# Algoritmos y Sistemas Computacionales

Repaso para:
Examen

Por Ale Tapia!!!!! (2023)
Aportes de zabi, seba y cabra uwu

# Temas a Estudiar:

- **Grafos:**
  - Grafos y su representación.

  Algoritmos de Búsqueda :
  - BFS
  - DFS
  - Dijkstra

- Grande “Grokking Algorithms” <33

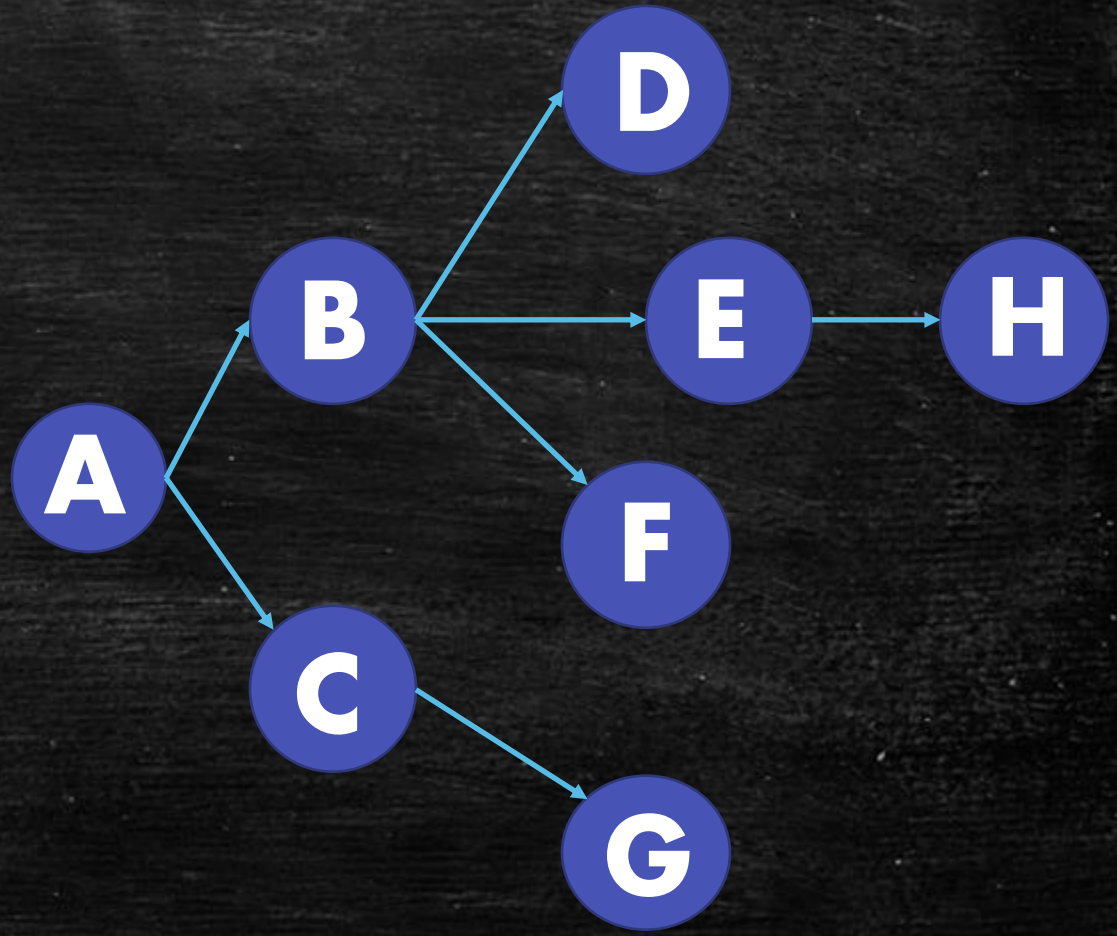
# Grafos

- Un grafo modela un grupo de conexiones entre distintos “nodos” o “vértices”, cuyas uniones se indican con aristas o vectores. A los nodos unidos se les llama vecinos.
- Permite representar una serie de elementos conectados
- Puedes seguir un camino de nodos conectados, aunque no conecten directamente.

| Nodo | Vecino(s) |
|---|---|
| A = | [B, C] |
| B = | [D, F, E] |
| C = | [G] |
| D = | [] |
| E = | [] |
| F = | [H] |
| G = | [] |
| H = | [] |

# Grafos

- Un grafo modela un grupo de conexiones entre distintos “nodos”, cuyas uniones se indican con vectores. A los nodos unidos se les llama vecinos.
- De manera que:
- (Cada grafo puede ser representado de muchas maneras)



Y al cabra tmbn gracias que me apañó caleta con algoritmos de búsqueda <3

# Algoritmos de Búsqueda

- Los algoritmos de Búsqueda son usados para encontrar información sobre la distancia entre Nodos en un Grafo.
- Ahora revisaremos 3 tipos de algoritmos de búsqueda, con el pseudocódigo visto en clases, paso a paso:
- BFS (Breadth-First Search)
- DFS (Depth-First Search)
- Dijkstra

| Nombre | Diferencias |
|---|---|
| BFS | Queue |
| DFS | Stack |
| Dijkstra | Aristas con Peso |

# BFS

Primero veremos, que elementos utiliza este algoritmo.

Recibe G, que es un grafo. La V y la E, representan todos los vértices (V de Vertex) y aristas (E de Edge). También recibe "node" que es nuestro nodo inicial.

Y nos entrega "visited", que es una lista con todos los valores True o False asignados a cada nodo si fueron revisados por el algoritmo. Esto nos dirá a qué elementos podemos llegar desde el nodo con el que comencemos.

De esta manera podemos averiguar el camino con menos aristas para llegar a otro nodo.

**Algorithm 2.5:** Graph breadth-first search.

```
BFS(G, node) → visited
    Input: G = (V, E), a graph
           node, the starting vertex in G
    Output: visited, an array of size |V| such that visited[i] is TRUE if we
            have visited node i, FALSE otherwise
1   Q ← CreateQueue()
2   visited ← CreateArray(|V|)
3   inqueue ← CreateArray(|V|)
4   for i ← 0 to |V| do
5       visited[i] ← FALSE
6       inqueue[i] ← FALSE
7   Enqueue(Q, node)
8   inqueue[node] ← TRUE
9   while not IsQueueEmpty(Q) do
10      c ← Dequeue(Q)
11      inqueue[c] ← FALSE
12      visited[c] ← TRUE
13      foreach v in AdjacencyList(G, c) do
14          if not visited[v] and not inqueue[v] then
15              Enqueue(Q, v)
16              inqueue[v] ← TRUE
17  return visited
```

# BFS

Luego, vemos en el código que se crea "Q", que es una lista de tipo Queue (que comienza vacía), "visited" e "inqueue", que son Arrays que contienen tantos elementos como nodos en el grafo.

Luego comienza un ciclo "for" que asigna a cada nodo en "visited" e "inqueue" el valor "False".

Lo siguiente es añadir al Queue nuestro nodo inicial y asignarle el valor True en inqueue para indicar que está dentro de Q.

**Algorithm 2.5:** Graph breadth-first search.

```
BFS(G, node) → visited
    Input: G = (V, E), a graph
           node, the starting vertex in G
    Output: visited, an array of size |V| such that visited[i] is TRUE if we
            have visited node i, FALSE otherwise
1   Q ← CreateQueue()
2   visited ← CreateArray(|V|)
3   inqueue ← CreateArray(|V|)
4   for i ← 0 to |V| do
5       visited[i] ← FALSE
6       inqueue[i] ← FALSE
7   Enqueue(Q, node)
8   inqueue[node] ← TRUE
9   while not IsQueueEmpty(Q) do
10      c ← Dequeue(Q)
11      inqueue[c] ← FALSE
12      visited[c] ← TRUE
13      foreach v in AdjacencyList(G, c) do
14          if not visited[v] and not inqueue[v] then
15              Enqueue(Q, v)
16              inqueue[v] ← TRUE
17  return visited
```

# BFS

Ahora comienza el ciclo principal, este ciclo While funcionará siempre que haya elementos en nuestro Q.

Lo primero que hace es crear una variable "c" y asignarle el valor que saca de nuestra Queue. Luego le asigna el valor False en el Array Inqueue, para indicar que lo sacamos de Q.

Luego, le asigna al actual elemento en c True en el Array visited, para recordar que ya fue visitado.

**Algorithm 2.5:** Graph breadth-first search.

```
BFS(G, node) → visited
    Input: G = (V, E), a graph
           node, the starting vertex in G
    Output: visited, an array of size |V| such that visited[i] is TRUE if we
            have visited node i, FALSE otherwise

1   Q ← CreateQueue()
2   visited ← CreateArray(|V|)
3   inqueue ← CreateArray(|V|)
4   for i ← 0 to |V| do
5       visited[i] ← FALSE
6       inqueue[i] ← FALSE

7   Enqueue(Q, node)
8   inqueue[node] ← TRUE
9   while not IsQueueEmpty(Q) do
10      c ← Dequeue(Q)
11      inqueue[c] ← FA
12      visited[c] ← TRUE
13      foreach v in AdjacencyList(G, c) do
14          if not visited[v] and not inqueue[v] then
15              Enqueue(Q, v)
16              inqueue[v] ← TRUE
17  return visited
```

# BFS

Ahora crea otro ciclo dentro del While, de tipo "For" que revisa cada nodo vecino a nuestro actual "c".

Si estos nodos no han sido visitados (not visited) ni están en la Queue (not inqueue), entonces lo que hará será añadirlos a Q, y cambiar su valor de Inqueue a True.

Una vez finalizado el ciclo principal While, el algoritmo devolverá el Array visited, en el que estarán todos los otros nodos que revisamos.

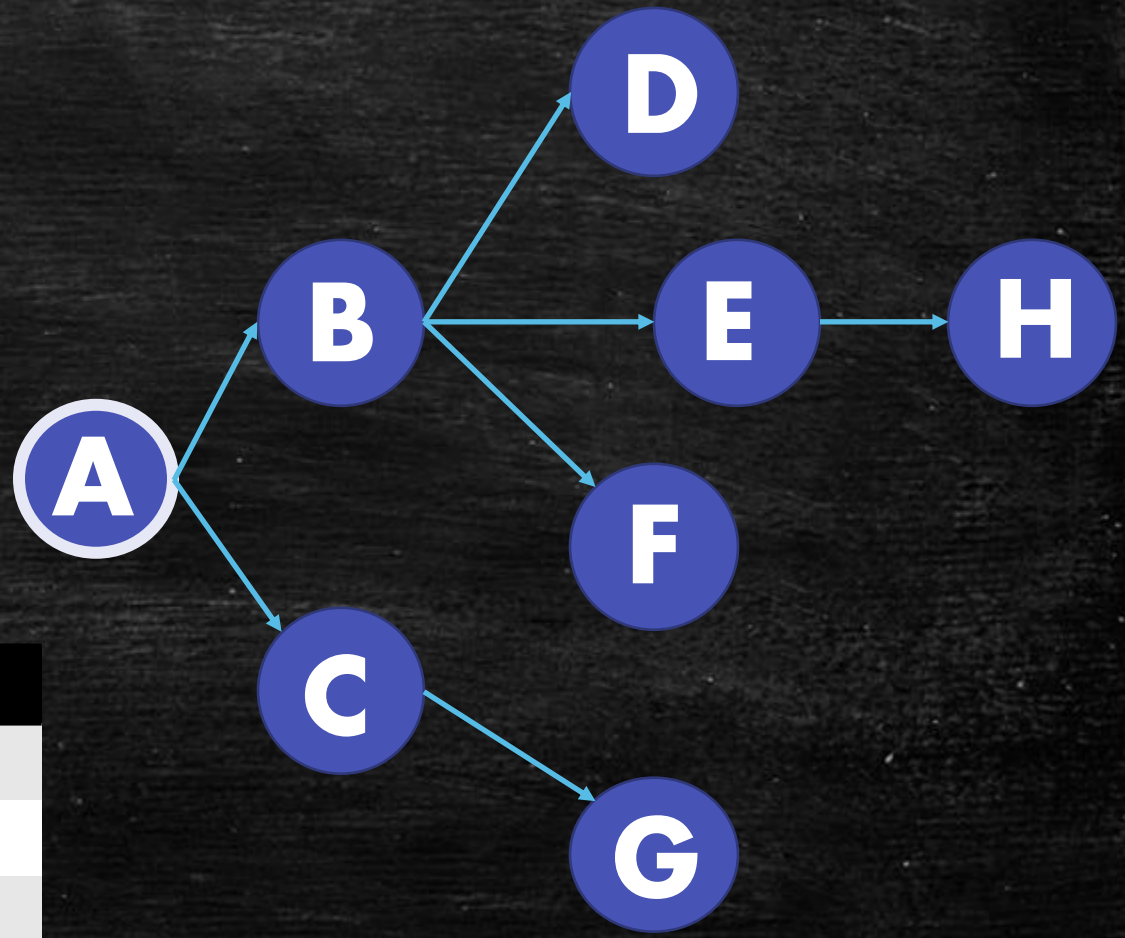
**Algorithm 2.5:** Graph breadth-first search.

```
BFS(G, node) → visited
    Input: G = (V, E), a graph
           node, the starting vertex in G
    Output: visited, an array of size |V| such that visited[i] is TRUE if we
            have visited node i, FALSE otherwise

1   Q ← CreateQueue()
2   visited ← CreateArray(|V|)
3   inqueue ← CreateArray(|V|)
4   for i ← 0 to |V| do
5       visited[i] ← FALSE
6       inqueue[i] ← FALSE

7   Enqueue(Q, node)
8   inqueue[node] ← TRUE
9   while not IsQueueEmpty(Q) do
10      c ← Dequeue(Q)
11      inqueue[c] ← FALSE
12      visited[c] ← TRUE
13      foreach v in AdjacencyList(G, c) do
14          if not visited[v] and not inqueue[v] then
15              Enqueue(Q, v)
16              inqueue[v] ← TRUE
17  return visited
```

# BFS

- Vamos a ver representado gráficamente el funcionamiento el algoritmo de BFS.
- De manera que este es nuestro G, y nuestro nodo inicial será A.
- Visited marcará todos los nodos como False. Añadiremos A a Q y le marcaremos True en inqueue.

| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | X | - | - | - | - | - | - | - |
| Visited | F | F | F | F | F | F | F | F |
| Inqueue | T | F | F | F | F | F | F | F |



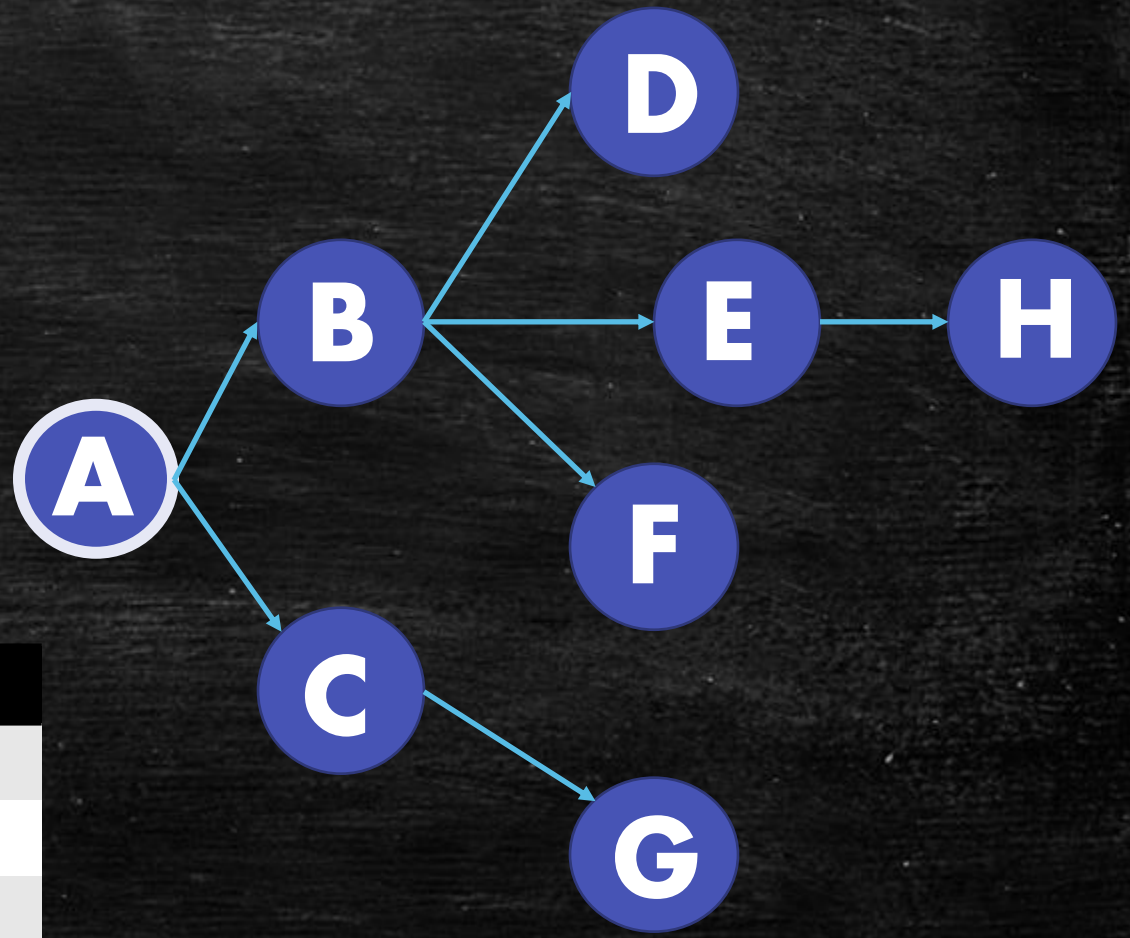
# BFS

- Ahora comenzamos el ciclo, partiendo por indicar que:

**c = A**

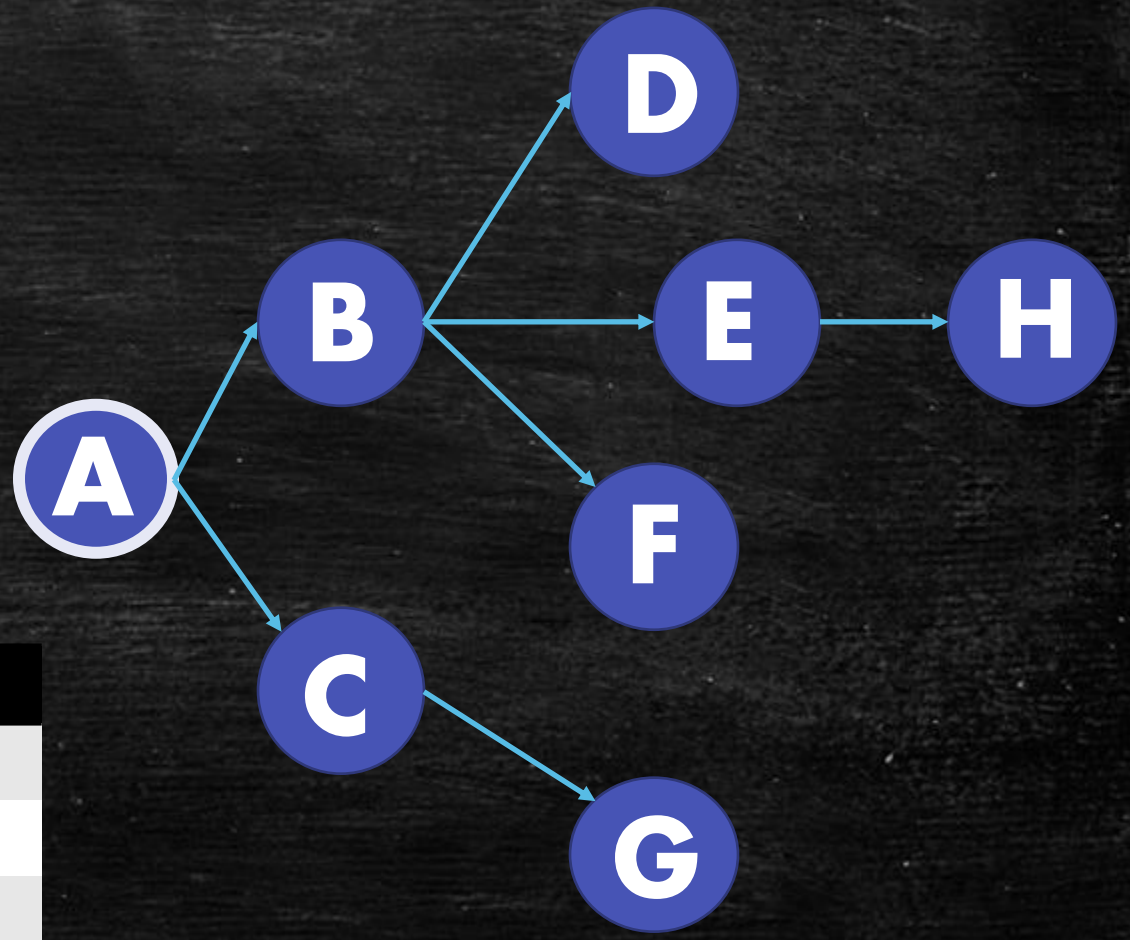
- Así que sacamos A de Q para luego marcamos False en Inqueue y True en Visited.

| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | - | - | - | - | - | - | - |
| Visited | **T** | F | F | F | F | F | F | F |
| Inqueue | **F** | F | F | F | F | F | F | F |



# BFS

- Y tenemos que ahora inicia el siguiente ciclo for, que añade a todos los nodos vecinos de c que no hayan sido visitados ni estén inqueued a Q, y los marca como True en inqueue.




| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | X | X | - | - | - | - | - |
| Visited | T | F | F | F | F | F | F | F |
| Inqueue | F | T | T | F | F | F | F | F |

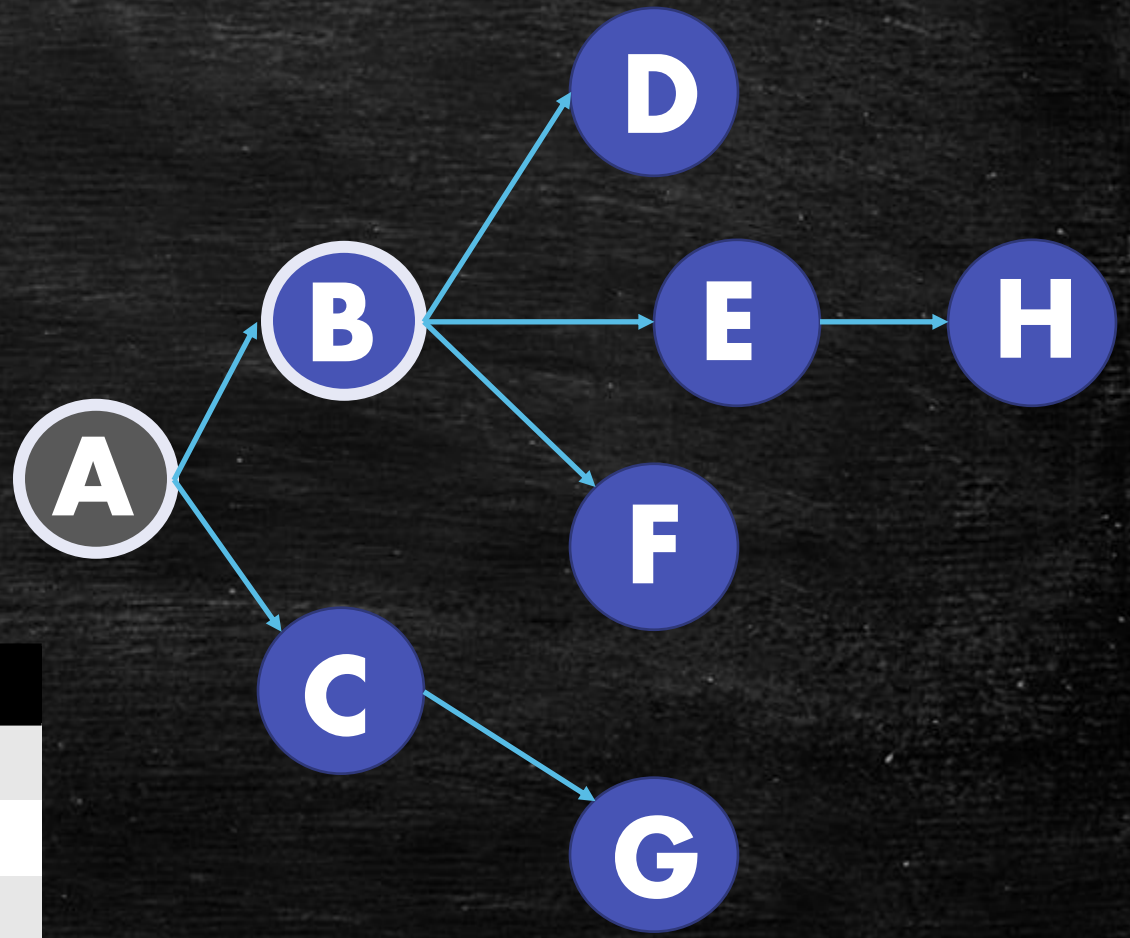
# BFS

- Ya que estamos en un Queue, el primero que entró es el primero que sacaremos, así que partiremos con B. Ahora:

**c = B**

- Y repetiremos el proceso, añadiendo D, E y F al Q.

| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | - | X | **X** | **X** | **X** | - | - |
| Visited | T | **T** | F | F | F | F | F | F |
| Inqueue | F | **F** | T | **T** | **T** | **T** | F | F |

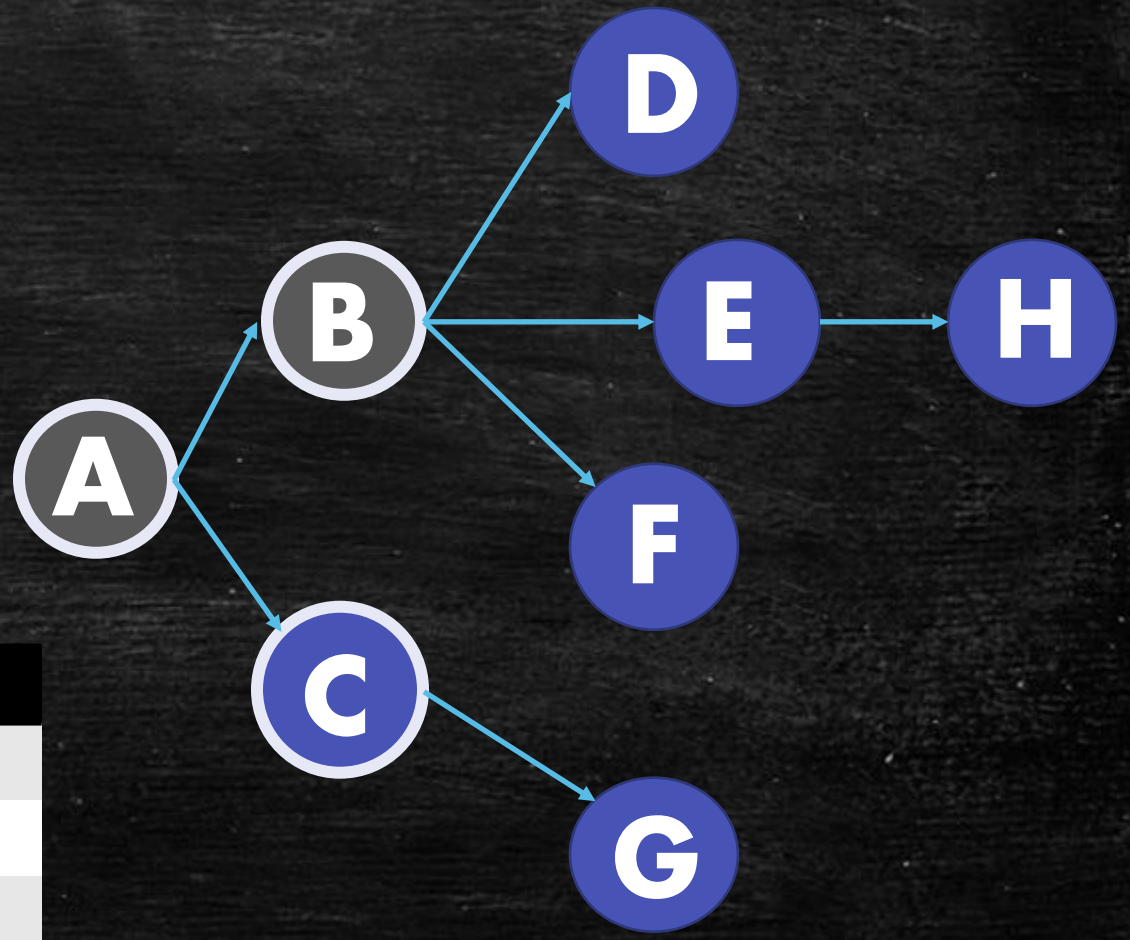


# BFS

- Ahora, por orden de Queue:

  **c = C**

- Y añadimos G a Q.

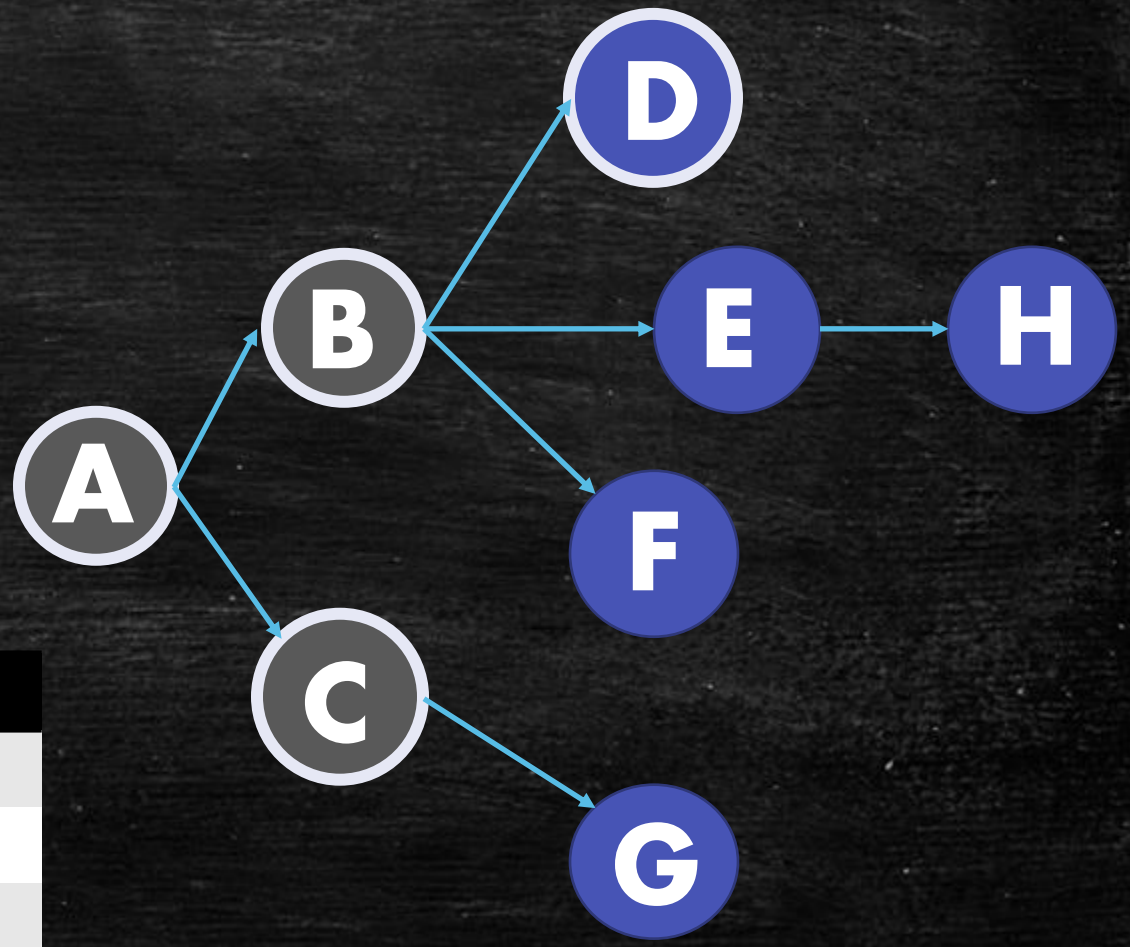



| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | - | - | X | X | X | **X** | - |
| Visited | T | T | **T** | F | F | F | F | F |
| Inqueue | F | F | **F** | T | T | T | **T** | F |

# BFS

- Ahora, por orden de Queue:

c = D



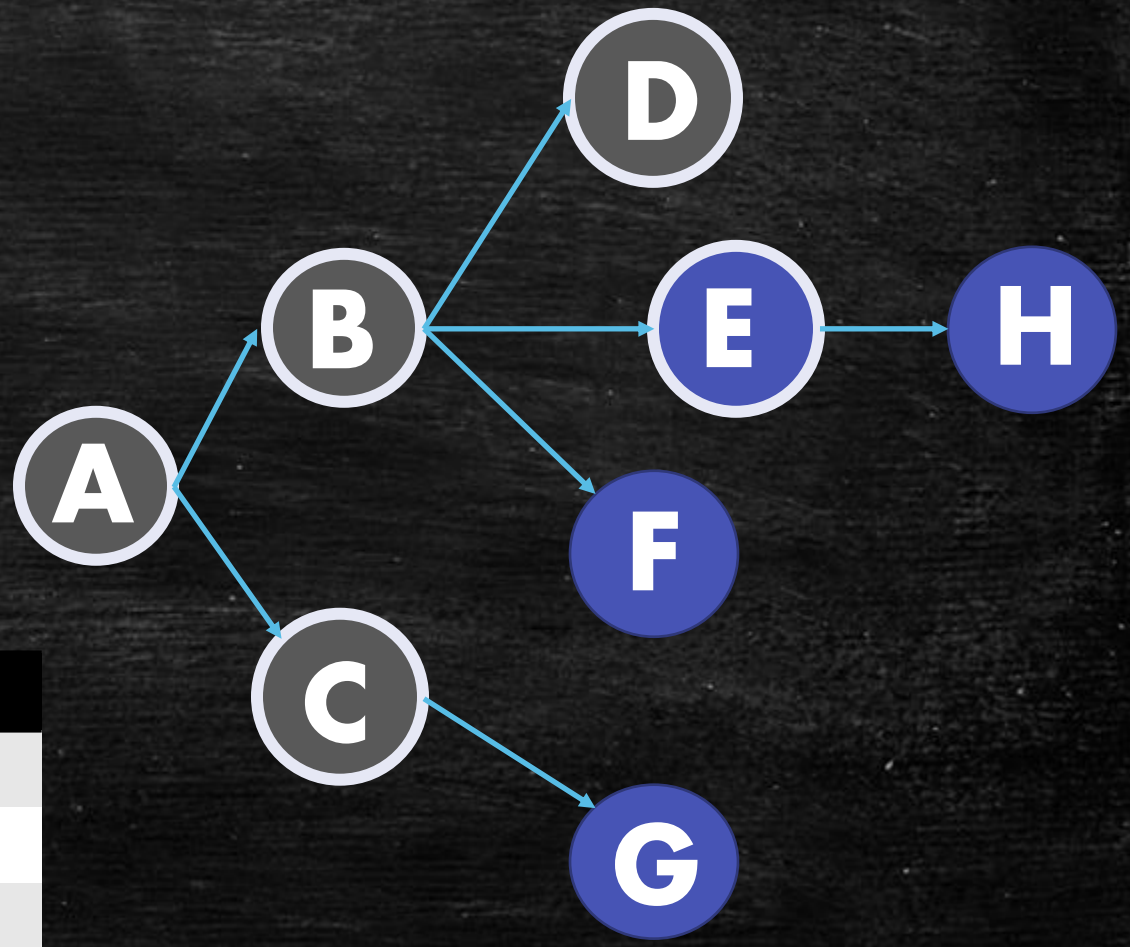

| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | - | - | - | X | X | X | - |
| Visited | T | T | T | **T** | F | F | F | F |
| Inqueue | F | F | F | **F** | T | T | T | F |

# BFS

- Ahora, por orden de Queue:

**c = E**

- Y añadimos H a Q.

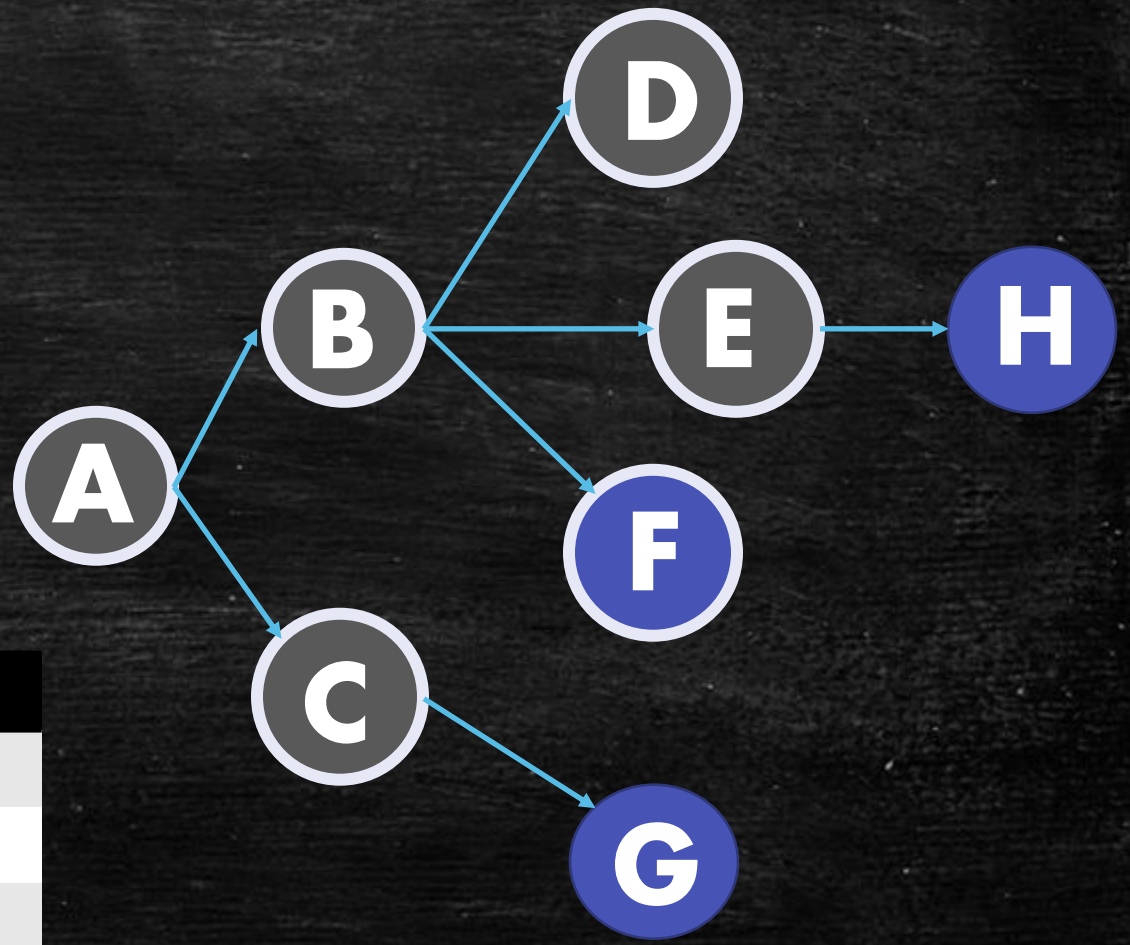



| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | - | - | - | - | X | X | X |
| Visited | T | T | T | T | T | F | F | F |
| Inqueue | F | F | F | F | F | T | T | T |

# BFS

- Ahora, por orden de Queue:

c = F

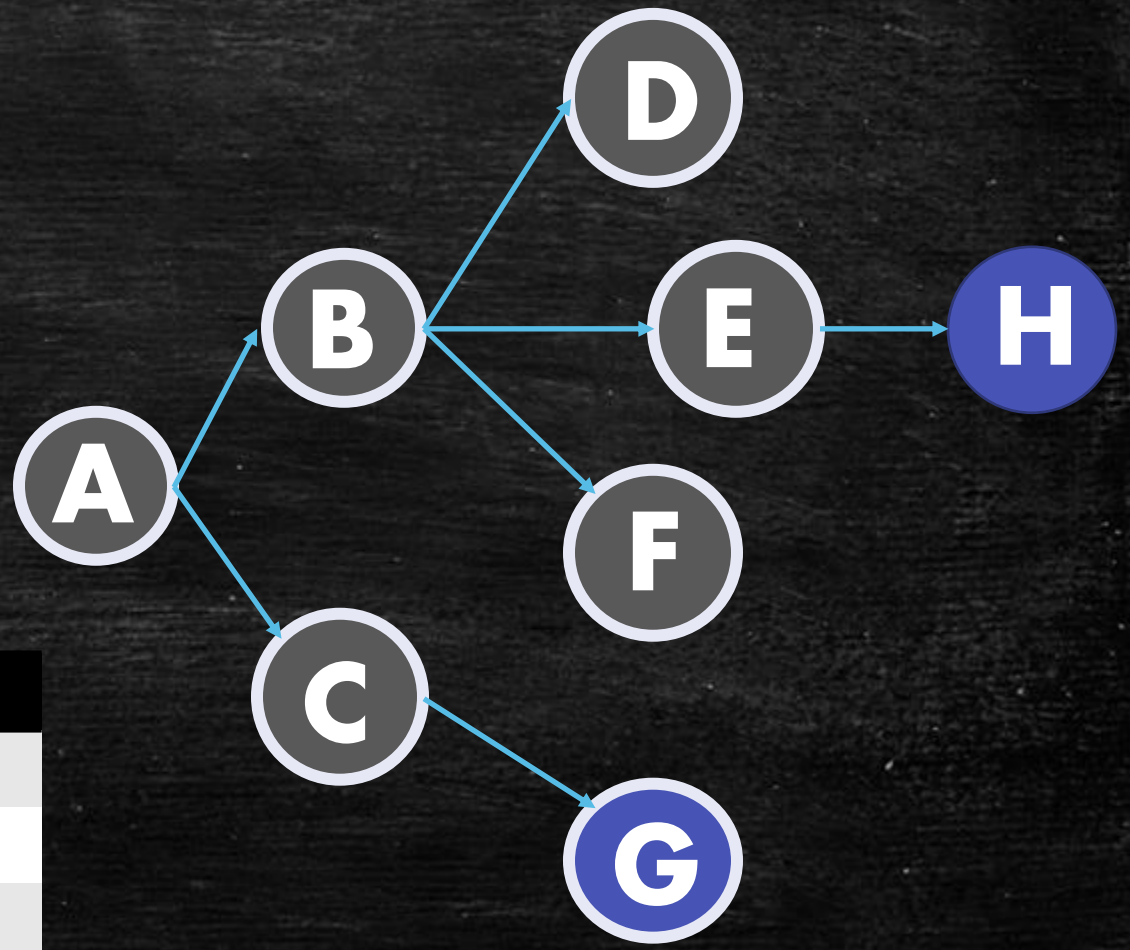



| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | - | - | - | - | - | X | X |
| Visited | T | T | T | T | T | **T** | F | F |
| Inqueue | F | F | F | F | F | **F** | T | T |

# BFS

- Ahora, por orden de Queue:

c = G

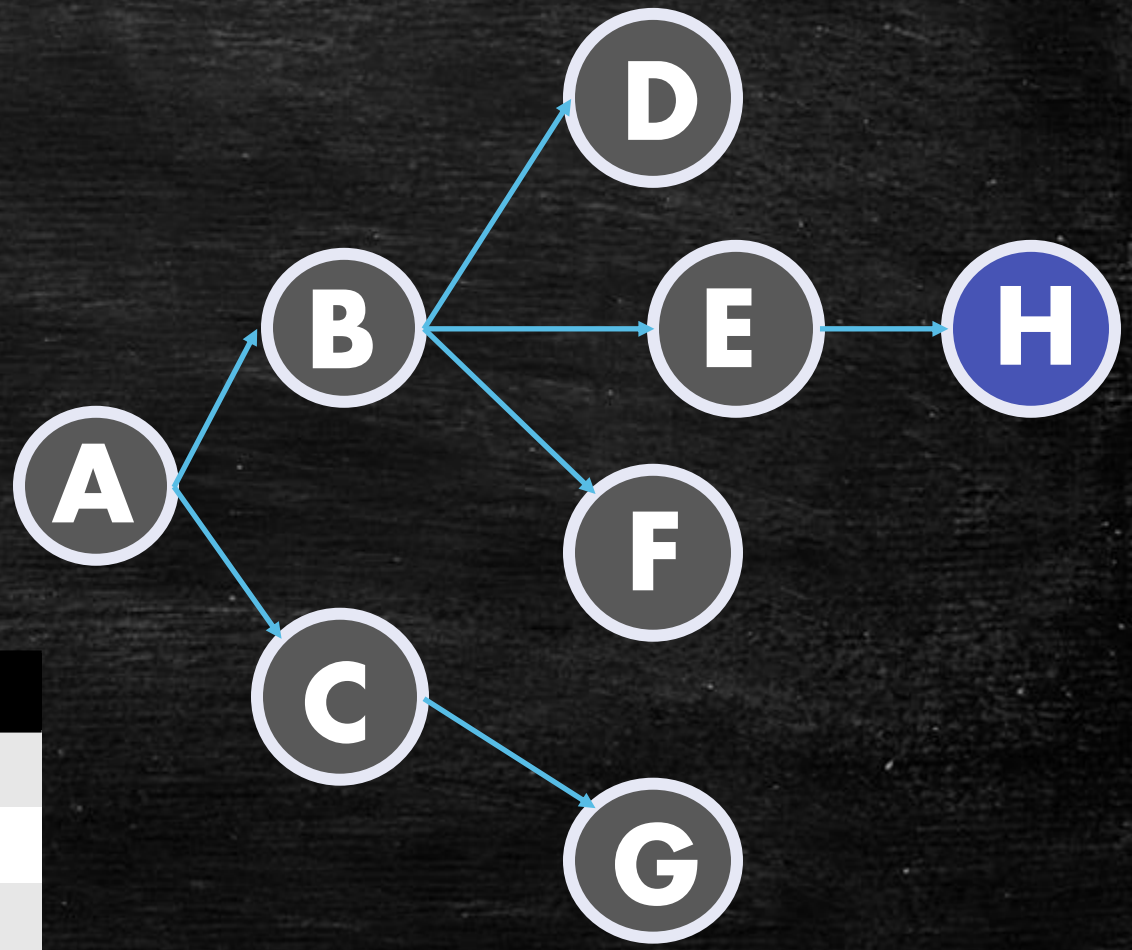



| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | - | - | - | - | - | - | X |
| Visited | T | T | T | T | T | T | **T** | F |
| Inqueue | F | F | F | F | F | F | **F** | T |

# BFS

- Ahora, por orden de Queue:

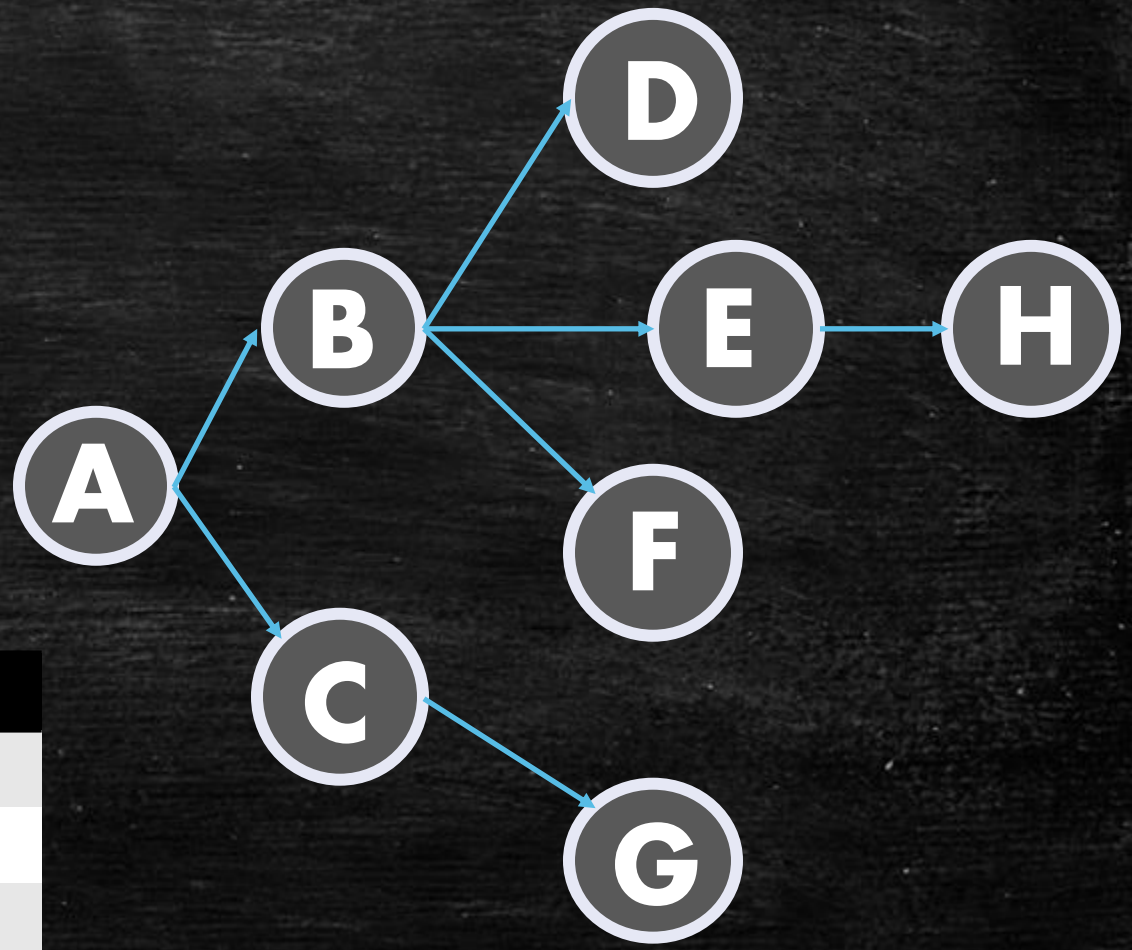
**c = H**




| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | - | - | - | - | - | - | - |
| Visited | T | T | T | T | T | T | T | **T** |
| Inqueue | F | F | F | F | F | F | F | **F** |

# BFS

- Y listo!!!




| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Q | - | - | - | - | - | - | - | - |
| Visited | T | T | T | T | T | T | T | T |
| Inqueue | F | F | F | F | F | F | F | F |

# DFS

Al igual que en el anterior, vemos que recibe G, que es un grafo. La V y la E, representan todos los vértices (V) y aristas (E). También recibe "node" que es nuestro nodo inicial.

Y nos entrega "visited", que es una lista con todos los valores True o False asignados a cada nodo si fueron revisados por el algoritmo. Esto nos dirá a qué elementos podemos llegar desde el nodo con el que comencemos.

**Algorithm 2.3:** Graph depth-first search with a stack.

```
StackDFS(G, node) → visited
    Input: G = (V, E), a graph
           node, the starting vertex in G
    Output: visited, an array of size |V| such that visited[i] is TRUE if we
            have visited node i, FALSE otherwise
1   S ← CreateStack()
2   visited ← CreateArray(|V|)
3   for i ← 0 to |V| do
4       visited[i] ← FALSE
5   Push(S, node)
6   while not IsStackEmpty(S) do
7       c ← Pop(s)
8       visited[c] ← TRUE
9       foreach v in AdjacencyList(G, c) do
10          if not visited[v] then
11              Push(S, v)
12  return visited
```

# DFS

Esta vez, comenzamos creando un Stack vacío "S" y el mismo Array "visited" con tantos elementos como nodos hay en el grafo.

Luego con el ciclo "For" marcamos como False a todos los nodos en nuestra lista visited.

Y antes de nuestro ciclo principal, hacemos Push a nuestro nodo inicial, para añadirlo al Stack.

Recordemos que en el Stack siempre se colocan los elementos sobre todos los anteriores, y se saca el que entró último antes que los demás.

**Algorithm 2.3:** Graph depth-first search with a stack.

```
StackDFS(G, node) → visited
    Input: G = (V, E), a graph
           node, the starting vertex in G
    Output: visited, an array of size |V| such that visited[i] is TRUE if we
            have visited node i, FALSE otherwise
1   S ← CreateStack()
2   visited ← CreateArray(|V|)
3   for i ← 0 to |V| do
4       visited[i] ← FALSE
5   Push(S, node)
6   while not IsStackEmpty(S) do
7       c ← Pop(s)
8       visited[c] ← TRUE
9       foreach v in AdjacencyList(G, c) do
10          if not visited[v] then
11              Push(S, v)
12  return visited
```

# DFS

En nuestro ciclo principal, asignamos a la variable "c" el nodo que salga del Stack al hacerle Pop. Luego marcamos el nodo en "c" como True en visited.

Ahora iniciamos otro ciclo "For" en el que revisaremos a todos los vecinos de nuestro nodo en "c". Si estos no han sido visitados, se les hará Push a nuestro Stack "S".

Esto hará que el ciclo siga funcionando hasta que ya no queden más nodos que conectar.

Y al terminar, devolverá el Array visited.

**Algorithm 2.3:** Graph depth-first search with a stack.

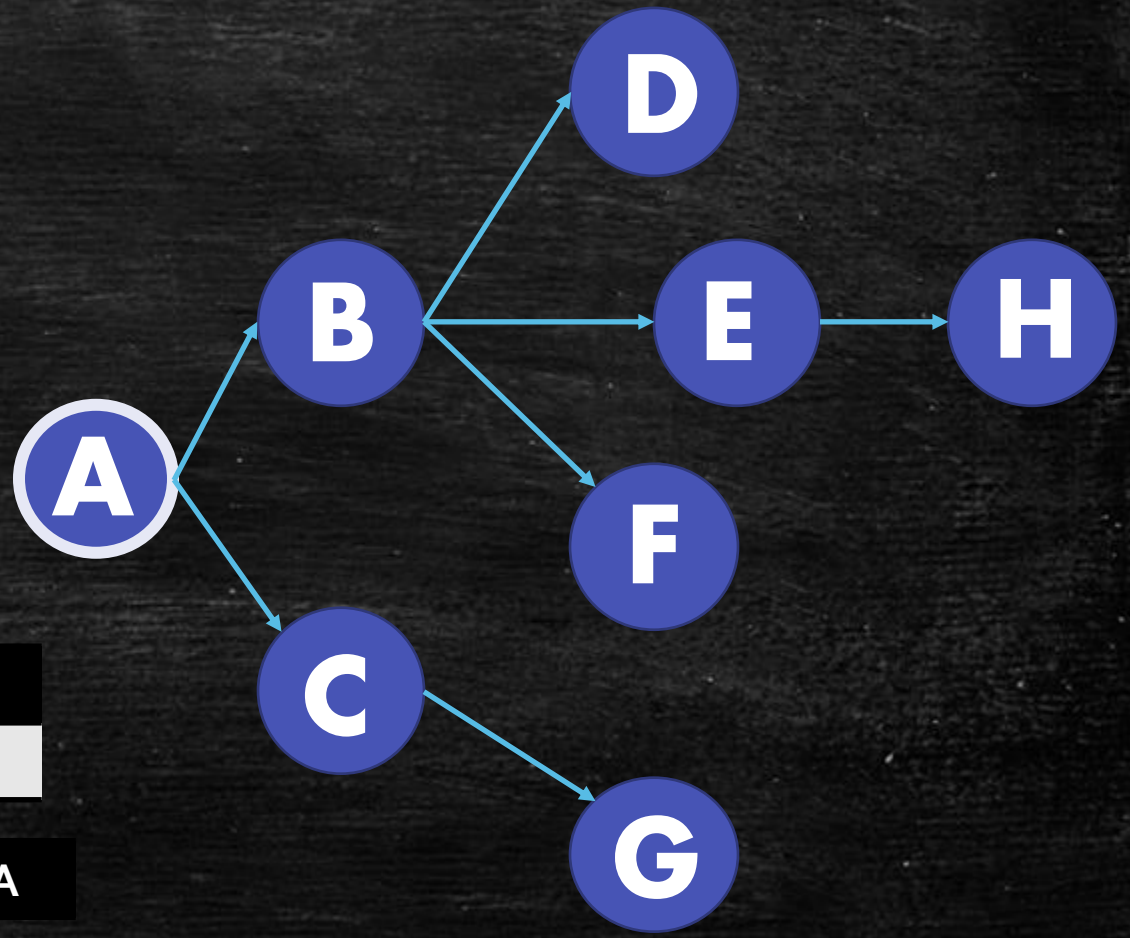
```
StackDFS(G, node) → visited
    Input: G = (V, E), a graph
           node, the starting vertex in G
    Output: visited, an array of size |V| such that visited[i] is TRUE if we
            have visited node i, FALSE otherwise

1   S ← CreateStack()
2   visited ← CreateArray(|V|)
3   for i ← 0 to |V| do
4       visited[i] ← FALSE
5   Push(S, node)
6   while not IsStackEmpty(S) do
7       c ← Pop(s)
8       visited[c] ← TRUE
9       foreach v in AdjacencyList(G, c) do
10          if not visited[v] then
11              Push(S, v)
12  return visited
```

# DFS

- Vamos a ver representado gráficamente el funcionamiento el algoritmo de DFS.
- De manera que este es nuestro G, y nuestro nodo inicial será A.
- Visited marcará todos los nodos como False. Y añadiremos A a S usando Push.

| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | F | F | F | F | F | F | F | F |

| Stack (S): | A |
|---|---|

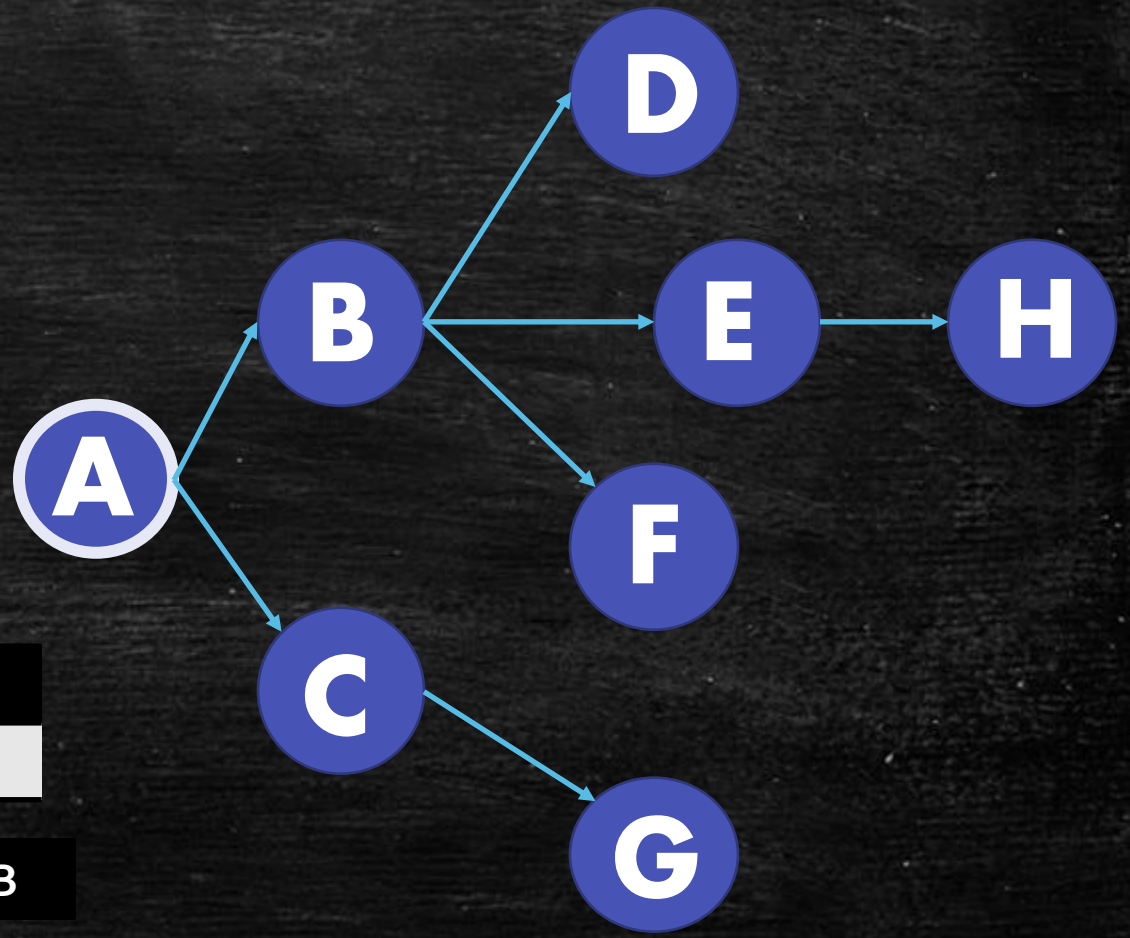


# DFS

- Ahora hacemos Pop a S, y asignamos:

$$c = A$$

- Se marcará en visited como True y se hará Push a sus vecinos.




| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | **T** | F | F | F | F | F | F | F |

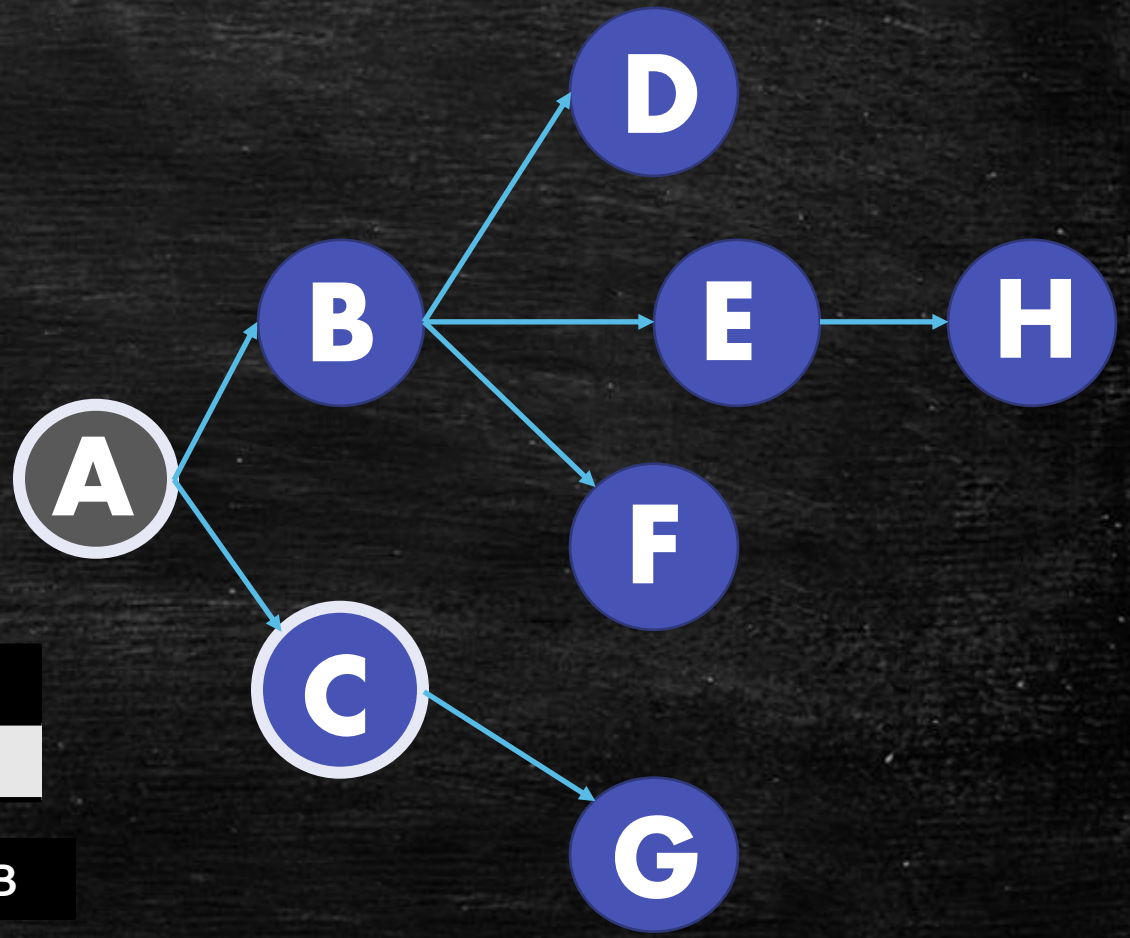
| Stack (S): | | | | | | | C | B |
|---|---|---|---|---|---|---|---|---|

# DFS

- Ahora al hacer Pop, sacamos el último dato en entrar, así que:

**c = C**

- Marcamos C como visited y hacemos Push a su vecino.



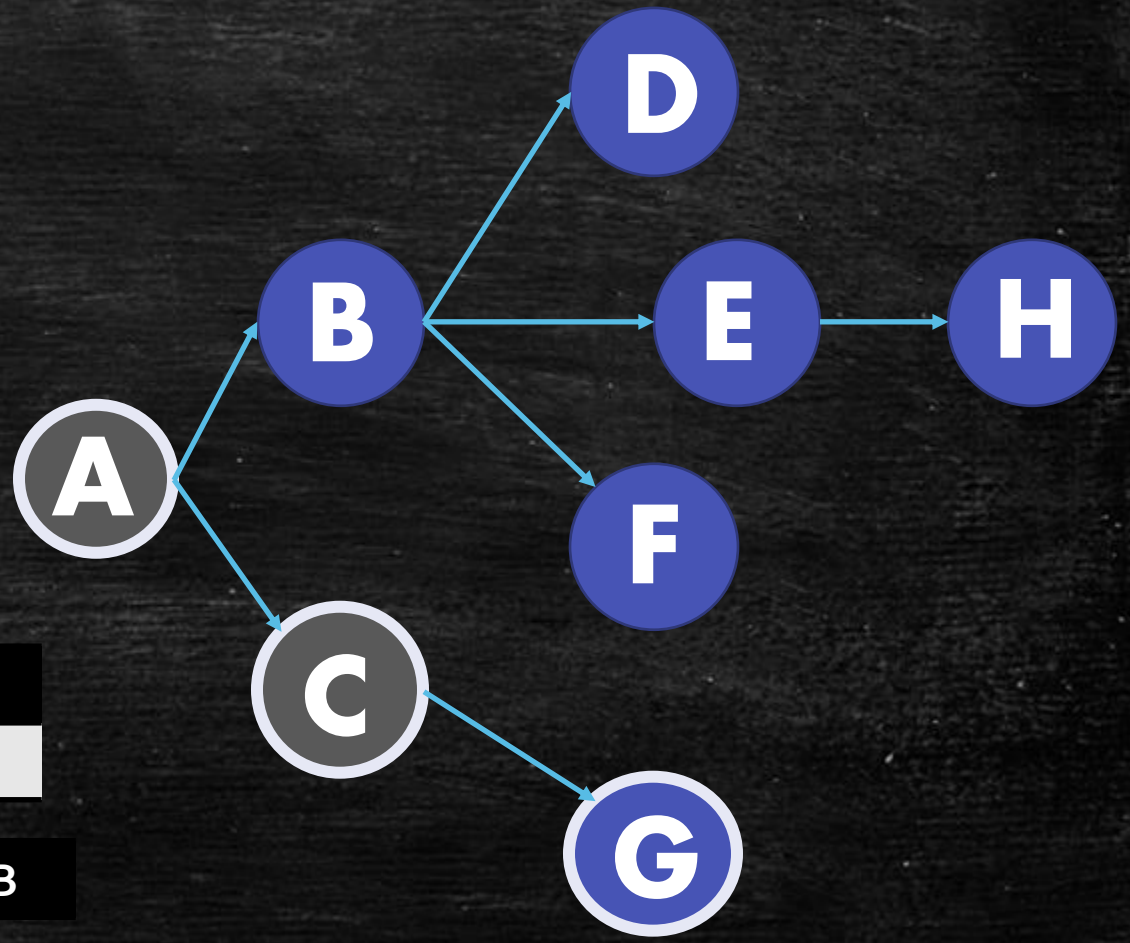

| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | T | F | **T** | F | F | F | F | F |

| Stack (S): | | | | | | | G | B |
|---|---|---|---|---|---|---|---|---|

# DFS

- Hacemos Pop, y:

c = G




| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | T | F | T | F | F | F | **T** | F |

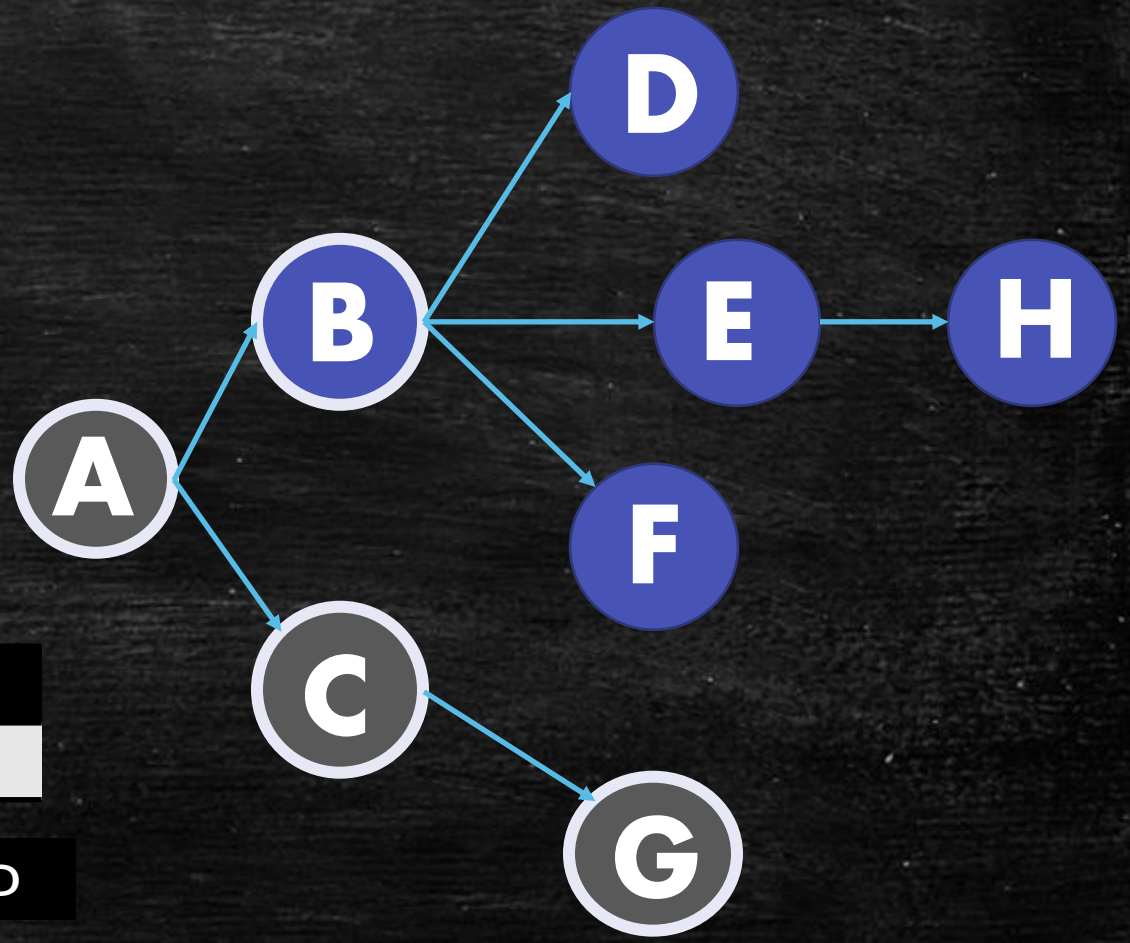
**Stack (S):** **B**

# DFS

- Hacemos Pop, y:

  **c = B**

- Lo que hará Push a D, E y F.



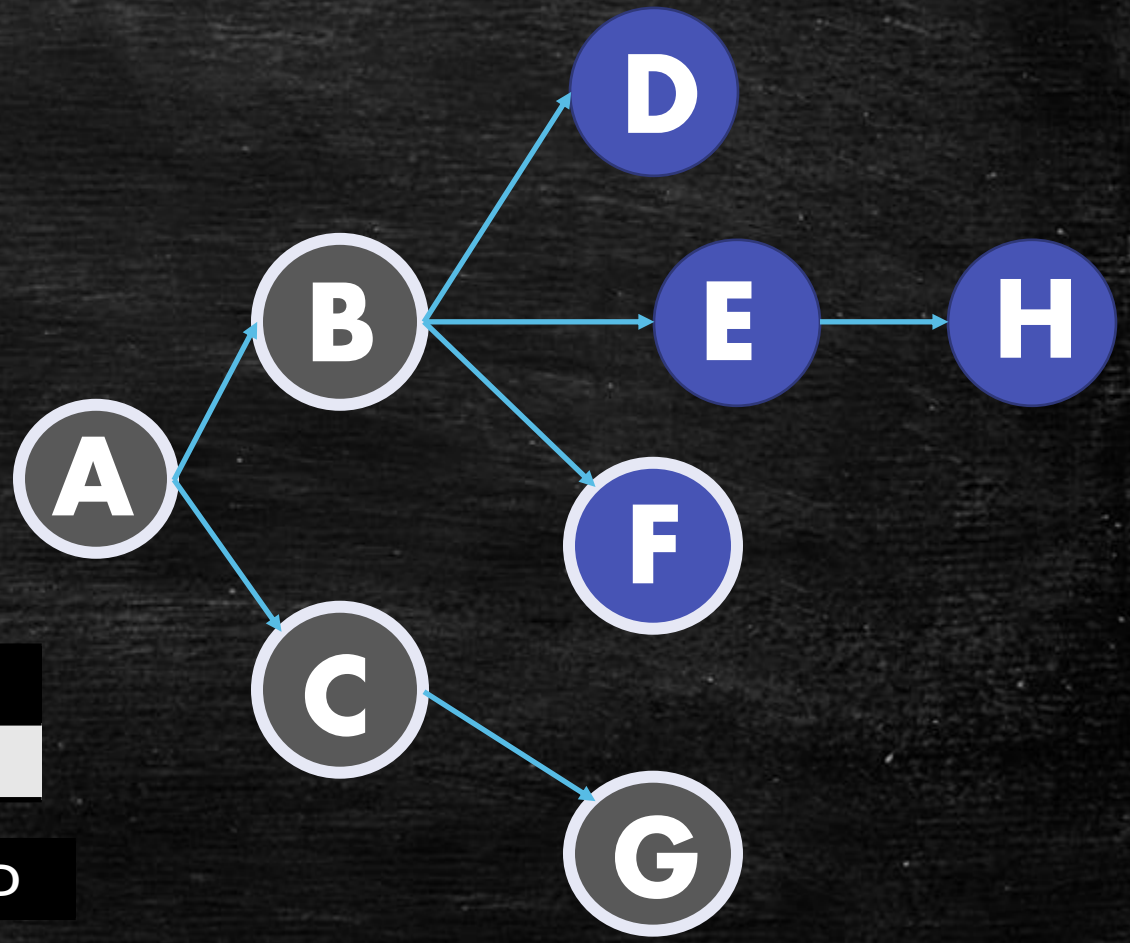

| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | T | **T** | T | F | F | F | T | F |

| Stack (S): | | | | | | F | E | D |
|---|---|---|---|---|---|---|---|---|

# DFS

- Hacemos Pop, y:

c = F




| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | T | T | T | F | F | T | T | F |

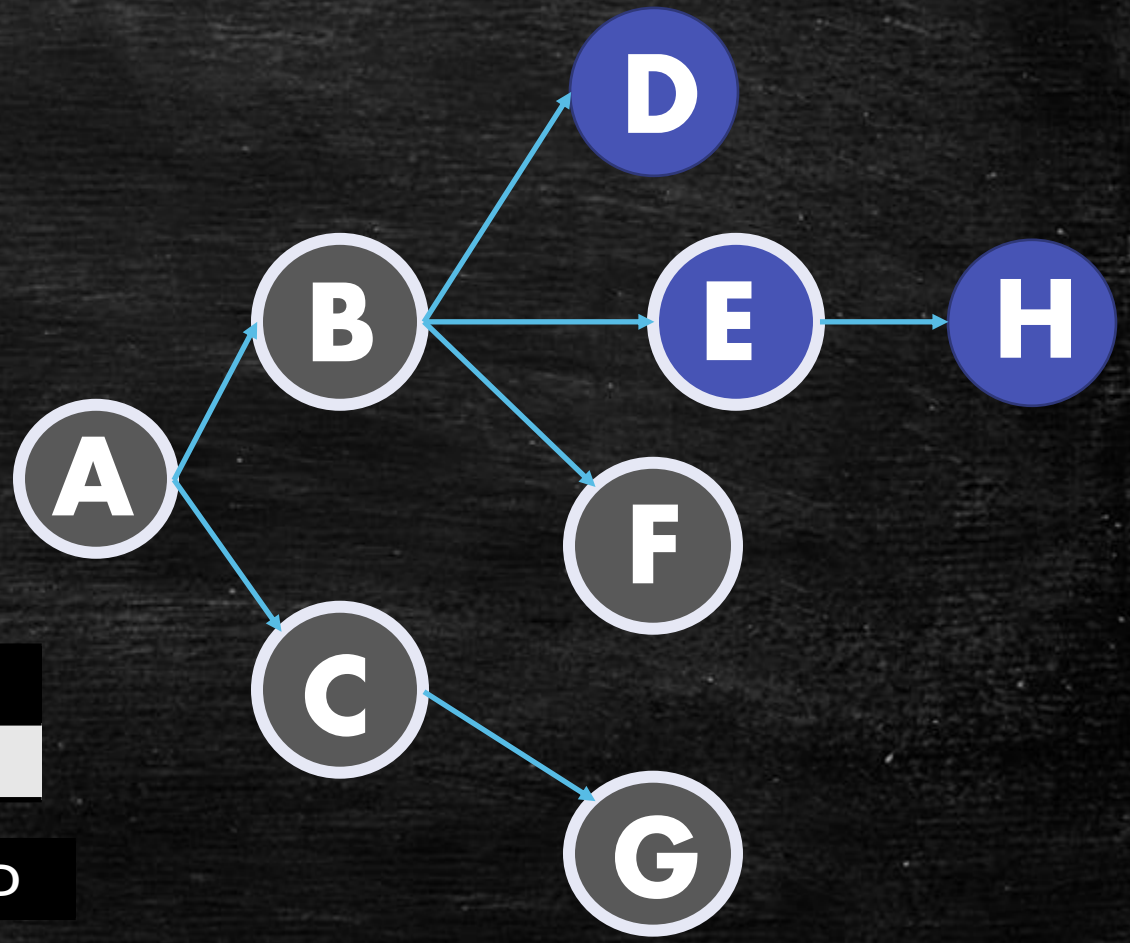
| Stack (S): | | | | | | | E | D |
|---|---|---|---|---|---|---|---|---|

# DFS

- Hacemos Pop, y:

  **c = E**

- Lo que hará Push a H.



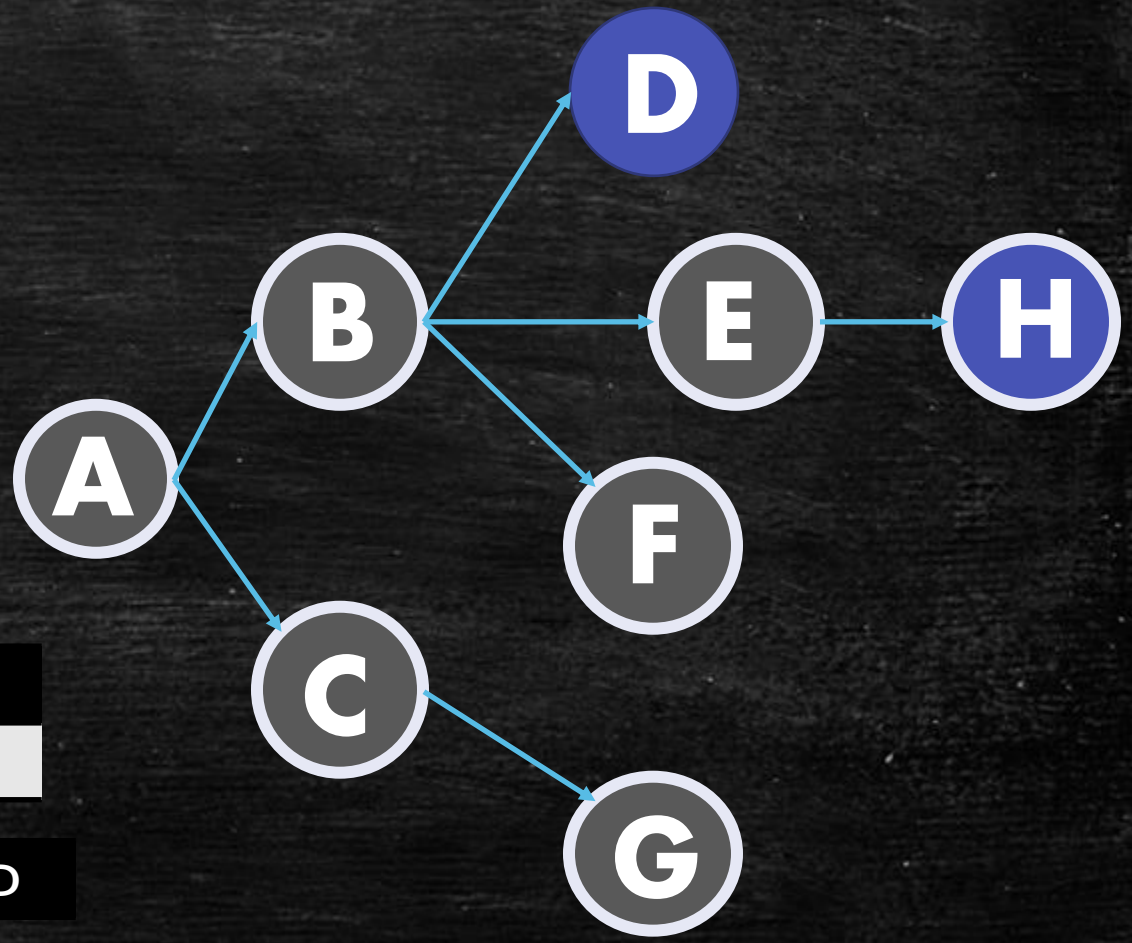

| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | T | T | T | F | **T** | T | T | F |

| Stack (S): | | | | | | | H | D |
|---|---|---|---|---|---|---|---|---|

# DFS

- Hacemos Pop, y:

**c = H**



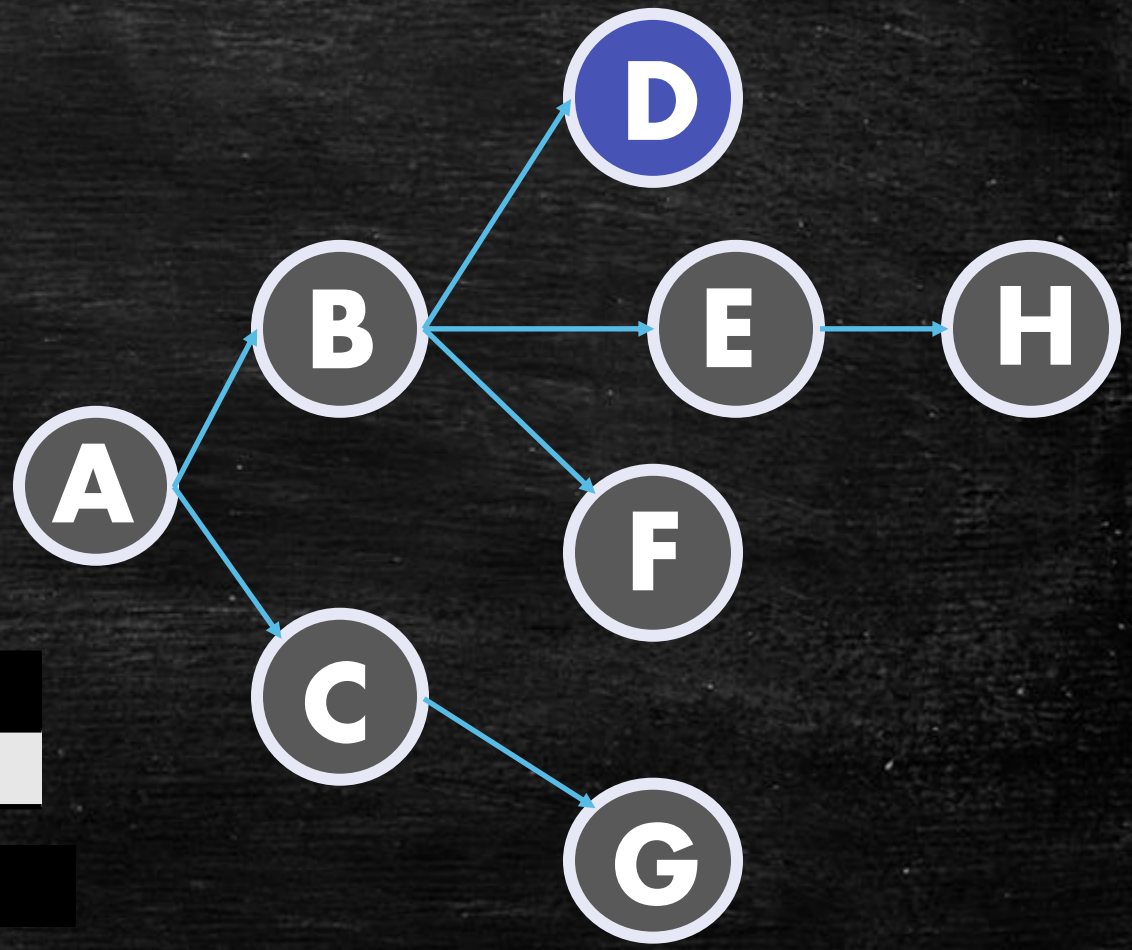

| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | T | T | T | F | T | T | T | T |

| Stack (S): | D |
|---|---|

# DFS

- Hacemos Pop, y:

c = D

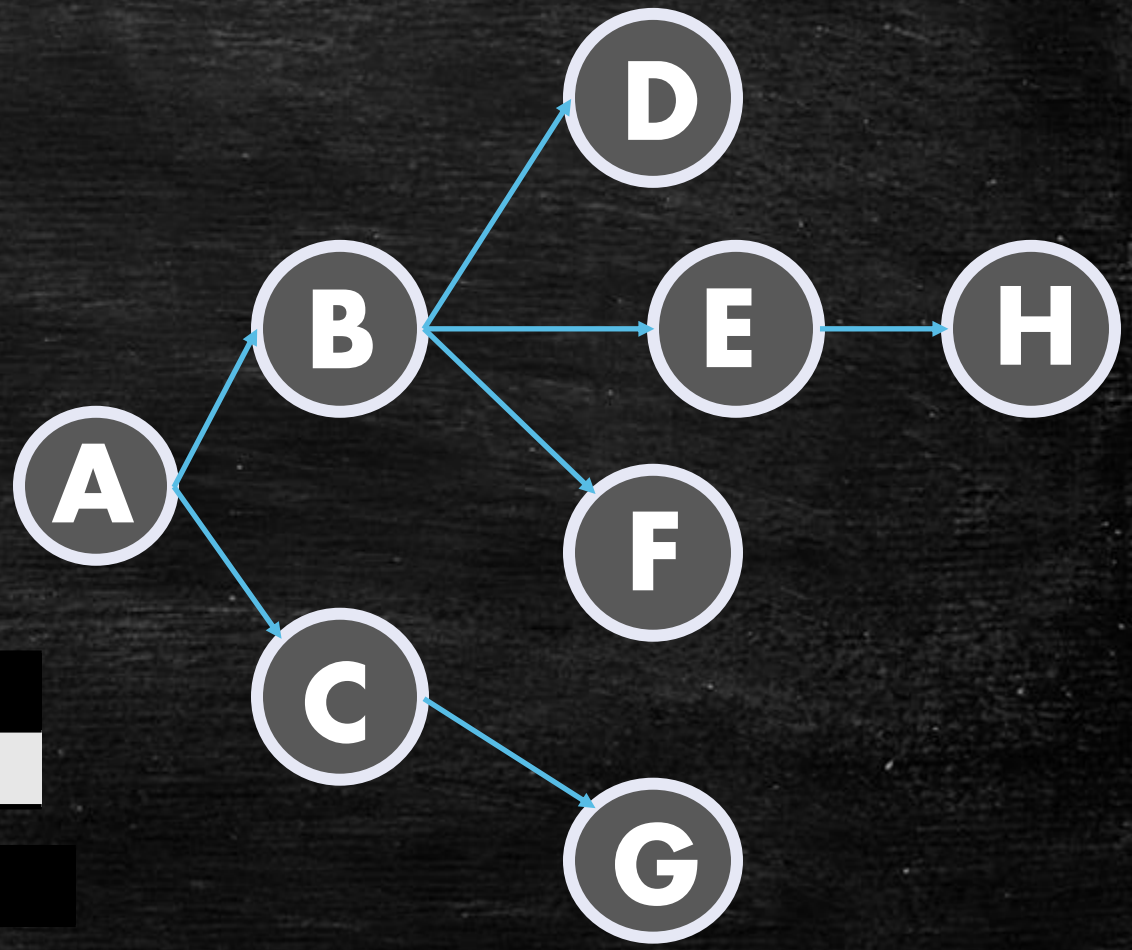



| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | T | T | T | **T** | T | T | T | T |

**Stack (S):**

# DFS

- Y terminamos de revisar todos los nodos!!!




| Nodos: | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Visited | T | T | T | **T** | T | T | T | T |

**Stack (S):**

# Algoritmo de Dijkstra

Este algoritmo tiene una función distinta a los anteriores, porque busca usarse en grafos que tengan aristas con peso (solo positivo). Esto significa que las aristas que conectan los nodos tienen un peso que indicará el tiempo o esfuerzo que requiere llegar de un nodo a otro.

Este algoritmo te entrega los caminos con menos peso para todos los nodos en un grafo, comenzando desde un nodo inicial. Y de este modo conocer el camino más eficiente para alcanzar a los otros nodos.

**Algorithm 7.1**: Dijkstra's algorithm.

`Dijkstra`$(G, s) \rightarrow (pred, dist)$
**Input:** $G = (V, E)$, a graph
$s$, the starting node
**Output:** *pred*, an array of size $|V|$ such that $pred[i]$ is the predecessor of node $i$ in the shortest path from $s$
*dist*, an array of size $|V|$ such that $dist[i]$ is the length of the shortest path calculated from node $s$ to $i$

```
1  pred ← CreateArray(|V|)
2  dist ← CreateArray(|V|)
3  pq ← CreatePQ()
4  foreach v in V do
5      pred[v] ← −1
6      if v ≠ s then
7          dist[v] ← ∞
8      else
9          dist[v] ← 0
10     InsertInPQ(pq, v, dist[v])
11 while SizePQ(pq) ≠ 0 do
12     u ← ExtractMinFromPQ(pq)
13     foreach v in AdjacencyList(G, u) do
14         if dist[v] > dist[u] + Weight(G, u, v) then
15             dist[v] ← dist[u] + Weight(G, u, v)
16             pred[v] ← u
17             UpdatePQ(pq, v, dist[v])
18 return (pred, dist)
```

# Dijkstra

Comenzamos recibiendo un Grafo y un nodo inicial, tal como los anteriores.

Pero lo que buscamos que nos entregue, son dos listas: una indicando el peso mínimo para llegar a cada nodo, y otra que almacena que nodos hay que seguir para llegar a esos nodos.

pred es un Array con tantos elementos como vértices en el grafo, que indica el nodo anterior al que buscamos llegar con menos peso acumulado, de manera que arma el camino que debemos tomar.

dist nos indica el peso mínimo para llegar a cada nodo.

**Algorithm 7.1:** Dijkstra's algorithm.

```
Dijkstra(G, s) → (pred, dist)
    Input: G = (V, E), a graph
           s, the starting node
    Output: pred, an array of size |V| such that pred[i] is the predecessor
            of node i in the shortest path from s
            dist, an array of size |V| such that dist[i] is the length of the
            shortest path calculated from node s to i
 1  pred ← CreateArray(|V|)
 2  dist ← CreateArray(|V|)
 3  pq ← CreatePQ()
 4  foreach v in V do
 5      pred[v] ← −1
 6      if v ≠ s then
 7          dist[v] ← ∞
 8      else
 9          dist[v] ← 0
10      InsertInPQ(pq, v, dist[v])
11  while SizePQ(pq) ≠ 0 do
12      u ← ExtractMinFromPQ(pq)
13      foreach v in AdjacencyList(G, u) do
14          if dist[v] > dist[u] + Weight(G, u, v) then
15              dist[v] ← dist[u] + Weight(G, u, v)
16              pred[v] ← u
17              UpdatePQ(pq, v, dist[v])
18  return (pred, dist)
```

# Dijkstra

Al empezar, crea los dos Arrays que buscamos entregar, pred y dist. Y también crea "pq" que es una Priority Queue. Es un tipo de Queue que además del elemento, guarda un valor de prioridad. Y los elementos se ordenan de manera que los que tengan un mayor valor de prioridad salgan primero de este Queue.

Luego de esto, con un ciclo "for" le asigna el valor -1 en la lista pred, y a todo nodo que no sea el inicial, le asigna una distancia de infinito. Esto de manera que todos los valores reales que nos marquen el peso de la arista sean menores a lo que asignemos inicialmente. Y al nodo inicial, se le asigna distancia de cero.

**Algorithm 7.1**: Dijkstra's algorithm.

```
Dijkstra(G, s) → (pred, dist)
    Input: G = (V, E), a graph
           s, the starting node
    Output: pred, an array of size |V| such that pred[i] is the predecessor
            of node i in the shortest path from s
            dist, an array of size |V| such that dist[i] is the length of the
            shortest path calculated from node s to i

1   pred ← CreateArray(|V|)
2   dist ← CreateArray(|V|)
3   pq ← CreatePQ()
4   foreach v in V do
5       pred[v] ← −1
6       if v ≠ s then
7           dist[v] ← ∞
8       else
9           dist[v] ← 0
10      InsertInPQ(pq, v, dist[v])
11  while SizePQ(pq) ≠ 0 do
12      u ← ExtractMinFromPQ(pq)
13      foreach v in AdjacencyList(G, u) do
14          if dist[v] > dist[u] + Weight(G, u, v) then
15              dist[v] ← dist[u] + Weight(G, u, v)
16              pred[v] ← u
17              UpdatePQ(pq, v, dist[v])
18  return (pred, dist)
```

# Dijkstra

Todo lo anterior se guarda en "pq". O sea, los nodos, con su valor de prioridad determinado por la distancia que acumulan.

Ahora comenzamos con nuestro ciclo principal, con la condición de que tienen que haber elementos en "pq". Primero, extrae el nodo con menor prioridad de "pq" y lo asigna a la variable u. La primera vez, será el nodo inicial, porque tiene un valor de distancia (y por ende de prioridad) igual a cero. Mientras que los demás tienen valor infinito.

**Algorithm 7.1:** Dijkstra's algorithm.

```
Dijkstra(G, s) → (pred, dist)
    Input: G = (V, E), a graph
           s, the starting node
    Output: pred, an array of size |V| such that pred[i] is the predecessor
            of node i in the shortest path from s
            dist, an array of size |V| such that dist[i] is the length of the
            shortest path calculated from node s to i
 1  pred ← CreateArray(|V|)
 2  dist ← CreateArray(|V|)
 3  pq ← CreatePQ()
 4  foreach v in V do
 5      pred[v] ← −1
 6      if v ≠ s then
 7          dist[v] ← ∞
 8      else
 9          dist[v] ← 0
10      InsertInPQ(pq, v, dist[v])
11  while SizePQ(pq) ≠ 0 do
12      u ← ExtractMinFromPQ(pq)
13      foreach v in AdjacencyList(G, u) do
14          if dist[v] > dist[u] + Weight(G, u, v) then
15              dist[v] ← dist[u] + Weight(G, u, v)
16              pred[v] ← u
17              UpdatePQ(pq, v, dist[v])
18  return (pred, dist)
```

# Dijkstra

Ahora tenemos un ciclo dentro del ciclo principal, que por cada vecino de u, calcula si el peso de la arista que haya entre ellos más el peso para llegar a ese nodo es menor que el peso que tenemos guardado en la lista. En ese caso, se le asigna a la nueva distancia la suma entre el peso acumulado del nodo que revisamos y la distancia de la arista para llegar al otro nodo.

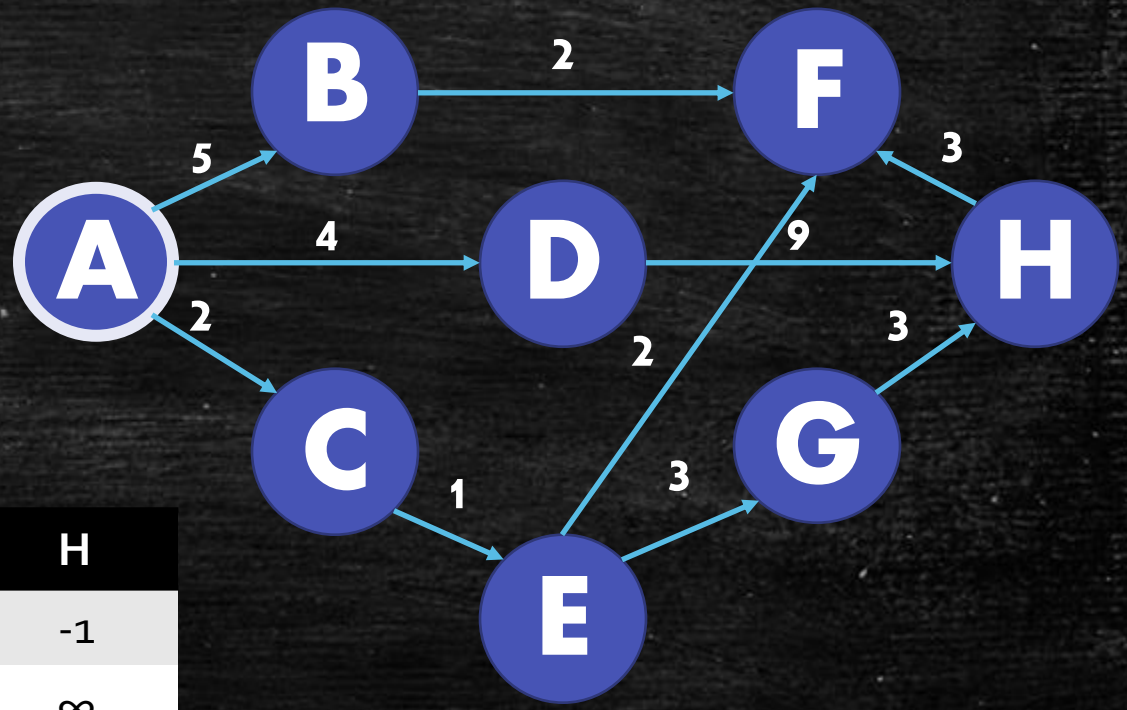
La primera vez de cada nodo siempre cambiará este valor, puesto a que tienen asignada una distancia infinita y cualquier distancia sería menor.

**Algorithm 7.1**: Dijkstra's algorithm.

```
Dijkstra(G, s) → (pred, dist)
    Input: G = (V, E), a graph
           s, the starting node
    Output: pred, an array of size |V| such that pred[i] is the predecessor
            of node i in the shortest path from s
            dist, an array of size |V| such that dist[i] is the length of the
            shortest path calculated from node s to i

 1  pred ← CreateArray(|V|)
 2  dist ← CreateArray(|V|)
 3  pq ← CreatePQ()
 4  foreach v in V do
 5      pred[v] ← −1
 6      if v ≠ s then
 7          dist[v] ← ∞
 8      else
 9          dist[v] ← 0
10      InsertInPQ(pq, v, dist[v])
11  while SizePQ(pq) ≠ 0 do
12      u ← ExtractMinFromPQ(pq)
13      foreach v in AdjacencyList(G, u) do
14          if dist[v] > dist[u] + Weight(G, u, v) then
15              dist[v] ← dist[u] + Weight(G, u, v)
16              pred[v] ← u
17              UpdatePQ(pq, v, dist[v])
18  return (pred, dist)
```

# Dijkstra

- Vamos a ver representado gráficamente el funcionamiento el algoritmo de Dijkstra.
- De manera que este es nuestro G, y nuestro nodo inicial será A.
- Creamos pred, dist y pq. Le asignamos a todos el valor -1 en pred, el infinito en dist, menos a A. Y en pq añadimos el nodo y su valor de dist



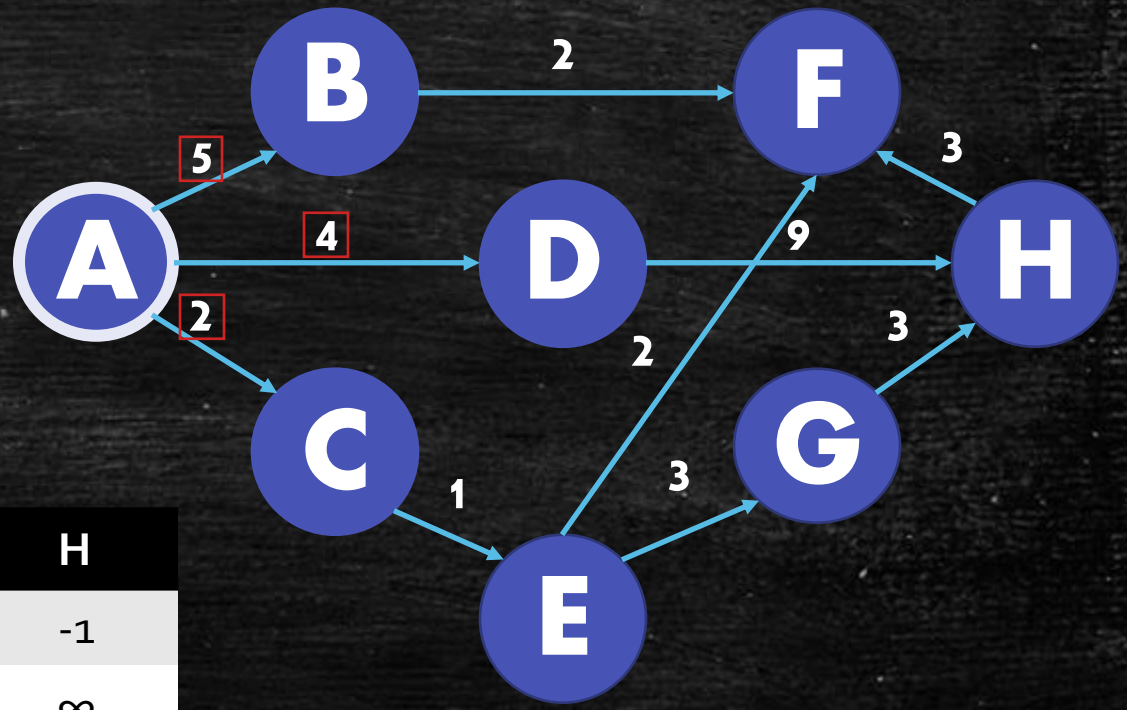

| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| Dist | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| pq | A, 0 | B, ∞ | C, ∞ | D, ∞ | E, ∞ | F, ∞ | G, ∞ | H, ∞ |

# Dijkstra

- Extraemos el nodo con menor prioridad en pq, al inicio el nodo inicial tendrá 0. Aquí

**u = A**

- Por cada vecino de A, vemos si la distancia de la arista es mayor a la distancia actual del nodo.
- Y todos son menores a ∞, así que reemplazamos en dist y a su vez, su prioridad en pq
- Además, marcamos A como su pred, porque es el camino que tomamos hasta esos nodos.



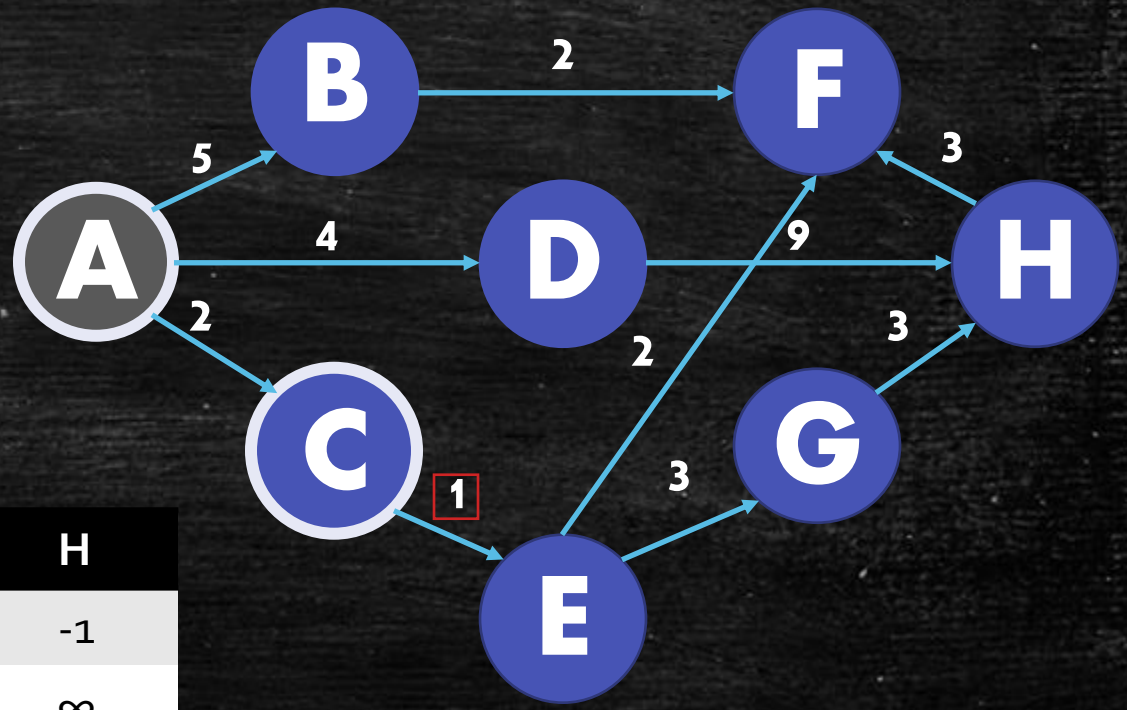

| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | -1 | -1 | -1 | -1 |
| Dist | 0 | 5 | 2 | 4 | ∞ | ∞ | ∞ | ∞ |
| pq | - | B, 5 | C, 2 | D, 4 | E, ∞ | F, ∞ | G, ∞ | H, ∞ |

# Dijkstra

- Volvemos a extraer el nodo con menor prioridad, ahora es C.

**u = C**

- El peso del vector, debemos sumarlo al peso del nodo en el que estamos, por ende, si el vector hacia E pesa 1, y el nodo C tiene distancia 2, la distancia de E es 1+2 = 3.
- Reemplazamos en nuestras listas, y marcamos C como pred.



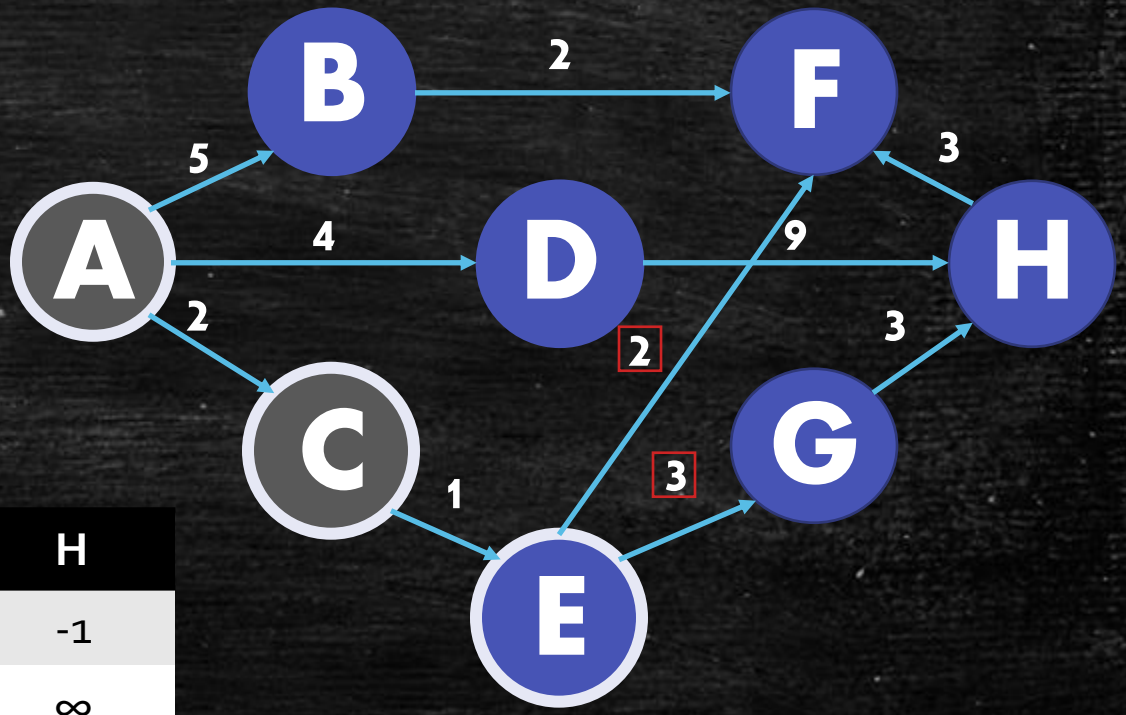

| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | **C** | -1 | -1 | -1 |
| Dist | 0 | 5 | 2 | 4 | **3** | ∞ | ∞ | ∞ |
| pq | - | B, 5 | - | D, 4 | **E, 3** | F, ∞ | G, ∞ | H, ∞ |

# Dijkstra

- Volvemos a extraer el nodo con menor prioridad, ahora es E.

**u = E**

- Volvemos a sumar el peso de las aristas con el peso de nuestro nodo, tenemos dist 5 hacia F y dist 6 hacia G.
- Reemplazamos en las listas.



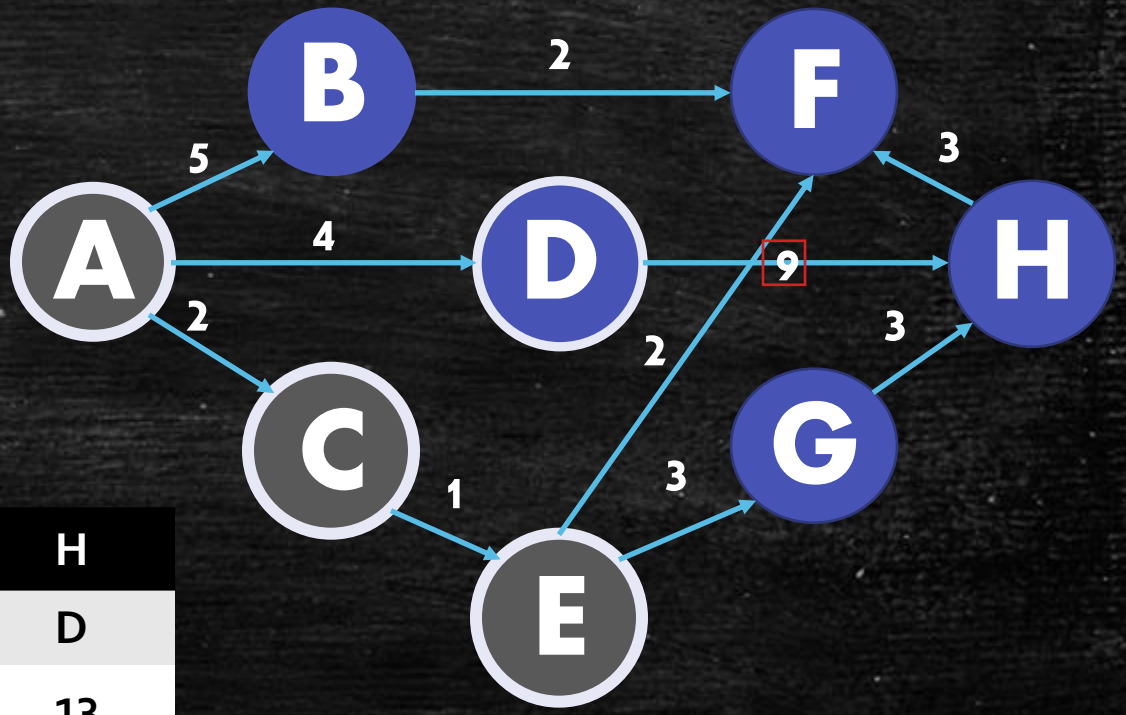

| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | C | **E** | **E** | -1 |
| Dist | 0 | 5 | 2 | 4 | 3 | **5** | **6** | ∞ |
| pq | - | B, 5 | - | D, 4 | - | **F, 5** | **G, 6** | H, ∞ |

# Dijkstra

- Volvemos a extraer el nodo con menor prioridad, ahora es D.

**u = D**

- Sumamos el peso de D al peso de la arista hacia H, nos da 13.
- Reemplazamos en las listas.



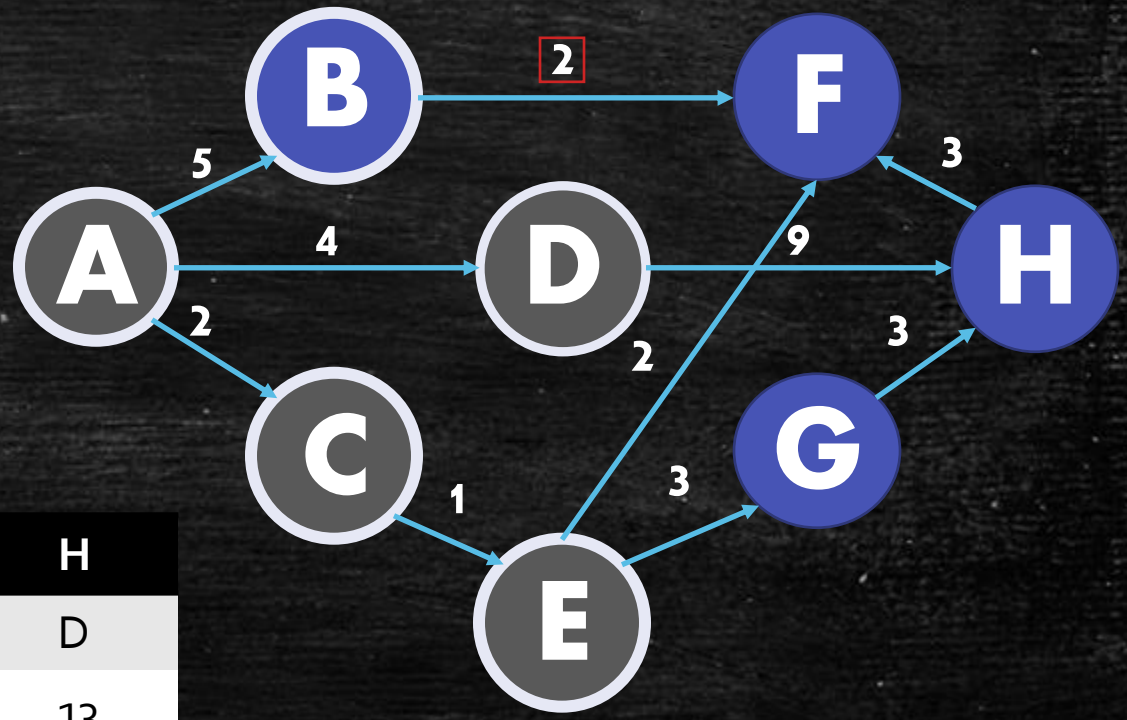

| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | C | E | E | **D** |
| Dist | 0 | 5 | 2 | 4 | 3 | 5 | 6 | **13** |
| pq | - | B, 5 | - | - | - | F, 5 | G, 6 | **H, 13** |

# Dijkstra

- Volvemos a extraer el nodo con menor prioridad, ahora son B y F, no hay mucha diferencia, así que partiremos por el primero que añadimos, B.

**u = B**

- Si sumamos su peso con el peso de la arista, vemos que es mayor al peso de F, por lo que no debemos reemplazar los valores, lo que tenemos ya es más eficiente.



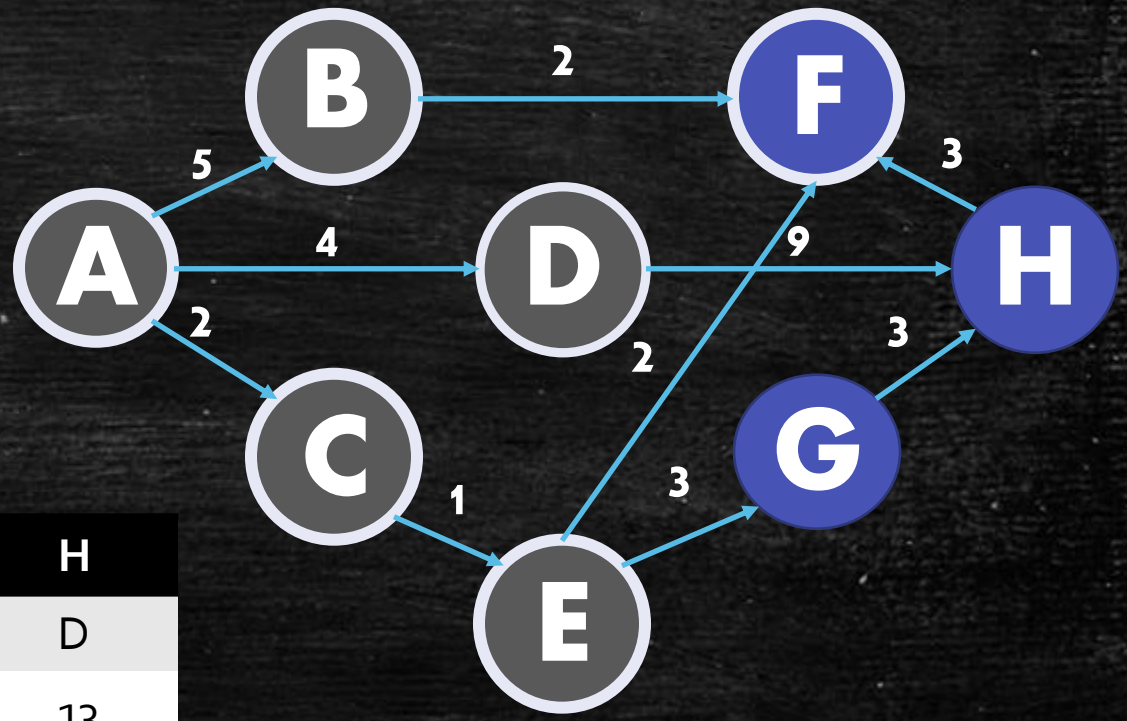

| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | C | E | E | D |
| Dist | 0 | 5 | 2 | 4 | 3 | 5 | 6 | 13 |
| pq | - | - | - | - | - | F, 5 | G, 6 | H, 13 |

# Dijkstra

- Volvemos a extraer el nodo con menor prioridad, ahora será F.

**u = F**

El cual no tiene vecinos, puesto a que las flechas solo apuntan hacia F, así que lo eliminamos de la lista pq.



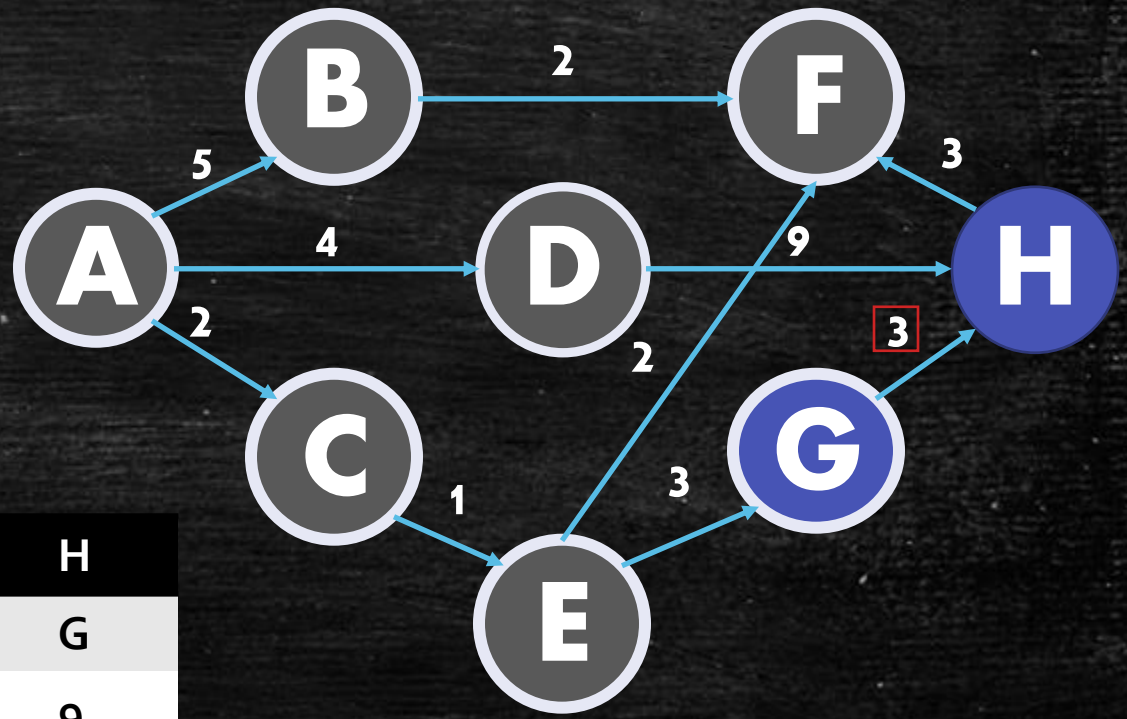

| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | C | E | E | D |
| Dist | 0 | 5 | 2 | 4 | 3 | 5 | 6 | 13 |
| pq | - | - | - | - | - | - | G, 6 | H, 13 |

# Dijkstra

- Volvemos a extraer el nodo con menor prioridad, ahora será G.

**u = G**

- Podemos ver que, si sumamos la dist de G con el vector hacia H, suma 9, que es menor a los 13 que habíamos guardado, así que como esto es más eficiente, reemplazamos la distancia, y su pred por G.



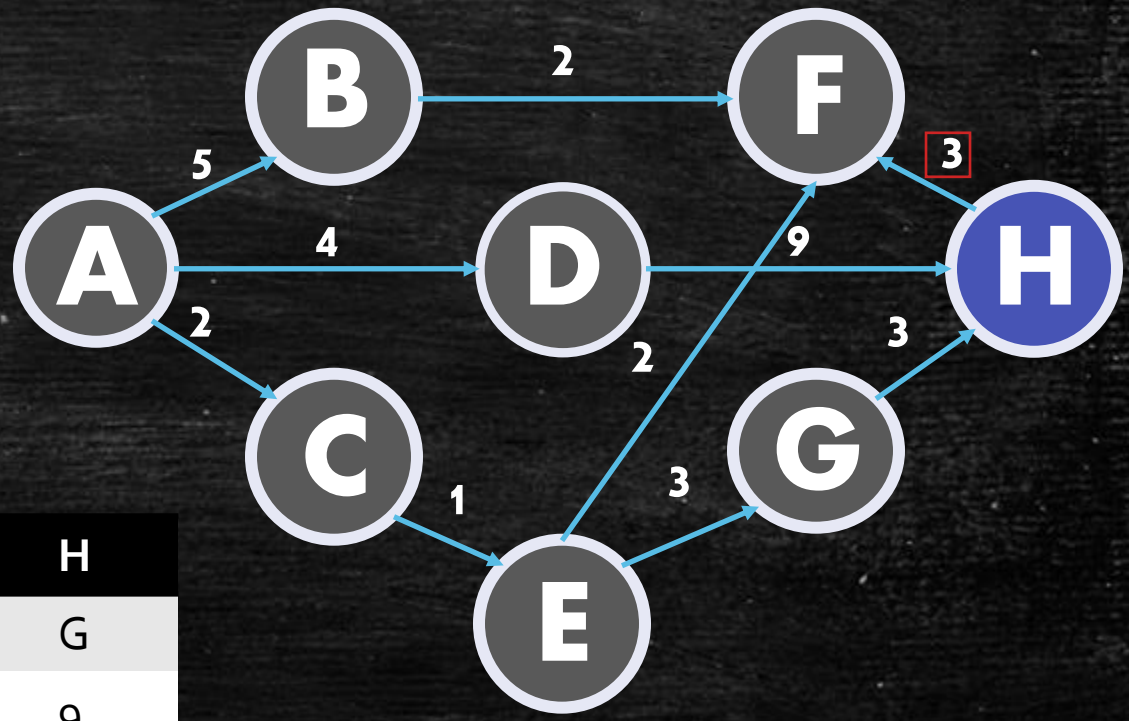

| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | C | E | E | **G** |
| Dist | 0 | 5 | 2 | 4 | 3 | 5 | 6 | **9** |
| pq | - | - | - | - | - | - | - | **H, 9** |

# Dijkstra

- Y por último H.

**u = H**

- El cual conecta solo con F, y no es más eficiente que el camino que ya descubrimos, así que solo eliminamos H de la lista pq.

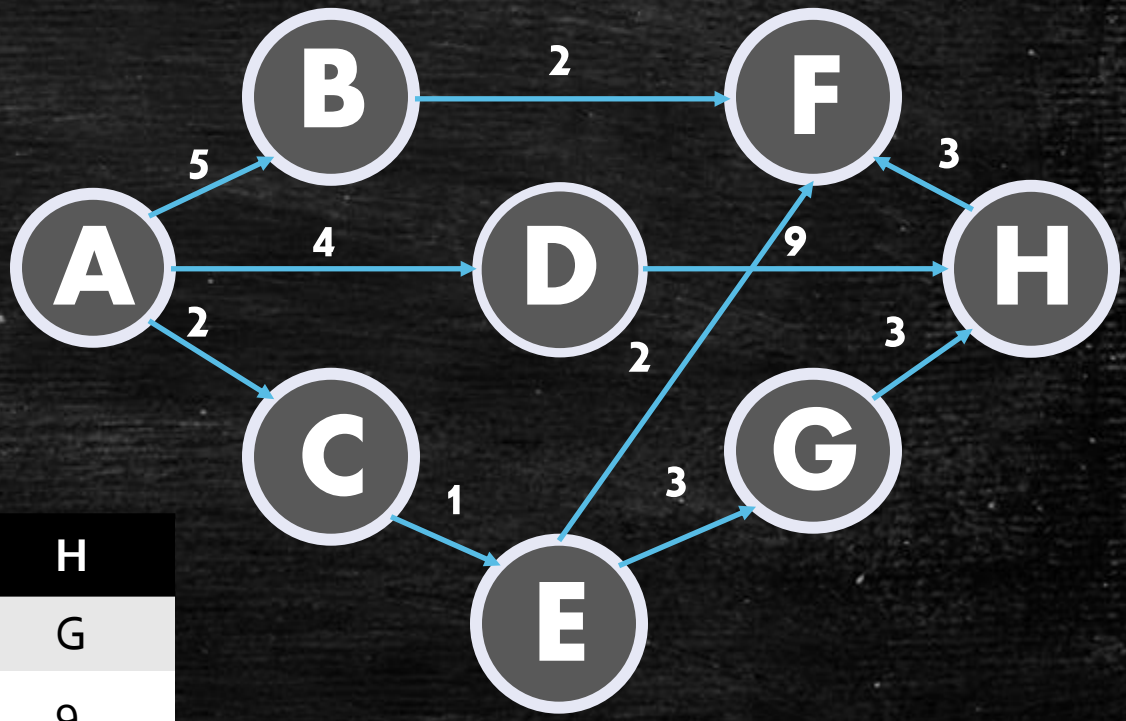



| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | C | E | E | G |
| Dist | 0 | 5 | 2 | 4 | 3 | 5 | 6 | 9 |
| pq | - | - | - | - | - | - | - | - |

# Dijkstra

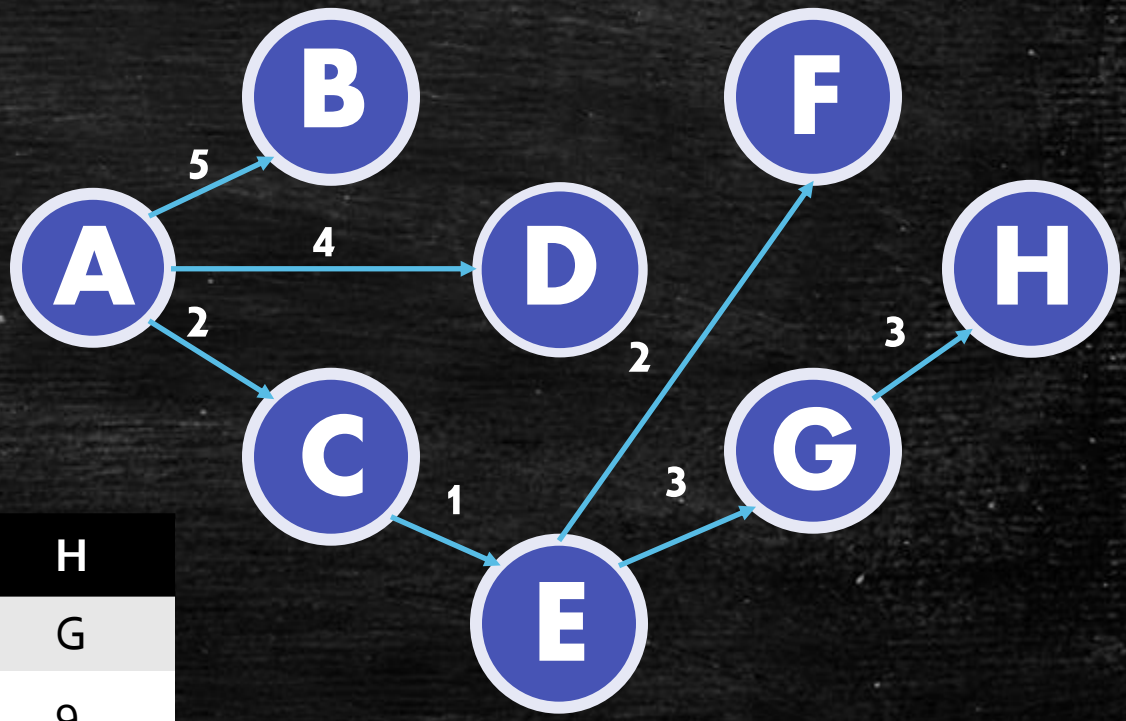
- Ahora terminamos!! Aunque podemos hacer algo más.
- Ahora que conocemos los caminos más eficientes, podemos rehacer nuestro grafo solo con las aristas que usaríamos. Ya que sabemos gracias a la lista pred que camino debemos tomar hacia cada vector.

| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | C | E | E | G |
| Dist | 0 | 5 | 2 | 4 | 3 | 5 | 6 | 9 |
| pq | - | - | - | - | - | - | - | - |



# Dijkstra

- Quedaría así!!




| Nodos | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| Pred | -1 | A | A | A | C | E | E | G |
| Dist | 0 | 5 | 2 | 4 | 3 | 5 | 6 | 9 |
| pq | - | - | - | - | - | - | - | - |

# Éxito en sus examenes!!! <3<3<3