uc3m | Universidad Carlos III de Madrid

OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 5 Árboles
5.1. Árboles Binarios

Profesores EDA

cc BY NC SA

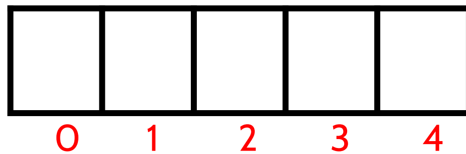
Índice

- **Introducción (conceptos básicos)**
- TAD Árbol Binario
  - Recorridos
  - Implementación
- TAD Árbol Binario de Búsqueda
- Equilibrado de árboles

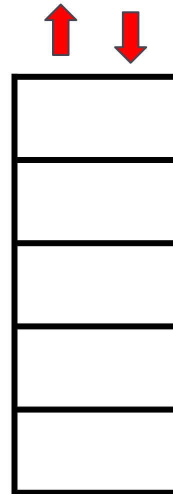
¿Cómo seleccionar una estructura de datos?

- ¿Qué tipo de datos necesitamos representar?
- Complejidad temporal de las operaciones.
- Complejidad espacial.
- Fácil de implementar.

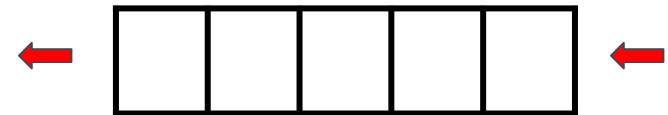
Estructuras de datos lineales



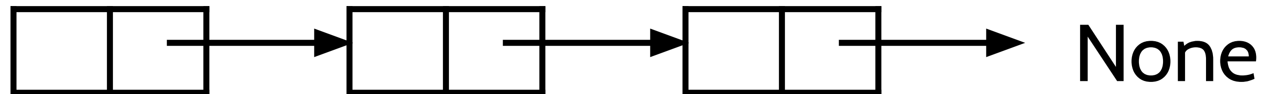
Lista de Python (array)



Pila



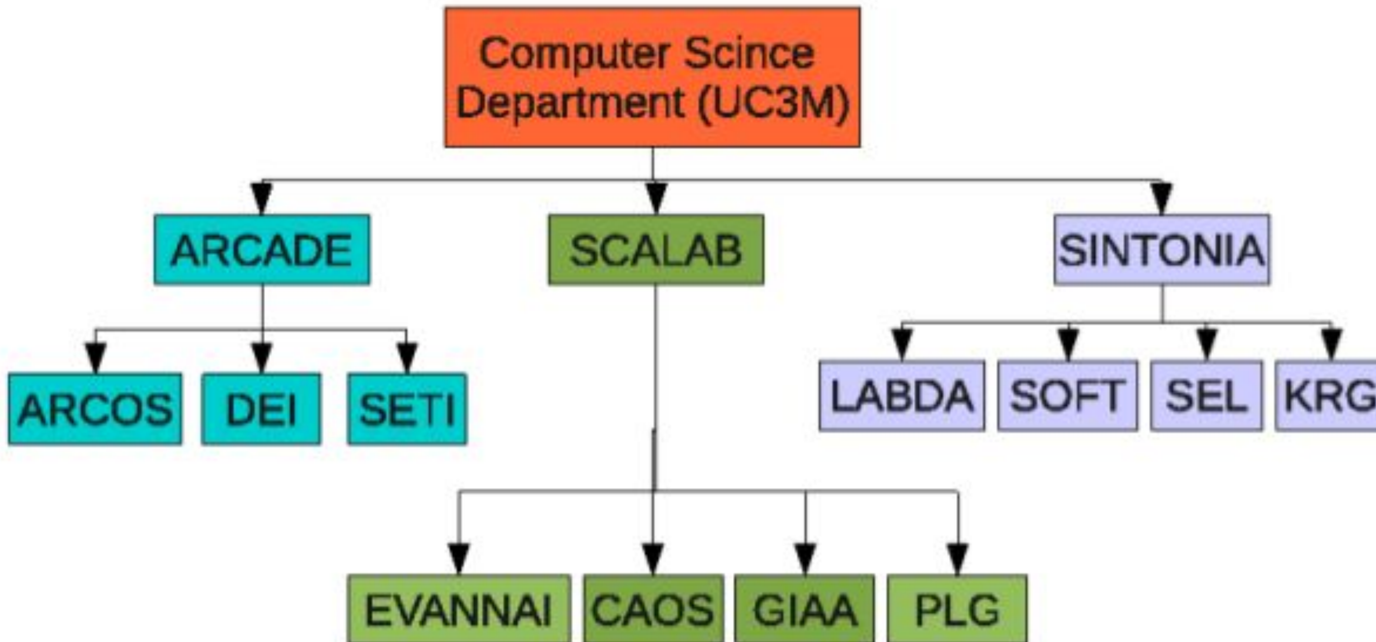
Cola



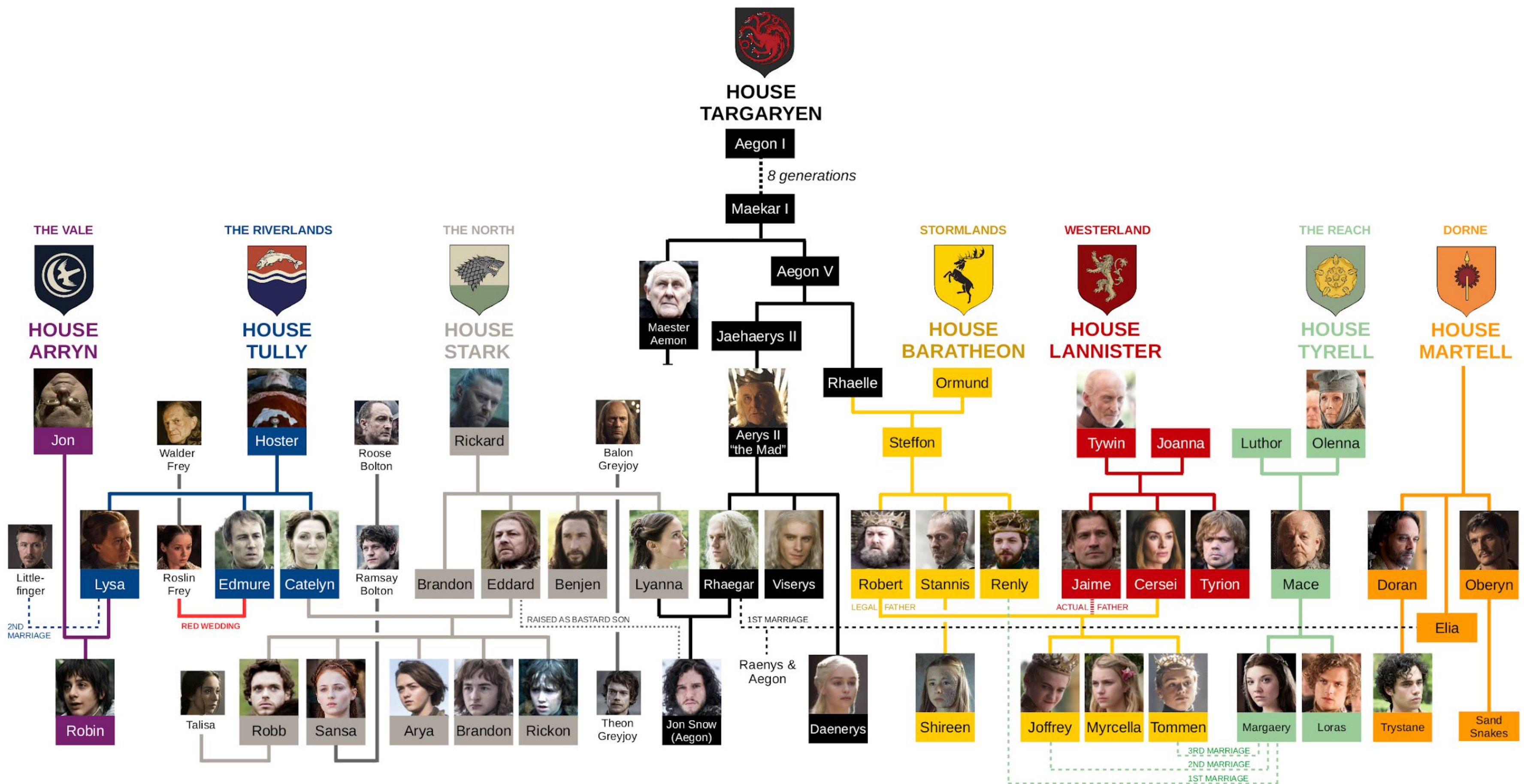
None

Lista Enlazada

¿Puedes representarlo con una estructura lineal?



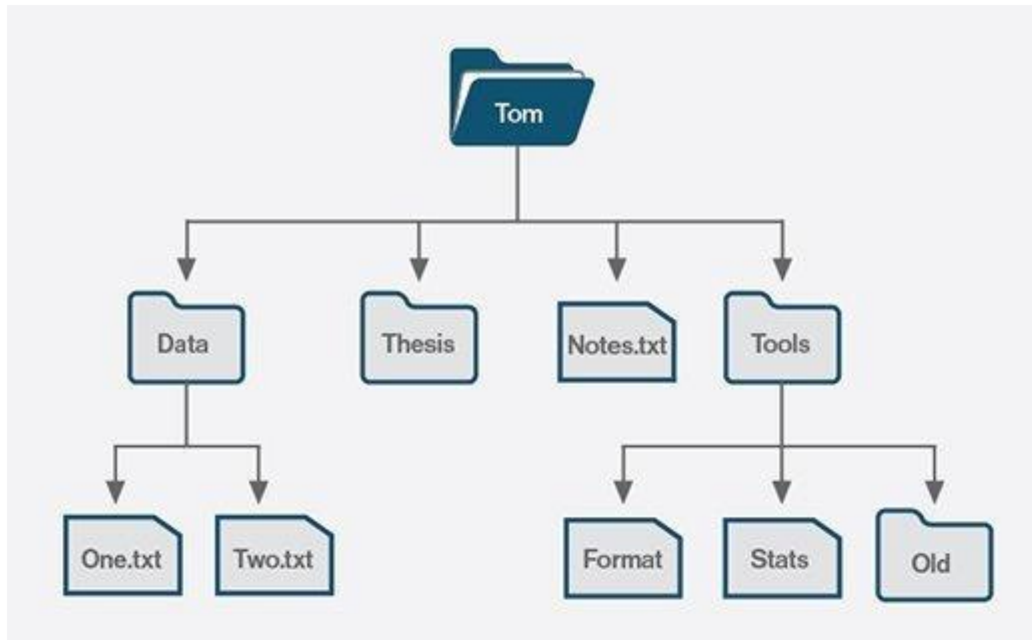
¿Y un árbol genealógico?



Game of Thrones Family Tree by Matt Baker - UsefulCharts.com. For educational purposes only - NOT FOR SALE.

Árboles

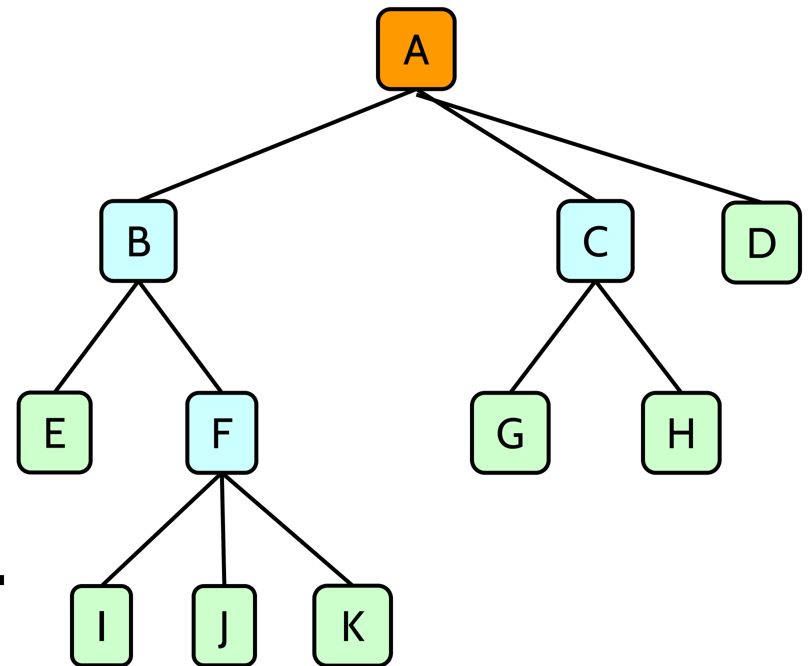
Representar datos jerárquicos



Sistemas de Ficheros

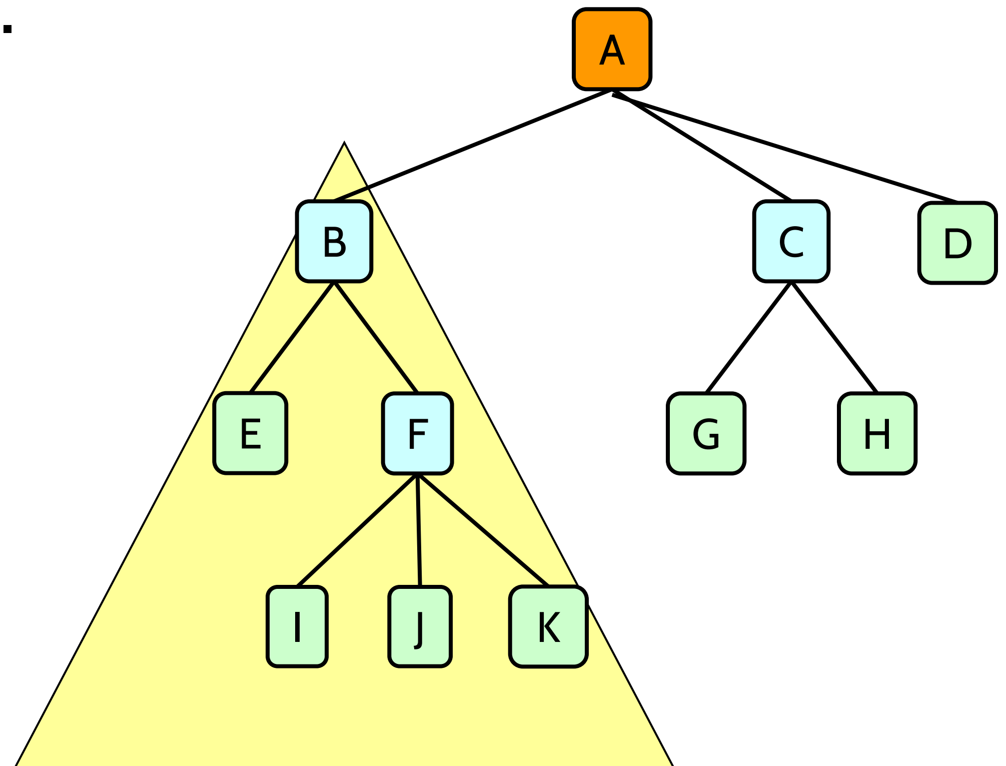
Conceptos básicos

- **Raíz**: el único nodo que no tiene padre (A)
- **Nodo interno**: un nodo que al menos tiene un hijo (A, B, C, F)
- **Nodo hoja (Externo)**: nodo sin hijos (E, I, J, K, G, H, D)
- **Hermanos**: nodos con el mismo padre.
- **Ascendientes y descendientes.**



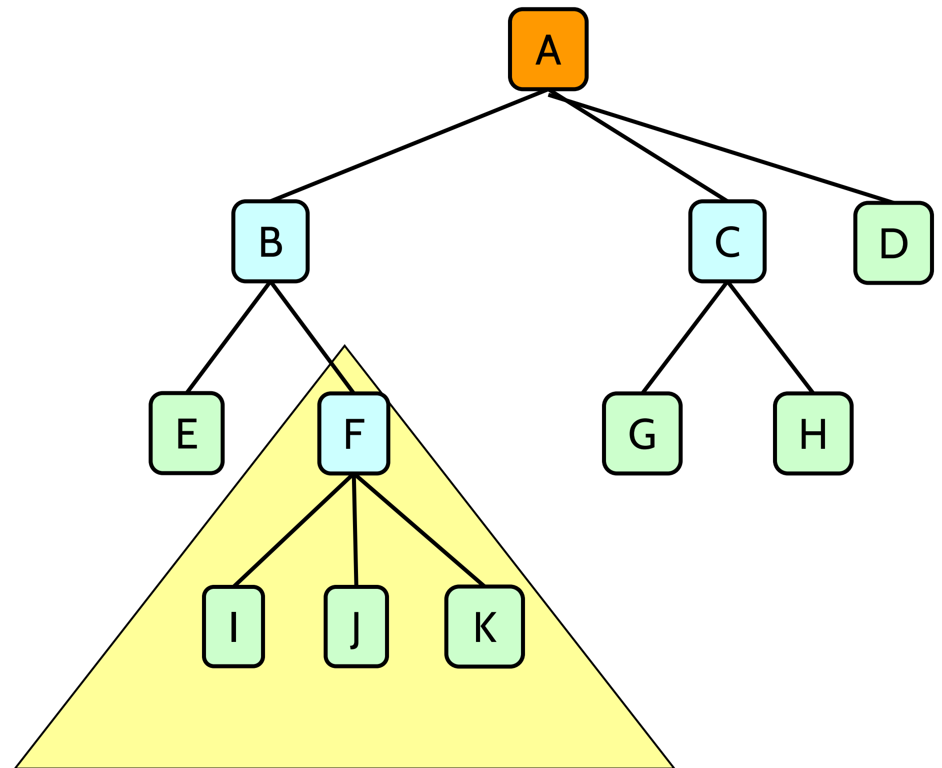
Conceptos básicos

- **Subárbol**: árbol formado por un nodo (por ejemplo, B) y todos sus descendientes.



Conceptos básicos

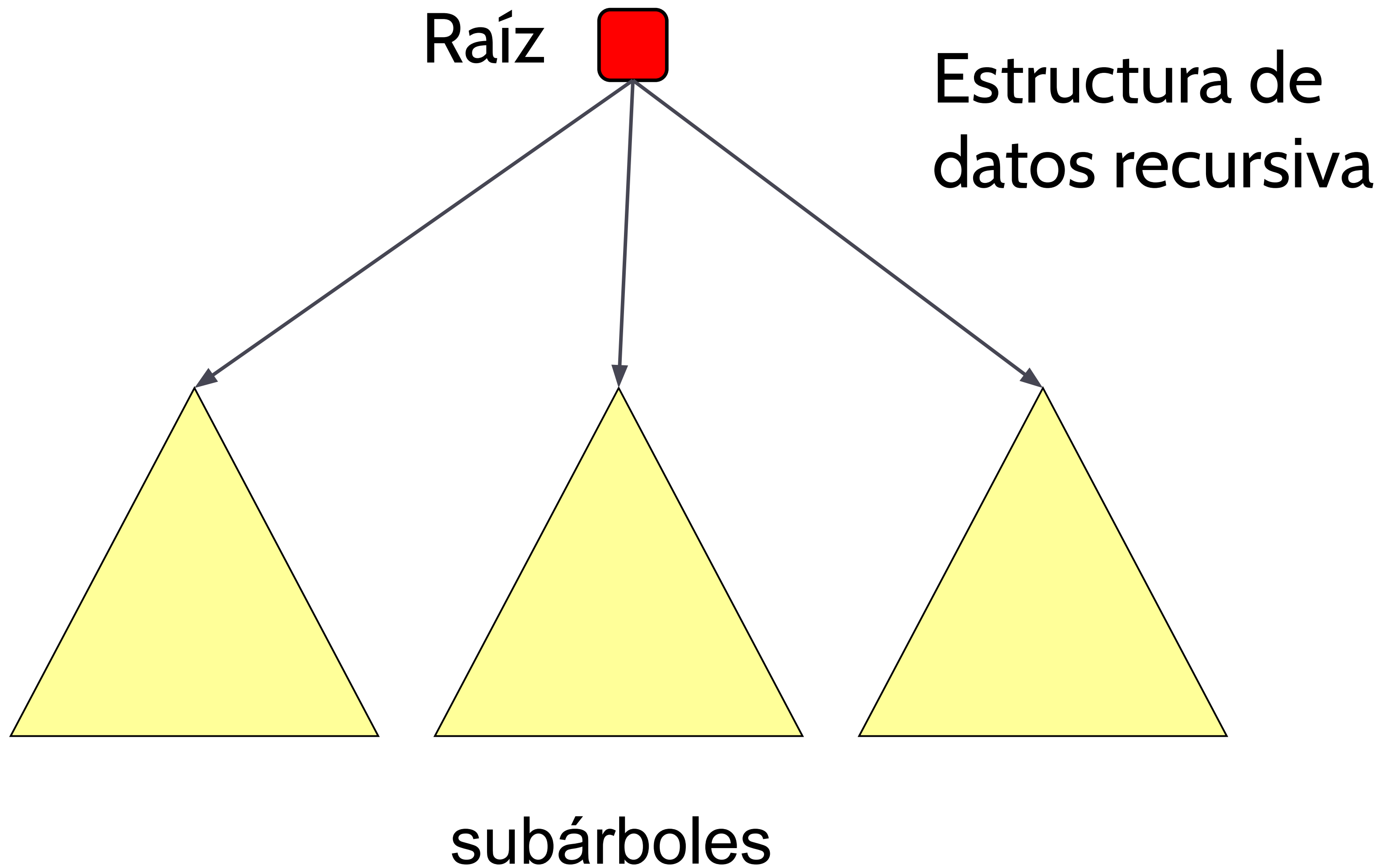
- En este caso, el triángulo amarillo está mostrando el subárbol que cuelga de F.



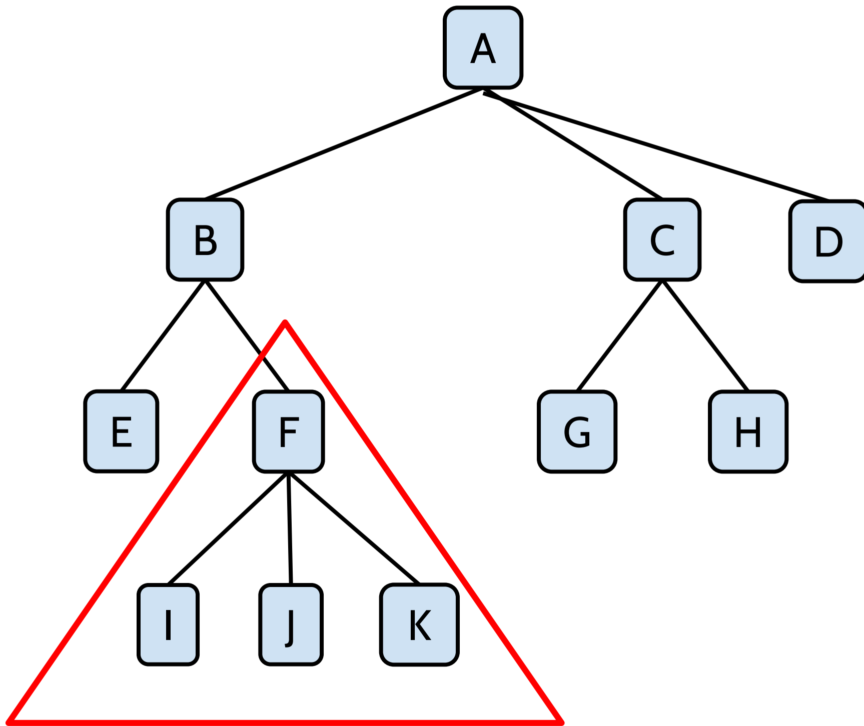
Definición (formal)

- Un árbol, T, es un conjunto de nodos con relaciones padre-hijo, que cumple:
  - El árbol (si no está vacío) tiene **una única raíz**. La raíz no tiene padre.
  - Cada nodo del árbol, T, tiene **un único padre**.

Árbol



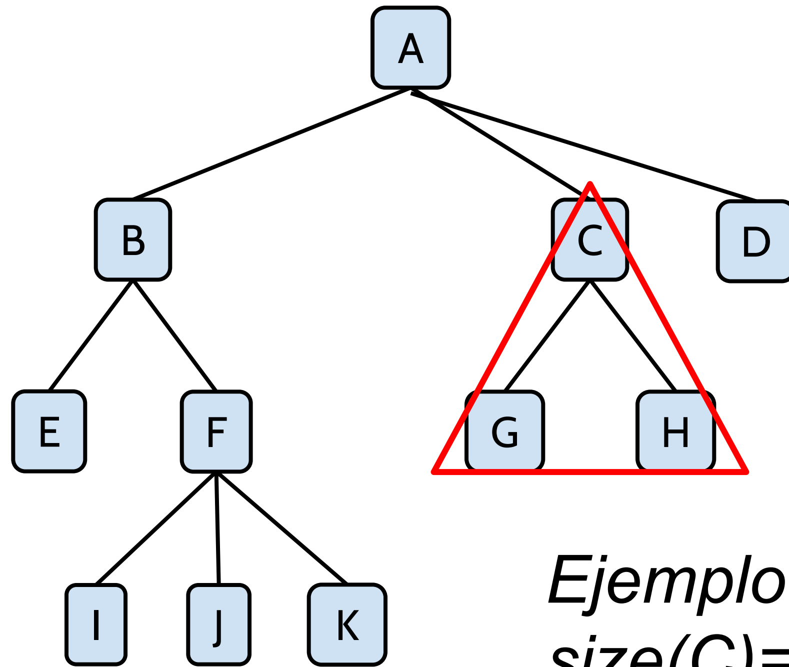
Tamaño de un nodo o subárbol



El tamaño de un nodo se define como el tamaño de su subárbol

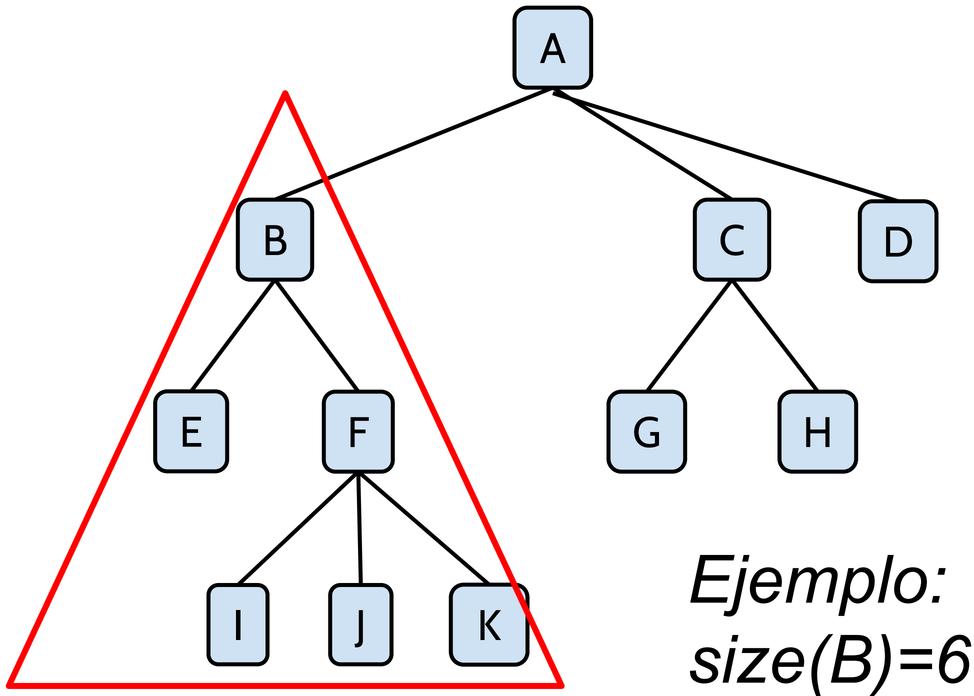
Ejemplo:
size(F)=4

Tamaño de un nodo o subárbol

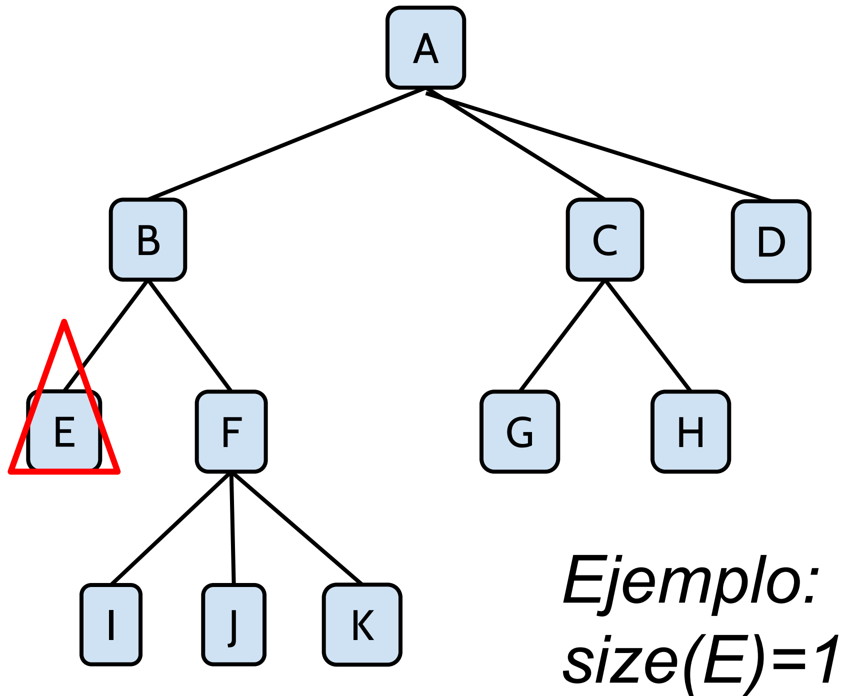


Ejemplo:
size(C)=3

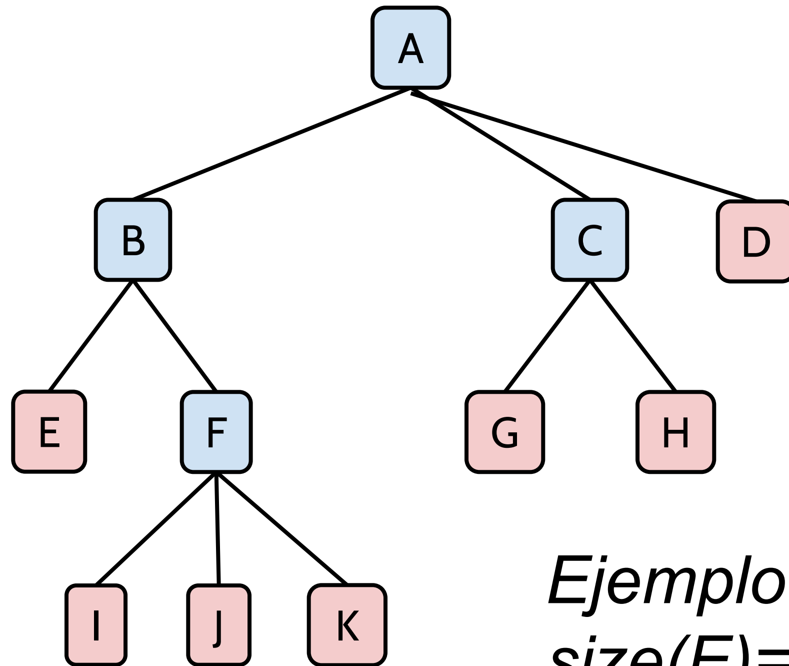
Tamaño de un nodo o subárbol



Tamaño de un nodo o subárbol



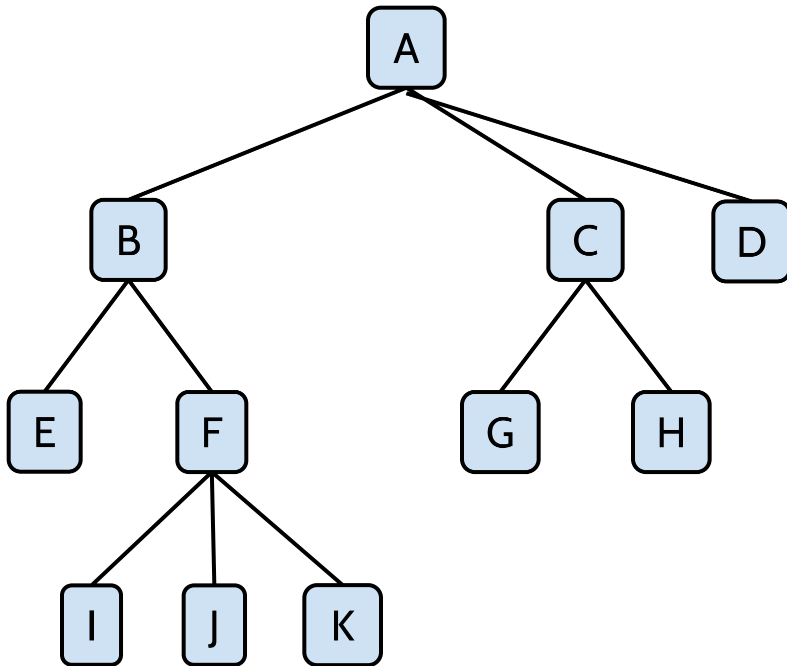
Tamaño de un nodo o subárbol



Todos los nodos hojas tienen tamaño 1

Ejemplo:
size(E)=1
size(I)=1
size(J)=1
size(K)=1
size(D)=1

Tamaño de un árbol



El tamaño de un árbol se define como el número de sus nodos = el tamaño de su nodo raíz

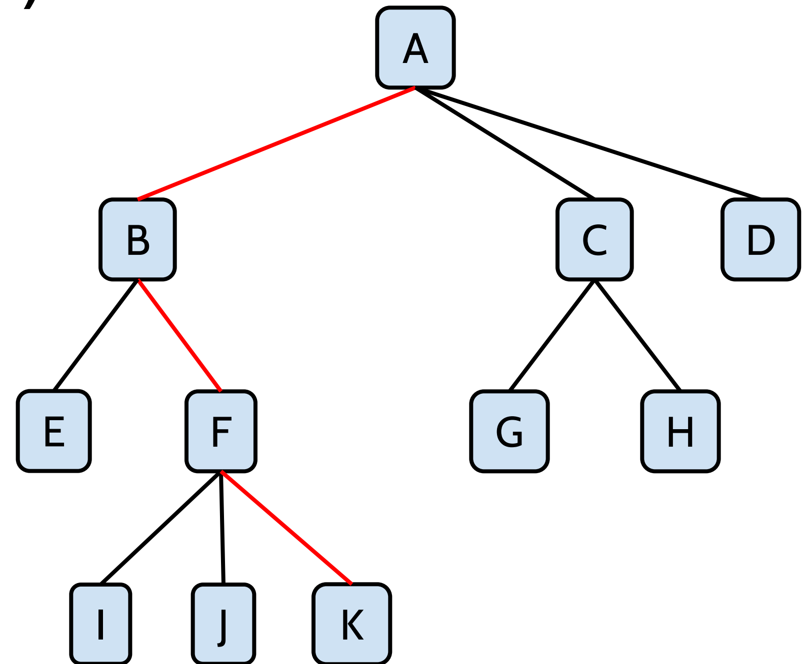
Ejemplo:
size(T)=size(A)=11

Camino

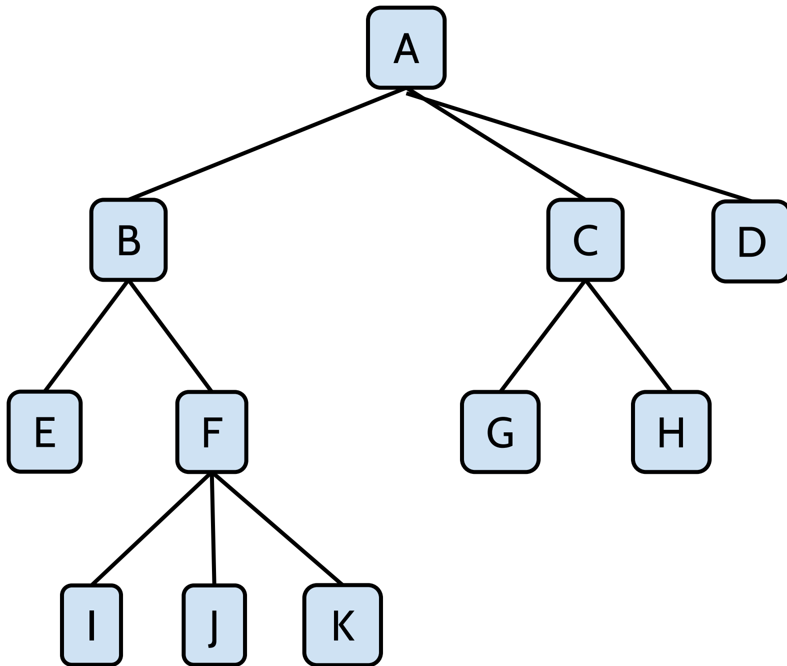
- **Camino:** existe un camino entre los nodos X e Y, si existe un secuencia de nodos que permita alcanzar Y desde X (siempre de forma descendente o ascendente, nunca ambas).

path(A,K)={A,B,F,K}

path(C,K)={}

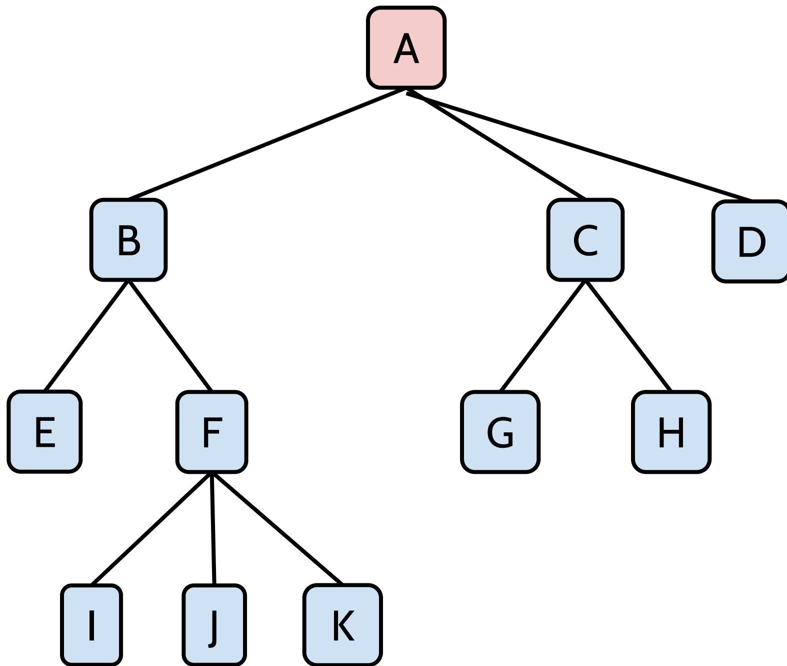


Profundidad (depth) de un nodo



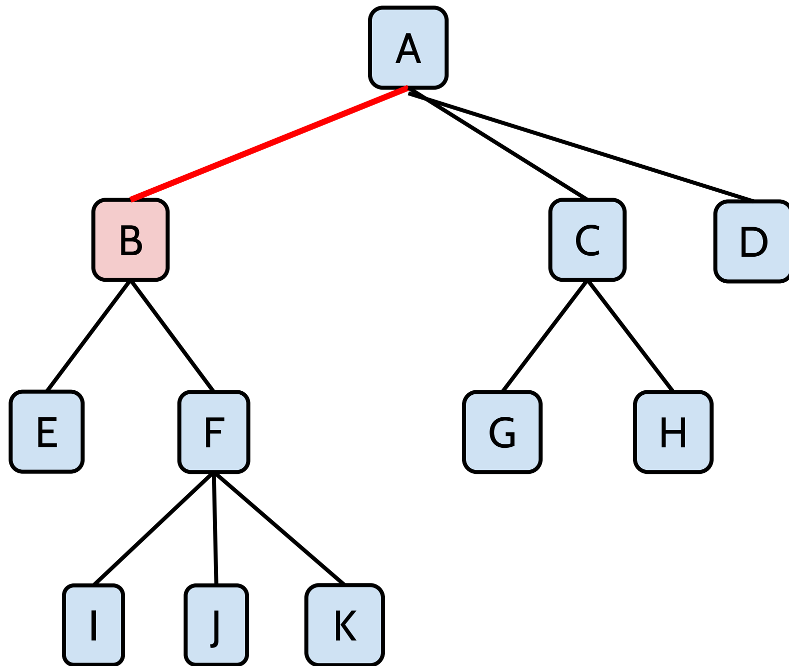
La profundidad de un nodo es la longitud del camino de la raíz a dicho nodo.

Profundidad (depth) de un nodo



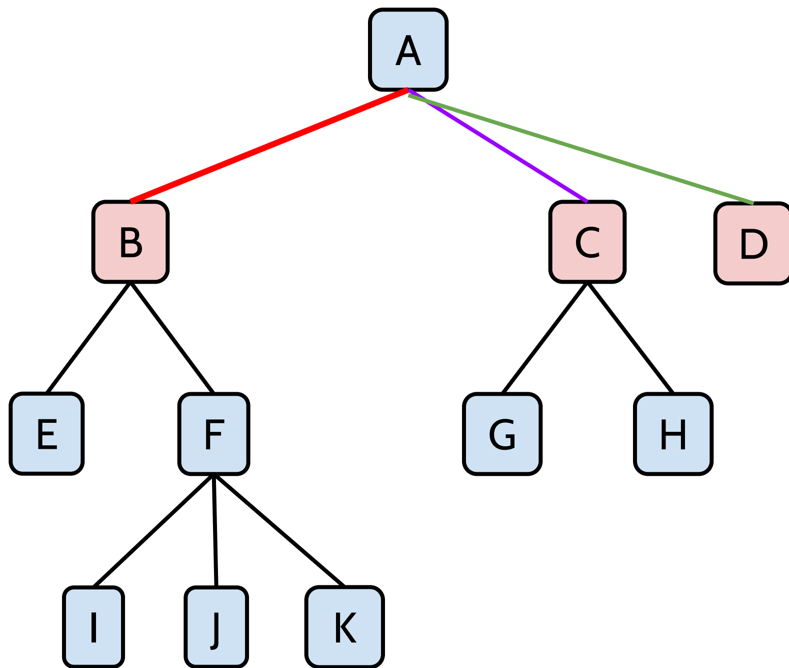
Ejemplos:
depth(A)=0

Profundidad (depth) de un nodo



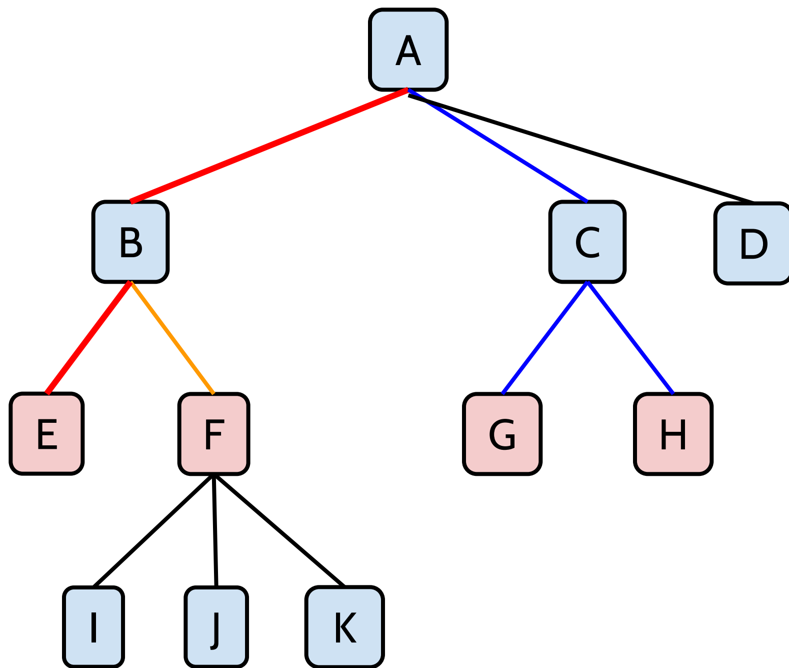
Ejemplos:
depth(B)=1

Profundidad (depth) de un nodo



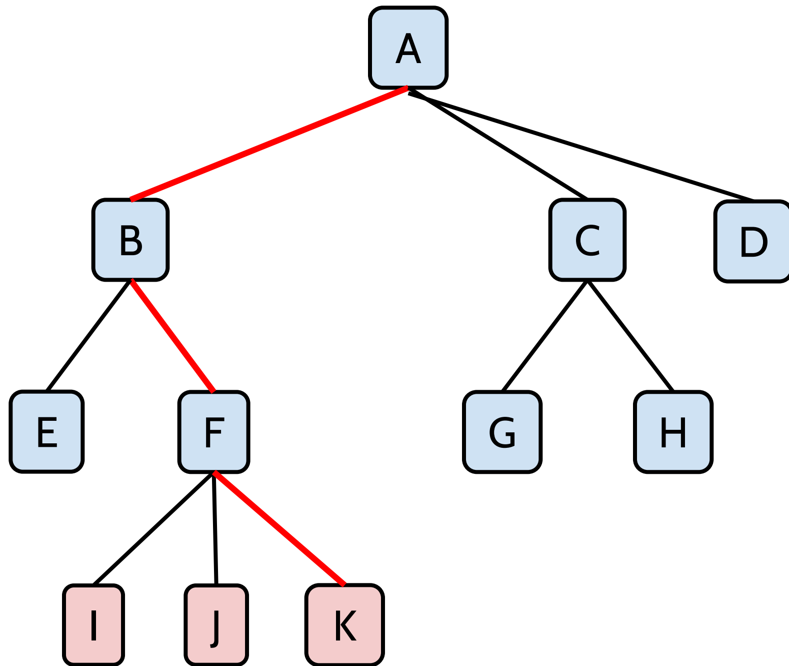
Ejemplos:
depth(B)=1
depth(C)=1
deptn(D)=1

Profundidad (depth) de un nodo



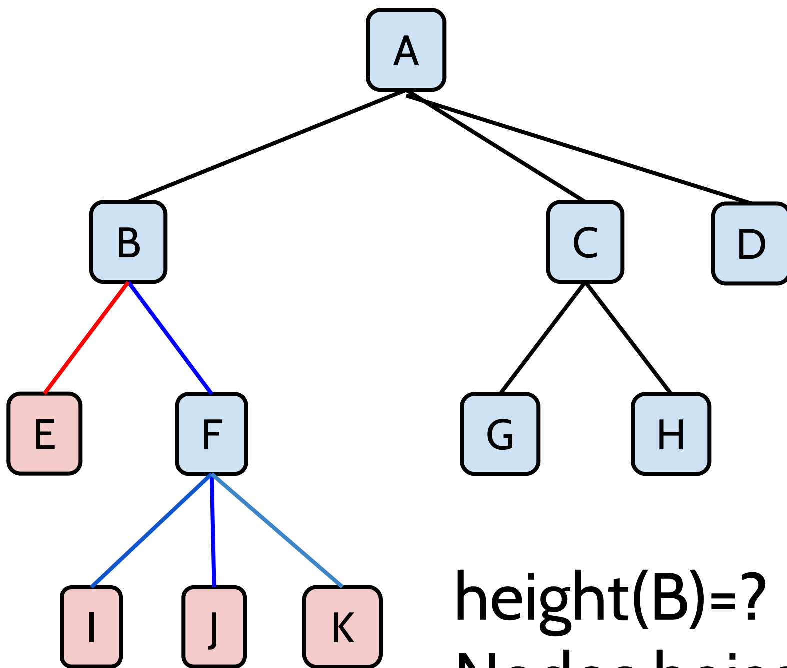
Ejemplos:
depth(E)=2
depth(F)=2
depth(G)=2
depth(H)=2

Profundidad (depth) de un nodo



Ejemplos:
depth(I)=3
depth(J)=3
depth(K)=3

Altura de un nodo



La altura de un nodo es la longitud del camino más largo desde dicho nodo a una hoja.

height(B)=?

Nodos hojas que cuelgan de B son: E, I, J, K

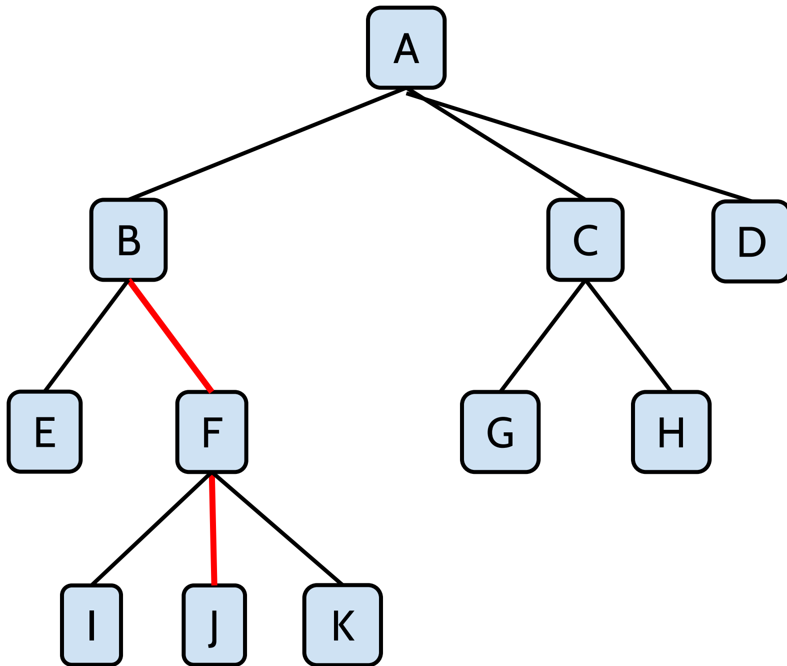
camino(B,E)=B->E, longitud=1

camino(B,I)=B->F->I, longitud=2

camino(B,J)=B->F->J, longitud=2

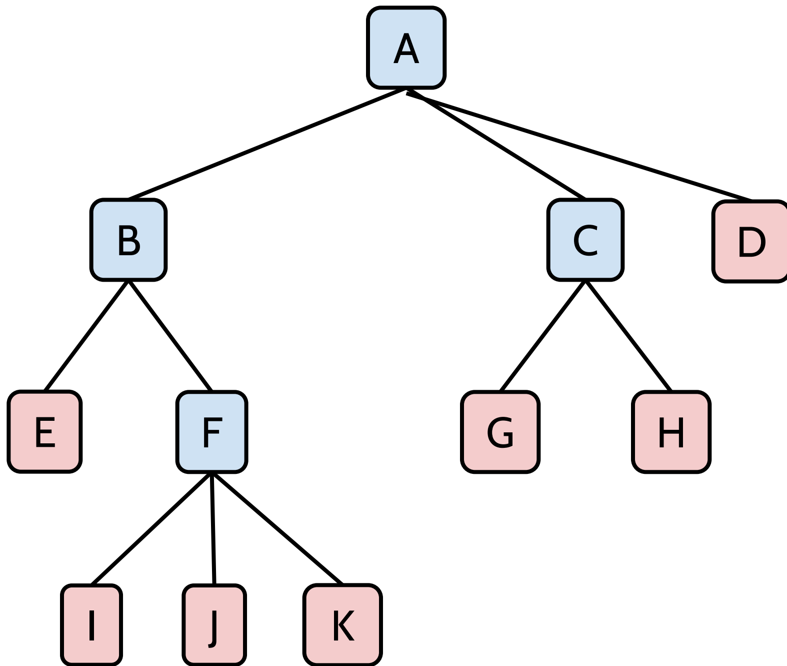
camino(B,K)=B->F->K, longitud=2

Altura de un nodo



Ejemplos:
height(B)=2

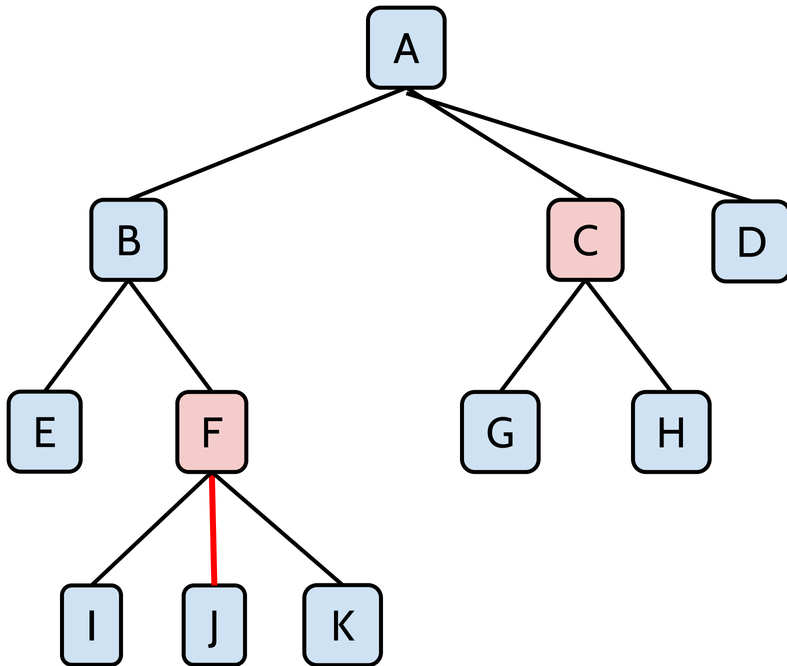
Altura de un nodo



Todos los nodos hojas tendrán altura cero.

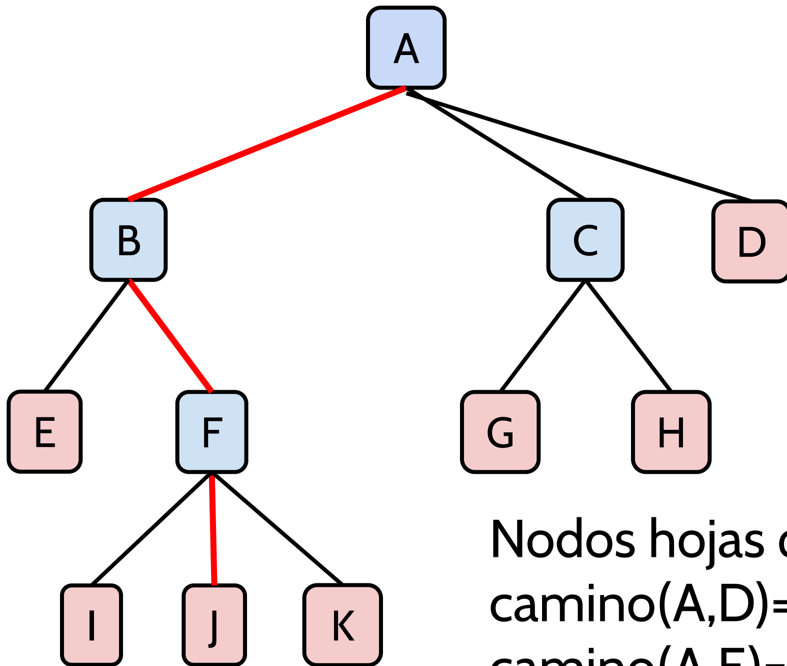
Ejemplos:
height(E)=0
height(I)=0
height(J)=0
height(K)=0
height(G)=0
height(H)=0
height(D)=0

Altura de un nodo



Ejemplos:
height(F)=1
height(C)=1

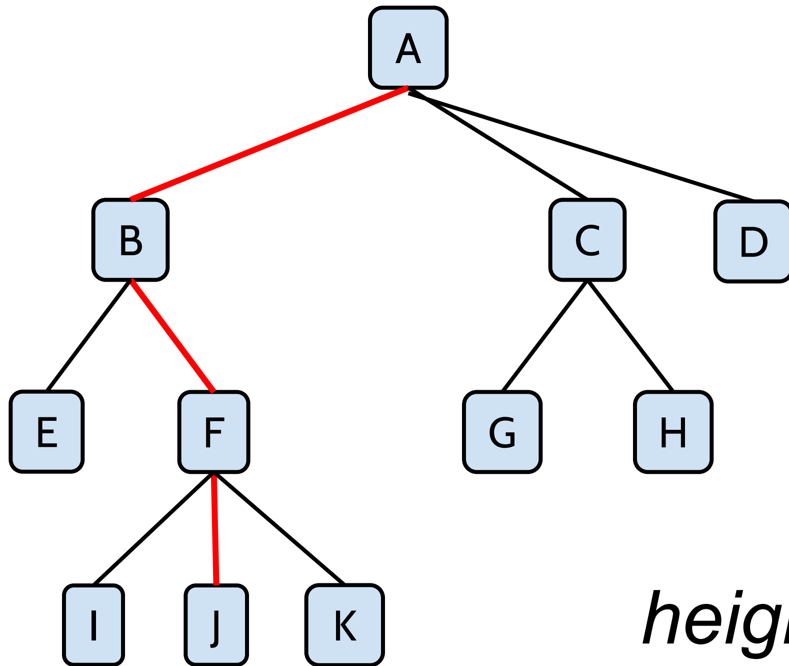
Altura de un nodo



Ejemplos:
height(A)=3

Nodos hojas que cuelgan de A son: E, I, J, K, G, H, D
camino(A,D)=A->D, longitud=1
camino(A,E)=A->B->E, longitud=2
camino(A,G)=A->C->G, longitud=2
camino(A,H)=A->C->H, longitud=2
camino(A,I)=A->F->I, longitud=3
camino(A,J)=A->F->J, longitud=3
camino(A,K)=A->F->K, longitud=3

Altura de un árbol

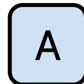


La altura de un árbol se define como la altura de su raíz

height(T)= height(A)=3

Altura de un árbol

Si un árbol sólo tiene un nodo, su altura será 0, porque su raíz es una hoja



Altura de un árbol

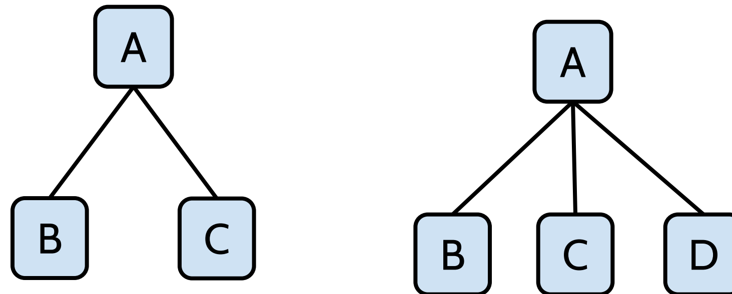
Un árbol nulo o vacía es aquel cuya raíz es None.
Para distinguir estos casos, vamos a definir la altura de un nodo nulo (None) como -1, es decir,

Si node=None, height(None)=-1

Por tanto, la altura de un árbol vacío será -1.

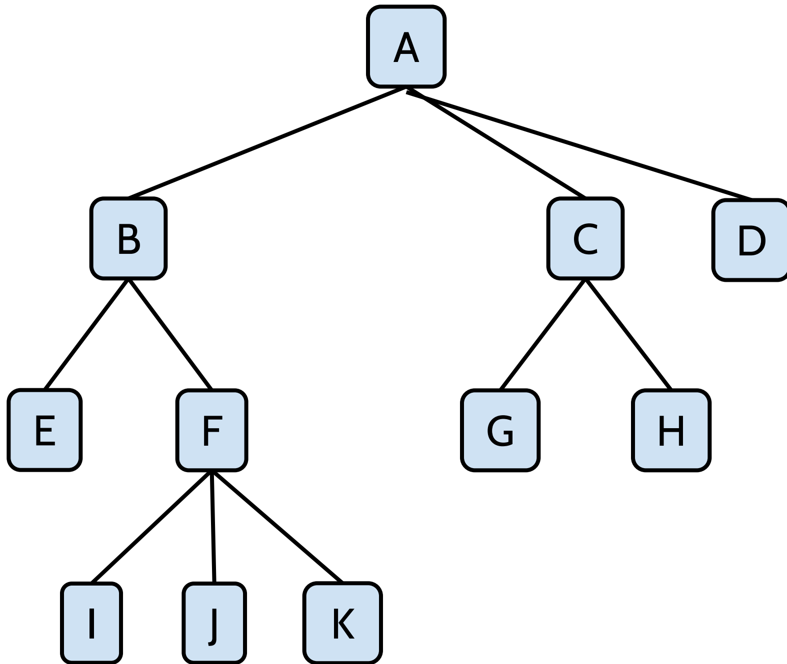
Grado de un nodo y de un árbol

- El grado de un nodo es el número de hijos directos.



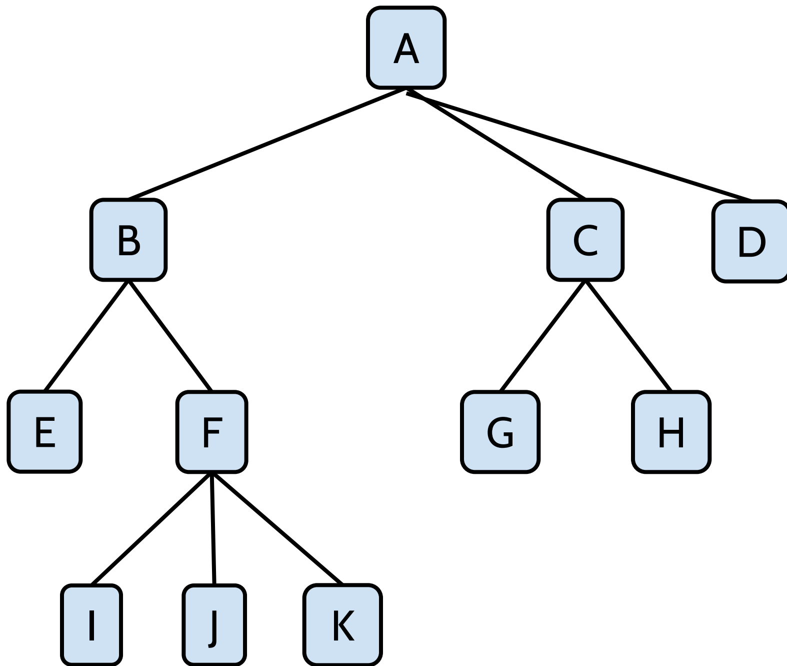
- El grado de un árbol es el grado mayor de todos sus nodos.

Pregunta:



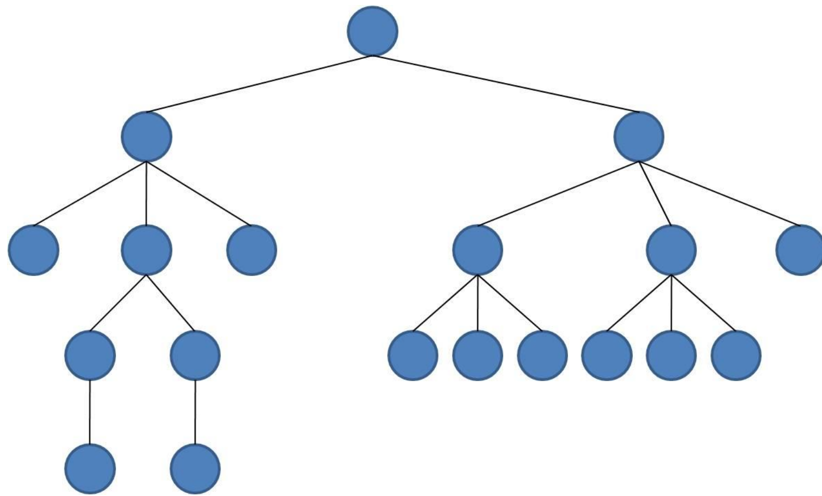
¿Cuál es el grado de este árbol?

Respuesta



Grado 3, porque el máximo grado de sus nodos es 3 (nodos A y F).

Ejercicio 1



Dado el árbol de la imagen:

- Grado del árbol?
- Altura?
- Número de nodos?
- Número de hojas?
- Número de nodos internos?
- Número de nodos con profundidad 2?

Ejercicio 2

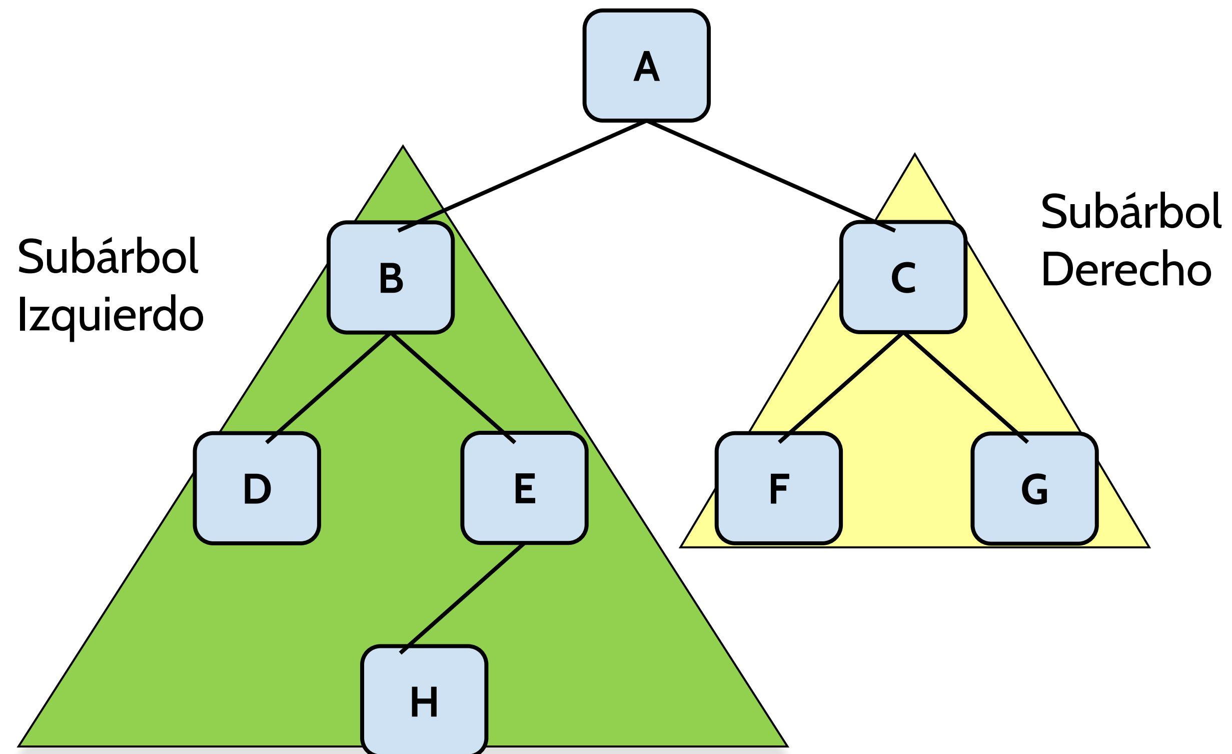
2. Dibuja un árbol que cumpla las siguientes propiedades:
   - Grado del árbol: 3
   - Número de nodos: 19
   - Número hojas:11
   - Nodos internos: 8
   - Altura del árbol: 4
   - Número de nodos con profundidad 2: 6

Índice

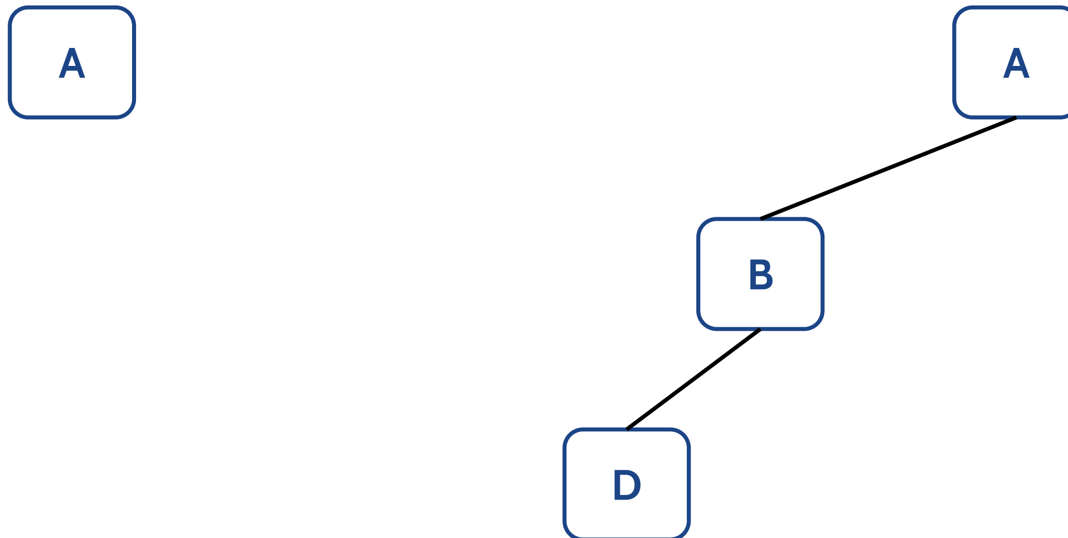
- Introducción (conceptos básicos)
- **TAD Árbol Binario**
  - Recorridos
  - Implementación
- TAD Árbol Binario de Búsqueda
- Equilibrado de árboles

Árbol binario

- Árbol de grado 2. Cada nodo tiene como máximo dos hijos



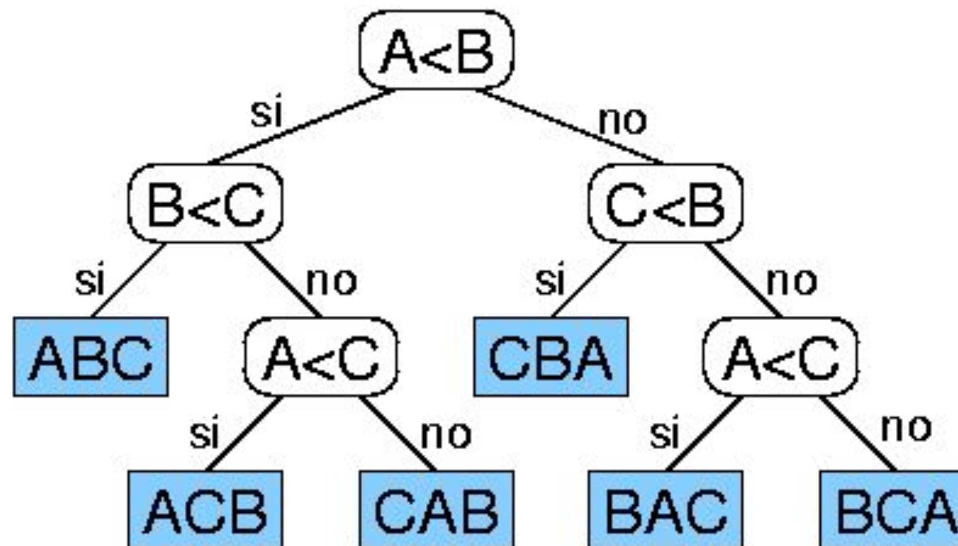
¿Son árboles binarios?



Árboles binarios: Aplicaciones

Ejemplo I: Árboles de decisión

- Nodos internos: preguntas con respuestas sí/no
- Nodos hojas: decisiones



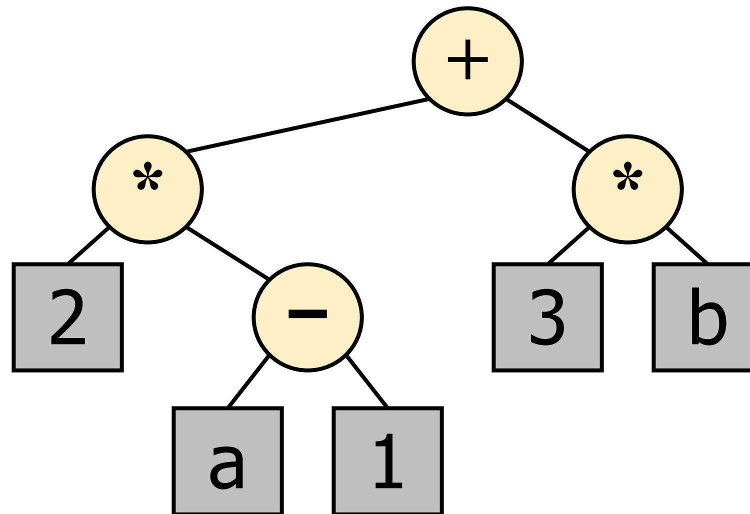
Ejemplo de árbol de decisión para ordenar tres números

Árboles binarios: Aplicaciones

Ejemplo 2: representación de expresiones aritméticas.

- Nodos internos: operadores
- Nodos hojas: operandos

2*(a-1)+3*b



Índice

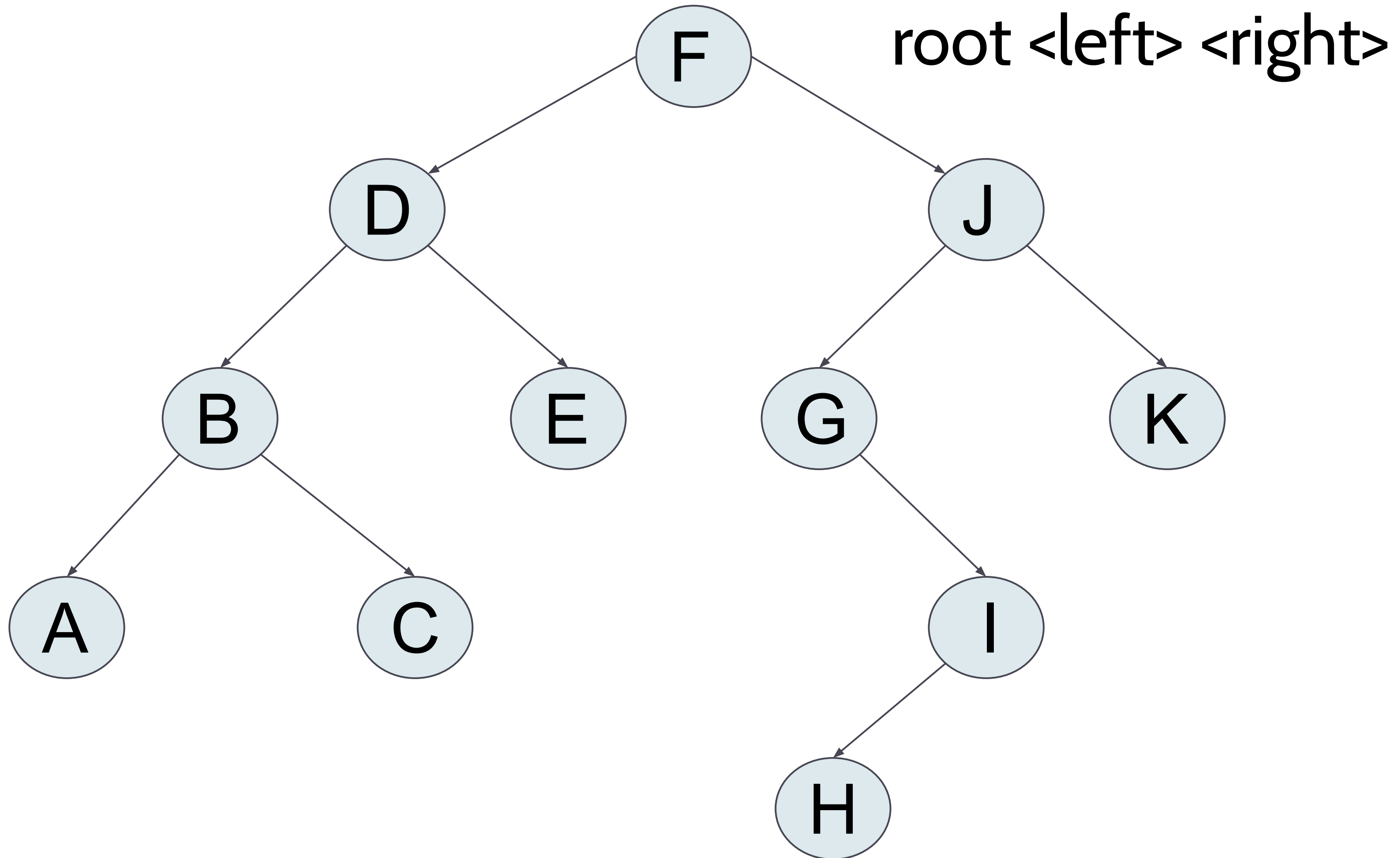
- Introducción (conceptos básicos)
- **TAD Árbol Binario**
  - **Recorridos**
  - Implementación
- TAD Árbol Binario de Búsqueda
- Equilibrado de árboles

Árboles Binarios: recorridos

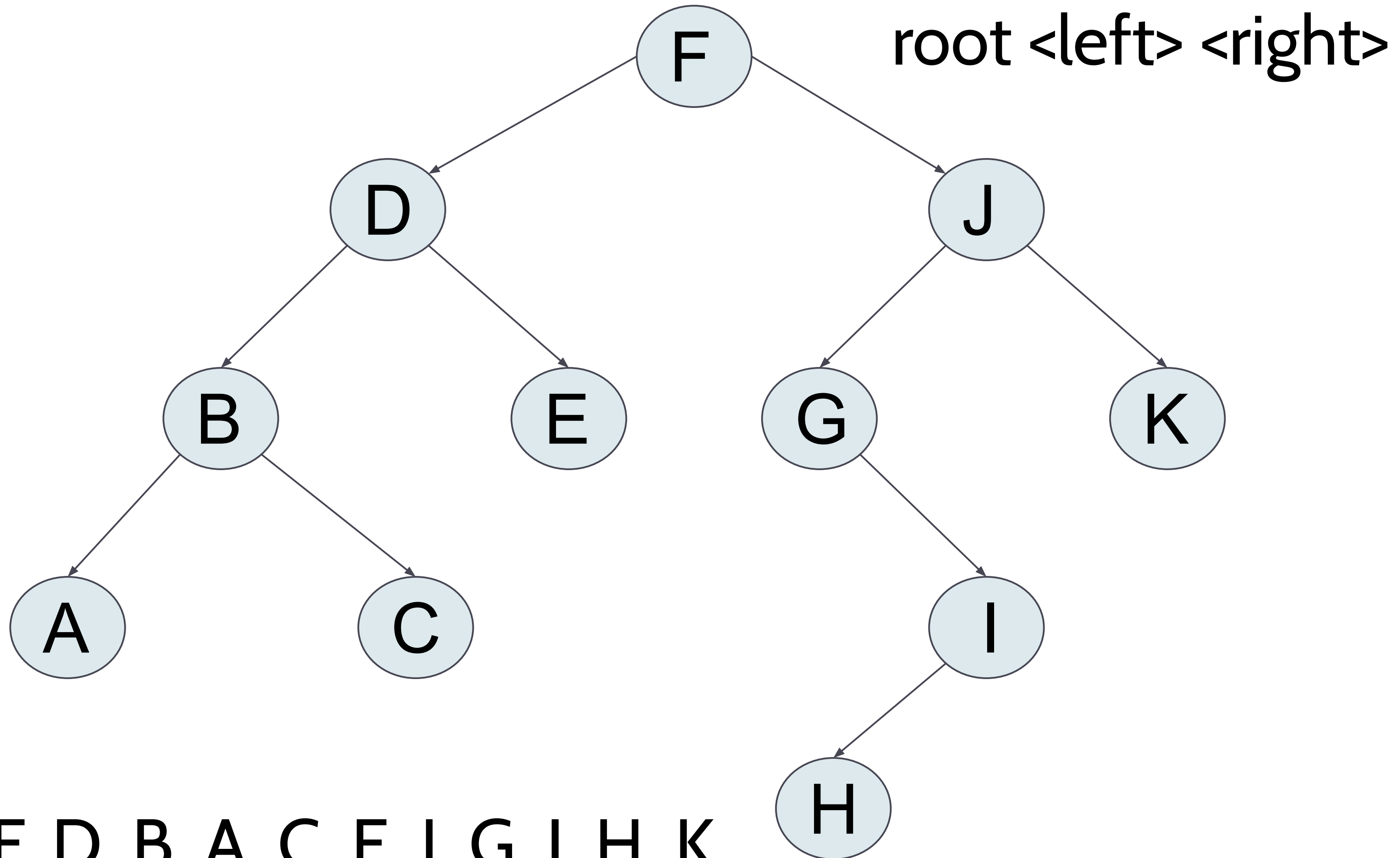
Recorridos pre-order:

- Primero, visitamos la raíz, entonces el árbol izquierdo, y finalmente, el subárbol derecho (**root, left, right**). Cada subárbol es visitado recursivamente aplicando pre-order.

Árboles Binarios: pre-order



Árboles Binarios: pre-order

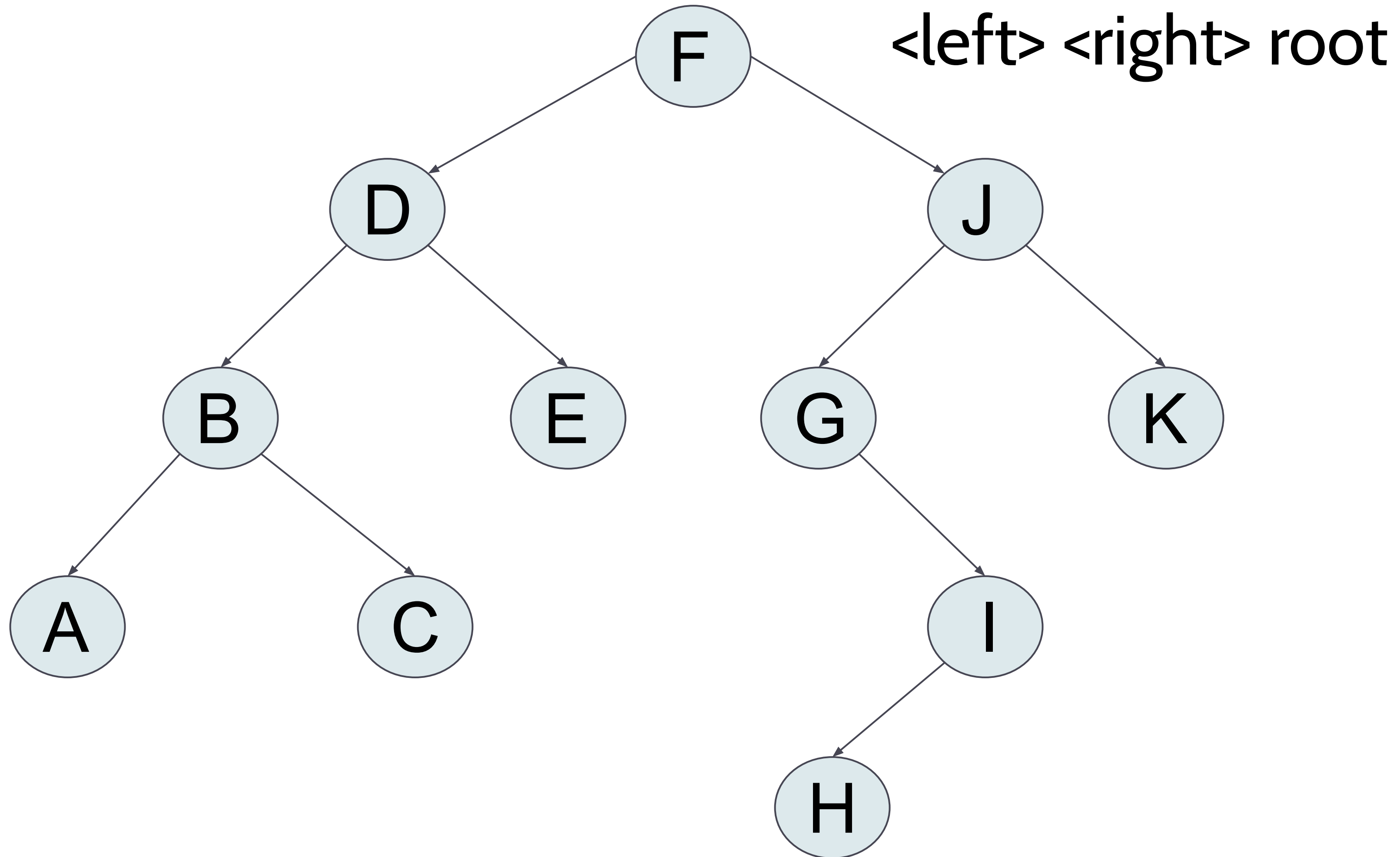


Árboles Binarios: recorridos

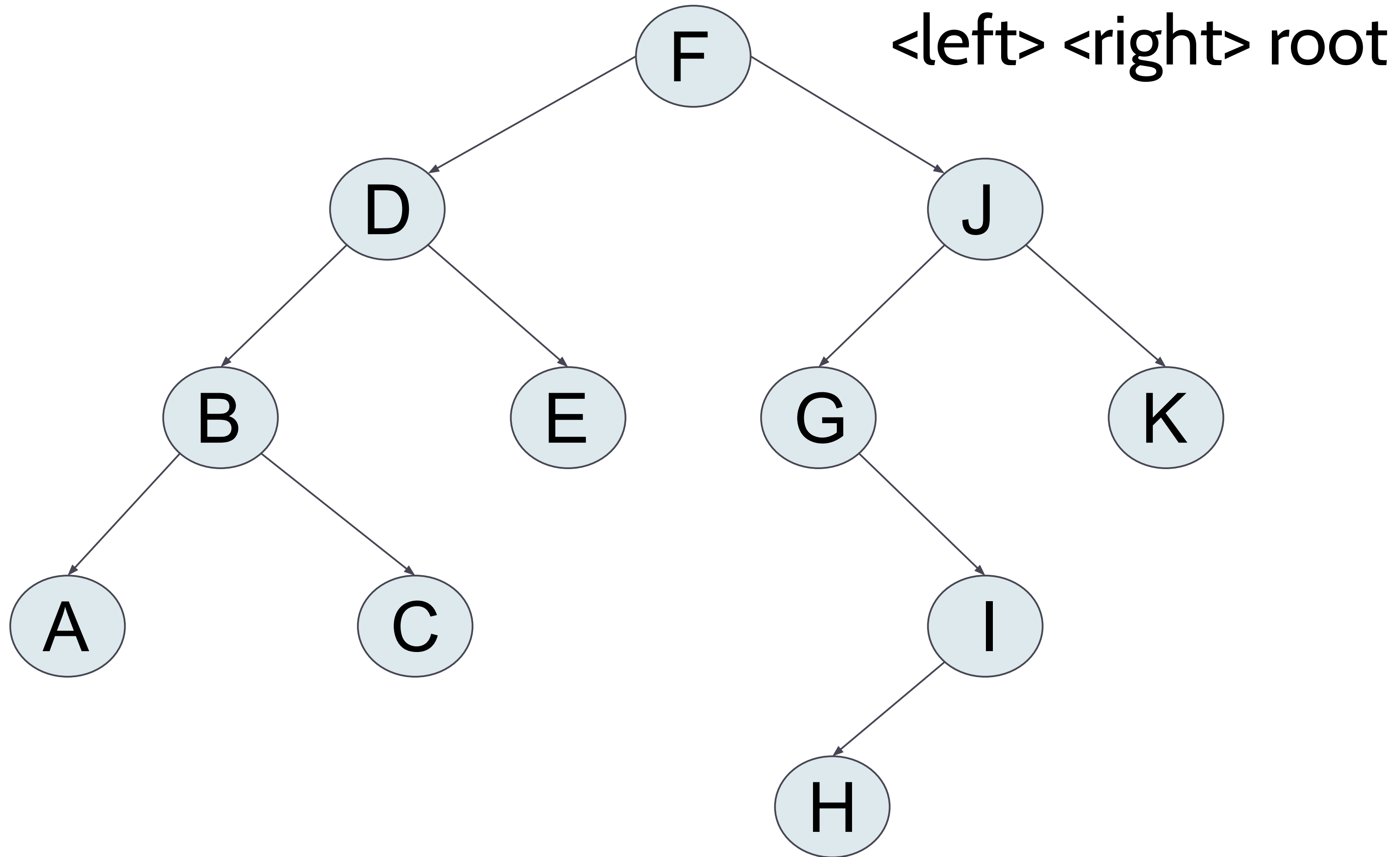
Recorrido post-order:

- Primero, visitamos el subárbol izquierdo, luego el subárbol derecho, y finalmente, la raíz. (**left, right, root**). Cada subárbol es visitado recursivamente aplicando pre-order.

Árboles Binarios: post-order



Árboles Binarios: post-order

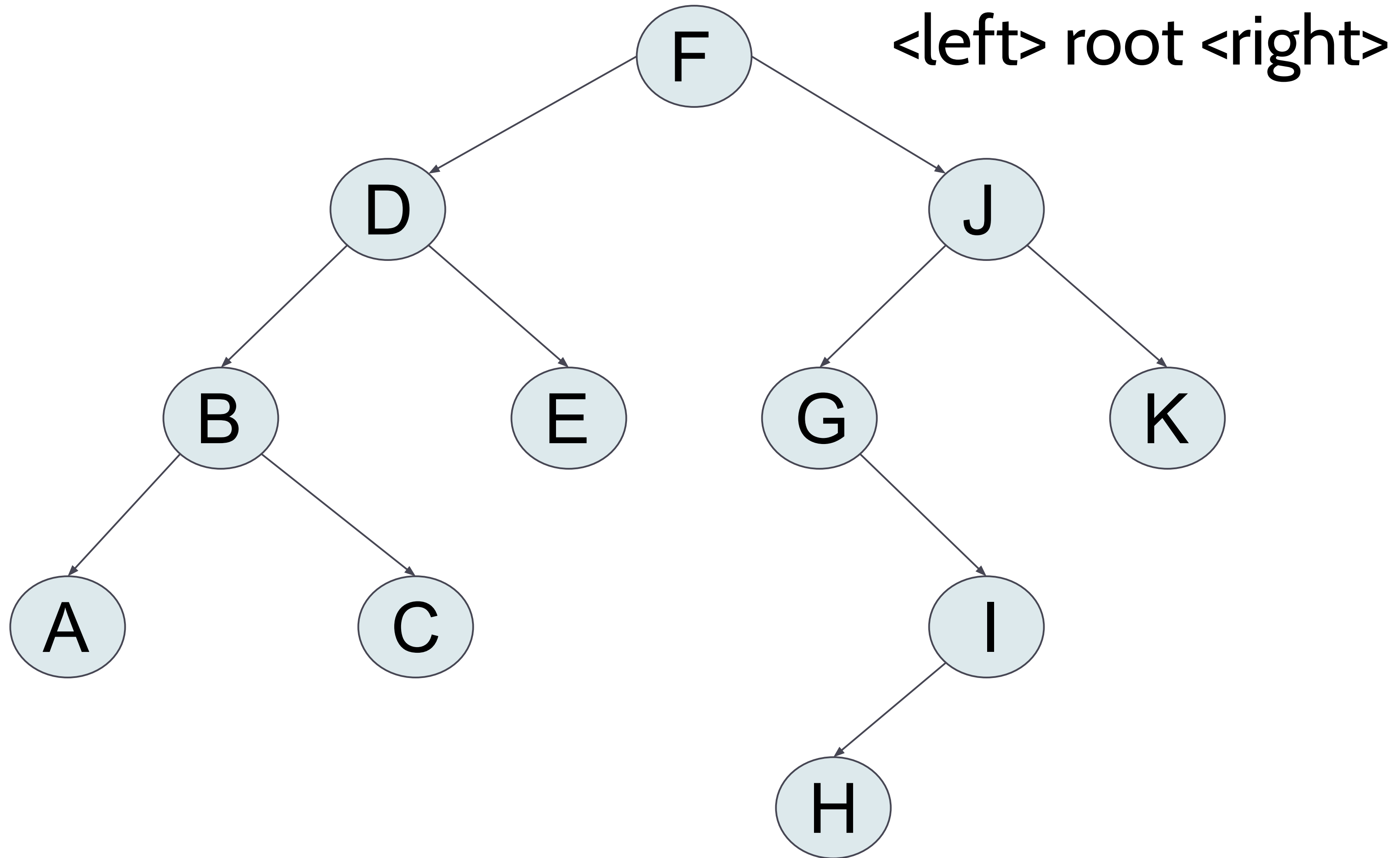


A, C, B, E, D, H, I, G, K, J, F

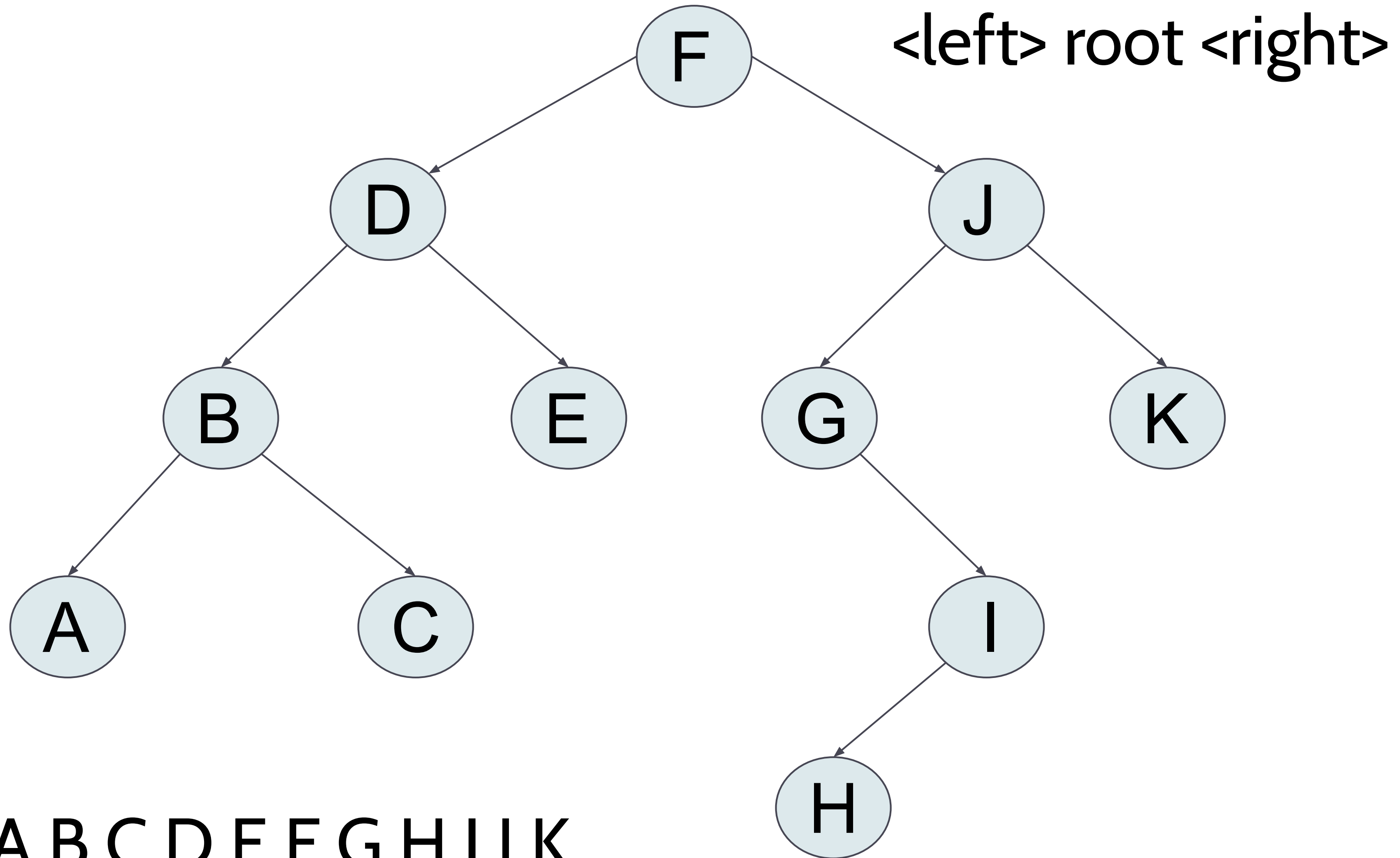
Árboles Binarios: recorridos

Recorrido in-order: primero visitamos el subárbol izquierdo, la raíz y el subárbol derecho. Cada subárbol es visitado recursivamente aplicando el recorrido in-order: (**left, root, right**).

Árboles Binarios: in-order



Árboles Binarios: in-order

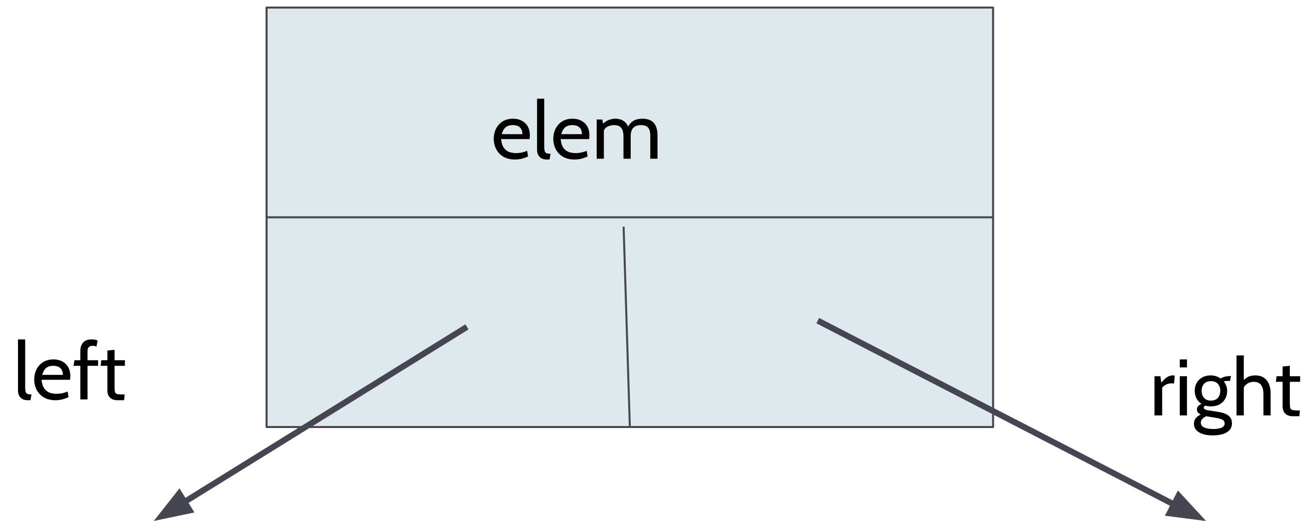


Índice

- Introducción (conceptos básicos)
- **TAD Árbol Binario**
  - Recorridos
  - **Implementación**
- TAD Árbol Binario de Búsqueda
- Equilibrado de árboles

¿Cómo representar un nodo de un árbol binario?

Nodo Binario (BinaryNode)

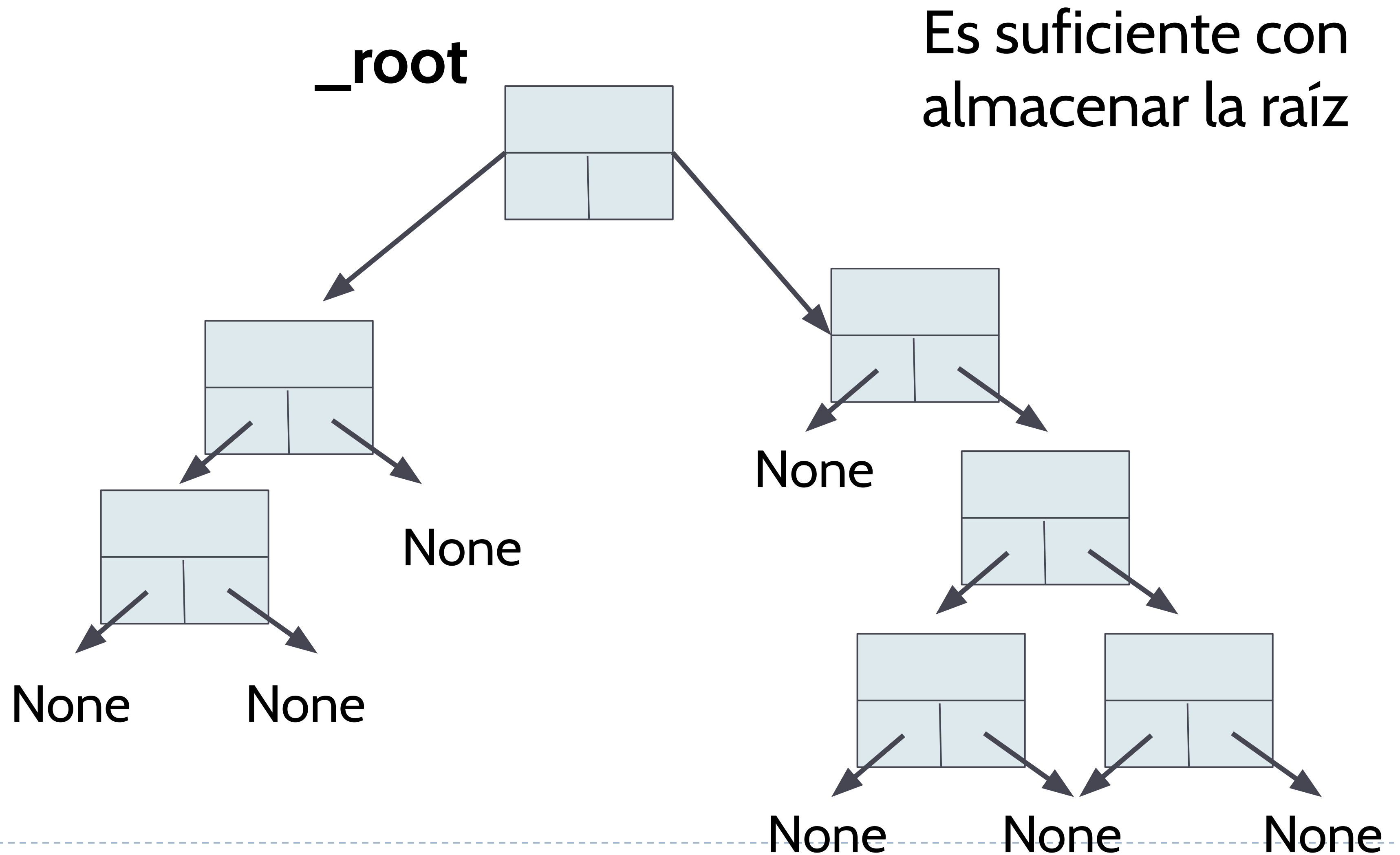


Implementación del nodo binario

```python
class BinaryNode:

    def __init__(self, elem: object,
                 node_left: 'BinaryNode' = None,
                 node_right: 'BinaryNode' = None) -> None:
        self.elem = elem
        self.left = node_left
        self.right = node_right
```

¿Cómo representar un árbol binario?



Implementación de un árbol binario

- El constructor creará un árbol vacío (es decir, su raíz será None).

```
class BinaryTree:
    def __init__(self) -> None:
        """creates an empty binary tree
        I only has an attribute: _root"""
        self._root = None
```

Comparando nodos

```python
class BinaryNode:
    def __init__(self, elem: object,
                 node_left: 'BinaryNode' = None,
                 node_right: 'BinaryNode' = None) -> None:
        self.elem = elem
        self.left = node_left
        self.right = node_right

    def __eq__(self, other: 'BinaryNode') -> bool:
        """checks if two nodes (subtrees) are equal o not"""
        return other is not None and self.elem == other.elem and \
               self.left == other.left and self.right == other.right
```

Comparando árboles

```python
class BinaryTree:
    def __init__(self) -> None:
        """creates an empty binary tree
        I only has an attribute: _root"""
        self._root = None

    def __eq__(self, other_tree: 'BinaryTree') -> bool:
        """checks if two binary trees are equal o not"""
        return other_tree is not None and self._root == other_tree._root
```

Implementación método tamaño

```
def size(self) -> int:
    """Returns the number of nodes"""
    return self._size(self._root)

def _size(self, node: BinaryNode) -> int:
    """return the size of the subtree from node"""
    ...
```

_size es un método recursiva

Implementación método tamaño

```python
def size(self) -> int:
    """Returns the number of nodes"""
    return self._size(self._root)

def _size(self, node: BinaryNode) -> int:
    """return the size of the subtree from node"""
    if node is None:
        return 0
    else:
        return 1 + self._size(node.left) + self._size(node.right)
```

Implementación método altura

```python
def height(self) -> int:
    """Returns the height of the tree"""
    return self._height(self._root)

def _height(self, node: BinaryNode) -> int:
    """return the height of node"""
    ...
```

_height es recursiva

Implementación método altura

```python
def height(self) -> int:
    """Returns the height of the tree"""
    return self._height(self._root)

def _height(self, node: BinaryNode) -> int:
    """return the height of node"""
    if node is None:
        return -1
    else:
        return 1 + max(self._height(node.left), self._height(node.right))
```

Implementación de preorder

```python
def preorder(self) -> None:
    """prints the preorder (root, left, right) traversal of the tree"""
    print('Preorder traversal: ', end=' ')  # end=' ' avoid the newline
    self._preorder(self._root)
    print()

def _preorder(self, node: BinaryNode) -> None:
    """prints the preorder (root, left, right) traversal of the subtree
    than hangs from node"""
    ...
```

_preorder es recursiva

Implementación de preorder

```python
def preorder(self) -> None:
    """prints the preorder (root, left, right) traversal of the tree"""
    print('Preorder traversal: ', end=' ')  # end=' ' avoid the newline
    self._preorder(self._root)
    print()

def _preorder(self, node: BinaryNode) -> None:
    """prints the preorder (root, left, right) traversal of the subtree
    than hangs from node"""

    if node is not None:
        print(node.elem, end=' ')  # end=' ' avoid new line
        self._preorder(node.left)
        self._preorder(node.right)
```

Ejercicio

- Implementa los recorridos postorder e indorder. Solución.

Ejercicio

Implementa una nueva versión de estas funciones para que en lugar de imprimir los elementos de los nodos, los devuelva en una lista (puedes usar una lista de python).
Solución

Recorrido - preorder usando una lista

```python
def preorder_list(self) -> list:
    """returns a list with the preorder traversal"""
    # self.draw()
    result = []
    self._preorder_list(self._root, result)
    return result

def _preorder_list(self, node: BinaryNode, pre_list: list) -> None:
    """populates pre_list with the preorder traversal of the subtree node"""
    if node is not None:
        pre_list.append(node.elem)
        self._preorder_list(node.left, pre_list)
        self._preorder_list(node.right, pre_list)
```

Recorrido - post-order usando una lista

```python
def postorder_list(self) -> list:
    """returns a list with the postorder traversal of the tree"""
    # self.draw()
    result = []
    self._postorder_list(self._root, result)
    return result

def _postorder_list(self, node: BinaryNode, post_list: list) -> None:
    """populates post_list with the postorder traversal of the subtree node"""
    if node is not None:
        self._postorder_list(node.left, post_list)
        self._postorder_list(node.right, post_list)
        post_list.append(node.elem)
```

Recorrido - in-order usando una lista

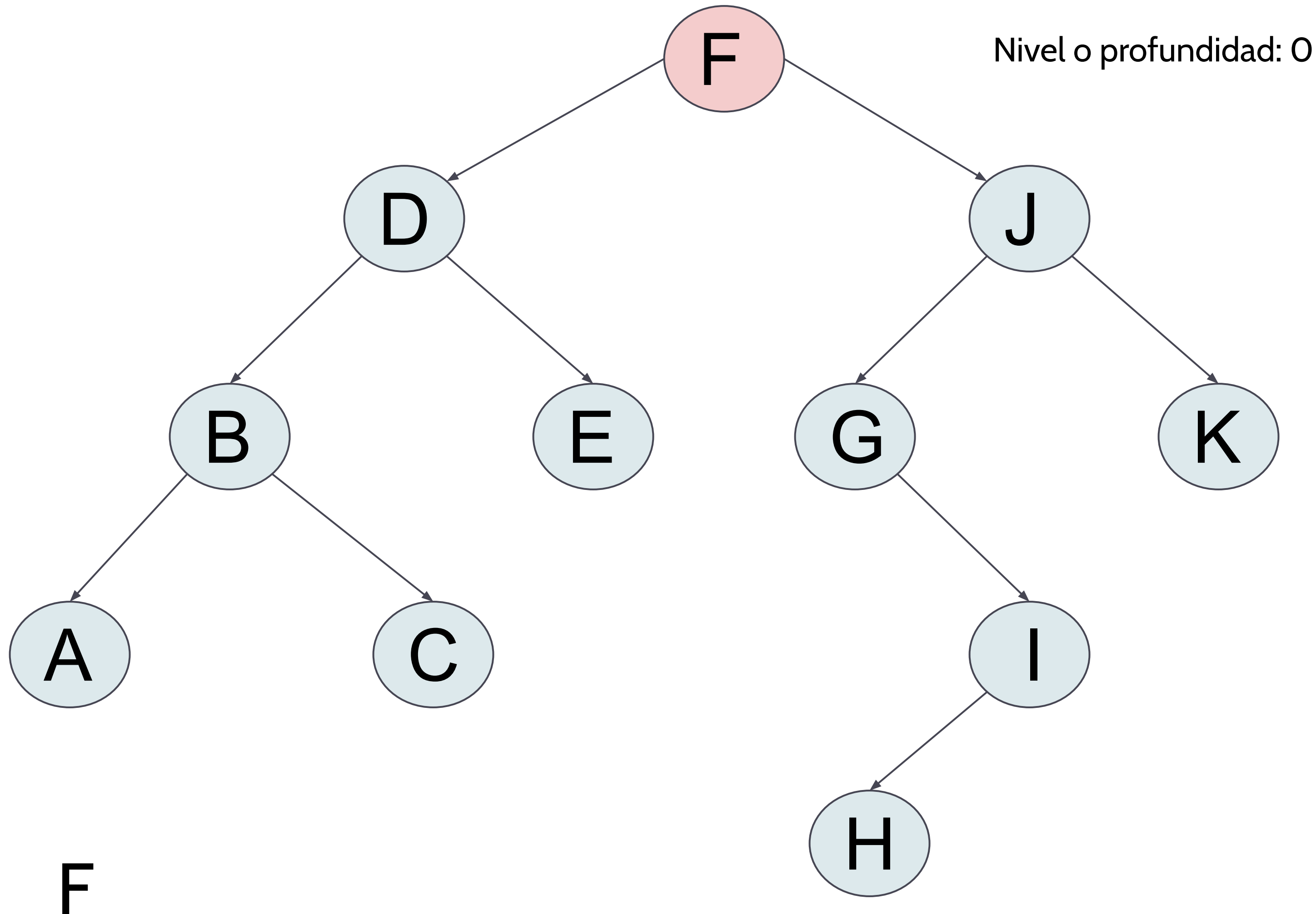
```python
def inorder_list(self) -> list:
    """returns a list with the inorder traversal of the tree"""
    # self.draw()
    result = []
    self._inorder_list(self._root, result)
    return result

def _inorder_list(self, node: BinaryNode, in_list: list) -> None:
    """populates in_list with the inorder traversal of the subtree node"""
    if node is not None:
        self._postorder_list(node.left, in_list)
        in_list.append(node.elem)
        self._postorder_list(node.right, in_list)
```

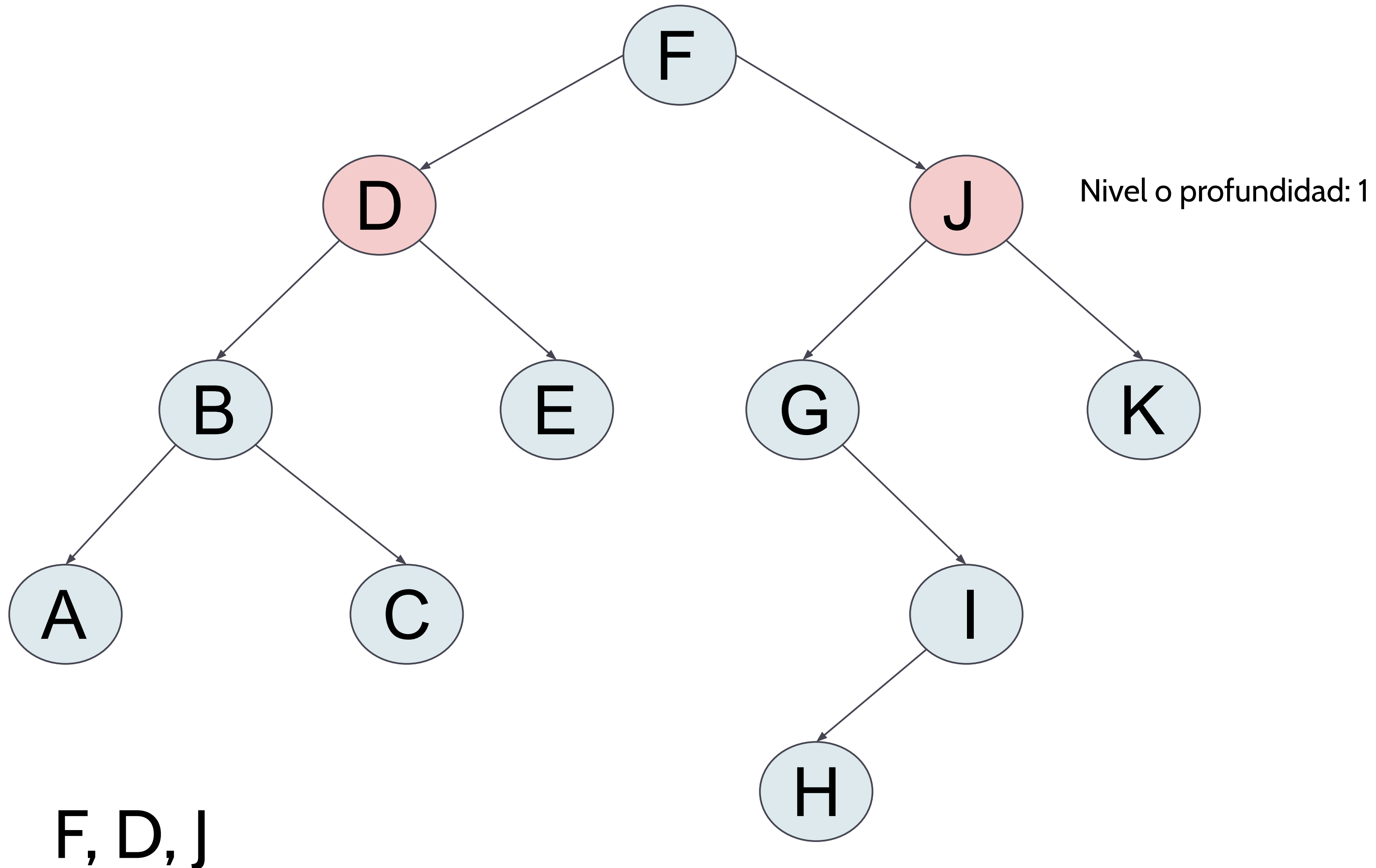
Recorrido por niveles

- Los nodos son visitados por niveles. Así, los nodos son visitados en el mismo nivel, de izquierda a derecha, y de forma descendente.

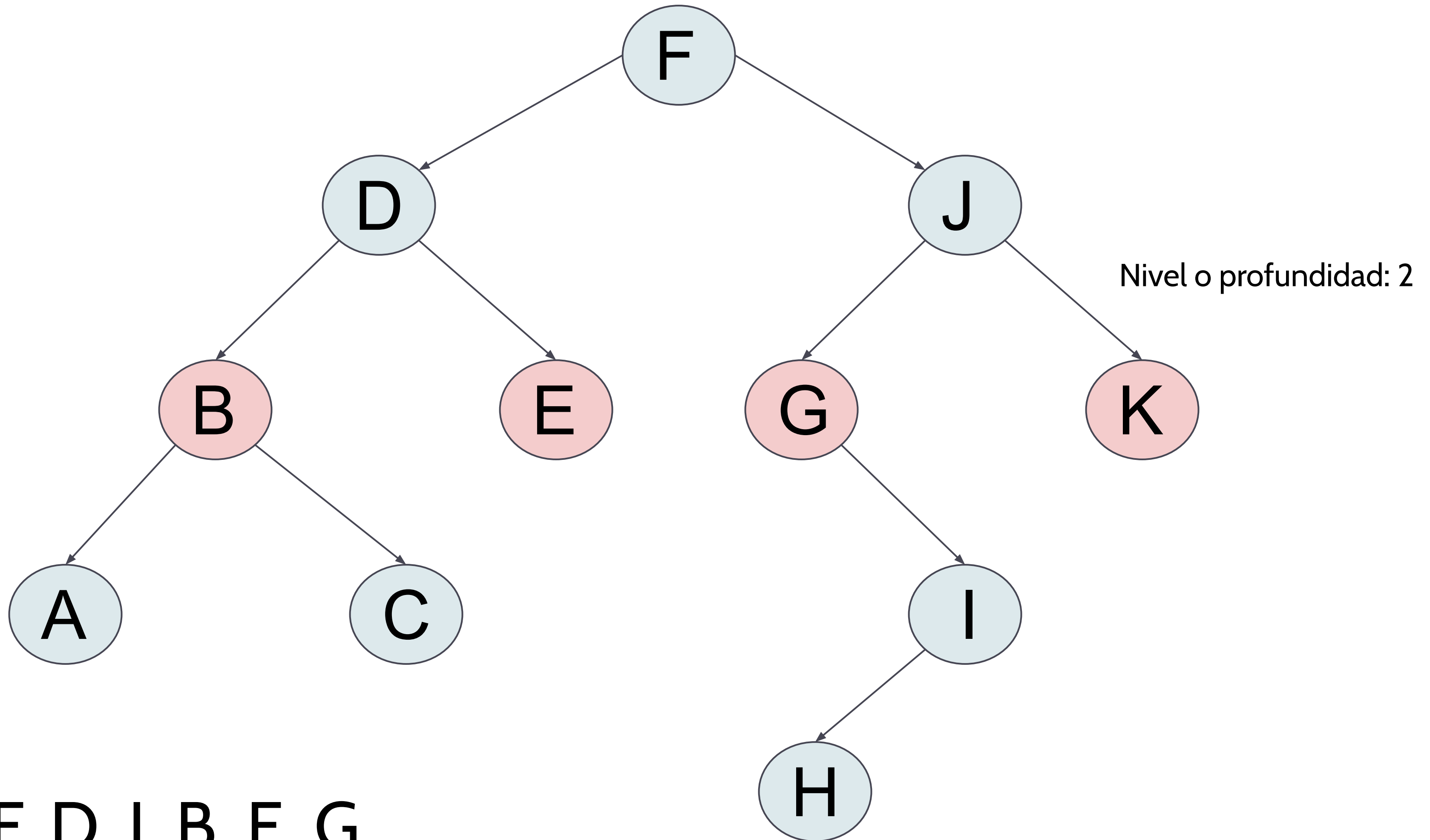
Recorrido por niveles



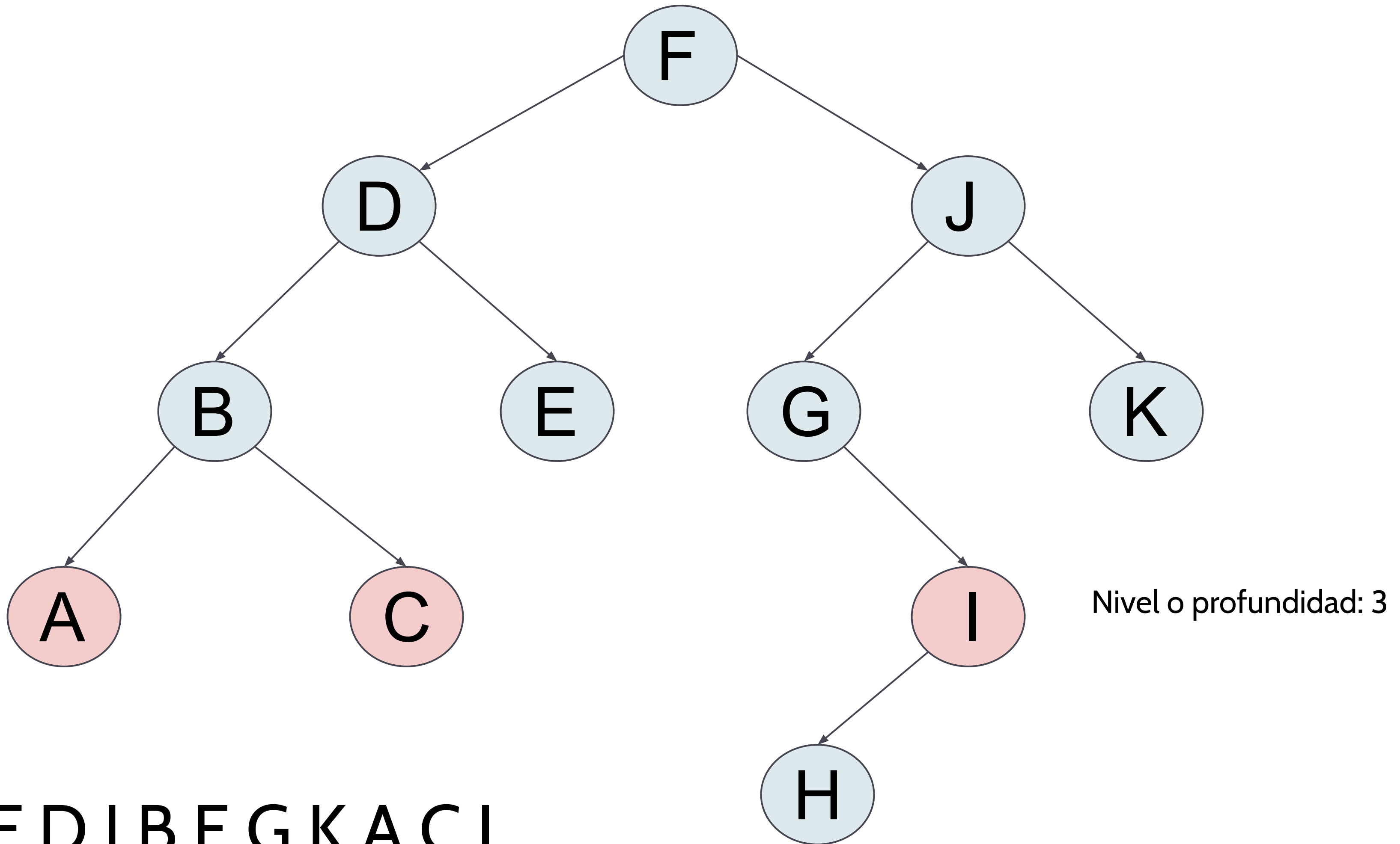
Recorrido por niveles



Recorrido por niveles

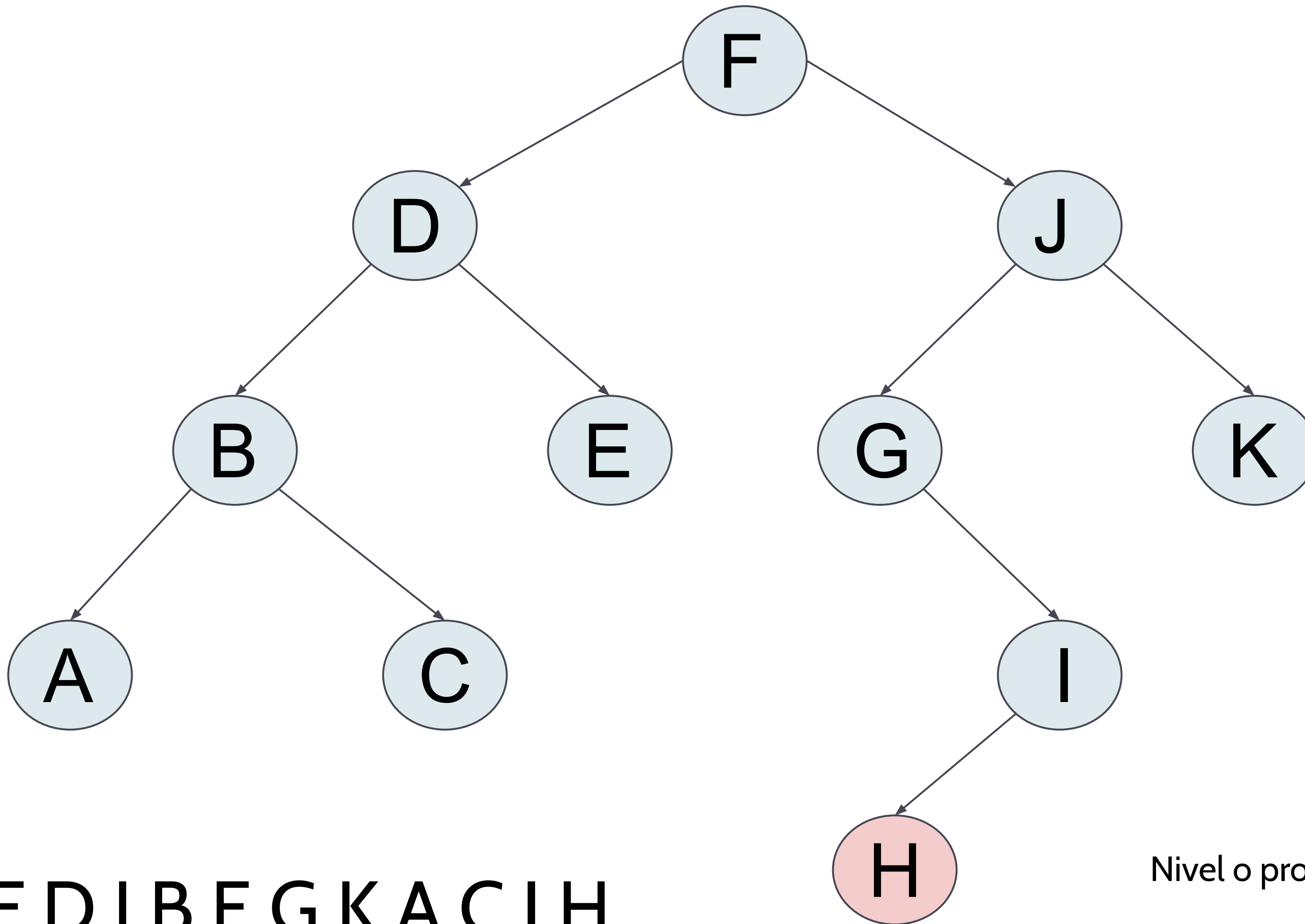


Recorrido por niveles



F D J B E G K A C I

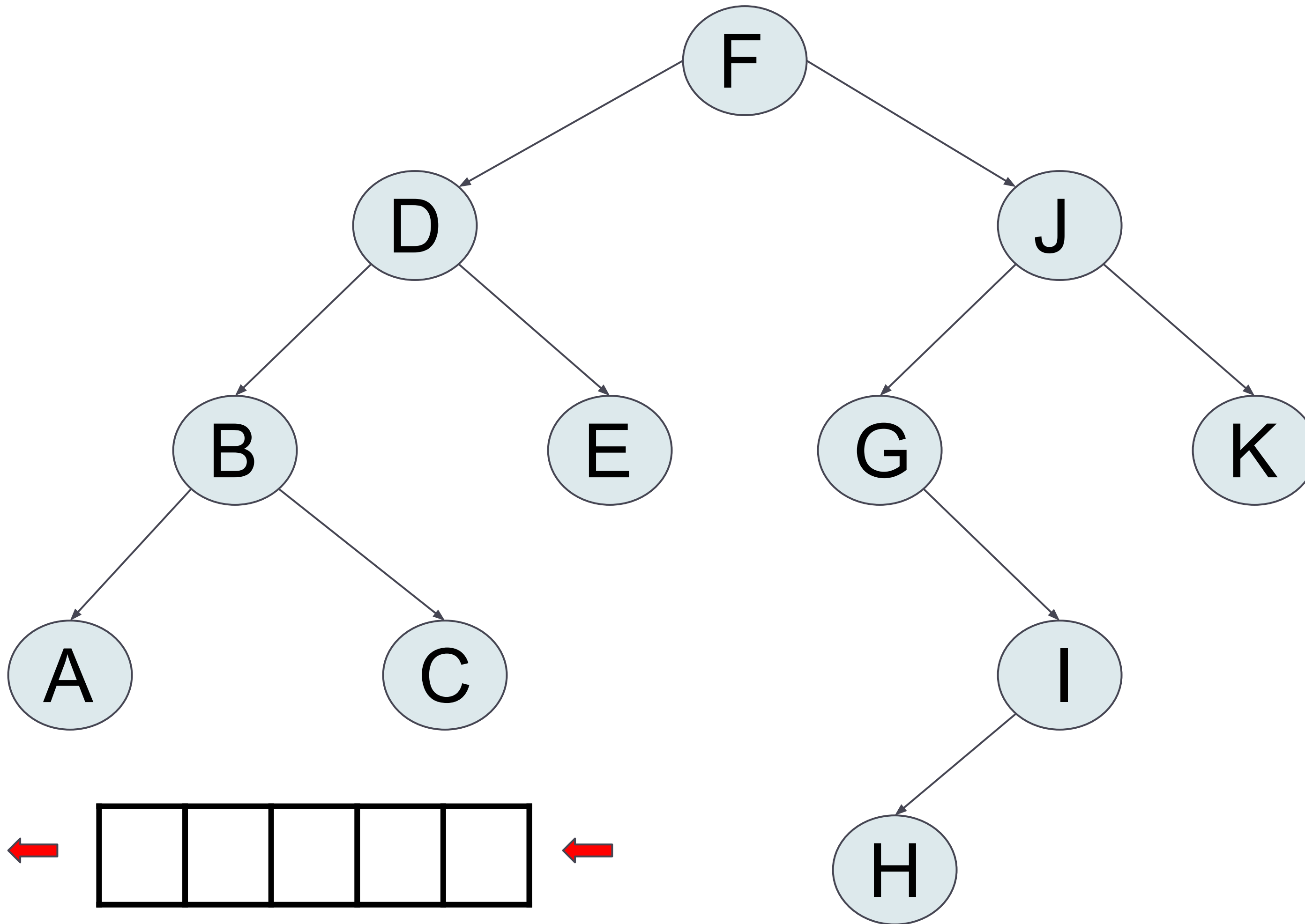
Recorrido por niveles



F D J B E G K A C I H

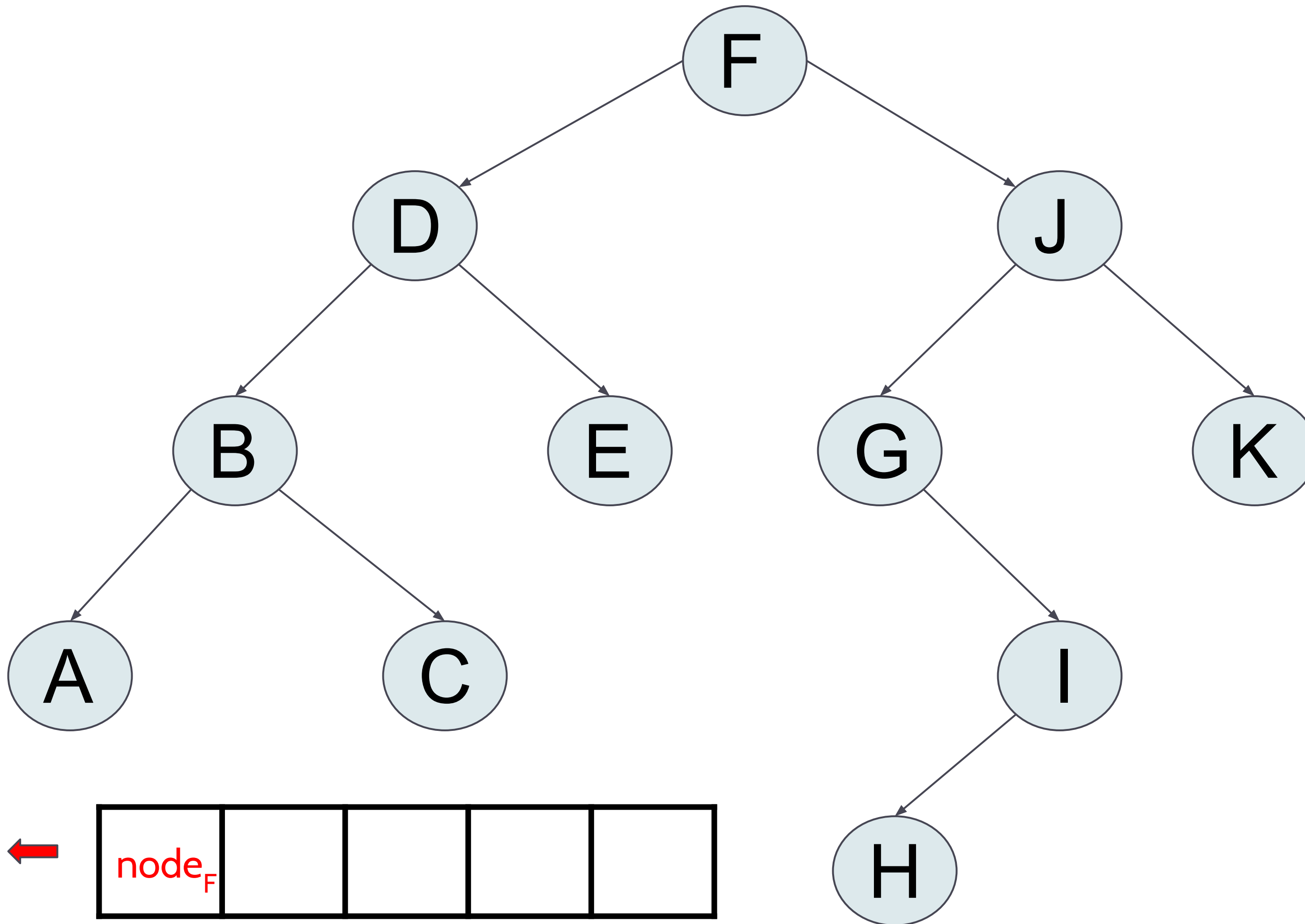
Nivel o profundidad: 4

Recorrido por niveles



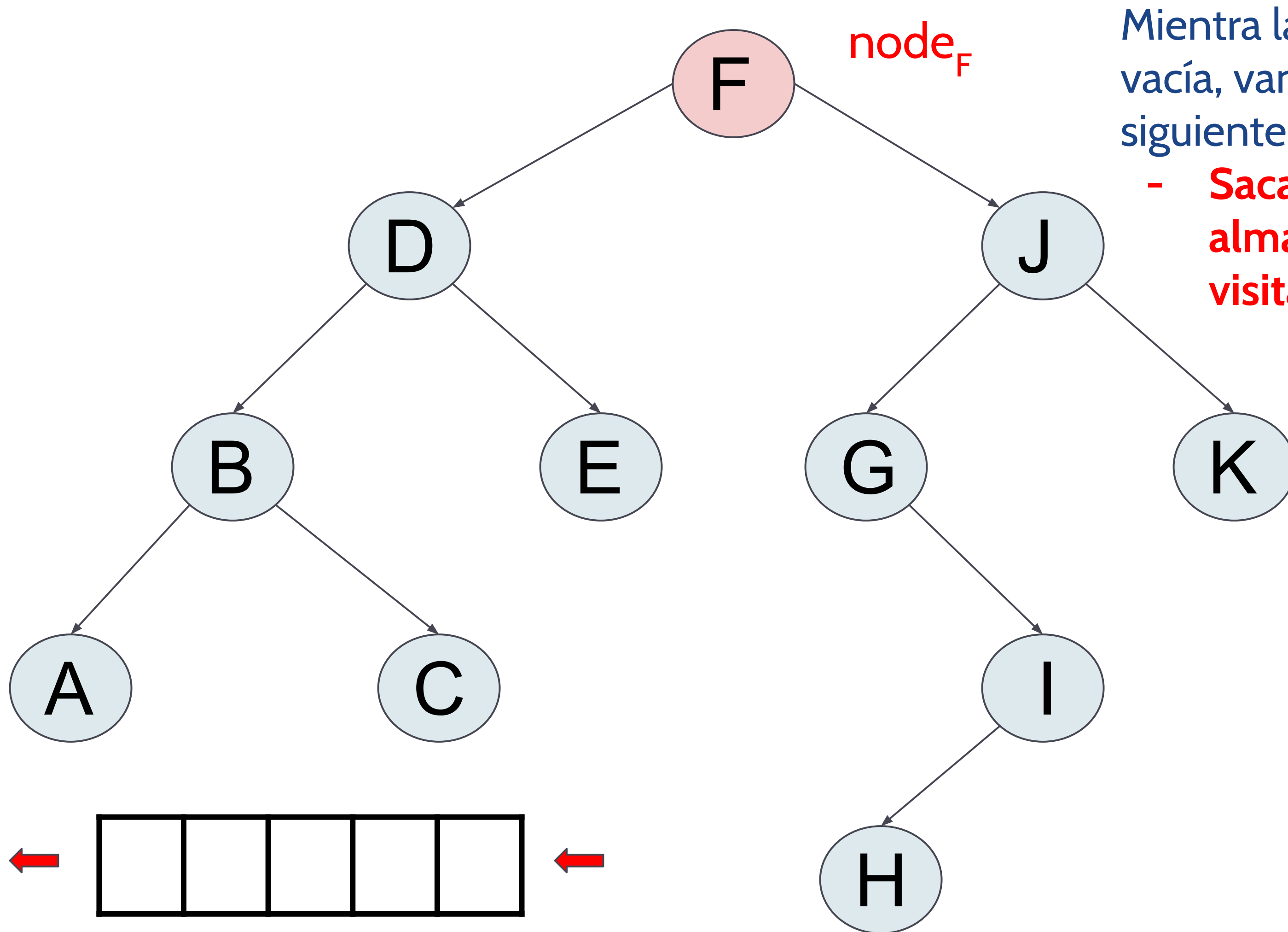
Cola o lista para almacenar nodos

Recorrido por niveles



Guardamos el nodo raíz

Recorrido por niveles

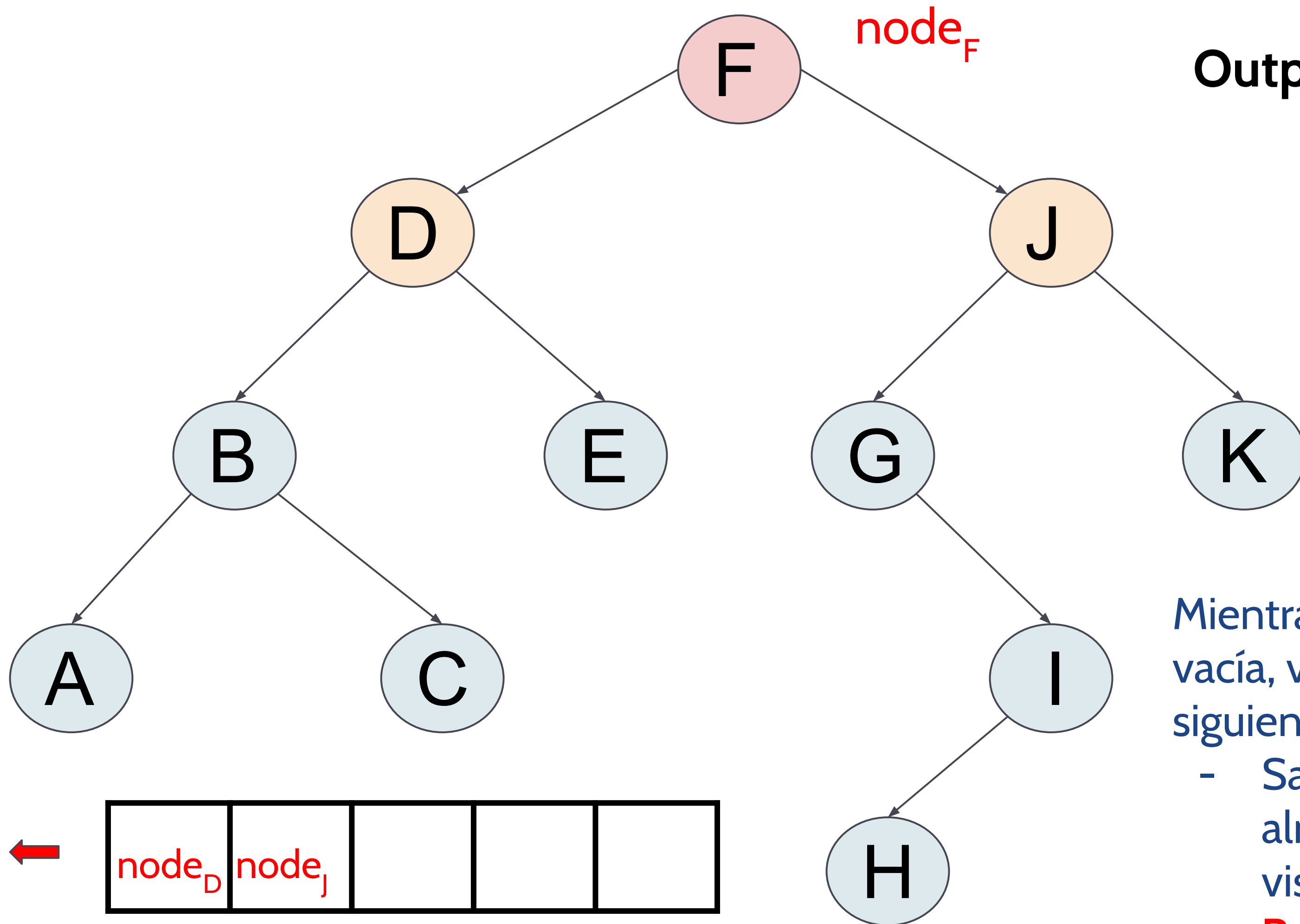


Mientra la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**

Output: F

Recorrido por niveles

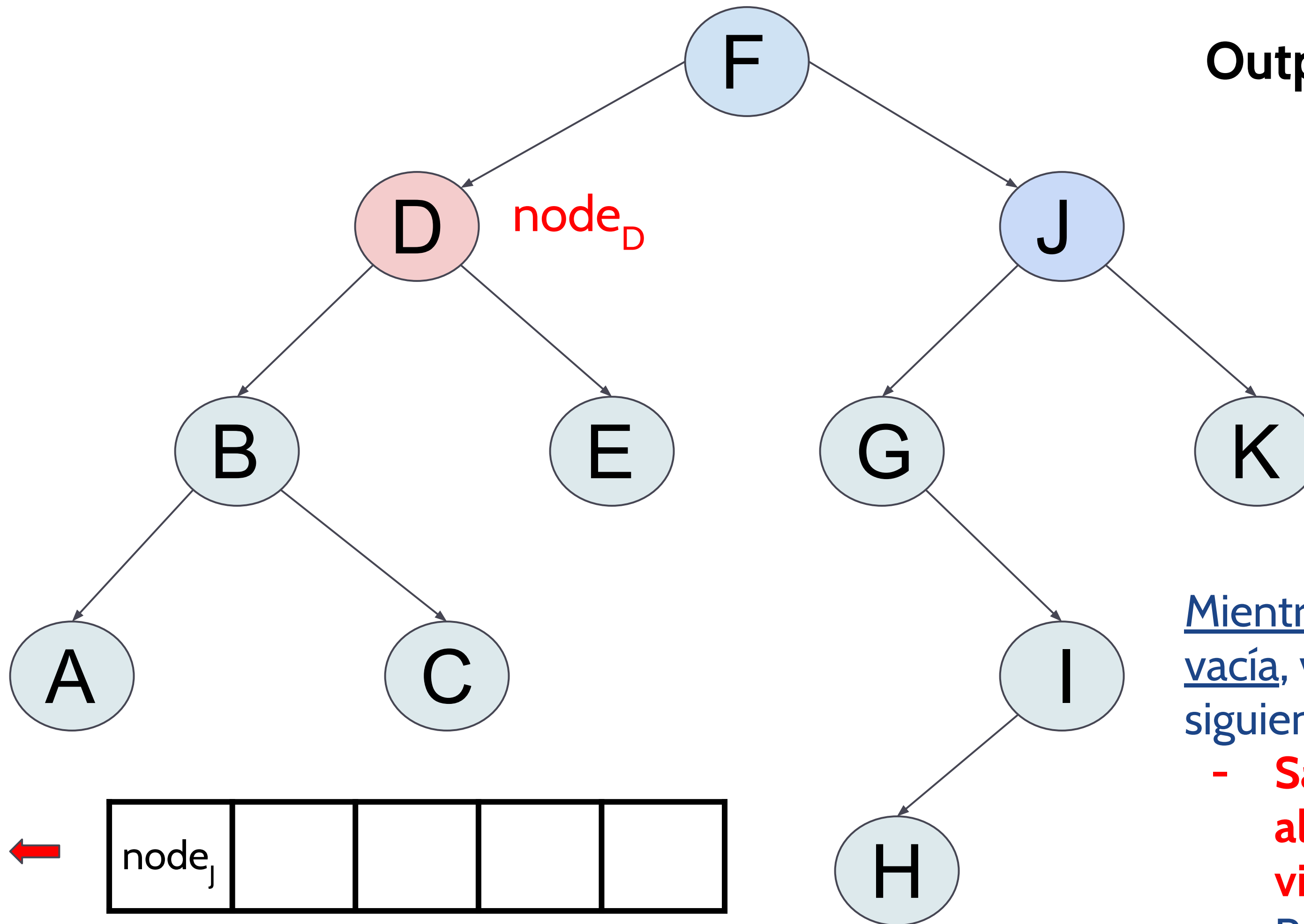


Output: F

Mientra la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

Recorrido por niveles

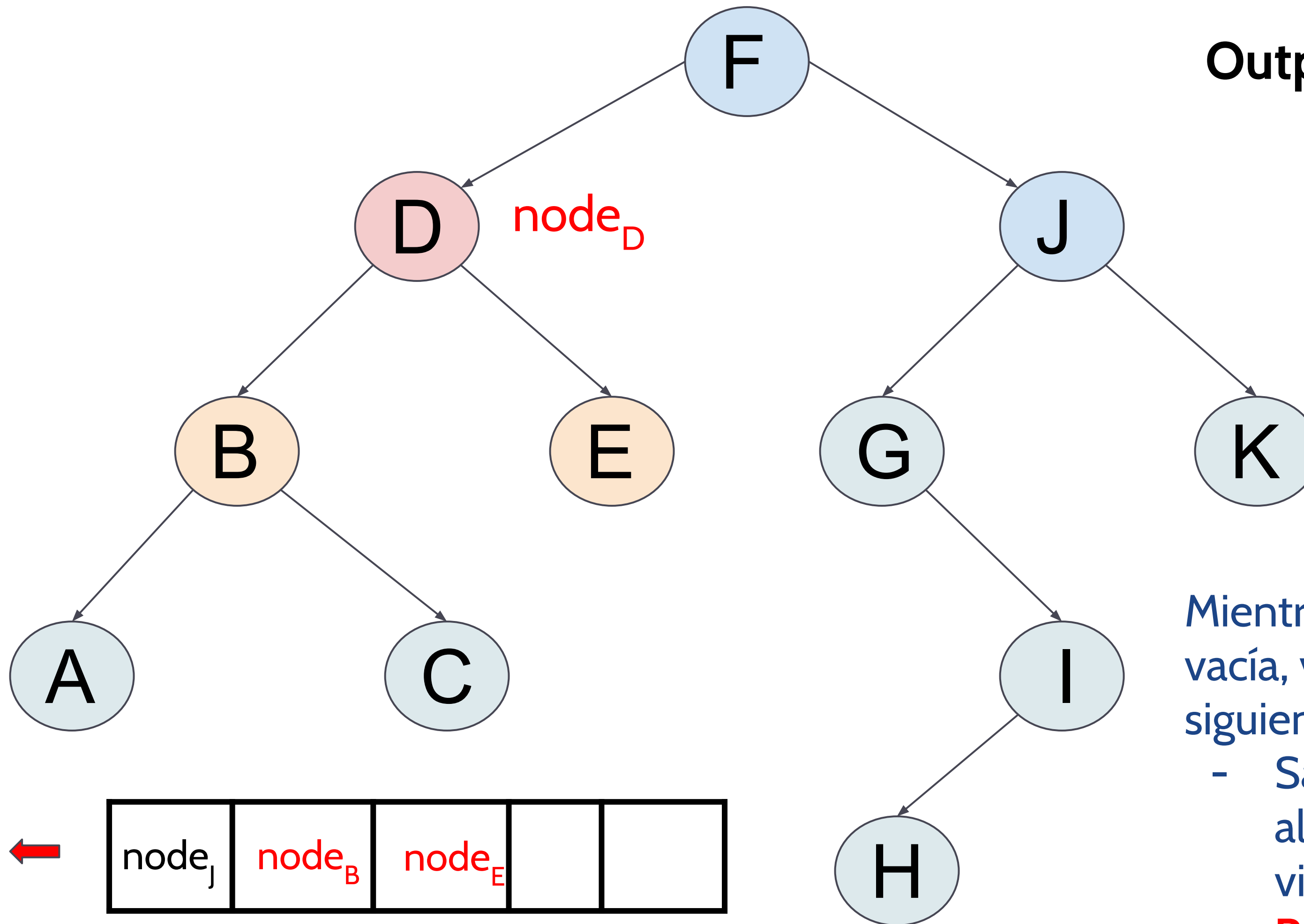


Output: F, D

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

Recorrido por niveles

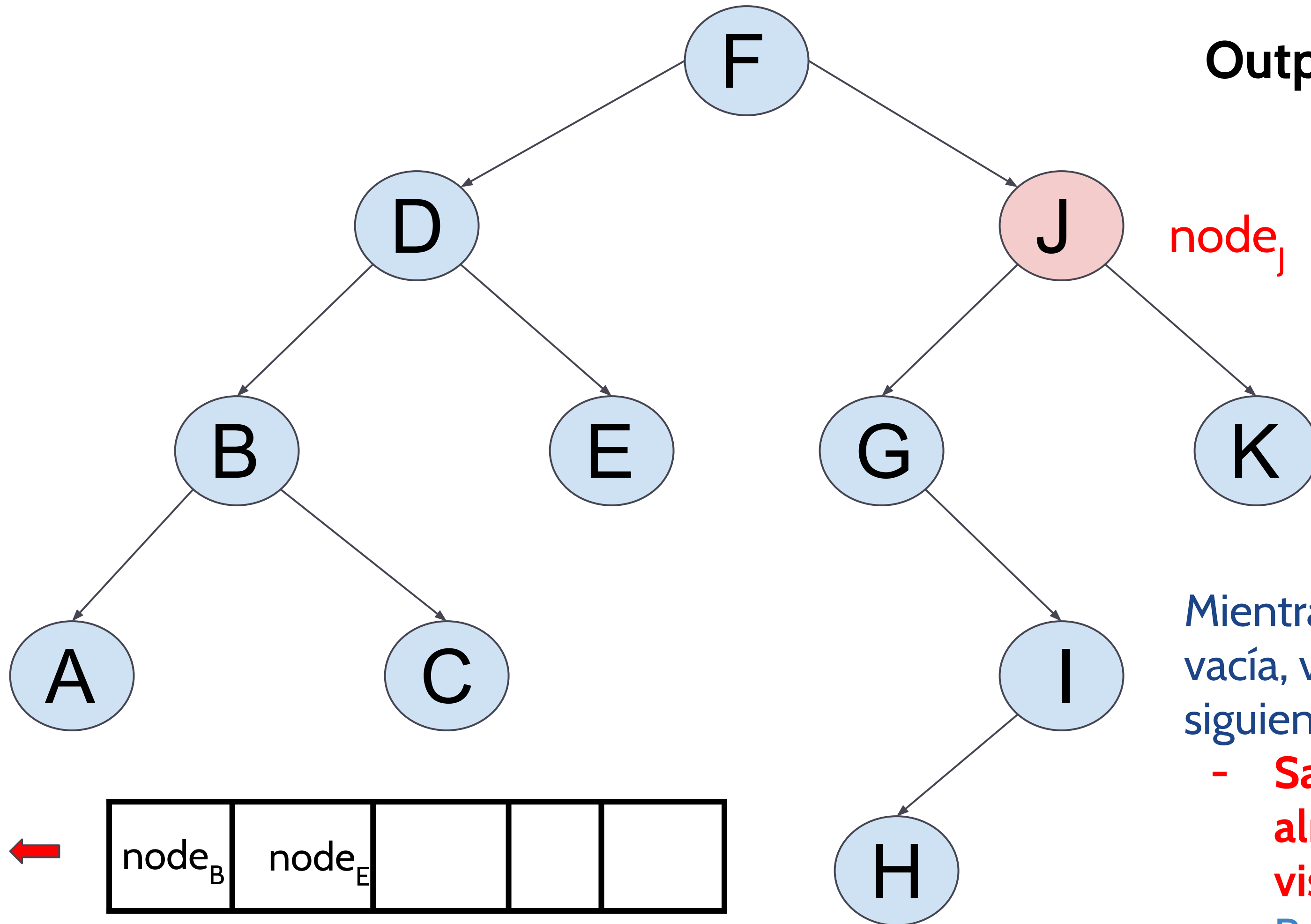


Output: F, D

Mientra la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

Recorrido por niveles

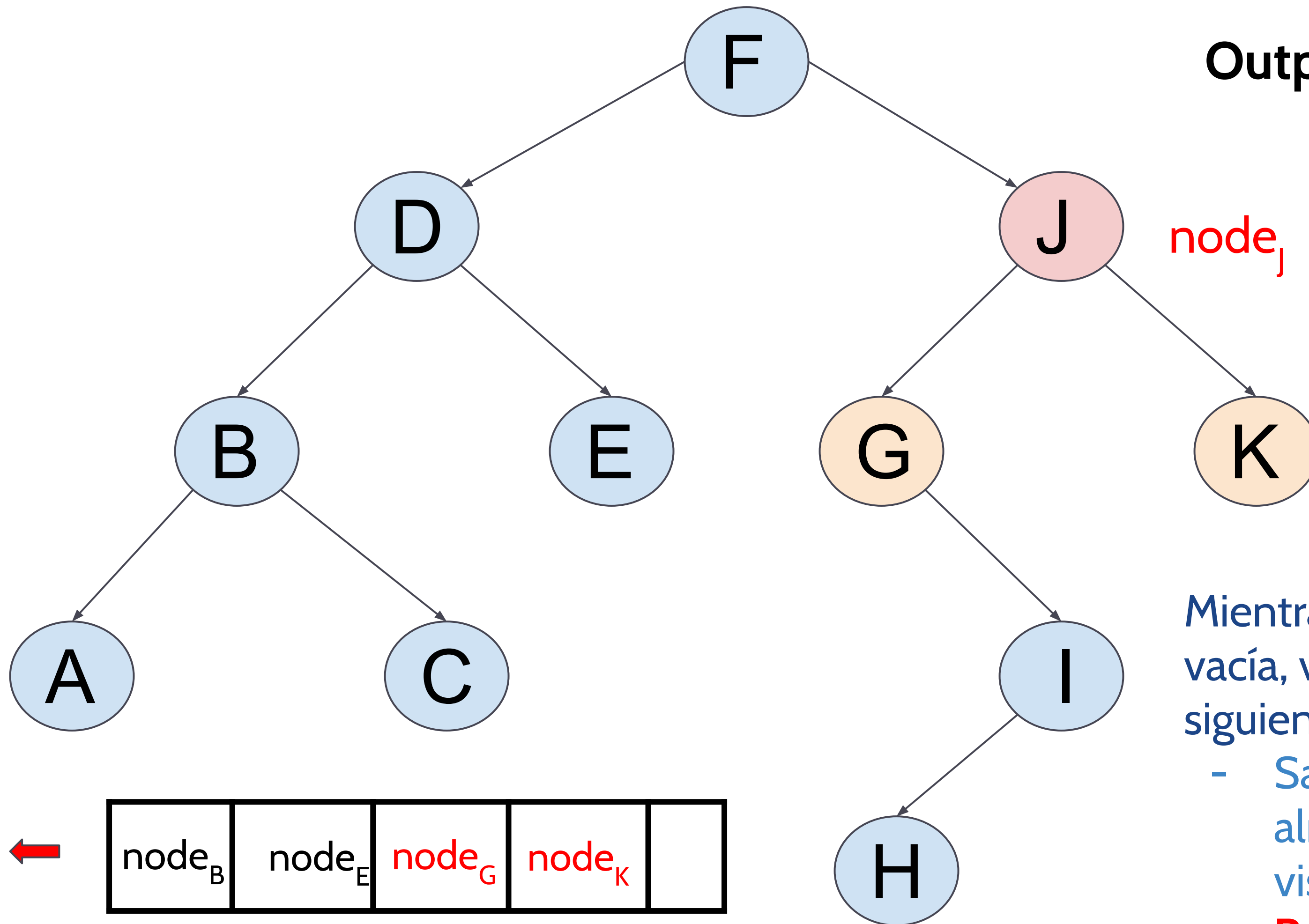


Output: F, D, J

Mientra la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

Recorrido por niveles

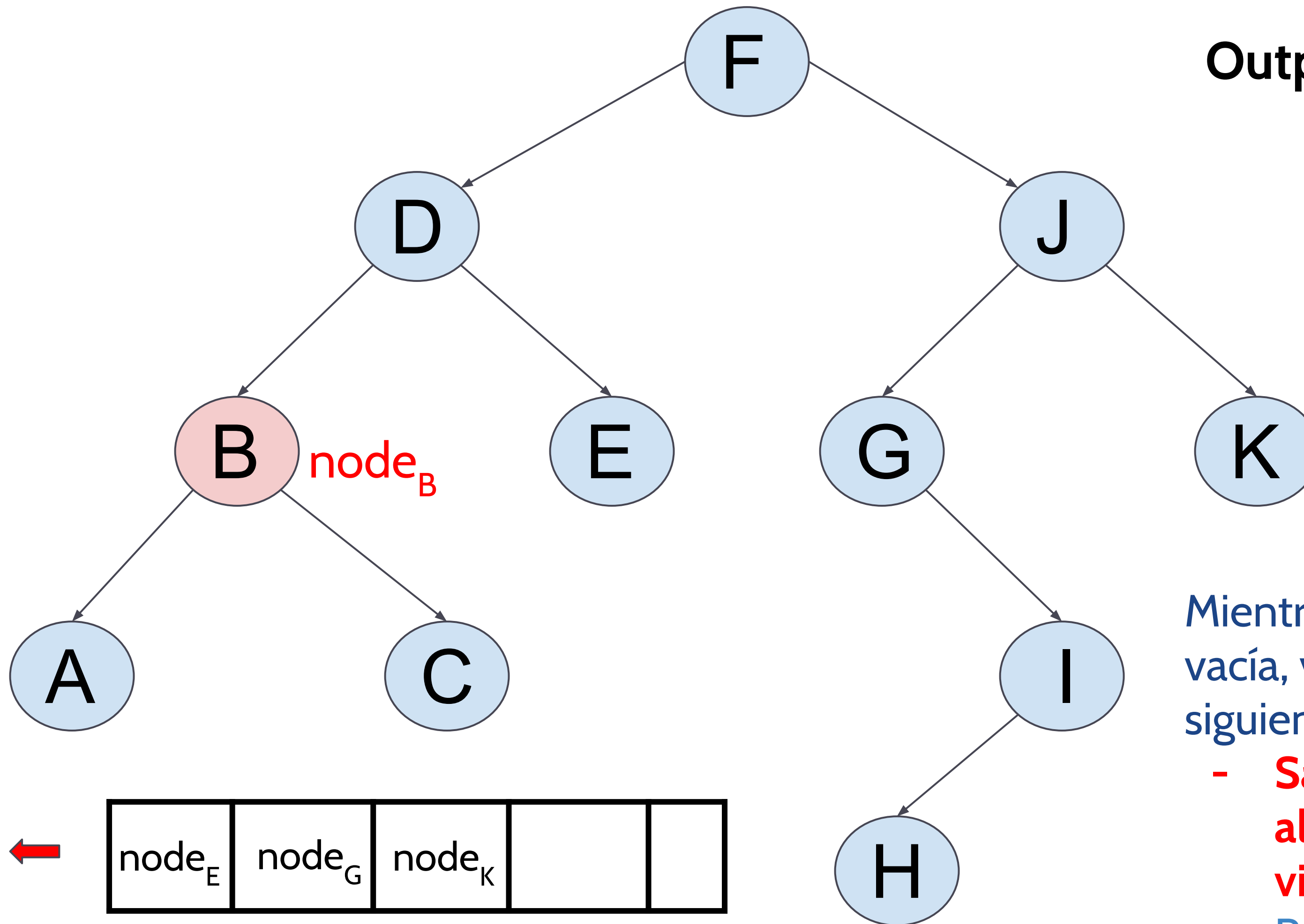


Output: F, D, J

Mientra la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

Recorrido por niveles

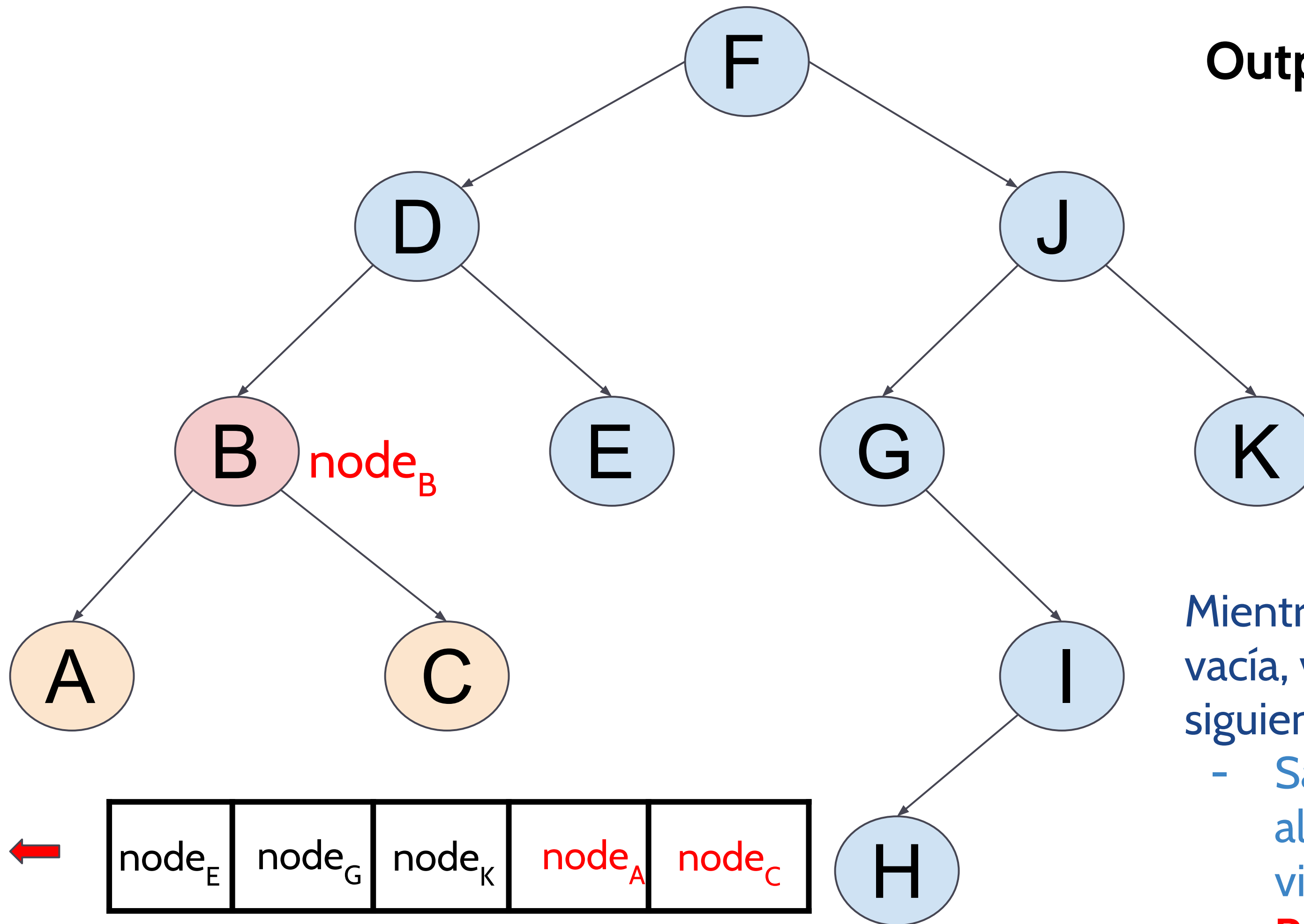


Output: F, D, J, B

Mientra la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

Recorrido por niveles

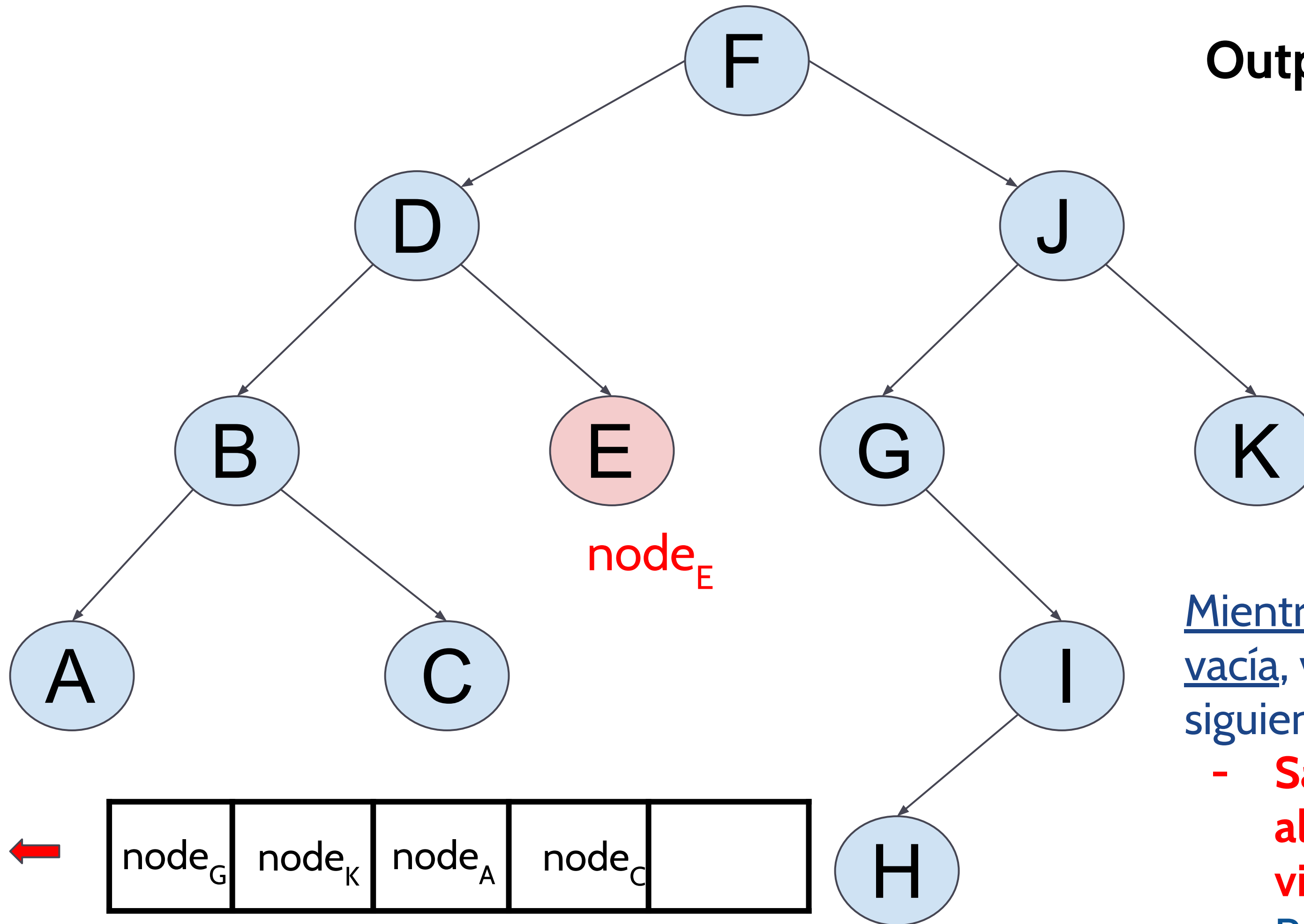


Output: F, D, J, B

Mientra la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

Recorrido por niveles

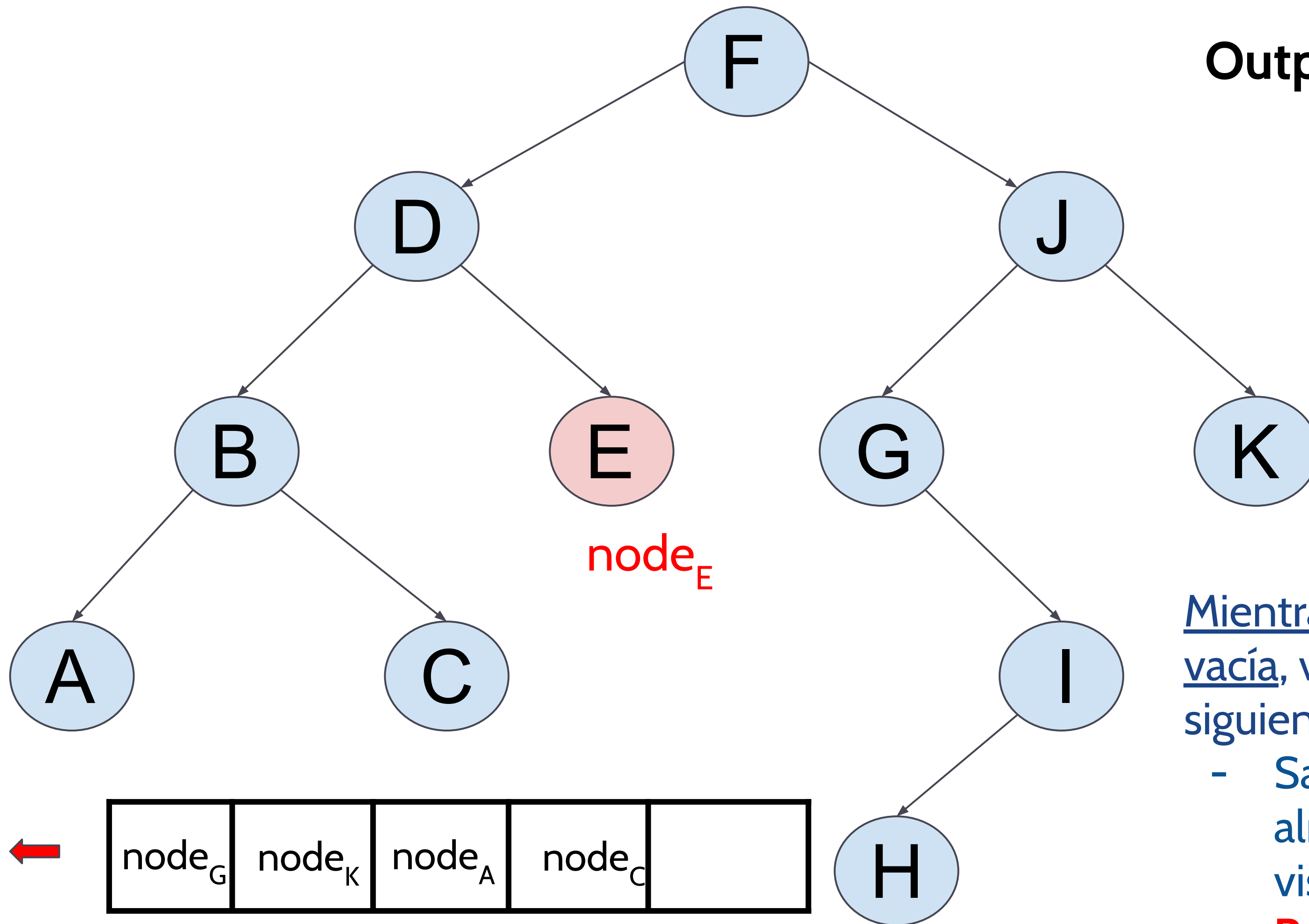


Output: F, D, J, B, E

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

Recorrido por niveles



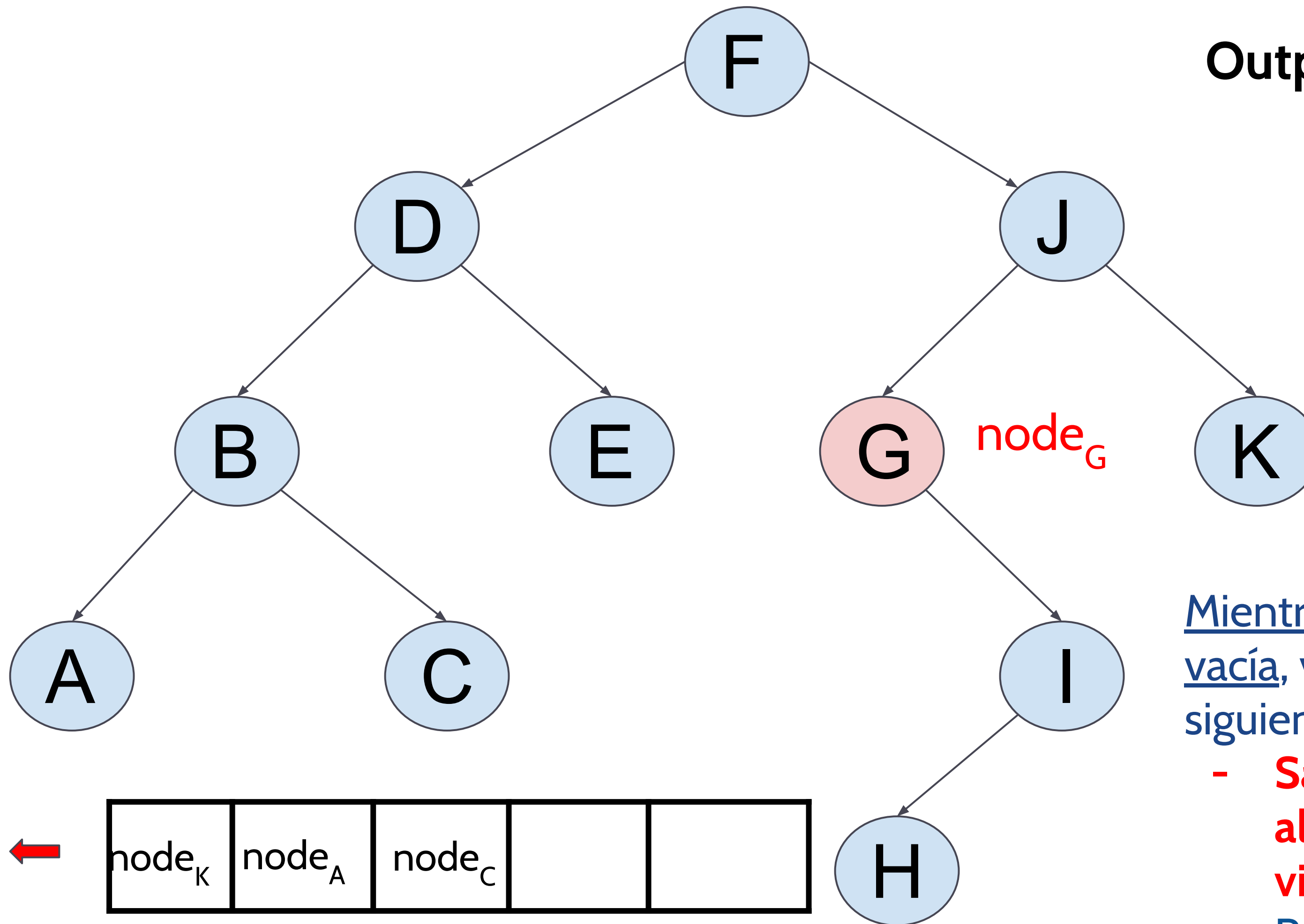
Output: F, D, J, B, E

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

node$_E$ no tiene hijos, no añadimos nada a la lista

Recorrido por niveles

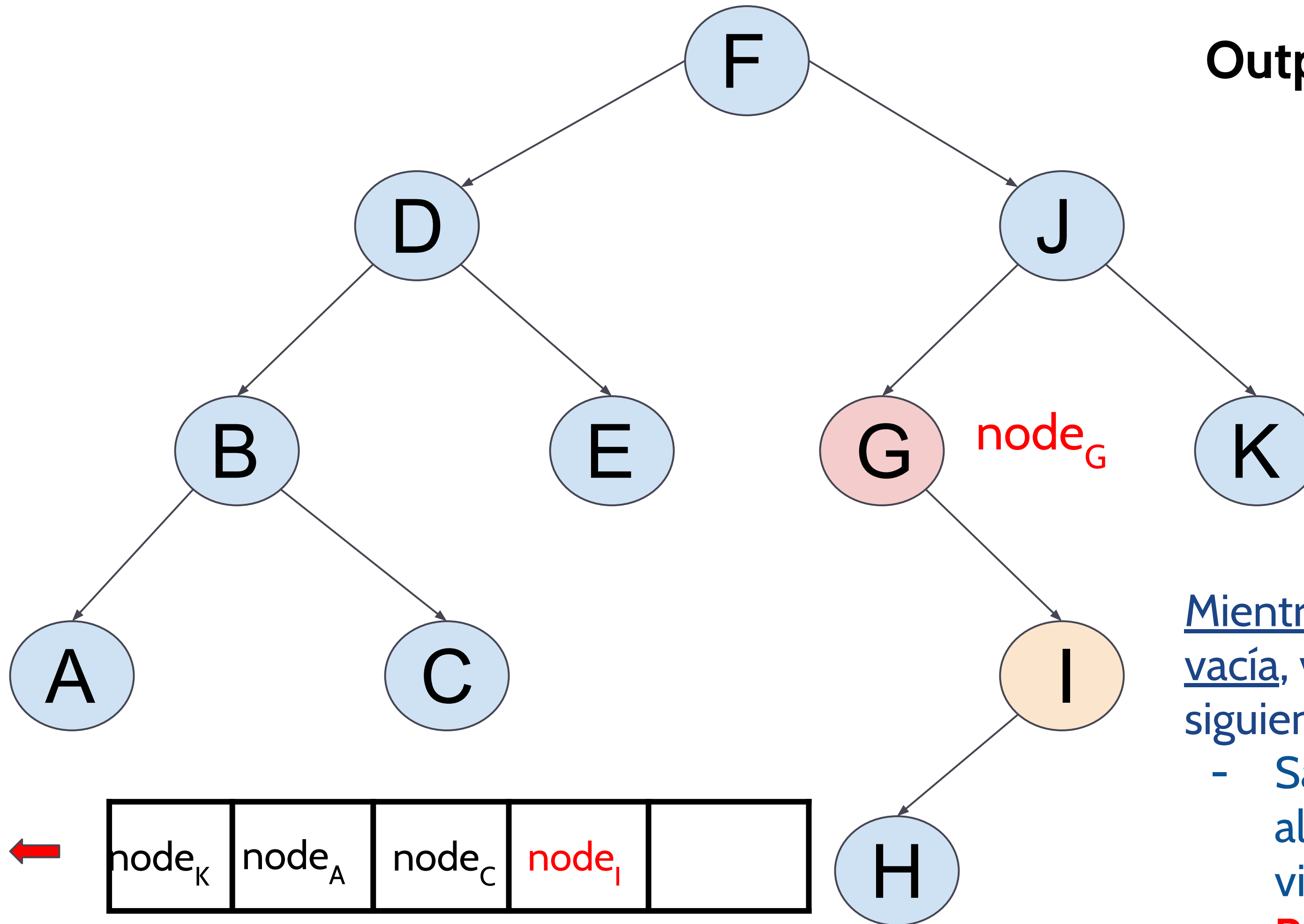


Output: F, D, J, B, E, G

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

Recorrido por niveles

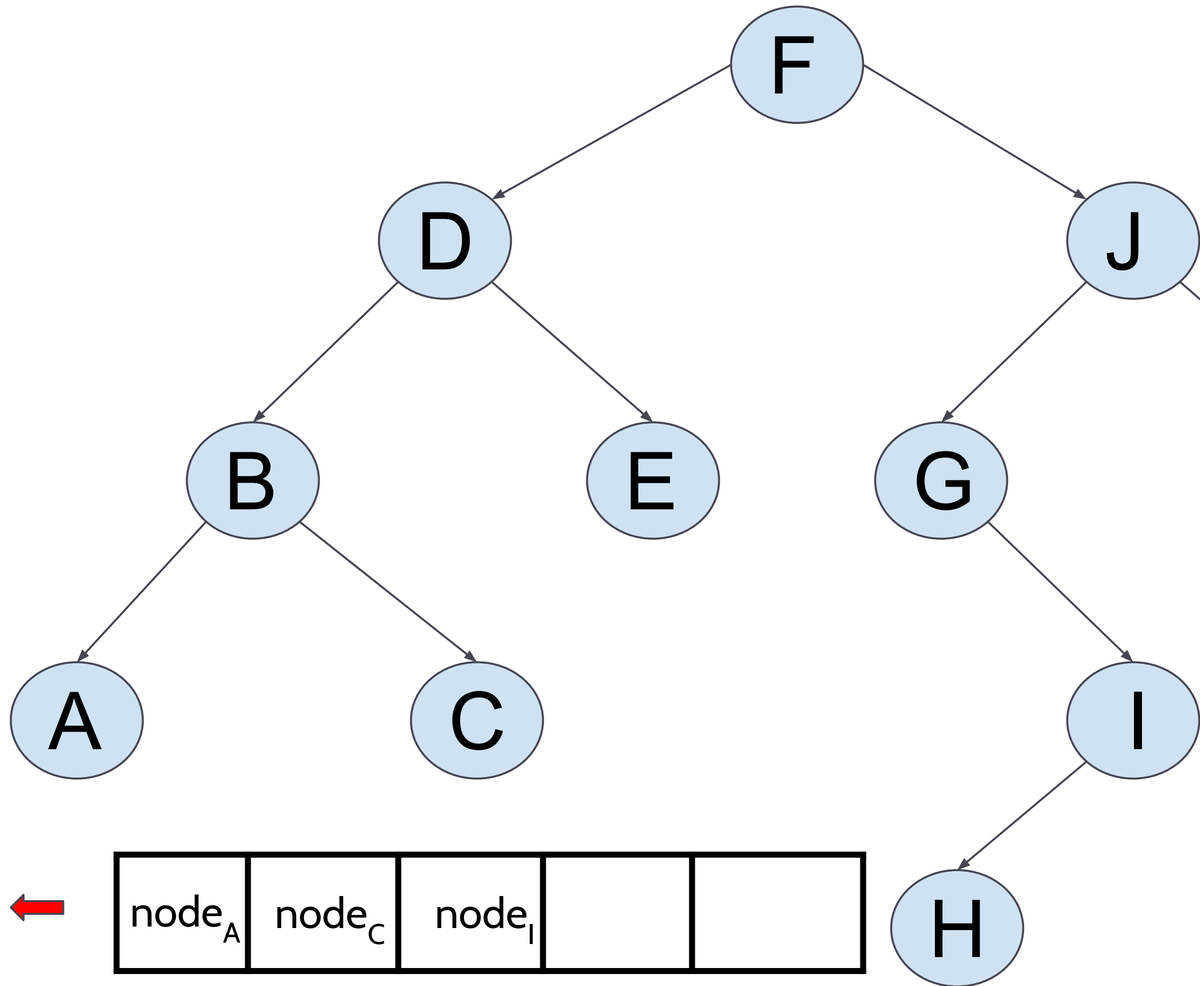


Output: F, D, J, B, E, G

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

Recorrido por niveles



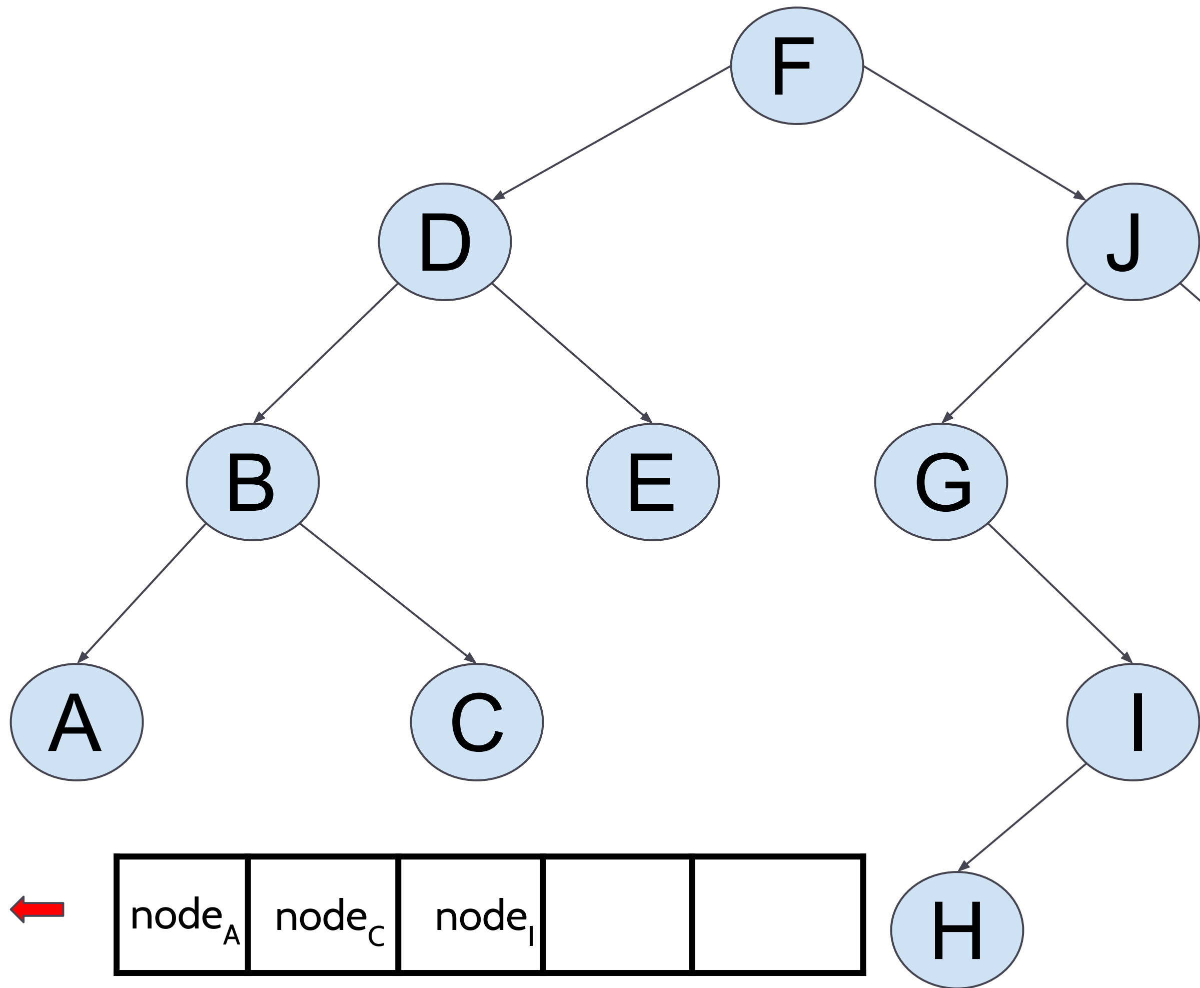
Output: F, D, J, B, E, G, K

node$_k$

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

Recorrido por niveles



Output: F, D, J, B, E, G, K

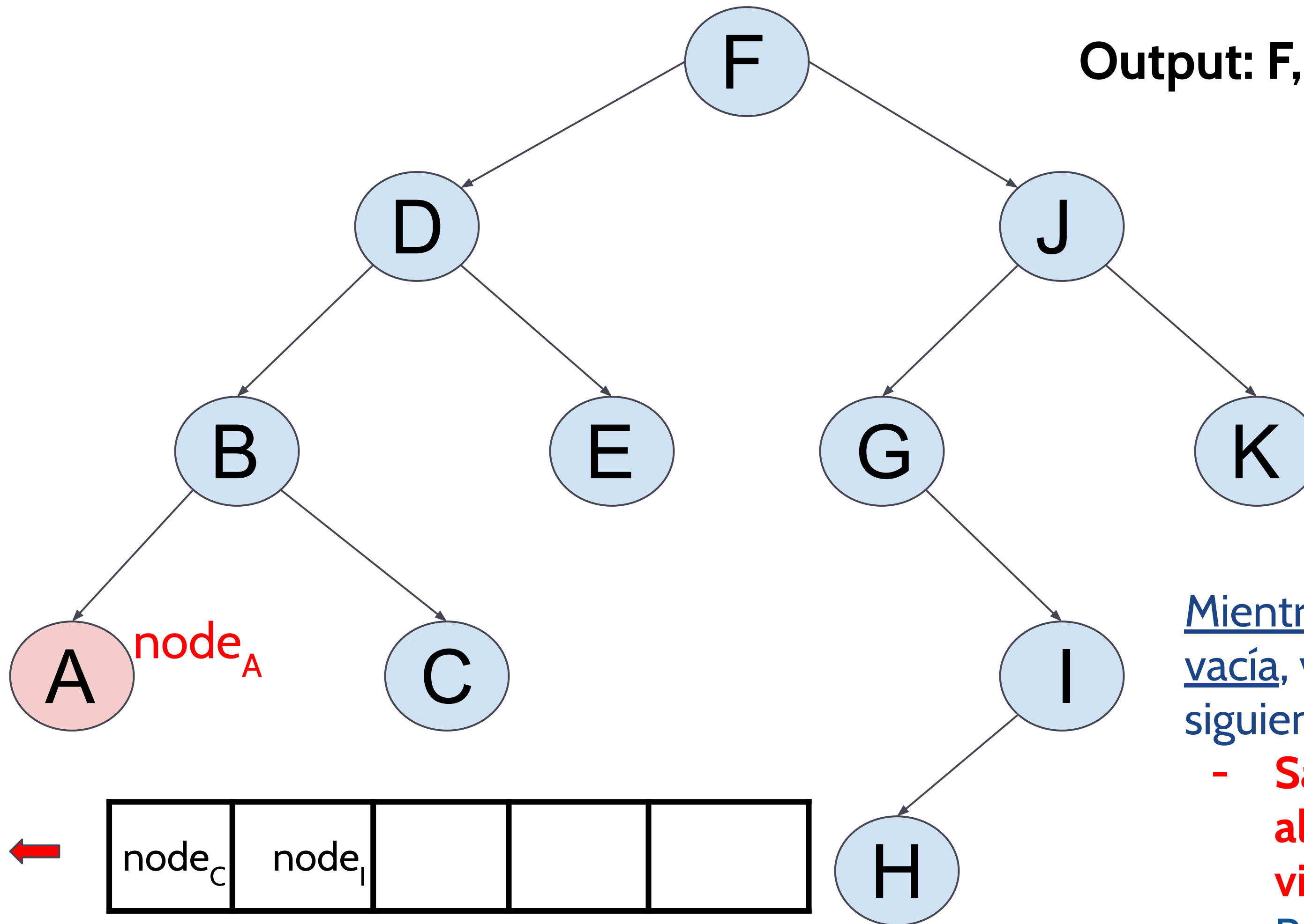
node$_k$

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

node$_k$ no tiene hijos, no añadimos nada a la lista

Recorrido por niveles

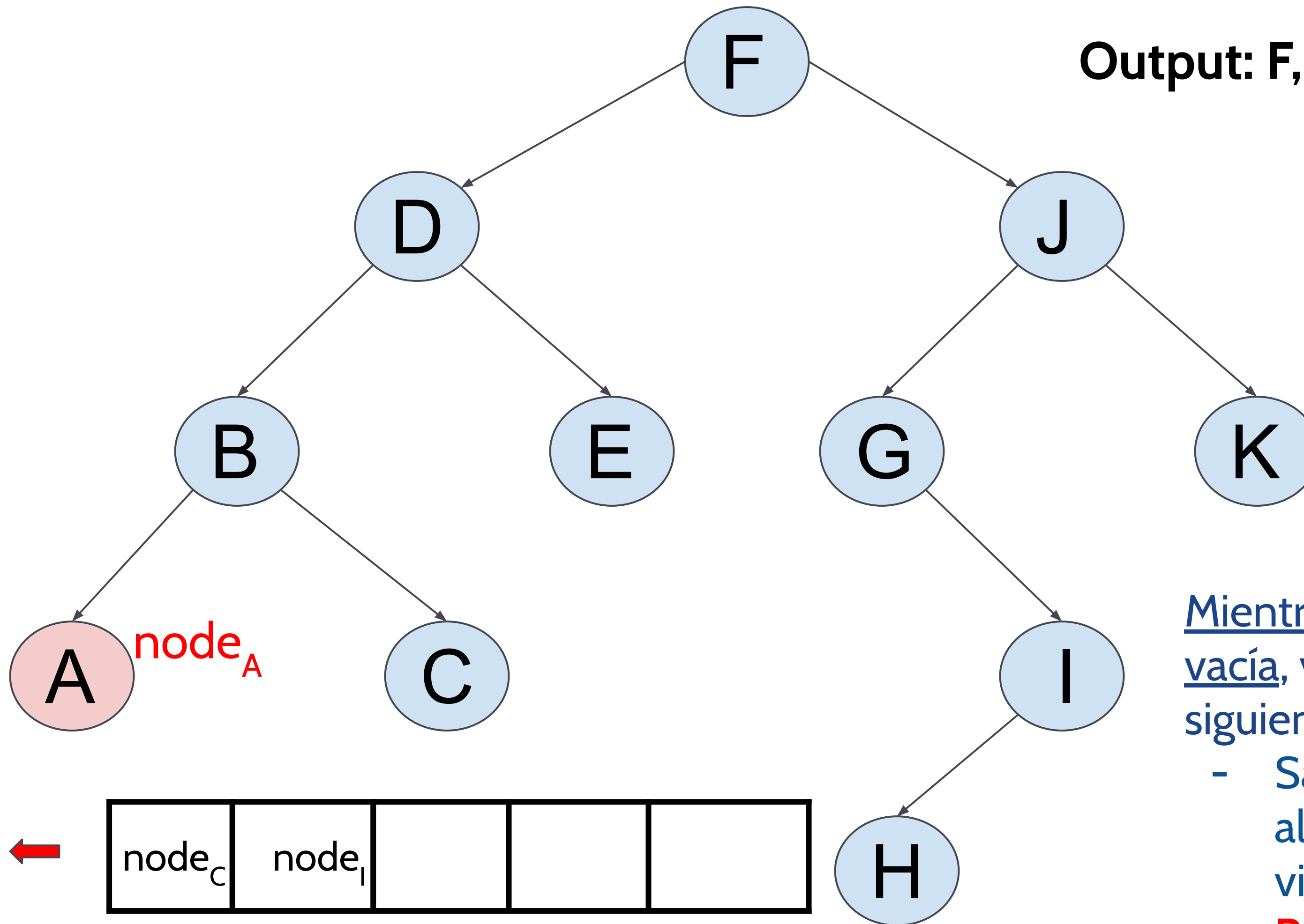


Output: F, D, J, B, E, G, K, A

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

Recorrido por niveles



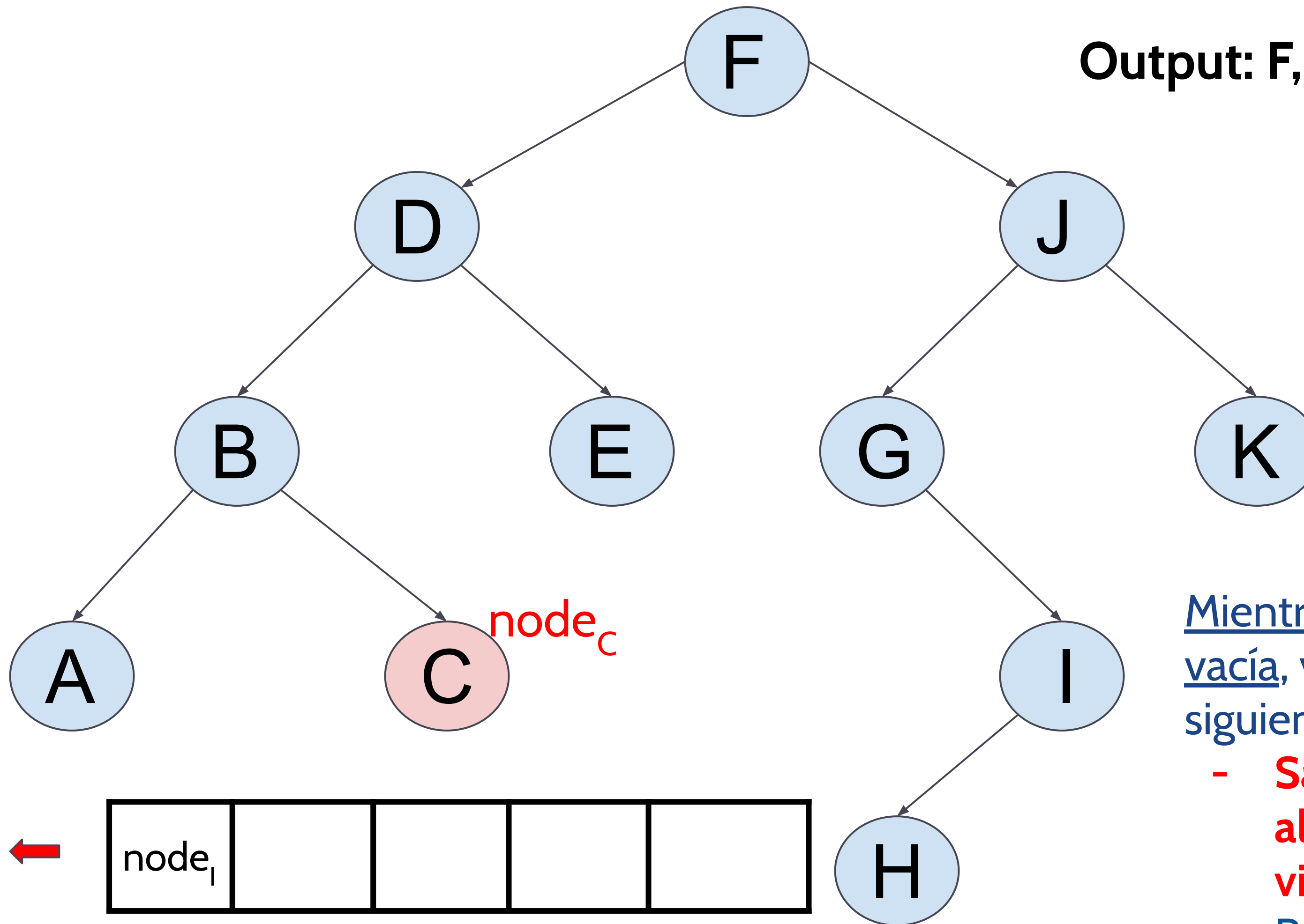
Output: F, D, J, B, E, G, K, A

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

node$_A$ no tiene hijos, no añadimos nada a la lista

Recorrido por niveles



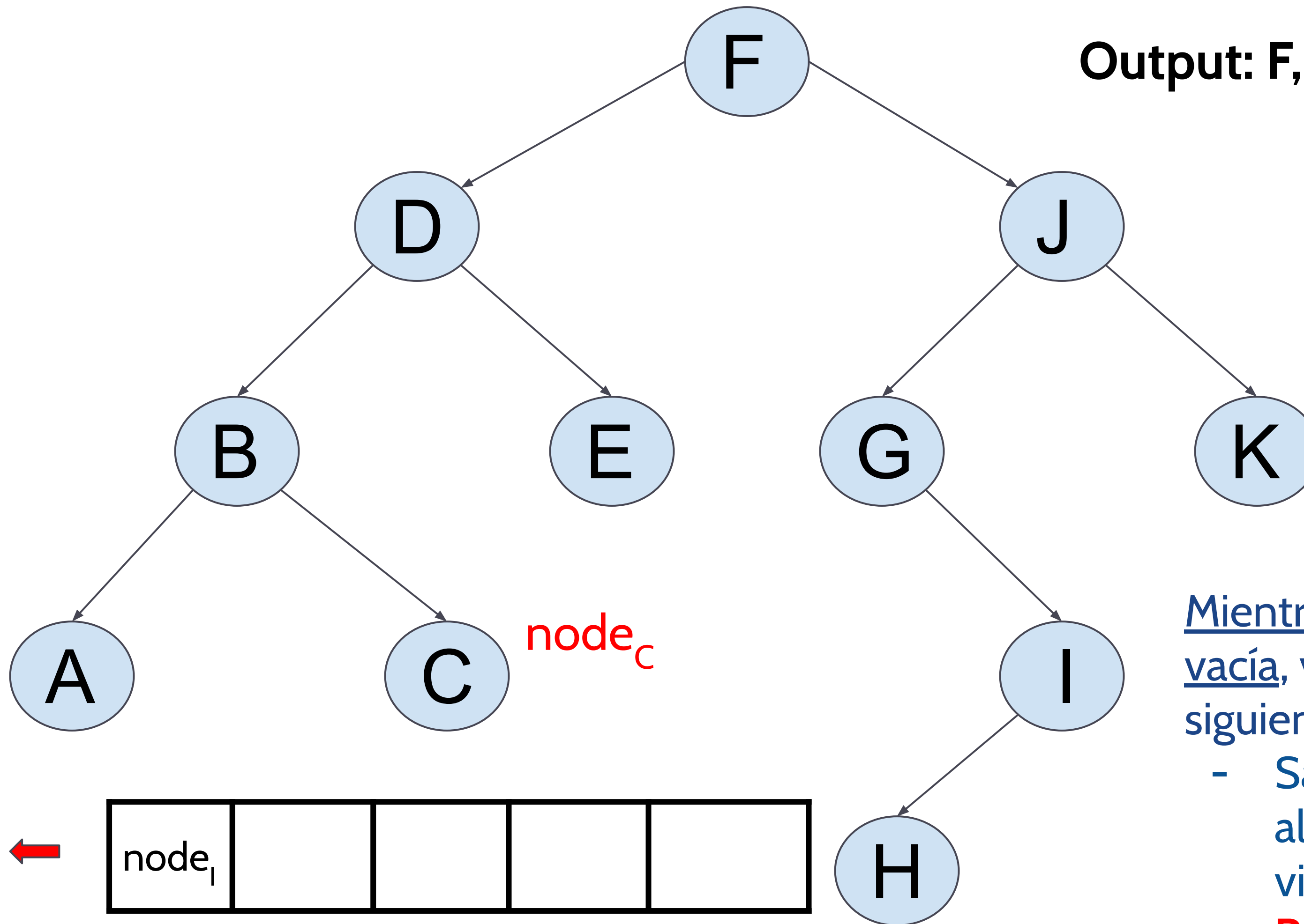
Output: F, D, J, B, E, G, K, A, C

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

Recorrido por niveles

Output: F, D, J, B, E, G, K, A, C

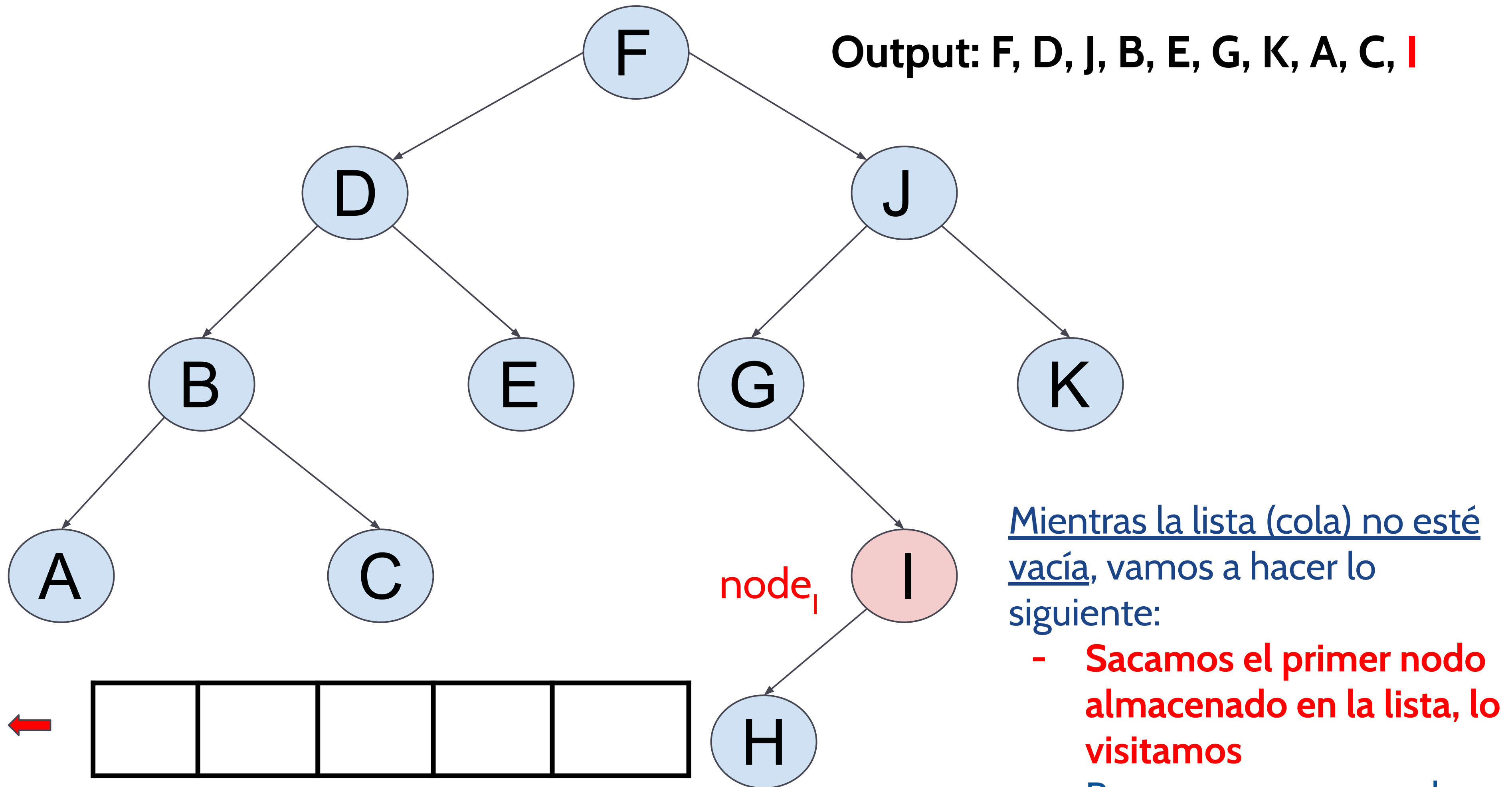


<u>Mientras la lista (cola) no esté vacía</u>, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

node$_C$ no tiene hijos, no añadimos nada a la lista

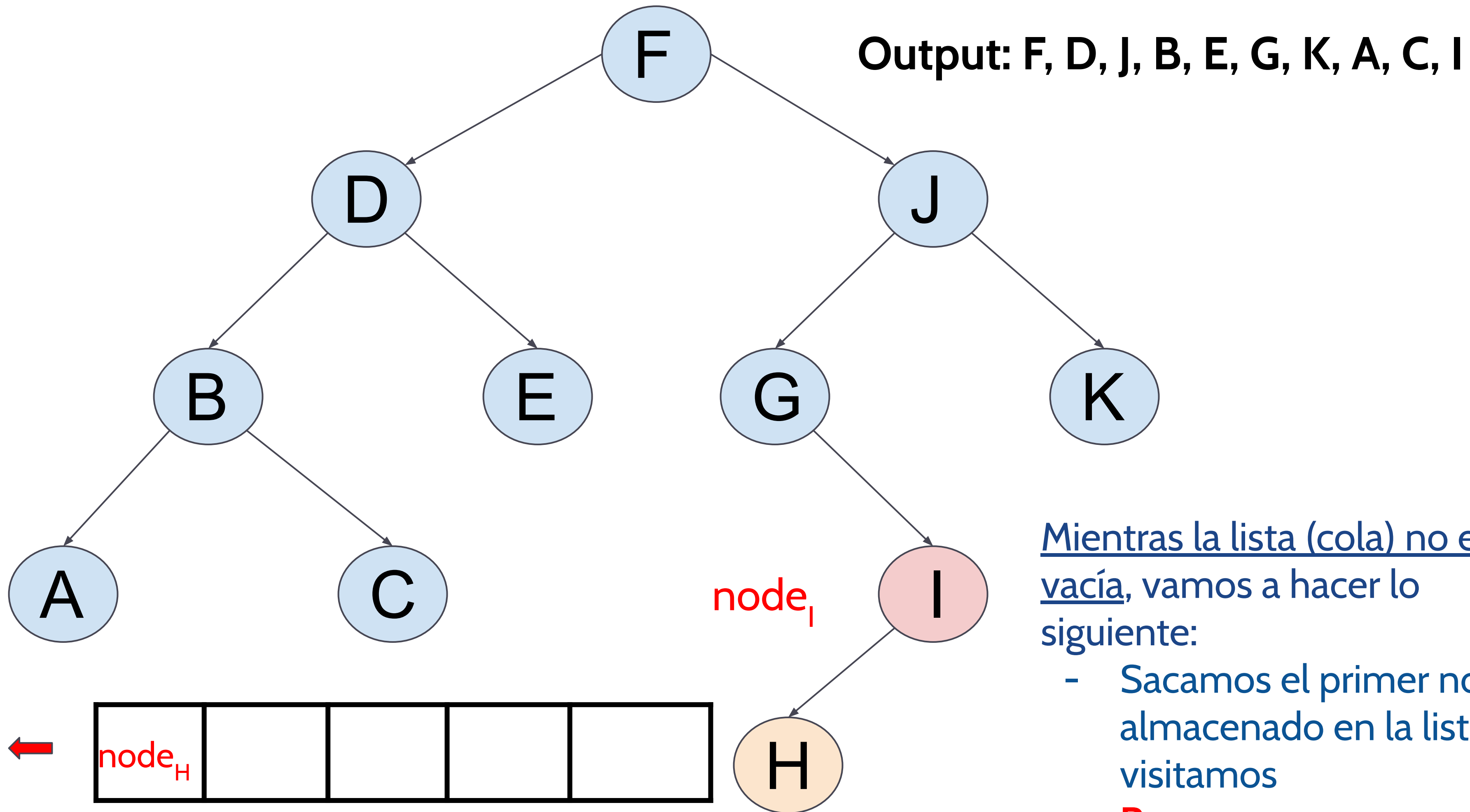
Recorrido por niveles



<u>Mientras la lista (cola) no esté vacía</u>, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

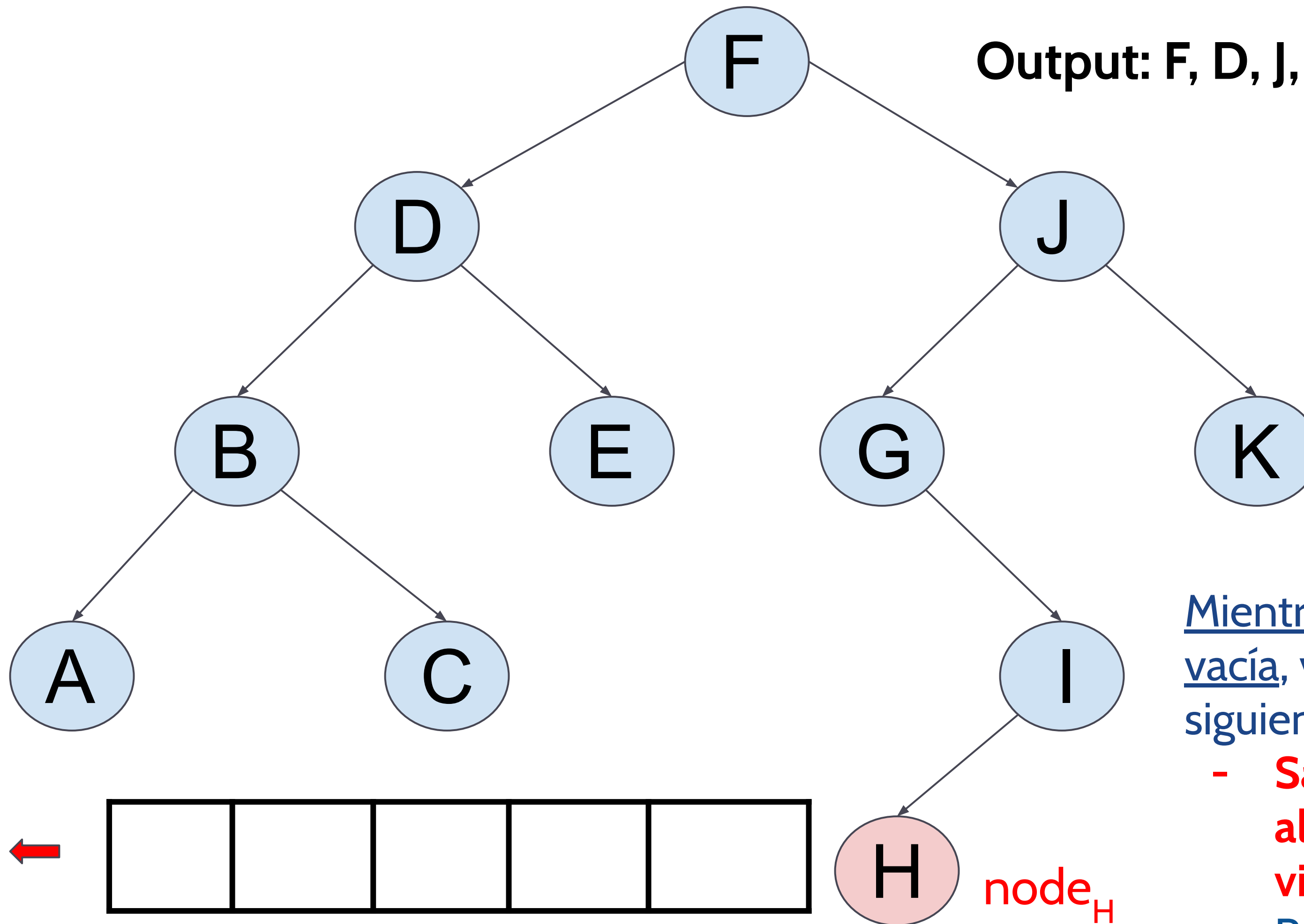
Recorrido por niveles



<u>Mientras la lista (cola) no esté vacía</u>, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

Recorrido por niveles

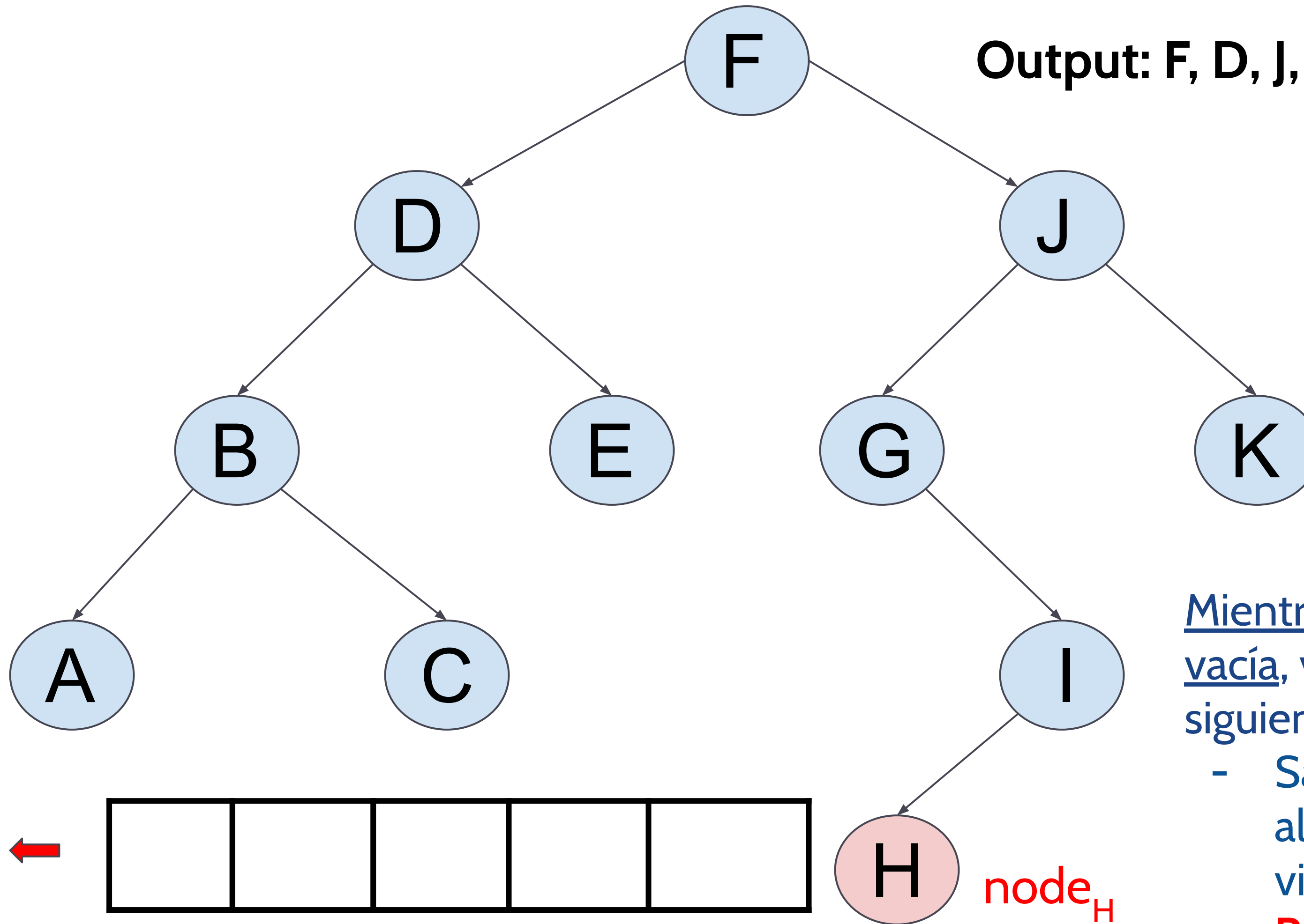


Output: F, D, J, B, E, G, K, A, C, I, H

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- **Sacamos el primer nodo almacenado en la lista, lo visitamos**
- Recuperamos sus nodos hijos y los almacenamos en la lista

Recorrido por niveles



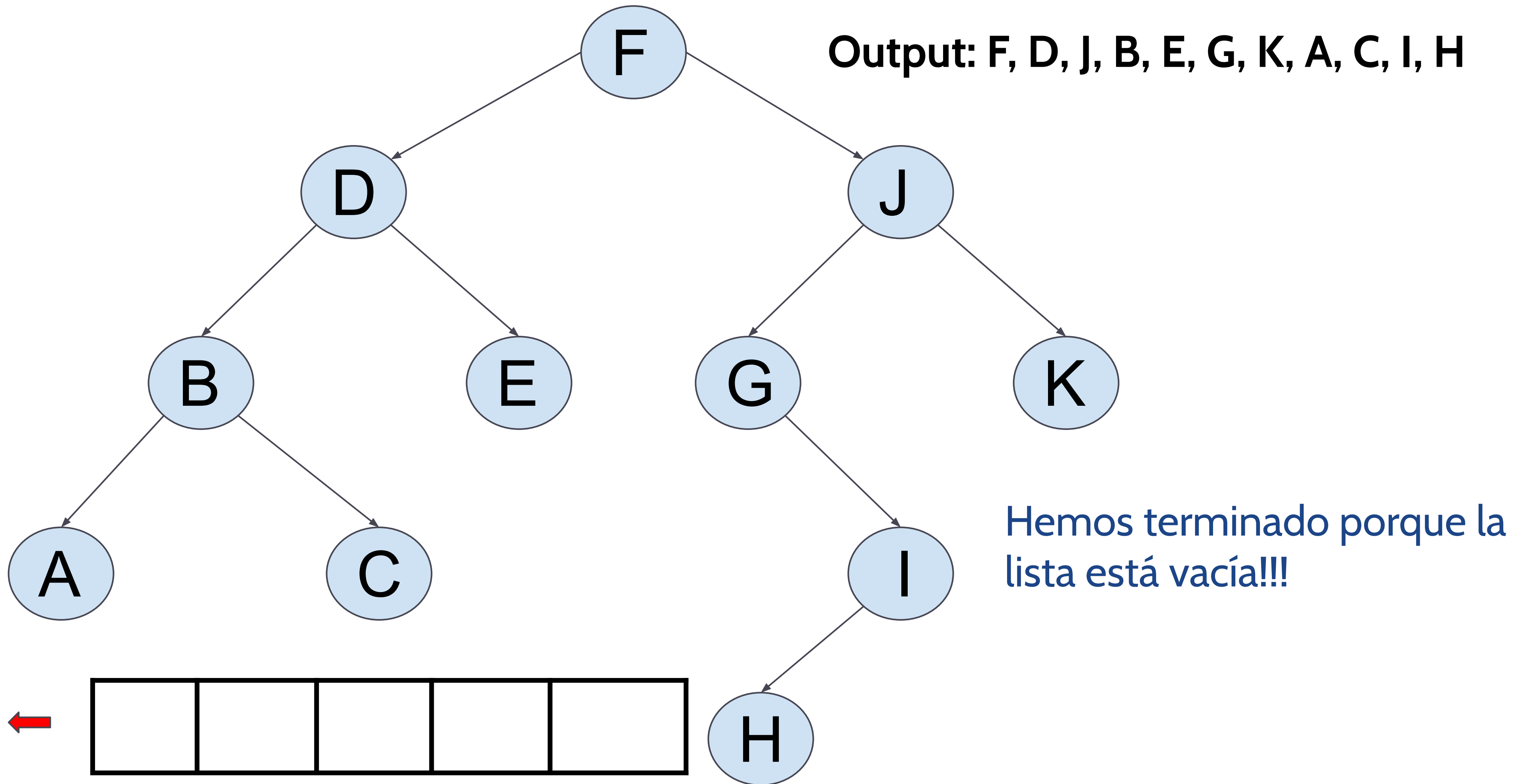
Output: F, D, J, B, E, G, K, A, C, I, H

Mientras la lista (cola) no esté vacía, vamos a hacer lo siguiente:

- Sacamos el primer nodo almacenado en la lista, lo visitamos
- **Recuperamos sus nodos hijos y los almacenamos en la lista**

$node_H$ no tiene hijos, no añadimos nada a la lista

Recorrido por niveles



Recorrido por niveles

```python
def level_order(self) -> None:
    """prints the level order of the tree."""
    if self._root is None:
        print('tree is empty')
    else:
        print("Level order: ", end=' ')  # avoid the new line

        # we use a Python list to save
        # the children of the nodes that we are visiting
        list_nodes = [self._root]
        while len(list_nodes) > 0:
            current = list_nodes.pop(0)  # get the first node
            print(current.elem, end=' ')
            # Each time we visit a node, we save their children into list_nodes
            if current.left is not None:
                list_nodes.append(current.left)
            if current.right is not None:
                list_nodes.append(current.right)

        print()
```

Recorrido por niveles

La complejidad espacial de la función es O(n) porque utiliza un array (lista de Python).
Su complejidad temporal es $O(n^2)$

```
while len(list_nodes) > 0:  # loop will be executed the size of tree: n
    current = list_nodes.pop(0)  # O(n)
    print(current.elem, end=' ')

    if current.left is not None:
        list_nodes.append(current.left)  # O(1)
    if current.right is not None:
        list_nodes.append(current.right)  # O(1)
```

Recorrido por niveles

Ejercicio: ¿Cómo puedes conseguir que la complejidad level_order sea lineal (O(n))?

```
while len(list_nodes) > 0:  # loop will be executed the size of tree: n
    current = list_nodes.pop(0)  # O(n)
    print(current.elem, end=' ')

    if current.left is not None:
        list_nodes.append(current.left)  # O(1)
    if current.right is not None:
        list_nodes.append(current.right)  # O(1)
```

Recorrido por niveles

```
def level_order(self) -> None:
    """prints the level order of the tree. O(n)"""
    if self._root is None:
        print('tree is empty')
    else:
        print("Level order: ", end=' ')  # avoid the new line

        # we can use SList with tail and head
        list_nodes = SList()
        list_nodes.addLast(self._root)
        while len(list_nodes) > 0:  # loop will be executed the size of tree: n
            current = list_nodes.removeFirst() # O(1)
            print(current.elem, end=' ')
            if current.left is not None:
                list_nodes.addLast(current.left)  # O(1)
            if current.right is not None:
                list_nodes.addLast(current.right)  # O(1)
```