

Tema 1:

**Resolución de problemas mediante técnicas de**

# **Búsquedas**

# Objetivos

- **Recordatorio** de las características de los problemas de búsqueda
- **Repaso** de los algoritmos principales
- Aplicación a distintos campos

# FORMULACIÓN GENERAL DE PROBLEMAS



# Formulación general

- La formulación de un problema es el proceso que consiste en **decidir qué acciones** y **qué estados** hay que considerar para la consecución de una **meta** u objetivo
- **Metodología** general

# Formulación

- El verdadero arte de la solución de problemas consiste en saber qué decidir y **qué es lo importante** para describir estados y operadores y **qué no**.
- El proceso de **eliminación de detalles** de una representación se denomina **abstracción**.

# Formulación

Vamos a considerar lo que se da en llamar **Inteligencia Artificial Clásica**:

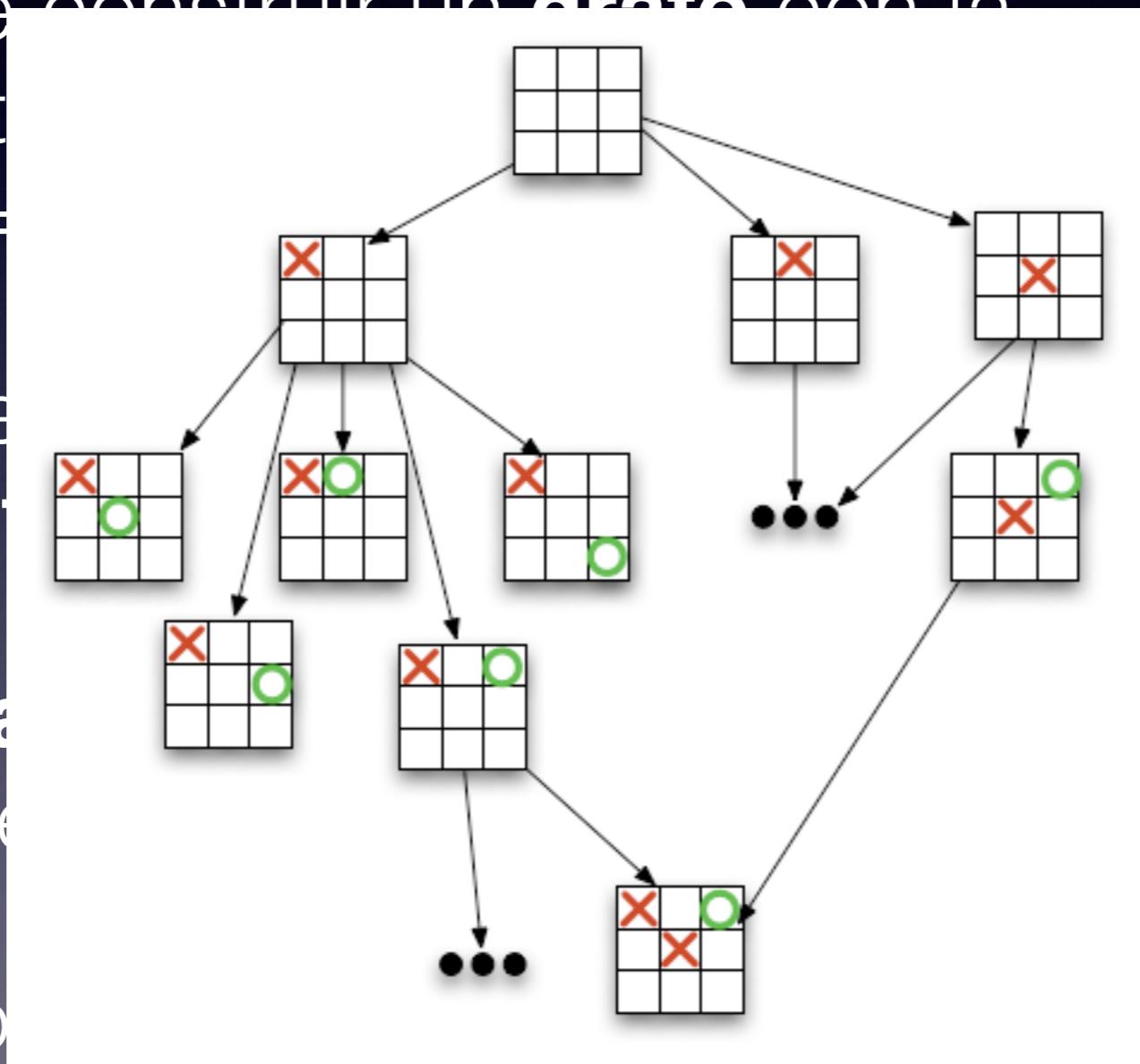
- **Estático**: no se “mueve” nada entre movimientos (o turnos).
- **Observable**: Conocemos todas las variables del problema.
- **Discreto**: las variables tienen un límite de división
- **Determinístico**: El estado siguiente de nuestro problema es consecuencia (solamente) del estado actual y de la acción que hagamos.

# Características

- Se puede construir un **grafo** con la representación del problema
  - Nodo = estado
- Deberá tener un **estado inicial** y, al menos, **un (estado) final.**
- Las **aristas** serán las transiciones (**acciones**) entre los estados.
- Ese grafo puede tener **ciclos**

# Características

- Se puede construir un **grafo** con la representación de los estados.
- Nodo = **Estado**
- Deberá tener una lista de **vecinos**, un **(estado)** que es el resultado de las **operaciones** posibles.
- Las **aristas** representan las transiciones entre los estados.
- Ese grafo es un **árbol**.

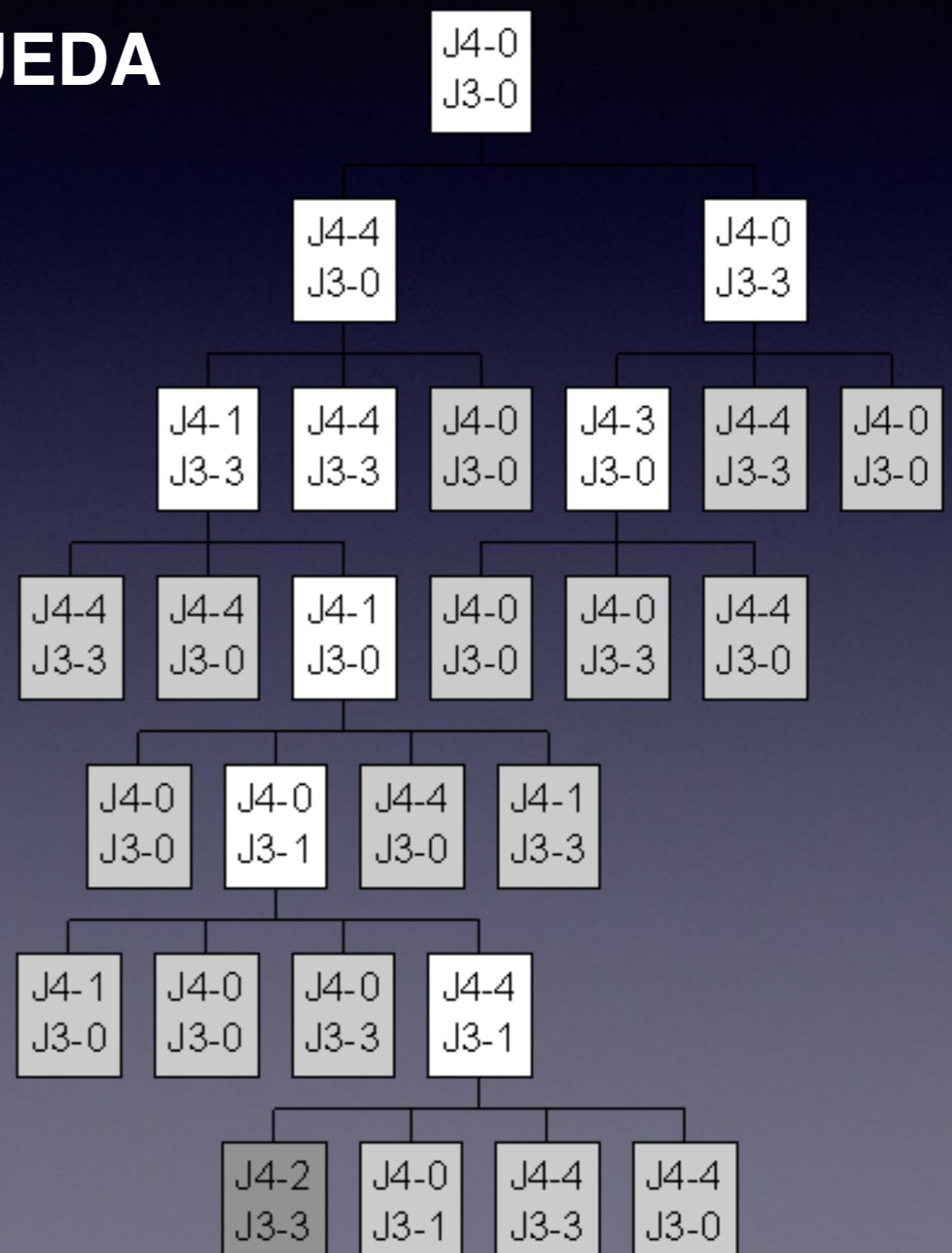


# Espacio de Estados

Espacio de estados: problema de los vasos de agua

## Se construye el ARBOL DE BÚSQUEDA

- Cada nodo es un estado
- El nodo raíz corresponde al estado inicial
- Cada arista es la aplicación de una acción
- los nodos hoja son los estados finales.



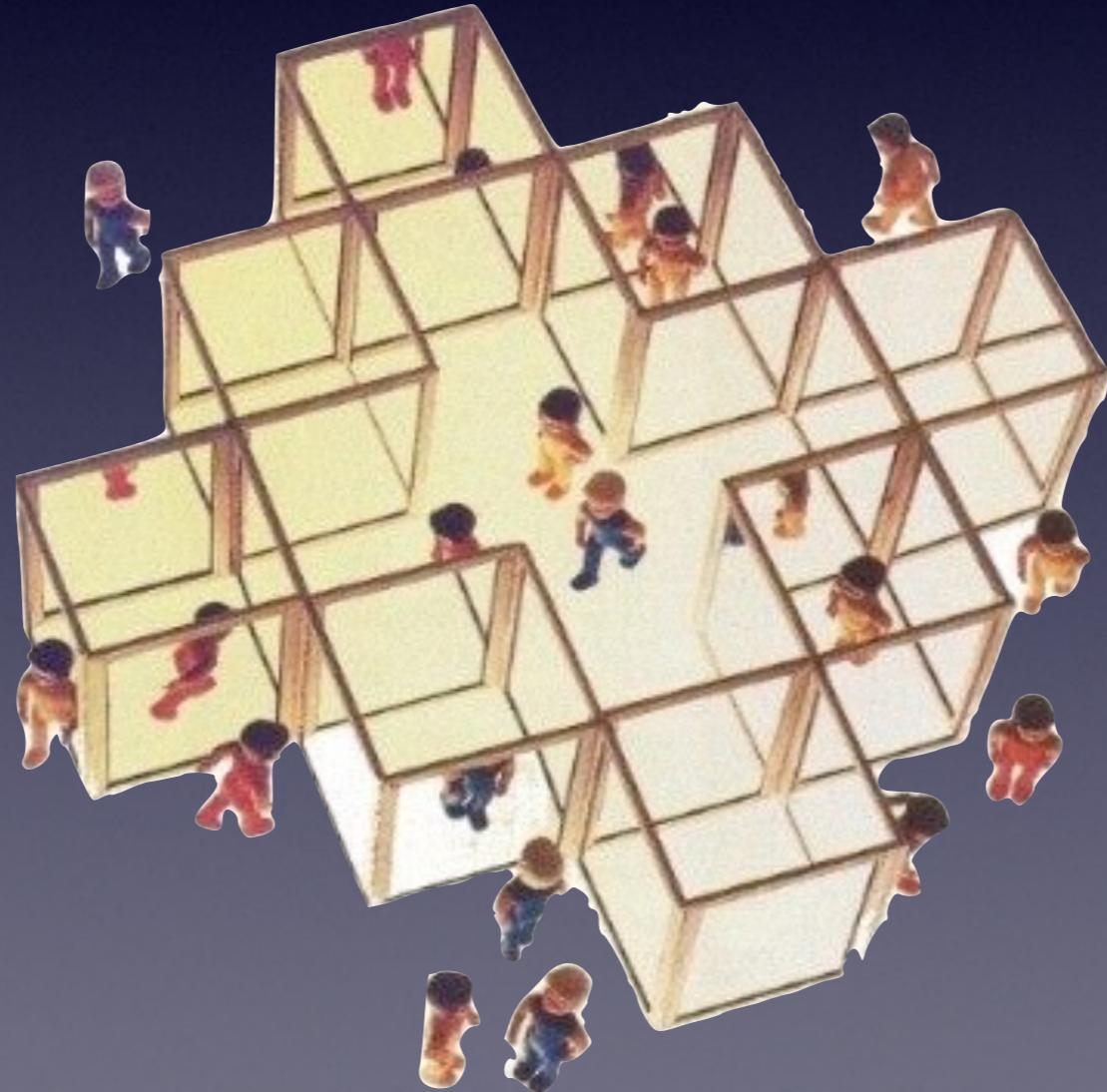
# Espacio de Estados

- Elementos básicos para definir un problema en el espacio de estados son:
  1. **Estado Inicial**
  2. **Operadores (Acciones)**
  3. **Prueba de Meta (Final)**
  4. **(Costo de Ruta)**



# 1. Estado Inicial

- El estado **inicial** es el estado con el que arranca el agente



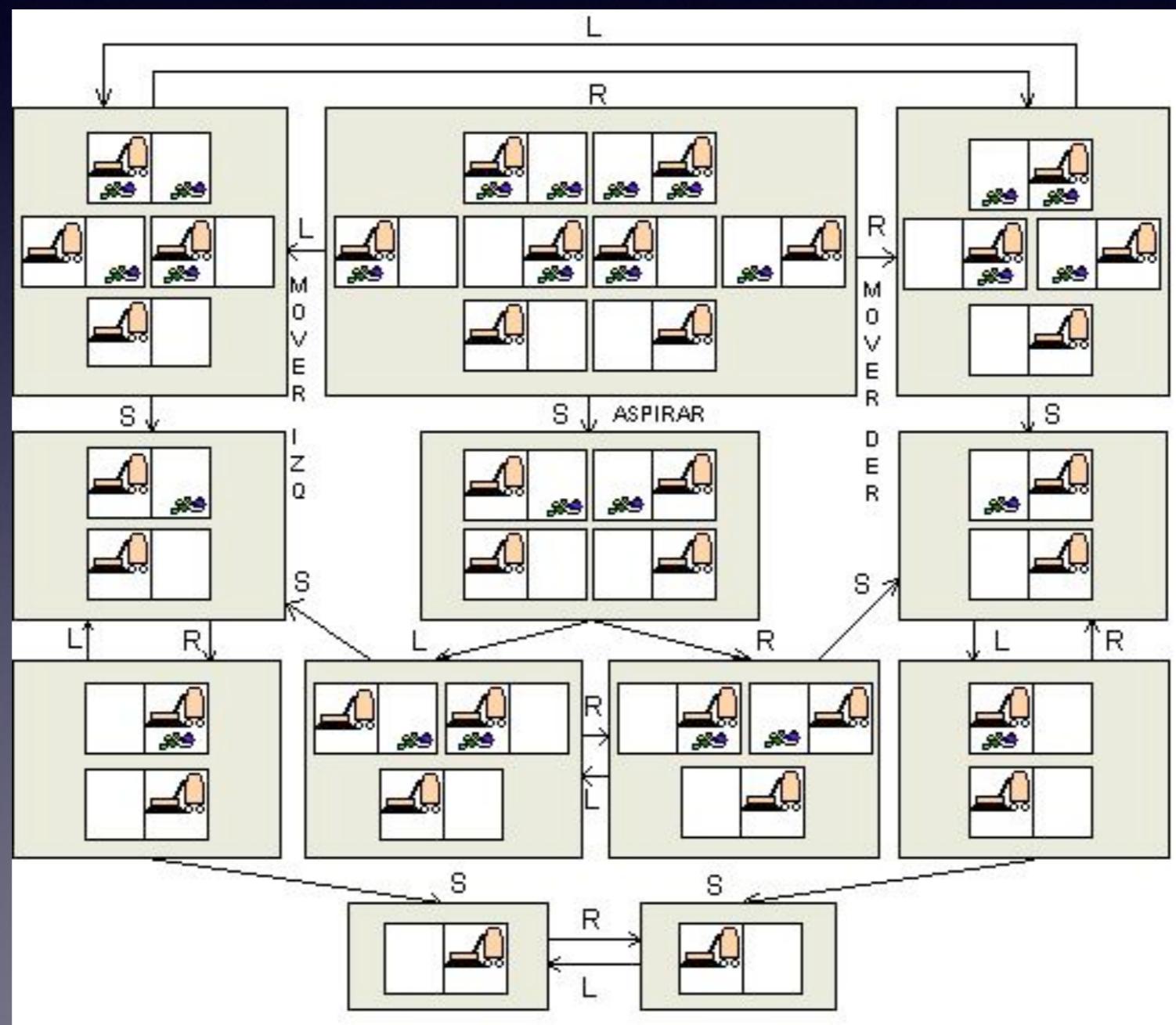
# 2. Operadores

- Los operadores son las **posibles acciones** que el agente puede emprender.
- La formulación mas común usa una función de sucesor. Dado un estado particular  $x$ ,  $\text{Suc}(x)$  regresa un conjunto de pares ordenados **<acción, sucesor>**



# 2. Operadores

- Estado (n) + Acción >> Estado (n+1)



# 3. Prueba de Meta

- La prueba de meta es una **condición** que tenemos del problema para saber que hemos **terminado**
- Esta condición puede ser un **estado concreto**
- o una **propiedad** (o conjunto de) que se haga cierta en un momento. Puede crear un conjunto de estados finales.

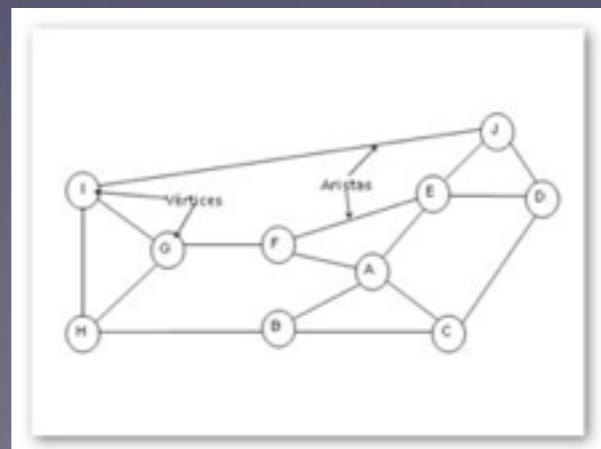


# 4. Costo de Ruta

- El costo de ruta es una **función** mediante la que se asigna un costo a una **acción** determinada.
- El costo **total** es la **suma de todos los costos** de cada una de las acciones individuales a lo largo de una ruta.

# Solución

- **Construcción del grafo** de estados, con sus posibles transiciones.
  - Algoritmo: **Búsqueda de camino** en grafos.
  - Solución: **Conjunto de acciones** que debe de realizar un agente para llegar desde el estado inicial a cualquiera de los estados finales.

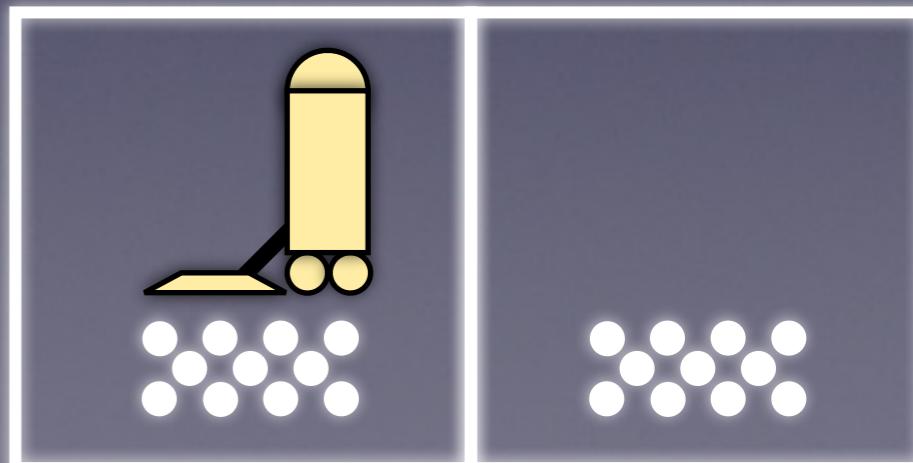


# EJEMPLOS



# 1. Mundo de la aspiradora

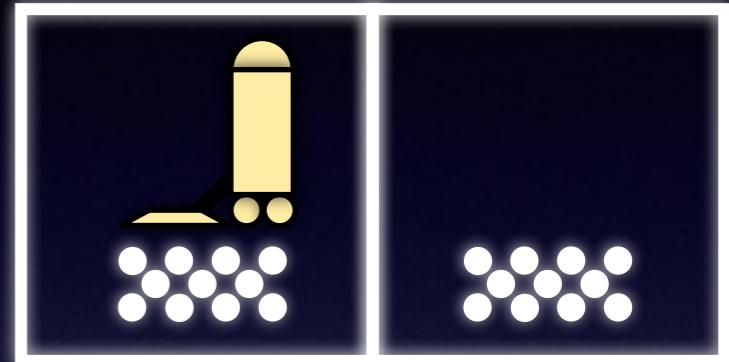
- En este mundo hay dos posibles habitaciones.
- El agente se encuentra en una de las dos
- En ellas puede o no haber suciedad.
- Son tres las acciones posibles: A la izquierda, A la derecha y Aspirar.
- Se puede suponer que la eficiencia del aspirado es 100%
- La meta es eliminar toda la suciedad.



# 1. Mundo de la aspiradora

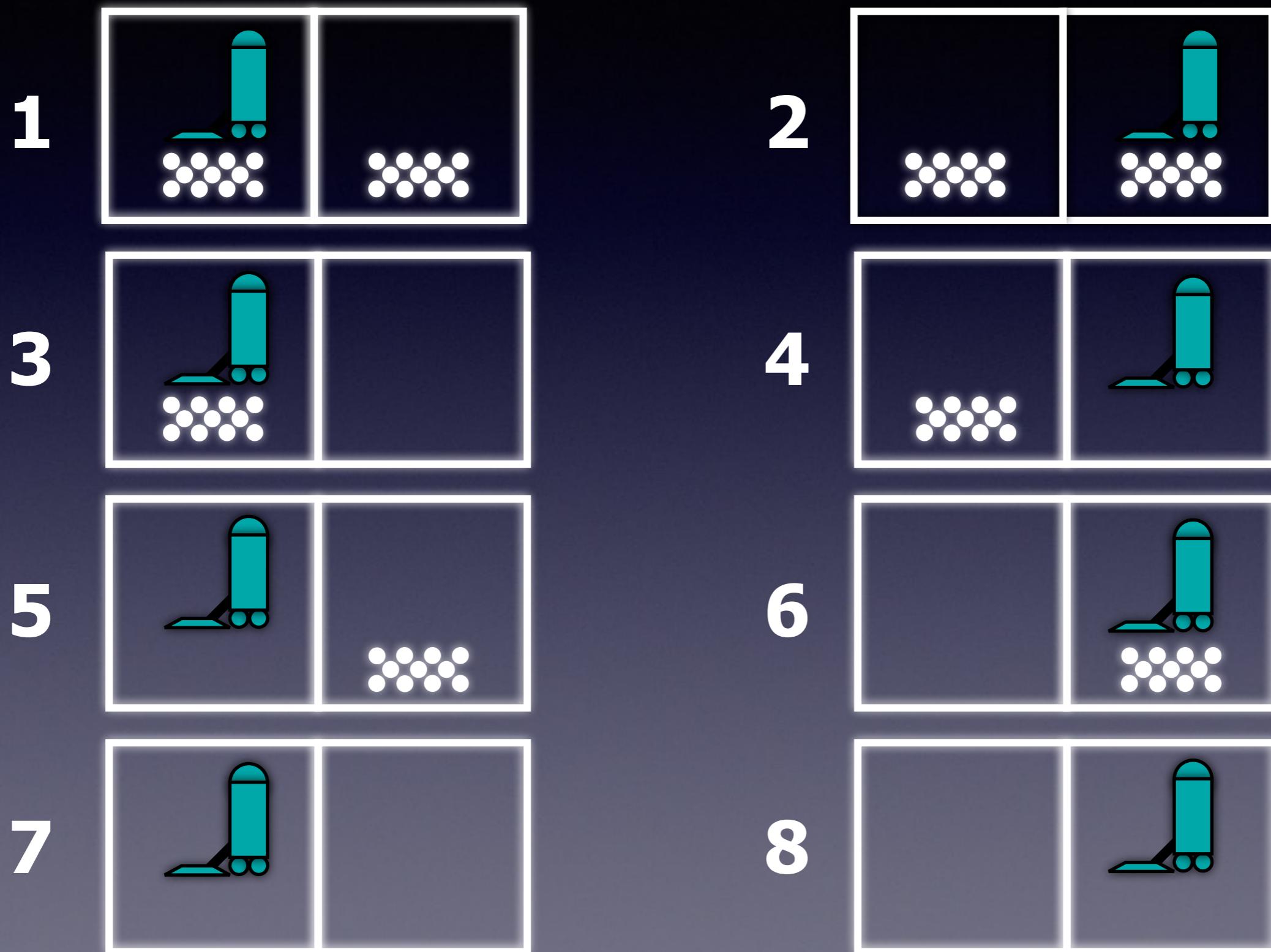
Representación del problema

- Estado:
  - El mapa de las dos habitaciones (2)
  - la situación del robot (2)
  - Situación de la limpieza (sucia o limpia)
- Por lo tanto, puede haber  $2 \times 2 \times 2 = 8$  estados posibles del mundo.





# 1. Mundo de la aspiradora





# 1. Mundo de la aspiradora



# 1. Mundo de la aspiradora

- **Estado inicial:** Cualquier estado puede ser designado como estado inicial.



# 1. Mundo de la aspiradora

- **Estado inicial:** Cualquier estado puede ser designado como estado inicial.
- **Operador:** las acciones posibles serán: izquierda, derecha y aspirar.



# 1. Mundo de la aspiradora

- **Estado inicial:** Cualquier estado puede ser designado como estado inicial.
- **Operador:** las acciones posibles serán: izquierda, derecha y aspirar.
- **Prueba de Meta:** Revisa si las dos ubicaciones están limpias.



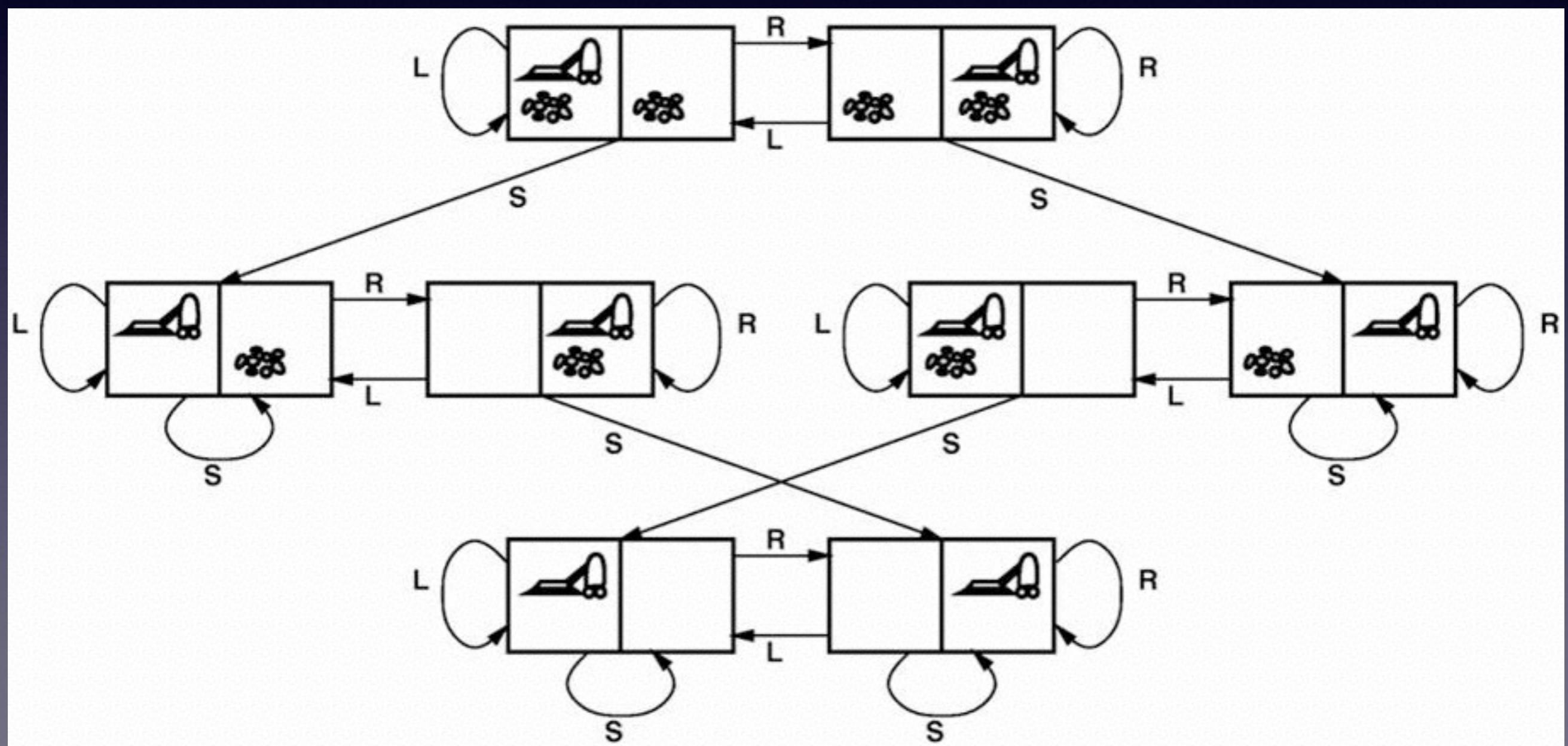
# 1. Mundo de la aspiradora

- **Estado inicial:** Cualquier estado puede ser designado como estado inicial.
- **Operador:** las acciones posibles serán: izquierda, derecha y aspirar.
- **Prueba de Meta:** Revisa si las dos ubicaciones están limpias.
- **Costo de ruta:** Cada paso (o aplicación de acción) cuesta 1, así el costo de ruta es el número de pasos en la ruta.



# 1. Mundo de la aspiradora

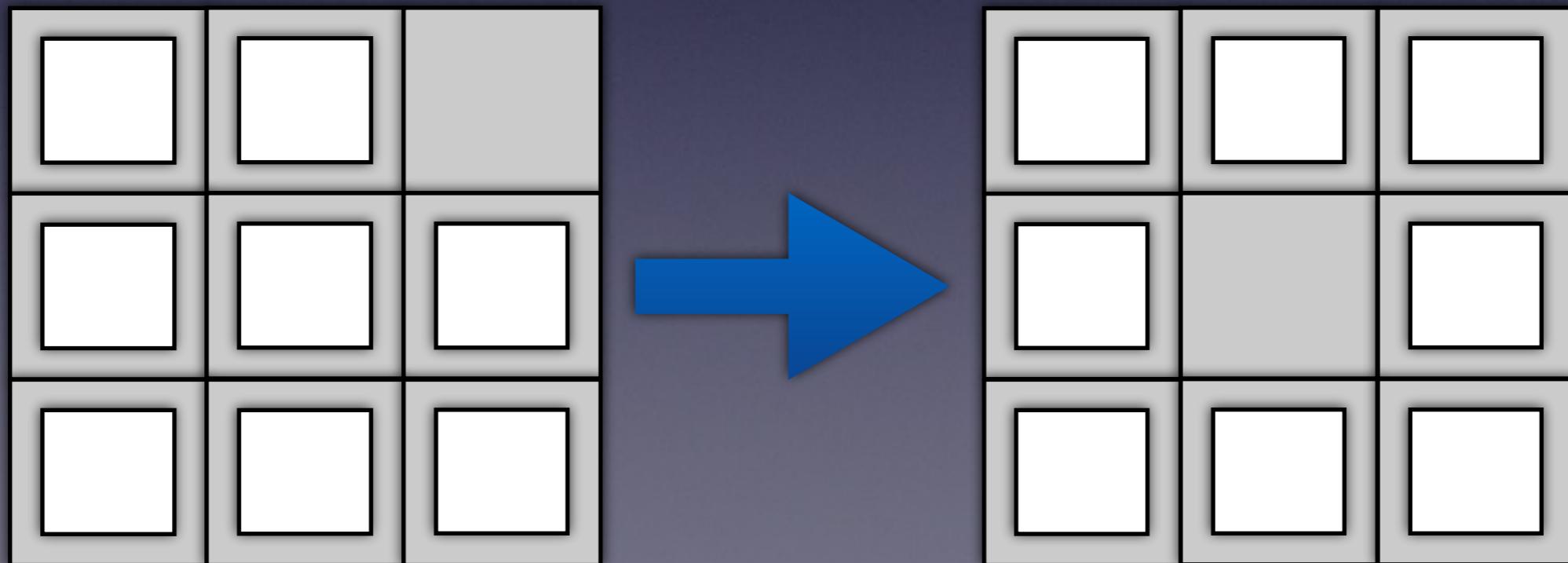
Diagrama de Estados





# 8-Puzzle

- Estados:
- Ubicación de cada una de las ocho placas en los nueve cuadrados. Conviene incluir la ubicación del espacio vacío.



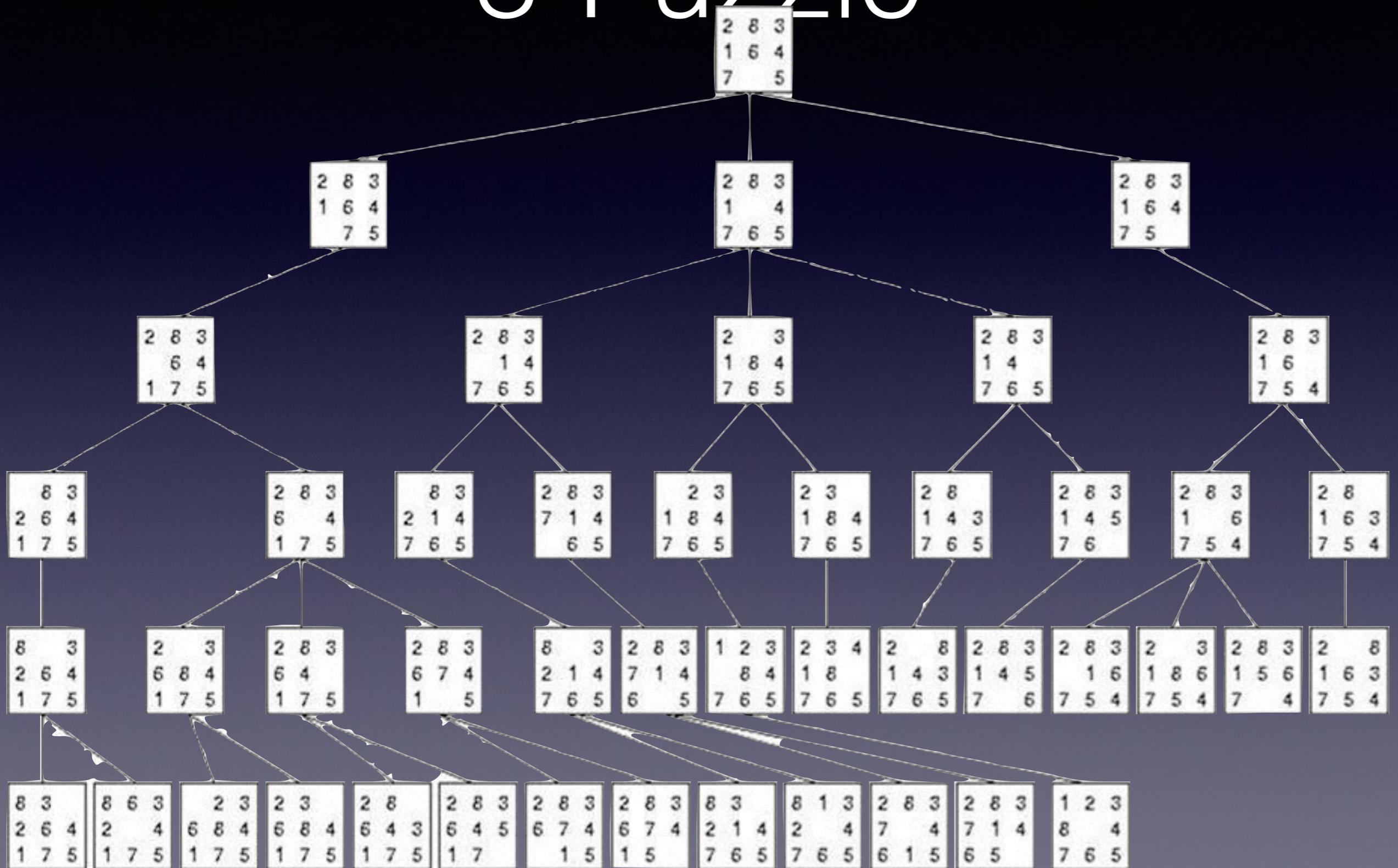


# 8-Puzzle

- Definición del estado:
  - Posición del tablero (de las 8 fichas y del espacio en blanco)
- Operadores:
  - ?



# 8-Puzzle



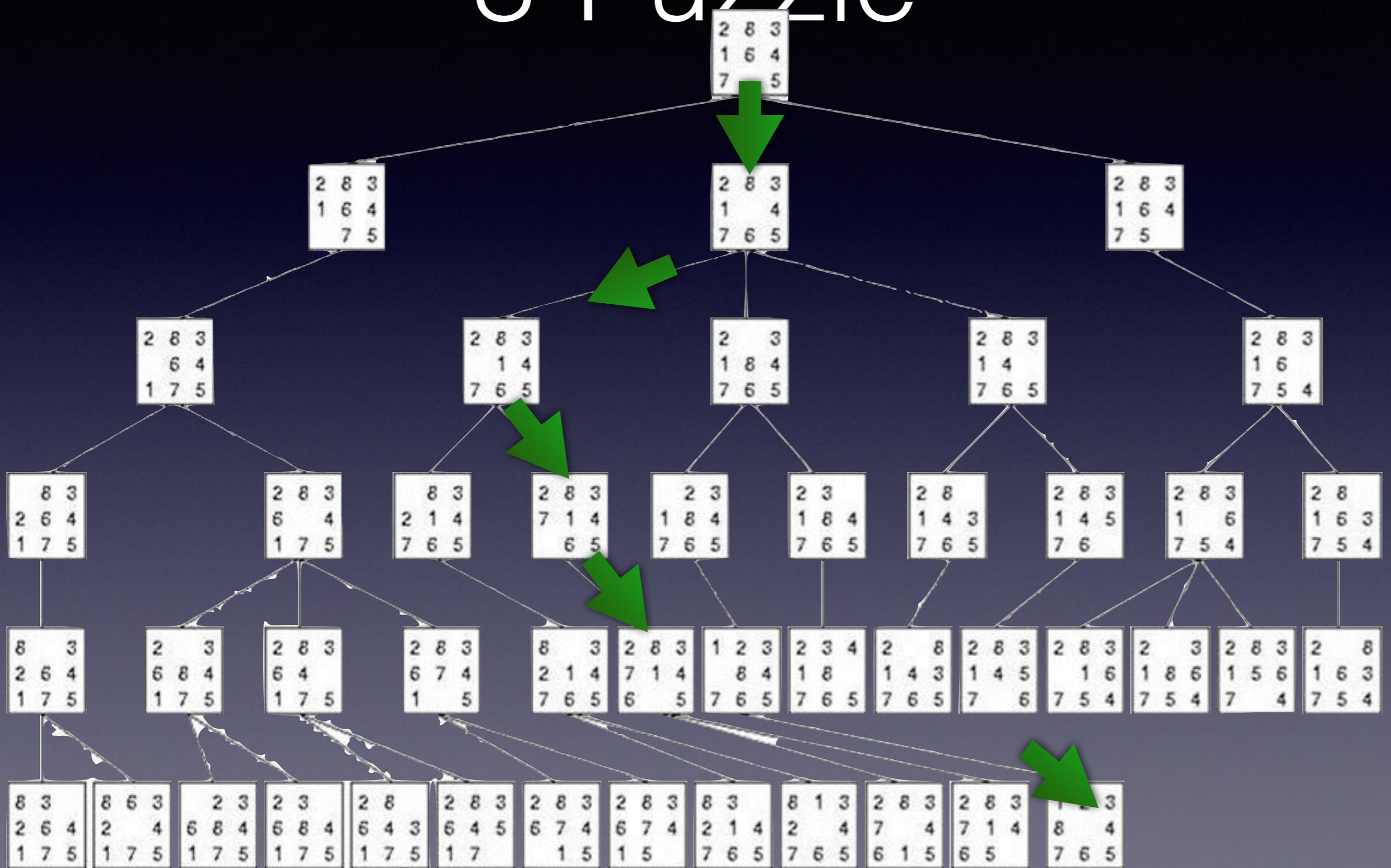


# 8-Puzzle

- Operadores:
  - Arriba, Abajo, Derecha e Izquierda.
- Prueba de meta:
  - Fichas ordenadas
- Coste
  - 1 por movimiento

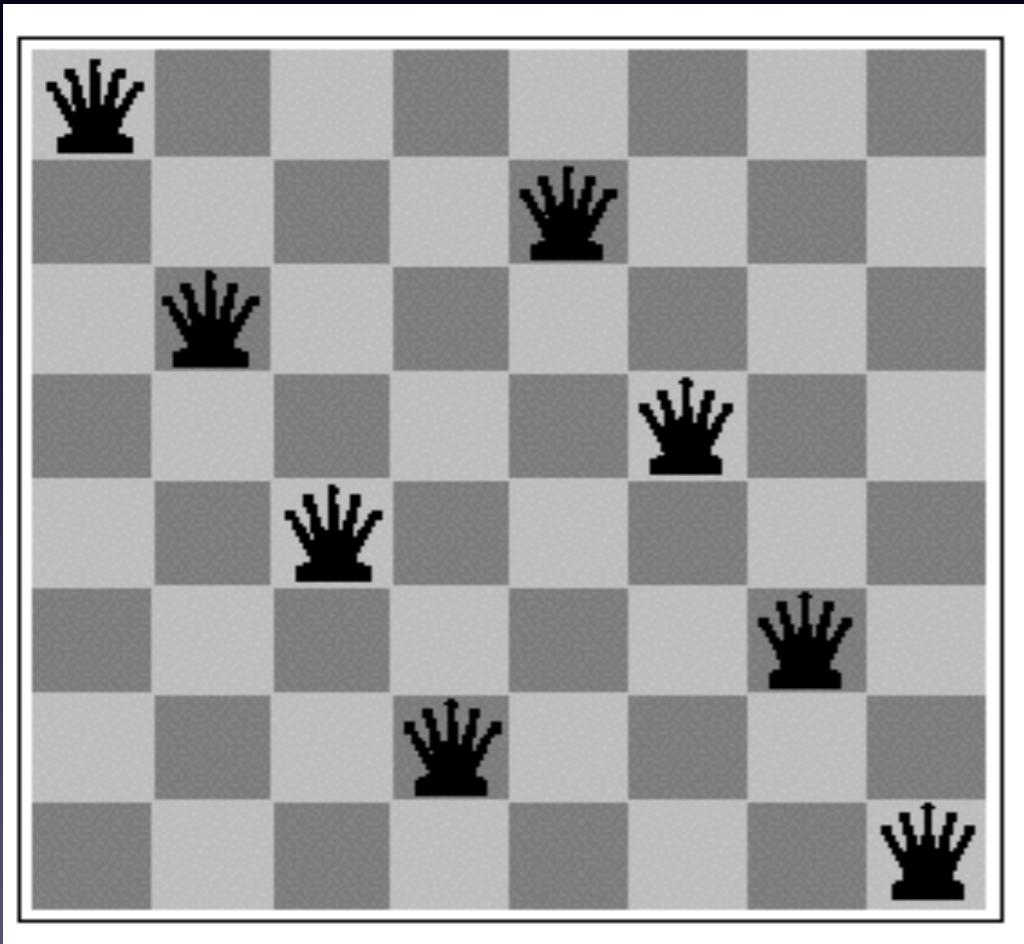


# 8-Puzzle





# 8-reinas

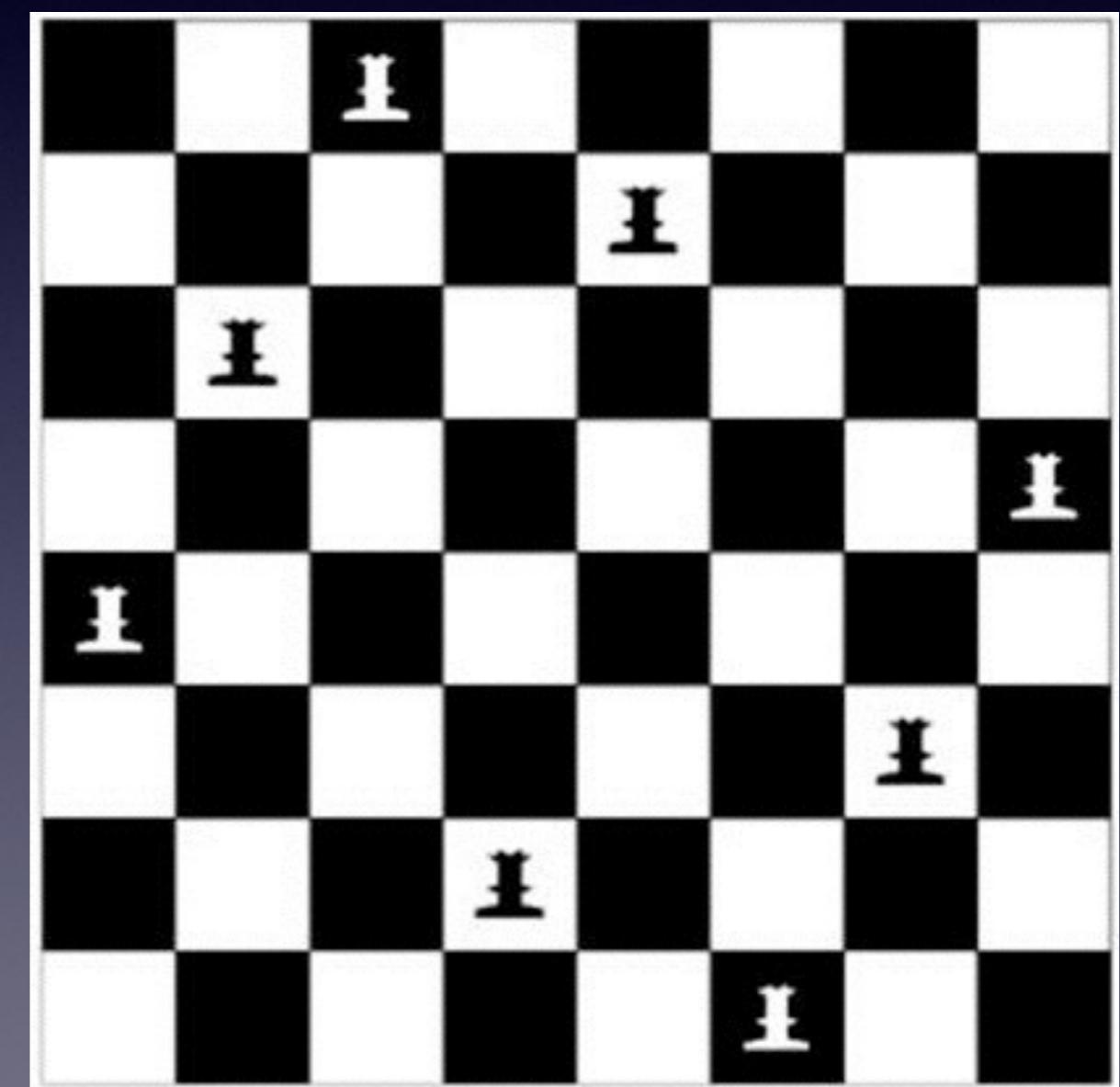
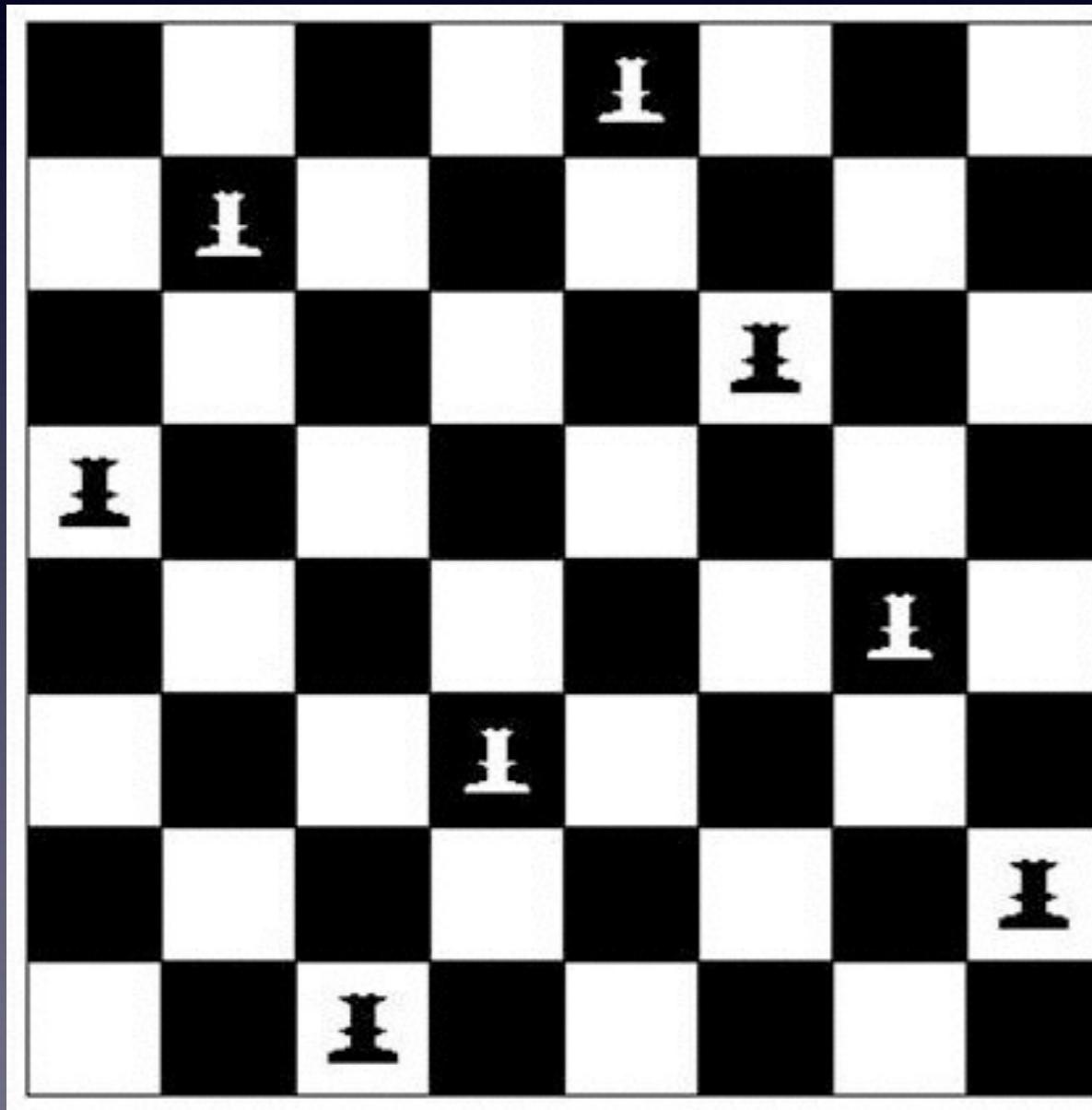


Colocar 8 reinas en un tablero de ajedrez de manera que ninguna de ellas esté en posibilidad de atacarse entre sí



# 3. El problema de las ocho reinas

Algunas soluciones al problema





# 4. Criptoaritmética

- Las letras representan dígitos, el objetivo es determinar una sustitución de dígitos por letras de manera que la operación resultante sea correcta aritméticamente. Por lo general cada letra representa un dígito distinto.

FORTY	29986
+ TEN	+ 850 {F=2, O=9, R=9, T=8, Y=6, ...}
+ TEN	+ 850
-----	-----
SIXTY	31486



# 5. Enrutamiento

- Consiste en definir una ruta en función de la especificación de ubicaciones y las transiciones mediante los vínculos que relacionan una y otra ubicación.
- Los algoritmos que resuelven estos problemas se usan en ruteo de redes de cómputo, sistemas automatizados de asesores de viajes, sistemas para planificación de viajes aéreos, bonificaciones a viajeros frecuentes, etc.



# Problema del viajante

- “Visitar todas las ciudades importantes del Perú por lo menos una vez, comenzando y terminando en Piura”
- La prueba de meta consistiría en haber visitado todas las ciudades UNA SOLA VEZ... en el camino más corto!!
- Los algoritmos que lo resuelven también se aplican para tareas tales como planeación de movimientos de taladros automáticos para circuitos impresos.





# Problema del viajante



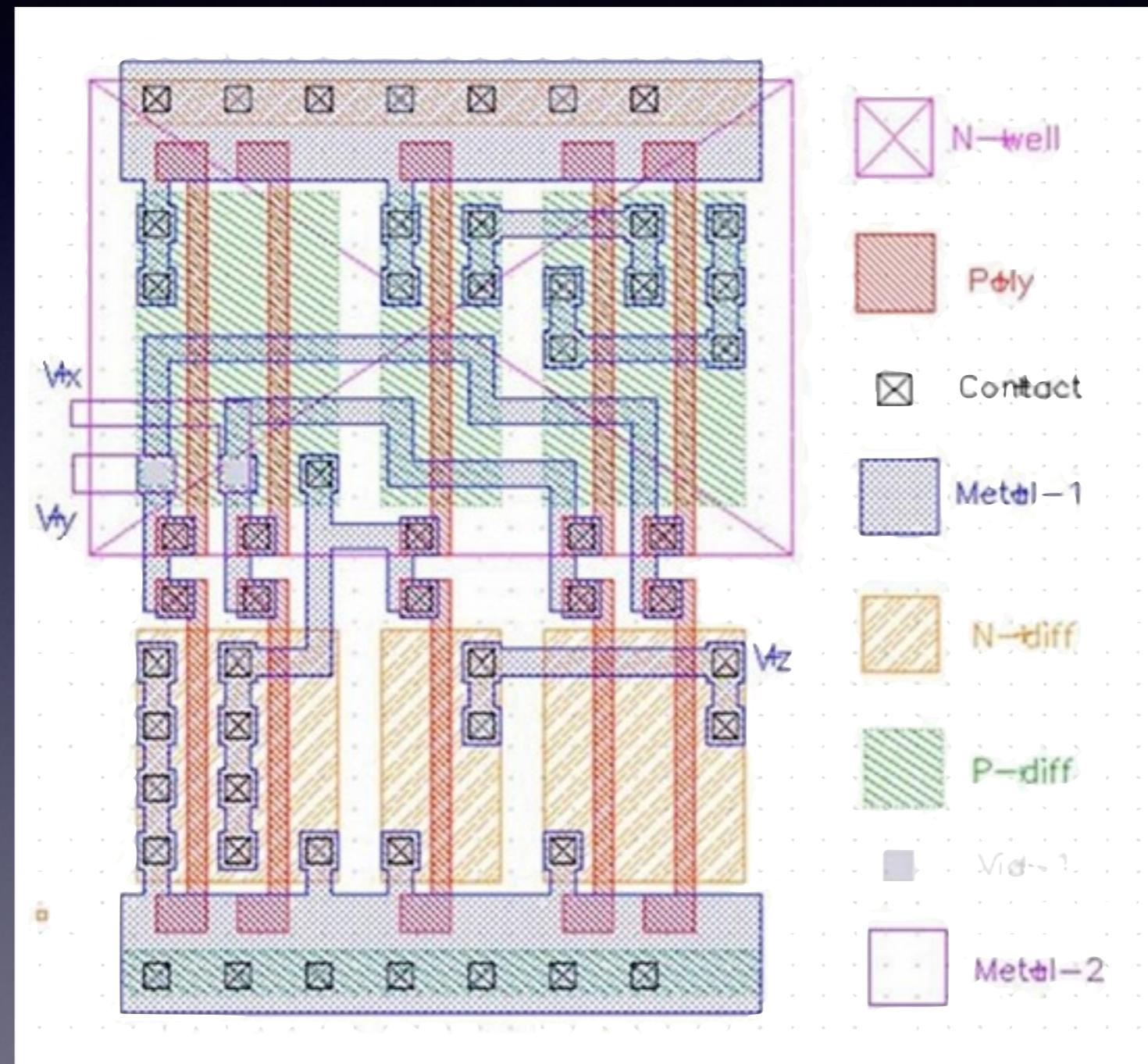
¿¿ RETO ??

# Distribución de componentes en circuitos VLSI

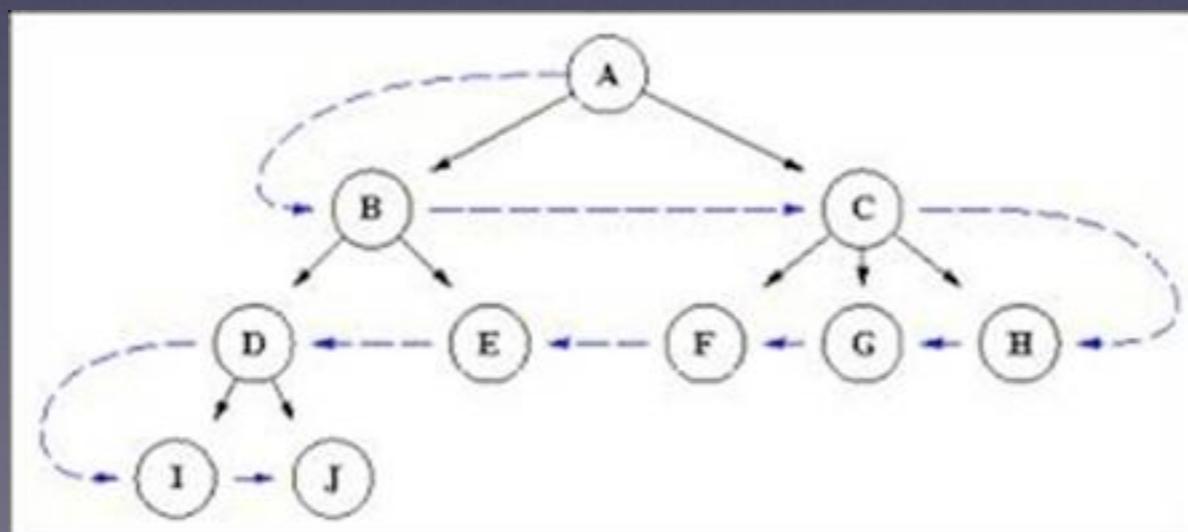
- Es una de las tareas más complejas de diseño en la ingeniería actual.
- El objetivo es colocar los miles de compuertas del circuito integrado de tal manera que no se encimen y ocupando un mínimo de área y de longitud de conexiones, con el fin de obtener un máximo de velocidad.



# Distribución de componentes en circuitos VLSI

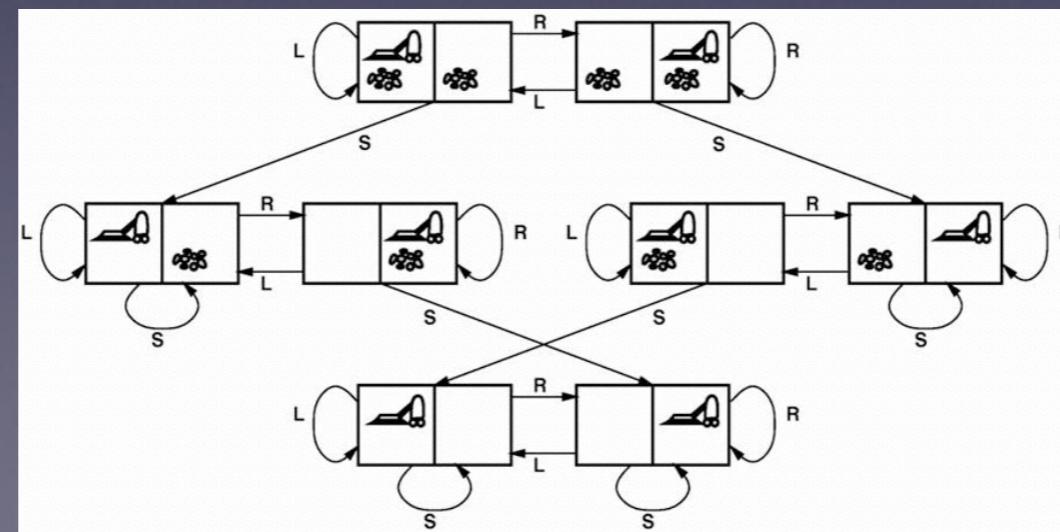


# BÚSQUEDA DE SOLUCIONES



# Búsqueda de Soluciones

- Se busca generar una **secuencia de acciones** para llegar desde el estado inicial al estado meta.
- La **búsqueda** se hace en el **espacio de estados**, y la idea es buscar una ruta en el grafo que nos lleve a un **nodo hoja** o **final**.



# Búsqueda de Soluciones

- Idea de **algoritmo**:
  - **Evaluar** el estado para determinar si es un estado **meta**.
  - **Si no** es meta:
    - Aplicar los operadores al estado actual, y **generar** un conjunto de estados **siguientes**.
    - **Elegir** el estado **siguiente** por el que continuar
      - **Aplicar** el operador
      - **Volver** al paso inicial

# Búsqueda de Soluciones

- La estrategia (o **inteligencia**) está en que criterio aplicar para **elegir** el siguiente estado a seguir desarrollando
- Conviene concebir el proceso de **búsqueda** como la **construcción** de un **árbol** de búsqueda sobrepuerto al espacio de estados.

Grafo de  
estados



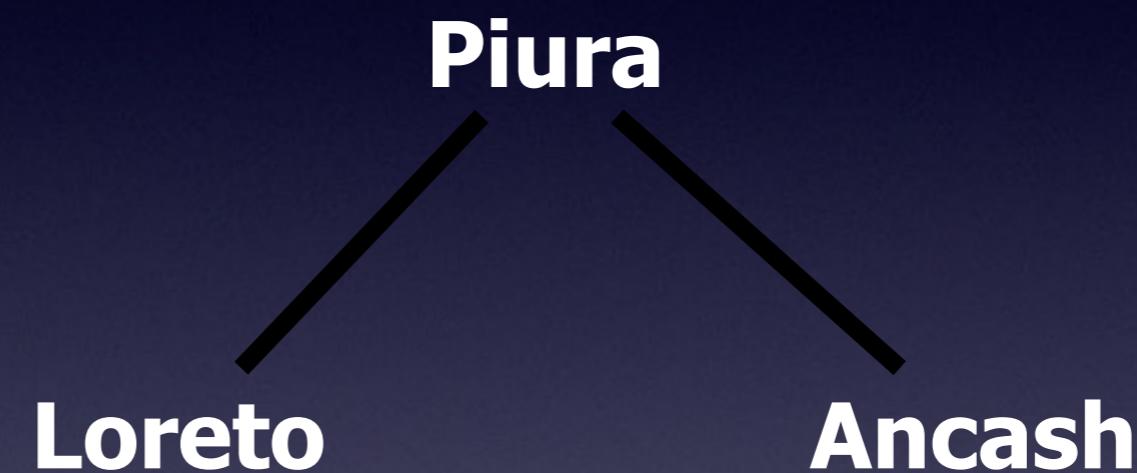
Árbol de  
Búsqueda

# Generación del árbol

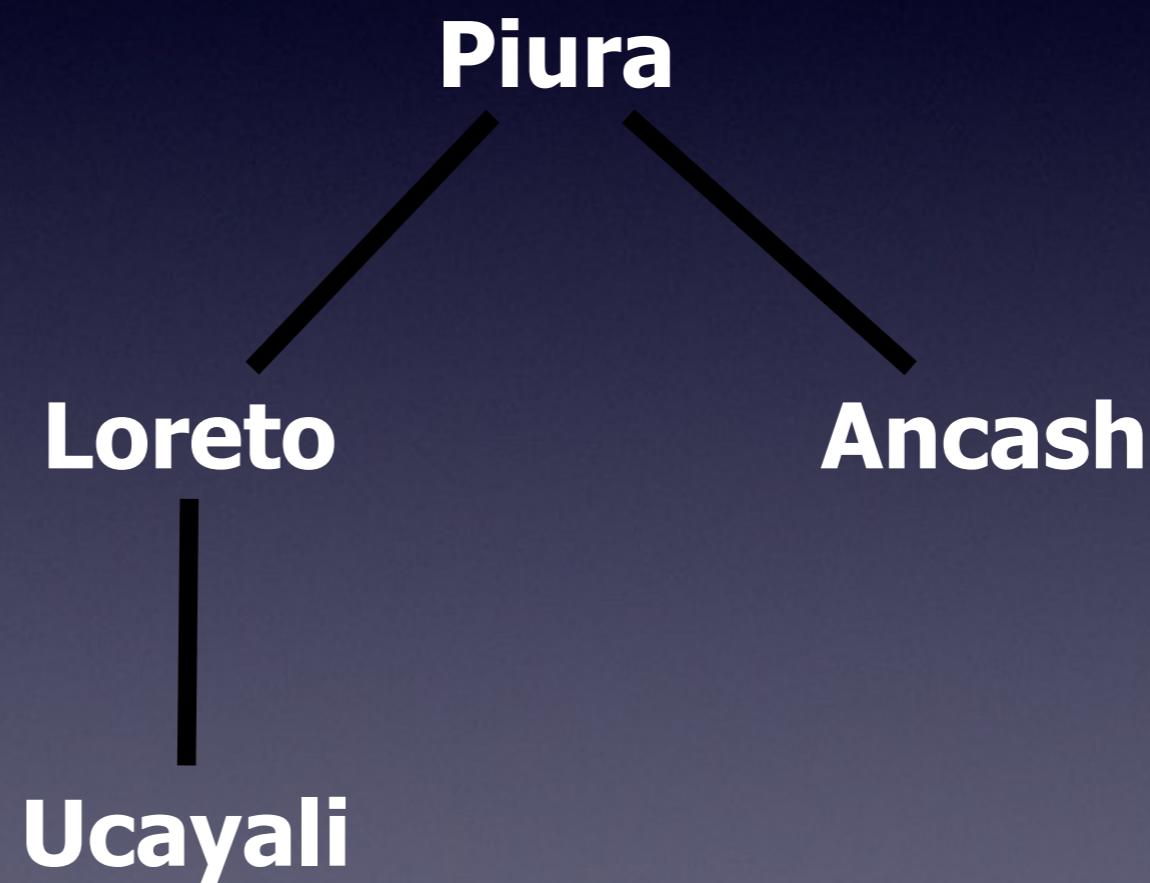
Estado inicial



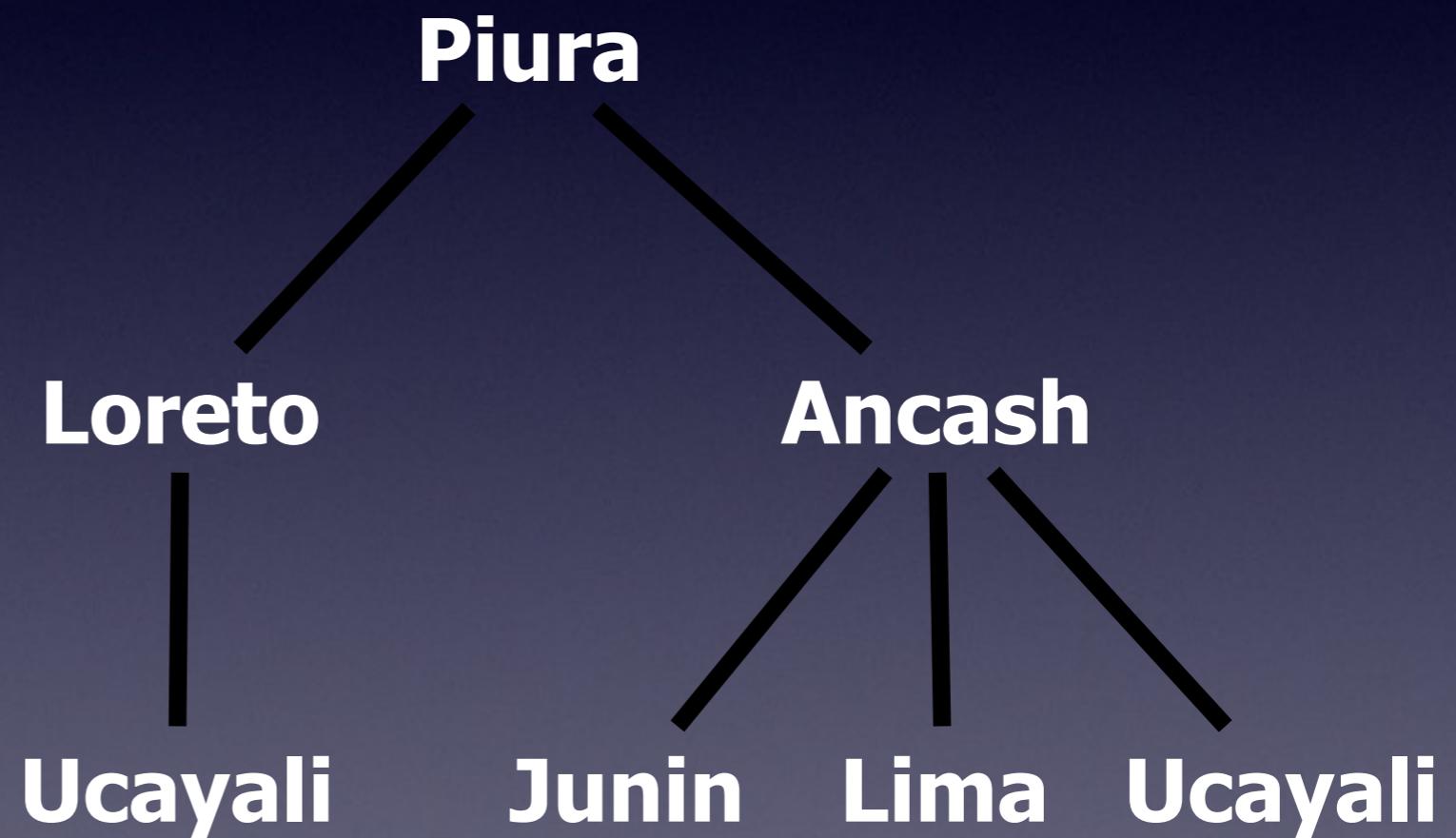
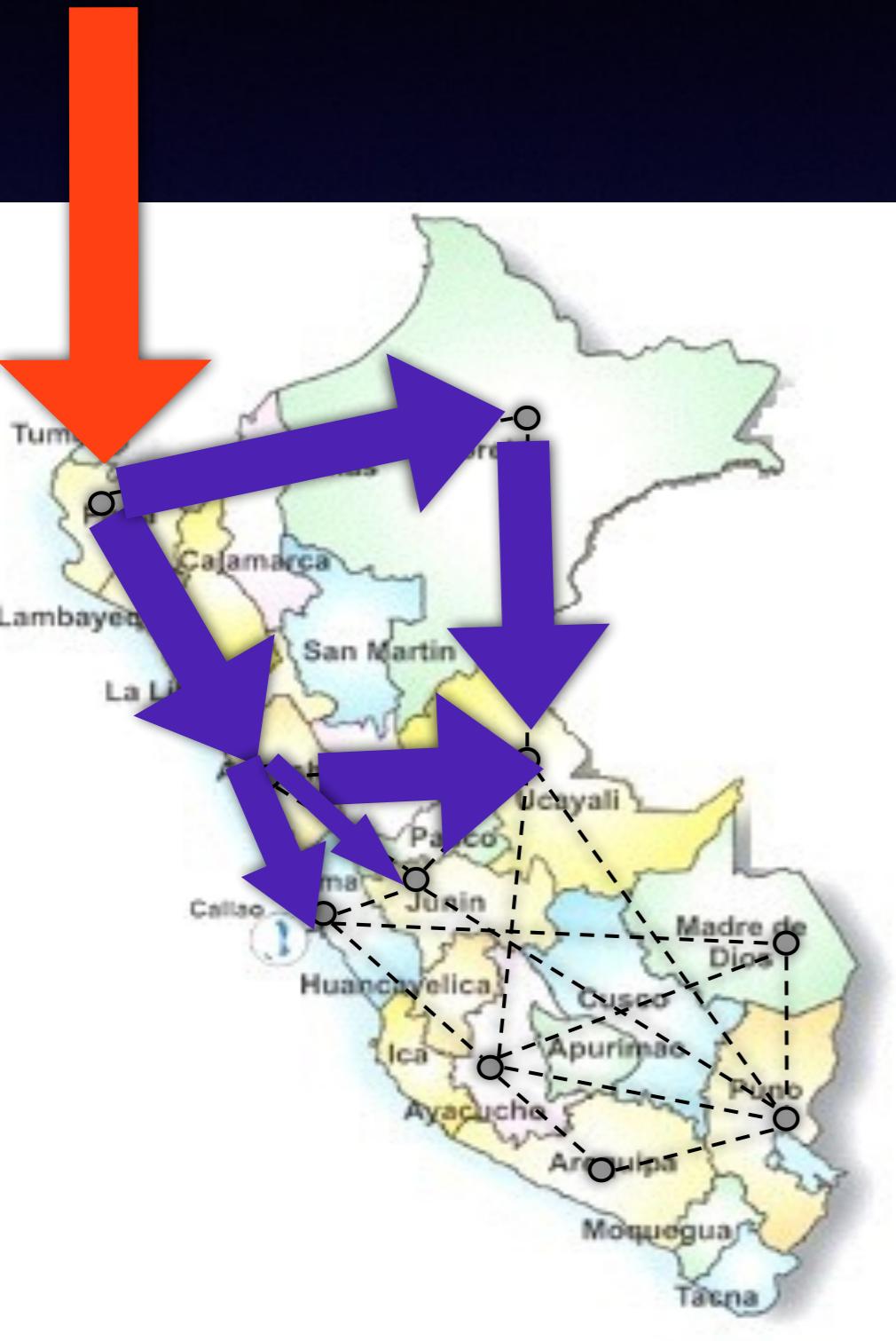
# Generación del árbol



# Generación del árbol



# Generación del árbol



¿ CÓMO ?

# Algoritmo GENERAL de búsqueda

```
public Lista<Accion> Busqueda(Nodo inicial, Nodo fin) {  
    Lista Abiertos = new Lista();  
    Lista Cerrados = new Lista();  
    Abiertos.Insertar(inicial);  
    while (!PruebaMeta()) {  
        Nodo Actual = Abiertos.SacarPrimero();  
        Cerrados.Insertar(Actual);  
        if (Actual == fin) {  
            return Recuperar_Camino(inic,Actual);  
        } else {  
            List<Nodo> sucesores = getSucesores(Actual);  
            sucesores = QuitarRepetidos(sucesores,Cerrados);  
            Abiertos.Insertar_Ordenado(sucesores);  
        }  
    }  
    return null;  
}
```

Estrategia

# Estrategias de búsquedas

# Estrategias de búsquedas

- **No Informadas (Ciegas):**
  - No contiene información de **cuanto de cerca** estamos de la solución
  - Sólo pueden distinguir si han **llegado** al objetivo **ó no.**
  - Son estrategias costosas, pero hay veces que son las únicas posibles.

# Estrategias de búsquedas

# Estrategias de búsquedas

- Informadas ó **Heurísticas**:
  - Hay una **ESTIMACIÓN** de cómo de cerca está de la solución
  - Medimos cuánto **CREEMOS** que nos falta para llegar al final
  - Hay veces que puede resultar peor que la estrategia ciega. Depende del problema

# Criterios para evaluar las estrategias

# Criterios para evaluar las estrategias

- **Corrección y Completitud**

¿La estrategia garantiza encontrar una solución **correcta**, si es que esta existe? ¿**Todas** las soluciones?

# Criterios para evaluar las estrategias

- **Corrección y Completitud**

¿La estrategia garantiza encontrar una solución **correcta**, si es que esta existe? ¿**Todas** las soluciones?

- **Complejidad**

# Criterios para evaluar las estrategias

- **Corrección y Completitud**

¿La estrategia garantiza encontrar una solución **correcta**, si es que esta existe? ¿**Todas** las soluciones?

- **Complejidad**

- **en tiempo:** ¿Cuánto tiempo se necesitara para encontrar una solución?

# Criterios para evaluar las estrategias

- **Corrección y Completitud**

¿La estrategia garantiza encontrar una solución **correcta**, si es que esta existe? ¿**Todas** las soluciones?

- **Complejidad**

- **en tiempo:** ¿Cuánto tiempo se necesitará para encontrar una solución?

- **en espacio:** ¿Cuánta memoria se necesita para efectuar la búsqueda?

# Criterios para evaluar las estrategias

- **Corrección y Completitud**

¿La estrategia garantiza encontrar una solución **correcta**, si es que esta existe? ¿**Todas** las soluciones?

- **Complejidad**

- **en tiempo:** ¿Cuánto tiempo se necesitará para encontrar una solución?

- **en espacio:** ¿Cuánta memoria se necesita para efectuar la búsqueda?

- **Optimización**

¿Con esta estrategia se encontrará una Solución Óptima?

# Estrategia NO informada

- Algoritmos
  - primero en anchura
  - primero en profundidad
  - primero en profundidad iterativa
  - ...

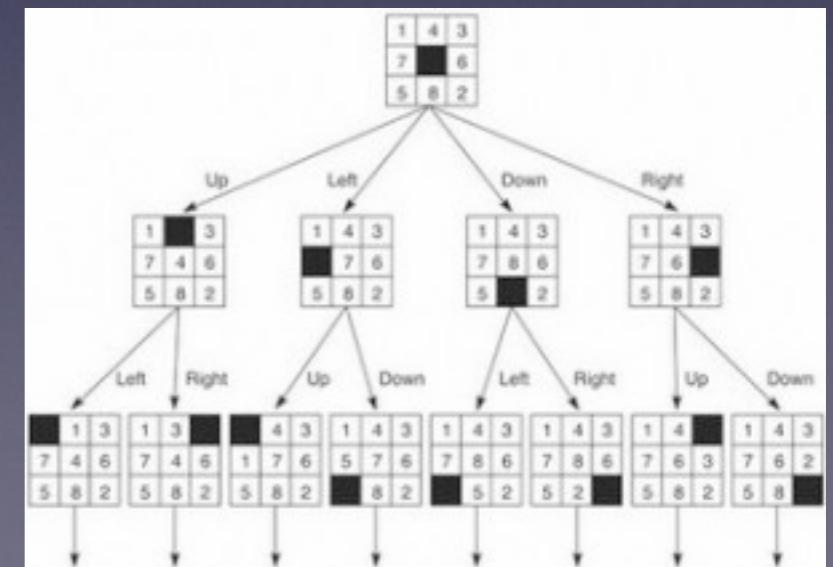
# Estrategia informada

- Algoritmos
  - primero el mejor (greedy)
  - Clase A
    - $A^*$
    - $A^*PI$
  - ...

# Otras estrategias

- Algoritmos
  - primero el mejor (greedy)
  - Clase A
    - A\*
    - A\*PI
  - ...

# Búsqueda No-informada



RECORRIDO EN  
ANCHURA

# Búsqueda en anchura

- No expandir nodos de nivel  $n$  hasta que todos los nodos de nivel  $n-1$  han sido expandidos
- Los nodos se visitan y generan por niveles.
- La estructura para los nodos abiertos es una cola (FIFO). (Se insertan al final)

# Búsqueda en anchura

- Completitud: el algoritmo siempre encuentra una solución (si existe).
- Complejidad temporal: exponencial respecto a la profundidad de la solución.  $O(r^p)$
- Complejidad espacial: exponencial respecto a la profundidad de la solución.  $O(r^p)$
- Optimización: la solución que se encuentra es óptima en número de niveles desde la raíz.

# Búsqueda en anchura

# Búsqueda en anchura

1

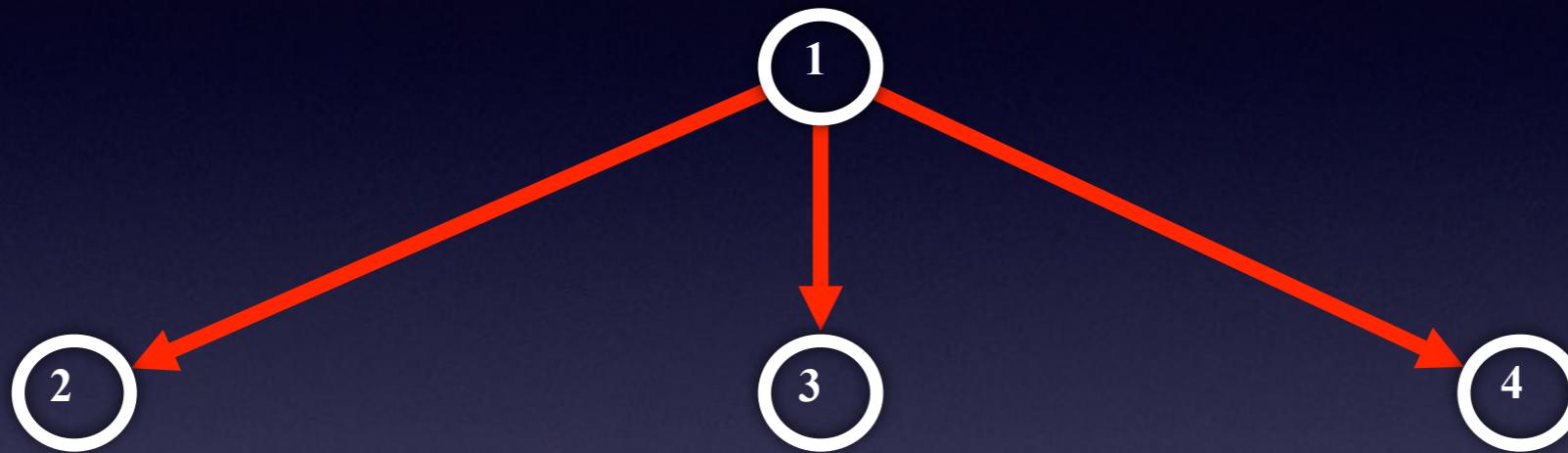
# Búsqueda en anchura



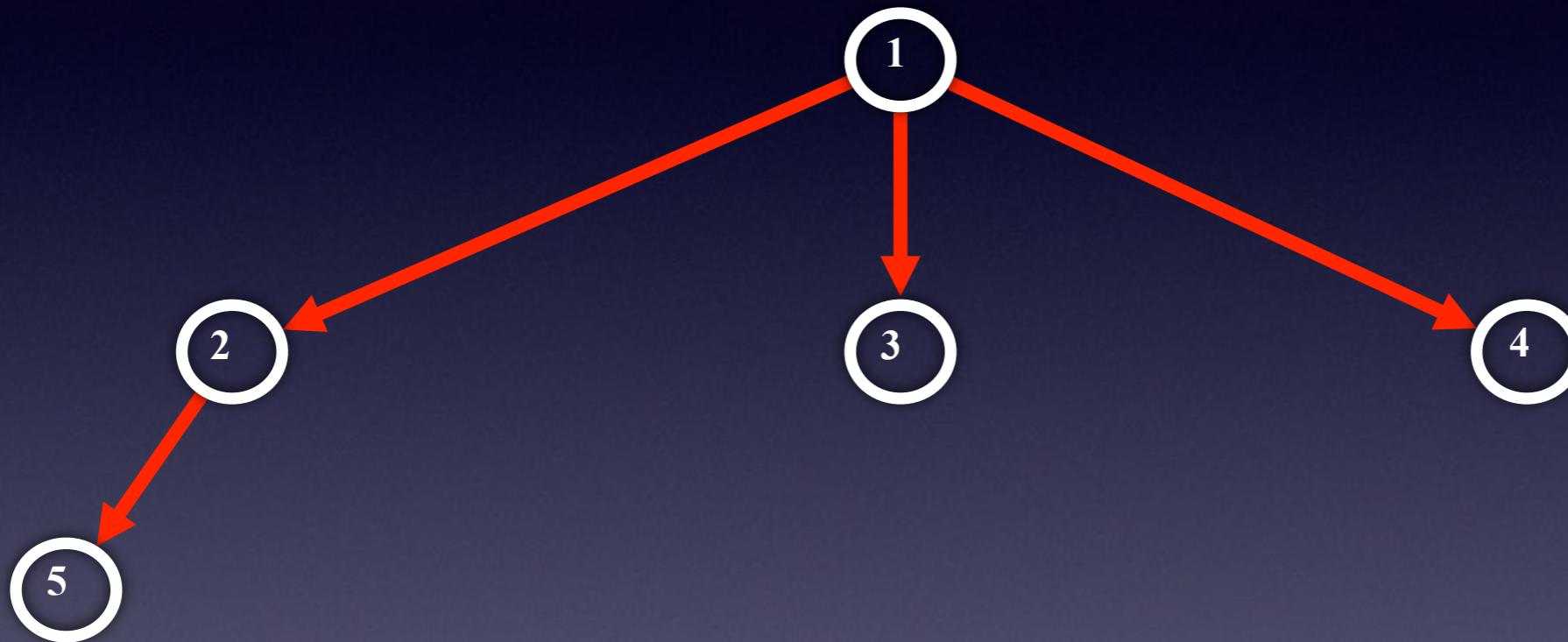
# Búsqueda en anchura



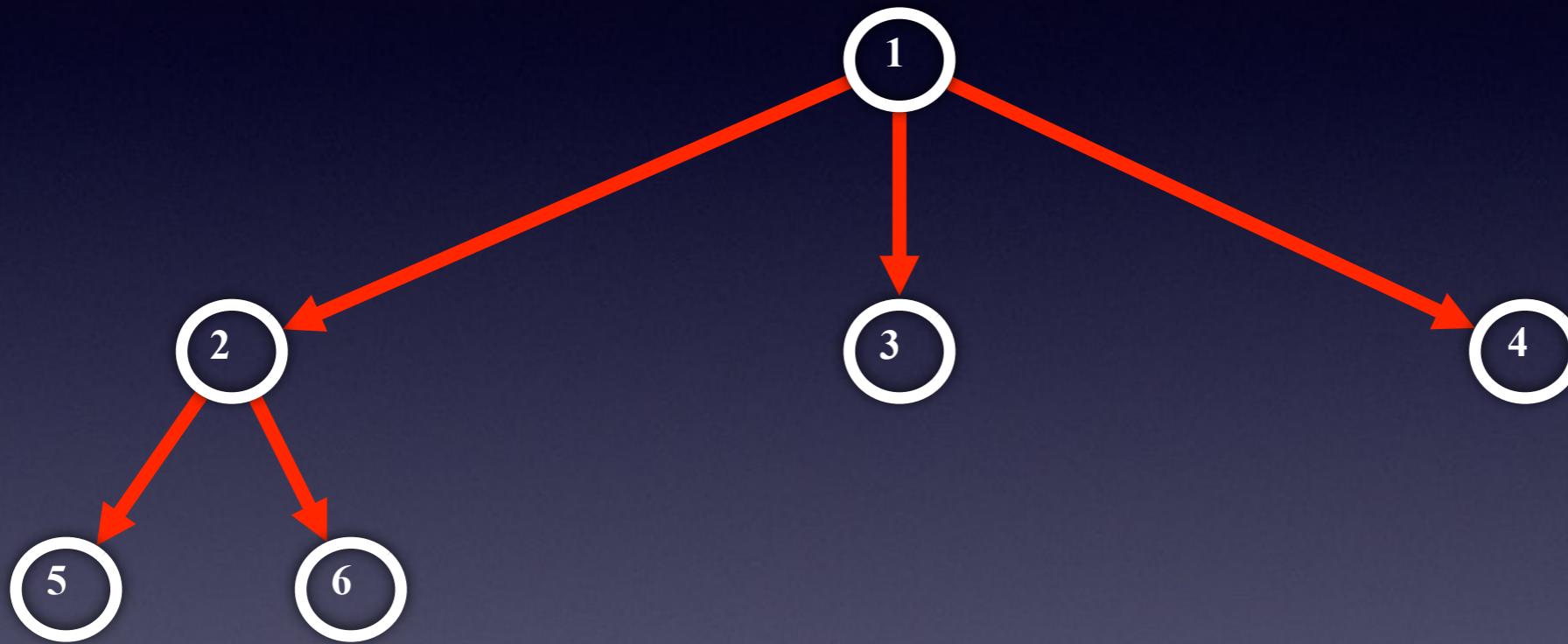
# Búsqueda en anchura



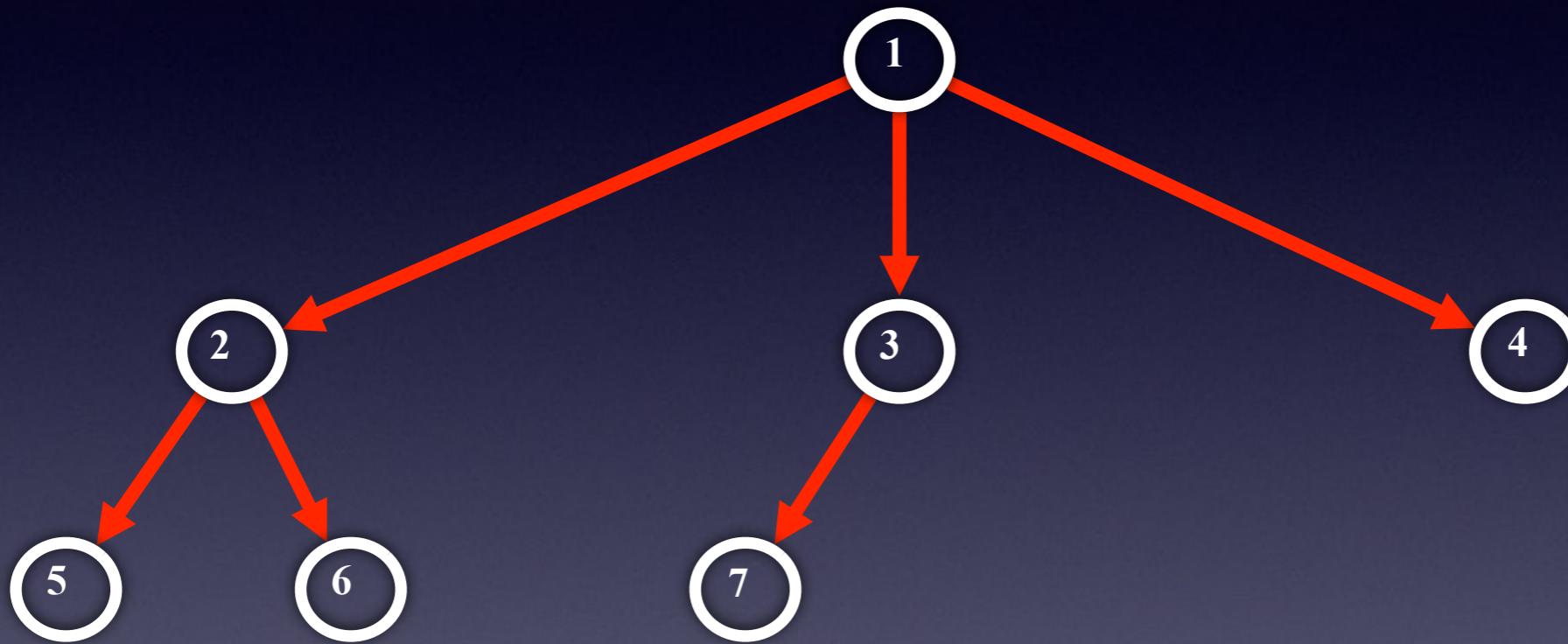
# Búsqueda en anchura



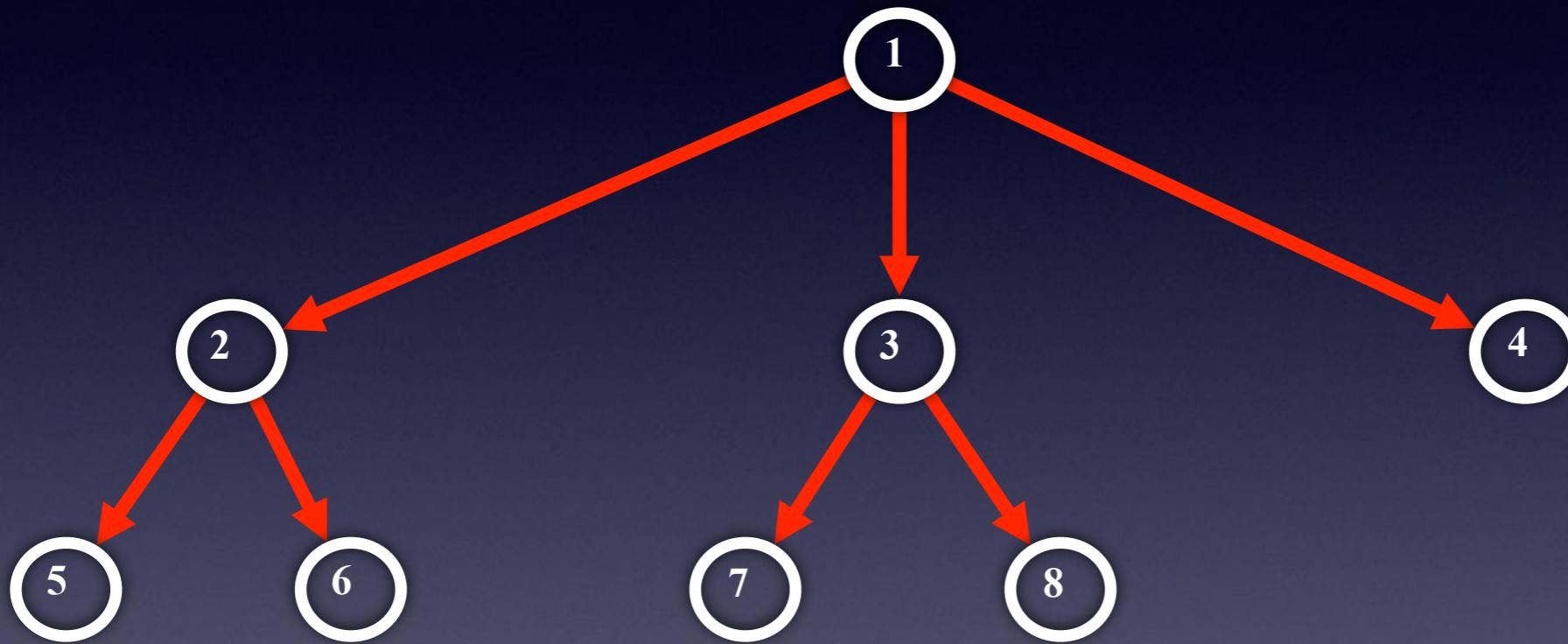
# Búsqueda en anchura



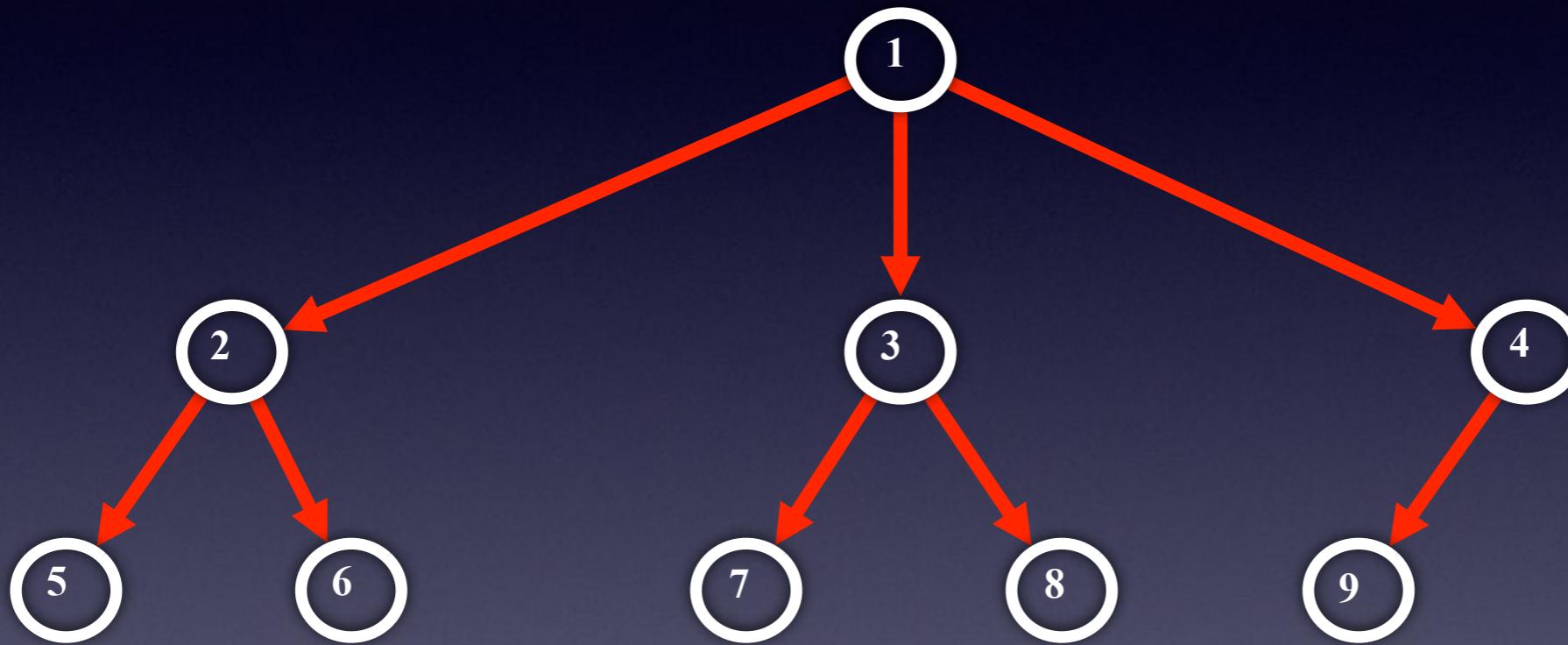
# Búsqueda en anchura



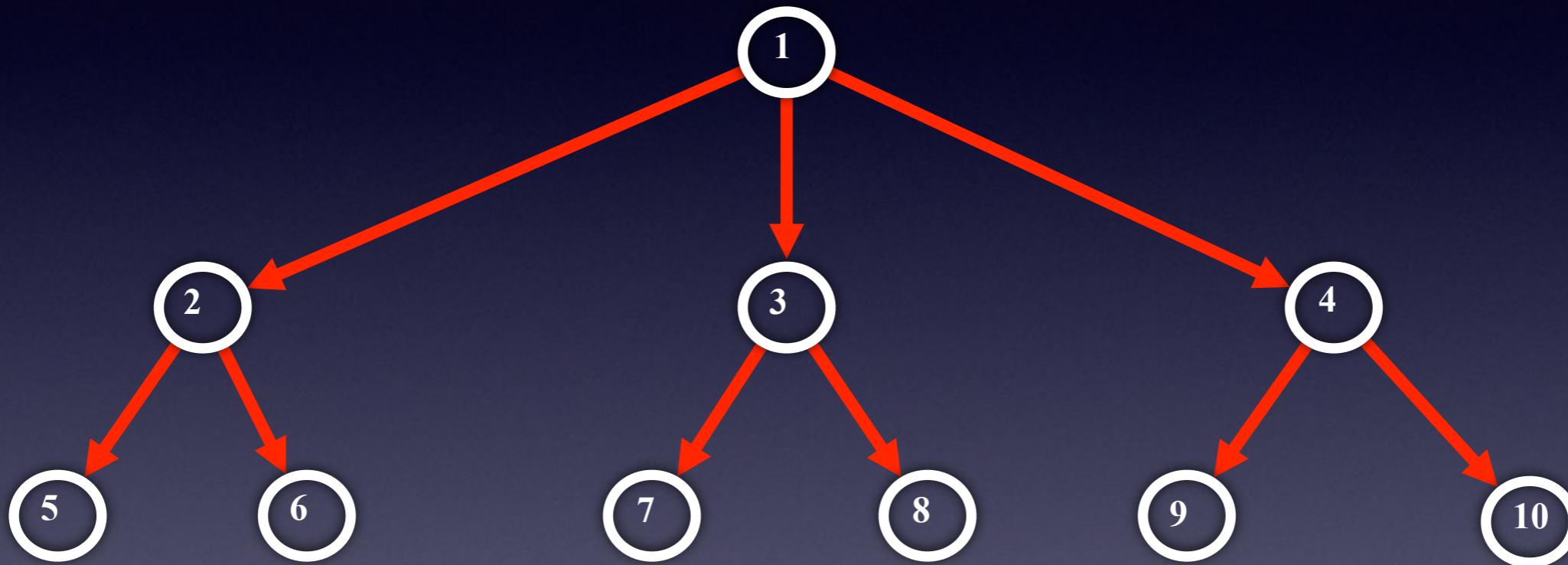
# Búsqueda en anchura



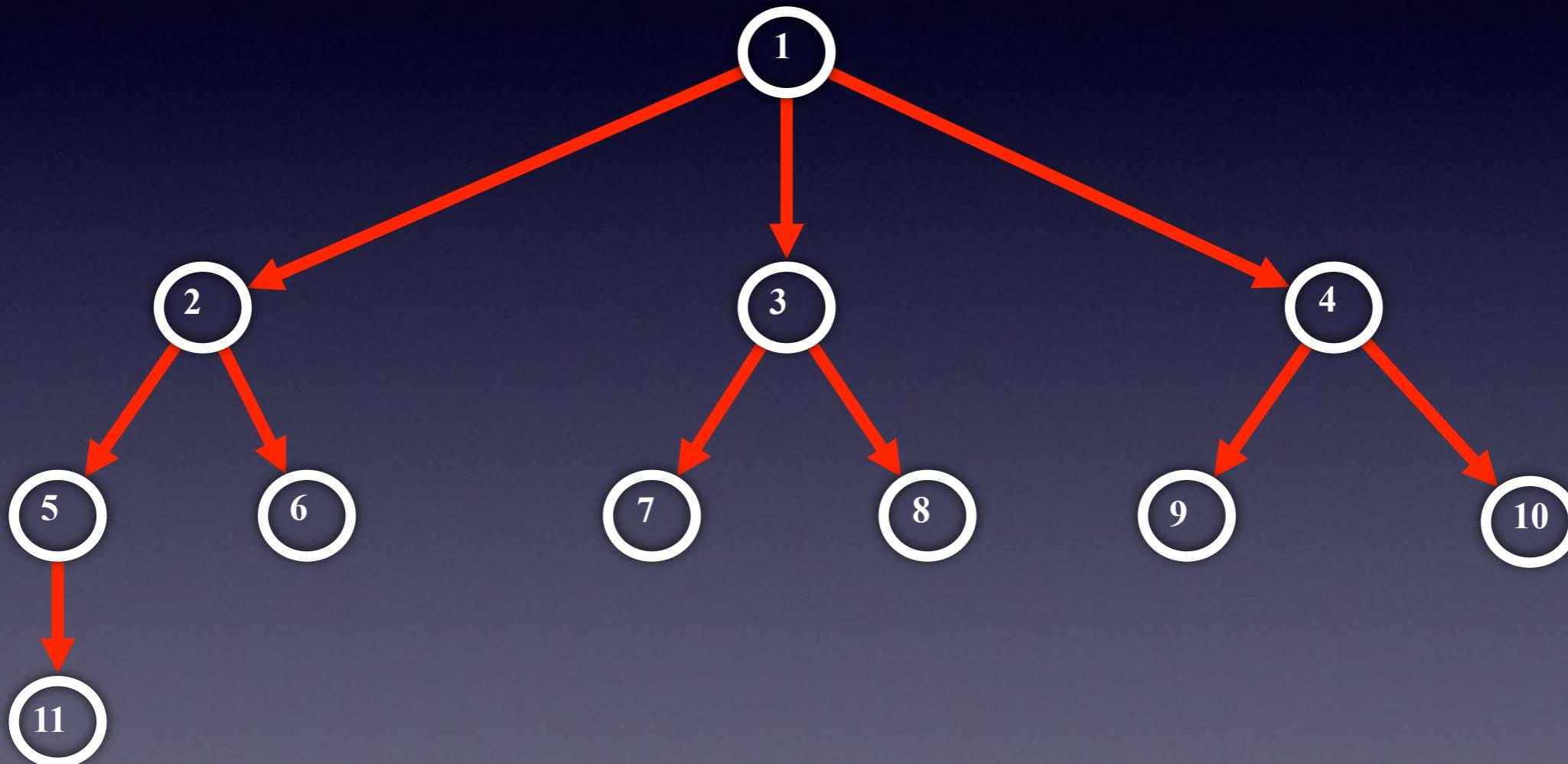
# Búsqueda en anchura



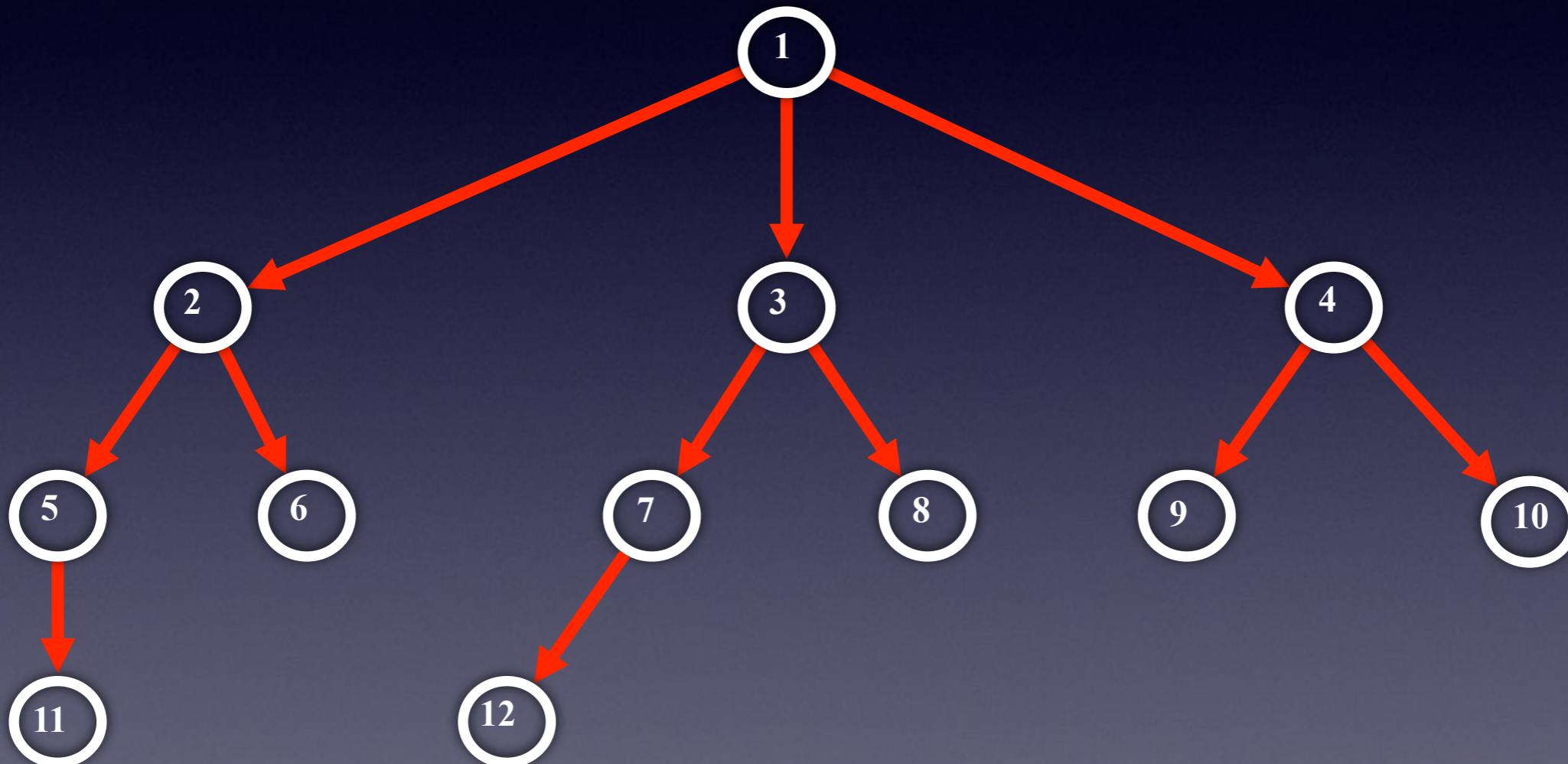
# Búsqueda en anchura



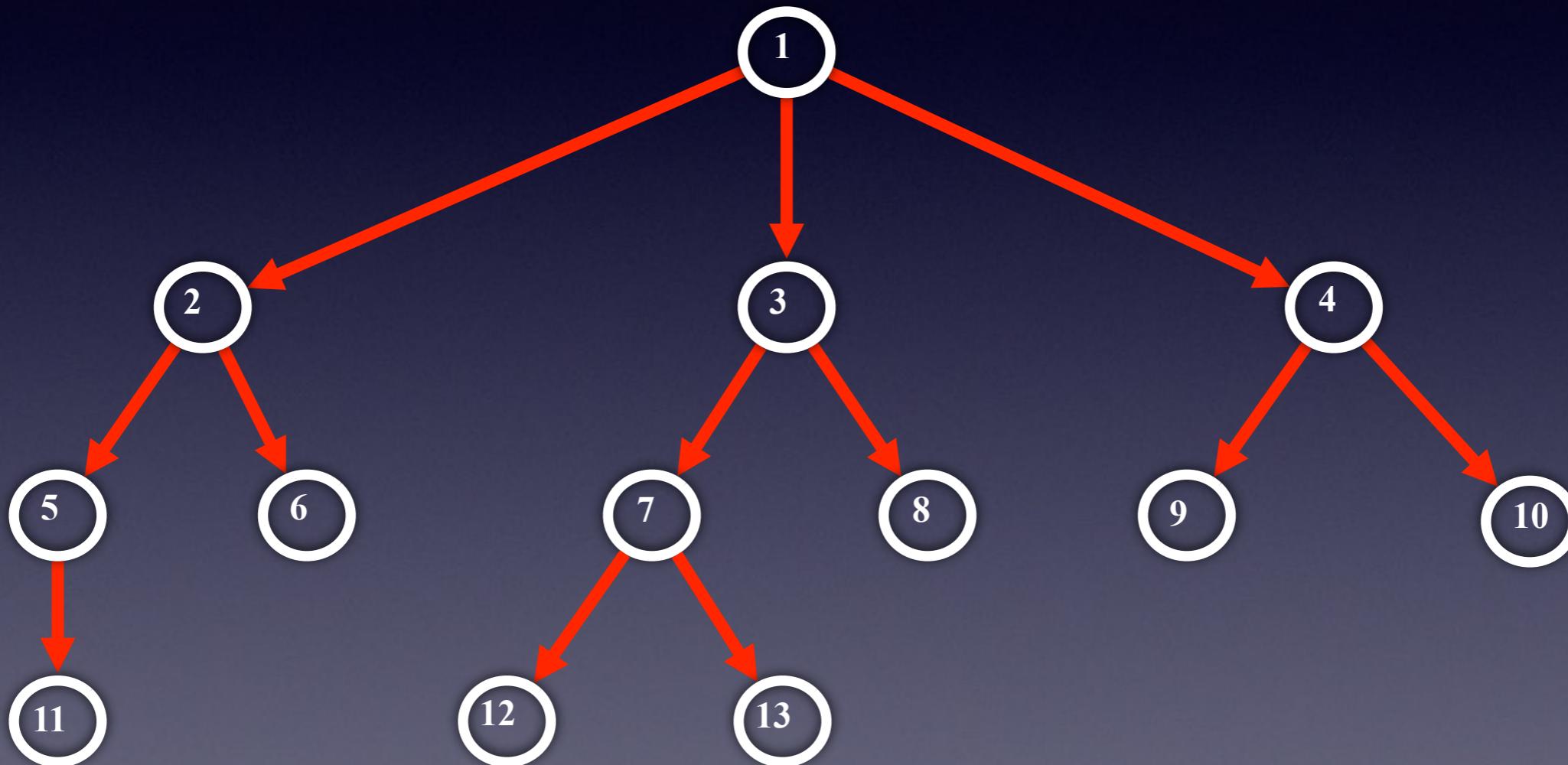
# Búsqueda en anchura



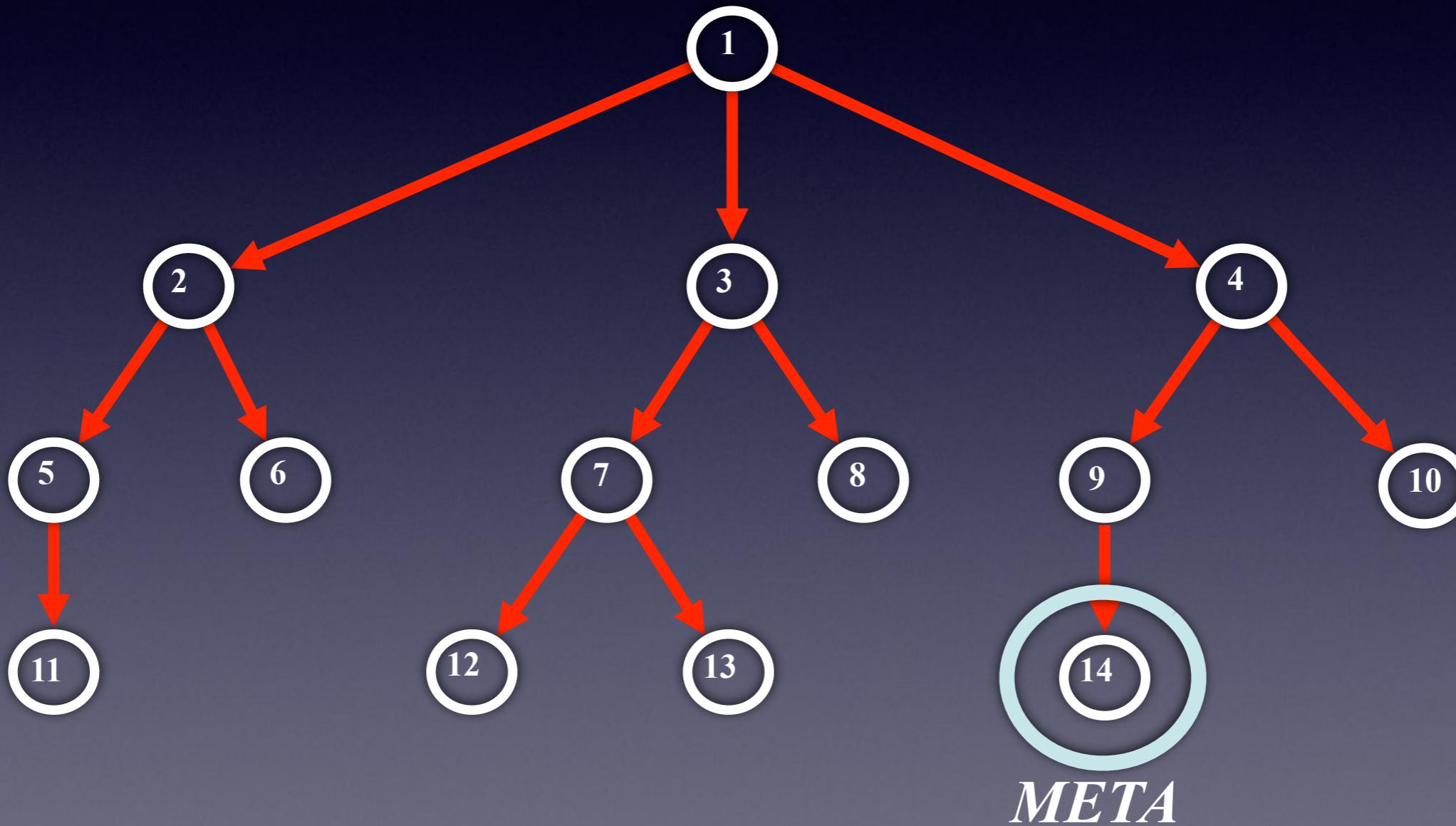
# Búsqueda en anchura



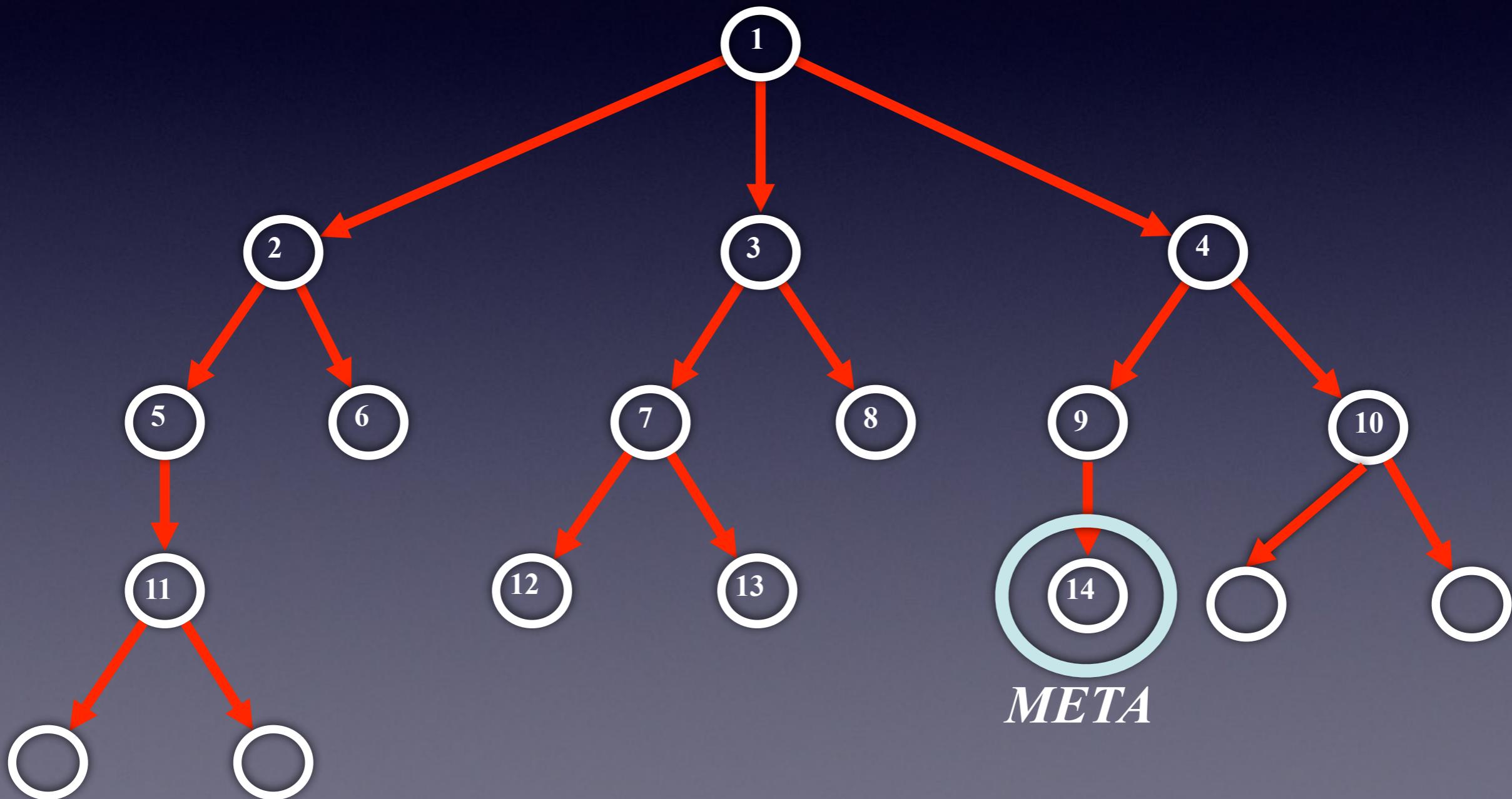
# Búsqueda en anchura



# Búsqueda en anchura



# Búsqueda en anchura



# Algoritmo: ANCHURA

```
public Lista<Accion> Busqueda(Nodo inicial,fin) {  
    Lista Abiertos = new Lista();  
    Lista Cerrados = new Lista();  
    Abiertos.Insertar(inicial);  
    while (Abiertos.Tamano()>0) {  
        Nodo Actual = Abiertos.SacarPrimero();  
        Cerrados.Insertar(Actual);  
        if (Actual == fin) {  
            return Recuperar_Camino(inic,Actual);  
        } else {  
            List<Nodo> sucesores = getSucesores(Actual);  
            sucesores = QuitarRepetidos(sucesores,Cerrados);  
            Abiertos.Insertar_AL_FINAL(sucesores);  
        }  
    }  
    return null;  
}
```

# Algoritmo: ANCHURA

```
public Lista<Accion> Busqueda(Nodo inicial,fin) {  
    Lista Abiertos = new Lista();  
    Lista Cerrados = new Lista();  
    Abiertos.Insertar(inicial);  
    while (Abiertos.Tamano()>0) {  
        Nodo Actual = Abiertos.SacarPrimero();  
        Cerrados.Insertar(Actual);  
        if (Actual == fin) {  
            return Recuperar_Camino(inic,Actual);  
        } else {  
            List<Nodo> sucesores = getSucesores(Actual);  
            sucesores = QuitarRepeticiones(sucesores,Cerrados);  
            Abiertos.Insertar_AL_FINAL(sucesores);  
        }  
    }  
    return null;  
}
```

# PROFUNDIDAD

# Búsqueda en Profundidad

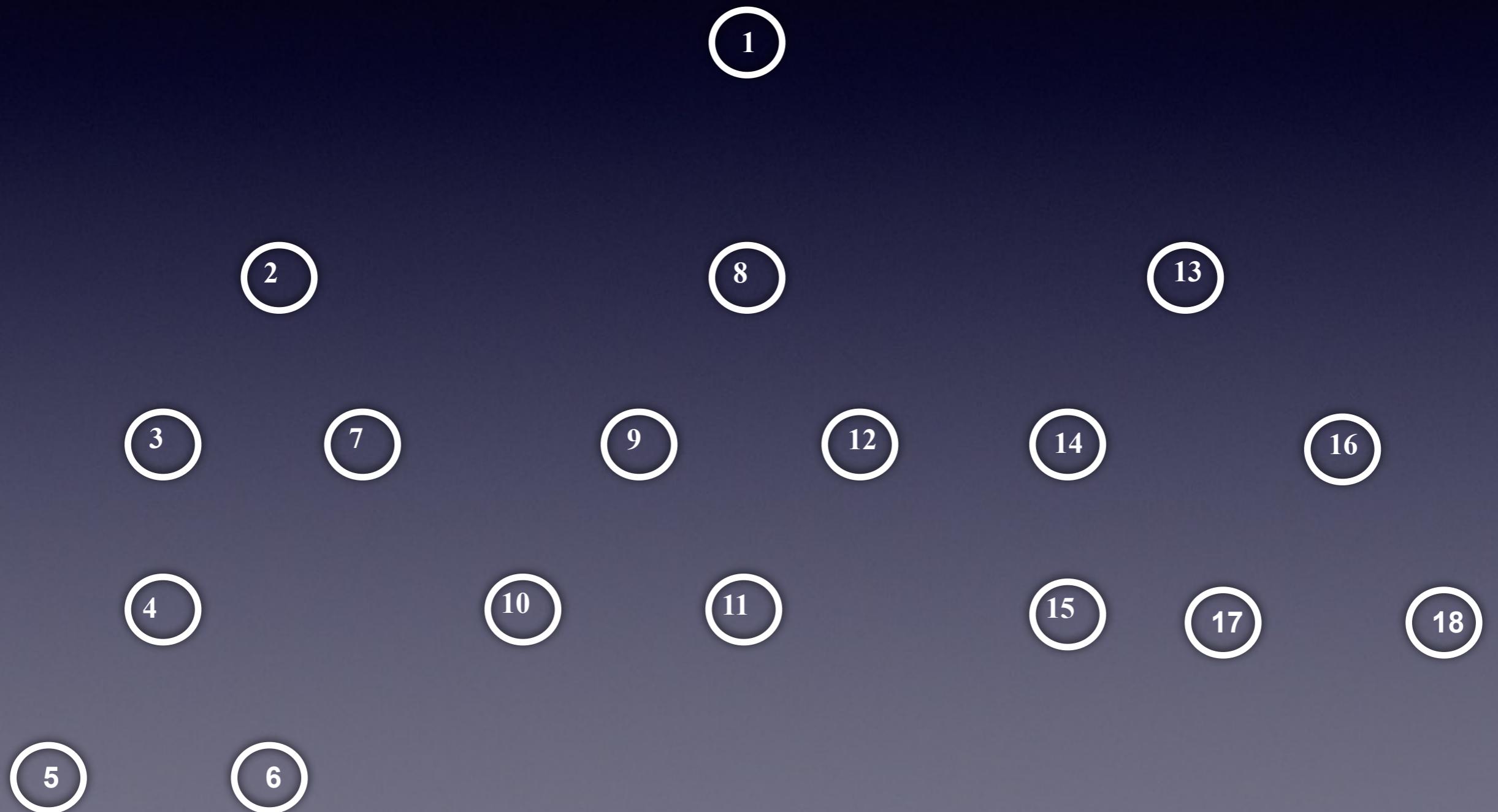
No expandir nodos de nivel  $n$  si hay todavía algún hijo de nivel  $> n$  pendiente de considerar

- Los nodos se visitan y generan buscando los nodos a mayor profundidad y retrocediendo cuando no se encuentran nodos sucesores.
- La estructura para los nodos abiertos es una pila (LIFO).
- Problema: profundidades grandes

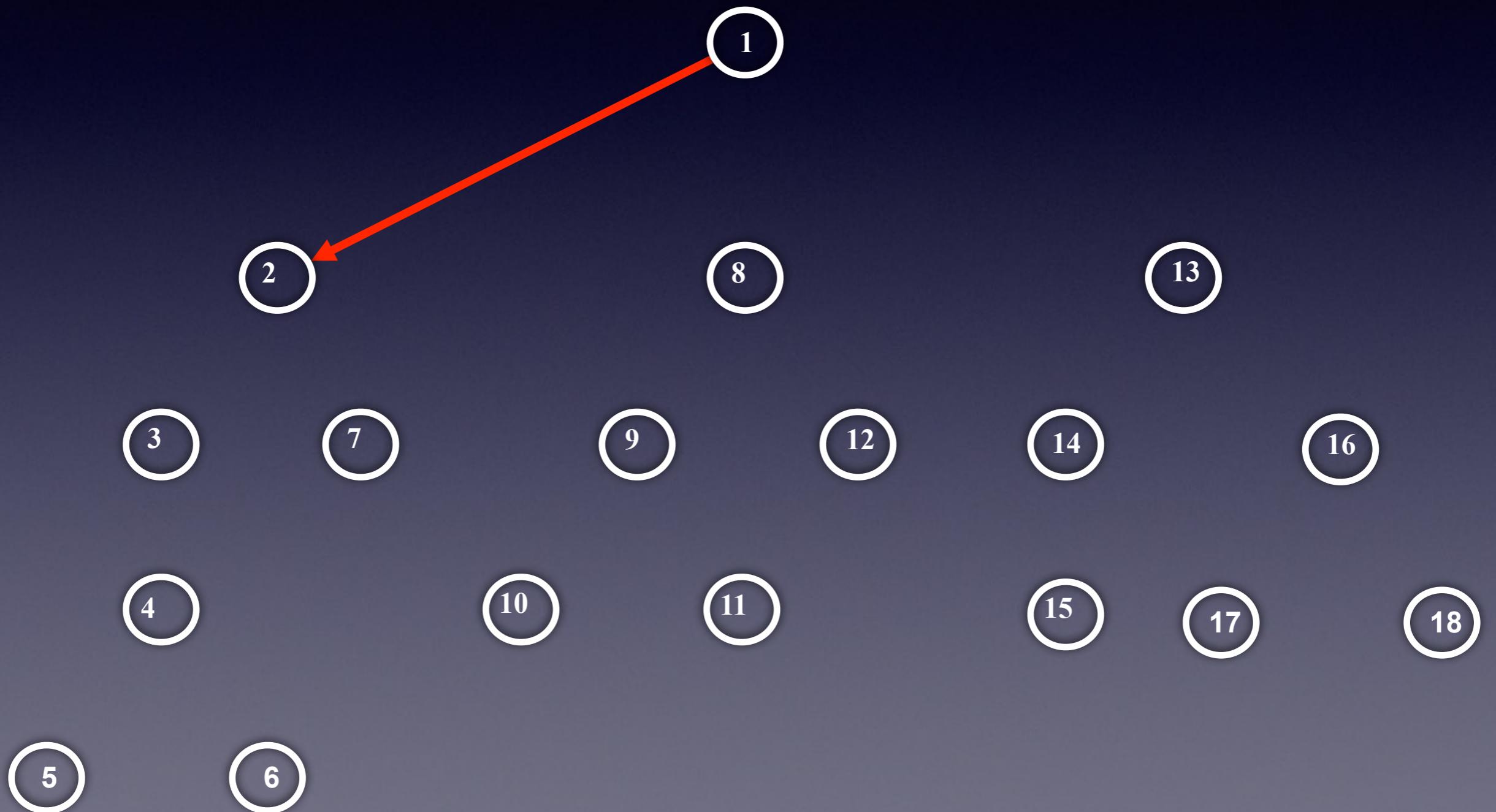
# Búsqueda en Profundidad

- Completitud: encuentra la solución siempre que no encuentre ramas infinitas antes de la solución
- Complejidad temporal: exponencial respecto a la profundidad del límite de exploración  $O(r^m)$ .
- Complejidad espacial: sin control de repetidos es lineal:  $O(r m)$ .
- Optimización: no se garantiza que la solución sea óptima.

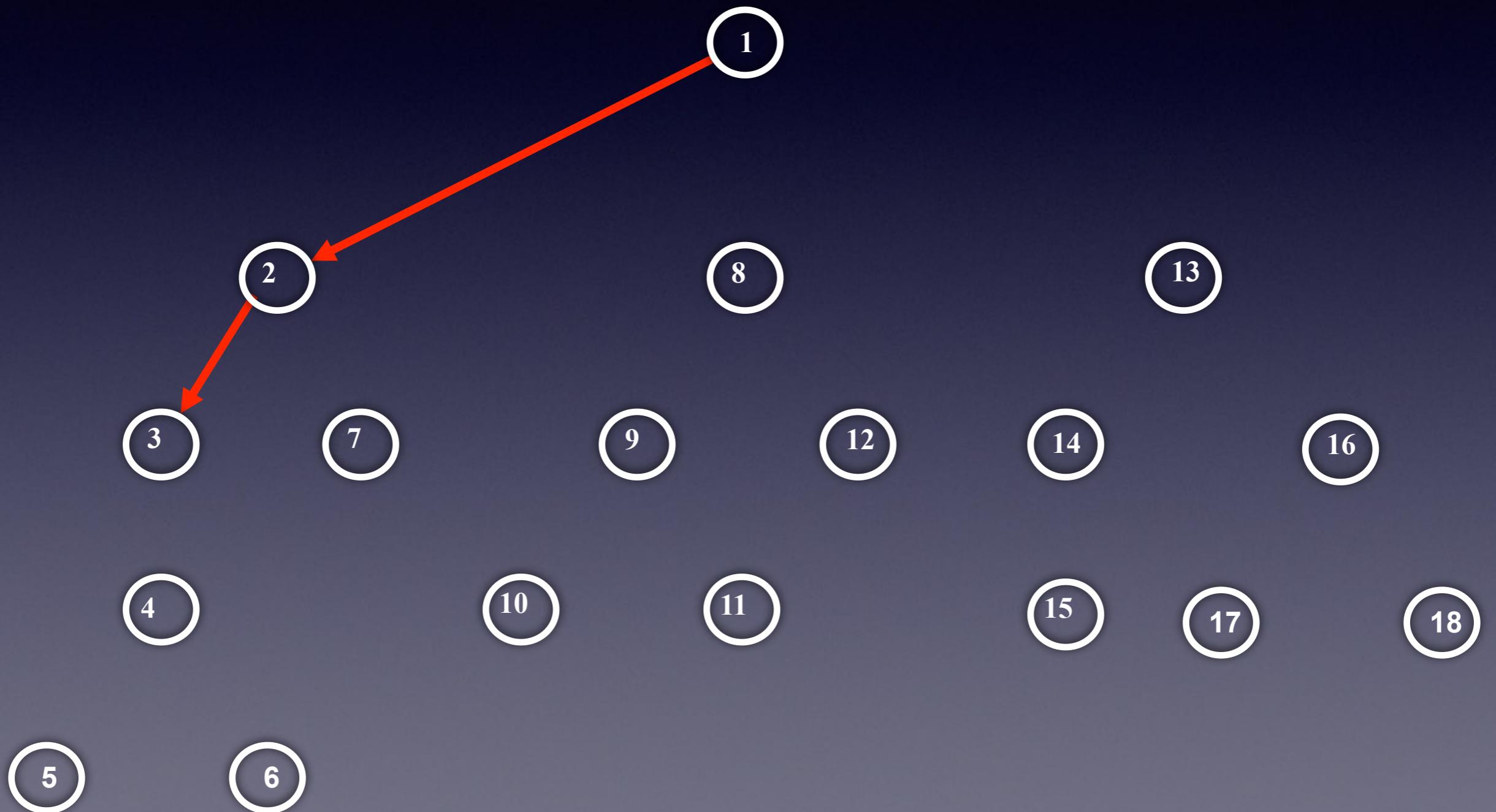
# Búsqueda en Profundidad



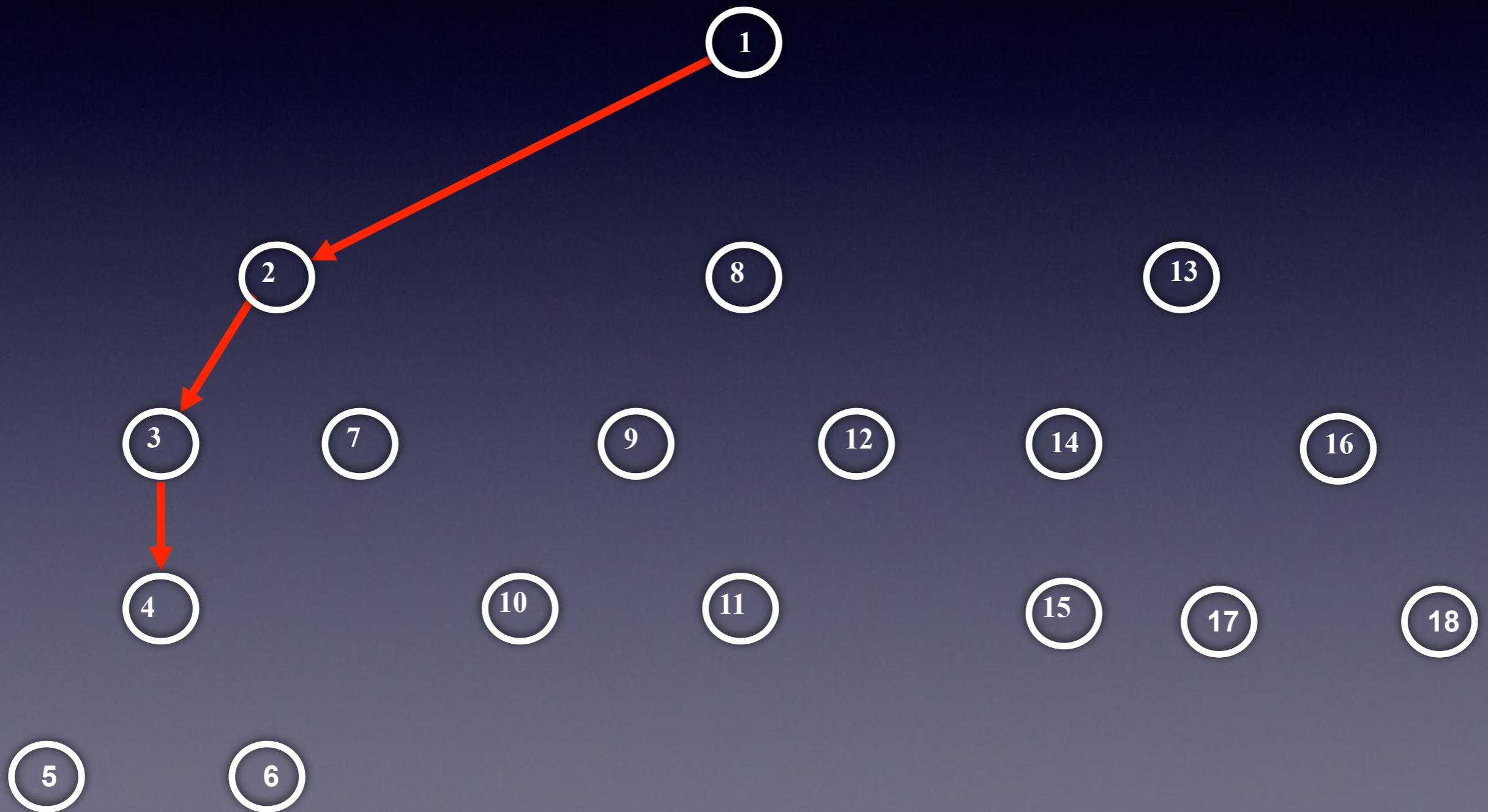
# Búsqueda en Profundidad



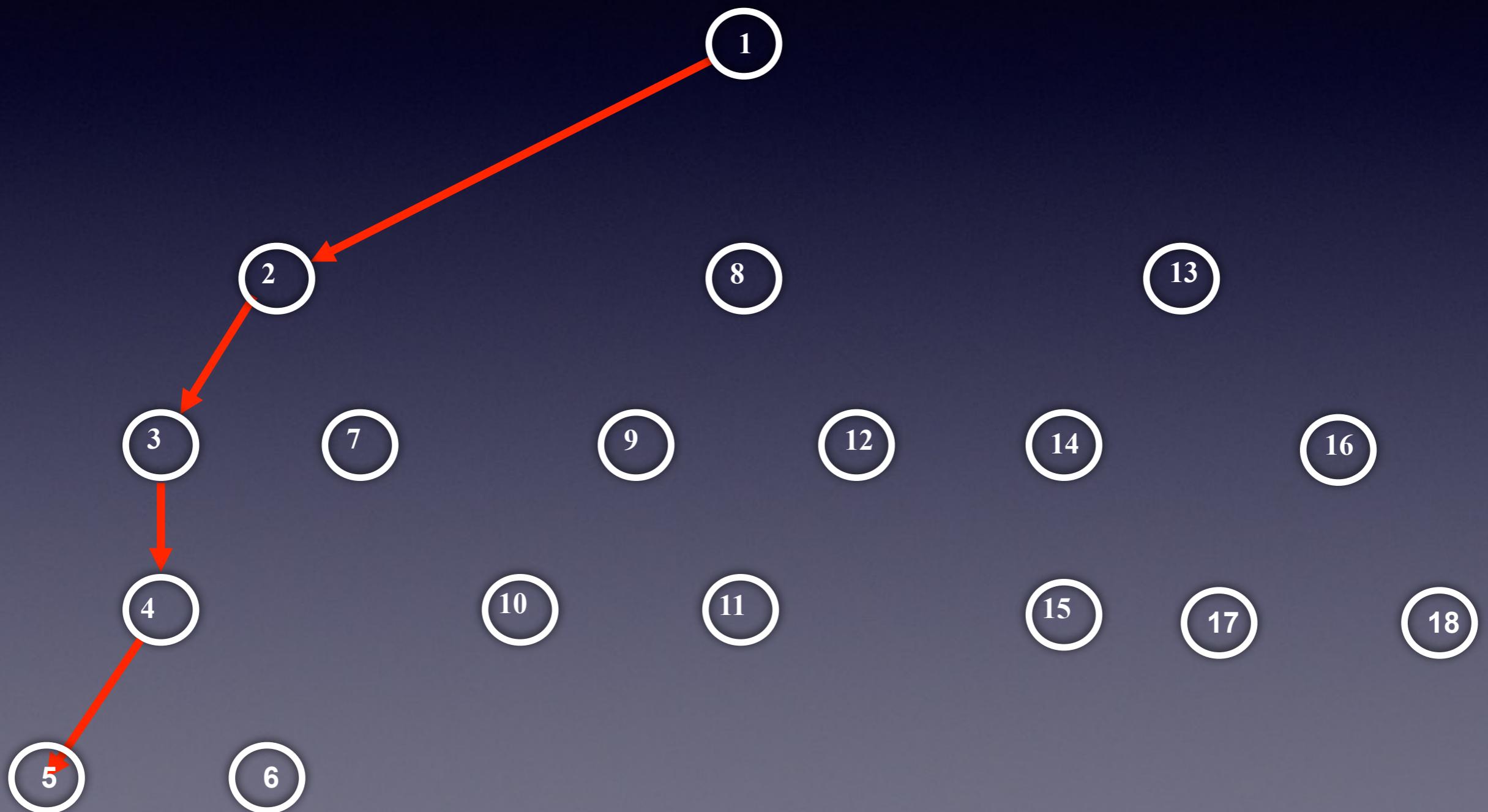
# Búsqueda en Profundidad



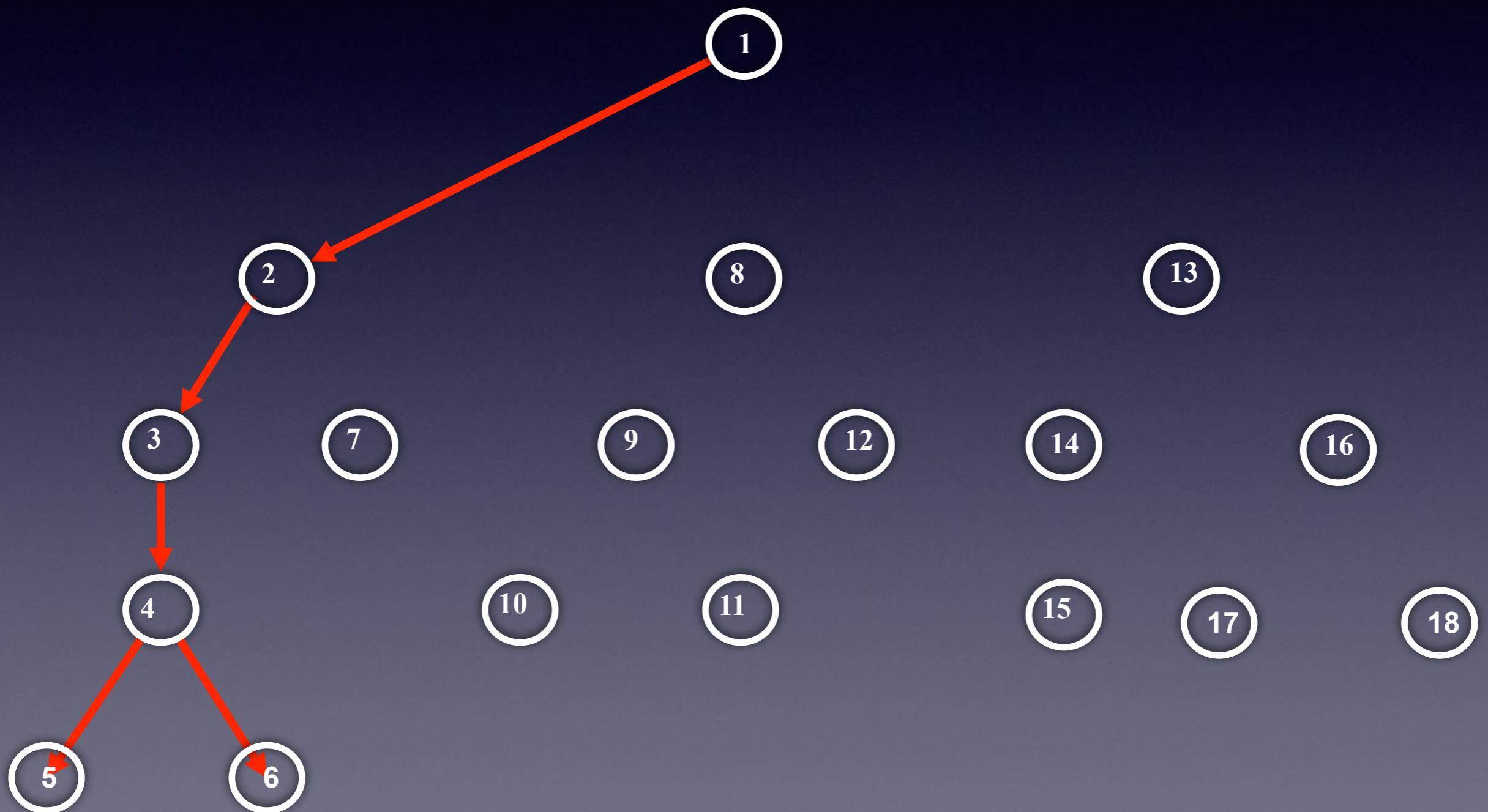
# Búsqueda en Profundidad



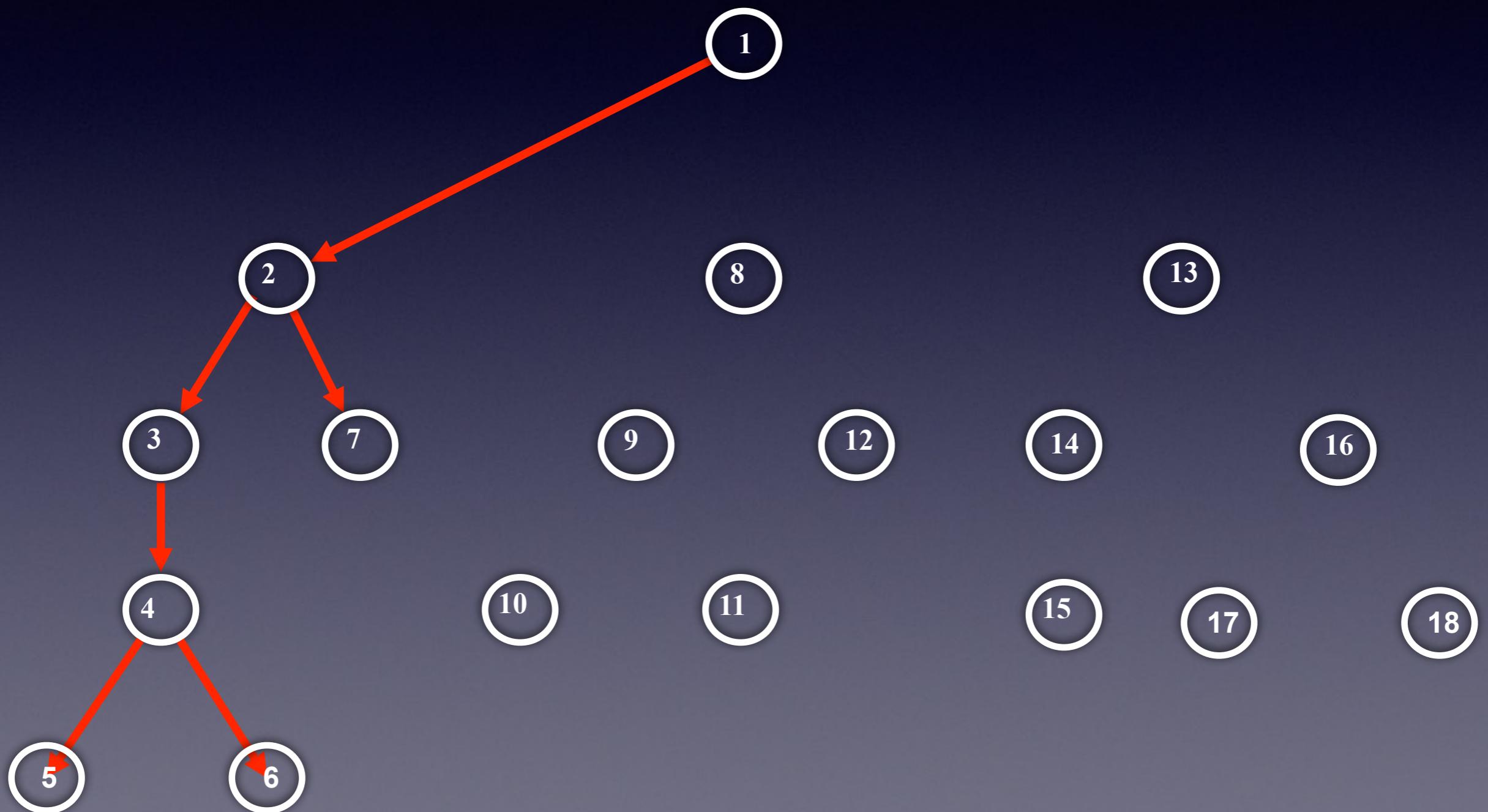
# Búsqueda en Profundidad



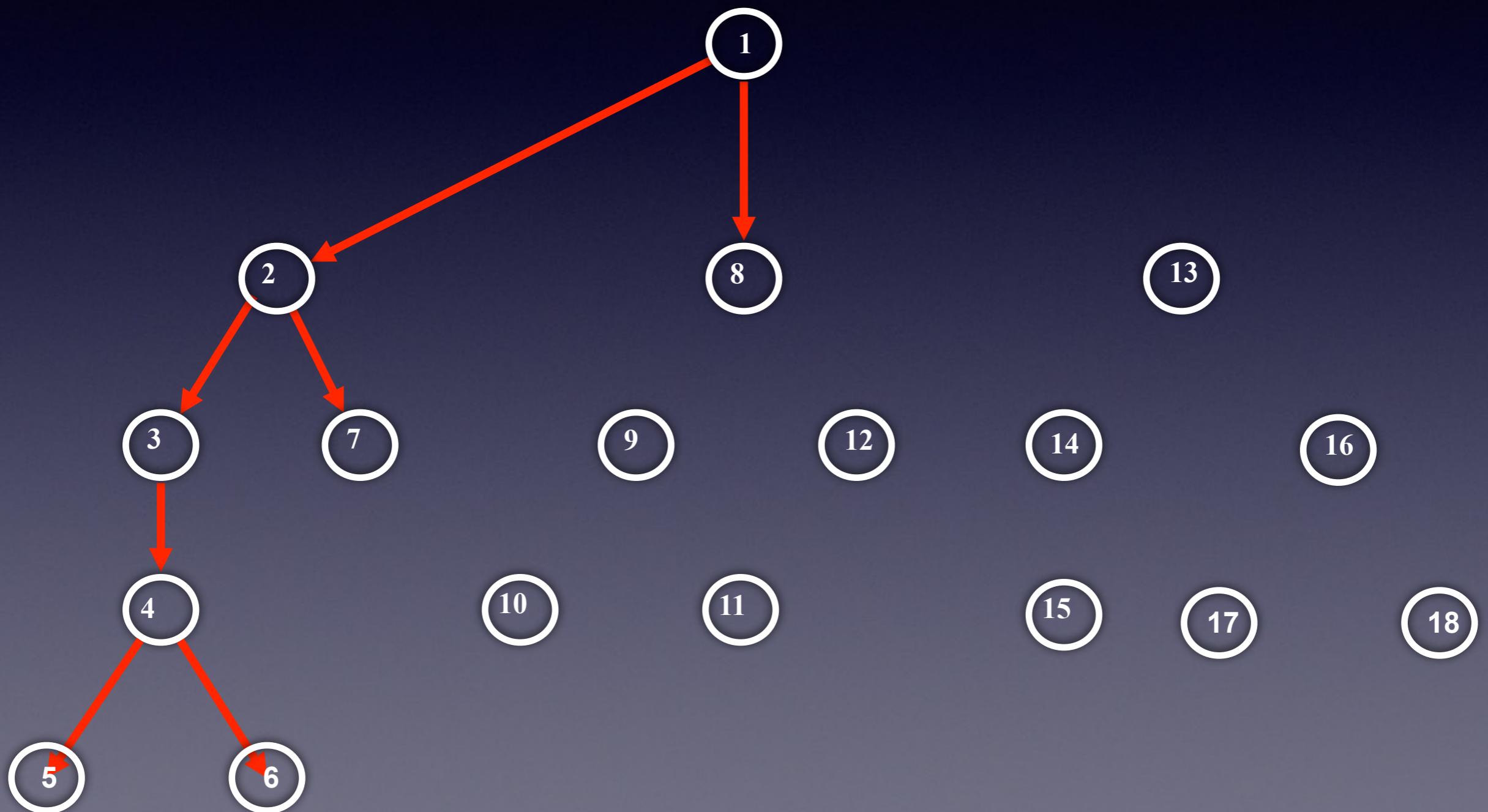
# Búsqueda en Profundidad



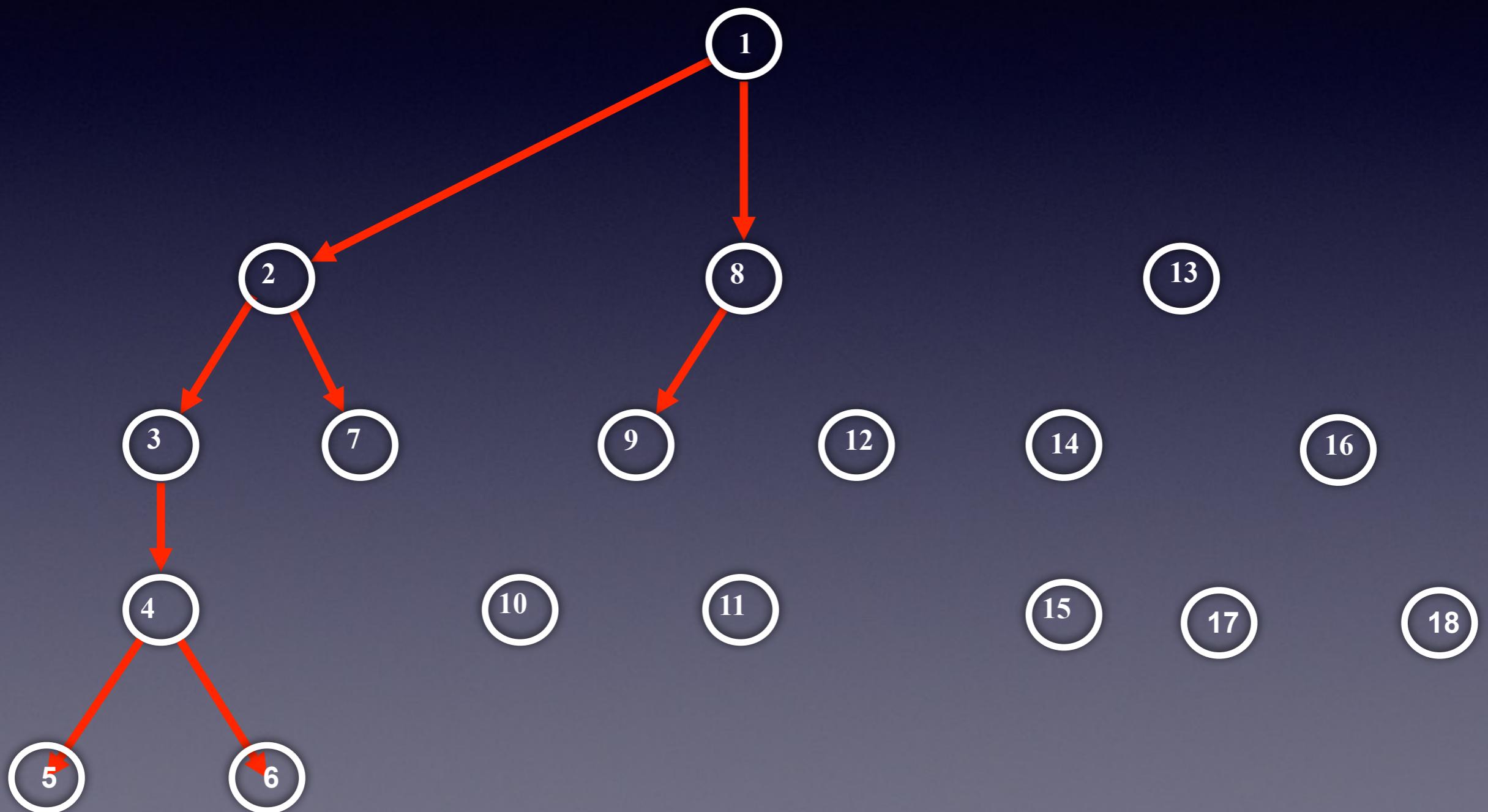
# Búsqueda en Profundidad



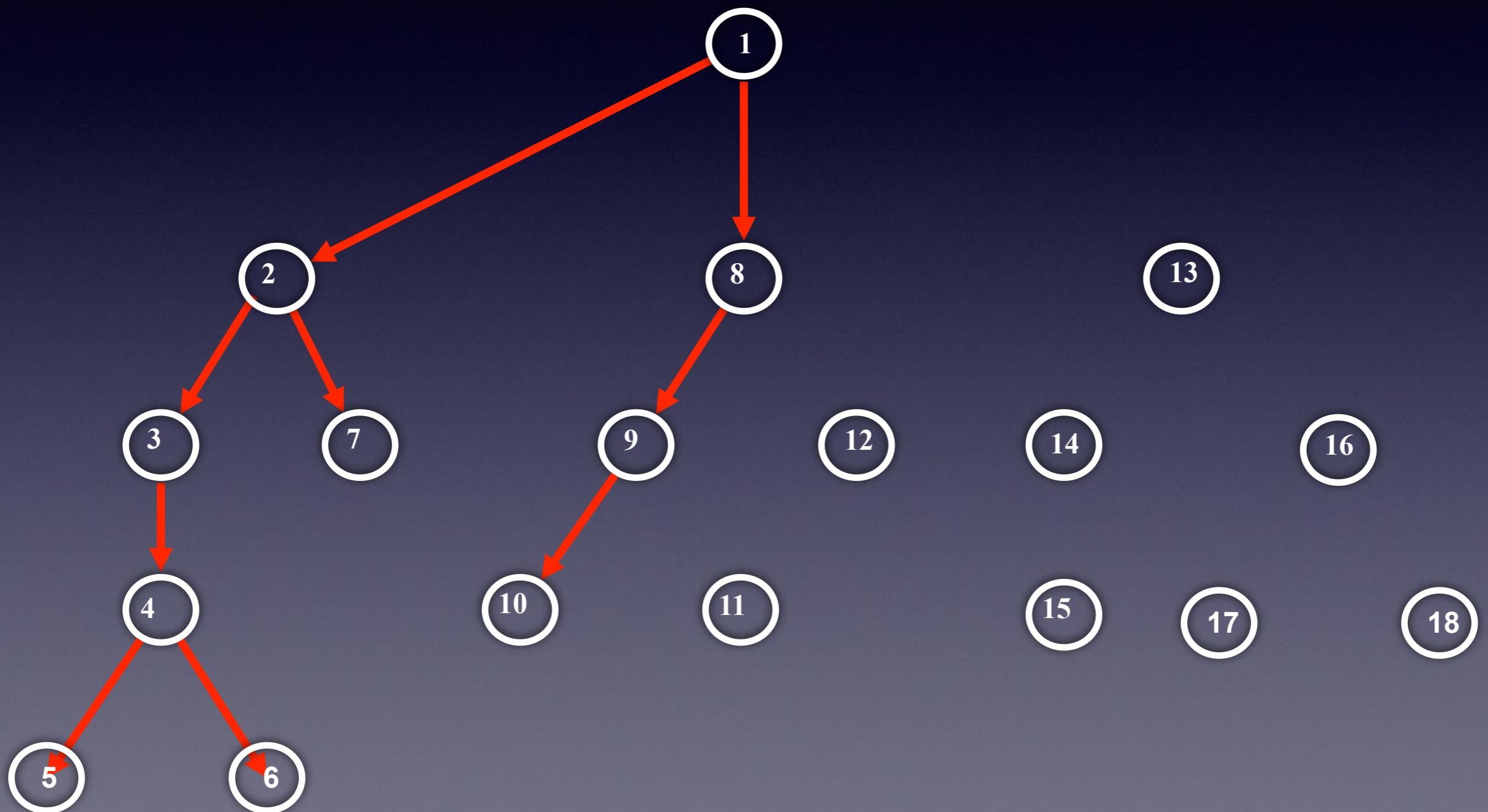
# Búsqueda en Profundidad



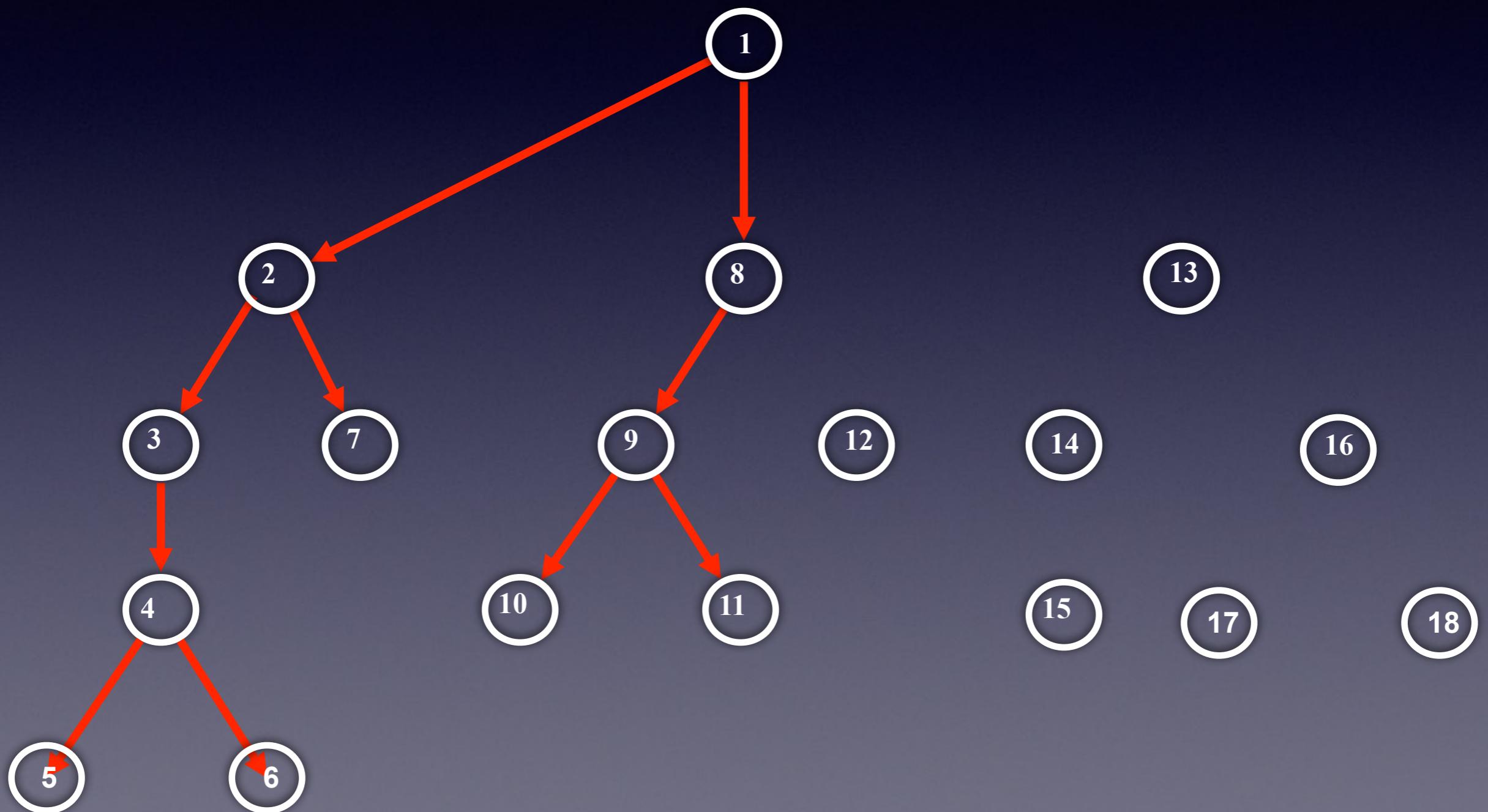
# Búsqueda en Profundidad



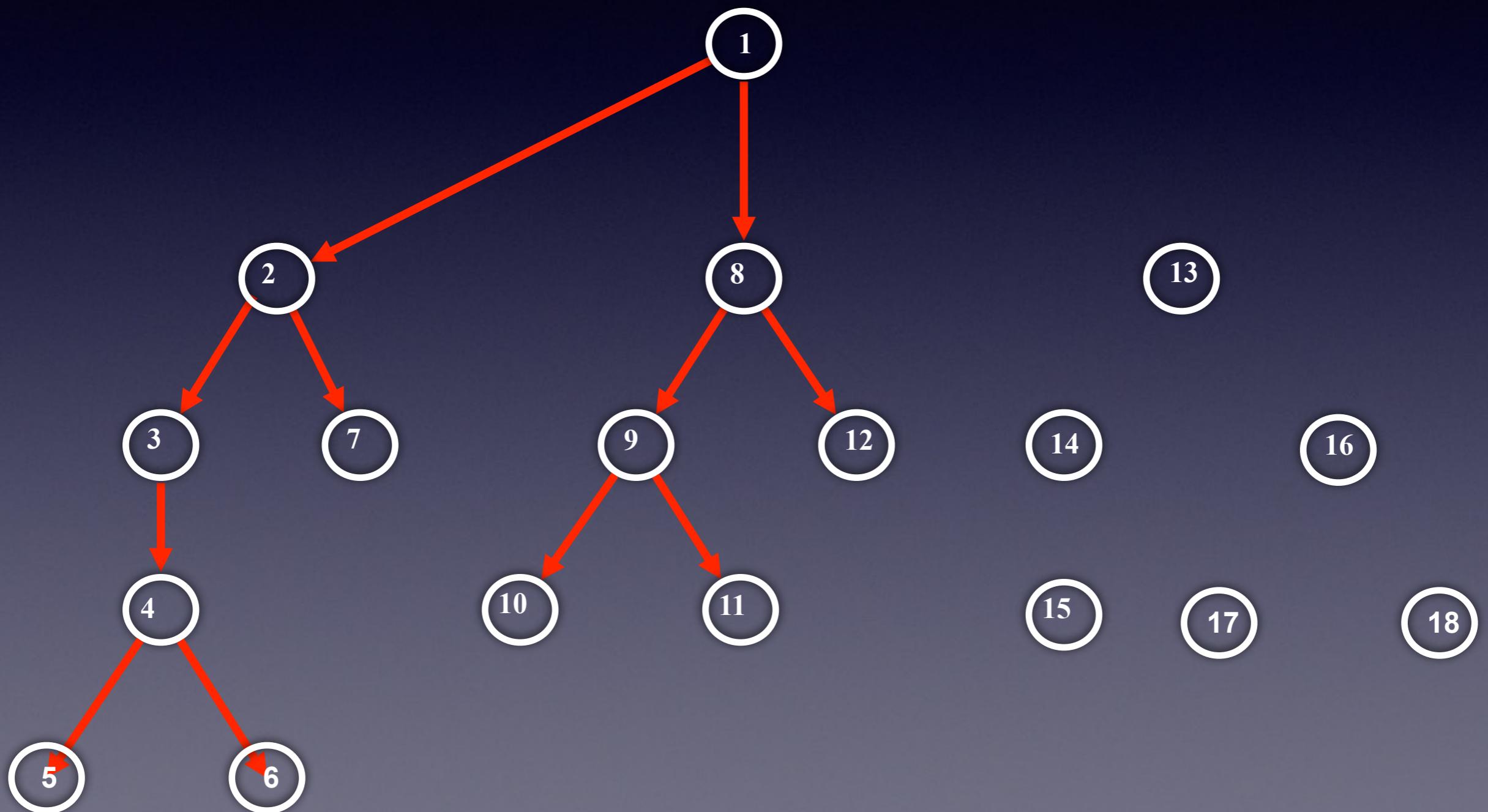
# Búsqueda en Profundidad



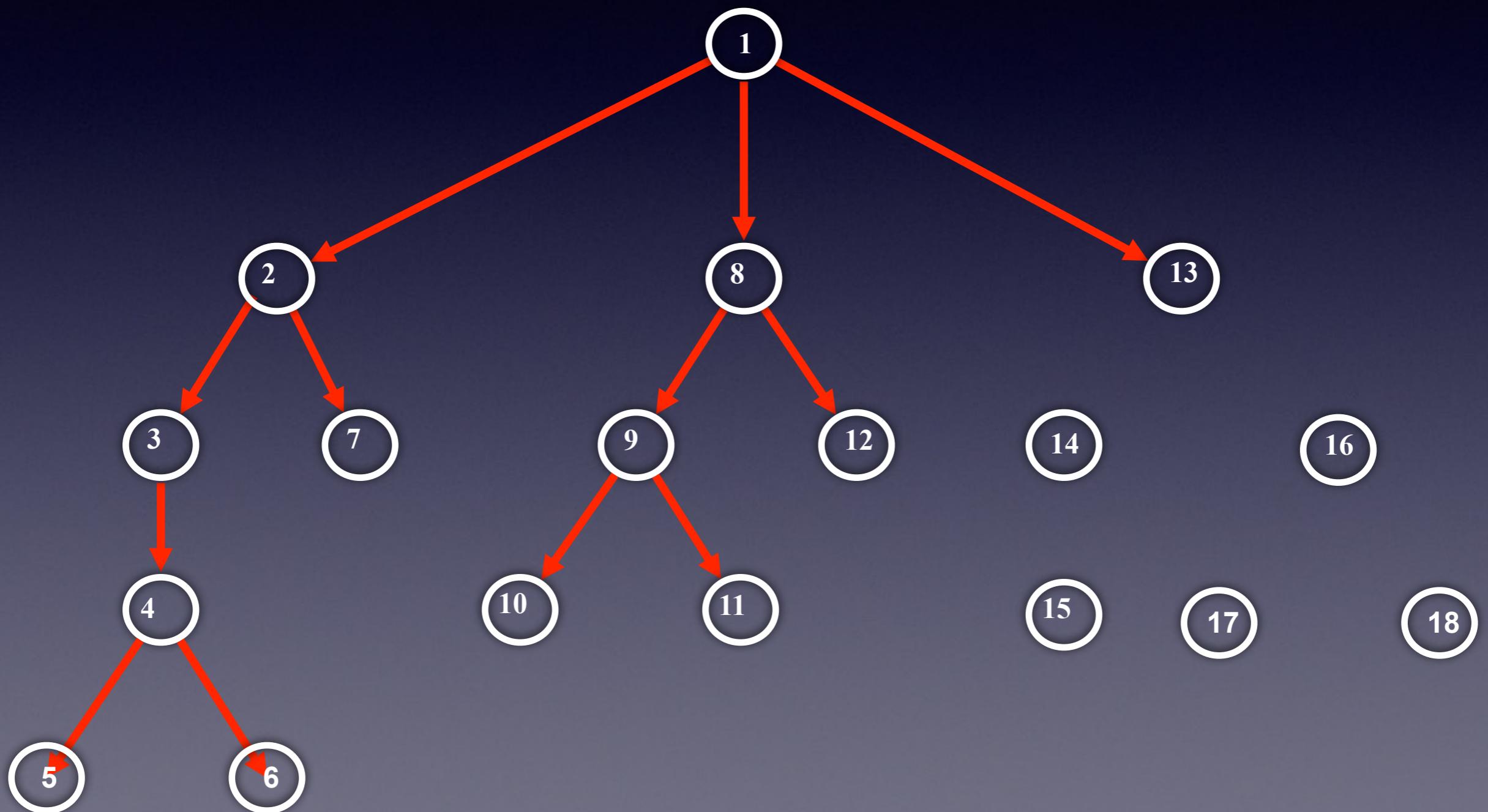
# Búsqueda en Profundidad



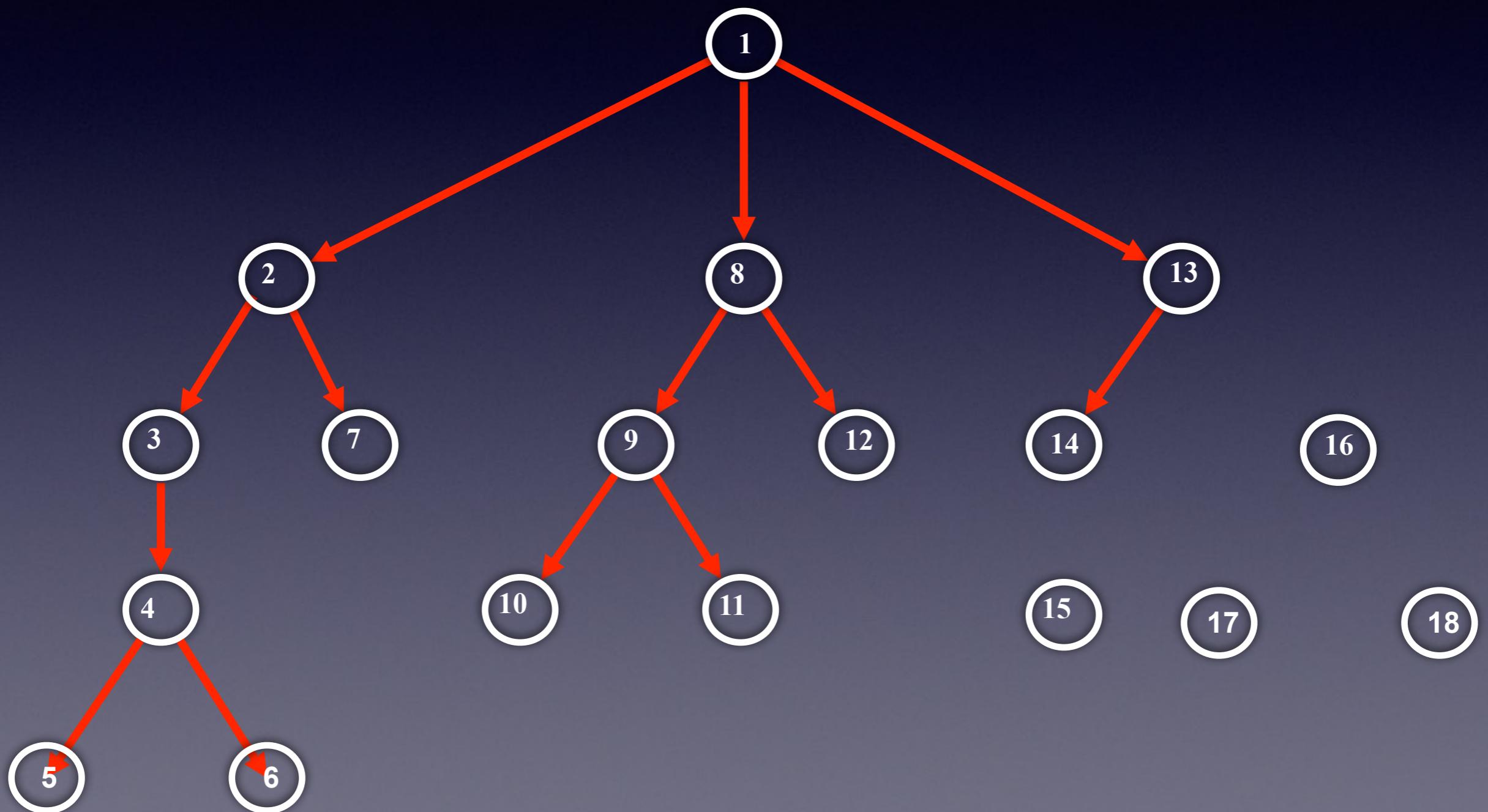
# Búsqueda en Profundidad



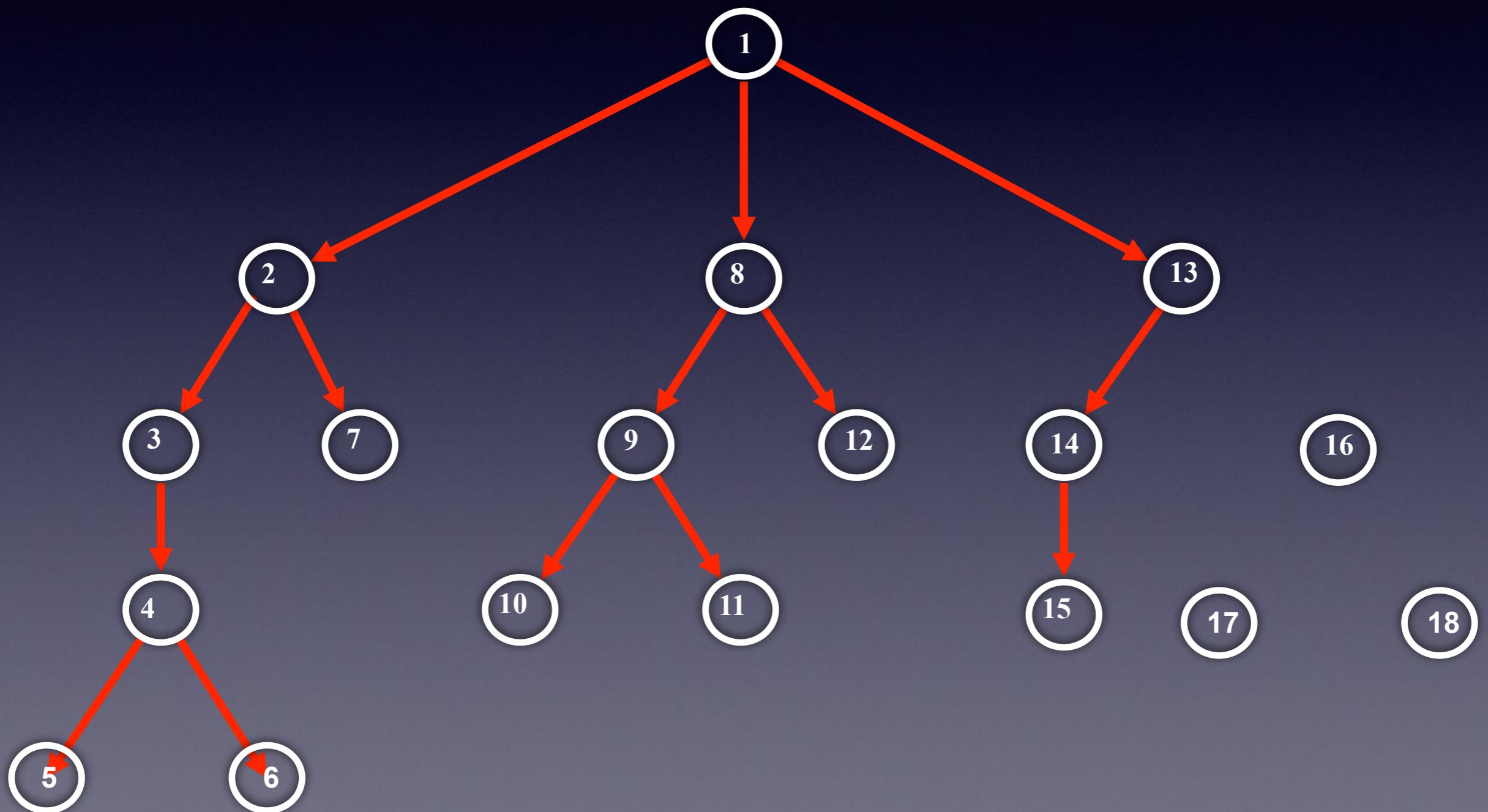
# Búsqueda en Profundidad



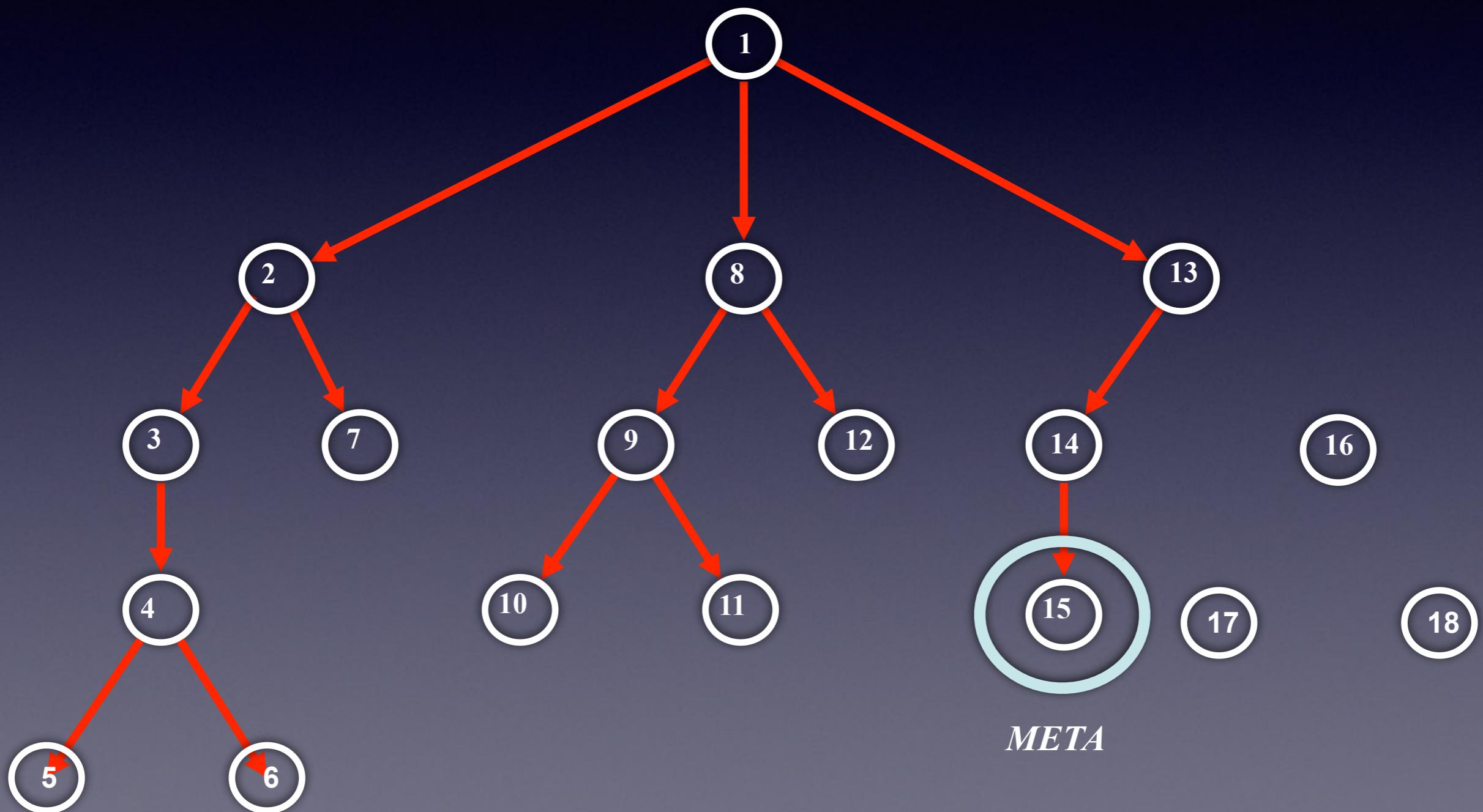
# Búsqueda en Profundidad



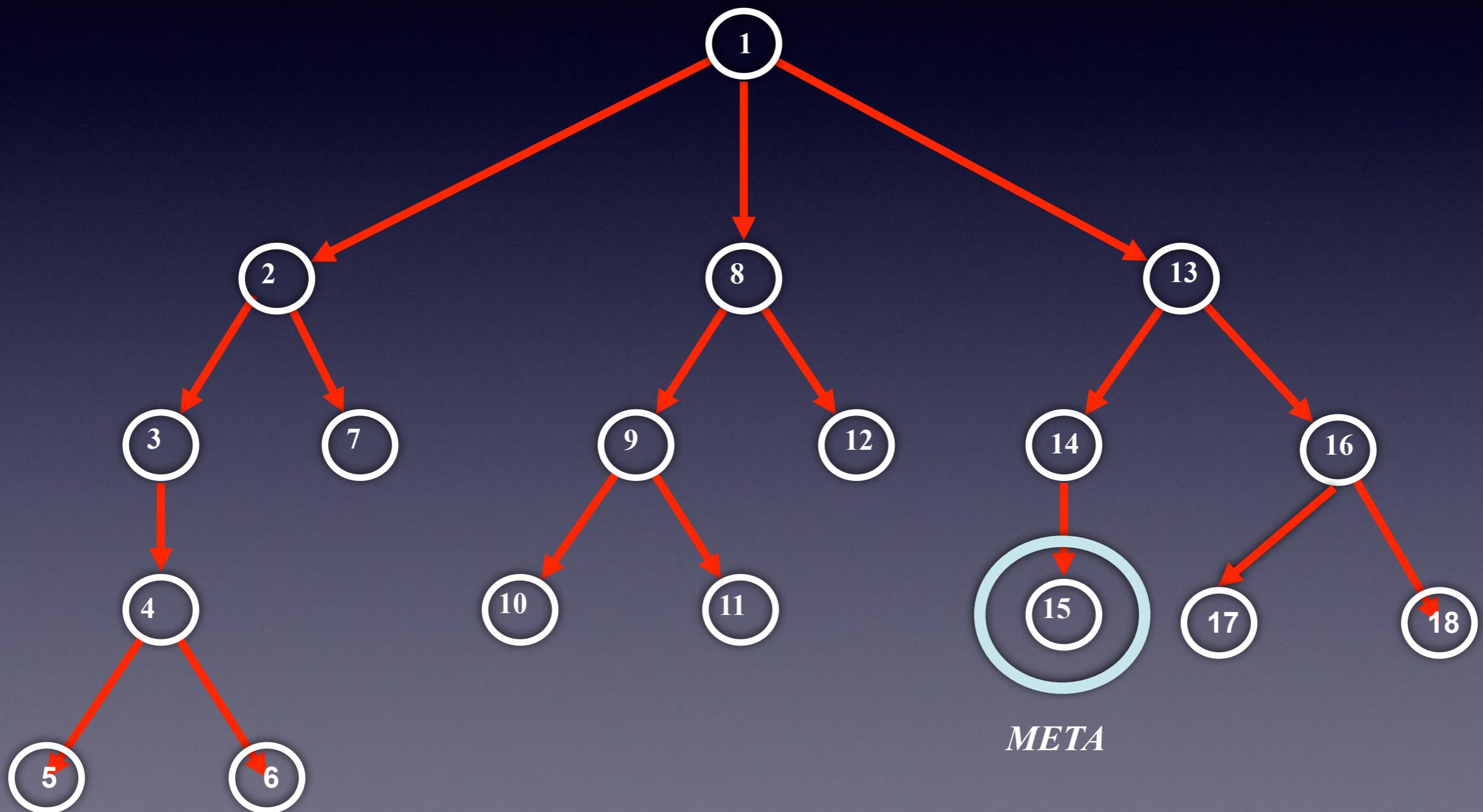
# Búsqueda en Profundidad



# Búsqueda en Profundidad



# Búsqueda en Profundidad



# Algoritmo: PROFUNDIDAD

```
public Lista<Accion> Busqueda(Nodo inicial,fin) {  
    Lista Abiertos = new Lista();  
    Lista Cerrados = new Lista();  
    Abiertos.Insertar(inicial);  
    while (Abiertos.Tamano()>0) {  
        Nodo Actual = Abiertos.SacarPrimero();  
        Cerrados.Insertar(Actual);  
        if (Actual == fin) {  
            return Recuperar_Camino(inic,Actual);  
        } else {  
            List<Nodo> sucesores = getSucesores(Actual);  
            sucesores = QuitarRepetidos(sucesores,Cerrados);  
            Abiertos.Insertar_PRINCIPIO(sucesores);  
        }  
    }  
    return null;  
}
```

# Algoritmo: PROFUNDIDAD

```
public Lista<Accion> Busqueda(Nodo inicial,fin) {  
    Lista Abiertos = new Lista();  
    Lista Cerrados = new Lista();  
    Abiertos.Insertar(inicial);  
    while (Abiertos.Tamano()>0) {  
        Nodo Actual = Abiertos.SacarPrimero();  
        Cerrados.Insertar(Actual);  
        if (Actual == fin) {  
            return Recuperar_Camino(inic,Actual);  
        } else {  
            List<Nodo> sucesores = getSucesores(Actual);  
            sucesores = QuitarRepeticiones(sucesores,Cerrados);  
            Abiertos.Insertar_PRINCIPIO(sucesores);  
        }  
    }  
    return null;  
}
```

# Profundidad limitada

# Profundidad limitada

- Se dejan de generar sucesores cuando se llega al límite de profundidad.
  - Esta modificación garantiza que el algoritmo acaba (en problemas con ramas infinitas)

# Algoritmo: PROF LIMITADA

```
public Lista<Accion> Busqueda(Nodo inicial,fin) {  
    Lista Abiertos = new Lista();  
    Lista Cerrados = new Lista();  
    Abiertos.Insertar(inicial);  
    while (Abiertos.Tamano()>0) {  
        Nodo Actual = Abiertos.SacarPrimero();  
        Cerrados.Insertar(Actual);  
        if (Actual == fin) {  
            return Recuperar_Camino(inic,Actual);  
        } else {  
            List<Nodo> sucesores = getSucesores(Actual);  
            sucesores = QuitarRepetidos(sucesores,Cerrados);  
            Quitar_Fuera_de_Nivel(sucesores);  
            Abiertos.Insertar_PRINCIPIO(sucesores);  
        }  
    }  
    return null;  
}
```

# Algoritmo: PROF LIMITADA

```
public Lista<Accion> Busqueda(Nodo inicial,fin) {  
    Lista Abiertos = new Lista();  
    Lista Cerrados = new Lista();  
    Abiertos.Insertar(inicial);  
    while (Abiertos.Tamano()>0) {  
        Nodo Actual = Abiertos.SacarPrimero();  
        Cerrados.Insertar(Actual);  
        if (Actual == fin) {  
            return Recuperar_Camino(inic,Actual);  
        } else {  
            List<Nodo> sucesores = getSucesores(Actual);  
            sucesores = QuitarRepetidos(sucesores,Cerrados);  
            Quitar_Fuera_de_Nivel(sucesores);  
            Abiertos.Insertar_PRINCIPIO(sucesores);  
        }  
    }  
    return null;  
}
```

# Profundidad Iterativa

# Profundidad iterativa

- PI combina la complejidad espacial de la “Profundidad” con la optimización de la “Anchura”.
- El algoritmo consiste en realizar búsquedas en profundidad limitada sucesivas con un nivel de profundidad máximo acotado y creciente en cada iteración.
- Se consigue el comportamiento de la búsqueda primero en anchura pero sin su coste espacial, ya que la exploración es en profundidad.

# Profundidad iterativa

- Los nodos se regeneran a cada iteración.
- PI permite evitar los casos en que la búsqueda primero en profundidad no acaba (existen ramas infinitas).
- En la primera iteración la profundidad máxima será 1.
  - Este valor irá aumentando en sucesivas iteraciones hasta llegar a la solución.
  - Para garantizar que el algoritmo acabe si no hay solución, se puede definir una cota máxima de profundidad en la exploración.

# Profundidad iterativa

```
public Lista<Accion> Busqueda(Nodo inicial,Nodo fin) {  
    int nivel = 0;  
    while (!Encontrado) {  
        Lista Abiertos = new Lista();  
        Lista Cerrados = new Lista();  
        Abiertos.Insertar(inicial);  
        while (!Abiertos.esVacio()) {  
            Nodo Actual = Abiertos.SacarPrimero();  
            Cerrados.Insertar(Actual);  
            if (Actual == fin) {  
                return Recuperar_Camino(inic,Actual);  
            } else {  
                List<Nodo> sucesores = getSucesores(Actual);  
                sucesores = QuitarRepetidos(sucesores,Cerrados);  
                sucesores2 = QuitarFueraNivel(sucesores,nivel);  
                Abiertos.Insertar_AL_PRINCIPIO(sucesores2);  
            }  
        }  
        nivel = nivel + 1;  
    }  
    return null;  
}
```

# Algoritmo: PROF ITERATIVA

```
public Lista<Accion> Busqueda(Nodo inicial,fin) {  
    int nivel = 0;  
    while (!encontrado){  
        Lista Abiertos = new Lista();  
        Lista Cerrados = new Lista();  
        Abiertos.Insertar(inicial);  
        while (Abiertos.Tamano()>0) {  
            Nodo Actual = Abiertos.SacarPrimero();  
            Cerrados.Insertar(Actual);  
            if (Actual == fin) {  
                return Recuperar_Camino(inic,Actual);  
            } else {  
                List<Nodo> sucesores = getSucesores(Actual);  
                sucesores = QuitarRepetidos(sucesores,Cerrados);  
                Quitar_Fuera_de_Nivel(sucesores);  
                Abiertos.Insertar_PRINCIPIO(sucesores);  
            }  
        }  
        nivel = nivel + 1  
    }  
    return null;  
}
```

# Algoritmo: PROF ITERATIVA

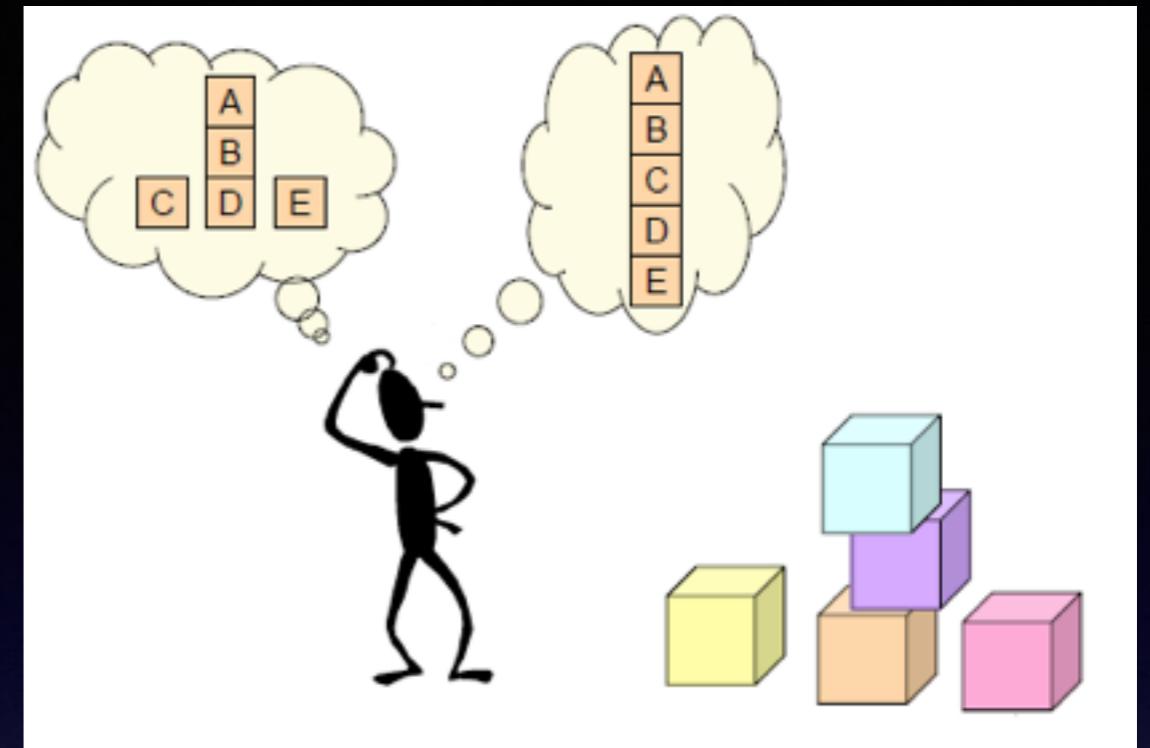
```
public class Busqueda_Busqueda(Nodo inicial,fin) {  
    int nivel = 0;  
    while (!encontrado){  
        Abiertos = new Lista();  
        Cerrados = new Lista();  
        Abiertos.Insertar(inicial);  
        while (Abiertos.Tamano()>0) {  
            Nodo Actual = Abiertos.SacarPrimero();  
            Cerrados.Insertar(Actual);  
            if (Actual == fin) {  
                return Recuperar_Camino(inic,Actual);  
            } else {  
                List<Nodo> sucesores = getSucesores(Actual);  
                sucesores = QuitarRepetidos(sucesores,Cerrados);  
                Quitar_Fuera_de_Nivel(sucesores);  
                Abiertos.Insertar_PRINCIPIO(sucesores);  
            }  
        }  
        nivel = nivel + 1  
    }  
    return null;  
}
```

# Algoritmo: PROF ITERATIVA

```
public class Busqueda_Busqueda(Nodo inicial,fin) {  
    int nivel = 0;  
    while (!encontrado){  
        Abiertos = new Lista();  
        Cerrados = new Lista();  
        Abiertos.Insertar(inicial);  
        while (Abiertos.Tamano()>0) {  
            Nodo Actual = Abiertos.SacarPrimero();  
            Cerrados.Insertar(Actual);  
            if (Actual == fin) {  
                return Recuperar_Camino(inic,Actual);  
            } else {  
                List<Nodo> sucesores = getSucesores(Actual);  
                sucesores = QuitarRepetidos(sucesores,Cerrados);  
                Quitar_Fuera_de_Nivel(sucesores);  
                Abiertos.Insertar_PRINCIPIO(sucesores);  
            }  
            nivel = nivel + 1  
        }  
        return null;  
    }  
}
```

# Profundidad iterativa

- Completitud: el algoritmo siempre encontrará la solución.
- Complejidad temporal: la misma que la búsqueda en anchura. El regenerar el árbol en cada iteración solo añade un factor constante a la función de coste -  $O(rp+1)$ .
- Complejidad espacial: igual que en la búsqueda en profundidad -  $O(r m)$ .
- Optimización: la solución es óptima igual que en la búsqueda en anchura.



# Búsqueda Informada

# Introducción

- La búsqueda informada utiliza el conocimiento específico del problema.
- Puede encontrar soluciones de una manera más eficiente.
- Una función heurística,  $h'(n)$ , mide el coste estimado más barato desde el nodo  $n$  a un nodo objetivo.
- $h'(n)$  se utiliza para guiar el proceso haciendo que en cada momento se seleccione el estado o las operaciones más prometedores.

# Función de coste

- La función de evaluación tiene dos componentes:
  - coste mínimo para ir desde el (un) inicio al nodo actual ( $g$ )
  - coste mínimo (estimado) para ir desde el nodo actual a una solución ( $h$ )

$$f'(n) = g(n) + h'(n)$$

# Importancia del estimador

Operaciones:

- situar un bloque libre en la mesa
- situar un bloque libre sobre otro bloque libre

A
H
G
F
E
D
C
B

Estado inicial

$$h'1 = 4$$

$$h'2 = -28$$

Estimador  $h'1$ :

- sumar 1 por cada bloque que esté colocado sobre el bloque que debe
- restar 1 si el bloque no está colocado sobre el que debe

Estimador  $h'2$ :

- si la estructura de apoyo es correcta sumar 1 por cada bloque de dicha estructura
- si la estructura de apoyo no es correcta restar 1 por cada bloque de dicha estructura

H
G
F
E
D
C
B
A

Estado final

$$h'1 = 8$$

$$h'2 = 28 (= 7+6+5+4+3+2+1)$$



# Estrategias de búsqueda informada (heurísticas)

- **No** siempre se **garantiza** encontrar una solución (de existir ésta).
- No siempre se garantiza encontrar la solución más próxima (la que se encuentra a una distancia, número de operaciones, menor).



# Estrategias de búsqueda informada (heurísticas)

- Búsqueda primero el mejor
- A, A\*, A\*PI
- Búsqueda local:
  - ascensión de colinas, temple simulado, algoritmos genéticos, búsqueda en línea, etc...

# Búsqueda voraz (primero el mejor)

- La búsqueda voraz primero el mejor expande el nodo más cercano al objetivo.
  - Probablemente conduce rápidamente a una solución.
- Evalúa los nodos utilizando solamente la función heurística, que, en general, se minimiza, porque se refiere a un coste:
  - La minimización de  $h'(n)$  es susceptible de ventajas falsas.

$$f'(n) = h'(n)$$

# Algoritmo: VORAZ

```
public Lista<Accion> Busqueda(Nodo inicial,fin) {  
    Lista Abiertos = new Lista();  
    Lista Cerrados = new Lista();  
    Abiertos.Insertar(inicial);  
    while (Abiertos.Tamano()>0) {  
        Nodo Actual = Abiertos.SacarPrimero();  
        Cerrados.Insertar(Actual);  
        if (Actual == fin) {  
            return Recuperar_Camino(inic,Actual);  
        } else {  
            List<Nodo> sucesores = getSucesores(Actual);  
            sucesores = QuitarRepetidos(sucesores,Cerrados);  
            Abiertos.Insertar_PRINCIPIO(sucesores);  
        }  
    }  
    return null;  
}
```

# Búsqueda voraz primero el mejor

- La estructura de abiertos es una cola con prioridad.
- La prioridad la marca la función heurística (coste estimado del camino que falta hasta la solución).
- En cada iteración se escoge el nodo más “cercano” a la solución (el primero de la cola).
- No se garantiza la solución óptima.

# Algoritmos de clase A

- La función de evaluación tiene dos componentes:
  - coste mínimo para ir desde el (un) inicio al nodo actual ( $g$ )
  - coste mínimo (estimado) para ir desde el nodo actual a una solución ( $h$ )

$$f'(n) = g(n) + h'(n)$$

# Algoritmos de clase A

- $f'$  es un valor estimado del coste total.
- $h'$  (función heurística) es un valor estimado del coste de alcanzar el objetivo.
- $g$  es un coste real: lo gastado por el camino más corto conocido.
- Preferencia: siempre al nodo con menor  $f'$ .
- En caso de empate: preferencia al nodo con una menor  $h'$ .

# Algoritmos de clase A\*

- Cuanto más  $h'$  se aproxime al verdadero coste, mejor.
- Si  $h'(n)$  nunca sobrestima el coste real, es decir  $\forall n: h'(n) \leq h(n)$ , se puede demostrar que el algoritmo encontrará (de haberlo) un camino óptimo.
- Se habla en este caso de algoritmos A\*.

# Algoritmo A\*

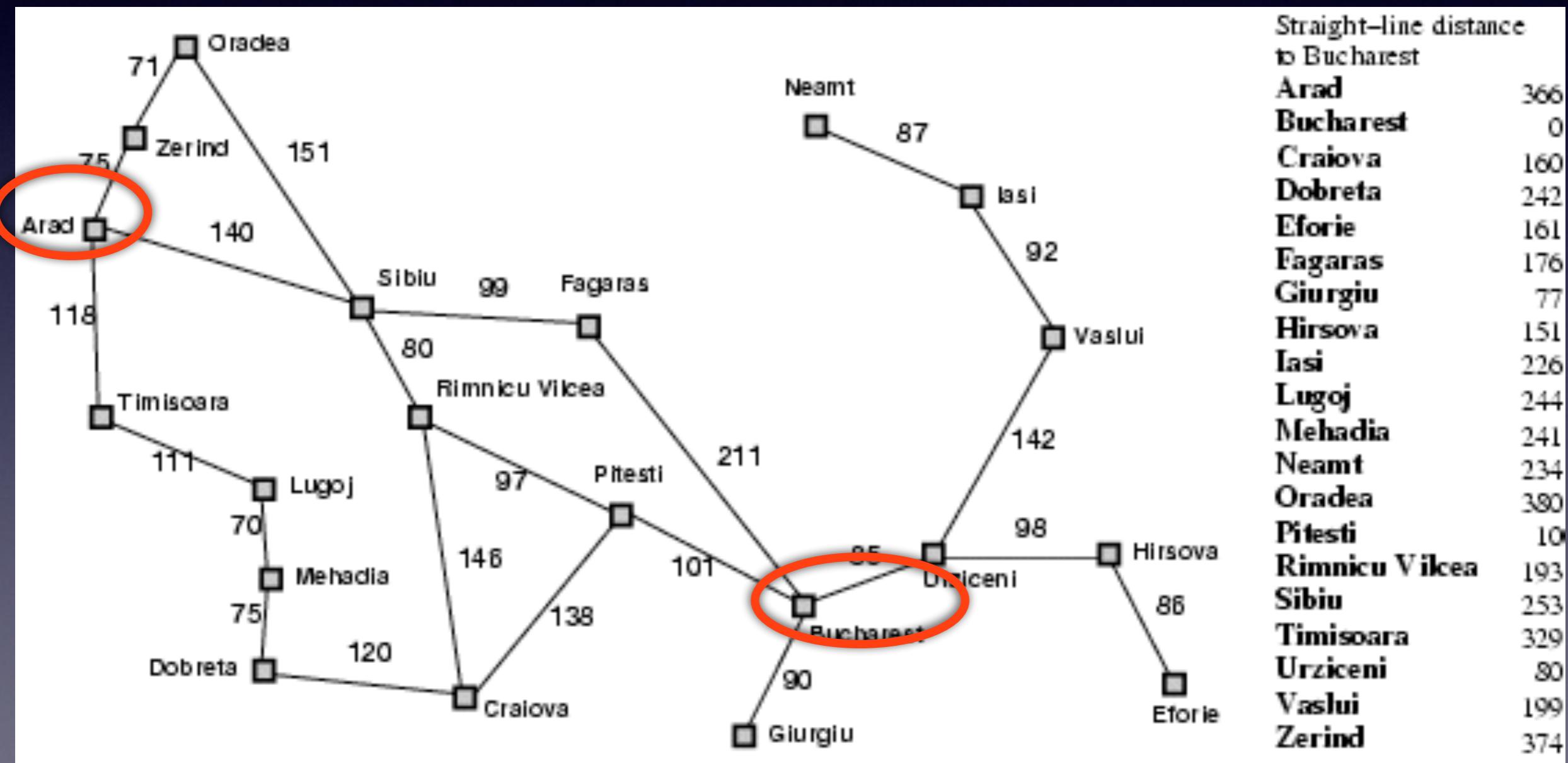
- La estructura de abiertos es una cola con prioridad.

```
public Lista<Accion> Busqueda(Nodo inicial,fin) {  
    Lista Abiertos = new Lista();  
    Lista Cerrados = new Lista();  
    Abiertos.Insertar(inicial);  
    while (Abiertos.Tamano()>0) {  
        Nodo Actual = Abiertos.SacarPrimero();  
        Cerrados.Insertar(Actual);  
        if (Actual == fin) {  
            return Recuperar_Camino(inic,Actual);  
        } else {  
            List<Nodo> sucesores = getSucesores(Actual);  
            sucesores = QuitarRepetidos(sucesores,Cerrados);  
            Abiertos.Insertar_Ordenado(sucesores);  
        }  
    }  
    return null;  
}
```

# Algoritmo A\*

- La prioridad la marca la función de estimación  $f'(n)=g(n)+h'(n)$ .
- En cada iteración se escoge el mejor camino estimado (el primero de la cola).
- A\* es completo cuando el factor de ramificación es finito y cada operador tiene un coste positivo fijo.

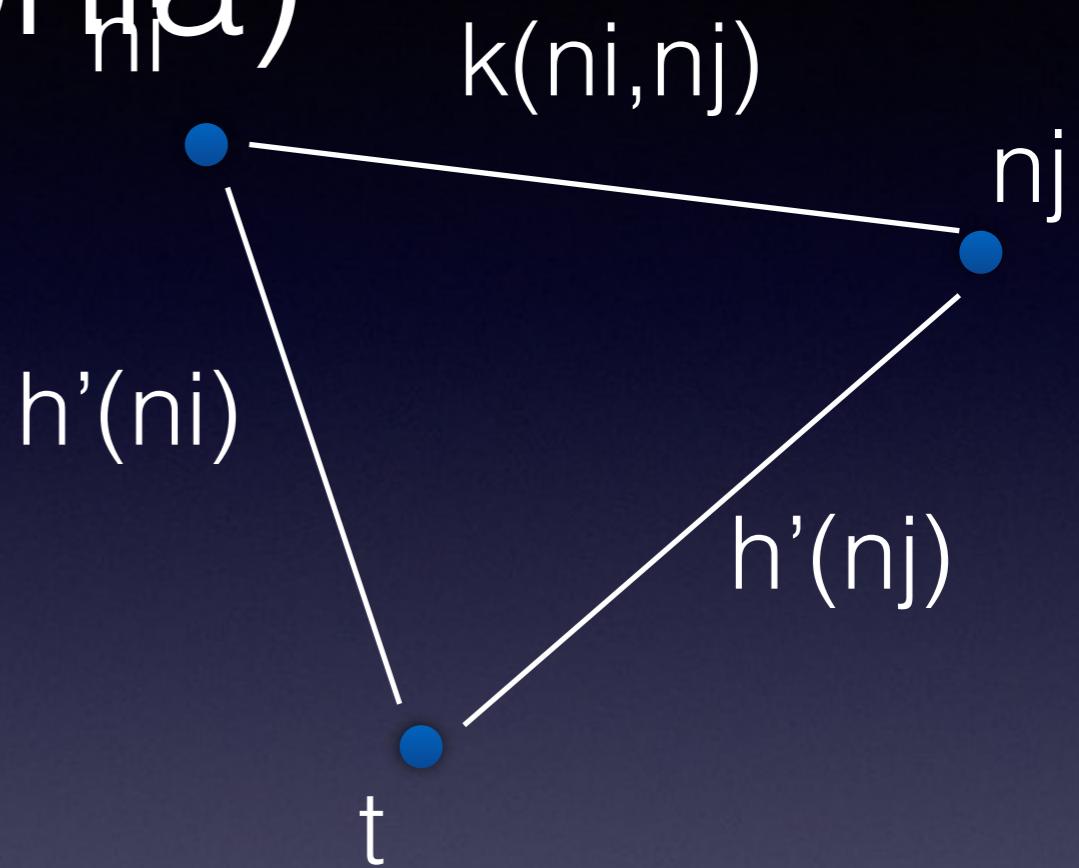
# Ejemplo: encontrar una ruta en Rumanía



# Optimización de A\*

- Un algoritmo A, dependiendo de la función heurística, encontrará o no una solución óptima.
- Si la función heurística es consistente, la optimización está asegurada.

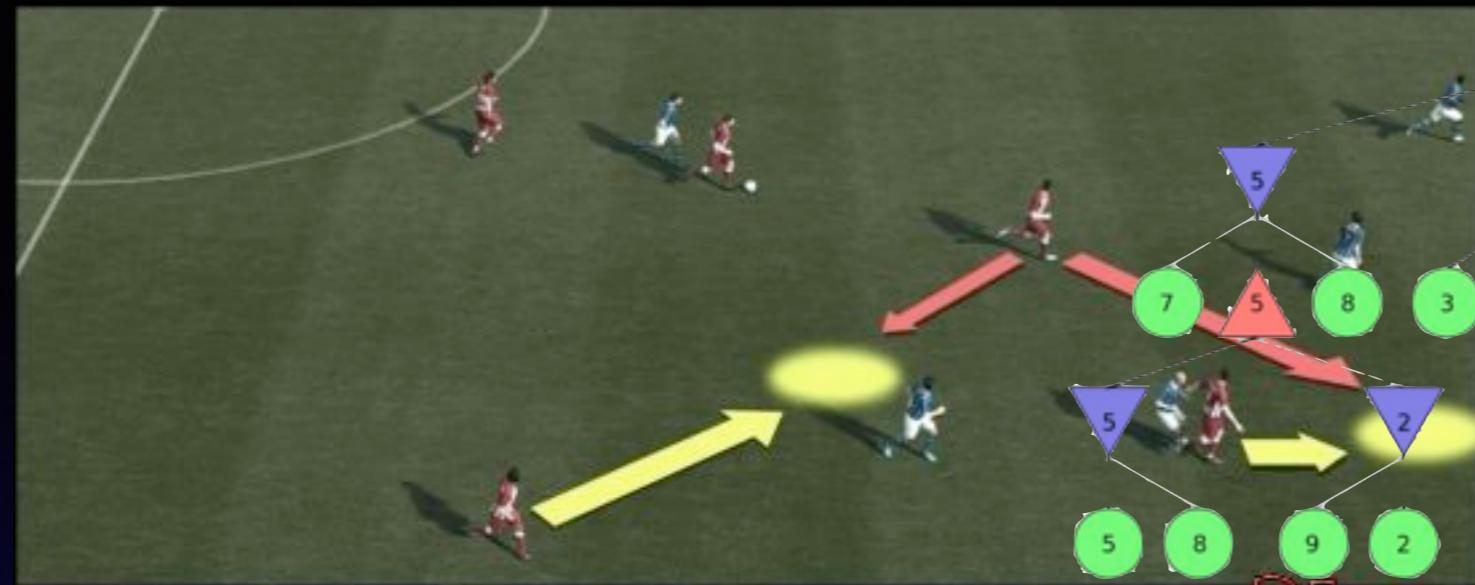
# Condición de consistencia (o monotonía)



- El nodo  $nj$  es sucesor de  $ni$
- $h'(ni) = \text{coste estimado desde } n \text{ a la solución } t$
- $h'(ni)$  cumple la desigualdad triangular si:
  - $h'(ni) \leq k(ni, nj) + h'(nj)$

# Admisibilidad de A\*

- Una función heurística es admisible si se cumple la siguiente propiedad:
- para todo  $n$ :  $0 \leq h'(n) \leq h(n)$
- Por lo tanto,  $h'(n)$  ha de ser un estimador optimista, nunca ha de sobreestimar  $h(n)$ .
- Usar una función heurística admisible garantiza que un nodo en el camino óptimo no pueda parecer tan malo como para no considerarlo nunca.

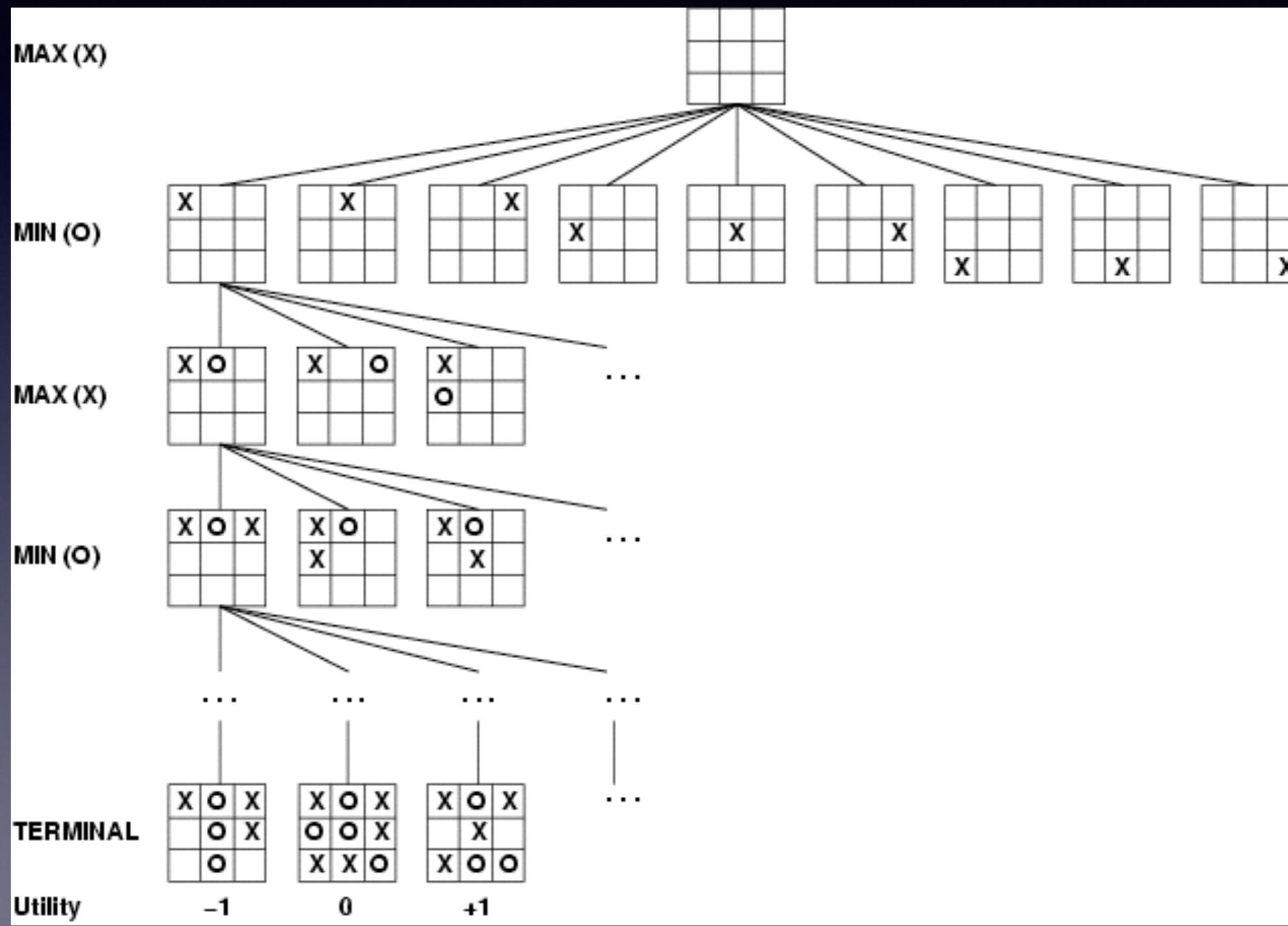


# Búsquedas con adversario

# Decisiones óptimas en juegos

- Un juego puede definirse formalmente como una clase de problemas de búsqueda con los componentes siguientes:
  - El estado inicial
  - Una función sucesor, que devuelve una lista de pares (movimiento, estado)
  - Un test terminal, que determina cuándo termina el juego (por estructura o propiedades o función utilidad)
  - Una función utilidad

# Búsqueda entre adversarios



# Búsqueda entre adversarios

# Búsqueda entre adversarios

- El objetivo es encontrar un conjunto de movimientos accesible que dé como ganador a MAX.

# Búsqueda entre adversarios

- El objetivo es encontrar un conjunto de movimientos accesible que dé como ganador a MAX.
- Algoritmo MiniMax:

# Búsqueda entre adversarios

- El objetivo es encontrar un conjunto de movimientos accesible que dé como ganador a MAX.
- Algoritmo MiniMax:
  - generar todo el árbol de jugadas.

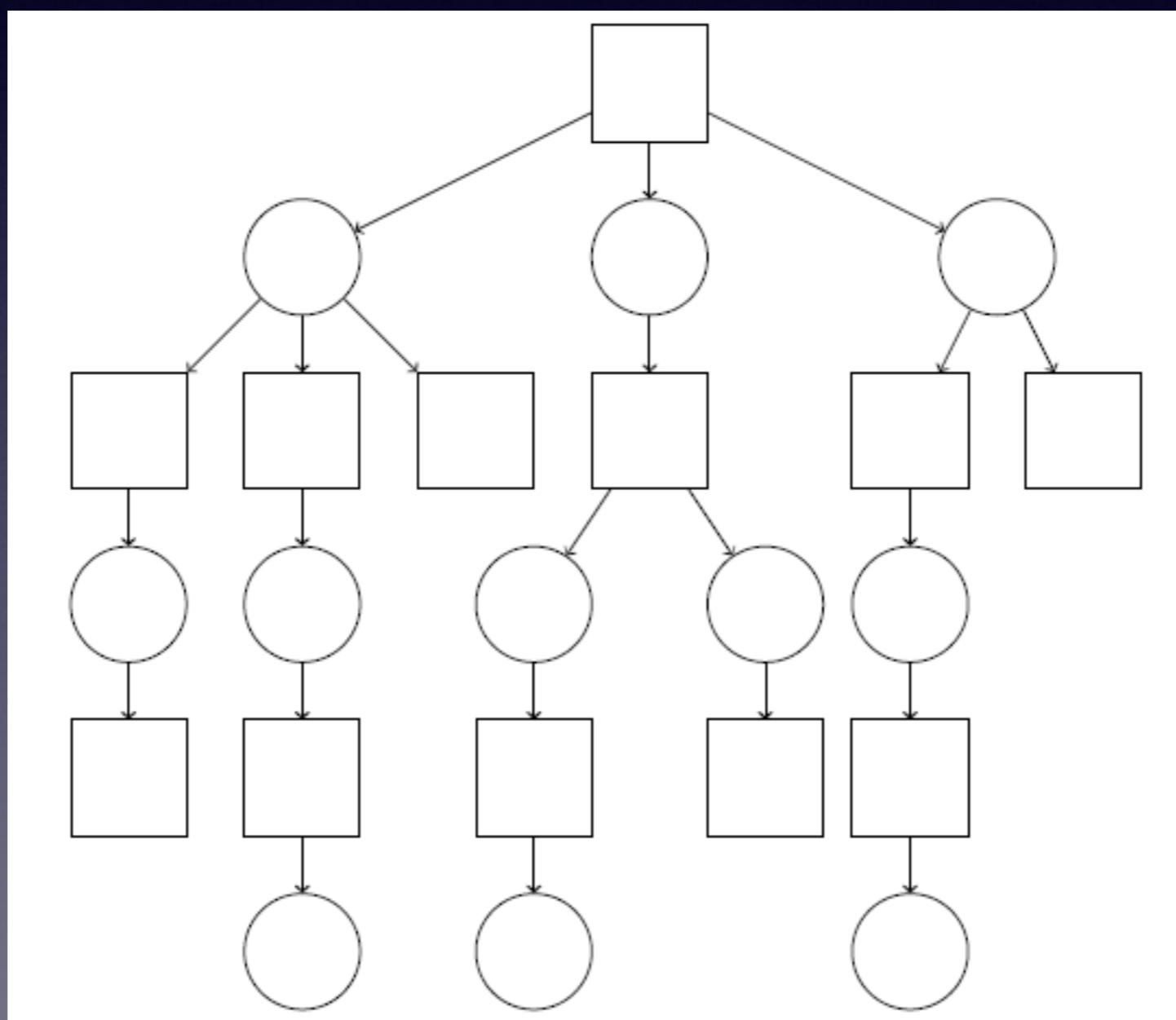
# Búsqueda entre adversarios

- El objetivo es encontrar un conjunto de movimientos accesible que dé como ganador a MAX.
- Algoritmo MiniMax:
  - generar todo el árbol de jugadas.
  - Se etiquetan las jugadas terminales, dependiendo de si gana MAX o MIN, con un valor de utilidad de, por ejemplo, “+1” o “-1”.

# Búsqueda entre adversarios

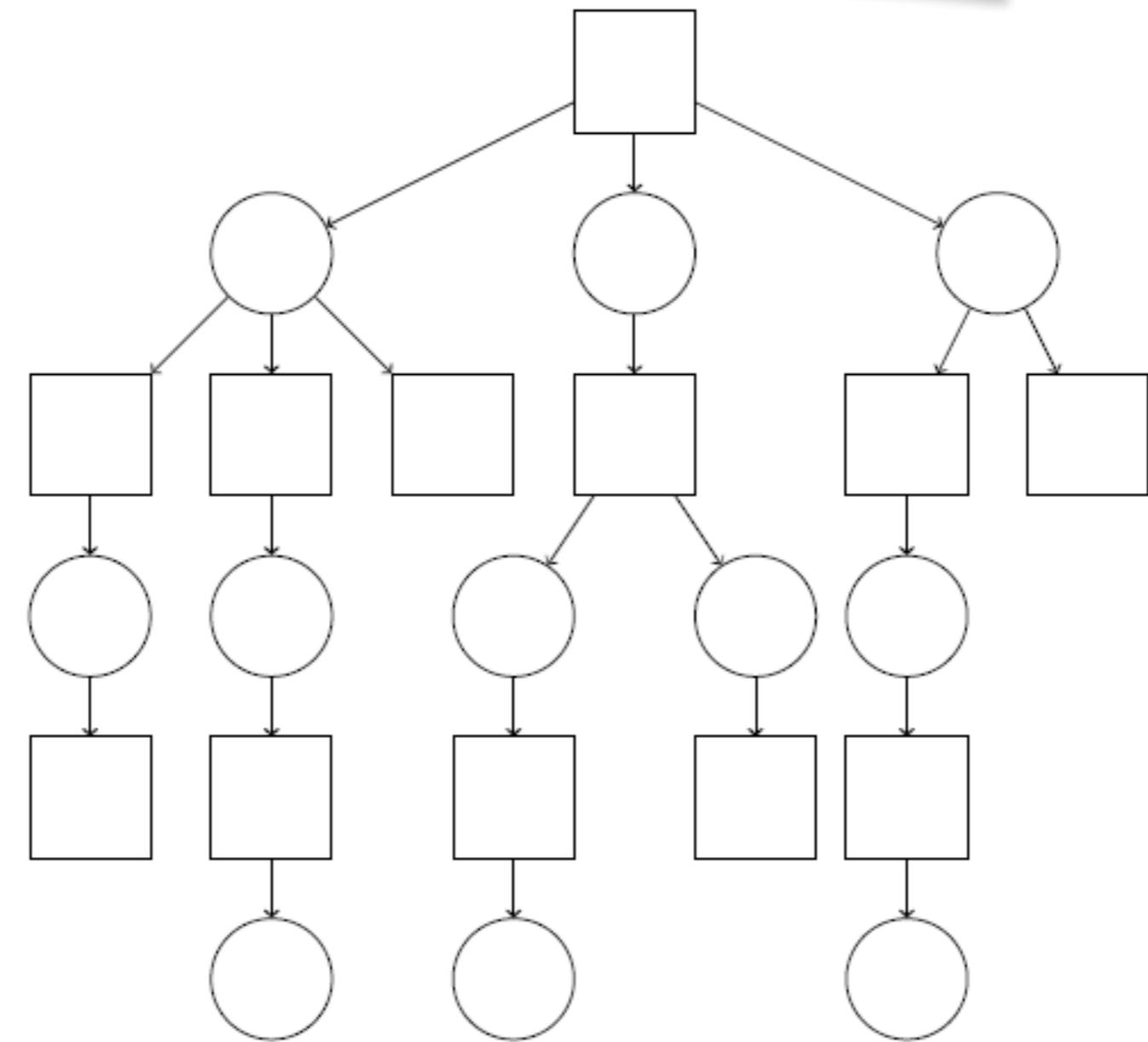
- El objetivo es encontrar un conjunto de movimientos accesible que dé como ganador a MAX.
- Algoritmo MiniMax:
  - generar todo el árbol de jugadas.
  - Se etiquetan las jugadas terminales, dependiendo de si gana MAX o MIN, con un valor de utilidad de, por ejemplo, “+1” o “-1”.
  - Se propagan los valores de las jugadas terminales de las hojas hasta la raíz.

# Búsqueda entre adversarios

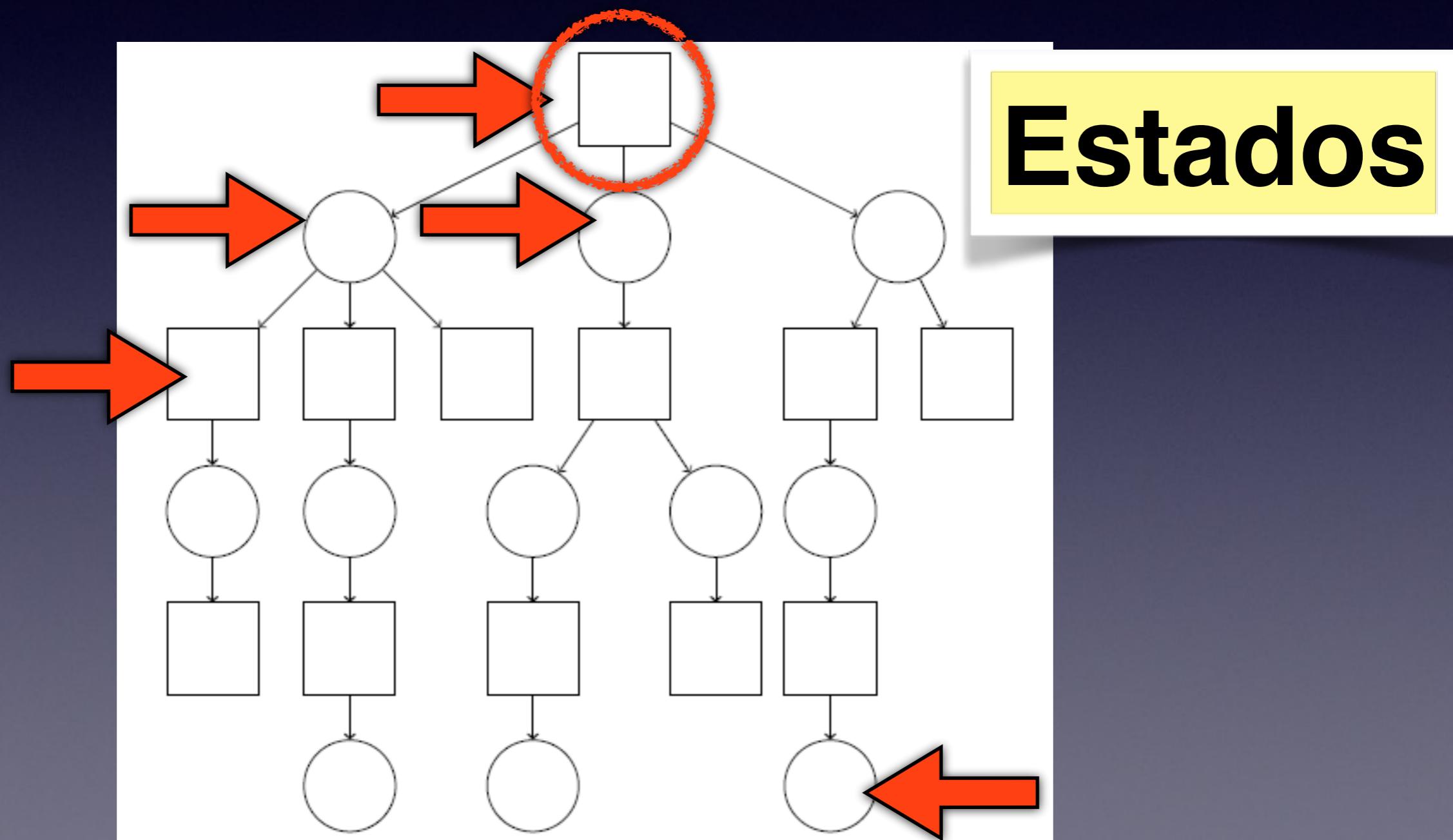


# 1. Desarrollo completo del árbol

versarios

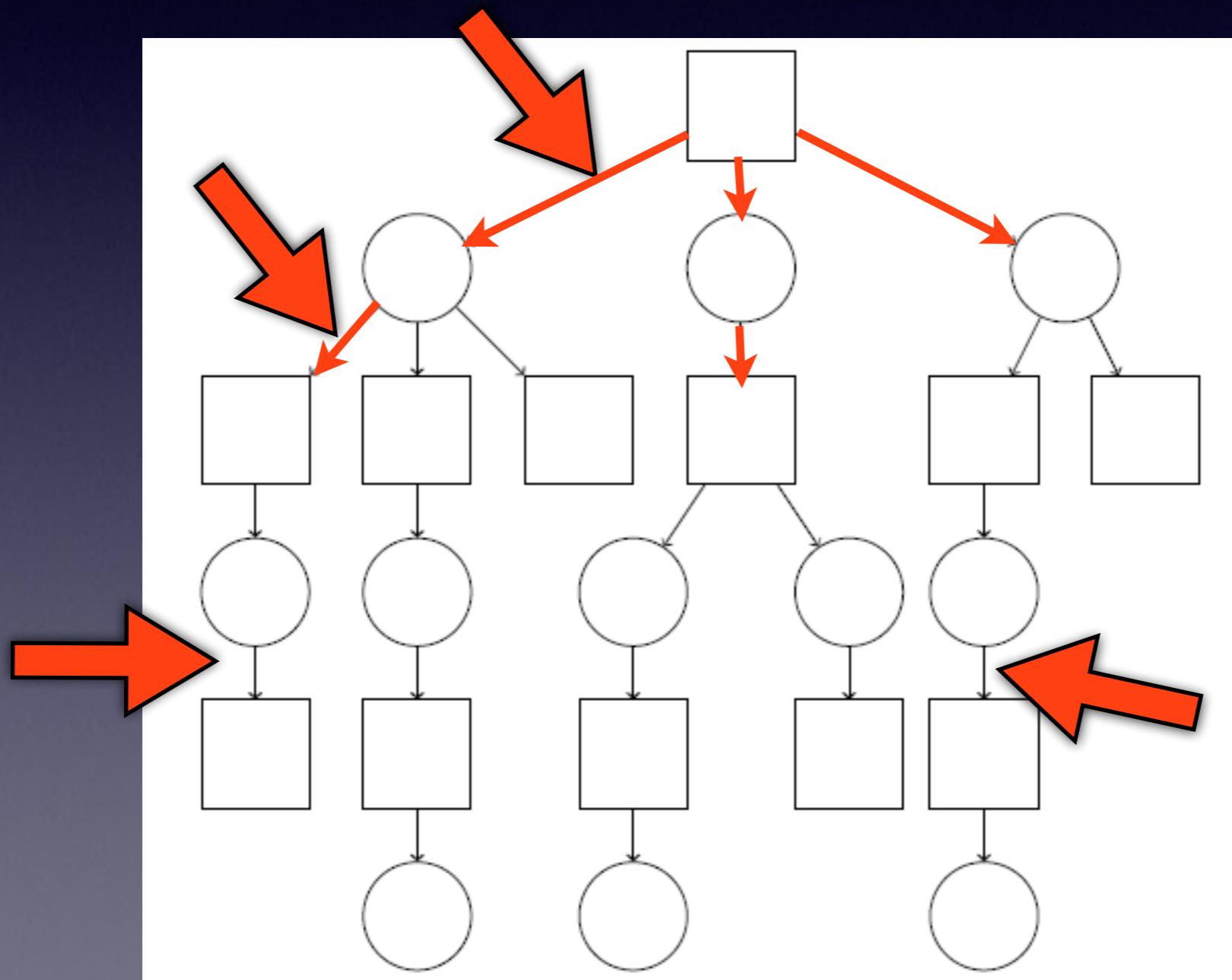


# Búsqueda entre adversarios



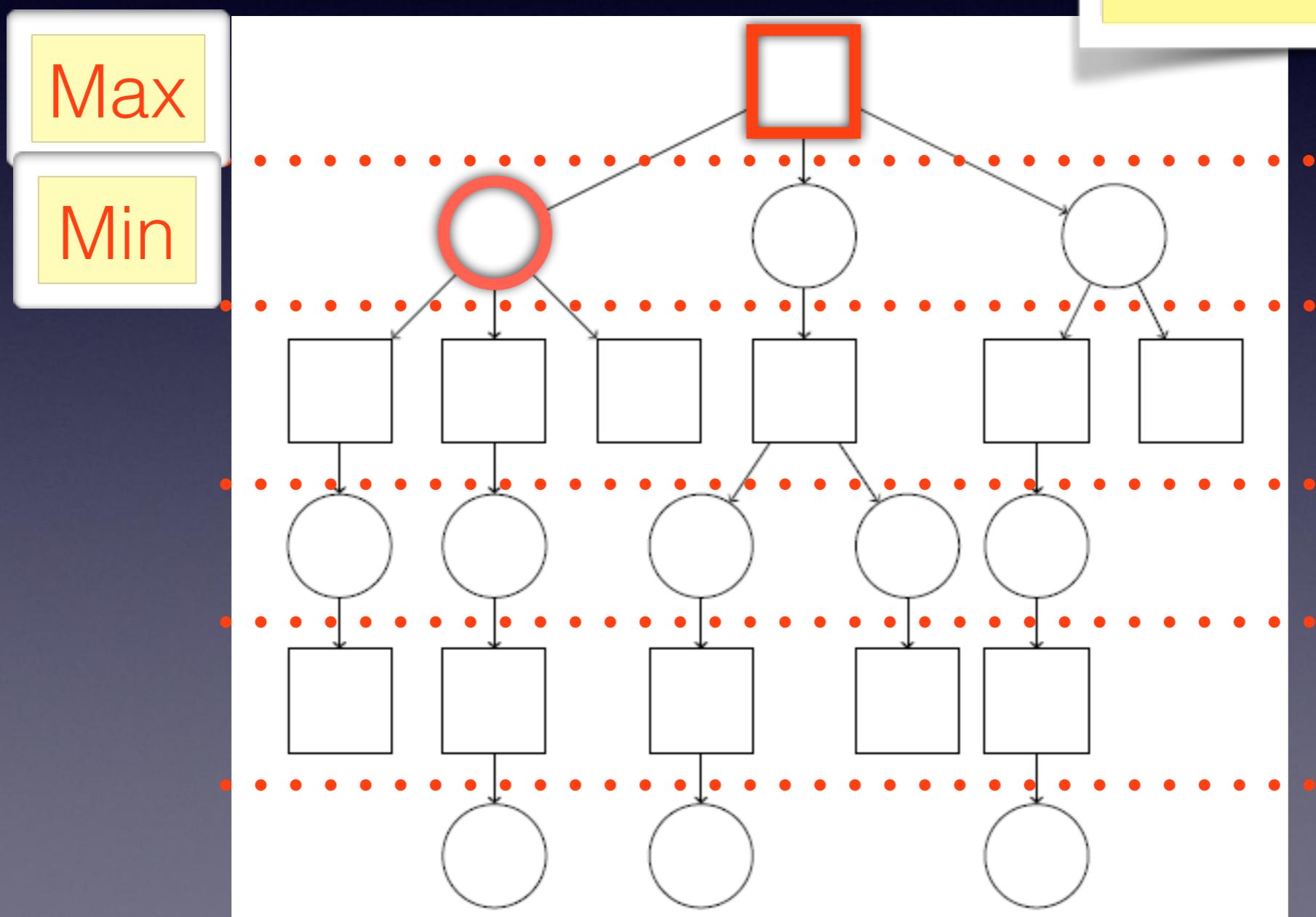
# Búsqueda entre adversarios

# Movimientos



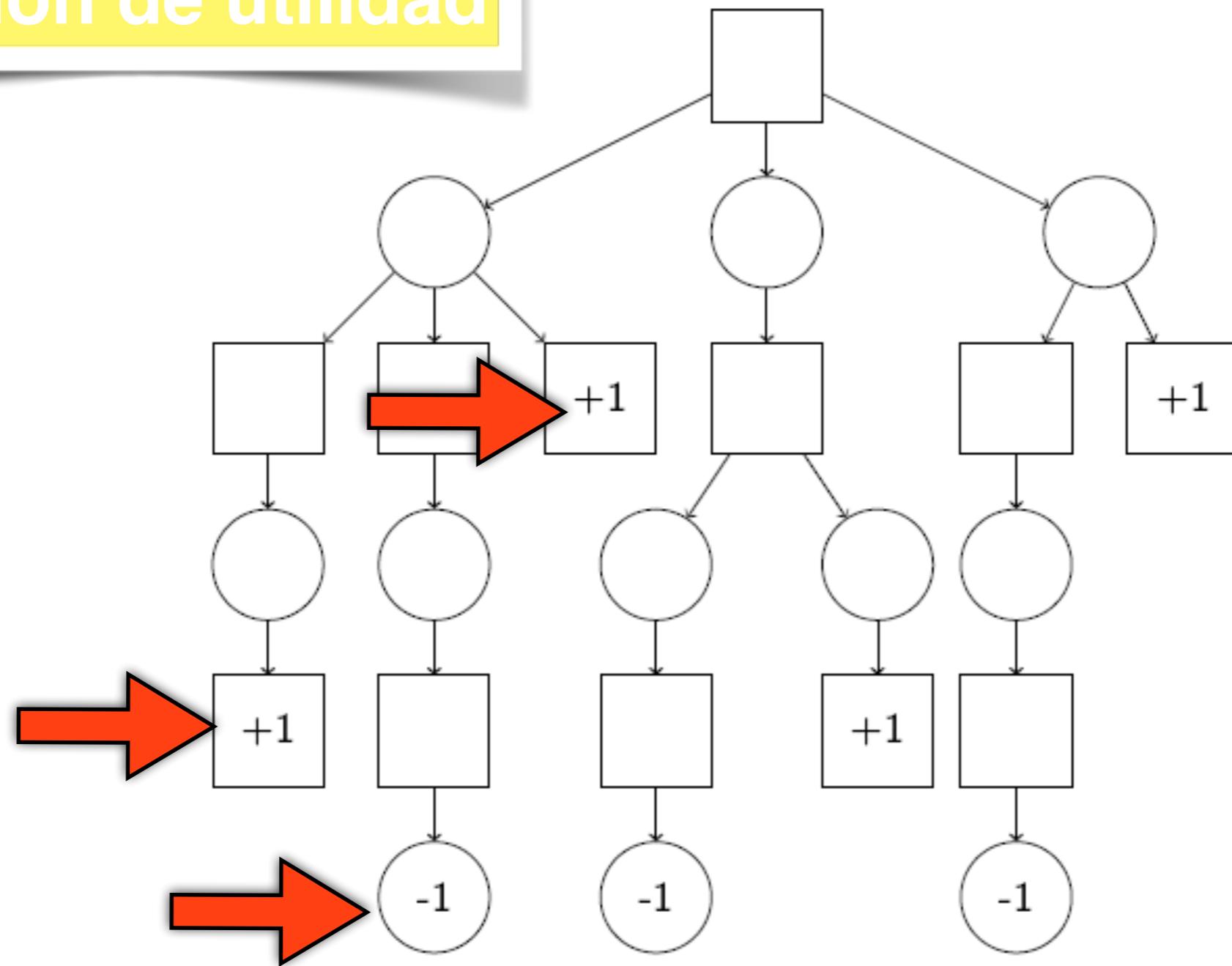
# Búsqueda entre adverberios

# Niveles MinMax



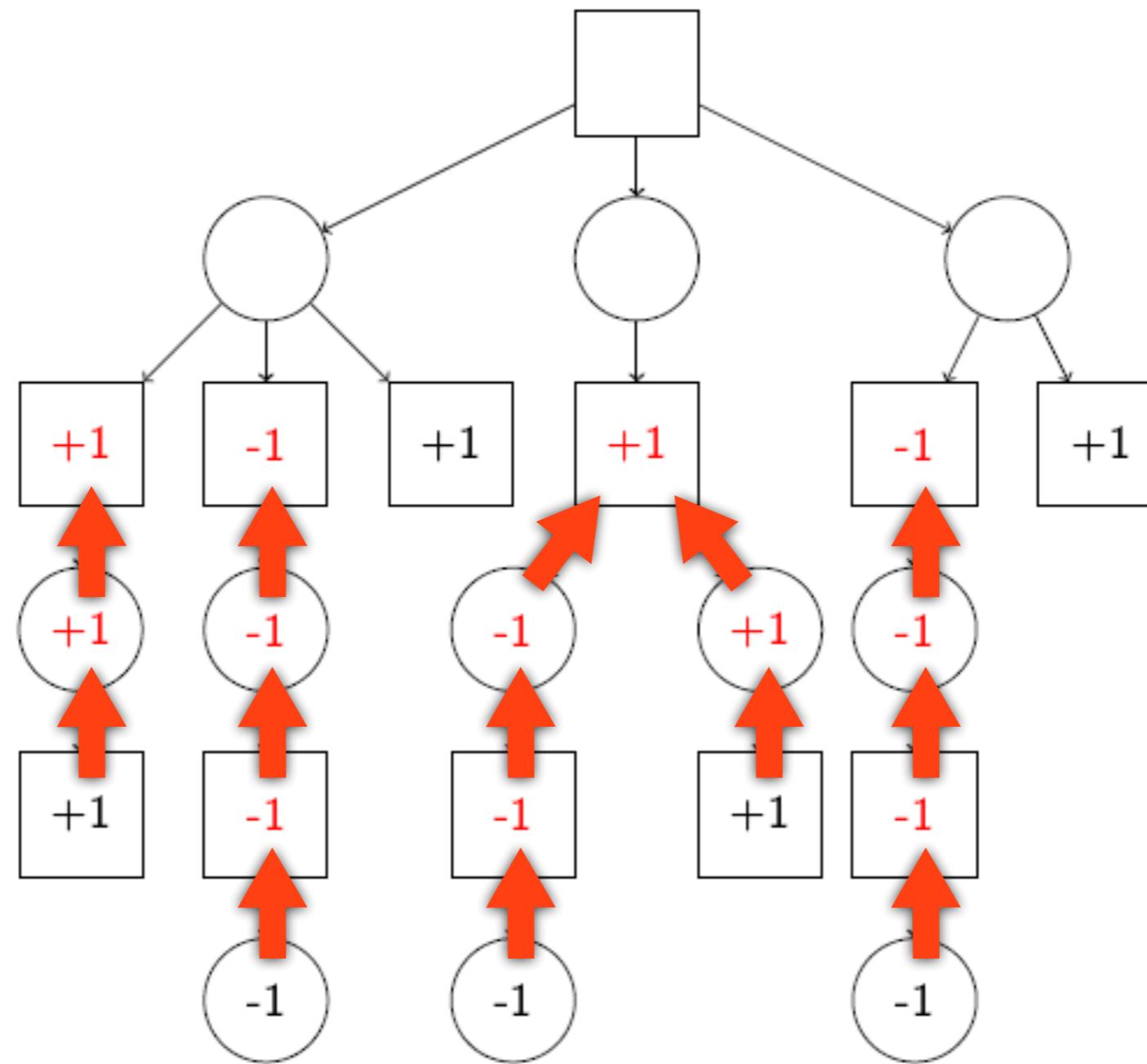
# Búsqueda entre adversarios

## Función de utilidad



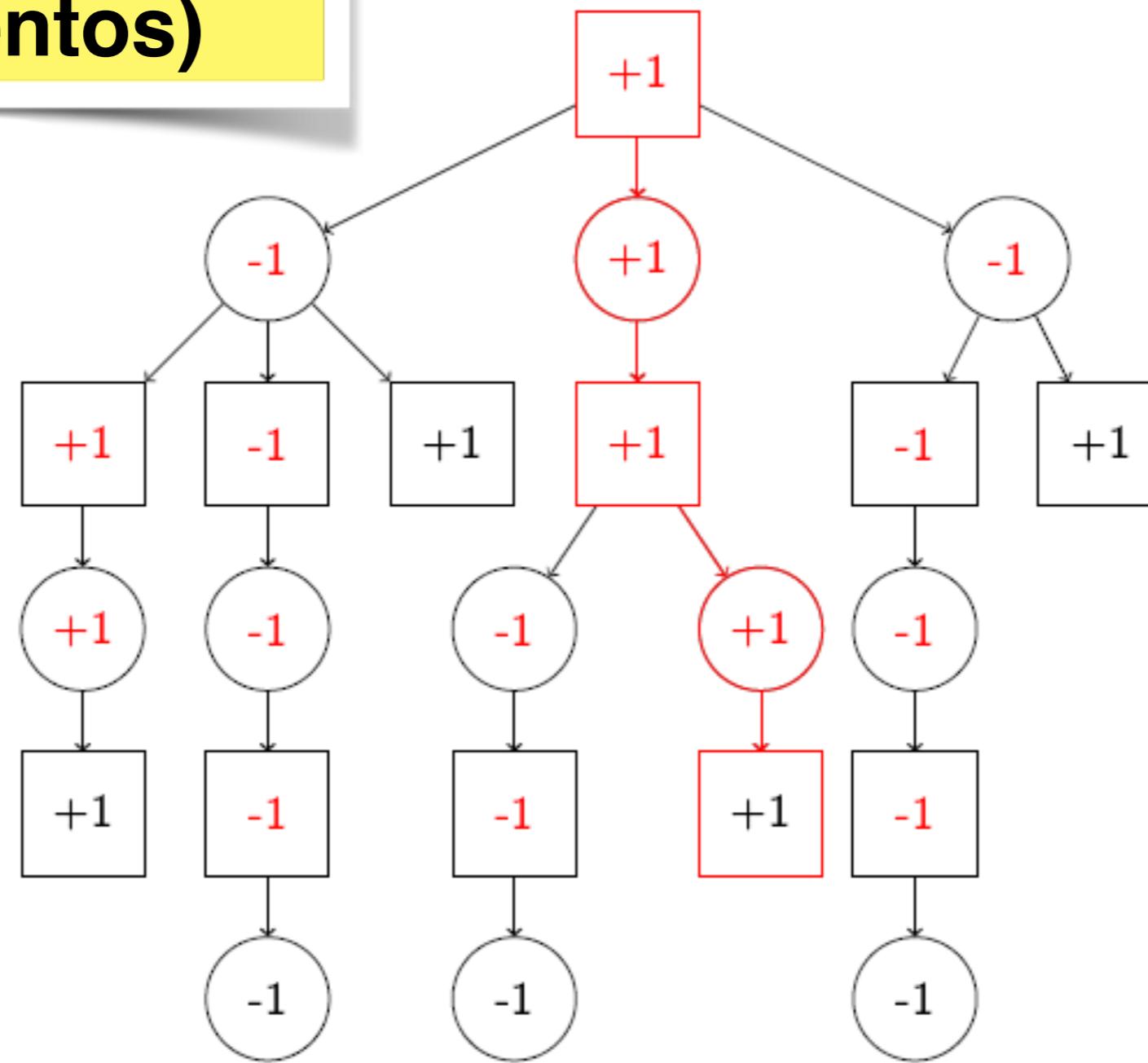
# Búsquedas entre adversarios

## 2. PROPAGACIÓN



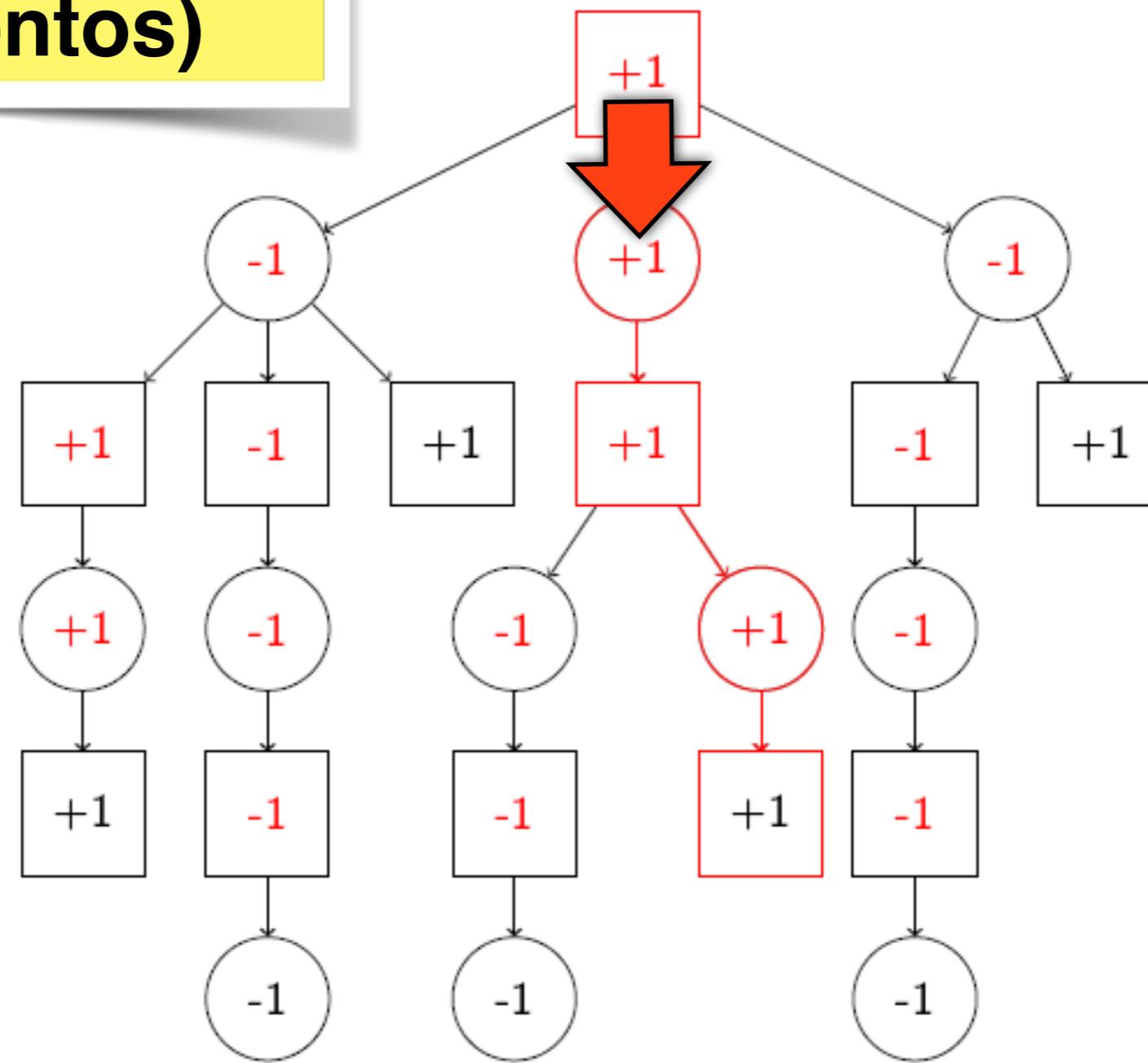
# Búsqueda entre adversarios

## 3. ELEGIR CAMINO (movimientos)



# Búsqueda entre adversarios

## 3. ELEGIR CAMINO (movimientos)



# Desventajas

- Cada nueva decisión por parte del adversario implicará repetir parte de la búsqueda.
  - Implementación alternativa puede mantener cierta estructura
- Incluso un juego simple como tic-tac-toe es demasiado complejo para dibujar el árbol de juegos entero.

# Aproximación heurística

- El algoritmo busca con profundidad limitada.
- Definir una función que nos indique lo cerca que estamos de una jugada ganadora (o perdedora).
- En esta función interviene información del dominio.
- Esta función no representa ningún coste, ni es una distancia en pasos.
- Cada nueva decisión por parte del adversario implicará repetir parte de la búsqueda.

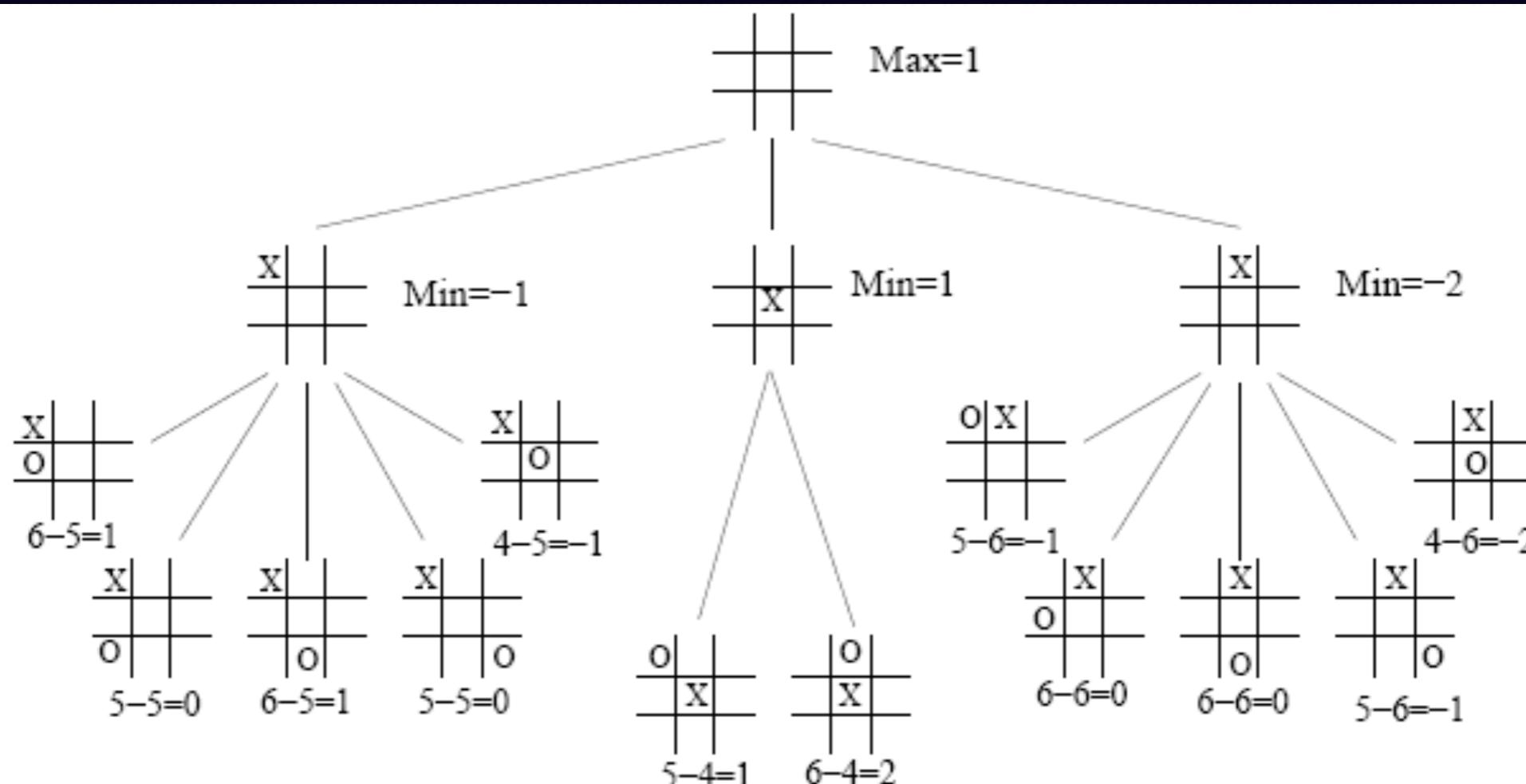
# Ejemplo: tic-tac-toe

- MAX juega con X y desea maximizar e
- MIN juega con 0 y desea minimizar e
- e (función utilidad) = número de filas, columnas y diagonales completas disponibles para MAX - número de filas, columnas y diagonales completas disponibles para MIN
- Valores absolutos altos de e: buena posición para el que tiene que mover
- Controlar las simetrías
- Utilizar una profundidad de parada (en el ejemplo: 2)

0	0	0
0	X	
X		X

# Ejemplo: tic-tac-toe

- Utilizar una profundidad de parada (en el ejemplo: 2)



Por convención:

las jugadas ganadoras se evalúan a “ $+\infty$ ”  
las jugadas perdedoras se evalúan a “ $-\infty$ ”

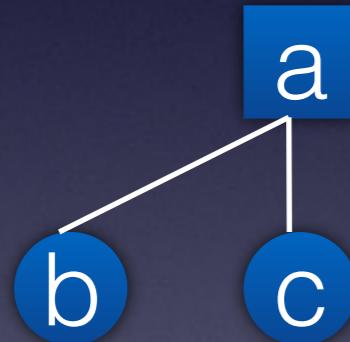
# Poda alfa-beta

- Problema de la búsqueda minimax: el número de estados que tiene que examinar es exponencial con el número de movimientos.
- Es posible calcular la decisión minimax correcta sin mirar todos los nodos en el árbol.
- La poda alfa-beta permite eliminar partes grandes del árbol, sin influir en la decisión final.

# Minimax con poda α-β

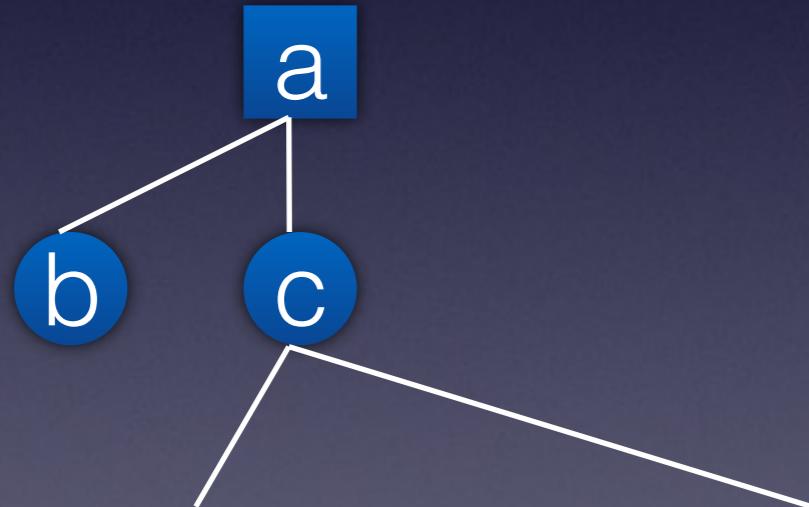
El valor de la raíz y la decisión minimax son independientes de los valores de las hojas podadas.

# Minimax con poda a- $\beta$



El valor de la raíz y la decisión minimax son independientes de los valores de las hojas podadas.

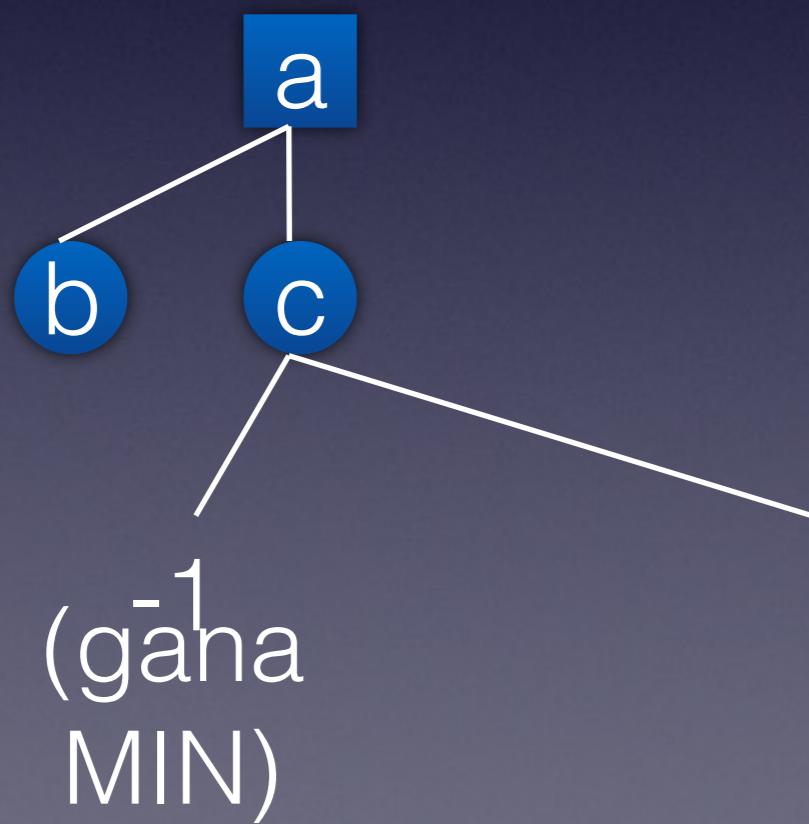
# Minimax con poda a- $\beta$



El valor de la raíz y la decisión minimax son independientes de los valores de las hojas podadas.

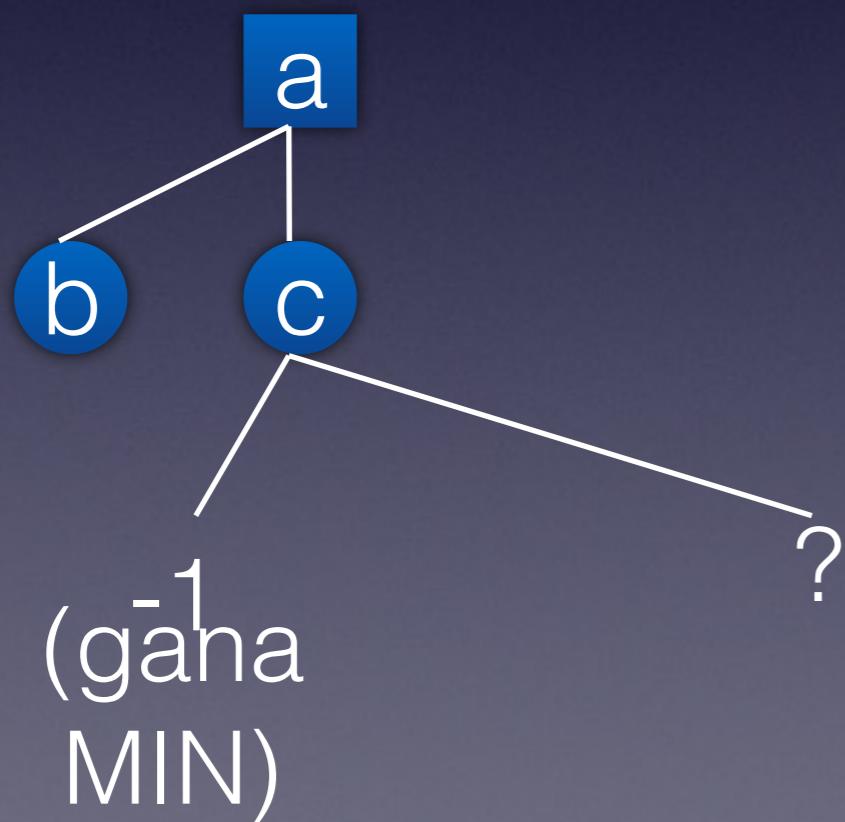
# Minimax con poda a- $\beta$

El valor de la raíz y la decisión minimax son independientes de los valores de las hojas podadas.



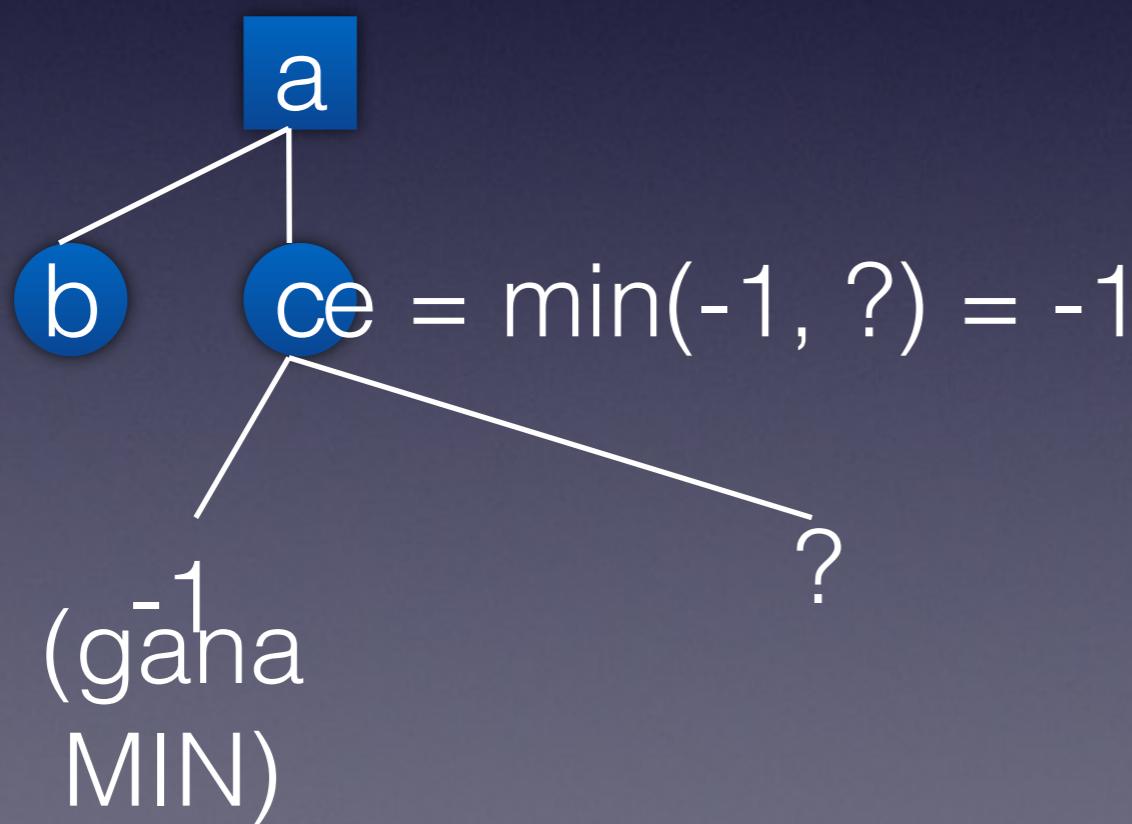
# Minimax con poda a- $\beta$

El valor de la raíz y la decisión minimax son independientes de los valores de las hojas podadas.



# Minimax con poda a-β

El valor de la raíz y la decisión minimax son independientes de los valores de las hojas podadas.

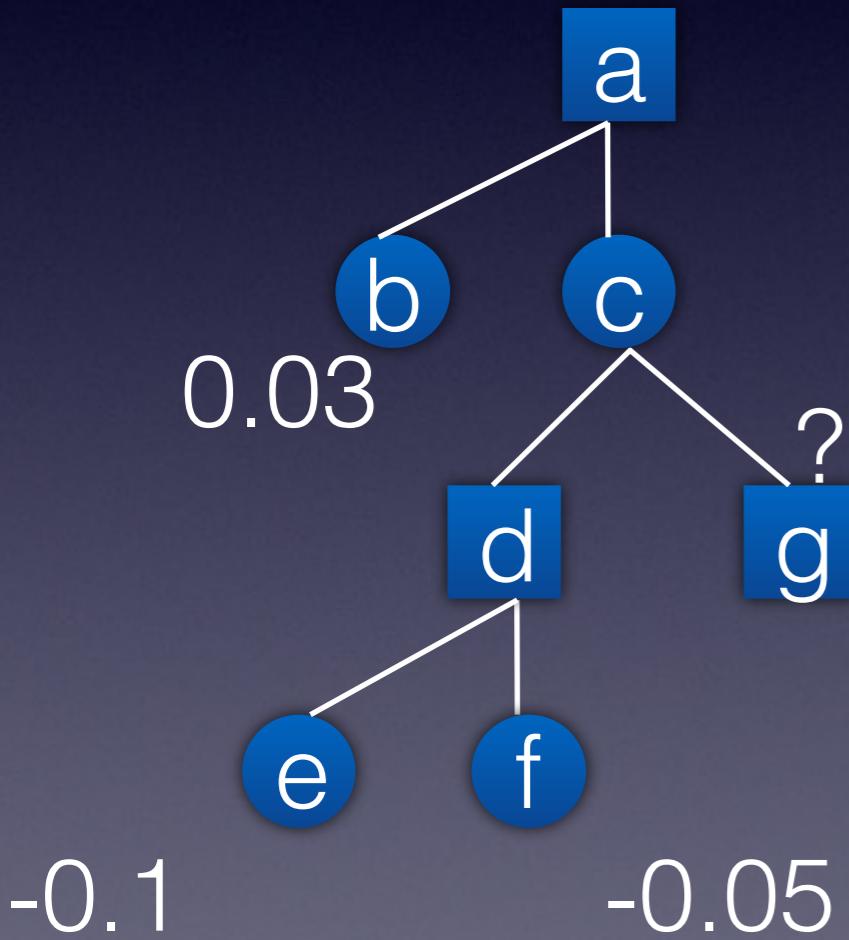


# Minimax con poda a-β

El valor de la raíz y la decisión minimax son independientes de los valores de las hojas podadas.



# Minimax con poda a- $\beta$



En c:

$$e = \min(-0.05, v(g))$$

por lo tanto en a:

$$e = \max(0.03, \min(-0.05, v(g))) = 0.03$$

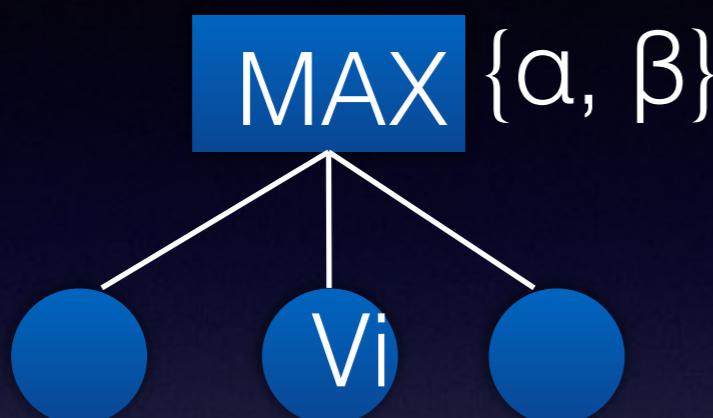
Se pueden pues podar los nodos bajo g; no aportan nada.

$$e = \max (-0.1, -0.05) = -0.05$$

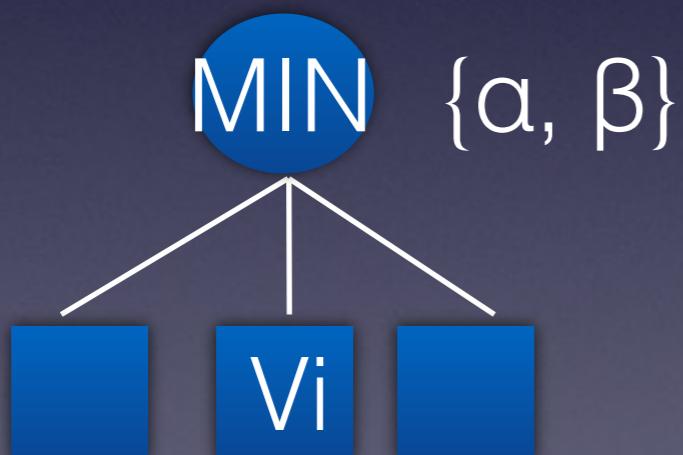
# Poda alfa-beta

- Los dos parámetros alfa y beta describen los límites sobre los valores que aparecen a lo largo del camino:
  - $\alpha$  = el valor de la mejor opción (el más alto) que se ha encontrado hasta el momento en cualquier punto del camino, para MAX
  - $\beta$  = el valor de la mejor opción (el más bajo) que se ha encontrado hasta el momento en cualquier punto del camino, para MIN
- La búsqueda alfa-beta actualiza el valor de  $\alpha$  y  $\beta$  según se va recorriendo el árbol y termina la recursión cuando encuentra un nodo peor que el actual valor  $\alpha$  o  $\beta$  correspondiente.

# Poda alfa-beta



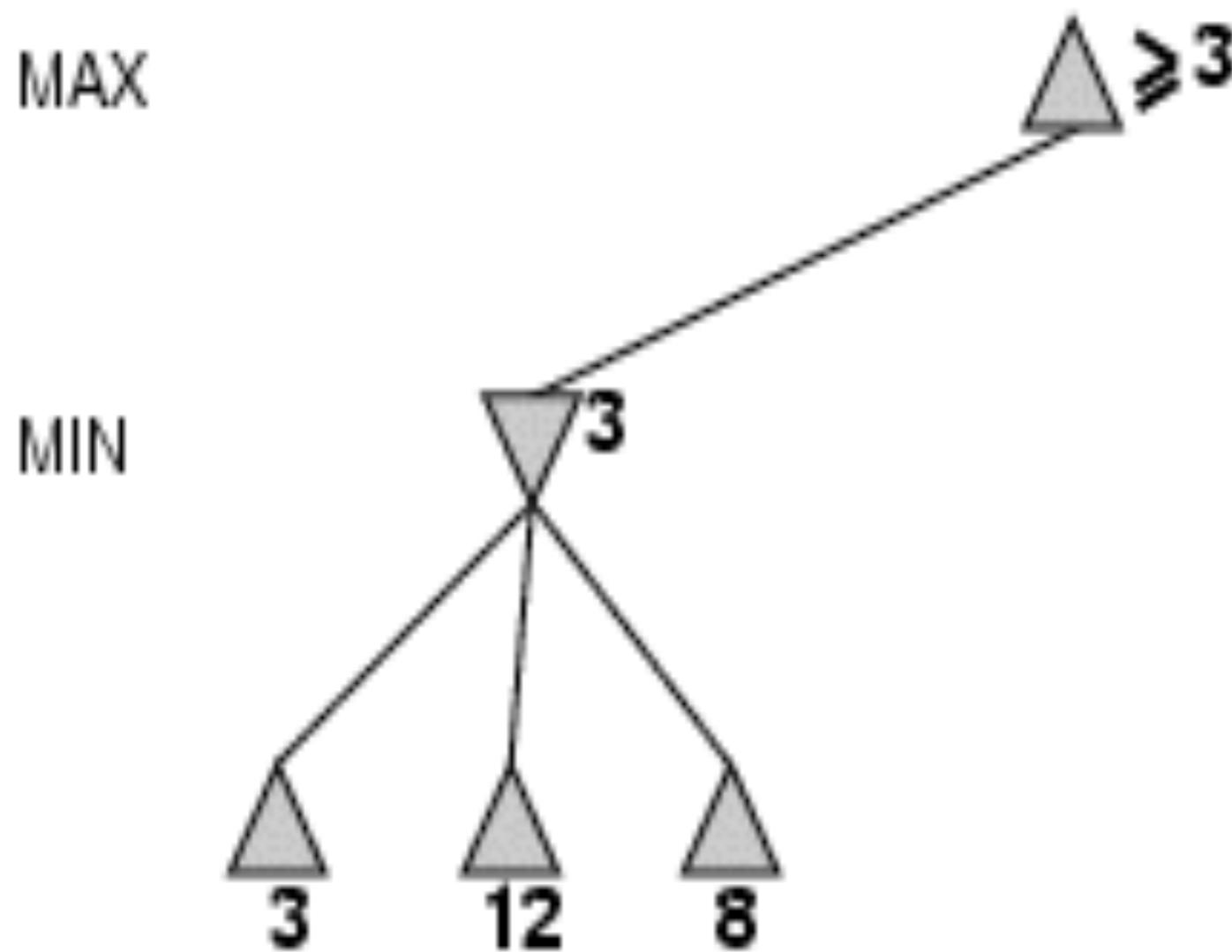
Si  $Vi \geq \beta$  poda  $\beta$   
Si  $Vi > a$  modificar a  
Retornar a



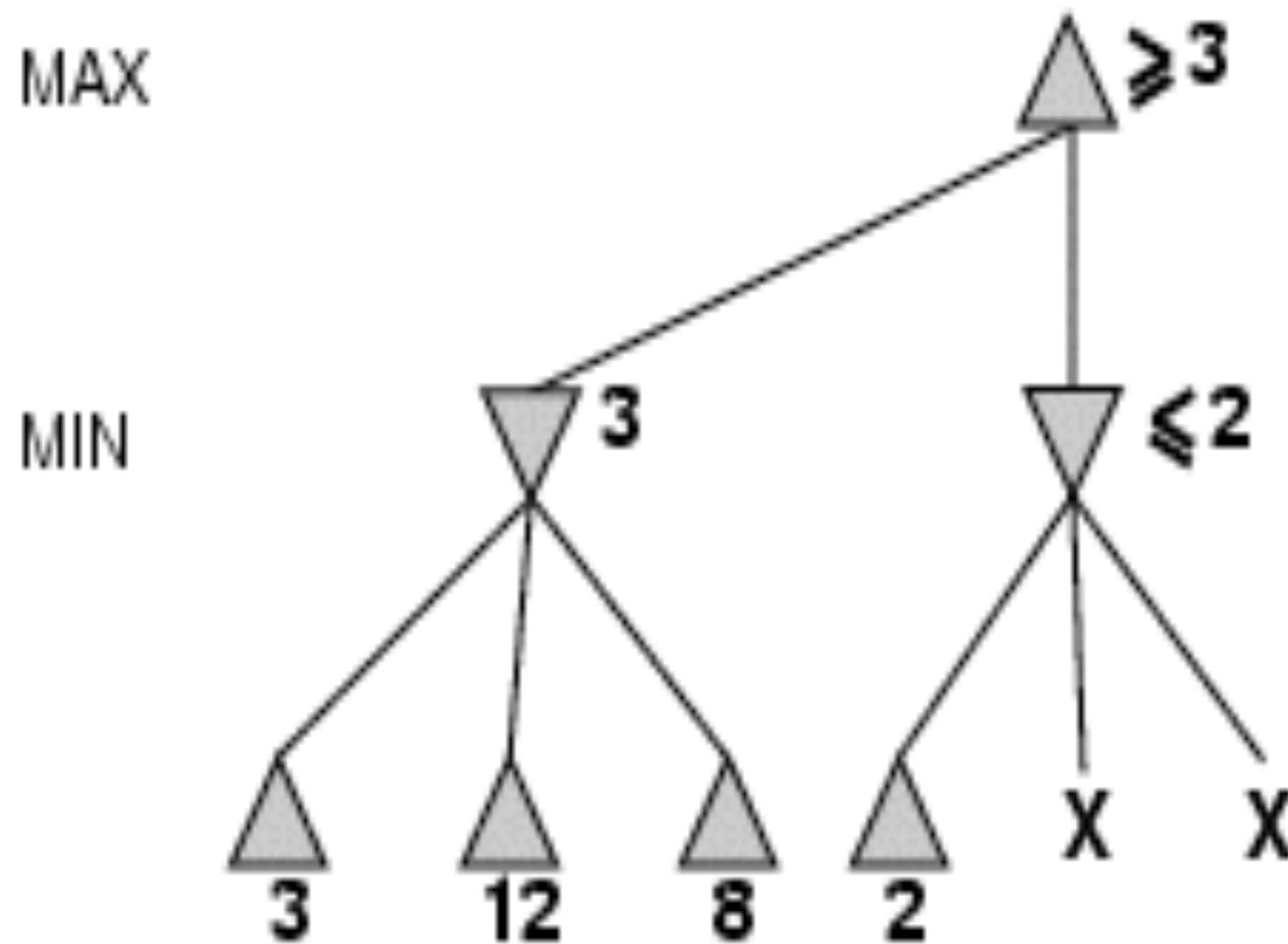
Si  $Vi \leq a$  poda a  
Si  $Vi < \beta$  modificar  $\beta$   
Retornar  $\beta$

Las cotas  $a$  y  $\beta$  se transmiten de padres a hijos de 1 en 1 y en el orden de visita de los nodos.

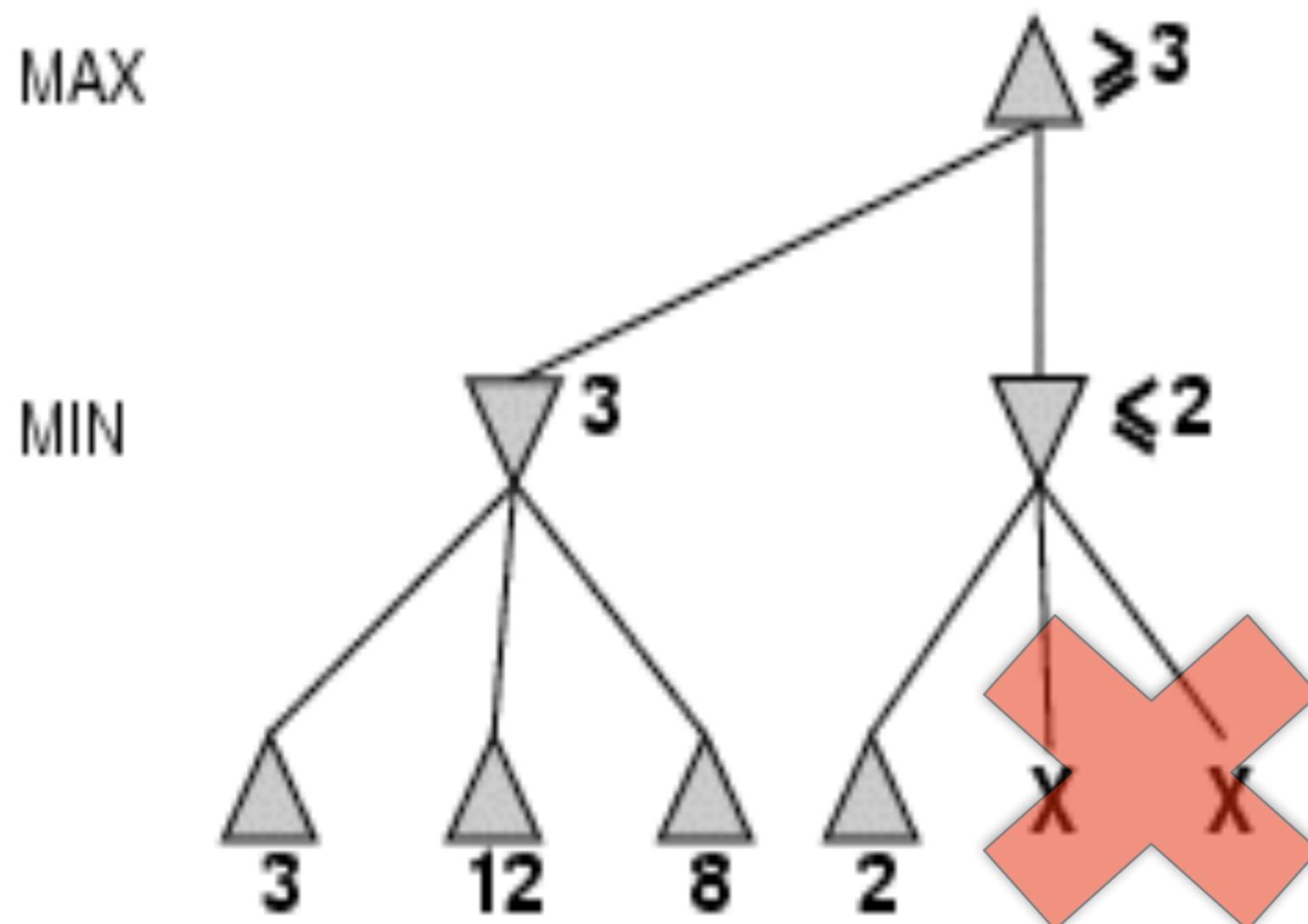
# Poda alfa-beta: ejemplo



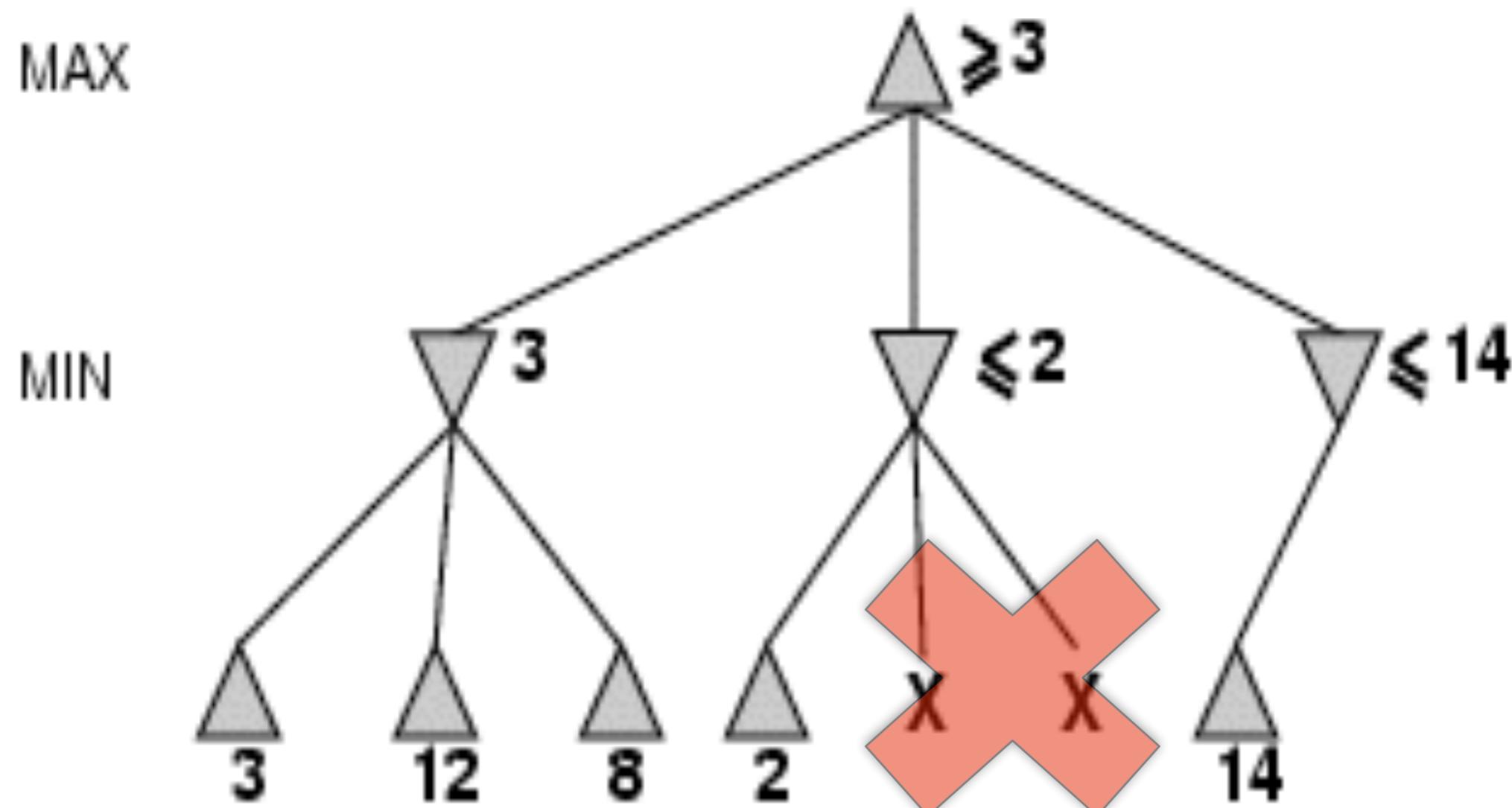
# Poda alfa-beta: ejemplo



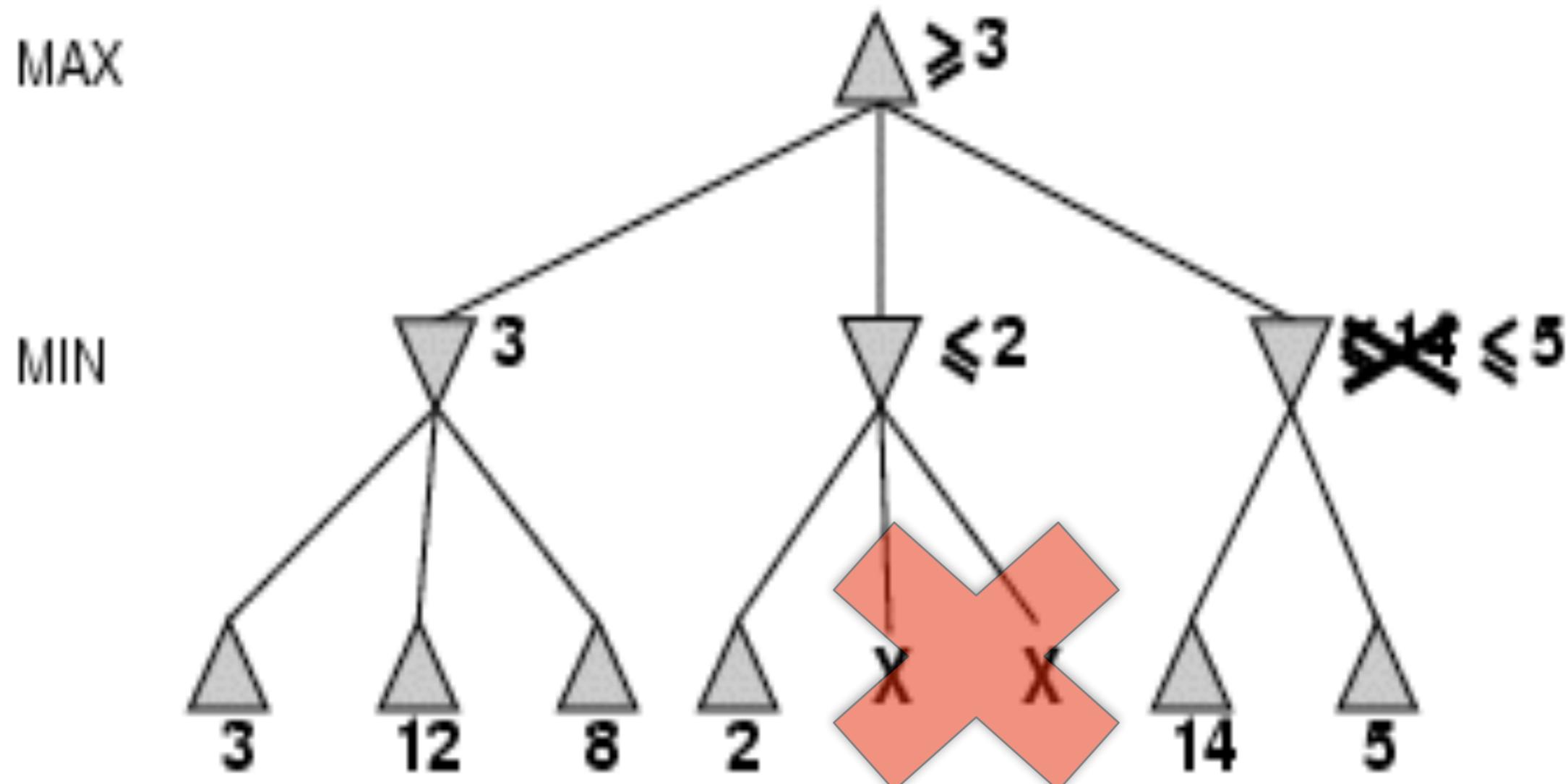
# Poda alfa-beta: ejemplo



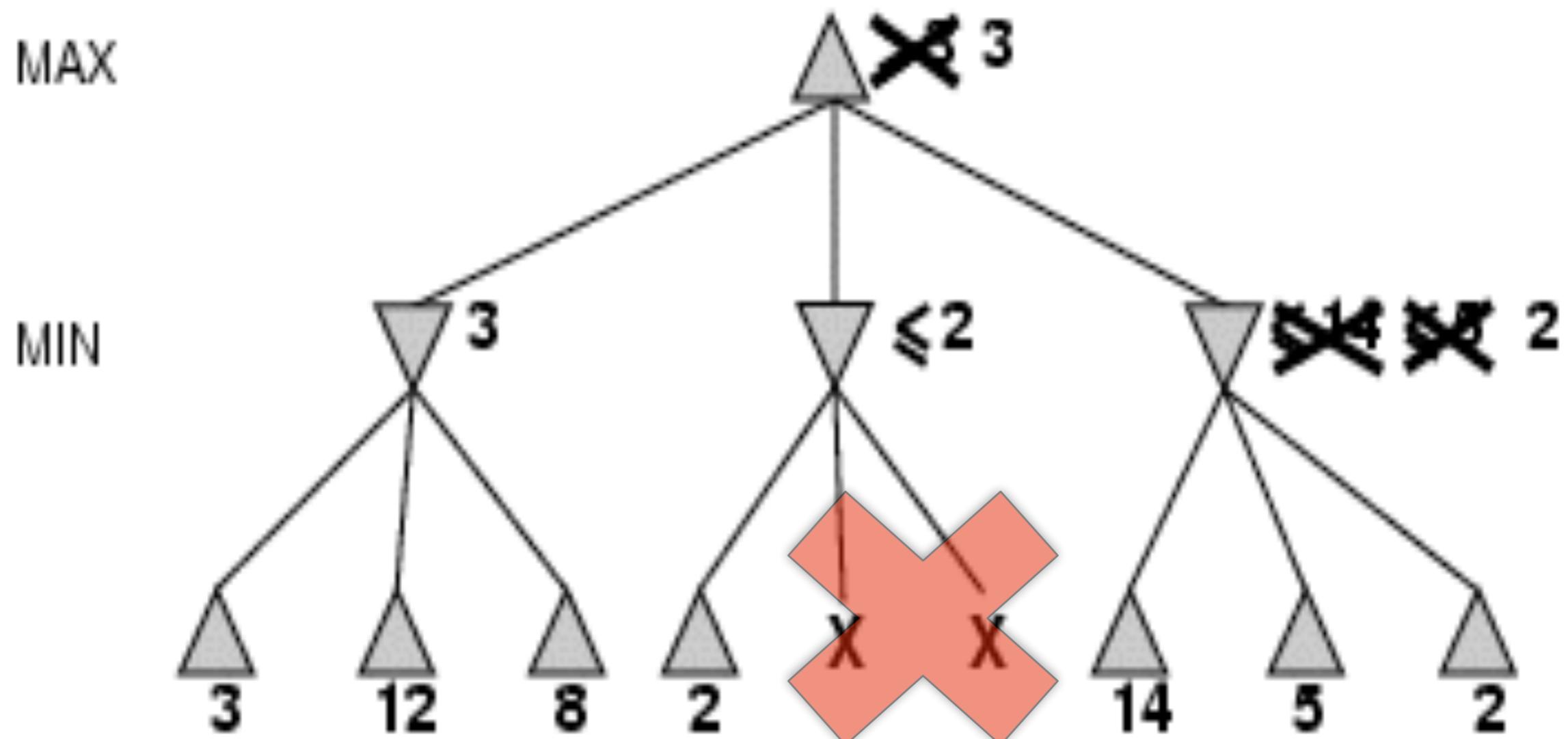
# Poda alfa-beta: ejemplo



# Poda alfa-beta: ejemplo



# Poda alfa-beta: ejemplo



# Minimax con poda α-β

- El recorrido se inicia llamando a la función valorMax con  $\alpha = -\infty$  y  $\beta = +\infty$ .
- En la función valorMax  $\alpha$  es el valor que se actualiza.
- En la función valorMin  $\beta$  es el valor que se actualiza.

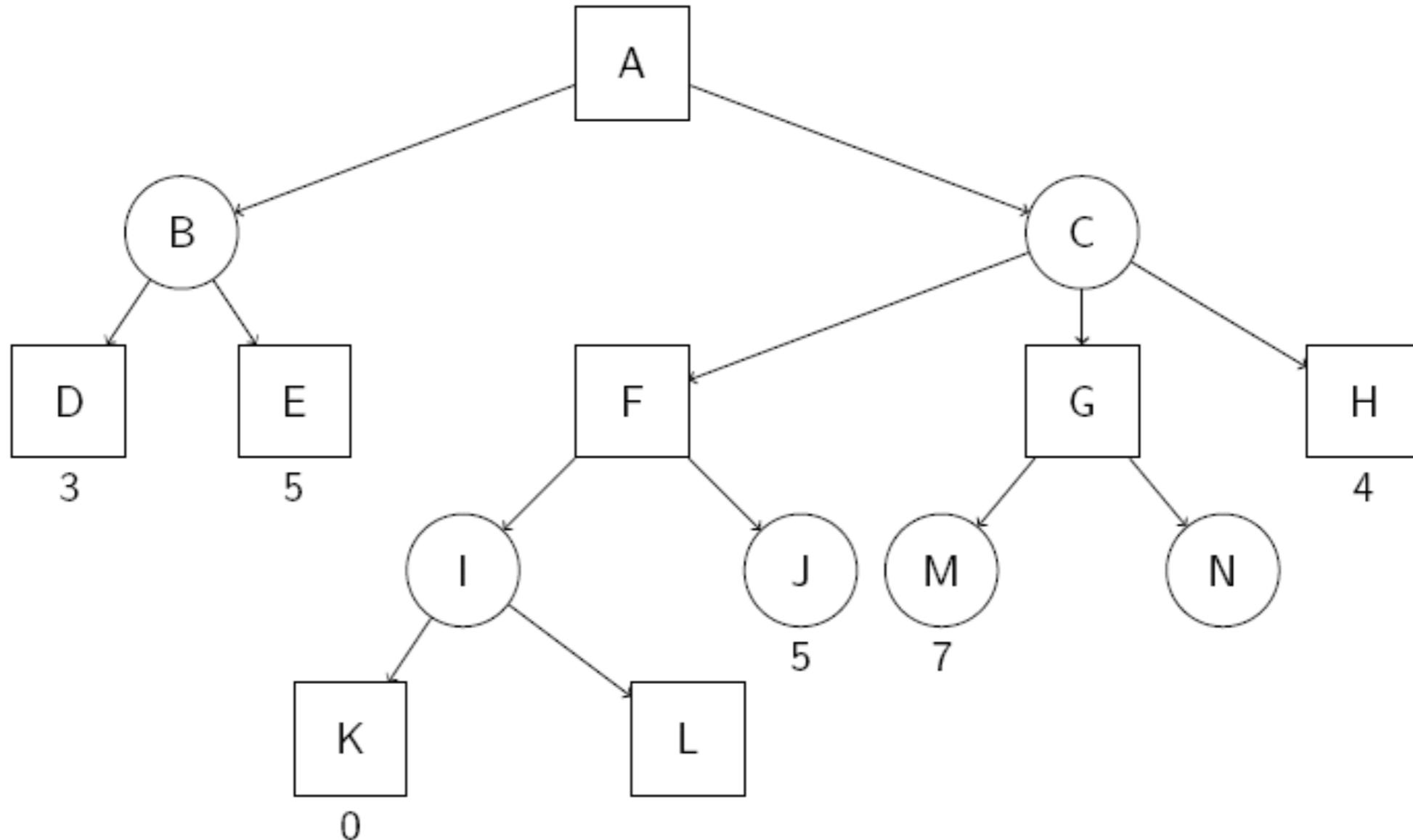
```
ion valorMax (g, alpha , beta) retorna entero
    si estado_terminal(g) entonces
        retorna( evaluacion(g))
    no
        para cada mov en movs_posibles(g)
            alfa=max( alfa , valorMin( aplicar(mov,g) , alfa , beta))
            si alfa >= beta entonces retorna(beta)
        para
        retorna( alfa)

    cion
```

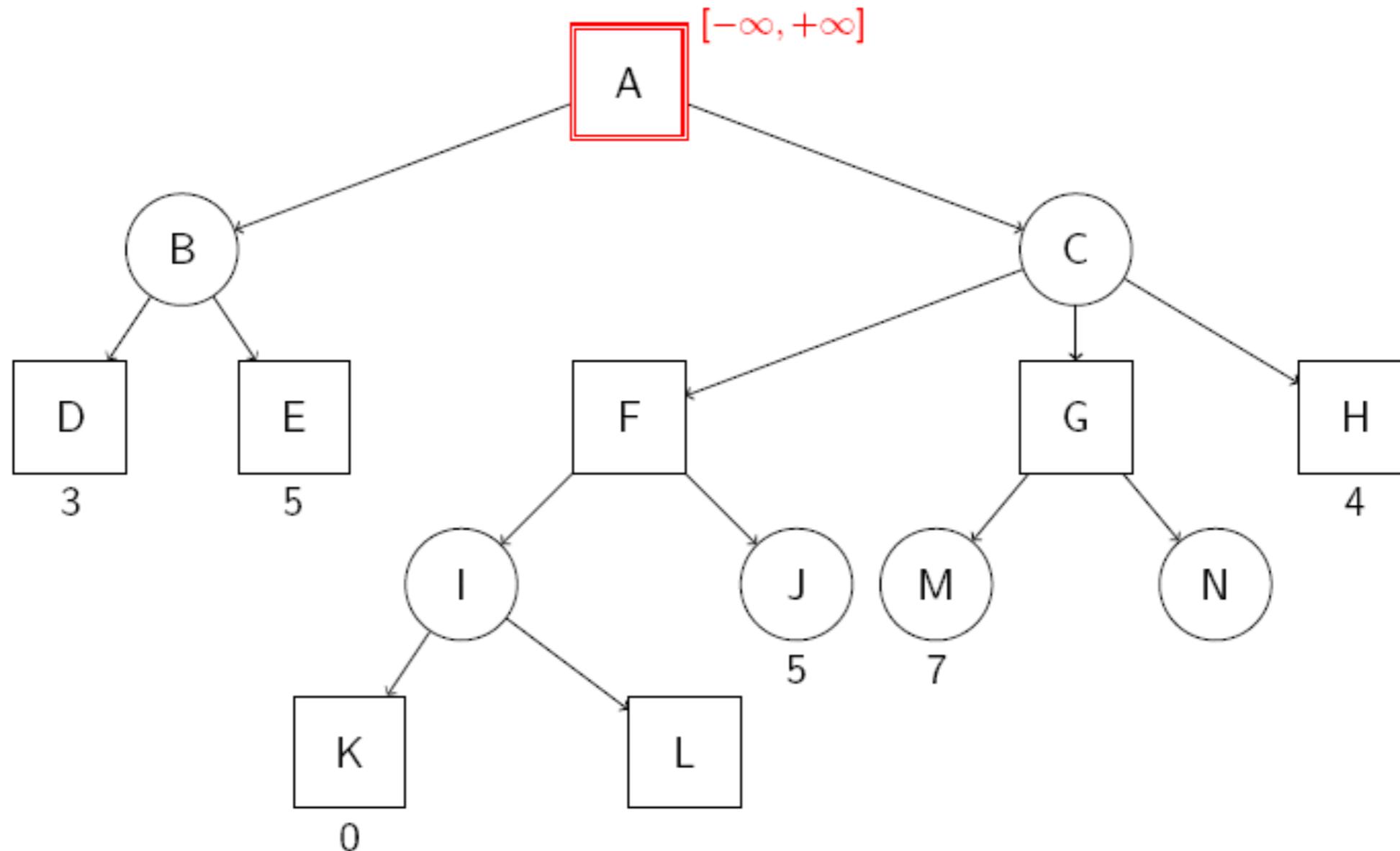
```
ion valorMin (g, alfa , beta) retorna entero
    si estado_terminal(g) entonces
        retorna( evaluacion(g))
    no
        para cada mov en movs_posibles(g)
            beta=min( beta , valorMax( aplicar(mov,g) , alfa , beta))
            si alfa >= beta entonces retorna( alfa)
        para
        retorna( beta)

    cion
```

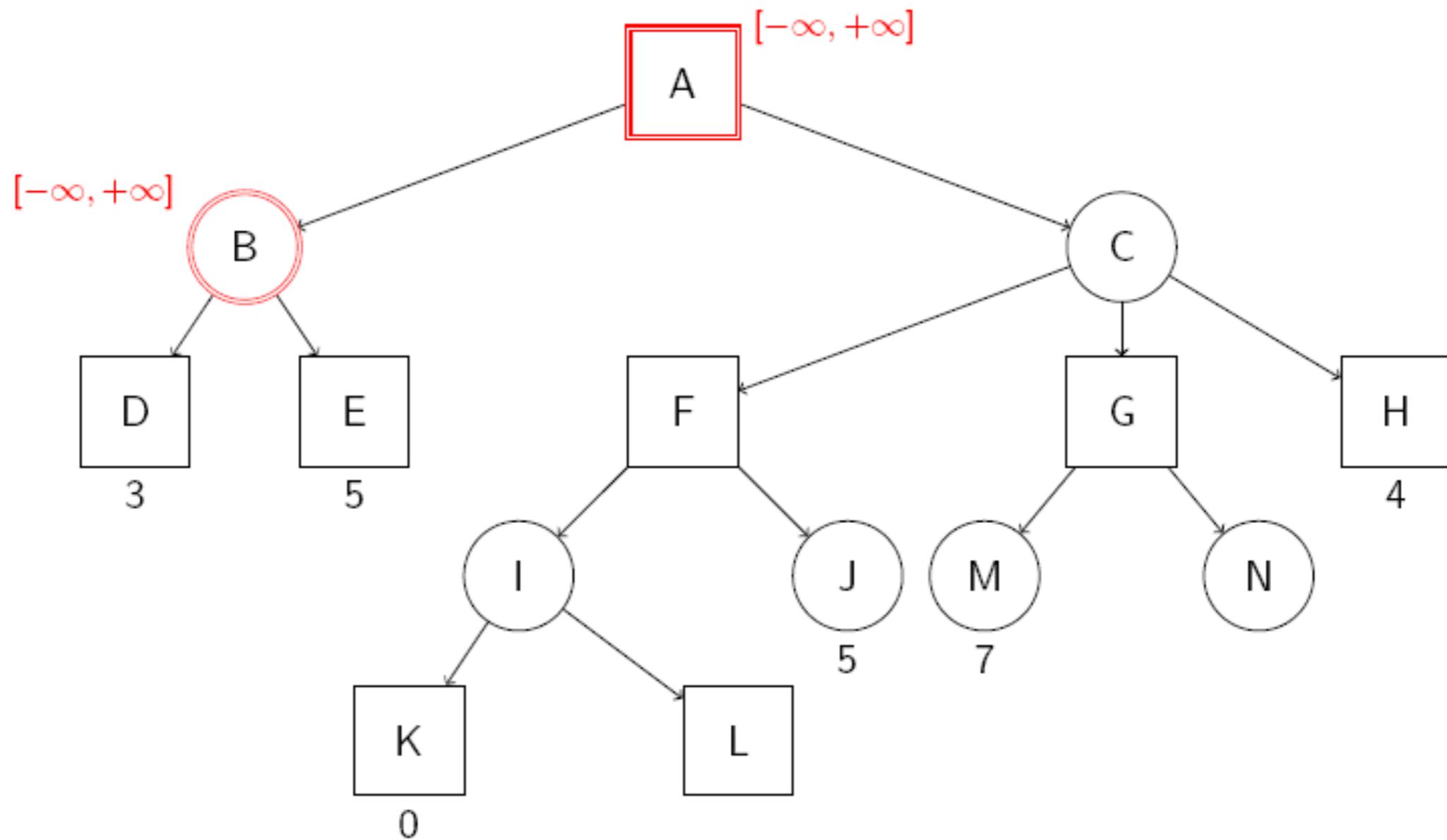
# Poda a- $\beta$ : ejemplo



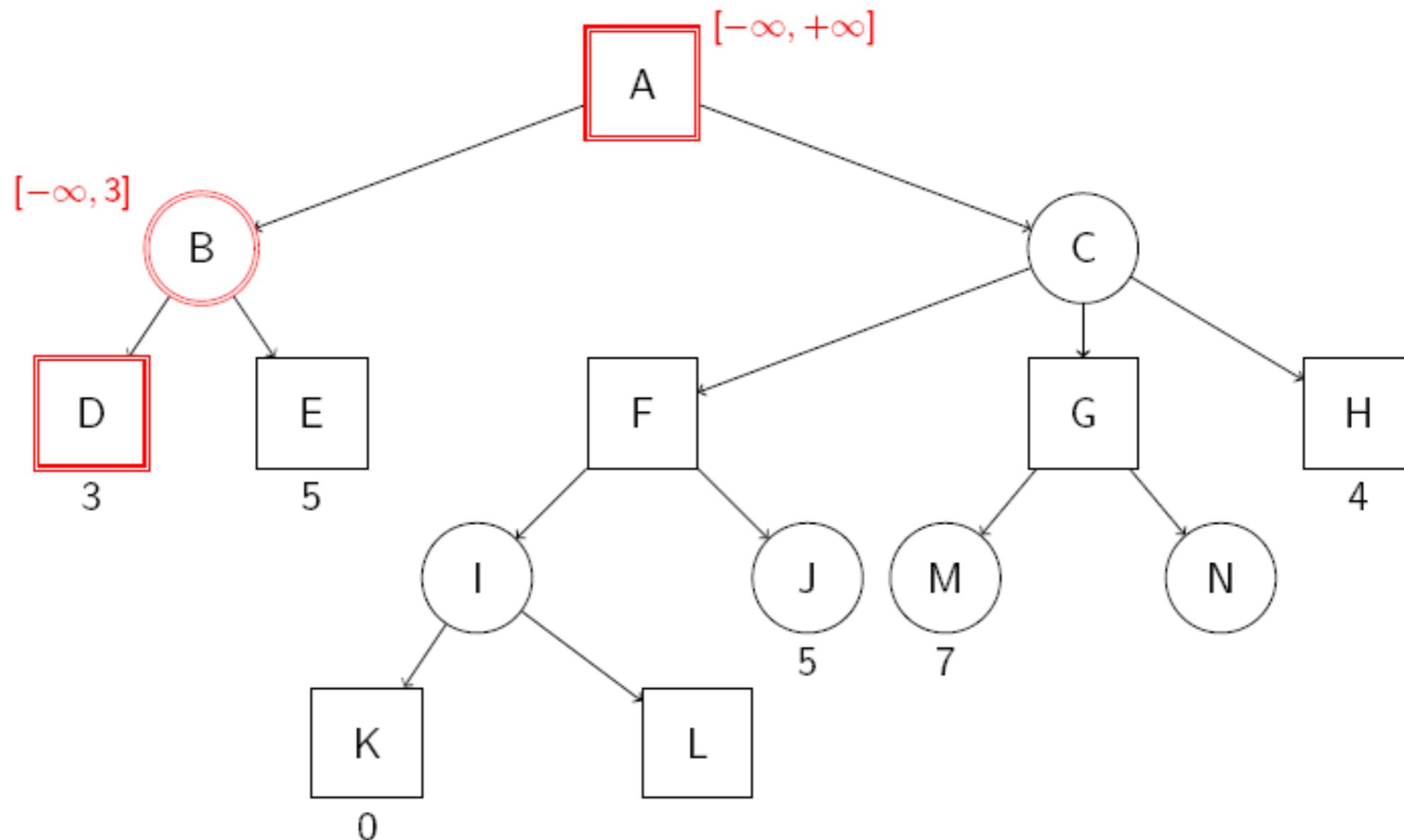
# Poda α-β: ejemplo



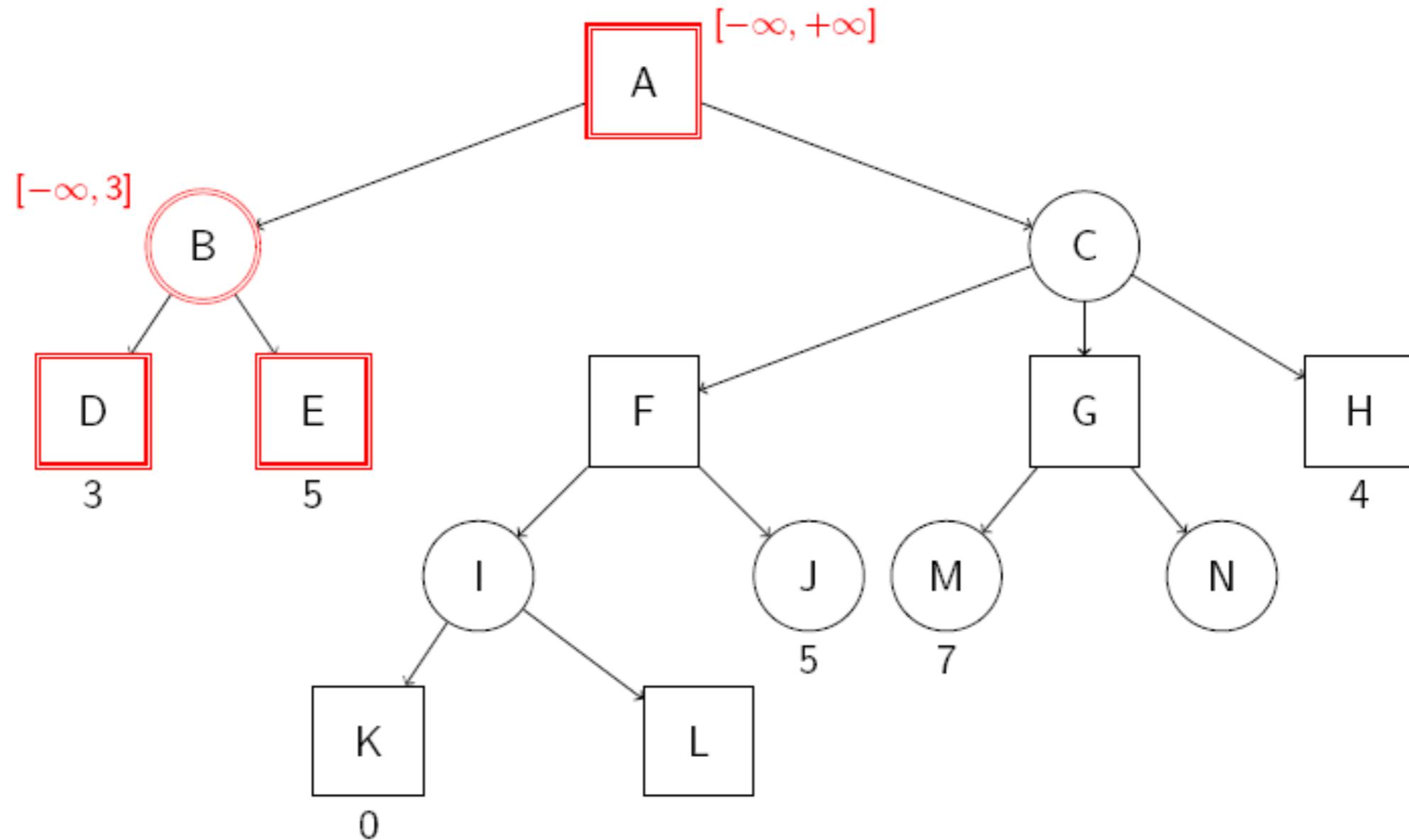
# Poda α-β: ejemplo



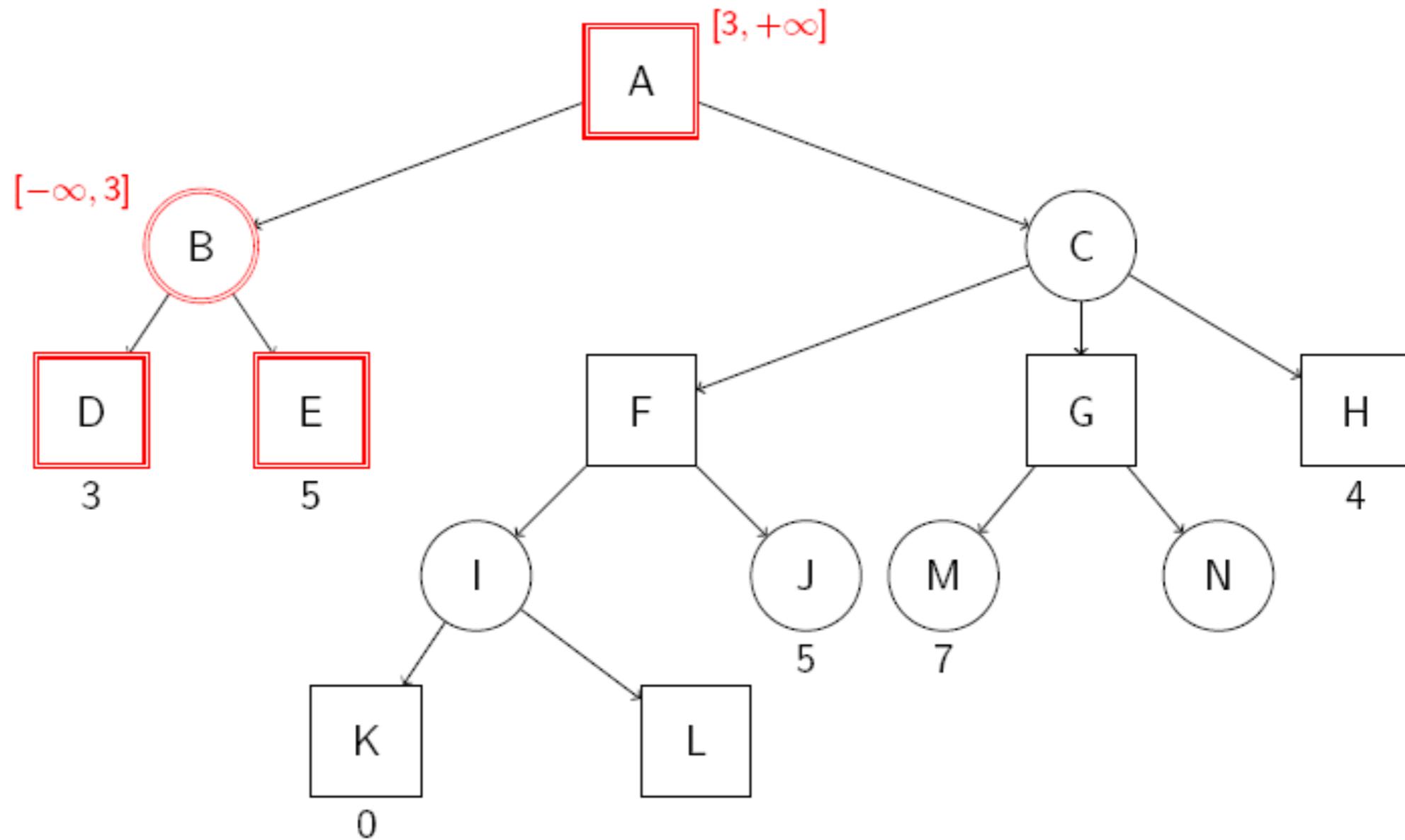
# Poda α-β: ejemplo



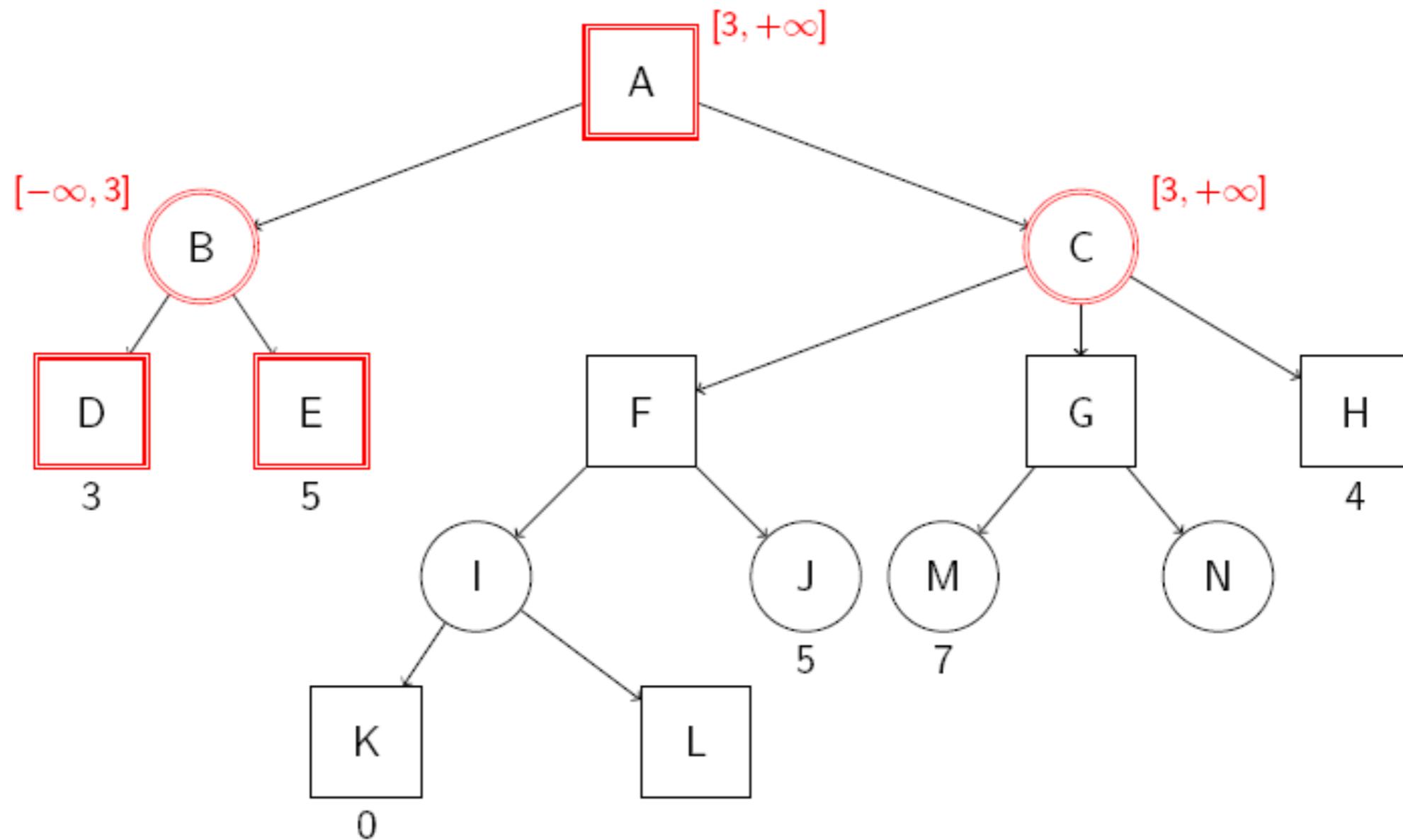
# Poda α-β: ejemplo



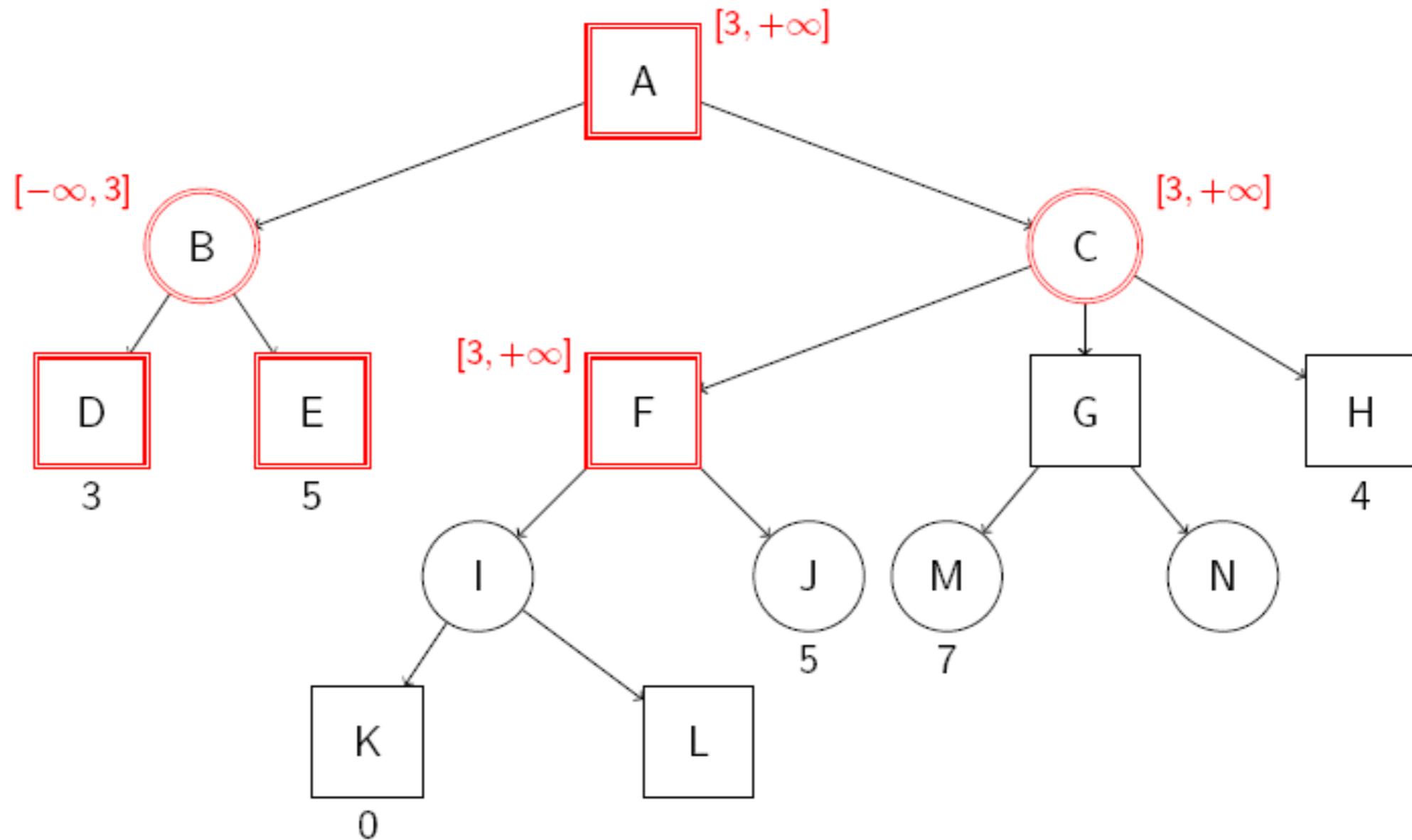
# Poda α-β: ejemplo



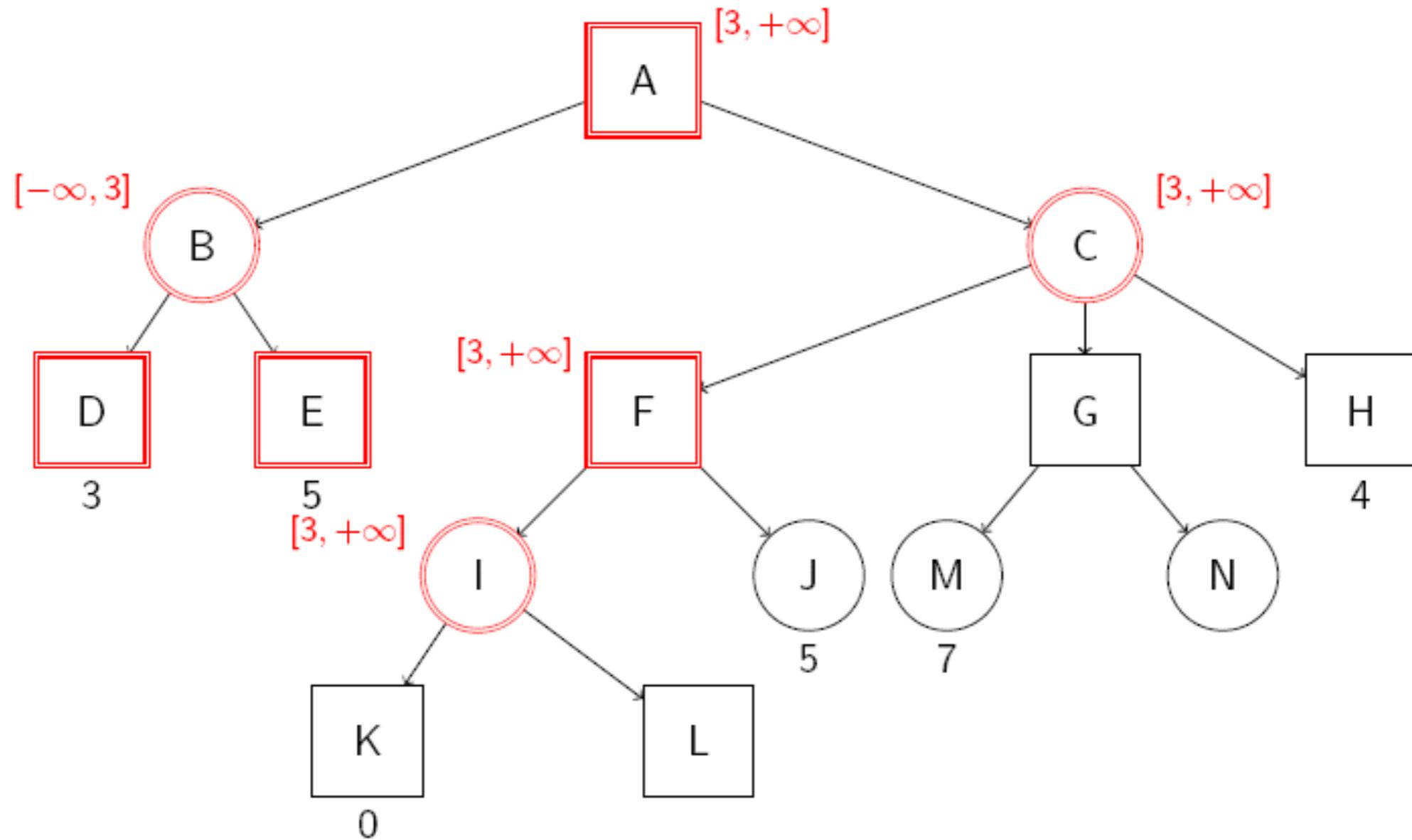
# Poda α-β: ejemplo



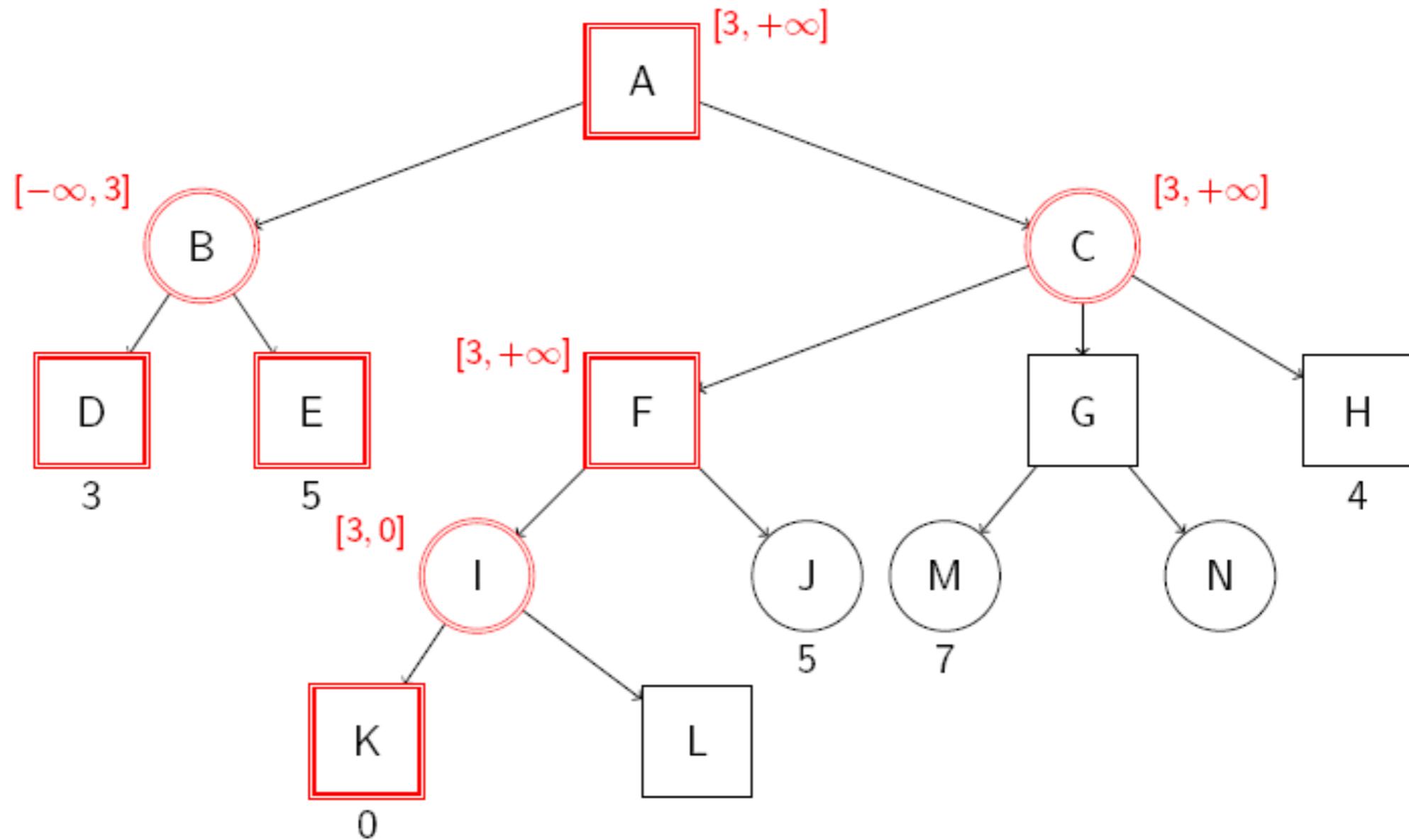
# Poda α-β: ejemplo



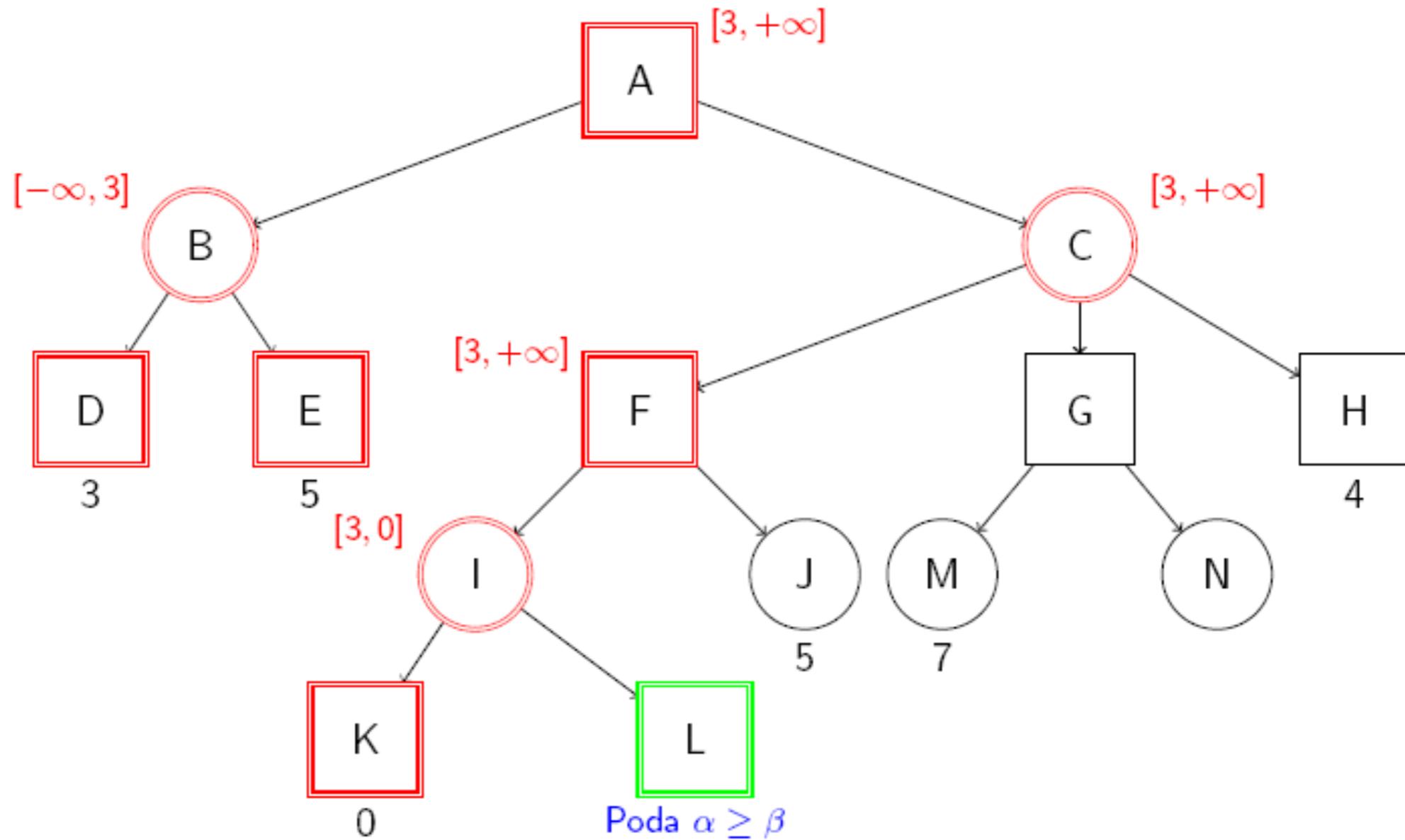
# Poda α-β: ejemplo



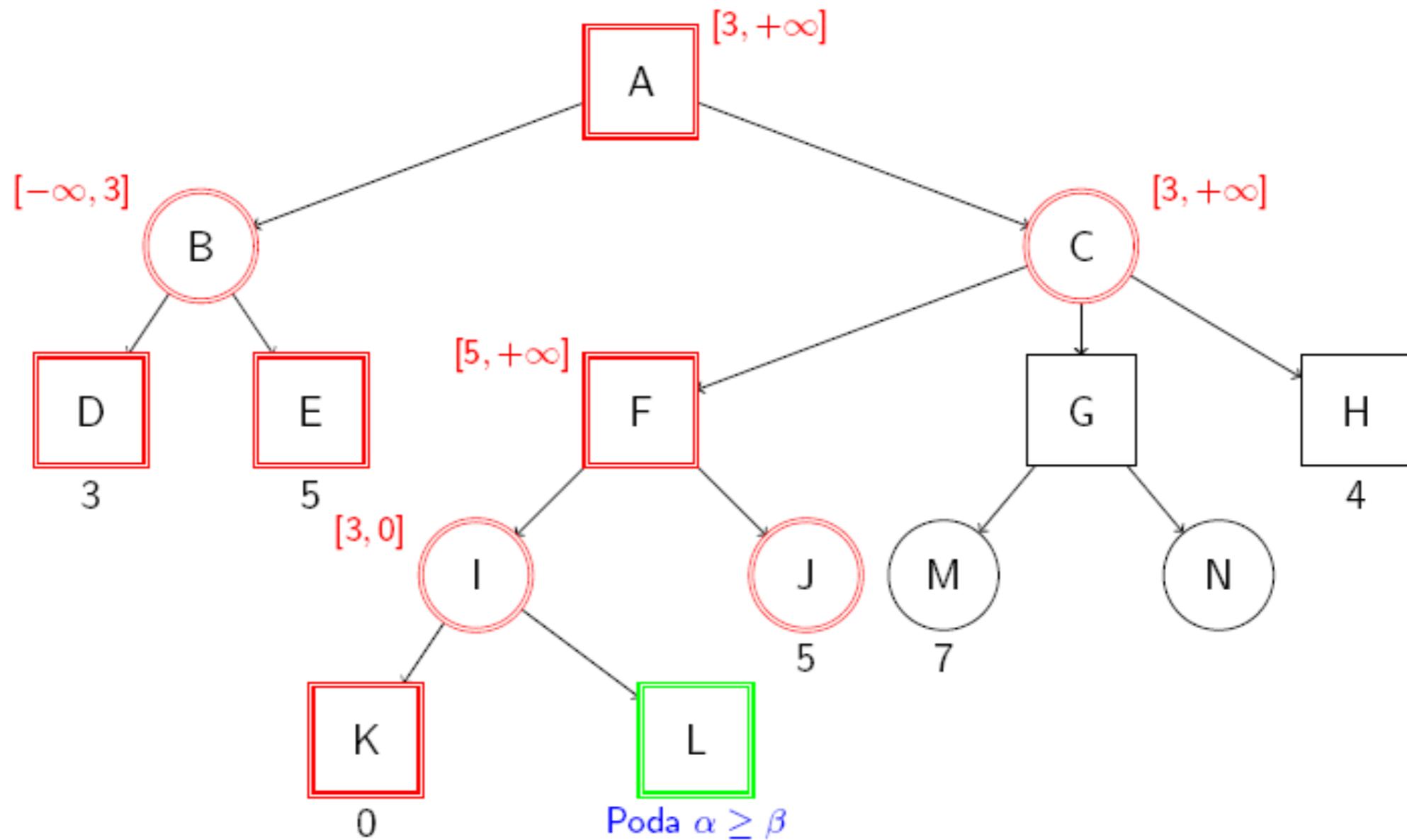
# Poda α-β: ejemplo



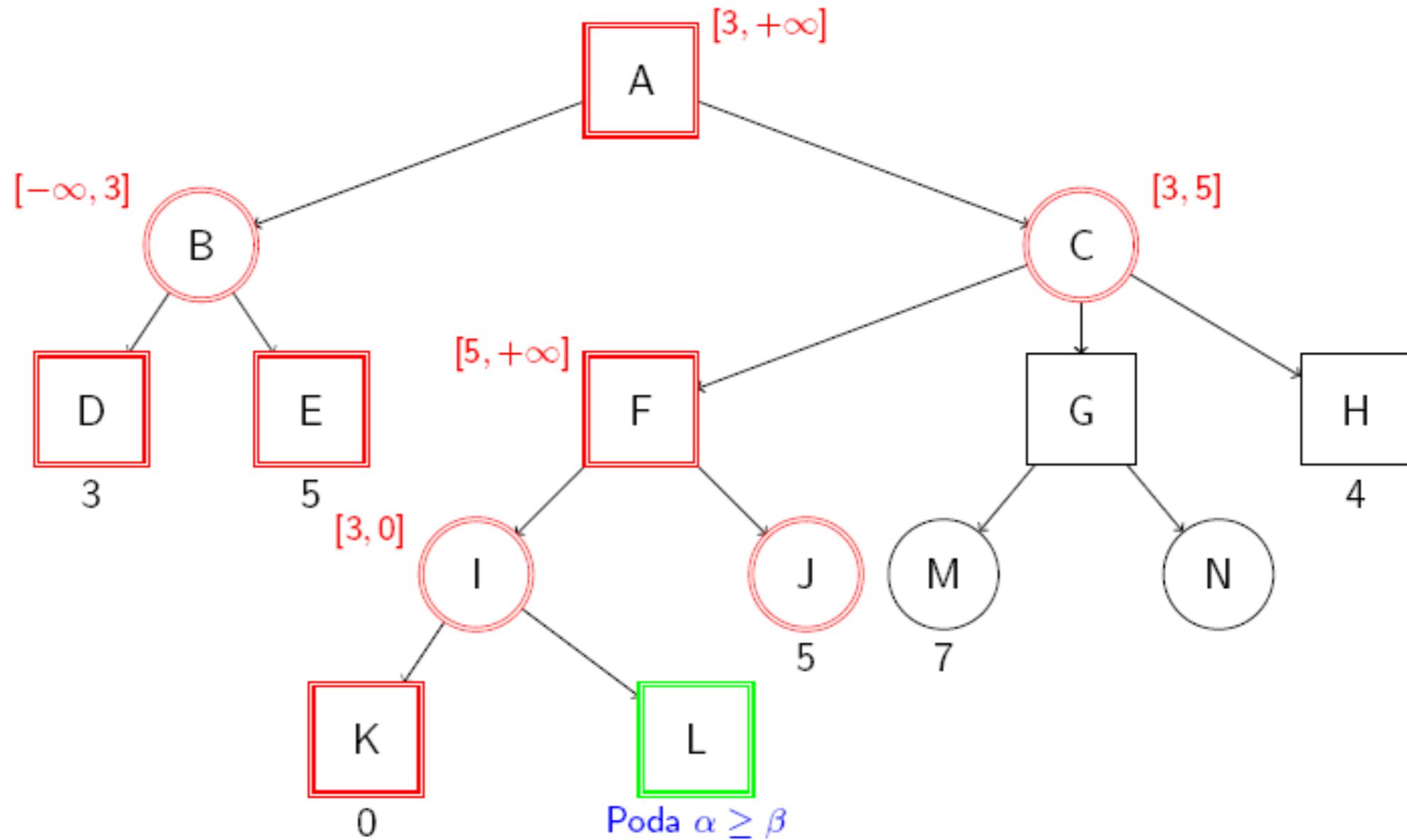
# Poda $\alpha$ - $\beta$ : ejemplo



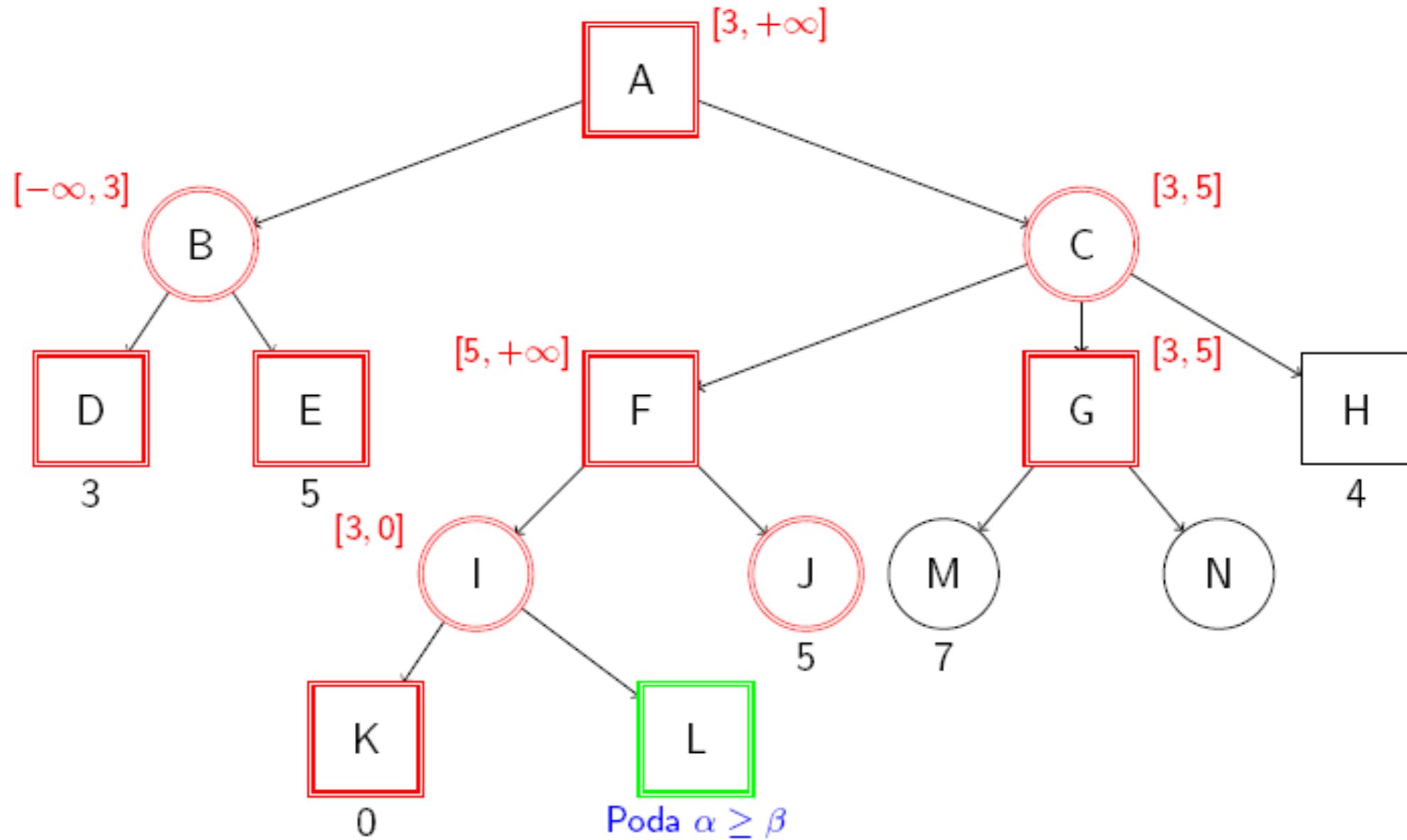
# Poda α-β: ejemplo



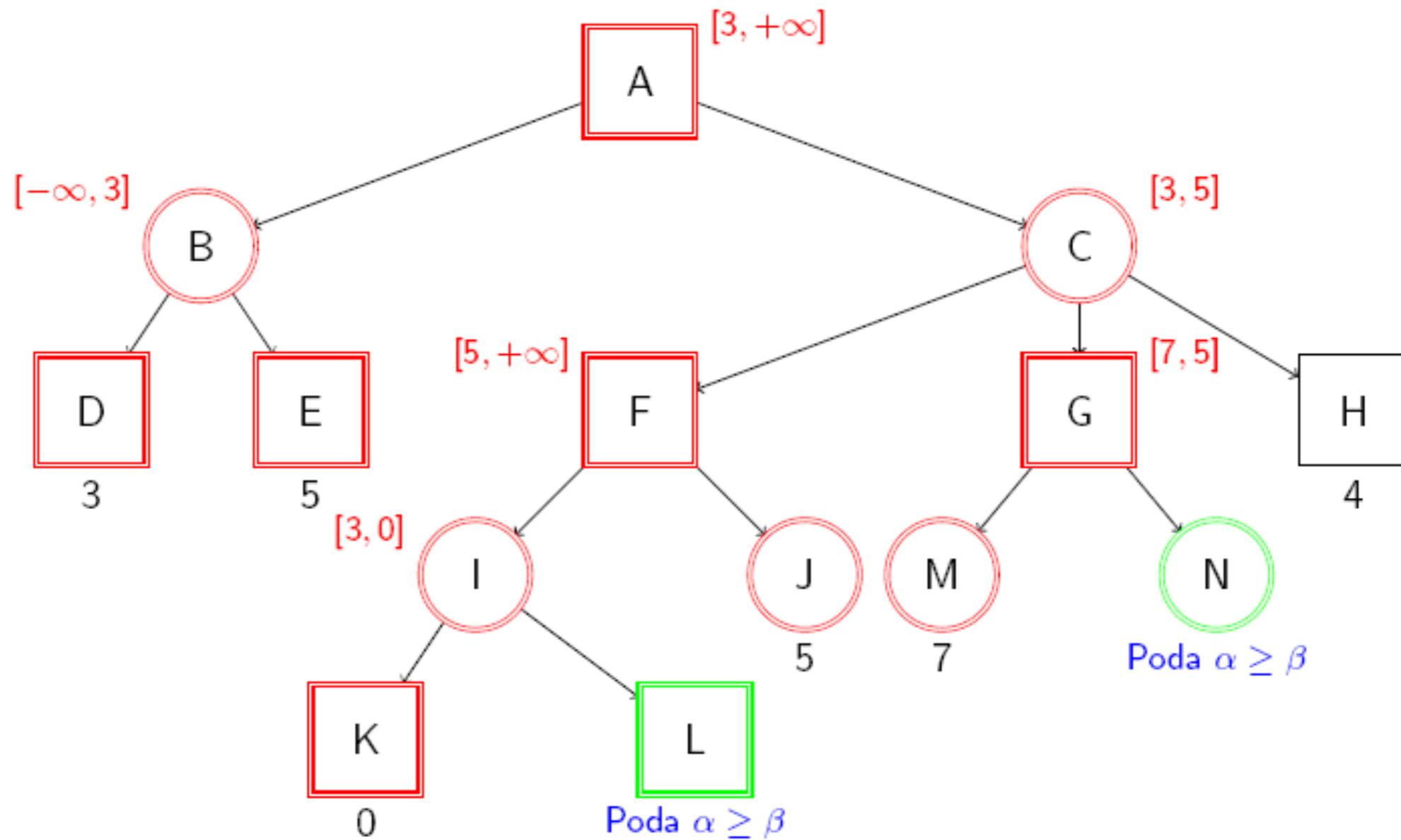
# Poda $\alpha$ - $\beta$ : ejemplo



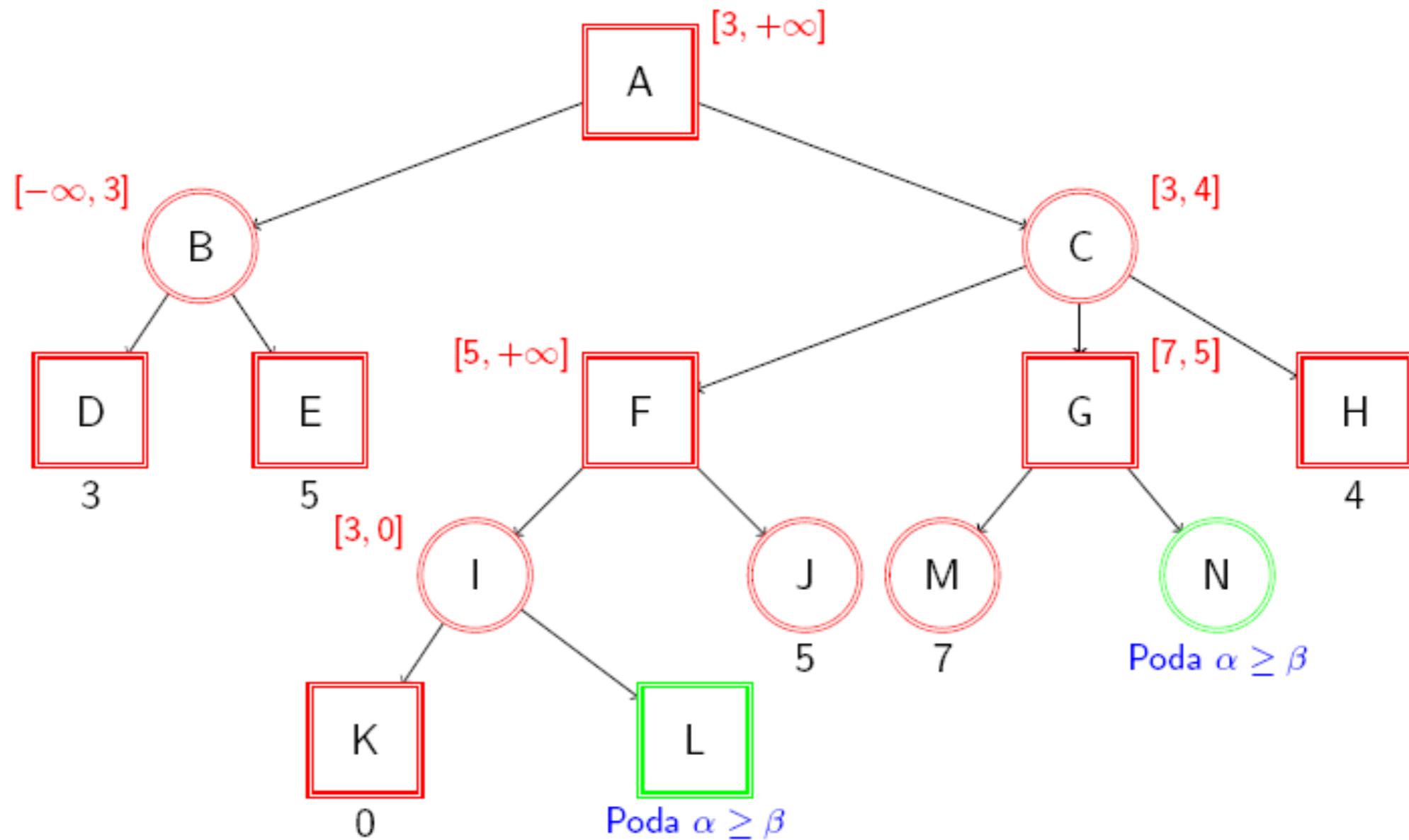
# Poda α-β: ejemplo



# Poda α-β: ejemplo



# Poda α-β: ejemplo



# Poda α-β: ejemplo

