# Estructuras de Datos

# Estructuras de Datos
# Objetivos

- Comprender la diferencia entre estructura de datos y TDA.
- Comprender el concepto de contenedor.
- Comprender el concepto de adaptador.
- Conocer los principales tipos de contenedores básicos.
- Entender el concepto de iterador en contenedores.
- Comprender el concepto de tabla hash.
- Comprender el concepto de árbol.
- Conocer las formas de recorrer un árbol.
- Comprender los conceptos de ABB, APO y AVL.
- Comprender el concepto de grafo.

# Estructuras de Datos
## Comprender la diferencia entre estructura de datos y TDA

Una **Estructura de Datos** es una forma lógica de organizar los datos. Por ejemplo, son estructuras de datos: una pila, una cola, un vector, una lista, un árbol, un grafo.

Un **Tipo de Dato Abstracto** es la implementación de una estructura de datos en una clase.

# Estructuras de Datos
# Comprender el concepto de contenedor

**Un contenedor es un TDA que puede contener un número arbitrario de otros TDAs**.

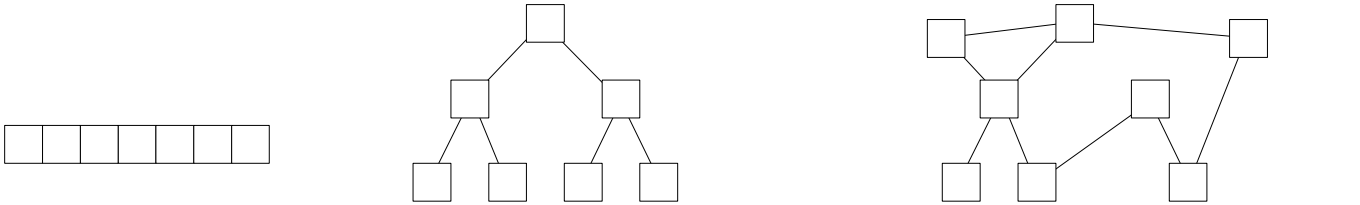
Un contenedor es una **colección de objetos** dotado de un conjunto de **métodos para gestionarlos** (acceder a ellos, eliminarlos del contenedor, añadir nuevos objetos, buscar, etc.).

Ofrecen también unas herramientas llamadas **iteradores**, para recorrerlos y poder revisar los objetos almacenados en ellos.

# Estructuras de Datos
# Comprender el concepto de contenedor

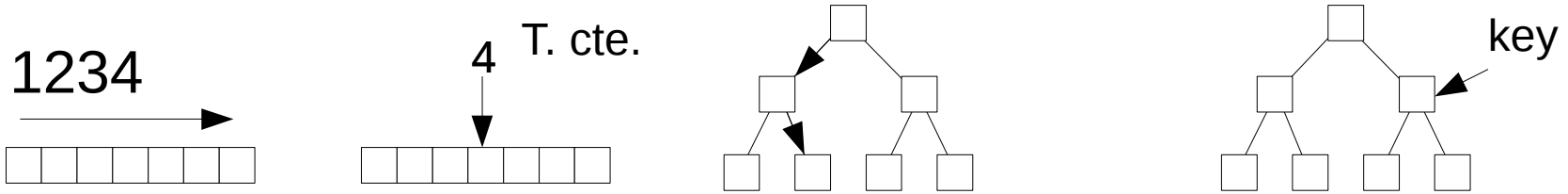
Los contenedores se pueden **clasificar** según distintos criterios:

- Forma de **organización**:



(lineal) (jerárquica) (interconexión total)

- Formas de **acceder a los componentes**:




(secuencial) (directa) (navegación) (clave/asociativos)

# Estructuras de Datos
# Comprender el concepto de contenedor

El **conjunto de operaciones** asociadas con los contenedores de cada tipo **suele ser común a todos ellos**, existiendo **operaciones adicionales** atendiendo a peculiaridades particulares.

Los contenedores pueden contener elementos de un Tipo de Dato Abstracto determinado, que puede ser a su vez otro tipo de contenedor (por ejemplo, vectores de vectores).

# Estructuras de Datos
# Comprender el concepto de adaptador

Un **adaptador** es un contenedor que está fundamentado en otro contenedor, y permite cambiar la interfaz del segundo (es decir, tiene una funcionalidad distinta).

Por ejemplo, en lugar de permitir el acceso libre a todos los elementos del contenedor, un adaptador puede limitar este acceso al primero o último introducido.

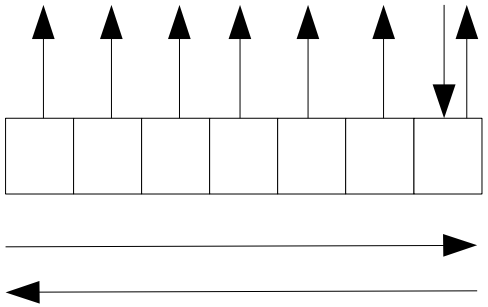
# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

## El TDA Vector dinámico (vector) *no adaptador*:

Almacena sus elementos en forma de sucesión secuencial.

Permite introducir o eliminar elementos eficientemente al final del vector (sin preocuparse por el tamaño del mismo, ya que se expande y se contrae automáticamente) y poco eficientemente en cualquier otro lugar.

Permite acceder a elementos eficientemente en cualquier lugar identificándolos por su índice.



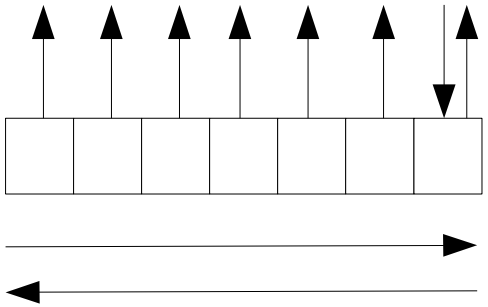
Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

## El TDA Vector dinámico (vector) *no adaptador*:

Ventajas:

- Se puede acceder a elementos individuales por su índice (O(1)).
- Se puede iterar por los elementos en cualquier orden (O(n)).
- Se pueden añadir y eliminar elementos eficientemente al final (O(1)).



Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

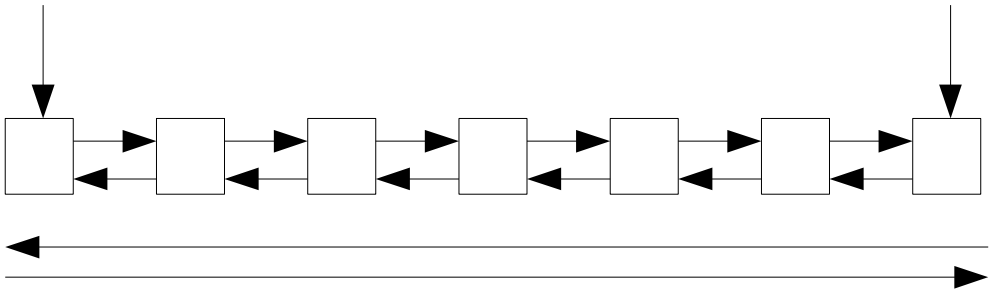
# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

## El TDA Lista (list) *no adaptador*:

Almacena sus elementos en forma de sucesión, y cada uno establece quién es el siguiente y el anterior.

Permite acceso secuencial a los elementos, hacia delante o hacia atrás.

No permite acceso directo, salvo al primer o último elemento.



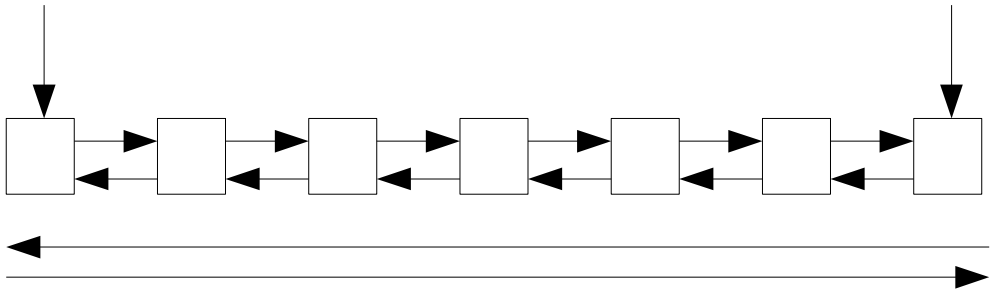
Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

## El TDA Lista (list) *no adaptador*:

Ventajas:

- Inserción y borrado eficiente (O(1)) de elementos en cualquier lugar de la lista.
- Se pueden mover elementos o grupos de elementos de una lista a otra eficientemente (O(1)).
- Recorrido o iteración por elementos hacia delante o atrás (O(n)).



Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

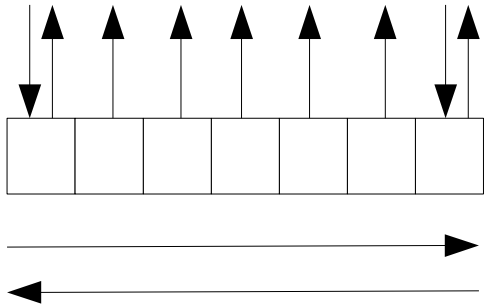
# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

## El TDA Cola con dos puntas (deque) *no adaptador*:

Almacena sus elementos en forma de sucesión secuencial (no contigua, no permite aritmética de punteros).

Permite introducir o eliminar elementos eficientemente por cualquiera de los dos extremos y poco eficientemente en cualquier otro lugar.

Permite acceder a elementos eficientemente en cualquier lugar identificándolos por su índice.



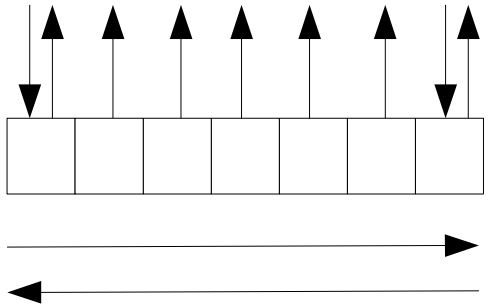
Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

## El TDA Cola con dos puntas (deque) *no adaptador*:

Ventajas:

- Se puede acceder a elementos individuales por su índice (O(1)).
- Se puede iterar por los elementos en cualquier orden (O(n)).
- Se pueden añadir y eliminar elementos eficientemente por cualquiera de las dos puntas (O(1)).



Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

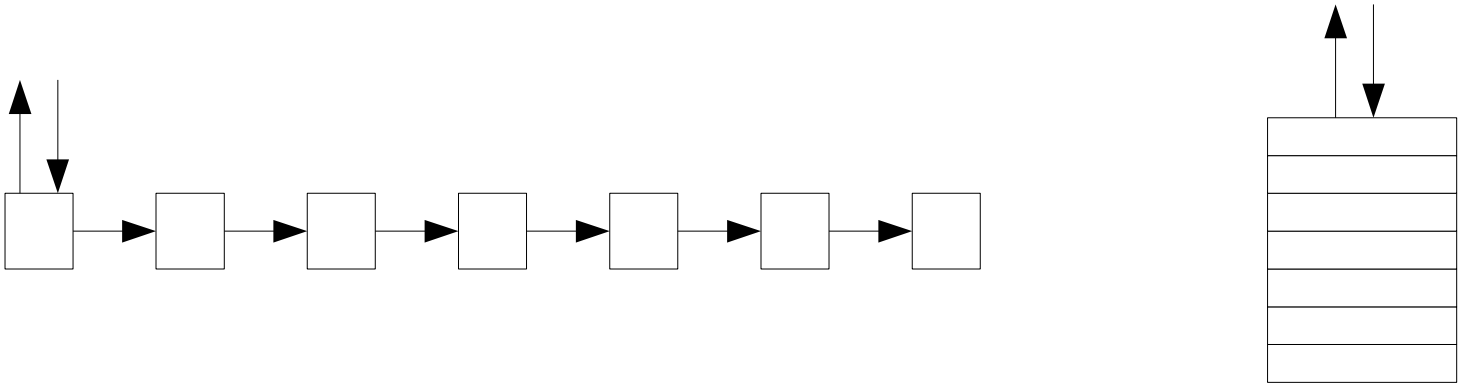
## El TDA Pila (stack):

Almacena sus elementos en forma de sucesión, y cada uno establece quién es el siguiente.

Permite introducir, acceder y borrar elementos **solo por un mismo extremo**, denominado tope (top).

Filosofía LIFO: Last-In First-Out.

Ejemplo: pila de platos en el fregadero.



Solo se puede acceder a un elemento en un instante <-> Fundamentado en otro contenedor.
Requiere poder insertar, acceder y eliminar elementos al final -> vector, list, o deque

# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

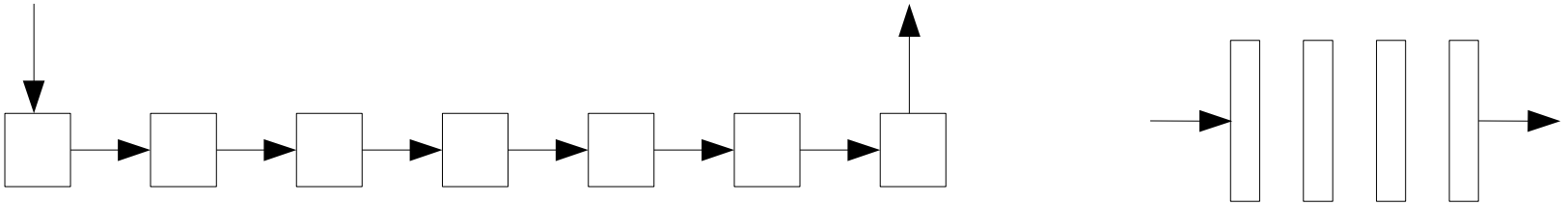
## El TDA Cola (queue):

Almacena sus elementos en forma de sucesión, y cada uno establece quién es el siguiente.

Permite introducir elementos sólo por un extremo (final), y permite recuperar/borrar el otro extremo (frente).

Filosofía FIFO: First-In First-Out.

Ejemplo: cola para entrar al cine



Solo se puede acceder a un elemento en un instante <-> Fundamentado en otro contenedor.
Requiere insertar elementos por un extremo y eliminar por otro -> list o deque, **NO vector (ineficiente)**

# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

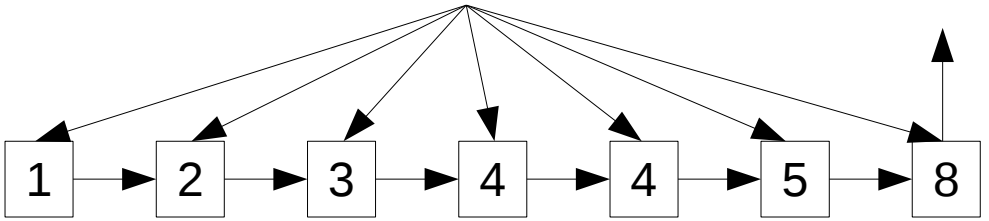
## El TDA Cola con prioridad (priority_queue):

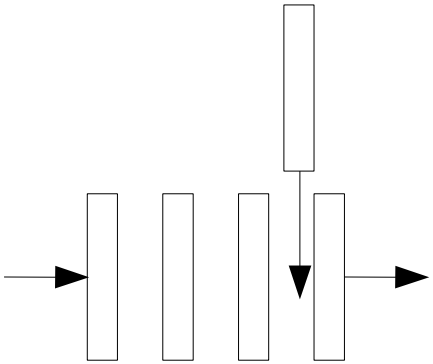
Es una cola en la que al insertar elementos, se ordenan por prioridad.

Primero se obtendrían los elementos de mayor prioridad.

La prioridad es por defecto el fruto de comparar elementos con el operador <, aunque se pueden definir otras prioridades.

¡¡¡OJO!!!



Solo se puede acceder a un elemento en un instante <-> Fundamentado en otro contenedor.
Requiere acceso aleatorio (búsqueda eficiente para insertar) -> Fundamentado en vector o deque, **NO listas**.
**INTERNAMENTE es un APO sobre vector o deque (ver las transparencias de APO más adelante). O(logn)**

# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

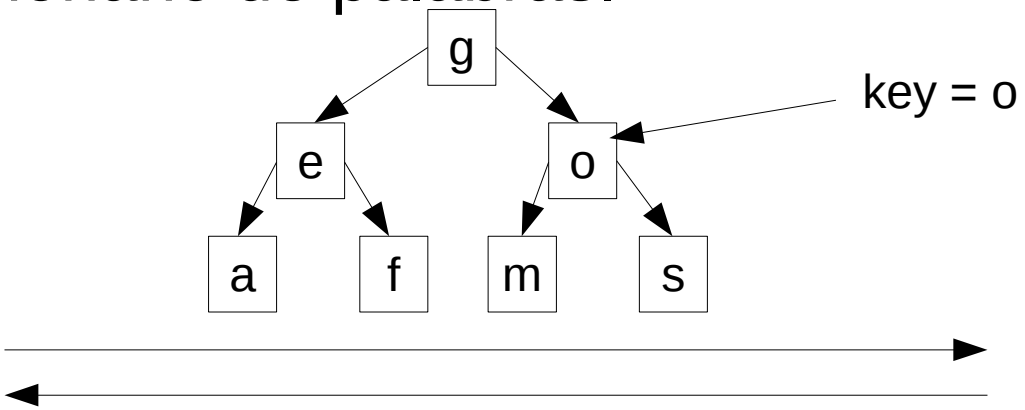
## El TDA Diccionario (map):

Contenedor asociativo, relaciona claves con objetos.

La clave **no puede repetirse** para dos elementos del diccionario, aunque varias claves sí pueden apuntar al mismo objeto.

Podemos introducir pares clave-objeto y buscar por clave.

Ejemplo: Diccionario de palabras.




Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

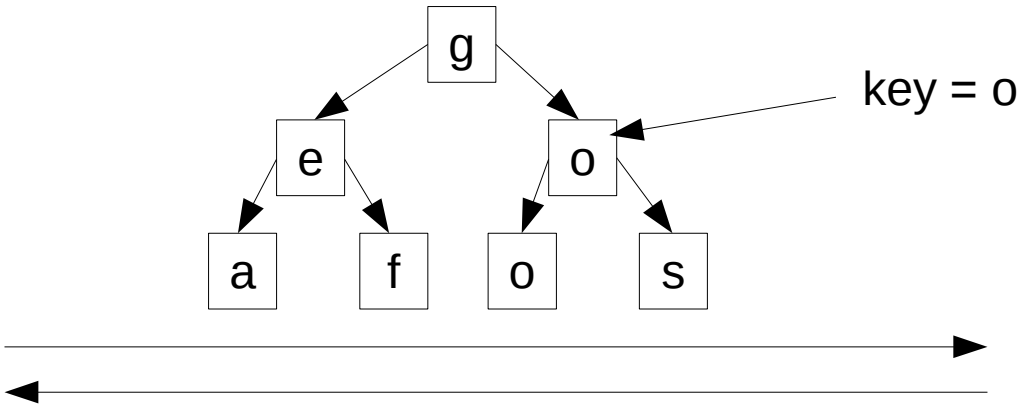
# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

## El TDA Diccionario múltiple (multimap):

Contenedor asociativo, relaciona claves con objetos.

La clave **puede repetirse** para dos elementos del diccionario, y varias claves pueden apuntar al mismo objeto.

Podemos introducir pares clave-objeto y buscar por clave.




Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

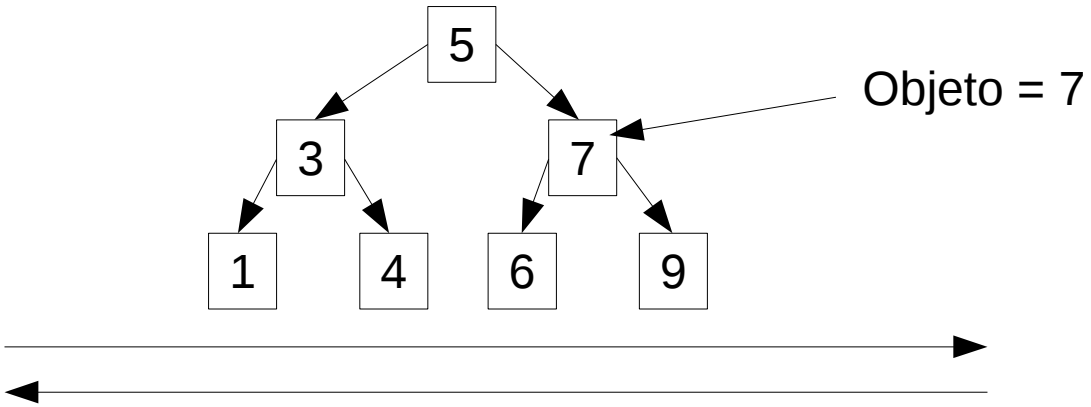
## El TDA Conjunto (set):

Contenedor asociativo, almacena relaciones claves->objetos.

No se pueden incluir objetos repetidos.

Podemos introducir objetos y buscar por objetos, ¡no por claves!

Se almacena como árbol binario.




Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

# Estructuras de Datos
# Conocer los principales tipos de contenedores básicos

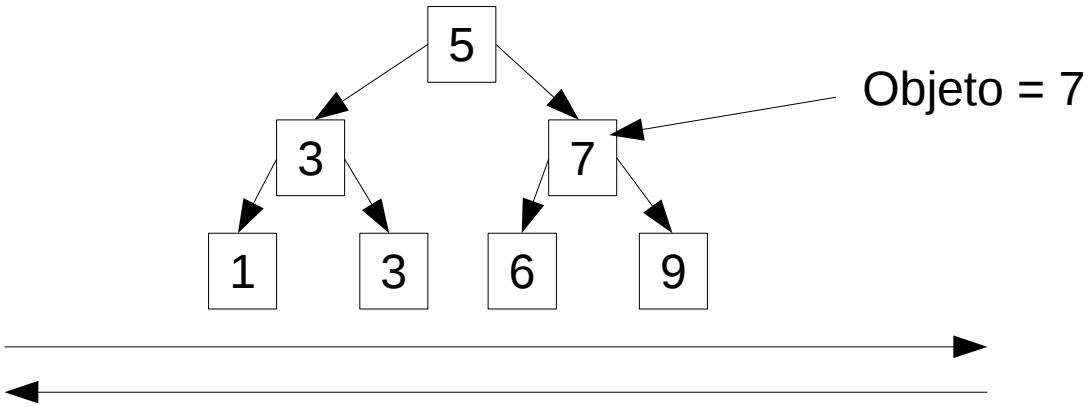
## El TDA Bolsa (multiset):

Contenedor asociativo, almacena objetos.

Se pueden incluir objetos repetidos.

Podemos introducir objetos y buscar por objetos, ¡no por claves!

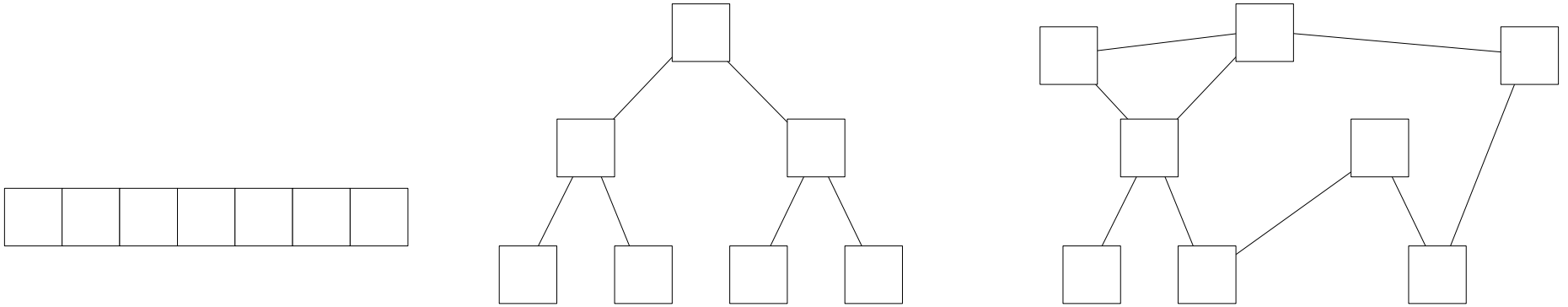
Ejemplo: Bolsa de fruta.



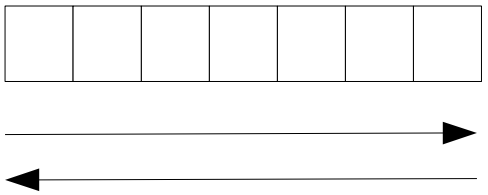

Se puede acceder a más de un elemento en un instante <-> Permite recorrido secuencial hacia delante o atrás

# Estructuras de Datos
# Entender el concepto de iterador en contenedores

Un **iterador** es una herramienta que permite recorrer, los elementos de un contenedor **independientemente de su organización interna**:



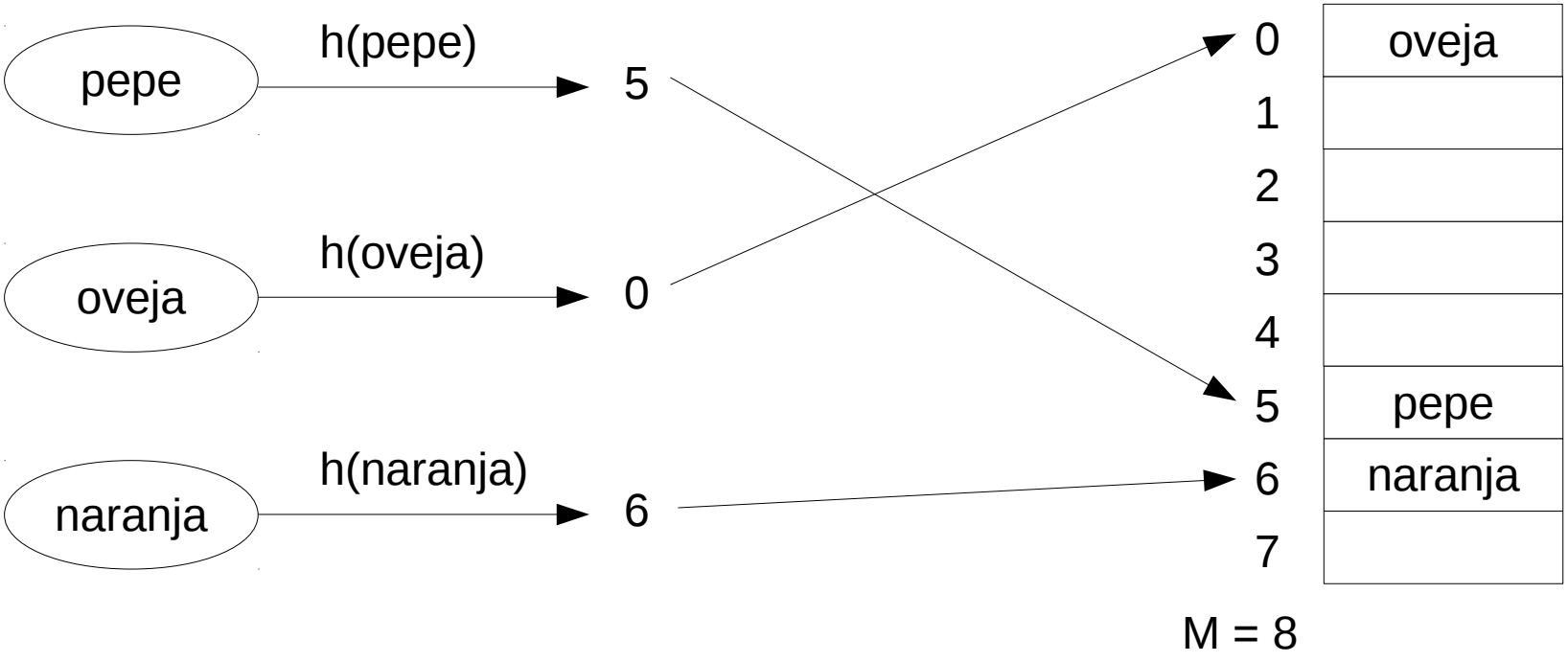
De forma aparentemente **secuencial**:



# Estructuras de Datos
# Comprender el concepto de tabla hash

Una **tabla hash** es un contenedor que calcula la posición a la que deben ir los elementos que se introducen aplicando una **función matemática** sobre las claves dadas para los mismos, en un tiempo O(1).

A esta función se la conoce como **función hash**.



# Estructuras de Datos
# Comprender el concepto de tabla hash

Si el conjunto de datos es fijo y predeterminado, y la función hash es biyectiva, hablamos de función **hash perfecta**.

Por lo general son **sobreyectivas**: Podemos obtener el mismo valor para dos claves distintas, por lo que hay colisiones en el valor de la función hash.

Podemos resolver colisiones realizando **encadenamiento separado**, o **direccionamiento abierto**.

Las funciones hash se deben definir tal que:

- Den valor para todos y cada uno de los posibles valores.
- Distribuyan de la forma lo más aleatoria posible las claves.
- Minimicen el número de colisiones.

Si la ocupación de la tabla hash es próxima al 100% se aumenta su tamaño (**rehashing**) y recolocan sus elementos.

# Estructuras de Datos
# Comprender el concepto de tabla hash

Diferentes métodos de **funciones hash**:

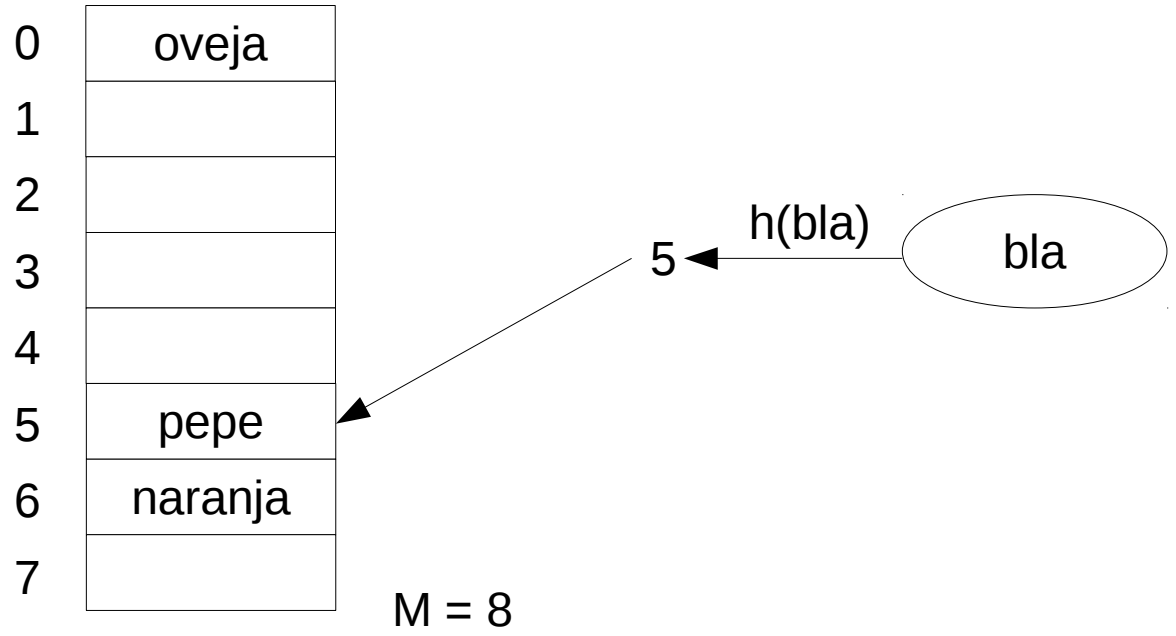
- **Multiplicación**: Multiplicar por un valor M y quedarnos con algunos de los bits de la expresión binaria del dato.
- **División**: Se calcula el resto módulo M a la clave, con M primo.
- **Multiplicación, suma y división**:
  h(x) = (**a**x+**b**)%M con a%M!=0.
- **Para los strings**:
  h(x) = (x[0]+...+x[n-1])%M.

# Estructuras de Datos
# Comprender el concepto de tabla hash

Técnica de **encadenamiento separado** para resolver colisiones:

Se resuelven insertándolas en listas. (vector de listas).

Se llama **factor de carga** al número medio de claves por lista.

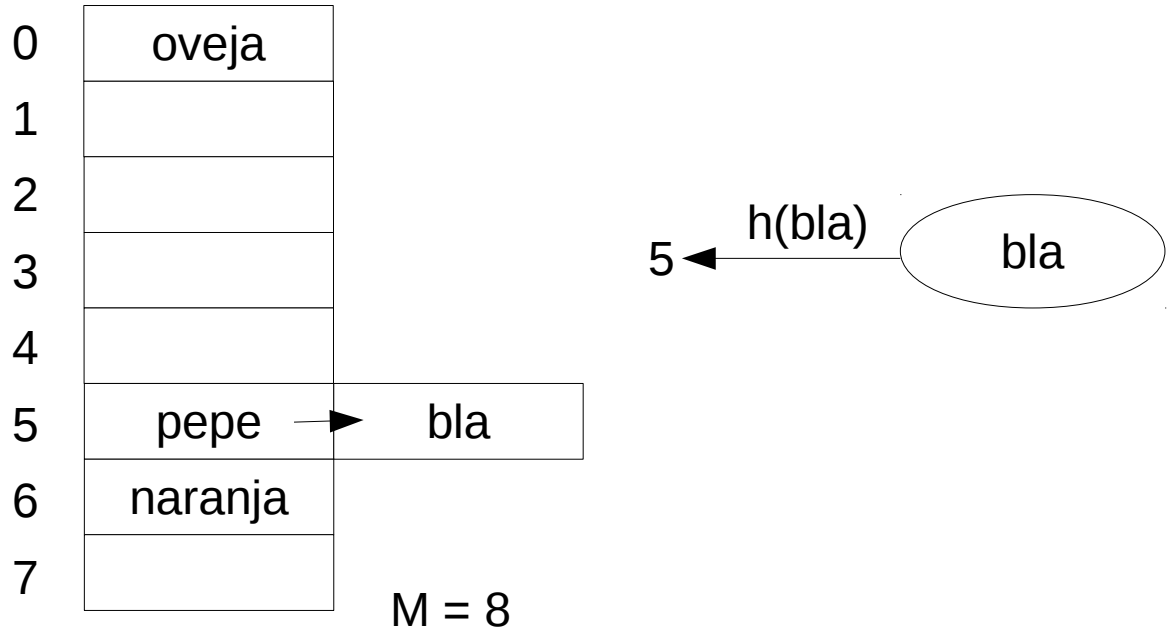


# Estructuras de Datos
# Comprender el concepto de tabla hash

Técnica de **encadenamiento separado** para resolver colisiones:

Se resuelven insertándolas en listas. (vector de listas).

Se llama **factor de carga** al número medio de claves por lista.



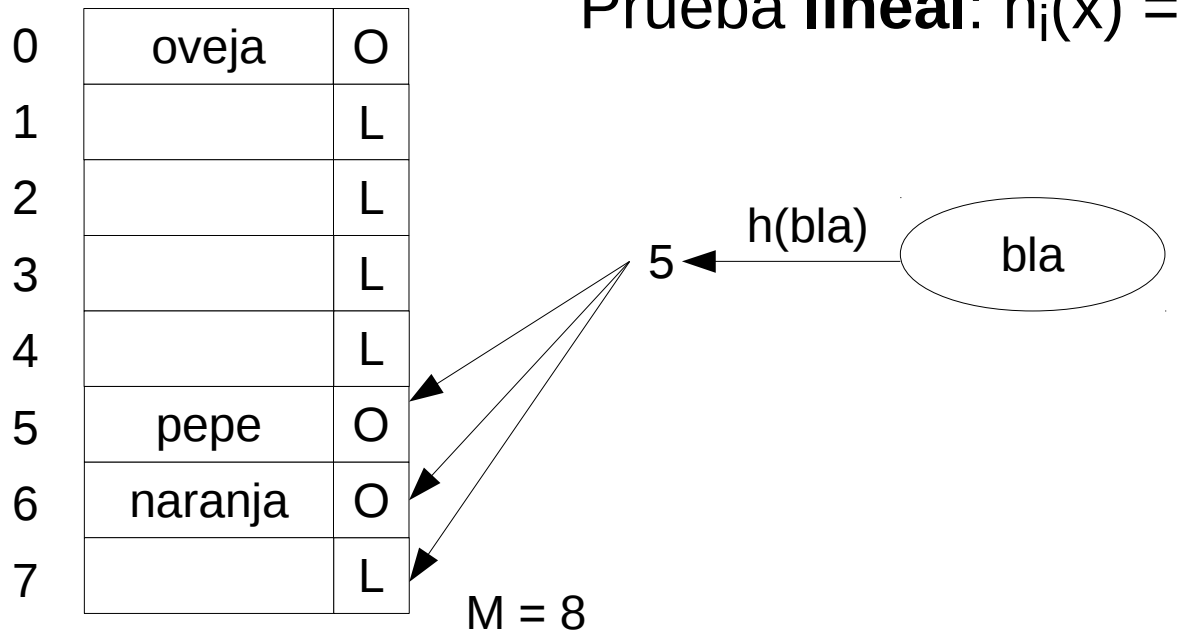
# Estructuras de Datos
# Comprender el concepto de tabla hash

Técnica de **direccionamiento abierto** para resolverlas:

Recordamos para cada entrada si está **Ocupada**, **Libre** o **Borrada**.

Volvemos a aplicar la función hash en base al intento i, o aplicamos otra distinta.

Prueba **lineal**: $h_i(x) = (h(x)+i)\%M$



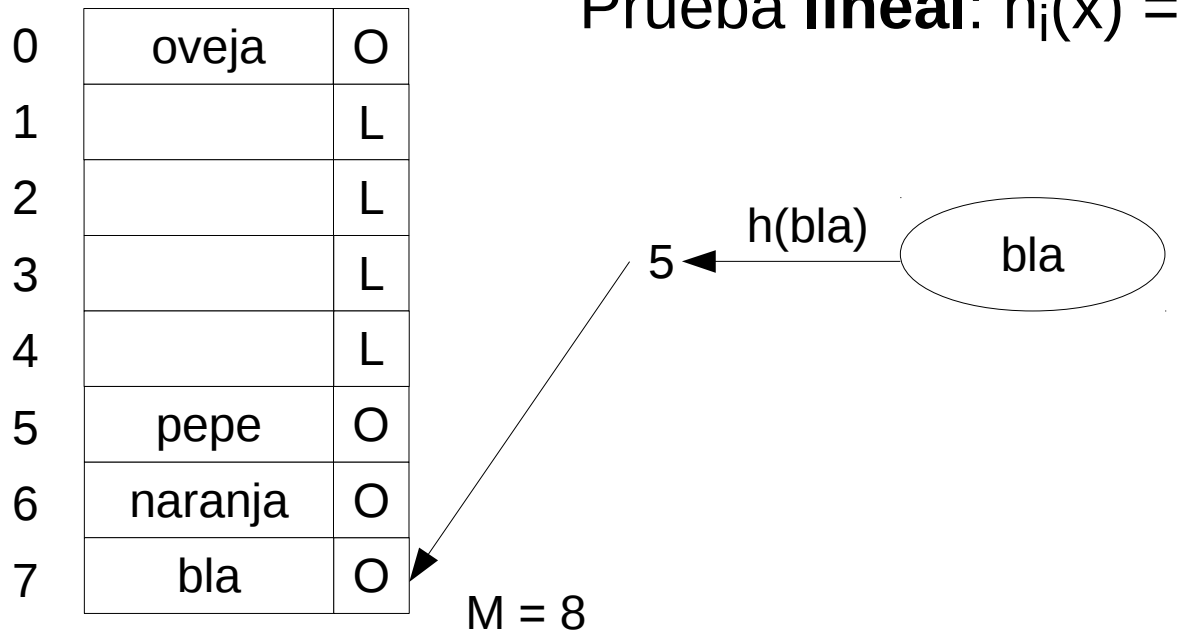
# Estructuras de Datos
# Comprender el concepto de tabla hash

Técnica de **direccionamiento abierto** para resolverlas:

Recordamos para cada entrada si está **Ocupada**, **Libre** o **Borrada**.

Volvemos a aplicar la función hash en base al intento i, o aplicamos otra distinta.

Prueba **lineal**: $h_i(x) = (h(x)+i)\%M$



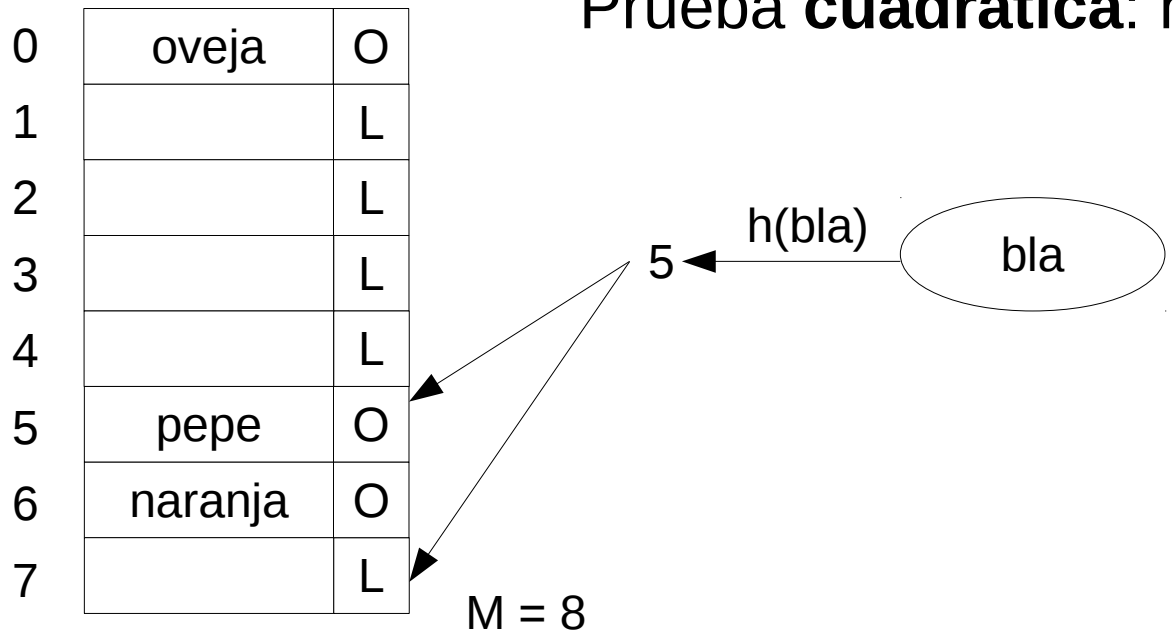
# Estructuras de Datos
# Comprender el concepto de tabla hash

Técnica de **direccionamiento abierto** para resolverlas:

Recordamos para cada entrada si está **Ocupada**, **Libre** o **Borrada**.

Volvemos a aplicar la función hash en base al intento i, o aplicamos otra distinta.

Prueba **cuadrática**: $h_i(x) = (h(x)+i^2)\%M$



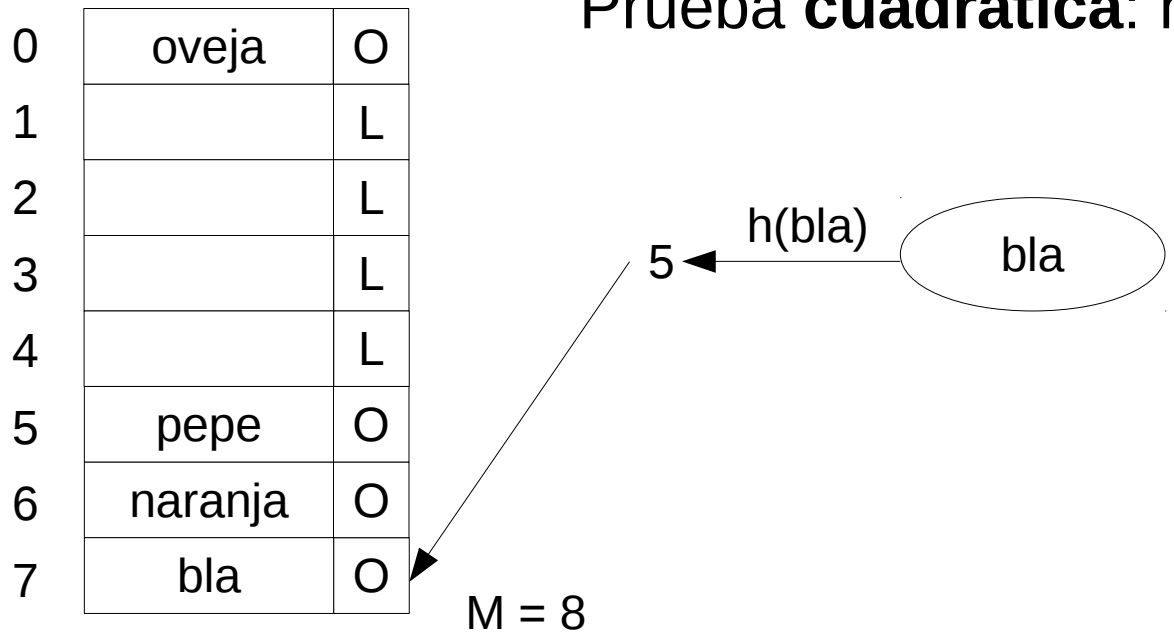
# Estructuras de Datos
# Comprender el concepto de tabla hash

Técnica de **direccionamiento abierto** para resolverlas:

Recordamos para cada entrada si está **Ocupada**, **Libre** o **Borrada**.

Volvemos a aplicar la función hash en base al intento i, o aplicamos otra distinta.

Prueba **cuadrática**: $h_i(x) = (h(x)+i^2)\%M$

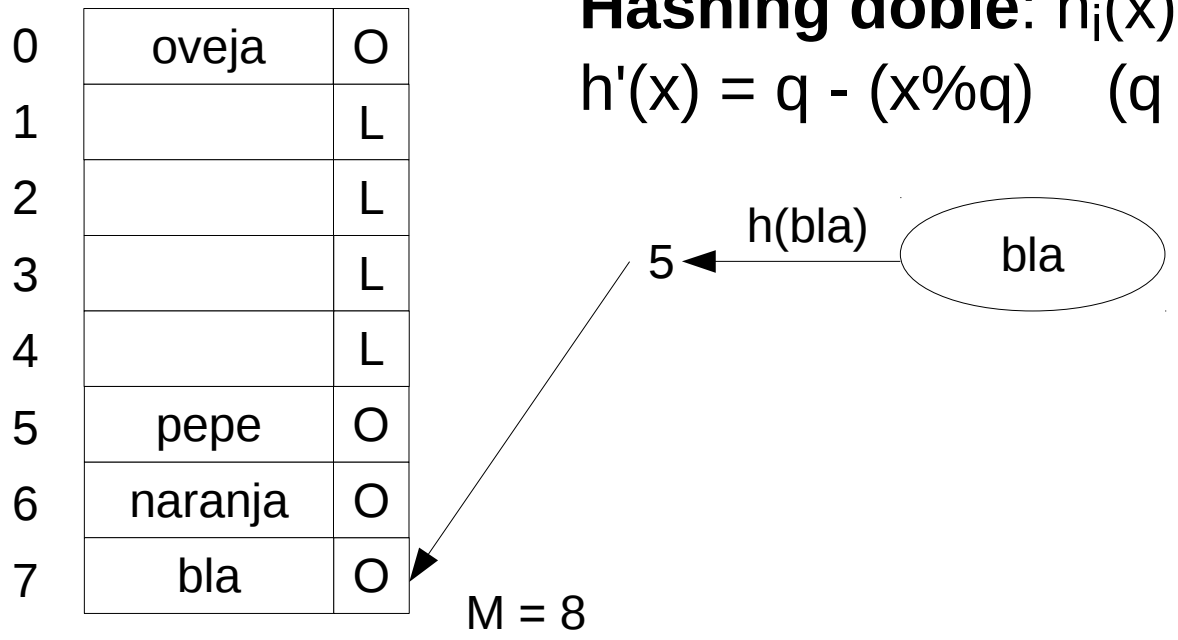


# Estructuras de Datos
# Comprender el concepto de tabla hash

Técnica de **direccionamiento abierto** para resolverlas:

Recordamos para cada entrada si está **Ocupada**, **Libre** o **Borrada**.

Volvemos a aplicar la función hash en base al intento i, o aplicamos otra distinta.




**Hashing doble**: $h_i(x) = (h(x)+i*h'(x))\%M$

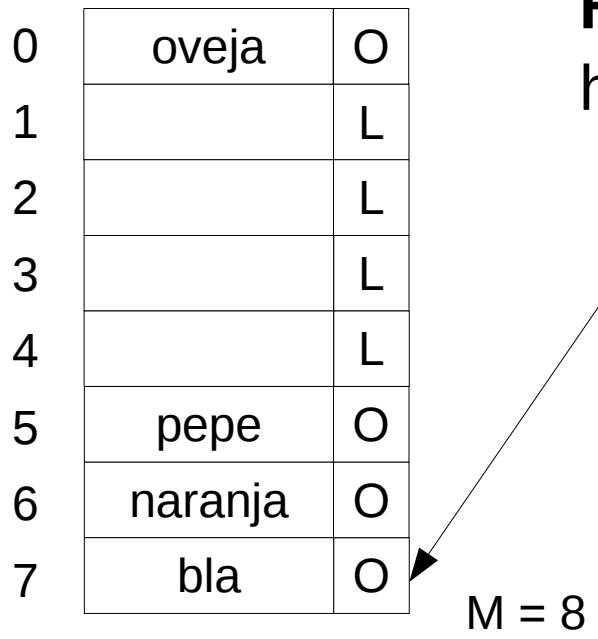
$h'(x) = q - (x\%q)$ (q número primo)

# Estructuras de Datos
# Comprender el concepto de tabla hash

Técnica de **direccionamiento abierto** para resolverlas:

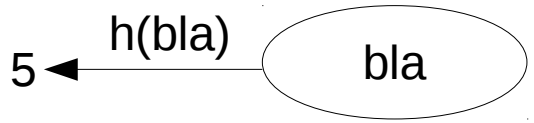
Recordamos para cada entrada si está **Ocupada**, **Libre** o **Borrada**.

Volvemos a aplicar la función hash en base al intento i, o aplicamos otra distinta.




**Hashing doble**: $h_i(x) = (h(x)+i*h'(x))\%M$

$h'(x) = q - (x\%q)$ (q número primo)



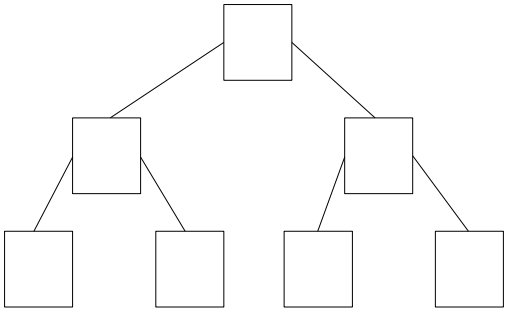

Las casillas borradas son libres para inserción (cabe un elemento) y ocupadas para las búsquedas (hay que seguir buscando). No se recuerda sólo la posición dada por la función hash, sino también la clave.

# Estructuras de Datos
# Comprender el concepto de árbol

Un árbol es una estructura **jerárquica**.

Para cada elemento no hay tan sólo un anterior y un siguiente, ya que existen elementos por encima (**padres**) y por debajo (**hijos**). Dos nodos que son hijos de un mismo nodo se llaman **hermanos**.

Así mismo, se suele llamar nodo **raíz** a aquel que no tiene padres, y nodos hoja a los que no tienen **hijos**.

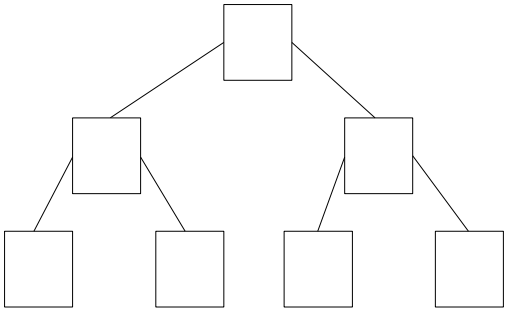


# Estructuras de Datos
# Comprender el concepto de árbol

Si los nodos hermanos están **ordenados** de izquierda a derecha (por cualquier relación), se dice que tenemos un **árbol ordenado**.

Un nodo v es **ancestro** de un nodo n si se puede llegar desde n a v siguiendo la relación padre.

Un nodo v es **descendiente** de un nodo n si se puede llegar desde n a v siguiendo la relación hijo.



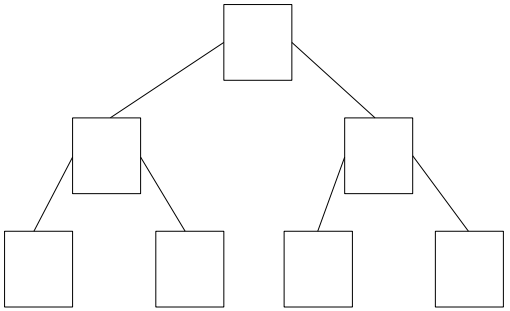
# Estructuras de Datos
# Comprender el concepto de árbol

Un nodo n es **interior** si tiene hijos.

Un nodo n es **exterior** si no tiene hijos (es hoja).

Un nodo puede contener información, a la que se llama **etiqueta del nodo**.

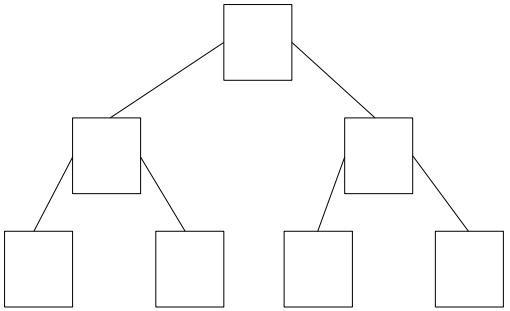
Un **árbol etiquetado** es aquel cuyos nodos poseen etiquetas.



# Estructuras de Datos
# Comprender el concepto de árbol

La **profundidad de un nodo** es el número de relaciones padre que hay que seguir para llegar desde ese nodo hasta el nodo raíz. La raíz tiene profundidad cero.

Desde la raíz de un árbol se puede llegar a **todos** los demás nodos siguiendo relaciones hijo.

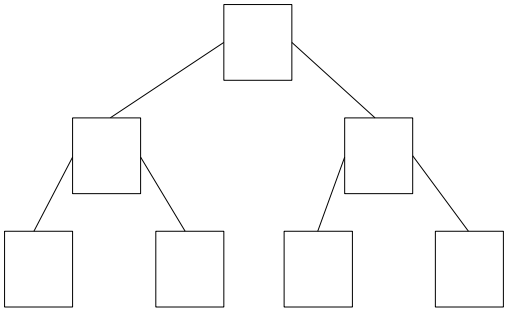


# Estructuras de Datos
# Comprender el concepto de árbol

Un **nivel** consiste en el conjunto de nodos que se encuentran a la misma profundidad.

La **altura de un nodo** es el número de relaciones hijo que hay que seguir hasta alcanzar su descendiente hoja más lejano. Una hoja tiene altura cero.

La **altura de un árbol** es la altura de su nodo raíz.

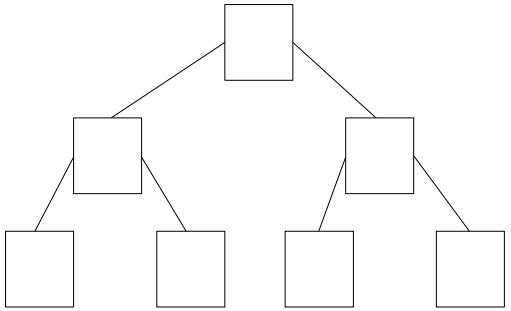


# Estructuras de Datos
# Comprender el concepto de árbol

Tenemos un **subárbol de un árbol** si cogemos un nodo y todos sus descendientes.

Un **subárbol parcial de un árbol** es un árbol que tan sólo tiene algunos de los nodos del árbol original.

Se llama **árbol propio** al árbol cuyos nodos internos tienen, cada uno de ellos, 2 hijos.



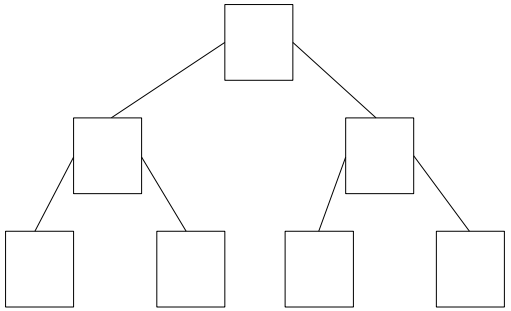
# Estructuras de Datos
# Comprender el concepto de árbol

En los árboles **no hay** un número máximo de hijos por nodo.

Pero si el número máximo de hijos por nodo es 2 diremos que el árbol es **binario** (hijo izquierdo, hijo derecho).

Si el número máximo de hijos por nodo es 2 diremos que el árbol es **ternario** (hijo izquierdo, hijo central, hijo derecho).

En general, tenemos **árboles n-arios** (máximo número de hijos por nodo).



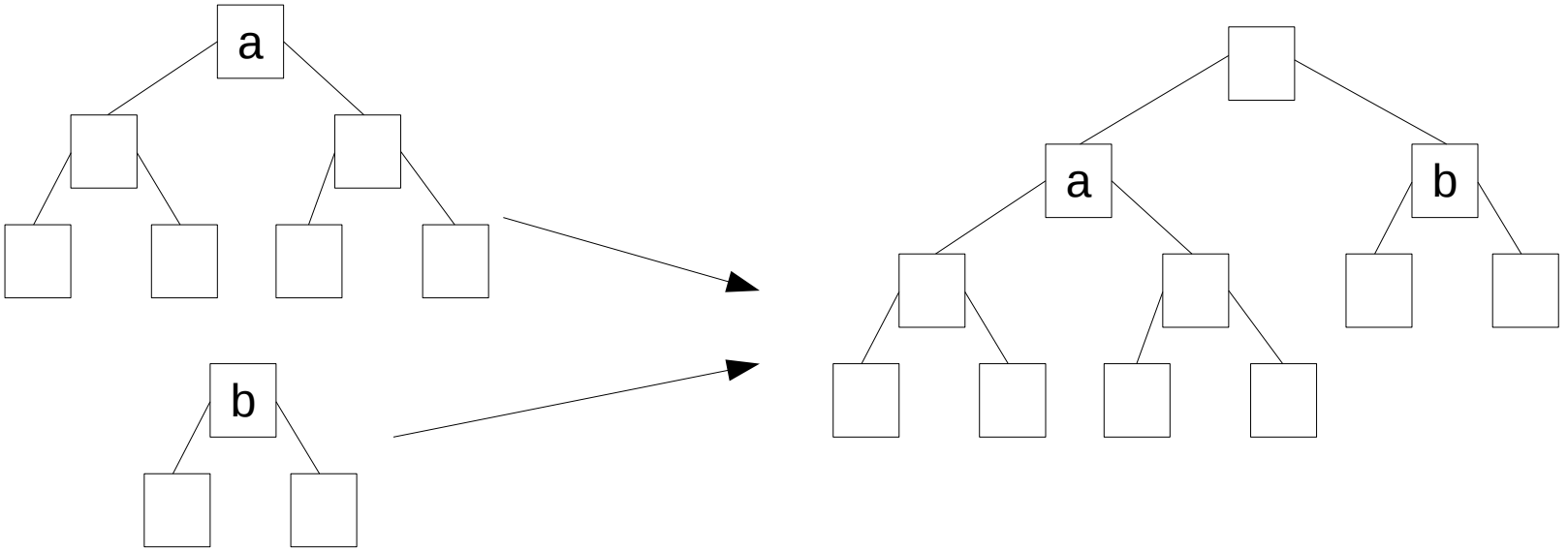
# Estructuras de Datos
# Comprender el concepto de árbol

**Propiedades de un árbol**:

Un nodo, por sí solo, es un árbol.

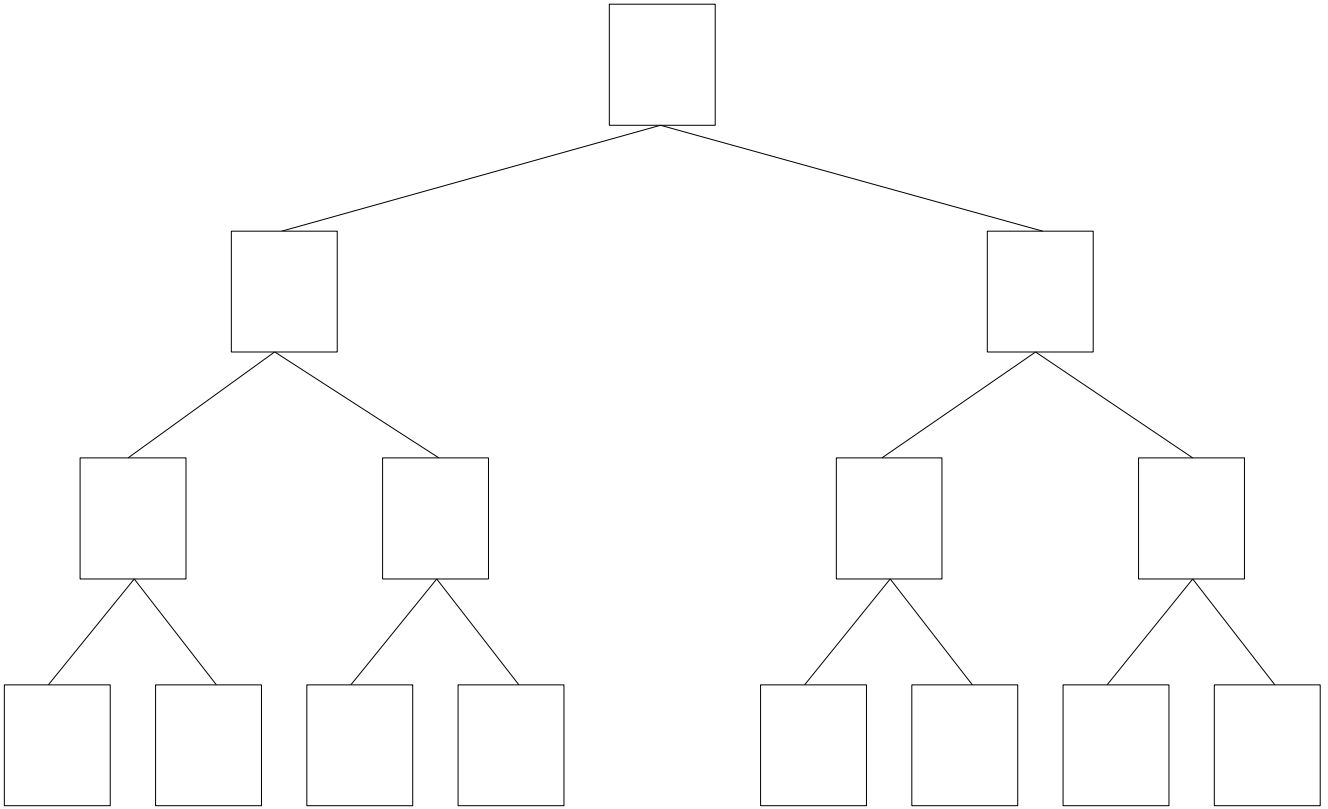
Un árbol vacío (con cero nodos), también es un árbol.

Si tomamos N árboles y cada una de sus raíces las hacemos hijo de un nodo, el resultado también es un árbol.



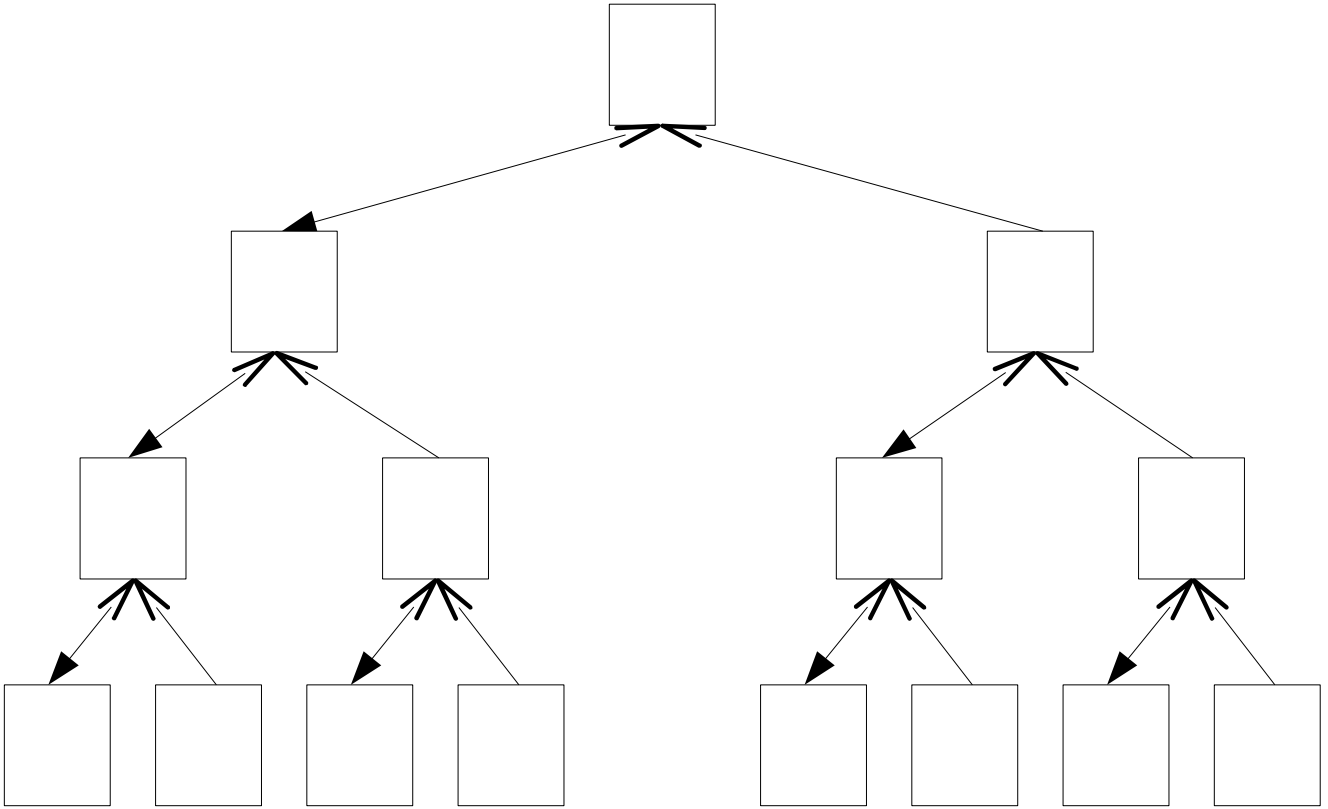
# Estructuras de Datos
# Comprender el concepto de árbol

Vamos a ver cómo se implementa un árbol usando sólo punteros (no listas).



# Estructuras de Datos
# Comprender el concepto de árbol

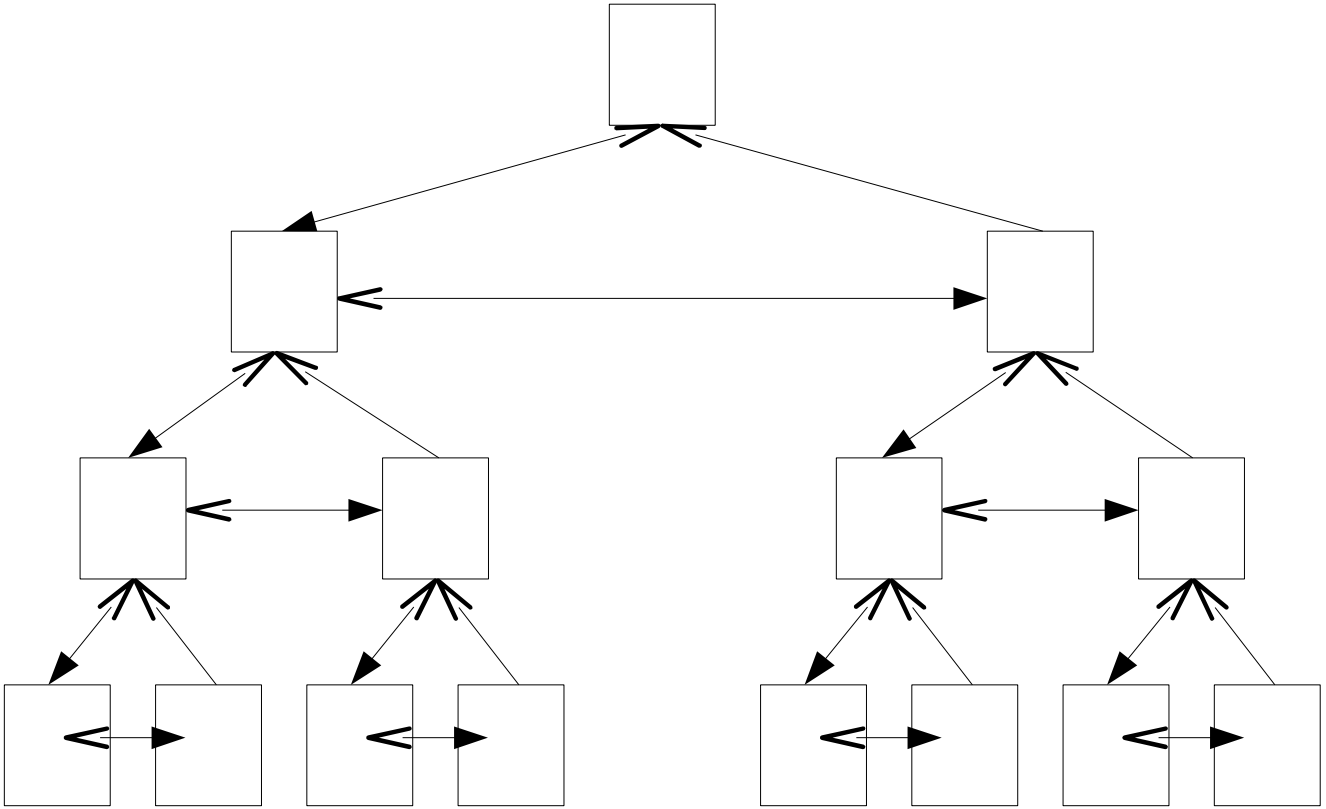
Navegabilidades padres-primer hijo e hjios-padres (opcional).



# Estructuras de Datos
# Comprender el concepto de árbol

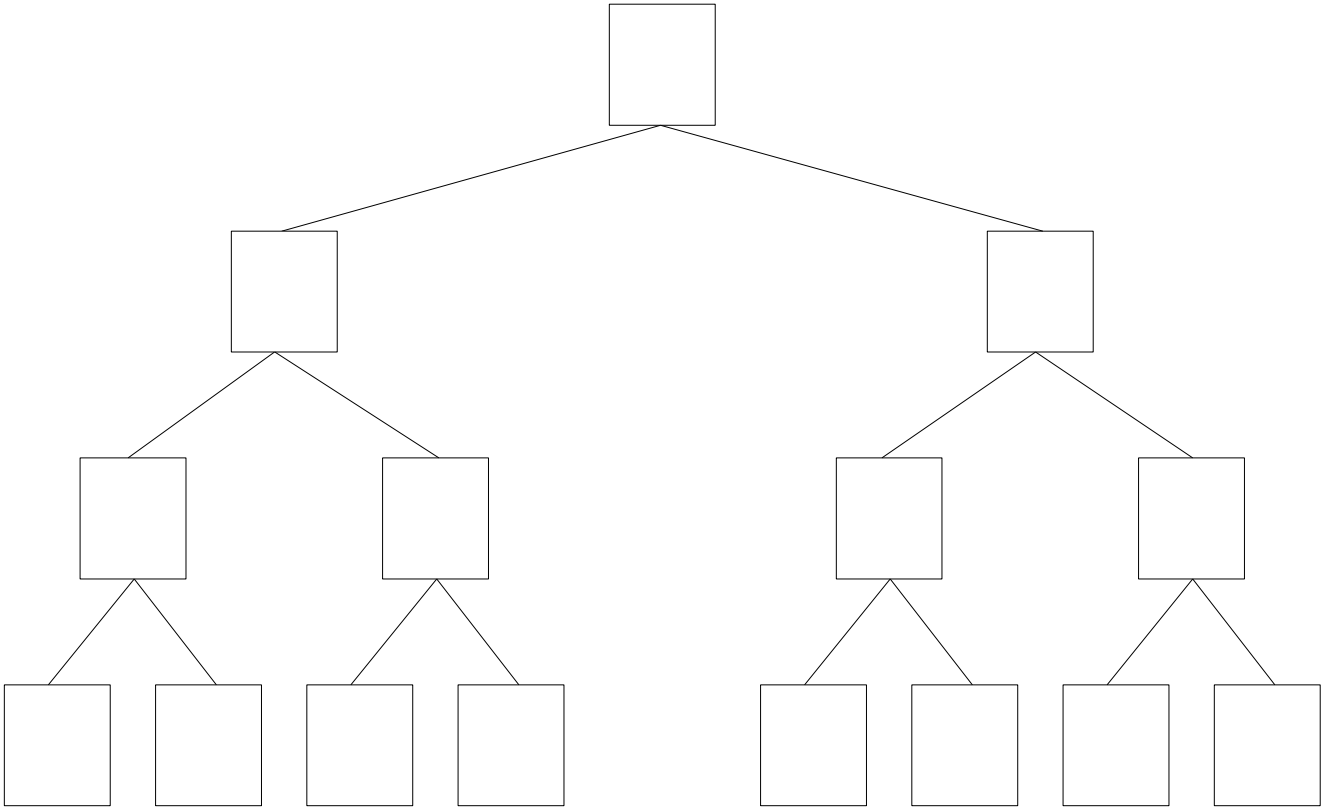
Navegabilidades padres-primer hijo e hjios-padres (opcional).

Navegabilidades hermano-der. y hermano-izq. (opcional).



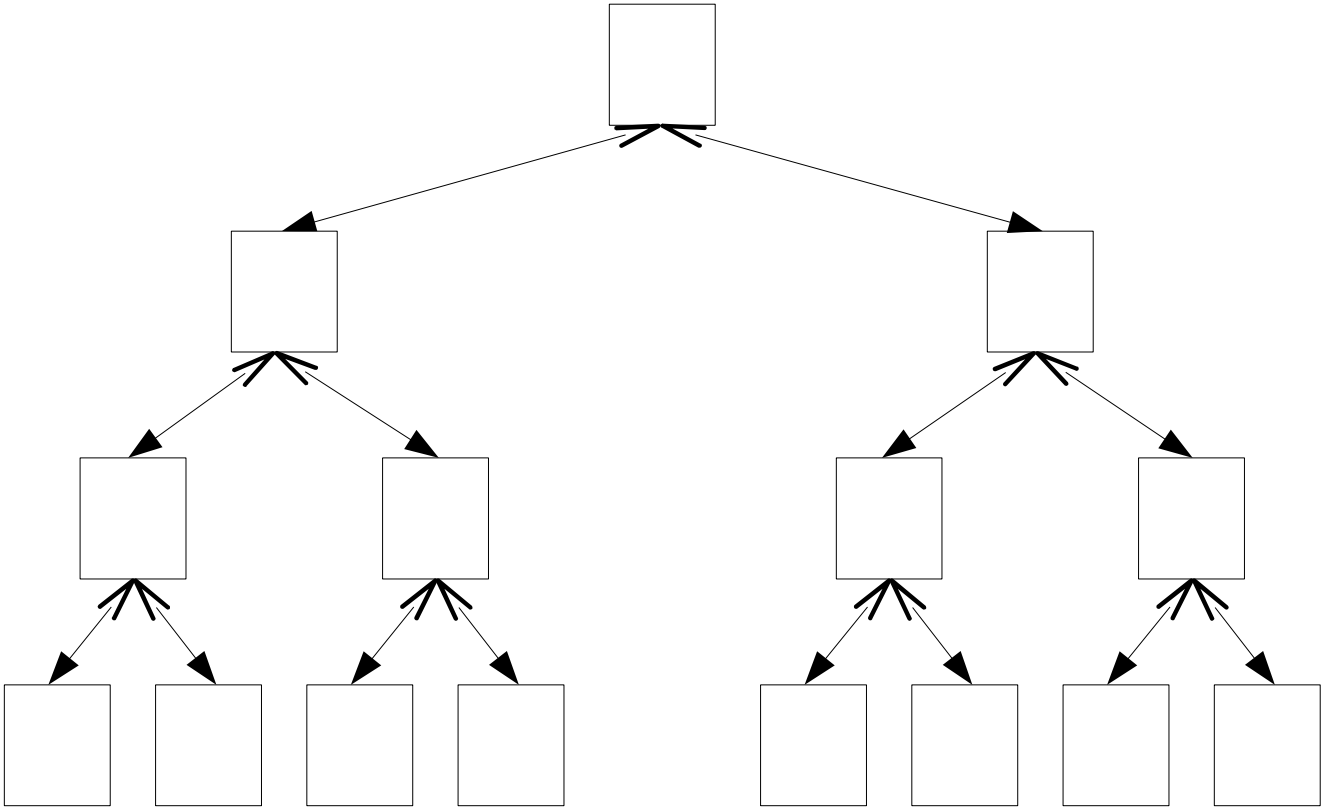
# Estructuras de Datos
# Comprender el concepto de árbol

Vamos a ver cómo se implementa un árbol usando punteros y listas.



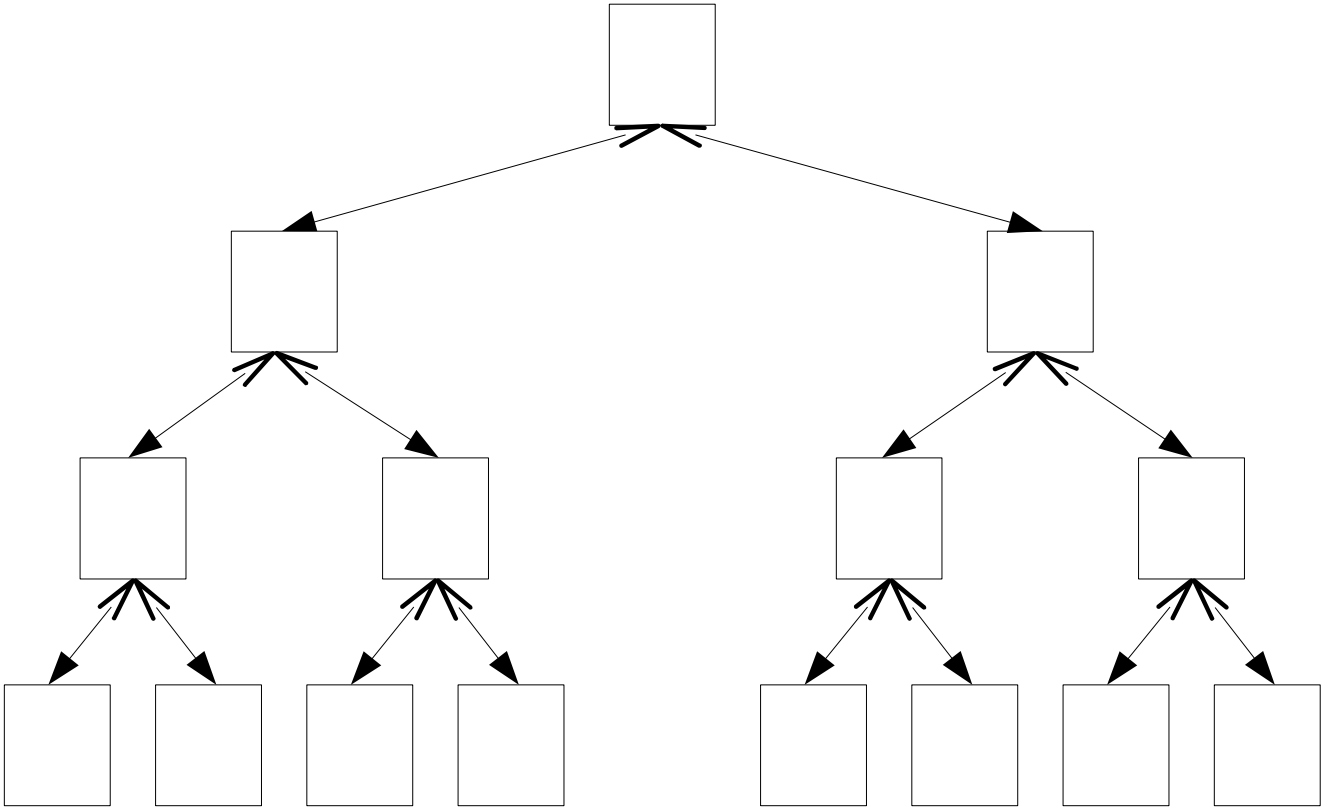
# Estructuras de Datos
# Comprender el concepto de árbol

Navegabilidades padres-hijos e hjios-padres (opcional).



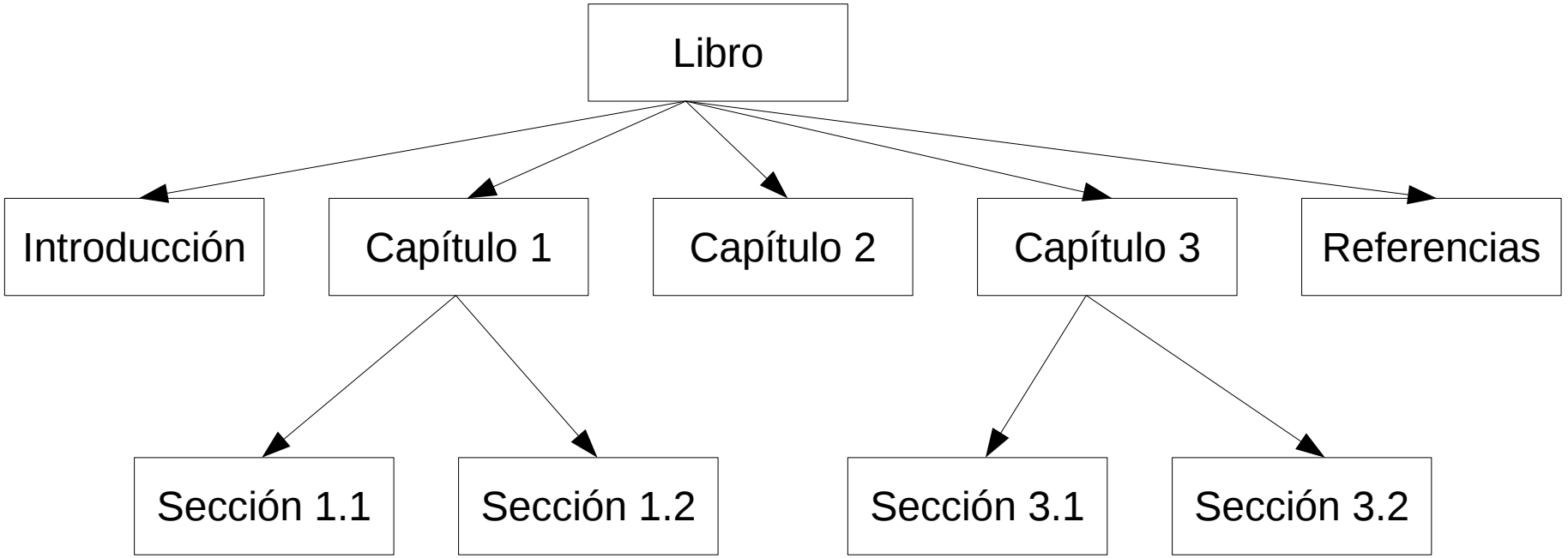
# Estructuras de Datos
# Comprender el concepto de árbol

En cualquiera de los casos, nos interesa definir un TDA arbol (que referencia el nodo raíz) y un TDA nodo (que referencia a su padre, hijos / hermanos (según la alternativa)).



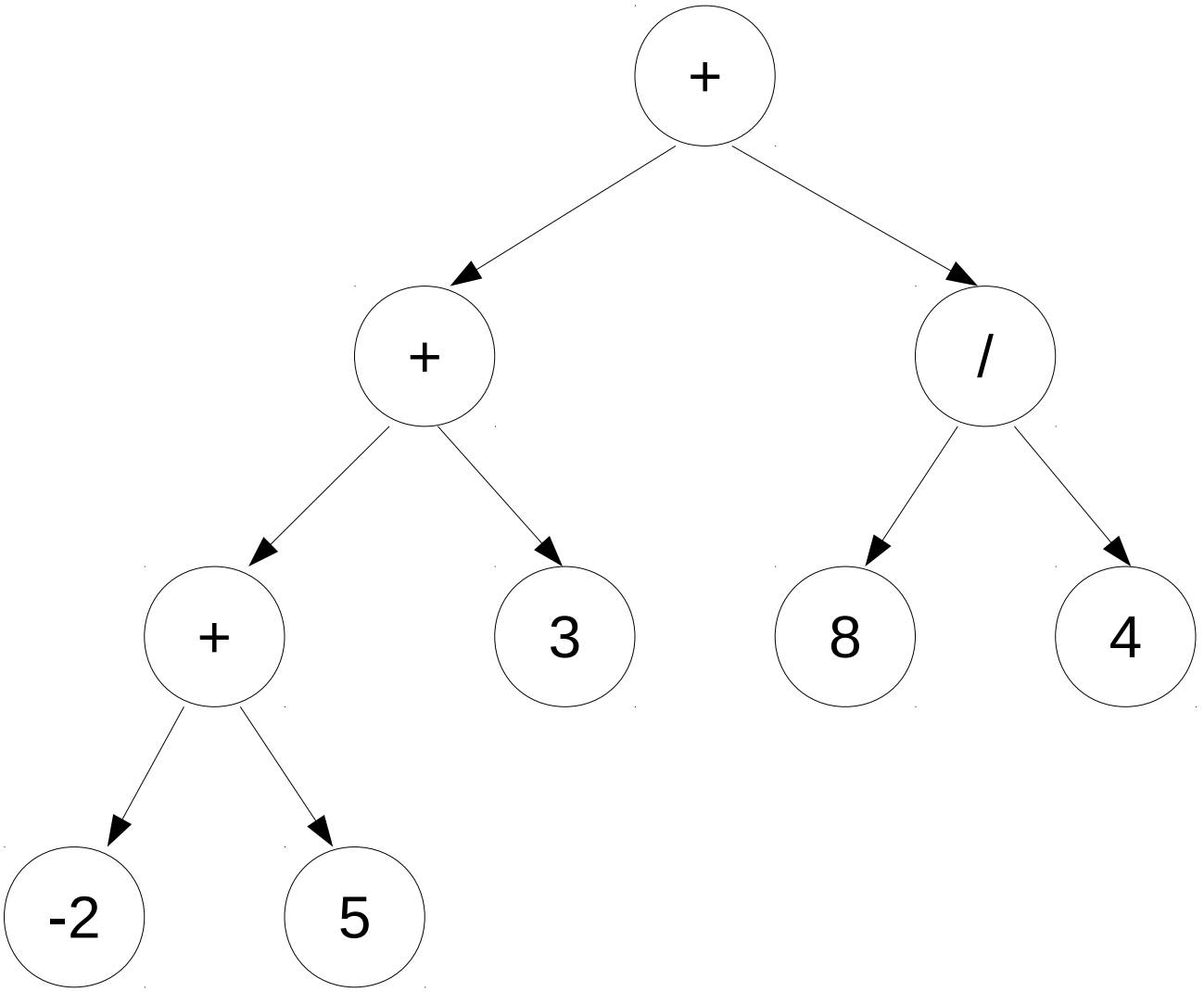
# Estructuras de Datos
# Comprender el concepto de árbol

Ejemplo: Libro



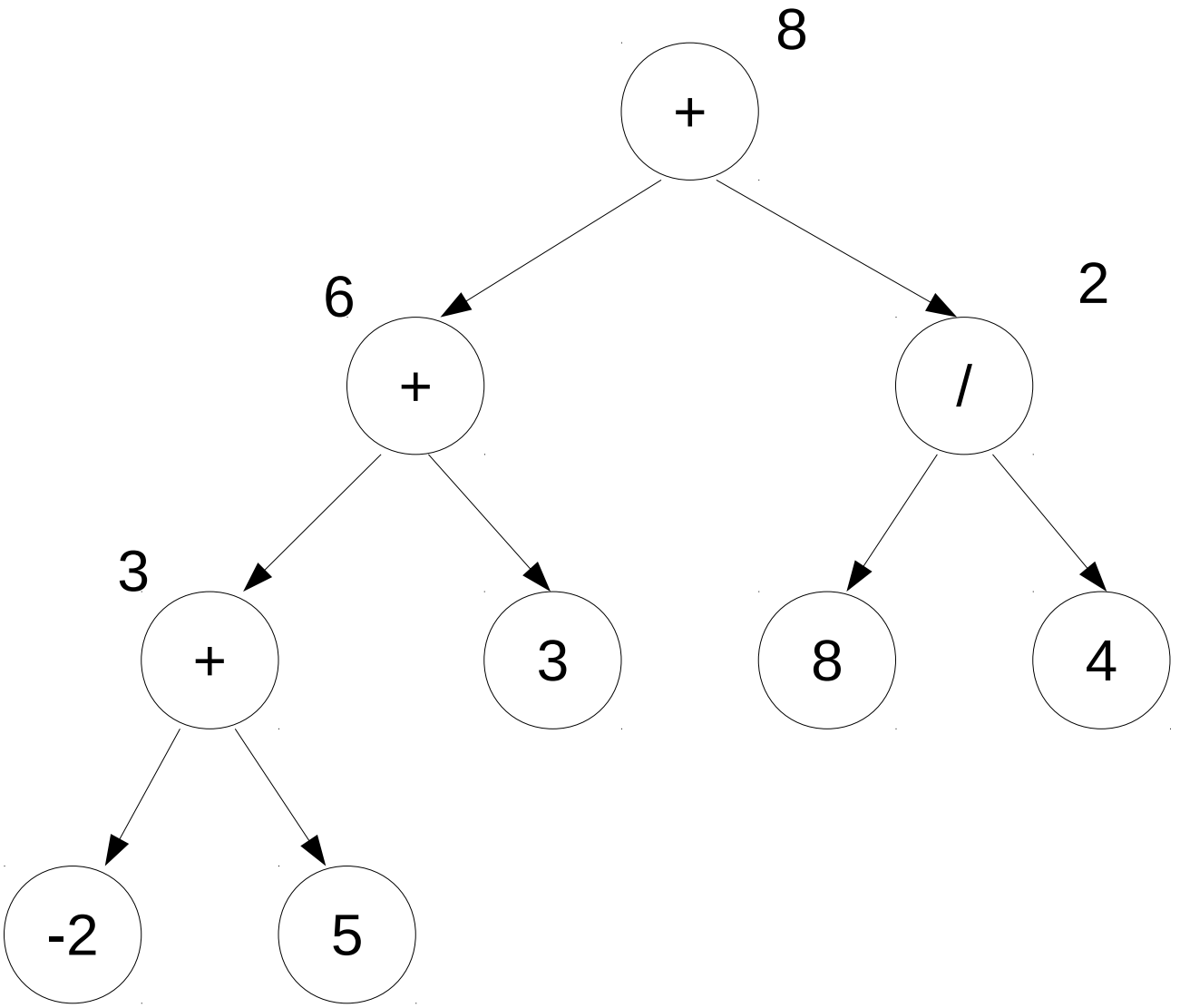
# Estructuras de Datos
# Comprender el concepto de árbol

Ejemplo: Expresión aritmética



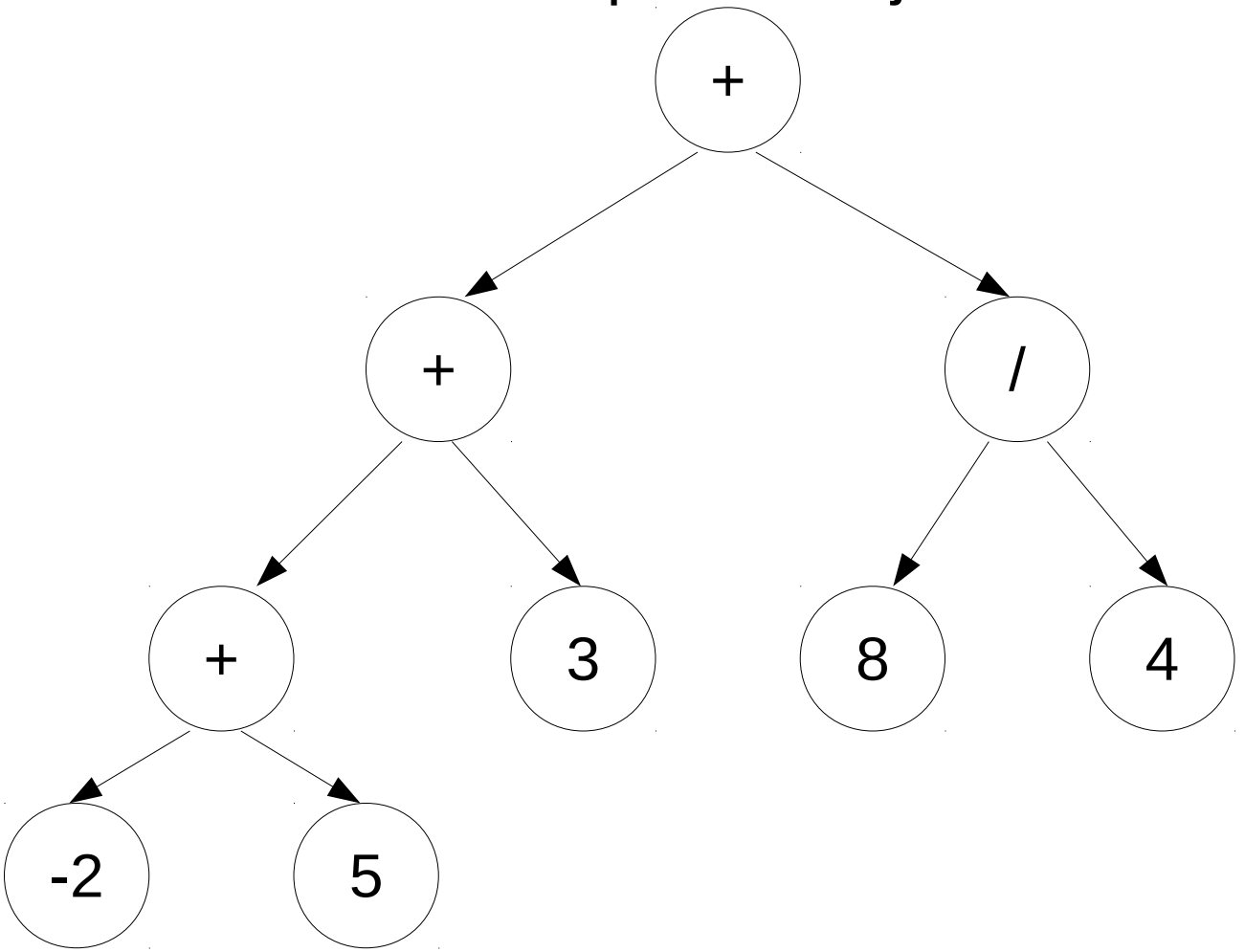
# Estructuras de Datos
# Comprender el concepto de árbol

Ejemplo: Expresión aritmética



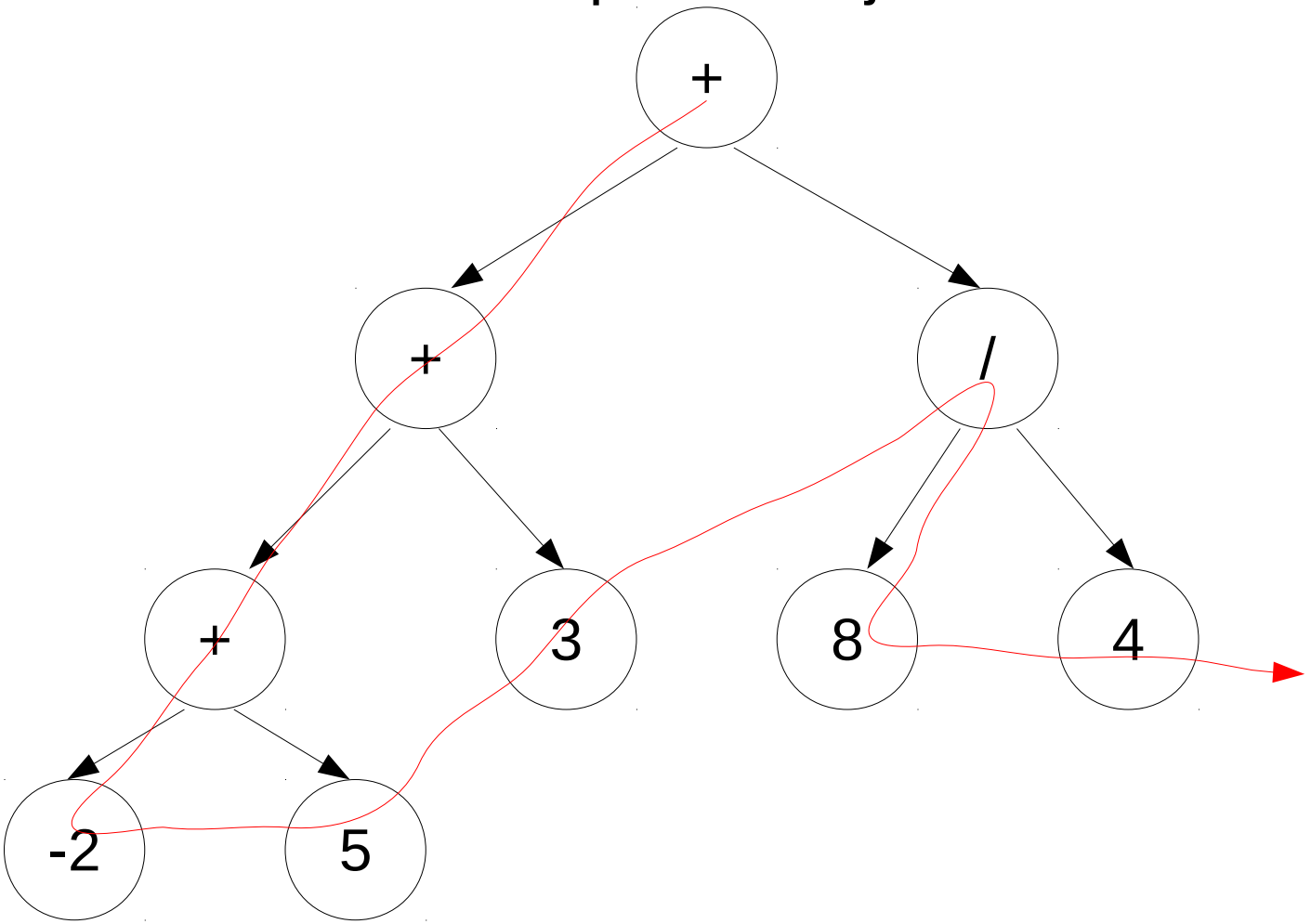
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Podemos recorrer los árboles en **preorden**: Primero visitamos la raíz y luego visitamos (en preorden también) cada uno de los subárboles que son hijos del nodo raíz.



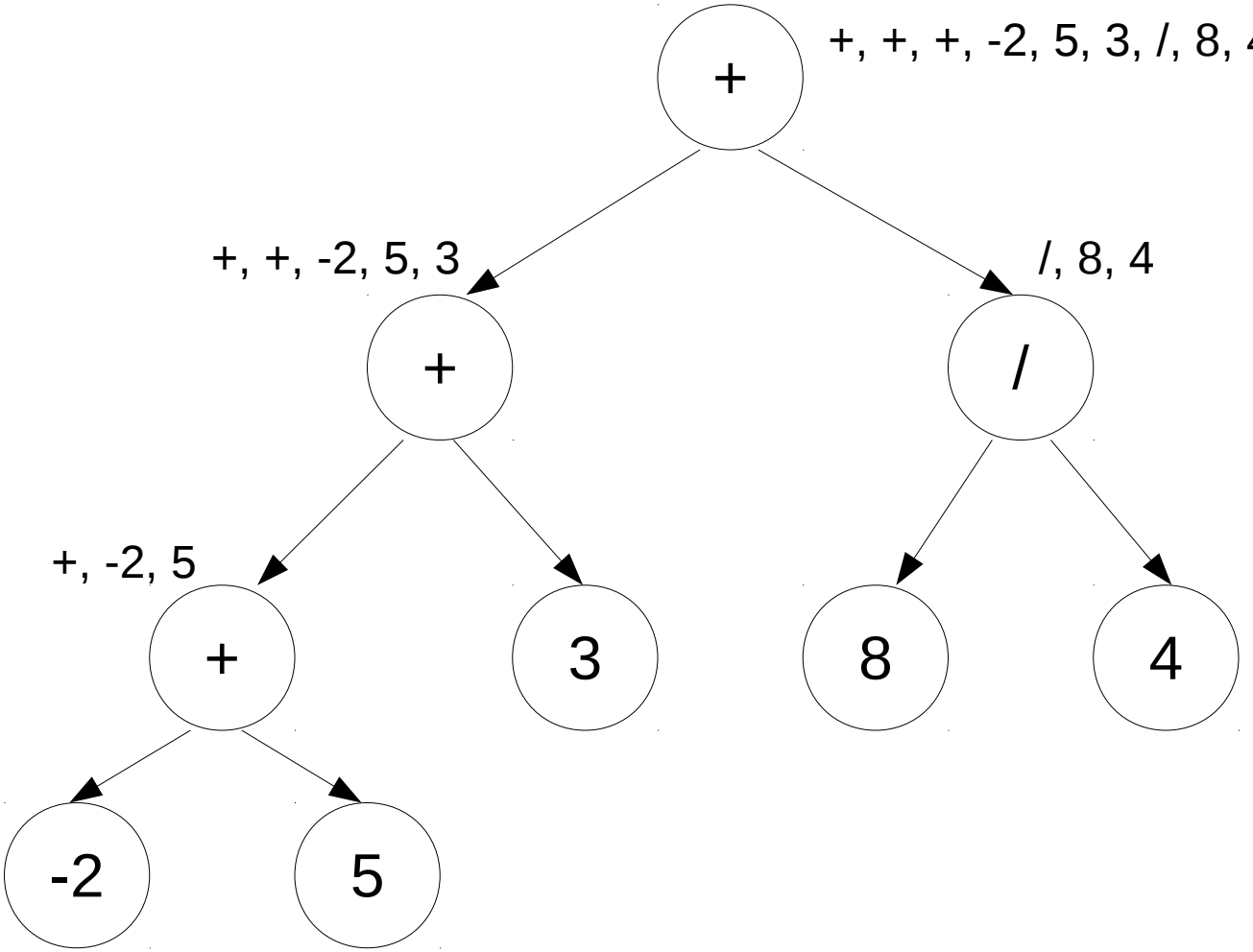
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Podemos recorrer los árboles en **preorden**: Primero visitamos la raíz y luego visitamos (en preorden también) cada uno de los subárboles que son hijos del nodo raíz.



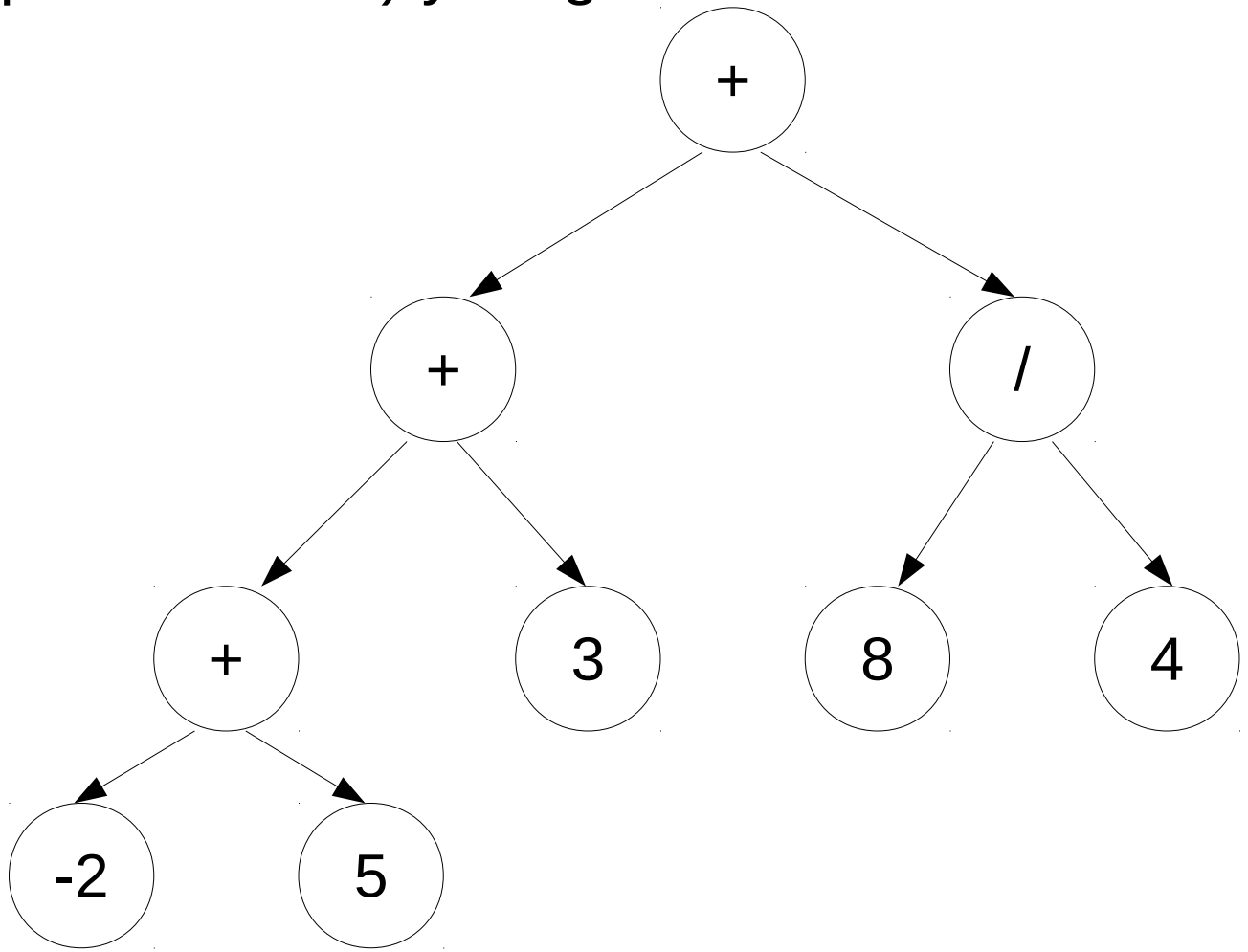
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Obtendríamos una notación **prefijo**, en la que cada operador afecta a los dos elementos siguientes.



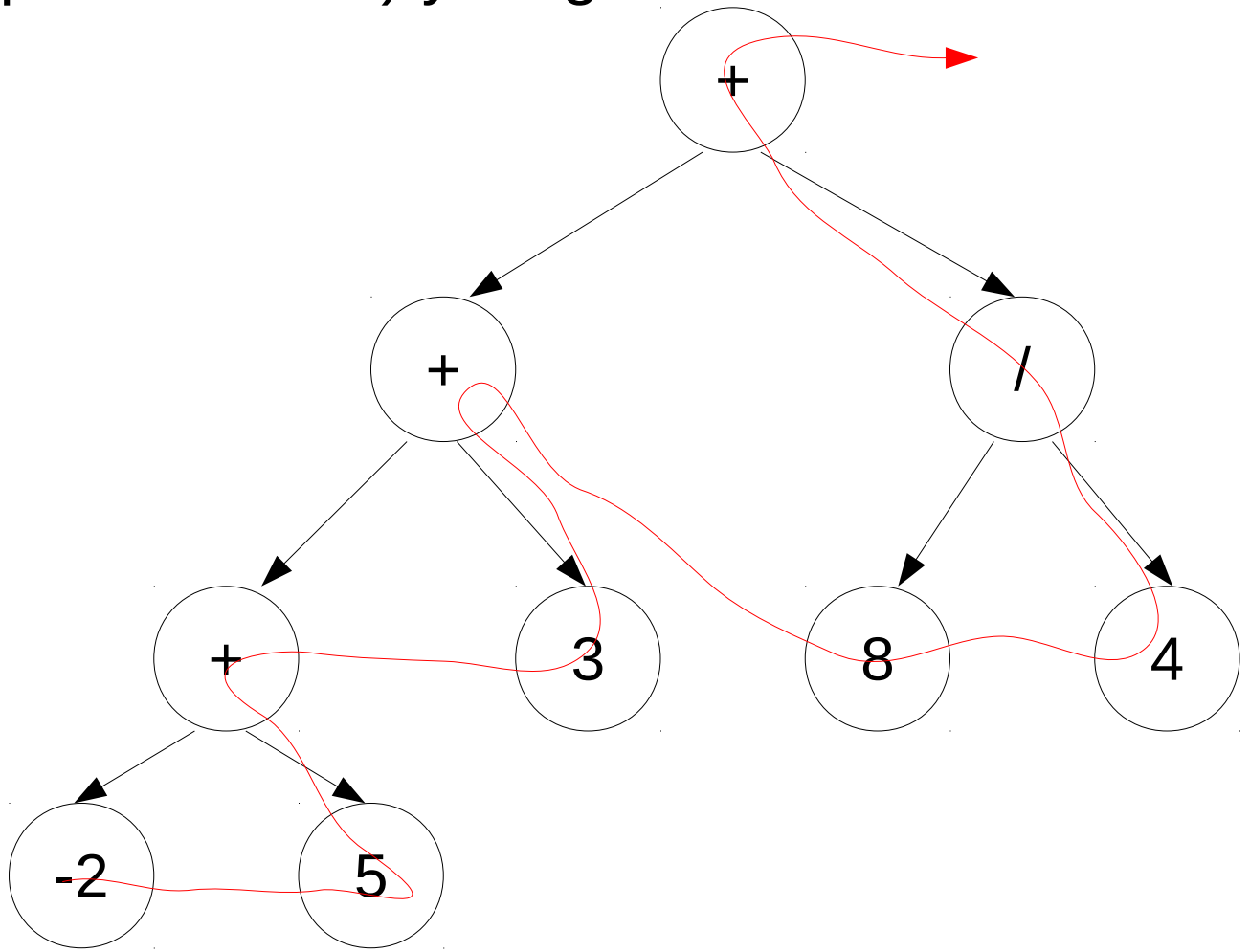
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Podemos recorrer los árboles en **postorden**: Primero visitamos cada uno de los subárboles que son hijos del nodo raíz (en post también) y luego visitamos la raíz.



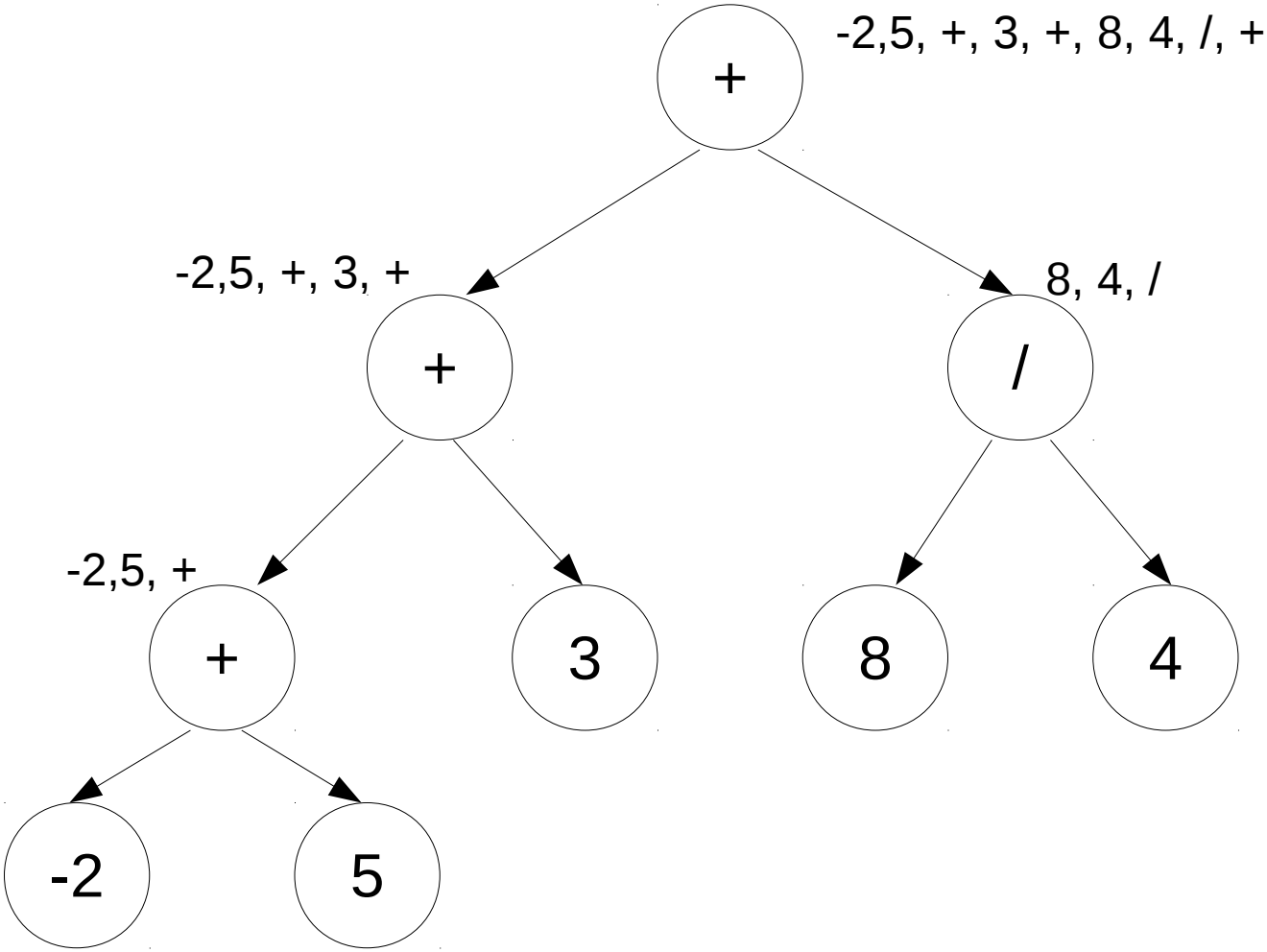
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Podemos recorrer los árboles en **postorden**: Primero visitamos cada uno de los subárboles que son hijos del nodo raíz (en post también) y luego visitamos la raíz.



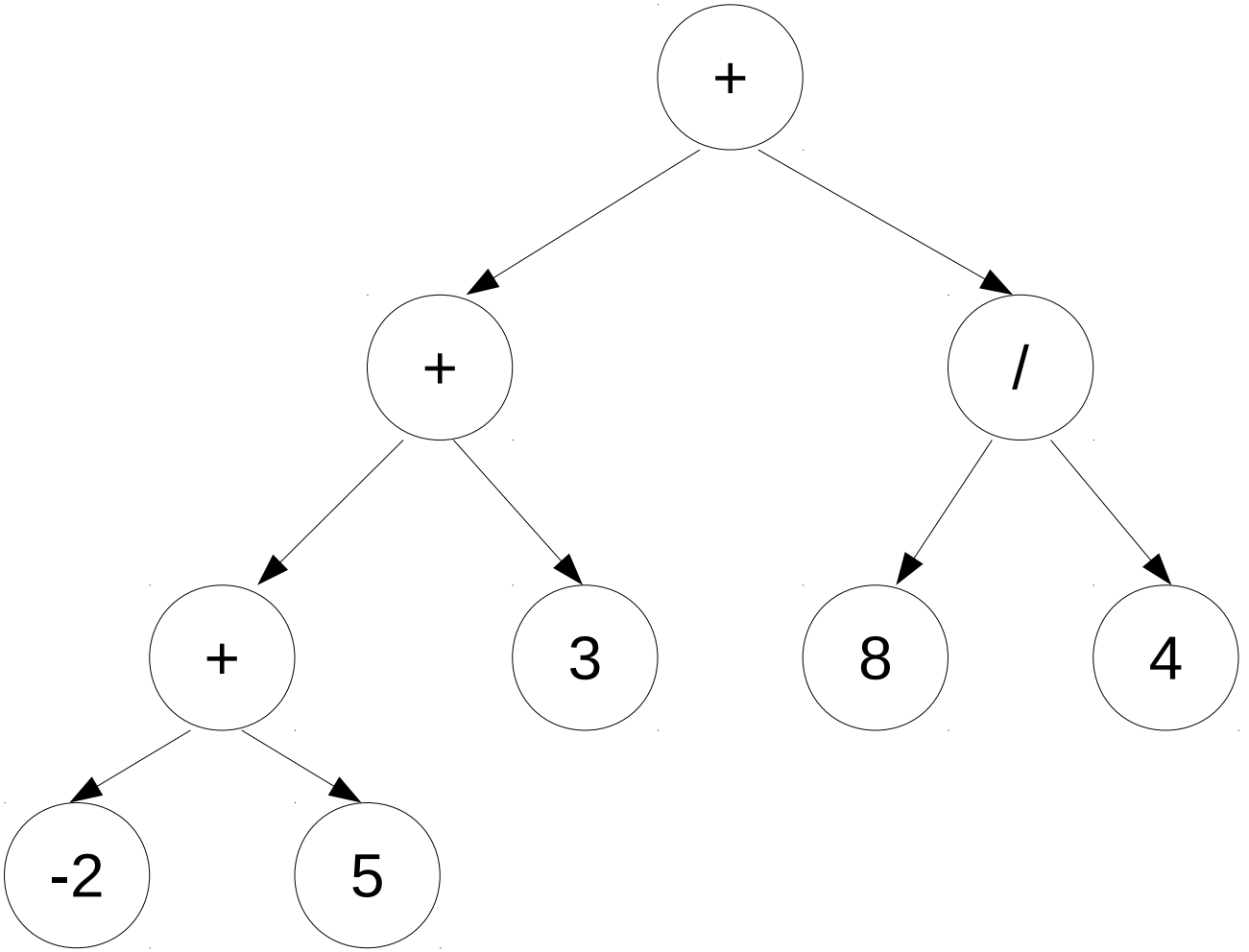
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Obtendríamos una notación **postfijo**, en la que cada operador afecta a los dos elementos anteriores.



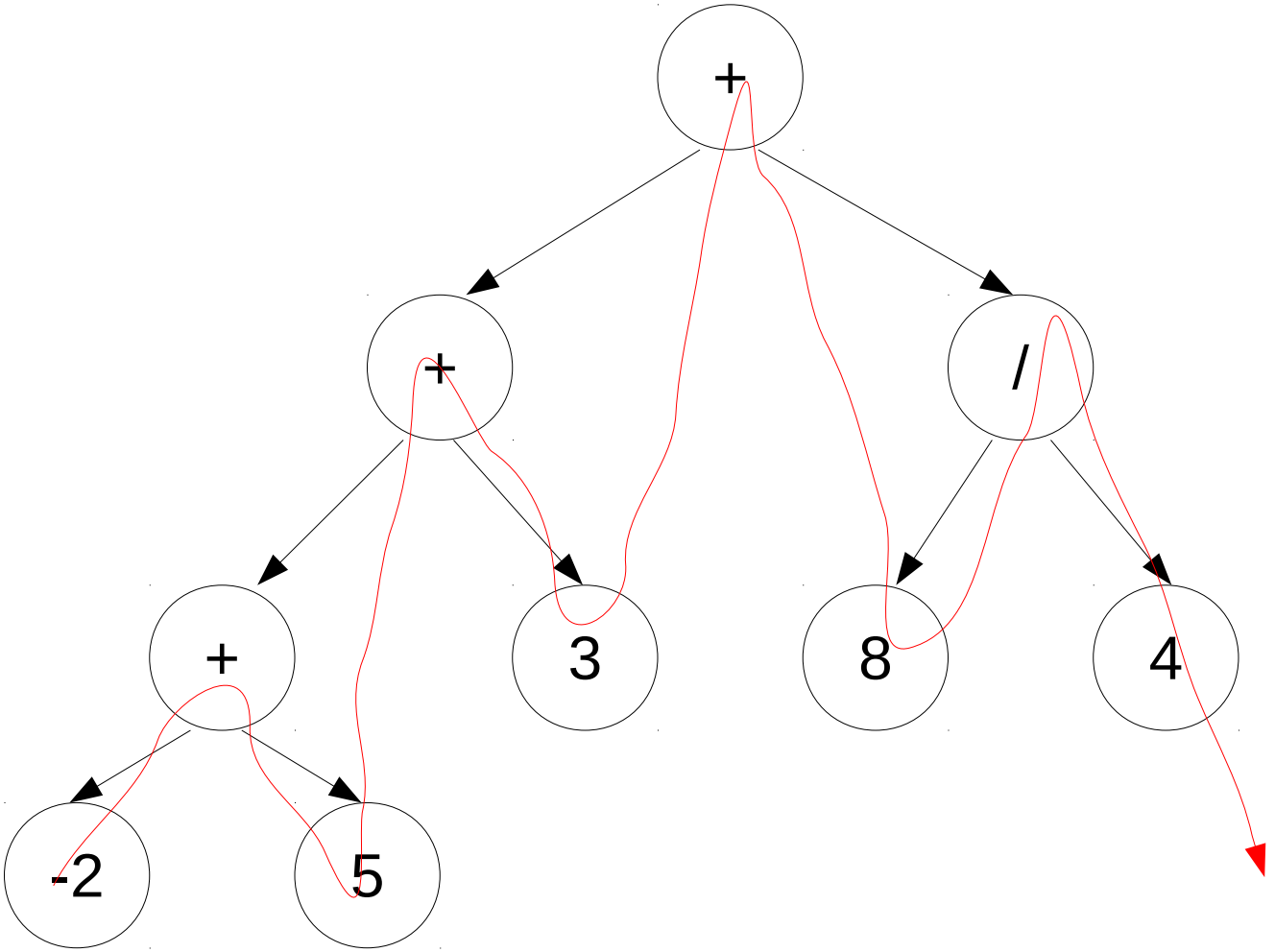
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Podemos recorrer los árboles en **inorden**: Primero visitamos en inorden el subárbol izquierdo, luego la raíz y luego en inorden el **resto** de subárboles.



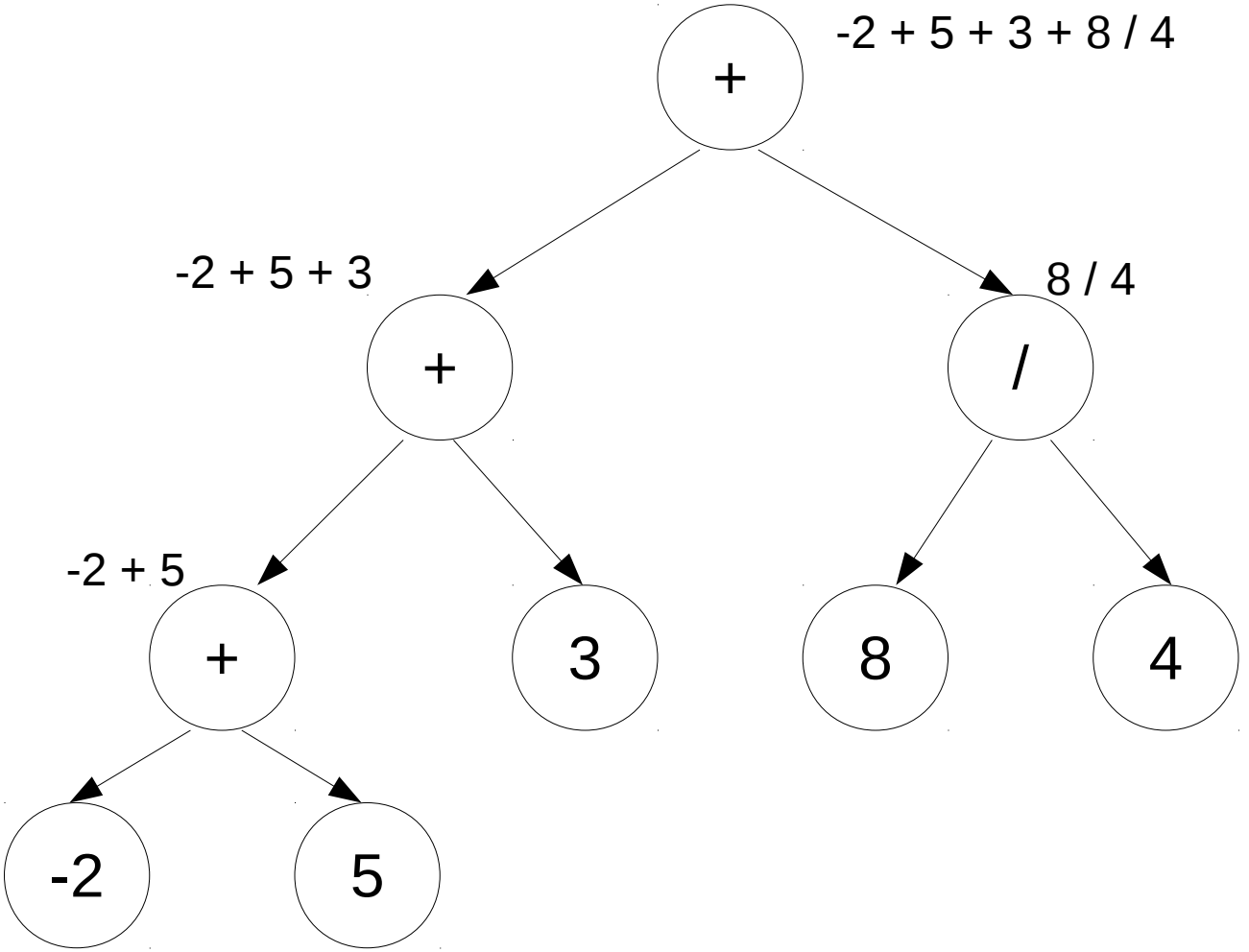
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Podemos recorrer los árboles en **inorden**: Primero visitamos en inorden el subárbol izquierdo, luego la raíz y luego en inorden el **resto** de subárboles.



# Estructuras de Datos
# Conocer las formas de recorrer un árbol

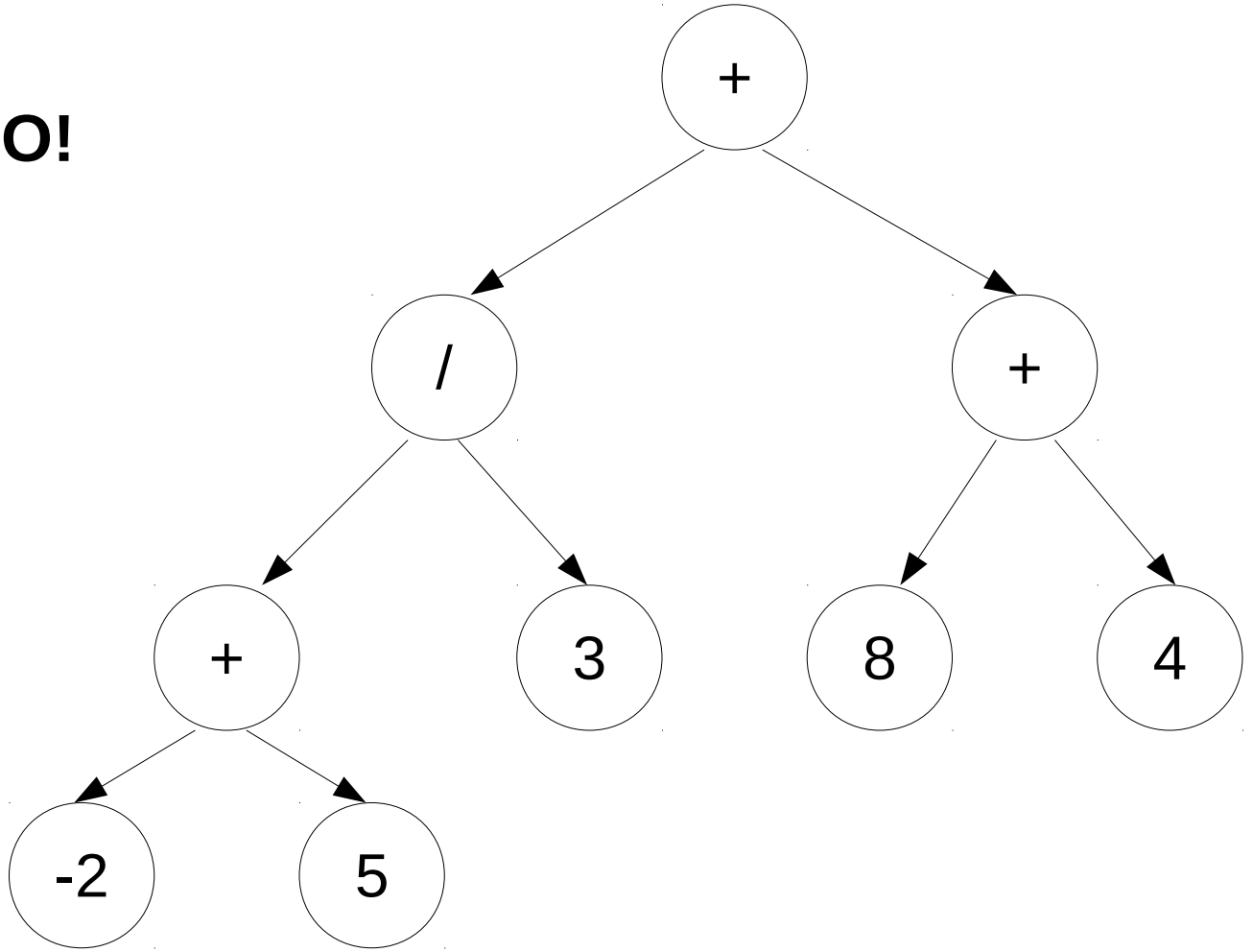
Podemos recorrer los árboles en **inorden**: Primero visitamos en inorden el subárbol izquierdo, luego la raíz y luego en inorden el **resto** de subárboles.



# Estructuras de Datos
# Conocer las formas de recorrer un árbol

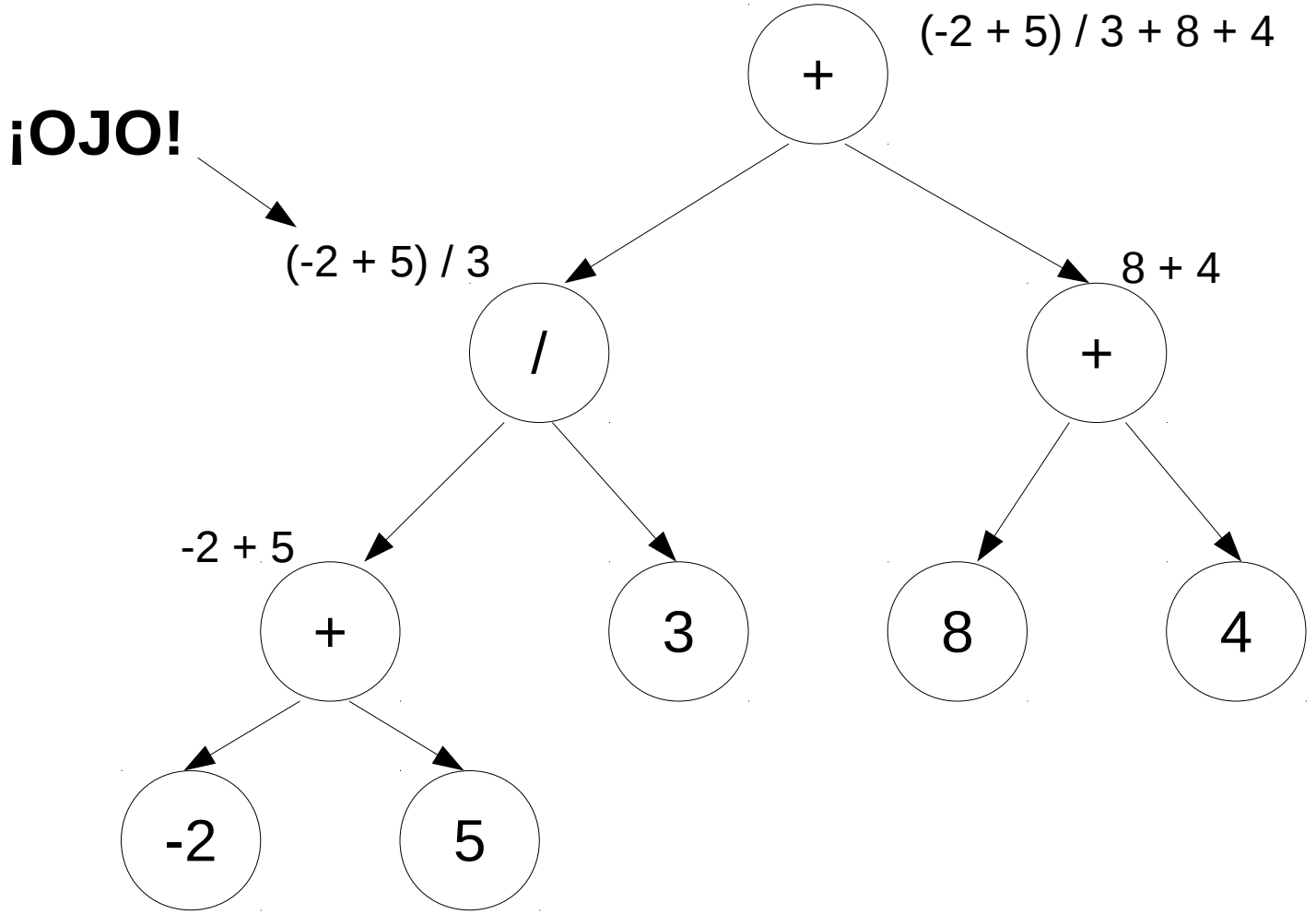
Podemos recorrer los árboles en **inorden**: Primero visitamos en inorden el subárbol izquierdo, luego la raíz y luego en inorden el **resto** de subárboles.

**¡OJO!**



# Estructuras de Datos
# Conocer las formas de recorrer un árbol

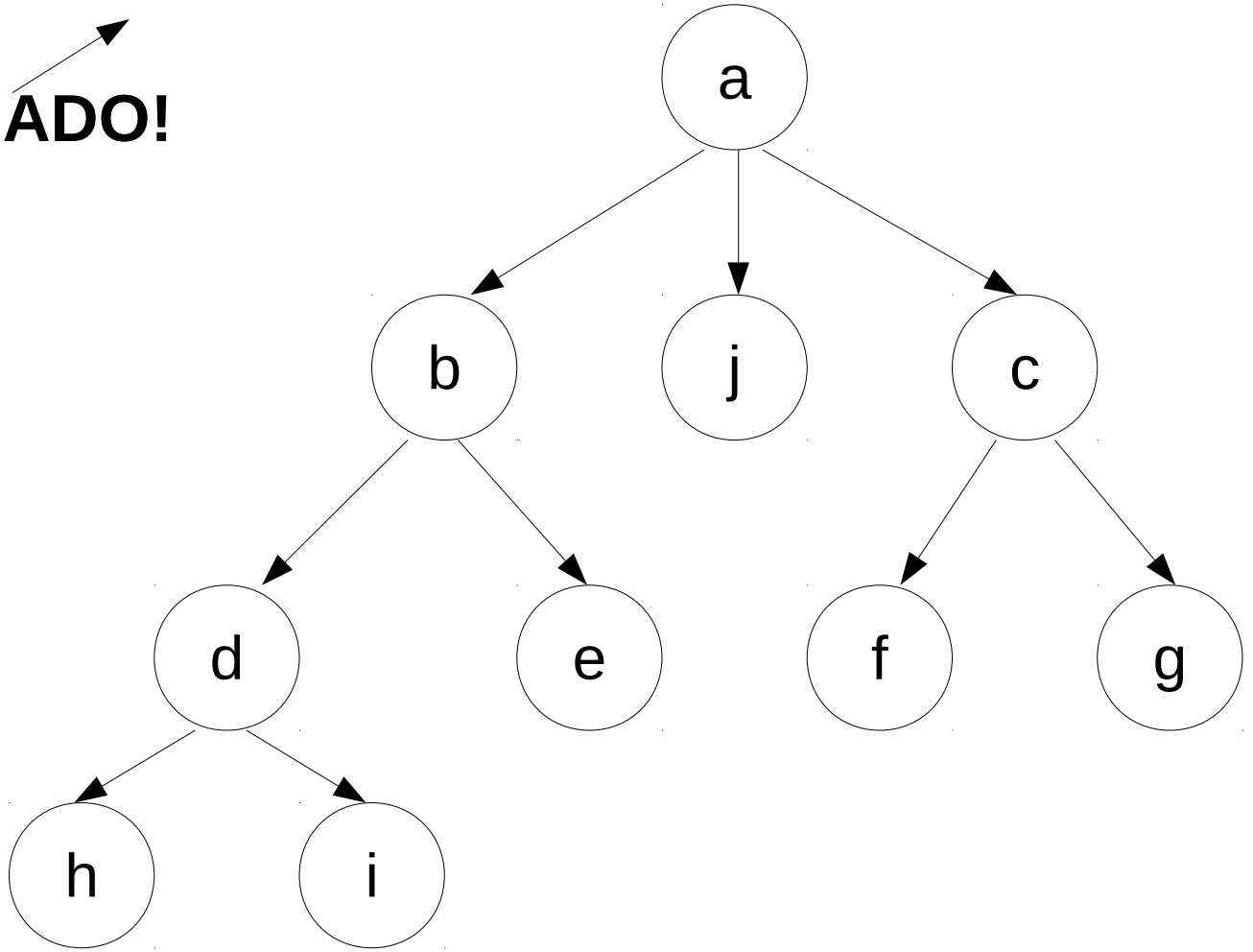
Podemos recorrer los árboles en **inorden**: Primero visitamos en inorden el subárbol izquierdo, luego la raíz y luego en inorden el **resto** de subárboles.



# Estructuras de Datos
# Conocer las formas de recorrer un árbol

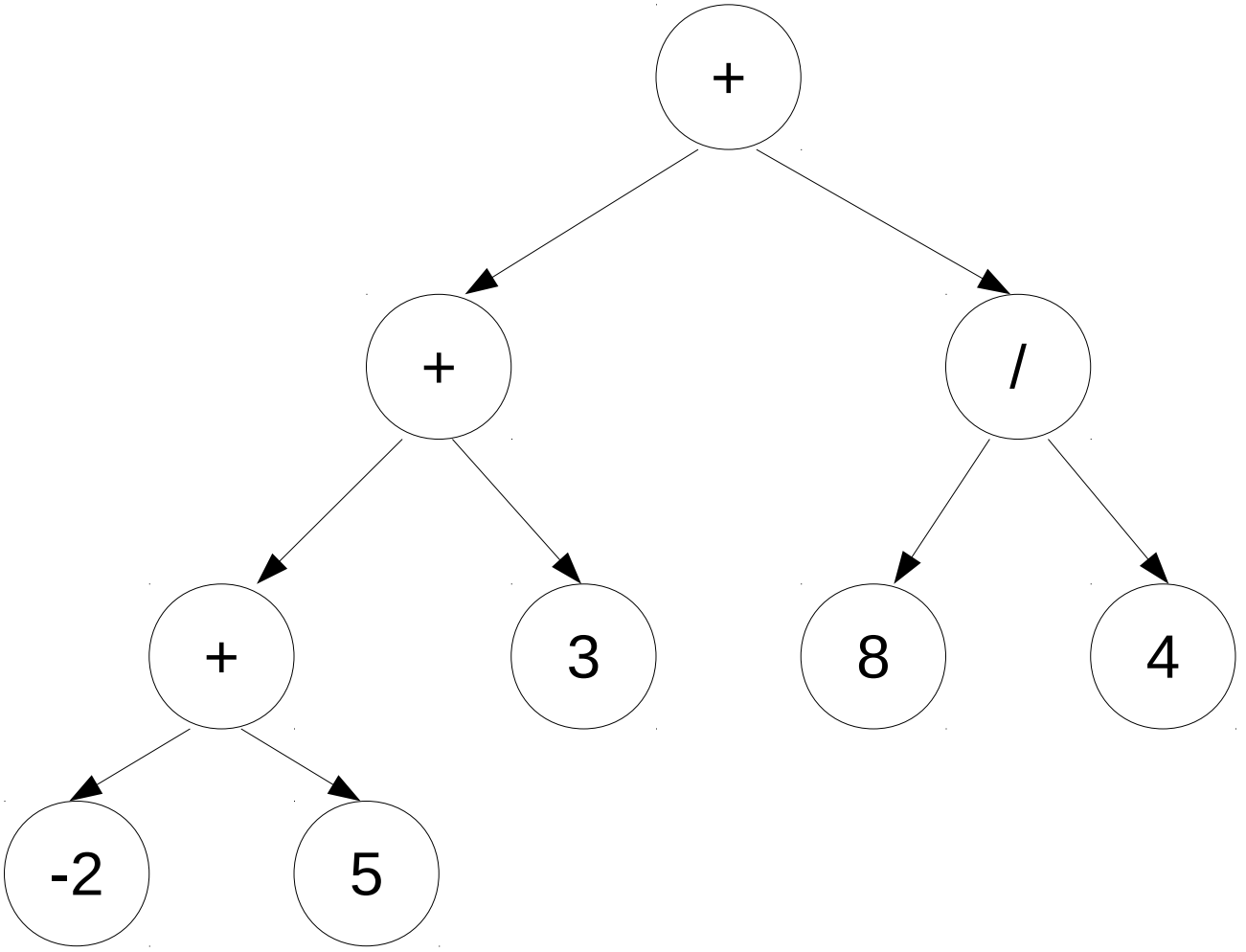
Podemos recorrer los árboles en **inorden**: Primero visitamos en inorden el subárbol izquierdo, luego la raíz y luego en inorden el **resto** de subárboles.

**¡CUIDADO!**



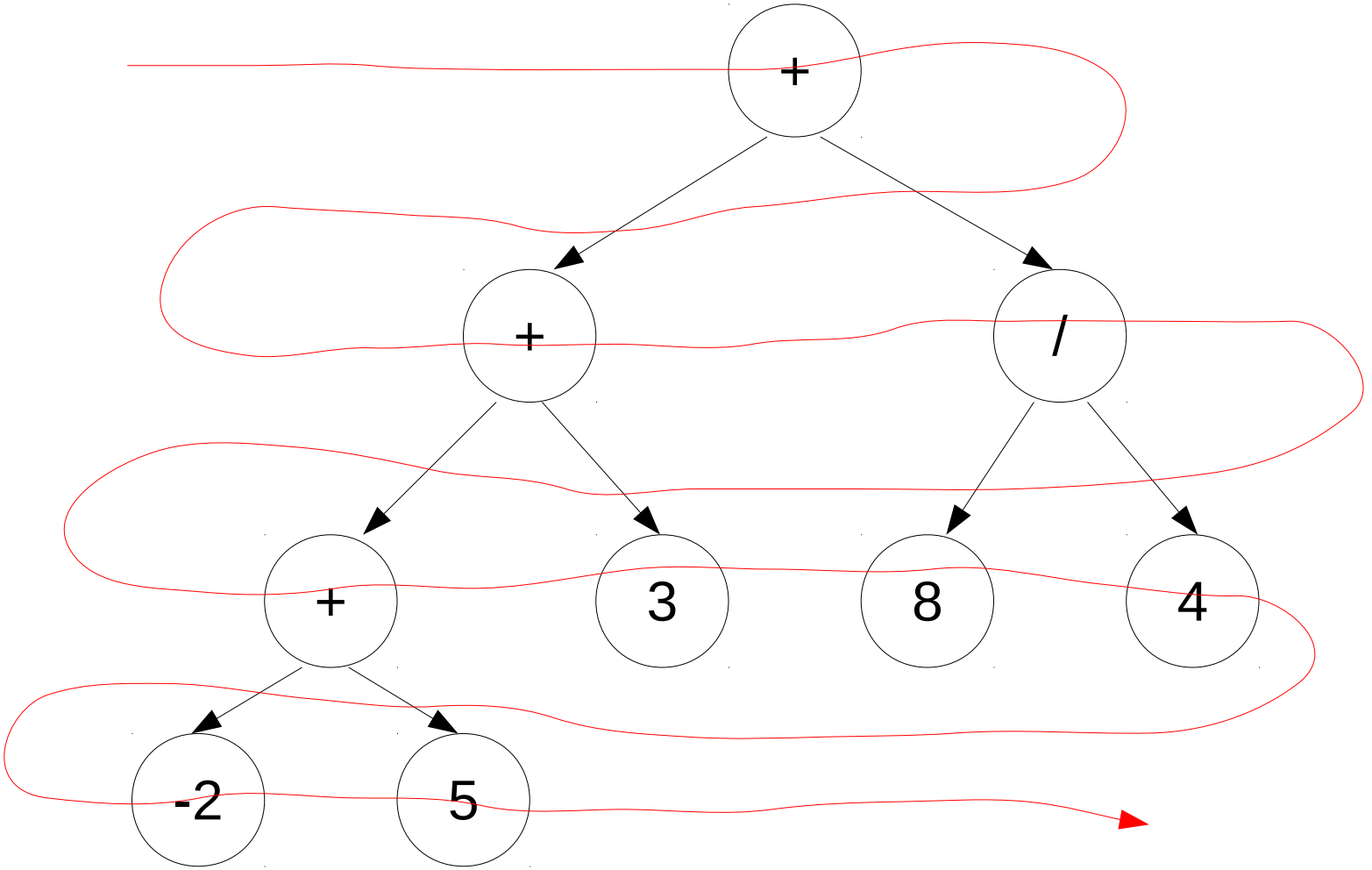
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Podemos recorrer los árboles en **anchura**: Primero visitamos los nodos en profundidad 0, luego en profundidad 1, etc.



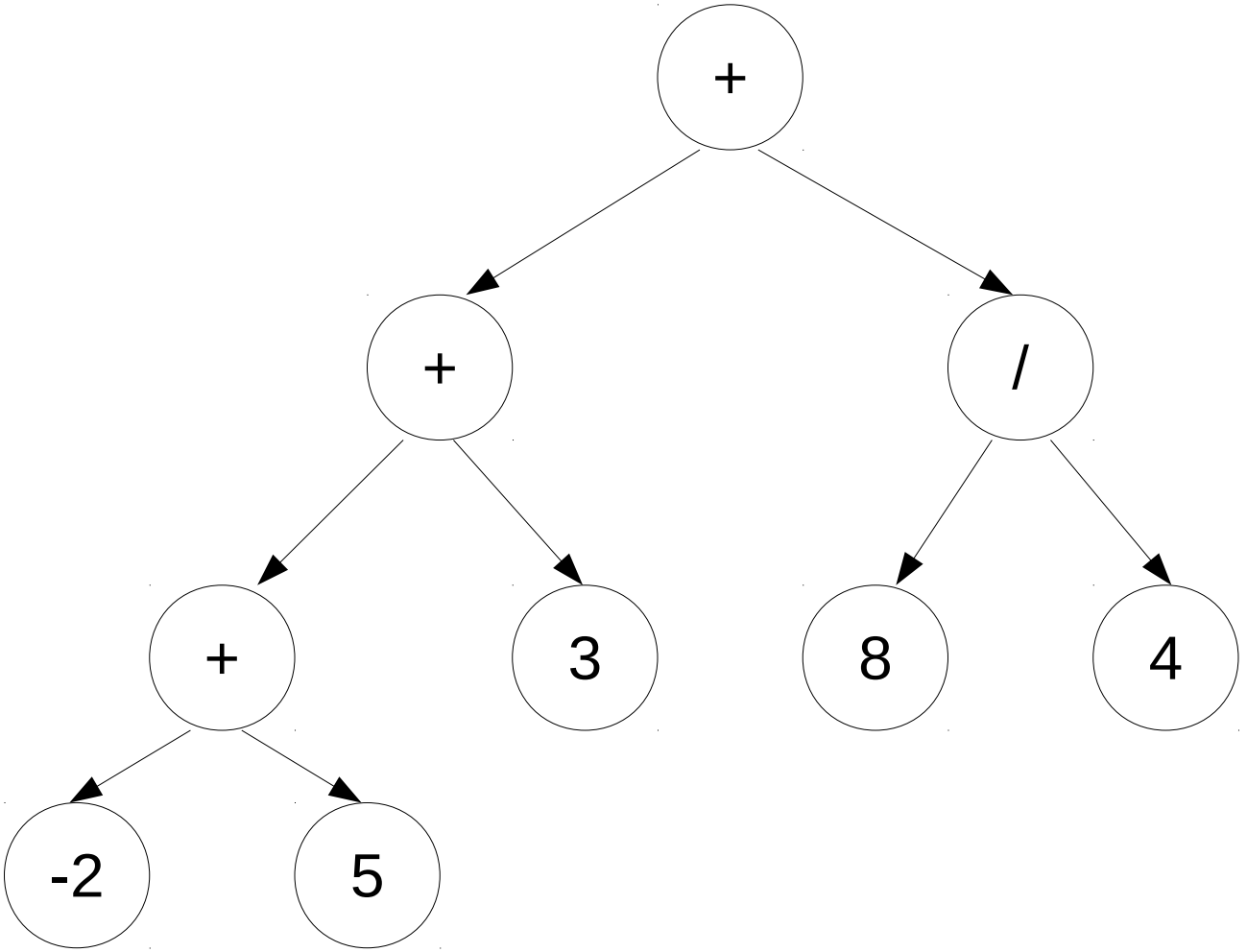
# Estructuras de Datos
# Conocer las formas de recorrer un árbol

Podemos recorrer los árboles en **anchura**: Primero visitamos los nodos en profundidad 0, luego en profundidad 1, etc.



# Estructuras de Datos
# Conocer las formas de recorrer un árbol

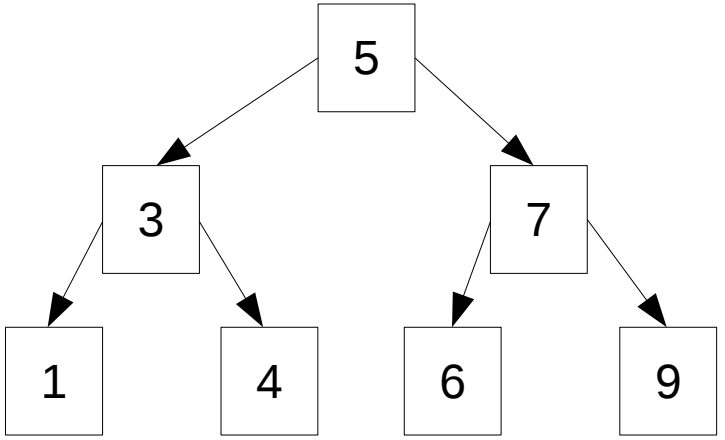
Para los árboles podemos tener iteradores preorden, postorden, inorden, y por anchura.



# Estructuras de Datos
# Comprender los conceptos de ABB, APO y AVL

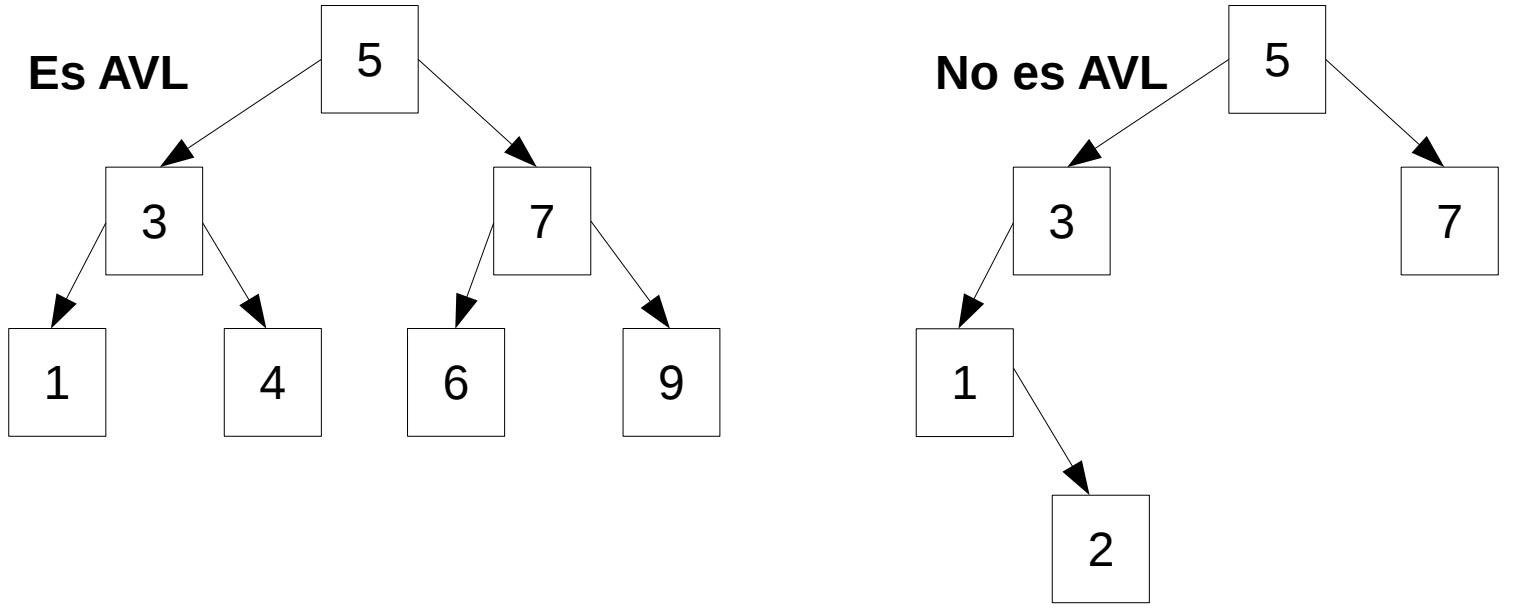
## ABB: Árbol binario de búsqueda.

Es un árbol en el que todos los elementos almacenados en el **subárbol izquierdo** de un nodo son **estrictamente menores** que el elemento almacenado en el nodo, y todos los elementos almacenados en el **subárbol derecho** del nodo son **estrictamente mayores** que el elemento almacenado en el nodo.



## AVL: Árbol equilibrado.

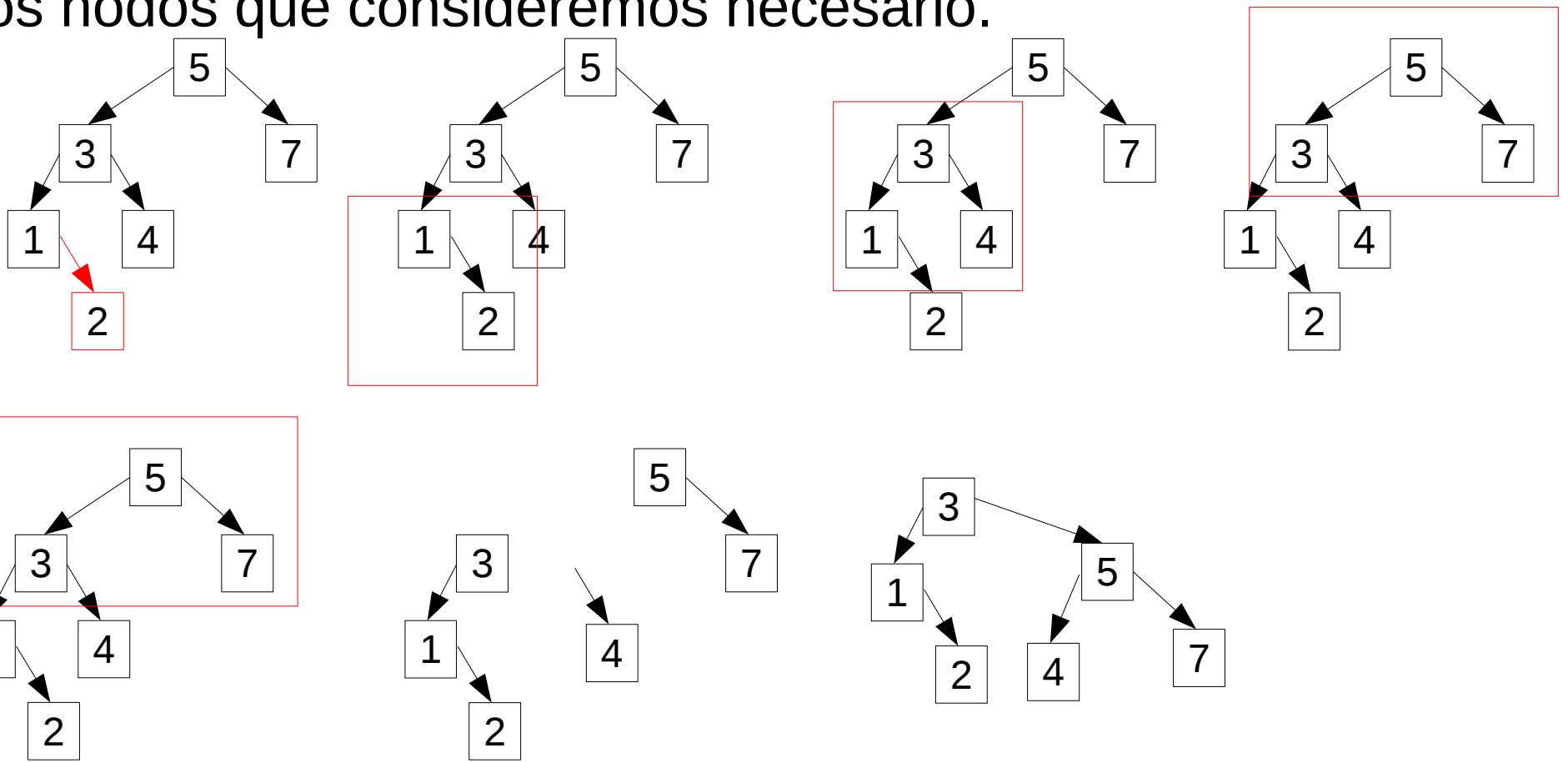
Es un ABB equilibrado. Su altura es O(log(n)), para cada nodo se cumple que la **diferencia de la altura** de sus dos hijos es menor o igual que uno (en valor absoluto). **Inserción y borrado** en orden de eficiencia O(log(n)).



# Estructuras de Datos
# Comprender los conceptos de ABB, APO y AVL

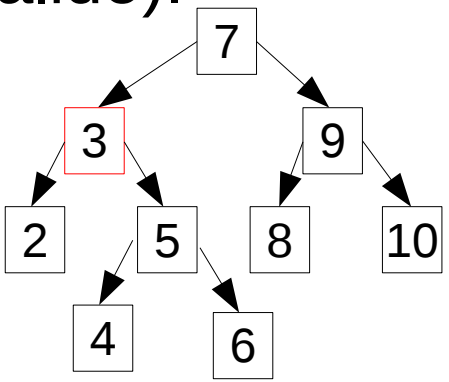
## AVL: Árbol equilibrado.

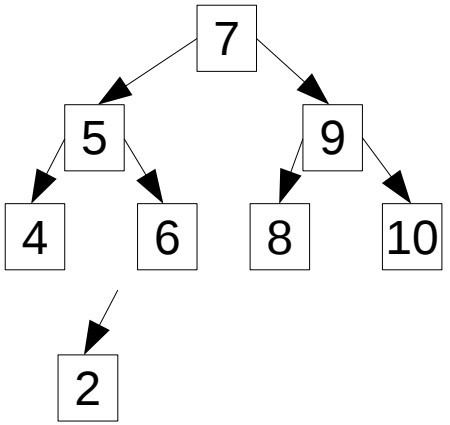
Al insertar en un AVL, lo hacemos en la posición oportuna según las etiquetas y vamos ascendiendo a la raíz rotando los nodos que consideremos necesario.

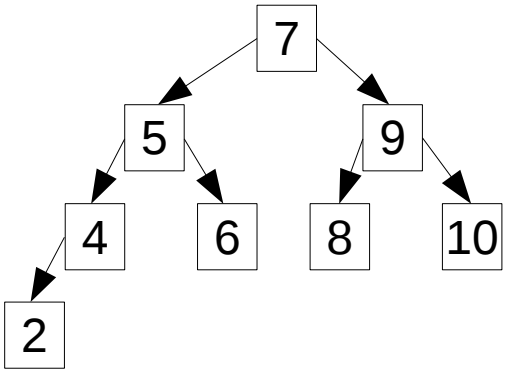


## AVL: Árbol equilibrado.

Al eliminar en un AVL, uno de los nodos hijos (el de mayor altura) asciende, el otro hijo que se queda descolgado se engancha donde debería estar (habrá un y solo un lugar válido).



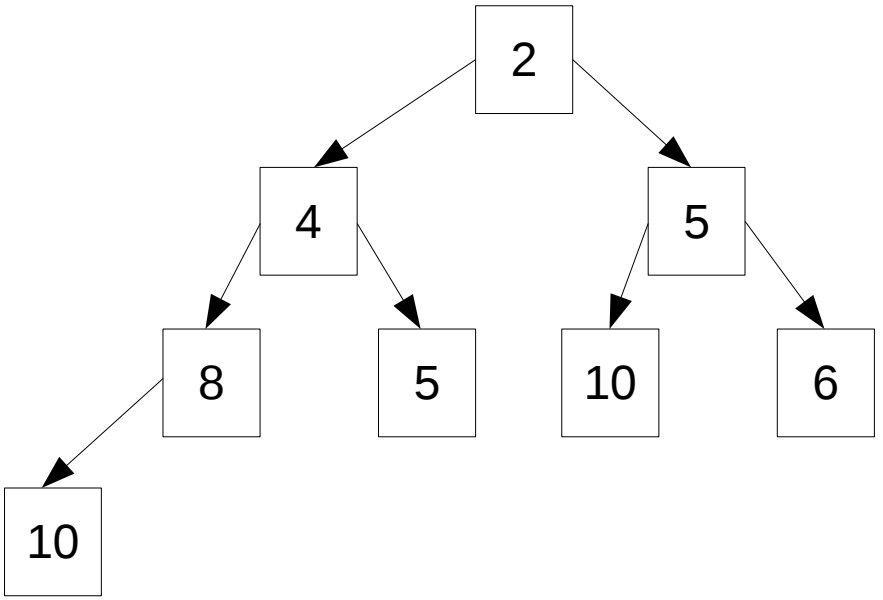



# Estructuras de Datos
# Comprender los conceptos de ABB, APO y AVL

## APO: Árbol parcialmente ordenado.

Es un árbol equilibrado que permite obtener el mínimo de un conjunto de datos de forma eficiente, en orden O(log(n)).

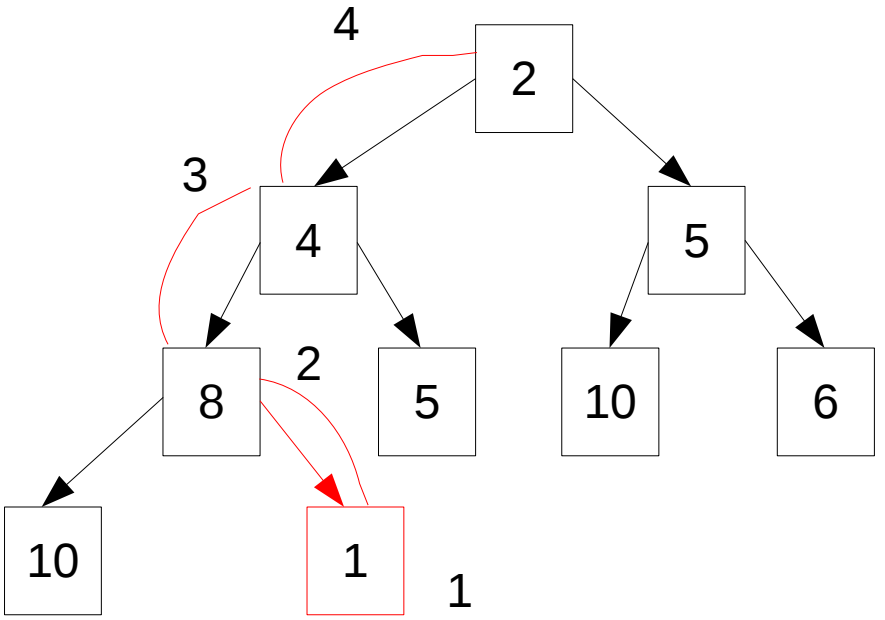
La etiqueta de cada nodo es menor o igual a la de sus hijos.



# Estructuras de Datos
# Comprender los conceptos de ABB, APO y AVL

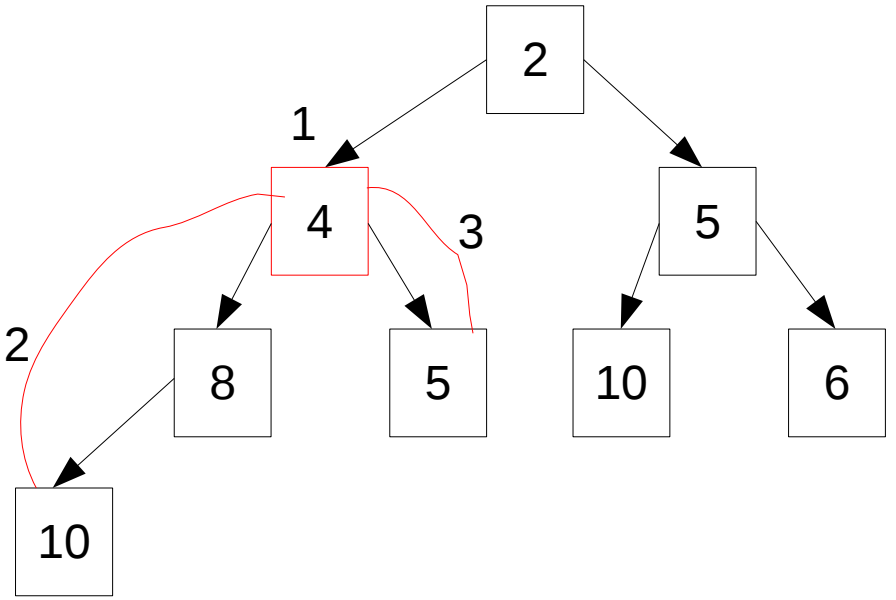
## APO: Árbol parcialmente ordenado.

Se insertan los elementos lo más a la izquierda posible en el último nivel y van ascendiendo mientras la etiqueta de sus padres sea mayor (o menor, según si una etiqueta más baja significa más o menos prioridad!).



## APO: Árbol parcialmente ordenado.

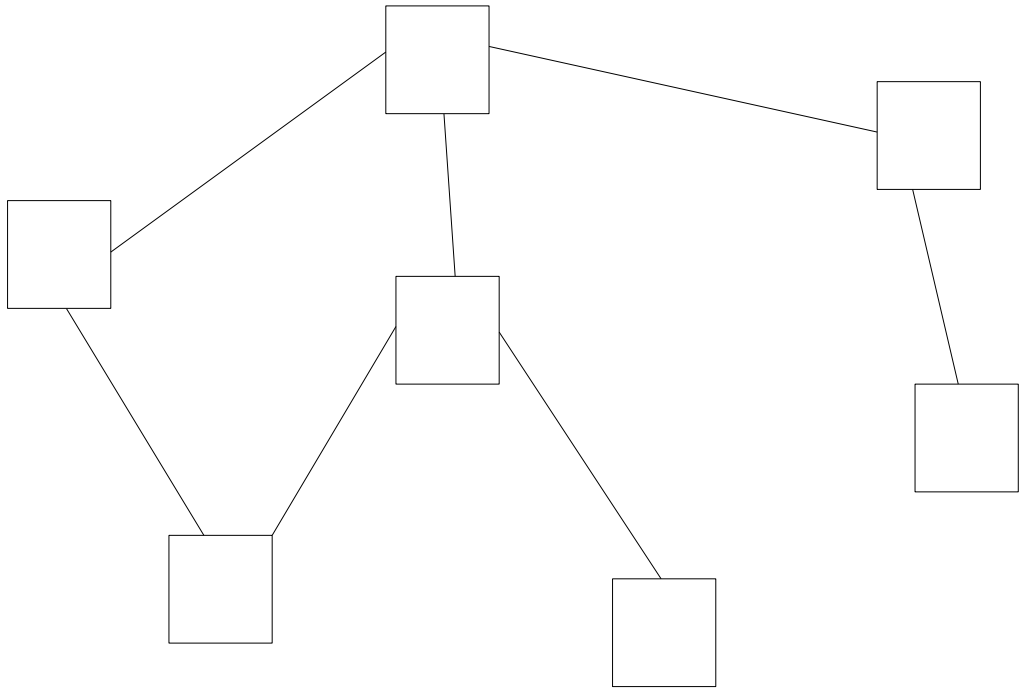
Se borran los elementos reemplazándolos por el último del último nivel y haciendo que este último descienda hasta donde deba, sustituyendo siempre a su hijo menor (o mayor, según si una etiqueta más baja significa más o menos prioridad!).



# Estructuras de Datos
# Comprender el concepto de grafo

Un grafo es un conjunto de **vértices** o **nodos (V)** y un conjunto de **lados (E)** que los unen.

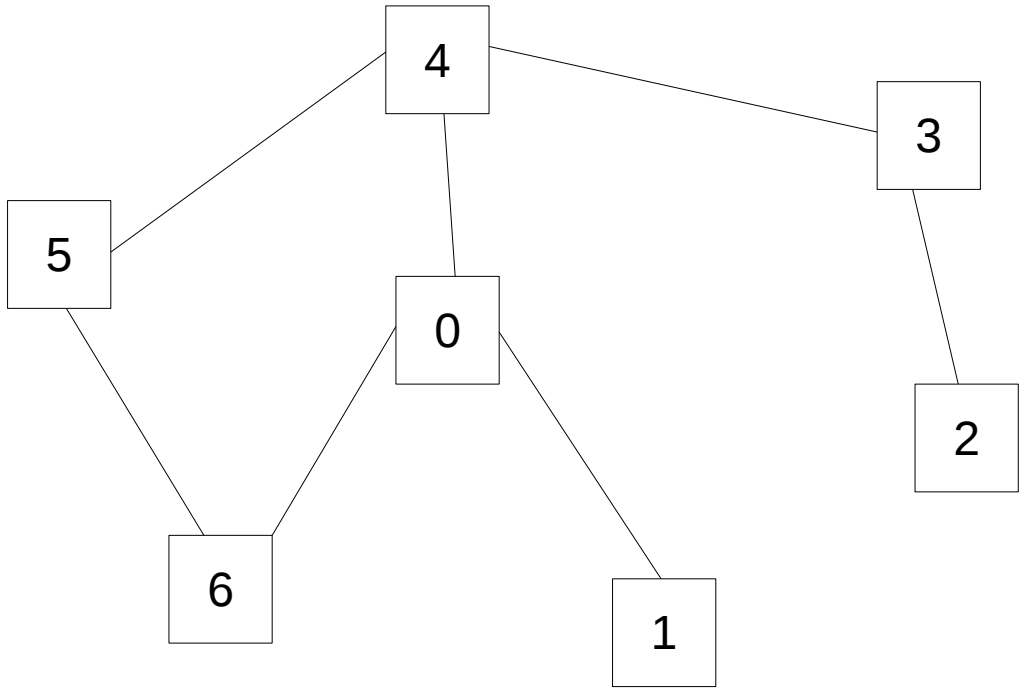
Se diferencia de un árbol en que puede haber **ciclos**. Un nodo no tiene padre o hijos, sino que esté conectado con otros.



# Estructuras de Datos
# Comprender el concepto de grafo

Supondremos que los vértices están numerados, para poder referirnos a ellos.

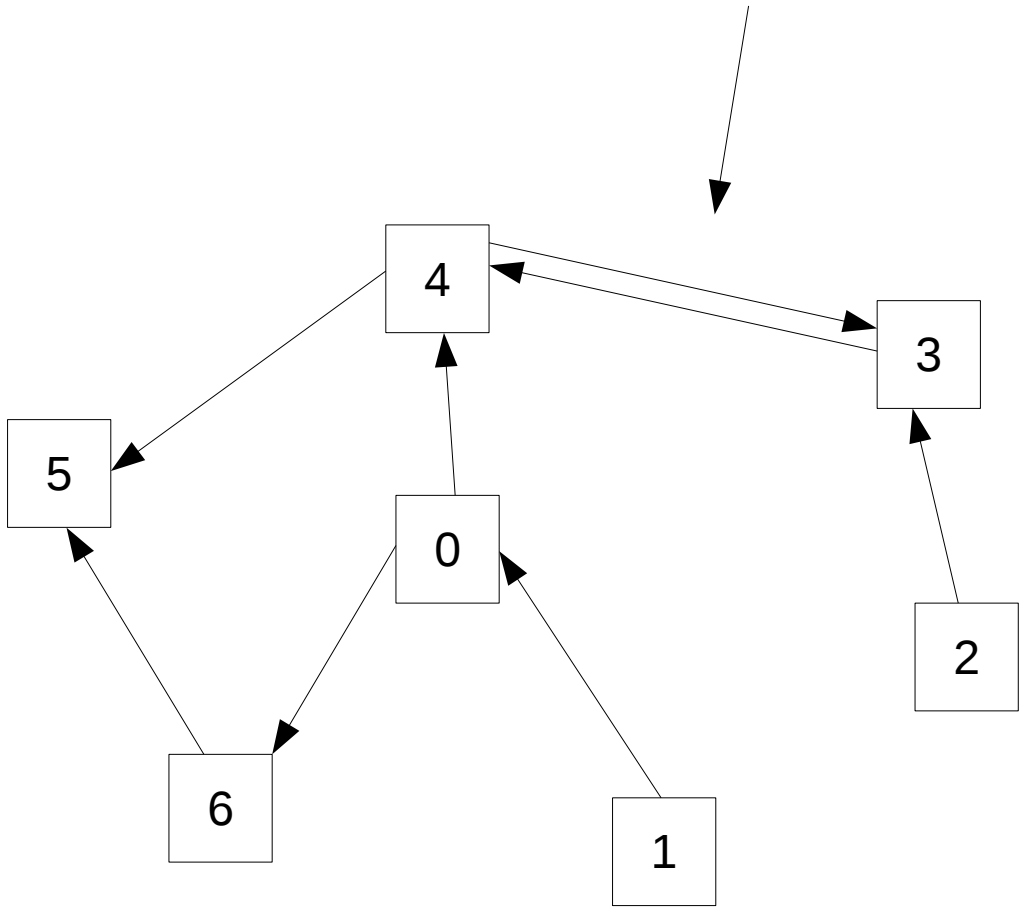
El número de lados será siempre igual o menor al número de vértices al cuadrado.



# Estructuras de Datos
# Comprender el concepto de grafo

Puede ser dirigido. En tal caso los lados se denominan arcos y permiten la navegabilidad en un sólo sentido.

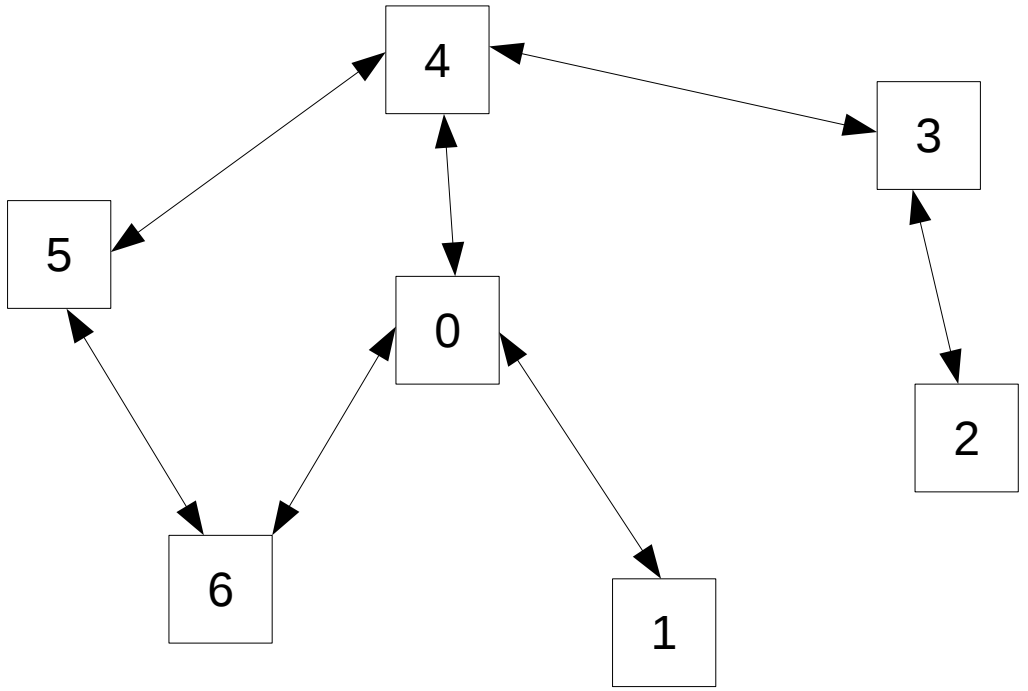
En este caso el lado (x,y) no equivale al lado (y,x).



# Estructuras de Datos
# Comprender el concepto de grafo

Puede ser no dirigido. En tal caso los lados se denominan aristas y permiten la navegabilidad en ambos sentidos.

En este caso, el lado (x,y) equivale a (y,x).

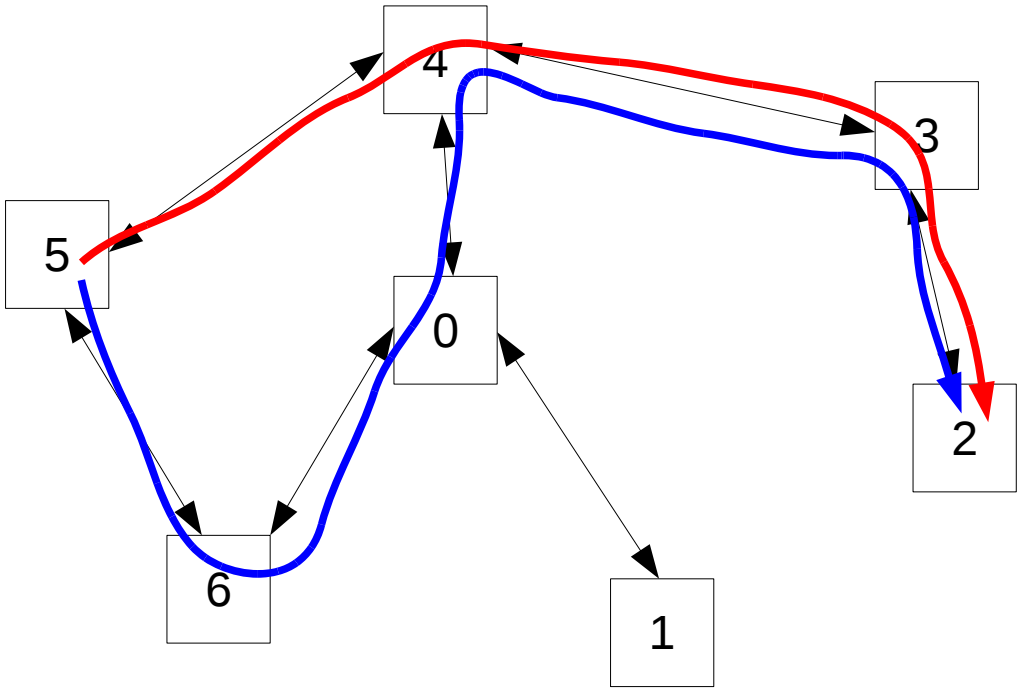


# Estructuras de Datos
# Comprender el concepto de grafo

Se dice que existe un camino entre dos vértices si podemos navegar de uno a otro. Hay dos caminos entre los vértices 5 y 2:

Camino 1 (rojo): (5,4) (4,3) (3,2)

Camino 2 (azul): (5,6) (6,0) (0,4) (4,3) (3,2)

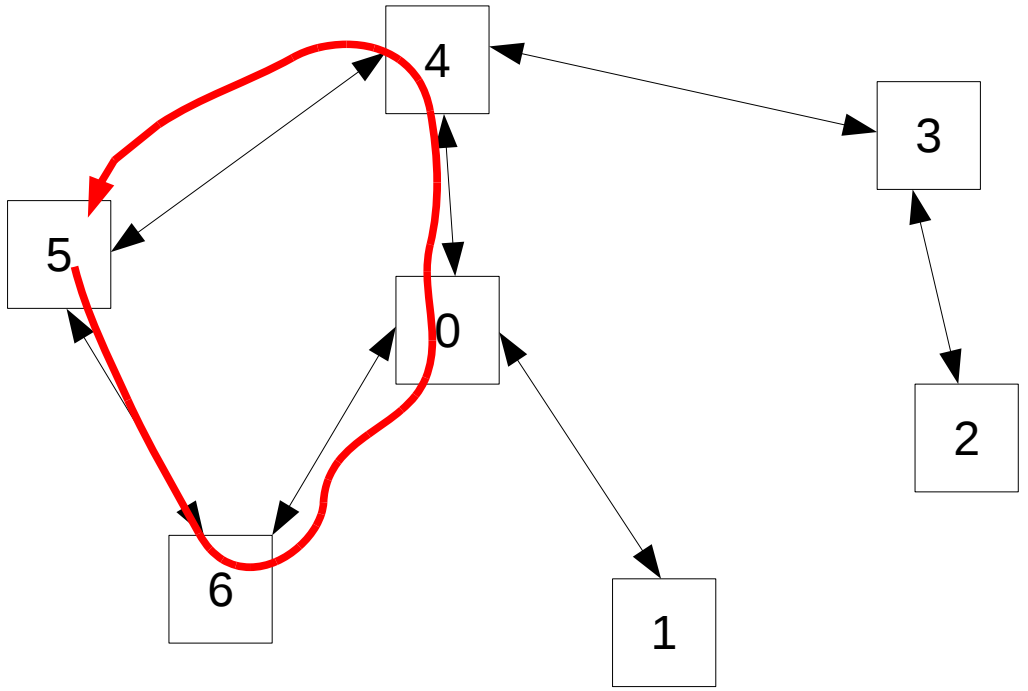


# Estructuras de Datos
# Comprender el concepto de grafo

Un camino es un ciclo si empieza en el mismo vértice en el que acaba.

Ciclo que empieza y acaba en 5 (rojo): (5,6) (6,0) (0,4) (4,5)

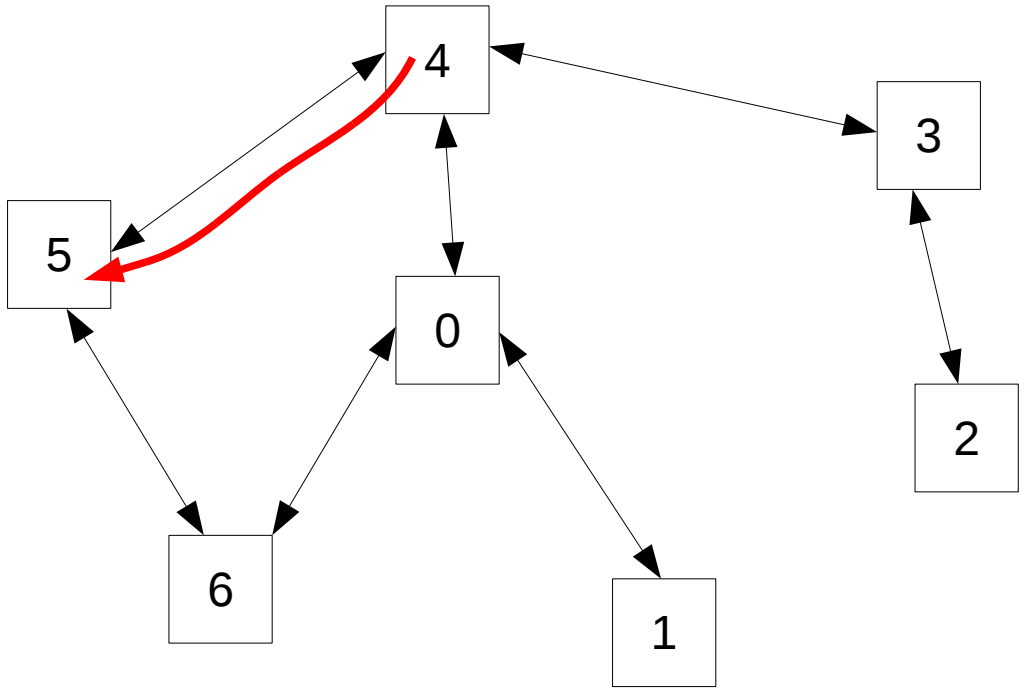
Un camino se dice simple si en él no hay ciclos.



# Estructuras de Datos
# Comprender el concepto de grafo

Se dice que un vértice v es adyacente a un vértice u si existe el lado u,v.

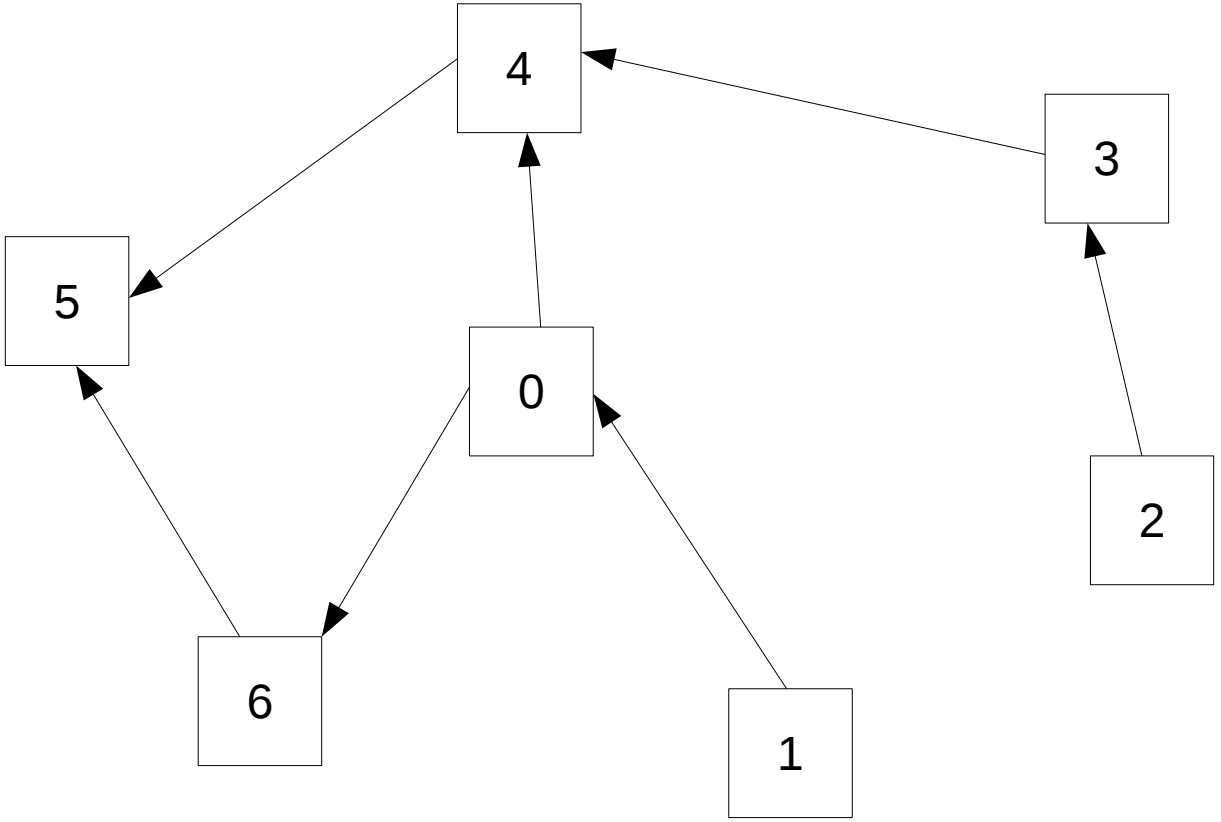
5 es adyacente a 4 porque existe (4,5)



# Estructuras de Datos
# Comprender el concepto de grafo

## Casos particulares importantes de grafos:

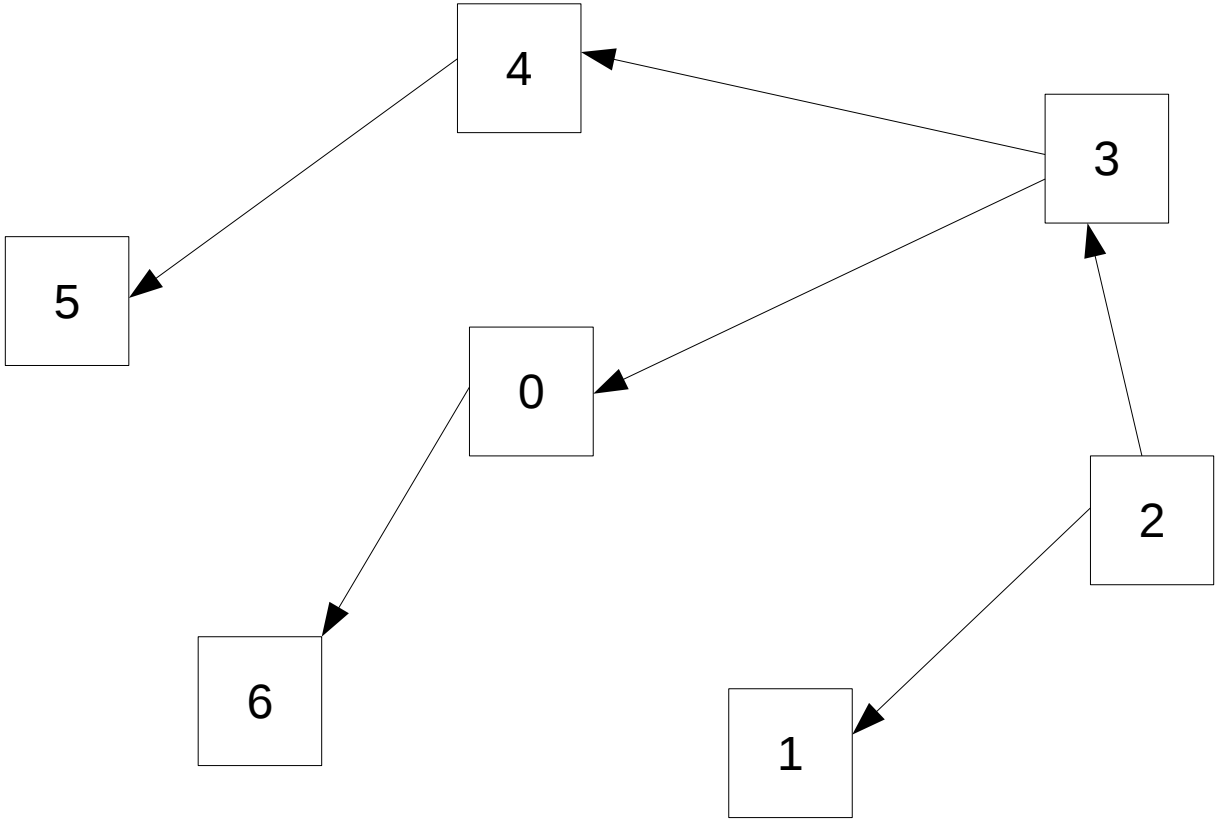
Grafos dirigidos acíclicos (GDA): sin ciclos, dirigidos.



# Estructuras de Datos
# Comprender el concepto de grafo

## Casos particulares importantes de grafos:

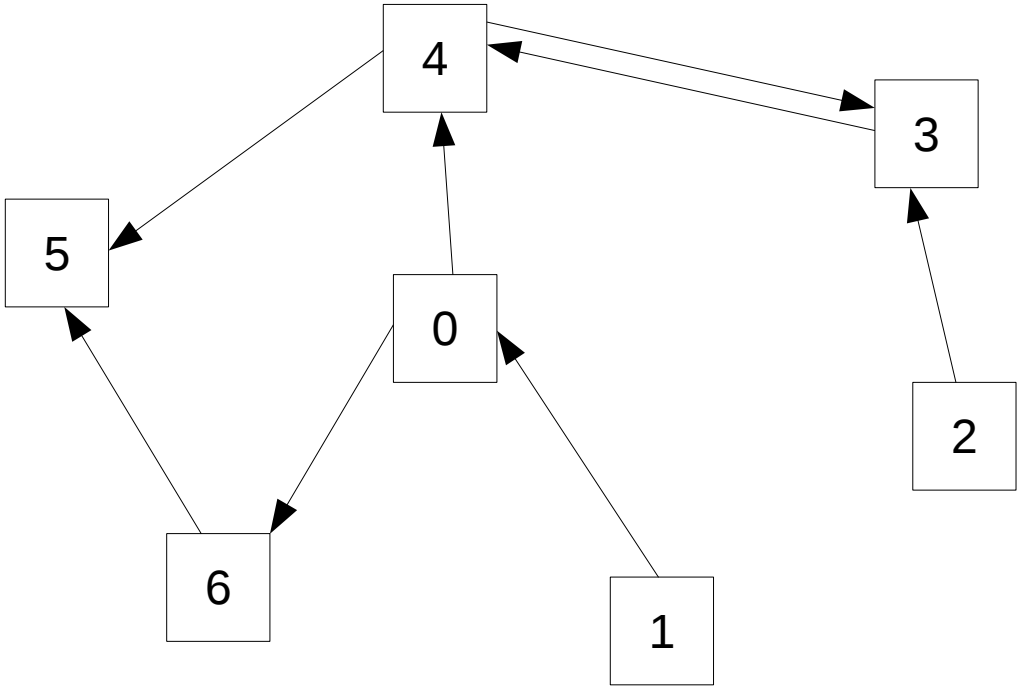
Árboles: sin ciclos, cada nodo es destino de un lado, salvo la raíz que lo es de ninguno. Se puede llegar desde la raíz hasta cualquier sitio.



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

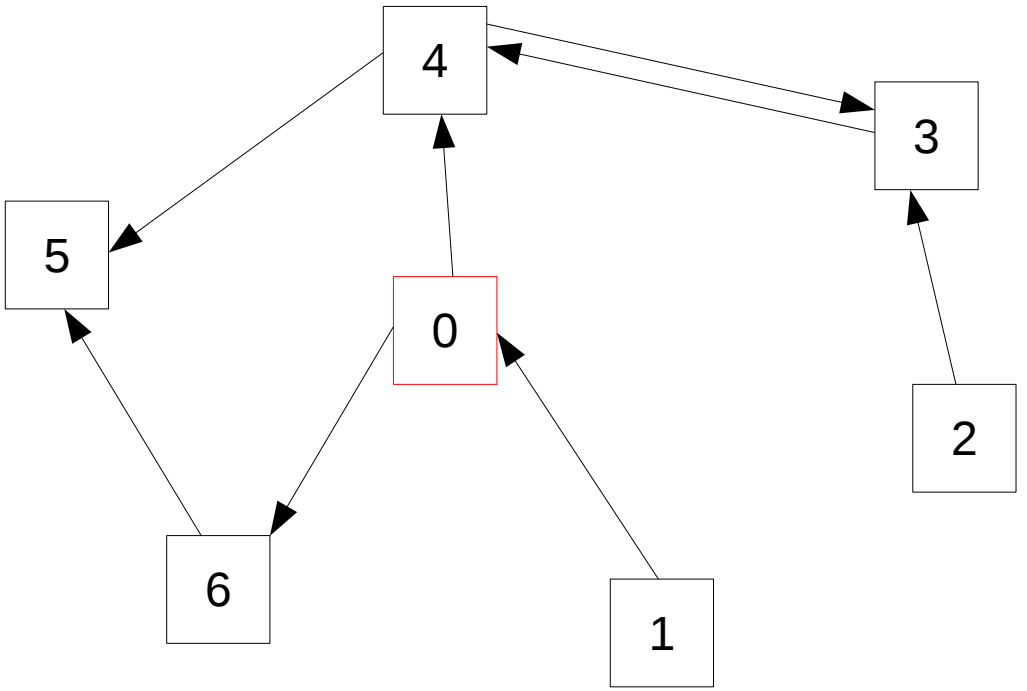
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

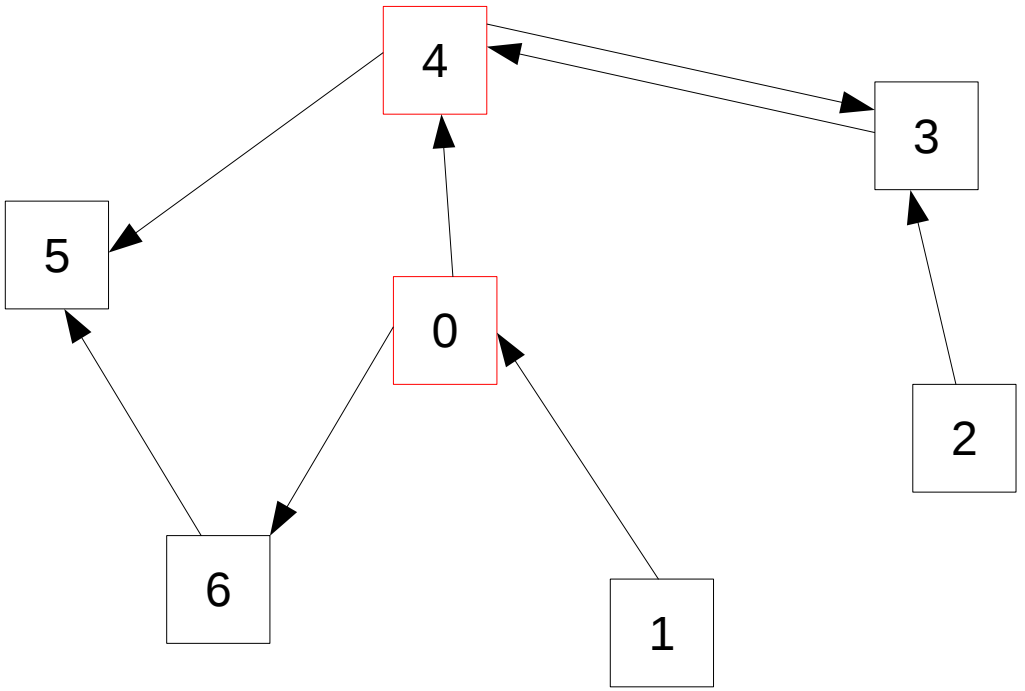
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

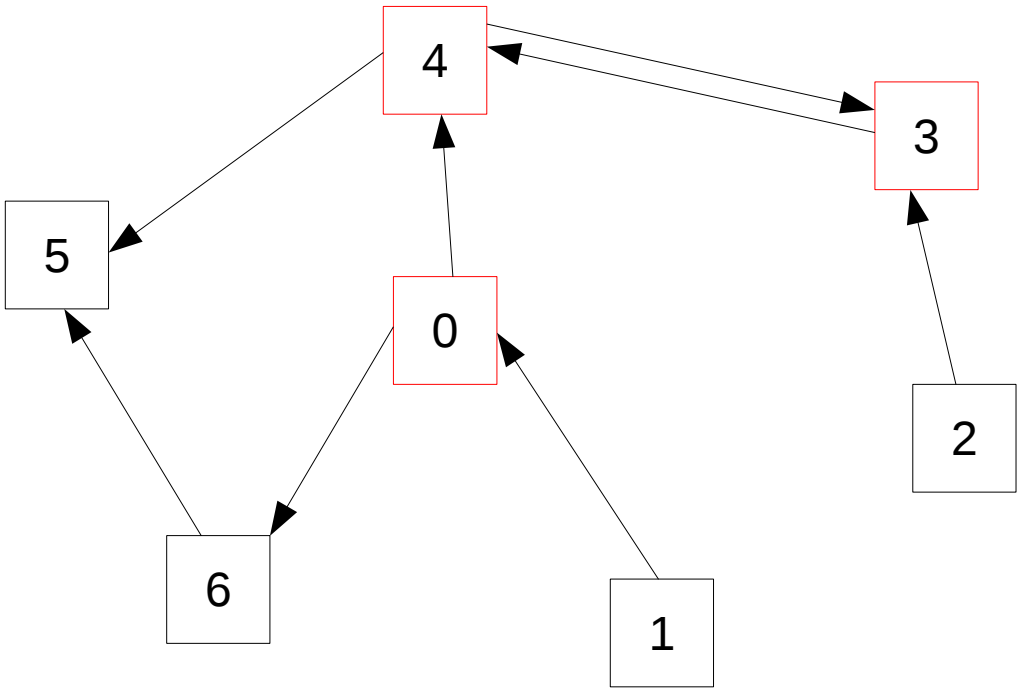
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

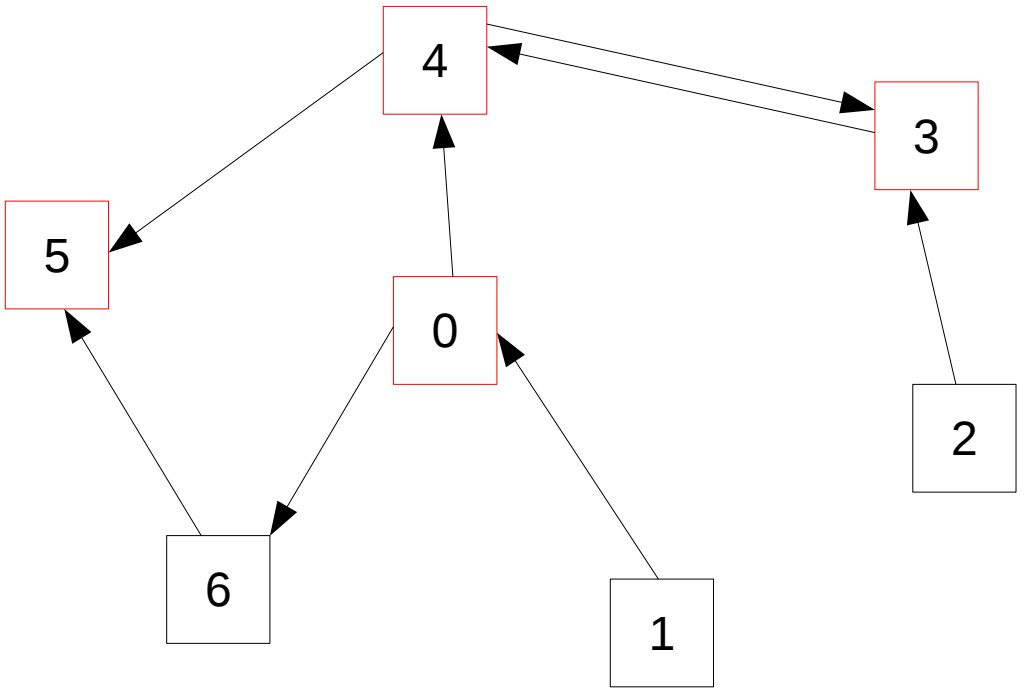
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

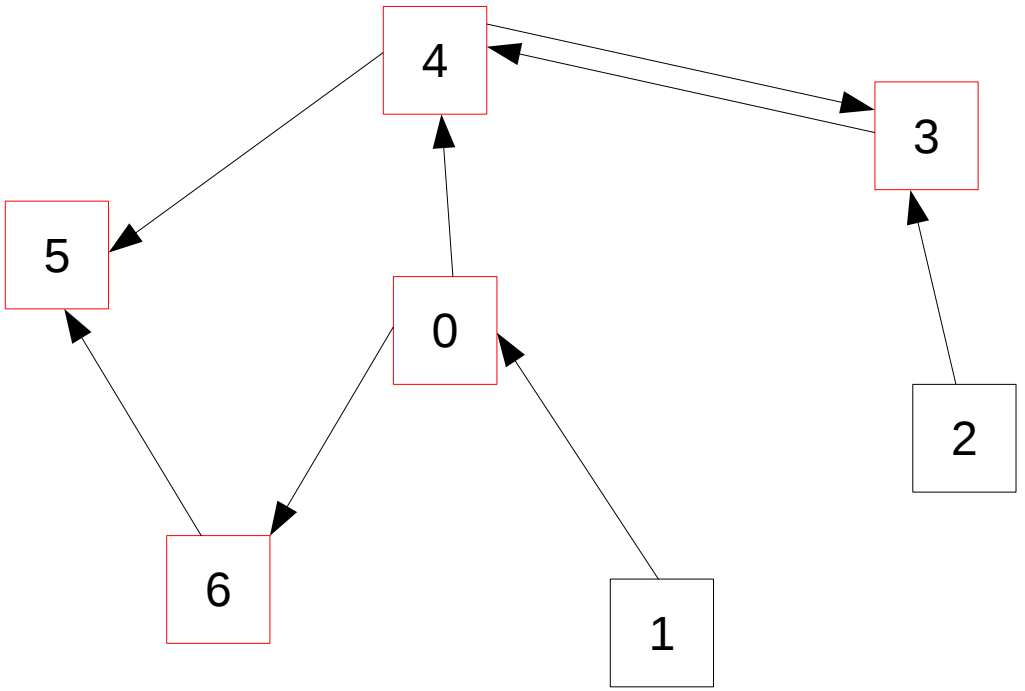
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

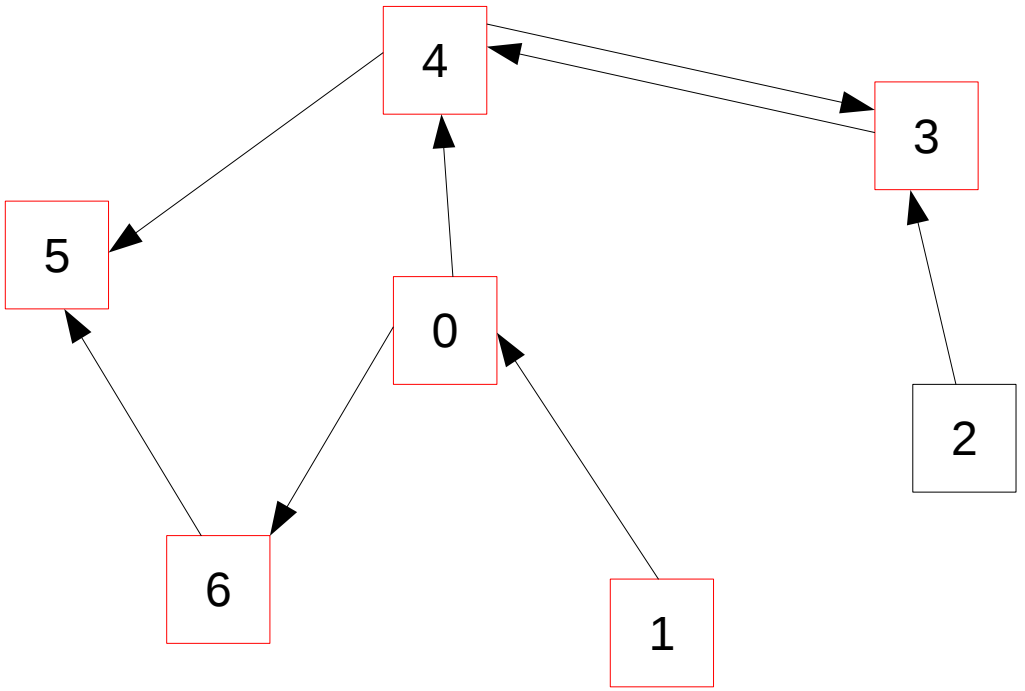
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

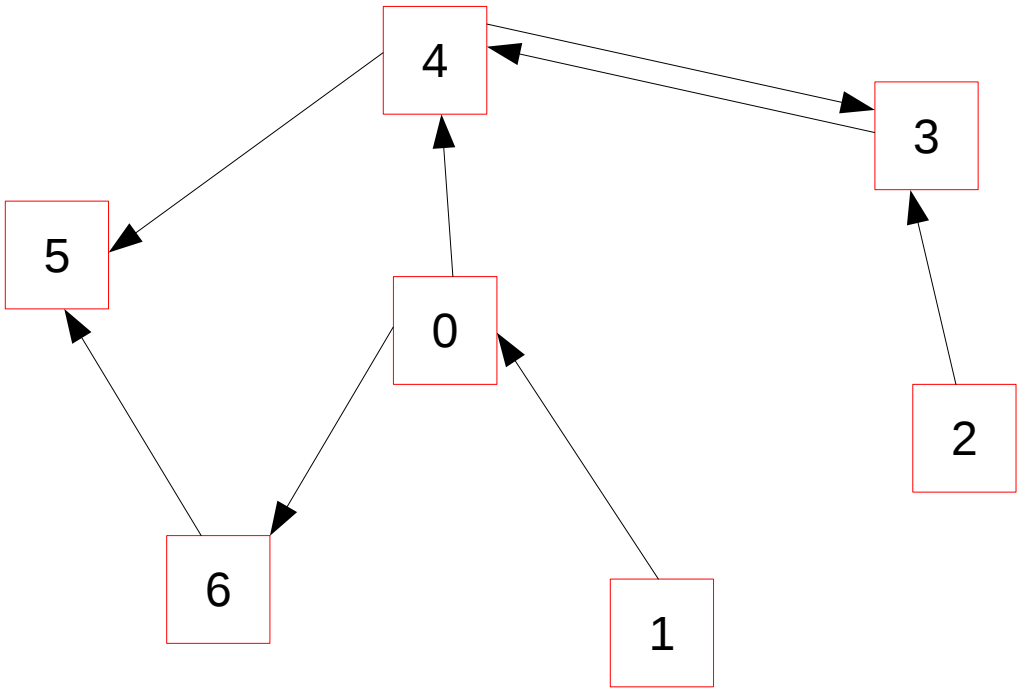
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

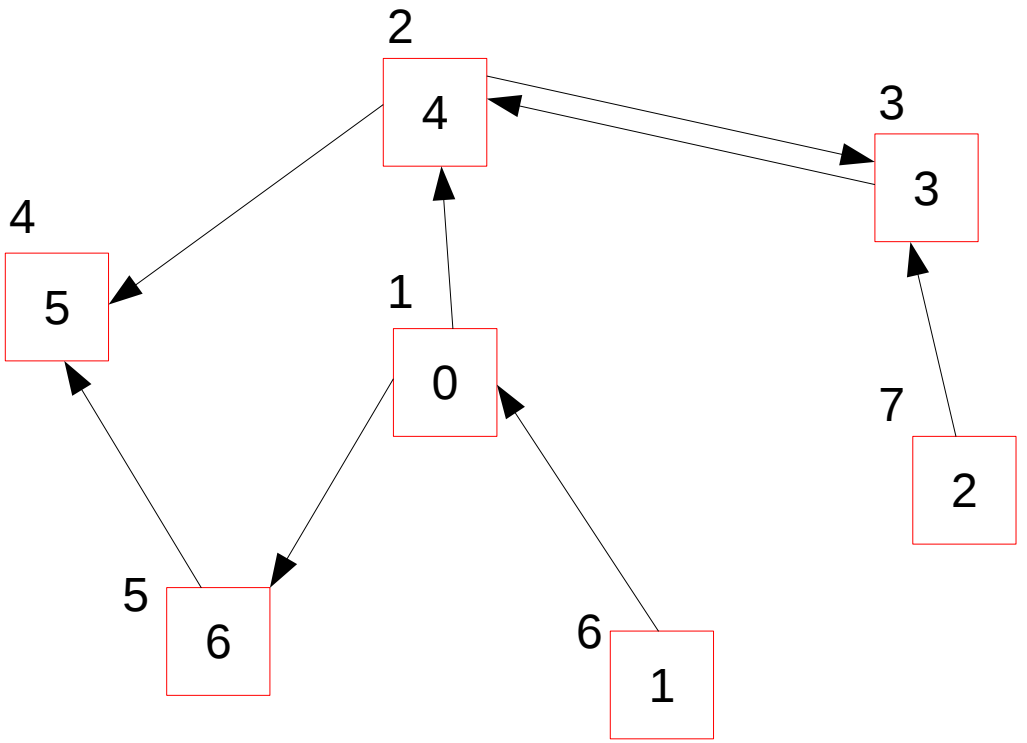
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

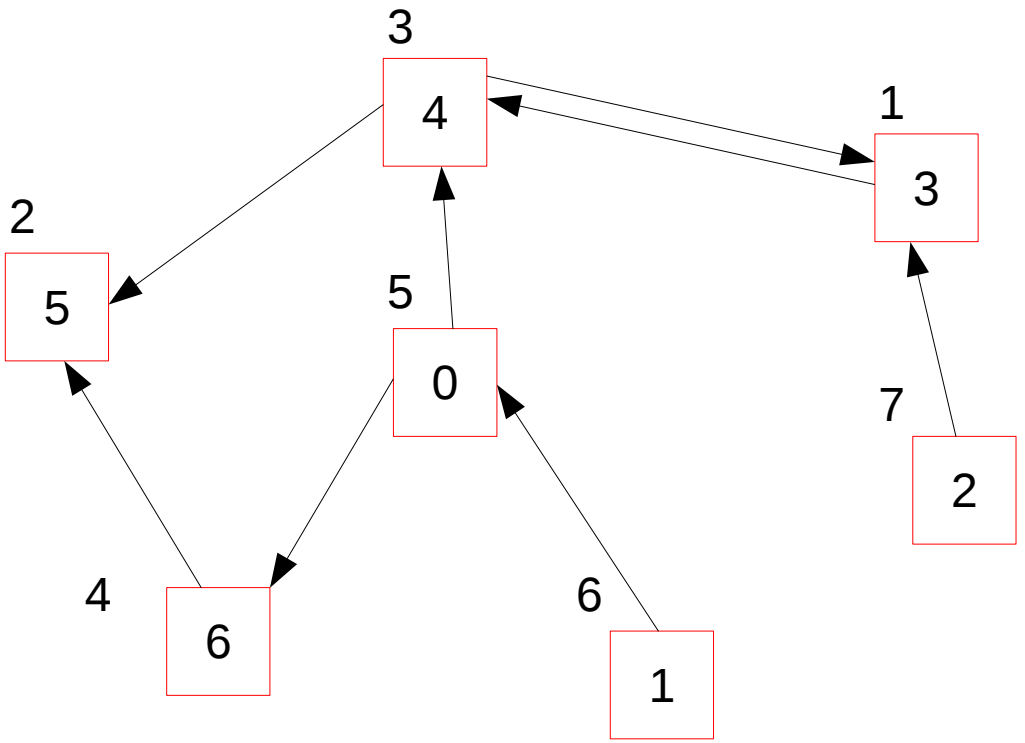
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  – PREORDEN
  nump = nump + 1;
  prenump[v] = nump;
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en profundidad (preorden/postorden) de un grafo:

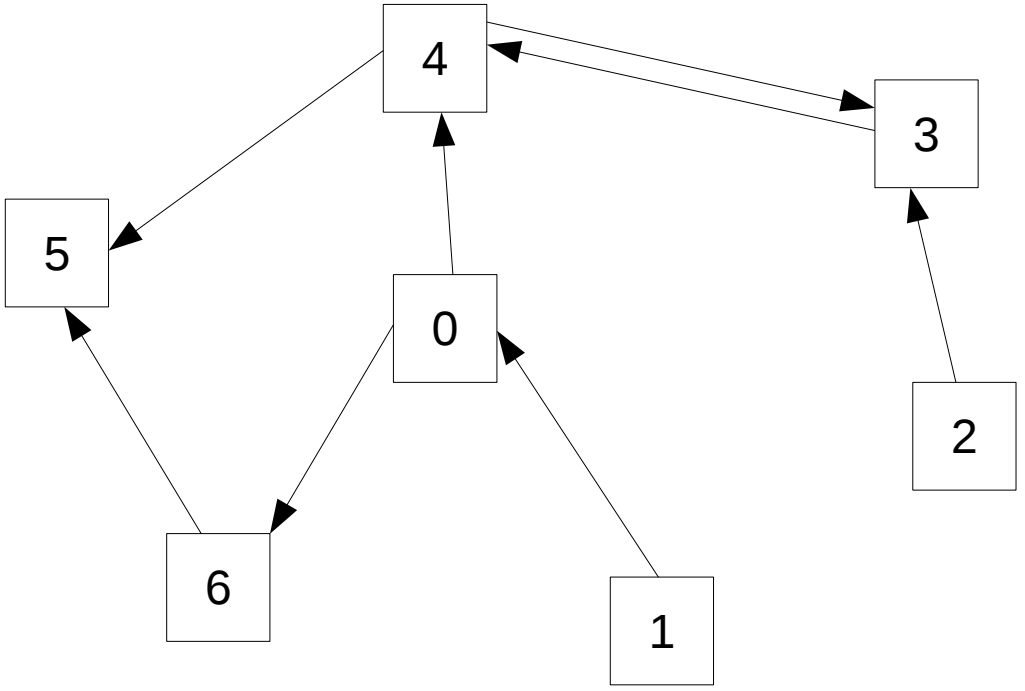
```
RecorridoProfundidad() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerP(ese vértice).
}
RecorrerP(vertice) {
  Se marca el vértice como visitado.
  Para cada vértice adyacente a ese:
    Si aún no está visitado:
      RecorrerP(ese vértice).
  – POSTORDEN
  nump = nump + 1;
  prenump[v] = nump;
}
```



# Estructuras de Datos
# Comprender el concepto de grafo

## Recorrido en anchura (generalización niveles) de un grafo:

```
RecorridoAnchura() {
  Se marcan todos los vértices como no visitados.
  Para cada vértice i=0->n:
    Si aún no está visitado:
      RecorrerA(ese vértice).
}
RecorrerA(vertice) {
  Se marca el vértice como visitado.
  Se inserta el vértice en una cola.
  Mientras queden vértices en la cola {
    Se obtiene el primero.
    Para cada vértice adyacente a ese:
      Si no está visitado:
        Se marca como visitado
        Y se inserta en la cola
}
```



# Estructuras de Datos
# ¿Qué hemos aprendido?

- **Diferencia entre estructura de datos y TDA.**
- **Concepto de contenedor.**
- **Concepto de adaptador.**
- **Principales tipos de contenedores básicos.**
- **Concepto de iterador.**
- **Concepto de tabla hash.**
- **Concepto de árbol.**
- **Formas de recorrer un árbol**.
- **Conceptos de ABB, APO y AVL.**
- **Concepto de grafo.**