



UNIVERSIDAD DE CARABOBO  
FACULTAD EXPERIMENTAL DE CIENCIAS Y TECNOLOGÍA  
DEPARTAMENTO DE COMPUTACIÓN  
CS218 – ALGORITMOS Y PROGRAMACIÓN II

## **TEMA 7**

### **ESTRUCTURAS JERÁRQUICAS: ÁRBOLES**

Mayo, 2005

## 1.- Árboles N-arios: Definiciones y Conceptos Básicos

Algunas veces, cuando se modela una situación, se determina que la mejor manera de representar la relación que existe entre los objetos involucrados es mediante una estructura jerárquica. Algunos ejemplos típicos de esta situación son los siguientes:

- Organigrama: Describe la organización administrativa (estructura) de una empresa a través de la relación entre las diferentes unidades de trabajo.
- Índice de un libro: Enumeración ordenada del contenido de un libro.
- Árboles genealógicos: Representación gráfica de las ancestros y descendientes de una persona.

### Definición: (Árbol N-ario no ordenado con tipo base T)

Formalmente, un árbol N-ario no ordenado con tipo base T se puede definir de manera recursiva como sigue:

1. La secuencia vacía representa al árbol nulo no ordenado. Se denota por  $\Lambda$ .
2. Sea  $e$  un elemento de tipo T y  $A_1, A_2, \dots, A_N$  árboles N-arios no ordenados con tipo base T. Entonces la secuencia  $A = \langle e, [A_1, A_2, \dots, A_N] \rangle$ , donde  $[A_1, A_2, \dots, A_N]$  denota un **multiconjunto** de árboles N-arios, es un árbol N-ario no ordenado con tipo base T y se definen las siguientes relaciones:
  - $e$  es la raíz de  $A$ .
  - $\forall i (1 \leq i \leq N) (A_i \text{ es un subárbol de } A)$
  - $\forall i (1 \leq i \leq N) (e \text{ es padre de la raíz de } A_i)$
  - $\forall i (1 \leq i \leq N) (\text{la raíz de } A_i \text{ es hijo de } e)$
3. Sólo son árboles N-arios no ordenados con tipo base T los objetos generados por la aplicación de las cláusulas (1) y (2).

**Grado de un nodo:** Es el número de hijos (subárboles) de ese nodo.

**Grado de un árbol:** Es el grado del nodo de mayor grado en el árbol.

**Hoja o nodo terminal:** Es un nodo de grado 0.

**Nodo no terminal o interior:** Es todo nodo de un árbol que no es una hoja.

**Nodos hermanos:** Dos nodos  $e_1$  y  $e_2$  de un árbol son hermanos si y sólo si tienen el mismo padre.

**Camino entre dos nodos:** Un camino entre dos nodos  $e$  y  $e'$  de un árbol es una secuencia de nodos  $\langle e_1, e_2, \dots, e_n \rangle$  tal que verifica las siguientes propiedades:

- $e_1 = e$
- $e_n = e'$
- $\forall i (1 \leq i < n) (e_i \text{ es padre de } e_{i+1})$

**Longitud de un camino:** La longitud de un camino  $C = \langle e_1, e_2, \dots, e_n \rangle$  se define como el número de veces que se debe aplicar la relación padre  $\rightarrow$  hijo durante el recorrido, es decir, es el número de nodos en el camino menos 1.

**Rama:** Es un camino que parte de la raíz y termina en una hoja.

**Nivel de un nodo:** Es la longitud del camino que parte de la raíz y llega a ese nodo.

**Altura de un nodo:** Es la longitud del camino más largo desde el nodo hasta una hoja.

**Altura de un árbol:** Es la altura de la raíz del árbol. La altura de un árbol nulo se define como  $-1$ .

**Peso de un árbol:** Es el número de nodos que contiene el árbol. El peso de un árbol nulo es 0.

## 2.- Class ArbolN

Para el desarrollo de la **Class ArbolN**, se supone que los elementos almacenados en los nodos o vértices son de un tipo genérico que llamaremos **Elemento**.

### Class ArbolN<Elemento>

#### ESPECIFICACIÓN OPERACIONAL

##### Sintaxis

##### Class ArbolN <Elemento>

nulo <sub>ArbolN</sub> :		→	ArbolN
es_nulo <sub>ArbolN</sub> :	ArbolN	→	Booleano
crear <sub>ArbolN</sub> :	Elemento × Lista[ ArbolN ]	→	ArbolN
raíz <sub>ArbolN</sub> :	ArbolN	→	Elemento
Hijos :	ArbolN	→	Lista[ ArbolN ]
ins_subárbol :	ArbolN × ArbolN	→	ArbolN
Elim._subárbol :	ArbolN × Natural	→	ArbolN

##### Semántica

Por convención se denota como  $A'$  el árbol  $A$  después de aplicar la operación.

{ Pre: }

**proc** nulo<sub>ArbolN</sub> (**var** A: ArbolN )

{ Post:  $A' \leftarrow \Lambda$  }

{ Pre: }

**func** es\_nulo<sub>ArbolN</sub> (A : ArbolN) : Booleano;

{ Post: es\_nulo<sub>ArbolN</sub>  $\leftarrow (A = \Lambda)$  }

{ Pre: }

**proc** crear<sub>ArbolN</sub> (**var** A: ArbolN; e : Elemento; lst : Lista[ ArbolN ])

{ Post: si lst =  $\langle A_1, A_2, \dots, A_N \rangle$  entonces  $A' \leftarrow \langle e, \langle A_1, A_2, \dots, A_N \rangle \rangle$  }

{ Pre:  $\neg \text{es\_nulo}_{\text{ArbolN}}(A)$  }

**func** raíz<sub>ArbolN</sub> (A : ArbolN) : Elemento;

{ Post: si A =  $\langle e, \langle A_1, A_2, \dots, A_N \rangle \rangle$  entonces raíz<sub>ArbolN</sub>  $\leftarrow e$  }

{ Pre:  $\neg \text{es\_nulo}_{\text{ArbolN}}(A)$  }

**func** hijos (A : ArbolN) : Lista[ ArbolN ];

{ Post: si A =  $\langle e, \langle A_1, A_2, \dots, A_N \rangle \rangle$  entonces hijos  $\leftarrow \langle A_1, A_2, \dots, A_N \rangle$  donde  
 $\forall i (1 \leq i \leq n)(A_i \text{ es el } i\text{-ésimo sub-árbol de } A)$  }

```

{ Pre:  $\neg \text{es\_nulo}_{\text{ArbolN}}(A) \wedge \neg \text{es\_nulo}_{\text{ArbolN}}(B)$  }
    proc ins_subárbol (var A: ArbolN; B : ArbolN)
{ Post: si  $A = \langle e, \langle A_1, A_2, \dots, A_N \rangle \rangle$  entonces  $A' \leftarrow \langle e, \langle A_1, A_2, \dots, A_N, B \rangle \rangle$  }

{ Pre:  $\neg \text{es\_nulo}_{\text{ArbolN}}(A) \wedge (1 \leq \text{pos} \leq \text{longitud}(\text{hijos}(A)))$  }
    proc elim_subárbol (var A: ArbolN; pos: entero)
{ Post: si  $A = \langle e, \langle A_1, A_2, \dots, A_{\text{pos}-1}, A_{\text{pos}}, A_{\text{pos}+1}, \dots, A_N \rangle \rangle$  entonces
     $A' \leftarrow \langle e, \langle A_1, A_2, \dots, A_{\text{pos}-1}, A_{\text{pos}+1}, \dots, A_N \rangle \rangle$  }

```

## 2.2.- IMPLEMENTACIÓN

En esta sección se presentan algunas de las representaciones más utilizadas para implementar árboles N-arios, con sus respectivos algoritmos.

En cada una de las representaciones se supone que se cumplen todas las precondiciones planteadas en la especificación pre / post de cada operación.

### 2.2.1.- Representación hijo–izquierdo hermano–derecho

En esta implementación, el árbol está representado por un apuntador a su nodo raíz y cada nodo tiene un apuntador a su primer hijo (hijo más a la izquierda) y a su hermano derecho.

```
class nodo
begin
  private:
    elem      : elemento;
    hijo_izq  : pointer to nodo;
    hno_der   : pointer to nodo;

  public:
    nodo();
    ~nodo();

    func get_elem() : elemento;
    func get_hijo_izq() : pointer to nodo;
    func get_hno_der() : pointer to nodo;

    proc set_elem(in e : elemento);
    proc set_hijo_izq(in p : pointer to nodo);
    proc set_hno_der(in p : pointer to nodo);
end

class ArbolN
begin
  private:
    arbn : pointer to nodo;

  public:
    ArbolN();
    ~ArbolN();

    func es_nulo() : logical;
    func raíz ()   : elemento;
    func hijos()   : lista<ArbolN>;

    proc crear(in e: elemento; in lst: lista<ArbolN>);
    proc ins_subárbol(in a : ArbolN);
    proc elim_subárbol (in pos : integer);
    proc destruir_árbol();
end
```

### 2.2.2.- Representación con vector de apuntadores

```
const
    MAX_HIJOS = ?;          (Constante predefinida)

class nodo
begin
    private:
        elem          : elemento;
        sub_arboles   : array [1 .. MAX_HIJOS] of pointer to nodo;

    public:
        nodo();
        ~nodo();
        func get_elem() : elemento;
        func get_sub_arbol(in pos) : pointer to nodo;

        proc set_elem(in e : elemento);
        proc set_sub_arbol(in p : pointer to nodo);
end

class ArbolN
begin
    private:
        arbn : pointer to nodo;

    public:
        ArbolN();
        ~ArbolN();

        func es_nulo() : logical;
        func raíz ()   : elemento;
        func hijos()   : lista<ArbolN>;

        proc crear(in e: elemento; in lst: lista<ArbolN>);
        proc ins_subárbol(in a : ArbolN);
        proc elim_subárbol (in pos : integer);
        proc destruir_árbol();
end
```

### 3.- Recorrido de árboles n-arios

Entendemos por recorrer un árbol visitar (y procesar) cada uno de los nodos que lo componen, una sola vez y en un determinado orden. Existen 4 métodos básicos, los cuales difieren en el orden en que procesamos los nodos:

#### Recorrido en preorden:

- i. Visitar la raíz (y procesarla).
- ii. Recorrer cada uno de los hijos en preorden

#### Recorrido en inorden:

- i. Recorrer el primer hijo en inorden.
- ii. Visitar la raíz (y procesarla)
- iii. Recorrer el resto de los hijos en inorden

#### Recorrido en postorden:

- i. Recorrer cada uno de los hijos en postorden
- ii. Visitar la raíz (y procesarla)

#### Recorrido por niveles:

para  $i \leftarrow 1$  hasta num\_niveles\_arbol hacer  
    recorrer nodos del nivel  $i$  partiendo del nodo más a la izquierda

fpara

#### 4.- Árboles binarios: Definiciones y Conceptos Básicos

**Árbol completo:** Un árbol binario es completo si y sólo si todo nodo no terminal tiene exactamente dos subárboles no vacíos.

**Árbol lleno:** Un árbol binario está lleno si y sólo si es completo y, además, todas las hojas están en el mismo nivel.

**Árbol casi lleno:** Un árbol binario está casi lleno si y sólo si está lleno hasta el penúltimo nivel y todas las hojas del último nivel están tan a la izquierda como es posible.

**Árboles iguales:** Dos árboles binarios son iguales si y sólo si ambos son nulos, o si sus raíces son iguales, lo mismo que sus respectivos subárboles izquierdo y derecho.

**Árboles isomorfos:** Dos árboles binarios son isomorfos si y sólo si tienen la misma estructura pero no necesariamente los mismos elementos.

**Árboles semejantes:** Dos árboles binarios son semejantes si y sólo si contienen los mismos elementos, aunque no sean isomorfos.

**Ocurrencia de un árbol binario en otro:** Un árbol binario  $A_1$  ocurre en otro árbol binario  $A_2$  si y sólo si  $A_1$  y  $A_2$  son iguales o si  $A_1$  es igual a alguno de los subárboles de  $A_2$ .



## 5.- Class ArBin

### ESPECIFICACIÓN OPERACIONAL

#### Sintaxis

**Class** ArBin [ Elemento ]

nulo <sub>ArBin</sub> :		→	ArBin
es_nulo <sub>ArBin</sub> :	ArBin	→	Booleano
crear <sub>ArBin</sub> :	Elemento × ArBin × ArBin	→	ArBin
raíz <sub>ArBin</sub> :	ArBin	→	Elemento
hijo_der :	ArBin	→	ArBin
hijo_izq :	ArBin	→	ArBin

#### Semántica

Por convención se denota como  $A'$  el árbol  $A$  después de aplicar la operación.

{ Pre: }  
**proc** nulo<sub>ArBin</sub>(var A: ArBin)  
{ Post:  $A' \leftarrow \Lambda$  }

{ Pre: }  
**func** es\_nulo<sub>ArBin</sub>(A: ArBin): Booleano;  
{ Post: es\_nulo<sub>ArBin</sub>  $\leftarrow (A = \Lambda)$  }

{ Pre: }  
**proc** crear<sub>ArBin</sub>(**var** A: ArBin; e: Elemento; A<sub>1</sub>, A<sub>2</sub>: ArBin)  
{ Post:  $A' \leftarrow \langle e, A_1, A_2 \rangle$  }

{ Pre:  $\neg$ es\_nulo<sub>ArBin</sub>(A) }  
**func** raíz<sub>ArBin</sub>(A: ArBin): Elemento;  
{ Post: si  $A = \langle e, A_1, A_2 \rangle$  entonces raíz<sub>ArBin</sub>  $\leftarrow e$  }

{ Pre:  $\neg$ es\_nulo<sub>ArBin</sub>(A) }  
**func** hijo\_der(A: ArBin): ArBin;  
{ Post: si  $A = \langle e, A_1, A_2 \rangle$  entonces hijo\_der  $\leftarrow A_2$  }

{ Pre:  $\neg$ es\_nulo<sub>ArBin</sub>(A) }  
**func** hijo\_izq(A: ArBin): ArBin;  
{ Post: si  $A = \langle e, A_1, A_2 \rangle$  entonces hijo\_izq  $\leftarrow A_1$  }

## 5.2.- IMPLEMENTACIÓN

En esta sección se presentan algunas de las representaciones más utilizadas para implementar árboles binarios, con sus respectivos algoritmos.

En cada una de las representaciones se supone que se cumplen todas las precondiciones planteadas en la especificación pre / post de cada operación.

### 5.2.1.- Árboles sencillamente enlazados (Hijo Izquierdo – Hijo Derecho)

Definición de la estructura de datos que representa a los objetos de la clase.

```
class nodo
begin
  private:
    elem : elemento;
    hijo_izq : pointer to nodo;
    hijo_der : pointer to nodo;

  public:
    nodo();
    ~nodo();

    func get_elem() : elemento;
    func get_hijo_izq() : pointer to nodo;
    func get_hijo_der() : pointer to nodo;

    proc set_elem(in e : elemento);
    proc set_hijo_izq(in p : pointer to nodo);
    proc set_hijo_der(in p : pointer to nodo);
end

class ArBin
begin
  private:
    arb : pointer to nodo;
  public:
    func es_nulo() : logical;
    func raíz () : elemento;
    func get_hijo_izq() : ArBin;
    func get_hijo_der() : ArBin;

    proc crear(in e: elemento; in A1, A2 : ArBin);
end
```

## 6.- Recorrido de árboles binarios

Los esquemas de recorrido para el caso de árboles binarios son los siguientes:

### Recorrido en preorden:

- i. Visitar la raíz (y procesarla)
- ii. Recorrer el subarbol izq en preorden
- iii. Recorrer el subarbol der en preorden

### Recorrido en inorden:

- i. Recorrer el subarbol izq en inorden.
- ii. Visitar la raíz (y procesarla)
- iii. Recorrer el subarbol der en inorden

### Recorrido en postorden:

- i. Recorrer el subarbol izq en postorden.
- ii. Recorrer el subarbol der en postorden
- iii. Visitar la raíz (y procesarla)

### Recorrido por niveles:

para  $i \leftarrow 1$  hasta num\_niveles\_arbol hacer

recorrer nodos del nivel  $i$  partiendo del nodo más a la izquierda

fpara

## 7.- Reconstrucción de un árbol binario a partir de sus recorridos

Si un árbol binario no tiene elementos repetidos, es posible reconstruirlo a partir de las secuencias de vértices producidas por dos de sus recorridos (preorden e inorden o postorden e inorden). El proceso de reconstrucción se muestra en el siguiente ejemplo:

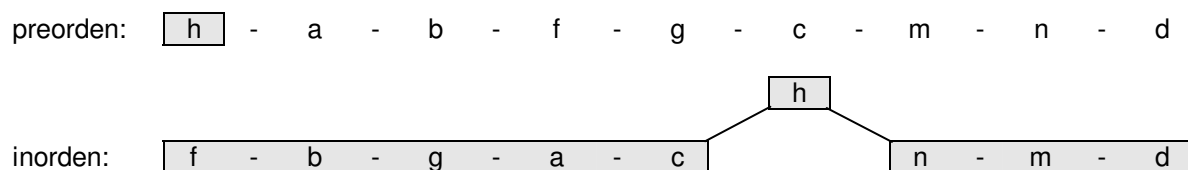
Suponga que se quiere reconstruir el árbol binario sin elementos repetidos cuyos recorridos en preorden e inorden vienen dados por las siguientes secuencias:

preorden: h - a - b - f - g - c - m - n - d  
inorden: f - b - g - a - c - h - n - m - d

### Paso 1:

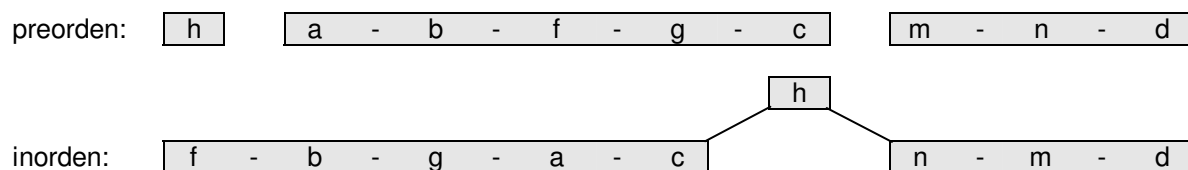
Encontrar la raíz, que siempre es el primer elemento del recorrido en preorden, y localizarla en el recorrido en inorden.

De esta manera, el recorrido en inorden queda dividido en los recorridos en inorden que corresponden a las secuencias en inorden de los dos subárboles del árbol original.



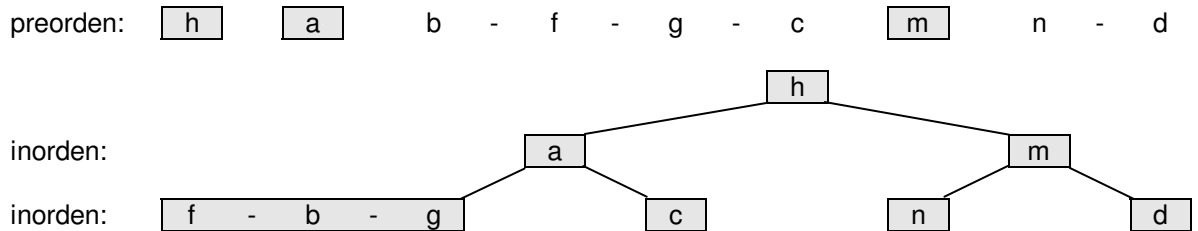
### Paso 2:

Conociendo el número de nodos de cada uno de los subárboles, que se determina por la longitud de las dos subsecuencias obtenidas en el paso anterior, puede extraerse del recorrido en preorden del árbol original, los recorridos en preorden de los subárboles.



### Paso 3:

Repetir los pasos 1 y 2 con cada uno de los subárboles encontrados.



## 8.- Árboles de Sintaxis:

Una aplicación bastante común de los árboles binarios es la representación de expresiones matemáticas. Un árbol de sintaxis es un árbol binario que permita representar expresiones aritméticas

### Expresiones Aritméticas:

Toda expresión aritmética está constituida por operadores y operandos, y representa la manera de obtener un resultado a partir del valor de sus componentes. Cada operador representa una operación aritmética, y tiene asociados dos operandos, los cuales son los valores a los que se les aplicará la operación para obtener un resultado. Este resultado puede ser a su vez, uno de los operandos de otra operación. Los operadores binarios que consideraremos en este tema son los siguientes: suma (+), resta (-), multiplicación (\*) y división (/). Los operandos serán constantes o variables. Las constantes son secuencias de dígitos y las variables son secuencias de letras.

Una expresión aritmética en notación infija puede estar formada por una variable, una constante o dos expresiones aritméticas infijas, cada una de ellas entre paréntesis, con un operador binario entre ellas. Nótese la naturaleza recursiva de esta descripción. En las expresiones infijas los paréntesis son indispensables para evitar ambigüedades en cuanto a la precedencia de los operadores.

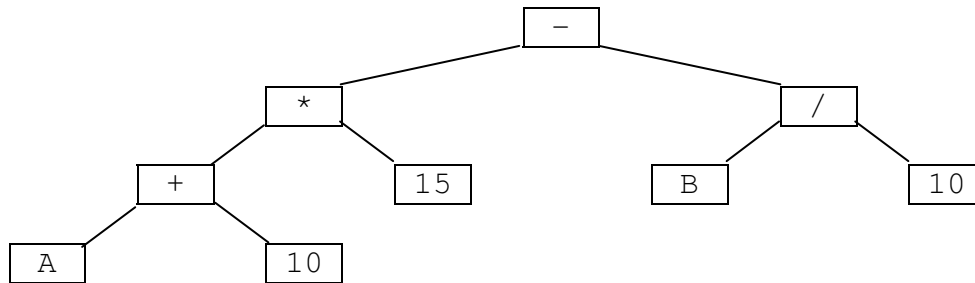
Una expresión aritmética en notación infija se define recursivamente de la siguiente manera:

- Si  $E$  es una constante numérica, entonces  $E$  es una expresión aritmética en notación infija.
- Si  $E$  es una variable numérica, entonces  $E$  es una expresión aritmética en notación infija.
- Si  $E_1$  y  $E_2$  son expresiones aritméticas en notación infija y  $op \in \{+, -, *, /\}$ , entonces  $(E_1 op E_2)$  es una expresión aritmética en notación infija.

C	→	(C * D)
D	→	((C * D) / (5 + G))
5	→	(5 + G)
G		

Un árbol de sintaxis es un árbol binario completo, en el cual los nodos interiores son operadores y las hojas son operandos. Un árbol de sintaxis representa sin ambigüedad, una expresión aritmética. Por ejemplo, si se tiene la expresión  $((A + 10) * 15) - (B / 10)$ , ésta puede ser representada, sin necesidad de los paréntesis,

con un árbol como el que se muestra a continuación, en el cual la relación **padre** → **hijo**, viene dada por la relación **operador** → **operando**, la cual es recursiva.



A partir del árbol de sintaxis asociado a una expresión aritmética, pueden obtenerse las representaciones en notación prefija, postfija e infija de la misma, mediante los recorridos en preorden, postorden e inorden del árbol, respectivamente.

## 9.- Árboles Binarios de Búsqueda

Una de las aplicaciones más frecuentes de los árboles binarios es el almacenamiento de conjuntos de datos entre los cuales puede establecerse una relación de orden. El almacenar datos de manera ordenada permite mejorar la eficiencia de las operaciones de acceso a la información. Si en un árbol binario los elementos están organizados de tal manera que aquellos menores que la raíz se encuentran en el subárbol izquierdo y los elementos mayores que la raíz están en el subárbol derecho, y esta propiedad se mantiene recursivamente en cada subárbol, se tiene una estructura llamada **árbol binario de búsqueda** o **árbol binario ordenado**.

De una manera más formal, se dice que un árbol binario es de búsqueda si todos los elementos del subárbol izquierdo son menores que la raíz, la raíz es menor que todos los elementos del subárbol derecho y los dos subárboles asociados son también árboles binarios de búsqueda. Por definición, el árbol vacío es un árbol binario de búsqueda.

En un árbol con estas características, la búsqueda de un elemento parte de la raíz y prosigue, en cada nodo, por el subárbol izquierdo o el subárbol derecho, de acuerdo al resultado de la inspección del valor almacenado en el nodo.

## 10.- Árboles Binarios Balanceados

Un problema de los árboles binarios de búsqueda es que no es suficiente con mantenerlos ordenados para garantizar la rapidez en el acceso a la información, ya que la complejidad del peor caso en la operación de acceso a la información es  $O(n)$ , a pesar de que la complejidad del caso promedio es  $O(\log_2 n)$ .

Una solución posible consiste en exigir que los dos sub-árboles asociados tengan, aproximadamente, el mismo número de componentes, garantizando de esta manera que, en cada paso de una búsqueda, se descarten aproximadamente la mitad de los elementos que aún no han sido inspeccionados.

Los árboles binarios con esta característica se denominan balanceados y garantizan que la operación de búsqueda de un elemento tiene una complejidad  $O(\log_2 n)$  en el peor caso. Básicamente un árbol binario balanceado es un árbol binario de búsqueda con una condición de equilibrio.

Esta característica tiene, sin embargo, un costo. Los algoritmos de inserción y eliminación sobre árboles balanceados, son más complicados, ya que deben considerar la modificación de la estructura del árbol, de tal manera de garantizar la condición de balance después de cada inserción o eliminación.

Existen dos tipos de árboles binarios balanceados:

- Árboles perfectamente balanceados (o balanceados por peso)
- Árboles AVL (o balanceados por altura)

### Árboles Perfectamente Balanceados

Sea  $A = \langle e, A_1, A_2 \rangle$  un árbol binario de búsqueda. Se dice que  $A$  es un árbol perfectamente balanceado si y sólo si satisface las siguientes condiciones:

- $| \text{peso}(A_1) - \text{peso}(A_2) | \leq 1$  (Condición de Equilibrio)
- $A_1$  y  $A_2$  son árboles perfectamente balanceados

El árbol vacío es, por definición, un árbol perfectamente balanceado.

### Árboles AVL

Adelson – Velskii y Landis introdujeron el 1962 el concepto de árbol balanceado por altura (o árbol AVL por las iniciales de los autores). En este tipo de árboles, las alturas de los dos subárboles asociados a cada elemento no pueden diferir en más de 1, y los dos subárboles deben ser también AVL. Por definición, el árbol vacío es un árbol AVL. Una definición más formal es la siguiente:

Sea  $A = \langle e, A_1, A_2 \rangle$  un árbol binario de búsqueda. Se dice que  $A$  es un árbol AVL si y sólo si satisface las siguientes condiciones:

- $| \text{altura}(A_1) - \text{altura}(A_2) | \leq 1$  (Condición de Equilibrio)
- $A_1$  y  $A_2$  son árboles AVL

Cada vez que se realiza una inserción o una eliminación en un árbol AVL, se debe comprobar que se preserva la condición de equilibrio en cada nodo del árbol AVL. Si la condición de equilibrio no se preserva en alguno de los nodos entonces se debe restablecer el equilibrio del árbol utilizando el proceso de rotación. Existen dos tipos principales de rotación:

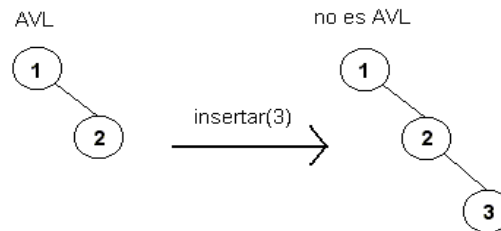


- Rotación Simple
- Rotación Doble

A continuación veremos una descripción detallada del proceso de rotación enmarcada en el proceso de inserción y eliminación de elementos de un árbol AVL.

### Inserción en un AVL

La inserción en un AVL se realiza de la misma forma que en un árbol binario de búsqueda, sin embargo, una vez hecha la inserción, se debe comprobar que se preserve la condición de equilibrio en cada nodo del árbol AVL. El problema potencial que se puede producir después de una inserción es que el árbol con el nuevo nodo no sea AVL:



En el ejemplo de la figura, la condición de balance se pierde al insertar el número 3 en el árbol, por lo que es necesario restaurar de alguna forma dicha condición. Esto siempre es posible de hacer a través de una modificación simple en el árbol, conocida como **rotación**.

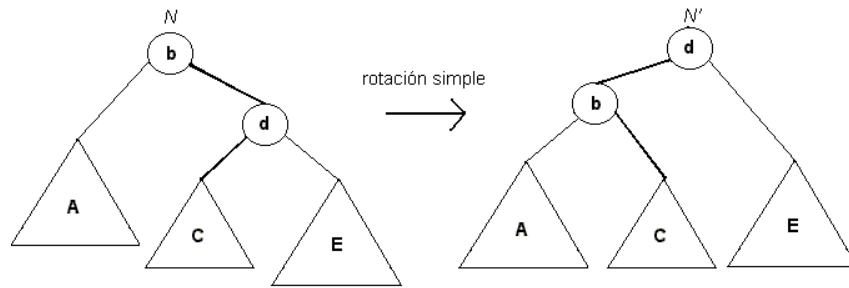
Suponga que después de la inserción de un elemento X el nodo desbalanceado más profundo en el árbol es N. Esto quiere decir que la diferencia de altura entre los dos hijos de N tiene que ser 2, puesto que antes de la inserción el árbol estaba balanceado. La violación del balance pudo ser ocasionada por alguno de los siguientes casos:

- El elemento X fue insertado en el subárbol izquierdo del hijo izquierdo de N.
- El elemento X fue insertado en el subárbol derecho del hijo izquierdo de N.
- El elemento X fue insertado en el subárbol izquierdo del hijo derecho de N.
- El elemento X fue insertado en el subárbol derecho del hijo derecho de N.

Dado que el primer y último caso son simétricos, así como el segundo y el tercero, sólo hay que preocuparse de dos casos principales: una inserción "hacia afuera" con respecto a N (primer y último caso) o una inserción "hacia adentro" con respecto a N (segundo y tercer caso).

### *Rotación simple*

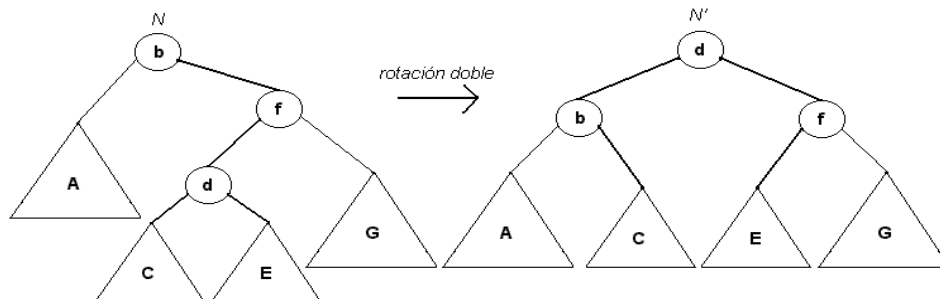
El desbalance por inserción "hacia afuera" con respecto a N se soluciona con una rotación simple.



La figura muestra la situación antes y después de la rotación simple, y ejemplifica el cuarto caso anteriormente descrito, es decir, el elemento  $X$  fue insertado en **E**, y **b** corresponde al nodo  $N$ . Antes de la inserción, la altura de  $N$  es la altura de **C**+1. Idealmente, para recuperar la condición de balance se necesitaría *bajar A* en un nivel y *subir E* en un nivel, lo cual se logra cambiando las referencias *derecha* de **b** e *izquierda* de **d**, quedando este último como nueva raíz del árbol,  $N'$ . Nótese que ahora el nuevo árbol *tiene la misma altura que antes de insertar el elemento*, **C**+1, lo cual implica que *no puede haber nodos desbalanceados más arriba en el árbol*, por lo que es necesaria una sola rotación simple para devolver la condición de balance al árbol. Nótese también que el árbol *sigue cumpliendo con la propiedad de ser ABB*.

## Rotación doble

Claramente un desbalance producido por una inserción "hacia adentro" con respecto a  $N$  no es solucionado con una rotación simple, dado que ahora es **C** quien produce el desbalance y, como se vio anteriormente, este subárbol mantiene su posición relativa con una rotación simple.

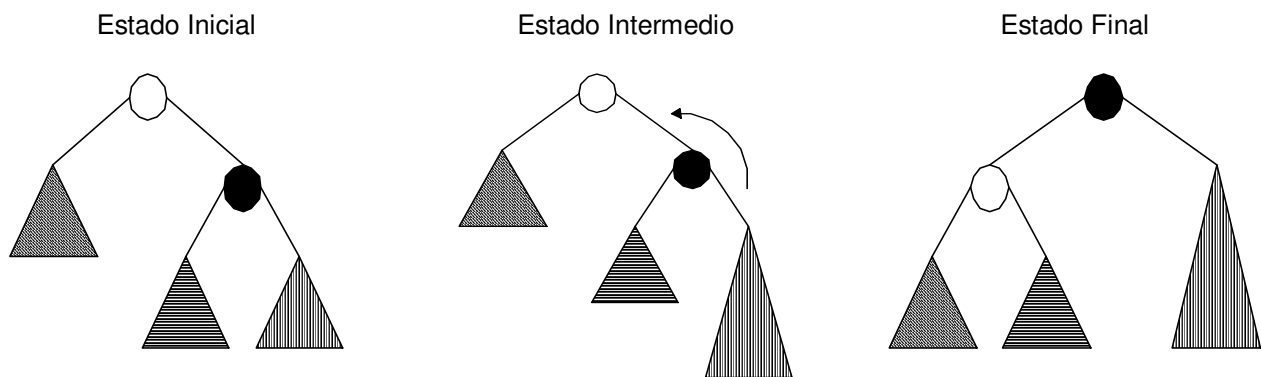


Para el caso de la figura (tercer caso), la altura de  $N$  antes de la inserción era **G**+1. Para recuperar el balance del árbol es necesario subir **C** y **E** y bajar **A**, lo cual se logra realizando dos rotaciones simples: la primera entre **d** y **f**, y la segunda entre **d**, ya rotado, y **b**, obteniéndose el resultado de la figura. A este proceso de dos rotaciones simples se le conoce como rotación doble, y como la altura del nuevo árbol  $N'$  es la misma que antes de la inserción del elemento, ningún elemento hacia arriba del árbol queda desbalanceado, por lo que sólo es necesaria una rotación doble para recuperar el balance del árbol después de la inserción. Nótese que el nuevo árbol cumple con la propiedad de ser árbol binario de búsqueda después de la rotación doble.

A continuación se presenta un esquema más detallado de cada una de las rotaciones mencionadas hasta ahora, así como los algoritmos para implementarlas.

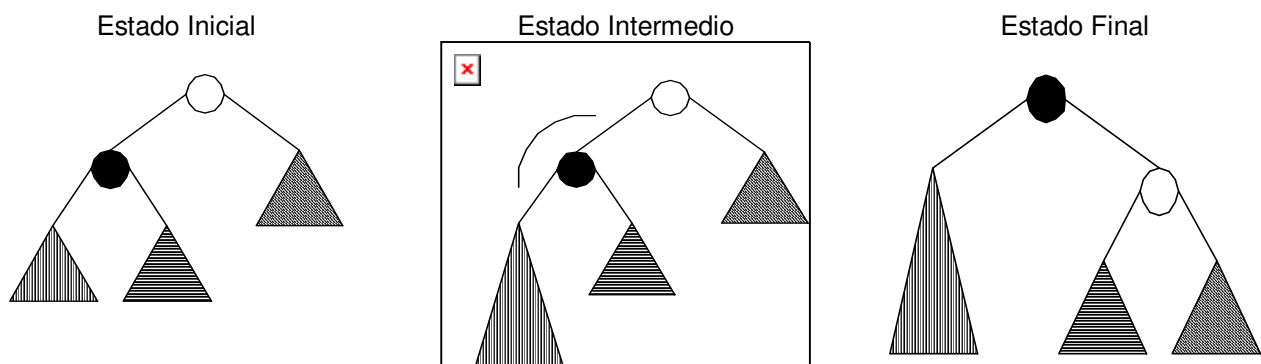
## Rotaciones Simples

### Rotación Izquierda



```
proc rotación_izq(var A: ArBin)
var
  temp: ArBin;
comienzo
  temp ← hijo_der(A);
  A ← crear_ArBin(raíz(A), hijo_izq(A), hijo_izq(temp));
  rotación_izq ← crear_ArBin(raíz(temp), A, hijo_der(temp))
fin;
```

### Rotación Derecha

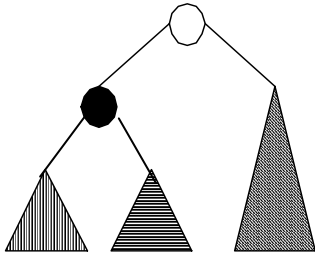


```
proc rotación_der(var A: ArBin)
var
  temp: ArBin;
comienzo
  temp ← hijo_izq(A);
  A ← crear_ArBin(raíz(A), hijo_der(temp), hijo_der(A));
  rotación_der ← crear_ArBin(raíz(temp), hijo_izq(temp), A)
fin;
```

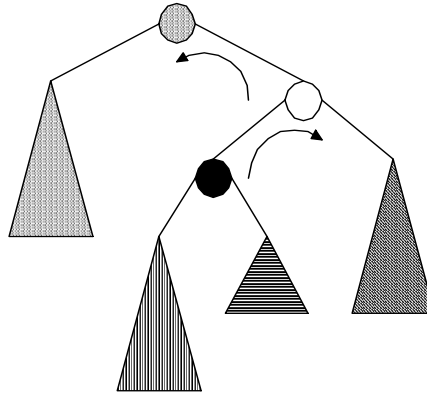
## Rotaciones Dobles

### Rotación Derecha – Izquierda

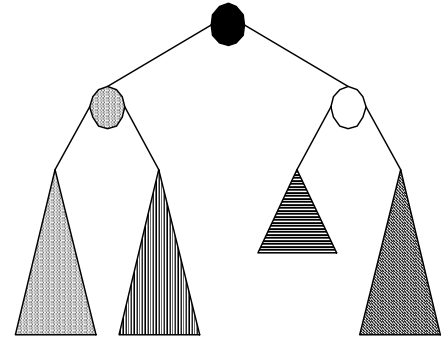
Estado Inicial



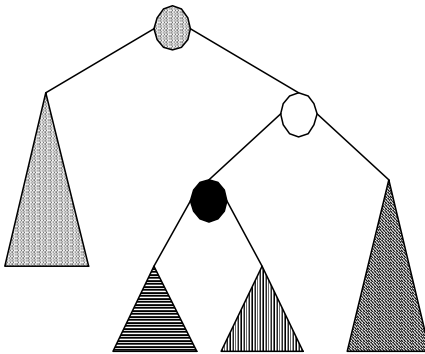
Estado Intermedio



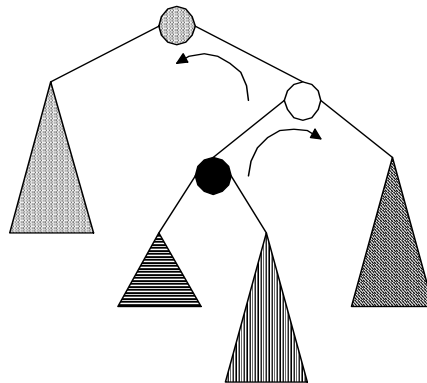
Estado Final



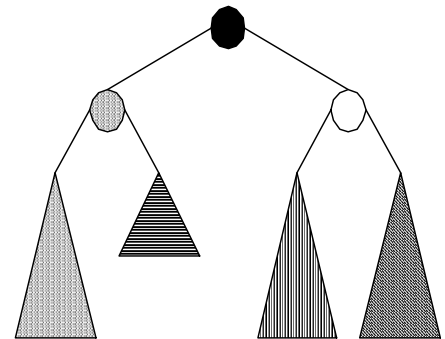
Estado Inicial



Estado Intermedio



Estado Final



```
Proc rotación_DI(var A: ArBin)
```

```
var
```

```
temp: ArBin;
```

```
comienzo
```

```
temp ← rotación_der(hijo_der(A));
```

```
rotación_DI ← rotación_izq(crearArBin(raíz(A), hijo_izq(A), temp)
```

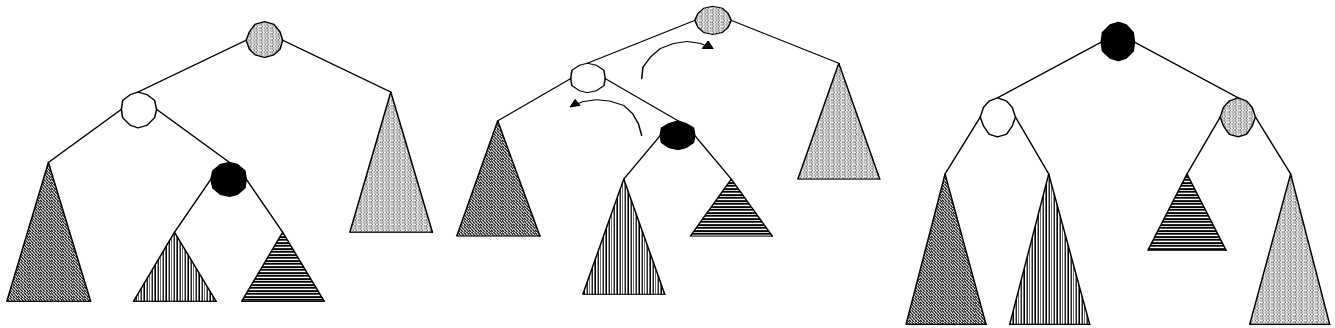
```
fin;
```

### Rotación Izquierda – Derecha

Estado Inicial

Estado Intermedio

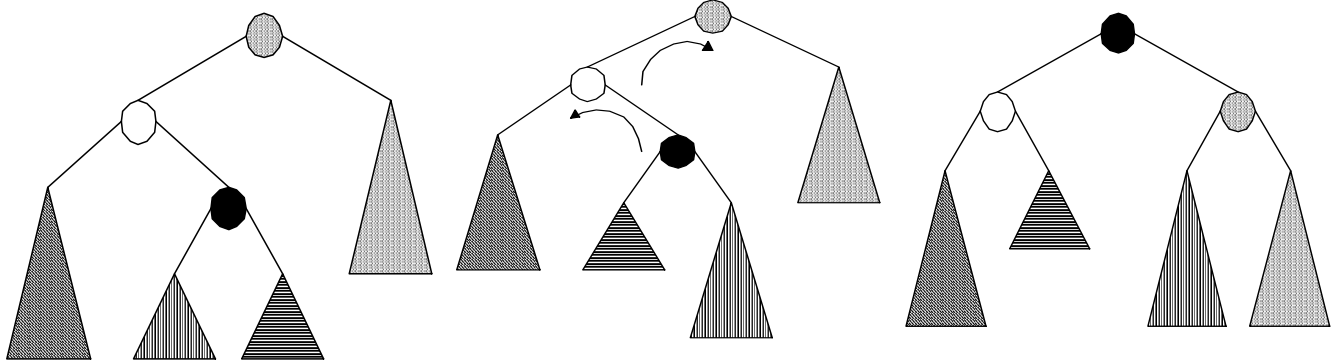
Estado Final



Estado Inicial

Estado Intermedio

Estado Final



```
proc rotación_ID(NS A: ArBin)
var
    temp: ArBin;
comienzo
    temp ← rotación_izq(hijo_izq(A));
    rotación_ID ← rotación_der(crearArBin(raíz(A), temp, hijo_der(A))
fin;
```

### Eliminación en un AVL

La eliminación en árbol AVL se realiza de manera análoga a un árbol binario de búsqueda, pero también es necesario verificar que la condición de balance se mantenga una vez eliminado el elemento. En caso que dicha condición se pierda, será necesario realizar una rotación simple o doble dependiendo del caso, pero es posible que se requiera más de una rotación para reestablecer el balance del árbol.

### Construcción de un árbol AVL

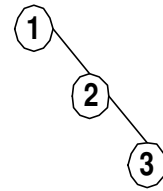
Construir un árbol AVL a partir de la siguiente secuencia de elementos.

⟨ 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8 ⟩

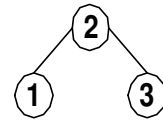
Realice el proceso de construcción paso a paso.

Los elementos 1 y 2 se insertan sin inconvenientes. Al

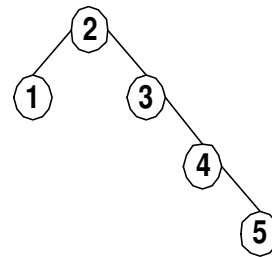
insertar el 3 se viola la condición de equilibrio en el nodo raíz:



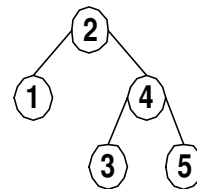
Se realiza una rotación simple para solucionar el problema:



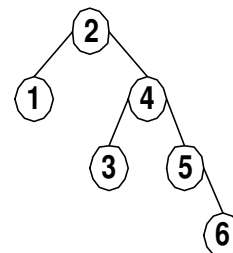
El elemento 4 se agrega sin problemas. Pero, la inserción del 5 produce una violación de la condición de equilibrio en el nodo 3:



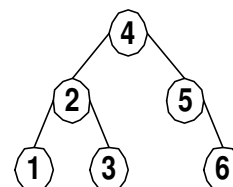
Se realiza una rotación simple para solucionar el problema:



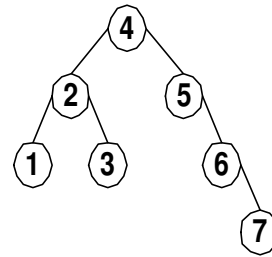
La inserción del 6 genera un problema de equilibrio en el nodo raíz:



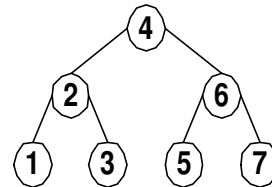
Se realiza una rotación simple para solucionar el problema:



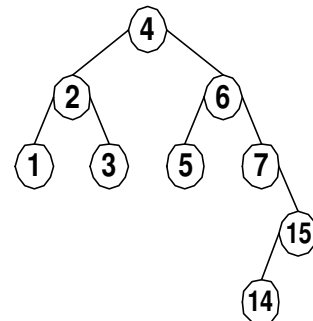
Al insertar el 7, se produce un problema de equilibrio en el nodo 5:



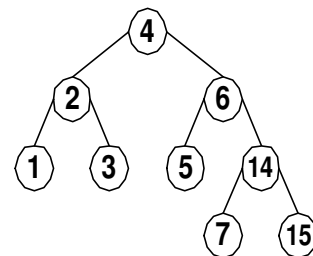
El problema se resuelve por medio de una rotación simple:



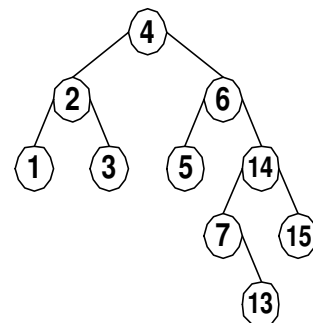
La inserción del 15 no destruye la propiedad de equilibrio. Pero, al insertar el 14 se produce un desequilibrio en el nodo 7:



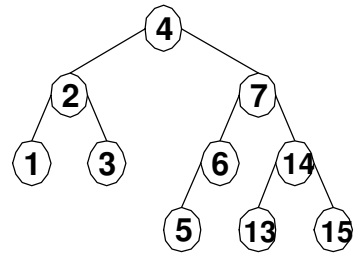
Para solucionar este inconveniente, se realiza una rotación doble derecha-izquierda:



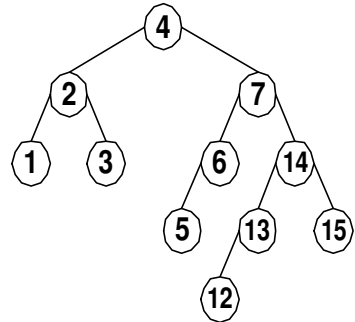
Al insertar el elemento 13, se produce un desequilibrio en el nodo 6:



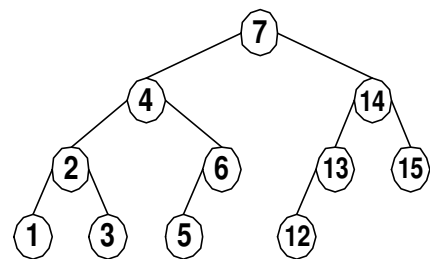
Para solucionar este inconveniente, se realiza una rotación doble derecha-izquierda:



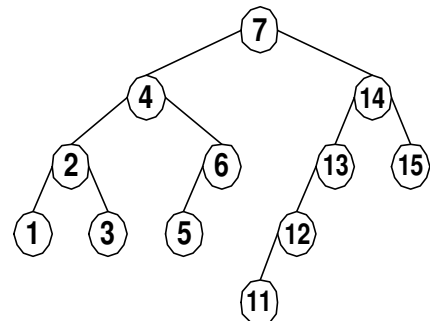
Al insertar el 12, se viola la condición de equilibrio en el nodo raíz:



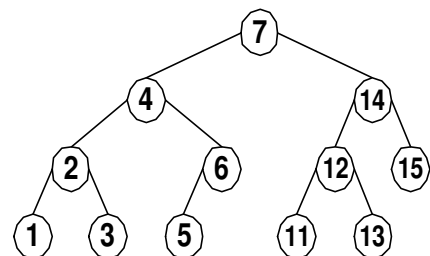
Se realiza una rotación simple para solucionar el problema:



La inserción de 11 produce un desequilibrio en el nodo 13:

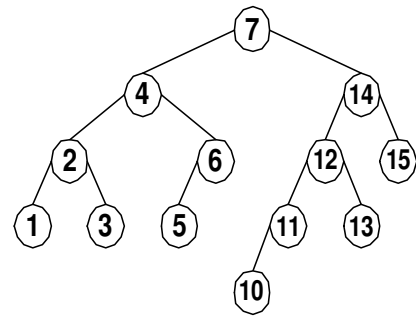


Se requiere una rotación simple para solucionar este inconveniente:

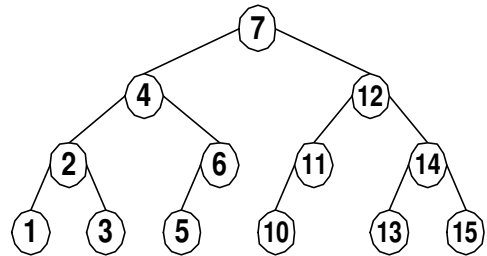




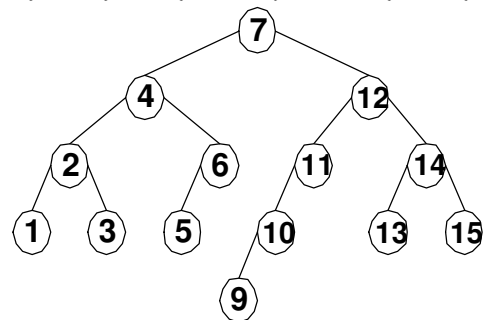
Al insertar el 10, se produce un desequilibrio en el nodo 14:



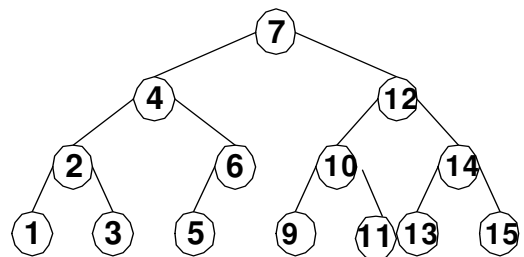
El equilibrio se restaura por medio de una rotación simple:



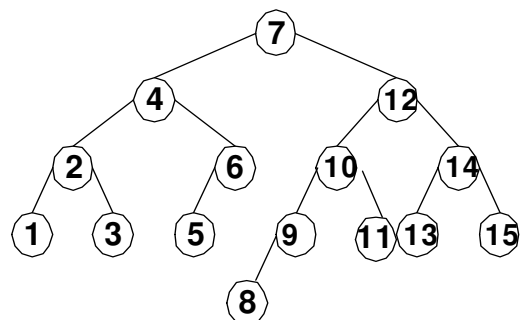
Al insertar el 9, se viola la condición de equilibrio en el nodo 11:



El equilibrio se restablece por medio de una rotación simple:



Finalmente, el 8 se inserta sin inconvenientes:





## 11.- Árboles B (B-trees)

Los árboles B son árboles cuyos nodos pueden tener un número múltiple de hijos y un número múltiple de elementos. La siguiente figura muestra un esquema general de un nodo de un árbol B.

$P_1$	$e_1$	$P_2$	$e_2$		$\dots$		$e_{n-1}$	$P_n$
-------	-------	-------	-------	--	---------	--	-----------	-------

donde

- $P_i$  : Apuntador al i-ésimo hijo del nodo.
- $e_i$  : i-ésimo elemento del nodo

El conjunto de elementos o claves que se encuentran en un nodo cumplen la condición:

$$e_1 < e_2 < \dots < e_{n-1}$$

de forma que los elementos que se encuentran en el primer hijo tienen una clave con valor menor que  $e_1$ , los que se encuentran en el segundo hijo tienen una clave con valor mayor que  $e_1$  y menor que  $e_2$ , etc. Finalmente, los elementos que se encuentran en el último hijo tienen una clave con valor mayor que la última clave. que el nodo puede tener menos de  $m$  hijos y por consiguiente menos de  $m-1$  claves).

De una manera más formal, un árbol B es un árbol n-ario ordenado y balanceado en el cual, cada nodo tiene como máximo  $(n-1)$  elementos y  $n$  subárboles asociados, los cuales son a su vez árboles B. Un árbol B de orden  $n$  se define como sigue:

1. Cada nodo en el árbol B tiene la siguiente estructura:

$$\langle P_1, e_1, P_2, e_2, \dots, P_{q-1}, e_{q-1}, P_q \rangle$$

donde:

- $q \leq n$
- $\forall i (1 \leq i \leq q) (P_i \text{ es un apuntador a otro nodo del árbol B})$
- $\forall i (1 \leq i < q) (e_i \text{ es una clave del árbol B})$

2. En cada nodo del árbol B se verifica que  $\forall i (1 \leq i < q-1) (e_i < e_{i+1})$

3. Para toda clave  $x$  en el sub-árbol apuntado por  $P_i$  se verifica que:

- $(i = 1) \rightarrow (x < e_1)$
- $(1 < i < q) \rightarrow (e_{i-1} < x < e_i)$
- $(i = q) \rightarrow (e_{i-1} < x)$

4. Todo nodo interior en un árbol B tiene a lo sumo  $n$  hijos.

5. Un nodo interior con  $q$  descendientes,  $q \leq n$ , contiene exactamente  $q-1$  claves.

6. Todo nodo interior, excepto el nodo raíz, tiene por lo menos  $\lceil (p/2) \rceil$  hijos.

7. Si el nodo raíz es un nodo interior entonces tiene por lo menos 2 hijos.

8. Todo nodo hoja contiene por lo menos  $\lceil (p/2) \rceil - 1$  claves y a lo sumo  $p-1$  claves.
9. Todos los nodos hoja se encuentran en el mismo nivel.

### ***Búsqueda en un árbol B***

Localizar una clave en un B-árbol es una operación simple, pues consiste en situarse en el nodo raíz del árbol, si la clave se encuentra allí la búsqueda finaliza, y si no es así, se selecciona de entre los hijos el que se encuentra entre dos valores de clave que son menor y mayor que la clave. El proceso continúa recursivamente hasta encontrar la clave o llegar a un nodo hoja. En caso de que se llegue a una hoja y no podamos proseguir la búsqueda la clave no se encuentra en el árbol. En definitiva, los pasos a seguir son los siguientes:

1. Seleccionar como nodo actual la raíz del árbol.
2. Comprobar si la clave se encuentra en el nodo actual:
  3. Si la clave está, fin.
  4. Si la clave no está:
    - Si estamos en una hoja, no se encuentra la clave. Fin.
    - Si no estamos en una hoja, hacer nodo actual igual al hijo que corresponde, según el valor de la clave a buscar y volver al segundo paso.

### ***Insertión en un árbol B***

Para insertar una nueva clave usaremos un algoritmo que consiste en dos pasos recursivos:

1. Buscamos la hoja donde debería encontrarse el valor de la clave de manera análoga a la descrita en la sección anterior (si el proceso de búsqueda indica que la clave se encuentra en el árbol, el algoritmo no debe hacer nada más). Si la clave no se encuentra en el árbol, el nodo actual será una hoja, que es justamente el lugar donde debe realizarse la inserción.
2. Situados en un nodo donde realizar la inserción, si este no está completo, es decir, si el número de claves que existen es menor que el orden menos 1 del árbol, el elemento puede ser insertado y el algoritmo termina. En caso de que el nodo esté completo, insertamos la clave en su posición y dado que la cantidad de claves excede la capacidad del nodo, éste se divide en dos nuevos nodos conteniendo cada uno de ellos la mitad de las claves y tomando una de éstas para insertarla en el padre (se usará la mediana). Si el padre está también completo, habrá que repetir el proceso de manera recursiva, hasta llegar a la raíz. En caso de que la raíz esté completa, la altura del árbol aumenta en uno creando un nuevo nodo raíz con una única clave.

En la figura 2 podemos observar el efecto de insertar una nueva clave en un nodo que está lleno.

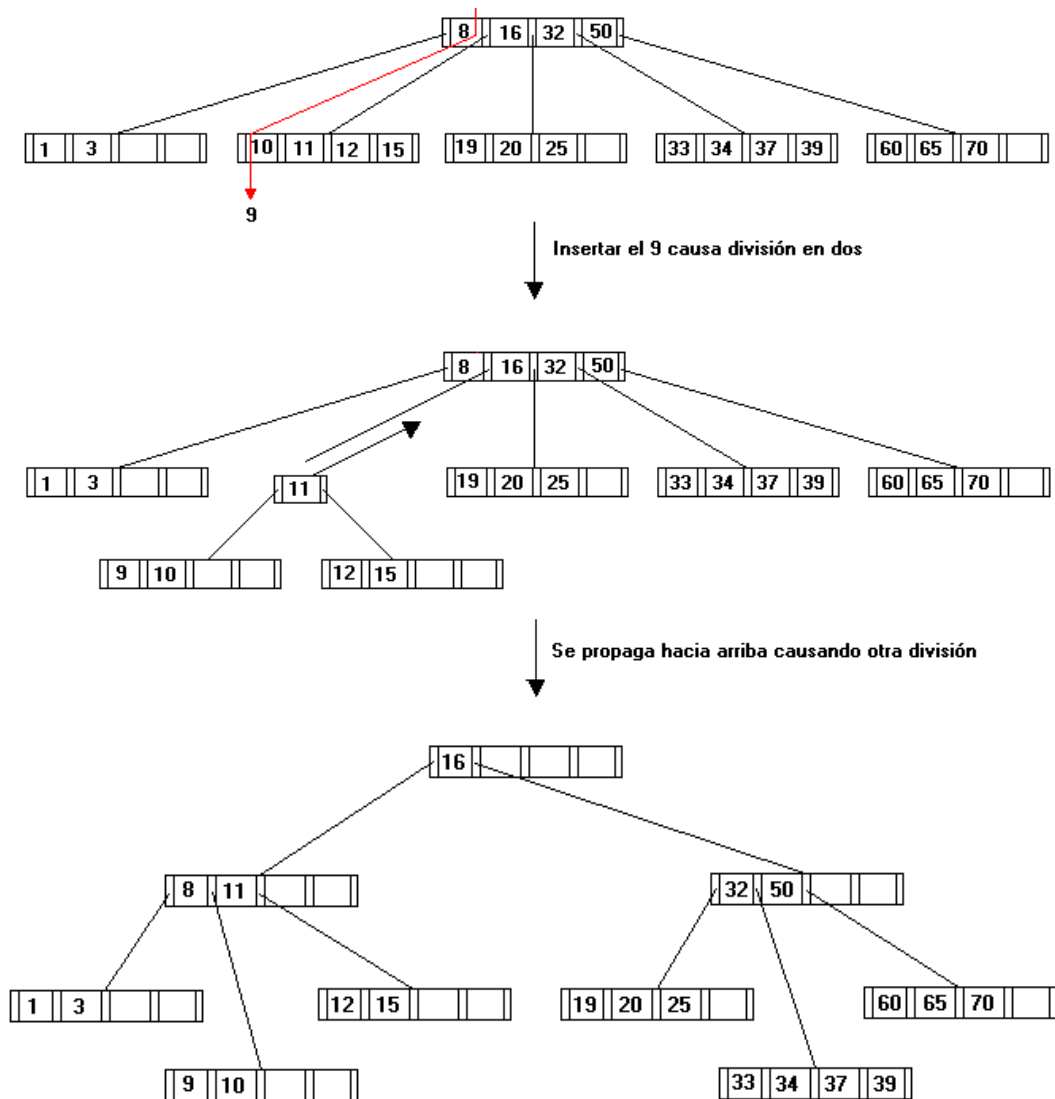


Figura 2: Inserción de un nuevo elemento en un B-árbol

## Eliminación en un árbol B

La idea para realizar el borrado de una clave es similar a la inserción, teniendo en cuenta que ahora, en lugar de divisiones, realizamos uniones. Existe un problema añadido, las claves a borrar pueden aparecer en cualquier lugar del árbol y por consiguiente no coincide con el caso de la inserción en la que siempre comenzamos desde una hoja y propagamos hacia arriba. La solución a esto es inmediata pues cuando borramos una clave que está en un nodo interior, lo primero que realizamos es un intercambio de este valor con el inmediato sucesor en el árbol, es decir, el hijo más a la izquierda del hijo derecho de esa clave.

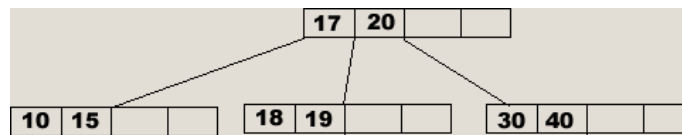
Las operaciones a realizar para poder llevar a cabo el borrado son por tanto:

1. Redistribución: La utilizaremos en el caso en que al borrar una clave el nodo se queda con un número menor que el mínimo y uno de los hermanos adyacentes tiene al menos uno más que ese mínimo, es decir, redistribuyendo podemos solucionar el problema.
2. Unión: La utilizaremos en el caso de que no sea posible la redistribución y por tanto sólo será posible unir los nodos junto con la clave que los separa y se encuentra en el padre.

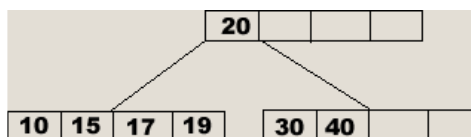
En definitiva, el algoritmo nos queda como sigue:

1. Localizar el nodo donde se encuentra la clave.
2. Si el nodo localizado no es una hoja, intercambiar el valor de la clave localizada con el valor de la clave más a la izquierda del hijo a la derecha. Esto tiene como objetivo colocar la clave a borrar en una hoja. Dicha hoja pasa a ser el nodo actual.
3. Borrar la clave.
4. Si el nodo actual contiene al menos el mínimo de claves como para seguir siendo un árbol B, fin.
5. Si el nodo actual tiene un número menor que el mínimo:
  6. Si un nodo hermano tiene más del mínimo de claves, se agrupa el nodo actual con el nodo hermano y con el elemento del nodo padre que está entre ambos, el elemento mediano del conjunto resultante se promueve al padre, y el resto de elementos se distribuye equitativamente entre las dos páginas.
  7. Si ninguno de los hermanos tiene más del mínimo, se produce el efecto inverso a la división, los dos nodos se agrupan en uno solo. A este nodo se le añade el elemento central que estaba situado en el nodo padre.

A continuación se verá un ejemplo gráfico. Consideremos el siguiente árbol B



Al eliminar la clave 18, el resultado es el siguiente:



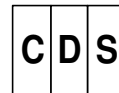
## Construcción de un árbol B: inserciones, divisiones (*split*) y promociones (*promotion*)

Construir un árbol B de orden 4 a partir de la siguiente secuencia de claves.

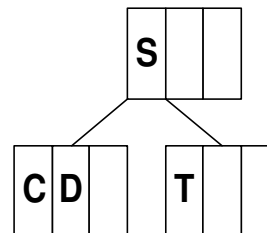
⟨ C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J, Y, Q, Z, F, X, V ⟩

Realice el proceso de construcción paso a paso.

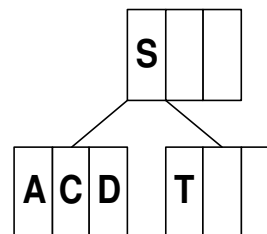
La inserción de las claves C, S y D se realiza en el nodo inicial:



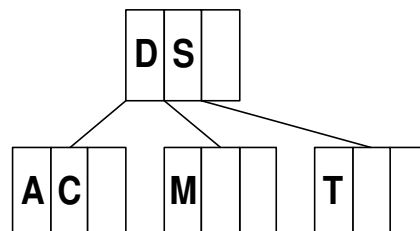
La inserción de T origina la división del nodo inicial y la promoción de S:



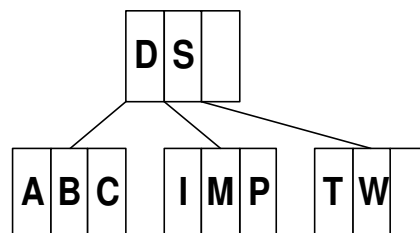
La inserción de A se realiza sin mayores inconvenientes:



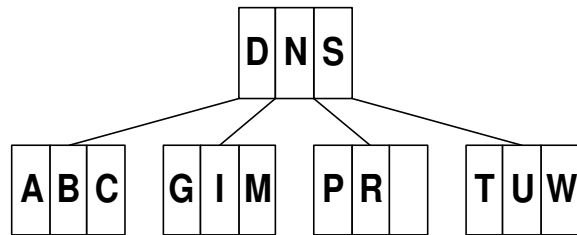
La inserción de M origina otra división y la promoción de D:



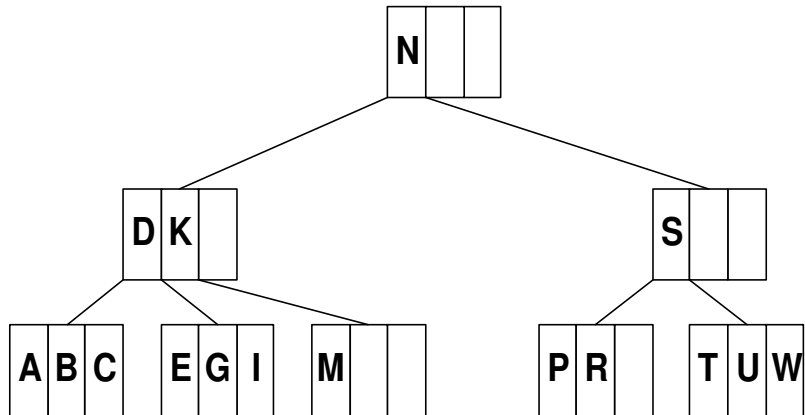
Las claves P, I, B y W son insertadas en los nodos existentes:



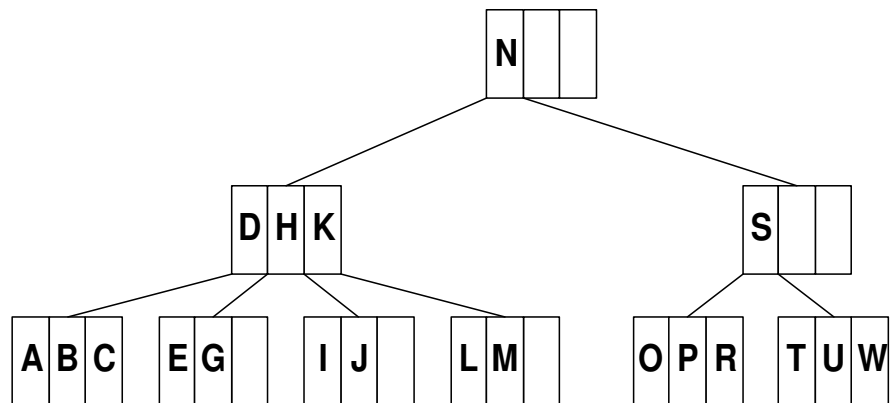
La inserción de N ocasiona otra división, seguida por la promoción de N. Las claves G, U y R son insertadas en los nodos existentes:



La inserción de K ocasiona una división en el nivel de las hojas, seguida por la promoción de K. Esta operación produce una división del nodo raíz. N es promovida y pasa a estar en el nuevo nodo raíz. E es insertado en una hoja:



La inserción de H origina la división de un nodo hoja. H es promovida. O, L y J son insertados en los nodos existentes:



La inserción de Y y Q origina dos divisiones a nivel de las hojas, seguidas de las promociones correspondientes. Las claves Z, F, X, y V son insertadas en los nodos existentes, sin mayores inconvenientes. El árbol B de orden 4 resultante es el siguiente:



