



**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**  
**FACULTAD DE INGENIERÍA**  
**ESTRUCTURAS DE DATOS Y ALGORITMOS II**

---

Tema 4

Árboles

# 4 Árboles

Objetivo: Aplicar las formas de representar y operar las listas lineales para representarlos en la computadora.

# 4 Árboles

4.1 Notaciones: infija, prefija, sufija.

4.2 Árboles binarios.

    4.2.1 Definiciones y operaciones.

    4.2.2 Transformación de árboles a árboles binarios.

    4.2.3 Recorrido de árboles.

    4.2.4 Representación en la computadora.

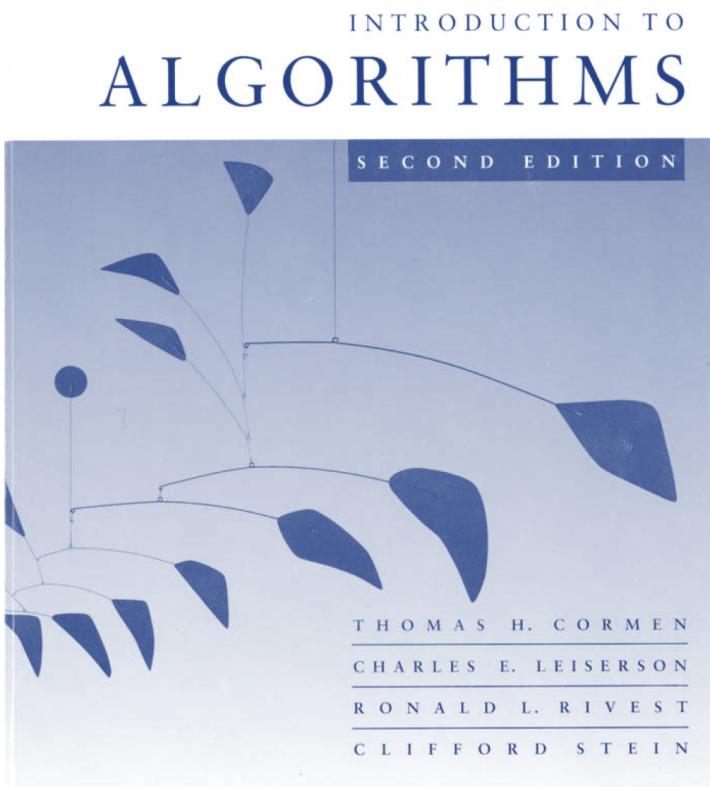
4.3 Árboles B.

    4.3.1 Árboles B.

    4.3.2 Árboles B+, algoritmos.

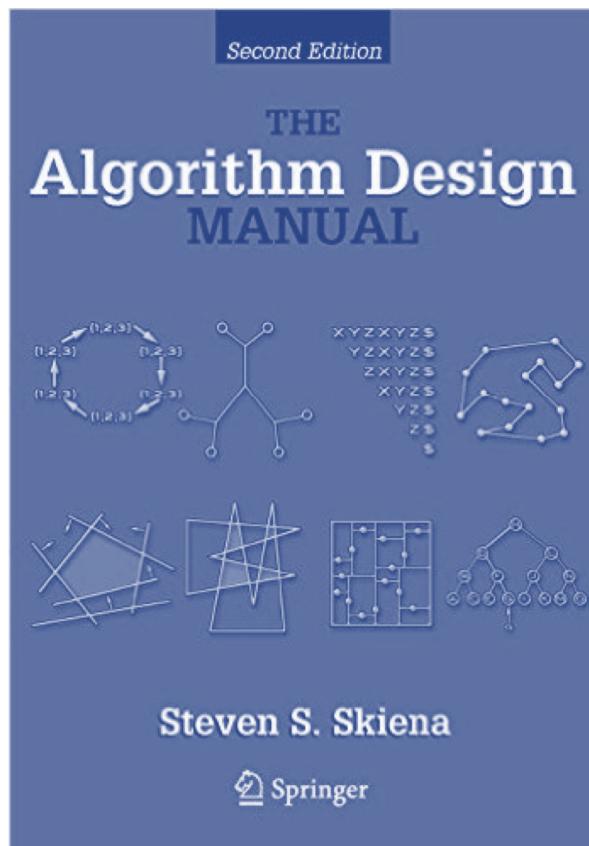
    4.3.3 Árboles B+ prefijos simples, algoritmos.

# Bibliografía



- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, McGraw-Hill.

# Bibliografía



- The Algorithm Design Manual.  
Steven S. Skiena, Springer.

# Árboles



- Tema 4

Los árboles son estructuras de datos no lineales y dinámicas. Estas estructuras son muy útiles en las ciencias de la computación (y en general en la rama de las TI).

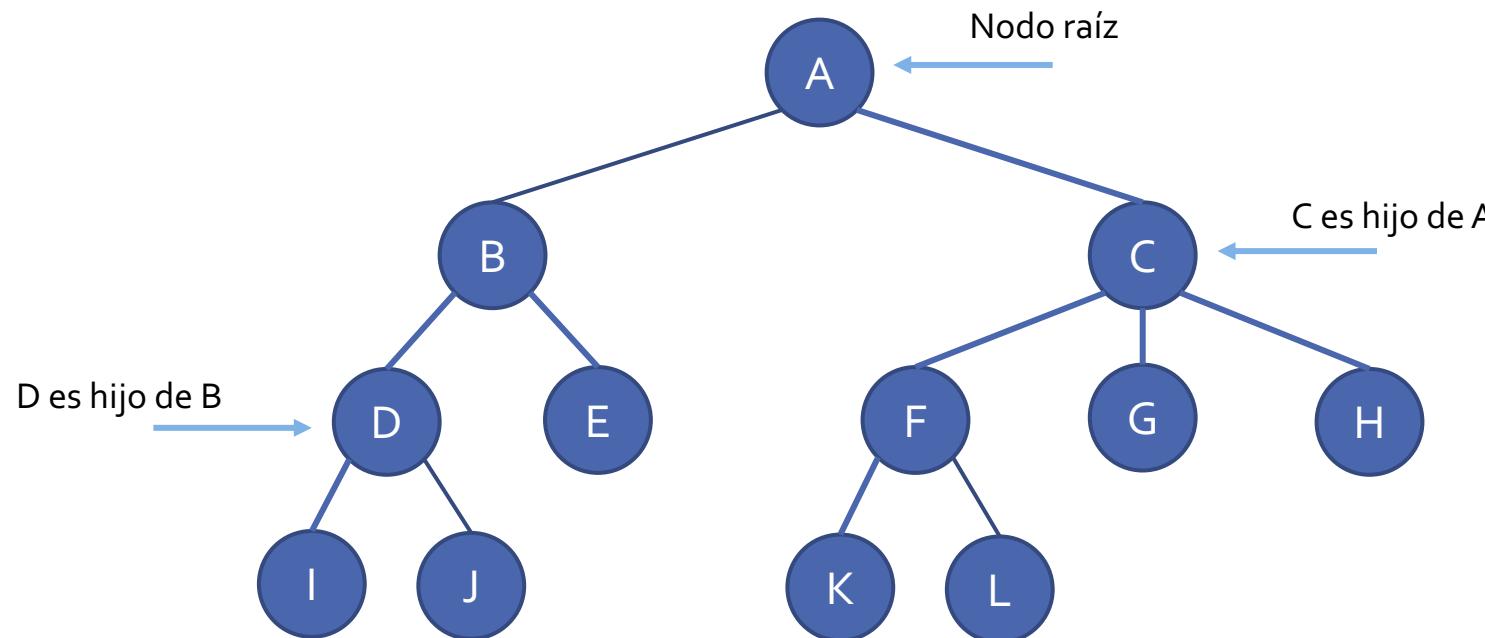
Un árbol es una estructura jerárquica aplicada sobre una colección de elementos u objetos llamados nodos. El primer nodo de un árbol se conoce como nodo raíz y a partir de él se dibuja la jerarquía.

Algunas características y propiedades que poseen las estructuras de datos no lineales tipo árbol son:

- Todo árbol que no es vacío posee un único nodo raíz.
- Un nodo X es descendente directo de un nodo Y si y solo si el nodo X es apuntado por el nodo Y (X es hijo de Y).
- Un nodo Y es antecesor directo de un nodo X si y solo si el nodo Y apunta al nodo X (Y es padre de X).
- Si varios nodos descienden de manera directa de un mismo nodo, entonces se dice que éstos son hermanos.
- Si un nodo no posee ramificaciones (o hijos) se denomina nodo terminal o nodo hoja.

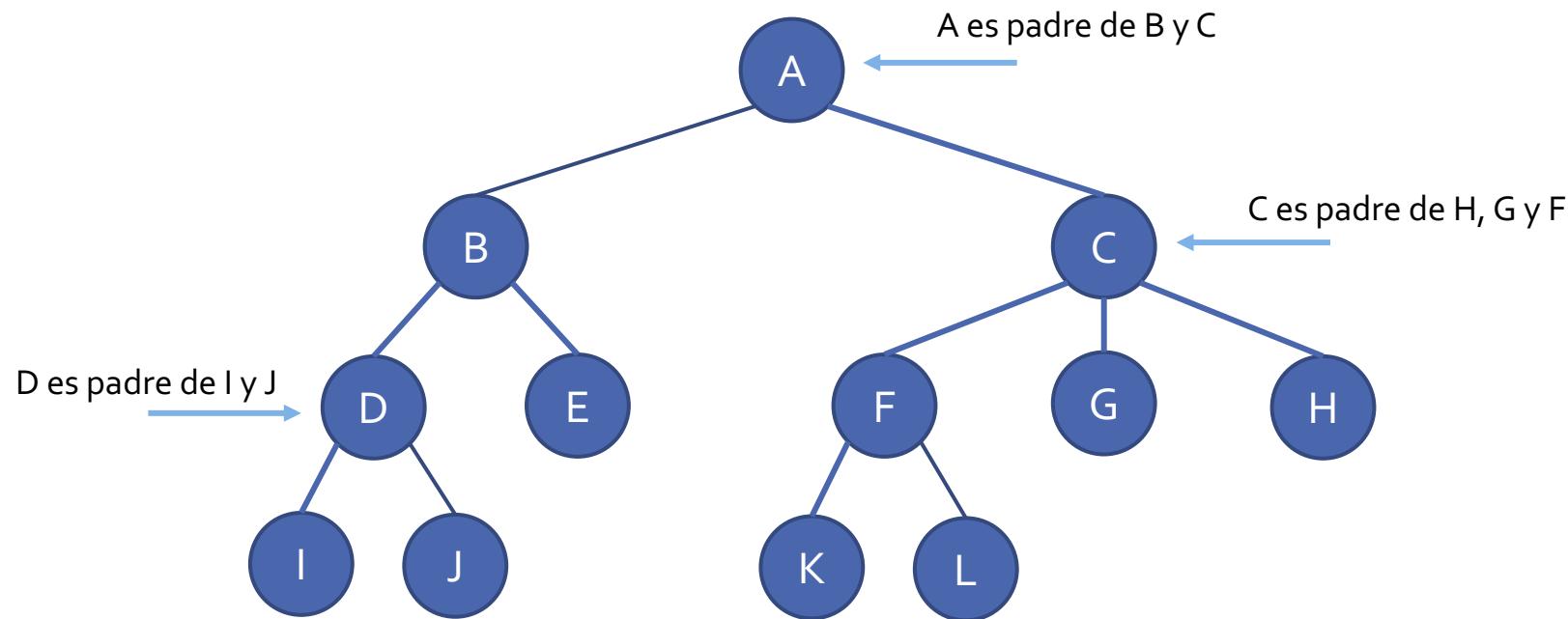
- Cualquier nodo diferente del nodo raíz o del nodo hoja se le conoce como nodo interior.
- El grado de un nodo está determinado por el número de descendentes directos.
- El grado del árbol está determinado por el máximo grado de todos los nodos.
- El nivel de un nodo está determinado por el número de arcos que deben ser recorridos para llegar a dicho nodo más uno.
- La altura de un árbol está determinada por el número máximo de niveles de todos los nodos del árbol.

Una estructura árbol se puede representar mediante un grafo de la siguiente manera:

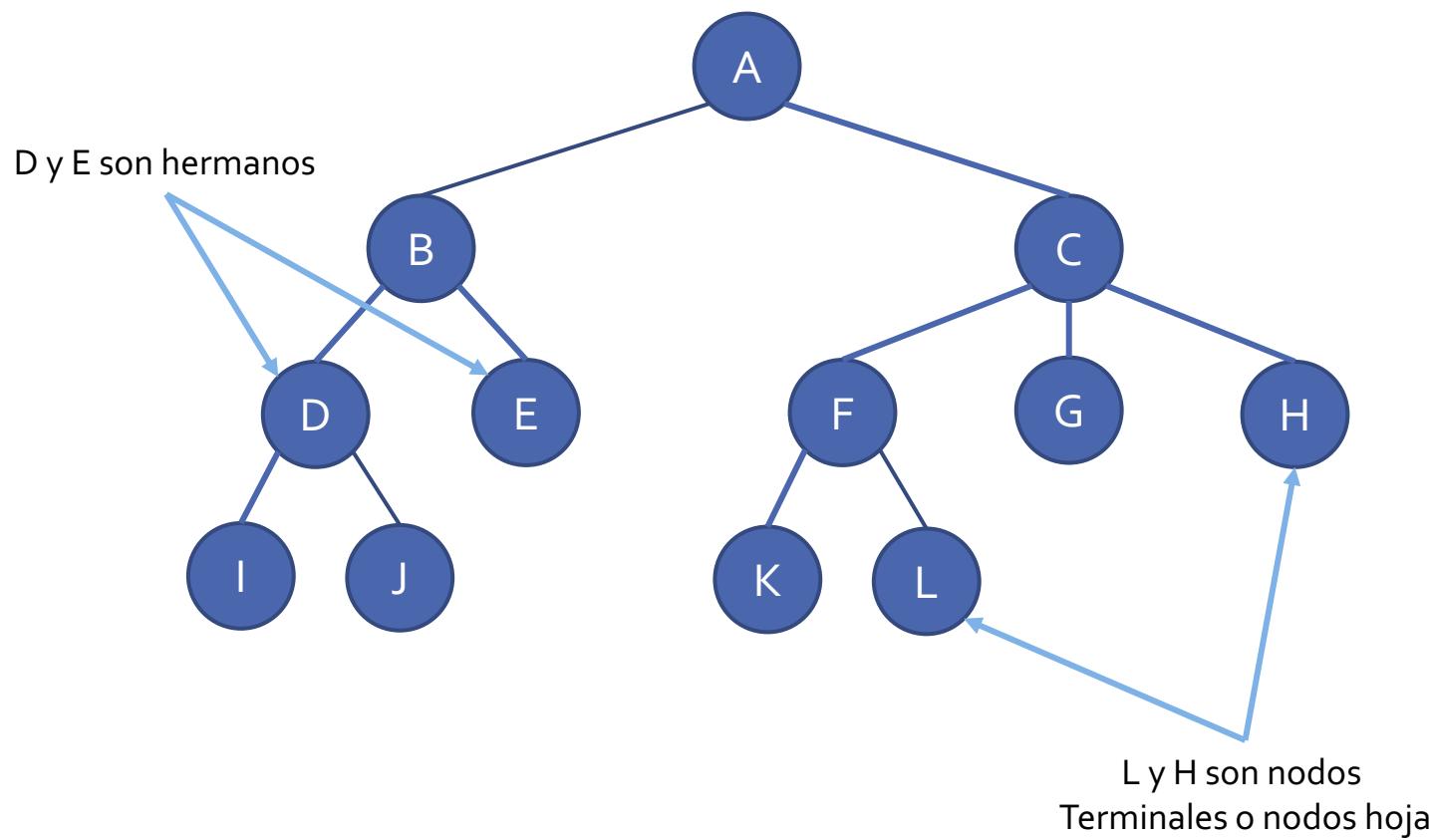


Árbol ≠ vacío

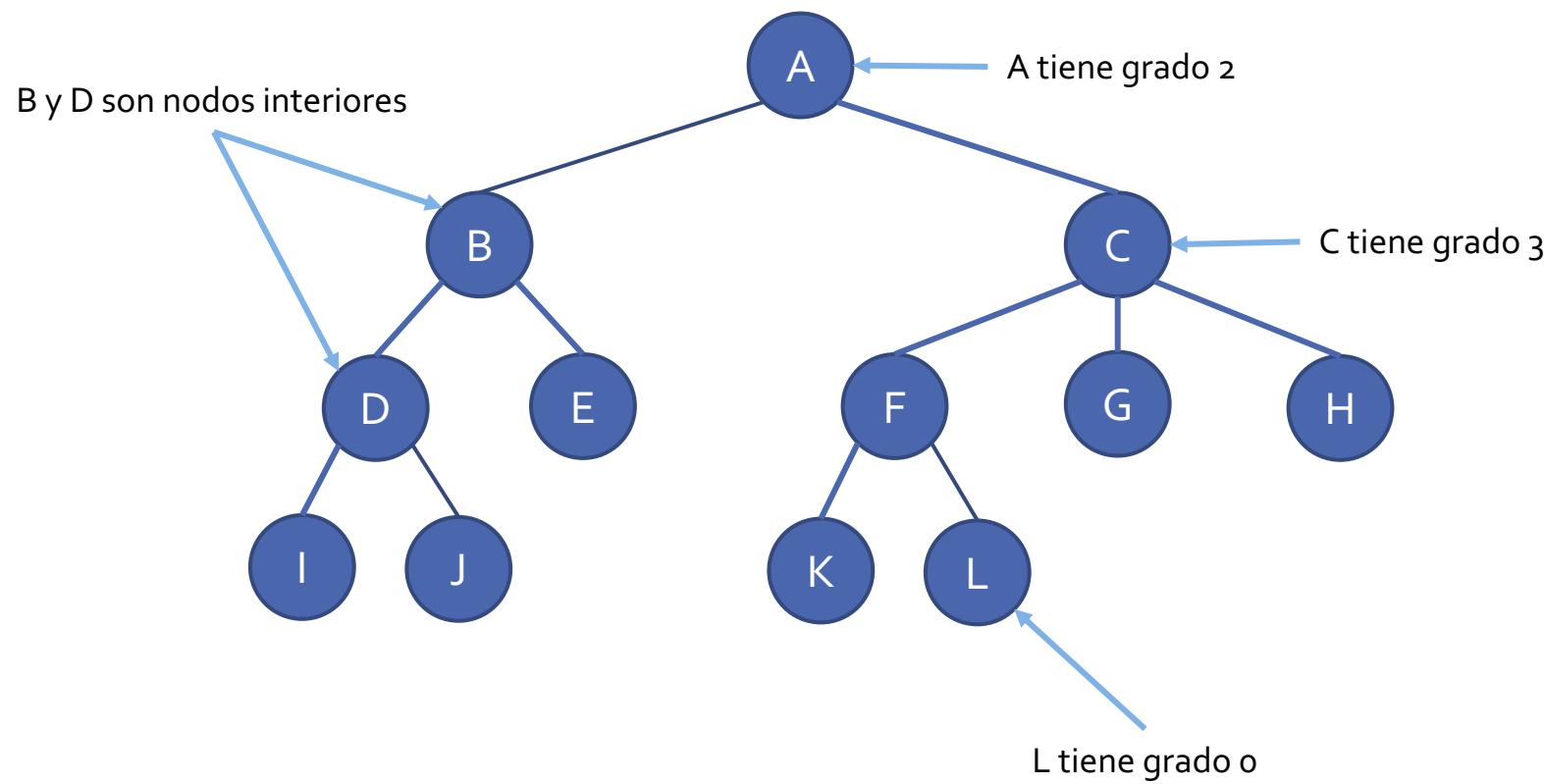
Una estructura árbol se puede representar mediante un grafo de la siguiente manera:



Una estructura árbol se puede representar mediante un grafo de la siguiente manera:

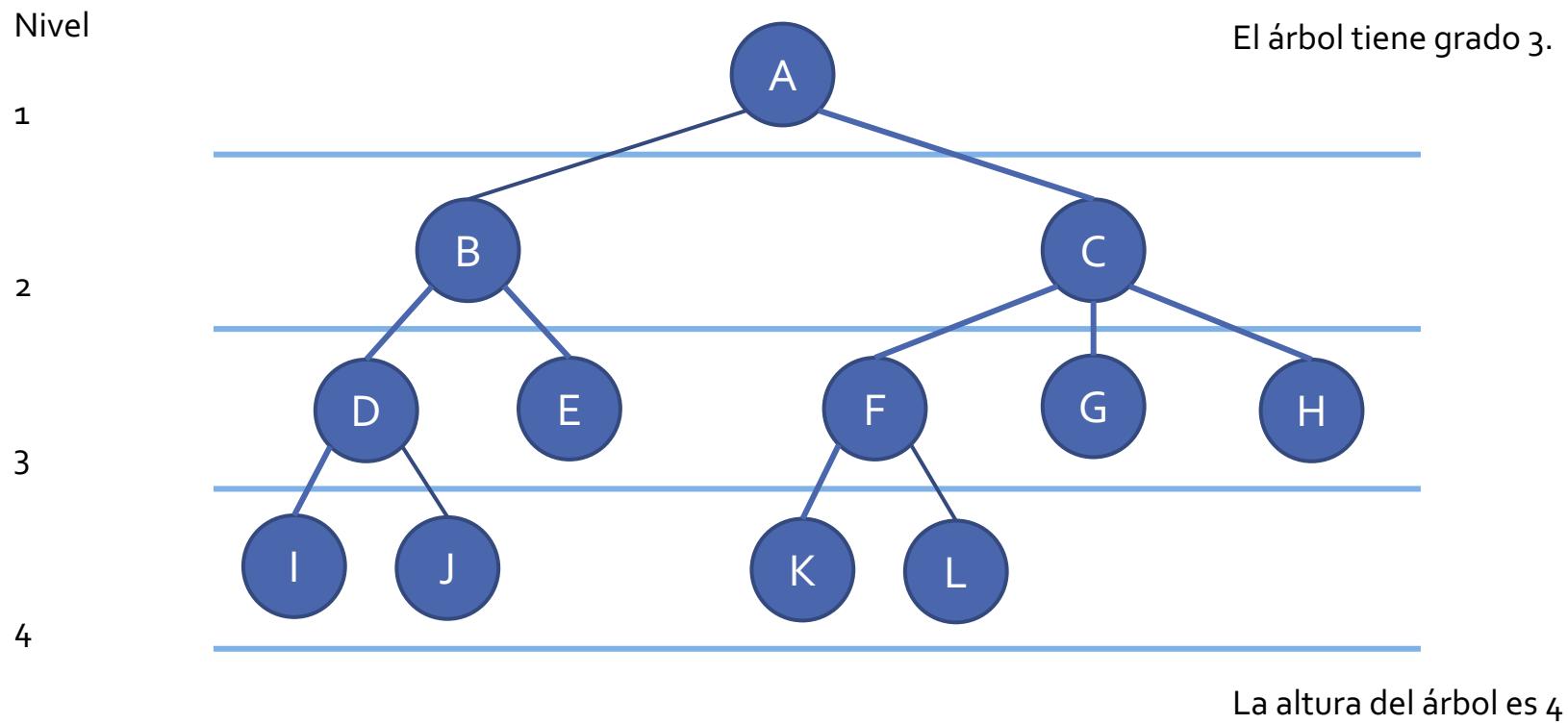


Una estructura árbol se puede representar mediante un grafo de la siguiente manera:



## Ejemplo

Una estructura árbol se puede representar mediante un grafo de la siguiente manera:



Un árbol es una gráfica  $G = \{V, A\}$  en donde el número de nodos es igual al número de aristas más uno:

$$V = A + 1$$

Por tanto, un árbol está formado por un conjunto de uno o más nodos en el que hay un nodo principal (el nodo raíz) y el resto de los nodos son subconjuntos disjuntos  $T_1, T_2, T_3, \dots, T_n$  ( $n > 0$ ), cada uno de los cuales es un árbol y, por ende, cada  $T_i$  es un subárbol de la raíz.

La propiedad de que los subconjuntos  $T_1, T_2, T_3, \dots, T_n$  sean disjuntos, deriva en la eficiencia de este tipo de estructura. De la teoría de conjuntos, un conjunto disjunto a otro cuando no tienen ningún elemento en común, es decir, la intersección de los dos conjuntos es el conjunto vacío. Por ende, dentro de un árbol no existen llaves repetidas.

Además, los árboles siguen una distribución lógica entre sus ramas. Dado un árbol  $T = \{V, A\}$  y un nodo  $n \in V$ , todos los elementos que se encuentran en el subárbol izquierdo de  $n$  son menores a éste y todos los elementos que se encuentran en el subárbol derecho de  $n$  son mayores a éste.

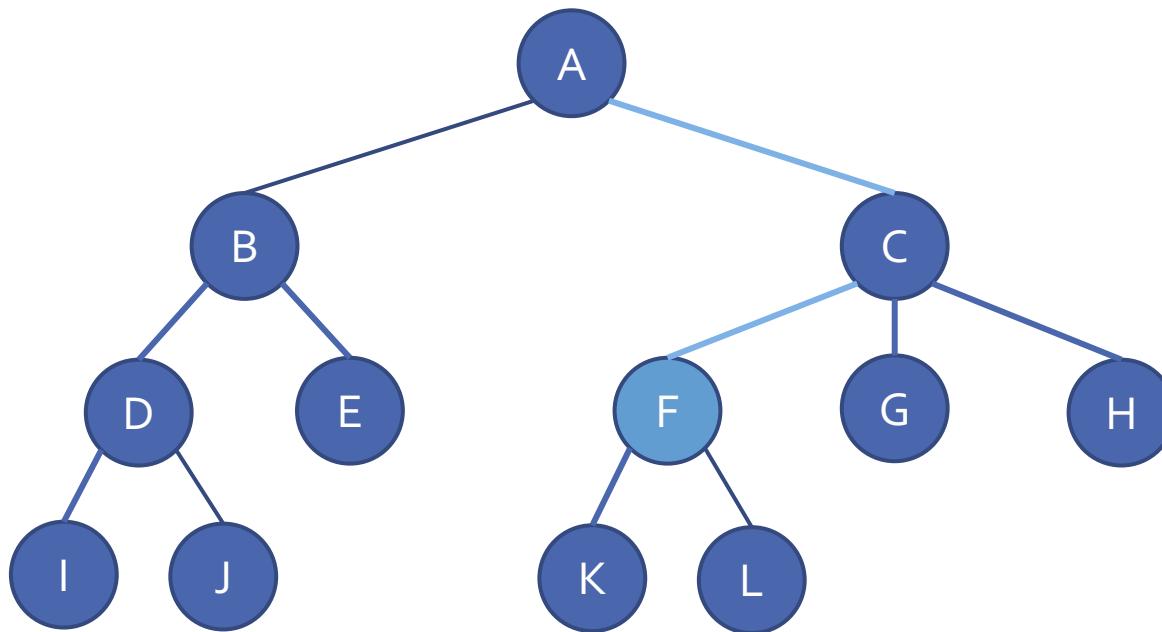
## ¿Cómo saber si un grafo es un árbol?

Para verificar si una gráfica es un árbol se toman en cuenta los siguientes elementos:

- El número de nodos es igual al número de arcos más uno ( $n = a+1$ ).
- Cualquier nodo diferente de raíz es de grado interno 1.
- No existen ciclos.
- Las trayectorias son simples.
- Entre dos nodos cualquiera solo hay una trayectoria.
- Si se retira cualquier arco de la gráfica, se desconecta una parte de ella.

## Longitud de camino

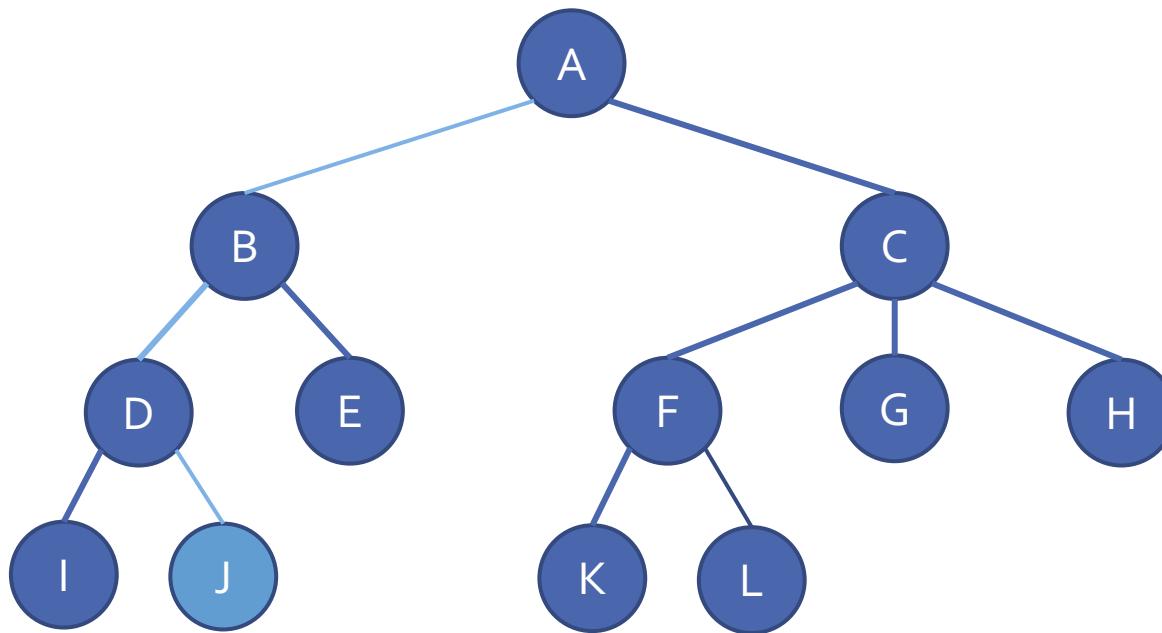
La longitud de camino de un nodo X está definida como el número de arcos que se deben recorrer para llegar desde la raíz hasta el nodo X.



Longitud de camino = 2

## Longitud de camino

La longitud de camino de un nodo X está definida como el número de arcos que se deben recorrer para llegar desde la raíz hasta el nodo X.



Longitud de camino = 3

## Longitud de camino interno

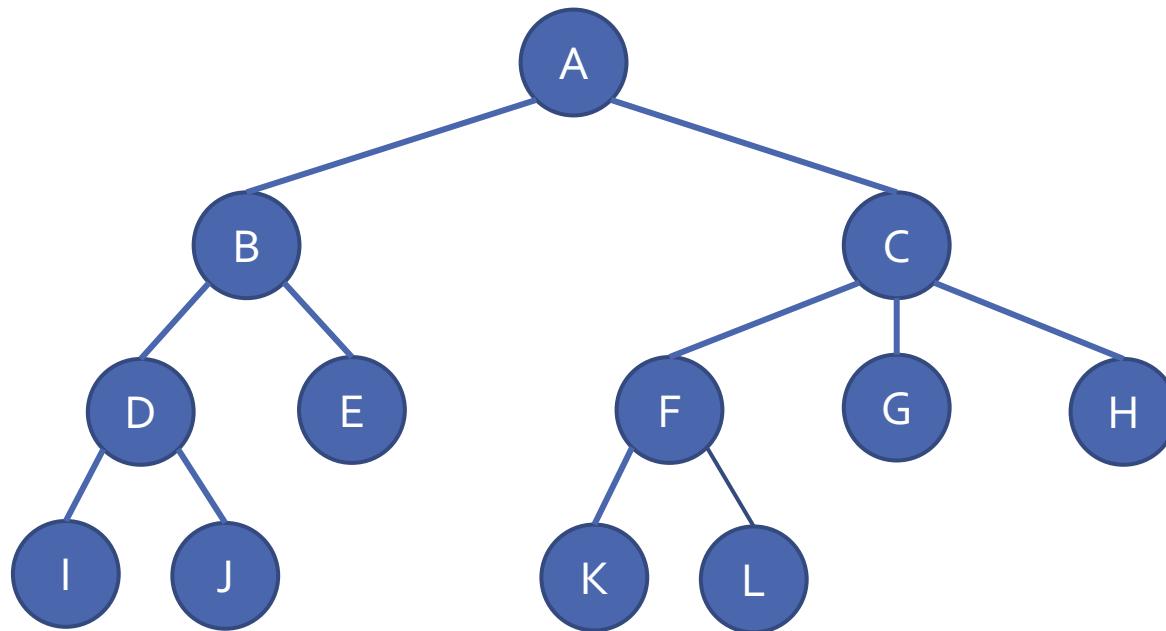
La longitud de camino interno (LCI) está dado por la suma de las longitudes de camino de todos los nodos del árbol. Se calcula de la siguiente manera:

$$LCI = \sum_{i=1}^h n_i * i$$

Donde  $i$  representa el nivel del árbol,  $h$  la altura del árbol y  $n_i$  el número de nodos en el nivel  $i$ .

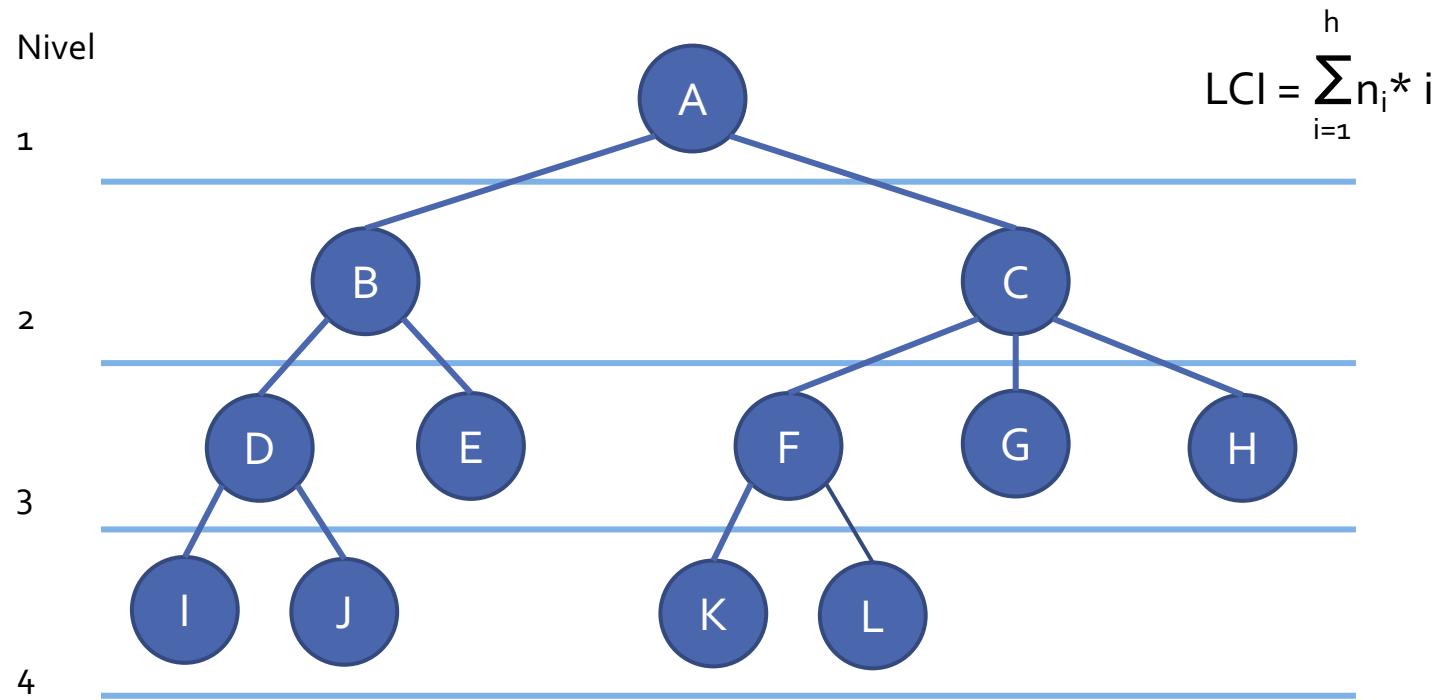
## Ejemplo

Calcular la longitud de camino interno del siguiente árbol:



## Ejemplo

Calcular la longitud de camino interno del siguiente árbol:



$$LCI = 1*1 + 2*2 + 5*3 + 4*4 = 36$$

## Longitud de camino interno promedio

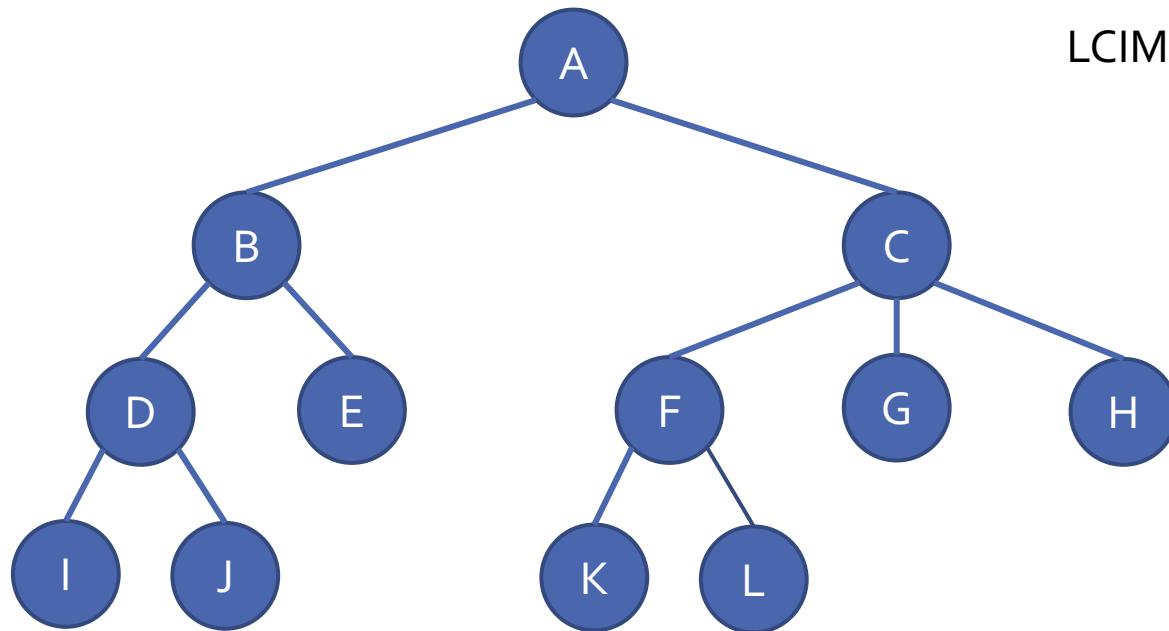
La media de la longitud de camino interno permite conocer el promedio del número máximo de decisiones (arcos) que se deben tomar para llegar a un determinado nodo partiendo de la raíz.

La media está dada por la longitud de camino interno entre el número de nodos del árbol.

$$\text{LCIM} = \text{LCI} / n$$

## Ejemplo

Calcular la longitud de camino interno promedio del siguiente árbol:



$$LCIM = LCI / n$$

$$LCIM = 36 / 12 = 3$$

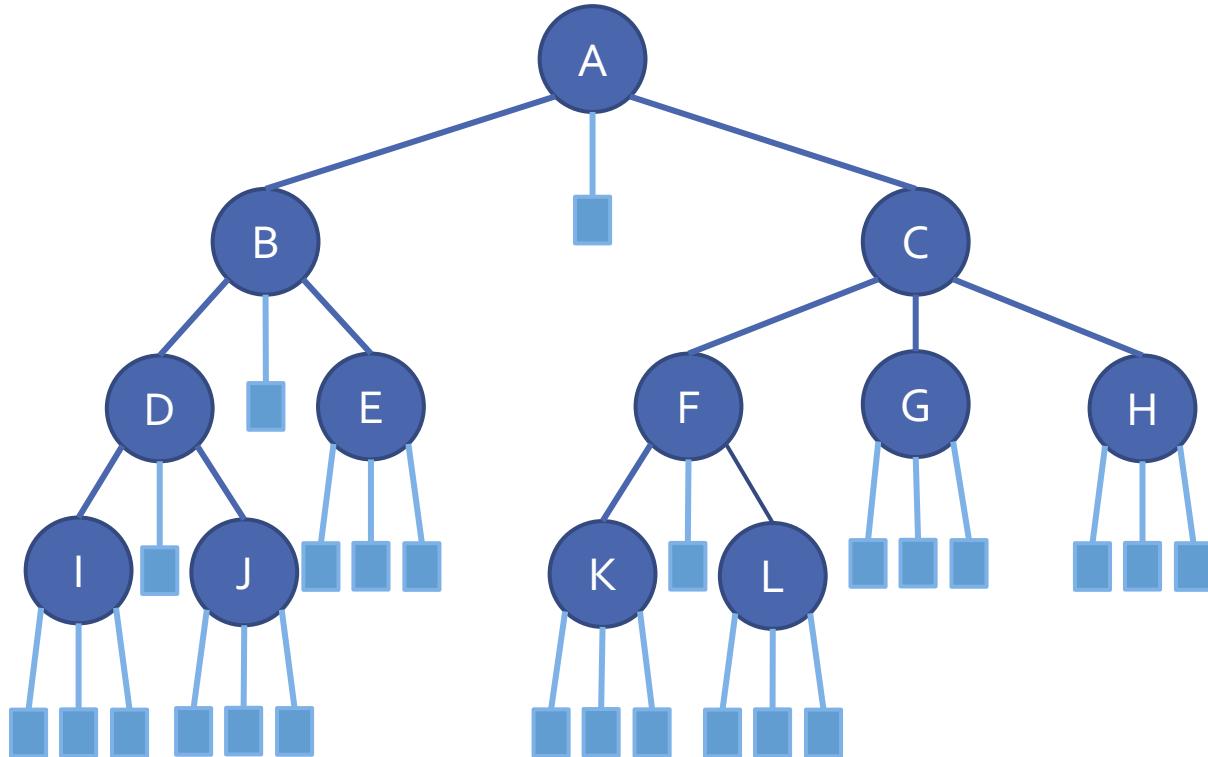
## Árbol extendido

Un árbol extendido es aquel donde el número de hijos de cada nodo es igual al grado del árbol (todos los nodos tienen el mismo número de descendientes). Si el árbol no cumple con la propiedad anterior, se pueden incorporar nodos especiales para hacerla cumplir.

Los nodos especiales completan la rama vacía (o nula) y no pueden tener descendencia. Normalmente, se representan con un cuadrado.

## Ejemplo

Dado el siguiente árbol, generar un árbol extendido:



El número de nodos especiales del árbol para hacerlo extendido es de 25.

## Longitud de camino externo

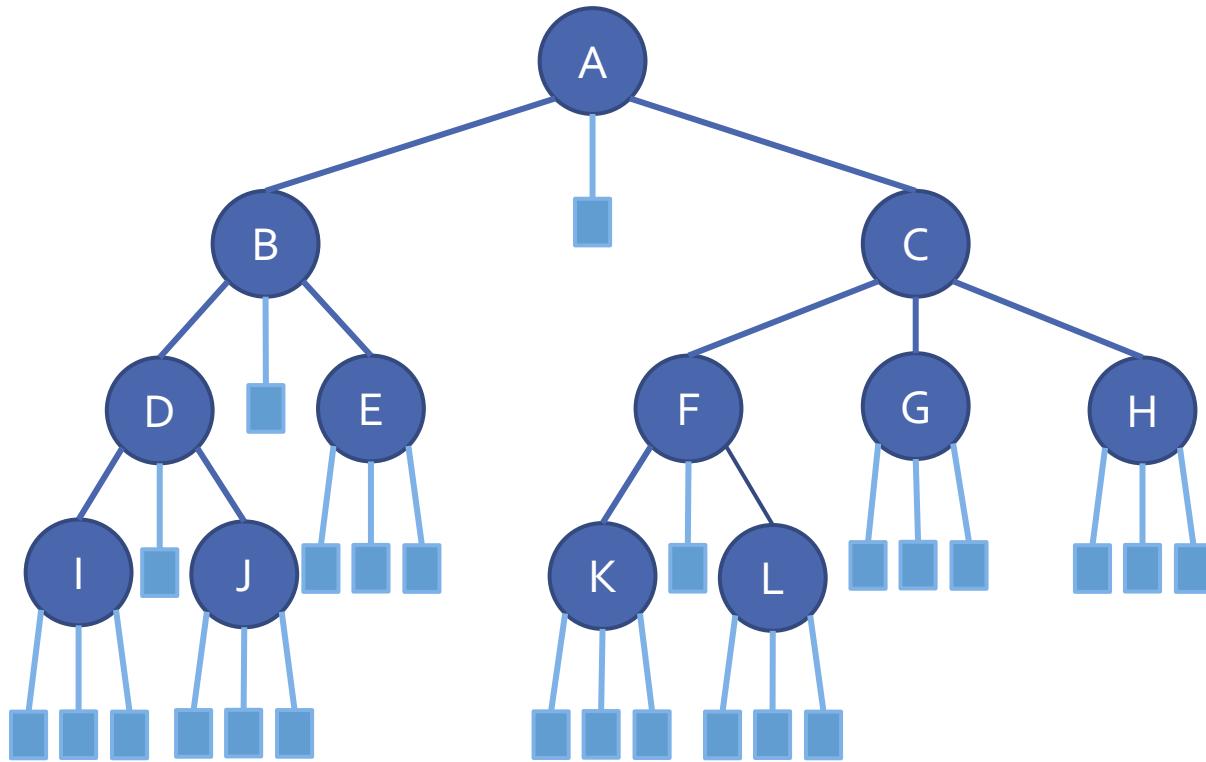
La longitud de camino externo (LCE) de un árbol está dada por las longitudes de camino de todos los nodos especiales del árbol. Se calcula de la siguiente manera:

$$LCE = \sum_{i=2}^{h+1} ne_i * i$$

Donde  $i$  representa el nivel del árbol,  $h$  la altura del árbol y  $ne_i$  el número de nodos en el nivel  $i$ . La ecuación inicia en dos porque a la altura del nodo raíz no puede haber nodos especiales.

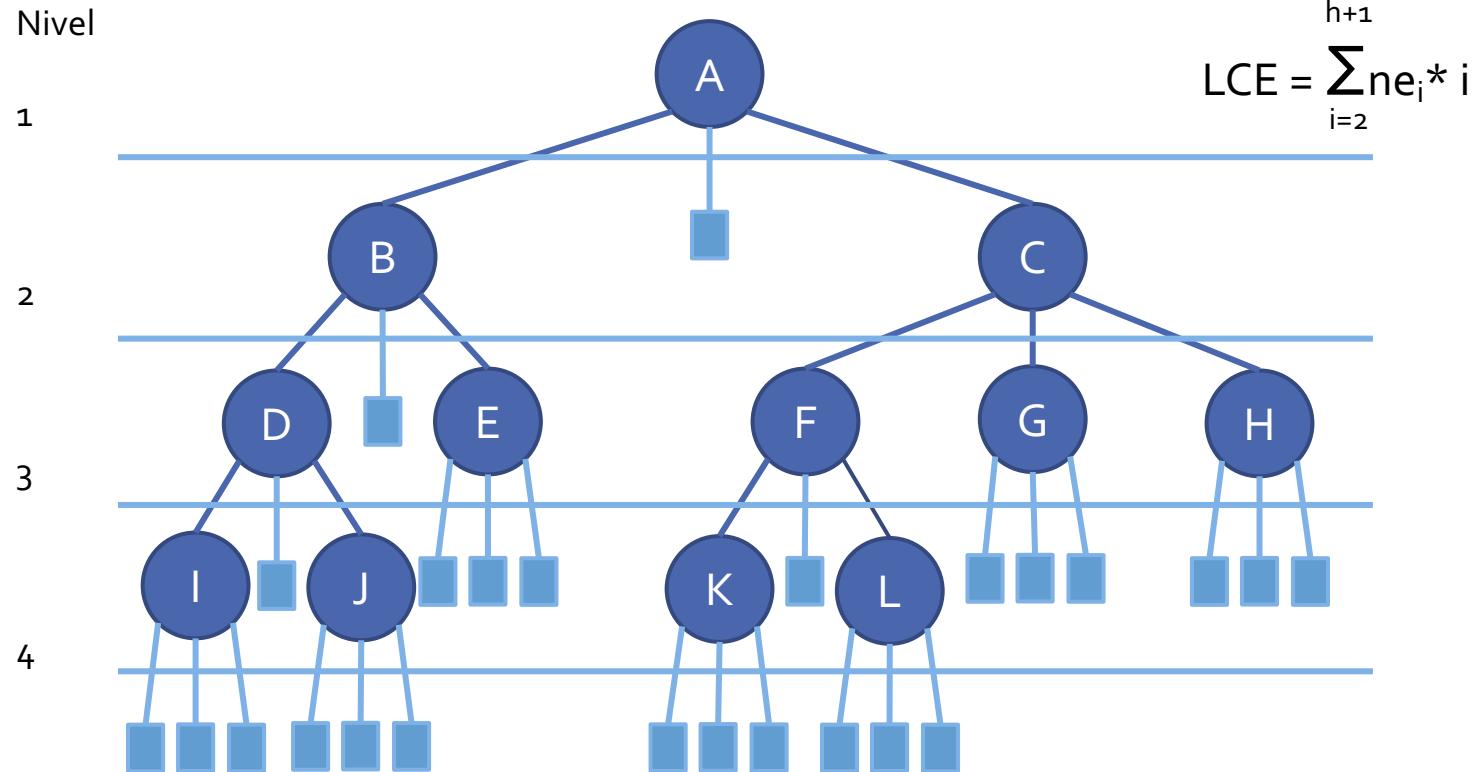
## Ejemplo

Dado el siguiente árbol, calcular la longitud de camino externo del árbol:



## Ejemplo

Dado el siguiente árbol, calcular la longitud de camino externo del árbol:



## Longitud de camino externo promedio

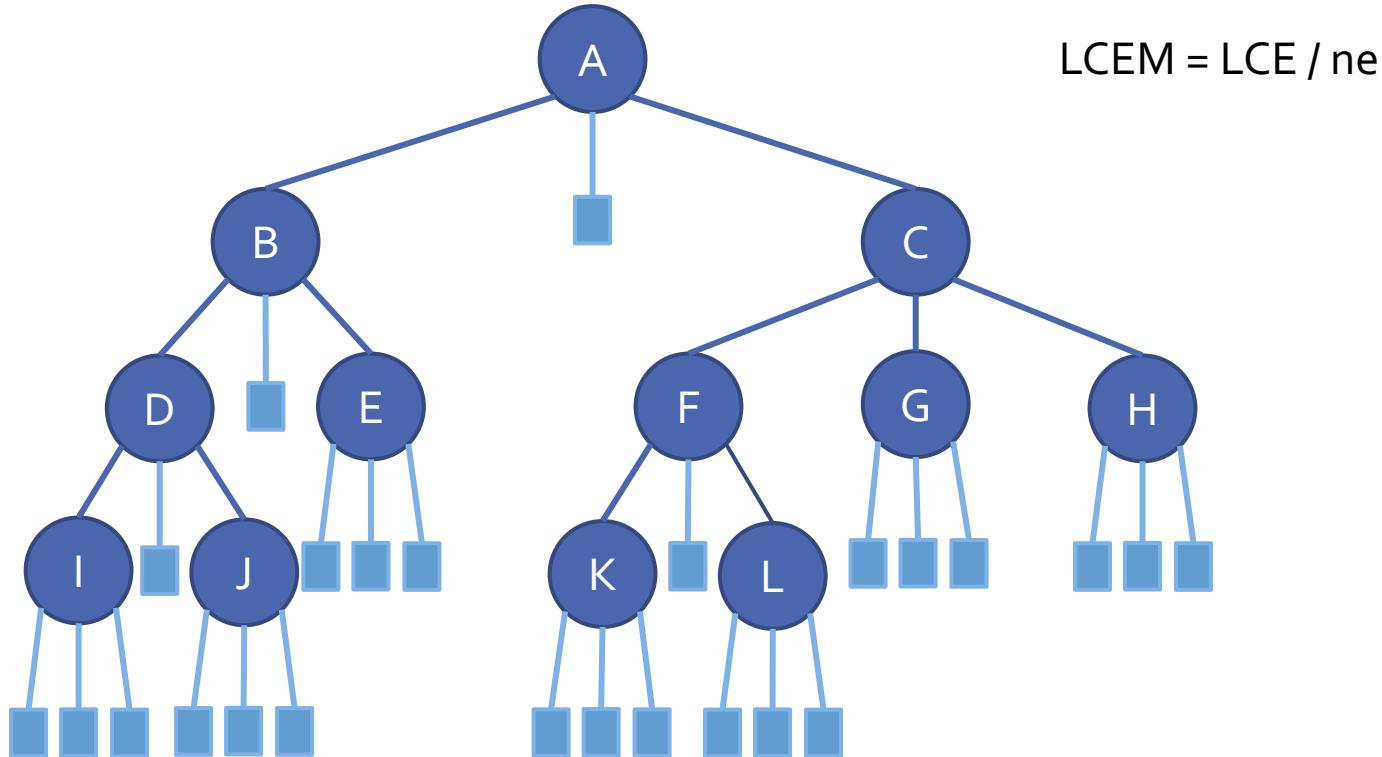
La media de la longitud de camino externo permite conocer el promedio del número máximo de arcos que se deben recorrer para llegar a un determinado especial partiendo de la raíz.

La media está dada por la longitud de camino externo entre el número de nodos especiales del árbol.

$$\text{LCEM} = \text{LCE} / n_e$$

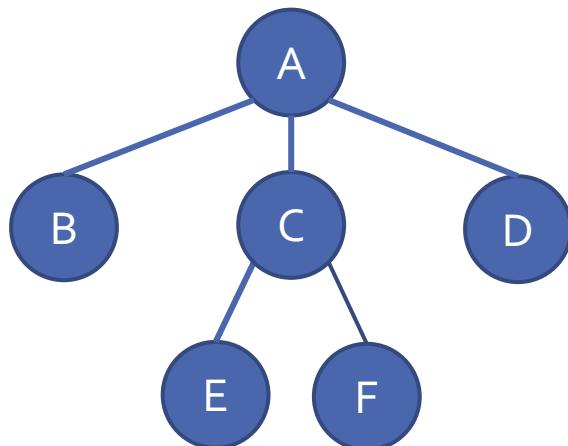
## Ejemplo

Dado el siguiente árbol, calcular la longitud de camino externo promedio del árbol:



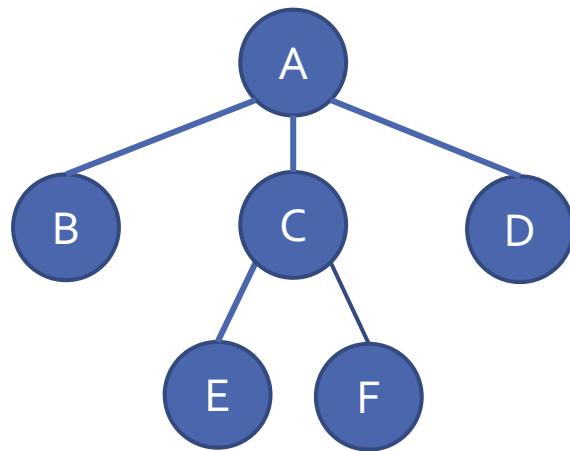
$$\text{LCEM} = 109 / 25 = 4.25$$

Debido a que un árbol es un tipo específico de gráfica, se puede representar a través de una matriz de adyacencia de forma idéntica a como se representa una gráfica dirigida. Para identificar al nodo raíz se pueden utilizar los elementos de la diagonal principal.



	A	B	C	D	E	F
A	*	1	1	1	0	0
B	0	0	0	0	0	0
C	0	0	0	0	1	1
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

Sin embargo, los árboles se pueden representar de manera más eficiente a través de un arreglo bidimensional, donde el primer elemento representa un nodo y el segundo representa a su antecesor.



A	B	C	D	E	F
*	A	A	A	C	C

“Programming is not about typing, it’s about thinking.”

Rich Hickey

(Rich Hickey is the creator of the Clojure language.  
Clojure is a dialect of the Lisp programming language.)

3 4 +

## 4.1 NOTACIONES: INFILA, PREFILA, SUFIJA.

---

## Notación infija

Dada la operación  $A * B$ , se sabe que la variable A será multiplicada por la variable B, debido a que el operando \* está entre ambas variables. A este tipo de notación se le conoce como infija.

Dentro de las operaciones infijas se tiene una jerarquía entre operandos, esto es, en la operación  $A + B * C$  primero se realizará la operación con mayor jerarquía (la multiplicación) y después la de menos jerarquía (la adición).

Sin embargo, si en la operación  $A + B * C$  primero desea realizar la operación de adición, entonces se debe especificar la operación entre paréntesis.

La jerarquía que sigue la notación infija entre sus operandos es:

1. Paréntesis
2. Potencias y raíces
3. Productos y cocientes
4. Adiciones y sustracciones

## Notación prefija

La notación prefija requiere que todos los operadores precedan a los dos operandos sobre los que actúan, es decir, dada la operación infija  $A + B$ , la operación prefija sería  $+ A B$ .

La ventaja de esta notación es que no requiere establecer una jerarquía de operadores, debido a que no existe ambigüedad en cuanto a qué operación realizar primero.

## Notación sufija (o posfija)

La notación sufija requiere que todos los operadores aparezcan después de los operandos sobre los que actúan, es decir, dada la operación infija  $A + B$ , la operación sufija sería  $A B +$ .

La ventaja de esta notación es que no requiere establecer una jerarquía de operadores, debido a que no existe ambigüedad en cuanto a qué operación realizar primero.

## Ejemplo

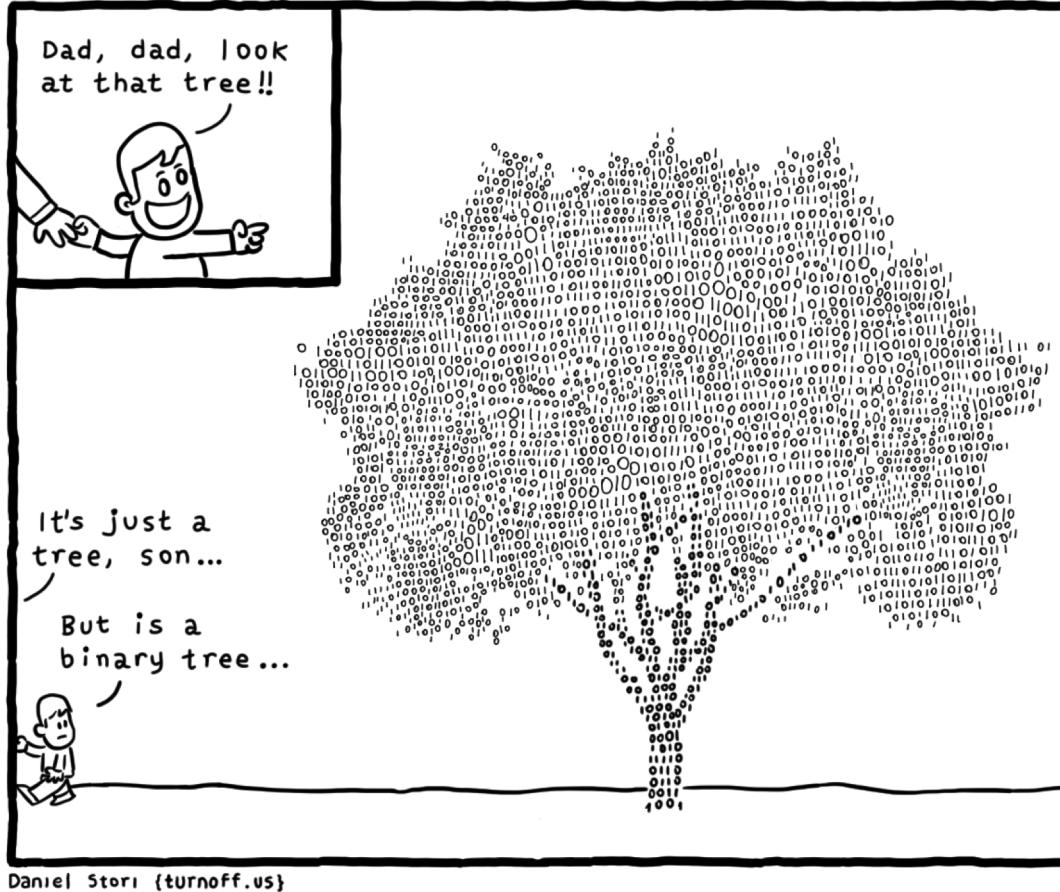
Dadas las siguientes expresiones infijas, obtener su notación prefija y sufija.

Expresión infija	Expresión prefija	Expresión sufija
$A + B * C$		
$(A + B) * C$		
$A + B * C + D$		
$(A + B) * (C + D)$		
$A * B + C * D$		
$A + B + C + D$		

## Ejemplo

Dadas las siguientes expresiones infijas, obtener su notación prefija y sufija.

Expresión infija	Expresión prefija	Expresión sufija
$A + B * C$	$+ A * B C$	$A B C * +$
$(A + B) * C$	$* + A B C$	$A B + C *$
$A + B * C + D$	$+ + A * B C D$	$A B C * + D +$
$(A + B) * (C + D)$	$* + A B + C D$	$A B + C D + *$
$A * B + C * D$	$+ * A B * C D$	$A B * C D * +$
$A + B + C + D$	$+ + + A B C D$	$A B + C + D +$



## 4.2 ÁRBOLES BINARIOS

Un tipo de árbol de gran importancia dentro de las ciencias de la computación es el conocido como árbol binario.

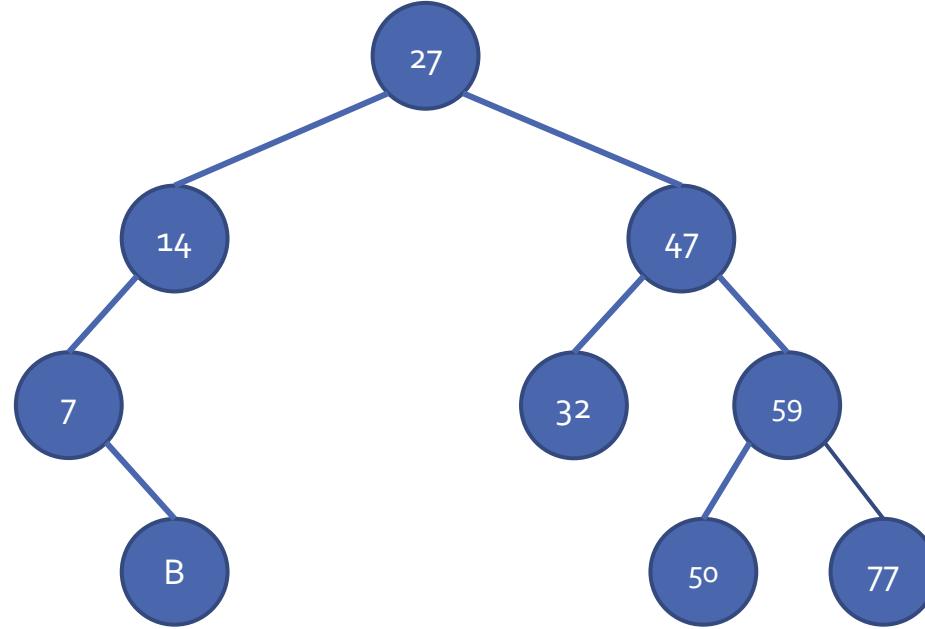
Un árbol ordenado es aquel en el cual la distribución de sus rama sigue una lógica determinada. Los árboles ordenados de grado 2, representan información relacionada con la solución de diversos problemas computacionales. A este tipo de grafos se les conoce como árboles binarios.

### **4.2.1 Definiciones y operaciones.**

En un árbol binario (árbol ordenado de grado 2) cada nodo puede tener como máximo dos subárboles (el izquierdo y el derecho).

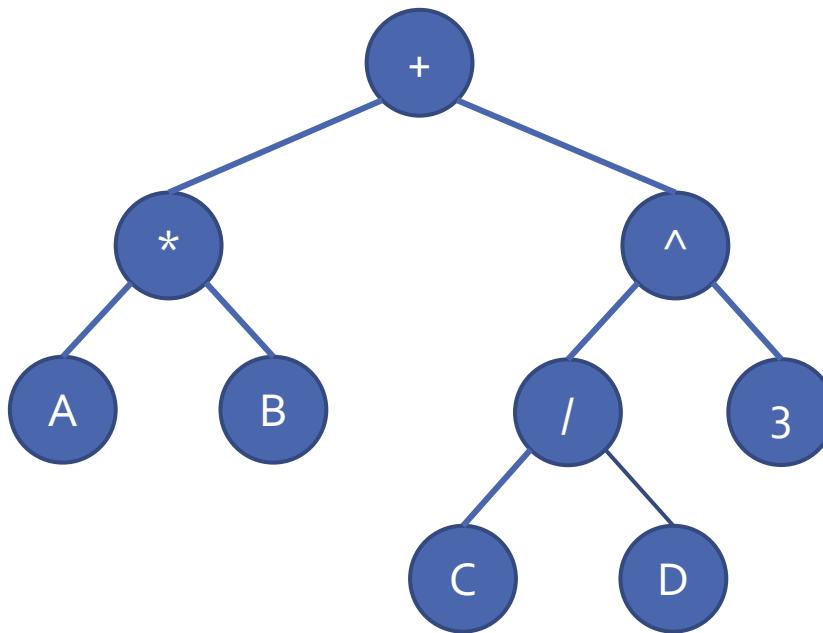
Por tanto, un árbol binario T es una estructura de datos homogénea, resultado de la unión de un elemento tipo (llamado raíz) con dos árboles binarios disjuntos (subárbol izquierdo y subárbol derecho).

## Ejemplo



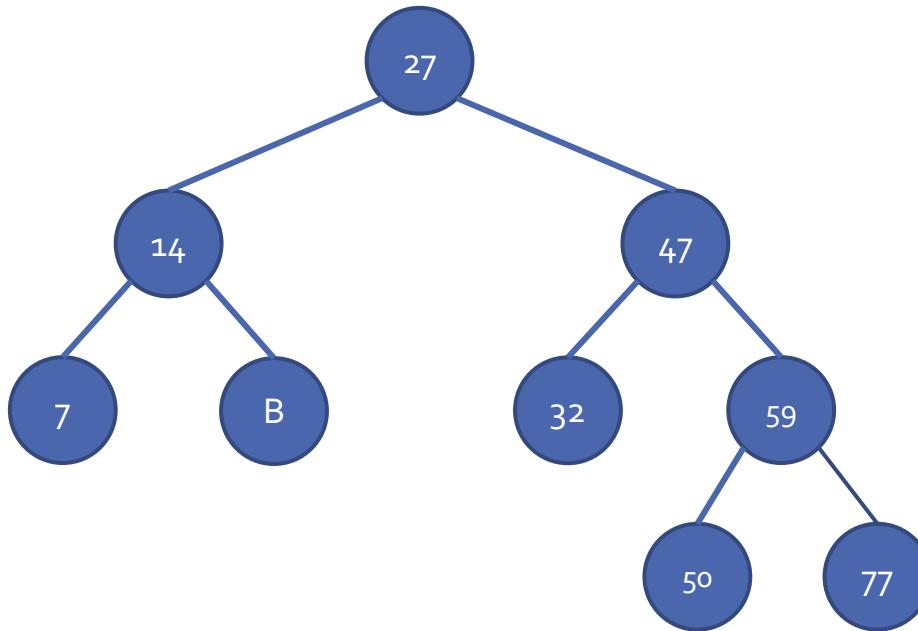
Árbol binario

Las notaciones algebraicas (infija, prefija y sufija) se representan en estructuras tipo árbol binario.



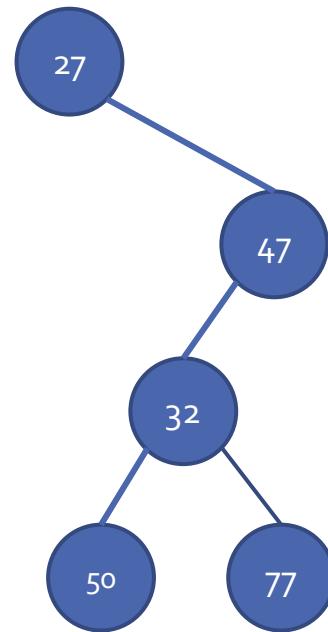
## Ejemplo

Si un árbol ordenado de grado dos que posee cero o dos subárboles, se considera que el árbol es un árbol estrictamente binario.



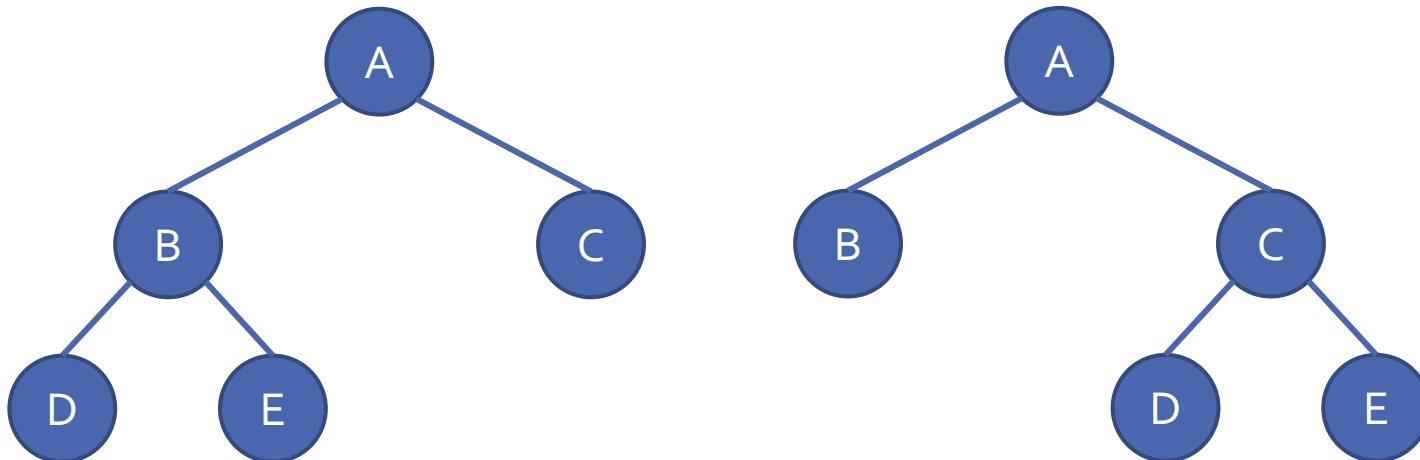
## Ejemplo

Si un árbol ordenado de grado dos posee solo un subárbol, se considera que el árbol es un árbol de Knuth.



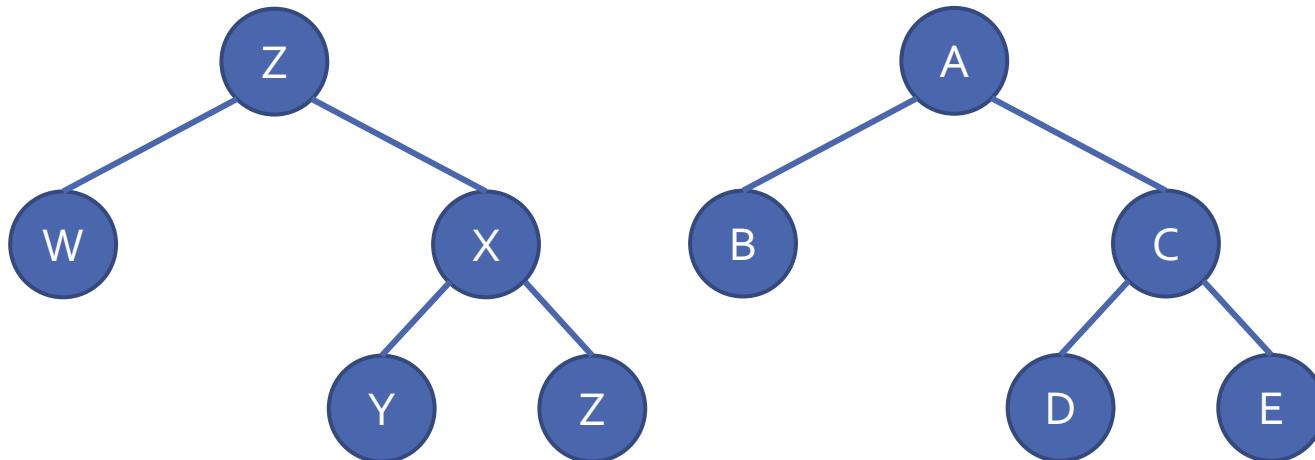
## Árboles binarios distintos

Dos árboles binarios son distintos cuando sus estructuras (nodos o arcos) son diferentes.



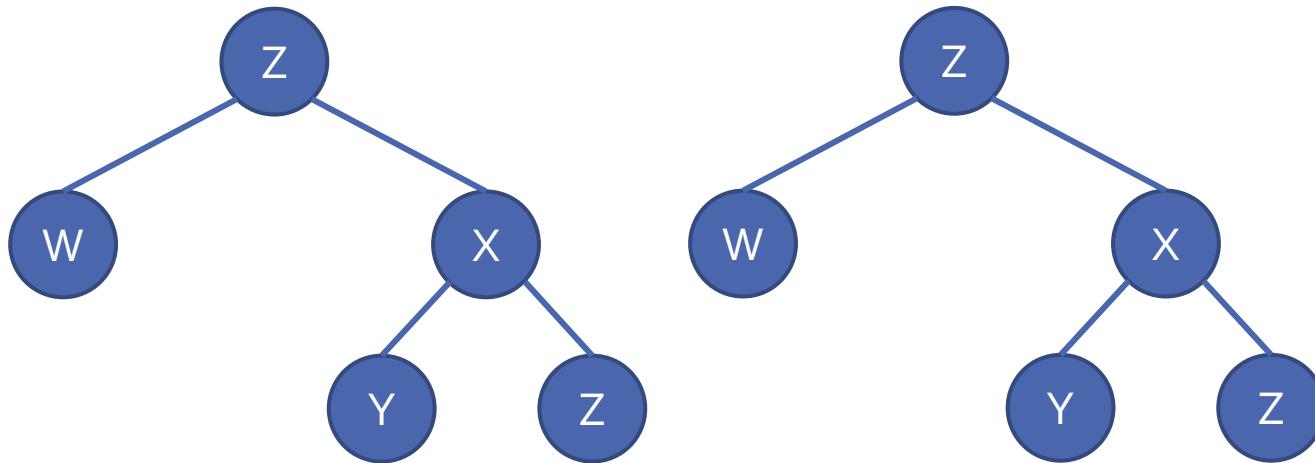
## Árboles binarios similares

Dos árboles binarios son similares cuando sus estructuras (nodos o arcos) son idénticas.



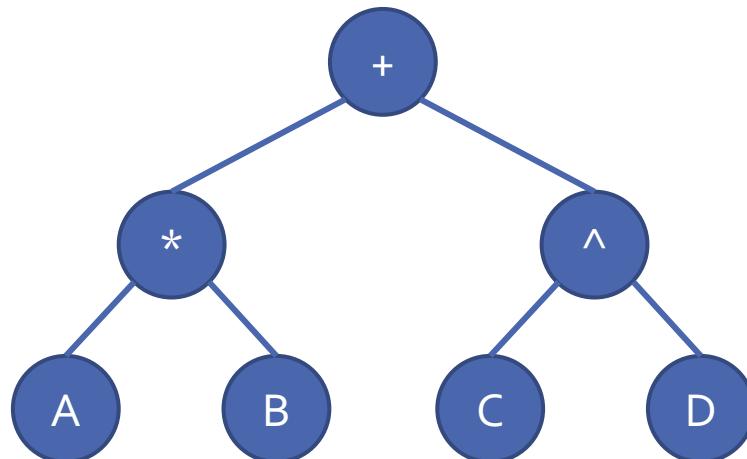
## Árboles binarios equivalentes

Dos árboles binarios son equivalentes, si son similares y además, sus nodos contienen la misma información.



## Árbol binario completo

Un árbol binario completo es aquel en el que todos los nodos (excepto los del último nivel) tienen dos hijos, es decir, el subárbol derecho y el subárbol izquierdo son similares.

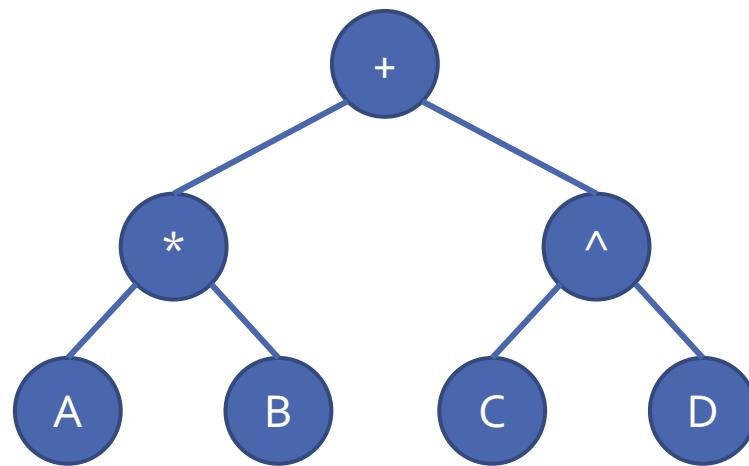


El número de nodos de un árbol binario completo de altura  $h$  se calcula utilizando la siguiente ecuación:

$$\text{num\_nodos} = 2^h - 1$$

## Ejemplo

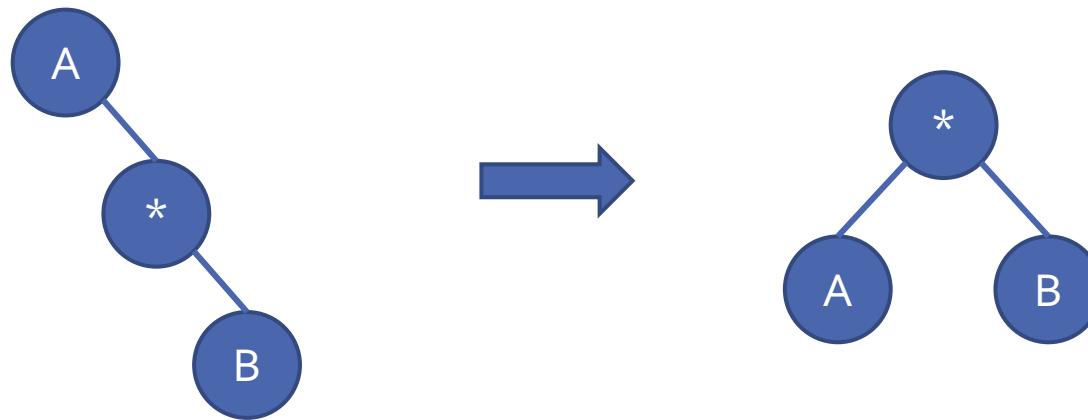
Calcular el número de nodos que puede tener un árbol de altura 3.



$$\text{num\_nodos} = 2^3 - 1 = 7$$

#### 4.2.2 Transformación de árboles a árboles binarios.

Como ya se mencionó, los árboles binarios son muy útiles en los sistemas computacionales. Por tanto, es muy útil convertir un árbol general en un árbol binario.

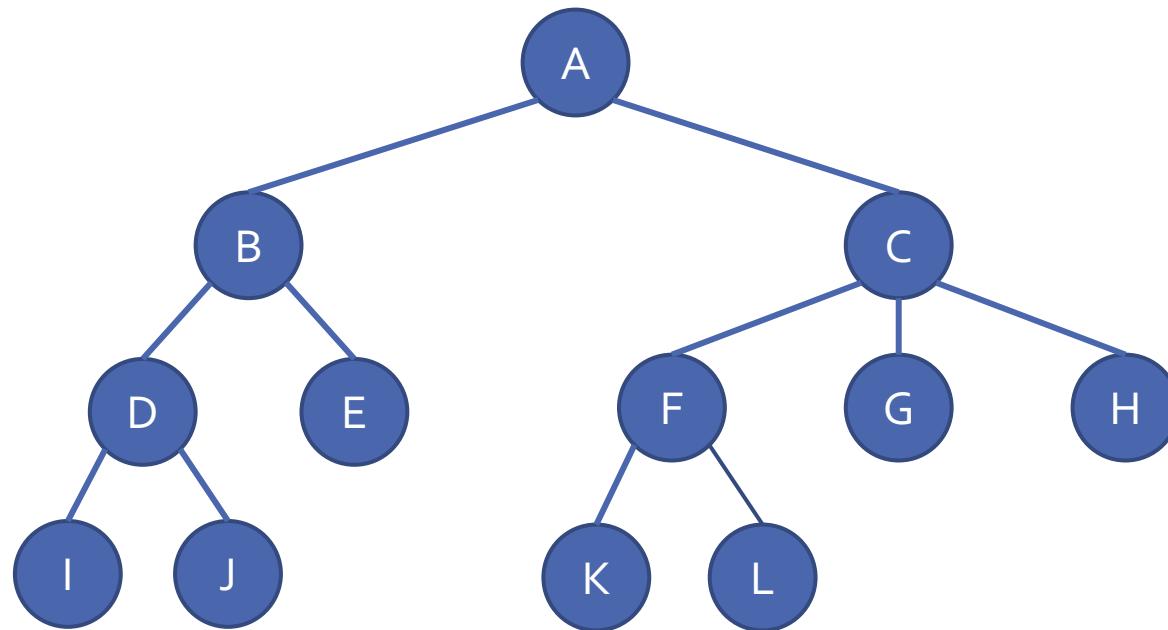


Las operaciones que se deben aplicar para convertir un árbol en un árbol binario son:

- Unir los hijos de cada nodo en forma horizontal (hermanos).
- Relacionar verticalmente el nodo padre con el hijo que se encuentra al extremo izquierdo. Además de eliminar el vínculo de ese padre con el resto de sus hijos.
- Acomodar (girar los nodos) la estructura resultante, obteniendo así el árbol binario correspondiente.

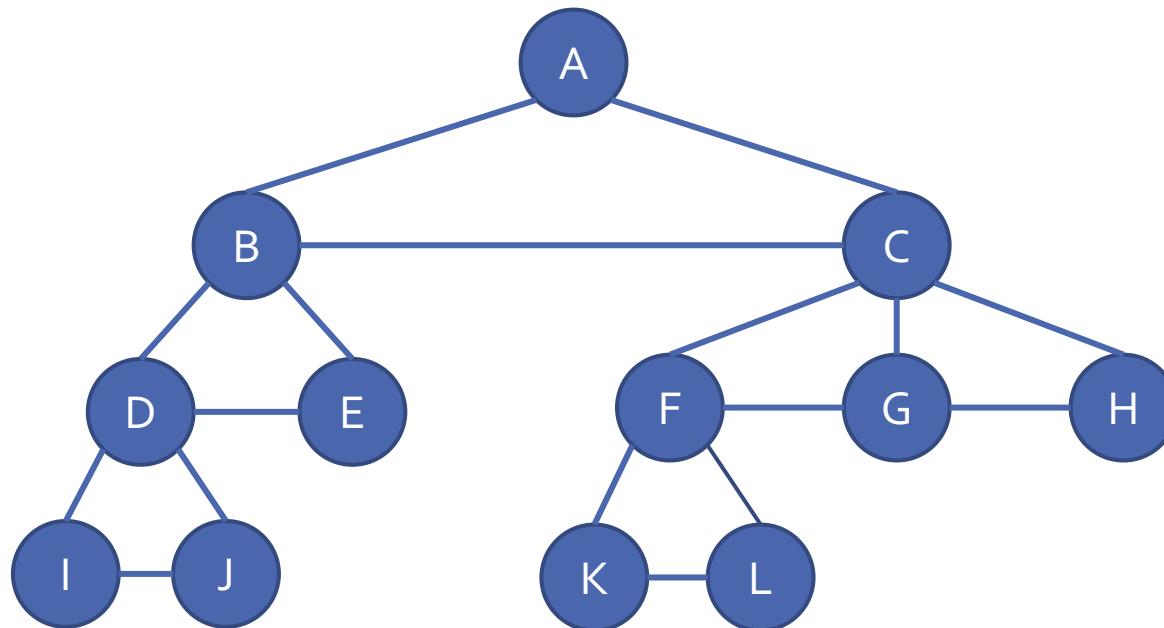
## Ejemplo

Convertir la siguiente estructura tipo árbol en un árbol binario:



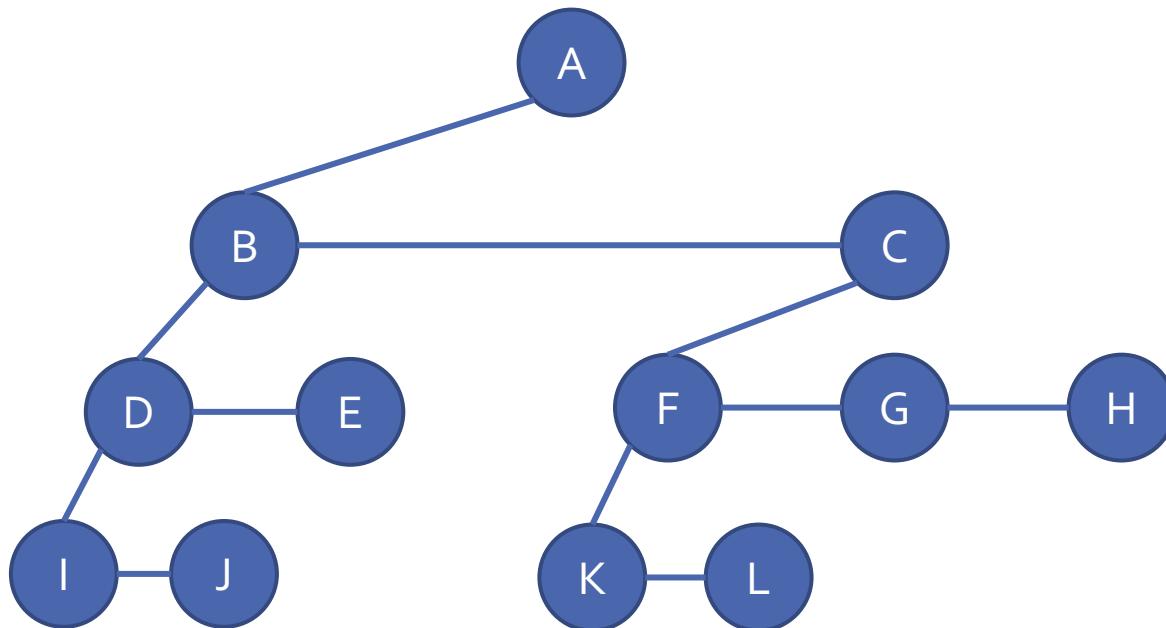
## Ejemplo

El primer paso es: Unir los hijos de cada nodo en forma horizontal (hermanos).



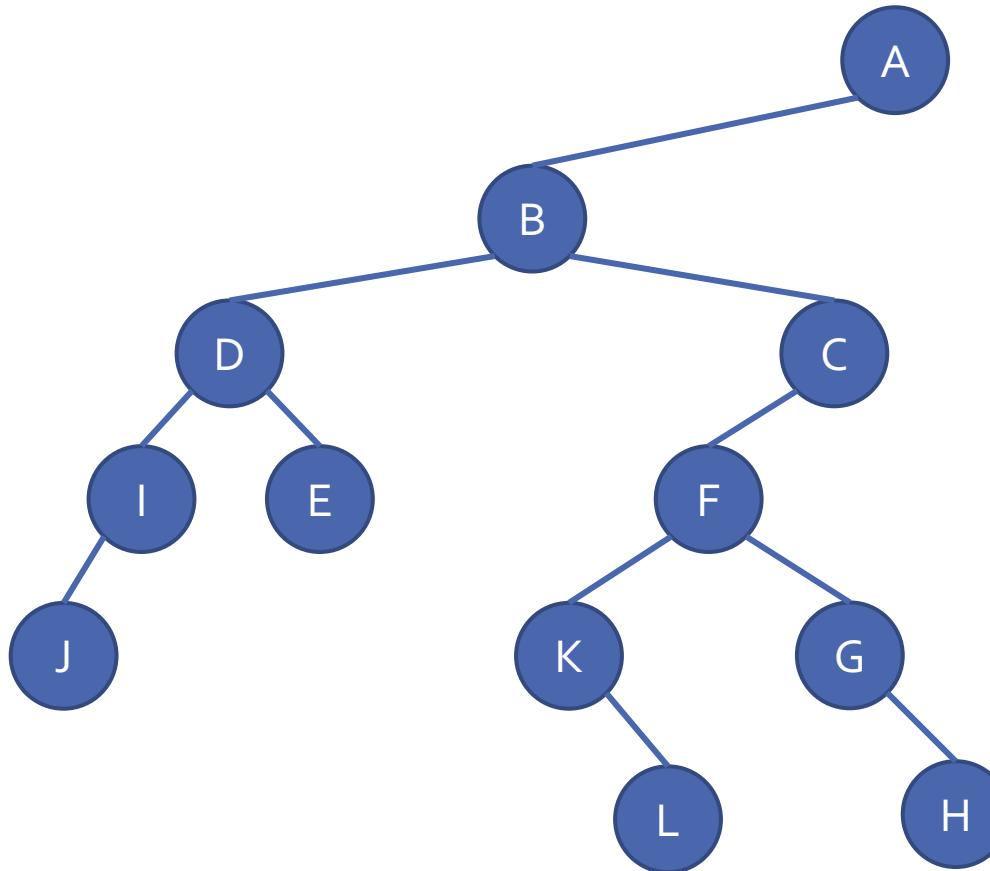
## Ejemplo

El segundo paso es: Relacionar verticalmente el nodo padre con el hijo que se encuentra al extremo izquierdo. Además de eliminar el vínculo de ese padre con el resto de sus hijos.



## Ejemplo

El tercer paso es: Acomodar (girar los nodos) la estructura resultante, obteniendo así el árbol binario correspondiente.



### **4.2.3 Recorrido de árboles.**

Recorrer un árbol implica visitar los nodos del árbol de manera ordenada y secuencial, de tal manera que éstos sean visitados una sola vez. Es posible recorrer un árbol de diversas maneras.

Existen 3 formas de realizar el recorrido de nodos, todas de naturaleza recursiva: infija, prefija y sufija.

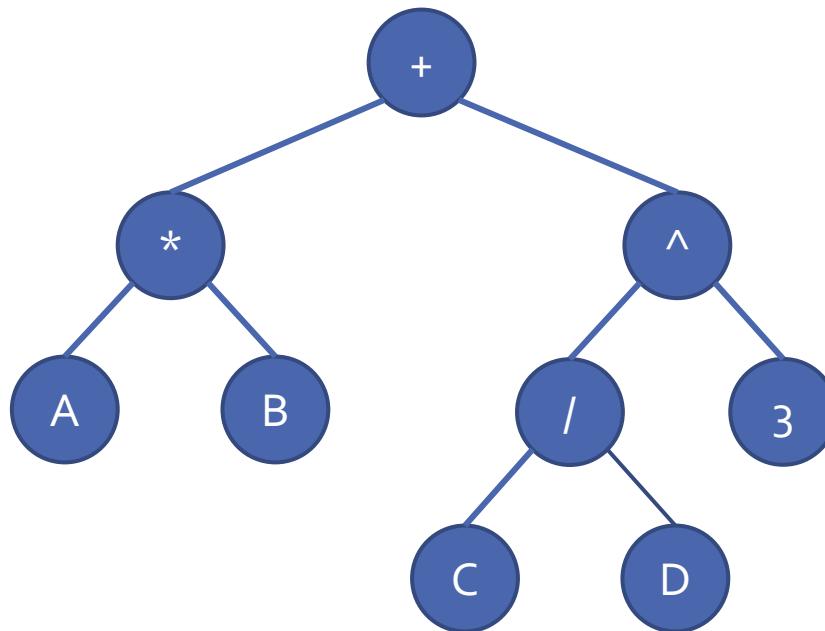
## Recorrido infijo o inorder

Para recorrer un árbol de manera infija se siguen los siguientes pasos:

- Se recorre el sub árbol izquierdo.
- Se visita la raíz del árbol.
- Se recorre el sub árbol derecho.

## Ejemplo

Dado el siguiente árbol A, realizar un recorrido infijo en el mismo.



$$A * B + C / D ^ 3$$

El siguiente algoritmo permite recorrer un árbol de manera infija o inorder. La llamada a la función debe iniciar del nodo raíz (root[t]):

```
INORDER-TREE-WALK(x)
    if x != NIL
        INORDER-TREE-WALK(left[x])
        print key[x]
        INORDER-TREE-WALK(right[x])
```

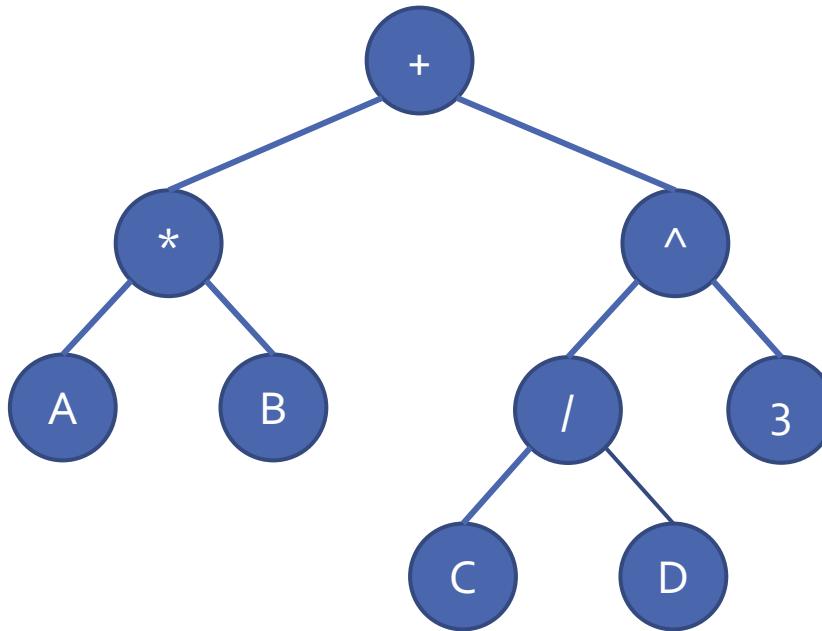
## Recorrido prefijo o preorden

Para recorrer un árbol de manera prefija se siguen los siguientes pasos:

- Visitar la raíz
- Recorrer el sub árbol izquierdo
- Recorrer el sub árbol derecho

## Ejemplo

Dado el siguiente árbol A, realizar un recorrido prefijo en el mismo.



+ \* A B ^ / C D 3

El siguiente algoritmo permite recorrer un árbol de manera prefija o preorden. La llamada a la función debe iniciar del nodo raíz (root[t]):

```
PREORDER-TREE-WALK(x)
    if x != NIL
        print key[x]
        PREORDER-TREE-WALK(left[x])
        PREORDER-TREE-WALK(right[x])
```

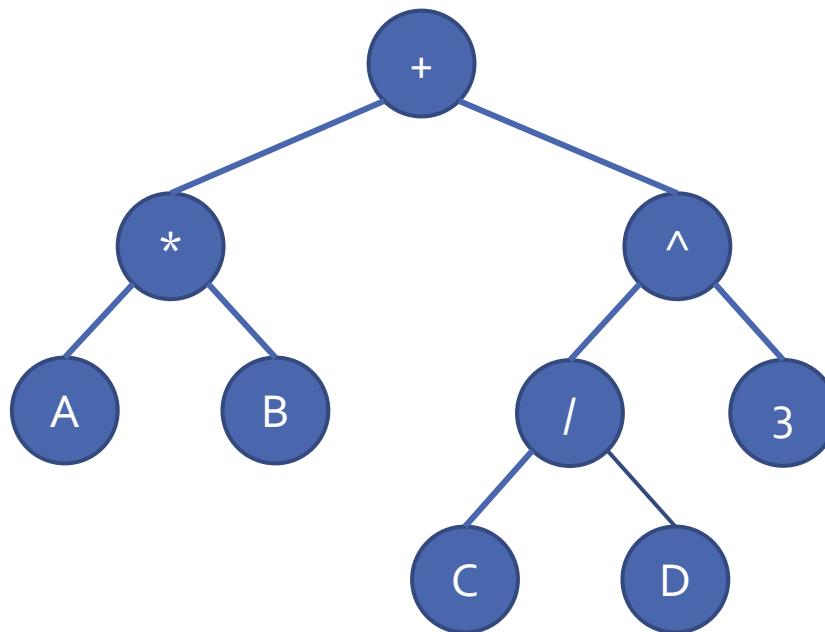
## Recorrido sufijo o posorden

Para recorrer un árbol de manera sufija se siguen los siguientes pasos:

- Recorrer el sub árbol izquierdo
- Recorrer el sub árbol derecho
- Visitar la raíz

## Ejemplo

Dado el siguiente árbol A, realizar un recorrido sufijo en el mismo.



A B \* C D / 3 ^ +

El siguiente algoritmo permite recorrer un árbol de manera sufija o posorden. La llamada a la función debe iniciar del nodo raíz (root[t]):

```
POSORDER-TREE-WALK(x)
    if x != NIL
        POSORDER-TREE-WALK(left[x])
        POSORDER-TREE-WALK(right[x])
        print key[x]
```

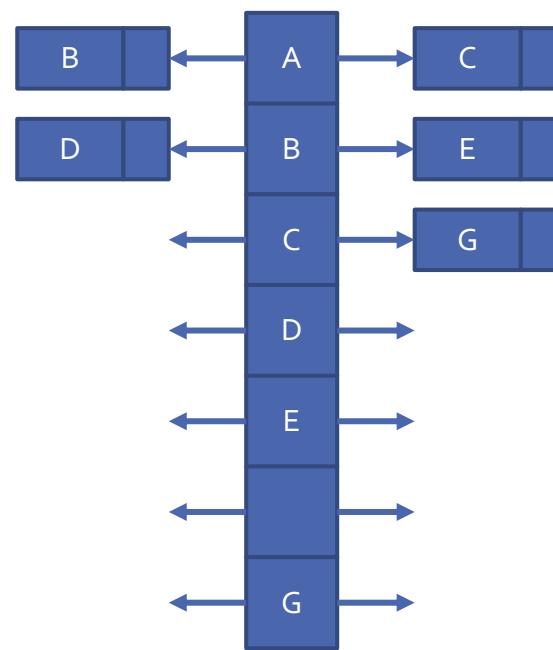
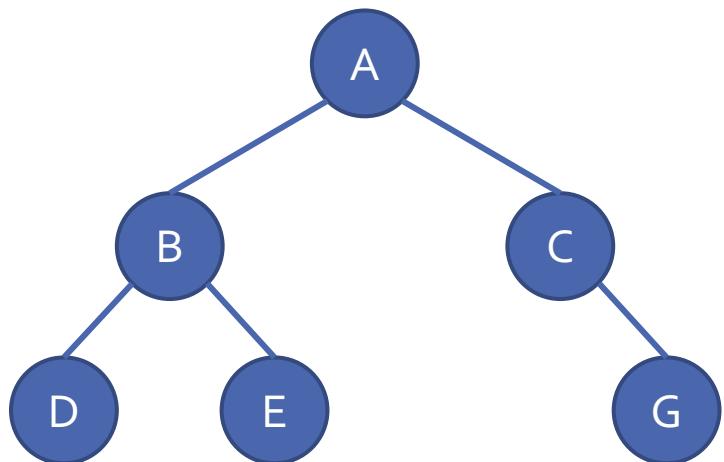
#### **4.2.4 Representación en la computadora.**

De manera similar a un grafo, un árbol binario se puede representar a través de una lista de adyacencia.

Para implementar un árbol binario en una lista de adyacencia, cada nodo debe poseer tres elementos: un campo para guardar la información del nodo, una referencia para almacenar el hijo izquierdo del nodo y otra referencia para almacenar el hijo derecho del nodo.

## Ejemplo

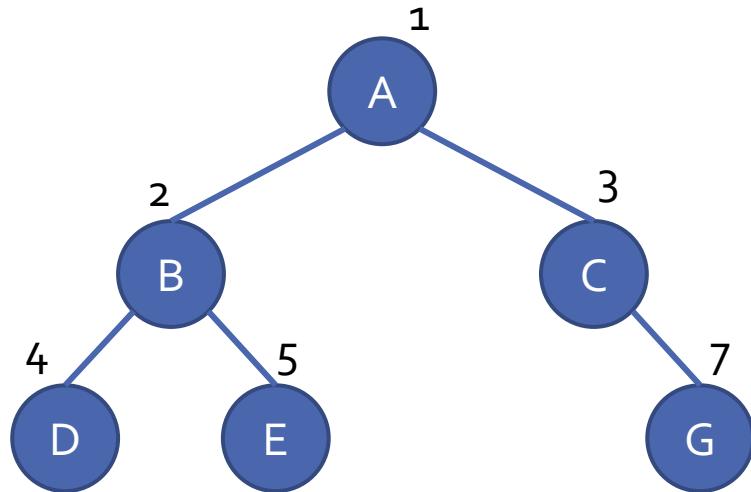
Dado el siguiente árbol binario T, realizar su representación con listas ligadas.



Así mismo, de manera similar a un árbol, un árbol binario se puede representar utilizando un arreglo, donde los hijos de cualquier nodo  $k$  se encuentran en el elemento  $2*k$  (nodo izquierdo) y  $2*k+1$  (nodo derecho). La longitud del arreglo es de  $2^n - 1$ , donde  $n$  es el número de niveles del árbol.

## Ejemplo

Dado el siguiente árbol binario T, realizar su representación en un arreglo.



1	2	3	4	5	6	7
A	B	C	D	E		G

“What is programming?... Some people call it a science, some people call it an art, some people call it a skill or trade.”

Charles Simonyi

(Is a Hungarian-born American computer businessman.)

# ÁRBOLES. PARTE I.

---

Práctica 8

# 8. Árboles. Parte I.

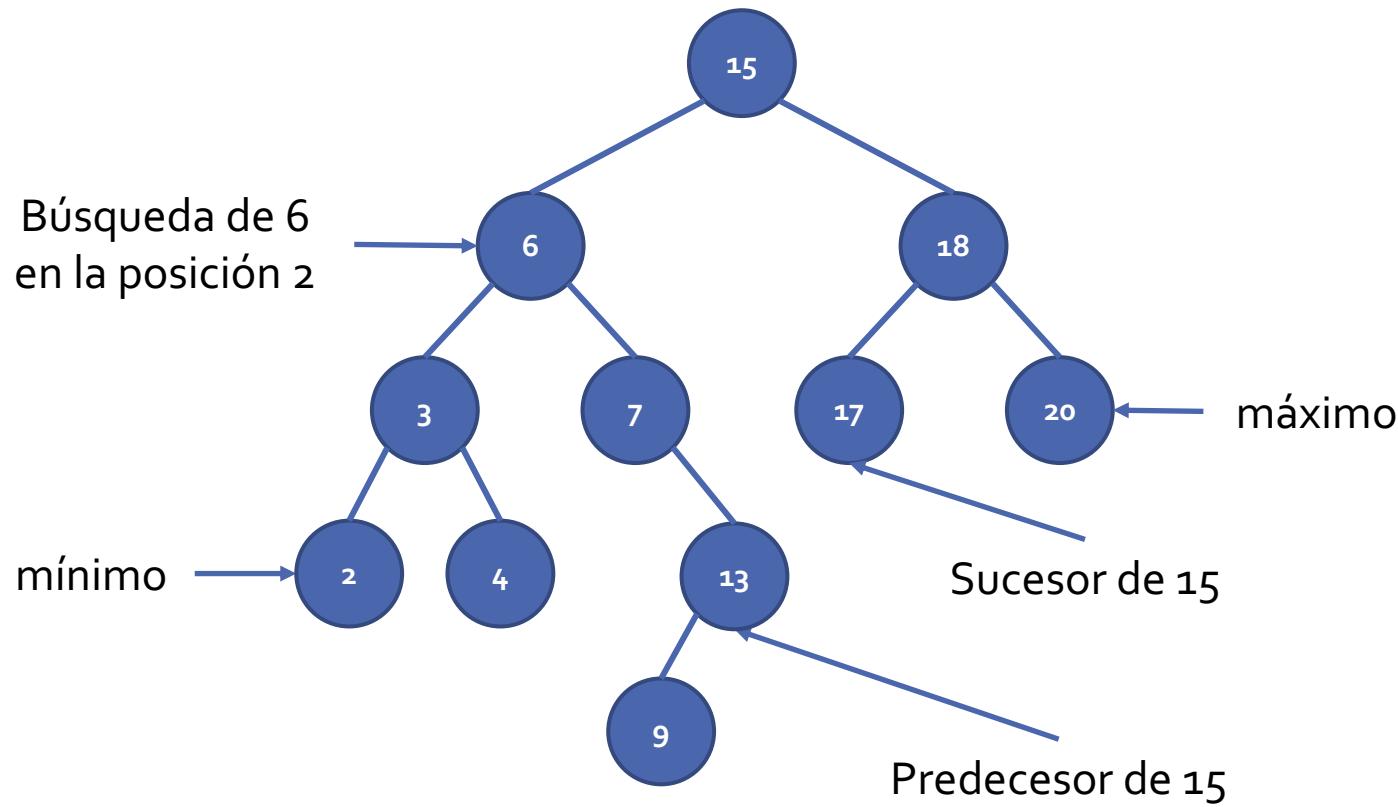
- Implementar en lenguaje Python los algoritmos que permitan realizar el recorrido de un árbol A en orden infijo, prefijo y sufijo, utilizando ecuaciones algebraicas, para obtener notaciones infija, prefija y sufija.
- Graficar la complejidad que tiene los algoritmos de recorridos infijo, prefijo y sufijo.

Una operación común dentro de cualquier estructura de datos es la BÚSQUEDA. Además de ésta, otras operaciones que soporta un árbol de búsqueda binaria es MINIMUM, MAXIMUM, SUCCESSOR y PREDECESSOR.

Lo interesante con las operaciones mencionadas es que todas se ejecutan en un tiempo  $O(h)$ , dentro de un árbol de búsqueda binario de altura  $h$ .

## Ejemplo

Operaciones dentro de un árbol.



Algoritmo de búsqueda en un árbol binario.

```
TREE-SEARCH(x, k)
    if x = NIL
        return -1
    if k = key[x]
        return x
    if k < key[x]
        return TREE-SEARCH(left[x], k)
    else
        return TREE-SEARCH(right[x], k)
```

El siguiente algoritmo encuentra el valor mínimo en un árbol binario.

```
TREE-MINIMUM(x)
    while left[x] != NIL
        x ← left[x]
    return x
```

¿Y el máximo?

Para encontrar el sustituto de un nodo se debe hacer una búsqueda del elemento que se encuentra más a la izquierda del subárbol derecho.

Por otro lado, si el subárbol derecho del nodo es vacío, se debe buscar el elemento que se encuentra más a la derecha del árbol de manera ascendente, es decir, rumbo a la raíz.

El siguiente algoritmo encuentra el sucesor de un valor en un árbol binario.

```
TREE-SUCCESSOR(x)
    if right[x] != NIL
        return TREE-MINIMUM(right[x])
    y ← parent[x]
    while y != NIL and x = right[y]
        x ← y
        y ← p[y]
    return y
```

¿Y el predecesor?

## Algoritmo de inserción.

La inserción en un árbol binario inicia desde la raíz y traza un camino hacia las hojas para localizar la posición donde se debe almacenar el nuevo valor, siguiendo la regla de los árboles binarios.

La operación de inserción en un árbol binario toma un tiempo de  $O(h)$  en un árbol de altura  $h$ .

```

TREE-INSERT(T, z)
    y ← NIL
    x ← root[ T ]
    while x != NIL
        y ← x
        if key[z] < key[x]
            x ← left[x]
        else
            x ← right[x]
    p[z]← y
    if y = NIL
        root[ T ]← z
    else
        if key[z] < key[y]
            left[y]← z
        else
            right[y] ← z

```

## Algoritmo de eliminación.

La operación de eliminación considera cuando el nodo a eliminar no tiene hijos, cuando tiene un hijo y cuando tiene ambos hijos.

La operación de eliminación en un árbol binario toma un tiempo de  $O(h)$  en un árbol de altura  $h$ .

```

TREE-DELETE(T, z)
    if left[z] = NIL or right[z] = NIL
        y ← z
    else y ← TREE-SUCCESSOR(z)
    if left[y] = NIL
        x ← left[y]
    else x ← right[y]
    if x = NIL
        p[x] ← p[y]
    if p[y] = NIL
        root[T] ← x
    else if y = left[p[y]]
        left[p[y]] ← x
    else right[p[y]] ← x
    if y = z
        key[z]← key[y]
        copy y's satellite data into z
    return y

```

# CÓDIGO HUFFMAN

---

Aplicación

Se refieren a una técnica de compresión ampliamente usada y muy efectiva. Puede llegar a comprimir entre 20 y 90 % dependiendo de las características de los datos a comprimir.

El algoritmo ávido de Huffman utiliza una tabla de frecuencia de ocurrencia de caracteres para crear una manera óptima de representar cada carácter como una cadena binaria.

Dado un archivo de datos de 100,000 caracteres (que en la memoria ocuparía 800,000 bits), se desea almacenar de manera compacta. Se puede observar que la frecuencia de caracteres en el archivo es la siguiente:

Caracteres	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5

De la tabla anterior se puede observar que solo aparecen 6 caracteres diferentes y que el carácter más repetitivo es 'a' (45,000 veces).

Para representar el archivo de información se considera el diseño de un código binario, donde cada carácter es representado por una cadena binaria única.

Caracteres	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5
Código de palabra de longitud fija.	000	001	010	011	100	101

Utilizando el código de longitud fija se observa que se necesitan 3 bits para representar los seis caracteres: a=000, b=001, c=010, d=011, e=100, f=101.

Dado que el archivo consta de 100,000 caracteres, este método requiere de 300,000 bits para codificar el archivo completo.

También se puede considerar un código de longitud variable con la siguiente distribución:

Caracteres	a	b	c	d	e	f
Frecuencia (en miles)	45	13	12	16	9	5
Código de palabra de longitud variable.	0	101	100	111	1101	1100

Un código de longitud variable hace un mejor trabajo que uno de longitud fija, debido a que a los caracteres frecuentes les asigna un código de palabra corto y a los caracteres que no tienen tanta frecuencia les asigna un código de palabra largo.

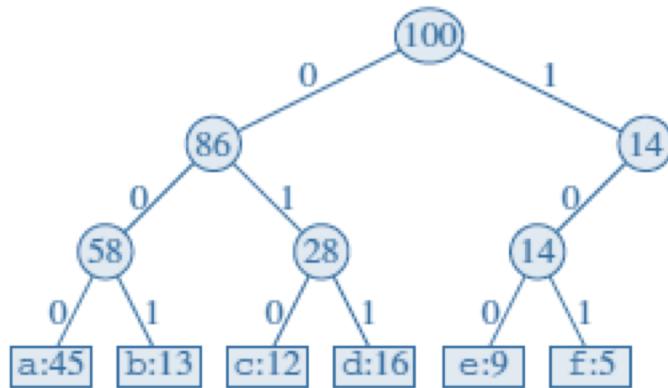
El código de longitud variable requiere  $(45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4) * 1000 = 224,000$  bits para codificar el archivo, genera un ahorro de un poco más del 25%.

La codificación variable se considera una codificación prefija, ya que ningún código de palabra es prefijo de algún otro código. La codificación prefija simplifica el proceso de decodificación.

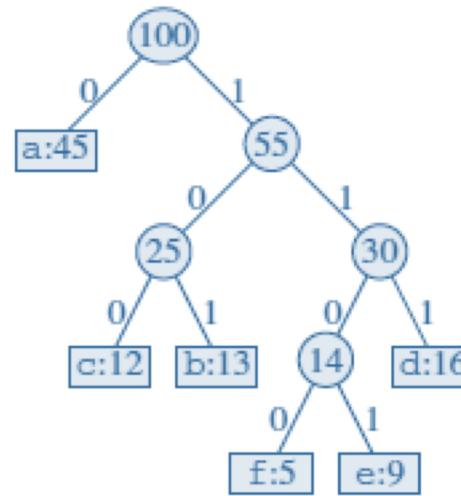
Por ejemplo, la cadena 001011101 se puede descodificar como 0.0.101.1101 que resulta en aabe.

La decodificación se puede representar mediante una estructura de árbol. Una estructura de árbol posee un nodo raíz (el primer nodo del árbol) y varios nodos hojas (los últimos nodos).

A partir de un árbol se puede interpretar el código de palabra de un carácter como la ruta que existe desde el nodo raíz hasta la hoja que contiene el carácter, donde 0 significa ir a la izquierda y 1 significa ir a la derecha.



Longitud fija



Longitud variable

Dado un árbol A que corresponde a un código prefijo, se puede calcular fácilmente el número de bits requeridos para codificar un archivo.

Para cada carácter c en el alfabeto C, se tiene  $f(c)$  (frecuencia de aparición del carácter c en el archivo) y  $p_A(c)$  (profundidad de la hoja que posee el carácter c dentro del árbol A).

El número de bits que se requieren para condificar el archivo está dado por:

$$B(A) = \sum f(c) p_A(c)$$

con  $c \in C$  que se define como el costo del árbol A.

Huffman inventó un algoritmo que construye un código prefijo óptimo. El siguiente pseudocódigo del algoritmo de Huffman asume que  $C$  es un conjunto de  $n$  caracteres y que cada carácter  $c \in C$  es un objeto con una frecuencia  $f(c)$ . El algoritmo crea un árbol  $A$  que corresponde a un código óptimo de arriba hacia abajo.

```

1 FUNC HUFFMAN(C)
2     n ← |C|                      ** longitud de C
3     Q ← C                         ** elementos de C
4     i ← 1                          ** índice
5     MIENTRAS i < n-1 HACER
6         crearNodo(z)
7         left[z]← x ← extraerMen(Q)
8         right[z]← y ← extraerMen(Q)
9         f [z]← f [x] + f [y]
10        INSERT(Q, z)
11    FIN_MIENTRAS
12    DEV EXTRACT-MIN(Q)
13 FIN_FUNC

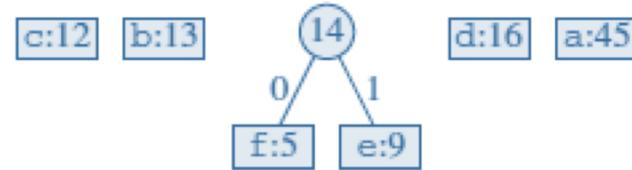
```

**1**

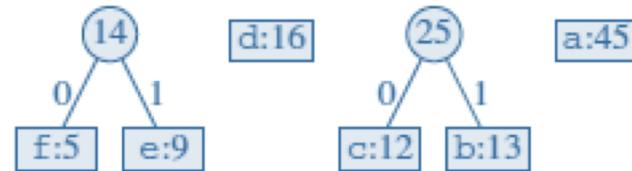
f:5 e:9 c:12 b:13 d:16 a:45

---

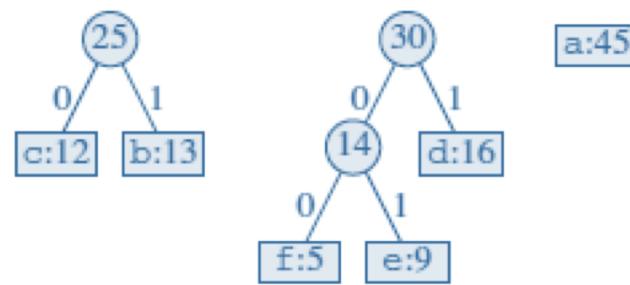
**2**



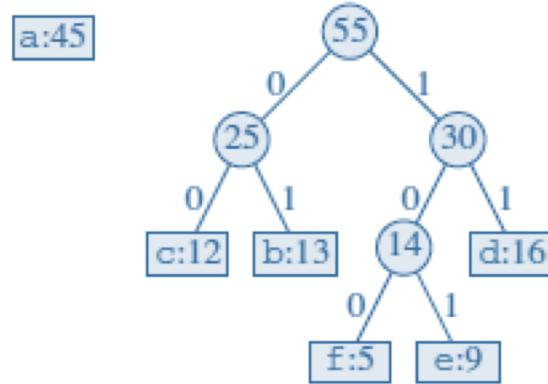
**3**



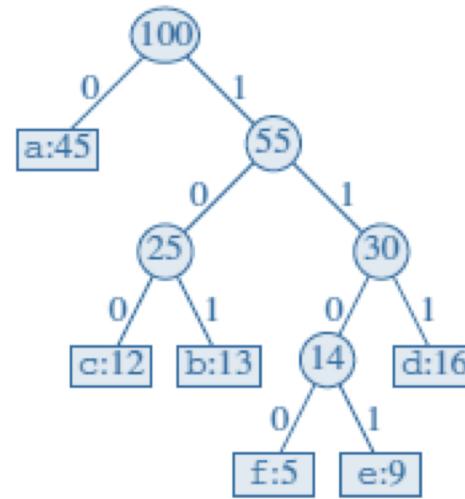
4



5



6



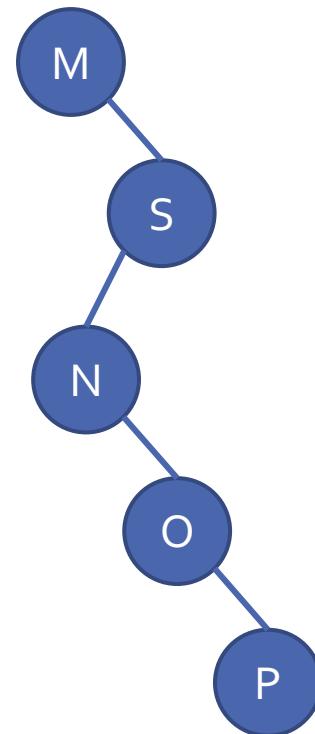
El algoritmo de Huffman es un algoritmo ávido, el cual va buscando la decisión que parece mejor en el momento, esperando con ello optimizar la solución general.

# ÁRBOL BALANCEADO

---

Árboles

Si en un árbol binario se inserta un conjunto de claves ordenadas (de forma ascendente o descendente), se produce un árbol que crece en una sola dirección.



Para conservar la eficiencia en la operación de búsqueda se diseñaron los árboles balanceados. Los árboles balanceados realizan un reacomodo o balanceo de los nodos después de insertar o eliminar un elemento.

Formalmente, un árbol balanceado se define como un árbol binario de búsqueda en el que se debe cumplir que para todo nodo T del árbol, la altura de los subárboles izquierdo y derecho no deben diferir en más de un elemento.

$$|HRI - HRD| < 1$$

Donde HRI es la altura del subárbol (rama) izquierdo y HRD es la altura del subárbol (rama) derecho.

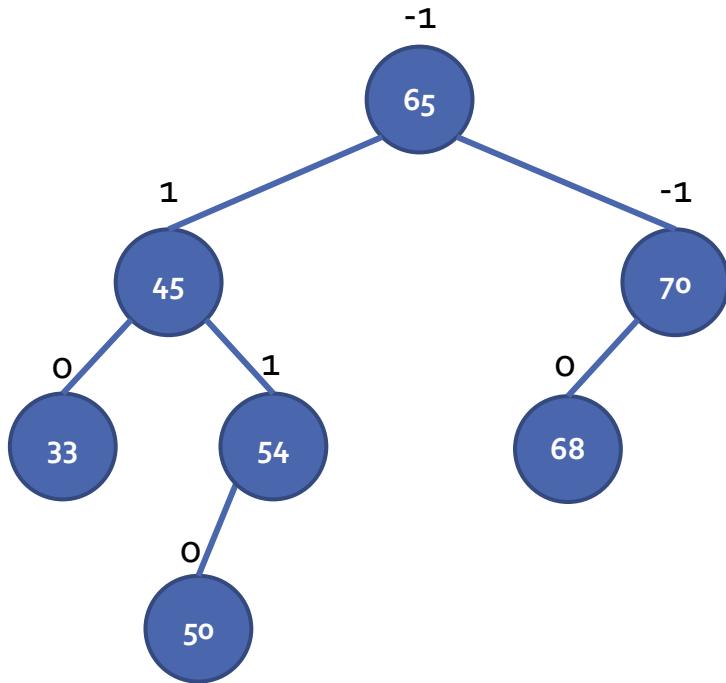
Para determinar si un árbol está balanceado se debe calcular el factor de equilibrio (FE) del nodo. El factor de equilibrio se define como la diferencia entre la altura del subárbol derecho con la altura del subárbol izquierdo:

$$FE = HRD - HRI$$

El factor de equilibrio puede ser -1, 0 ó 1. Cuando el FE genera un valor de -2 ó 2, el árbol debe ser restructurado.

## Ejemplo

Dado el siguiente árbol T, calcular el FE del nodo 65.



$$FE_{65} = 2 - 3 = -1$$

## **Inserción en un árbol balanceado.**

Para mantener un árbol balanceado es necesario comprobar el factor de equilibrio una vez realizada la inserción.

El proceso de inserción en un árbol balanceado es sencillo, empero, para mantener el equilibrio, se requieren operaciones auxiliares, lo que complica el algoritmo.

Para insertar un nodo se debe seguir el camino de búsqueda del árbol. Una vez insertado se calcula el FE (que será o para dicho nodo) y se regresa por el camino de búsqueda calculando el FE de los nodos visitados. Si en algún nodo se rompe el criterio de equilibrio se debe reestructurar el árbol.

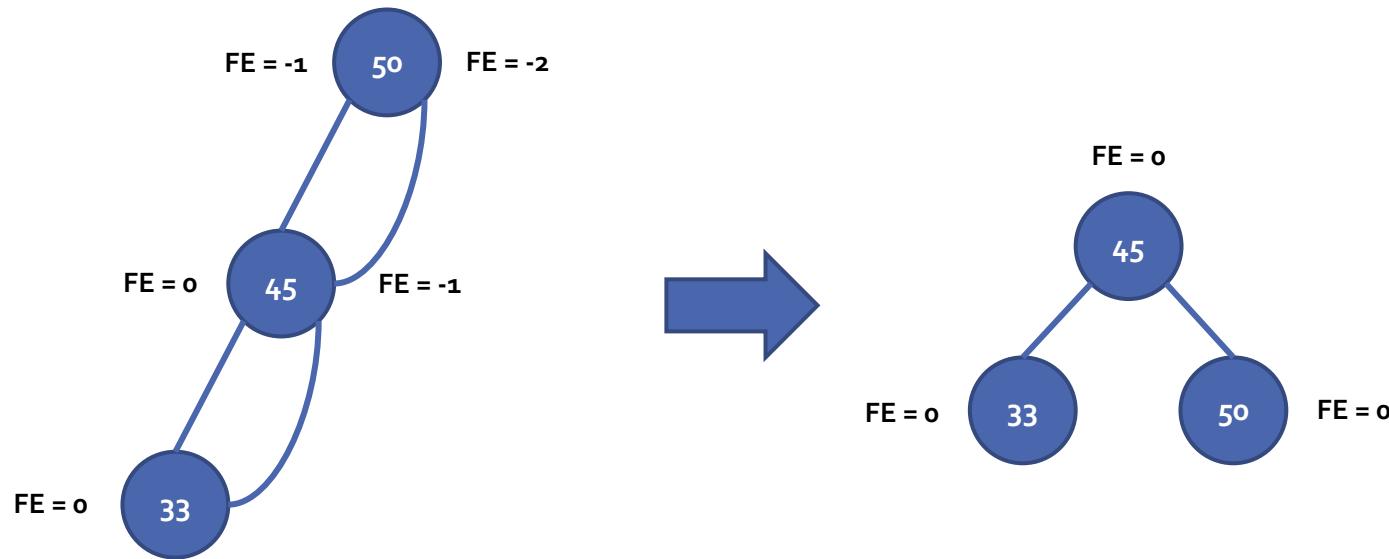
El ciclo anterior se rompe cuando se llega a la raíz del árbol o cuando se realiza el reacomodo en un nodo.

El reacomodo del árbol implica rotar los nodos para equilibrar la estructura. Existen 2 tipos de rotaciones: simple (que involucra la rotación de dos nodos) y compuesta (que involucra la rotación de tres nodos).

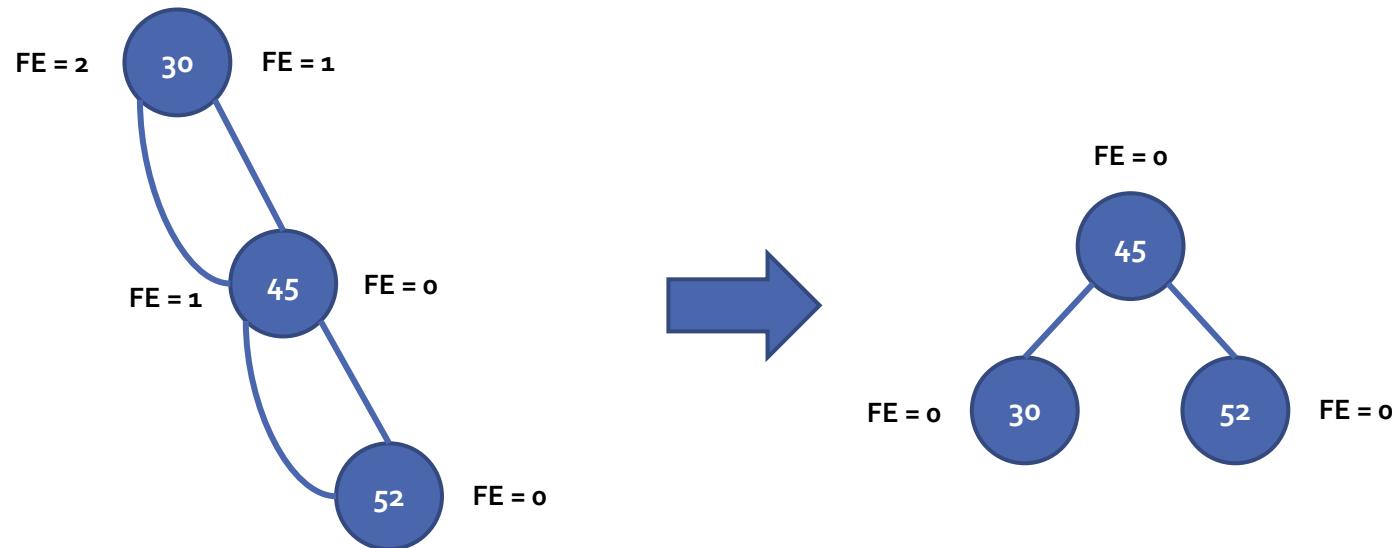
Una rotación simple se puede realizar por la rama derecha (DD) o por la rama izquierda (II) de los nodos involucrados.

Una rotación compuesta se puede realizar por la rama derecha e izquierda (DI) o izquierda y derecha (ID) de los nodos involucrados.

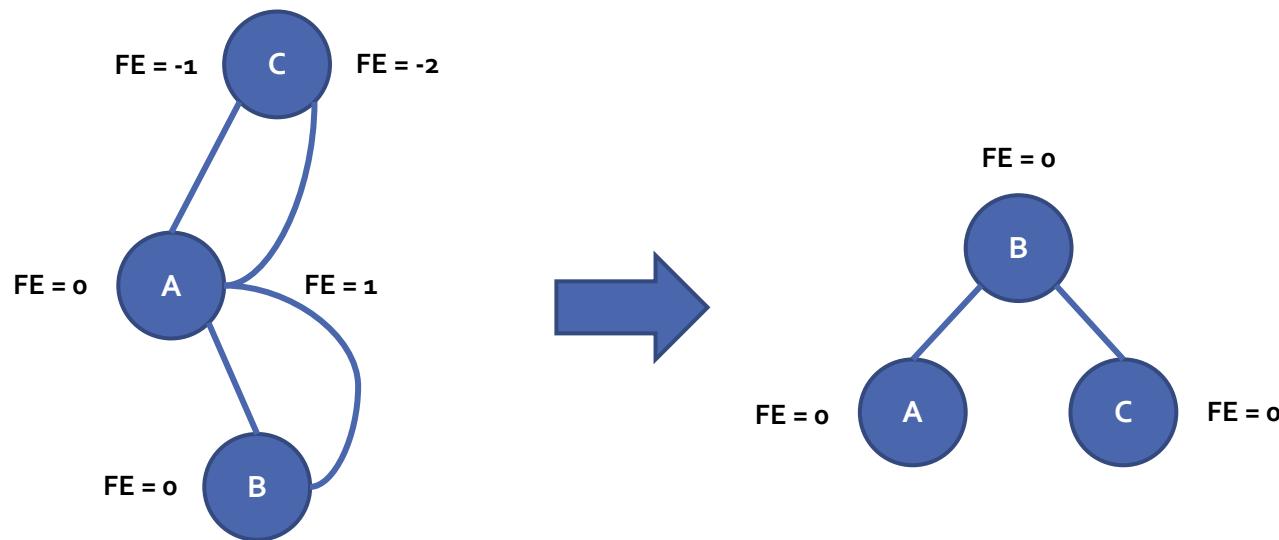
La rotación simple II se realiza cuando el factor de equilibrio se rompe en una rama izquierda-izquierda de la siguiente manera:



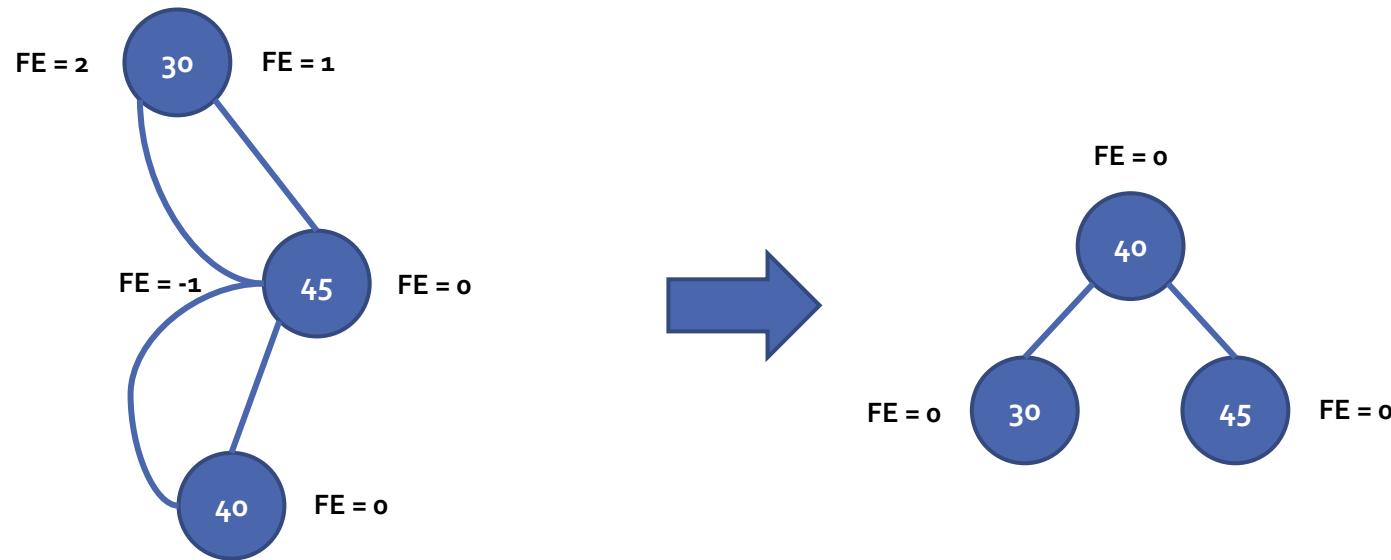
La rotación simple DD se realiza cuando el factor de equilibrio se rompe en una rama derecha-derecha de la siguiente manera:



La rotación compuesta ID se realiza cuando el factor de equilibrio se rompe en una rama izquierda-derecha de la siguiente manera:



La rotación compuesta DI se realiza cuando el factor de equilibrio se rompe en una rama derecha-izquierda de la siguiente manera:



## Ejemplo

Se desean insertar las siguientes claves en un árbol binario y balanceado vacío:

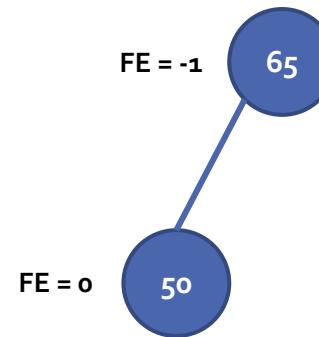
65 – 50 – 23 – 70 – 82 – 68 – 39

## Ejemplo

1) Se inserta la clave 65 en el árbol vacío

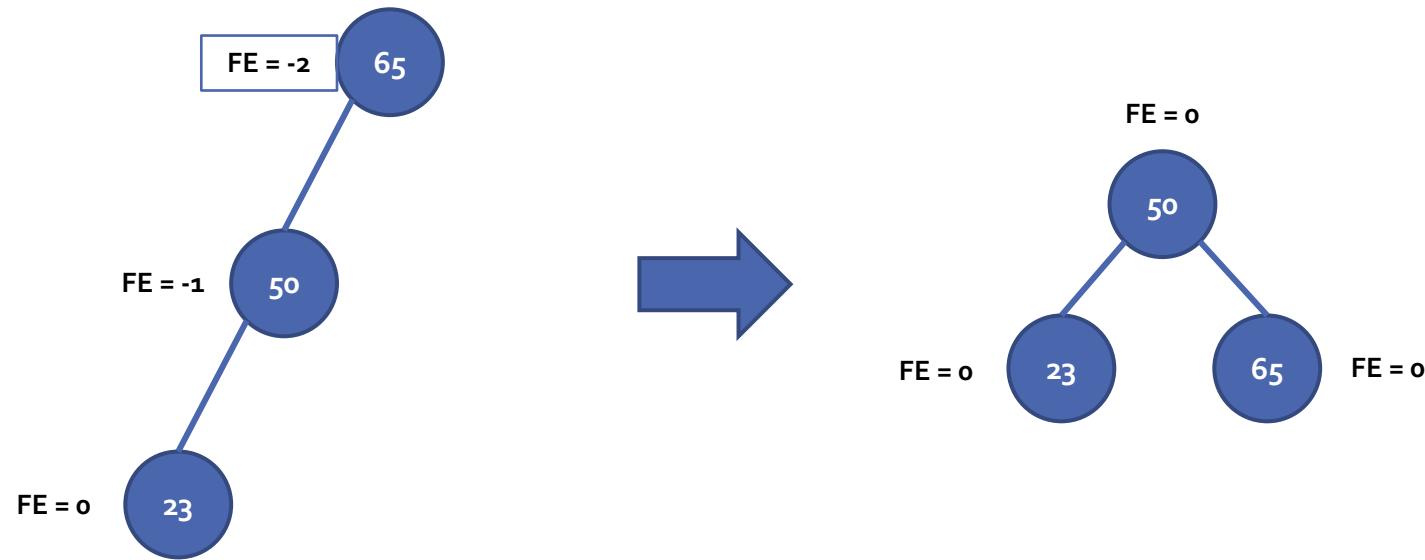


2) Se inserta la clave 50.



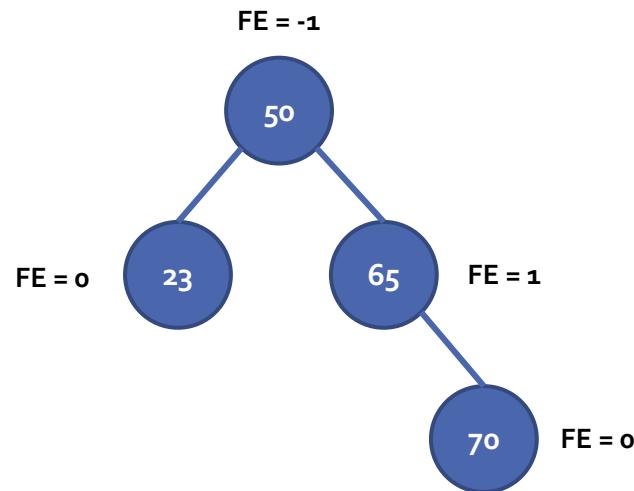
## Ejemplo

3) Se inserta la clave 23.



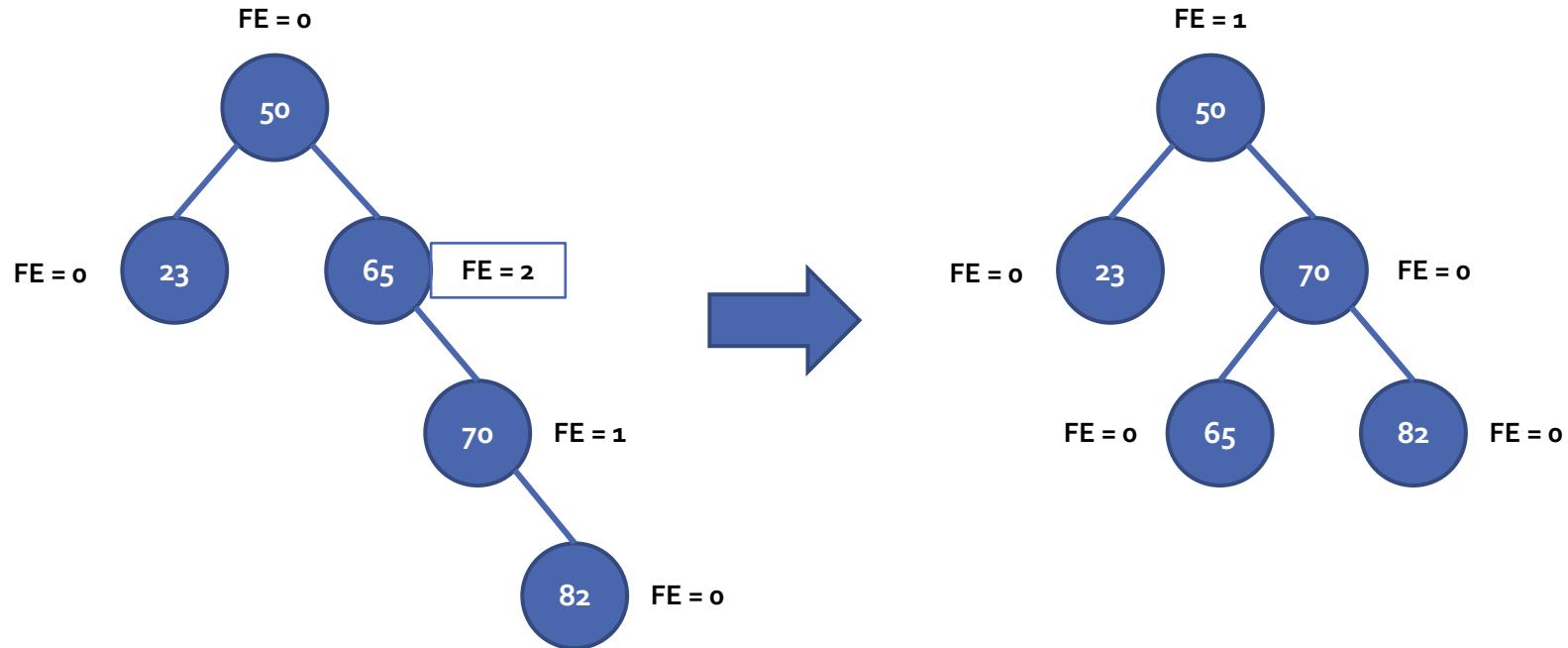
## Ejemplo

4) Se inserta la clave 70.



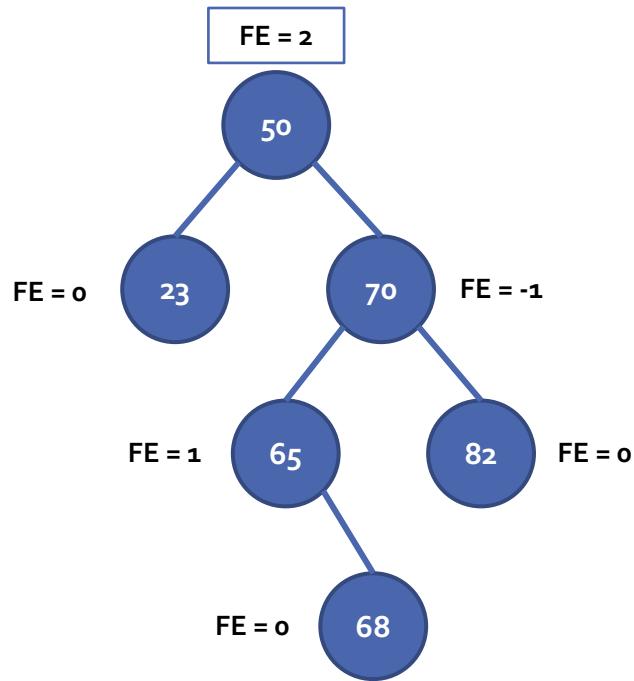
## Ejemplo

5) Se inserta la clave 82.



## Ejemplo

6) Se inserta la clave 68.



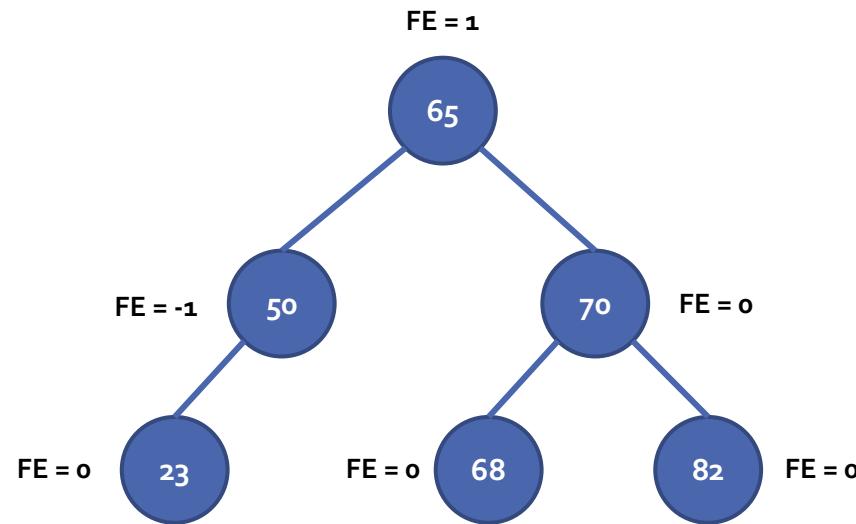
El nodo 50 rompe el factor de equilibrio. En este caso, se tiene una rotación compuesta derecha-izquierda (debido a que la rama más larga la forman los nodos 50, 70, 65).

Si  $n_1 \rightarrow 50$ ,  $n_2 \rightarrow 70$  y  $n_3 \rightarrow 65$ , las rotaciones que se deben realizar son:

$n_2.\text{left} \rightarrow n_3.\text{right}$   
 $n_3.\text{right} \rightarrow n_2$   
 $n_1.\text{right} \rightarrow n_3.\text{left}$   
 $n_3.\text{left} \rightarrow n_1$

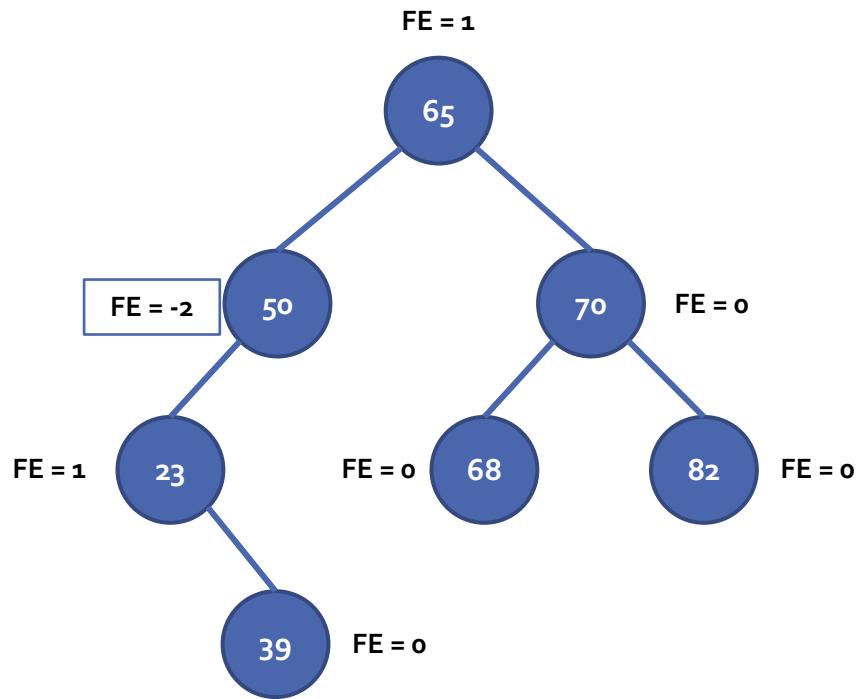
## Ejemplo

Al final de la rotación, el árbol queda de la siguiente manera:



## Ejemplo

7) Se inserta la clave 39.



El nodo 50 rompe el factor de equilibrio. En este caso, se tiene una rotación compuesta izquierda-derecha.

Si  $n_1 \rightarrow 50$ ,  $n_2 \rightarrow 23$  y  $n_3 \rightarrow 39$ , las rotaciones que se deben realizar son:

$n_2.\text{right} \rightarrow n_3.\text{left}$

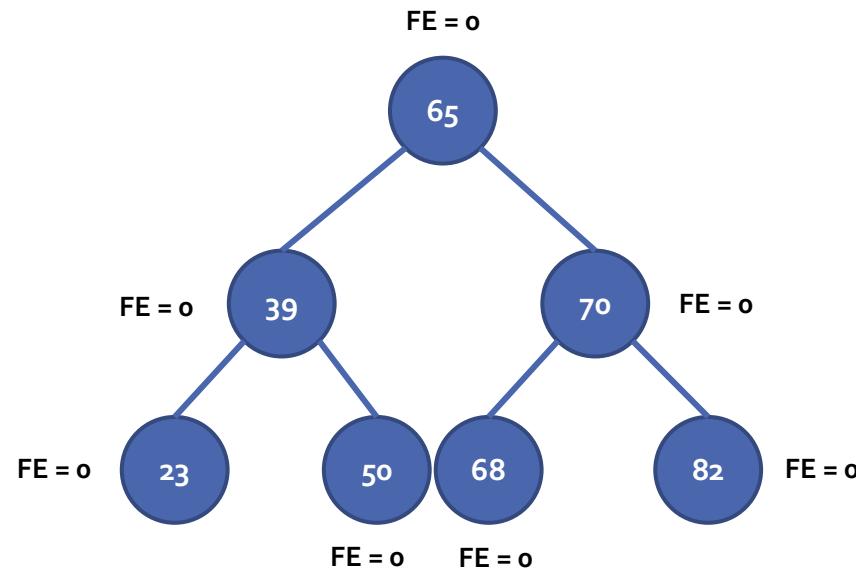
$n_3.\text{left} \rightarrow n_2$

$n_1.\text{left} \rightarrow n_3.\text{right}$

$n_3.\text{right} \rightarrow n_1$

## Ejemplo

Al final de la rotación, el árbol queda de la siguiente manera:



## **Eliminación en un árbol balanceado.**

Para eliminar elementos en un árbol balanceado una vez que se remueve el nodo deseado, se debe verificar que no se violen los principios del mismo, es decir, que se cumpla el factor de equilibrio.

Para eliminar datos de un árbol balanceado se utiliza un algoritmo de eliminación para árboles binarios y las operaciones de reacomodo del algoritmo de inserción de árboles balanceados.

El proceso de eliminación de un nodo debe tomar en cuenta los siguientes casos:

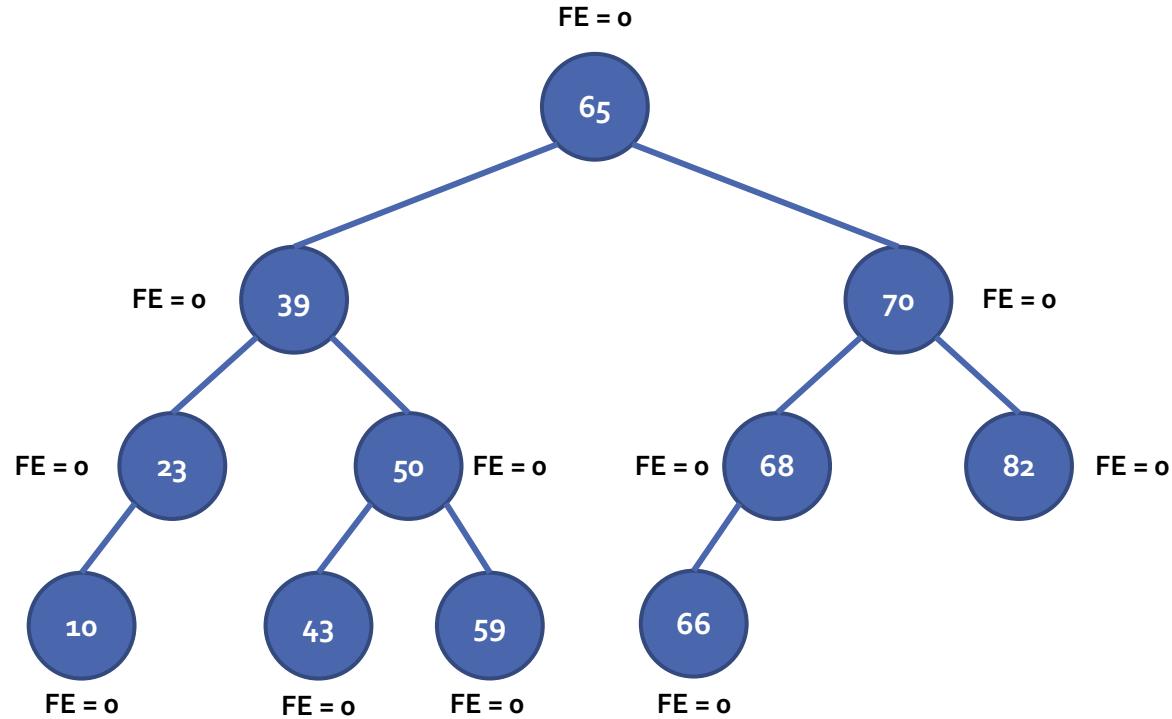
- Si el elemento a eliminar es una hoja, simplemente se suprime.
- Si el elemento a eliminar posee un solo hijo, entonces se tiene que sustituir por ese nodo.
- Si el elemento a eliminar tiene los dos hijos, entonces se sustituye por el nodo que se encuentra más a la izquierda en el subárbol derecho o por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

Una vez que se haya realizado el proceso de eliminación se debe regresar por el camino de búsqueda, calculando el factor de equilibrio de los nodos visitados. Si en algún nodo se rompe el criterio de equilibrio, se debe reestructurar el árbol.

A diferencia del algoritmo de inserción, el ciclo anterior se rompe cuando se llega a la raíz del árbol, es decir, se pueden efectuar más de una rotación en el camino hacia atrás.

## Ejemplo

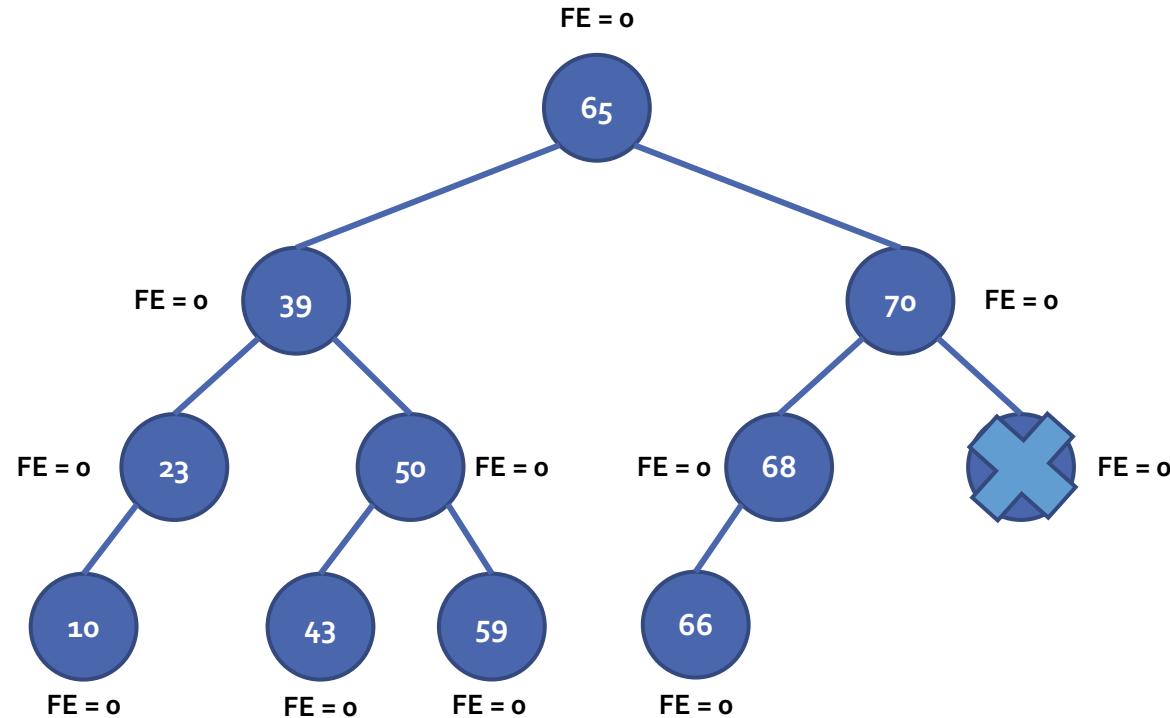
Dado el siguiente árbol:



Eliminar los elementos: 82, 10, 39, 65, 70, 23, 50 y 39.

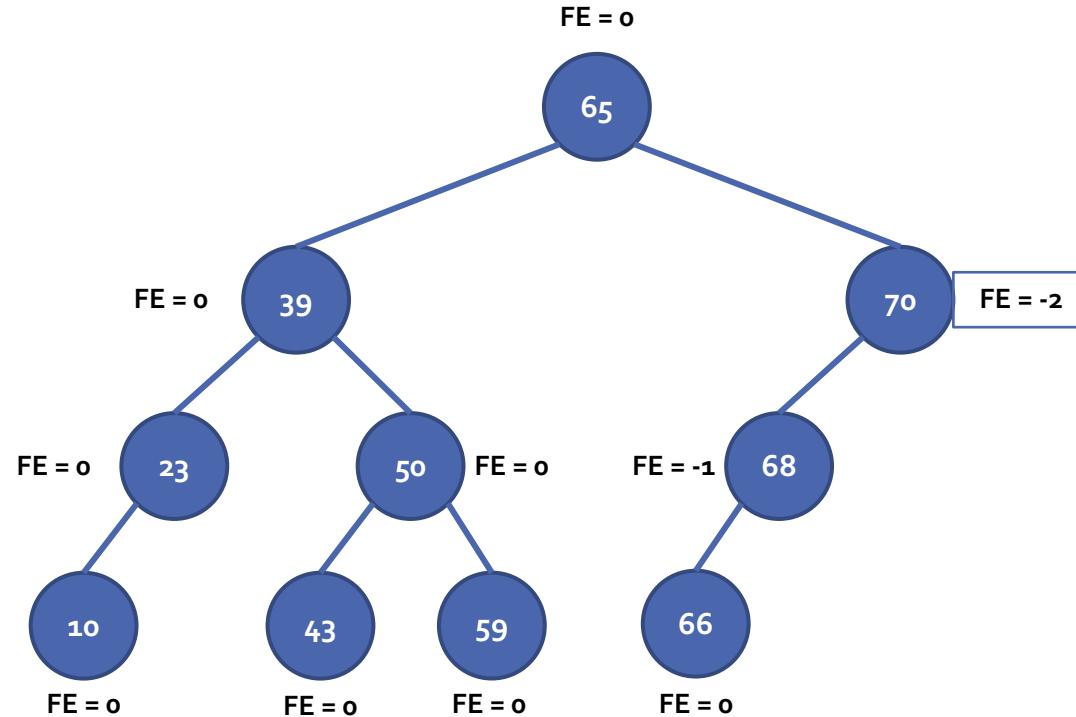
## Ejemplo

1) Se elimina la clave 82:



## Ejemplo

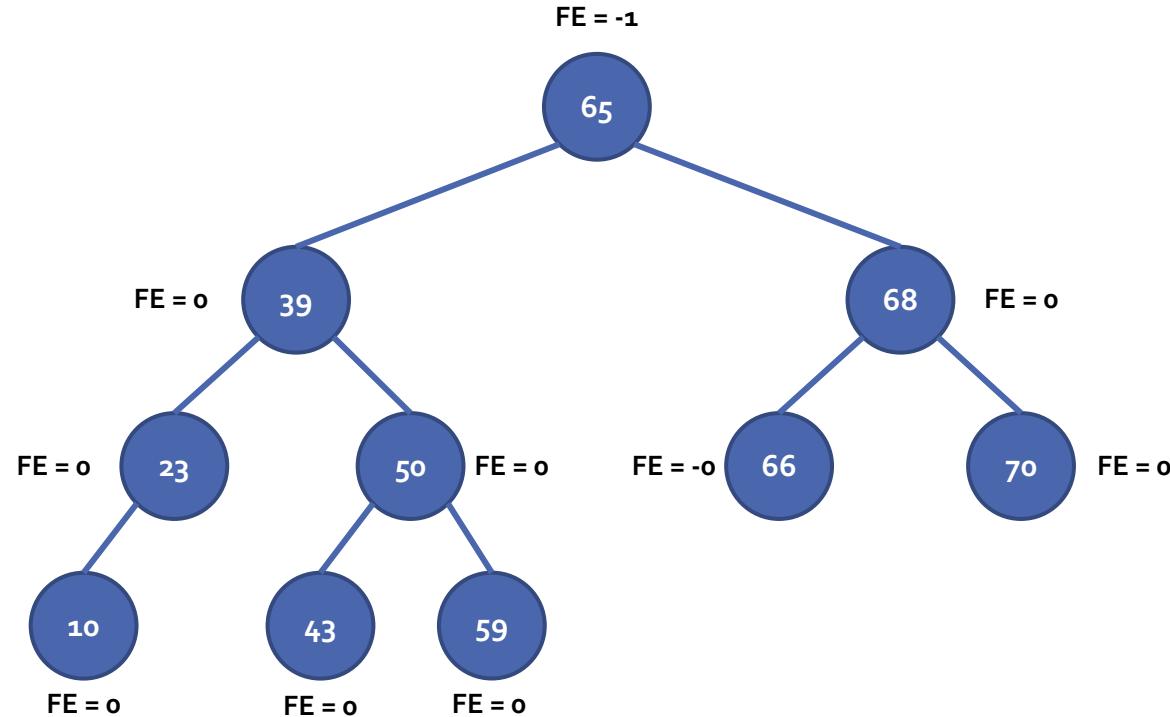
Al eliminar el nodo 82, en el nodo 70 se rompe el criterio de equilibrio



Se debe realizar una rotación II, donde  $n_1 \rightarrow 70$ ,  $n_2 \rightarrow$  rama izquierda de  $n_1$  (68) y, como  $FE(n_2) = -1$ , entonces  $n_3 \rightarrow$  rama izquierda de  $n_2$  (66).

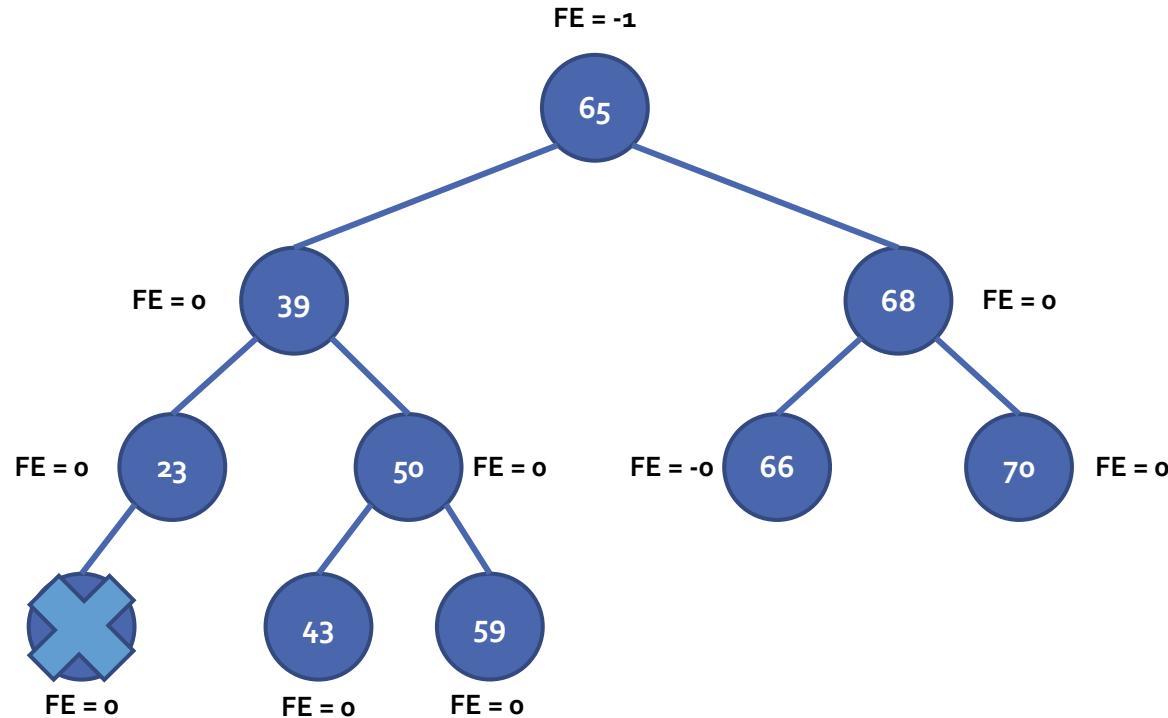
## Ejemplo

Después de la rotación II, el árbol queda balanceado hasta la raíz.



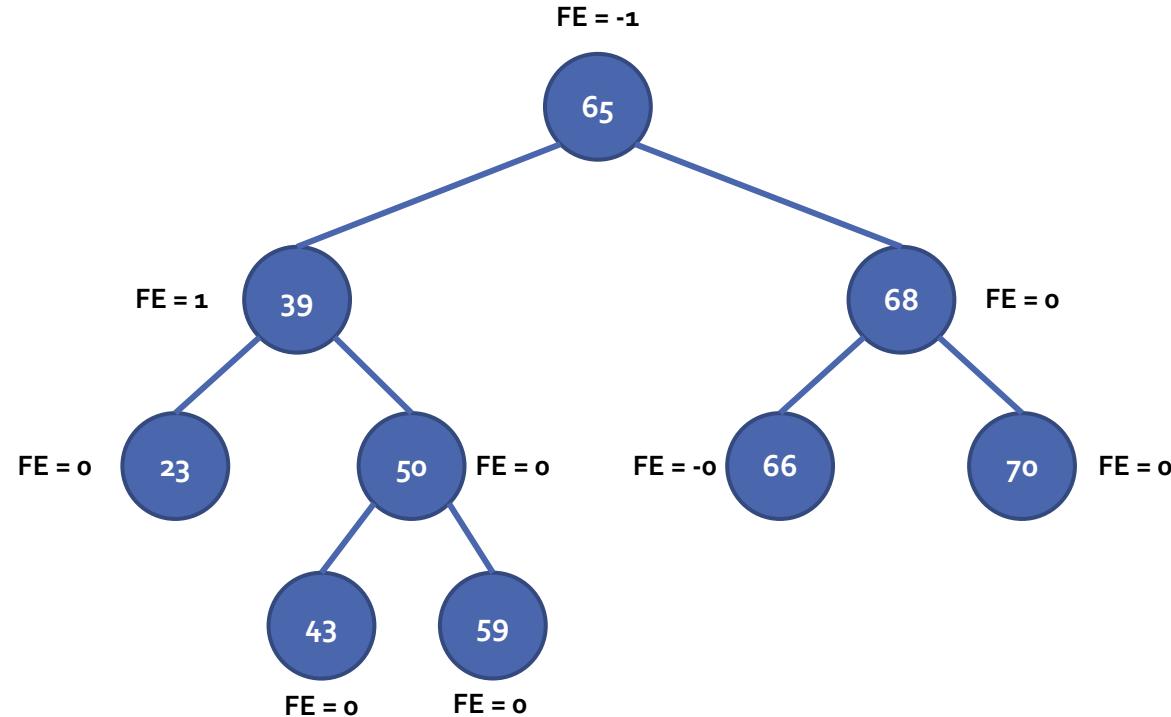
## Ejemplo

2) Se elimina la clave 10.



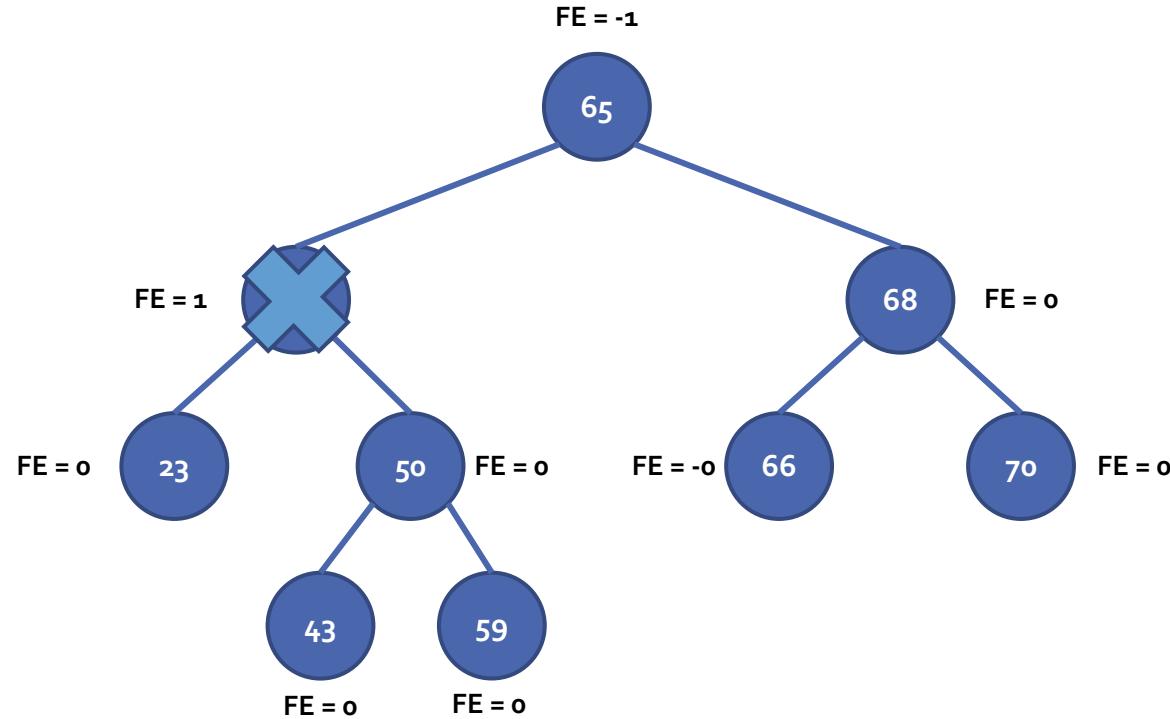
## Ejemplo

La eliminación del nodo 10 no genera reacomodo en el árbol.



## Ejemplo

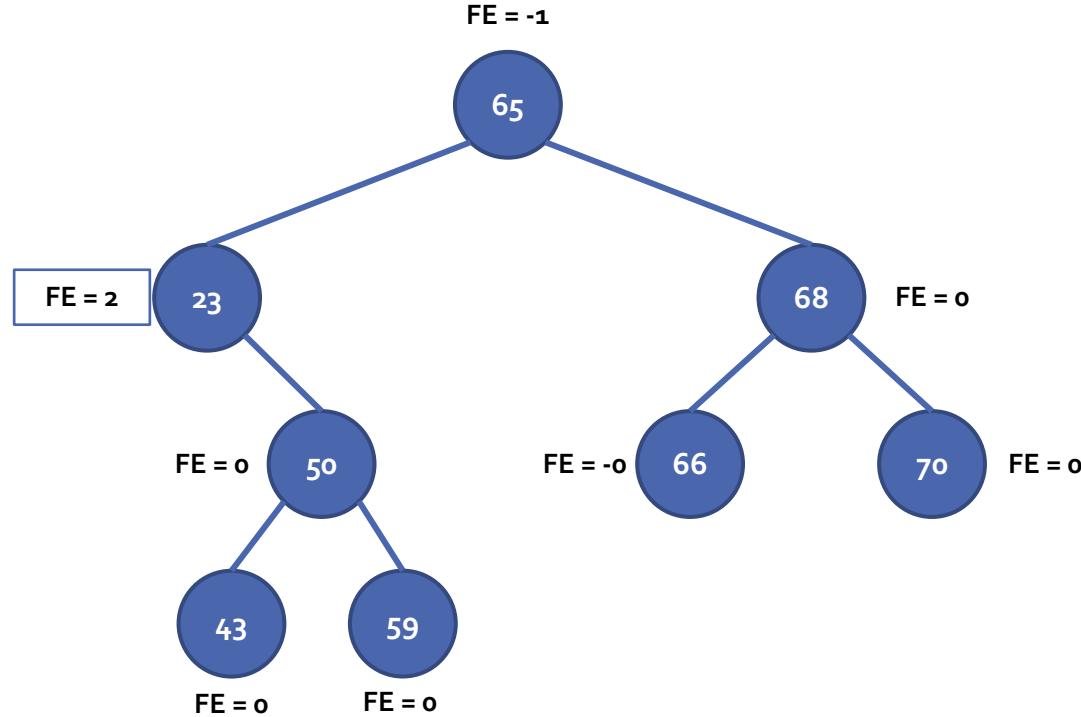
3) Se elimina la clave 39.



El nodo 39 posee dos hijos, por lo tanto, se realiza la sustitución por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

## Ejemplo

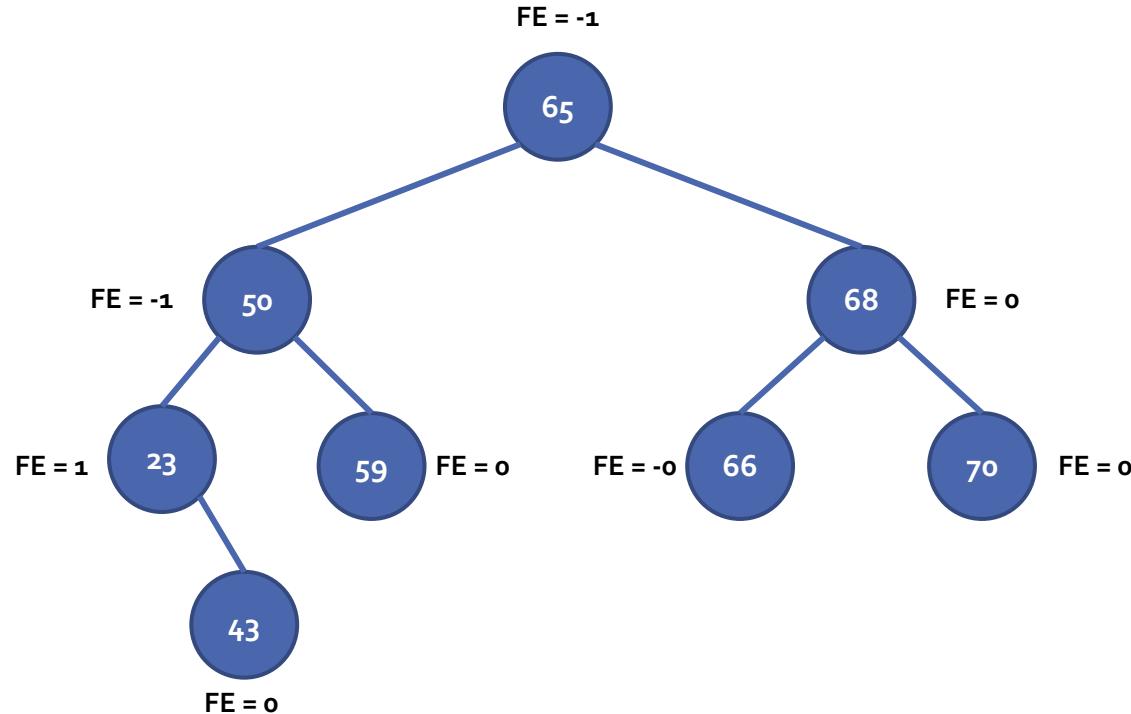
Después de la sustitución se altera el factor de equilibrio en el nodo con clave 23:



Por tanto, se debe realizar una rotación DD, donde  $n_1 \rightarrow 23$ ,  $n_2 \rightarrow$  rama derecha de  $n_1$  (50) y, como  $FE(n_2) = 0$ ,  $n_3 \rightarrow$  rama derecha de  $n_2$  (59).

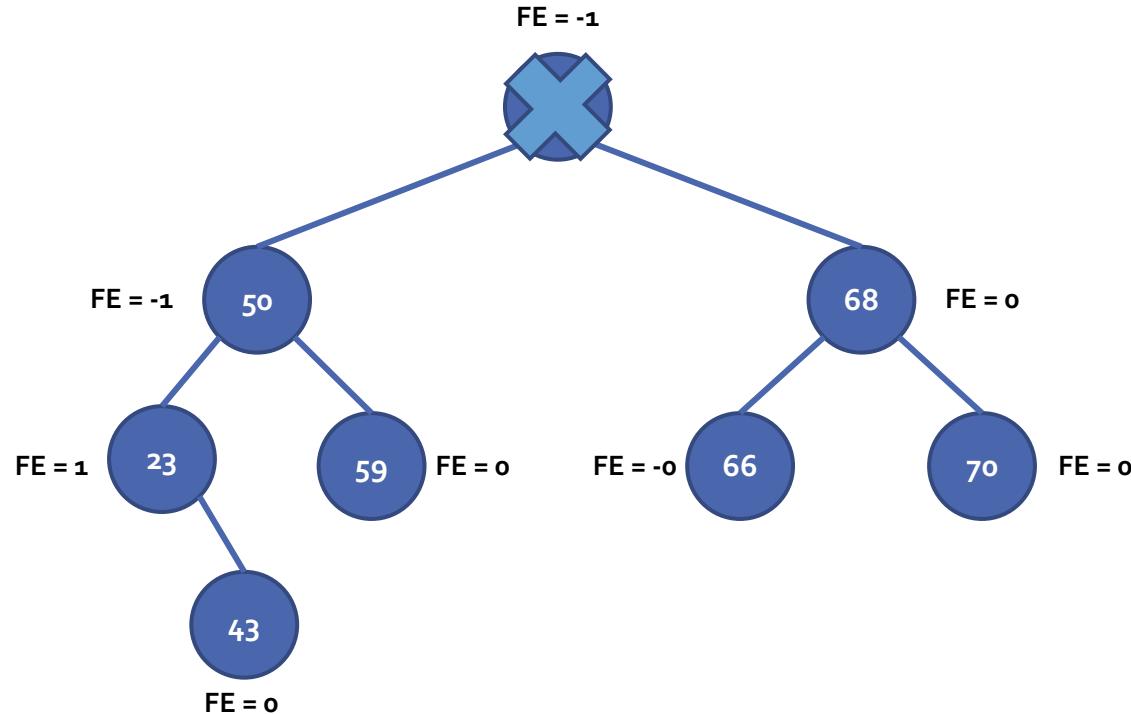
## Ejemplo

Después de la rotación DD, el árbol queda balanceado hasta la raíz.



## Ejemplo

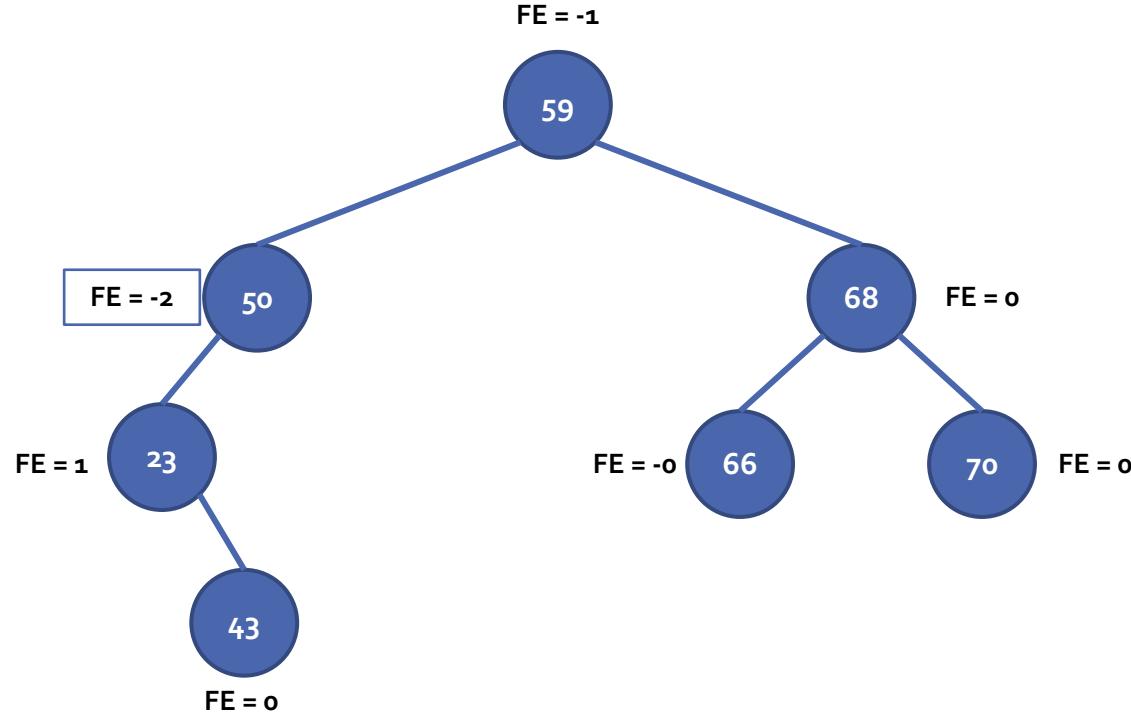
4) Se elimina la clave 65



El nodo 65 posee dos hijos, por lo tanto, se realiza la sustitución por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

## Ejemplo

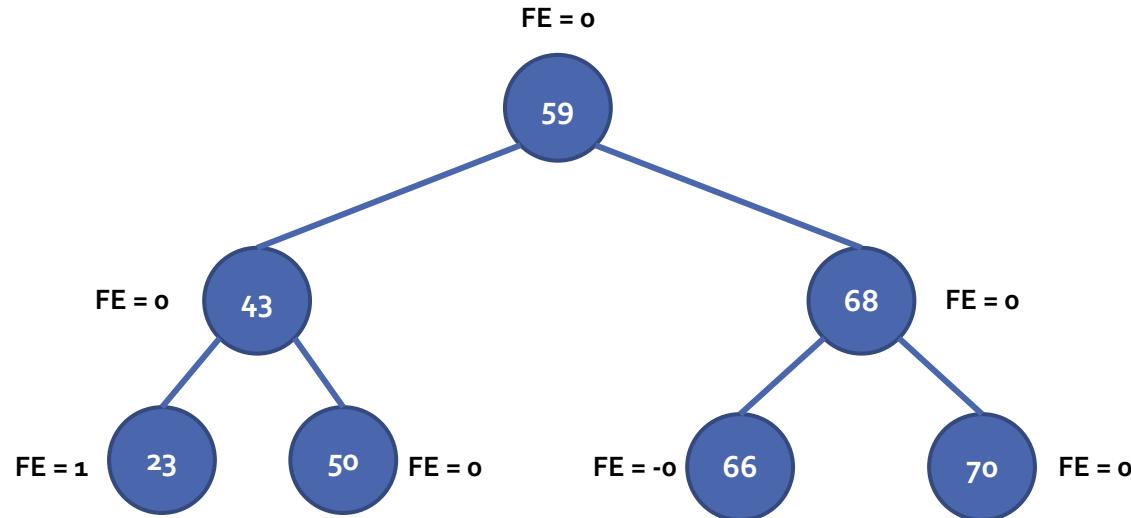
Después de la sustitución se altera el factor de equilibrio en el nodo con clave 50:



Por tanto, se debe realizar una rotación ID, donde  $n_1 \rightarrow 50$ ,  $n_2 \rightarrow$  rama izquierda de  $n_1$  (23) y, como  $FE(n_2) = 1$ ,  $n_3 \rightarrow$  rama derecha de  $n_2$  (59).

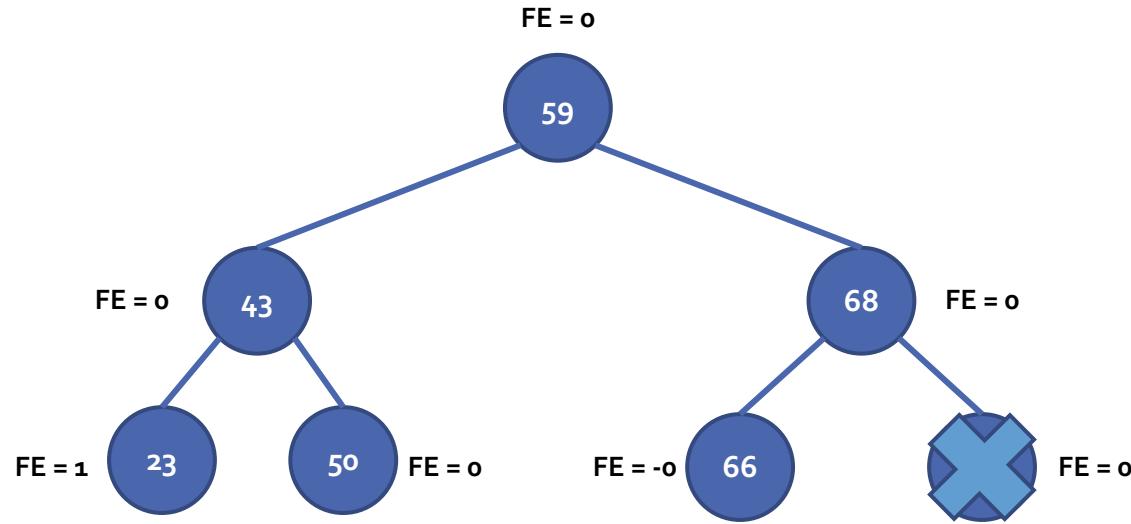
## Ejemplo

Después de la rotación ID, el árbol queda balanceado hasta la raíz.



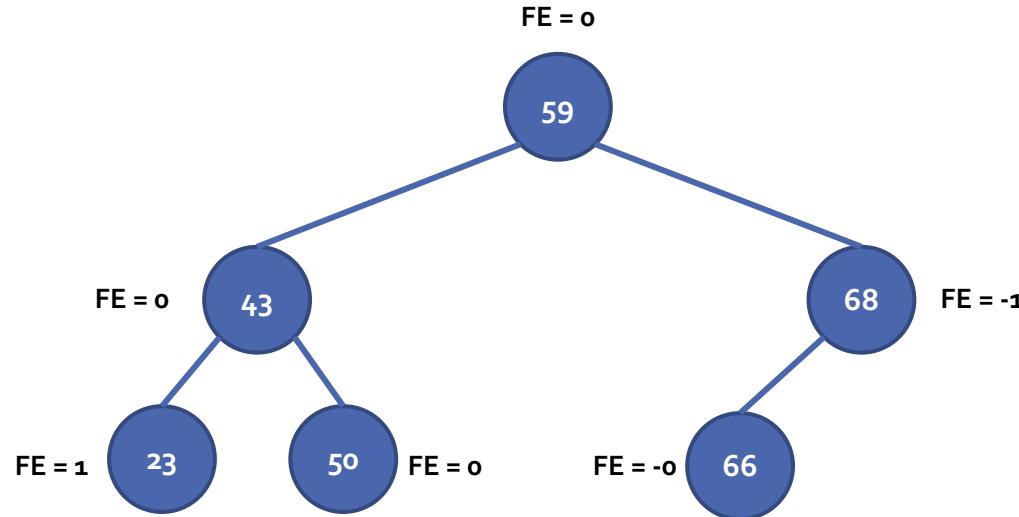
## Ejemplo

5) Se elimina la clave 70.



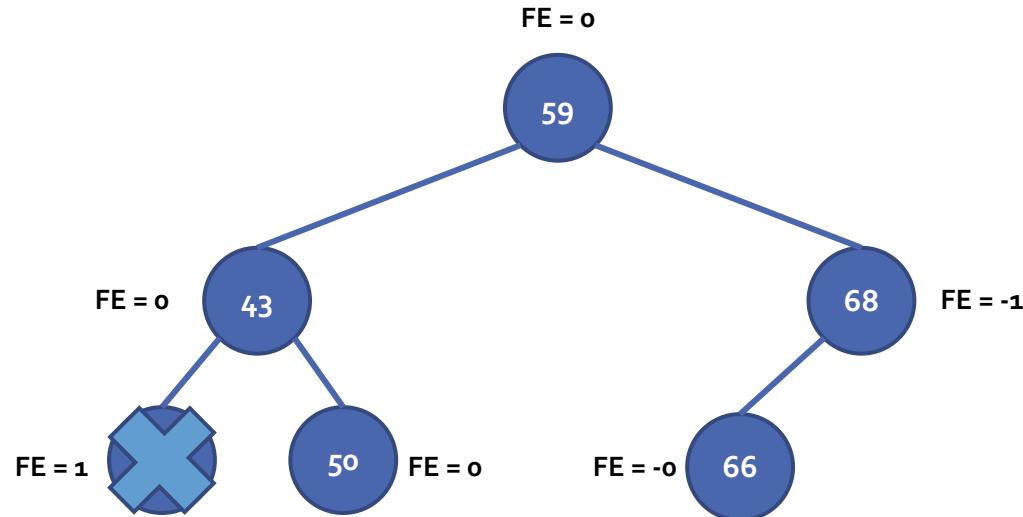
## Ejemplo

La eliminación del nodo 70 no genera reacomodo en el árbol.



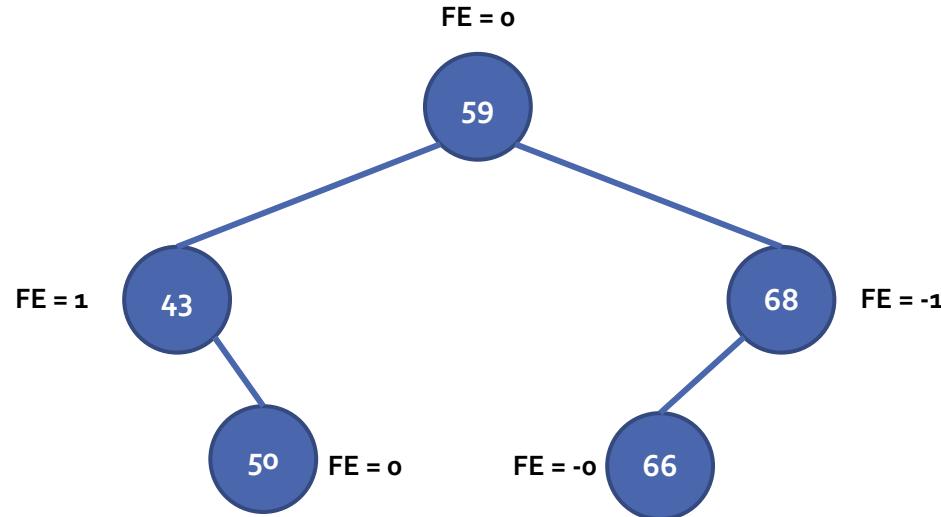
## Ejemplo

6) Se elimina la clave 23.



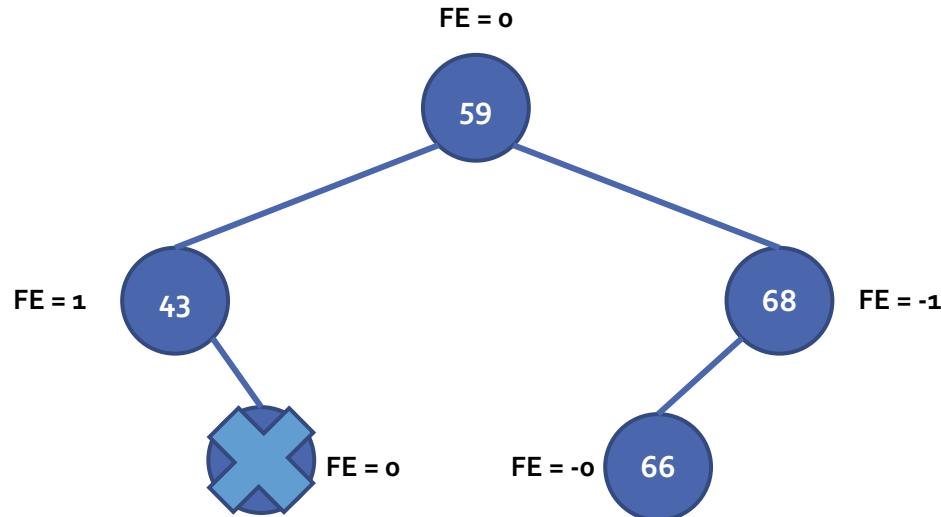
## Ejemplo

La eliminación del nodo 23 no genera reacomodo en el árbol.



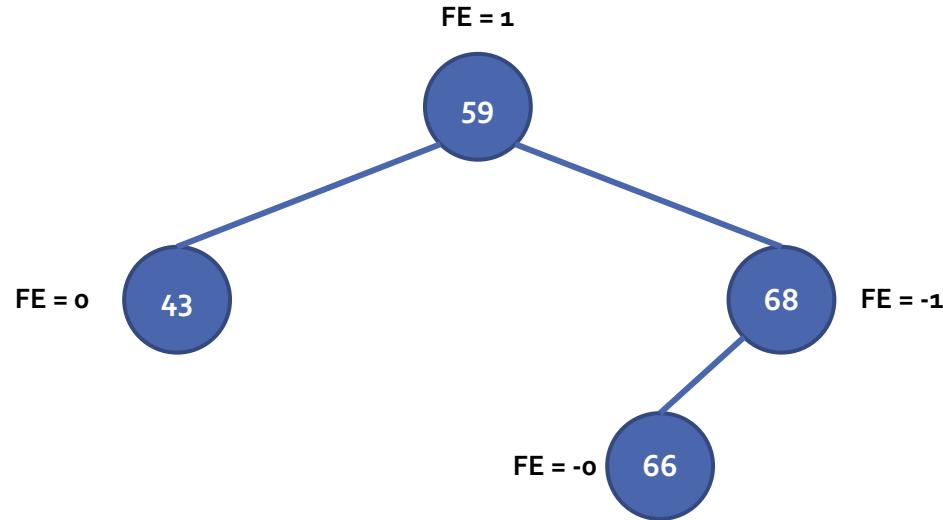
## Ejemplo

7) Se elimina la clave 50.



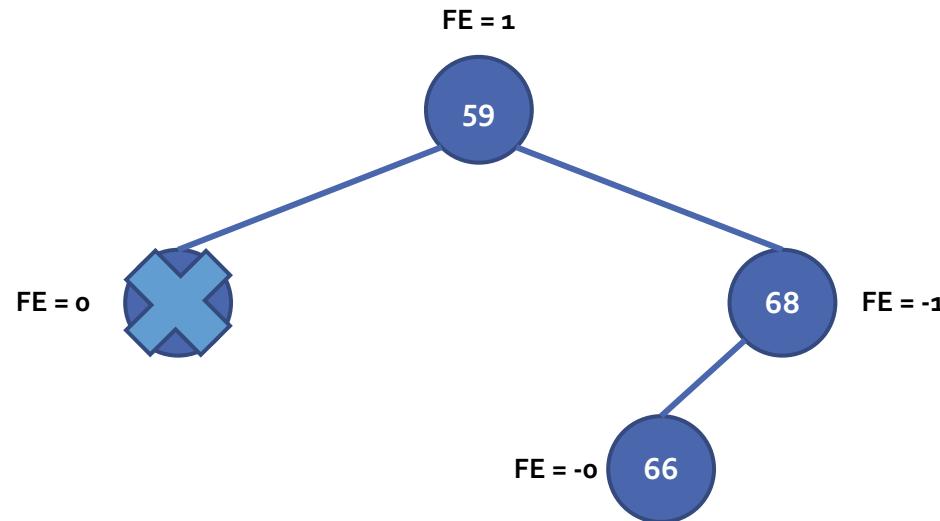
## Ejemplo

La eliminación del nodo 50 no genera reacomodo en el árbol.



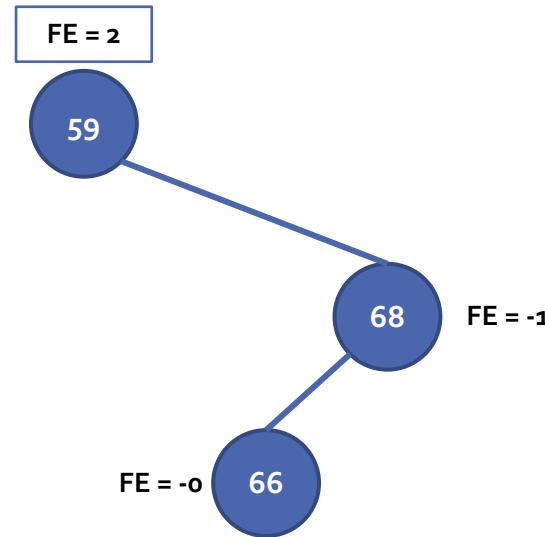
## Ejemplo

8) Se elimina la clave 43.



## Ejemplo

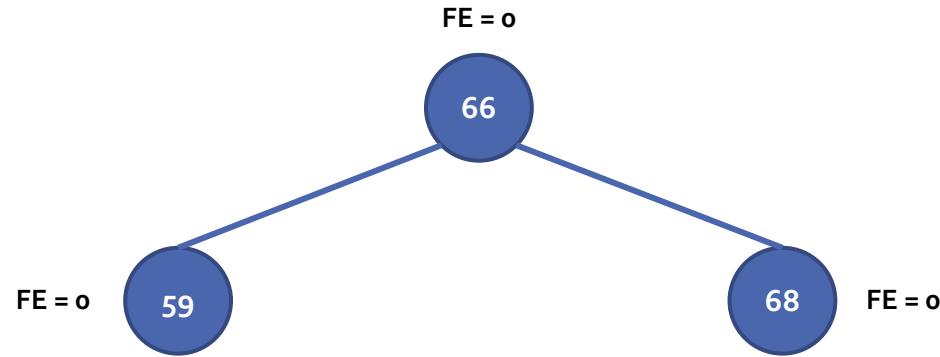
Al eliminar el nodo 43, en el nodo 59 se rompe el criterio de equilibrio



Se debe realizar una rotación DI, donde  $n_1 \rightarrow 59$ ,  $n_2 \rightarrow$  rama derecha de  $n_1$  (68) y, como  $FE(n_2) = -1$ , entonces  $n_3 \rightarrow$  rama izquierda de  $n_2$  (66).

## Ejemplo

Después de la rotación DI, el árbol queda balanceado hasta la raíz.



En general, las rotaciones son más frecuentes en las operaciones de inserción que en las operaciones de eliminación. Aproximadamente, por cada dos inserciones se produce una rotación, mientras que por cada cinco eliminaciones se produce una rotación.



## 4.3 ÁRBOLES B.

Los árboles que se han visto hasta el momento trabajan en la memoria principal de la computadora. Empero, existen aplicaciones en las que el volumen de información es tan grande que los datos no caben en la memoria principal y es necesario guardarlos en dispositivos de almacenamiento secundario (organizados en archivos).

Normalmente, el tiempo que se necesita para localizar un registro en la memoria principal está dado en microsegundos, mientras que el tiempo que se lleva localizar una página (con varios registros) en memoria secundaria está dado en el orden de milisegundos.

Por tanto, el tiempo de acceso es miles de veces más rápido en la memoria principal que en la memoria secundaria.

Si se tiene un árbol binario almacenado en disco, el tiempo promedio para localizar un nodo está dado por  $\log(n)$  accesos a disco, con  $n$  igual al número de nodos del árbol.

Si el árbol contiene 1,000,000 de elementos, se necesitaría, en promedio, 20 accesos al disco para encontrar un elemento. Si el árbol se organizara en páginas, de tal forma que cada página contenga 100 elementos, el tiempo de acceso está dado por  $\log_{100} (1000000)$ , por lo que se necesitarían solo 3 accesos a disco.

### **4.3.1 Árboles B.**

Los árboles B son un tipo de árboles balanceados que permiten organizar archivos con índices, que permiten el almacenamiento y recuperación de información en medios externos.

En los árboles B manejan páginas. Cada página posee, a su vez, un grupo de nodos, los cuales poseen una clave o llave. A partir de esa llave es como se indexan los valores en cada página y en el árbol en sí.

Cada página de un árbol B de orden d contiene  $2d$  claves como máximo y  $d$  claves como mínimo. La última condición garantiza que cada página esté llena, como mínimo, hasta la mitad.

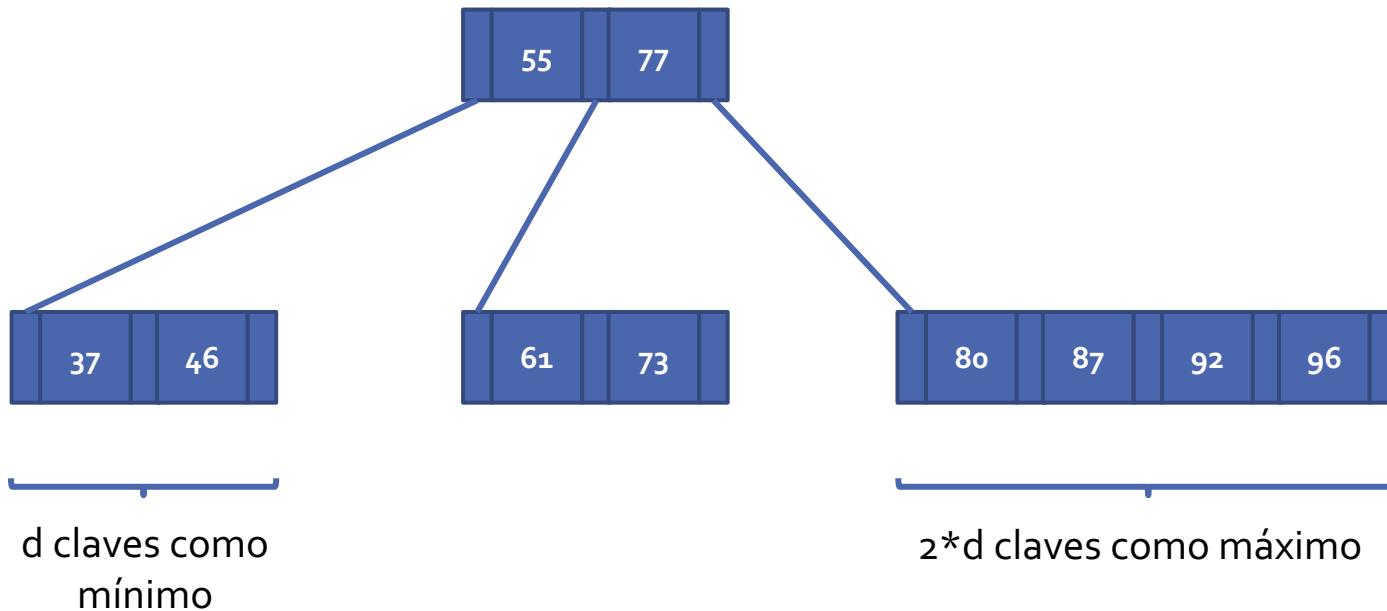
Además, cada página en un árbol B de orden d contiene como máximo  $2d$  nodos y como mínimo  $d$  nodos, excepto la página raíz que puede contener solo un dato (un nodo). Así mismo, cada página puede tener como mínimo  $d+1$  hijos y como máximo  $2d+1$  hijos, excepto el nodo raíz, el cual puede tener  $d$  hijos.

## Ejemplo

El siguiente grafo representa un árbol B de orden 2:

$$d(A) = 2$$

$$\begin{aligned} \text{MIN(hijos)} &= d+1 \\ \text{MAX(hijos)} &= 2*d+1 \end{aligned}$$



Con el diagrama anterior podemos confirmar las siguientes afirmaciones:

- Cada página, con excepción de la raíz, contiene de  $d$  a  $2d$  elementos, donde  $d$  es el grado del árbol.
- La raíz puede almacenar de 1 a  $2*d$  elementos.
- Cada página (con excepción de la raíz y las hojas) posee de  $d+1$  a  $2*d+1$  descendientes.
- La página raíz tiene al menos dos descendientes.
- Las páginas hoja están todas al mismo nivel.

### **4.3.2 Árboles B, algoritmos.**

Los algoritmos básicos dentro de los árboles B son dos: Inserción y eliminación. Tanto al insertar como al eliminar un nodo, es importante conservar el orden en cada página, lo que vuelve un poco más complicados los algoritmos antes mencionados.

## **Inserción en un árbol B.**

Debido a que todas las hojas se encuentran al mismo nivel, por lo tanto, cualquier camino desde la raíz hasta alguna hoja tiene la misma longitud.

Los árboles B crecen de abajo hacia arriba, es decir, desde las hojas hasta la raíz.

Para insertar un nodo en un árbol B se siguen los siguientes pasos:

1. Localizar la página donde se va a insertar la clave.
2. Si el número de elementos de la página ( $m$ ) es menor o igual a  $2*d$ ,  
ENTONCES

Se inserta la clave en el lugar que le corresponde  
DE LO CONTRARIO (si  $m$  es mayor a  $2*d$ )

La página afectada se divide en dos y se distribuyen las  
 $m+1$  claves de manera equitativa.

La clave intermedia sube a la página antecesora.

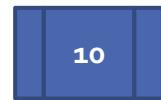
## Ejemplo

Se desean insertar las siguientes claves en un árbol B de orden 2:

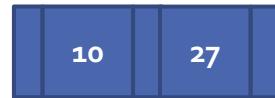
10 - 27 - 29 - 17 - 25 - 21 - 15 - 31 - 13 - 51 - 20 - 24 - 48 - 19 - 60 - 35 - 66

## Ejemplo

1) Se inserta la clave 10.

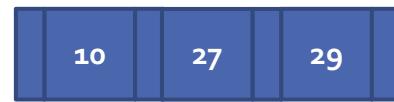


2) Se inserta la clave 27.

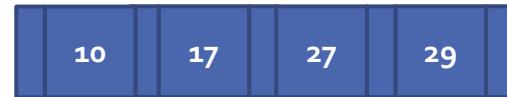


## Ejemplo

3) Se inserta la clave 29.

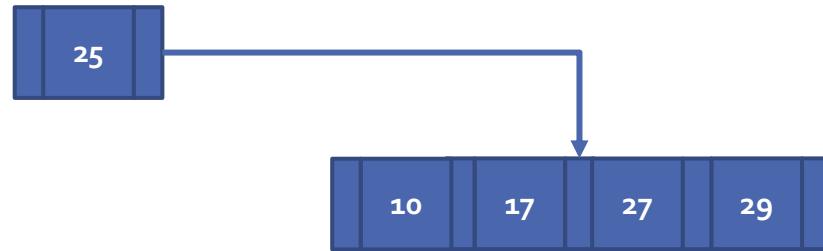


4) Se inserta la clave 17.



## Ejemplo

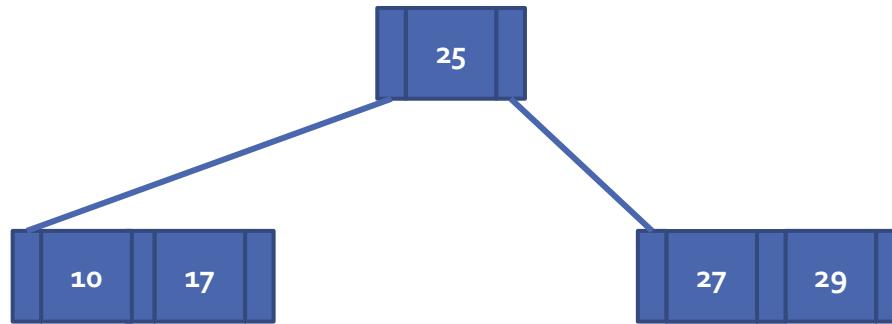
5) Se inserta la clave 25.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

## Ejemplo

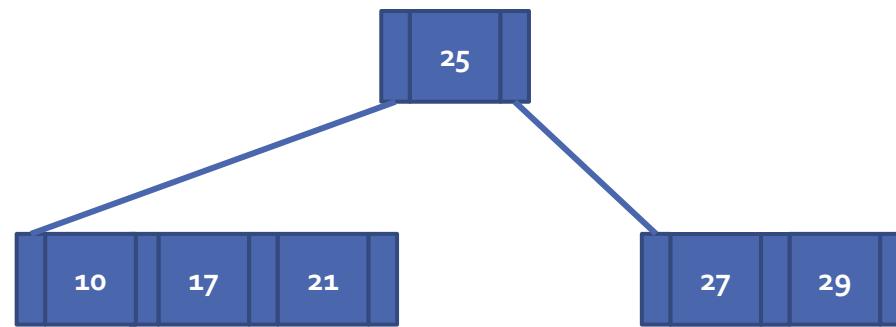
5) Se inserta la clave 25.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

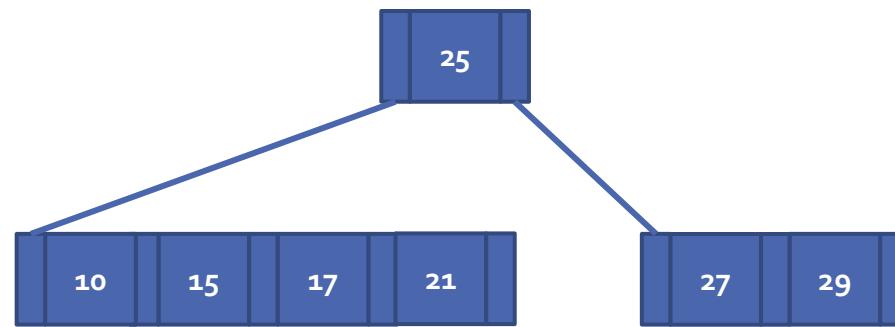
## Ejemplo

6) Se inserta la clave 21.



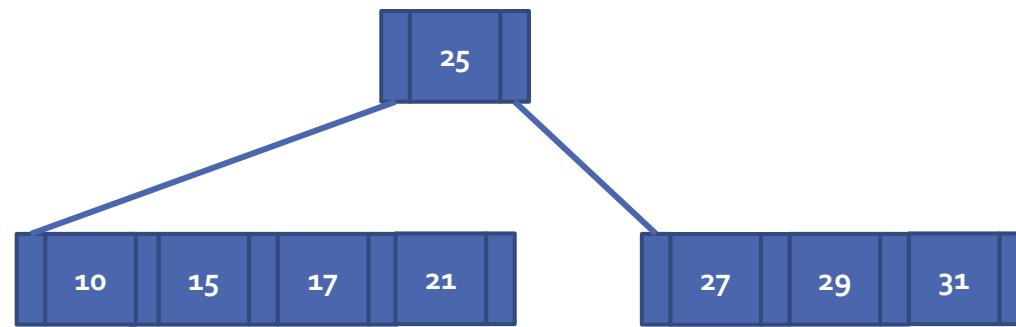
## Ejemplo

7) Se inserta la clave 15.



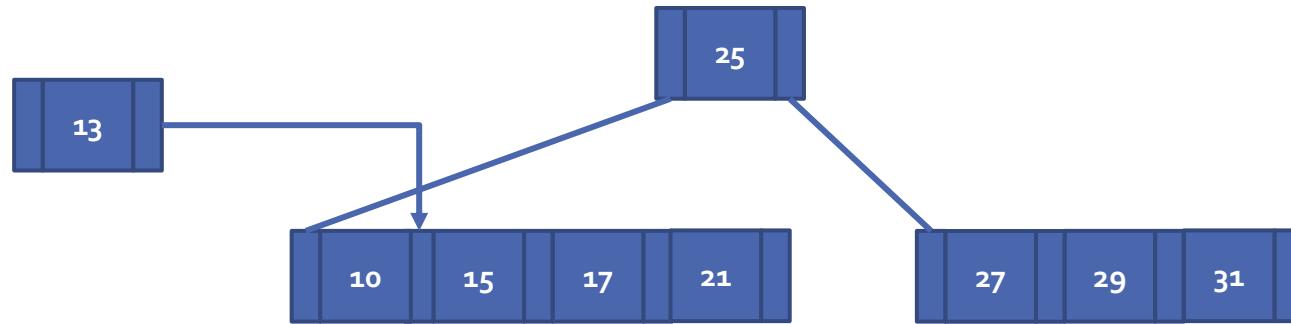
## Ejemplo

8) Se inserta la clave 31.



## Ejemplo

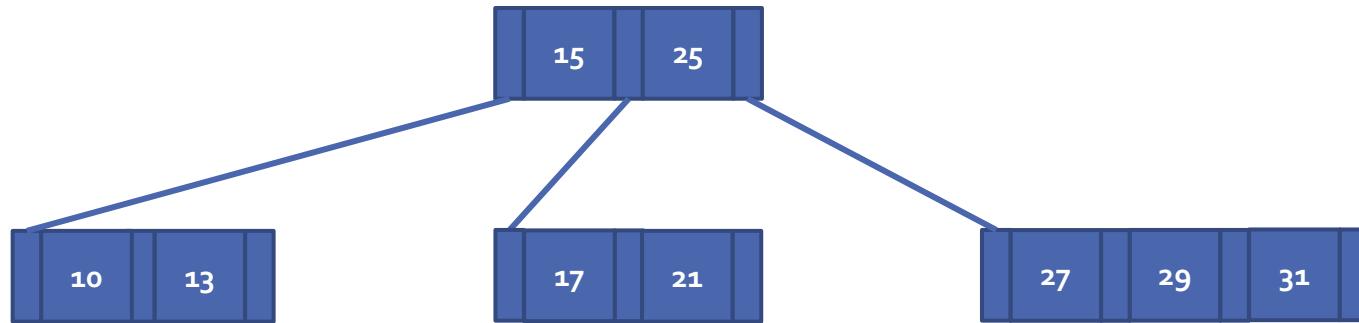
9) Se inserta la clave 13.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2^*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

## Ejemplo

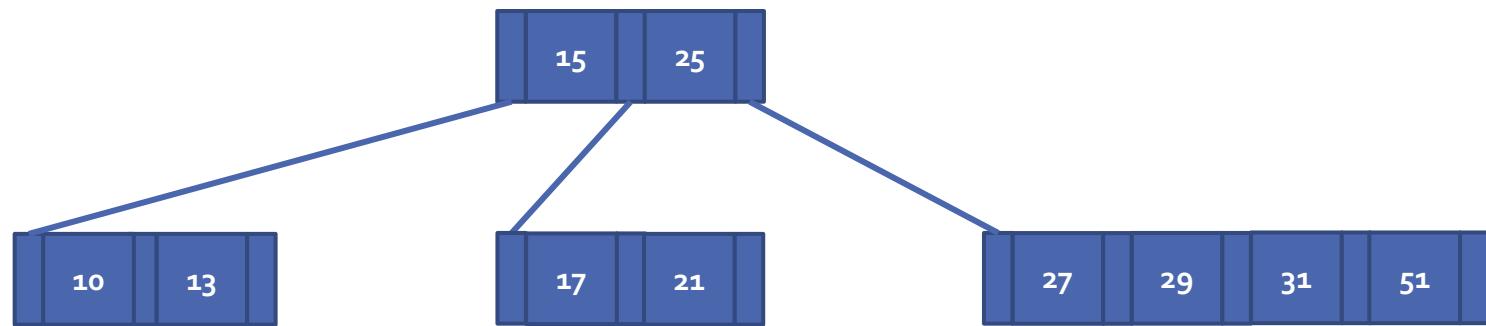
9) Se inserta la clave 13.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

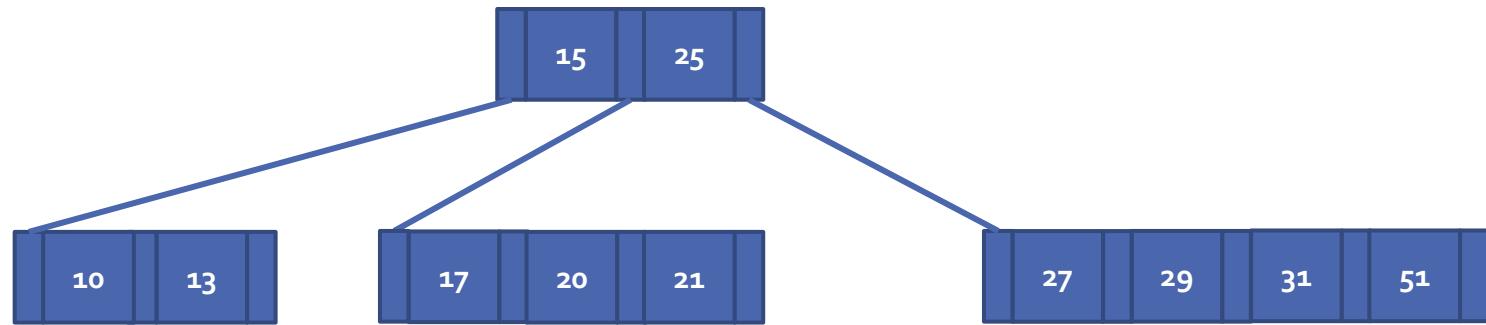
## Ejemplo

10) Se inserta la clave 51.



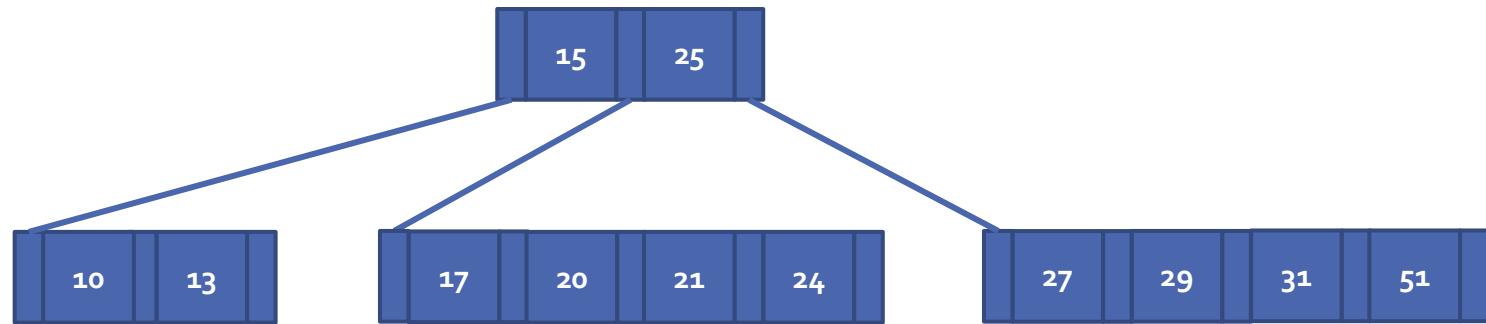
## Ejemplo

11) Se inserta la clave 20.



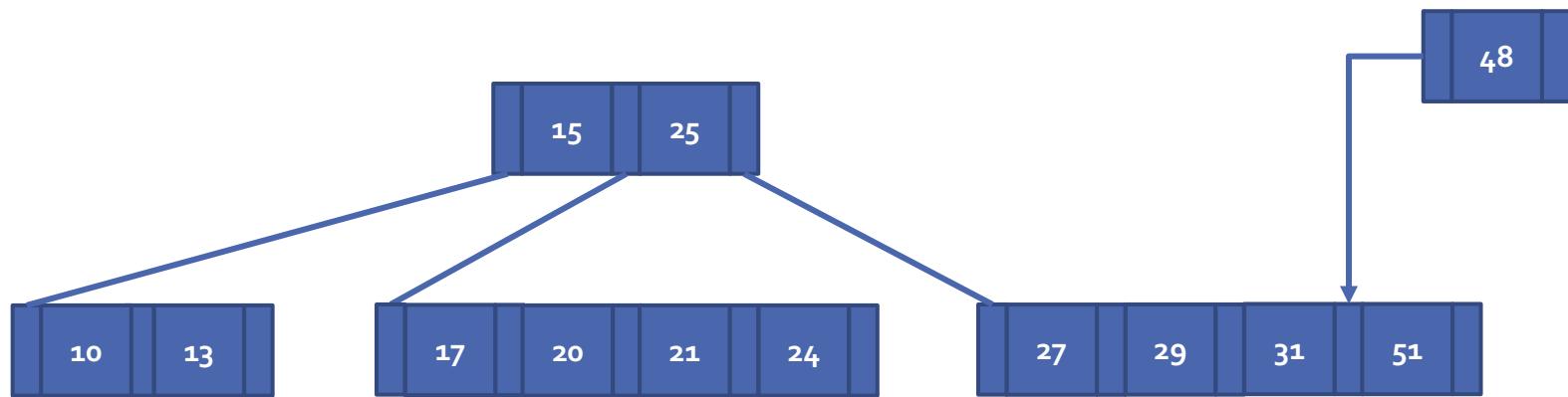
## Ejemplo

12) Se inserta la clave 24.



## Ejemplo

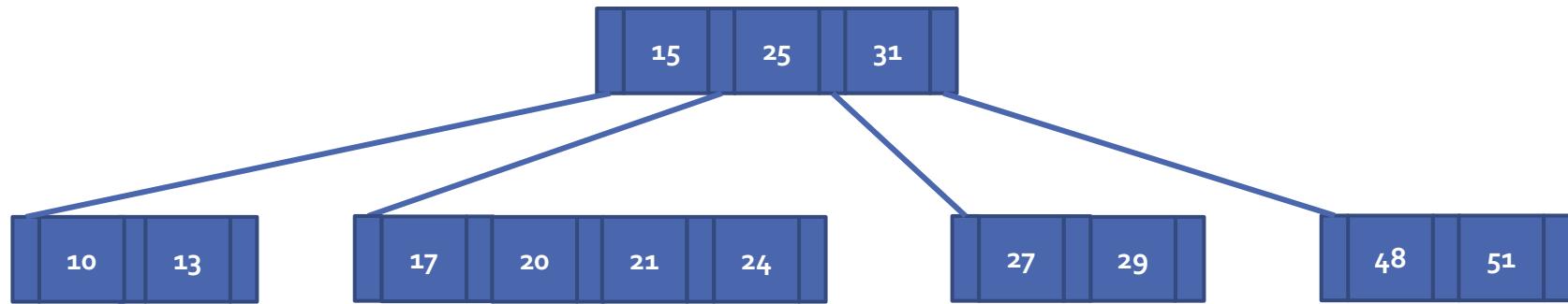
13) Se inserta la clave 48.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

## Ejemplo

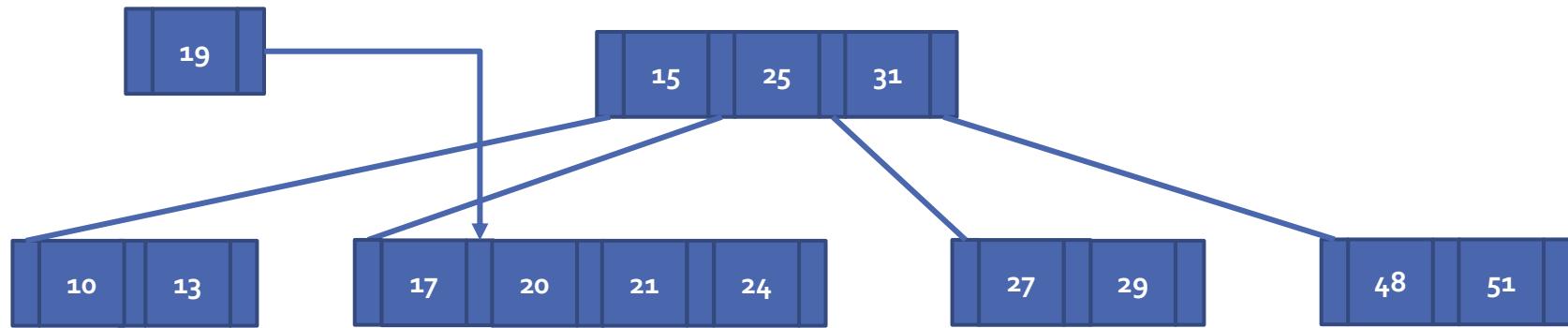
13) Se inserta la clave 48.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

## Ejemplo

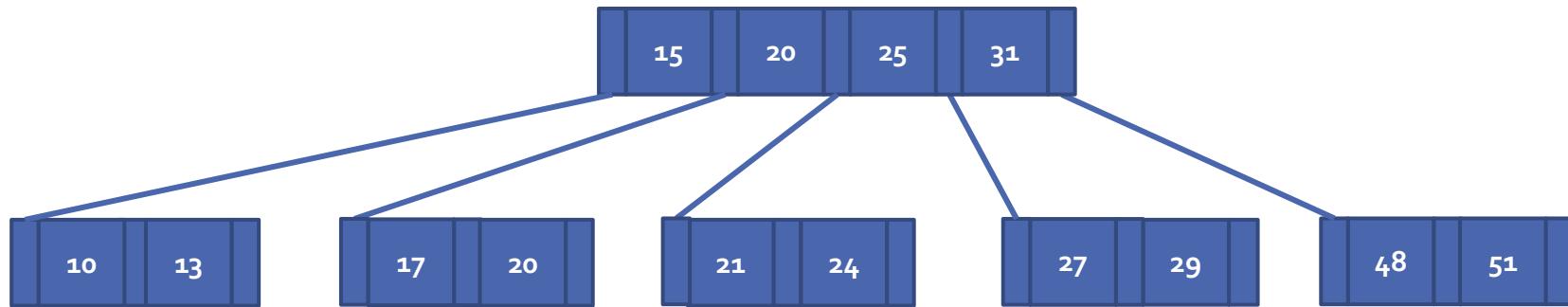
14) Se inserta la clave 19.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

## Ejemplo

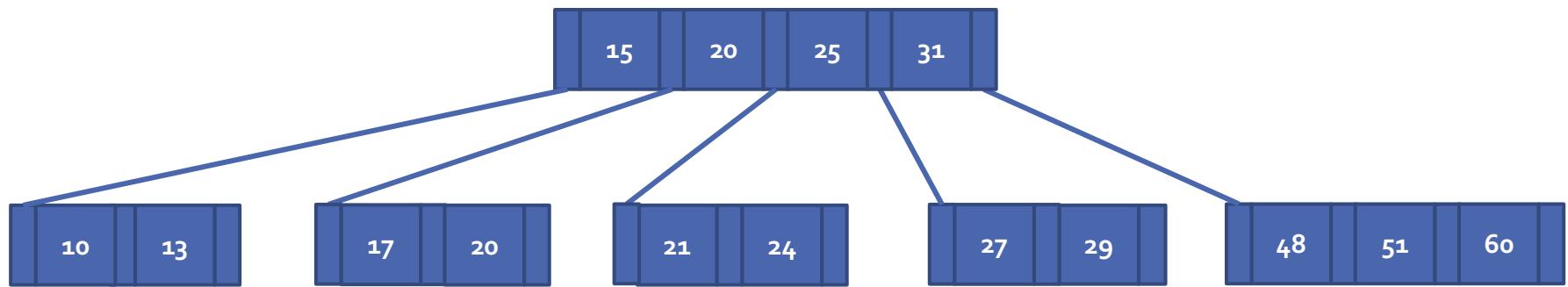
14) Se inserta la clave 19.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

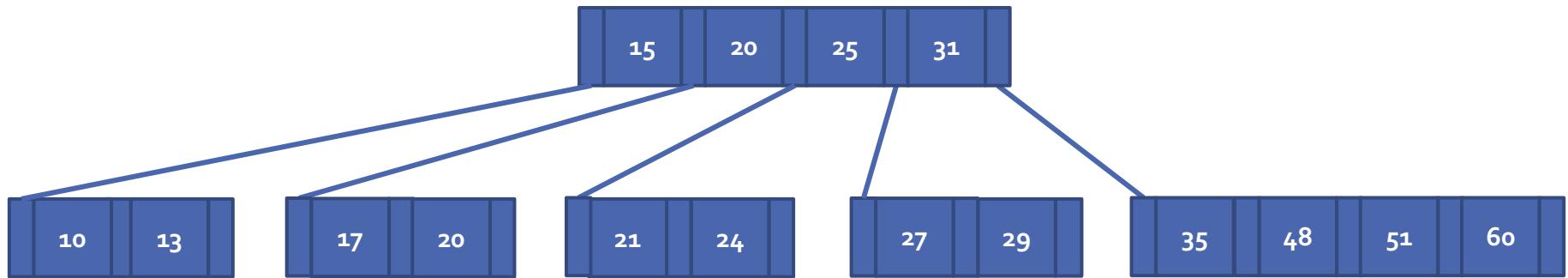
## Ejemplo

15) Se inserta la clave 60.



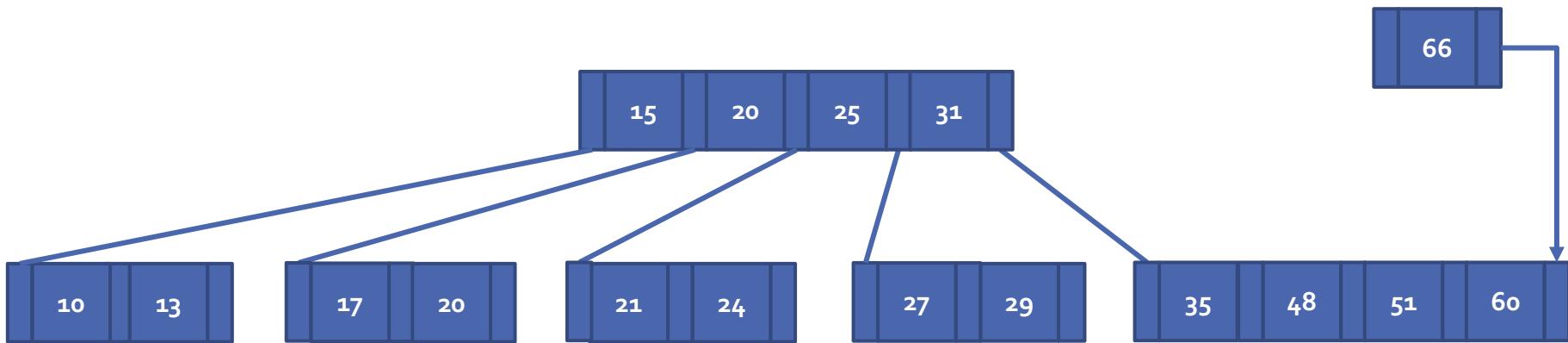
## Ejemplo

16) Se inserta la clave 35.



## Ejemplo

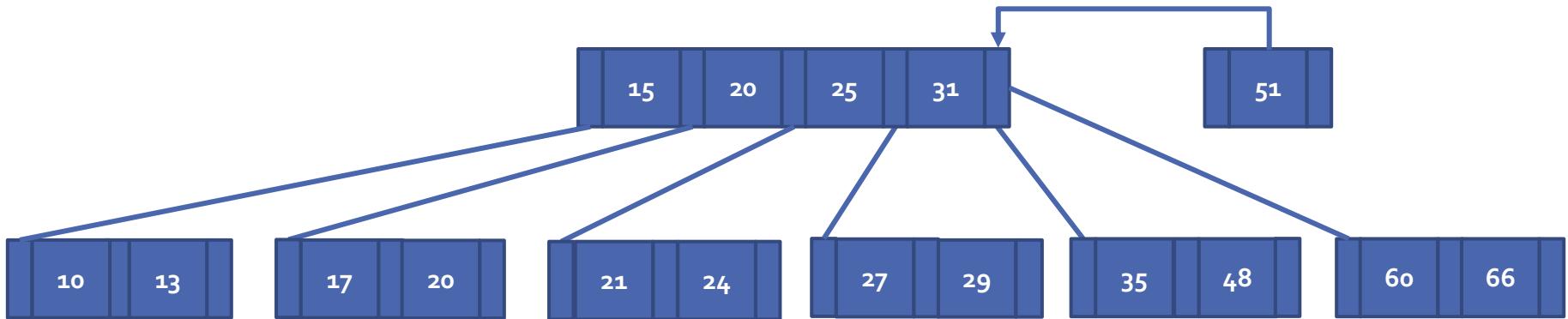
17) Se inserta la clave 66.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

## Ejemplo

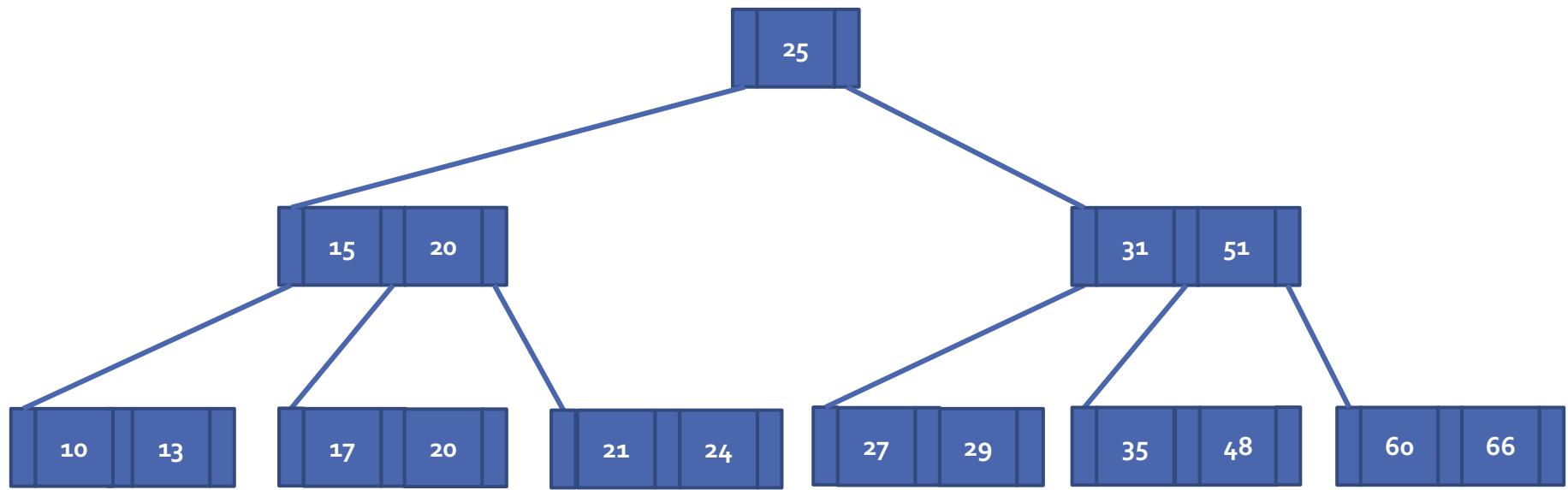
17) Se inserta la clave 66.



Debido a que el número de elementos de la página ( $m$ ) es igual a  $2*d$ , la página afectada se divide en dos y se distribuyen las  $m+1$  claves de manera equitativa. La clave de en medio sube a la página antecesora.

## Ejemplo

17) Se inserta la clave 66.



## **Eliminación en un árbol B.**

Cuando se elimina un elemento de un árbol B, se debe tener cuidado de no violar la condición del número de elementos de la página del árbol, es decir, por página debe haber entre  $d$  y  $2*d$  elementos (con excepción del nodo raíz). Por tanto, es una operación más complicada que la inserción.

Para eliminar un nodo en un árbol B se siguen los siguientes pasos:

1. Si el elemento a eliminar se encuentra en una página hoja:

Si el número de elementos de la página ( $m$ ) es mayor a d

Se elimina el elemento

De lo contrario

Se baja el elemento adyacente de la página antecesora y se sustituye por el nodo que se encuentre más a la derecha en el subárbol izquierdo o por el nodo que se encuentre más a la izquierda en el subárbol derecho.

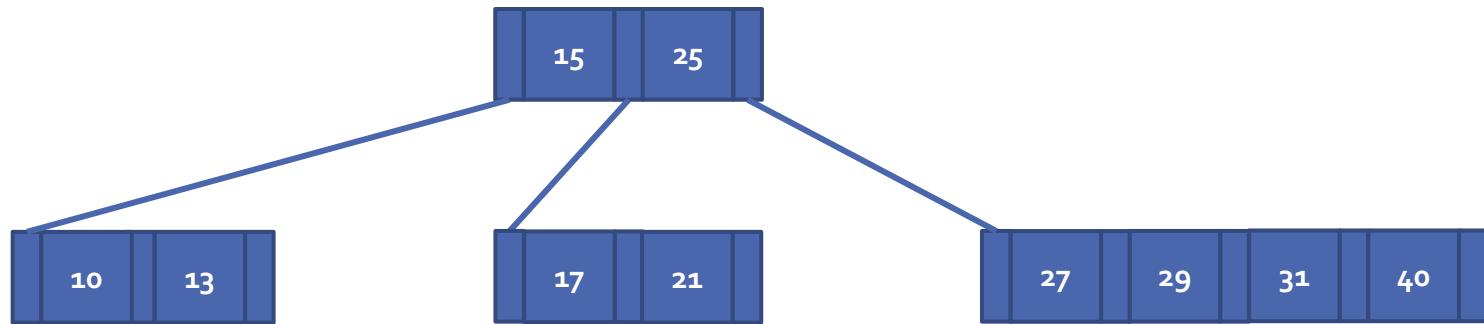
Si lo anterior no se puede hacer, se deben unirlas páginas que son descendentes directos del elemento que se baja.

2. Si el elemento a eliminar no se encuentra en una página hoja

Se sustituye por el nodo que se encuentra más a la izquierda del subárbol derecho o por el nodo que se encuentra más a la derecha del subárbol izquierdo.  
(Se debe mantener la relación  $d \leq m \leq 2*d$ )

## Ejemplo

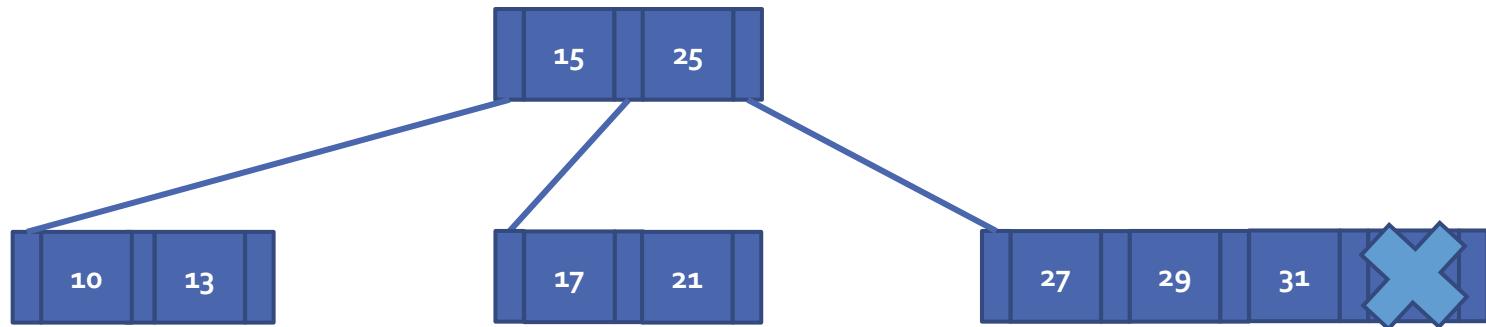
Dado el siguiente árbol T:



Eliminar los nodos: 40 - 21 - 10 - 27.

## Ejemplo

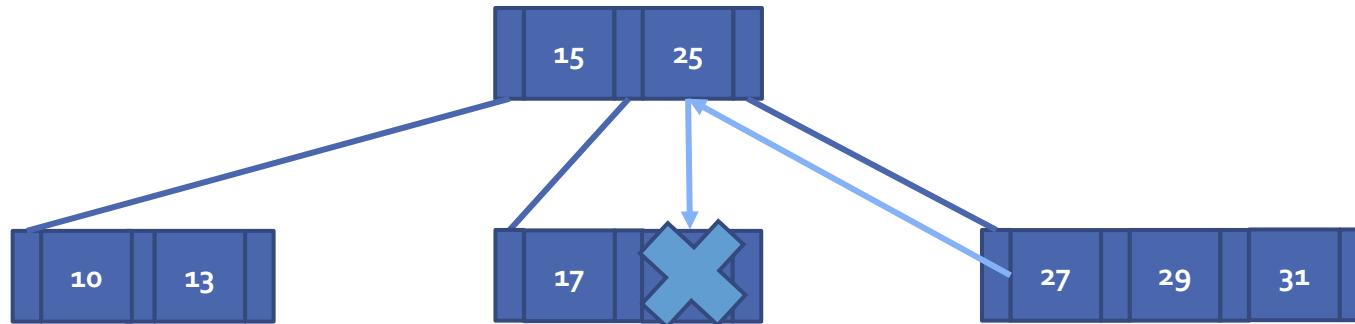
- 1) Eliminar el nodo con clave 40:



Debido a que el nodo es una hoja y el tamaño de la página m es mayor o igual d, éste simplemente se elimina.

## Ejemplo

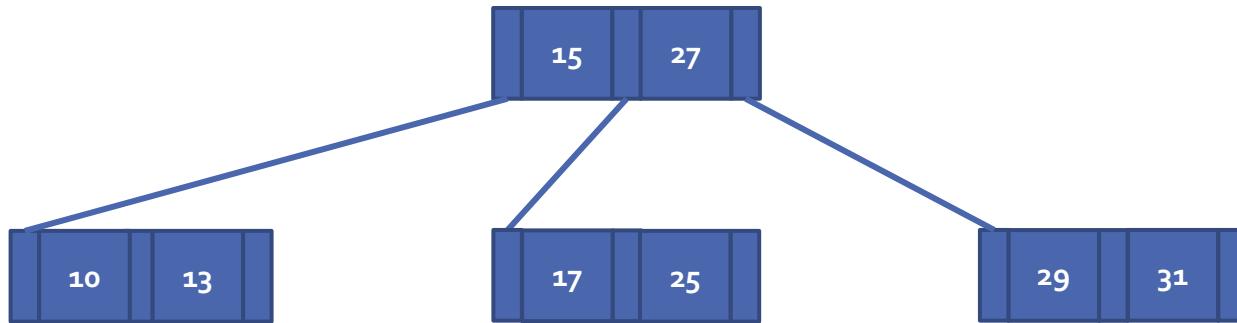
2) Eliminar el nodo con clave 21:



Debido a que el nodo es una hoja, éste se elimina, pero el tamaño de la página  $m$  es menor a  $d$ , por tanto, se baja el elemento adyacente de la página antecesora y se sustituye por la clave que se encuentre más a la derecha en el subárbol izquierdo o la que se encuentre más a la izquierda en el subárbol derecho.

## Ejemplo

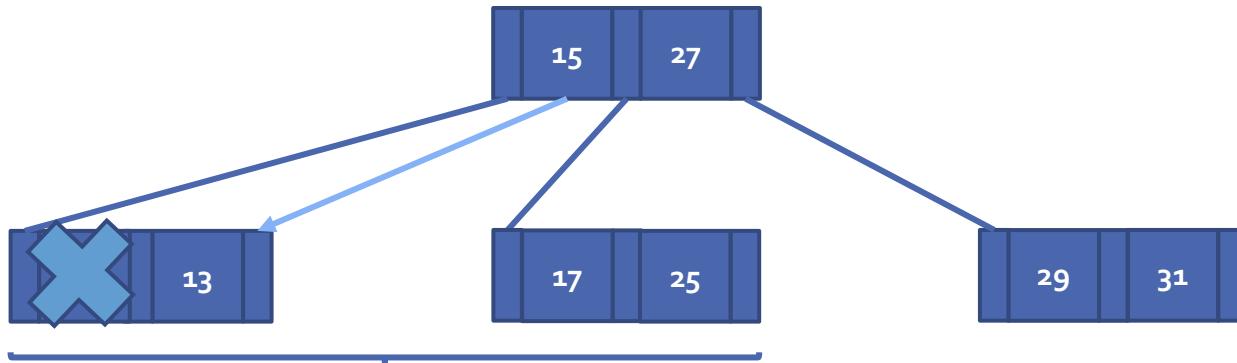
2) Eliminar el nodo con clave 21:



Debido a que el nodo es una hoja, éste se elimina, pero el tamaño de la página m es menor a d, por tanto, se baja el elemento adyacente de la página antecesora y se sustituye por la clave que se encuentre más a la derecha en el subárbol izquierdo o la que se encuentre más a la izquierda en el subárbol derecho.

## Ejemplo

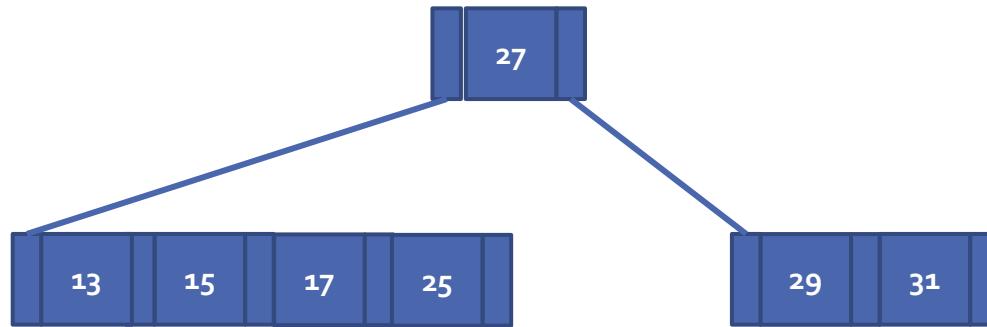
3) Eliminar el nodo con clave 10:



El nodo es una hoja, pero, al eliminar el nodo, m es menor a d, por lo tanto, se deben unir las páginas que son descendientes directos del elemento que se baja.

## Ejemplo

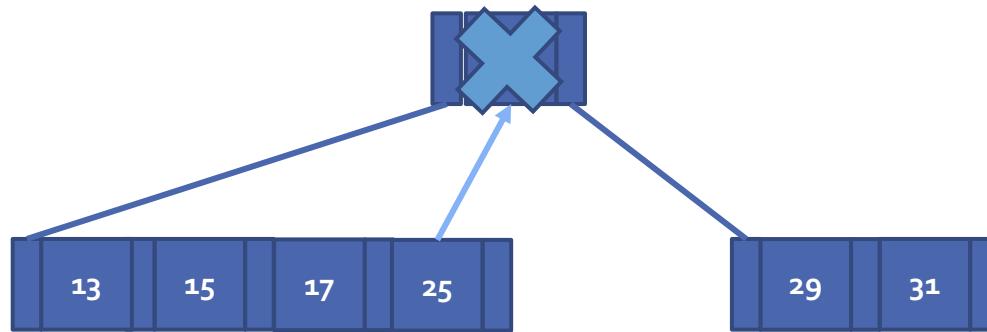
3) Eliminar el nodo con clave 10:



El nodo es una hoja, pero, al eliminar el nodo, m es menor a d, por lo tanto, se deben unir las páginas que son descendentes directos del elemento que se baja.

## Ejemplo

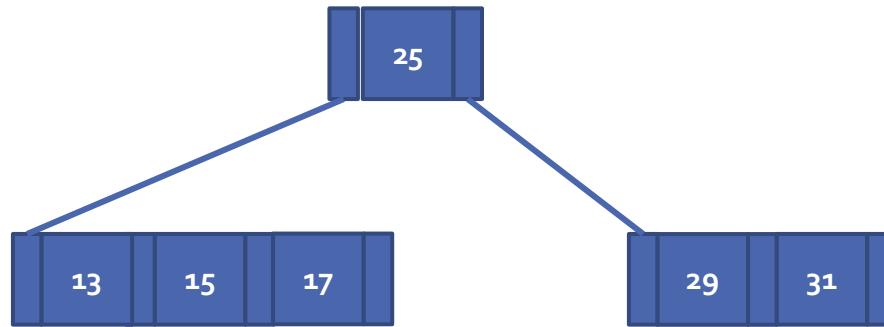
4) Eliminar el nodo con clave 27:



Debido a que el elemento a eliminar no se encuentra en una página hoja, se sustituye por el elemento que se encuentra más a la izquierda del subárbol derecho o por el elemento que se encuentra más a la derecha del subárbol izquierdo.

## Ejemplo

4) Eliminar el nodo con clave 27:



Debido a que el elemento a eliminar no se encuentra en una página hoja, se sustituye por el elemento que se encuentra más a la izquierda del subárbol derecho o por el elemento que se encuentra más a la derecha del subárbol izquierdo.

### **4.3.3 Árboles B +, prefijos simples, algoritmos.**

Los árboles B+ son una variante de los árboles B, con la diferencia de que en los árboles B+ toda la información se encuentra almacenada en las hojas. En la raíz y en las páginas internas se almacenan índices o claves para llegar a los datos en las hojas.

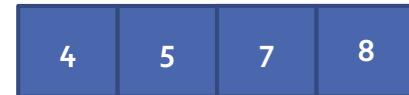
El árbol B+ combina la estructura de un árbol B con un conjunto secuencial, permitiendo el acceso a la información tanto por referencia como de manera secuencial.

Las hojas están formadas por un conjunto secuencial, esto es, un bloque de tamaño fijo que contiene un número máximo de elementos ordenados con base en una clave. Además, cada nodo cuenta con una referencia al elemento siguiente y una referencia al elemento anterior (como una estructura de datos lineal).

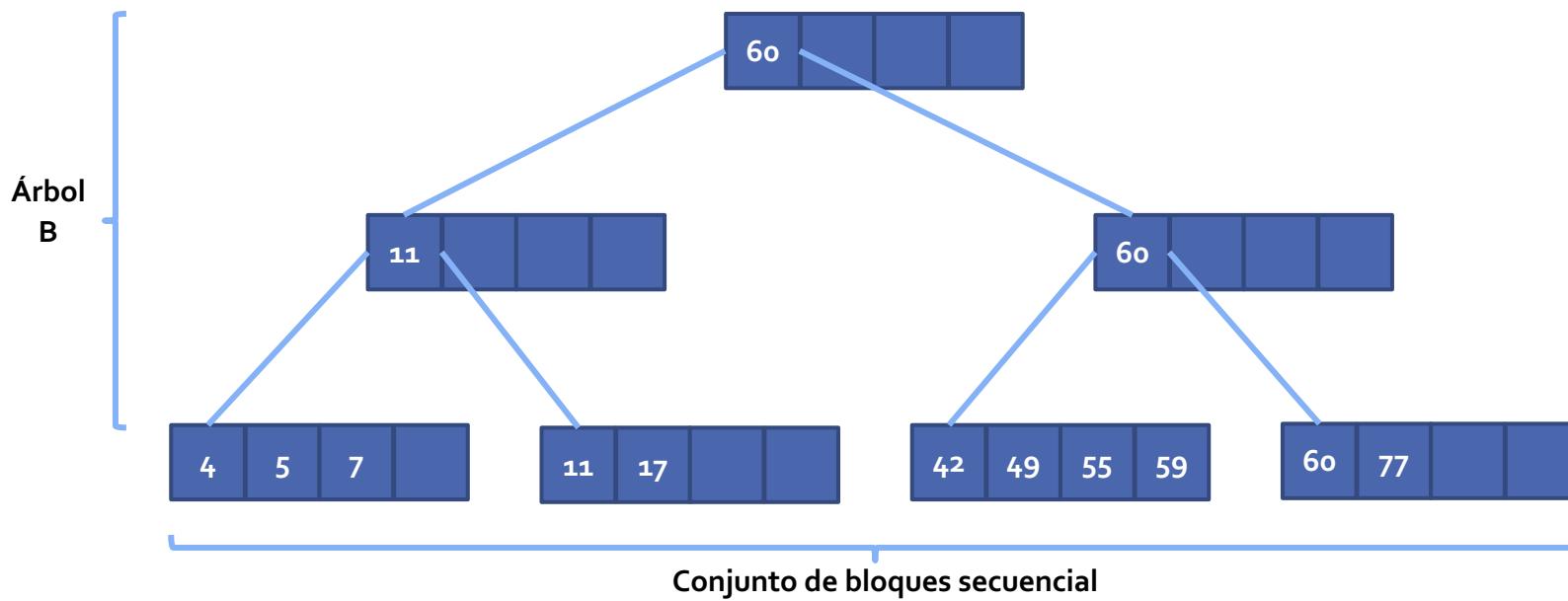


Los bloques se trabajan de manera idéntica a como se realiza en los árboles B de tal manera que siempre se debe cumplir  $d < m < 2*d$ .

Esta estructura permite recorrer secuencialmente todas las claves por bloque.



Sobre el conjunto de bloques secuencial se construye un conjunto indexado (árbol B), donde cada nodo interno o raíz del árbol puede contener un separador que permite saber qué ruta seguir para encontrar el elemento deseado.

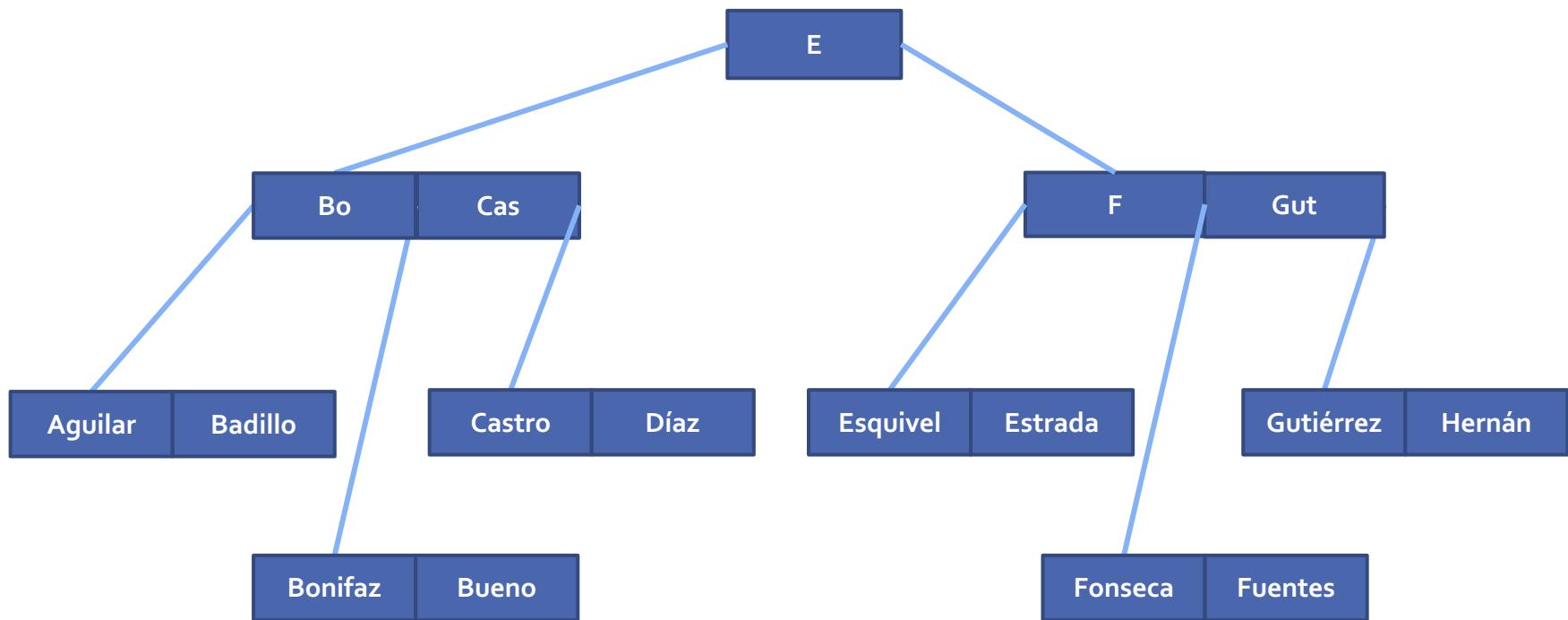


## Prefijos simples

Los separadores del árbol pueden ser una clave o un prefijo que permita hacer la separación. A los árboles B+ que guardan prefijos para separar en vez de claves completas se les llama árboles B+ de prefijo simple y utilizan un algoritmo para determinar el prefijo más corto que pueda usarse como separador en cada caso.

## Ejemplo

Un árbol B+ de prefijo simple.



## **Inserción en un árbol B+**

Para realizar una inserción se debe identificar el bloque donde se desea ingresar el dato, para ello se busca el bloque a través del árbol B.

Si el bloque encontrado no está lleno, se inserta la clave y se verifica que el separador siga siendo válido, si no es así se actualiza (cuando se inserta la clave al principio de un bloque).

Si el bloque está lleno se debe realizar una separación en la cual no se promueven claves si no que todas se distribuyen en los dos bloques resultantes. Una vez distribuidas las claves se selecciona un separador y éste es el que se promueve como índice.

Este proceso se repite hasta que el separador no genere un sobre cupo en el bloque interno, es decir, mientras el nodo superior no esté lleno ( $2^*d$ ).

## **Eliminación en un árbol B+**

La eliminación se realiza sobre un bloque secuencial. Si al hacer una baja el bloque es menor a d, se intenta redistribuir las claves con los bloques vecinos y, si se puede redistribuir, se actualiza el separador de bloques.

Si no se puede redistribuir porque los bloques contiguos tienen la cantidad mínima de claves se hace una concatenación y se elimina del árbol el separador actual de los bloques, este proceso se sigue hacia arriba hasta que todos los nodos cumplan con la restricción  $d < m < 2*d$ .

Si al eliminar un elemento, su clave se encuentra dentro del árbol B, no es necesario eliminarla, ya que el índice solo actúa como separador de elementos.

## **Python orientado a objetos**

En Python, una clase se define de la siguiente manera:

```
class Nombre:  
    # comentarios  
    # atributos  
    # métodos
```

## **self**

La palabra reservada `self` hace referencia al objeto que está en ejecución. Se utiliza para acceder a los atributos y métodos del objeto.

Además, debe ser el primer parámetro de todos los métodos, es decir, siempre se debe incluir en la definición. Sin embargo, al llamar a la función no se debe colocar el objeto, ya que éste se agrega automáticamente.

## Ejemplo

```
# -*- coding: utf-8 -*-

class A:
    i = 5

    def getI(self):
        print ("self.i: ", self.i)

one = A()
one.getI()
```

## **init**

El método `__init__` permite inicializar los atributos de una instancia. Este método es lo que se conoce en la programación orientada a objetos como el método constructor.

`__init__` soporta cualquier número de parámetros, siendo el primero `self`. Este método se ejecuta siempre que se crea una objeto de la clase (implícitamente).

## Ejemplo

```
# -*- coding: utf-8 -*-

class Human(object):
    def __init__(self, age, height, weight):
        self.age = age
        self.height = height
        self.weight = weight

h1 = Human(12, 1.60, 50)
print ("Human:\nEdad = %i, Altura = %.2f, Peso = %i "
       %(h1.age, h1.height, h1.weight))
```

## **str**

El método str se hereda de la súper clase object y, por defecto, imprime la localidad de memoria donde se guarda la referencia del objeto.

Es posible sobrescribir el comportamiento el método str para imprimir el estado (los atributos) del objeto.

## Ejemplo

```
# -*- coding: utf-8 -*-

class Restaurant(object):
    def __init__(self, cad_nombre, cad_propietario, cad_chef):
        self.nombre = cad_nombre
        self.propietario = cad_propietario
        self.chef = cad_chef
        self.__dishes = 15

    def __str__(self):
        return (self.nombre + " (Propietario: "
               + self.propietario + ", Chef: " + self.chef + ")")

mcAlgo = Restaurant("Hamburguesas McAlgo", "Josue", "Alison")
print(mcAlgo)
print()

# dir muestra la lista de métodos que posee el objeto
print(dir(mcAlgo))
```

## **Parámetros por defecto**

Si al crear un objeto o llamar un método se omite algún parámetro del, la ejecución del programa producirá un error.

Python permite inicializar los valores que recibe un método cuando éstos no son pasados como parámetros.

## Ejemplo

```
class Saucer(object):
    def __init__(self, cadNombre, realPrecio, cadDescription=None,
                 cadImagen=None, boolVegetariano=False, entCoccion=1):
        self.nombre = cadNombre
        self.precio = realPrecio
        self.descripcion = cadDescription
        self.imagen = cadImagen
        self.esVegetariano = boolVegetariano
        self.coccion = entCoccion

    def __str__(self):
        return "{nombre}{esVeg}: {precio:.2f}{desc}".format(
            nombre=self.nombre,
            desc=' (' + self.descripcion + ')' if self.descripcion else '',
            precio=self.precio, esVeg='*' if self.esVegetariano else "")

burgerPython = Saucer("Hamburguesa de Python", 0.13,
                      cadDescription="Barely an eigth of a byte")
print(burgerPython)
```

## B-TREE (coding)

Para crear un árbol B se debe modelar el elemento básico del árbol, es decir, el nodo página. Un nodo página consta de un conjunto de hijos y un conjunto de valores.

Para crear un árbol B se requiere una raíz, por lo tanto, el método CREATE\_B\_TREE crea la raíz del árbol. Una vez creado el árbol, es posible insertar datos a través de la función INSERT\_B\_TREE. Ambas funciones requieren reservar en memoria un nodo página, lo que toma  $O(1)$ .

Para insertar valores en un árbol B, a diferencia de en un árbol binario, no se puede insertar directamente como nodo hoja, ya que se debe validar que la página no rompa la regla del máximo número de elementos que puede guardar. Por lo tanto, se requiere una función SPLIT\_B\_TREE que permita separar una página llena.

Independientemente de que exista o no separación, al final se debe insertar el valor en una página que no está llena, para ello se manda llamar al método INSERT\_NON\_FULL\_B\_TREE.

## Ejemplo

```
class Node:  
    def __init__(self, d):  
        self.sons = list()  
        self.keys = list()  
        self.leaf = 1  
        self.n = 0  
        for k in range(2*d):  
            self.keys.append([None])  
        for k in range(2*d + 1):  
            self.sons.append([None])
```

## Ejemplo

```
class b_tree:  
    def __init__(self, min_grade):  
        self.d = min_grade  
        self.root = None  
  
    def create_b_tree(self):  
        if self.root == None:  
            self.root = Node(self.d)  
        return self.root
```

## Ejemplo

```
def split_b_tree(self, x, i):
    z = Node(self.d)
    y = x.sons[i]
    z.leaf = y.leaf
    z.n = self.d-1

    for j in range(1, self.d):
        z.keys[j] = y.keys[j+self.d]
        y.keys[j+self.d] = None

    if y.leaf == 0:
        for j in range(1, self.d + 1):
            z.sons[j] = y.sons[j + self.d]
            y.sons[j + self.d] = None
```

## Ejemplo

```
y.n = self.d - 1
```

```
for j in range(x.n + 1, i, -1):
    x.sons[j+1] = x.sons[j]
```

```
x.sons[i+1] = z
```

```
for j in range (x.n, i-1, -1):
    x.keys[j+1] = x.keys[j]
```

```
x.keys[i] = y.keys[self.d]
```

```
y.keys[self.d] = None
```

```
x.n = x.n + 1
```

## Ejemplo

```
def insert_non_full_b_tree(self, x, k):
    i = x.n
    if x.leaf == 1:
        while i >= 1 and k < x.keys[i]:
            x.keys[i+1] = x.keys[i]
            i -= 1
        x.keys[i+1] = k
        x.n = x.n + 1
    else:
        # Not a leaf
        while i >= 1 and k < x.keys[i]:
            i -= 1
        i += 1
        if x.sons[i].n == 2*self.d - 1:
            self.split_b_tree(x, i)
            if k > x.keys[i]:
                i += 1
        self.insert_non_full_b_tree(x.sons[i], k)
```

## Ejemplo

```
def insert_b_tree(self, k):
    r = self.root
    # Full node
    if r.n == 2*self.d - 1:
        s = Node(self.d)
        self.root = s
        s.leaf = 0
        s.n = 0
        s.sons[1] = r
        self.split_b_tree(s, 1)
        self.insert_non_full_b_tree(s, k)
    else:
        self.insert_non_full_b_tree(r, k)
```

## Ejemplo

```
def print_node(self, node):
    string = ""
    for i in range(1, 2*self.d, 1):
        if node.keys[i] != None:
            if type(node.keys[i]) is int:
                string += str(chr(node.keys[i])) + "\t"
            else:
                string += str(node.keys[i]) + "\t"
    print(string)
```

## Ejemplo

```
print("Insert B")
bt.insert_b_tree(ord("B"))
print("Insert T")
bt.insert_b_tree(ord("T"))
print("Insert H")
bt.insert_b_tree(ord("H"))

print ("Root")
bt.print_node(bt.root)

# After insert 'M', root node is split
print("Insert M")
bt.insert_b_tree(ord("M"))

print("root")
bt.print_node(bt.root)
bt.print_node(bt.root.sons[1])
bt.print_node(bt.root.sons[2])
```

# ÁRBOLES. PARTE II.

---

Práctica 9

# 9. Árboles. Parte II.

- Implementar en lenguaje Python el programa propuesto de la práctica 9.
- Crear una función que permita mostrar el árbol creado a partir del código del punto anterior.
- Graficar la complejidad que tiene el algoritmo de inserción en un árbol B.

# 4. Árboles

Objetivo: Aplicar las formas de representar y operar las listas lineales para representarlos en la computadora.

4.1 Notaciones: infija, prefija, sufija.

4.2 Árboles binarios.

    4.2.1 Definiciones y operaciones.

    4.2.2 Transformación de árboles a árboles binarios.

    4.2.3 Recorrido de árboles.

    4.2.4 Representación en la computadora.

4.3 Árboles B.

    4.3.1 Árboles B.

    4.3.2 Árboles B+, algoritmos.

    4.3.3 Árboles B+ prefijos simples, algoritmos.