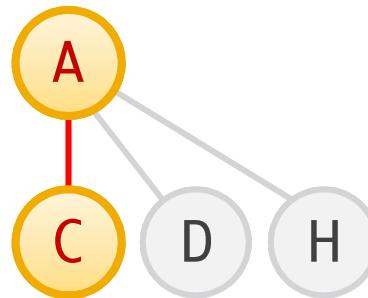
# Recorrido de un Grafo en Profundidad (II<sub>3</sub>)

- DFT comenzando en A



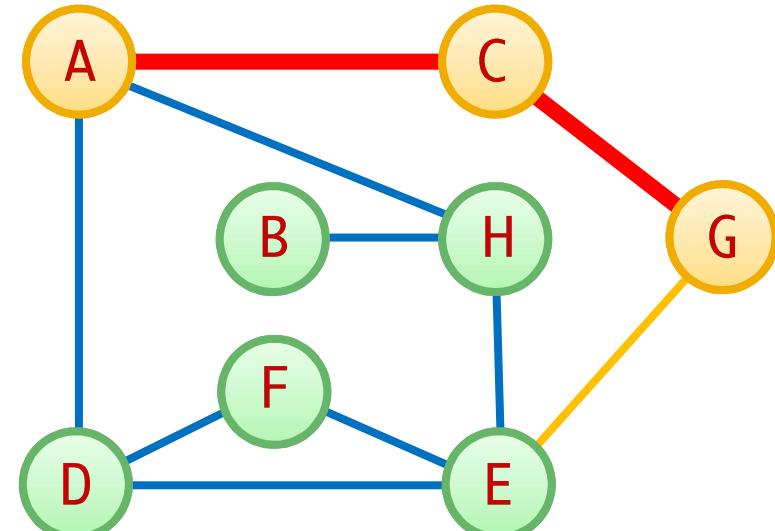
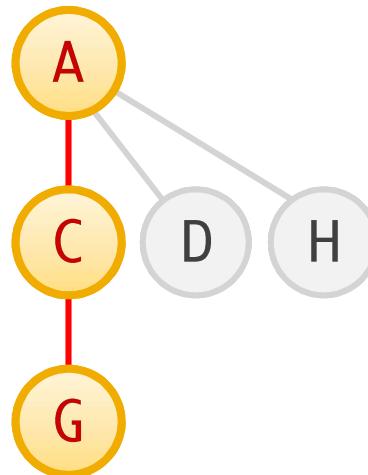
# Recorrido de un Grafo en Profundidad (II<sub>4</sub>)

- DFT comenzando en A



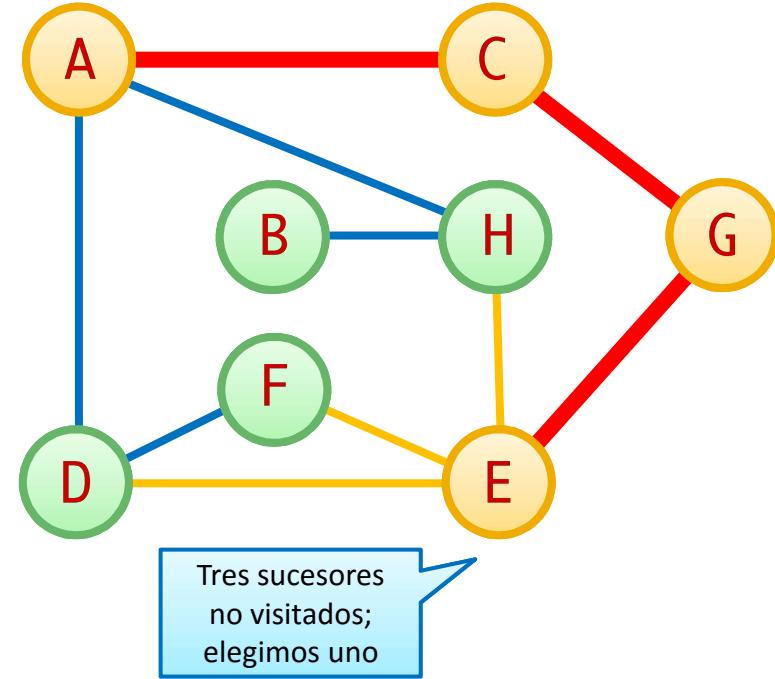
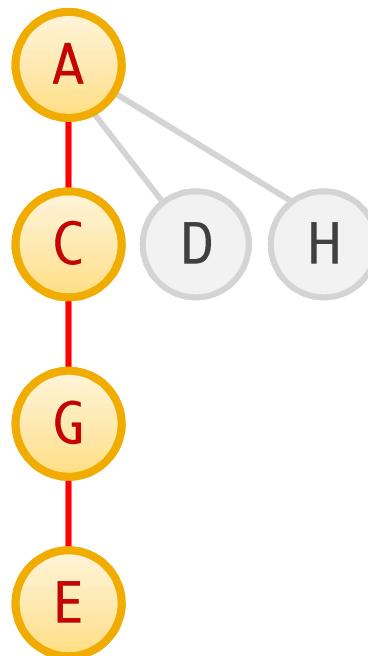
# Recorrido de un Grafo en Profundidad (II<sub>5</sub>)

- DFT comenzando en A



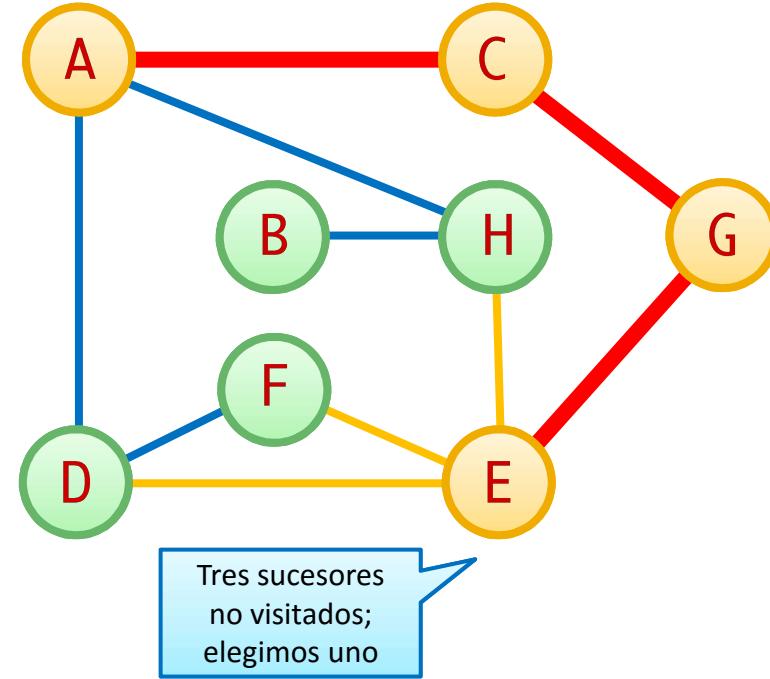
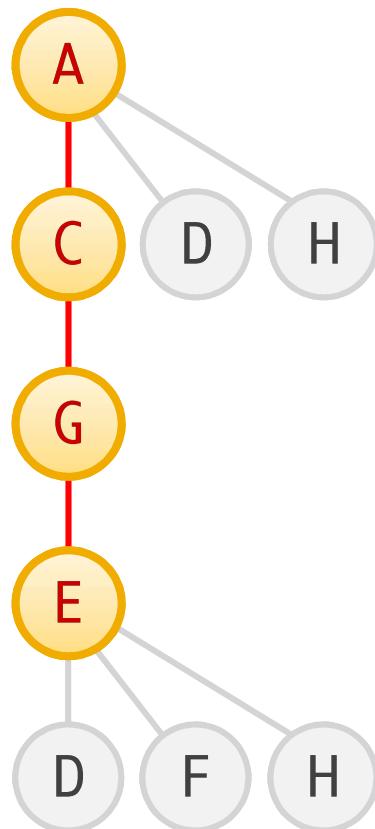
# Recorrido de un Grafo en Profundidad (II<sub>6</sub>)

- DFT comenzando en A



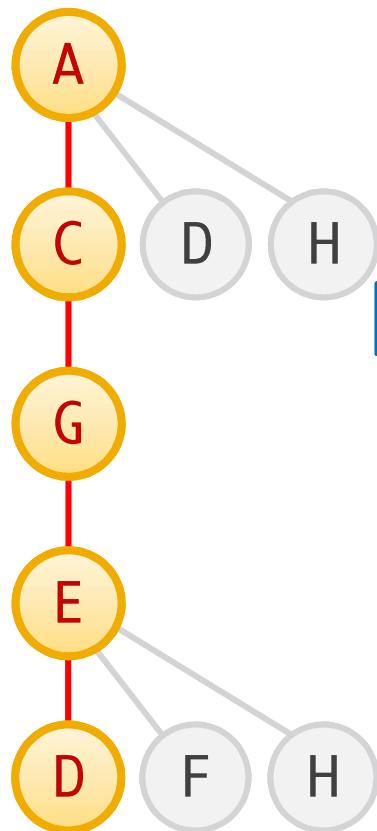
# Recorrido de un Grafo en Profundidad (II<sub>7</sub>)

- DFT comenzando en A

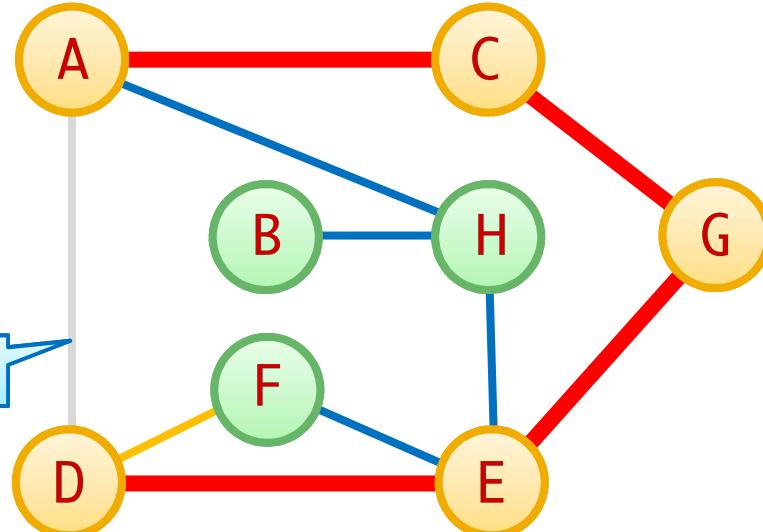


# Recorrido de un Grafo en Profundidad (II<sub>8</sub>)

DFT comenzando en A

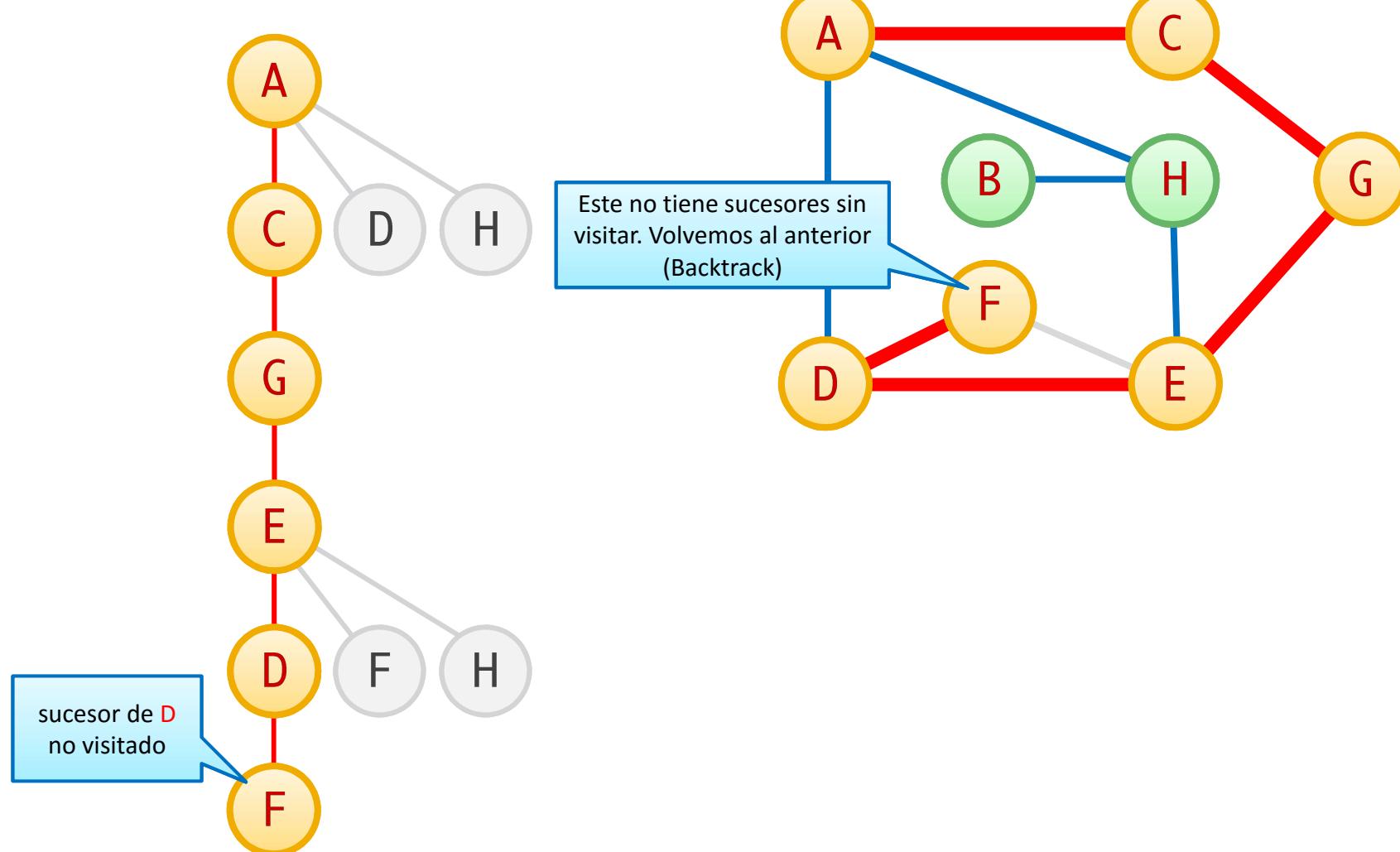


A ha sido visitado



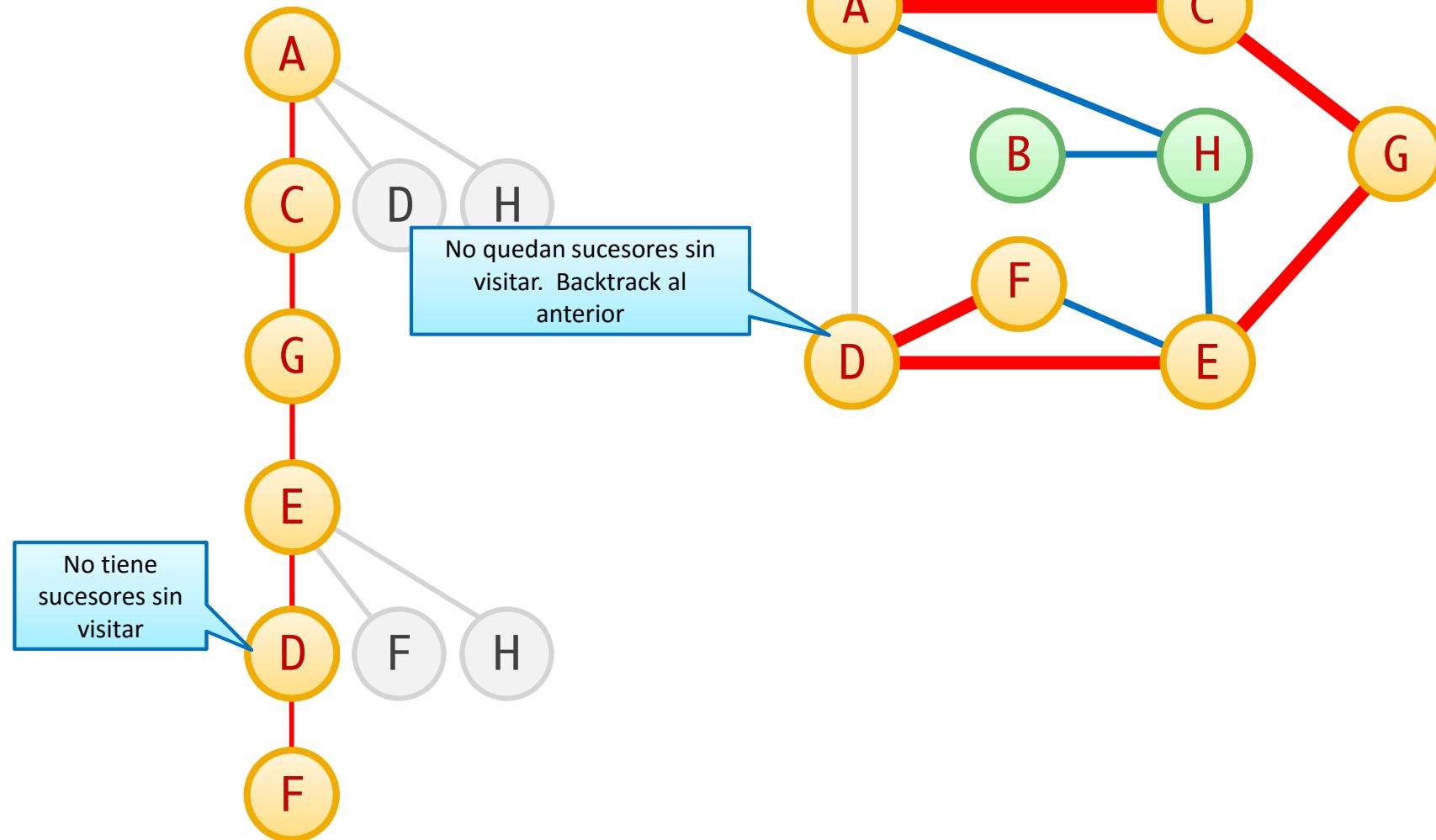
# Recorrido de un Grafo en Profundidad (II<sub>9</sub>)

- DFT comenzando en A



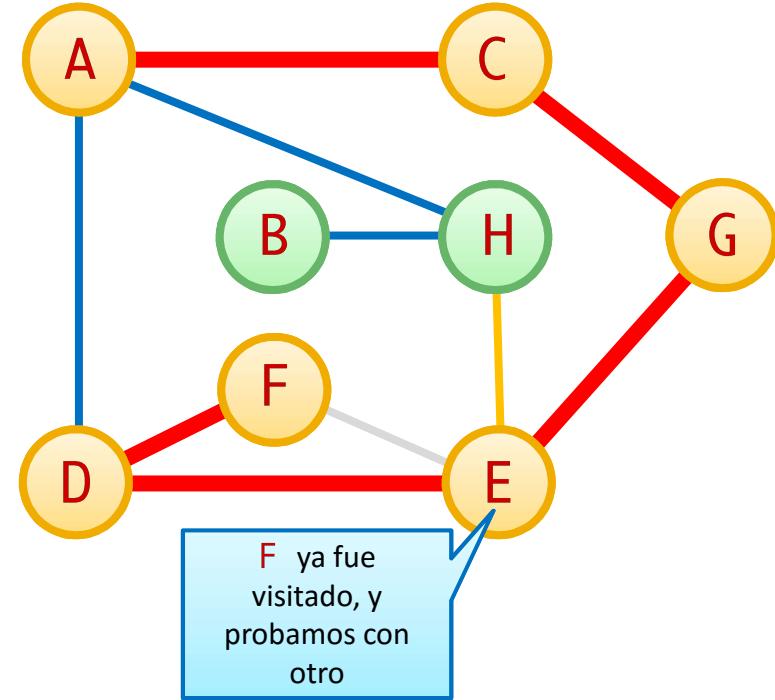
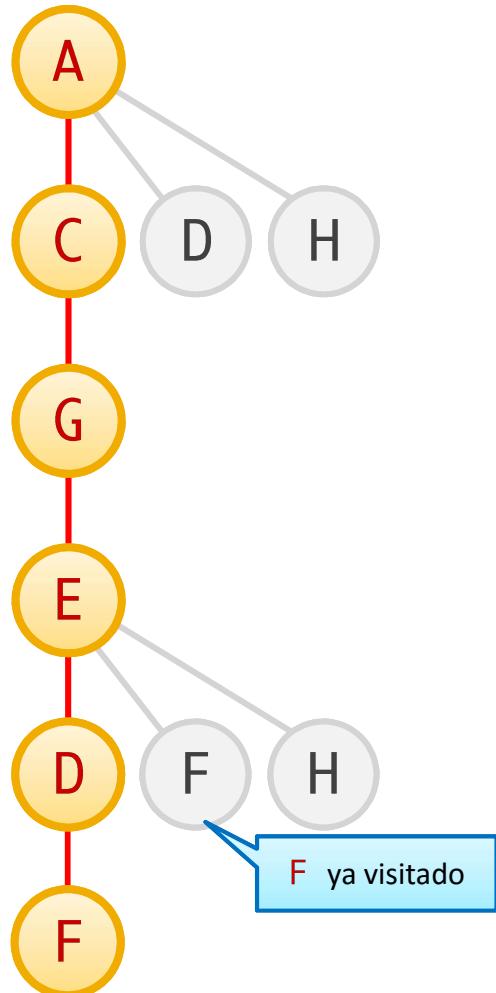
# Recorrido de un Grafo en Profundidad (II<sub>10</sub>)

- DFT comenzando en A



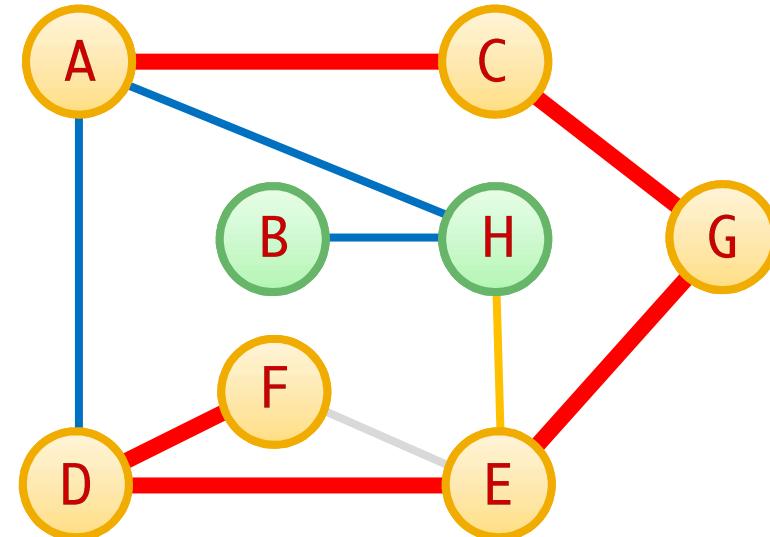
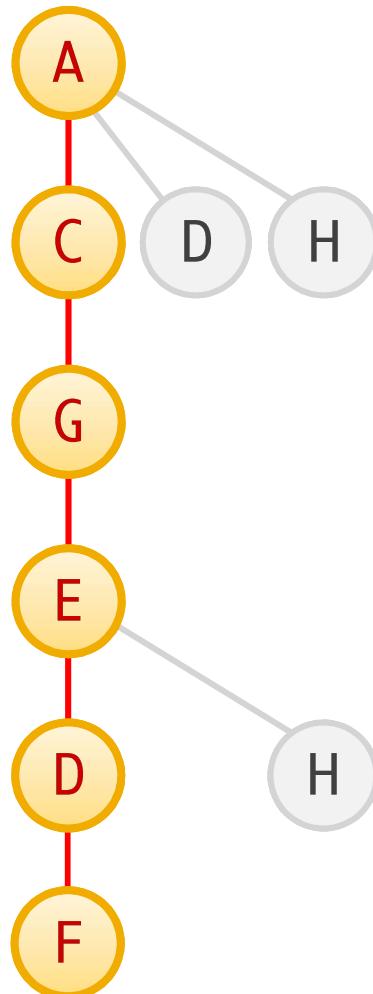
# Recorrido de un Grafo en Profundidad (II<sub>11</sub>)

- DFT comenzando en A



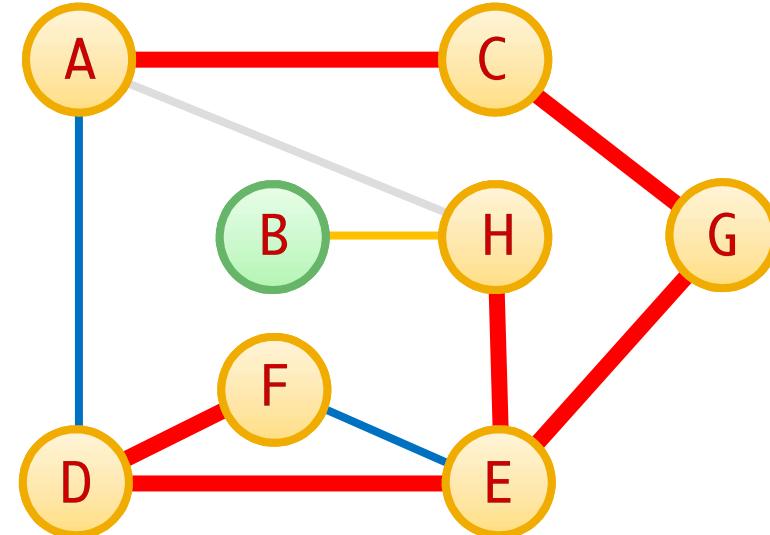
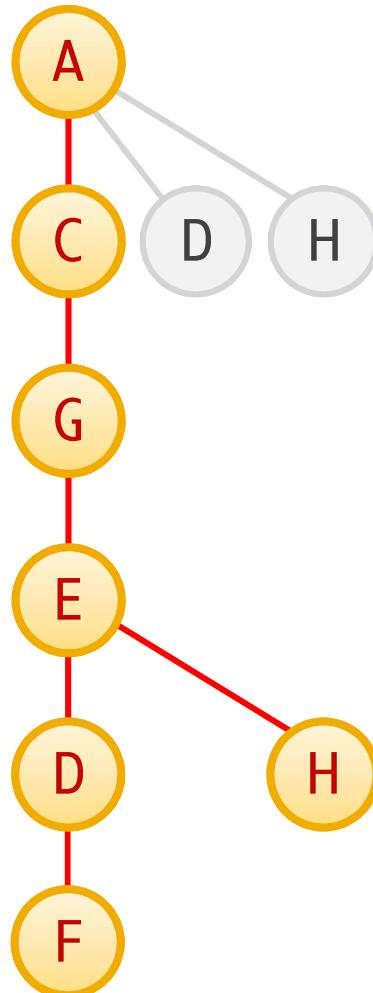
# Recorrido de un Grafo en Profundidad (II<sub>12</sub>)

- DFT comenzando en A



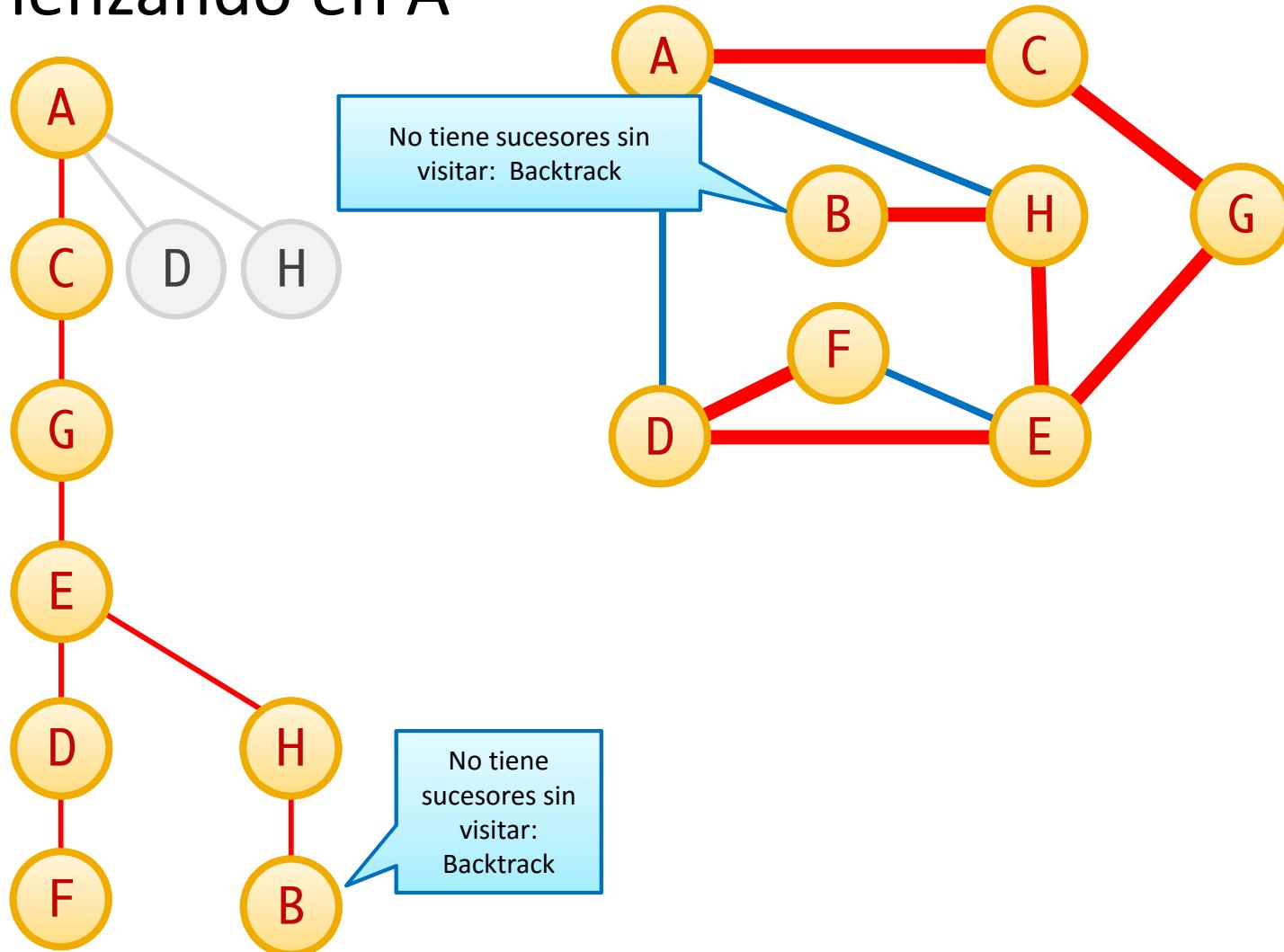
# Recorrido de un Grafo en Profundidad (II<sub>13</sub>)

- DFT comenzando en A



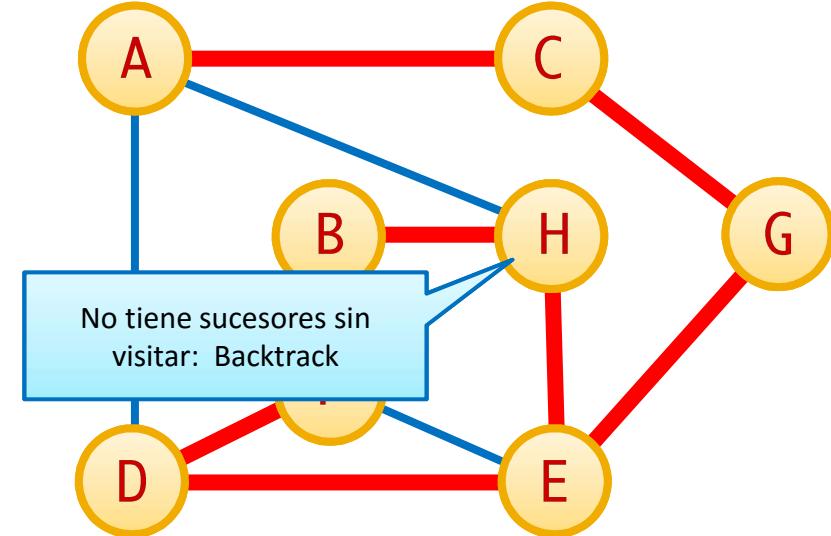
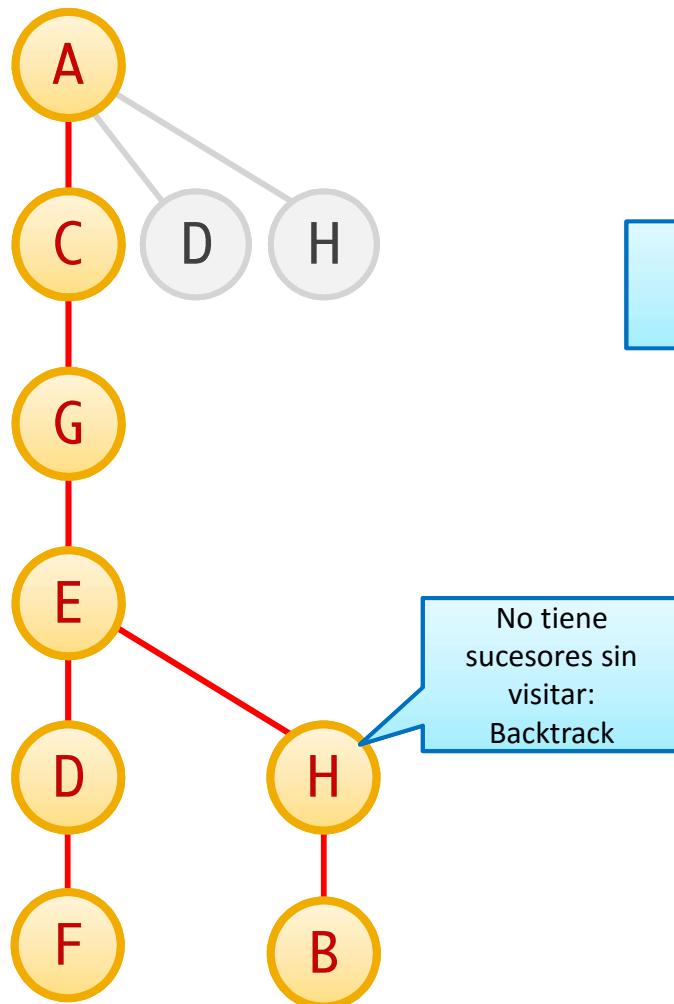
# Recorrido de un Grafo en Profundidad (II<sub>14</sub>)

- DFT comenzando en A



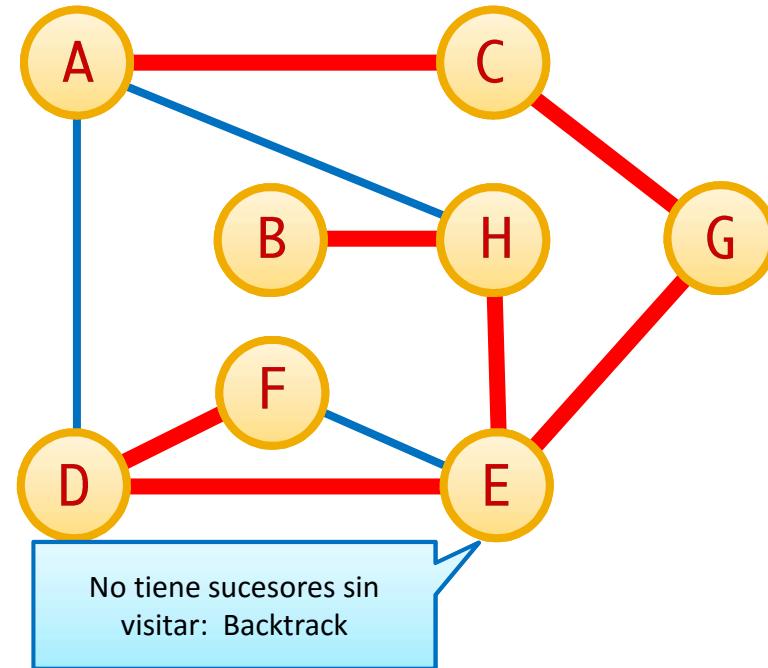
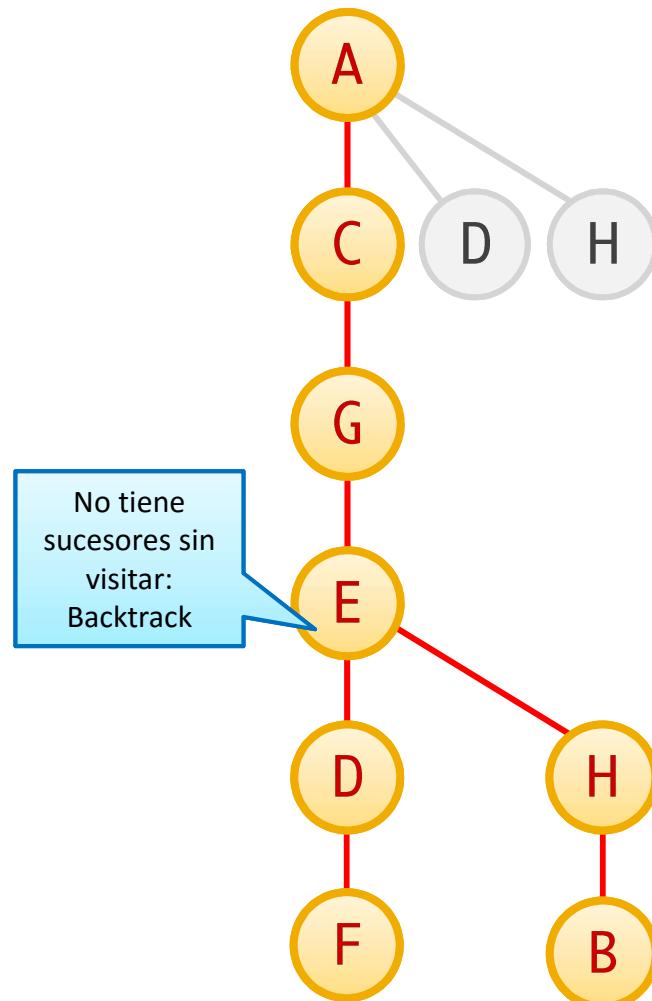
# Recorrido de un Grafo en Profundidad (II<sub>15</sub>)

- DFT comenzando en A



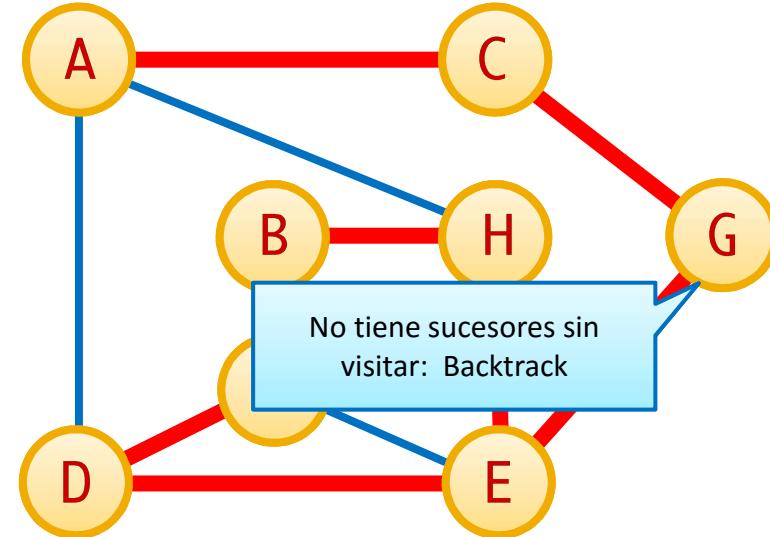
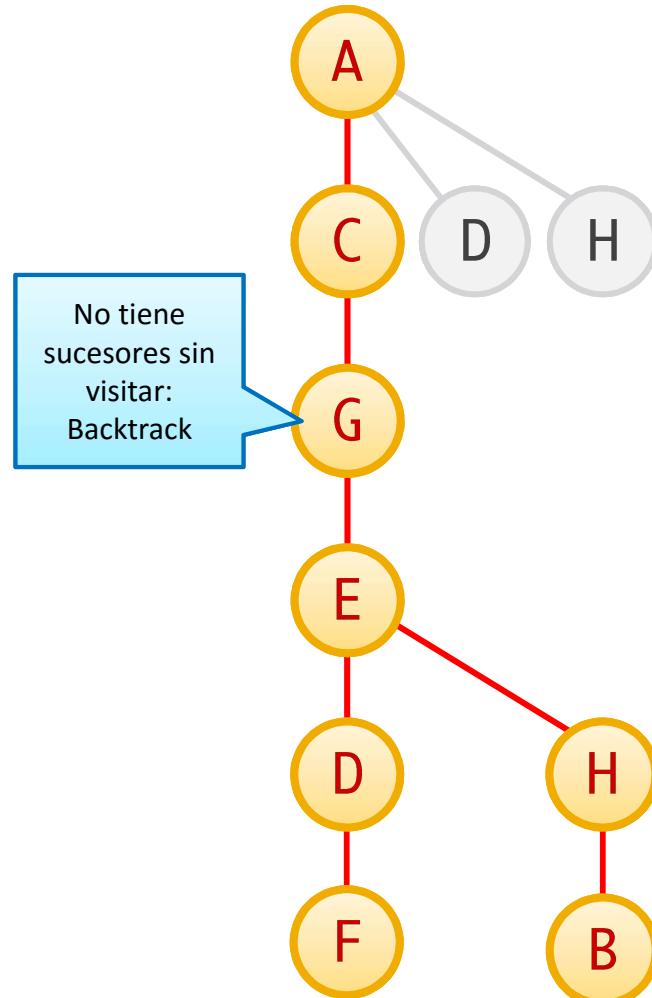
# Recorrido de un Grafo en Profundidad (II<sub>16</sub>)

- ## ■ DFT comenzando en A



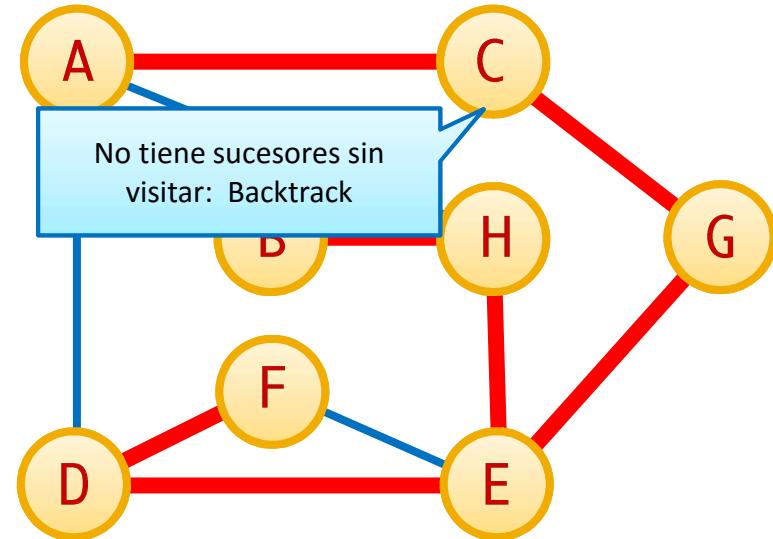
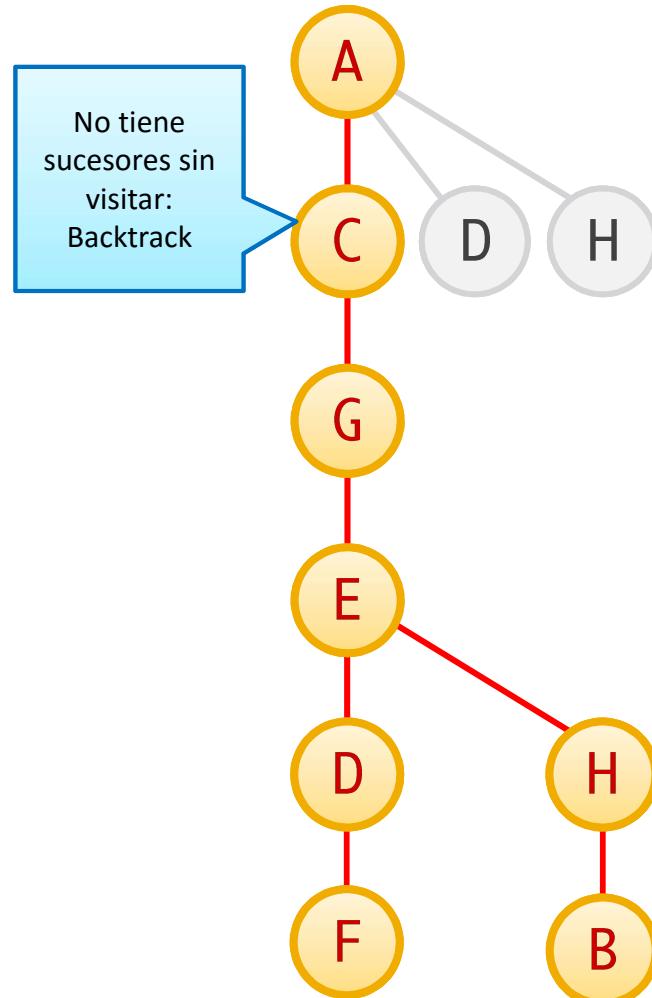
# Recorrido de un Grafo en Profundidad (II<sub>17</sub>)

- DFT comenzando en A



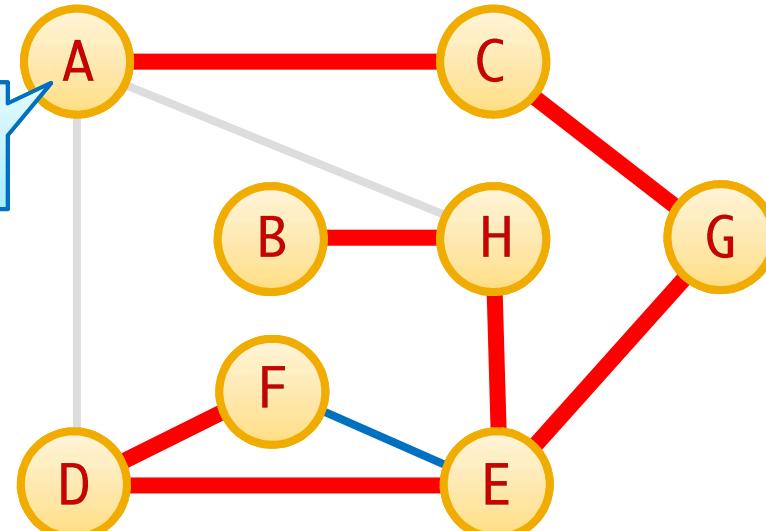
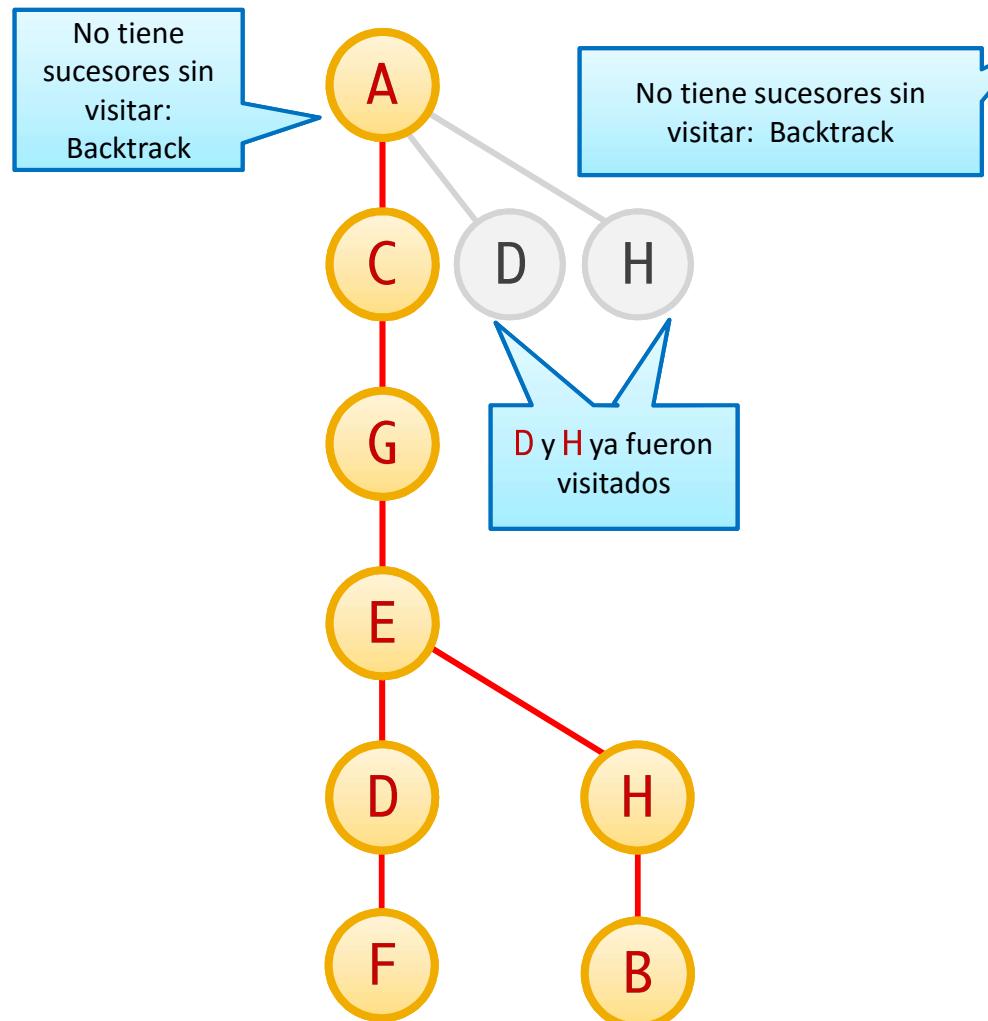
# Recorrido de un Grafo en Profundidad (II<sub>18</sub>)

- DFT comenzando en A



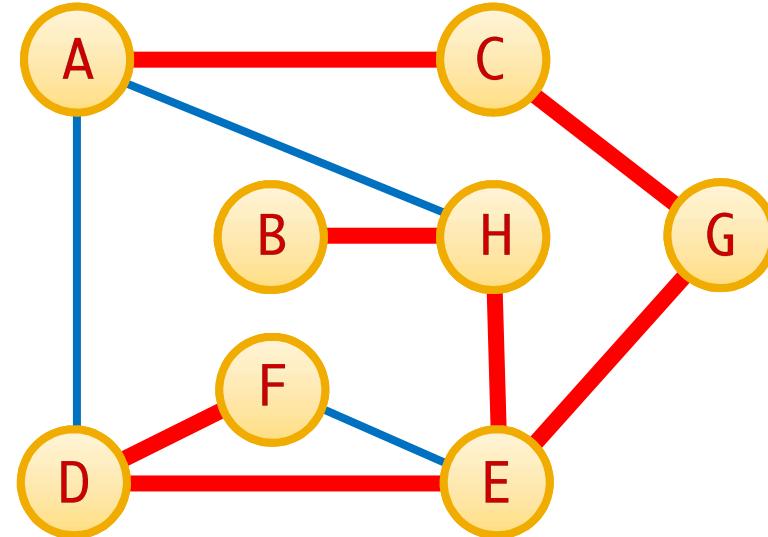
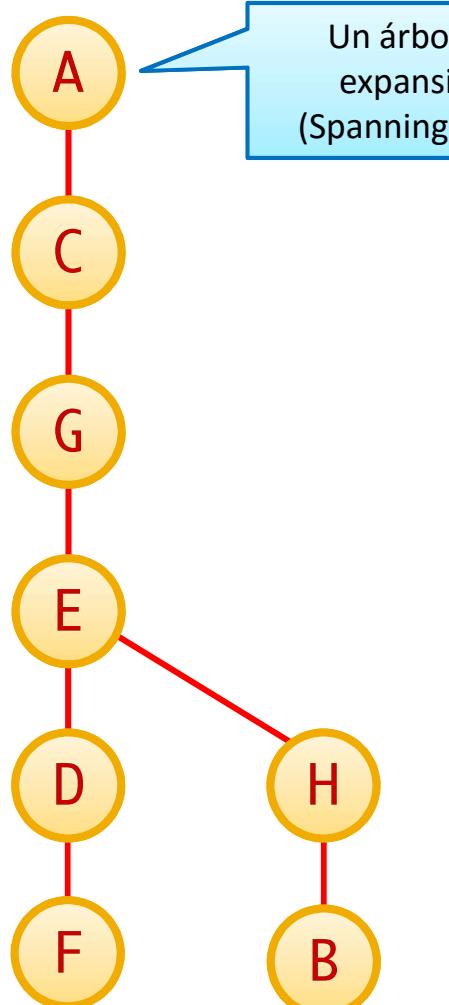
# Recorrido de un Grafo en Profundidad (II<sub>19</sub>)

- ## ■ DFT comenzando en A



# Recorrido de un Grafo en Profundidad (II<sub>20</sub>)

## ■ DFT comenzando en A



Todos los vértices han sido visitados: fin del DFT

Una secuencia de vértices en el orden de visita

Depth First Traversal: A C G E D F H B

# Recorrido en Profundidad. Propiedades

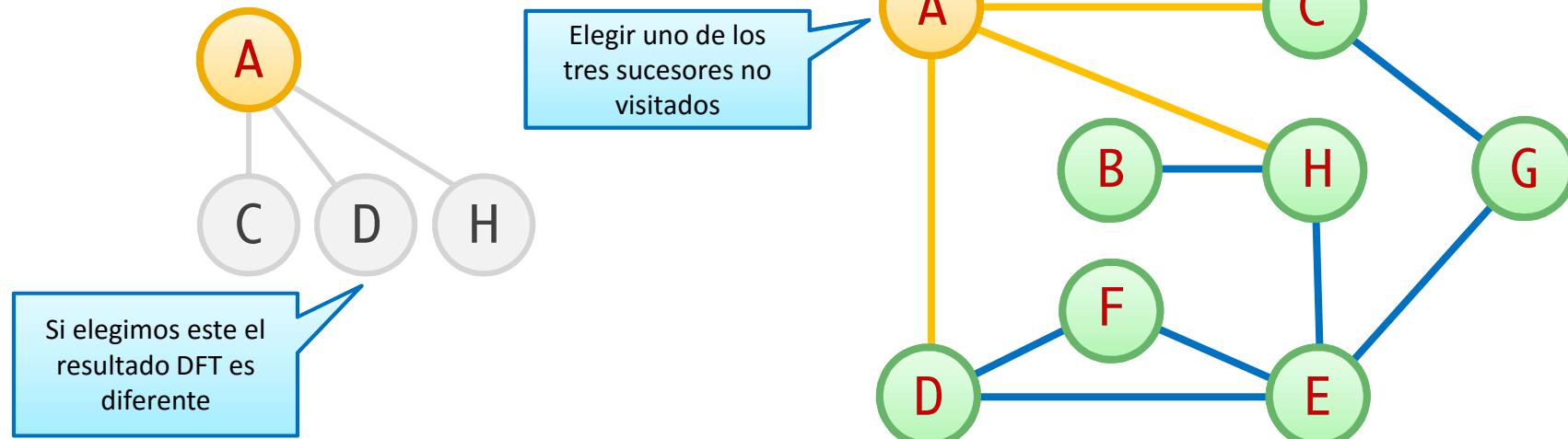
- DFT( $v$ ) visita todos los vértices de la componente conexa de  $v$  (los conectados a  $v$ )
- Las aristas utilizadas forman un **árbol de expansión en profundidad (Depth First Spanning Tree)** de la componente conexa
- DFT es invocada *a lo sumo* una sola vez para cada vértice
- Cada arista es *inspeccionada* en los dos sentidos
- La complejidad de DFT está en  $O(|V| + |E|)$  donde
  - $|V|$  es el número de vértices
  - $|E|$  es el número de aristas

## Recorrido de un Grafo en Profundidad (III)

- Un grafo puede admitir **distintos** recorridos en profundidad desde cada vértice.
- Si existe un vértice de la componente conexa de  $v$  con varios sucesores, entonces existen distintos recorridos DFT desde  $v$ .
- Los recorridos dependerán del vértice inicial y del orden de elección de sucesores.

# Recorrido de un Grafo en Profundidad (IV<sub>1</sub>)

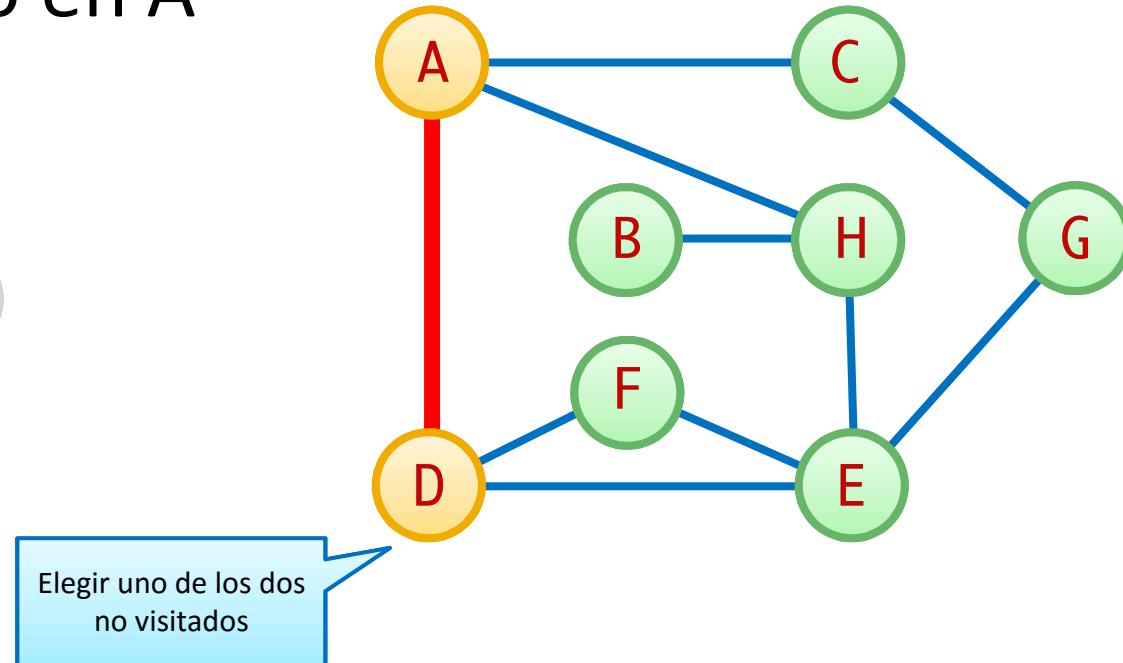
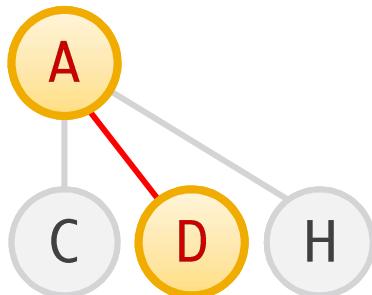
## ■ DFT comenzando en A



Depth First Traversal: A ...

# Recorrido de un Grafo en Profundidad (IV<sub>2</sub>)

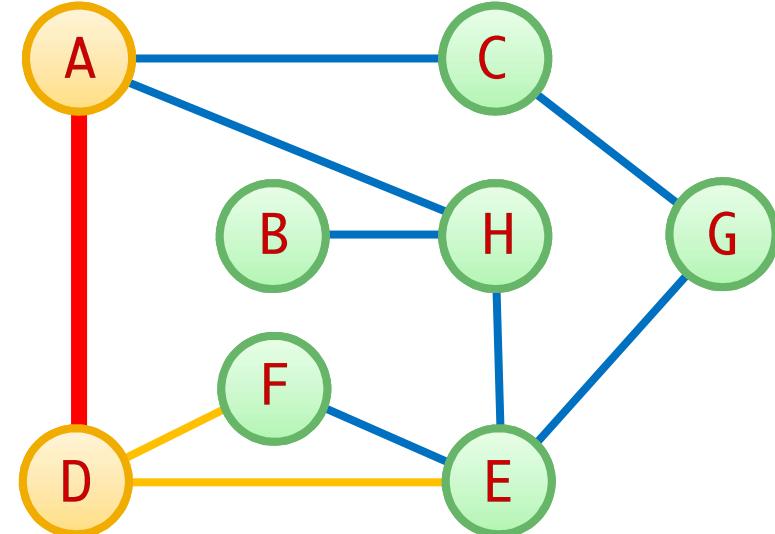
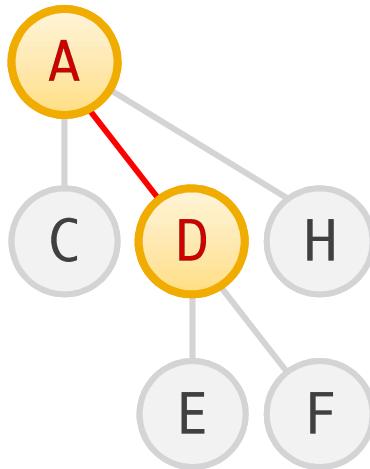
- ## ■ DFT comenzando en A



Depth First Traversal: A D ...

# Recorrido de un Grafo en Profundidad (IV<sub>3</sub>)

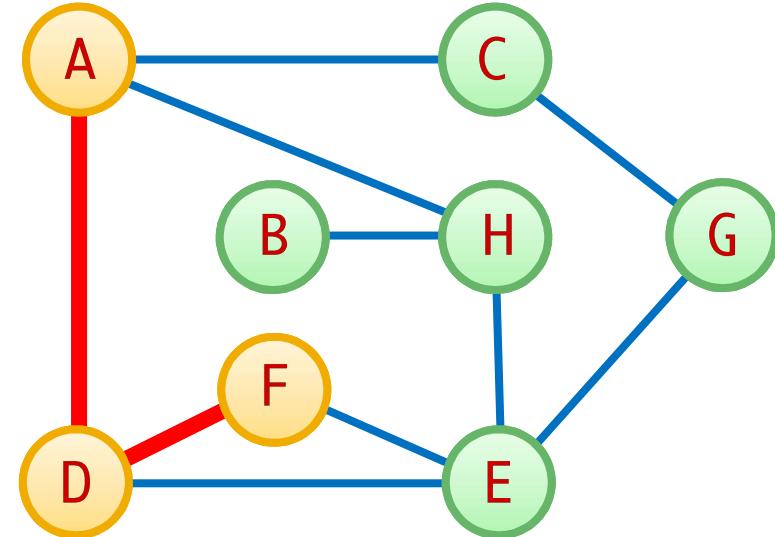
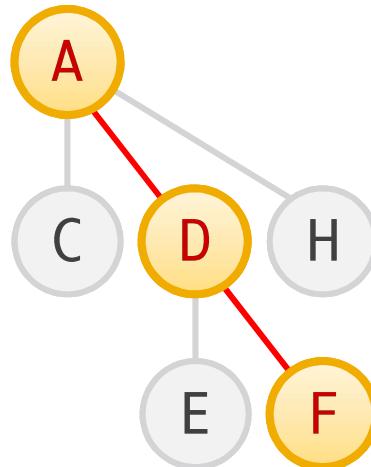
- DFT comenzando en A



Depth First Traversal: A D ...

# Recorrido de un Grafo en Profundidad (IV<sub>4</sub>)

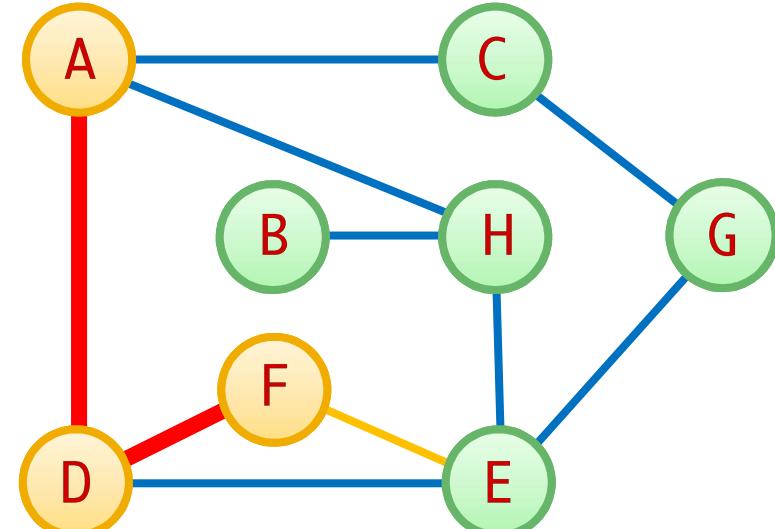
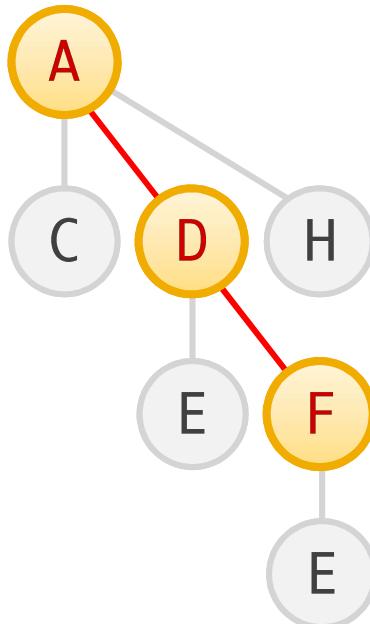
- DFT comenzando en A



Depth First Traversal: A D F ...

# Recorrido de un Grafo en Profundidad (IV<sub>5</sub>)

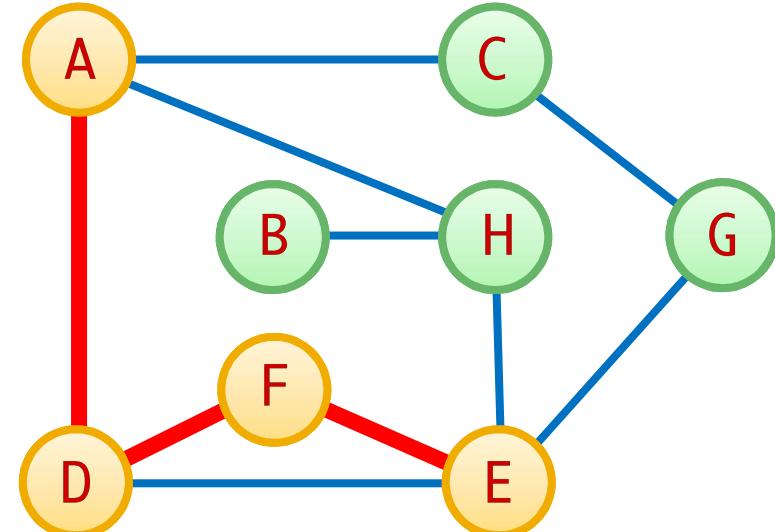
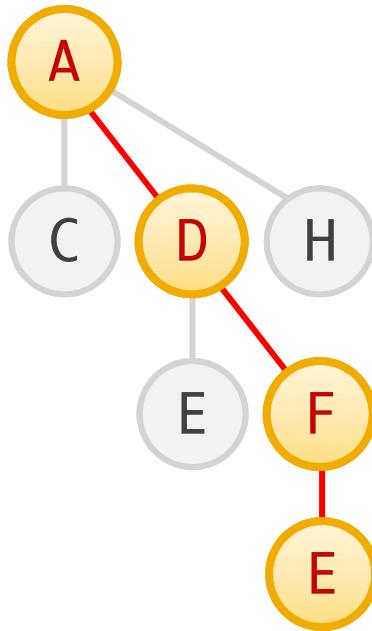
- DFT comenzando en A



Depth First Traversal: A D F ...

# Recorrido de un Grafo en Profundidad (IV<sub>6</sub>)

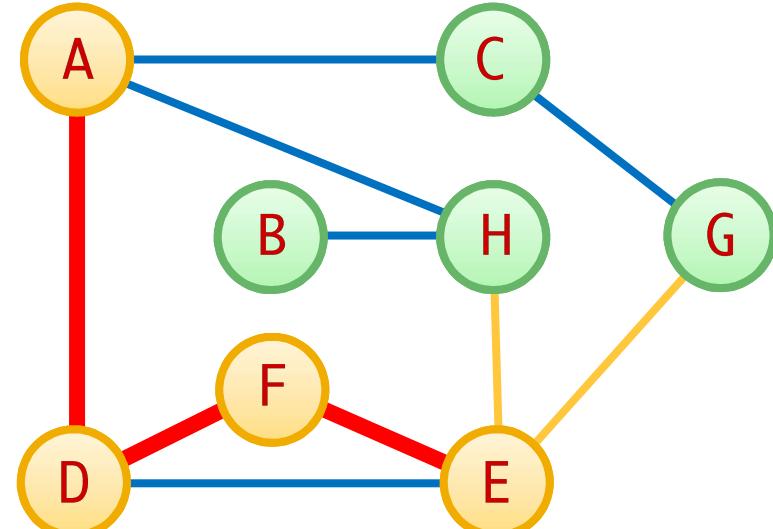
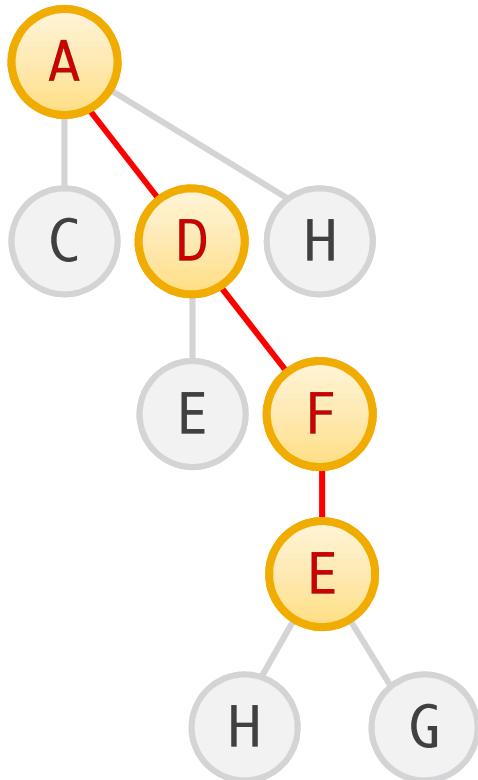
- DFT comenzando en A



Depth First Traversal: A D F E ...

# Recorrido de un Grafo en Profundidad (IV<sub>7</sub>)

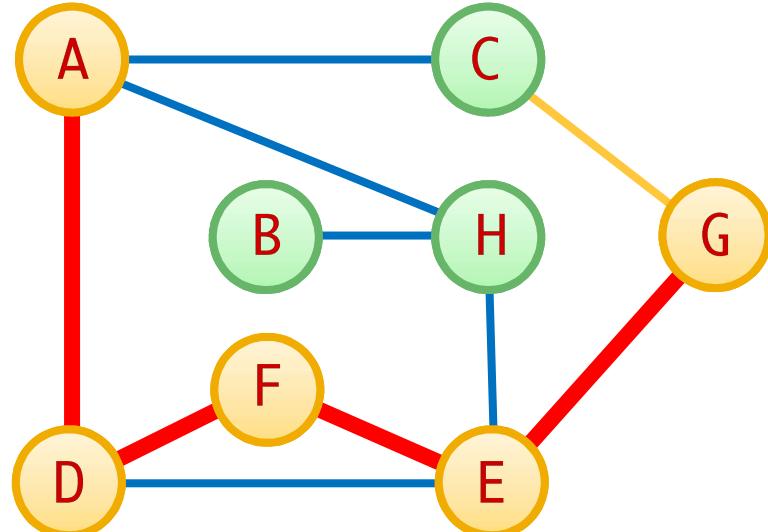
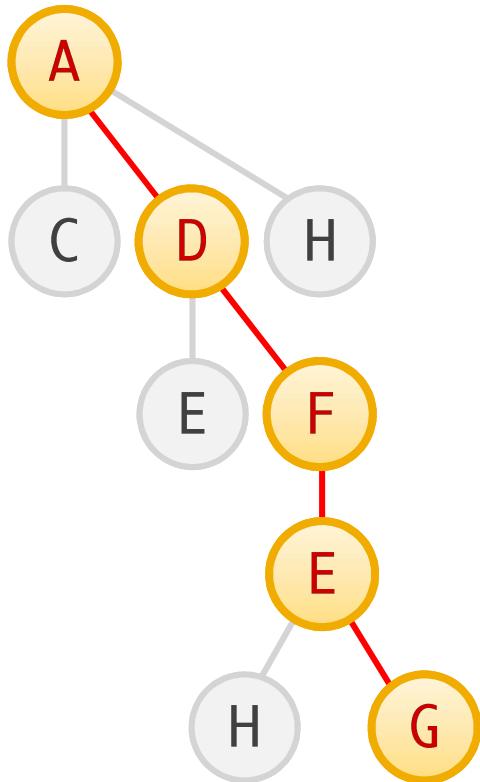
- DFT comenzando en A



Depth First Traversal: A D F E ...

# Recorrido de un Grafo en Profundidad (IV<sub>8</sub>)

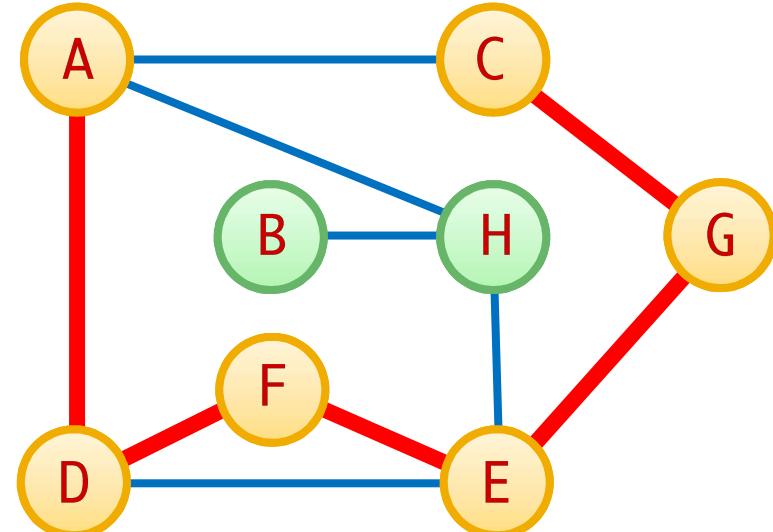
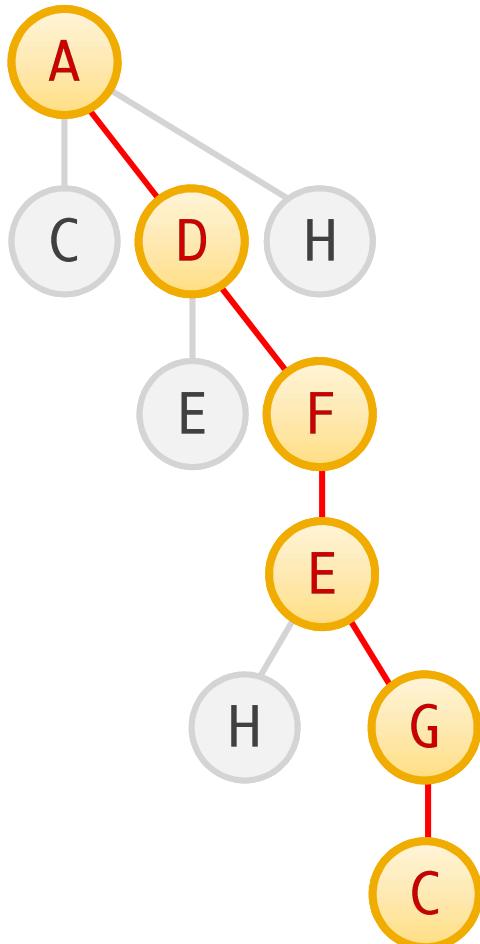
- DFT comenzando en A



Depth First Traversal: A D F E G ...

# Recorrido de un Grafo en Profundidad (IV<sub>9</sub>)

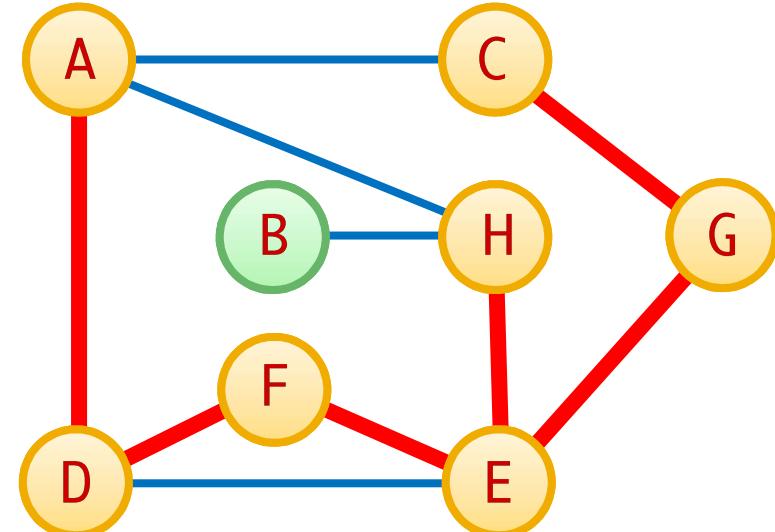
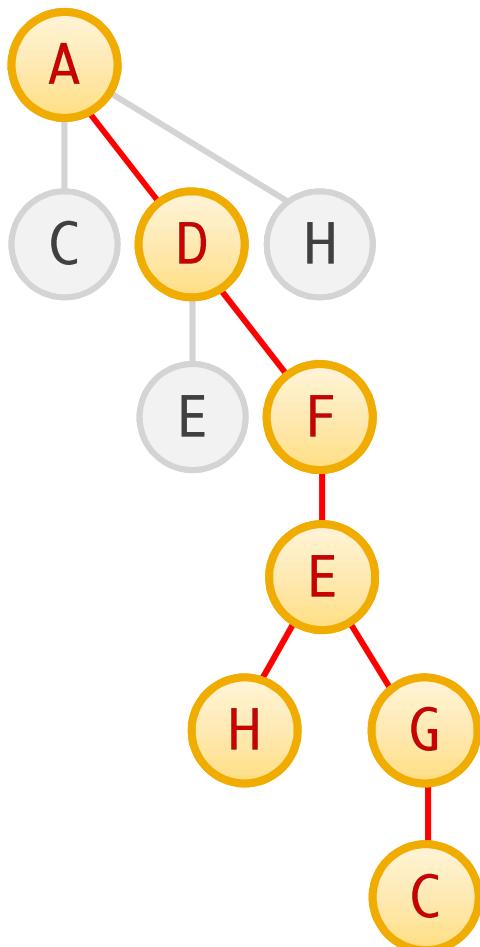
- DFT comenzando en A



Depth First Traversal: A D F E G C ...

# Recorrido de un Grafo en Profundidad (IV<sub>10</sub>)

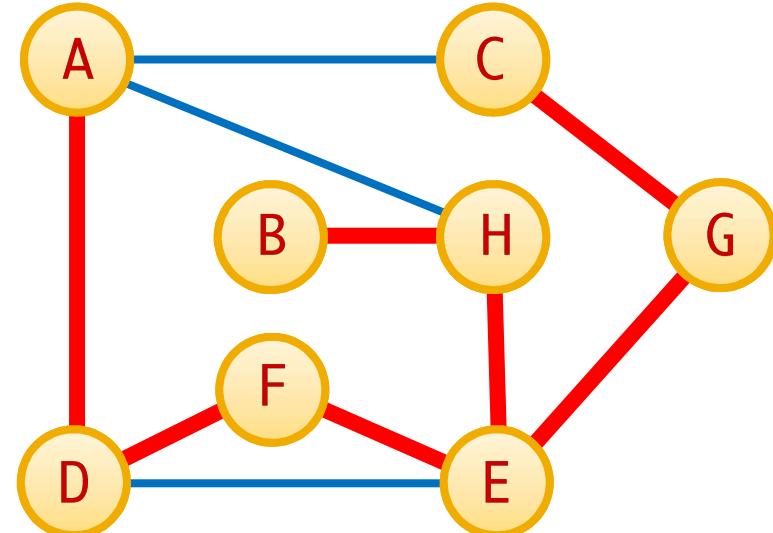
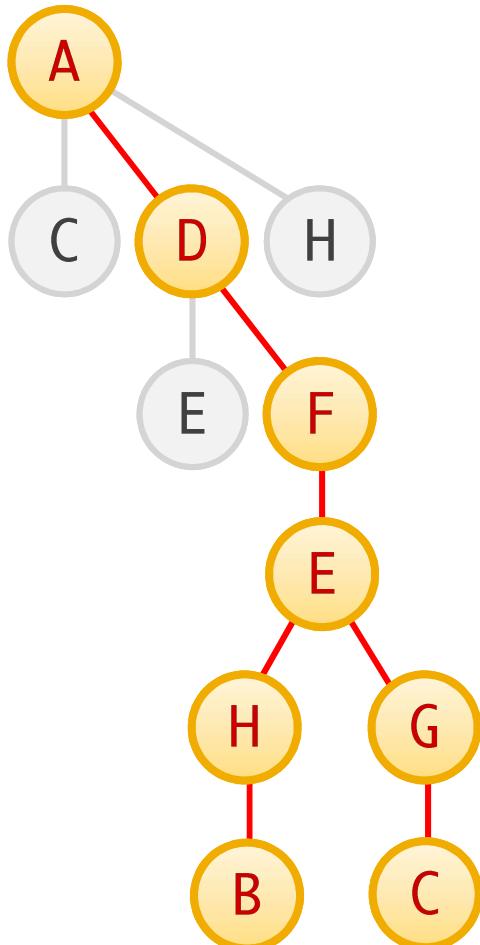
- DFT comenzando en A



Depth First Traversal: A D F E G C H ...

# Recorrido de un Grafo en Profundidad (IV<sub>11</sub>)

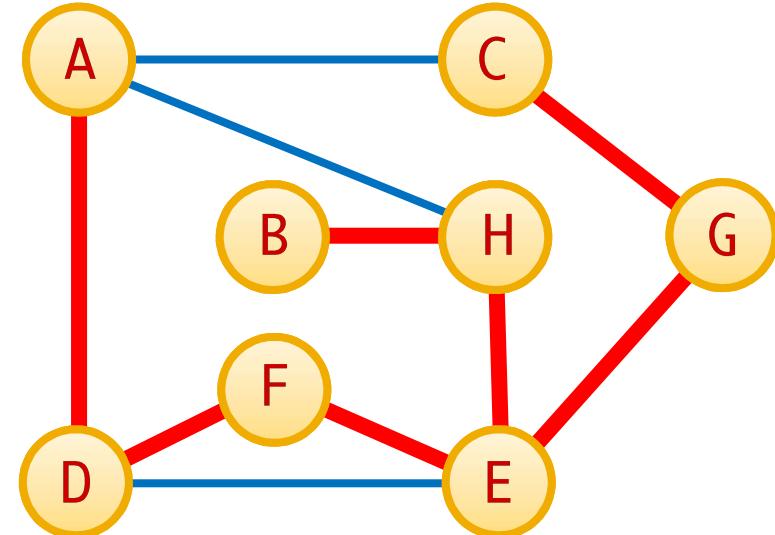
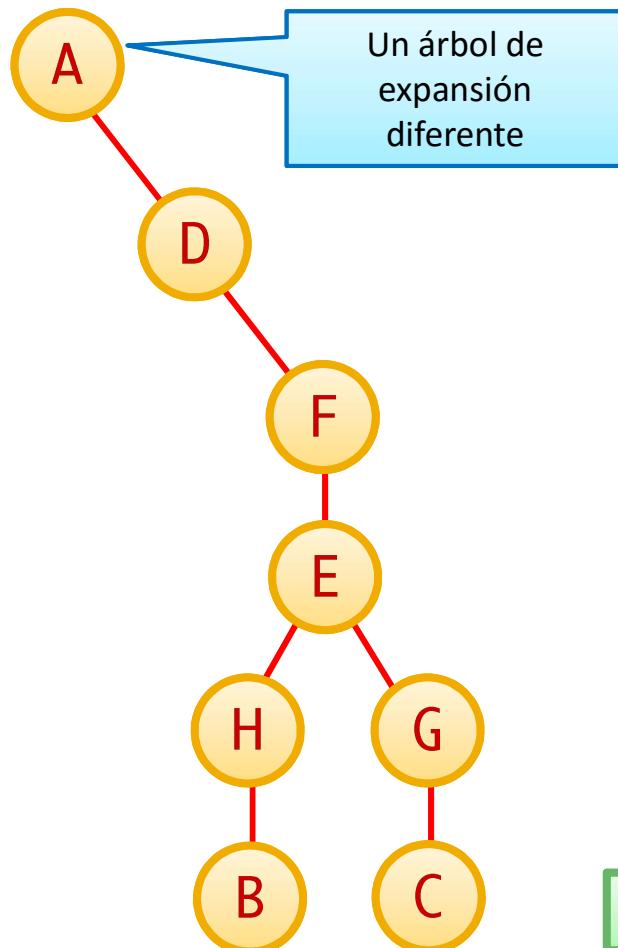
- DFT comenzando en A



Depth First Traversal: A D F E G C H B

# Recorrido de un Grafo en Profundidad (IV<sub>12</sub>)

- DFT comenzando en A



Otro recorrido DFT

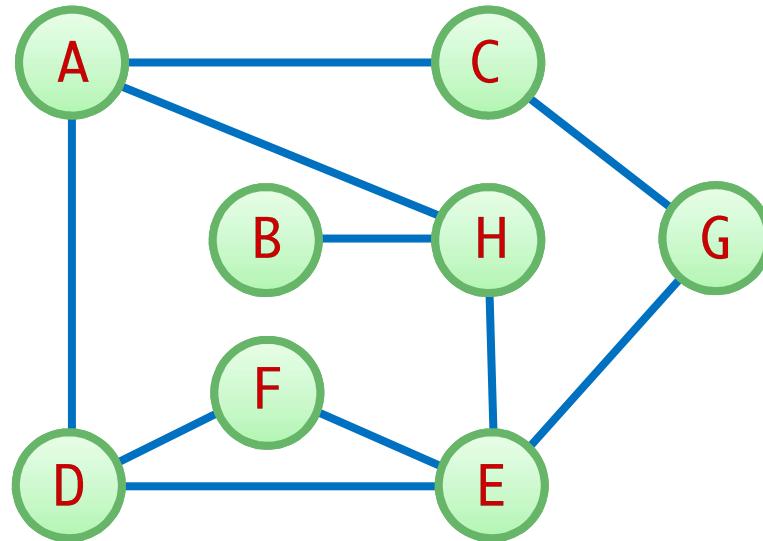
Depth First Traversal: A D F E G C H B

# DFT: Implementación con un Stack

- El recorrido en profundidad puede implementarse a través de una Pila (Stack)  $S$  que *recuerda* los vértices pendientes de explorar
  - Al principio, colocamos el vértice inicial en un stack vacío  $S$
  - Mientras  $S$  no sea vacío:
    - Extraer (Pop) el elemento  $v$  de la cima de  $S$
    - Si  $v$  no ha sido visitado:
      - Visitarlo (colecciónarlo, guardarlo, ... )
      - Apilar en  $S$  los sucesores de  $v$  que no han sido visitados

# DFT: Implementación con un Stack (II<sub>1</sub>)

## ■ DFT comenzando en A



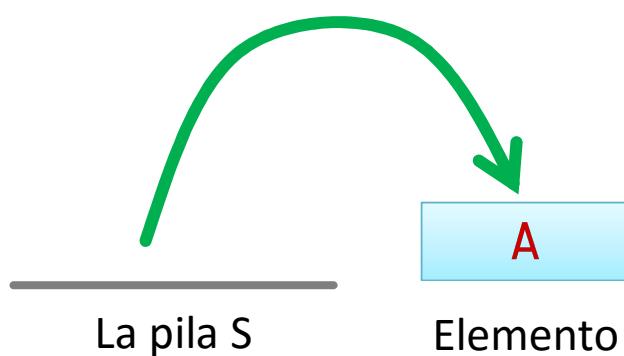
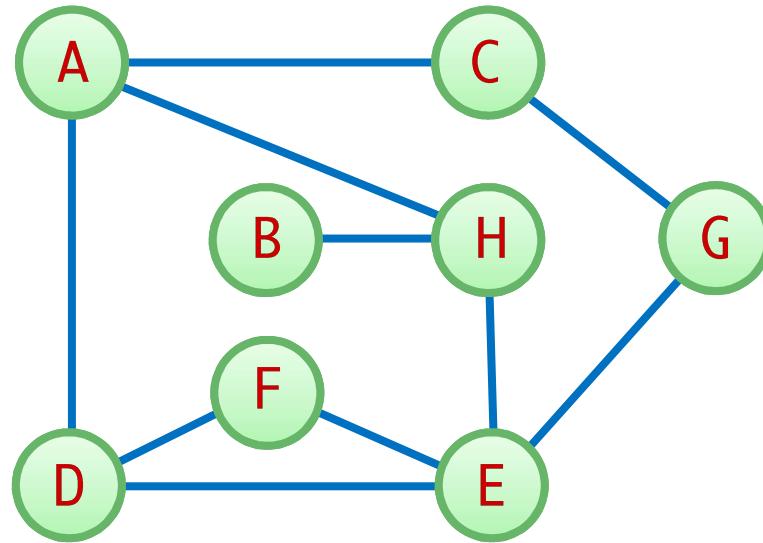
A

La pila S

- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>2</sub>)

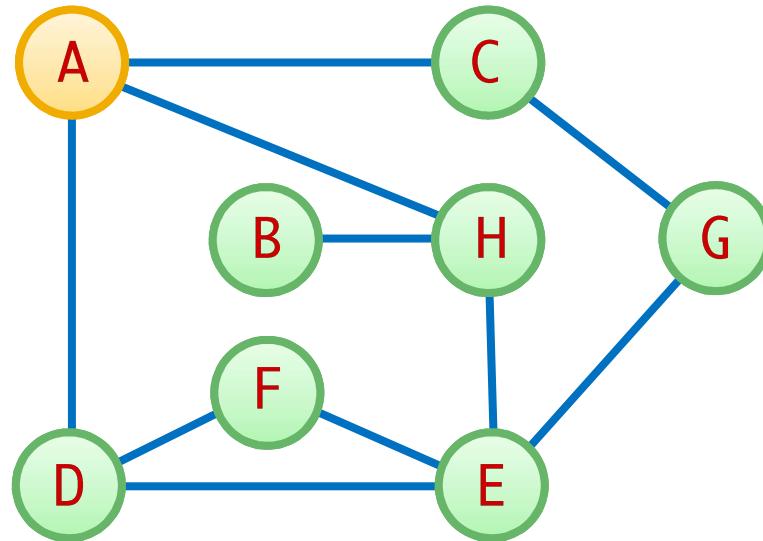
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>3</sub>)

## ■ DFT comenzando en A



La pila S

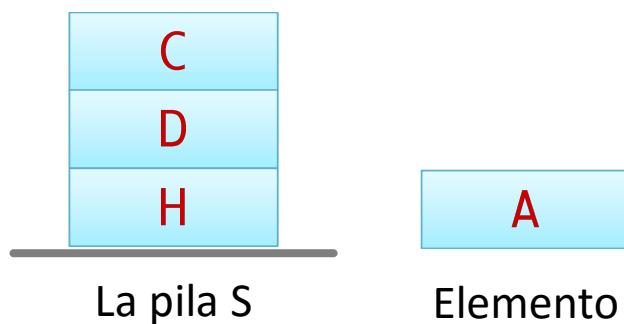
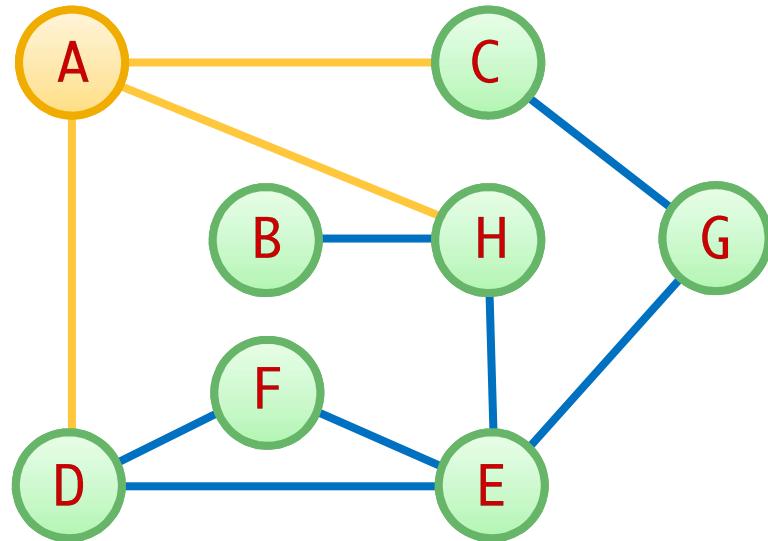
---

Elemento

- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack ( $\Pi_4$ )

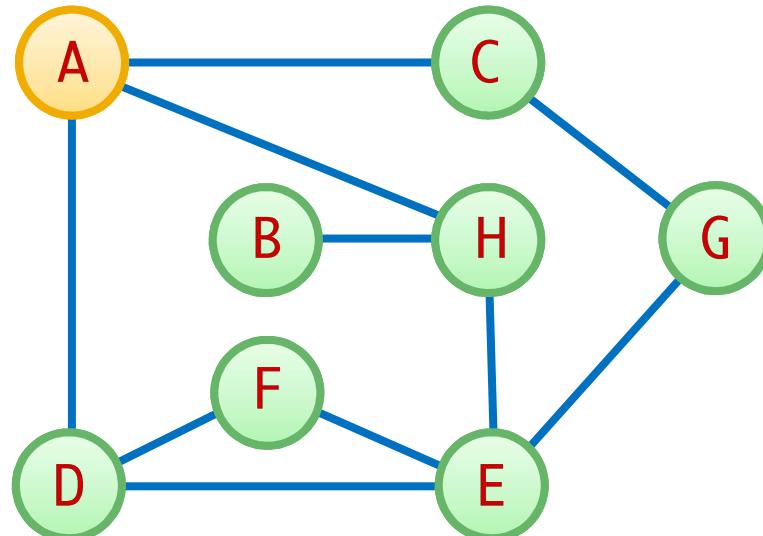
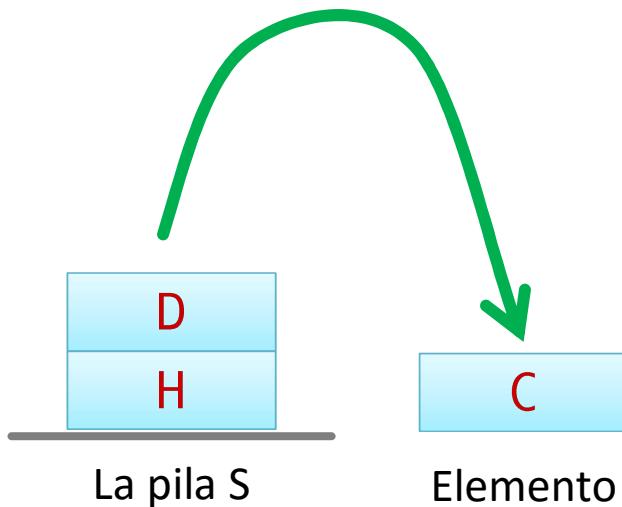
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>5</sub>)

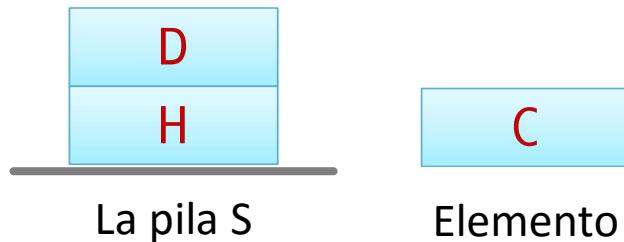
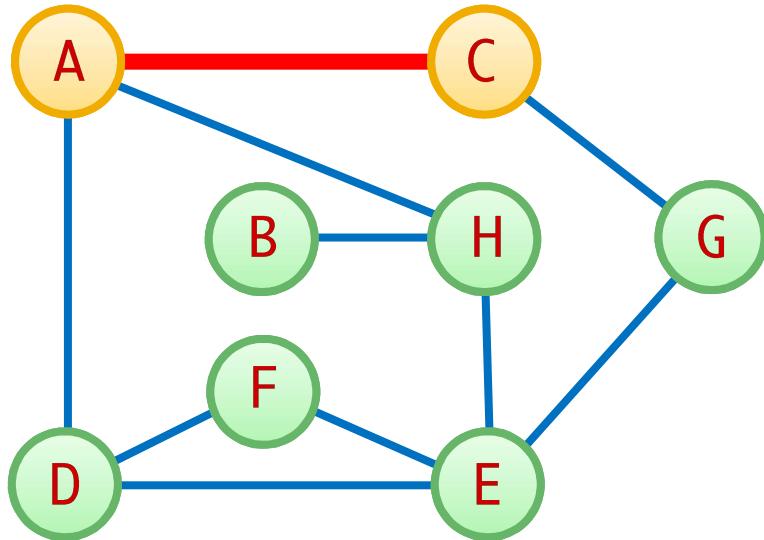
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>6</sub>)

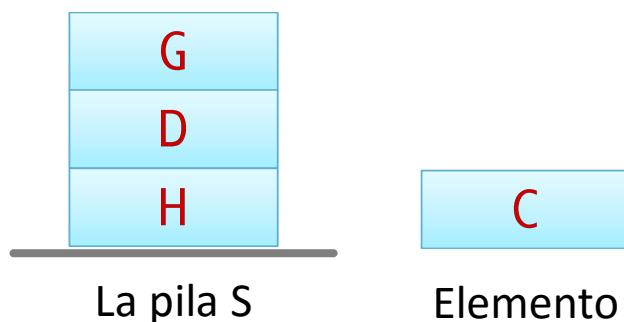
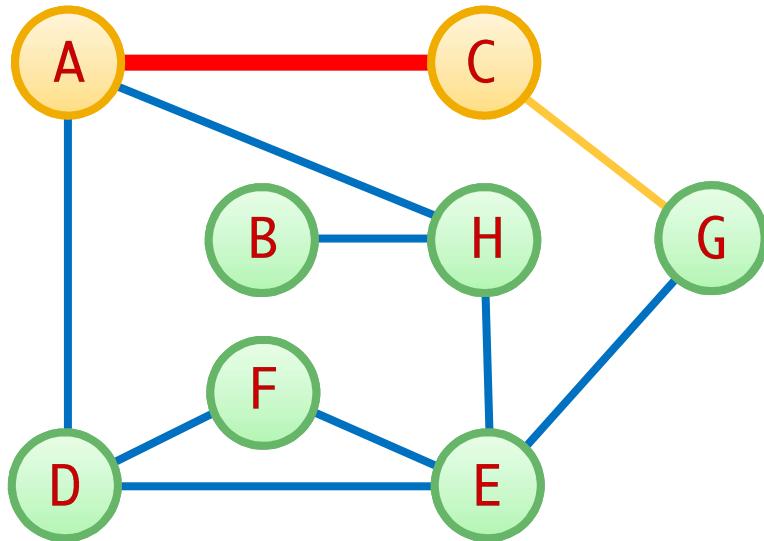
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
  - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>7</sub>)

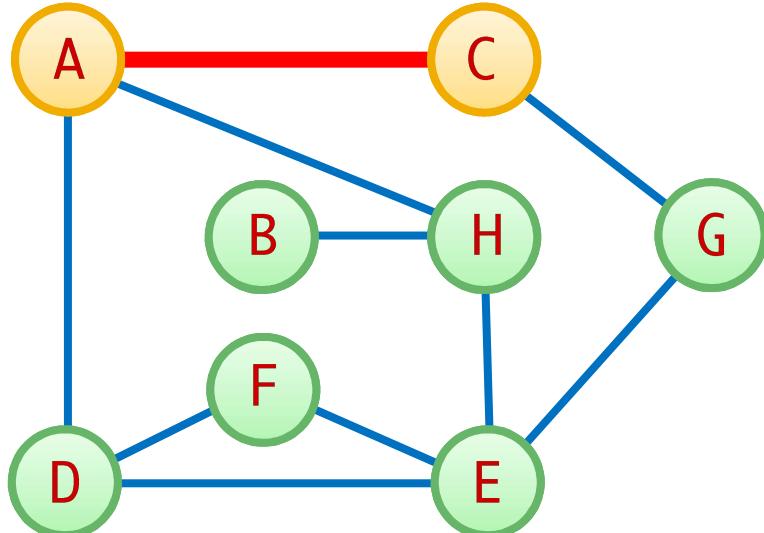
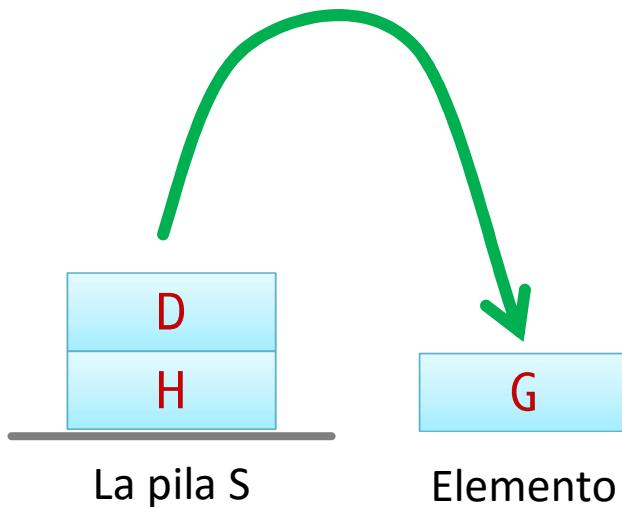
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>8</sub>)

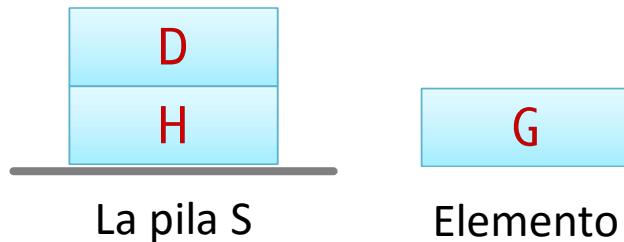
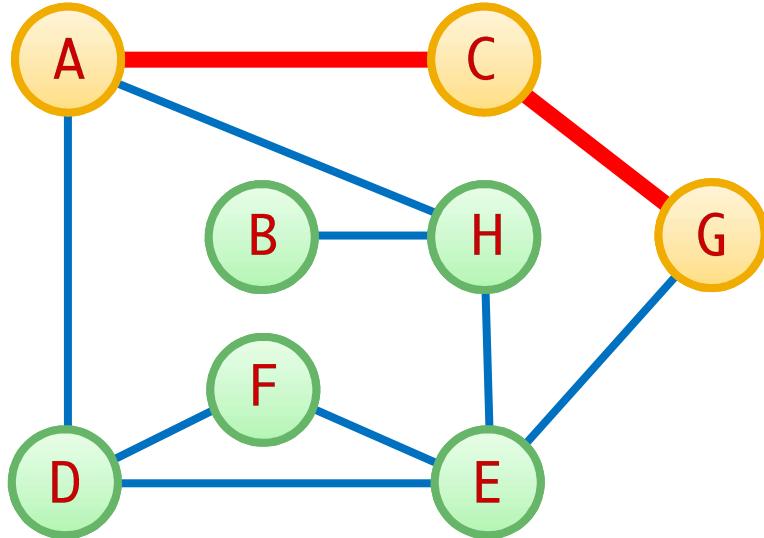
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>9</sub>)

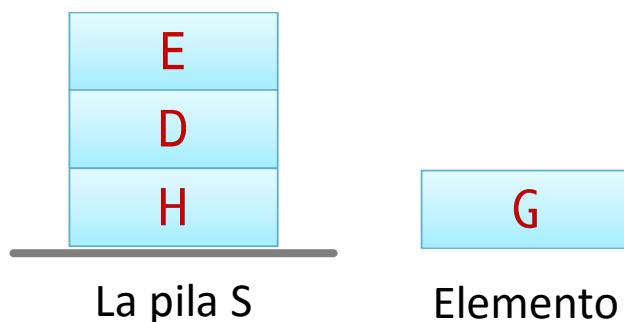
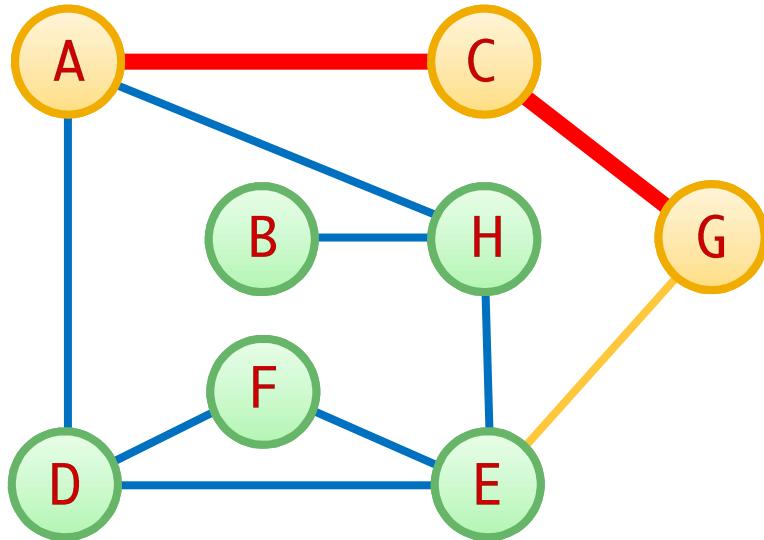
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
  - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>10</sub>)

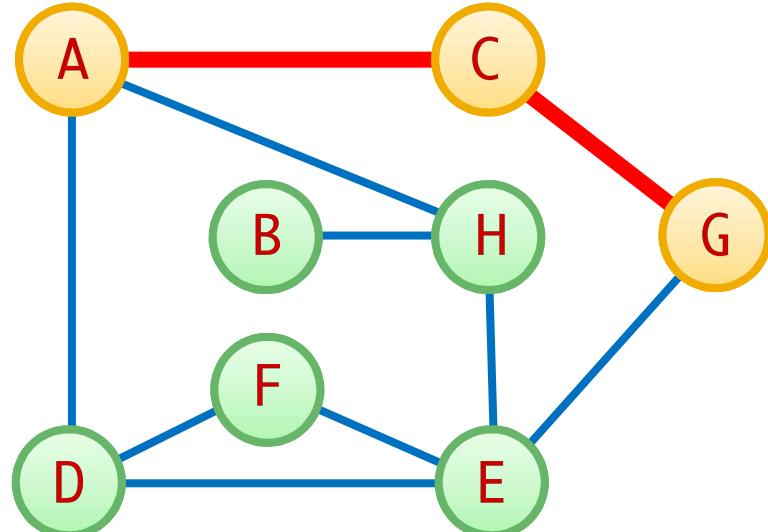
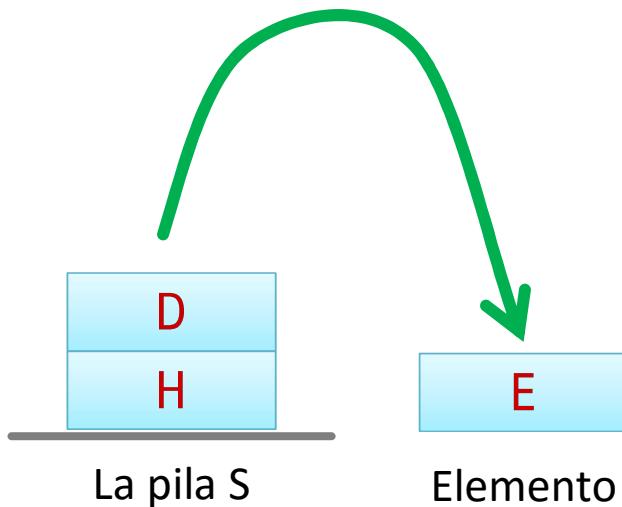
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>11</sub>)

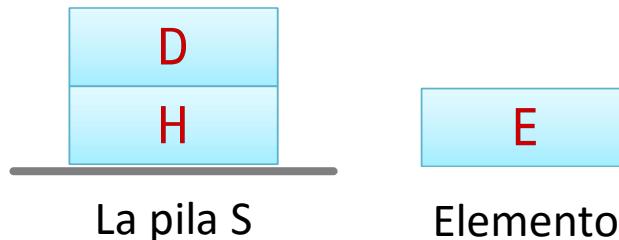
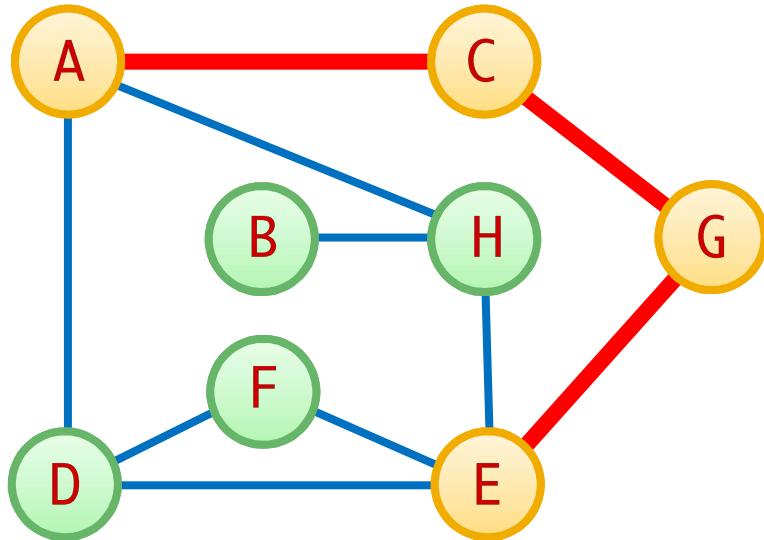
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>12</sub>)

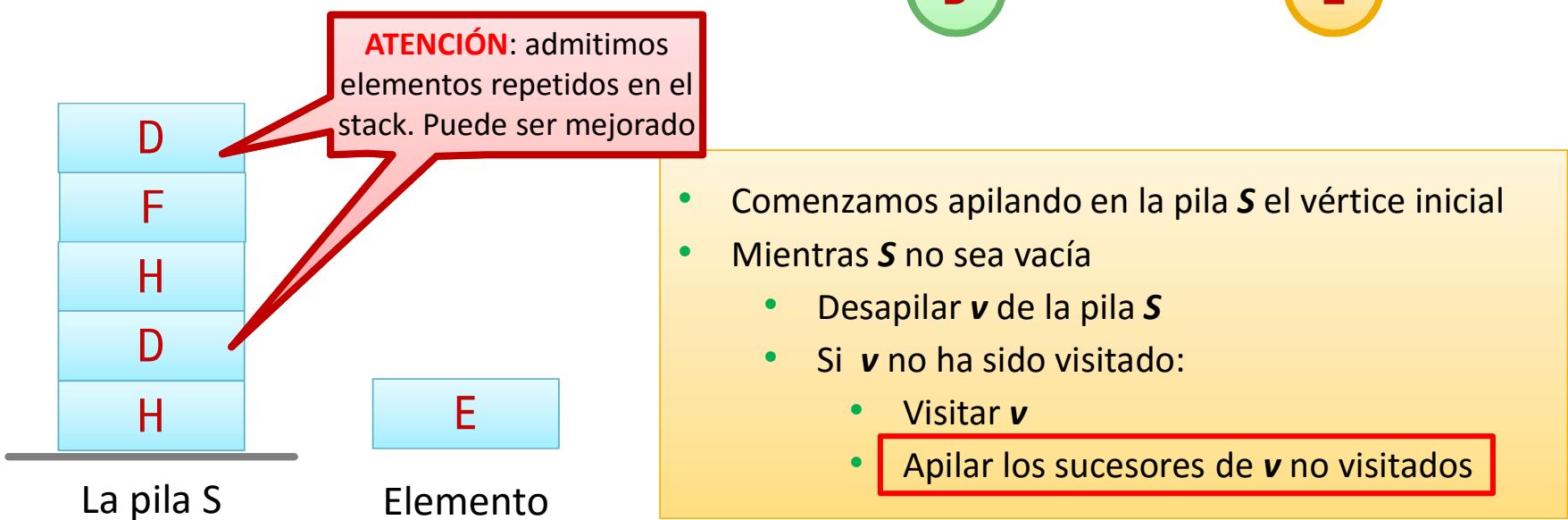
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

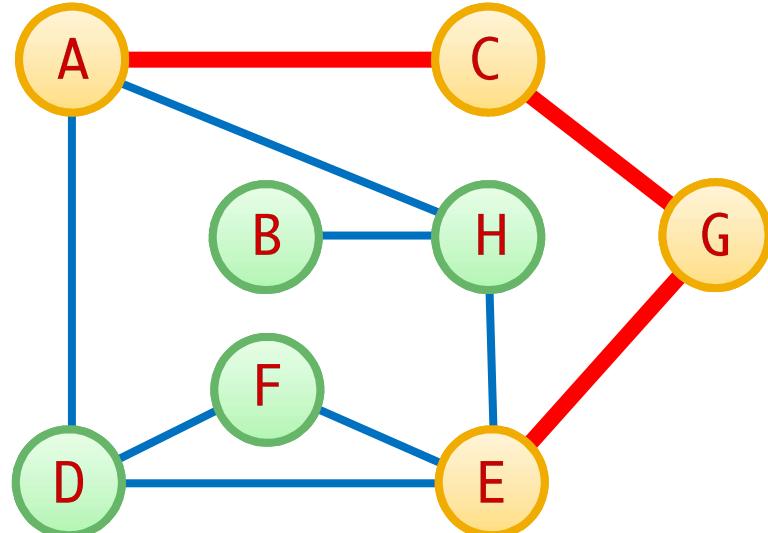
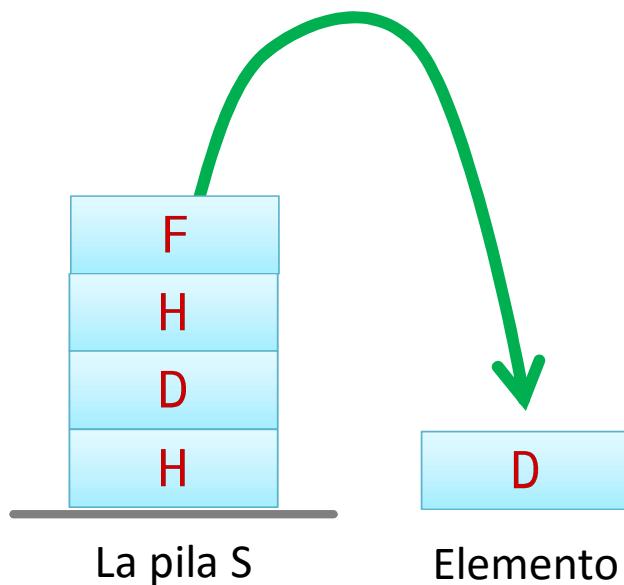
# DFT: Implementación con un Stack (II<sub>13</sub>)

## ■ DFT comenzando en A



# DFT: Implementación con un Stack (II<sub>14</sub>)

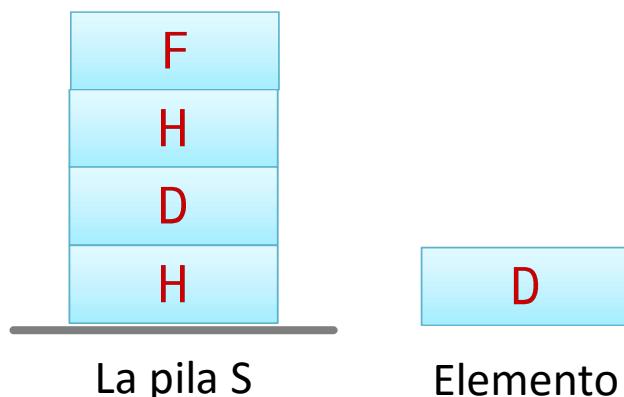
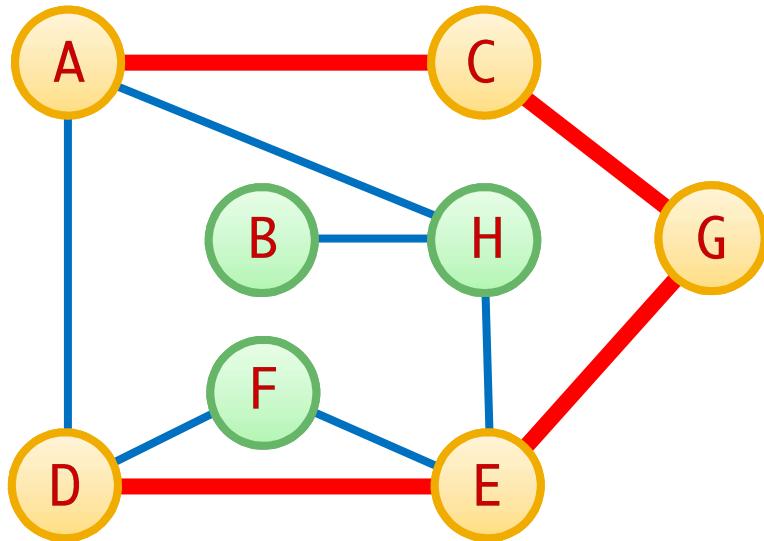
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>15</sub>)

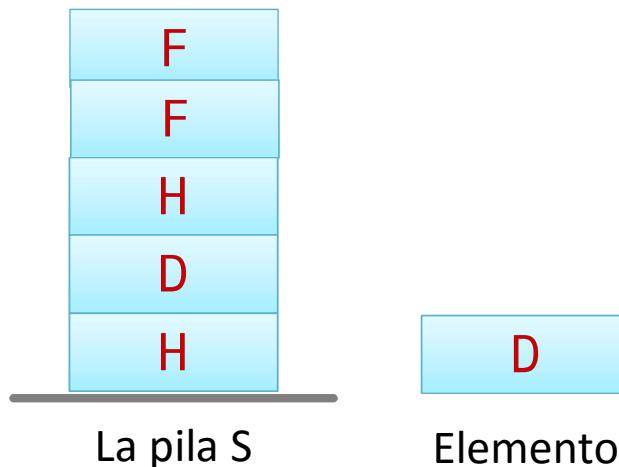
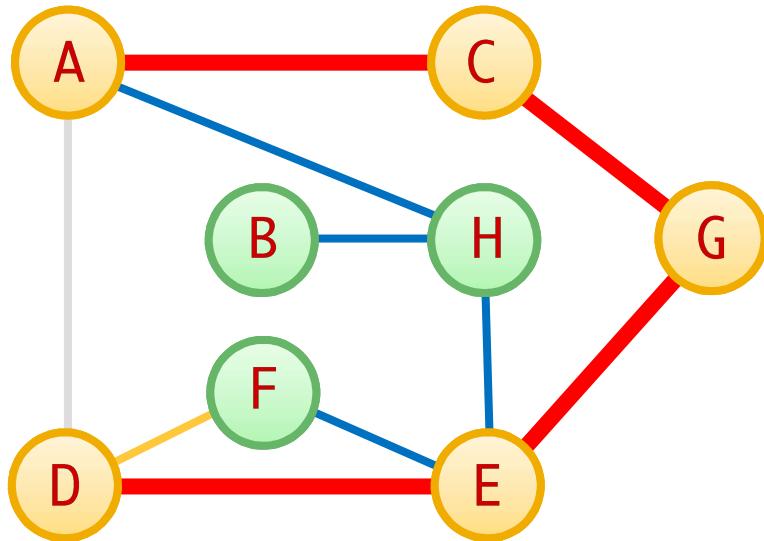
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
  - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>16</sub>)

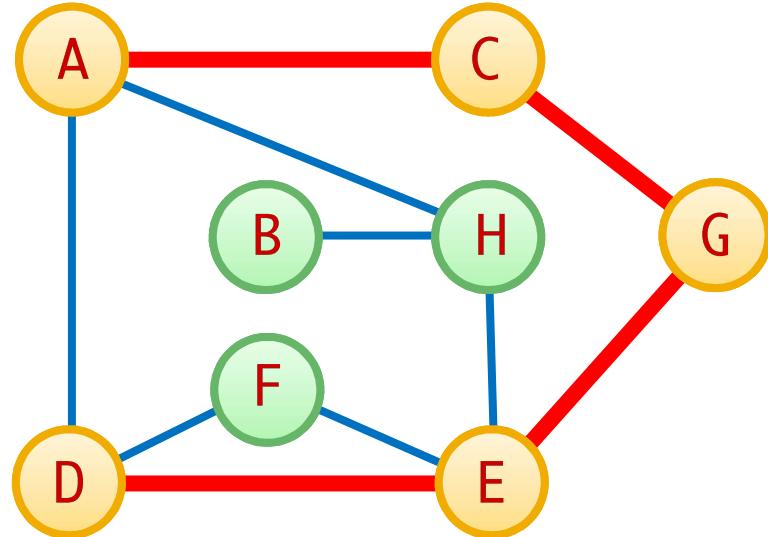
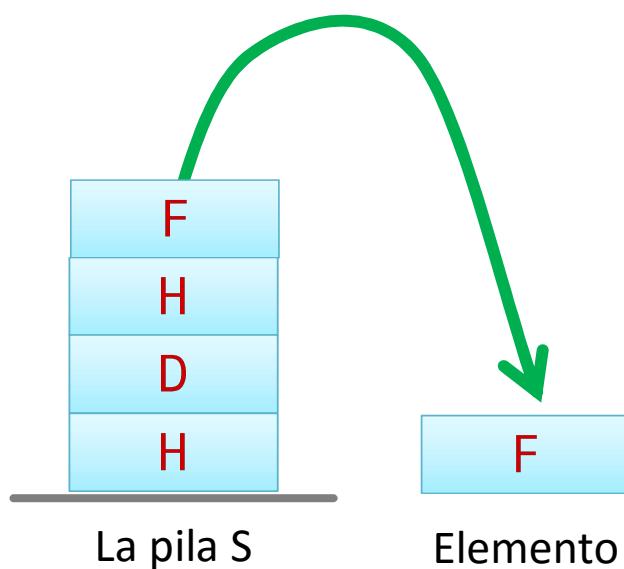
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>17</sub>)

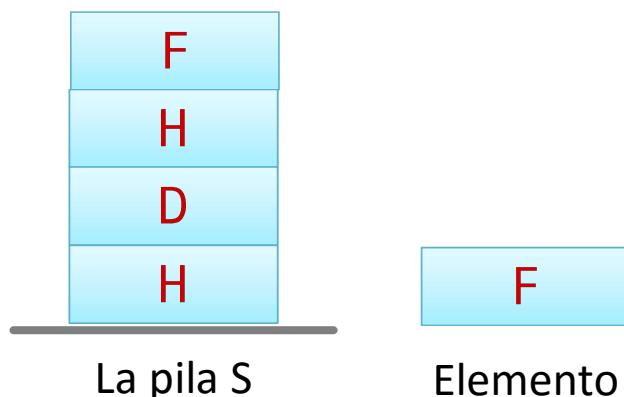
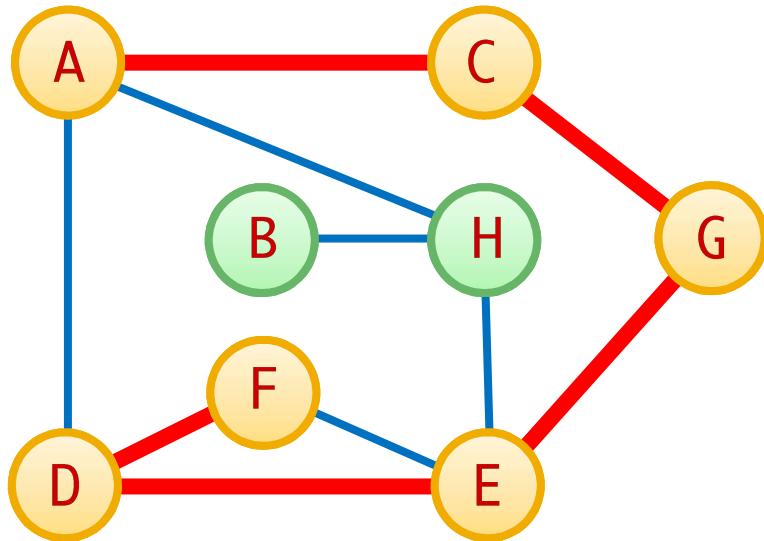
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>18</sub>)

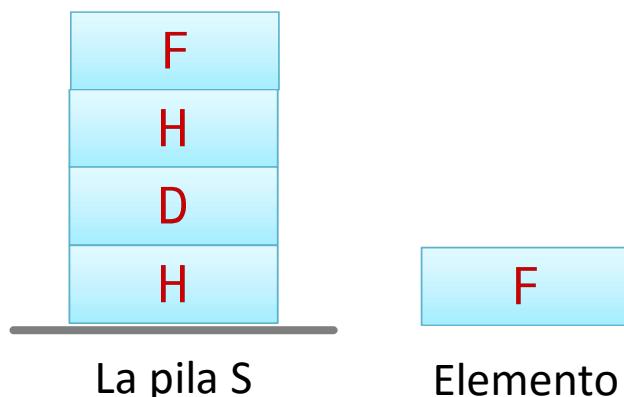
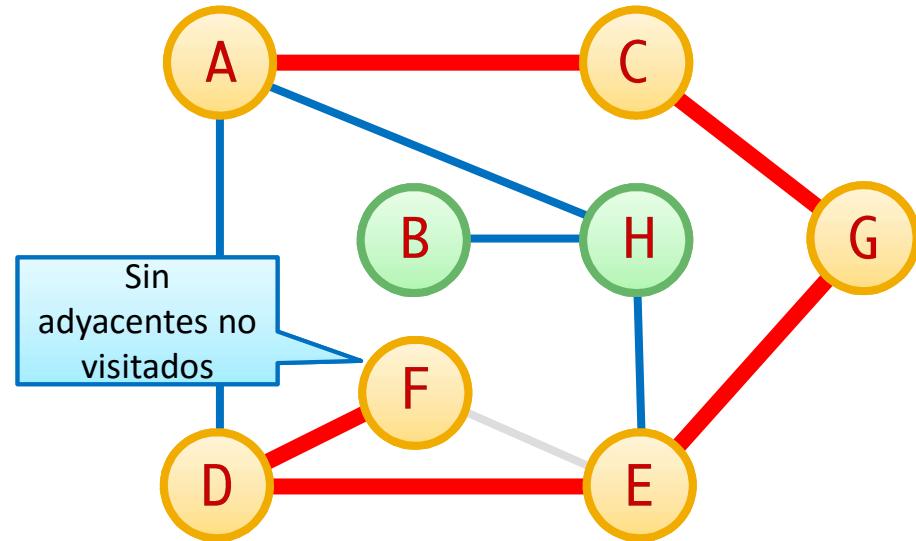
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
  - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>19</sub>)

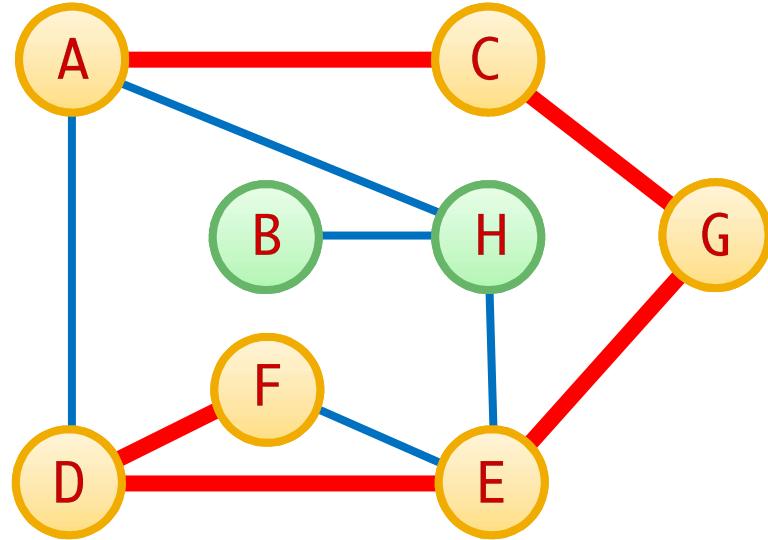
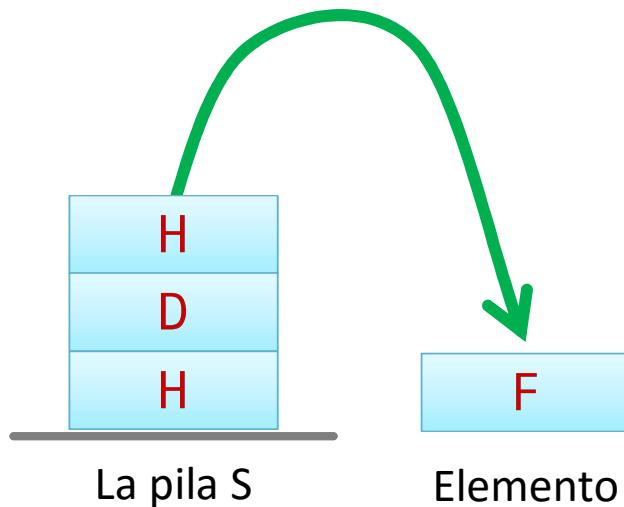
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>20</sub>)

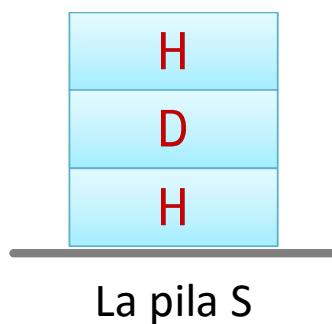
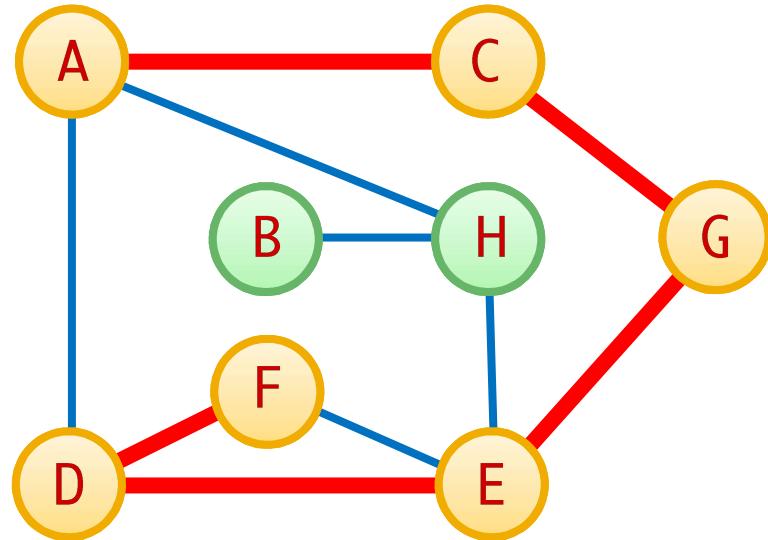
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>21</sub>)

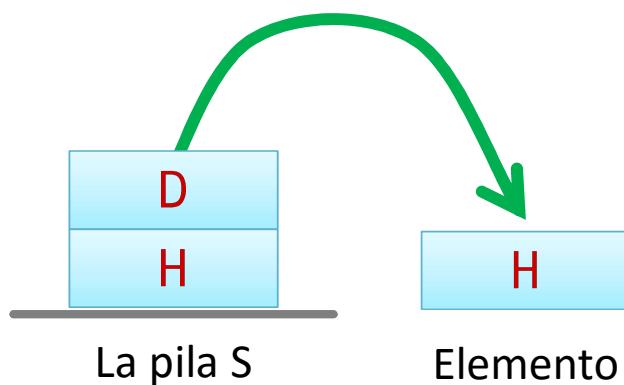
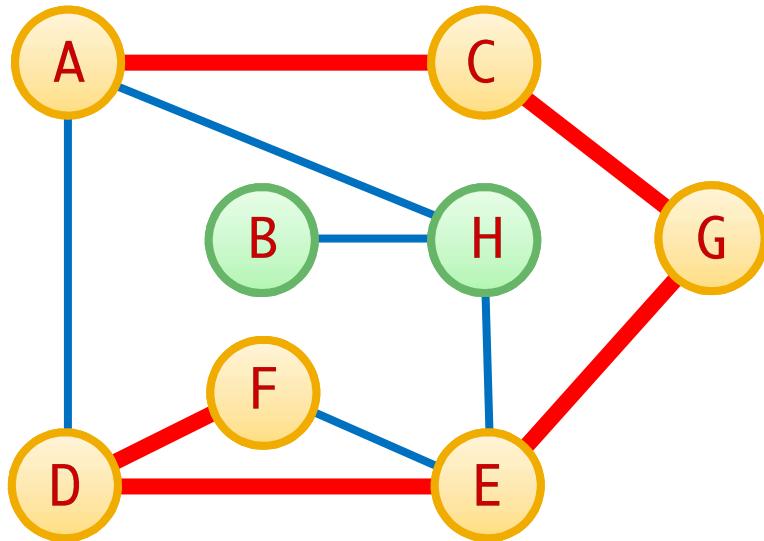
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>22</sub>)

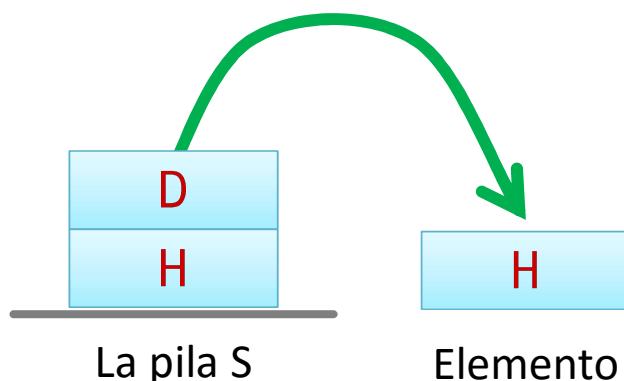
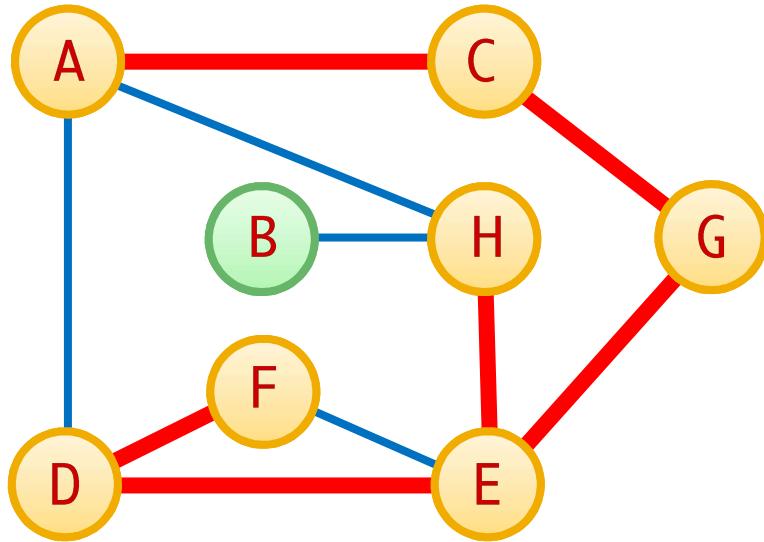
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>23</sub>)

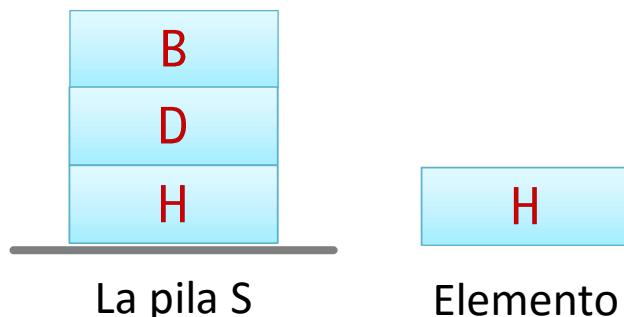
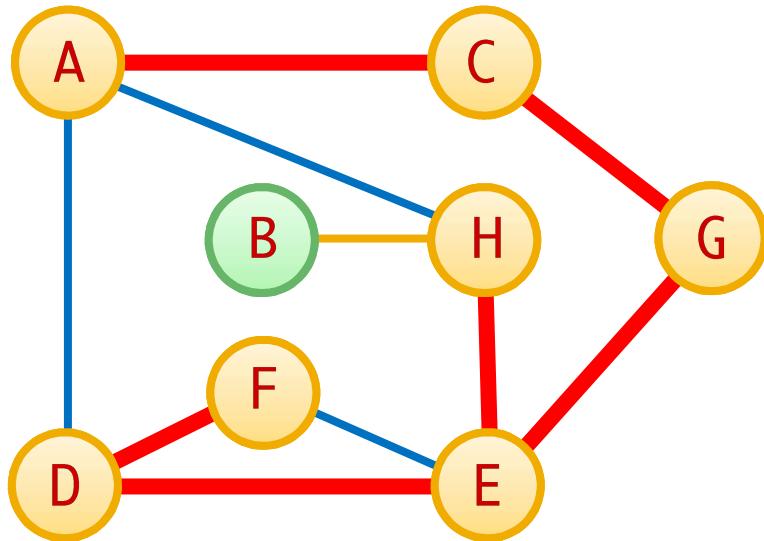
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
  - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>24</sub>)

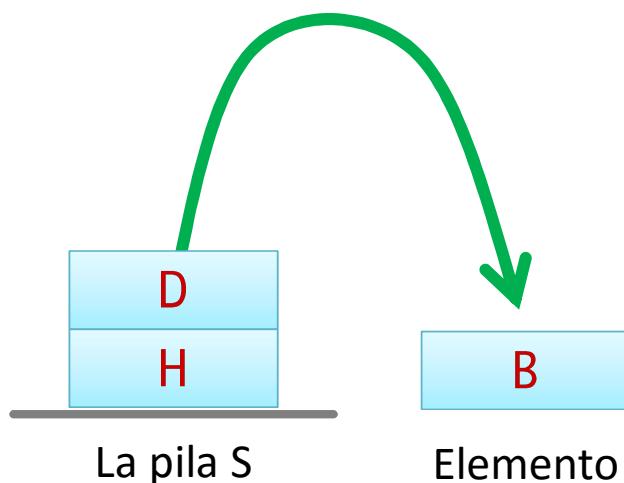
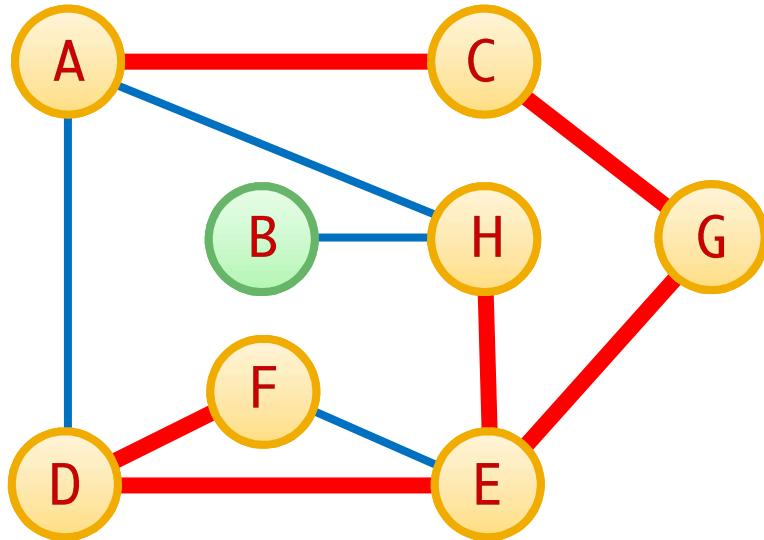
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>25</sub>)

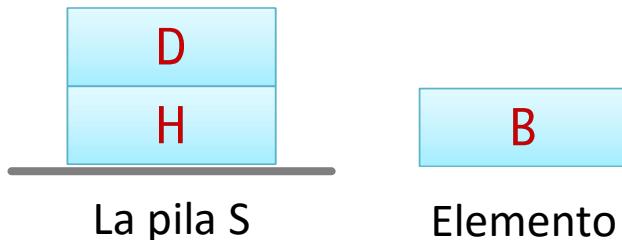
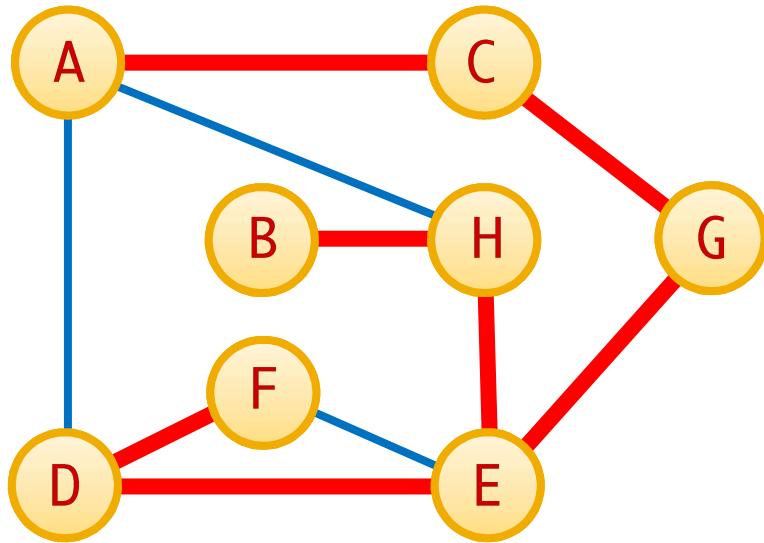
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>26</sub>)

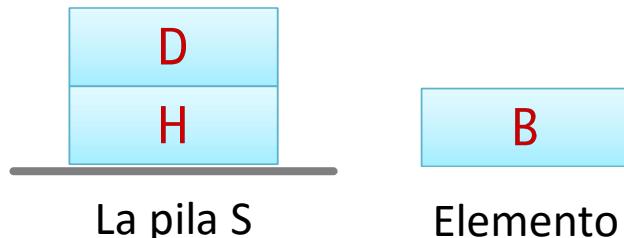
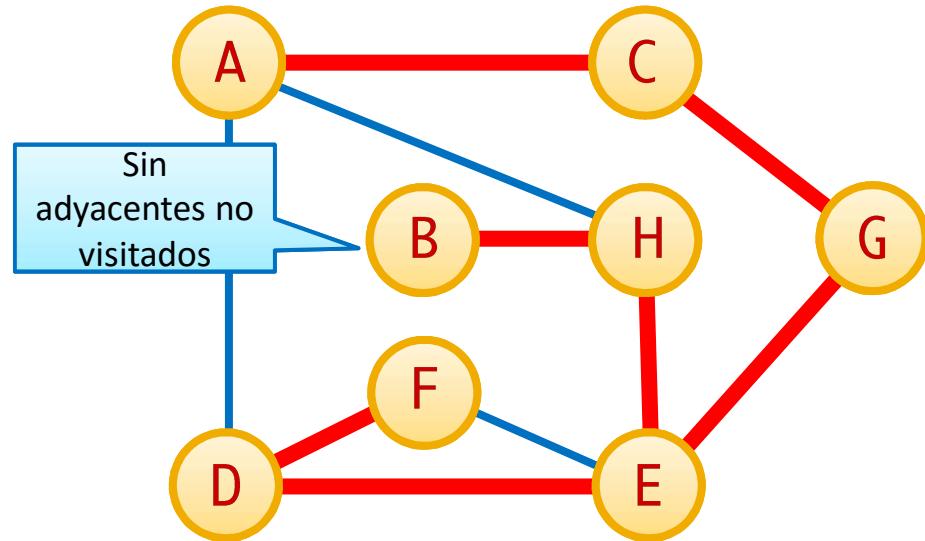
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
  - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>27</sub>)

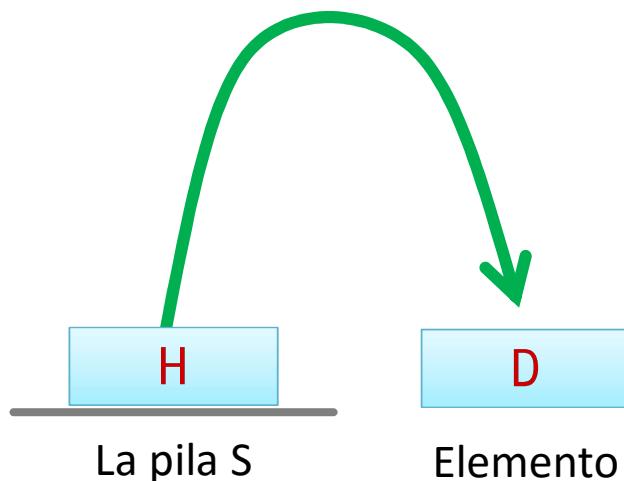
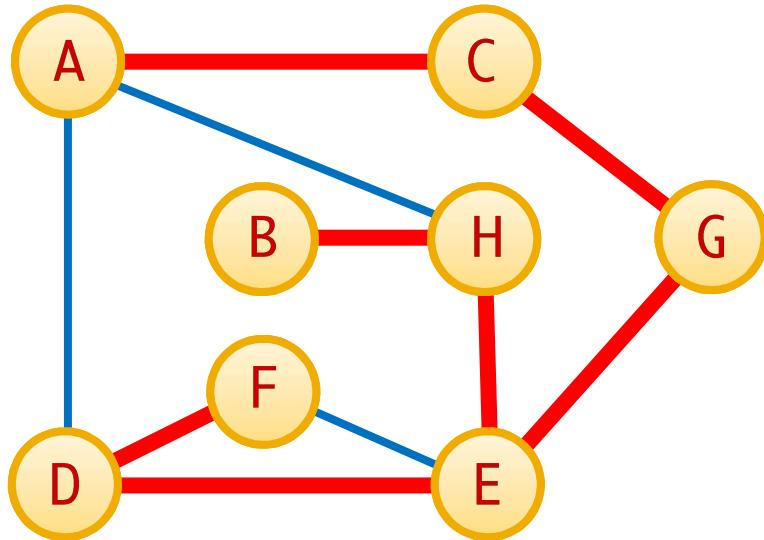
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (II<sub>28</sub>)

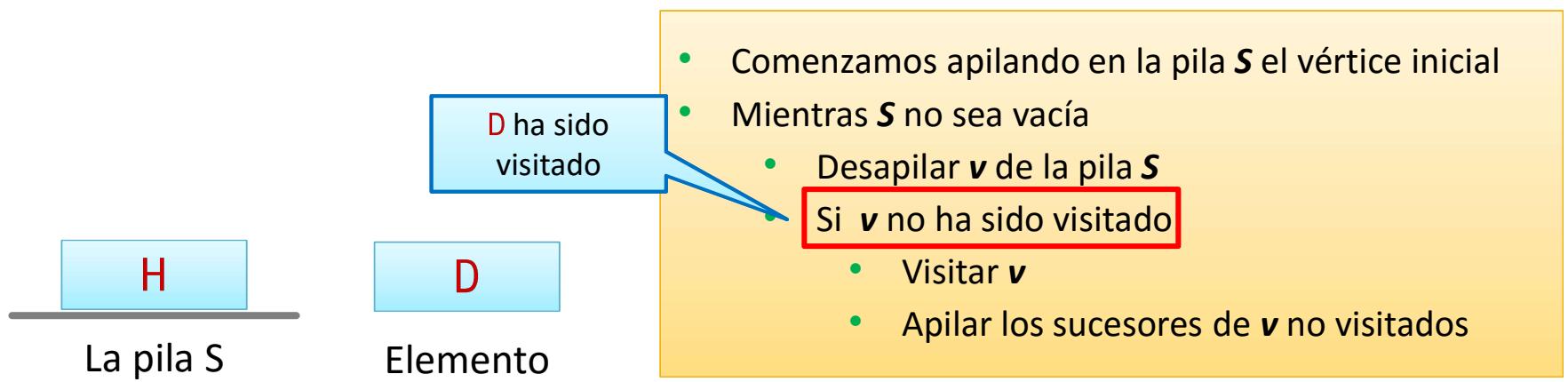
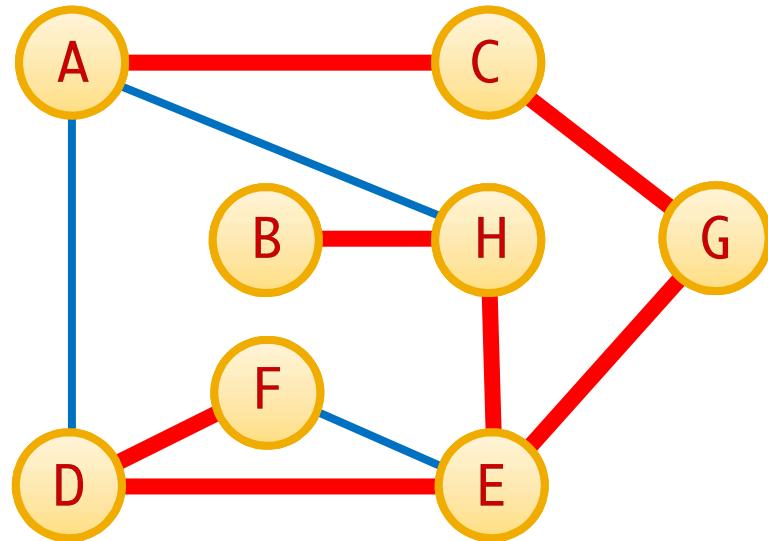
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

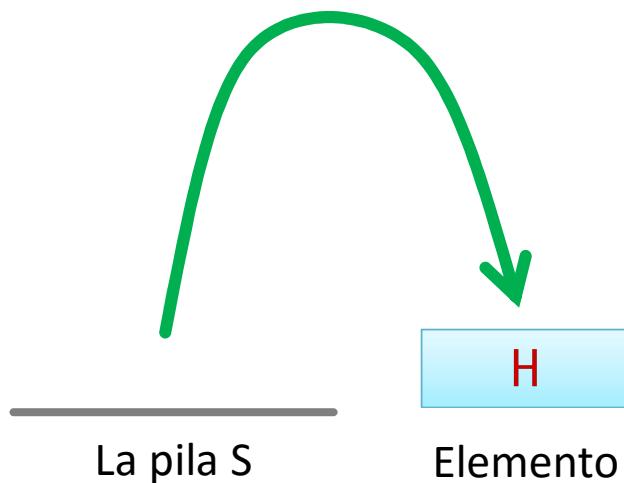
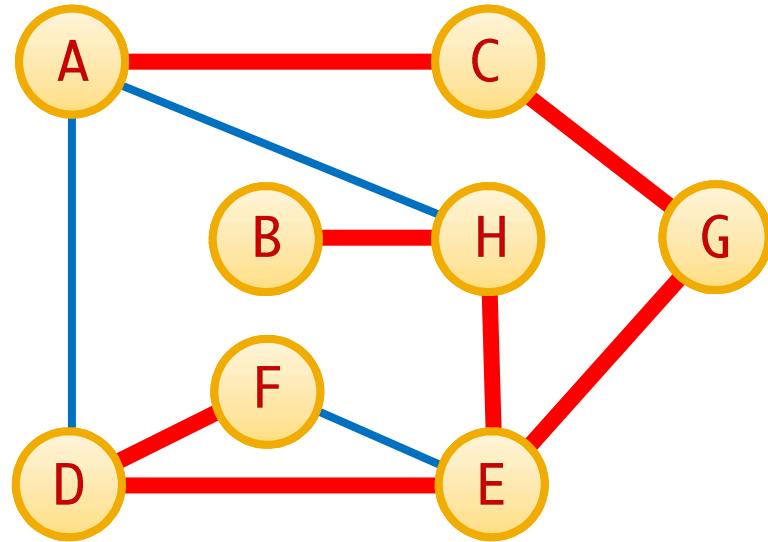
# DFT: Implementación con un Stack (II<sub>29</sub>)

## ■ DFT comenzando en A



# DFT: Implementación con un Stack ( $\Pi_{30}$ )

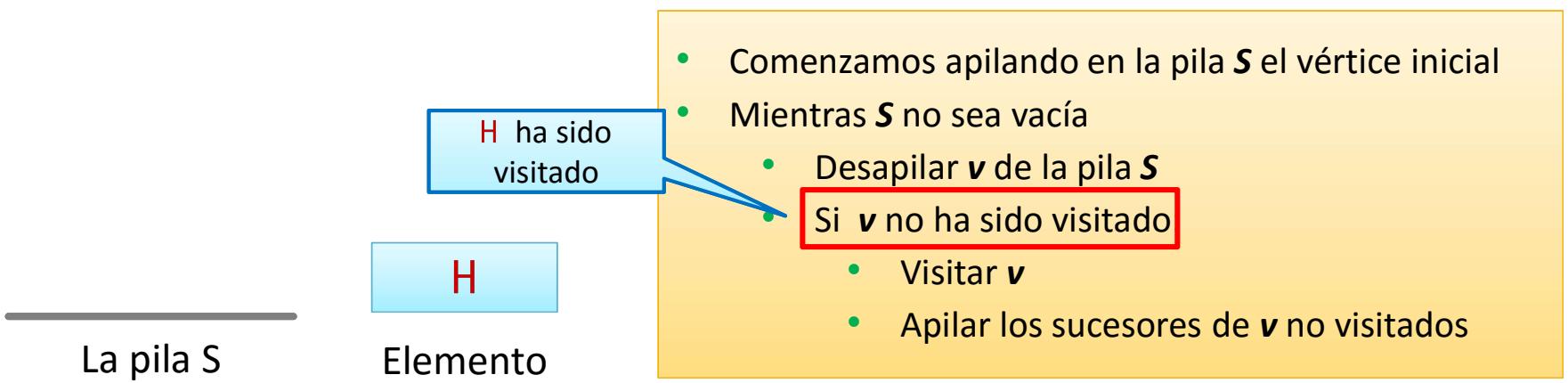
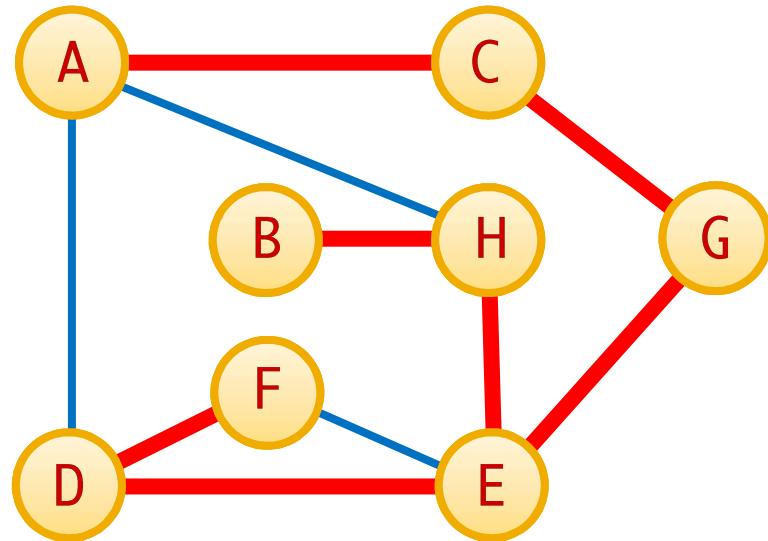
## ■ DFT comenzando en A



- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

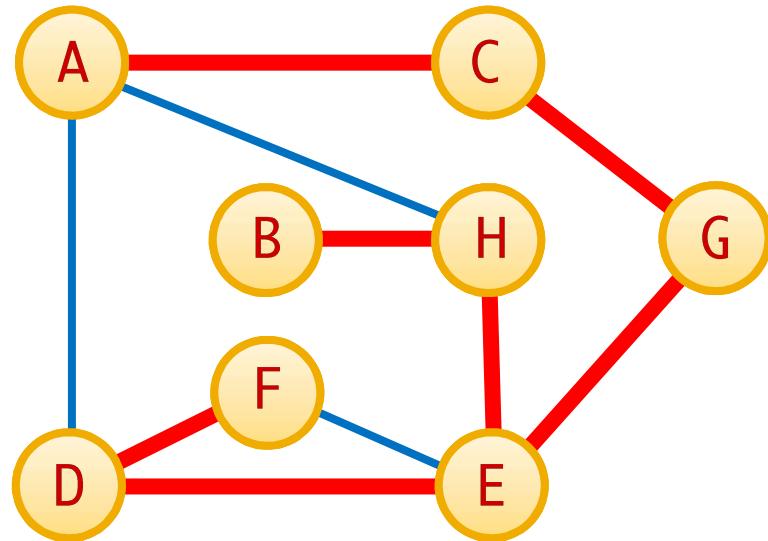
# DFT: Implementación con un Stack (II<sub>31</sub>)

## ■ DFT comenzando en A



# DFT: Implementación con un Stack ( $\Pi_{32}$ )

## ■ DFT comenzando en A



Pila vacía. Fin del recorrido

H

La pila S

Elemento

- Comenzamos apilando en la pila  $S$  el vértice inicial
- Mientras  $S$  no sea vacía
  - Desapilar  $v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Apilar los sucesores de  $v$  no visitados

# DFT: Implementación con un Stack (III)

- Recordemos la interfaz Pila (Stack):

**data** Stack a

empty :: Stack a

isEmpty :: Stack a -> Bool

push :: a -> Stack a -> Stack a

top :: Stack a -> a

pop :: Stack a -> Stack a

# DFT: Implementación con un Stack (IV)

- Recordemos también la interfaz Conjunto (Set):

**data** Set a

empty :: Set a

isEmpty :: Set a -> Bool

insert :: (Ord a) => a -> Set a -> Set a

isElem :: (Ord a) => a -> Set a -> Bool

notIsElem :: (Ord a) => a -> Set a -> Bool

delete :: (Ord a) => a -> Set a -> Set a

# DFT: Implementación con un Stack (V)

```
import DataStructures.Stack.LinearStack
import qualified DataStructures.Set.BSTSet as S
import DataStructures.Graph.Graph

pushAll :: Stack a -> [a] -> Stack a
pushAll s xs = foldr push s xs
```

Apila en s todos los elementos de la lista xs

```
dft :: (Ord a) => Graph a -> a -> [a]
dft g v0 = aux S.empty (push v0 empty)
```

Devuelve la lista de vértices de la DFT comenzando en v0

where

```
aux visited stack
| isEmpty stack      = [] -- fin de recorrido
| v `S.isElem` visited = aux visited stack' -- v ha sido visitado
| otherwise           = v : aux visited' (pushAll stack' us)
```

Conjunto vacío de vértices visitados y pila con el vértice inicial

where

```
v = top stack
stack' = pop stack
visited' = S.insert v visited
us = [ u | u <- successors g v, u `S.notIsElem` visited ]
```

vértice visitado

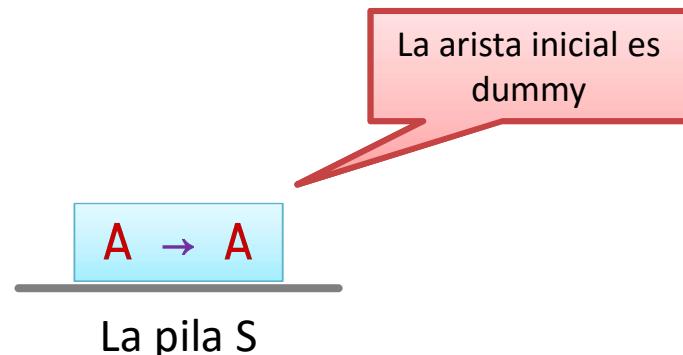
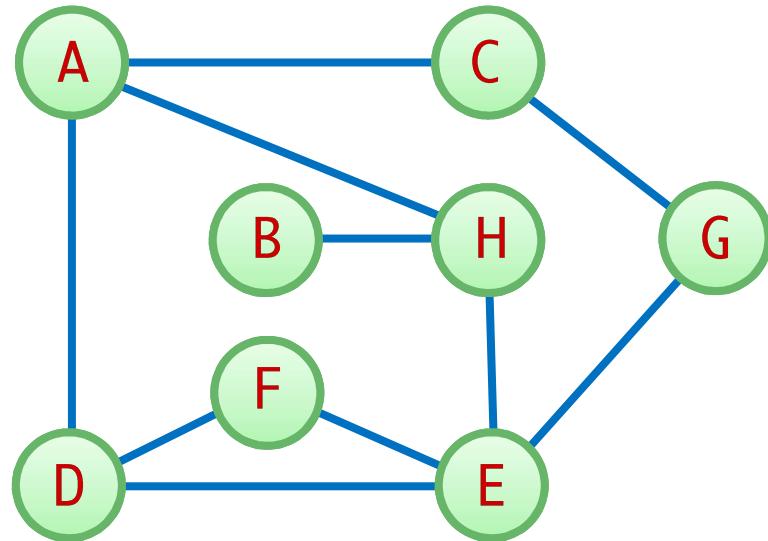
sucesores de v no visitados

# DFT: Caminos a vértices

- Para la mayoría de aplicaciones una lista con los vértices recorridos es insuficiente 😞
- Normalmente es necesario considerar los caminos (desde el vértice inicial) utilizados para alcanzar los vértices visitados:
  - Un **diccionario** puede *guardar* cada arista recorrida durante la visita (las aristas del *spanning tree*)
  - Con este diccionario podremos reconstruir el camino a cada vértice visitado.
  - Para conseguir esto, guardaremos en el stack las aristas pendientes de recorrer

# DFT: Caminos a vértices (II<sub>1</sub>)

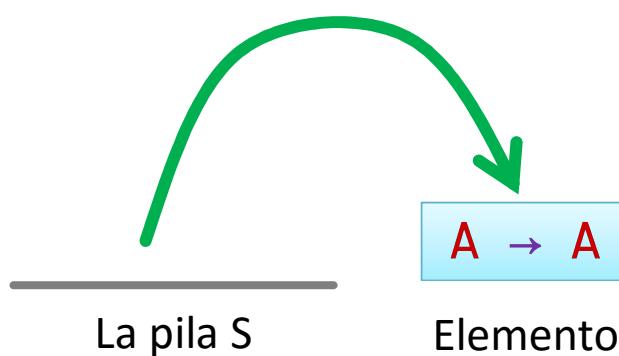
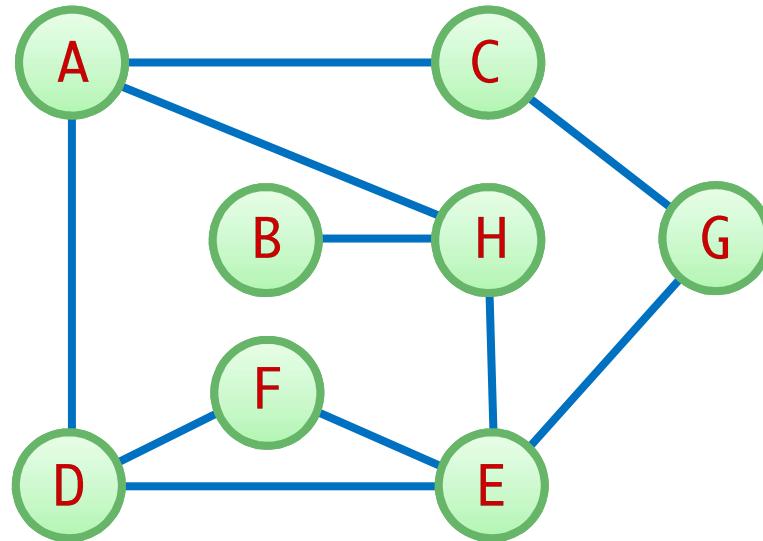
## ■ DFT comenzando en A



- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Añadir  $v \leftarrow w$  al diccionario
    - Apilar en  $S$  las aristas de suc. no visitados

# DFT: Caminos a vértices (II<sub>2</sub>)

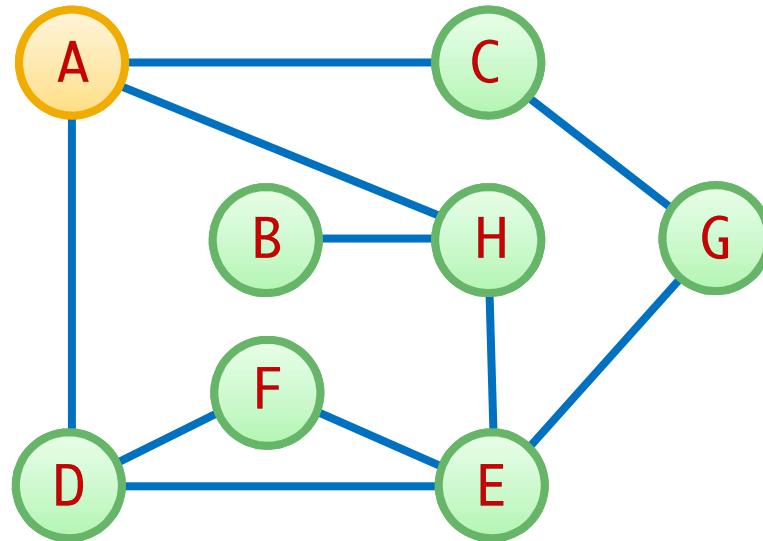
## ■ DFT comenzando en A



- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Añadir  $v \leftarrow w$  al diccionario
    - Apilar en  $S$  las aristas de suc. no visitados

# DFT: Caminos a vértices (II<sub>3</sub>)

## ■ DFT comenzando en A



- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Añadir  $v \leftarrow w$  al diccionario
    - Apilar en  $S$  las aristas de suc. no visitados

$A \rightarrow A$

La pila S

Elemento

# DFT: Caminos a vértices (II<sub>4</sub>)

## ■ DFT comenzando en A

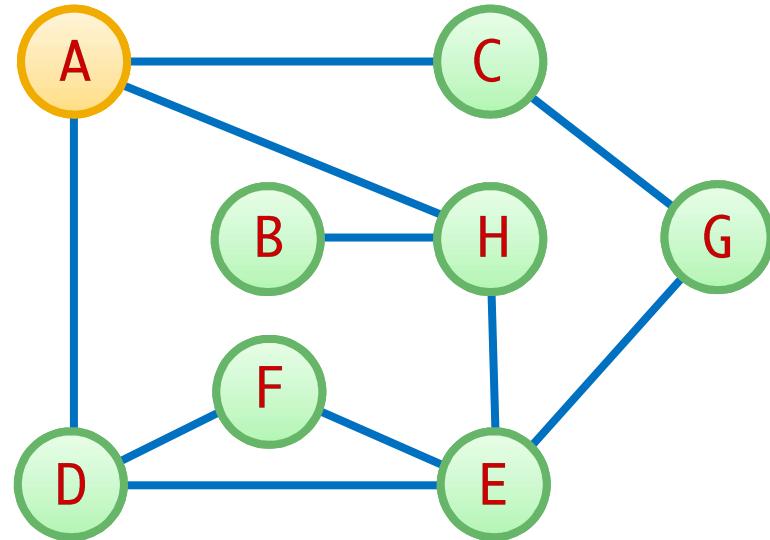
El diccionario

A  $\leftarrow$  A

La pila S

Elemento

A  $\rightarrow$  A



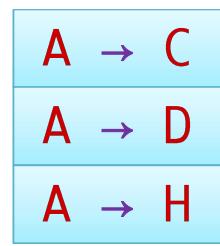
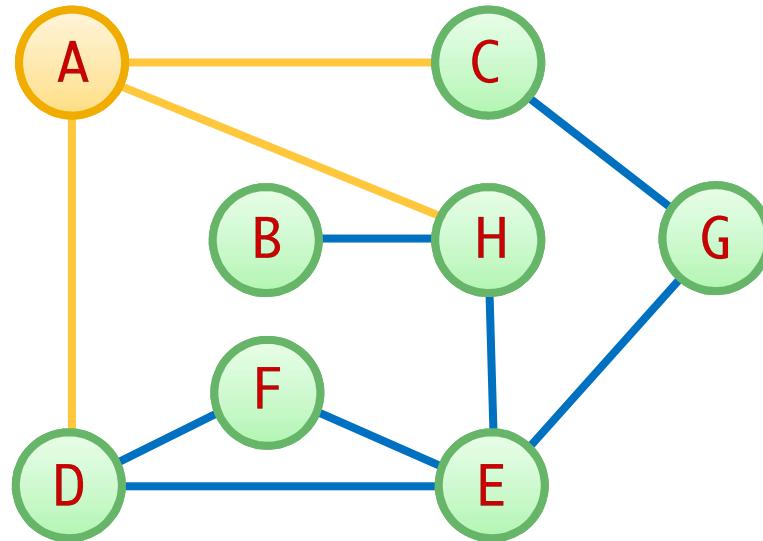
- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - **Añadir  $v \leftarrow w$  al diccionario**
    - Apilar en  $S$  las aristas de suc. no visitados

# DFT: Caminos a vértices (II<sub>5</sub>)

## ■ DFT comenzando en A

El diccionario

$A \leftarrow A$



La pila S

$A \rightarrow A$

Elemento

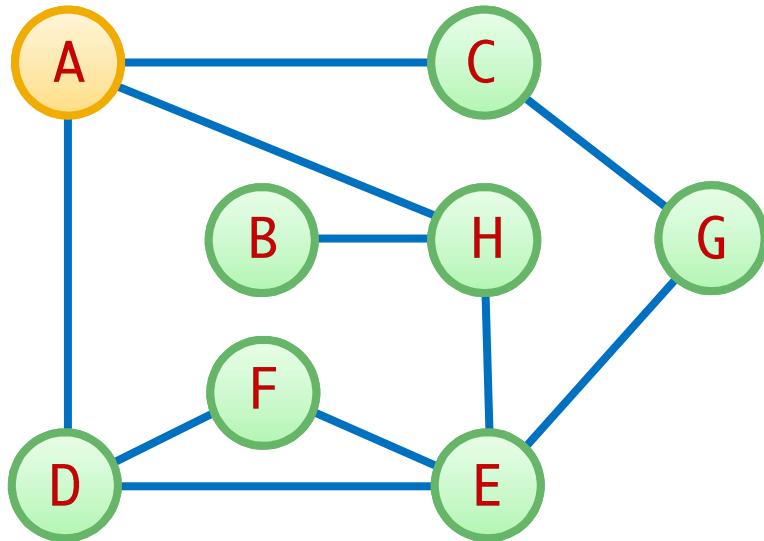
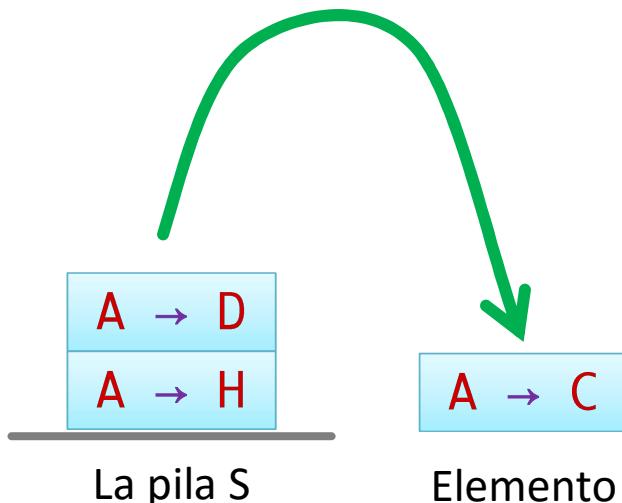
- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Añadir  $v \leftarrow w$  al diccionario
    - Apilar en  $S$  las aristas de suc. no visitados

# DFT: Caminos a vértices (II<sub>6</sub>)

## ■ DFT comenzando en A

El diccionario

$A \leftarrow A$



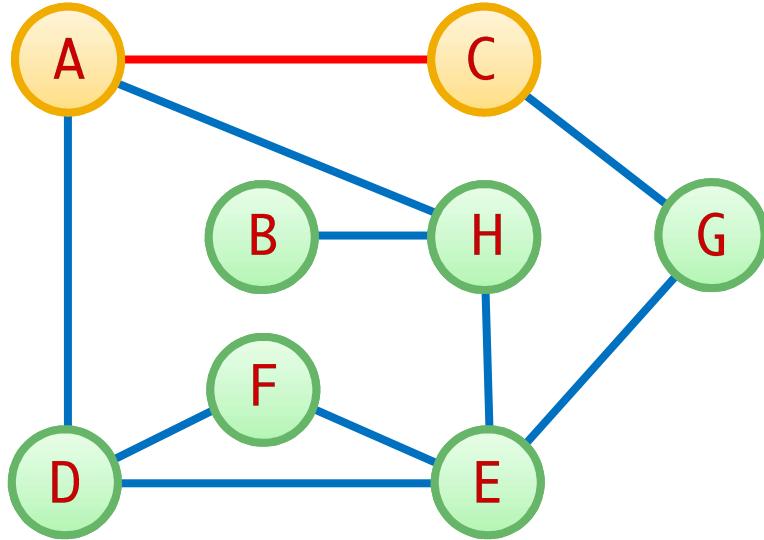
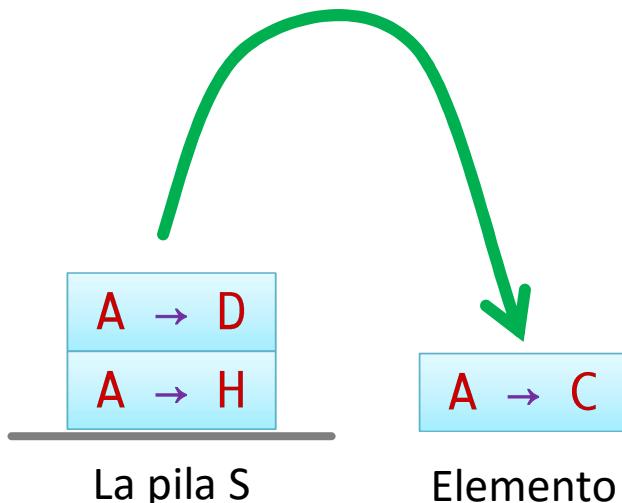
- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Añadir  $v \leftarrow w$  al diccionario
    - Apilar en  $S$  las aristas de suc. no visitados

# DFT: Caminos a vértices (II<sub>7</sub>)

## ■ DFT comenzando en A

El diccionario

$A \leftarrow A$



- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Añadir  $v \leftarrow w$  al diccionario
    - Apilar en  $S$  las aristas de suc. no visitados

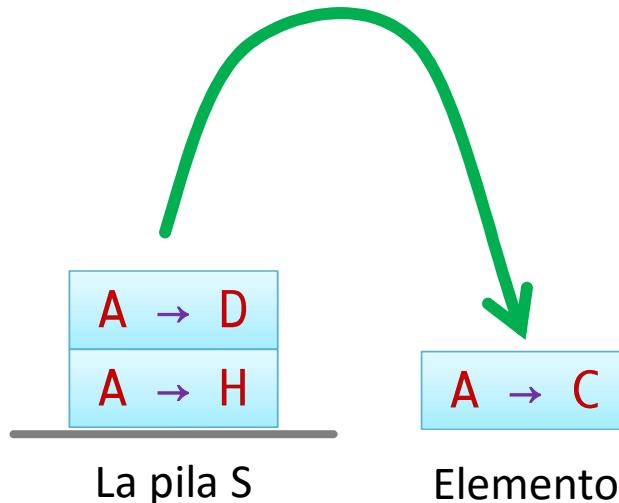
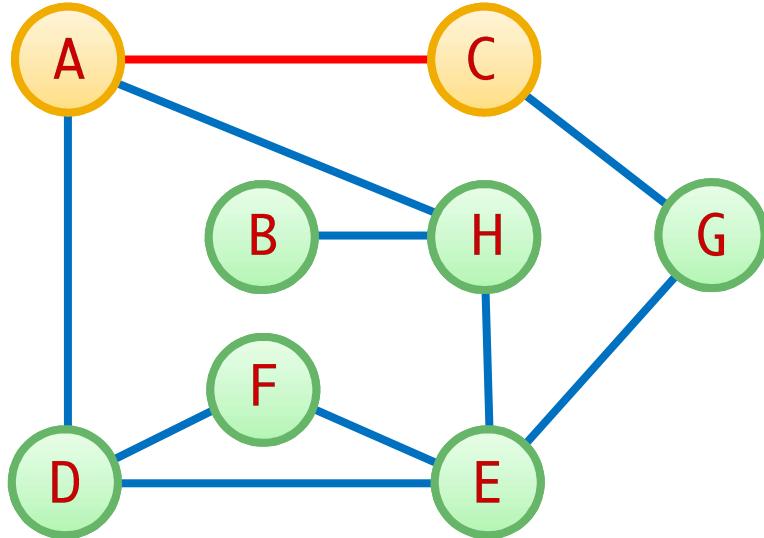
# DFT: Caminos a vértices (II<sub>8</sub>)

## ■ DFT comenzando en A

El diccionario

A  $\leftarrow$  A C  $\leftarrow$  A

Significa que podemos ir a C desde A



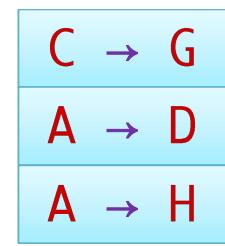
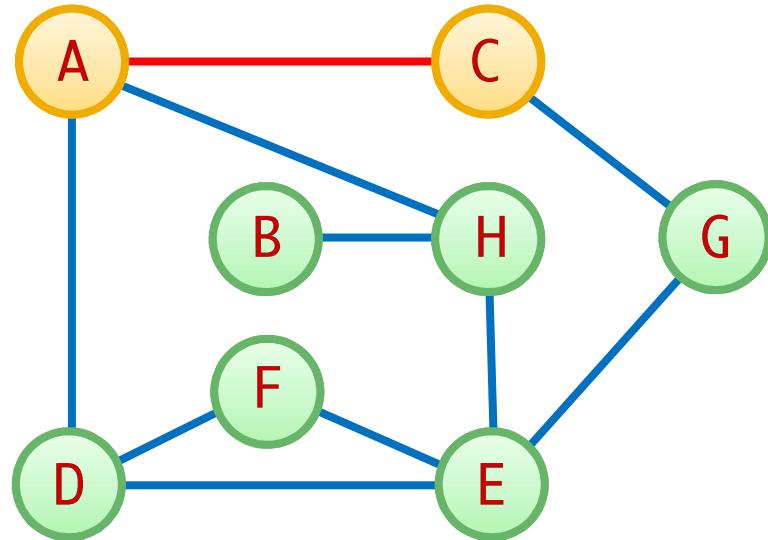
- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - **Añadir  $v \leftarrow w$  al diccionario**
    - Apilar en  $S$  las aristas de suc. no visitados

# DFT: Caminos a vértices (II<sub>9</sub>)

## ■ DFT comenzando en A

El diccionario

A  $\leftarrow$  A C  $\leftarrow$  A



La pila S

A  $\rightarrow$  C

Elemento

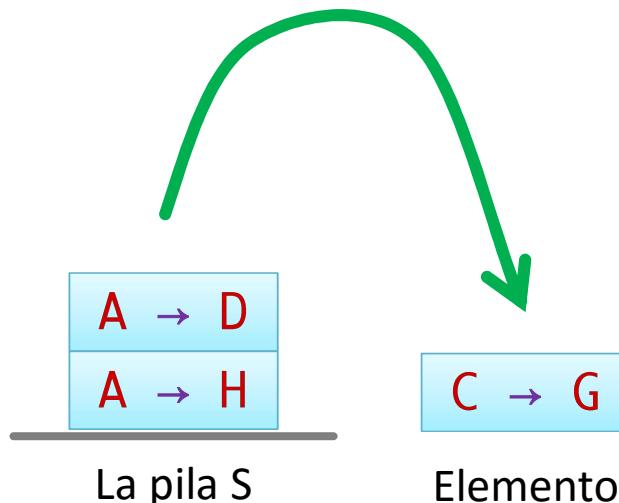
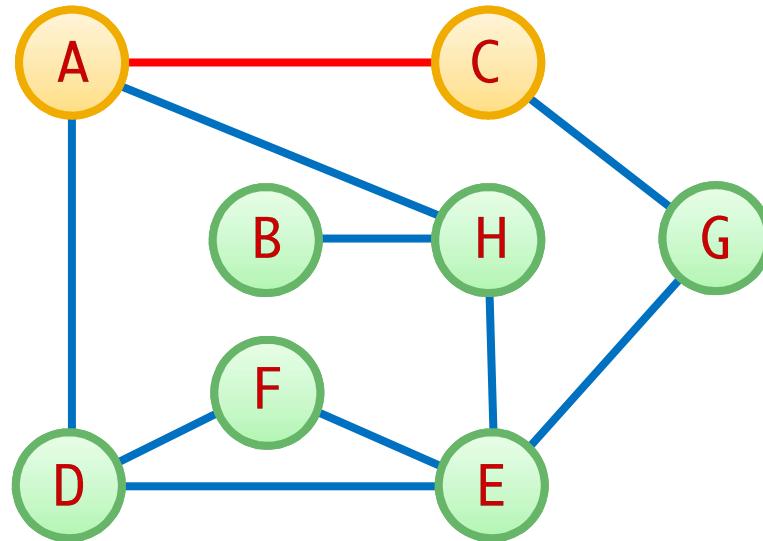
- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Añadir  $v \leftarrow w$  al diccionario
    - Apilar en  $S$  las aristas de suc. no visitados

# DFT: Caminos a vértices (II<sub>10</sub>)

## ■ DFT comenzando en A

El diccionario

A  $\leftarrow$  A C  $\leftarrow$  A



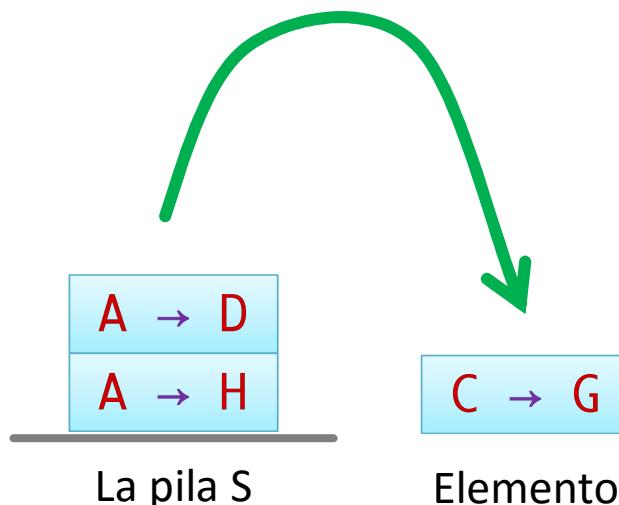
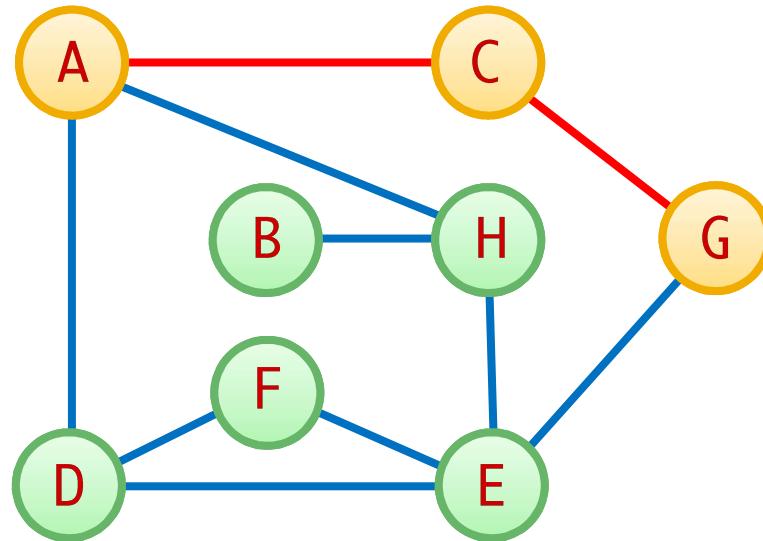
- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Añadir  $v \leftarrow w$  al diccionario
    - Apilar en  $S$  las aristas de suc. no visitados

# DFT: Caminos a vértices (II<sub>11</sub>)

## ■ DFT comenzando en A

El diccionario

A  $\leftarrow$  A C  $\leftarrow$  A



- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Añadir  $v \leftarrow w$  al diccionario
    - Apilar en  $S$  las aristas de suc. no visitados

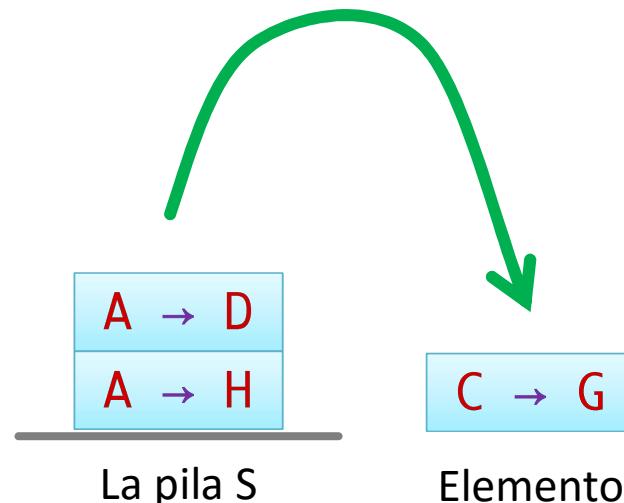
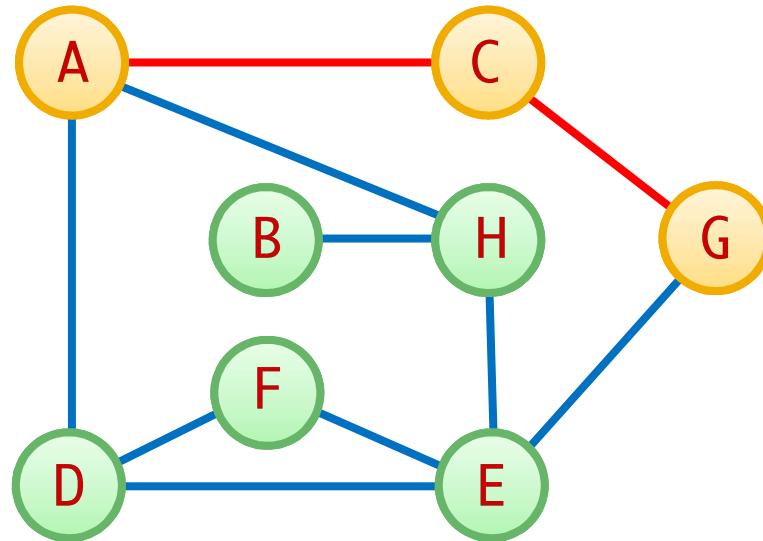
# DFT: Caminos a vértices (II<sub>12</sub>)

## ■ DFT comenzando en A

El diccionario

A  $\leftarrow$  A   C  $\leftarrow$  A   G  $\leftarrow$  C

Podemos ir a G  
desde C



- Comenzamos apilando en  $S$  una arista inicial  $A \rightarrow A$
- Mientras  $S$  no sea vacía
  - Desapilar  $w \rightarrow v$  de la pila  $S$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - **Añadir  $v \leftarrow w$  al diccionario**
    - Apilar en  $S$  las aristas de suc. no visitados

# DFT: Caminos a vértices ( $\Pi_{13}$ )

Tras completar el recorrido DFT...

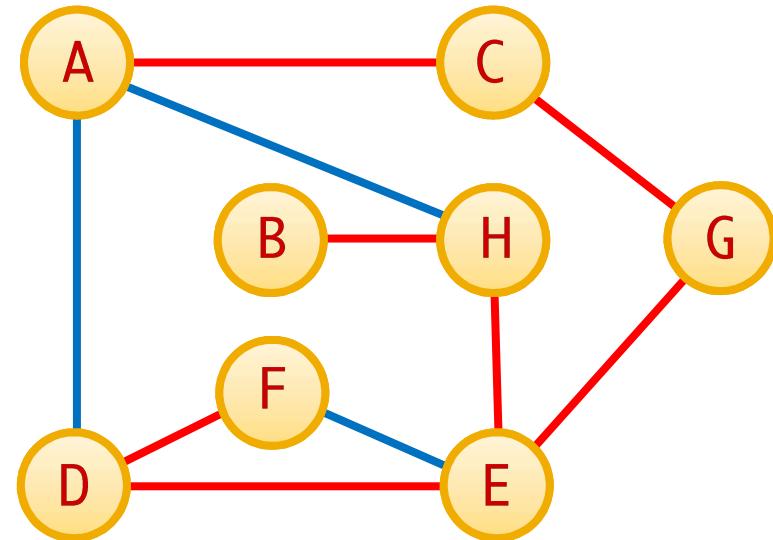
# DFT: Caminos a vértices (II<sub>14</sub>)

## ■ DFT comenzando en A

El diccionario

A $\leftarrow$ A	C $\leftarrow$ A	G $\leftarrow$ C
E $\leftarrow$ G	D $\leftarrow$ E	F $\leftarrow$ D
H $\leftarrow$ E	B $\leftarrow$ H	

Información suficiente  
para reconstruir los  
caminos desde A



# DFT: Caminos a vértices (III)

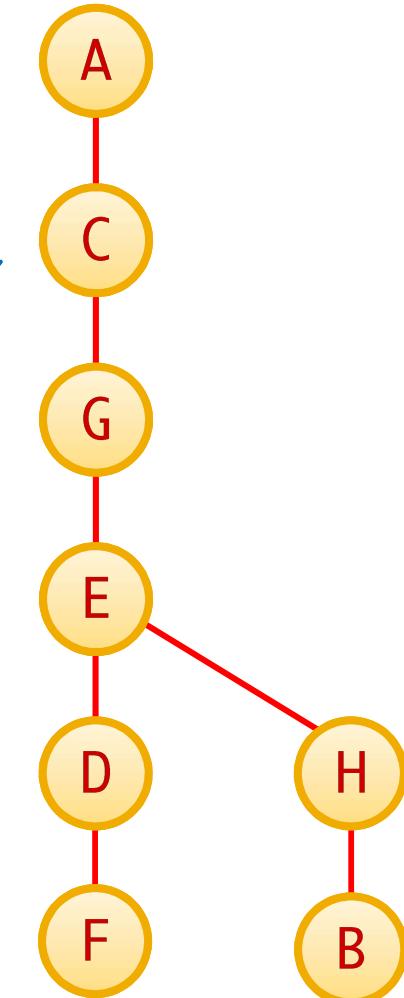
## ■ DFT comenzando en A

El diccionario

A ← A	C ← A	G ← C
E ← G	D ← E	F ← D
H ← E	B ← H	

C es padre de G  
en el árbol de  
expansión

Diccionario asociado a  
este árbol de expansión



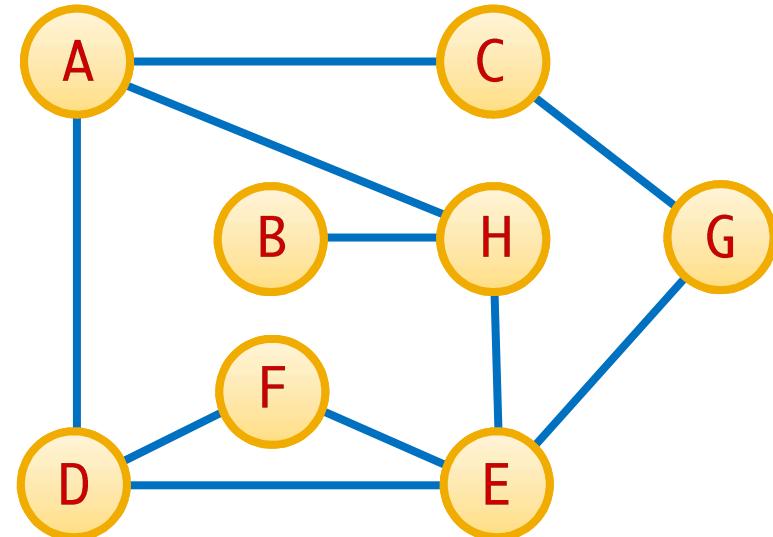
# DFT: Reconstrucción de un camino

## ■ DFT comenzando en A

El diccionario

A $\leftarrow$ A	C $\leftarrow$ A	G $\leftarrow$ C
E $\leftarrow$ G	D $\leftarrow$ E	F $\leftarrow$ D
H $\leftarrow$ E	B $\leftarrow$ H	

¿Cómo llegamos desde  
A hasta el vértice B ?



# DFT: Reconstrucción de un camino (II<sub>1</sub>)

## ■ DFT comenzando en A

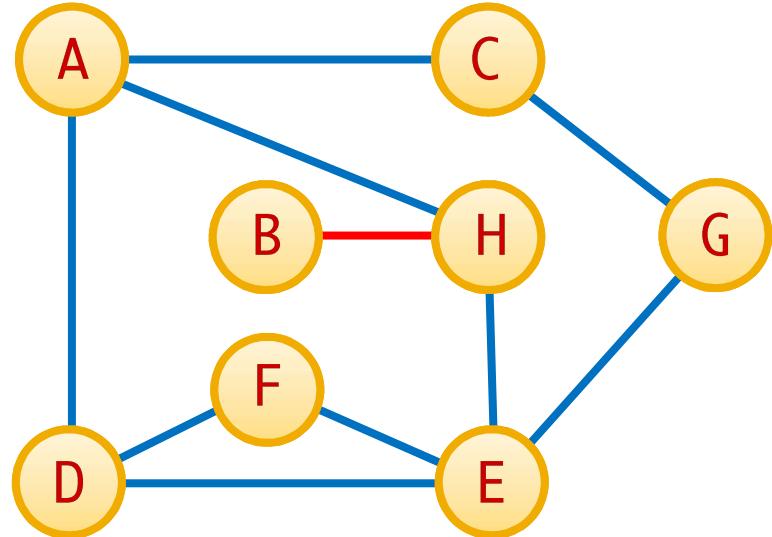
El diccionario

A $\leftarrow$ A	C $\leftarrow$ A	G $\leftarrow$ C
E $\leftarrow$ G	D $\leftarrow$ E	F $\leftarrow$ D
H $\leftarrow$ E	B $\leftarrow$ H	

¿Cómo ir desde A hasta el vértice B ?

B  $\leftarrow$  H

¿Cómo llegamos a H ?



# DFT: Reconstrucción de un camino (II<sub>2</sub>)

## ■ DFT comenzando en A

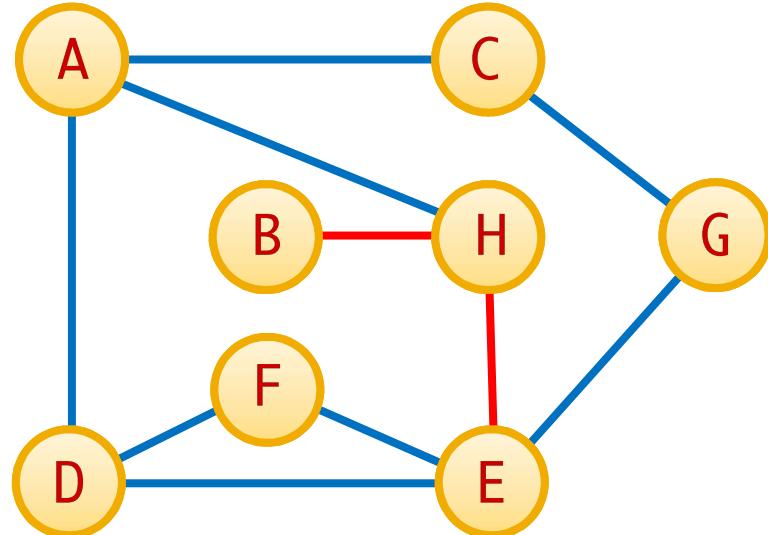
El diccionario

A $\leftarrow$ A	C $\leftarrow$ A	G $\leftarrow$ C
E $\leftarrow$ G	D $\leftarrow$ E	F $\leftarrow$ D
H $\leftarrow$ E		B $\leftarrow$ H

¿Cómo ir desde A hasta el vértice B ?

B  $\leftarrow$  H  $\leftarrow$  E

¿Cómo llegamos a E ?



# DFT: Reconstrucción de un camino (II<sub>3</sub>)

## ■ DFT comenzando en A

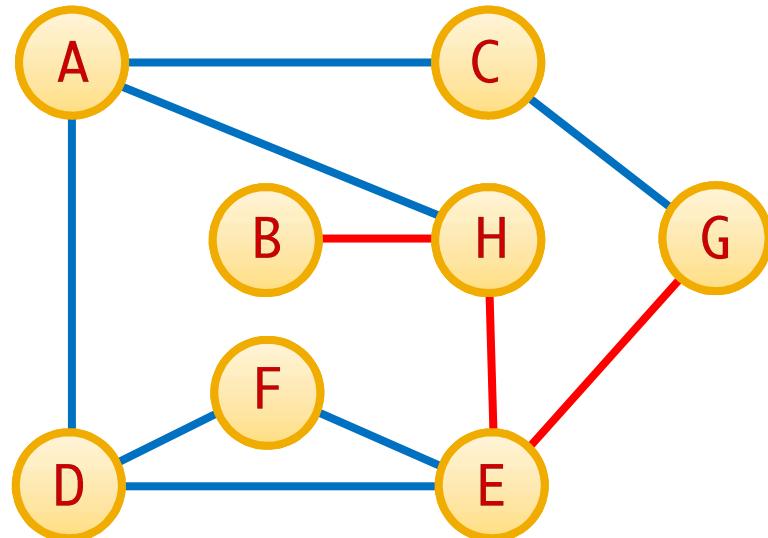
El diccionario

A $\leftarrow$ A	C $\leftarrow$ A	G $\leftarrow$ C
E $\leftarrow$ G	D $\leftarrow$ E	F $\leftarrow$ D
H $\leftarrow$ E	B $\leftarrow$ H	

¿Cómo ir desde A hasta el vértice B ?

B  $\leftarrow$  H  $\leftarrow$  E  $\leftarrow$  G

¿Cómo llegamos a G ?



# DFT: Reconstrucción de un camino (II<sub>4</sub>)

## ■ DFT comenzando en A

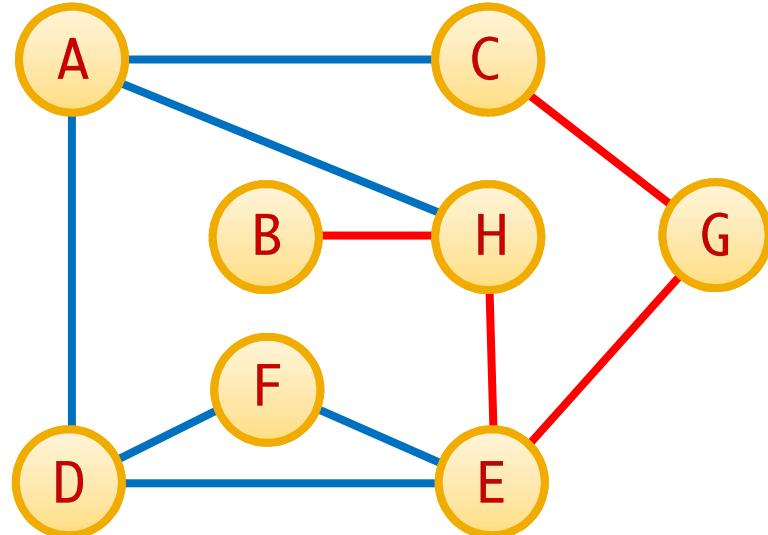
El diccionario

A $\leftarrow$ A	C $\leftarrow$ A	G $\leftarrow$ C
E $\leftarrow$ G	D $\leftarrow$ E	F $\leftarrow$ D
H $\leftarrow$ E	B $\leftarrow$ H	

¿Cómo ir desde A hasta el vértice B ?

B  $\leftarrow$  H  $\leftarrow$  E  $\leftarrow$  G  $\leftarrow$  C

¿Cómo llegamos a C ?



# DFT: Reconstrucción de un camino (II<sub>5</sub>)

## ■ DFT comenzando en A

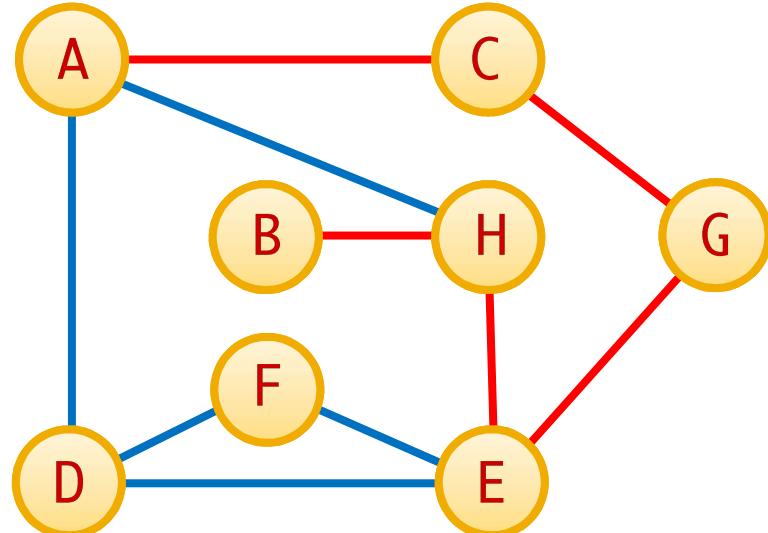
El diccionario

A $\leftarrow$ A	C $\leftarrow$ A	G $\leftarrow$ C
E $\leftarrow$ G	D $\leftarrow$ E	F $\leftarrow$ D
H $\leftarrow$ E	B $\leftarrow$ H	

¿Cómo ir desde A hasta el vértice B ?

B  $\leftarrow$  H  $\leftarrow$  E  $\leftarrow$  G  $\leftarrow$  C  $\leftarrow$  A

A es el inicial: hemos construimos un camino desde A hasta B:  
A-C-G-E-H-B



# DFT: Caminos a vértices (IV)

## ■ Recordemos la interfaz Dictionary

```
data Dictionary a b

empty :: Dictionary a b

isEmpty :: Dictionary a b -> Bool

insert :: (Ord a) => a -> b -> Dictionary a b -> Dictionary a b

valueOf :: (Ord a) => a -> Dictionary a b -> Maybe b

isDefinedAt :: (Ord a) => a -> Dictionary a b -> Bool

delete :: (Ord a) => a -> Dictionary a b -> Dictionary a b

keys :: (Ord a) => Dictionary a b -> [a]

values :: (Ord a) => Dictionary a b -> [b]
```

# DFT: Caminos a vértices (V)

```
import DataStructures.Stack.LinearStack
import qualified DataStructures.Set.BSTSet as S
import qualified DataStructures.Dictionary.BSTDictionary as D
import DataStructures.Graph.Graph

data AnEdge a = a :-> a -- w :-> v significa que llegamos a v desde w
-- la siguiente devuelve los caminos desde un vértice
-- en una visita DFT
dftPaths :: (Ord a) => Graph a -> a -> [Path a]
dftPaths g v0 = aux S.empty (push (v0 :-> v0) empty) D.empty
where
  aux visited stack dict
    | isEmpty stack      = [] -- fin de recorrido
    | v `S.isElem` visited = aux visited stack' dict -- v fue visitado
    | otherwise           =
        pathFromTo v0 v dict' :-- devolvemos un camino desde v0 hasta v
        aux visited' (pushAll stack' es) dict'
  where
    w :-> v = top stack
    stack' = pop stack
    visited' = S.insert v visited
    dict' = D.insert v w dict -- el padre v es w
    es = [ v :-> u | u <- successors g v, u `S.notIsElem` visited ]
```

Sucesores de v no visitados

# DFT: Caminos a vértices (VI)

-- reconstrucción de un camino desde  $v_0$  hasta  $w$

-- a través de un diccionario

`pathFromTo :: (Ord a) => a -> a -> D.Dictionary a a -> [a]`

`pathFromTo v0 w dict = reverse (aux w)`

**where**

`aux w`

|  $w == v_0 = [w]$

| otherwise =  $w : aux u$

**where Just u = D.valueOf w dict**

u es el padre de w en el correspondiente  
spanning tree

# Recorrido en amplitud (Breadth First Traversal, o BFT)

- Recorrido en amplitud desde cierto vértice  $v$
- $BFT(v)$ 
  - $S_0 = \{ v \}$
  - Visitar todos los vértices de  $S_0$
  - $S_1 = \{ w \mid w' \in S_0, w \in \text{sucesores de } w' \text{ no visitados} \}$
  - Visitar todos los vértice de  $S_1$
  - $S_2 = \{ w \mid w' \in S_1, w \in \text{sucesores de } w' \text{ no visitados} \}$
  - Visitar todos los vértice de  $S_2$
  - ...
  - Hasta visitar todos los vértices

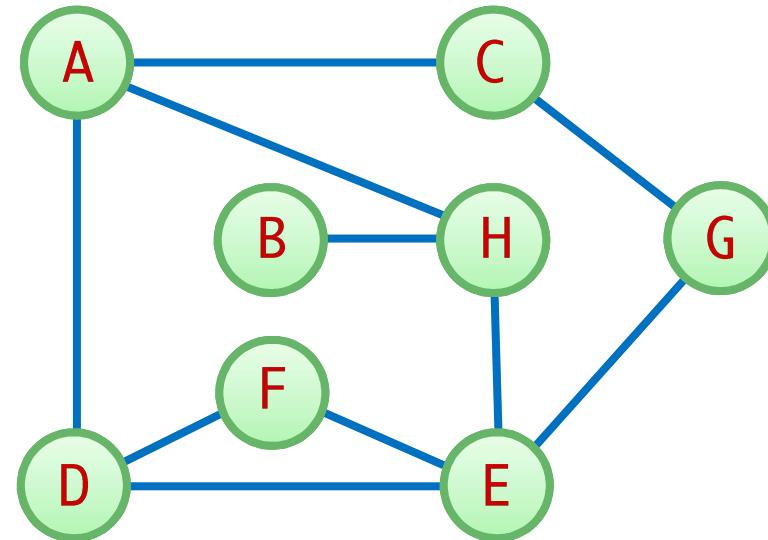
$BFT(v)$  encuentra el camino  
más corto hasta cualquier  
nodo conectado a  $v$  😊

Vértices no visitados que  
están a distancia 1 desde  $v$

Vértices no visitados que  
están a distancia 2 desde  $v$

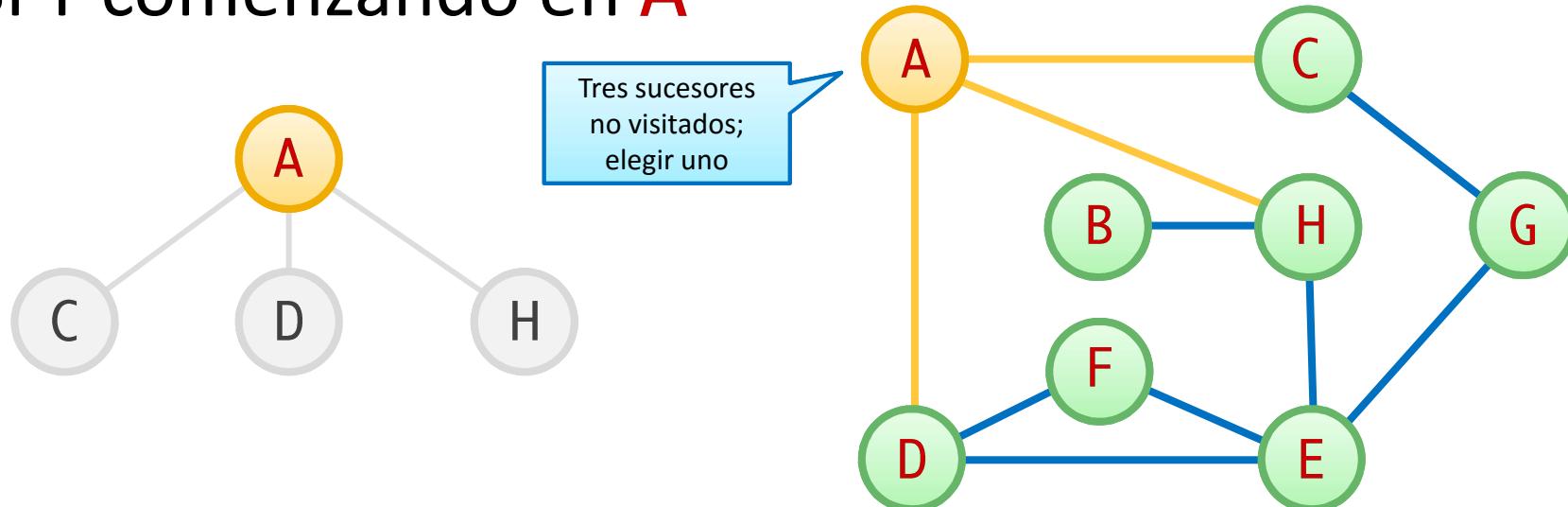
# Recorrido en amplitud ( $lI_1$ )

- BFT comenzando en A



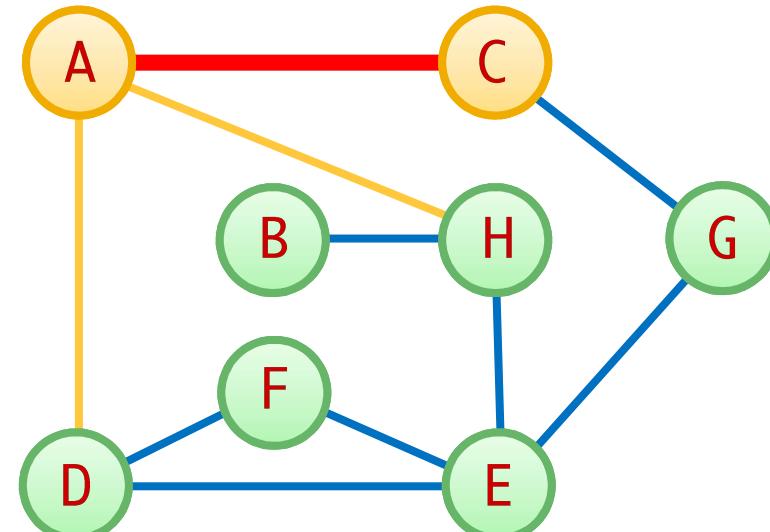
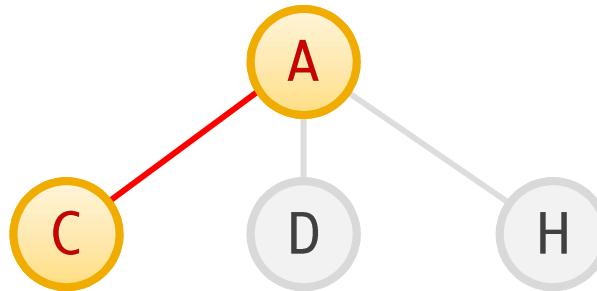
# Recorrido en amplitud ( $II_2$ )

- BFT comenzando en A



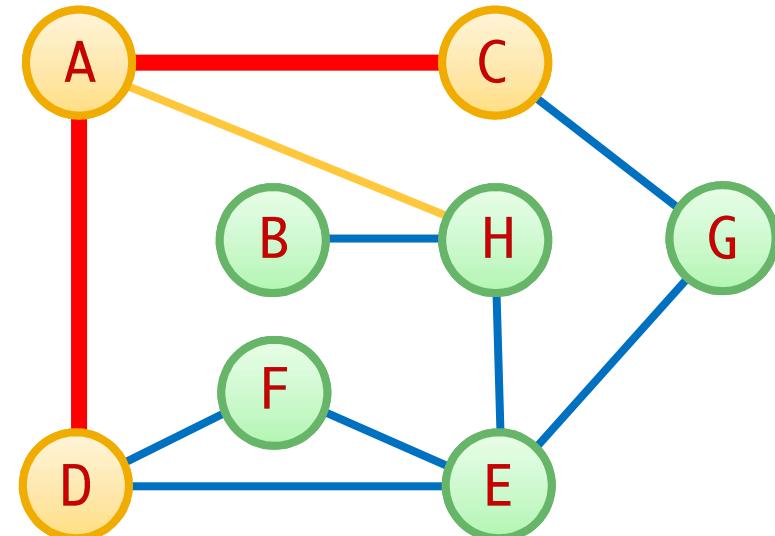
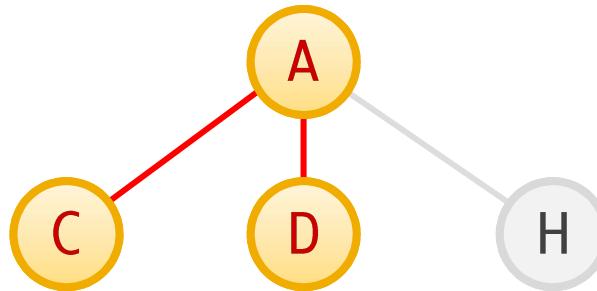
# Recorrido en amplitud ( $II_3$ )

- BFT comenzando en A



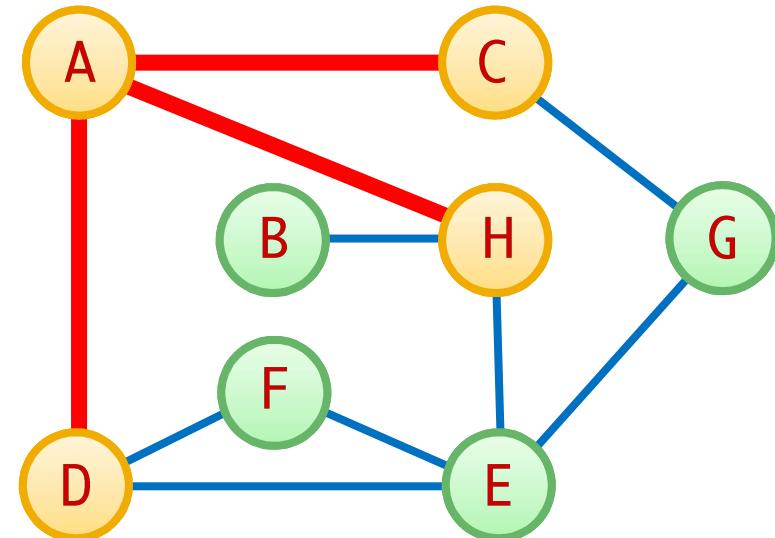
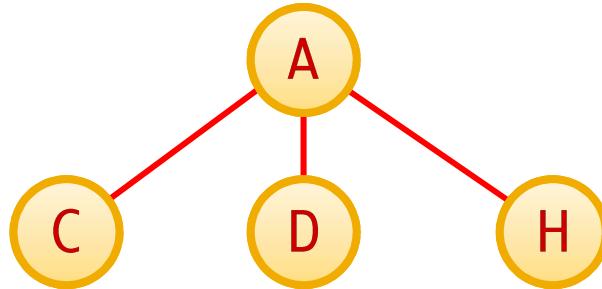
# Recorrido en amplitud ( $II_4$ )

- BFT comenzando en A



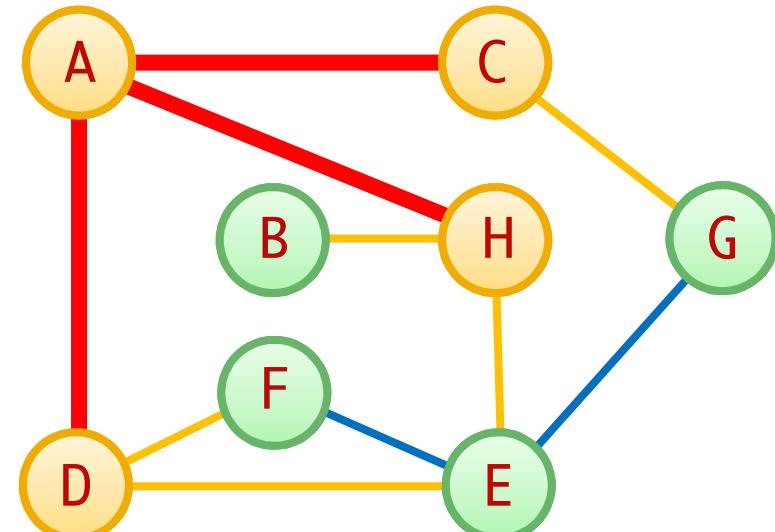
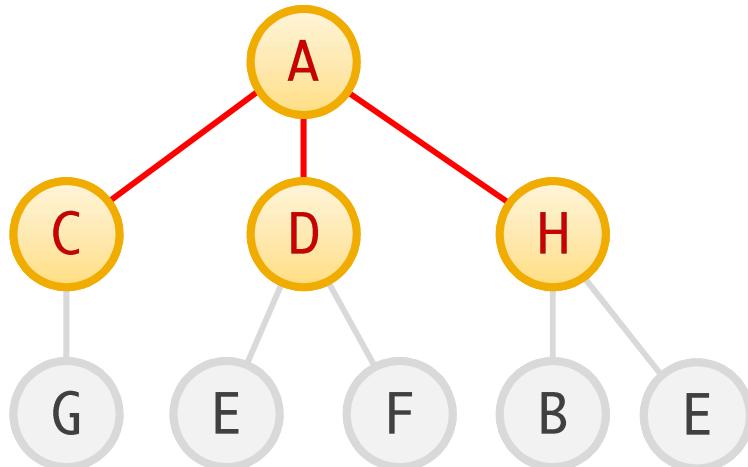
# Recorrido en amplitud ( $II_5$ )

- BFT comenzando en A



# Recorrido en amplitud (II<sub>6</sub>)

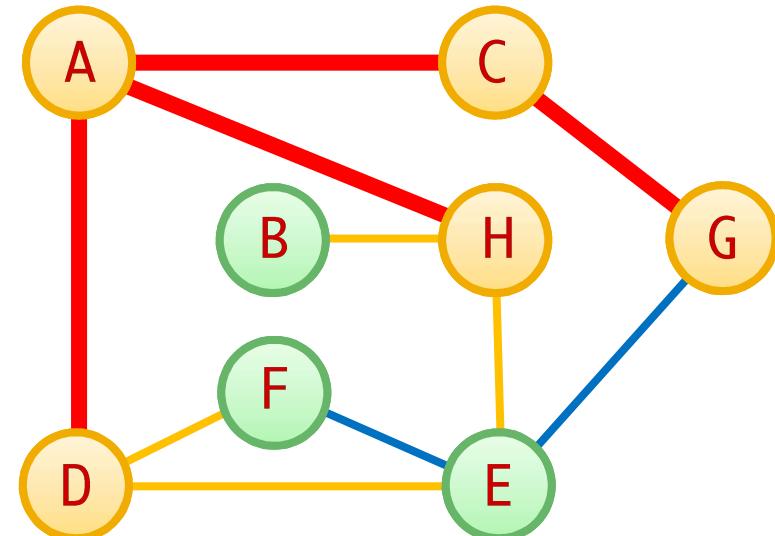
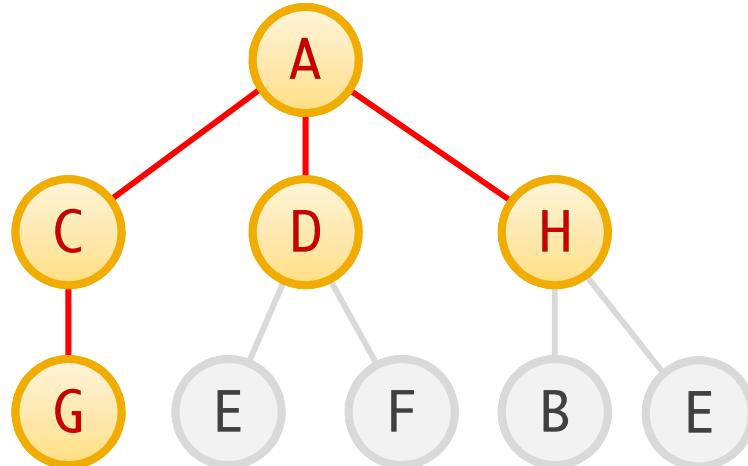
- BFT comenzando en A



Visitar los sucesores no visitados de C, D y H

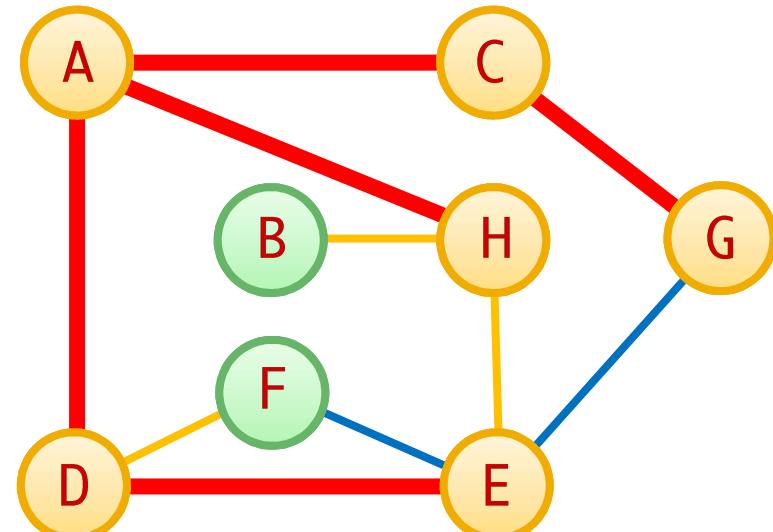
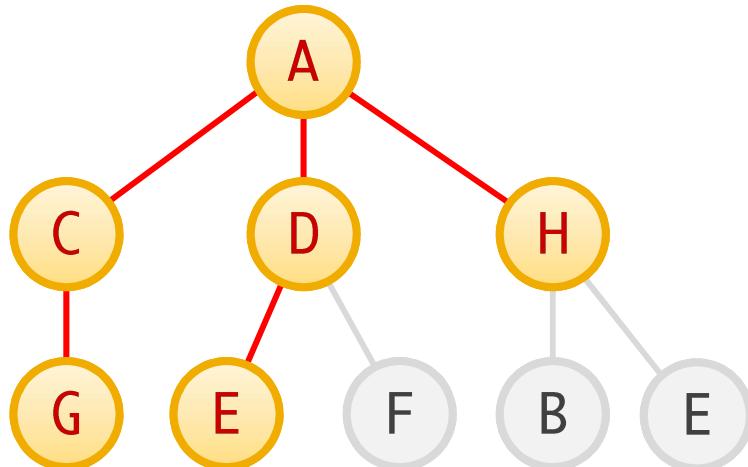
# Recorrido en amplitud (II<sub>7</sub>)

- BFT comenzando en A



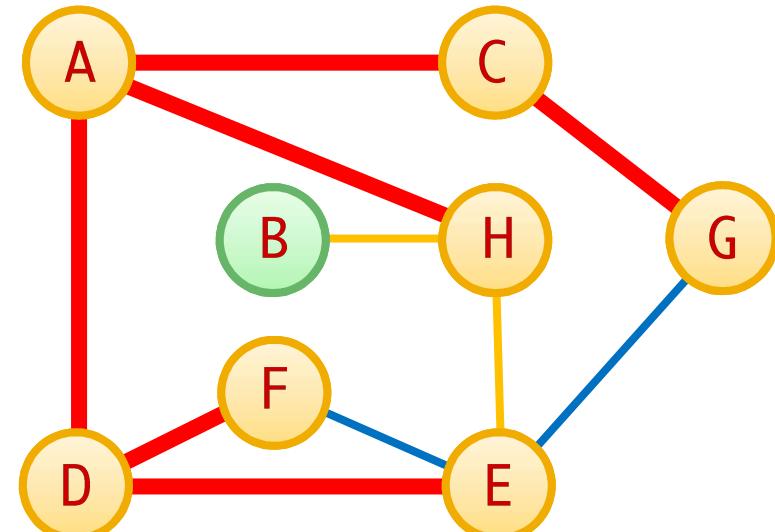
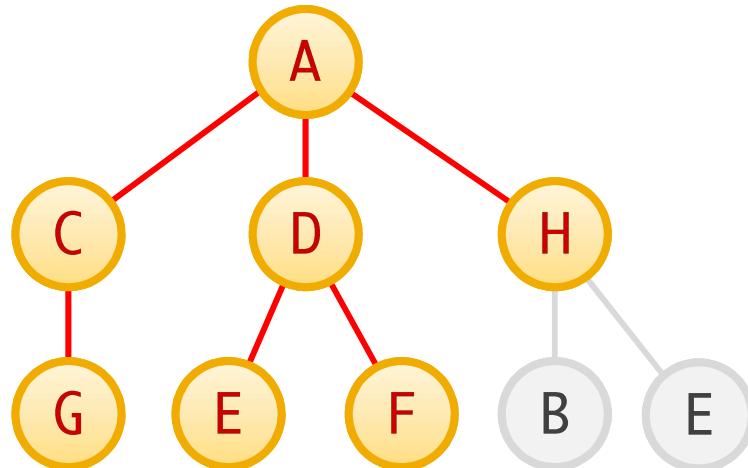
# Recorrido en amplitud ( $II_8$ )

- BFT comenzando en A



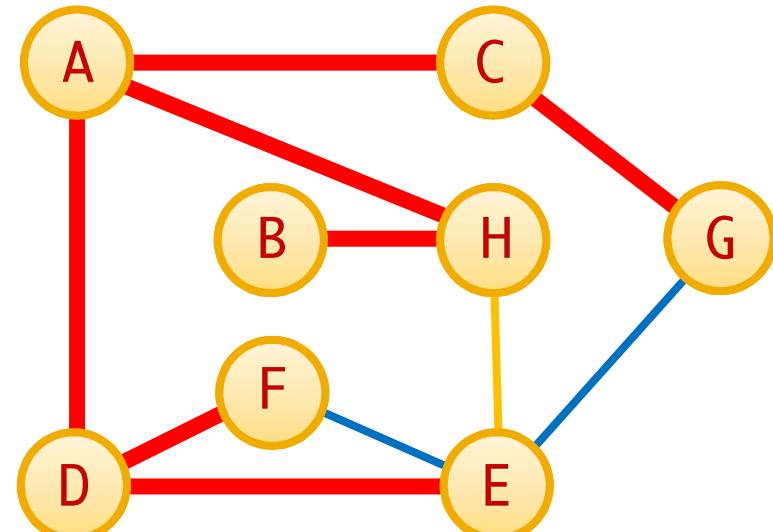
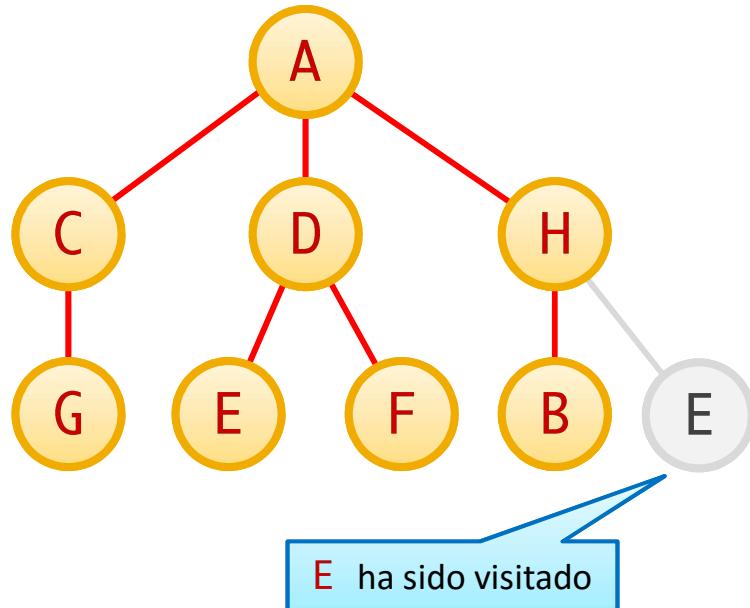
# Recorrido en amplitud (II<sub>9</sub>)

- BFT comenzando en A



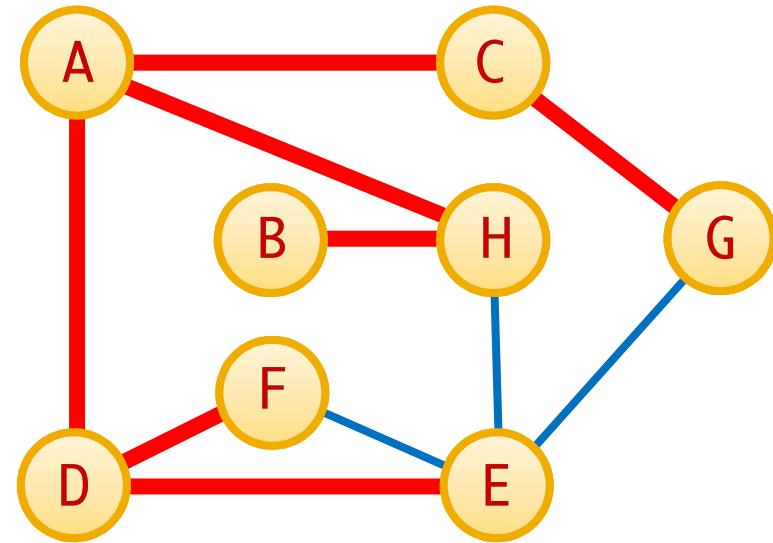
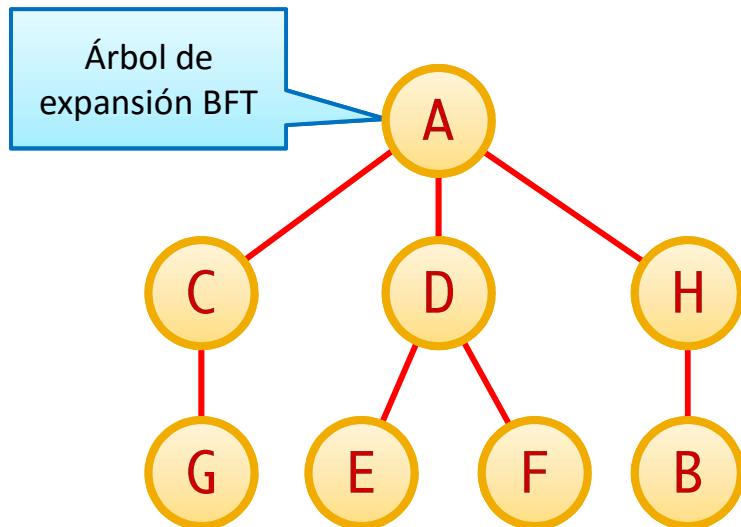
# Recorrido en amplitud ( $\text{II}_{10}$ )

- BFT comenzando en A



# Recorrido en amplitud (II<sub>11</sub>)

- BFT comenzando en A



Todos los vértices han sido visitados. Sin del recorrido BFT

# Breadth First Traversal. Propiedades

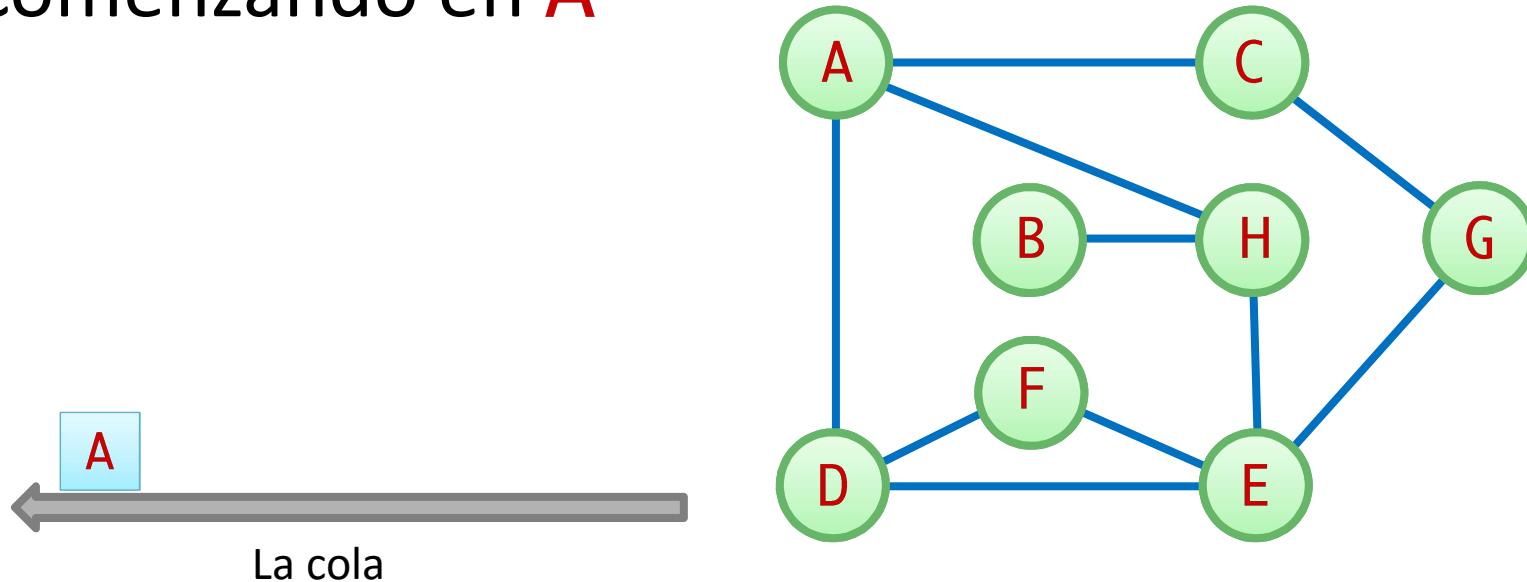
- El recorrido en amplitud BFT ( $v$ ) visita todos los vértices de la componente conexa (CC) de  $v$
- Las aristas recorridas forman el **árbol de búsqueda en amplitud (Breadth First Spanning Tree)** de la CC
- Este árbol contiene los **caminos más cortos** desde  $v$  a cualquier otro nodo de la CC 😊
- BFT es invocada *a lo sumo* una sola vez para cada vértice
- Cada arista es *inspeccionada* en los dos sentidos
- La complejidad de BFT está en  $O(|V| + |E|)$

# BFT: Implementación con una cola

- El algoritmo BFT es el mismo que el DFT ...
- pero, usa una cola en lugar de una pila
  - Inicialmente, encolamos el origen **A**
  - Mientras la cola no es vacía
    - Desencolar el elemento **v**
    - Si **v** no ha sido visitado:
      - Visitar **v** (anotar, marcar, colecciónar, ...)
      - Encolar los sucesores de **v** no visitados

# BFT: Implementación con una cola (II<sub>1</sub>)

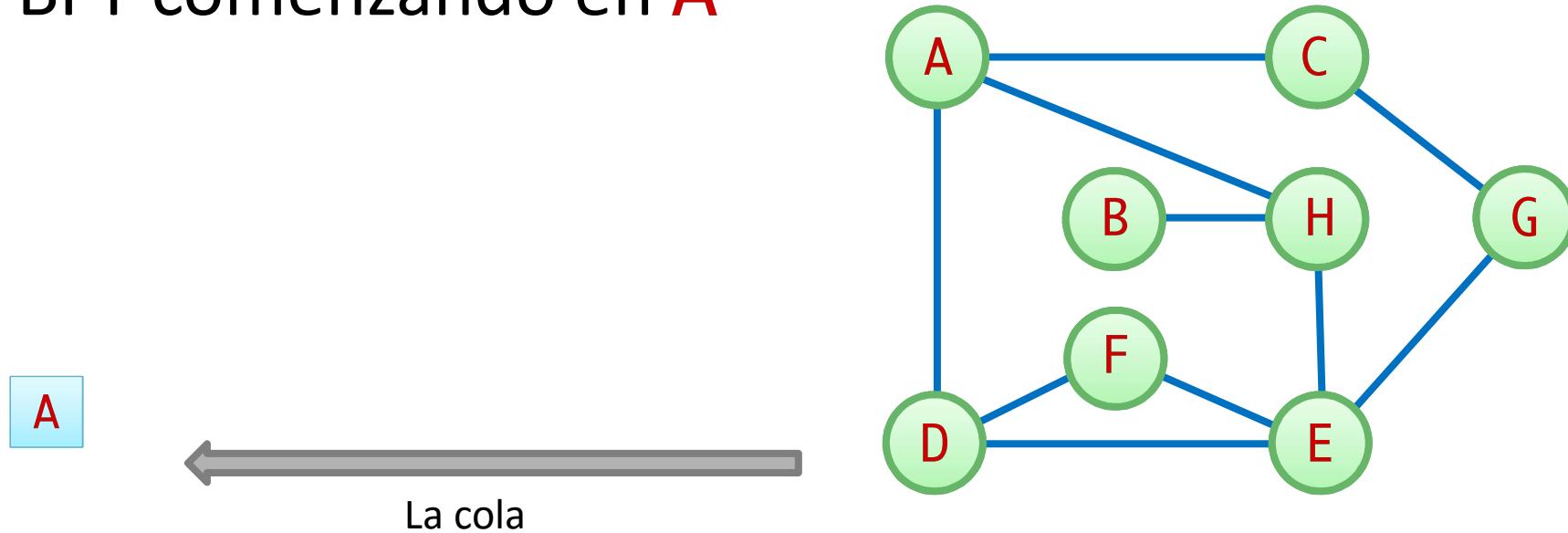
## ■ BFT comenzando en A



- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>2</sub>)

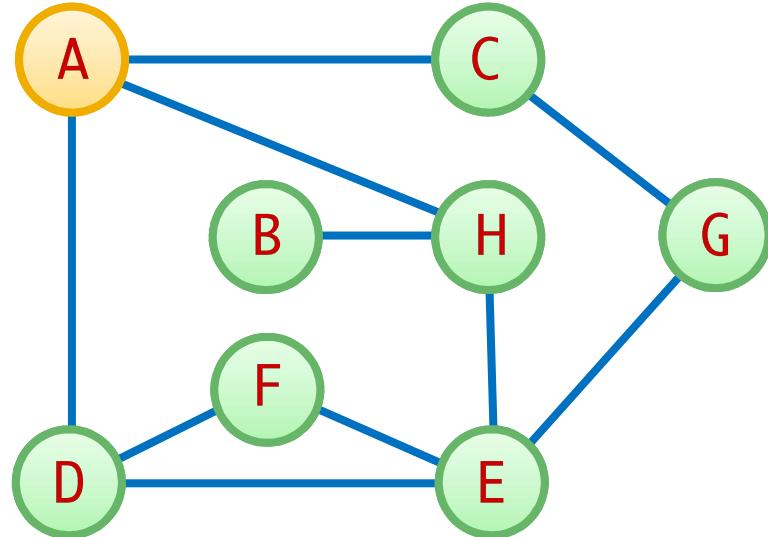
## ■ BFT comenzando en A



- Inicialmente, encolamos el origen  $A$
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>3</sub>)

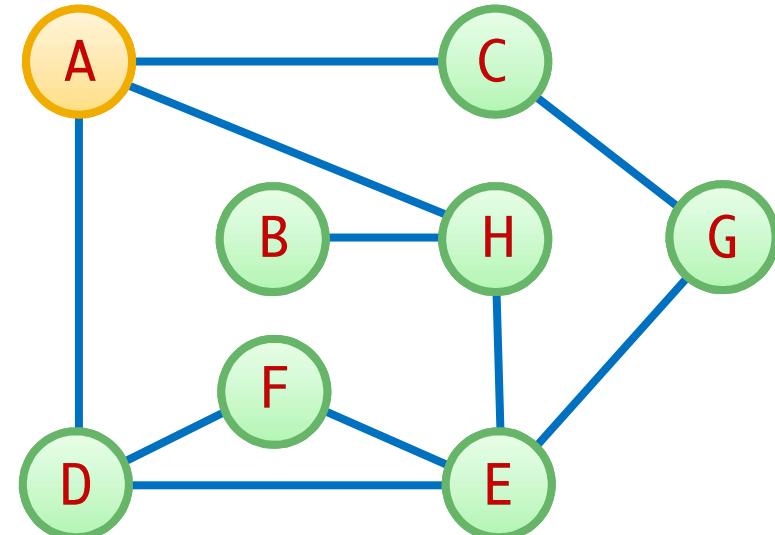
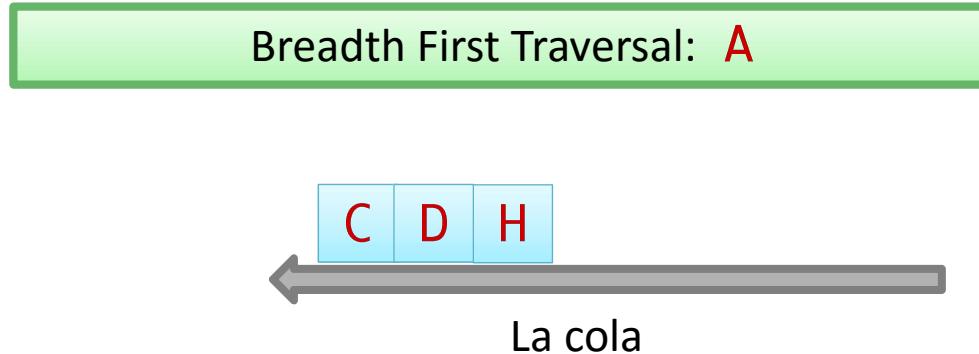
## ■ BFT comenzando en A



- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>4</sub>)

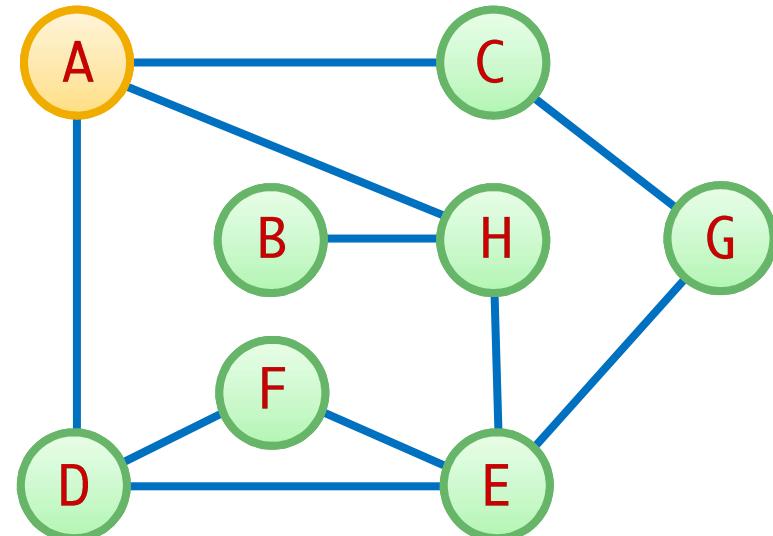
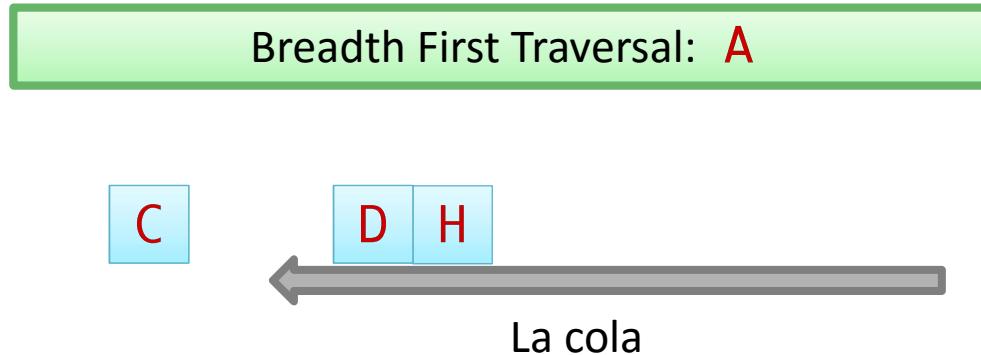
## ■ BFT comenzando en A



- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - **Encolar los sucesores de  $v$  no visitados**

# BFT: Implementación con una cola (II<sub>5</sub>)

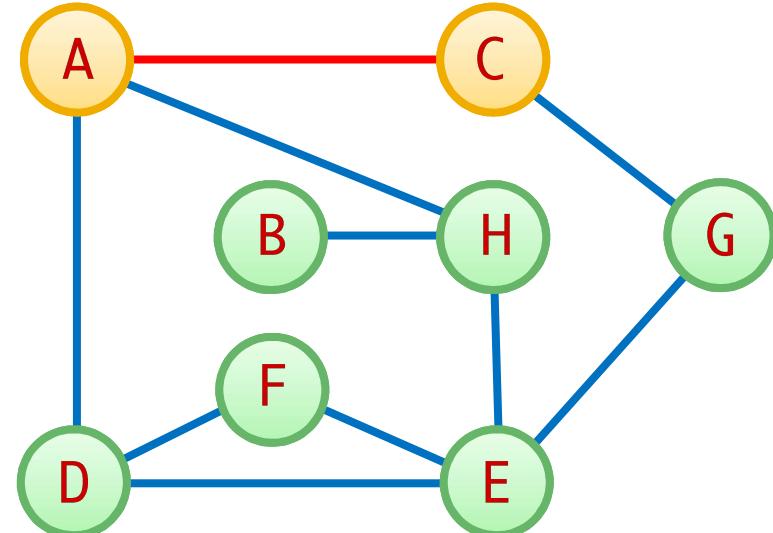
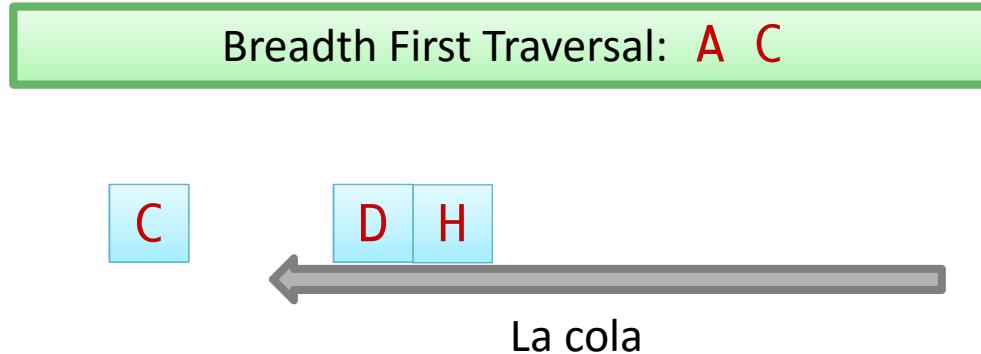
## ■ BFT comenzando en A



- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>6</sub>)

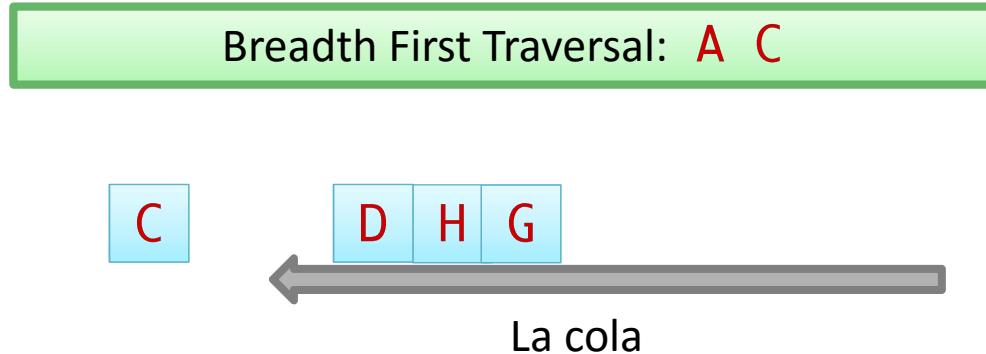
## ■ BFT comenzando en A



- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>7</sub>)

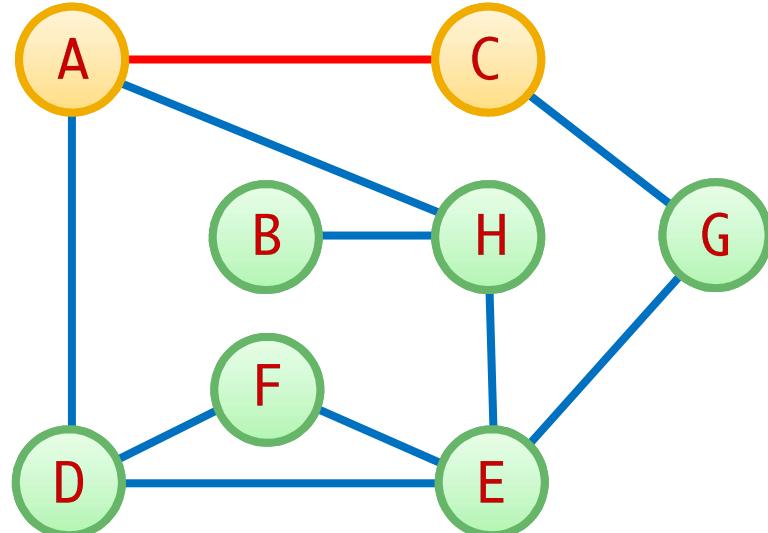
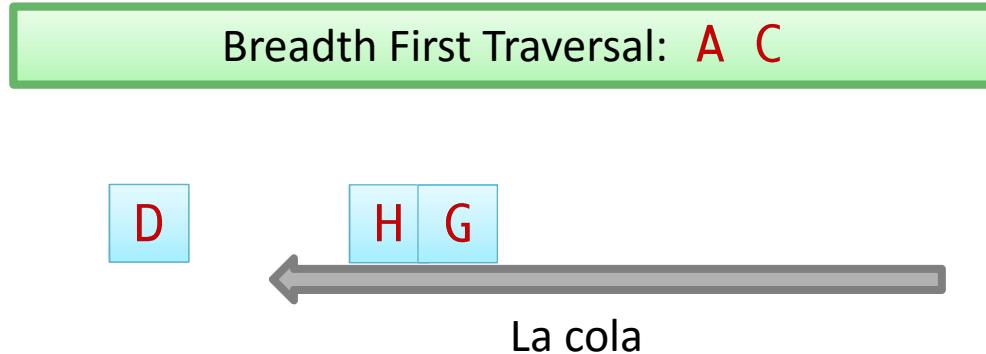
## ■ BFT comenzando en A



- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - **Encolar los sucesores de  $v$  no visitados**

# BFT: Implementación con una cola (II<sub>8</sub>)

## ■ BFT comenzando en A

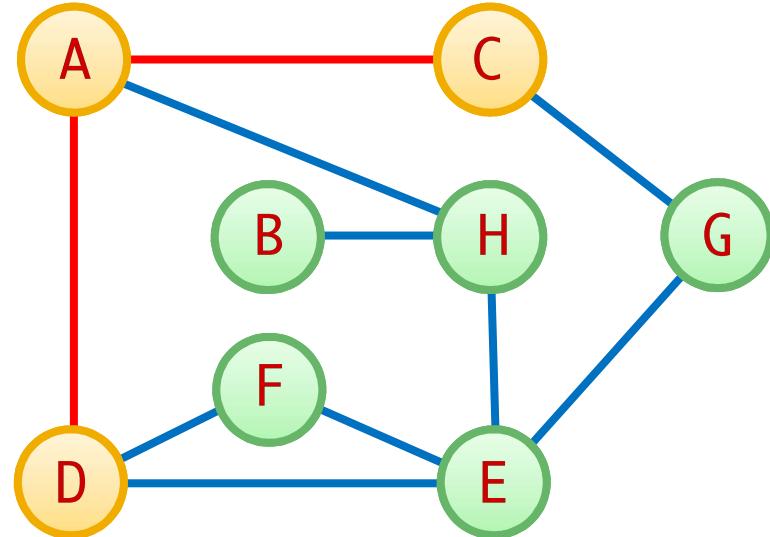


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>9</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D

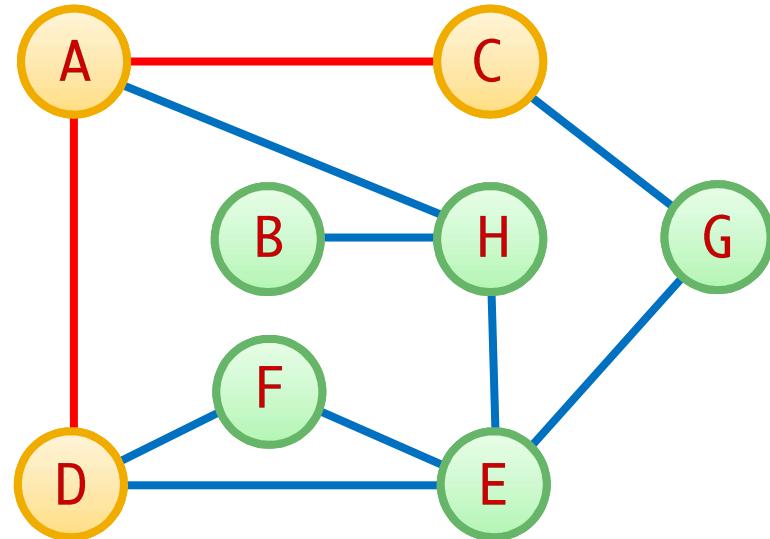


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>10</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D

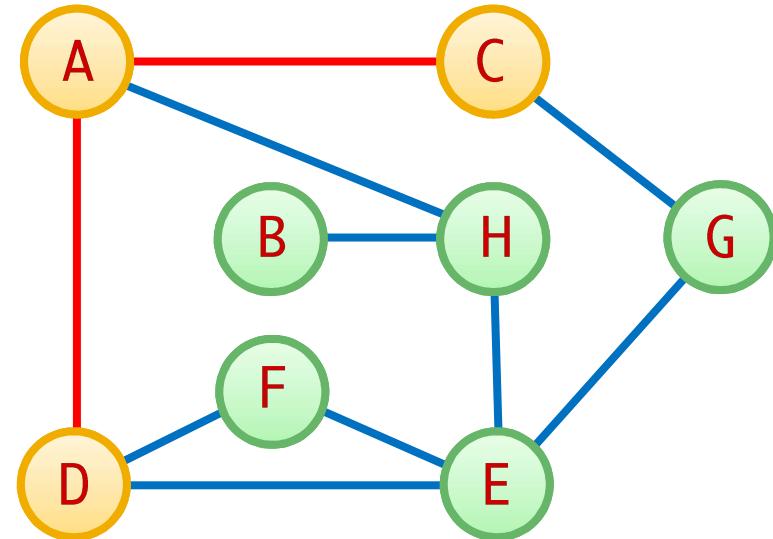


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - **Encolar los sucesores de  $v$  no visitados**

# BFT: Implementación con una cola (II<sub>11</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D

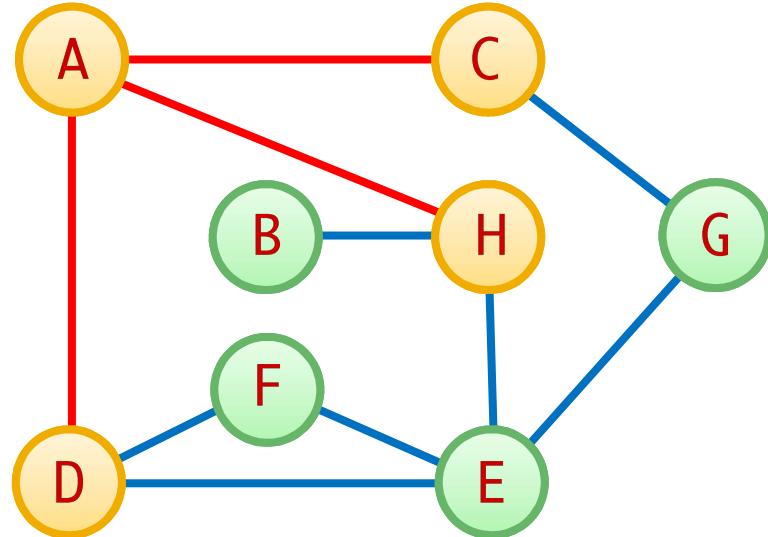


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>12</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H

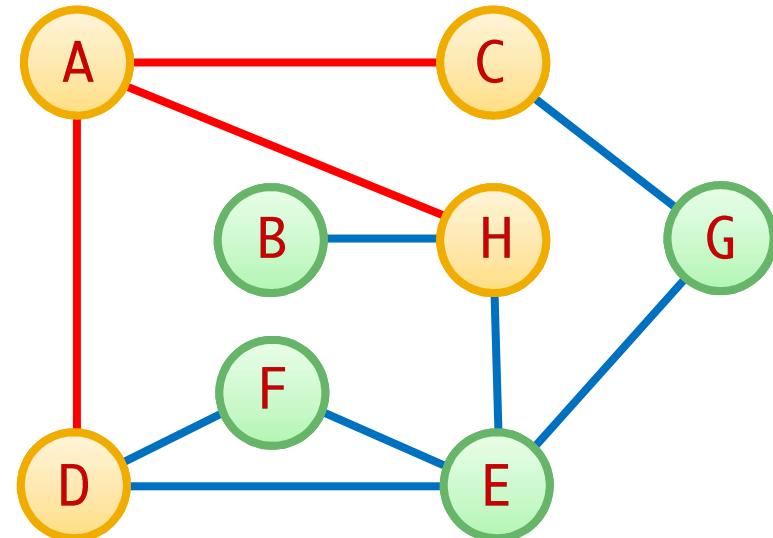
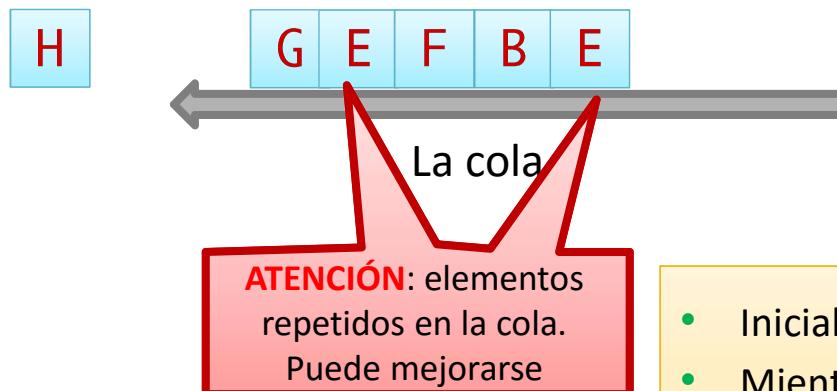


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>13</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H

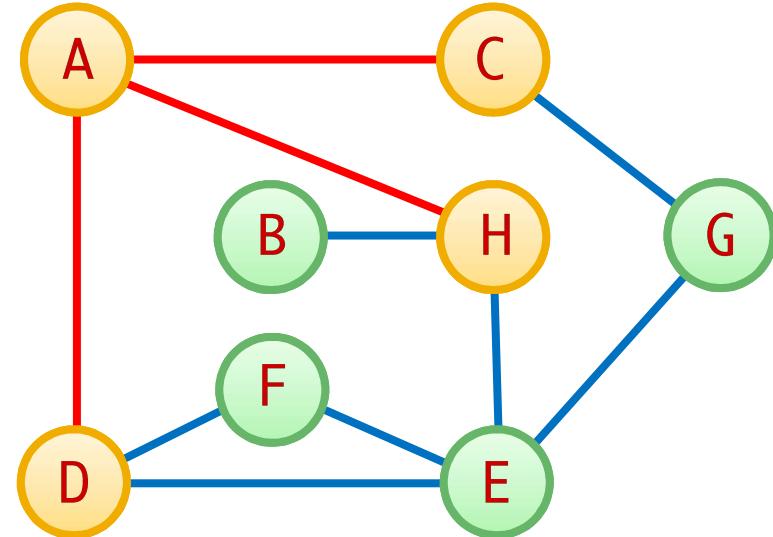


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>14</sub>)

## ■ BFT comenzando en A

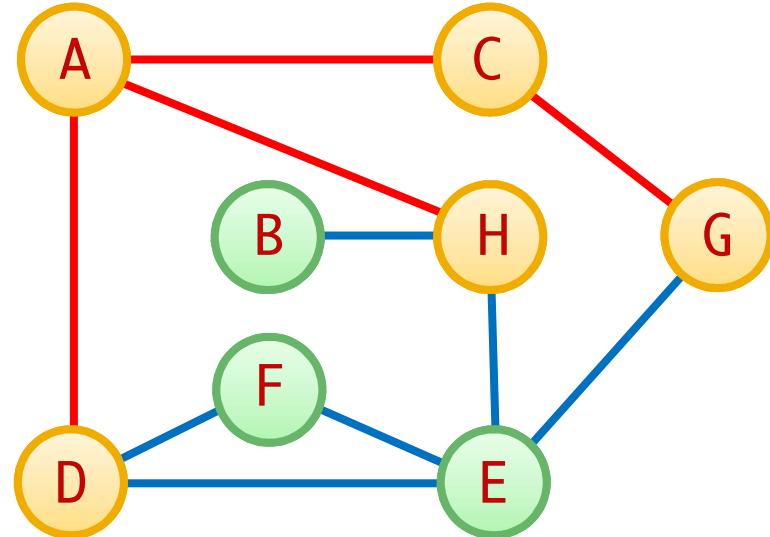
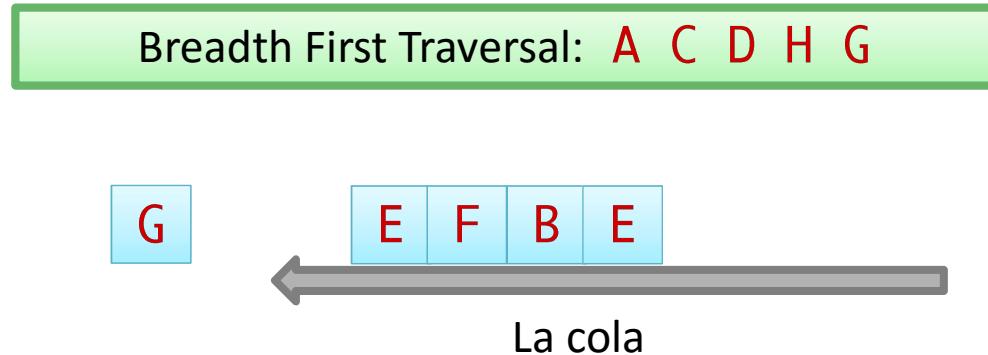
Breadth First Traversal: A C D H



- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>15</sub>)

## ■ BFT comenzando en A

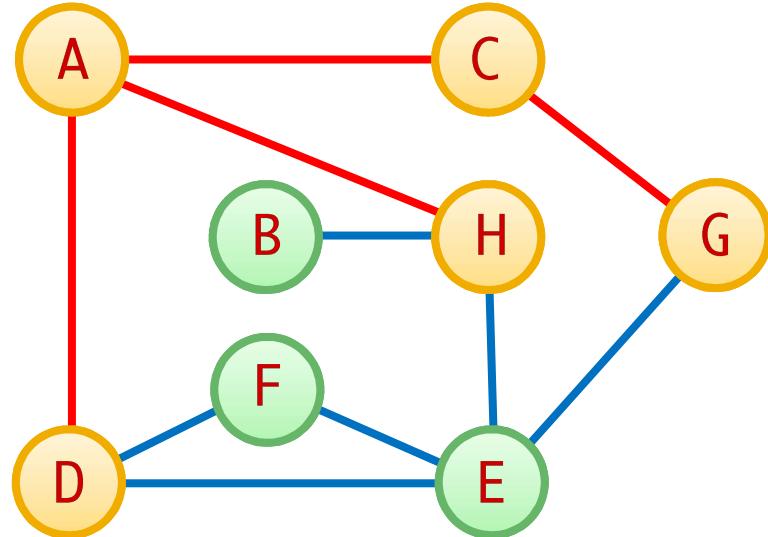


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>16</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G

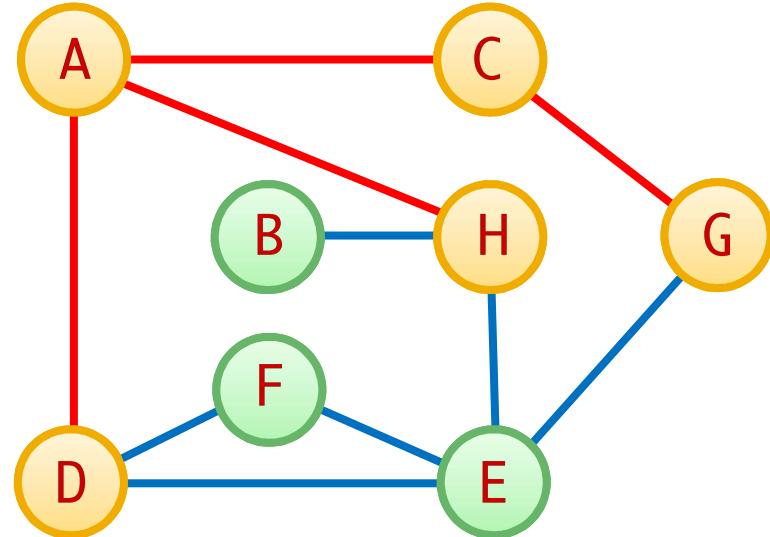


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - **Encolar los sucesores de  $v$  no visitados**

# BFT: Implementación con una cola (II<sub>17</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G

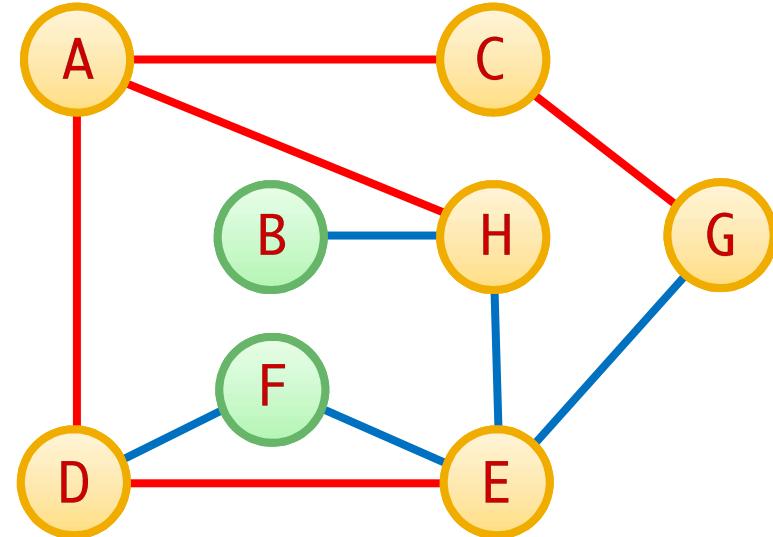


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>18</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E

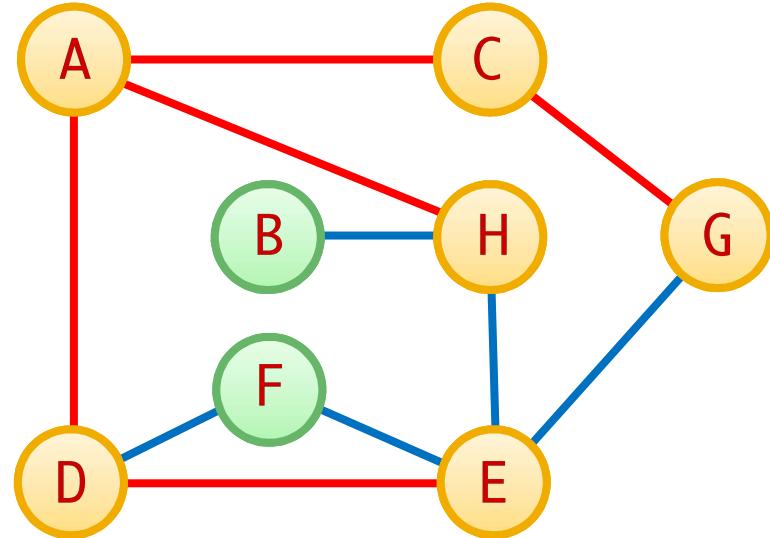


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>19</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E

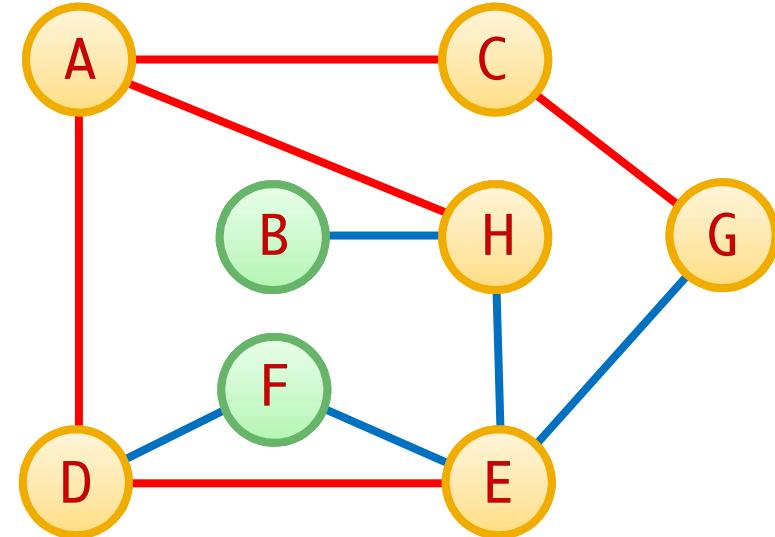


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - **Encolar los sucesores de  $v$  no visitados**

# BFT: Implementación con una cola (II<sub>20</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E

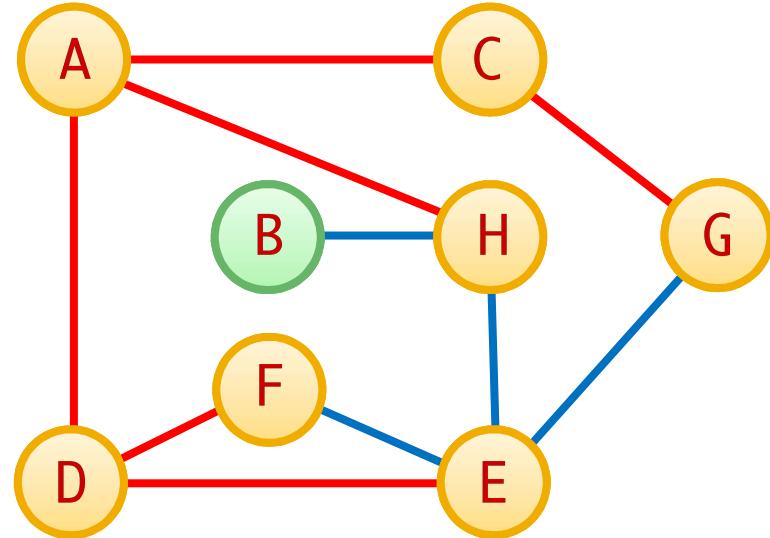


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>21</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F

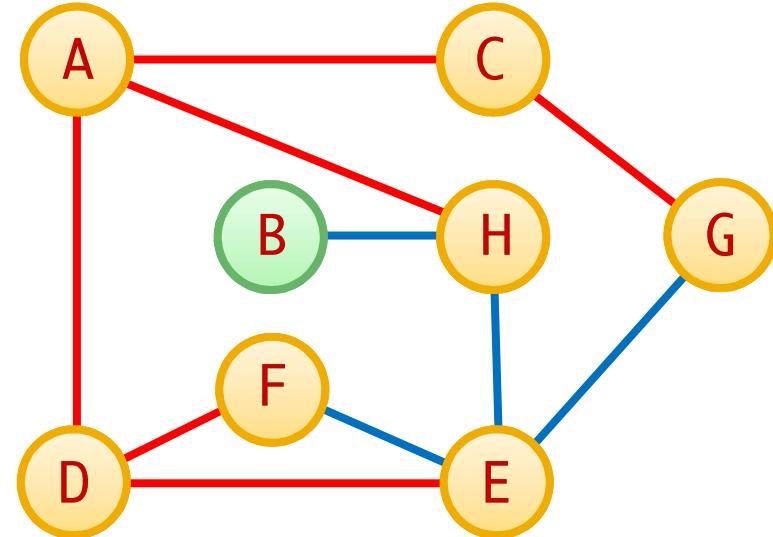


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>22</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F



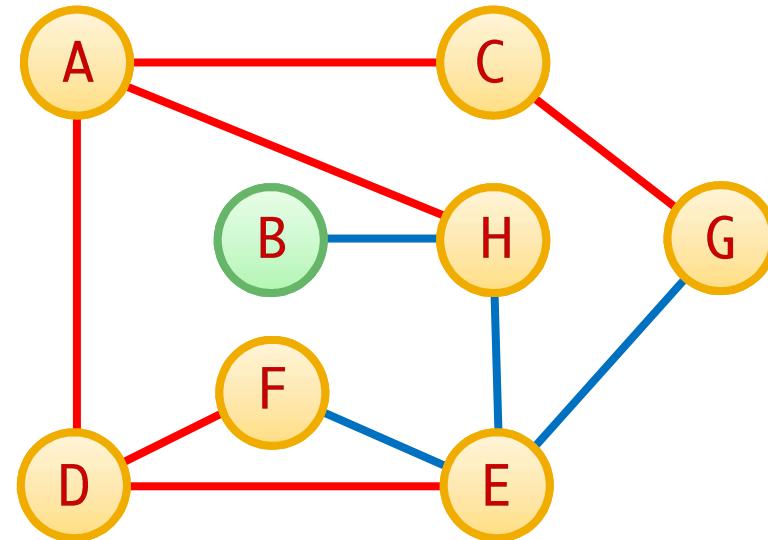
- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

No hay

# BFT: Implementación con una cola (II<sub>23</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F

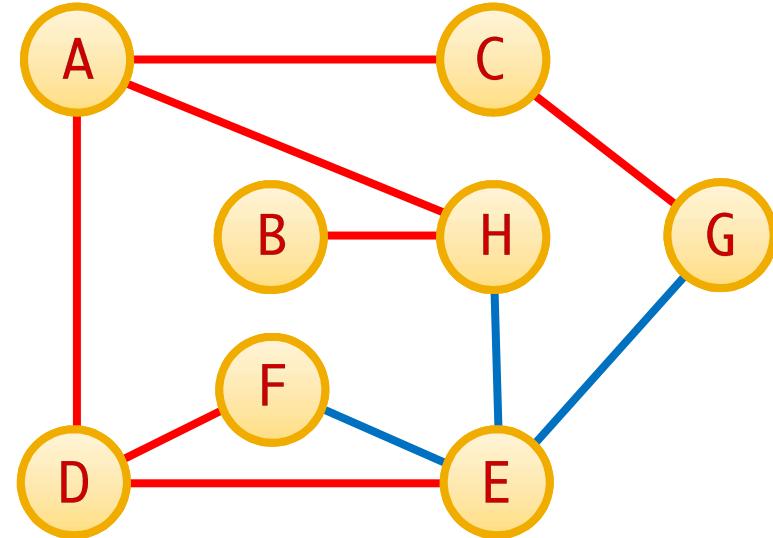


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>24</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F B

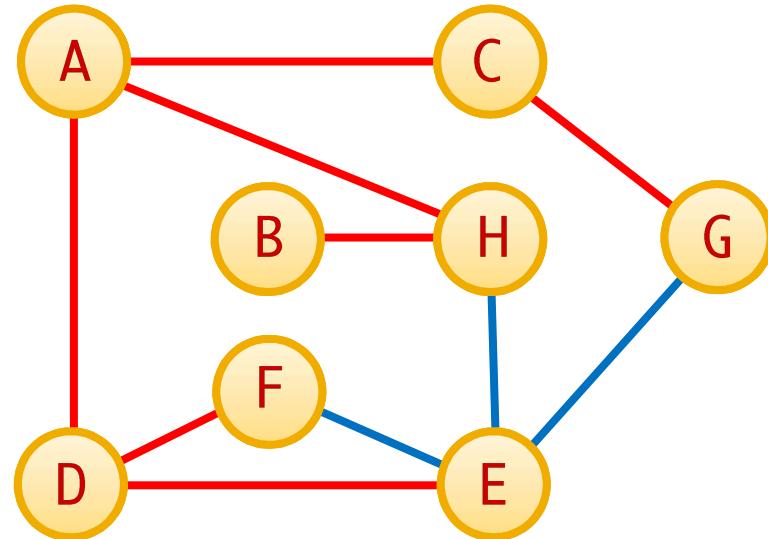


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>25</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F B



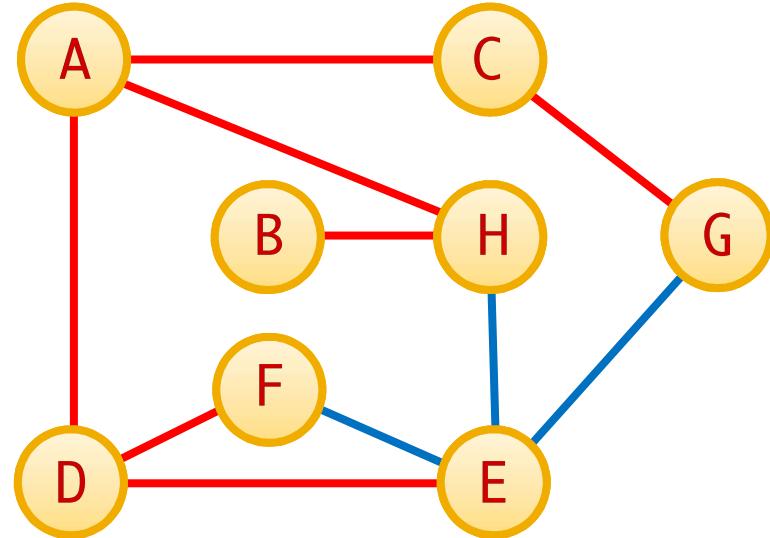
- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

No hay

# BFT: Implementación con una cola (II<sub>26</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F B

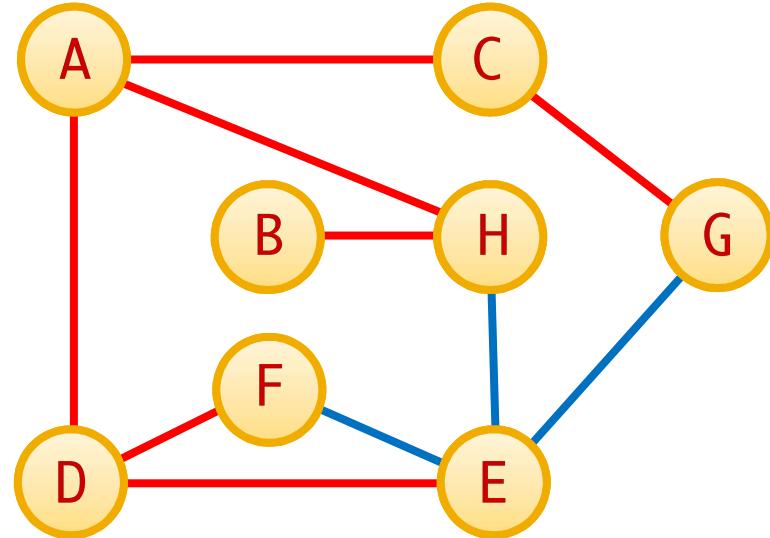
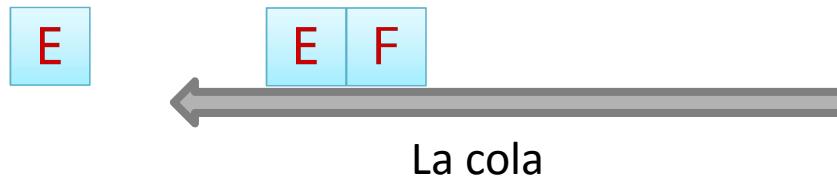


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>27</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F B



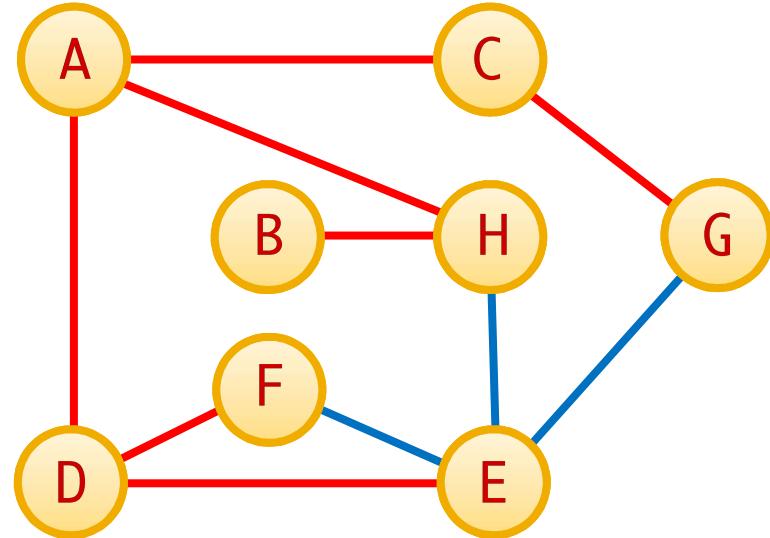
E fue visitado

- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>28</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F B

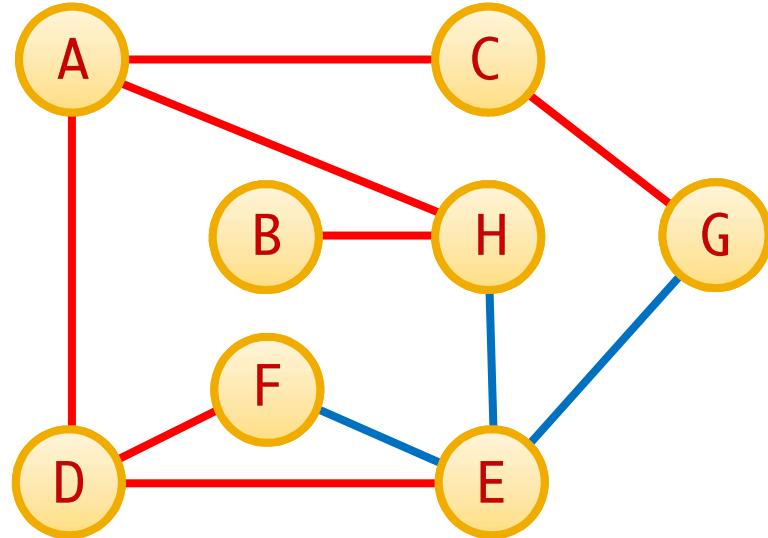


- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>29</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F B



E fue visitado

# BFT: Implementación con una cola (II<sub>30</sub>)

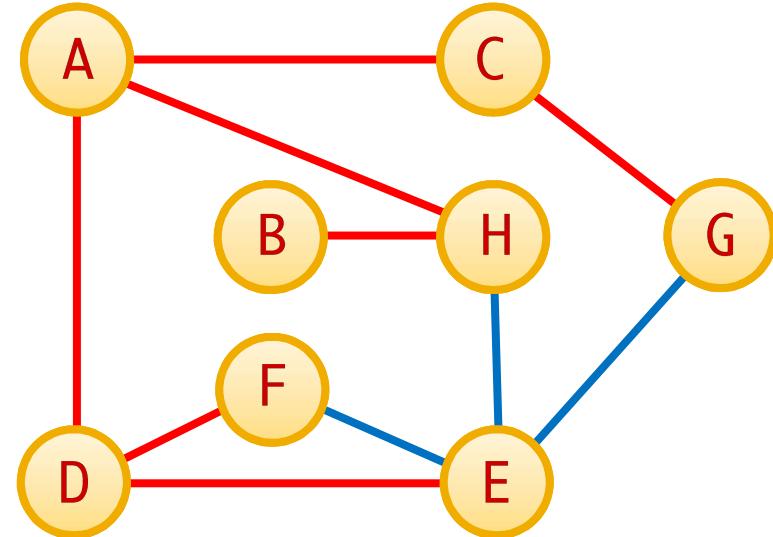
## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F B

F



La cola



- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>31</sub>)

## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F B

F



F fue visitado

- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (II<sub>32</sub>)

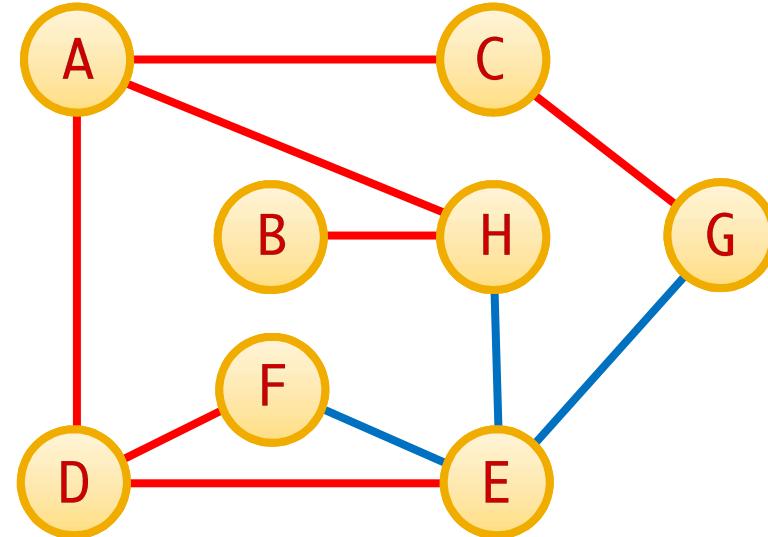
## ■ BFT comenzando en A

Breadth First Traversal: A C D H G E F B



La cola

Cola vacía: fin de  
recorrido



- Inicialmente, encolamos el origen A
- Mientras la cola no es vacía
  - Desencolar el elemento  $v$
  - Si  $v$  no ha sido visitado:
    - Visitar  $v$
    - Encolar los sucesores de  $v$  no visitados

# BFT: Implementación con una cola (III)

- Recordemos la interfaz Cola

**data** Queue a

empty :: Queue a

isEmpty :: Queue a -> Bool

enqueue :: a -> Queue a -> Queue a

first :: Queue a -> a

dequeue :: Queue a -> Queue a

# BFT: Implementación con una cola (IV)

```
import DataStructures.Queue.TwoListsQueue
import qualified DataStructures.Set.BSTSet as S
import DataStructures.Graph.Graph

enqueueAll :: Queue a -> [a] -> Queue a
enqueueAll s xs = foldr enqueue s xs

bft :: (Ord a) => Graph a -> a -> [a]
bft g v0 = aux S.empty (enqueue v0 empty)
where
    aux visited queue
        | isEmpty queue      = []
        | v `S.isElem` visited = aux visited' -- v fue visitado
        | otherwise            = v : aux visited' (enqueueAll queue' us)
    where
        v = first queue
        queue' = dequeue queue
        visited' = S.insert v visited
        us = [ u | u <- successors g v, u `S.notIsElem` visited ]

```

Encola en s los elementos de la lista xs

Devuelve la lista de vértices del recorrido BFT comenzando en v0

conjunto de visitados inicialmente vacío, y cola con el vértice inicial

Visitamos v

Sucesores de v no visitados

# BFT: Caminos a vértices

```
import import DataStructures.Queue.TwoListsQueue
import qualified DataStructures.Set.BSTSet as S
import DataStructures.Graph.Graph
import qualified DataStructures.Dictionary.BSTDictionary as D

data DiEdge a = a :> a -- w :> v vamos a v desde w

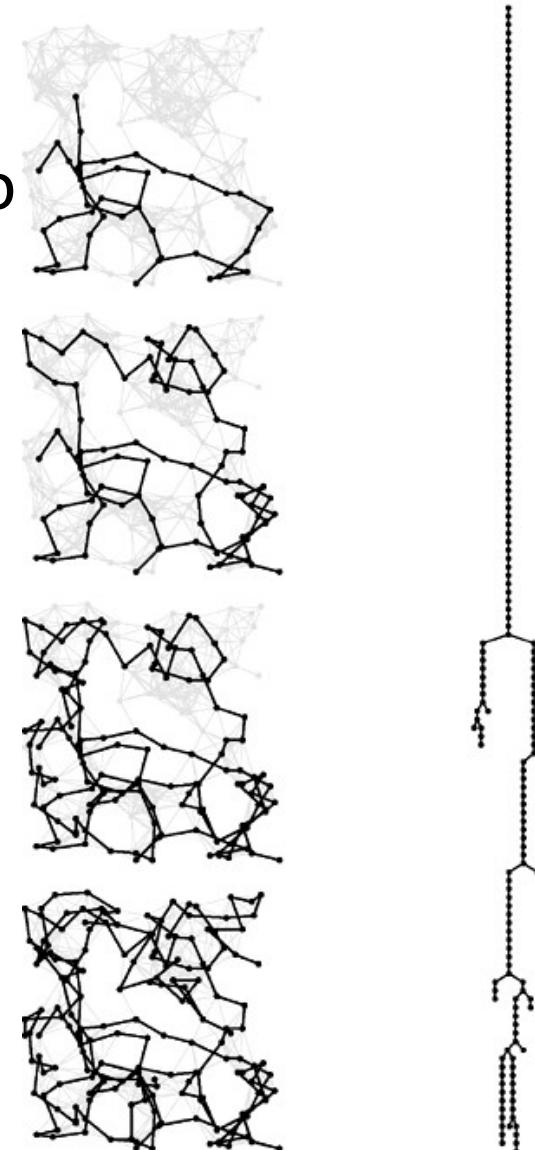
bftPaths :: (Ord a) => Graph a -> a -> [Path a]
bftPaths g v0 = aux S.empty (enqueue (v0 :> v0) empty) D.empty
  where
    aux visited queue dict
      | isEmpty queue = [] -- fin de recorrido
      | v `S.isElem` visited = aux visited queue' dict -- v fue visitado
      | otherwise =
          pathFromTo v0 v dict' :-- v fue visitado: devolvemos un camino
          aux visited' (enqueueAll queue' es) dict'
    where
      w :> v = first queue
      queue' = dequeue queue
      visited' = S.insert v visited
      dict' = D.insert v w dict -- parent of v is w
      es = [ v :> u | u <- successors g v, u `S.notIsElem` visited ]
```

Sucesores de v no visitados

# DFT *versus* BFT

- Depth First Traversal
  - El árbol de expansión es muy alto

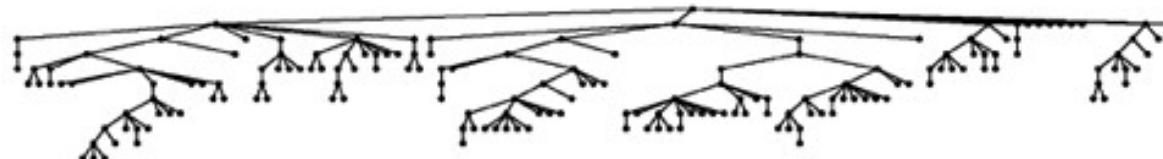
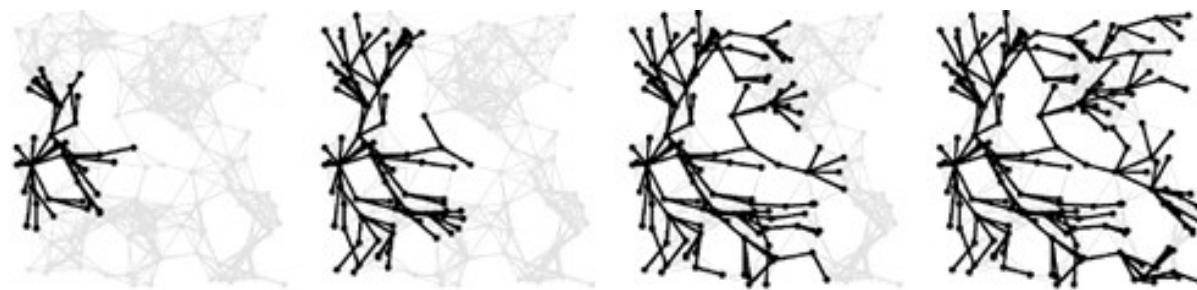
Evolución del  
recorrido DFT



# DFT versus BFT (II)

- Breadth First Traversal
  - El árbol de expansión es corto y ancho

Evolución del  
recorrido BFT

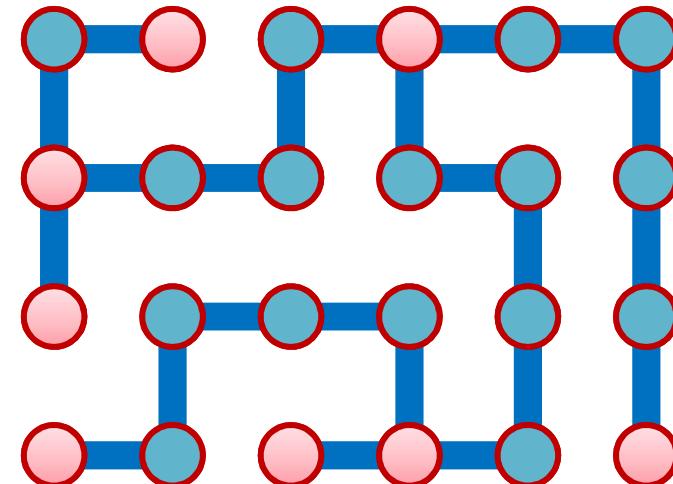
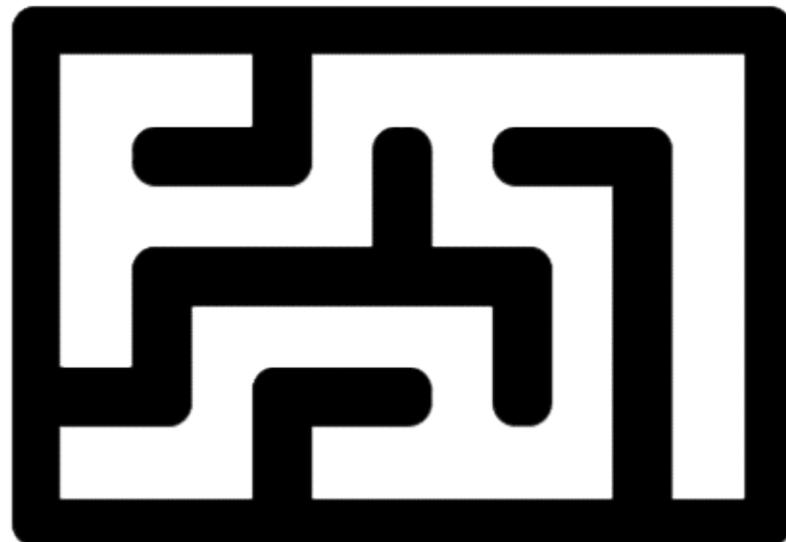


# DFT y BFT: Aplicaciones

- ¿Existe un camino desde  $v$  hasta  $w$  en el Grafo  $g$ ?
  - $w \in \text{dft } g v$        $w \in \text{bft } g v$
- Encontrar un camino desde  $v$  hasta  $w$  en el Grafo  $g$ 
  - Usar  $\text{dftPaths } g v$  o  $\text{bftPathsTo } g v$
- Encontrar el camino más corto (menor número de aristas) desde  $v$  hasta  $w$  en el Grafo  $g$ 
  - Usar  $\text{bftPaths } g v$
- Encontrar el camino más largo que conecta dos vértices
  - No se conocen algoritmos eficientes ☹

# DFT y BFT: Aplicaciones (II)

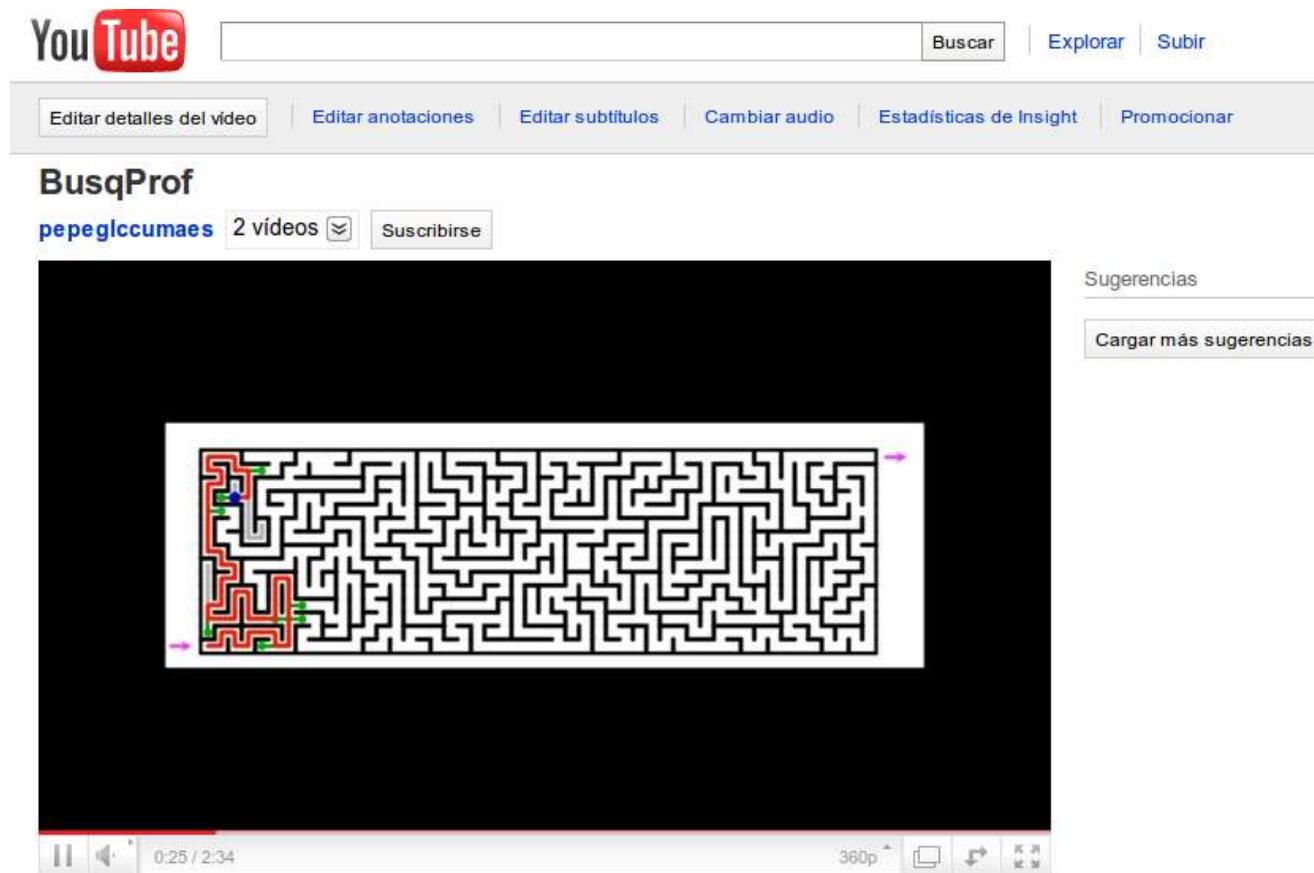
- Salida de un laberinto
- Un laberinto se modela como un grafo



# Saliendo de un laberinto

## Depth First Traversal

- Video <http://www.youtube.com/watch?v=AKgo5I5eYAg>



# Saliendo de un laberinto

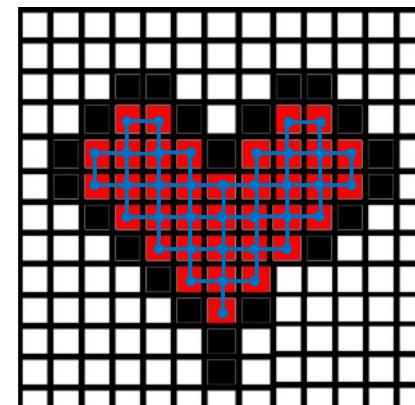
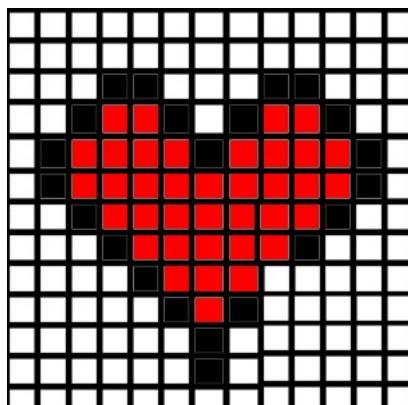
## Breadth First Traversal

- Video <http://www.youtube.com/watch?v=tDtMj9wWtEk>



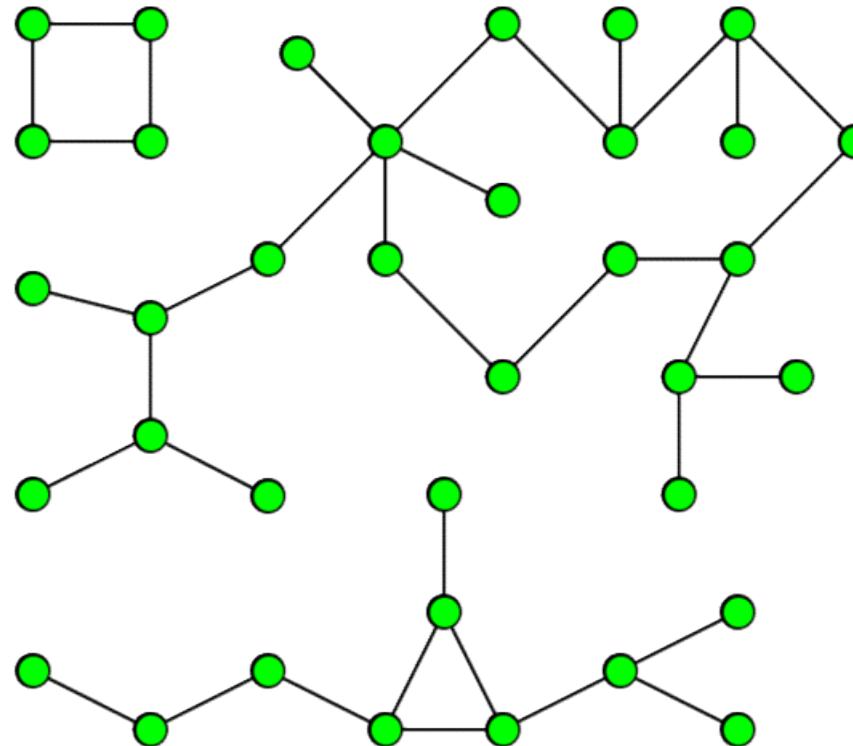
# DFT y BFT: Aplicaciones (III)

- **Conectividad:** Encontrar todos los vértices conectados a  $v$  (componente conexa de  $v$  en el Grafo  $g$ )
  - Los vértices de la lista `dft g v`
- **Coloreado de una región**
  - Cambiar el color de una región: recorrer un grafo cambiando el color de los vértices visitados, donde:
    - Vértice: pixel. Arco: dos píxeles adyacentes.
    - Región conexa: píxeles **conectados** a un vértice de la región.



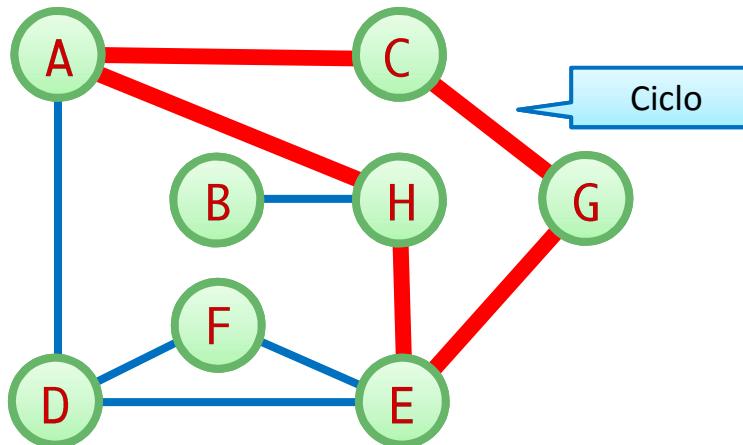
# DFT y BFT: Aplicaciones (IV)

- Encontrar las **componentes conexas** de un grafo:
  - Usamos dft reiteradamente (ver transparencia 228)



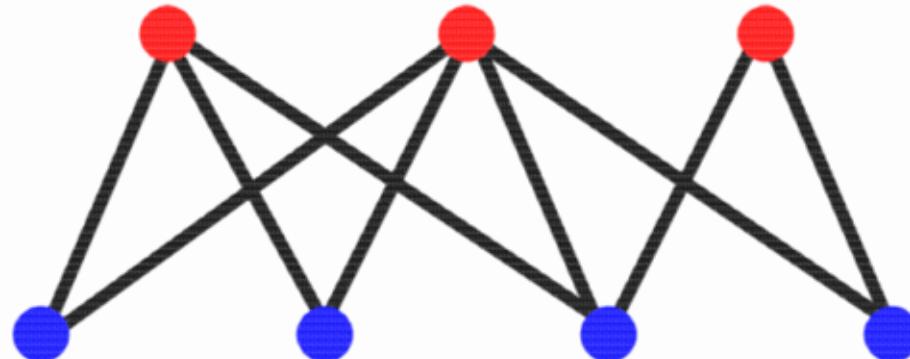
# DFT y BFT: Aplicaciones (IV)

- Detección de **ciclos** : extraer un ciclo de un grafo
  - Vía DFT( $v$ ) localizamos un camino que termine en un vértice conectado a  $v$ .



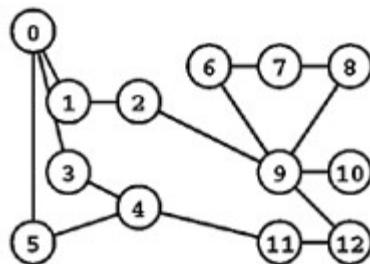
# Grafo 2-Coloreable o Bipartito

- ¿Podemos colorear los vértices de forma que dos vértices adyacentes tengan distinto color? ¿Podemos usar dos colores?
- **Grafo bipartito:** existe una partición del conjunto de vértices en dos grupos, de forma que no existan arcos entre los grupos
- Equivalentemente: no contiene ciclos de longitud impar

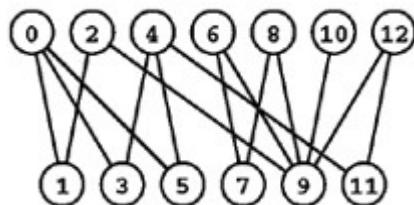


# Grafo 2-Coloreable o Bipartito (II)

- Vértice: una persona
- Arista: dos personas que se gustan



- Gran Hermano: repartir los participantes en temporadas
- En cada temporada no debe haber dos candidatos que se gusten en la casa

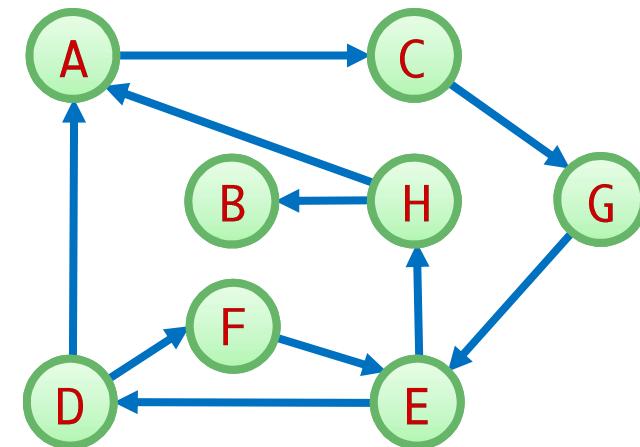


Si es necesario repartir en 3 o más temporadas, no se conocen algoritmos eficientes 😞

# Grafos Dirigidos

- También conocidos como diGrafos
- Las aristas son flechas en lugar de arcos

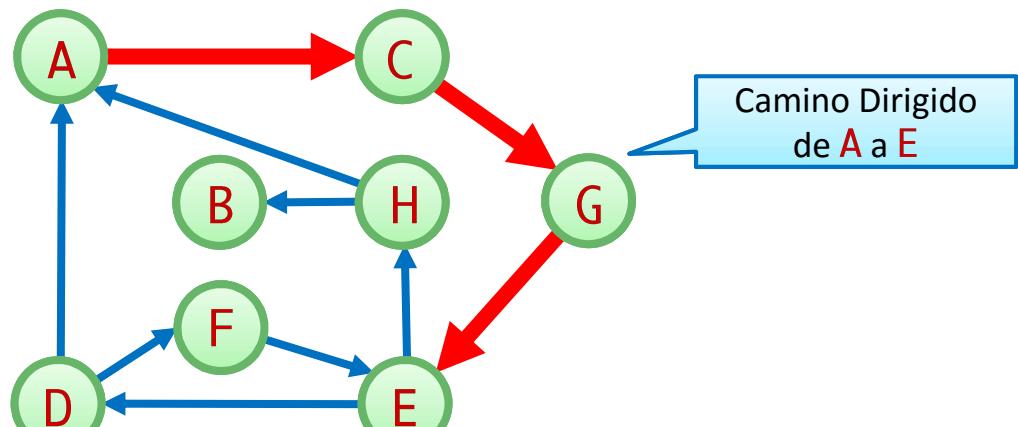
- En un arco  $v \rightarrow w$ 
  - $v$  es el **fuente**
  - $w$  es el **destino**
  - $w$  es **sucesor** de  $v$
  - $v$  es **predecesor** de  $w$



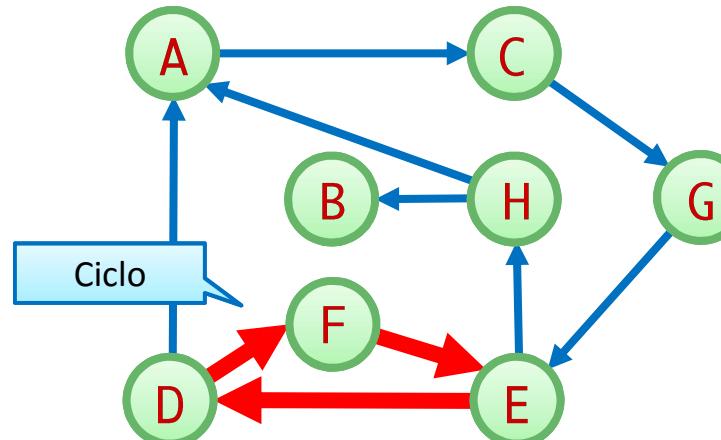
- Grado de salida de  $v$  (**out degree**): nº de arcos que parten de  $v$
- Grado de entrada  $v$  (**in degree**): nº de arcos que llegan a  $v$

# Grafos Dirigidos (II)

- **Camino Dirigido**: secuencia de vértices conectados por flechas



- **Ciclo Dirigido**: camino dirigido tal que el primer y el último vértice son el mismo



# Construyendo DiGrafos en Haskell

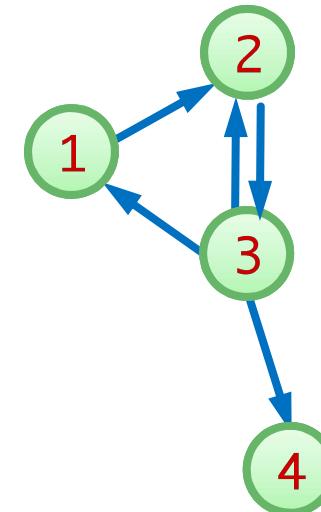
- Proporcionando vértices y la función sucesor:

```
import DataStructures.Graph.DiGraph
```

```
g1 :: DiGraph Int
g1 = mkDiGraphSuc [1,2,3,4] suc
where
    suc 1 = [2]
    suc 2 = [3]
    suc 3 = [1,2,4]
    suc 4 = []
```

lista de los vértices del grafo

función que retorna los vértices sucesores



- Proporcionando vértices y arcos:

```
g1' :: DiGraph Int
g1' = mkDiGraphEdges [1,2,3,4] [ 1 :> 2
                                    , 2 :> 3
                                    , 3 :> 1, 3 :> 2, 3 :> 4 ]
```

lista de los arcos del grafo

lista con todos los vértices del grafo

# Implementando DiGrafos en Haskell

```
module DataStructures.Graph.DiGraph
( DiGraph
, DiEdge((:->))
, Path
, mkDiGraphSuc
, mkDiGraphEdges
, successors
, predecesors
, vertices
, diEdges
, outDegree
, inDegree
, deleteVertices
) where
```

# Implementando DiGrafos en Haskell (II)

```
data DiGraph a = DG [a] (a -> [a])
```

La implementación contiene la lista de vértices y la función sucesor

```
mkDiGraphSuc :: [a] -> (a -> [a]) -> DiGraph a  
mkDiGraphSuc vs suc = DG vs suc
```

Trivial

```
data DiEdge a = a :-> a deriving Show
```

```
mkDiGraphEdges :: (Eq a) => [a] -> [DiEdge a] -> DiGraph a  
mkDiGraphEdges vs es = DG vs suc
```

where

```
suc v = [ y | x :-> y <- es, x==v ]
```

Se define la función sucesor a partir de la lista de arcos

# Implementando DiGrafos en Haskell (III)

```
successors :: DiGraph a -> a -> [a]  
successors (DG vs suc) v = suc v
```

```
predecesors :: (Eq a) => DiGraph a -> a -> [a]  
predecesors (DG vs suc) v =  
  [ w | w <- vs, v `elem` suc w ]
```

```
vertices :: DiGraph a -> [a]  
vertices (DG vs suc) = vs
```

```
outDegree :: DiGraph a -> a -> Int  
outDegree g v = length (successors g v)
```

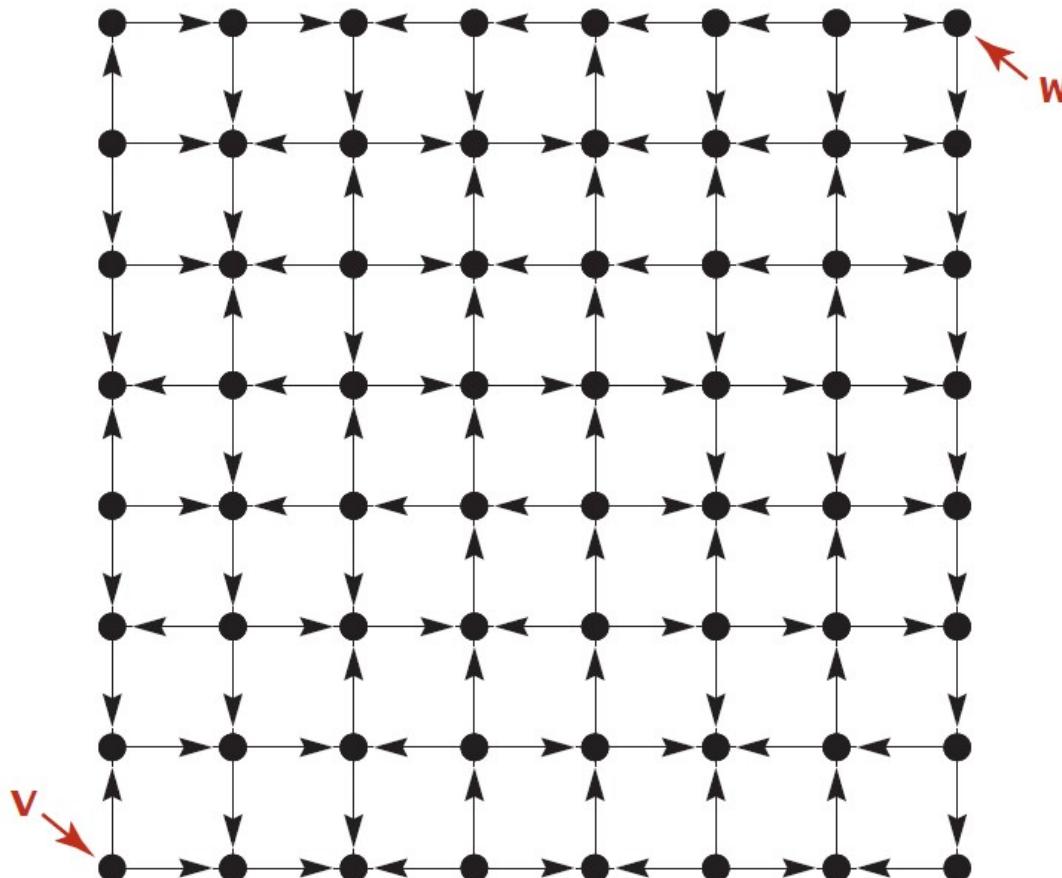
```
inDegree :: (Eq a) => DiGraph a -> a -> Int  
inDegree g v = length (predecesors g v)
```

# Recorridos en Digrafos

- Los mismos algoritmos para recorrido en profundidad (Depth First) y recorrido en anchura (Breadth First) de Grafos pueden ser usados con Digrafos
- Solo se necesita especificar la función sucesor apropiadamente
- Ver los módulos DiGraphDFT y DiGraphBFT

# Alcanzabilidad en DiGrafos

- ¿Es  $w$  alcanzable desde  $v$  en este DiGrafo?

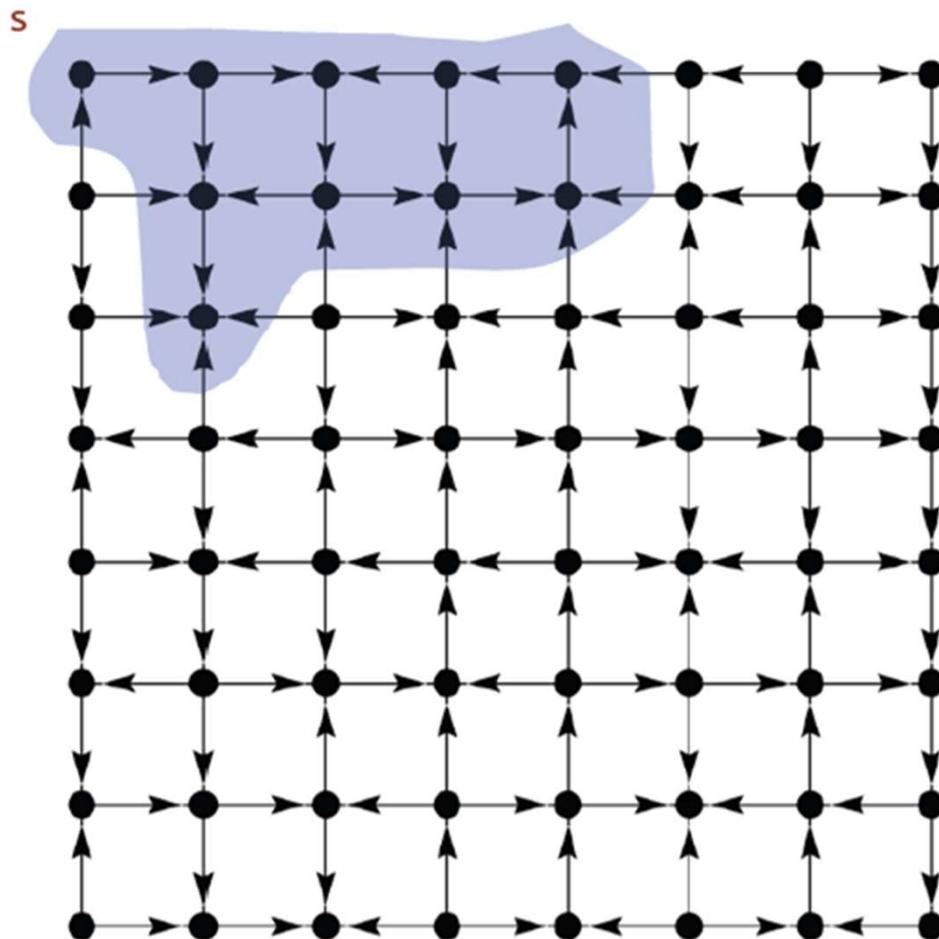


Un camino dirigido  
es difícil de  
vislumbrar 😞

Usar DFT o BFT  
desde  $v$  😊

# Alcanzabilidad en DiGrafos

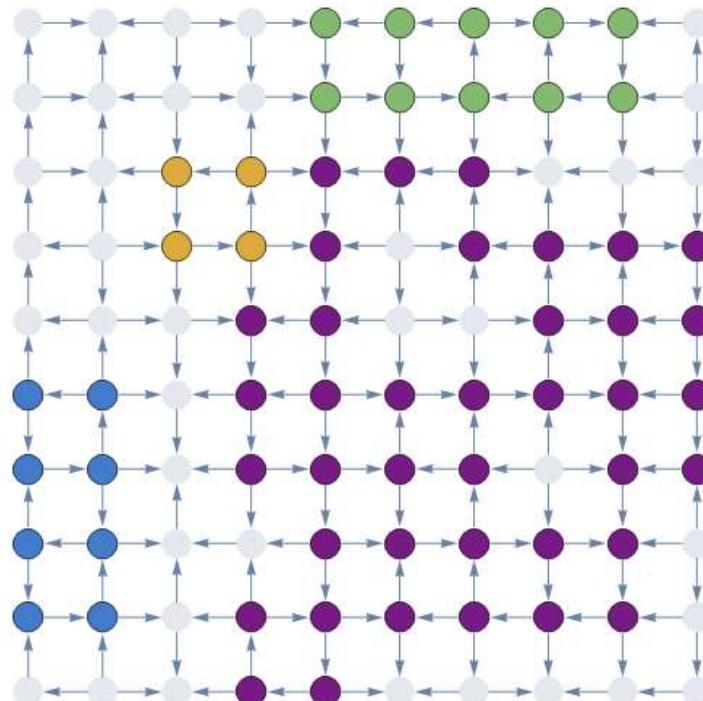
- Encontrar todos los vértices alcanzables desde  $s$  a lo largo de un camino dirigido



Usar DFT or BFT  
desde s 😊

# Componentes Fuertemente Conexas

- Dos vértices se dice que están **Fuertemente Conectados** si el primero es alcanzable desde el segundo y el segundo es alcanzable desde el primero
- **Componentes Fuertemente Conexas**: máximo conjunto de vértices de manera que todos están fuertemente conectados



El algoritmo de Kosaraju usa DFT

# Representación clásica de Grafos

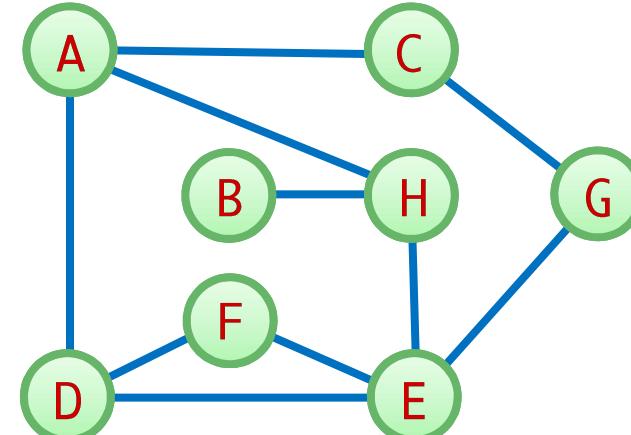
## Matriz de adyacencias

### ■ Matriz de adyacencias para un Grafo

	A	B	C	D	E	F	G	H
A			X	X				X
B								X
C	X							X
D	X				X	X		
E			X		X	X	X	X
F				X	X			
G			X		X			
H	X	X			X			

La matriz es simétrica

Hay una arista de B a H



Hay una arista de H a B

# Representación clásica de Grafos

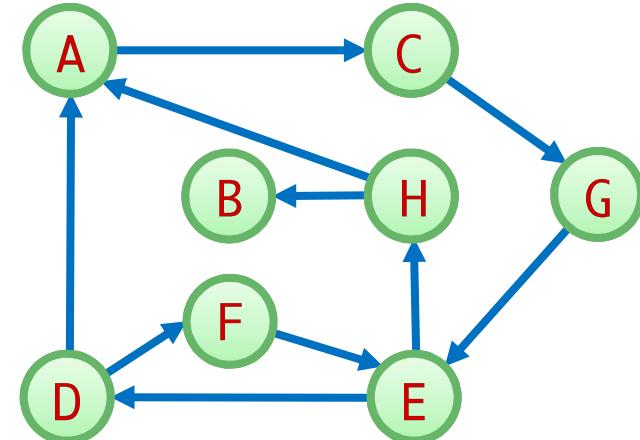
## Matriz de adyacencias (II)

- Matriz de adyacencias para un DiGrafo

La matriz **NO** es necesariamente simétrica

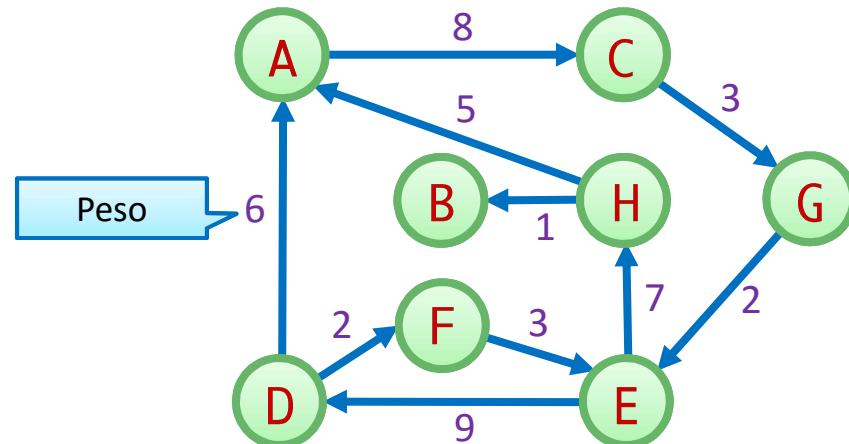
	A	B	C	D	E	F	G	H
A			X					
B								
C						X		
D	X					X		
E			X					X
F				X				
G					X			
H	X	X						

Hay un arco de C a G



# (Di)Grafos con pesos

- Existe un valor (**peso**) asociado a cada arista (**arco**)
- Es usado para expresar **costes** (distancia, dinero, ...) para ir de la fuente al destino

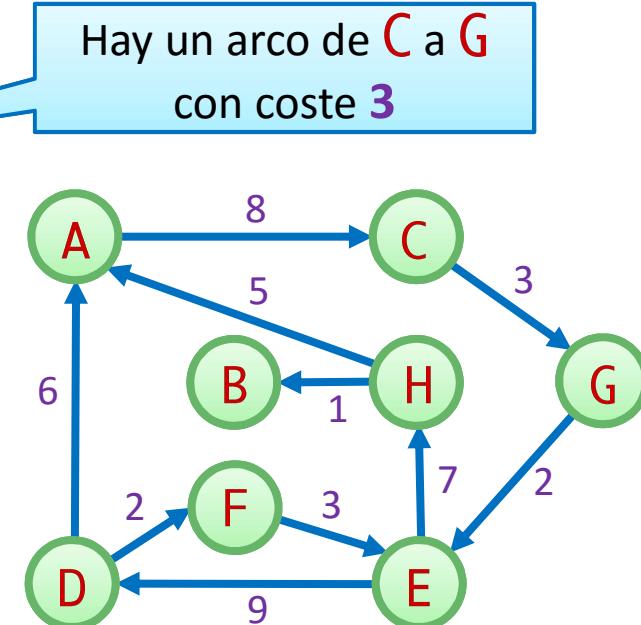


# Representación clásica de Grafos

## Matriz de adyacencias (III)

- Matriz de adyacencia para un DiGrafo con pesos

	A	B	C	D	E	F	G	H
A			8					
B								
C						3		
D	6				2			
E			9				7	
F					3			
G					2			
H	5	1						



# Representación clásica de Grafos

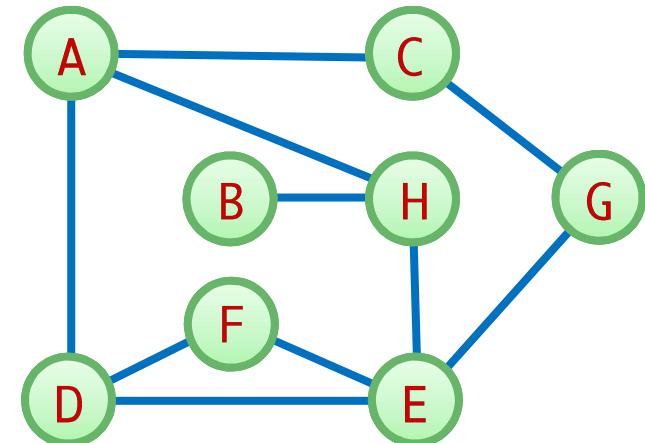
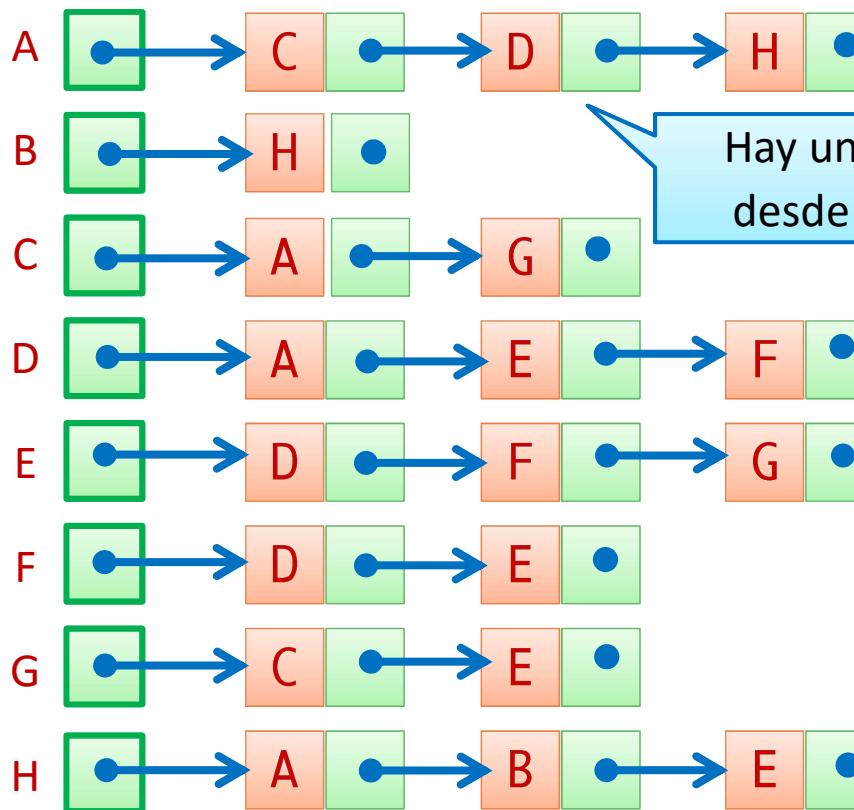
## Matriz de adyacencias (IV)

- Buen uso de la memoria para grafos **densos** 😊
- Para grafos **dispersos** el uso de memoria es ineficiente 😞
- La mayoría de los lenguajes de programación solo permiten enteros como índice de arrays:
  - Además de la matriz, se necesita un diccionario de vértices a enteros
- $O(1)$  para determinar si el arco  $v \rightarrow w$  está en el grafo 😊
  -   $|V|$  es el número de vértices en el grafo
- $O(|V|)$  para devolver todos los sucesores de un vértice 😞

# Representación clásica de Grafos

## Listas de adyacencias

### ■ Lista de adyacencias para un Grafo

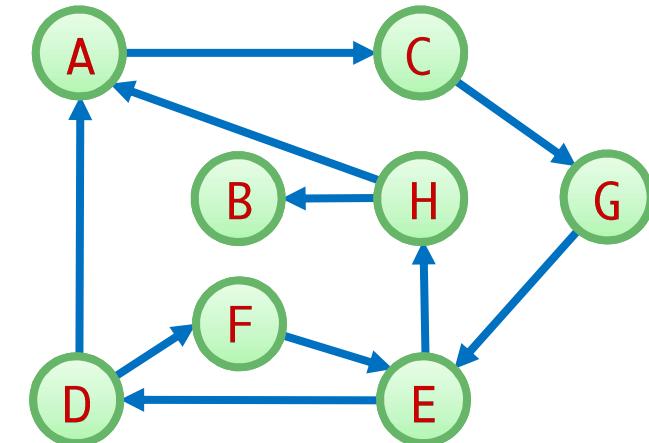
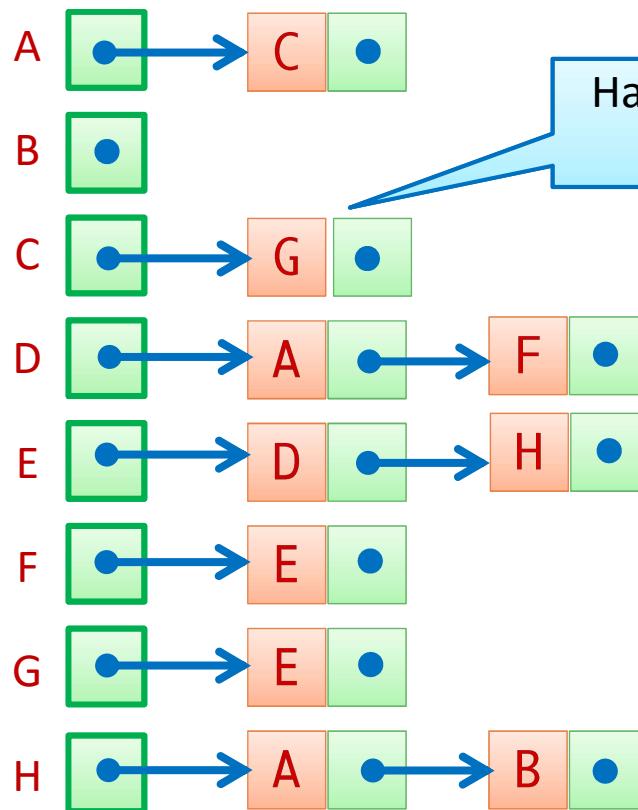


Una arista es representada por dos arcos.  
Para la arista  $u - v$ :  
•  $v$  está en la lista asociada a  $u$  y  
•  $u$  está en la lista asociada a  $v$

# Representación clásica de Grafos

## Listas de adyacencias (II)

### ■ Lista de adyacencias para un DiGrafo



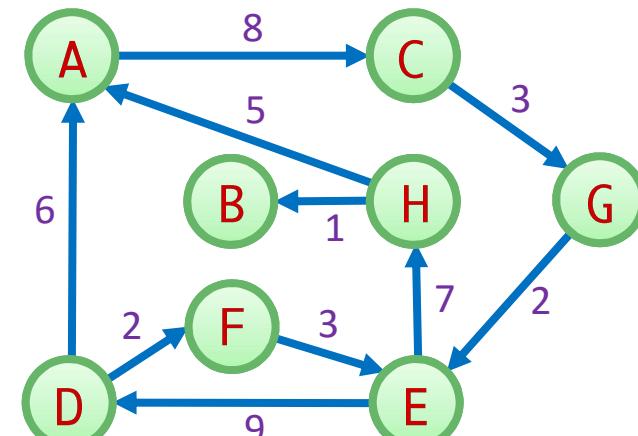
Para un arco  $u \rightarrow v$ :

- $v$  está en la lista asociada a  $u$

# Representación clásica de Grafos con pesos

## Listas de adyacencias (III)

- Lista de adyacencia para un DiGrafo con pesos



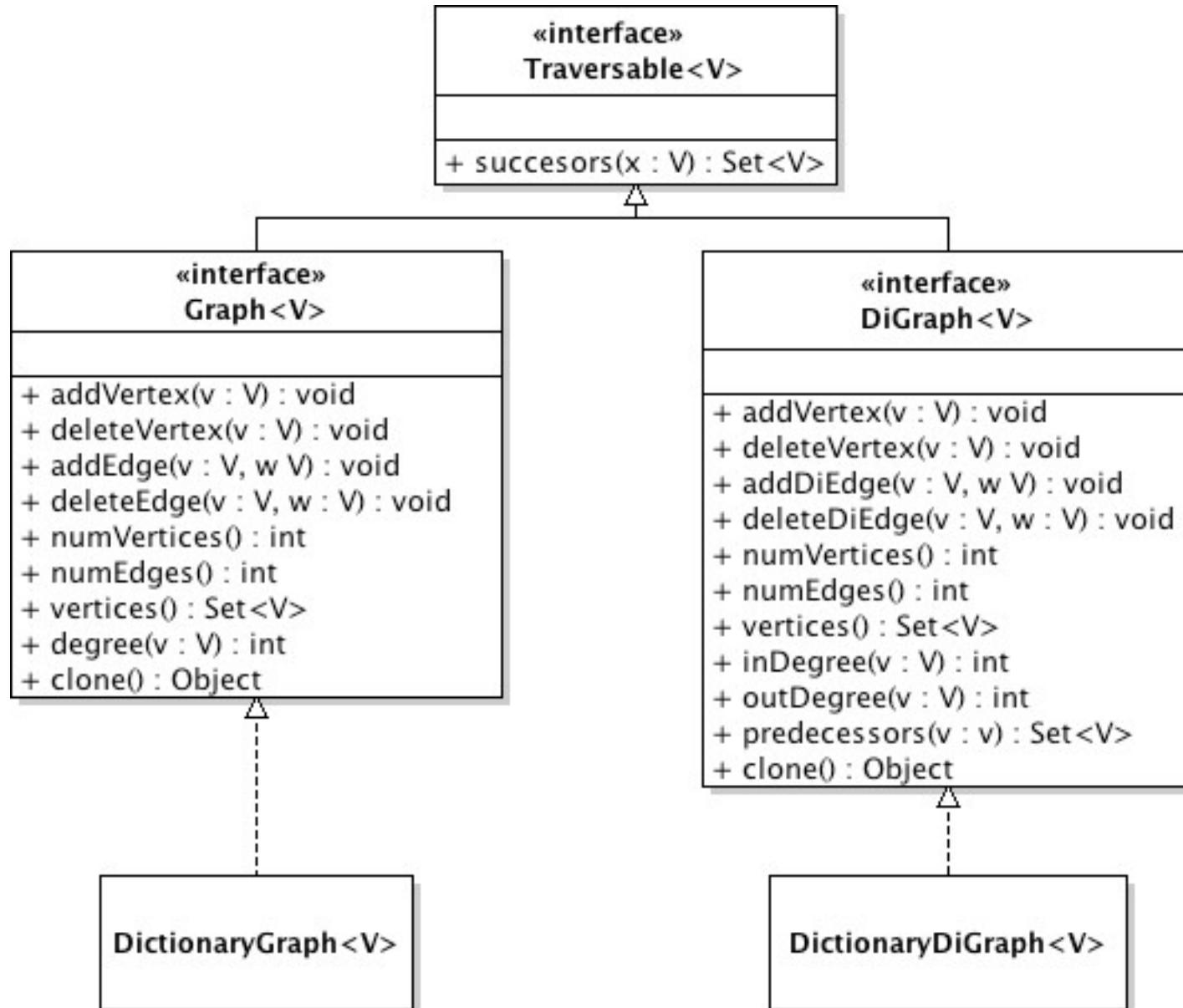
# Representación clásica de Grafos

## Listas de adyacencias (IV)

- Buen uso de memoria, incluso para grafos dispersos 😊
- Para grafos sin pesos y muy densos, el uso de una matriz de adyacencia puede ofrecer mejores resultados 😞
- La complejidad de las operaciones dependen de la implementación. Típicamente:
  - $O(\text{degree } v)$  para determinar si la arista  $v \rightarrow w$  está en el grafo 😞
  - $O(1)$  para devolver todos los sucesores de un vértice 😊

# Grafos y DiGrafos en Java

## Diagrama UML



# Grafos en Java

- Recordamos la interfaz de conjuntos:

```
public interface Set<T> extends Iterable<T> {  
    boolean isEmpty();  
    int size();  
    void insert(T elem);  
    boolean isElem(T elem);  
    void delete(T elem);  
}
```

# Grafos en Java (II)

- Interfaz común para objetos que pueden ser recorridos:

```
package dataStructures.graph;

public interface Traversable<T> {

    // devuelve los elementos alcanzables
    // directamente desde x
    Set<T> successors(T x);
}
```

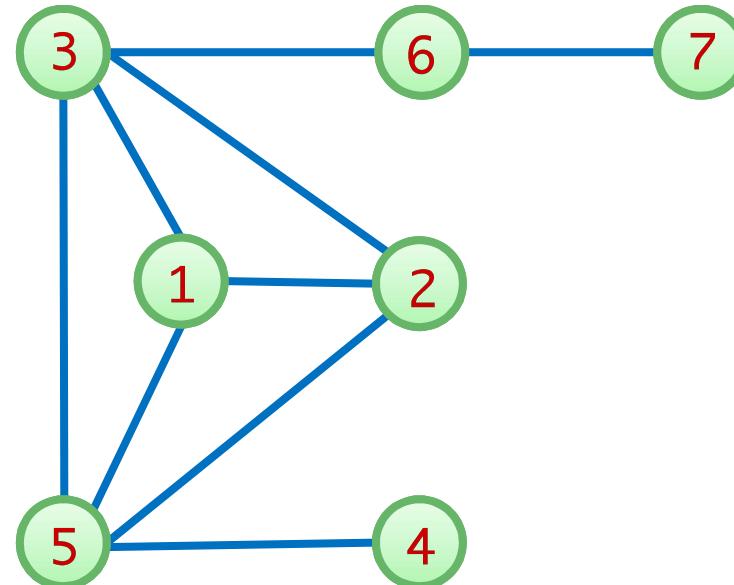
successors devuelve los vértices adyacentes o sucesores

# Grafos en Java (III)

```
public interface Graph<V> extends Traversable<V>, Cloneable {  
    void addVertex(V v); // add vertex to graph  
    // delete from graph a vertex and corresponding edges  
    void deleteVertex(V v);  
  
    void addEdge(V v, V w); // add undirected edge to graph  
    void deleteEdge(V v, V w); // delete undirected edge from graph  
  
    int numVertices(); // returns |V|  
    int numEdges(); // returns |E|  
  
    Set<V> vertices(); // returns all vertices in graph  
    int degree(V v); // returns degree of vertex v  
  
    Object clone(); // returns a copy of the graph  
}
```

# Grafos en Java. Ejemplo

```
import dataStructures.graph.Graph;
import dataStructures.graph.DictionaryGraph;
public class GraphDemo {
    public static void main(String[] args) {
        Graph<Integer> g = new DictionaryGraph<>();
        g.addVertex(1);
        g.addVertex(2);
        g.addVertex(3);
        g.addVertex(4);
        g.addVertex(5);
        g.addVertex(6);
        g.addVertex(7);
        g.addEdge(1,2);
        g.addEdge(1,5);
        g.addEdge(2,5);
        g.addEdge(3,1);
        g.addEdge(3,2);
        g.addEdge(3,6);
        g.addEdge(5,3);
        g.addEdge(5,4);
        g.addEdge(6,7);
```



# Grafos en Java (IV)

- Recordemos la interfaz de diccionarios:

```
public interface Dictionary<K, V> {  
  
    void insert(K k, V v);  
  
    void delete(K k);  
  
    V valueOf(K k); // value associated to k or null  
  
    Iterable<K> keys();  
  
    int size();  
  
    boolean isEmpty();  
  
    boolean isDefinedAt(K k); . . .  
  
}
```

# Grafos en Java (V)

```
public class DictionaryGraph<V> implements Graph<V> {  
    // Set with all vertices in graph  
    protected Set<V> vertices;  
  
    // Dictionary with source of DiEdge as keys and set with  
    // all destinations as values (Adjacency set)  
    protected Dictionary<V,Set<V>> diEdges;  
  
    public DictionaryGraph() {  
        vertices = new HashSet<V>();  
        diEdges = new HashDictionary<V,Set<V>>();  
    }  
  
    public void addVertex(V v) {  
        vertices.insert(v);  
    }  
    ...
```

Una arista se representa con dos arcos.

Para la arista  $u - v$ :

- $v$  está en el conjunto asociado a  $u$  y
- $u$  está en el conjunto asociado a  $v$

# Grafos en Java (VI)

```
// add directed edge from src to dst
private void addDiEdge(V src, V dst) {
    if(!vertices.isElem(src))
        throw new GraphException("vertex "+src+" is not in graph");
    if(!vertices.isElem(dst))
        throw new GraphException("vertex "+dst+" is not in graph");

    Set<V> destinations = diEdges.valueOf(src);
    if(destinations == null) { // first edge from src being added to graph
        destinations = new HashSet<V>();
        diEdges.insert(src, destinations);
    }
    destinations.insert(dst);
}

// undirected edge as two directed edges
public void addEdge(V v, V w) {
    addDiEdge(v, w);
    addDiEdge(w, v);
}
```

# Grafos en Java (VII)

```
// delete directed egde from src to dst
private void deleteDiEdge(V src, V dst) {
    Set<V> destinations = diEdges.valueOf(src);
    if(destinations != null)
        destinations.delete(dst);
}

// delete undirected egde from src to dst
public void deleteEdge(V v, V w) {
    deleteDiEdge(v, w);
    deleteDiEdge(w, v);
}

public void deleteVertex(V v) {
    vertices.delete(v); // remove vertex
    diEdges.delete(v); // remove all edges from v
    // remove all edges to v
    for(V w : vertices)
        deleteDiEdge(w,v);
}
```

# Grafos en Java (VIII)

```
public Set<V> successors(V v) {
    Set<V> destinations = diEdges.valueOf(v);
    return destinations == null ? new HashSet<V>() : destinations;
}

public int degree(V v) {
    return successors(v).size();
}

public Set<V> vertices() {
    return vertices;
}

public int numVertices() {
    return vertices.size();
}

public int numEdges() {
    int directedEdges = 0;
    for(V src : diEdges.keys())
        directedEdges += successors(src).size();
    return directedEdges / 2; // number of undirected edges
}
```

# DiGrafos en Java

```
public interface DiGraph<V> extends Traversable<V>, Cloneable {  
    void addVertex(V v);  
    void deleteVertex(V v);  
    void addDiEdge(V v, V w);  
    void deleteDiEdge(V v, V w);  
    int numVertices();  
    int numEdges();  
    Set<V> vertices();  
    int inDegree(V v);  
    int outDegree(V v);  
    Set<V> predecessors(V dst); // returns vertices leading to dst  
    Object clone();  
}
```

# DiGrafos en Java (II)

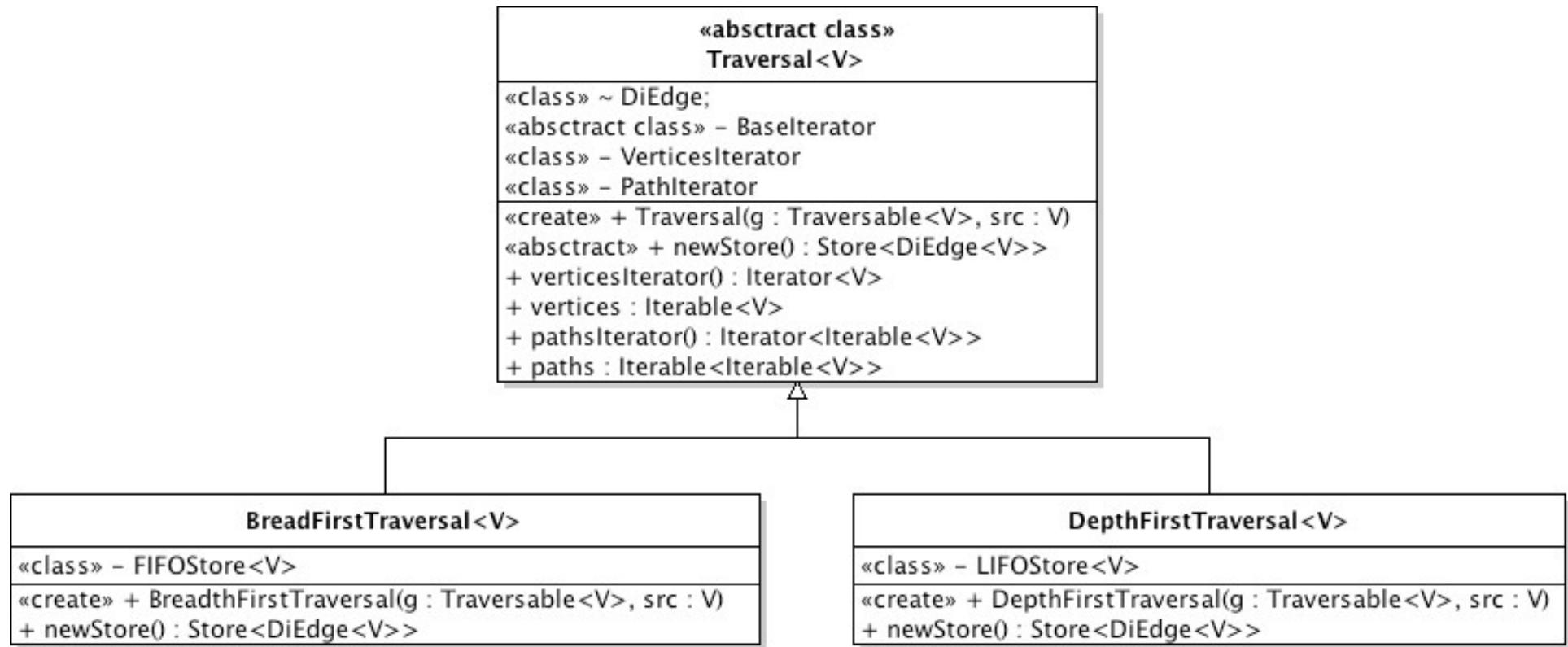
```
public class DictionaryDiGraph<V> implements DiGraph<V> {  
  
    // Set with all vertices in graph  
    protected Set<V> vertices;  
  
    // Dictionary with source of DiEdge as keys and set with  
    // all destinations as values (Adjacency set)  
    protected Dictionary<V,Set<V>> diEdges;  
  
    ...  
}
```

Para un arco  $u \rightarrow v$ :

- $v$  está en el conjunto asociado a  $u$

Implementación similar a  
DictionaryGraph pero las aristas son  
arcos  
Ver el fichero DictionaryDiGraph.java

# Recorridos en Grafos en Java (I)



# Recorridos en Grafos en Java (II)

```
// Generic traversal over a graph
public abstract class Traversal<V> {

    static class DiEdge<T> { // Edge from src vertex to dst vertex
        T src, dst;
        public DiEdge(T s, T d) {
            src = s;
            dst = d;
        }
    }

    private Traversable<V> graph; // graph to explore
    private V source; // initial node for exploration

    // Will be defined in subclasses
    // to return different kinds of stores
    abstract public Store<DiEdge<V>> newStore();

    public Traversal(Traversable<V> g, V src) {
        graph = g;
        source = src;
    }

    ...
}
```

# Recorridos en Grafos en Java (III)

## Baselteator 1/2

```
...
public private abstract class BaseIterator {
    // Vertices ya visitados
    protected Set<V> visited;
    // diEdges aun por explorar
    protected Store<DiEdge<V>> store;
    // origen de cada nodo ya visitado
    protected Dictionary<V,V> sources;
    // siguiente vertice a visitar (o null si se han visitado todos)
    protected V nextVertex;

    public BaseIterator() {
        visited = new HashSet<>();
        store = newStore(); // Un Stack o una Queue
        store.insert(new DiEdge<V>(source,source)); // enqueue o push
        sources = new HashDictionary<>();
        advanceTraversal();
    }
    ...
}
```

```
class DiEdge<T> {
    T src, dst;
    public DiEdge(T s, T d) {
        src = s;
        dst = d;
    }
}
```

- BaseIterator implementa un recorrido genérico sobre un grafo como un iterador (incompleto).
- Las subclases deben implementar el método next para proporcionar la implementación completa del Iterator
- El orden del recorrido depende del objeto almacenado en el método newStore.

# Recorridos en Grafos en Java (IV)

## Baselteator 2/2

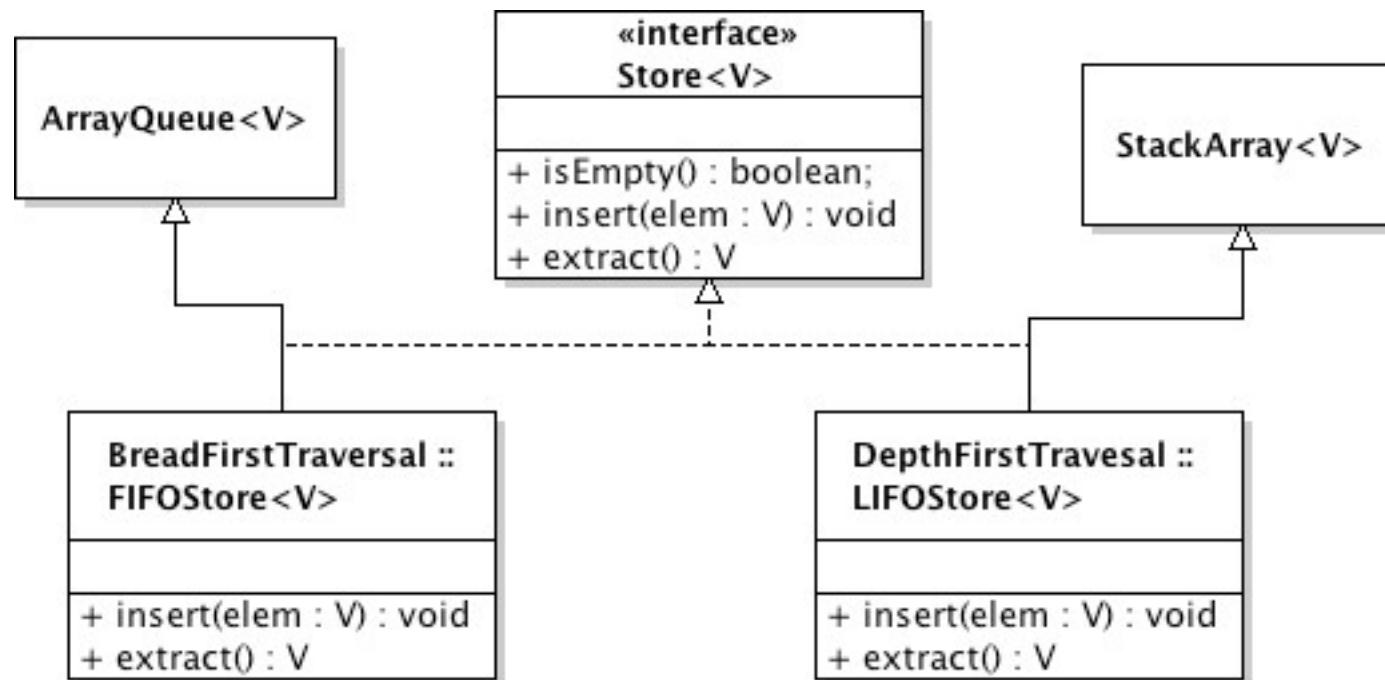
```
...
// Encontrar el siguiente vertice a visitar.
// Deja el vertice (o null si es el final) en nextVertex
protected void advanceTraversal() {
    nextVertex = null;
    while(!store.isEmpty() && nextVertex==null) {
        // first y dequeue o top y pop
        DiEdge<V> edge = store.extract();
        V v = edge.dst;
        if(!visited.isElem(v)) {
            nextVertex = v;
            visited.insert(v);
            sources.insert(v, edge.src);
            for(V w : graph.successors(v))
                if (!visited.isElem(w))
                    store.insert(new DiEdge<V>(v, w)); // enqueue o push
        }
    }
}

public boolean hasNext() {
    return nextVertex != null;
}
...

```

```
class DiEdge<T> {
    T src, dst;
    public DiEdge(T s, T d) {
        src = s;
        dst = d;
    }
}
```

# Recorridos en Grafos en Java (V)



# Recorridos en Grafos en Java (VI)

- La interfaz Store se utilizará para incluir un objeto que mantenga (según convenga) las aristas aún no visitadas

```
interface Store<T> {  
    boolean isEmpty();  
    void insert(T elem);  
    T extract();  
}
```

# Recorridos en Grafos en Java (VII)

## En Profundidad

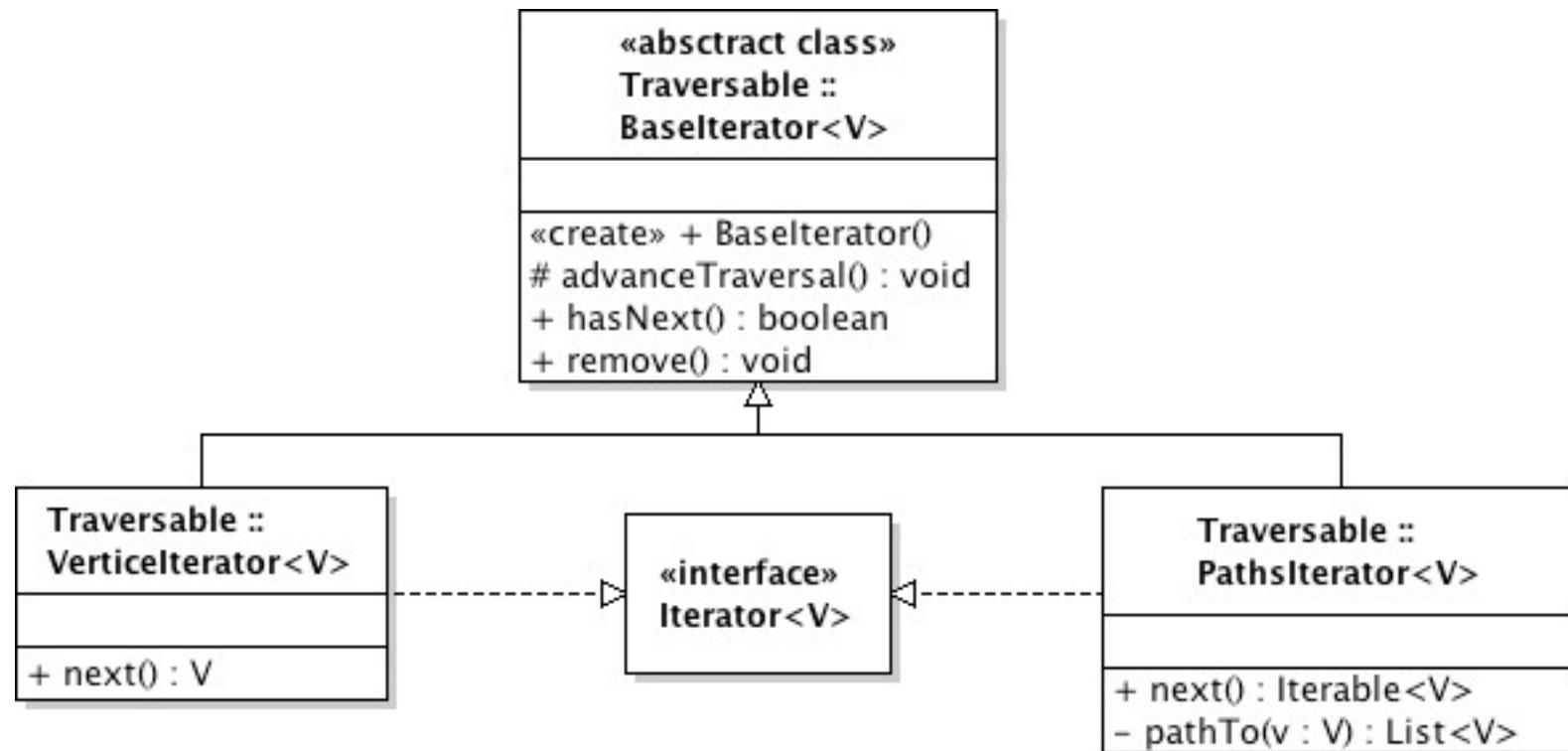
```
public class DepthFirstTraversal<V> extends Traversal<V> {  
    // DFT starting from source  
    public DepthFirstTraversal(Traversable<V> g, V source) {  
        super(g,source);  
    }  
  
    private class LIFOStore<T> extends ArrayStack<T>  
        implements Store<T> {  
  
        public void insert(T elem) {  
            push(elem);  
        }  
  
        public T extract() {  
            T elem = top();  
            pop();  
            return elem;  
        }  
        public Store<DiEdge<V>> newStore() {  
            return new LIFOStore<>();  
        }  
    }  
}
```

# Recorridos en Grafos en Java (VIII)

## En Anchura

```
public class BreadthFirstTraversal<V> extends Traversal<V> {  
    // BFT starting from source  
    public BreadthFirstTraversal(Traversable<V> g, V source) {  
        super(g,source);  
    }  
  
    private class FIFOStore<T> extends ArrayQueue<T>  
        implements Store<T> {  
  
        public void insert(T elem) {  
            enqueue(elem);  
        }  
  
        public T extract() {  
            T elem = first();  
            dequeue();  
            return elem;  
        }  
  
        public Store<DiEdge<V>> newStore() {  
            return new FIFOStore<>();  
        }  
    }  
}
```

# Recorridos en Grafos en Java (IX)



# Recorridos en Grafos en Java (X)

```
...  
  
private class VerticesIterator extends BaseIterator  
    implements Iterator<V> {  
  
    public V next() {  
        if (!hasNext())  
            throw new NoSuchElementException();  
  
        V vertex = nextVertex;  
  
        advanceTraversal(); //for next iteration of iterator  
  
        return vertex;  
    }  
  
    public Iterator<V> verticesIterator() {  
        return new VerticesIterator();  
    }  
  
    public Iterable<V> vertices() {  
        return new Iterable<V>(){  
            public Iterator<V> iterator() {  
                return verticesIterator();  
            }  
        };  
    }  
}
```

Un recorrido por los vértices

# Recorridos en Grafos en Java (XI)

- Recordemos la interfaz de listas:

```
public interface List<T> extends Iterable<T> {  
    boolean isEmpty();  
    int size();  
    T get(int i);  
    void set(int i, T elem);  
    void add(int i, T elem);  
    void append(T elem);  
    void remove(int i);  
}
```

# Recorrido de Grafos en Java (XII)

```
...
private class PathsIterator extends BaseIterator
    implements Iterator<Iterable<V>> {

    // returns path from initial source to vertex v
    private List<V> pathTo(V v) {
        List<V> path = new LinkedList<V>();
        while(v!=source) {
            path.add(0, v);
            v = sources.valueOf(v);
        }
        path.insert(0, v);
        return path;
    }

    public Iterable<V> next() {
        if (!hasNext())
            throw new NoSuchElementException();

        // reconstruct path from source to visited vertex
        Iterable<V> path = pathTo(nextVertex);

        advanceTraversal(); // for next iteration of iterator

        return path;
    }
}
...

```

Un recorrido sobre  
paths(Iterables sobre  
vértices)

# Recorridos de Grafos en Java (XIII)

```
public Iterator<Iterable<V>> pathsIterator() {  
    return new PathsIterator();  
}  
  
public Iterable<Iterable<V>> paths() {  
    return new Iterable<Iterable<V>>(){  
        public Iterator<Iterable<V>> iterator() {  
            return pathsIterator();  
        }  
    };  
}
```

# Recorridos de Grafos en Java (XIV)

## Ejemplo

```
Graph<Integer> g = new DictionaryGraph<>();
```

```
g.addVertex(1);  
g.addVertex(2);  
...
```

```
g.addEdge(1,2);  
g.addEdge(1,5);  
...
```

```
Traversal<Integer> dfs = new DepthFirstTraversal<>(g,1);  
System.out.println("DF traversal from node 1:");  
for(Integer vertex : dfs.vertices())  
    System.out.println(vertex);
```

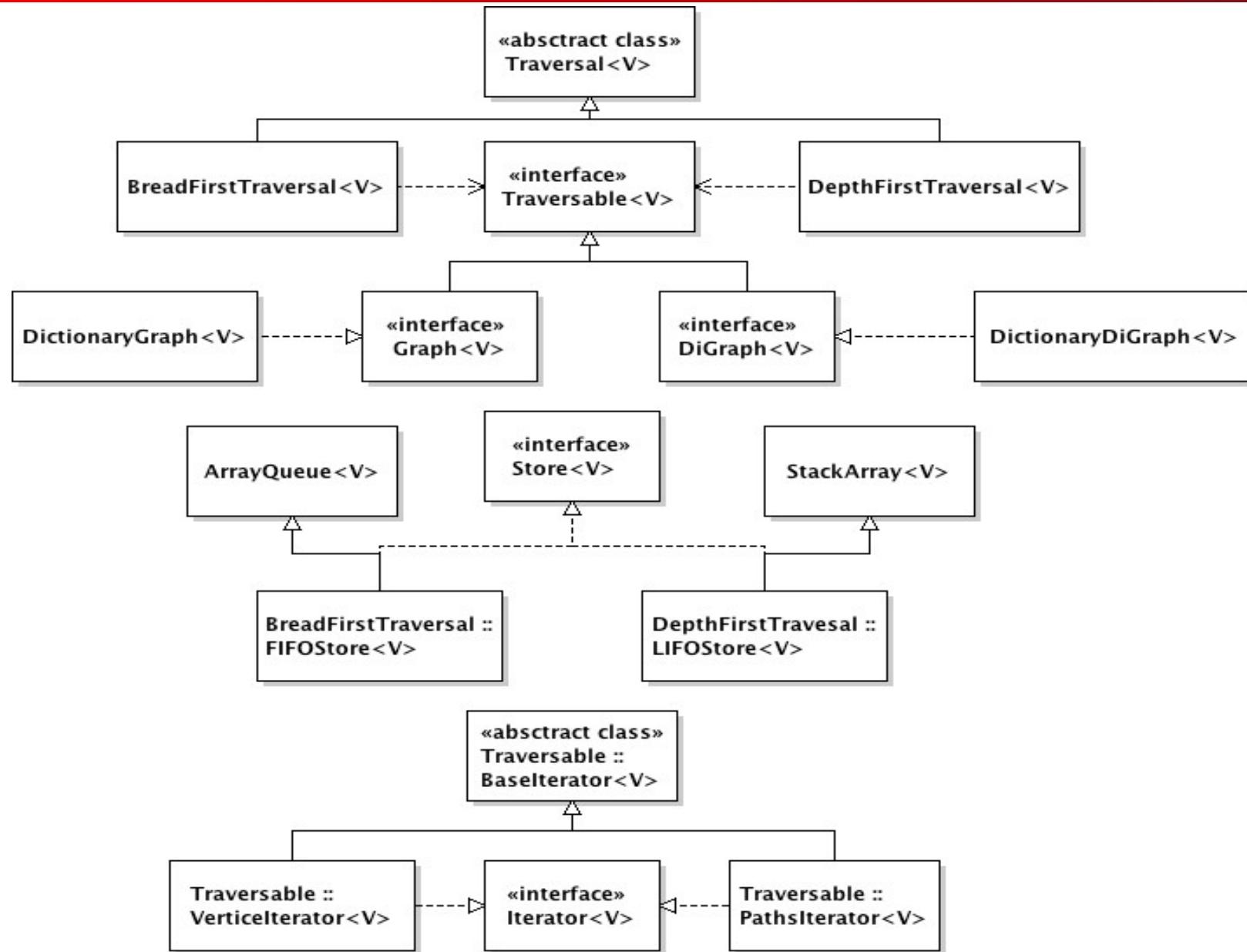
Visita los vértices en profundidad

```
Traversal<Integer> bfs = new BreadthFirstTraversal<>(g,1);  
System.out.println("BF paths traversal from node 1:");  
for(Iterable<Integer> path: bfs.paths())  
    System.out.println(path);
```

Visita de los vértices en profundidad pero devolviendo los paths desde 1

# Recorridos de Grafos en Java (XV)

## Modelo de datos



# Componentes Conexas de un Grafo (I)

- Los recorridos en profundidad o anchura pueden usarse para calcular todos los vértices conectados a un vértice SRC en un grafo g. En Haskell es una de las siguientes listas :
  - `dft g src`
  - `bft g src`
- Una función Haskell que devuelve las Componentes Conexas en un grafo:

```
type ConnectedComponent a = [a]
```

```
dftConnectedComps :: (Ord a) => Graph a -> [ConnectedComponent a]
dftConnectedComps g = aux (vertices g)
where
  aux [] = []
  aux (v:vs) = comp : aux (vs \\ comp)
  where
    comp = dft g v
```

Todos los vértices  
van a ser  
visitados

Se calcula la componente de v, y se  
eliminan todos sus vértices

# Componentes Conexas de un Grafo (II)

- En Java, podemos obtener la componente conexa del vértice `src` en forma parecida:
  - - bien con un recorrido en profundidad

```
Set<V> component = new HashSet<O>;
for(V v : new DepthFirstTraversal<O>(g, src).vertices()) {
    component.insert(v); // add to component
}
```

- - bien con un recorrido en anchura

```
Set<V> component = new HashSet<O>;
for(V v : new BreadthFirstTraversal<O>(g, src).vertices()) {
    component.insert(v); // add to component
}
```

- Para obtener todas las componentes ...

# Componentes Conexas de un Grafo (II)

```
public class ConnectedComponents<V> {
    private Set<Set<V>> components; // Components as sets of vertices
    private Dictionary<V, Integer> inComponent; // in which component is a vertex?

    public ConnectedComponents(Graph<V> g) {
        components = new HashSet<>();
        inComponent = new HashDictionary<>();

        Set<V> unvisited = new HashSet<V>(); // no vertex has been visited yet
        for(V v : g.vertices())
            unvisited.insert(v);

        for(int c = 0; !unvisited.isEmpty(); c++) {
            V src = unvisited.iterator().next(); // an unvisited vertex
            inComponent.insert(src, c); // store component number for src

            Set<V> component = new HashSet<>();
            for(V v : new DepthFirstTraversal<>(g, src).vertices()) {
                component.insert(v); // add to component
                inComponent.insert(v, c); // store component number for v
            }

            components.insert(component); // add component to set of components
            for(V v : component)
                unvisited.delete(v); // all vertex in component have been visited
        }
    }
}
```

# Componentes Conexas de un Grafo (III)

```
public Set<Set<V>> components() { // return all components
    return components;
}

// u and v are connected if in same component
public boolean areConnected(V v, V w) {
    return inComponent.valueOf(v).equals(inComponent.valueOf(w));
}

}

public static void main(String[] args) {
    Graph<Integer> g = new DictionaryGraph<>();
    g.addVertex(1);
    ...
    g.addEdge(1,2);
    ...

    ConnectedComponents<Integer> cc = new ConnectedComponents<>(g);

    System.out.println("Connected components: "+cc.components());
    System.out.println("Are 1 and 2 connected? "+cc.areConnected(1, 2));
    System.out.println("Are 1 and 6 connected? "+cc.areConnected(1, 6));
}
```

Ejemplo

# Orden Topológico en DiGrafos

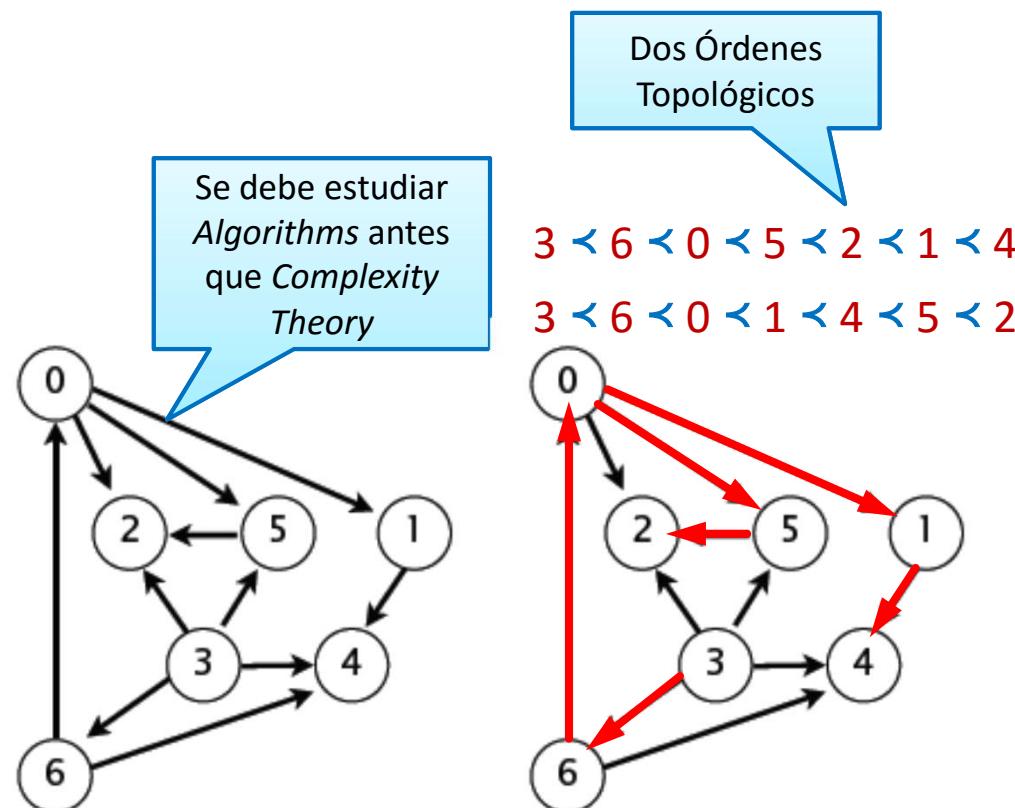
- Dado un DiGrafo acíclico (DAG, Directed Acyclic Graph), un **orden Topológico** es una relación de orden total ( $\prec$ ) entre vértices tal que:
  - Si existe un arco desde  $v$  a  $w$ , entonces  $w$  es mayor que  $v$  en el orden.
  
$$v \rightarrow w \quad \Rightarrow \quad v \prec w$$
  
- Un DiGrafo puede tener diferentes Ordenes Topológicos
- Si el DiGrafo es cíclico, el Orden Topológico no existe ☹

# Orden Topológico en DiGrafos (II)

- **Objetivo:** dado un conjunto de tareas que deben terminar antes que otras comiencen (prerrequisitos),  
¿En qué orden deben ser planificadas las tareas?

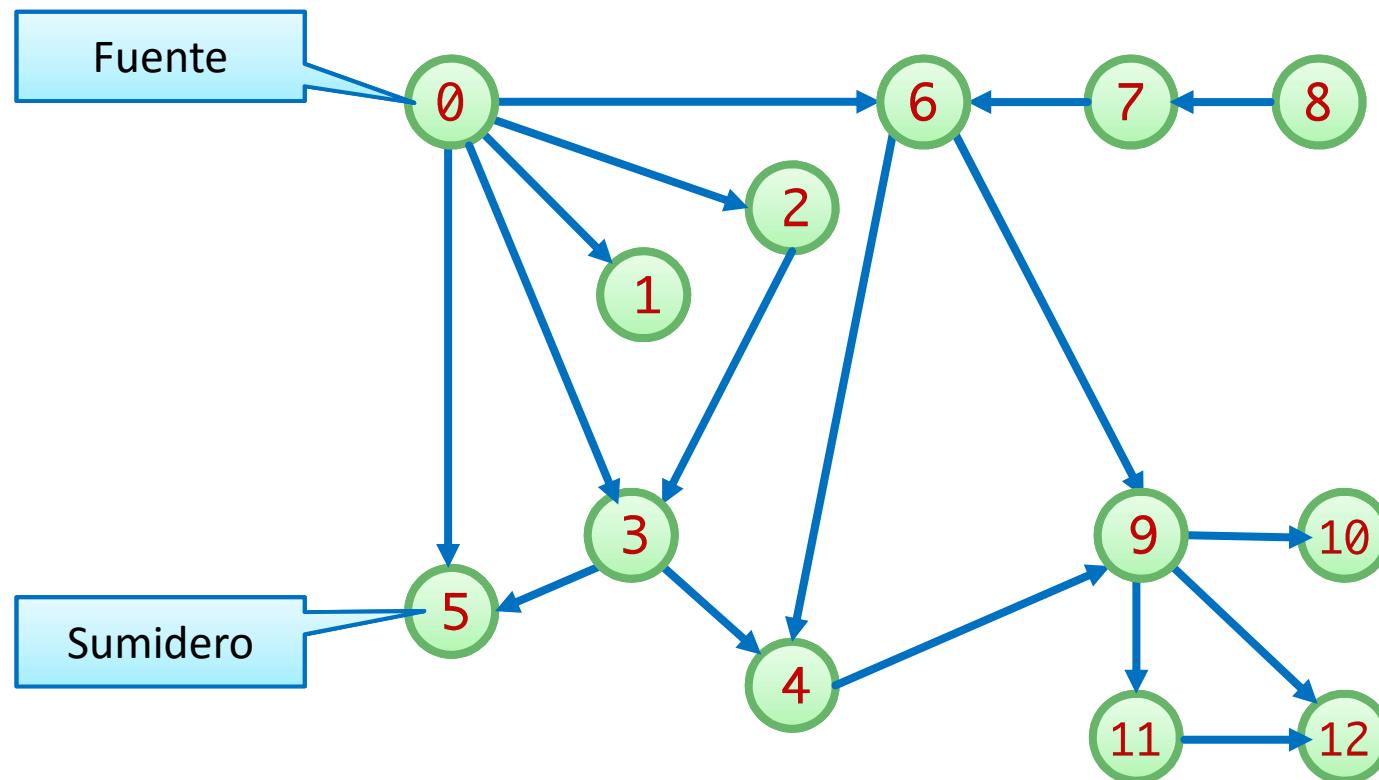
- Modelado por un DiGrafo.
  - vértice= tarea
  - Arco = prerequisito.

0. Algorithms  
1. Complexity Theory  
2. Artificial Intelligence  
3. Intro to CS  
4. Cryptography  
5. Scientific Computing  
6. Advanced Programming



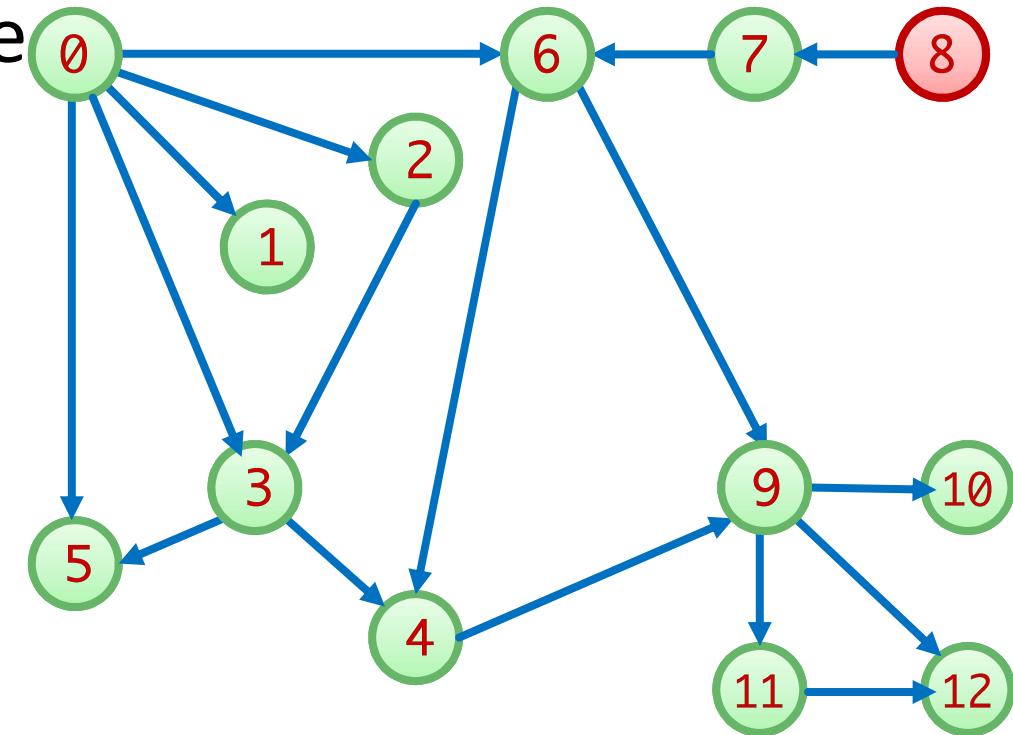
# Orden Topológico en DiGrafos (III)

- **Fuente**: vértice cuyo grado de entrada es 0
- **Sumidero**: vértice cuyo grado de salida es 0



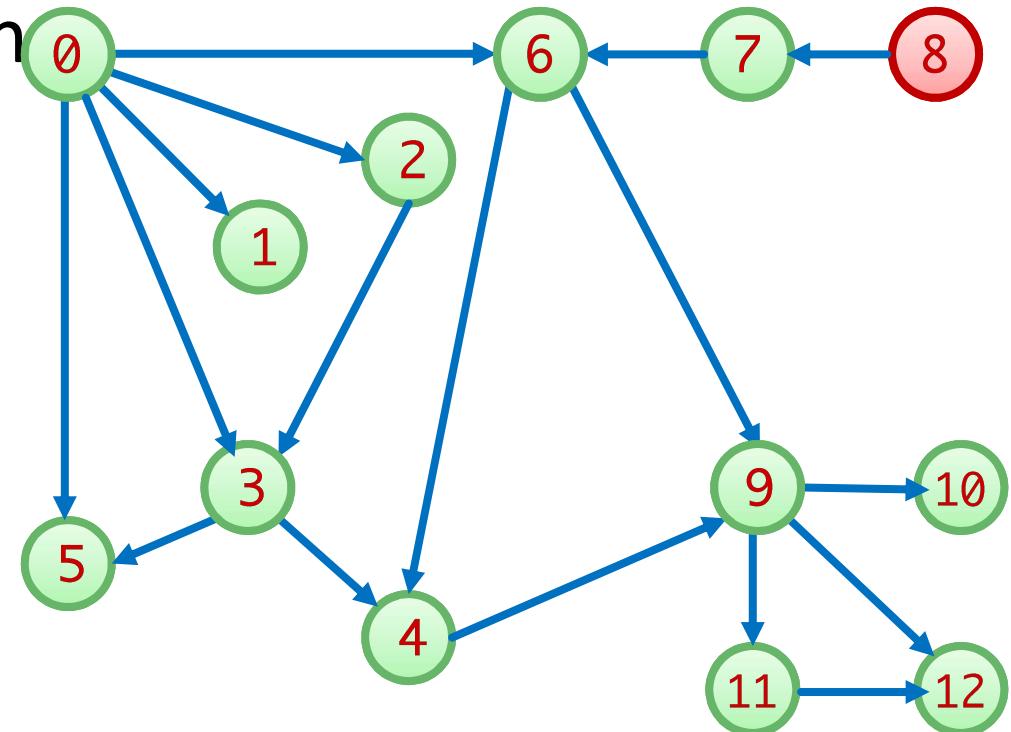
# Orden Topológico en DiGrafos ( $IV_1$ )

- Tomamos una fuente



# Orden Topológico en DiGrafos ( $IV_2$ )

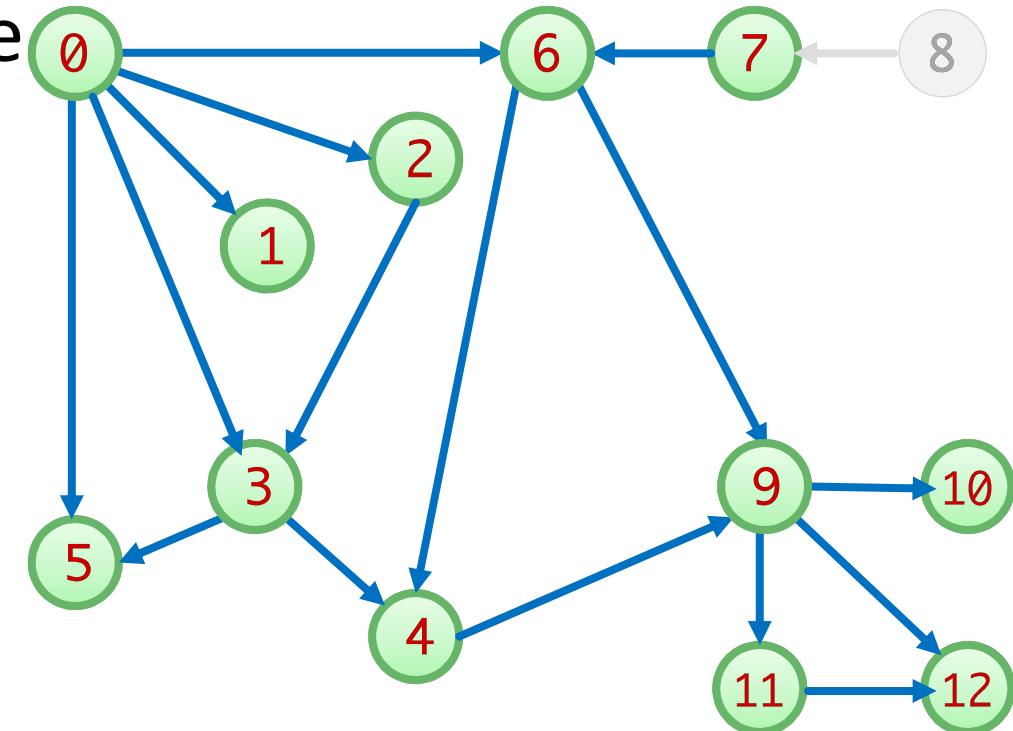
- La añadimos al orden



Orden Topológico: 8

# Orden Topológico en DiGrafos (IV<sub>3</sub>)

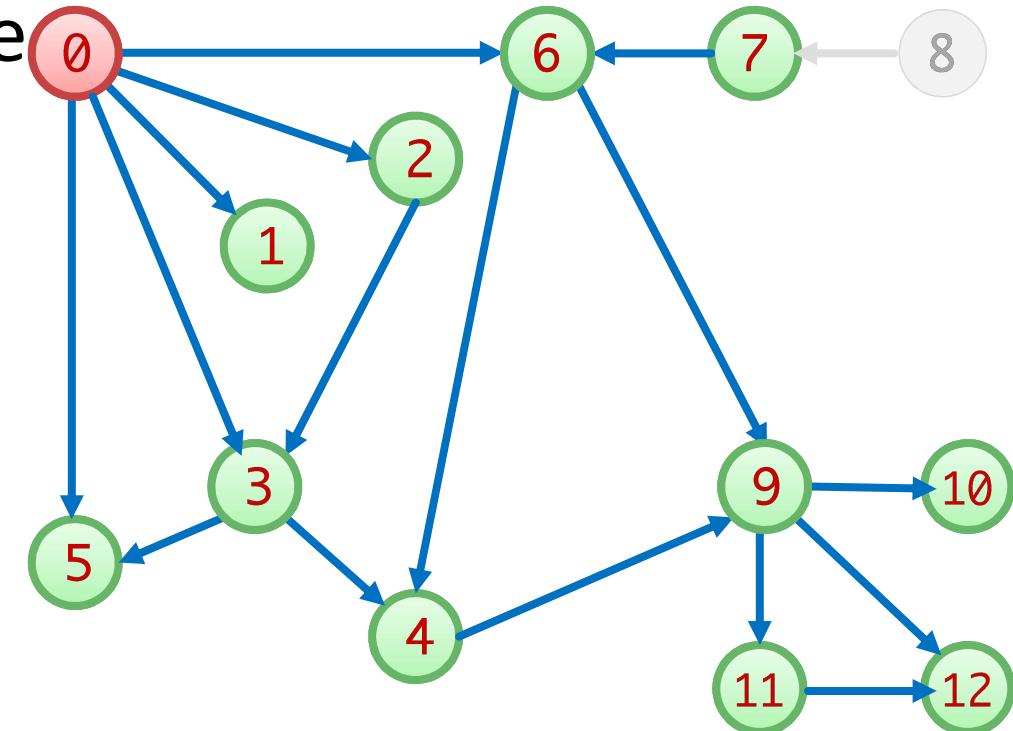
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8

# Orden Topológico en DiGrafos (IV<sub>4</sub>)

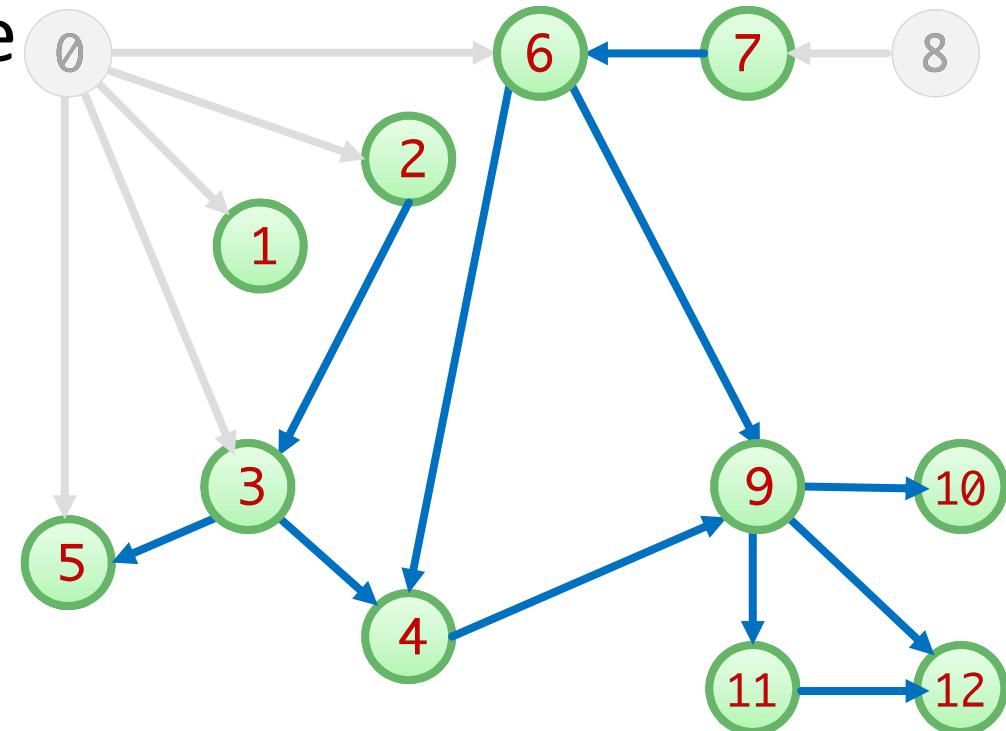
- Tomamos una fuente



Orden Topológico:  $8 < 0$

# Orden Topológico en DiGrafos (IV<sub>5</sub>)

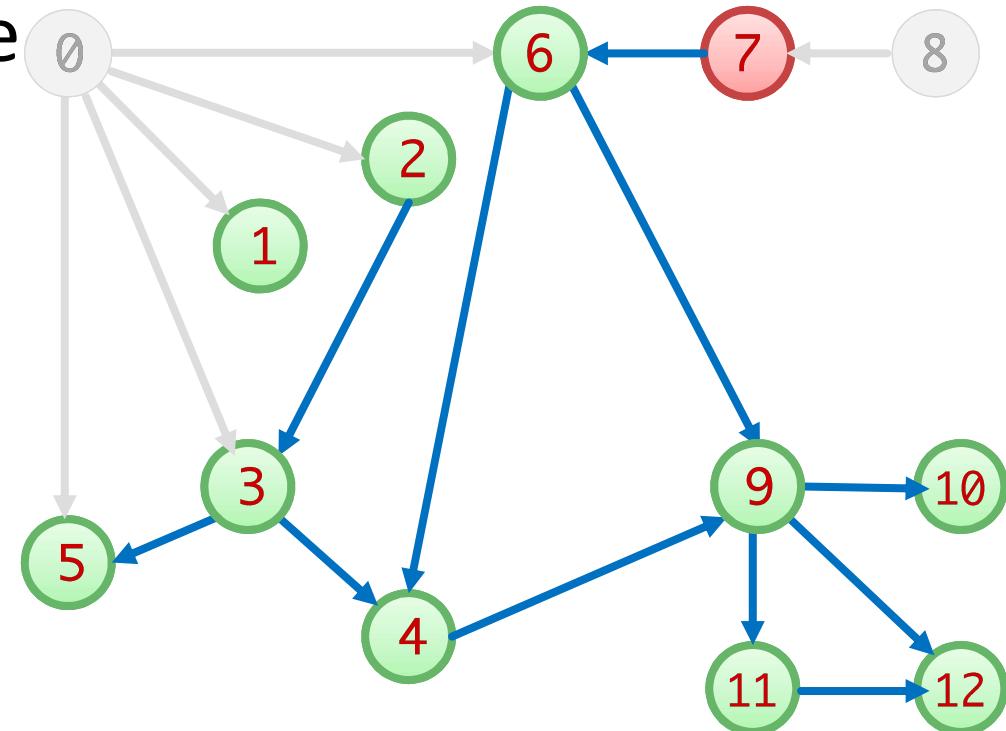
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0

# Orden Topológico en DiGrafos (IV<sub>6</sub>)

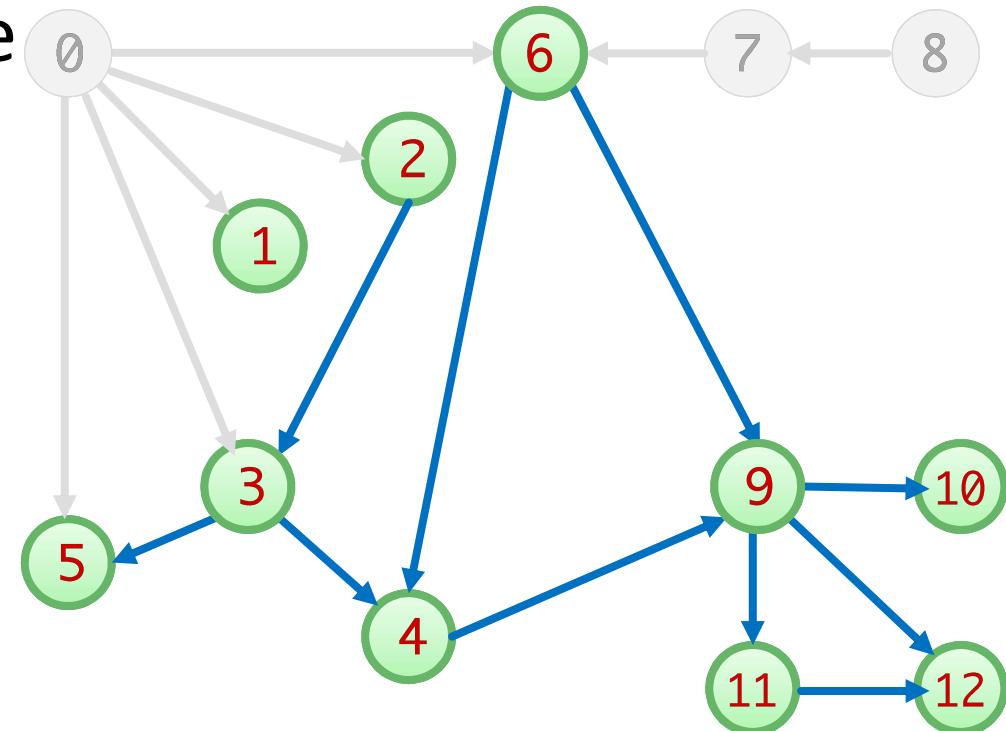
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7

# Orden Topológico en DiGrafos (IV<sub>7</sub>)

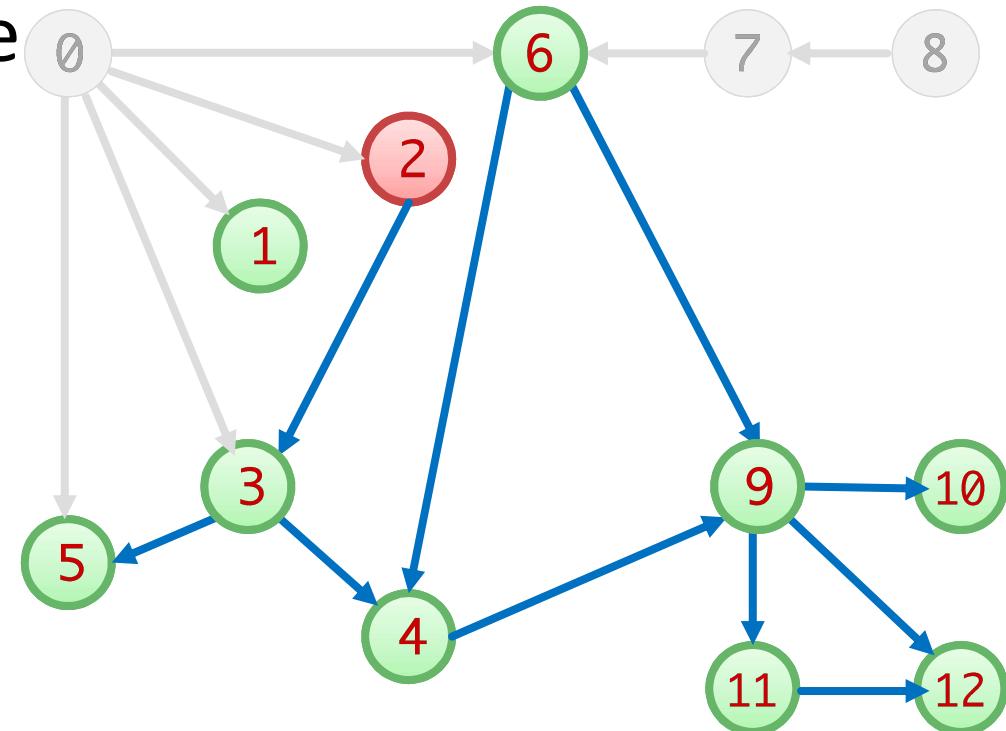
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico:  $8 < 0 < 7$

# Orden Topológico en DiGrafos (IV<sub>8</sub>)

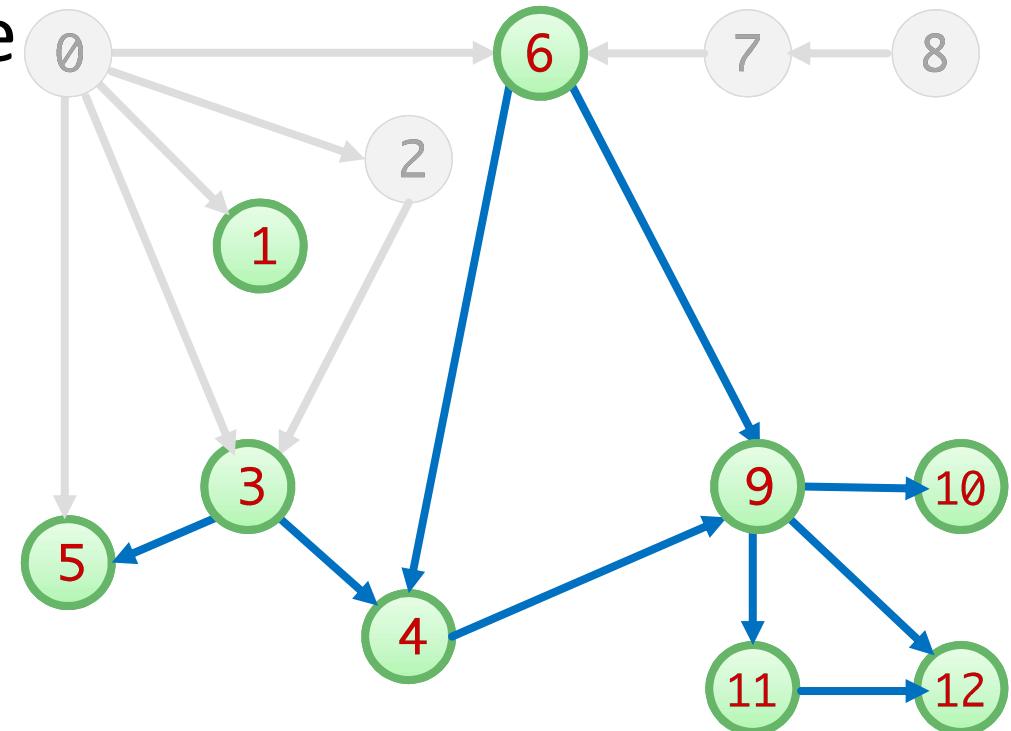
- Tomamos una fuente



Orden Topológico:  $8 < 0 < 7 < 2$

# Orden Topológico en DiGrafos (IV<sub>9</sub>)

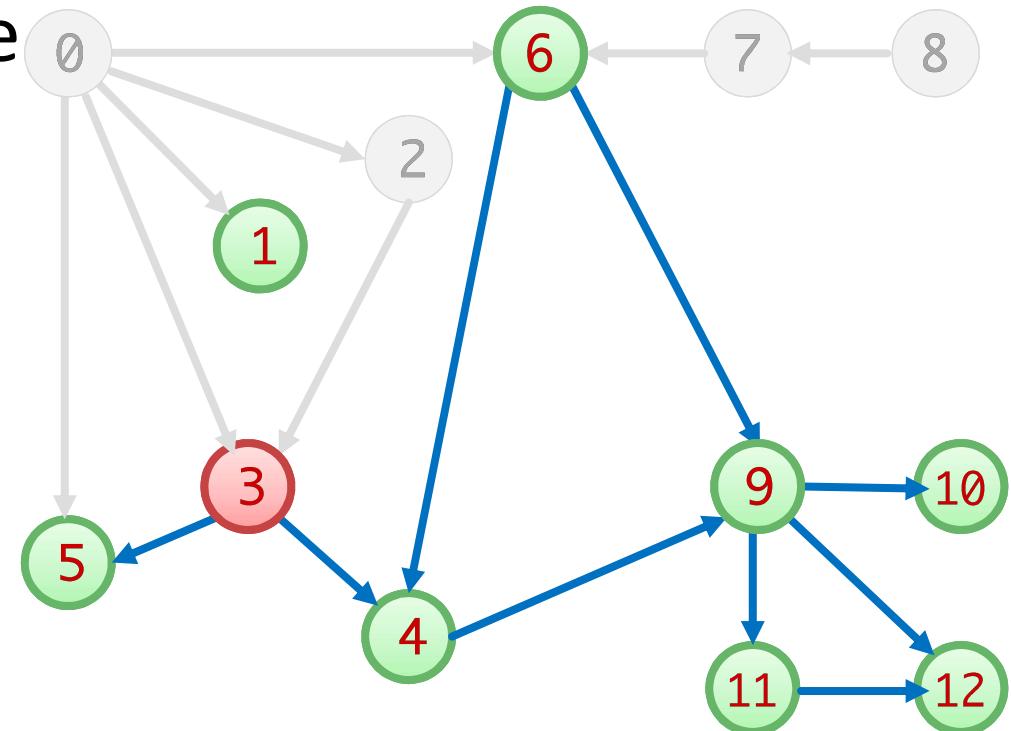
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0 < 7 < 2

# Orden Topológico en DiGrafos (IV<sub>10</sub>)

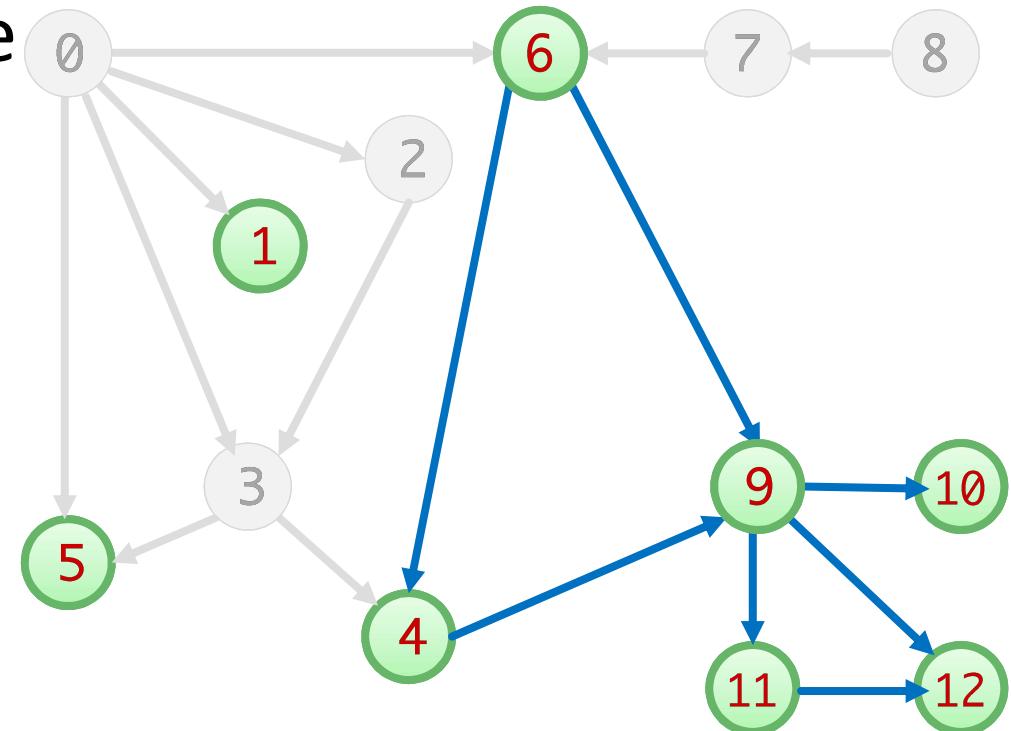
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7 < 2 < 3

# Orden Topológico en DiGrafos (IV<sub>11</sub>)

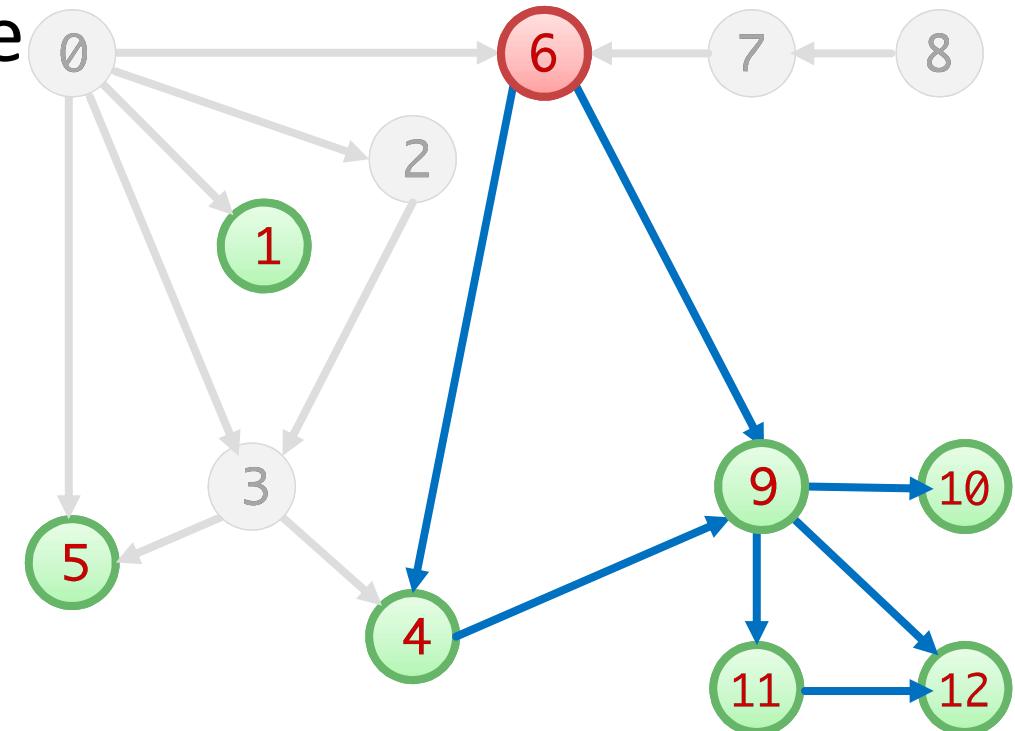
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0 < 7 < 2 < 3

# Orden Topológico en DiGrafos (IV<sub>12</sub>)

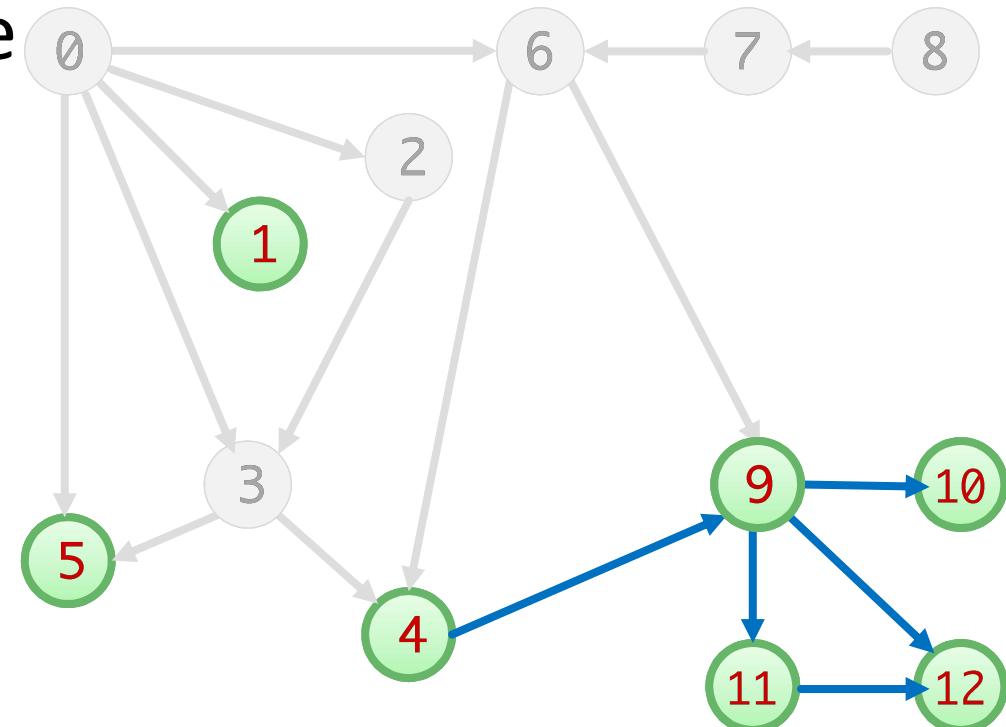
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6

# Orden Topológico en DiGrafos (IV<sub>13</sub>)

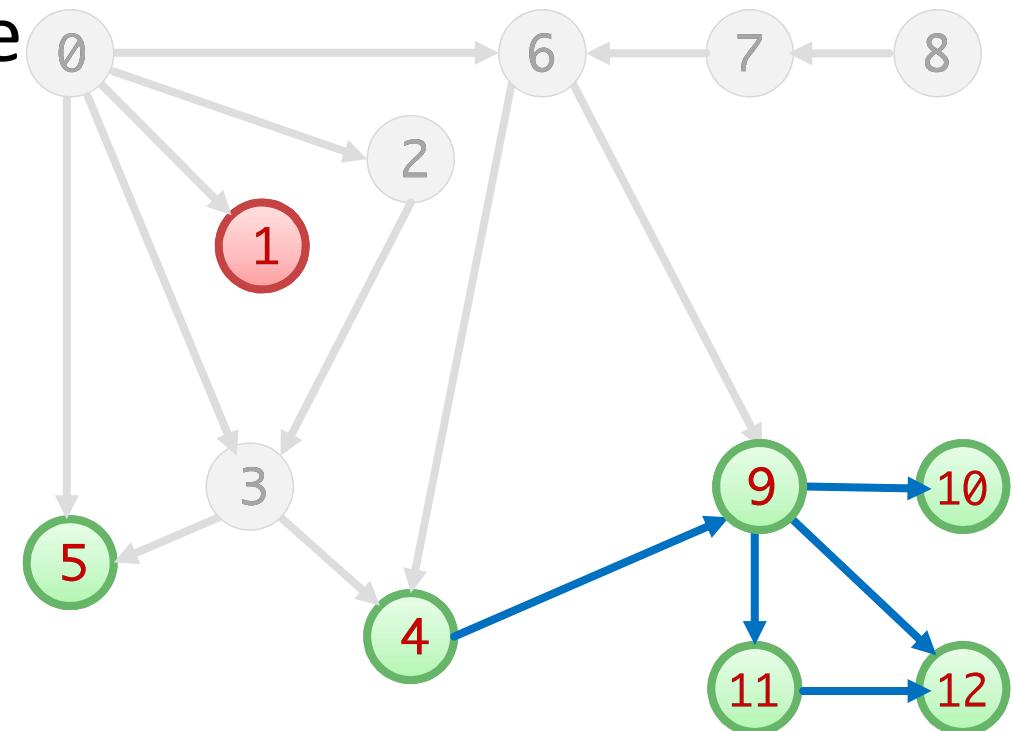
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6

# Orden Topológico en DiGrafos (IV<sub>14</sub>)

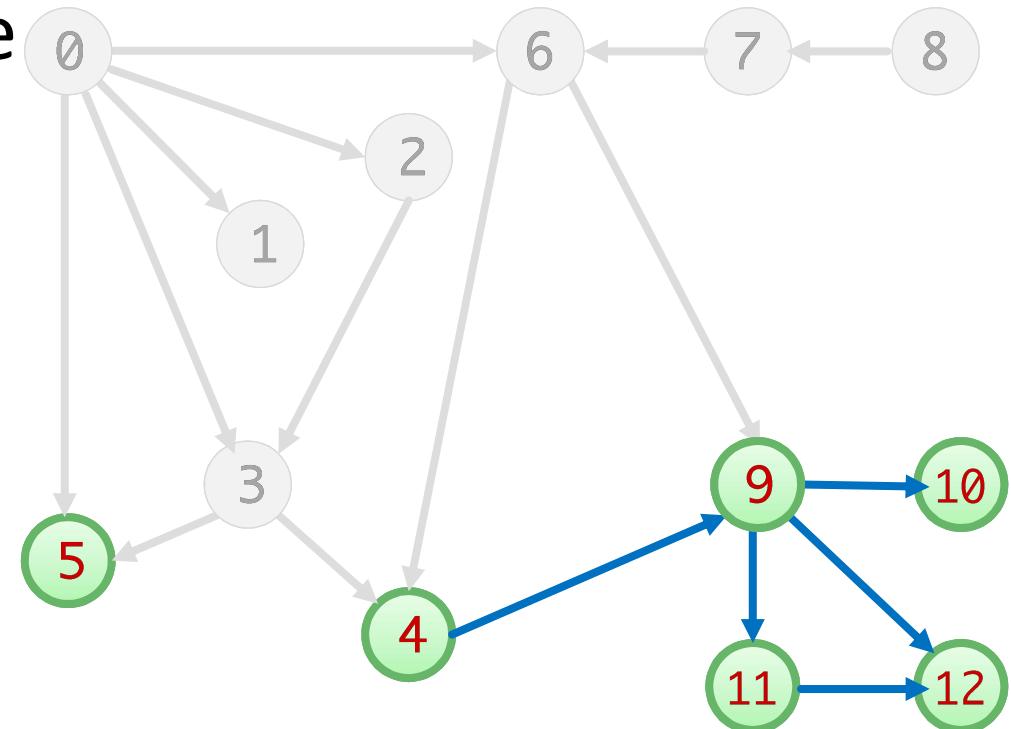
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1

# Orden Topológico en DiGrafos (IV<sub>16</sub>)

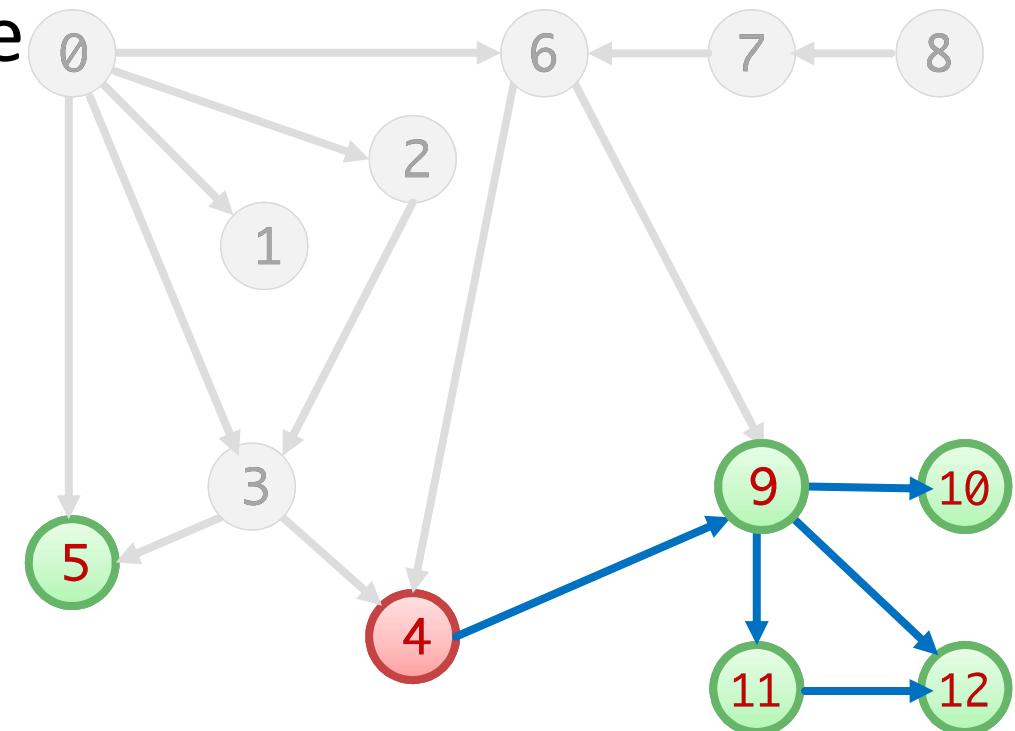
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1

# Orden Topológico en DiGrafos (IV<sub>17</sub>)

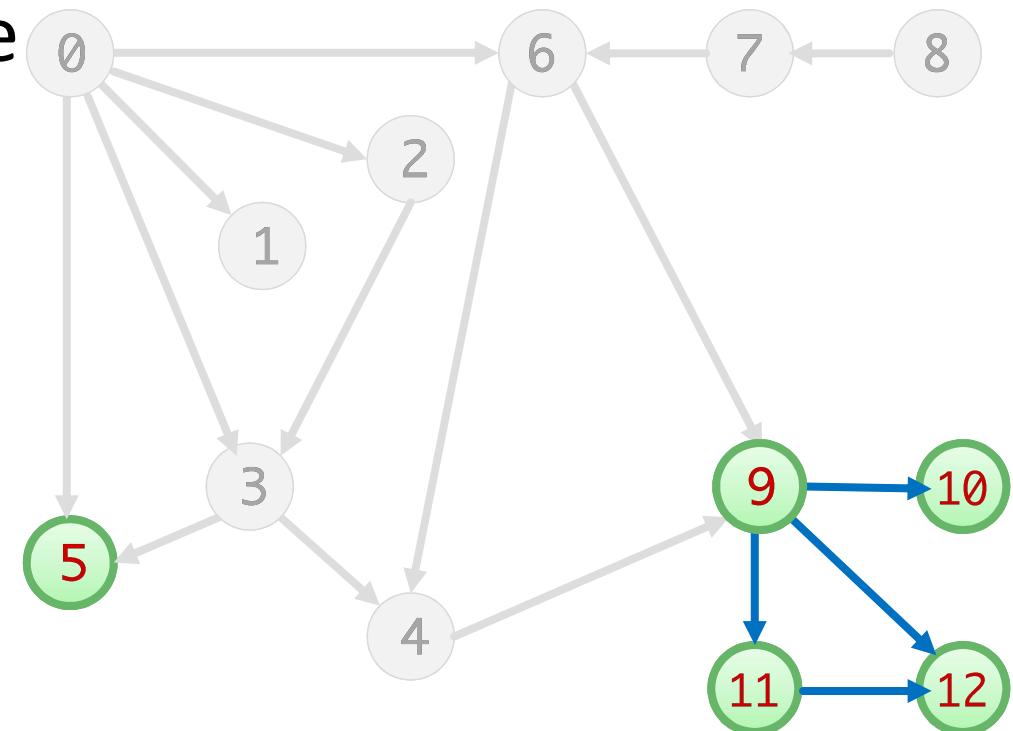
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4

# Orden Topológico en DiGrafos (IV<sub>18</sub>)

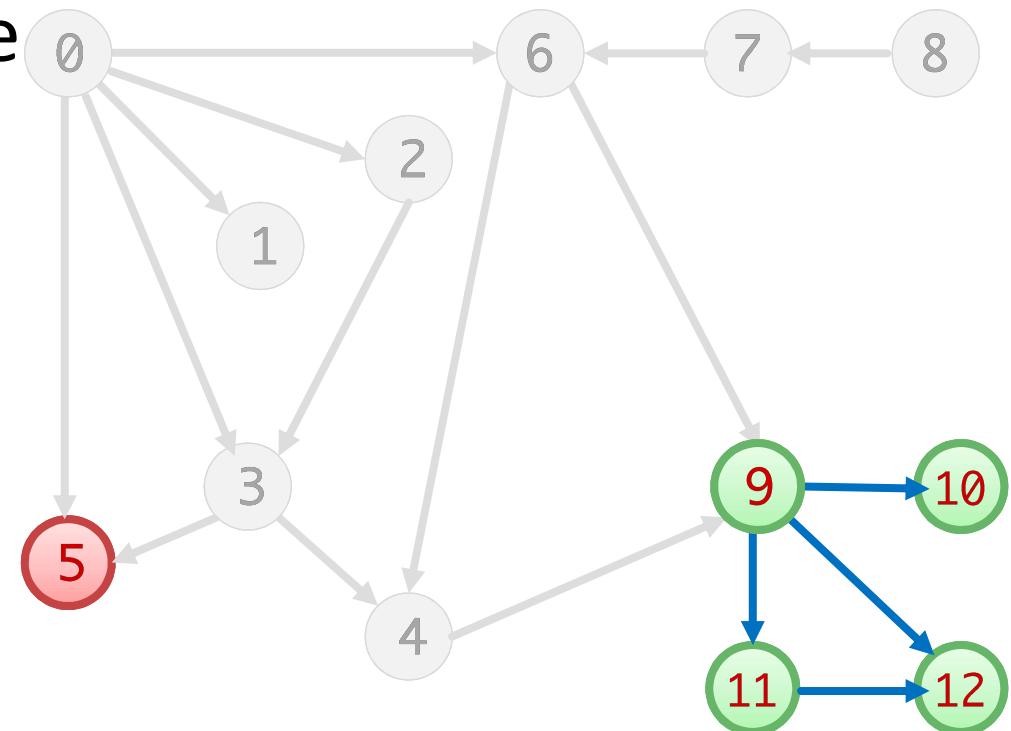
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4

# Orden Topológico en DiGrafos (IV<sub>19</sub>)

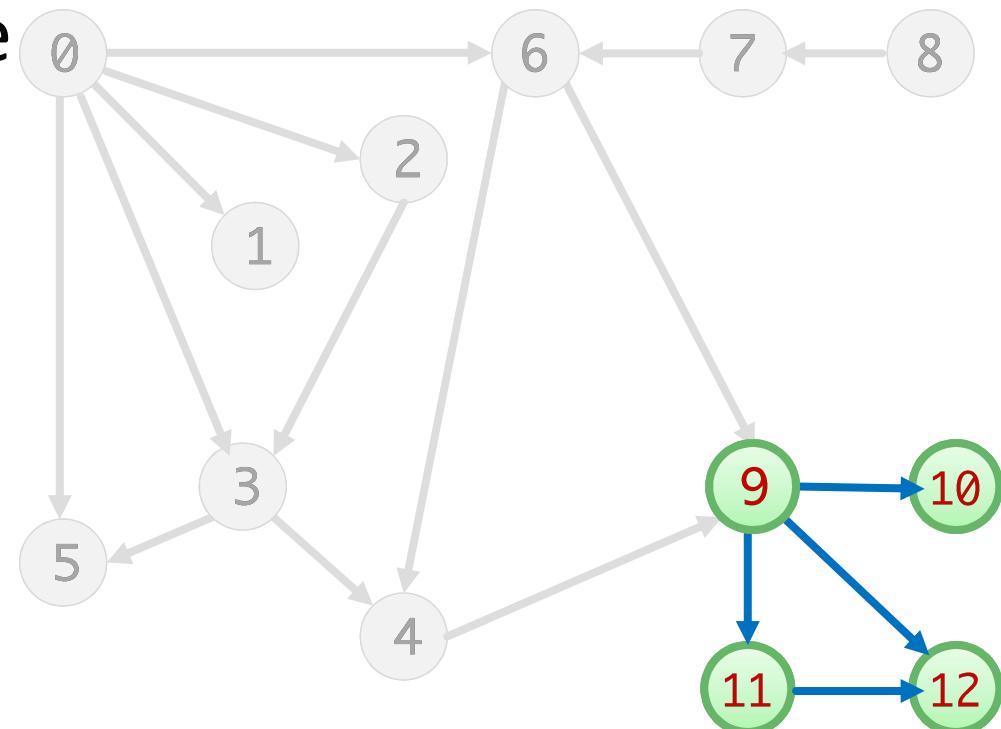
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5

# Orden Topológico en DiGrafos (IV<sub>20</sub>)

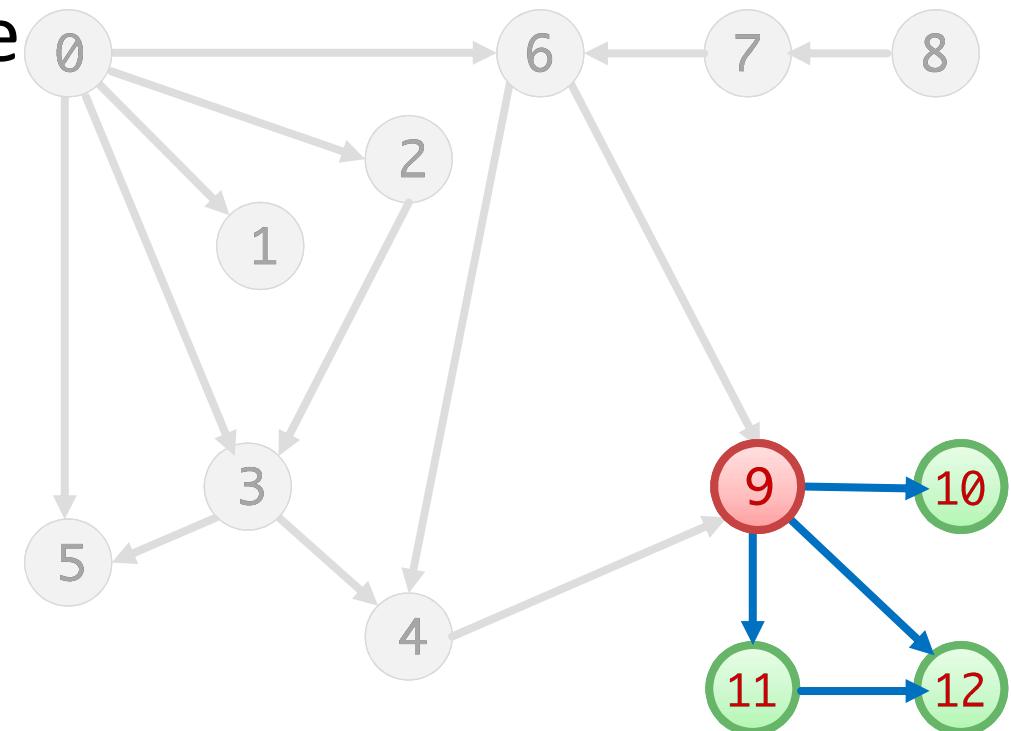
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5

# Orden Topológico en DiGrafos (IV<sub>21</sub>)

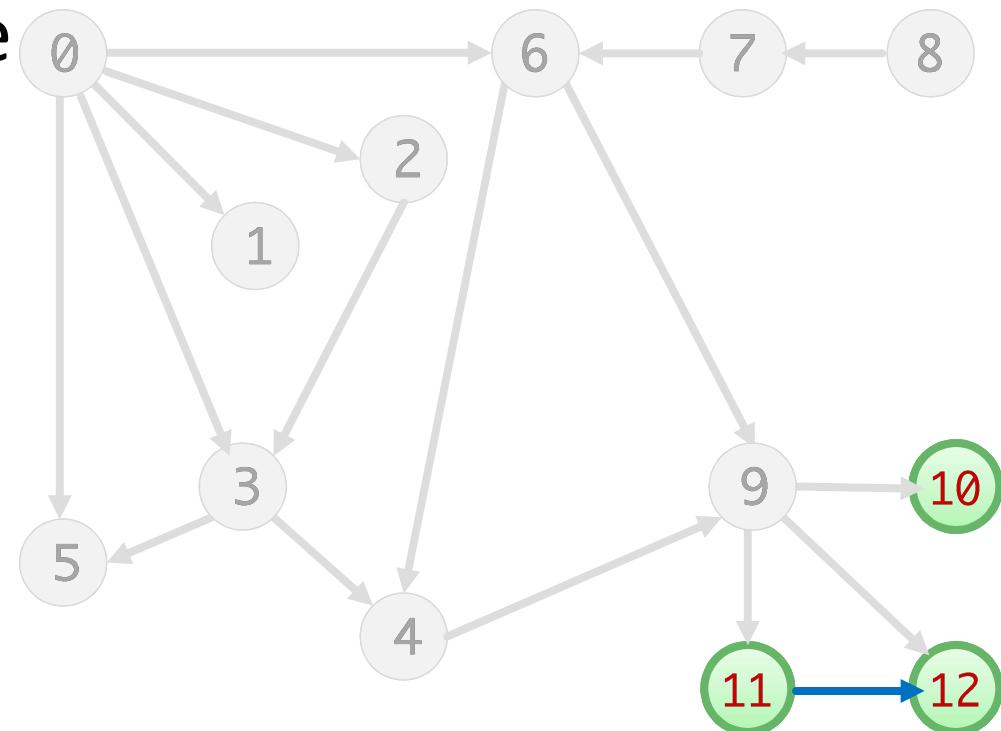
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5 < 9

# Orden Topológico en DiGrafos (IV<sub>22</sub>)

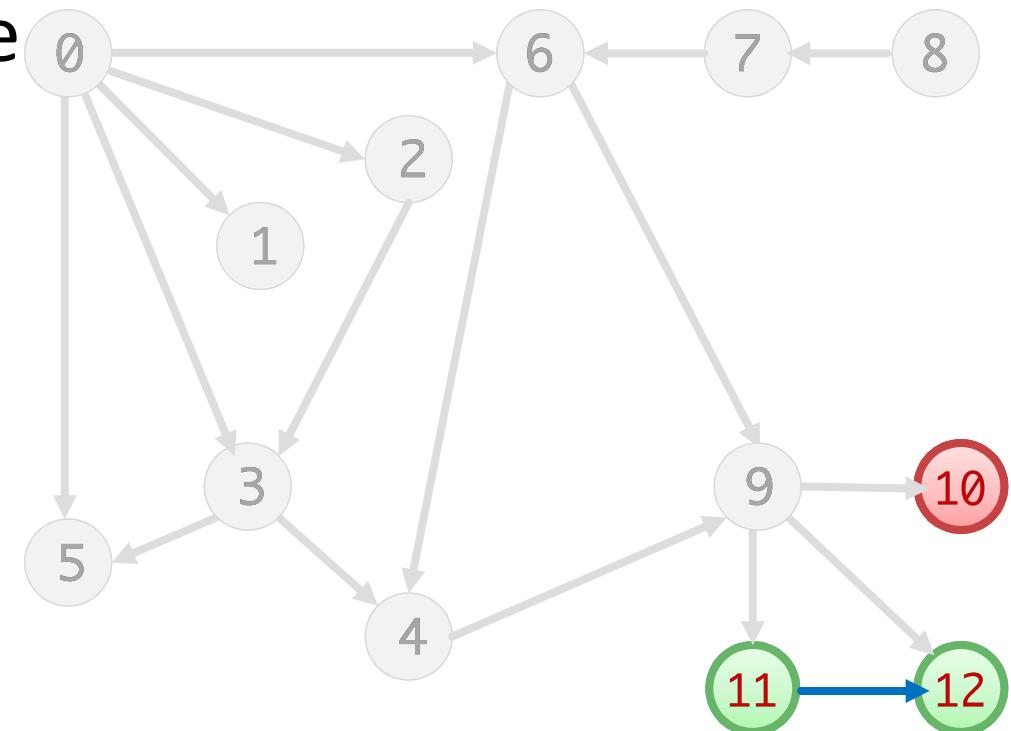
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5 < 9

# Orden Topológico en DiGrafos (IV<sub>23</sub>)

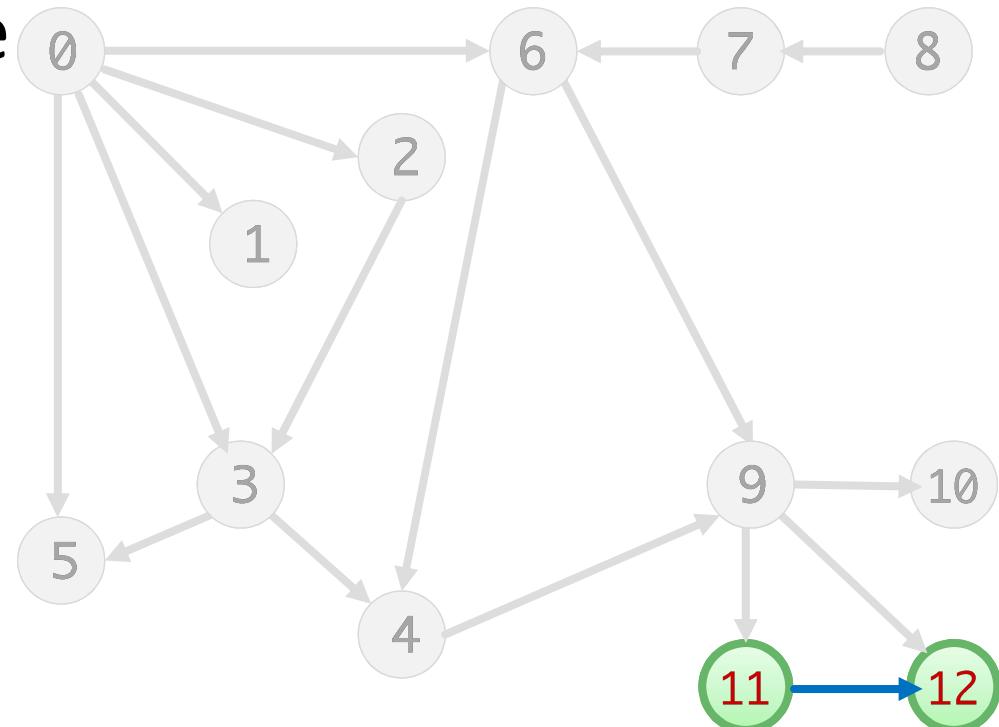
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5 < 9 < 10

# Orden Topológico en DiGrafos (IV<sub>24</sub>)

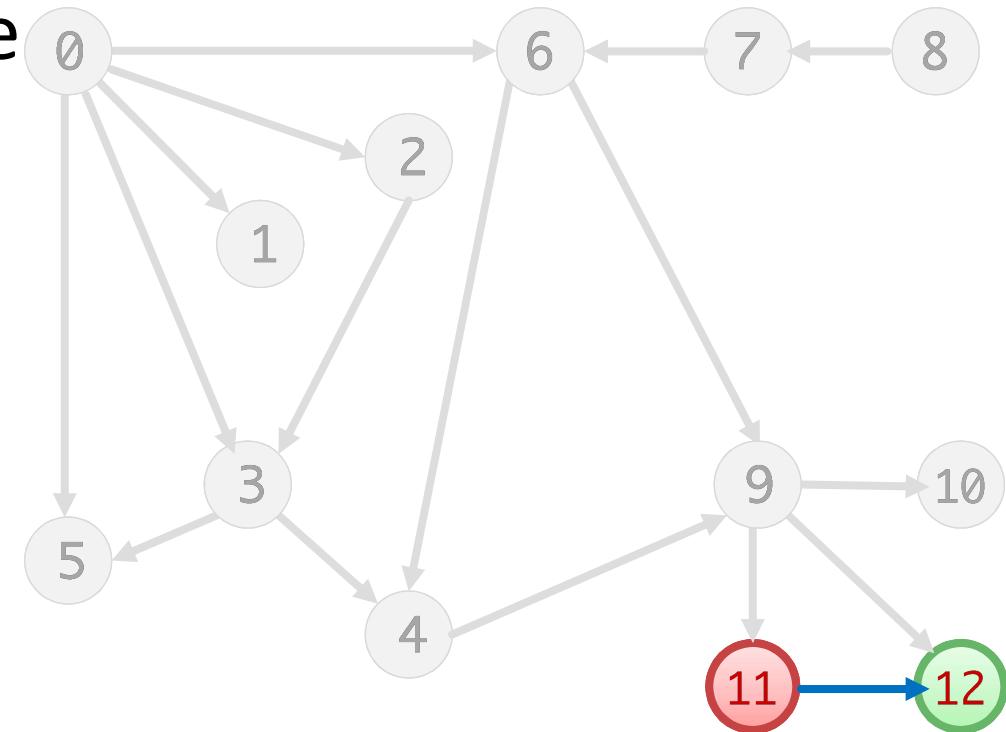
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5 < 9 < 10

# Orden Topológico en DiGrafos (IV<sub>25</sub>)

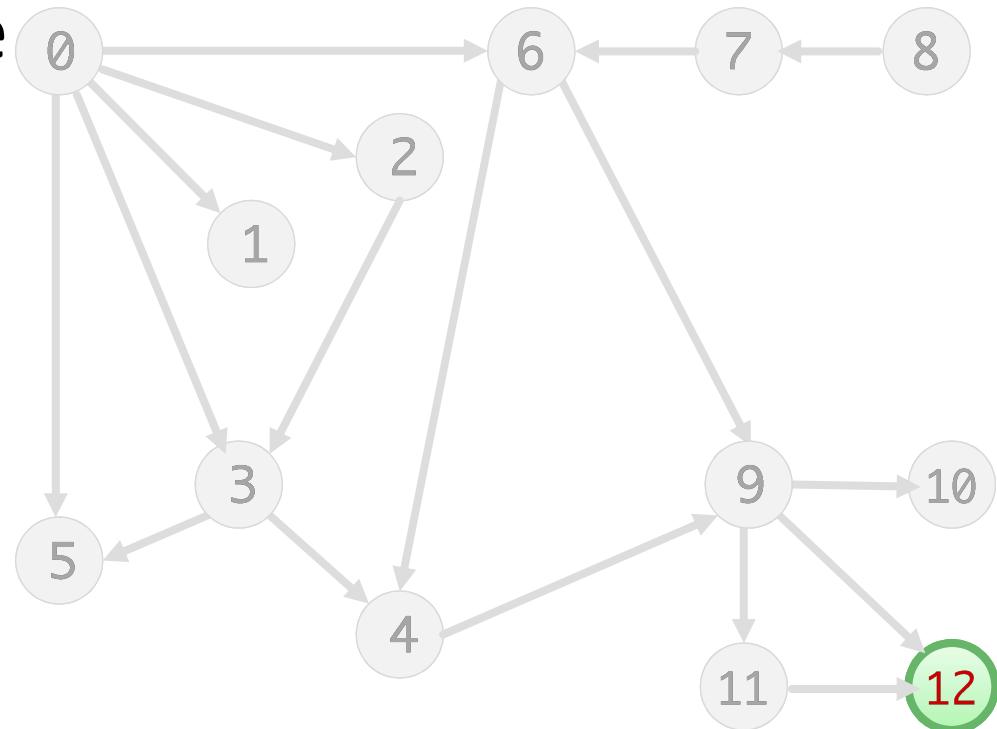
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5 < 9 < 10 < 11

# Orden Topológico en DiGrafos (IV<sub>26</sub>)

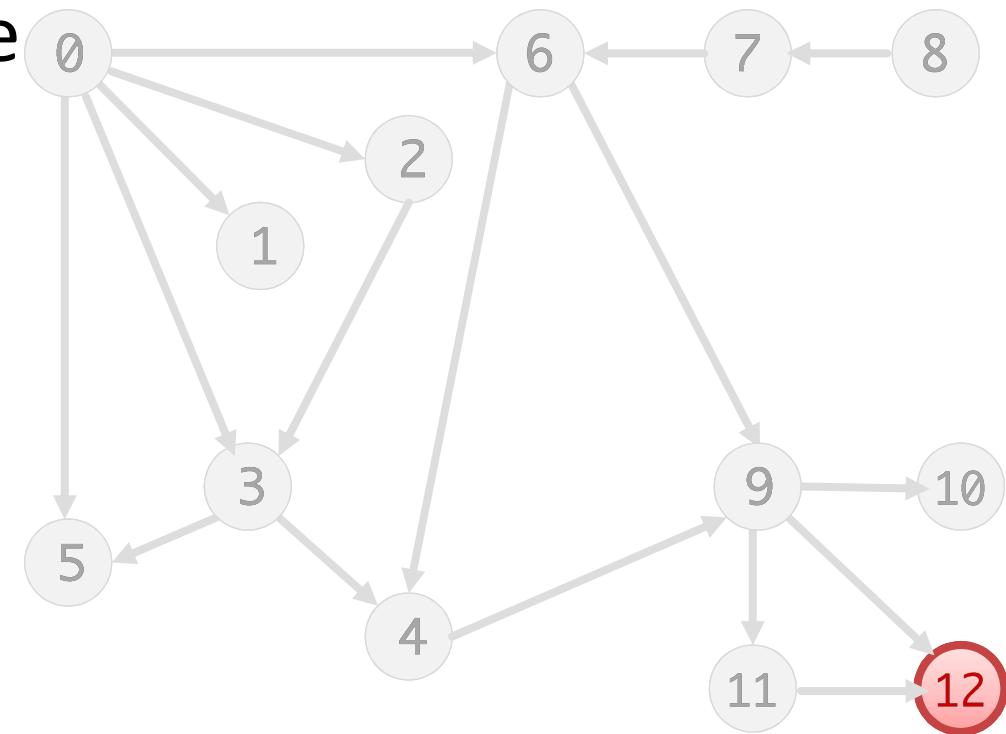
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5 < 9 < 10 < 11

# Orden Topológico en DiGrafos (IV<sub>27</sub>)

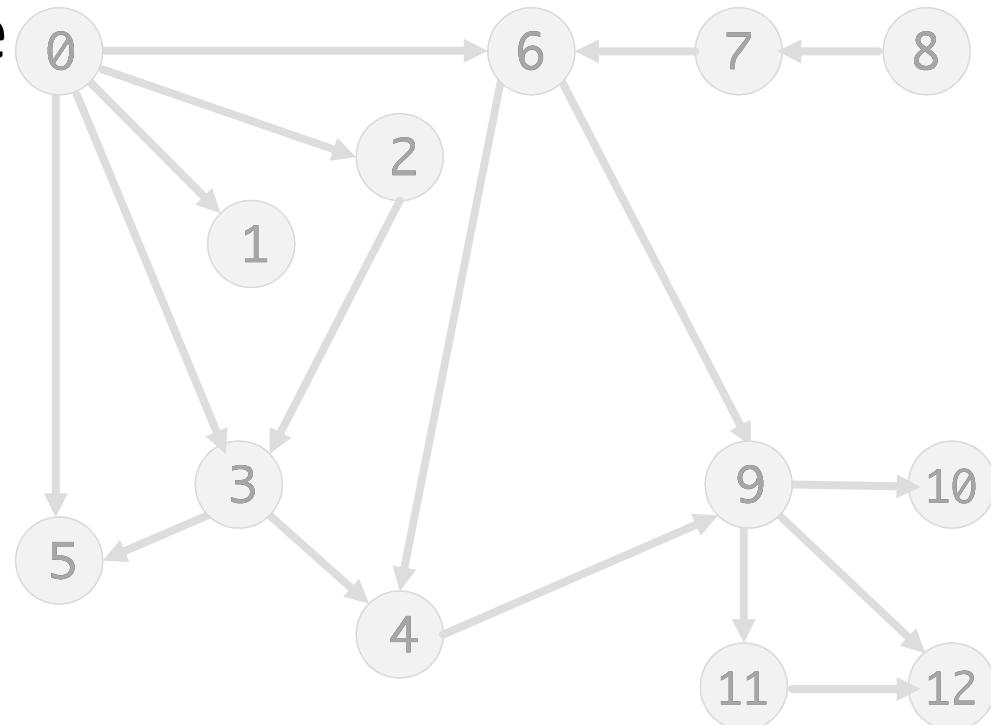
- Tomamos una fuente



Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5 < 9 < 10 < 11 < 12

# Orden Topológico en DiGrafos (IV<sub>28</sub>)

- Eliminamos la fuente y los arcos correspondientes

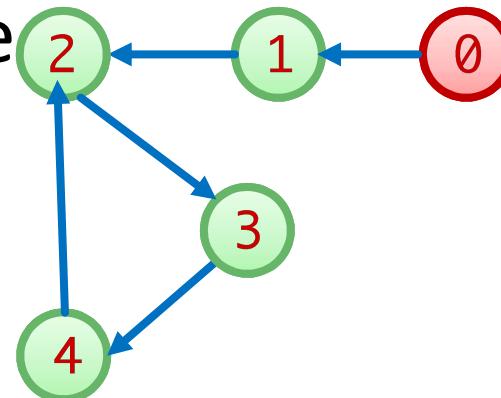


Se han añadido todos los vértices. Se ha computado el Orden Topológico

Orden Topológico: 8 < 0 < 7 < 2 < 3 < 6 < 1 < 4 < 5 < 9 < 10 < 11 < 12

# Orden Topológico. DiGrafo cíclico

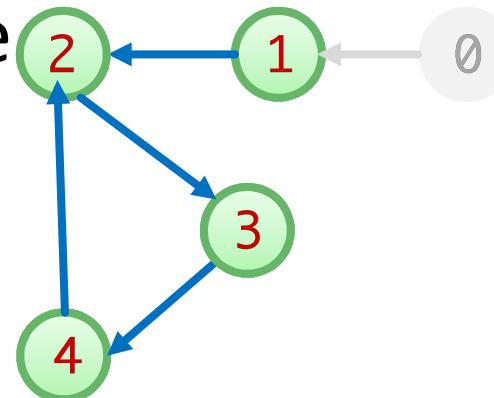
- Tomamos una fuente



Orden Topológico: 0

# Orden Topológico. DiGrafo cíclico<sub>1</sub>

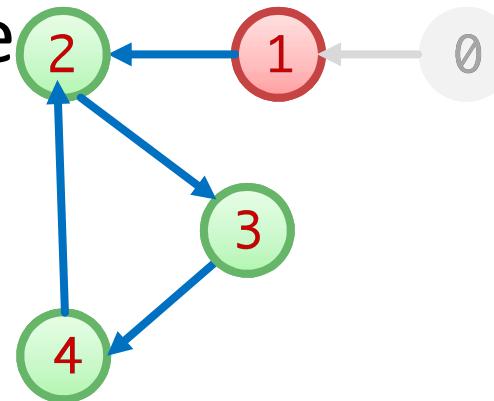
- Eliminamos la fuente y los arcos correspondientes



Orden Topológico: 0

# Orden Topológico. DiGrafo cíclico<sub>2</sub>

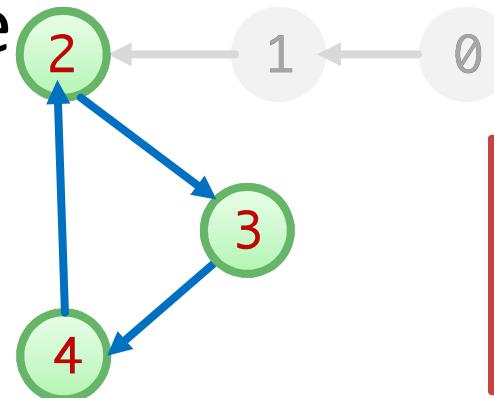
- Tomamos una fuente



Orden Topológico:  $0 < 1$

# Orden Topológico. DiGrafo cíclico<sub>3</sub>

- Eliminamos la fuente y los arcos correspondientes



No hay más fuentes. El DiGrafo tiene un ciclo y no existe un Orden Topológico

- En resumen: después de eliminar en forma reiterada todos los vértices fuentes, llegamos a un subgrafo sin fuentes, y
  - - si éste es no vacío, es cíclico así como el original.
  - - si éste es vacío, el grafo original es acíclico y los vértices eliminados forman un orden topológico.
  - Probaremos esto a continuación

# Ciclos y Orden Topológico. Corrección

Un vértice  $v$  se dice **fuente** si no es destino de ningún arco.

**Un digrafo es cíclico si y sólo si contiene un subgrafo finito no vacío sin fuentes.**

Demostración.-

- (i) Si  $G$  es cíclico entonces contiene un subgrafo finito no vacío sin fuentes (p.e., un ciclo).
- (ii) Si un grafo no vacío no tiene ningún vértice fuente, entonces es cíclico.

En efecto.- Sea  $G$  un digrafo no vacío y sea  $S_0$  un vértice. Por ser no fuente existe  $S_1 \prec S_0$ . Como  $S_1$  no es fuente, existirá otro vértice  $S_2 \prec S_1$ ; así construimos una sucesión de vértices

$$S_n \prec \dots \prec S_1 \prec S_0$$

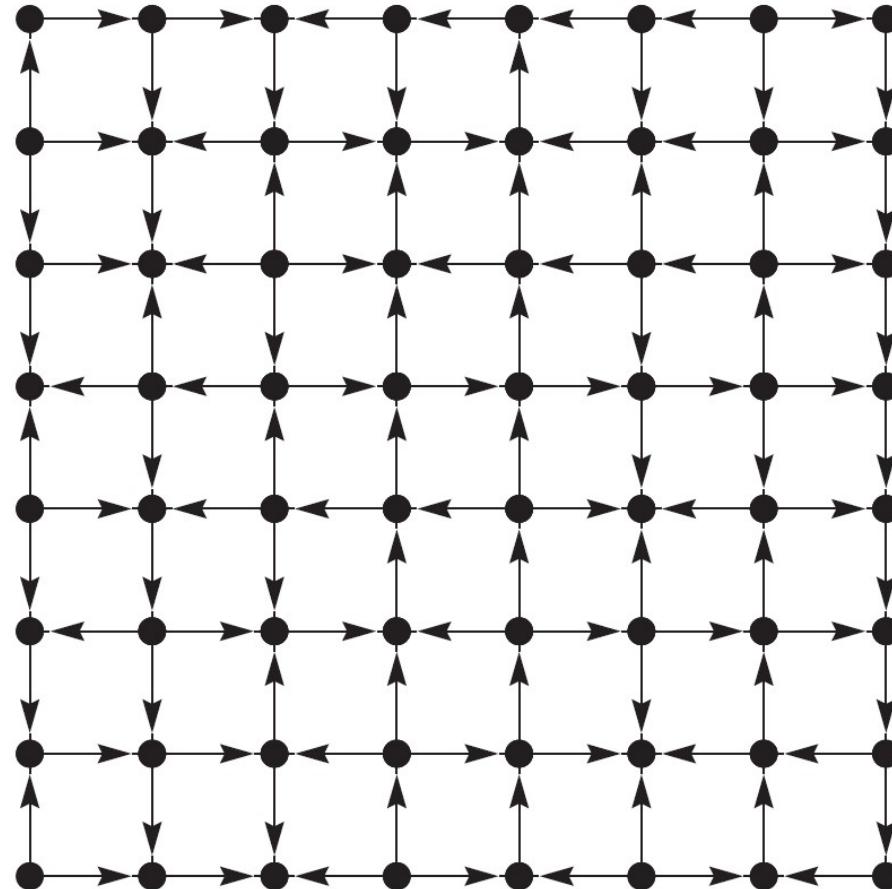
Puesto que el grafo es finito, se repetirá algún vértice en esta sucesión, es decir, existen dos valores  $k$  y  $c$  tales que  $S_{k+c} = S_k$ , y necesariamente la siguiente sucesión es un ciclo

$$S_{k+c} \prec S_{k+c-1} \prec \dots \prec S_k$$

Lo anterior conduce a un algoritmo **correcto** para construir un orden topológico o un ciclo.

# Ciclos en DiGrafos

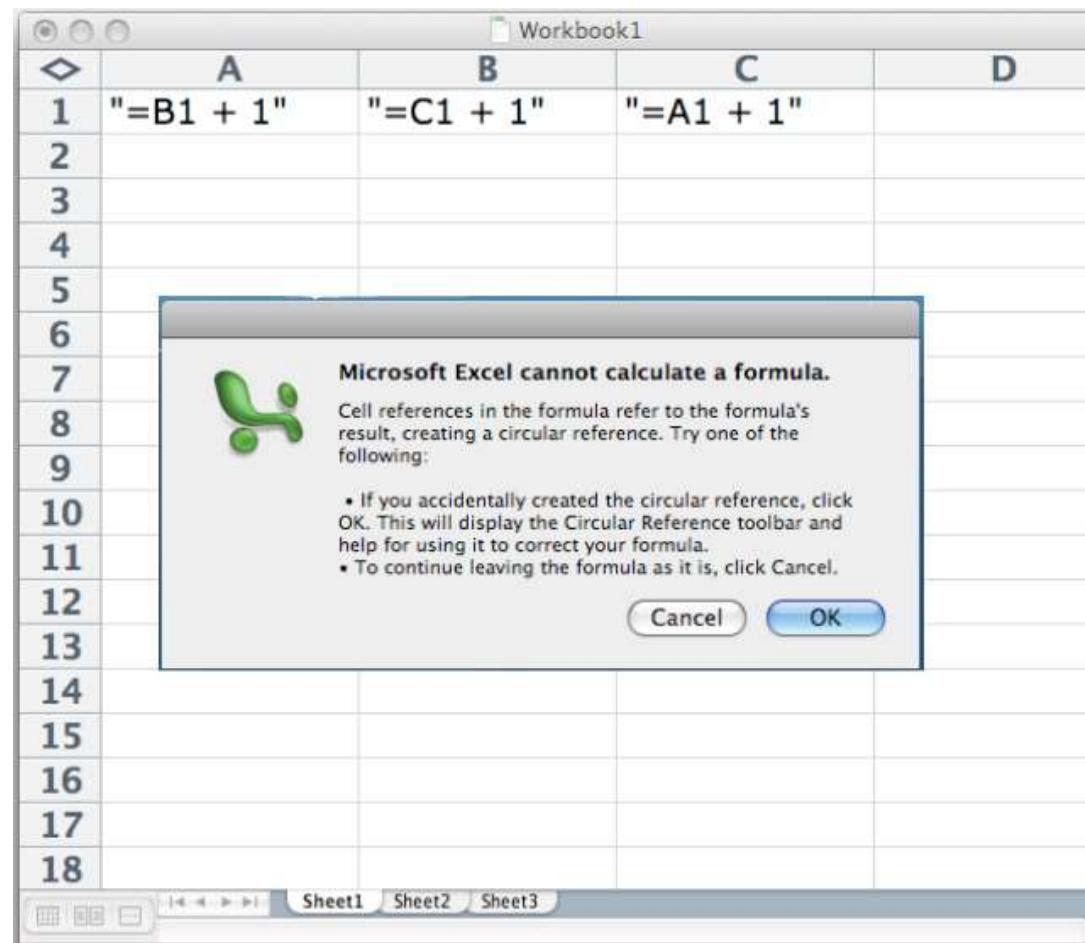
- ¿Tiene este grafo un ciclo?



Identificar ciclos en  
un DiGrafo puede  
ser un reto 😞

# Ciclos en DiGrafos (II)

- Microsoft Excel hace detección de ciclos 😊



# Ciclos en DiGrafos (III)

- El compilador de Java detecta ciclos 😊

```
public class A extends B {  
    ...  
}  
  
public class B extends C {  
    ...  
}  
  
public class C extends A {  
    ...  
}
```

```
% javac A.java  
A.java:1: cyclic inheritance  
involving A  
public class A extends B {}  
^  
1 error
```

# Ciclos en DiGrafos (IV)

- El sistema de ficheros de Linux **no** detecta ciclos



```
% ln -s a.txt b.txt  
% ln -s b.txt c.txt  
% ln -s c.txt a.txt  
% more a.txt  
a.txt: Too many levels of symbolic links
```

# Orden Topológico en Java

```
public class TopologicalSorting<V> {  
    private List<V> order;  
    private boolean cycle;  
  
    public TopologicalSorting(DiGraph<V> graph) {  
        order = new ArrayList<>();  
        cycle = false;  
  
        DiGraph<V> g = (DiGraph<V>) graph.clone();  
        while(!cycle && !g.vertices().isEmpty()) {  
            V next = null;  
            for(V v : g.vertices())  
                if(g.inDegree(v)==0) {  
                    next = v;  
                    break;  
                }  
  
            if(next!=null) {  
                order.append(next);  
                //also deletes corresponding edges  
                g.deleteVertex(next);  
            }  
            else  
                cycle = true;  
        }  
    }  
}
```

```
public boolean hasCycle() {  
    return cycle;  
}  
  
public List<V> order() {  
    return cycle ? null : order;  
}
```

# Orden Topológico en Java. Ejemplo

```
DiGraph<Integer> g = new DictionaryDiGraph<>();  
  
g.addVertex(2);  
g.addVertex(3);  
g.addVertex(5);  
g.addVertex(7);  
g.addVertex(8);  
g.addVertex(9);  
g.addVertex(10);  
g.addVertex(11);  
  
g.addDiEdge(7,5);  
g.addDiEdge(7,8);  
g.addDiEdge(5,11);  
g.addDiEdge(3,8);  
g.addDiEdge(3,10);  
g.addDiEdge(11,2);  
g.addDiEdge(11,9);  
g.addDiEdge(11,10);  
g.addDiEdge(8,9);  
  
TopologicalSorting<Integer> topSort = new TopologicalSorting<>(g);  
if(!topSort.hasCycle())  
    System.out.println("A topological sorting: "+topSort.order());  
else  
    System.out.println("DiGraph is cyclic");
```

# Ciclos y Orden Topológico en Haskell (I)

Un algoritmo **correcto** para construir un orden topológico o un ciclo.

```
topologicalSortAndCycle :: Eq v => DiGraph v -> (Maybe [v], Maybe (Path v))
topologicalSortAndCycle g
| null (DiGraph.vertices g') = (Just orderTop, Nothing)
| otherwise                  = (Nothing, Just $ extractCycle g')
```

where

```
(g', orderTop) = collectSources g []
```

```
collectSources :: Eq v => DiGraph v -> [v] -> (DiGraph v, [v])
```

```
collectSources g as
```

```
| null ss  = (g,as)
```

```
| otherwise = collectSources newg (as++ss)
```

where

```
ss = sources g
```

```
newg = deleteVertices ss g
```

```
deleteVertices :: Eq v => [v] -> DiGraph v -> DiGraph v
deleteVertices xs (DG vs suc) = DG (vs \\ xs) suc'
where suc' v = suc v \\ xs
```

```
extractCycle :: Eq v => DiGraph v -> [v]
```

```
extractCycle g = extractCycleAux (head $ DiGraph.vertices g) []
```

where

```
extractCycleAux v as
```

```
| v `elem` as = v: takeWhile (/=v) as ++ [v]
```

```
| otherwise    = extractCycleAux (head $ predecesors g v) (v:as)
```

# Ciclos y Orden Topológico en Haskell (II)

Otro programa simplificado para calcular un Orden Topológico

```
-- returns vertices in vs with no predecessor (in vs)
noPreds :: (Eq a) => DiGraph a -> [a] -> [a]
noPreds g vs = [ v | v <- vs, null (preds v) ]
where
    -- returns predecessors (in vs) of v
    preds v = [ w | w <- vs, v `elem` successors g w ]

topSorting :: (Ord a) => DiGraph a -> [a]
topSorting g = aux (vertices g)
where
    aux [] = []
    aux vs
        | null ws  = error "DiGraph is cyclic"
        | otherwise = v : aux (vs \\ [v])
    where
        ws = noPreds g vs
        v = head ws
```

# Ciclos y Orden Topológico en Haskell (III)

Otro programa *sencillo* para calcular un Orden Topológico

```
topologicalSort :: (Ord a) => DiGraph a -> [a]
topologicalSort g = tSort [ v | v <- vertices g,
                             inDegree g v == 0 ]
                     []
```

where

```
tSort []      vis = vis
tSort (v:vs)  vis
| v `elem` vis = tSort vs vis
| otherwise     = tSort vs (v:tSort (successors g v) vis)
```

Corrección.- por inducción ...

# Ciclos y Orden Topológico en Haskell (IV)

- Corrección.- Es fácil demostrar por inducción (sobre el primer argumento de tSort) que:
  - Cada vértice del segundo argumento (*vis*) aparecerá detrás de cada vértice del primer argumento (*v:vs*).
  - Por tanto, la expresión
    - $tSort\ vs\ (v:tSort\ (successors\ g\ v)\ vis)$  significa que *v* aparecerá antes que sus sucesores (*successors g v*)
- La siguiente relación  $\prec$  entre subconjuntos (listas) de vértices es inductiva si el grafo es acíclico y finito:

$$\frac{W \subset V}{W \prec V} \qquad \frac{x \in V}{(successors x) \prec V}$$

( donde  $W \subset V$  denota una inclusión propia)

# Grafos: Problemas Interesantes (I)

- **Conexión**: ¿Existe un camino entre dos vértices?
- **Camino más corto**: ¿Cuál es el más corto?
- **Camino más largo**: ¿Cuál es el camino más largo?
- **Detección de ciclos**: ¿Existen ciclos en un grafo?
- **Ciclo de Euler**: ¿Existe un ciclo que recorra cada arista una sola vez?
- **Ciclo de Hamilton**: ¿Existe un ciclo que recorra cada vértice una sola vez?

# Grafos: Problemas Interesantes (II)

- **2-colores**: ¿Pueden colorearse los vértices con dos colores de manera que dos adyacentes tengan colores distintos?  
¿Es un grafo bipartito?
- **Árbol de Expansión Mínimo**: para un grafo con pesos, encontrar un árbol de expansión con coste total mínimo.
- **Biconectividad**: ¿Hay algún vértice que al quitarlo desconecte el grafo?
- **Planaridad**: ¿Puede pintarse un grafo en un plano sin que se crucen las aristas?
- **Isomorfismo de Grafos**: ¿Son dos grafos idénticos salvo el renombrado de vértices?

# Grafos: Problemas Interesantes (III)

## ■ Dificultad de algunos problemas en Grafos

	Efficient	Intractable (NP-hard)	Unknown
Shortest paths	*		
Longest paths		*	
Euler tour	*		
Hamilton tour		*	
Minimum spanning tree	*		
Traveling Salesperson		*	
Isomorphism			*

Problemas similares  
pueden tener  
dificultades diferentes

# Material Complementario

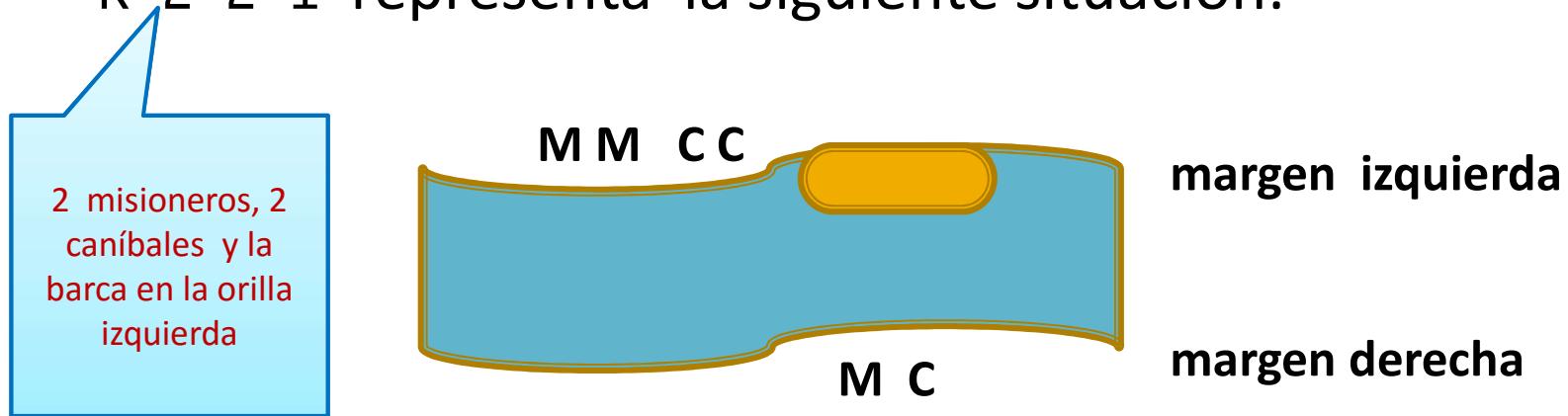
Para profundizar

# El problema de los caníbales y misioneros (I)

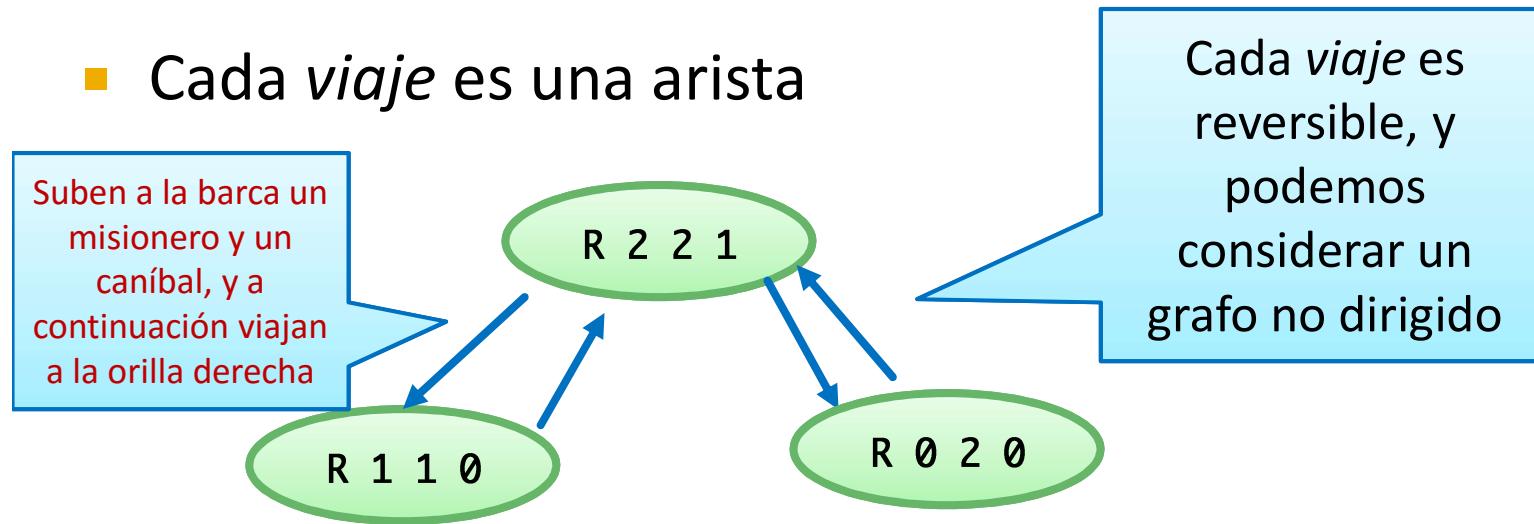
- Tres misioneros y tres caníbales se encuentran en la orilla izquierda de un río y quieren pasar a la orilla derecha con ayuda de una barca.
- Si la barca puede transportar un máximo de dos personas, describid el juego de movimientos más corto si en ningún momento, ni en la barca ni en cualquier orilla, puede haber más caníbales que misioneros (de lo contrario se los comen).
- El problema se reduce a la exploración **bft** del grafo donde cada vértice viene dado por las personas en la orilla izquierda y la posición de la barca:
- `type PosiciónBarca = Int -- 1 denota la orilla izda`
- `type Caníbales = Int`
- `type Misioneros = Int`
- `data Río = R Misioneros Caníbales PosiciónBarca`

# El problema de los caníbales y misioneros (II)

- R 2 2 1 representa la siguiente situación:



- Cada *viaje* es una arista



# El problema de los caníbales y misioneros (III)

La solución (solo hay una óptima) con 11 viajes (aristas)

```
*Barca> bftPathsTo gRio (R 3 3 1) (R 0 0 0)
[[R 3 3 1,R 2 2 0,R 3 2 1,R 3 0 0,R 3 1 1,R 1 1 0,R 2 2 1,R 0 2
0,R 0 3 1,R 0 1 0,R 1 1 1,R 0 0 0]]
```

```
*Barca> map length $ bftPathsTo gRio (R 3 3 1) (R 0 0 0)
[12]
```

Construcción del grafo

```
tcm = 3 -- total de caníbales y de misioneros
total_barca = 2 -- en la barca caben 2
```

```
gRio = mkGraphSuc situaciones suc
situaciones = [ R m c p | c<-[0..tcm], m<-[0..tcm],
                  no_come c m, no_come (tcm-c) (tcm-m), p<-[0,1] ]
```

```
no_come      :: Caníbales -> Misioneros -> Bool
no_come _ 0 = True
no_come c m = m >= c
```

# El problema de los caníbales y misioneros (IV)

La función que construye los sucesores será:

```
suc (R m c 1) =  
  [ R m' c' 0 | (mb,cb)<- pares,  
    let m'=m-mb, m'>=0,  
    let c'=c-cb, c'>=0,  
    no_come c' m',  
    no_come (tcm-c') (tcm-m') ]
```

```
suc (R m c 0) =  
  [ R m' c' 1 | ... ]
```

```
pares = [ (m,c) |  
  suben <- [1..total_barca], -- suben > 0  
  m<-[0..suben], let c = suben-m,  
  no_come c m]
```

# El problema de los caníbales y misioneros (V)

- Estudio de las componentes conexas (transp. 229) para  $(tcm, total\_barca) = (3,2)$

```
*Barca> map length $ dftConnectedComps gRio  
[16,1,1,1,1]
```

Es decir, hay 5 componentes conexas, y cuatro de ellas son 4 vértices aislados (imposible viajar):

```
*Barca> filter (\x -> length x ==1 ) (dftConnectedComps gRio)  
[[R 0 0 1],[R 3 0 1],[R 0 3 0],[R 3 3 0]]
```

El *spanning tree* bft es muy degenerado:

```
*Barca> map length $ bftPaths gRio (R 3 3 1)  
[1,2,2,2,3,4,5,6,7,8,9,10,11,11,12,13]
```

y los vértices más alejados son:

- \*Barca> reverse \$ bft gRio (R 3 3 1)  
■ [R 0 1 1,R 0 0 0,R 0 2 1, . . .

# Puzzle de las jarras

- Disponemos de dos jarras ( $X$  e  $Y$ ) con las correspondientes capacidades  $capX$  y  $capY$  en litros

- Las jarras no tienen marcas

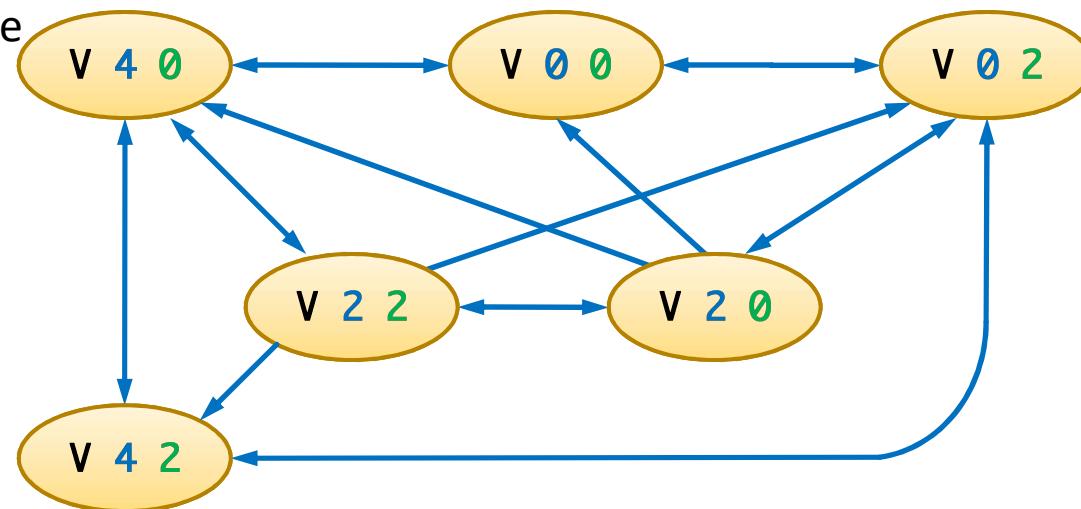
- Operaciones permitidas:
  - Las jarras puede llenarse completamente
  - Las jarras pueden vaciarse completamente
  - El contenido de una jarra puede ser decantada en la otra
    - Bien totalmente (dejando la primera jarra vacía)
    - o parcialmente (se completa la segunda)



¿ Es posible aislar **UN** litro en alguna jarra? ¿ y **1** litros ?

# Puzzle de las jarras (II)

- El puzzle puede representarse como un DiGrafo
- El vértice  $V \ x \ y$  corresponde a la configuración donde la primera jarra contiene  $x$  litros y la segunda contiene  $y$  litros
- Habrá un arco de un vértice  $V \ x \ y$  a otro  $V \ x' \ y'$  si es posible ir de una configuración a la otra realizando un único transvase permitido
- Para aislar  $l$  litros, buscaremos un camino dirigido desde el vértice  $V \ 0 \ 0$  hasta el vértice  $V \ l \ y$  o hasta el  $V \ x \ l$
- DiGrafo correspondiente al puzzle de las jarras con 4 y 2 litros:



# Puzzle de las jarras (III)

```
data Jugs = V Int Int deriving (Eq, Ord, Show)
```

```
jugs :: Int -> Int -> DiGraph Jugs
```

```
jugs capX capY = mkDiGraphSuc undefined suc  
where
```

```
suc v = nub [ op v | op <- ops ] \\ [v]
```

Los vértices no están predefinidos

```
ops = [fillX, fillY, emptyX, emptyY, decantXY, decantYX]
```

```
fillX (V _ y) = V capX y
```

nub y \\ están en la librería Data.List:

```
fillY (V x _) = V x capY
```

nub [1,2,1,3,2] => [1,2,3]

```
emptyX (V _ y) = V 0 y
```

[1,2,3,4] \\ [1,3] => [2,4]

```
emptyY (V x _) = V x 0
```

```
decantXY (V x y) -- decant X to Y
```

| s <= capY = V 0 s -- X can be totally decanted

| otherwise = V (s-capY) capY

```
where s = x + y
```

```
decantYX (V x y) -- decant Y to X
```

| s <= capX = V s 0

| otherwise = V capX (s-capX)

```
where s = x + y
```

Función con los sucesores eliminando repetidos y ciclos

# Puzzle de las jarras (IV)

```
contains1Litre :: Jugs -> Bool  
contains1Litre (V x y) = x==1 || y==1
```

```
isSol :: Path Jugs -> Bool  
isSol vs = contains1Litre (last vs)
```

```
sols :: [Path Jugs]  
sols = filter isSol (bftPathsTo (jugs 3 5) (V 0 0))
```

La búsqueda bft proporciona los caminos ordenados por su longitud

```
Main> head sols  
[V 0 0, V 3 0, V 0 3, V 3 3, V 1 5]
```

Una solución óptima para aislar UN litro

```
Main> gcd 3 7  
1
```

```
Main> sort . nub $ [ l | V x y <- bft (jugs 3 7) (V 0 0), l <- [x,y] ]  
[0,1,2,3,4,5,6,7]
```

```
Main> gcd 3 9  
3
```

Se aislan todos los múltiplos del  $MCD(capX, capY)$

```
Main> sort . nub $ [ l | V x y <- bft (jugs 3 9) (V 0 0), l <- [x,y] ]  
[0,3,6,9]
```