# Tema 3
# Estructuras Jerárquicas
## *Árboles*

## 1. Introducción

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:

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

En estos casos se necesita una estructura adecuada que permita representar este tipo de estructuras jerárquicas, los árboles.

## 2. Árboles: Definiciones Básicas

a) **Árbol:** Un árbol es un grafo conexo y acíclico.
b) **Grado de un nodo:** En los grafos no dirigidos es el número de arcos que inciden sobre un vértice. En los grafos dirigidos existen dos tipos de grado: (1) Grado de entrada, el cual es la cantidad de arcos que inciden positivamente sobre un vértice (gráficamente son las flechas que llegan), y (2) Grado de salida, el cual es la cantidad de arcos que inciden negativamente sobre un vértice (gráficamente son las flechas que salen).
(Nota: En este tema cuando se hable del grado de salida de un nodo se le llamará simplemente grado, y representará el número de hijos (subárboles) de ese nodo.)
c) **Grado de un árbol:** Es el grado del nodo de mayor grado en el árbol.
d) **Peso de un árbol:** Es el número de nodos que contiene el árbol. El peso de un árbol nulo es 0.
e) **Nodo terminal u hoja:** Es un nodo de grado 0.
f) **Nodo no terminal o interior:** Es todo nodo de un árbol que no es una hoja.
g) **Nodos hermanos:** Dos nodos $e_1$ y $e_2$ de un árbol son hermanos si y sólo si tienen el mismo padre.
h) **Camino entre dos nodos:** Un camino entre dos nodos $e$ y $e'$ de un árbol es una secuencia de nodos $<e_1, e_2, \ldots, e_n>$ tal que verifica las siguientes propiedades:
   - $e_1 = e$
   - $e_n = e'$
   - $\forall i\ (\ 1 \le i < n\ )$: $e_i$ es padre de $e_{i+1}$.

i) **Longitud de un camino:** La longitud de un camino C = <$e_1$, $e_2$, …, $e_n$> se define como el número de veces que se debe aplicar la relación padre → hijo durante el recorrido, es decir, es el número de arcos en el camino o el número de nodos en el camino menos 1.

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

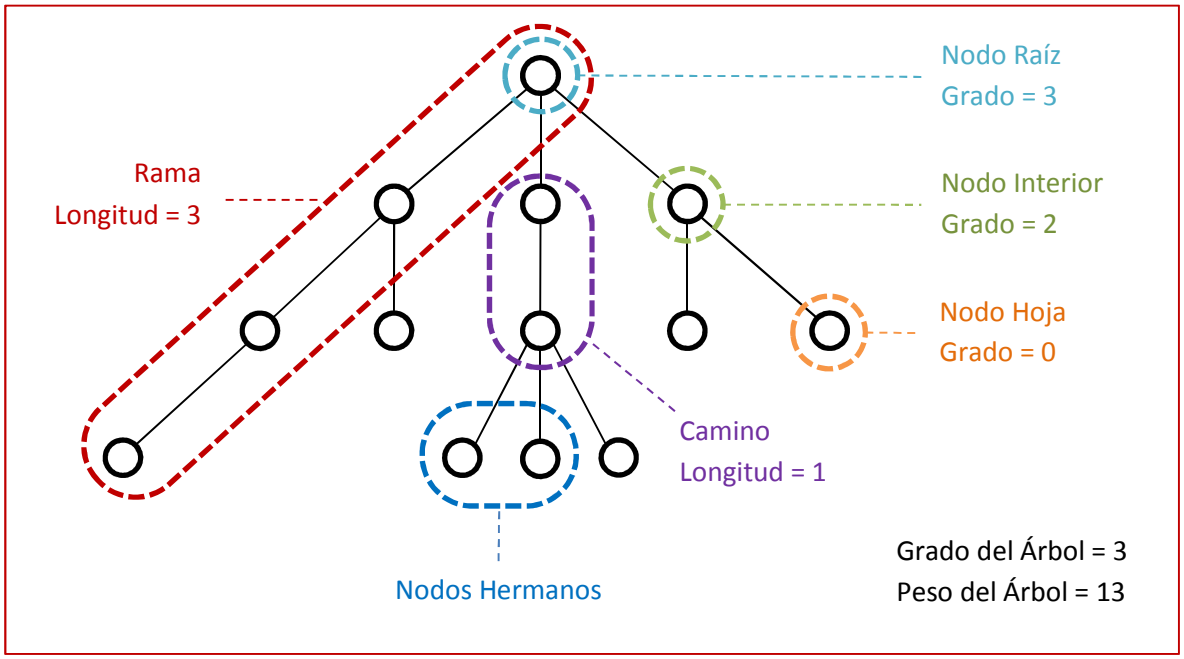



Figura 1. Algunos conceptos básicos de un árbol.

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

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

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

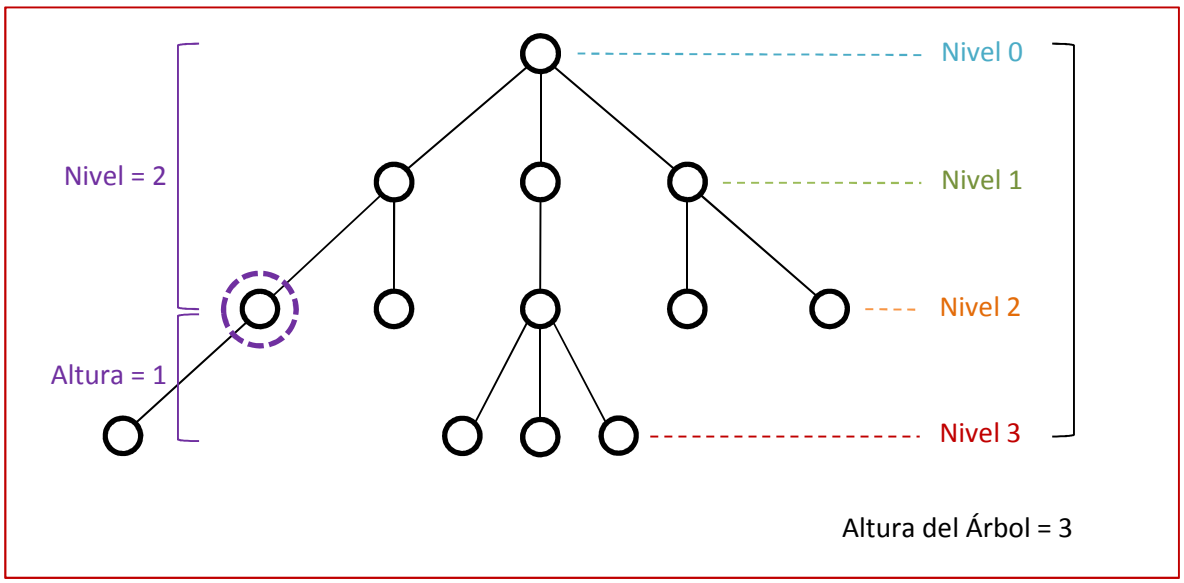



Figura 2. Nivel y altura de un nodo. Altura de un árbol.

## 3. Árboles N-Arios

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

a) La secuencia vacía representa al árbol nulo no ordenado. Se denota por $\Lambda$.

b) Sea *e* un elemento de tipo T y $A_1$, $A_2$, …, $A_n$ árboles *n*-arios no ordenados con tipo base T. Entonces la secuencia A = <*e*, [$A_1$, $A_2$, …, $A_n$]>, donde [$A_1$, $A_2$, …, $A_n$] 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.
   - $\forall i$ ( $1 \le i \le n$ ): $A_i$ es un subárbol de A.
   - $\forall i$ ( $1 \le i \le n$ ): *e* es padre de la raíz de $A_i$.
   - $\forall i$ ( $1 \le i \le n$ ): La raíz de $A_i$ es hijo de *e*.

c) Sólo son árboles *n*-arios no ordenados con tipo base T los objetos generados por la aplicación de las cláusulas (1) y (2).

### 3.1. Especificación Operacional

**ArbolN[Elemento]**

Invariante:

ArbolN = (<*e*, lst>) $\wedge$ (*e* $\in$ Elemento) $\wedge$ (lst = <$A_1$, $A_2$, …, $A_{k-1}$, $A_k$, $A_{k+1}$, …, $A_n$>) $\wedge$ ($\forall i$ ($1 \le i \le n$): $A_i \in$ ArbolN)

Sintaxis:

| | | | |
|---|---|---|---|
| nulo: | | $\rightarrow$ | ArbolN |
| esNulo: | ArbolN | $\rightarrow$ | Lógico |
| crear: | Elemento × Lista[ArbolN] | $\rightarrow$ | ArbolN |
| raíz: | ArbolN | $\rightarrow$ | Elemento |
| hijos: | ArbolN | $\rightarrow$ | Lista[ArbolN] |
| insertarSubarbol: | ArbolN × ArbolN | $\rightarrow$ | ArbolN |
| eliminarSubarbol: | ArbolN × Natural | $\rightarrow$ | ArbolN |

Semántica:

{ Pre: }
    **func** nulo(): ArbolN
{ Post: nulo $\leftarrow \Lambda$ }

{ Pre: }
    **func** esNulo(ArbolN: A): **Lógico**
{ Post: esNulo $\leftarrow$ (A = $\Lambda$) }

{ Pre: }
    **func** crear(Elemento: e; Lista[ ArbolN ]: lst): ArbolN
{ Post: crear $\leftarrow$ <e, lst> }

{ Pre: ¬esNulo(A) }
**func** raiz(ArbolN: A) : Elemento
{ Post: raíz ← e }

{ Pre: ¬esNulo(A) }
**func** hijos(ArbolN: A): Lista[ ArbolN ]
{ Post: hijos ← lst }

{ Pre: ¬esNulo(A) ∧ ¬esNulo(B) }
**proc** insertarSubarbol(**ref** ArbolN: A; ArbolN: B)
{ Post: A' ← <e, <$A_1$, $A_2$, …, $A_{k-1}$, $A_k$, $A_{k+1}$, …, $A_N$, B>> }

{ Pre: ¬esNulo(A) ∧ ( 1 ≤ k ≤ longitud(hijos(A)) ) }
**proc** eliminarSubarbol(**ref** ArbolN: A; **entero**: k)
{ Post: A' ← <e, <$A_1$, $A_2$, …, $A_{k-1}$, $A_{k+1}$, …, $A_N$>> }

## 3.2. 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.

**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.

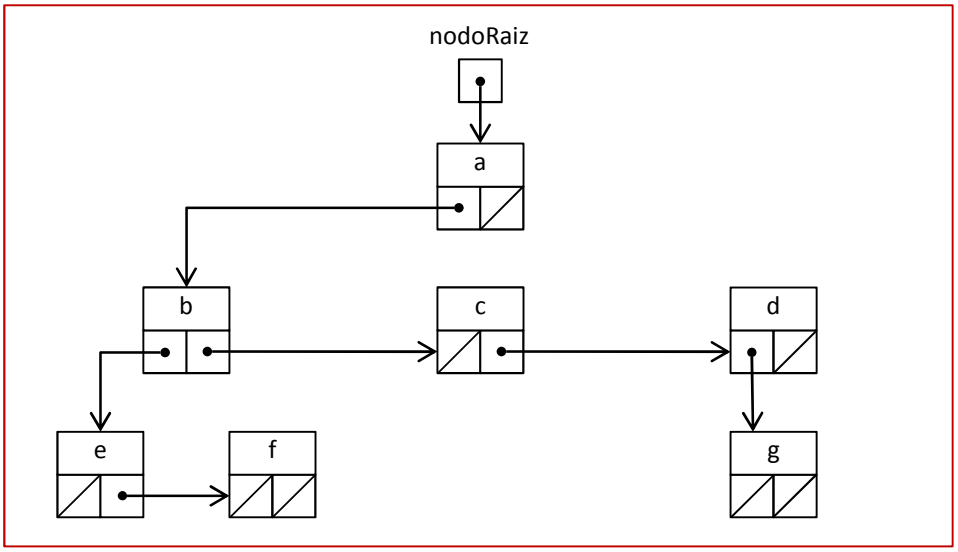



Figura 3. Representación gráfica de la estructura de un árbol n-ario.

```
clase NodoArbol
      atributos:
            privado:
                  Elemento: e
                  Apuntador a  NodoArbol: hijoIzq, hnoDer
      métodos:
            público:
                  proc construir()
                  proc construir(Elemento: e; Apuntador a NodoArbol: izq, der)
                  func getElemento(): Elemento
                  func getHijoIzquierdo(): Apuntador a  NodoArbol
                  func getHermanoDerecho(): Apuntador a NodoArbol
                  proc setElemento(Elemento: e)
                  proc setHijoIzquierdo(Apuntador a NodoArbol: p)
                  proc setHermanoDerecho(Apuntador a NodoArbol: p)
fclase

clase ArbolN
      atributos:
            privado:
                  Apuntador a NodoArbol: nodoRaiz
      métodos:
            privado:
                  estático func copiarNodos(Apuntador a NodoArbol: ptrNodo):
                        Apuntador a NodoArbol
                  estático proc destruirNodos(ref Apuntador a NodoArbol: p)
            público:
                  proc construir()                                  // nulo
                  proc construir(Elemento: e; Lista[ArbolN]: lst) // crear
                  proc copiar(ref ArbolN: A)
                  func esNulo(): Lógico
                  func raiz(): Elemento
                  func hijos(): Lista[ArbolN]
                  proc insertarSubarbol(ArbolN: arbol);
                  proc eliminarSubarbol (Entero: pos);
                  proc destruir()
fclase

proc ArbolN::construir()
inicio
      nodoRaiz ← nulo
fproc

estático func ArbolN::copiarNodos(Apuntador a NodoArbol: p): Apuntador a NodoArbol
var
      Apuntador a NodoArbol: nuevoNodo
inicio
      si(p = nulo) entonces
            retornar(nulo)
```

```
      sino
            nuevoNodo.crear(NodoArbol)
            nuevoNodo↑.construir(p↑.getElemento(),
                  copiarNodos(p↑.getHijoIzquierdo()),
                  copiarNodos(p↑.getHermanoDerecho()))
            retornar(nuevoNodo)
      fsi
ffunc

proc ArbolN::construir(Elemento: e; Lista[ArbolN]: L)
var
      Apuntador a NodoArbol: aux
inicio
      nodoRaiz.crear(NodoArbol)
      nodoRaiz↑.construir(e, nulo, nulo)
      si(¬L.esVacia()) entonces
            nodoRaiz↑.setHijoIzquierdo(copiarNodos(L.consultar(1).nodoRaiz))
            L.eliminar(1)
            aux ← nodoRaiz↑.getHijoIzquierdo()
            mientras(¬L.esVacia()) hacer
                  aux↑.setHermanoDerecho(copiarNodos(L.consultar(1).nodoRaiz))
                  L.eliminar(1)
                  aux ← aux↑.getHermanoDerecho()
            fmientras
      fsi
fproc

proc ArbolN::copiar(ref ArbolN: A)
inicio
      nodoRaiz ← copiarNodos(A.nodoRaiz)
fproc

func ArbolN::esNulo(): Lógico
inicio
      retornar(nodoRaiz = nulo)
ffunc

func ArbolN::raiz(): Elemento
inicio
      retornar(nodoRaiz↑.getElemento())
ffunc

func ArbolN::hijos(): Lista[ArbolN]
var
      Apuntador a NodoArbol: aux
      ArbolN: arbolAux
      Lista[ArbolN]: L
inicio
```

```
      L.construir()
      aux ← nodoRaiz↑.getHijoIzquierdo()
      mientras(aux ≠ nulo) hacer
            arbolAux.nodoRaiz ← aux
            L.insertarAlFinal(arbolAux) // Inserta una copia
            aux ← aux↑.getHermanoDerecho()
      fmientras
      retornar(L)
ffunc

proc ArbolN::insertarSubarbol(ArbolN: subarbol)
var
      Apuntador a NodoArbol: aux
inicio
      si(nodoRaiz↑.getHijoIzquierdo() = nulo) entonces
            nodoRaiz↑.setHijoIzquierdo(copiarNodos(subarbol.nodoRaiz))
      sino
            aux ← nodoRaiz↑.getHijoIzquierdo()
            mientras(aux↑.getHermanoDerecho() ≠ nulo) hacer
                  aux ← aux↑.getHermanoDerecho()
            fmientras
            aux↑.setHermanoDerecho(copiarNodos(subarbol.nodoRaiz))
      fsi
fproc

estático proc ArbolN::destruirNodos(ref Apuntador a NodoArbol: p)
inicio
      si(p ≠ nulo) entonces
            si(p↑.getHermanoDerecho() ≠ nulo) entonces
                  destruirNodos(p↑.getHermanoDerecho())
            fsi
            si(nodoRaiz↑.getHijoIzquierdo() ≠ nulo) entonces
                  destruirNodos(p↑.getHijoIzquierdo())
            fsi
            p.destruir() // Destruir el nodo
            p ← nulo    // Colocar en nulo
      fsi
fproc

proc ArbolN::eliminarSubarbol(Entero: pos)
var
      Apuntador a NodoArbol: aux, elim
      Entero: i
inicio
      si(pos = 1) entonces
            elim ← nodoRaiz↑.getHijoIzquierdo()
            nodoRaiz↑.setHijoIzquierdo(nodoRaiz↑.getHijoIzquierdo()↑.getHermanoDerecho())
```

```
      sino
            aux ← nodoRaiz↑.getHijoIzquierdo()
            para i ← 2 hasta pos – 1 hacer
                  aux ← aux↑.getHermanoDerecho()
            fpara
            elim ← aux↑.getHermanoDerecho()
            aux↑.setHermanoDerecho(aux↑.getHermanoDerecho()↑.getHermanoDerecho())
      fsi
      elim↑.setHermanoDerecho(nulo)
      destruirNodos(elim)
fproc

// Versión Recursiva
proc ArbolN::destruir()
inicio
      destruirNodos(nodoRaiz)
fproc

// Versión Iterativa (con una Pila)
proc ArbolN::destruir()
var
      Pila[Apuntador a NodoArbol]: p
      Apuntador a NodoArbol: aux
inicio
      si(nodoRaiz ≠ nulo) entonces
            p.apilar(nodoRaiz)
            mientras(¬p.esVacia()) hacer
                  aux ← p.tope()
                  p.desapilar()
                  si(aux↑.getHijoIzquierdo() ≠ nulo) entonces
                        p.apilar(aux↑.getHijoIzquierdo())
                  fsi
                  si(aux↑.getHermanoDerecho() ≠ nulo) entonces
                        p.apilar(aux↑.getHermanoDerecho())
                  fsi
                  aux.destruir()
            fmientras
            nodoRaiz ← nulo
      fsi
fproc
```

**Representación con Vector de Apuntadores**

```
clase NodoArbol
      atributos:
            privado:
                  Elemento: e
                  estático const Entero: MAX ← 100
                  Arreglo [1..MAX] de Apuntador a  NodoArbol: subarboles
      métodos:
            público:
                  proc construir()
                  proc construir(Elemento: e; Arreglo [1..MAX] de Apuntador a
                        NodoArbol: subarboles)
                  func getElemento(): Elemento
                  func getSubarbol(Entero: pos): Apuntador a  Nodo
                  proc setElemento(Elemento: e)
                  proc setSubarbol(Apuntador a  Nodo: p; Entero: pos)
fclase

clase ArbolN
      atributos:
            privado:
                  Apuntador a NodoArbol: nodoRaiz
      métodos:
            público:
                  proc construir()
                  proc construir(Elemento: e; Lista[ArbolN]: lst)
                  func esNulo(): Lógico
                  func raiz(): Elemento
                  func hijos(): Lista[ArbolN]
                  proc insertarSubarbol(ArbolN: arbol);
                  proc eliminarSubarbol (Entero: pos);
                  proc destruir()
fclase
```

**Ejercicios:**

1. Implemente los métodos de la representación que utiliza un vector de apuntadores.
2. Implemente una clase ArbolN con la siguiente clase NodoArbol:

```
clase NodoArbol
   atributos:
         privado:
               Elemento: e
               Apuntador a NodoArbol: padre, hno_der
   métodos:
   ...
fclase
```

## 3.3. Recorrido

### Recorrido en preorden

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

```
proc ArbolN::preorden()
inicio
      recorridoPreorden(nodoRaiz)
fproc

// Procedimiento privado de ArbolN, solo para uso del procedimiento preorden()
estático proc ArbolN::recorridoPreorden(Apuntador a NodoArbol: ptrRaiz)
var
      Apuntador a  NodoArbol: aux
inicio
      si(ptrRaiz ≠ nulo) entonces
            ptrRaiz↑.getElemento().imprimir() // imprimir() debe estar definido
            aux ← ptrRaiz↑.getHijoIzquierdo()
            mientras(aux ≠ nulo) hacer
                  recorridoPreorden(aux)
                  aux ← aux↑.getHermanoDerecho()
            fmientras
      fsi
fproc
```

### Recorrido en inorden

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

```
proc ArbolN::inorden()
inicio
      recorridoInorden(nodoRaiz)
fproc

// Procedimiento privado de ArbolN, solo para uso del procedimiento inorden()
proc ArbolN::recorridoInorden(Apuntador a NodoArbol: ptrRaiz)
var
      Apuntador a NodoArbol: aux
inicio
      si(ptrRaiz ≠ nulo) entonces
            recorridoInorden(ptrRaiz↑.getHijoIzquierdo())
            nodoRaiz↑.getElemento().imprimir()
```

```
            aux ← ptrRaiz↑.getHijoIzquierdo()↑.getHermanoDerecho()
            mientras(aux ≠ nulo) hacer
                  recorridoInorden(aux)
                  aux ← aux↑.getHermanoDerecho()
            fmientras
      fsi
fproc
```

**Recorrido en postorden**

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

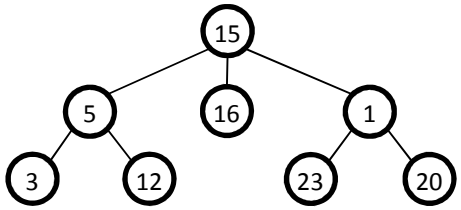
```
proc ArbolN::postorden()
inicio
      recorridoPostorden(nodoRaiz)
fproc

// Procedimiento privado de ArbolN, solo para uso del procedimiento postorden()
estático proc ArbolN::recorridoPostorden(Apuntador a NodoArbol: ptrRaiz)
var
      Apuntador a  NodoArbol: aux
inicio
      si(ptrRaiz ≠ nulo) entonces
            aux ← ptrRaiz↑.getHijoIzquierdo()
            mientras(aux ≠ nulo) hacer
                  recorridoPostorden(aux)
                  aux ← aux↑.getHermanoDerecho()
            fmientras
            ptrRaiz↑.getElemento().imprimir()
      fsi
fproc
```

**Recorrido por niveles**

**Ejercicio:**

1. Diseñe un procedimiento que haga el recorrido por niveles de un árbol *n*-ario.




Para este árbol el recorrido por niveles es 15 5 16 1 3 12 23 20.

# 4. Arboles Binarios

## 4.1. Definiciones Básicas

a) **Árbol completo:** Un árbol binario es completo si y sólo si todo nodo no terminal tiene exactamente dos subárboles no vacíos.
b) **Árbol lleno:** Un árbol binario está lleno si y sólo si es completo y, además, todas las hojas están en el mismo nivel.
c) **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.
d) **Arboles 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.
e) **Arboles isomorfos:** Dos árboles binarios son isomorfos si y sólo si tienen la misma estructura pero no necesariamente los mismos elementos.
f) **Arboles semejantes:** Dos árboles binarios son semejantes si y sólo si contienen los mismos elementos, aunque no sean isomorfos.
g) **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 $A_2$.

## 4.2. Especificación Operacional

**ArBin[Elemento]**

Invariante:
ArBin = (<*e*, $A_1$, $A_2$>) $\wedge$ (*e* $\in$ Elemento) $\wedge$ ( $A_1$, $A_2$ $\in$ ArBin)

Sintaxis:

| | | | |
|---|---|---|---|
| nulo: | | $\rightarrow$ | ArBin |
| esNulo: | ArBin | $\rightarrow$ | Lógico |
| crear: | Elemento × ArBin × ArBin | $\rightarrow$ | ArBin |
| raíz: | ArBin | $\rightarrow$ | Elemento |
| hijoIzquierdo: | ArBin | $\rightarrow$ | ArBin |
| hijoDerecho: | ArBin | $\rightarrow$ | ArBin |

Semántica:

{ Pre: }
**func** nulo(): ArBin
{ Post: nulo $\leftarrow \Lambda$ }

{ Pre: }
**func** esNulo(ArBin: A): **Lógico**
{ Post: esNulo $\leftarrow$ (A = $\Lambda$) }

{ Pre: }
**func** crear(Elemento: e; ArBin: A1, A2): ArBin
{ Post: crear ← <e, A1, A2> }

{ Pre: ¬esNulo(A) }
**func** raiz(ArBin: A) : Elemento
{ Post: raiz ← e }

{ Pre: ¬esNulo(A) }
**func** hijoIzquierdo(ArBin: A): ArBin
{ Post: hijoIzquierdo ← A1 }

{ Pre: ¬esNulo(A) }
**func** hijoDerecho(ArBin: A): ArBin
{ Post: hijoDerecho ← A2 }

## 4.3. Implementación

```
clase NodoAB
      atributos:
            privado:
                  Elemento: e
                  Apuntador a  NodoAB: hijoIzq, hijoDer
      métodos:
            público:
                  proc construir()
                  proc construir(Elemento: e; Apuntador a NodoAB: izq, der)
                  func getElemento(): Elemento
                  func getHijoIzquierdo(): Apuntador a  NodoAB
                  func getHijoDerecho(): Apuntador a NodoAB
                  proc setElemento(Elemento: e)
                  proc setHijoIzquierdo(Apuntador a NodoAB: p)
                  proc setHijoDerecho(Apuntador a NodoAB: p)
fclase

clase ArBin
      atributos:
            privado:
                  Apuntador a NodoAB: nodoRaiz
      métodos:
            privado:
                  estático func copiarNodos(Apuntador a NodoAB: ptrNodo): Apuntador
                        a NodoAB
            público:
                  proc construir()                          // nulo
                  proc construir(Elemento: e; ArBin: A1, A2) // crear
```

```
                proc copiar(ref ArBin: A)
                func esNulo(): Lógico
                func raiz(): Elemento
                func hijoIzquierdo(): ArBin
                func hijoDerecho(): Arbin
                proc destruir()
fclase
```

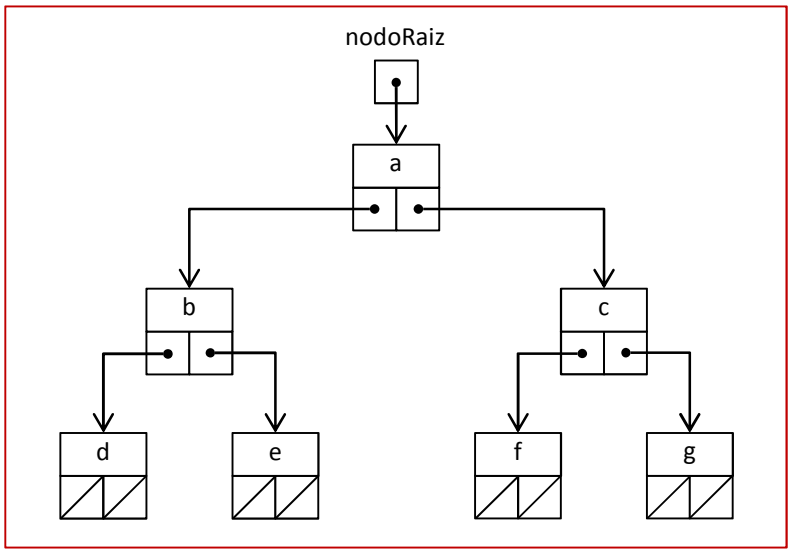



Figura 4. Representación gráfica de la estructura de un árbol binario.

```
proc ArBin::construir()
inicio
      nodoRaiz ← nulo
fproc

estático func ArBin::copiarNodos(Apuntador a NodoAB: ptrNodo): Apuntador a NodoAB
var
      Apuntador a NodoAB: nuevoNodo
inicio
      si(ptrNodo = nulo) entonces
            retornar(nulo)
      sino
            nuevoNodo.crear(NodoAB)
            nuevoNodo↑.construir(ptrNodo↑.getElemento(),
                  copiaNodos(ptrNodo↑.getHijoIzquierdo()),
                  copiaNodos(ptrNodo↑.getHermanoDerecho()))
            retornar(nuevoNodo)
      fsi
ffunc

proc ArBin::construir(Elemento: e; ArBin: A1, A2)
var
      ArBin: arbolAux
```

```
inicio
      nodoRaiz.crear(NodoAB)
      nodoRaiz↑.setElemento(e)
      nodoRaiz↑.setHijoIzquierdo(copiarNodos(A1.nodoRaiz))
      nodoRaiz↑.setHijoDerecho(copiarNodos(A2.nodoRaiz))
fproc

proc ArBin::copiar(ref ArBin: A)
inicio
      nodoRaiz ← copiarNodos(A.nodoRaiz)
fproc

func ArBin::esNulo(): Lógico
inicio
      retornar(nodoRaiz = nulo)
ffunc

func ArBin::raiz(): Elemento
inicio
      retornar(nodoRaiz↑.getElemento())
ffunc

func ArBin::hijoIzquierdo(): ArBin
var
      ArBin: hijo
inicio
      si(nodoRaiz ≠ nulo) entonces
            hijo.nodoRaiz ← copiarNodos(nodoRaiz↑.getHijoIzquierdo())
      sino
            hijo.nodoRaiz ← nulo
      fsi
      retornar(hijo)
ffunc

func ArBin::hijoDerecho(): ArBin
var
      ArBin: hijo
inicio
      si(nodoRaiz ≠ nulo) entonces
            hijo.nodoRaiz ← copiarNodos(nodoRaiz↑.getHijoDerecho())
      sino
            hijo.nodoRaiz ← nulo
      fsi
      retornar(hijo)
ffunc
```

## 4.4. Recorridos

**Recorrido en preorden**

- Visitar la raíz (y procesarla)).
- Recorrer el hijo izquierdo en preorden
- Recorrer el hijo derecho en preorden

```
proc ArBin::preorden()
inicio
      recorridoPreorden(nodoRaiz)
fproc

// Procedimiento privado de la clase ArBin, para uso del procedimiento preorden()
estático proc ArBin::recorridoPreorden(Apuntador a NodoAB: ptrRaiz)
var
      Apuntador a  NodoAB: aux
inicio
      si(ptrRaiz ≠ nulo) entonces
            ptrRaiz↑.getElemento().imprimir() // imprimir() debe estar definido
            recorridoPreorden(ptrRaiz↑.getHijoIzquierdo())
            recorridoPreorden(ptrRaiz↑.getHijoDerecho())
      fsi
fproc
```

**Recorrido en inorden**

- Recorrer el hijo izquierdo en inorden.
- Visitar la raíz (y procesarla).
- Recorrer el hijo derecho en inorden.

```
proc ArBin::inorden()
inicio
      recorridoInorden(nodoRaiz)
fproc

// Procedimiento privado de la clase ArBin, para uso del procedimiento inorden()
estático proc ArBin::recorridoInorden(Apuntador a NodoAB: ptrRaiz)
var
      Apuntador a  NodoAB: aux
inicio
      si(ptrRaiz ≠ nulo) entonces
            recorridoInorden(ptrRaiz↑.getHijoIzquierdo())
            ptrRaiz↑.getElemento().imprimir()
      recorridoInorden(ptrRaiz↑.getHijoDerecho())
      fsi
fproc
```

**Recorrido en postorden**

- Recorrer el hijo izquierdo en postorden
- Recorrer el hijo derecho en postorden
- Visitar la raíz (y procesarla)

```
proc ArBin::postorden()
inicio
      recorridoPostorden(nodoRaiz)
fproc

// Procedimiento privado de la clase ArBin, para uso del procedimiento postorden()
estático proc ArBin::recorridoPostorden(Apuntador a NodoAB: ptrRaiz)
var
      Apuntador a  NodoAB: aux
inicio
      si(ptrRaiz ≠ nulo) entonces
            recorridoPostorden(ptrRaiz↑.getHijoIzquierdo())
            recorridoPostorden(ptrRaiz↑.getHijoDerecho())
            ptrRaiz↑.getElemento().imprimir()
      fsi
fproc
```

**Recorrido por niveles**

**Ejercicio:**

1. Diseñe un procedimiento que haga el recorrido por niveles de un árbol binario.

## 5. Arboles de Sintaxis

Un árbol de sintaxis, también llamado árbol de sintaxis abstracto, es un árbol binario que permite representar expresiones sin ambigüedad. Las expresiones pueden ser aritméticas, lógicas, relacionales o sentencias propias de un lenguaje de programación. Por ejemplo la siguiente expresión en el lenguaje de programación C, `posicion = inicial + velocidad * 60`, se puede ver como el árbol:

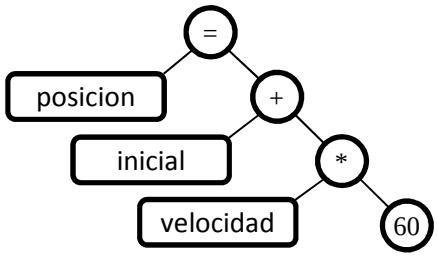



Figura 5. Árbol de sintaxis de la expresión posicion = inicial + velocidad * 60.

Estos árboles son usados por los compiladores como estructuras intermedias para la representación del código fuente, y también son usados por los intérpretes al ejecutar un programa. (Aho *et al*, 1998; Lee, 2008).

Como se puede notar, en este tipo de árboles los nodos interiores son los operadores y las hojas son los operandos.

En este tema se tomarán en cuenta solamente expresiones aritméticas. Los operadores que se considerarán son los siguientes: suma (+), resta (–), multiplicación (*) y división (/). Los operandos serán constantes (secuencias de dígitos) o variables (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. Ejemplos: A, 15, (A + 15) * (B – C).

Nótese la naturaleza recursiva de la 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.

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.

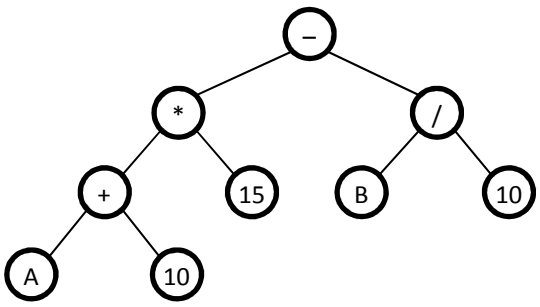



Figura 6. Árbol de sintaxis de la expresión la expresión ((A + 10) * 15) – (B / 10).

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.

**Ejercicios:**

1. Haga una función que permita evaluar un árbol de sintaxis, con operandos constantes. Asuma que cuenta con la operación **`func Cadena::obtenerReal(): Real`** que permite obtener el valor numérico de un objeto de la clase **Cadena**.

2. Realice una función que permita construir un árbol de sintaxis a partir de una expresión guardada en una lista de Cadenas. Asuma que cuenta con la operación **`func Cadena::esNumero(): Lógico`** que devuelve **`verdadero`** en caso de que el objeto sea un número y **`falso`** en caso contrario. Tome en cuenta que la expresión está bien formada y totalmente parentizada. Ejemplo: ((2 * 3) + (5 – (1 + 2))).

## 6. Arboles Binarios de Búsqueda (ABB)

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 o iguales 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 A = <*e*, $A_1$, $A_2$>, donde *e* ∈ Elemento y $A_1$, $A_2$ ∈ ArBin es de búsqueda si y solo si es nulo o cumple con las siguientes propiedades:

- *e* > elementoMaximo($A_1$)
- *e* ≤ elementoMinimo($A_2$)
- $A_1$ y $A_2$ también son ABBs.

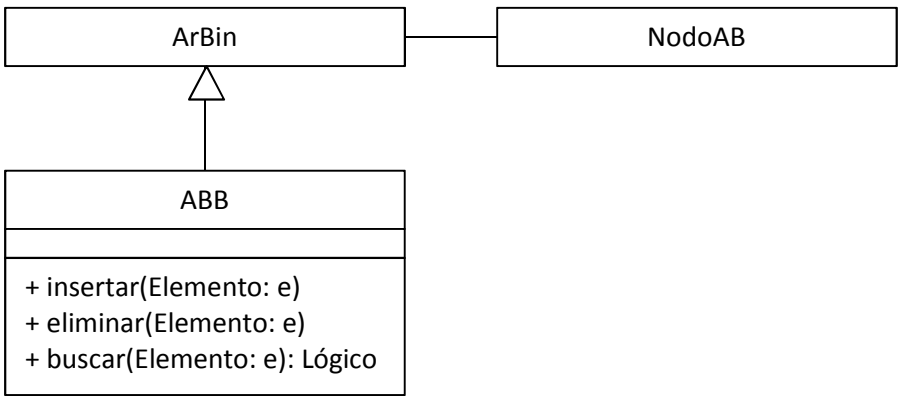



Figura 7. Diagrama de clases de un árbol binario de búsqueda.

## 6.1. 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.

```
// Función Privada
func ABB::buscarNodo(Elemento: e): Apuntador a NodoAB
var
      Apuntador a NodoAB: act
inicio
      act ← nodoRaiz
      mientras(act ≠ nulo ∧ act↑.getElemento() ≠ e) hacer
            si(e < act↑.getElemento()) entonces
                  act ← act↑.getHijoIzquierdo()
            sino
                  act ← act↑.getHijoDerecho()
            fsi
      fmientras
      retornar(act)
ffunc

func ABB::buscar(Elemento: e): Lógico
var
      Apuntador a NodoAB: ptrNodo
inicio
      ptrNodo ← instancia.buscarNodo(e)
      retornar(ptrNodo ≠ nulo)
ffunc
```

## 6.2. Inserción

```
// Procedimiento Privado
estático proc ABB::insertarNodo(ref Apuntador a NodoAB: ptrRaiz;  Elemento: e)
var
      Apuntador a NodoAB: nodo
inicio
      si(ptrRaiz = nulo) entonces
            ptrRaiz.crear(NodoAB)
            ptrRaiz↑.construir()
            ptrRaiz↑.setElemento(e)
      sino
            si(e < ptrRaiz↑.getElemento()) entonces
                  si(ptrRaiz↑.getHijoIzquierdo() = nulo) entonces
                        nodo.crear(NodoAB)
                        nodo↑.construir()
```

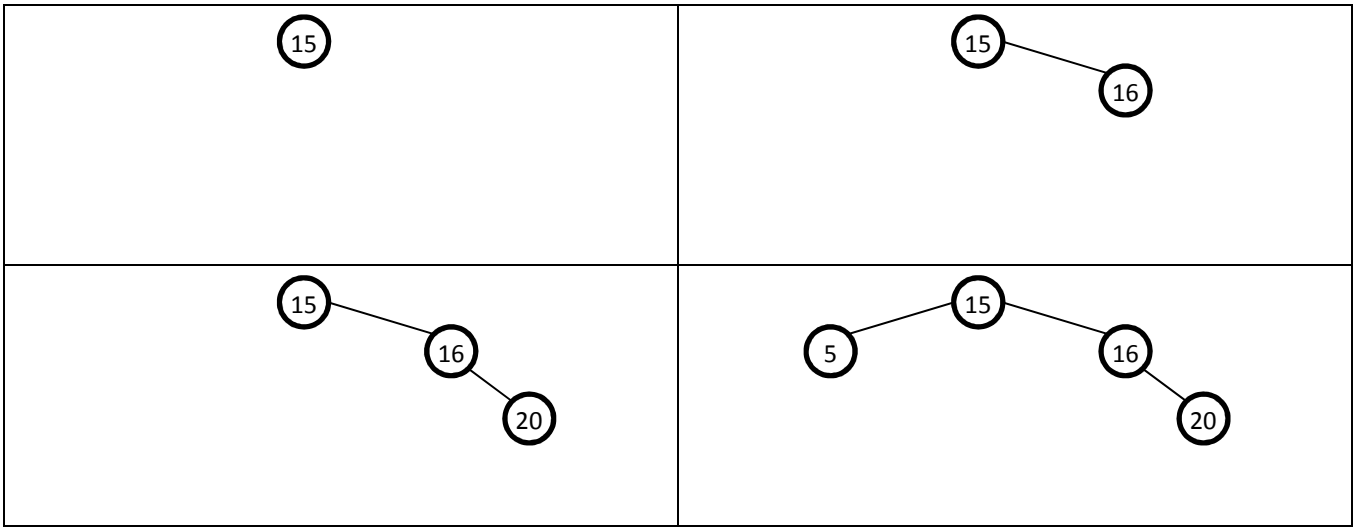
```
                        nodo↑.setElemento(e)
                        ptrRaiz↑.setHijoIzquierdo(nodo)
                  sino
                        insertarNodo(ptrRaiz↑.getHijoIzquierdo(), e)
                  fsi
            sino
                  si(ptrRaiz↑.getHijoDerecho() = nulo) entonces
                        nodo.crear(NodoAB)
                        nodo↑.construir()
                        nodo↑.setElemento(e)
                        ptrRaiz↑.setHijoDerecho(nodo)
                  sino
                        insertarNodo(ptrRaiz↑.getHijoDerecho(), e)
                  fsi
            fsi
      fsi
fproc

proc ABB::insertar(Elemento: e)
inicio
      { Para la versión que no permite elementos repetidos se debe aplicar
        primero el algoritmo de búsqueda }
      insertarNodo(nodoRaiz, e)
fproc
```

Se desean insertar los siguientes elementos en un ABB nulo:

15 16 20 5 12 18 3 23 10 13

La secuencia de ABBs generados por cada inserción es:



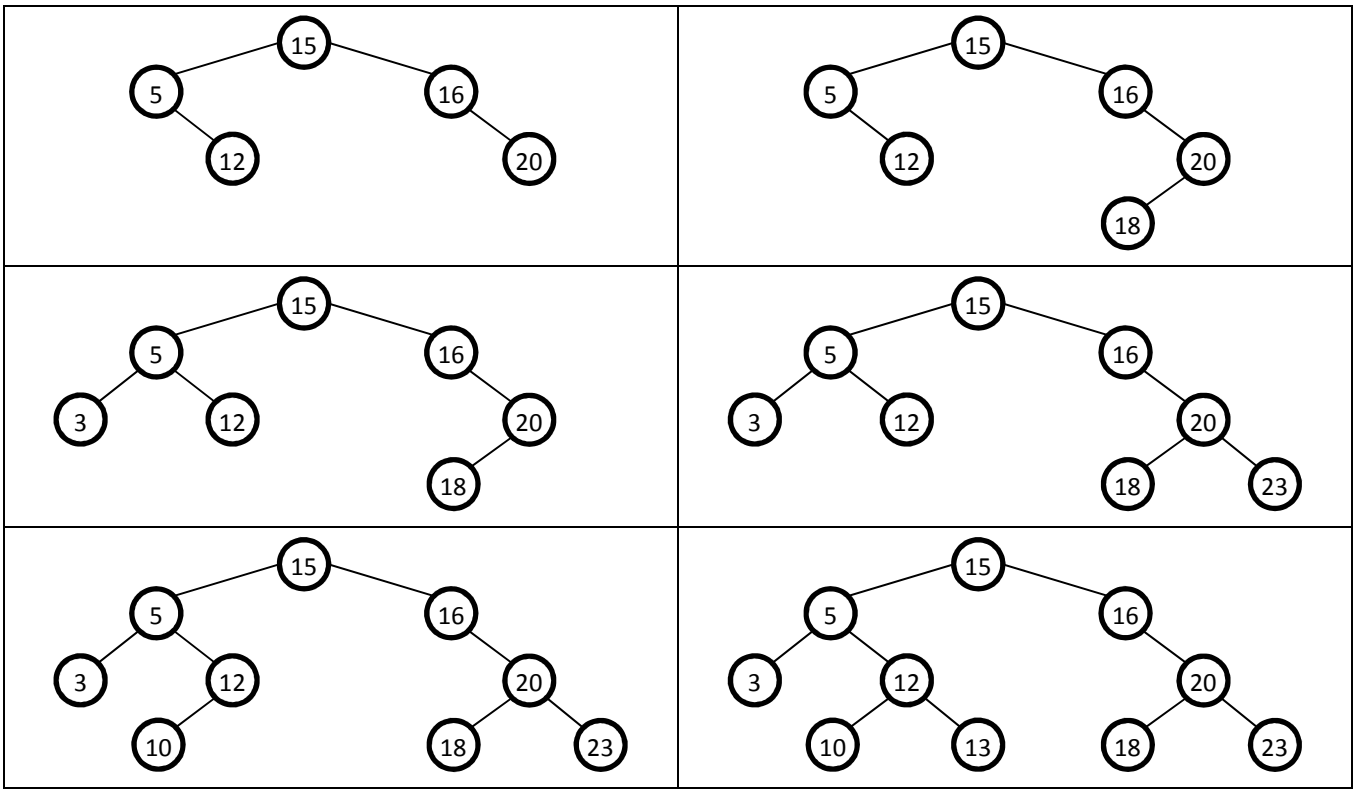



Figura 8. Proceso de inserción en un ABB.

## 6.3. Eliminación en un ABB

La eliminación de un nodo en un árbol binario de búsqueda es un poco más complicada que la inserción, dado que exige enlazar los subárboles del nodo eliminado en una posición correcta del árbol. Existen tres posibles casos cuando se elimina un nodo:

a) El nodo no tiene hijos (por lo tanto es un nodo hoja), en cuyo caso la eliminación es directa.
b) El nodo tiene un solo hijo, en cuyo caso esta pasa a sustituir al padre.
c) El nodo tiene dos hijos, en cuyo caso se debe buscar en su hijo derecho, el nodo que contenga el elemento con menor valor, el cual pasará a tomar su lugar.

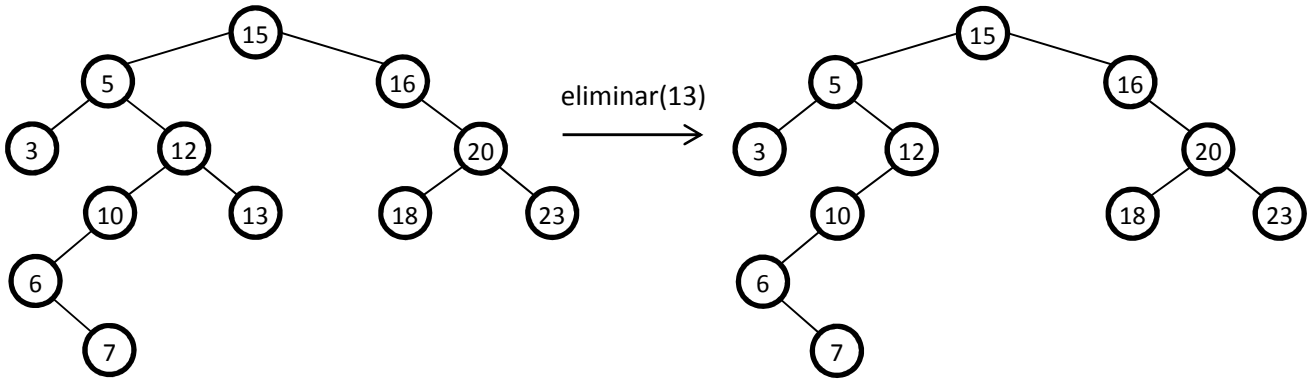



Figura 9. Eliminación de un nodo sin hijos.

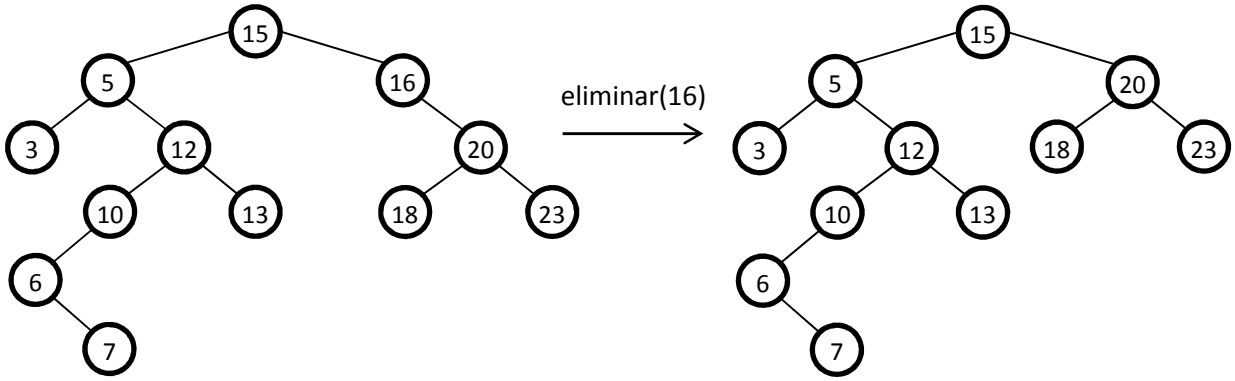



Figura 10. Eliminación de un nodo con un solo hijo.

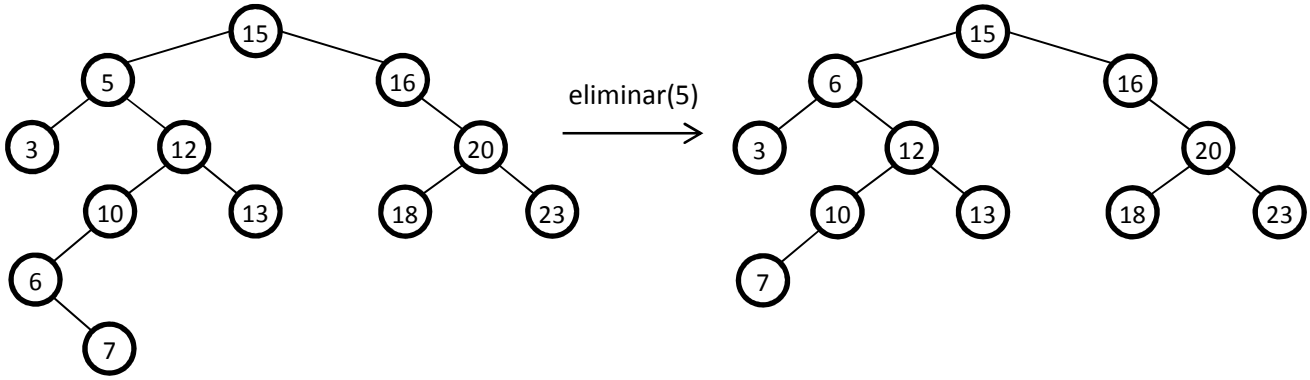



Figura 11. Eliminación de un nodo con dos hijos.

**Ejercicios:**

1. Realice una función que compruebe si un árbol binario es de búsqueda.
2. Realice el procedimiento de eliminación de un nodo en un árbol binario de búsqueda.

## 7. Arboles 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 subárboles 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).

## 7.1. Árboles Perfectamente Balanceados

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

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

Por definición, el árbol nulo es un árbol perfectamente balanceado.

## 7.2. Árboles AVL

Es un árbol binario de búsqueda auto-balanceado. En un árbol AVL las alturas de los subárboles asociados a cualquier nodo difieren a lo sumo en uno; por esto también son llamados árboles balanceados por altura (Gupta *et al*, 2008). Este árbol toma su nombre de las iniciales de sus dos inventores, Adelson – Velskii y Landis quienes introdujeron el concepto en el año 1962.

Formalmente, sea A = <*e*, $A_1$, $A_2$> un árbol binario de búsqueda. Se dice que A es un árbol AVL si y sólo si satisface las siguientes condiciones:

- $|altura(A_1) - altura(A_2)| \leq 1$.
- $A_1$ y $A_2$ son árboles AVL.

Por definición, el árbol nulo es un árbol AVL.

**Condición de Equilibrio**

Los árboles AVL son árboles binarios de búsqueda con una condición de equilibrio: Para todo nodo del árbol, la altura de los subárboles izquierdo y derecho puede diferir a lo sumo en uno. De acuerdo a esto para cada nodo se define un factor de equilibrio (FE) que se calcula como sigue:

$$FE = altura(A_2) - altura(A_1)$$

Esto quiere decir que el factor de equilibrio para cualquier nodo en un árbol AVL es un número entero perteneciente al conjunto {-1, 0, 1}.

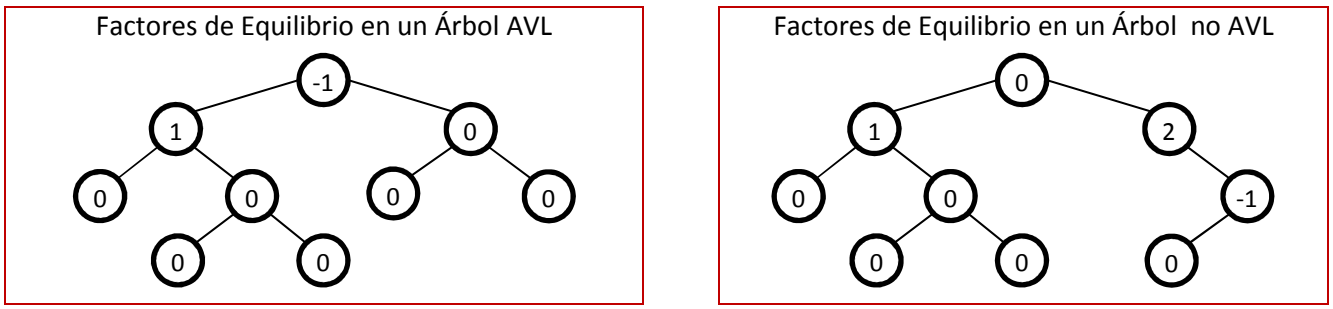



Figura 12. Factores de equilibrio de un árbol. A la izquierda, cuando es AVL. A la derecha, cuando no es AVL.

Cuando un nodo tiene un factor de equilibrio negativo se dice que es pesado a la izquierda, cuando su factor de equilibrio es 0 se dice que es balanceado y cuando su factor de equilibrio es positivo se dice que es pesado a la derecha.

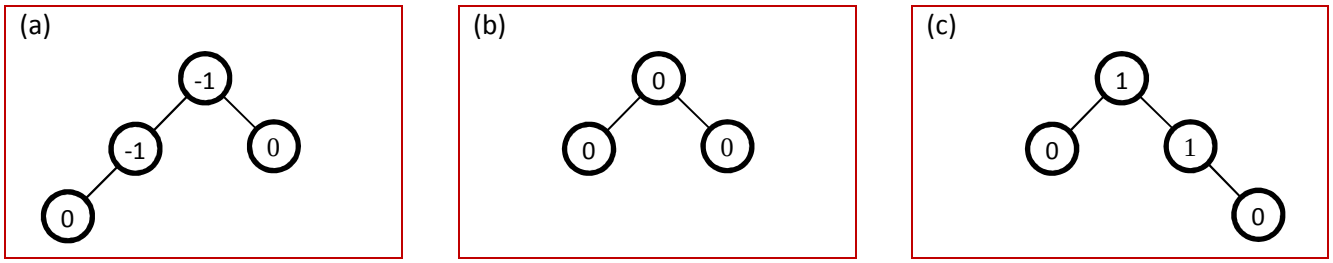



Figura 13. (a) Nodo pesado a la izquierda. (b) Nodo balanceado. (c) Nodo pesado a la derecha.

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.

**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 preserva la condición de equilibrio en cada nodo del árbol AVL. El problema potencial que puede producirse luego de una inserción es que el árbol con el nuevo nodo no sea AVL.

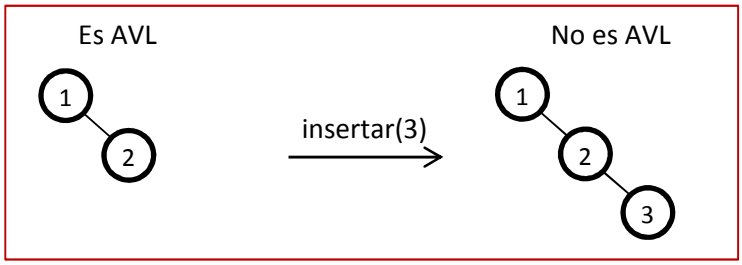



Figura 14. Problema al realizar el proceso de inserción en un árbol AVL.

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:

a) El elemento **x** fue insertado en el subárbol izquierdo del hijo izquierdo de **N**.
b) El elemento **x** fue insertado en el subárbol derecho del hijo izquierdo de **N**.
c) El elemento **x** fue insertado en el subárbol izquierdo del hijo derecho de **N**.
d) El elemento **x** fue insertado en el subárbol derecho del hijo derecho de **N**.

Dado que el primer y último caso son simétricos, producidos por una inserción hacia afuera con respecto a **N**, y se resuelven mediante una rotación simple; igualmente el segundo y el tercer caso son simétricos, producidos por una inserción hacia adentro con respecto a **N**, y se resuelven mediante una rotación doble.

*Rotación simple*

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

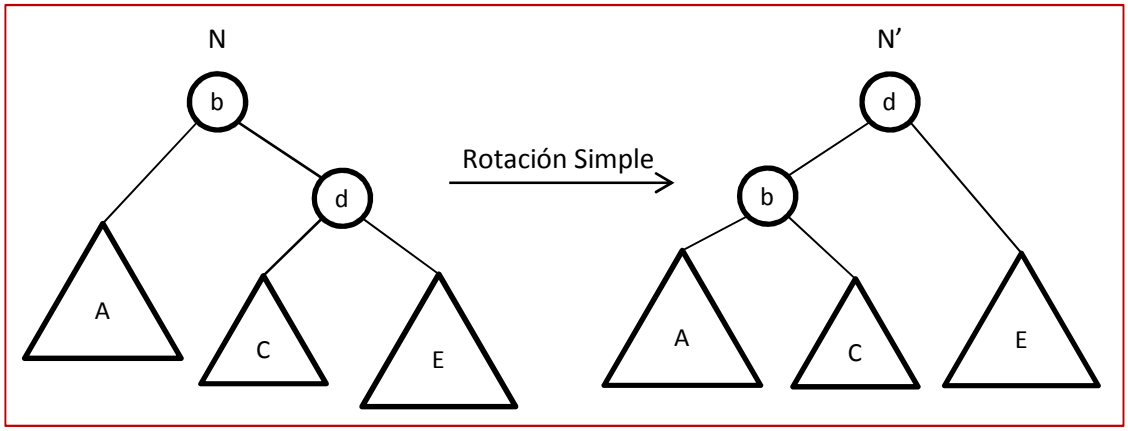



Figura 15. Rotación Simple.

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.

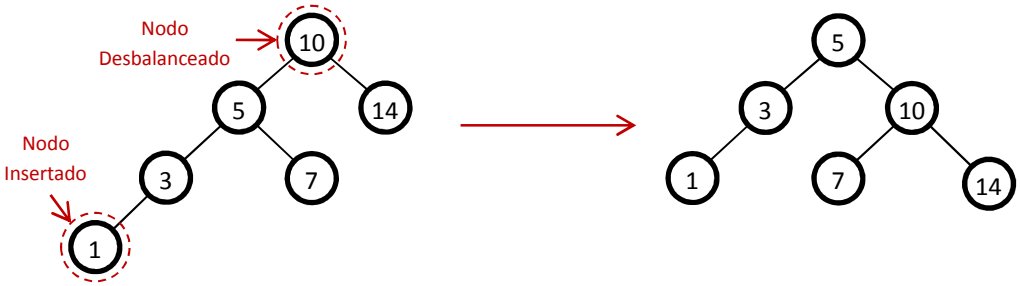



Figura 16. Caso a: Rotación Simple hacia la derecha.

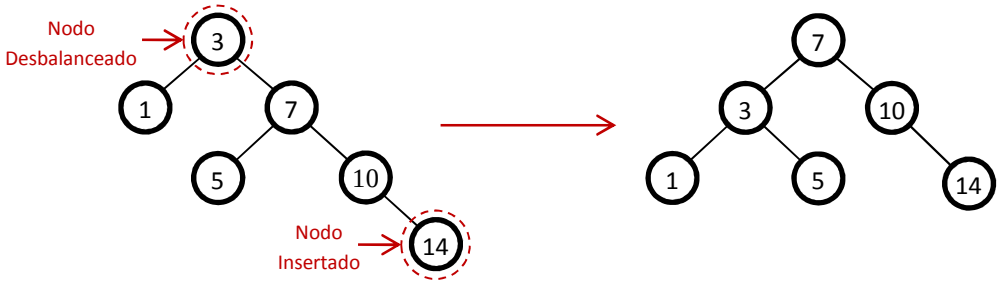



Figura 17. Caso d: Rotación Simple hacia la izquierda.

*Rotación Doble*

En el siguiente ejemplo se muestra un árbol AVL en el que se insertó un elemento en el hijo derecho del hijo izquierdo de la raíz, por lo tanto se necesitaría una rotación a la derecha:

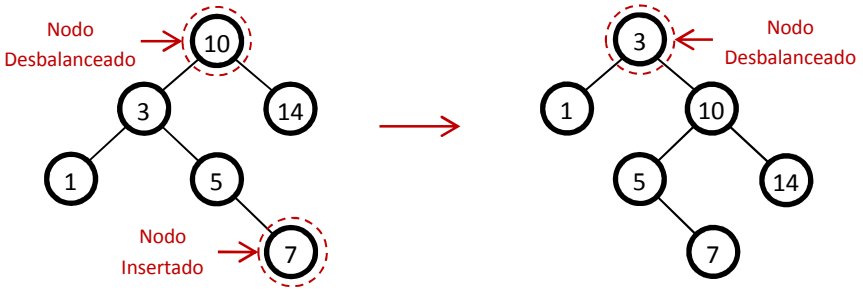



Figura 18. Rotación simple hacia la derecha que produce un árbol desbalanceado.

Sin embargo, la rotación a la derecha no logró la condición de equilibrio en el árbol. La razón por la cual la rotación simple a la derecha no funcionó es porque el subárbol izquierdo tenía un factor de equilibrio positivo, es decir, era pesado a la derecha, por lo cual una rotación a la derecha en un árbol con un subárbol izquierdo que es pesado a la derecha dará como resultado un árbol que seguirá estando desbalanceado.

De manera análoga, una rotación simple a la izquierda no funcionará en un nodo que tenga un subárbol derecho con un factor de equilibrio negativo, es decir, pesado a la izquierda.

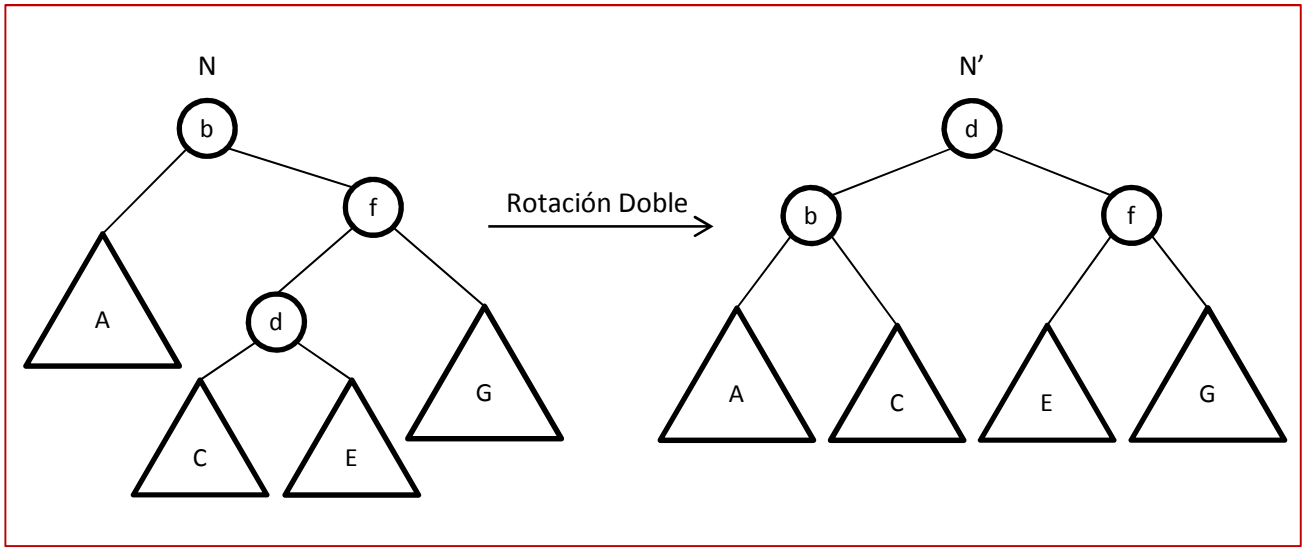



Figura 19. Rotación doble.

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 en el ejemplo general de rotación simple, este subárbol mantiene su posición relativa con esta rotación.

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.

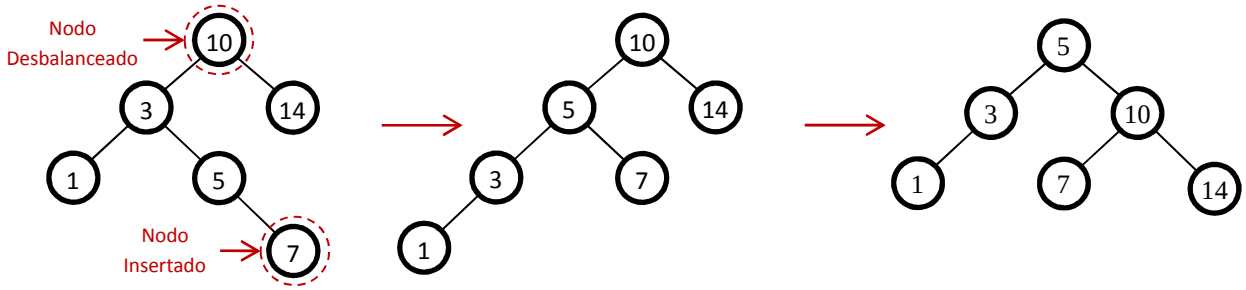



Figura 20. Caso b: Rotación doble hacia la derecha.

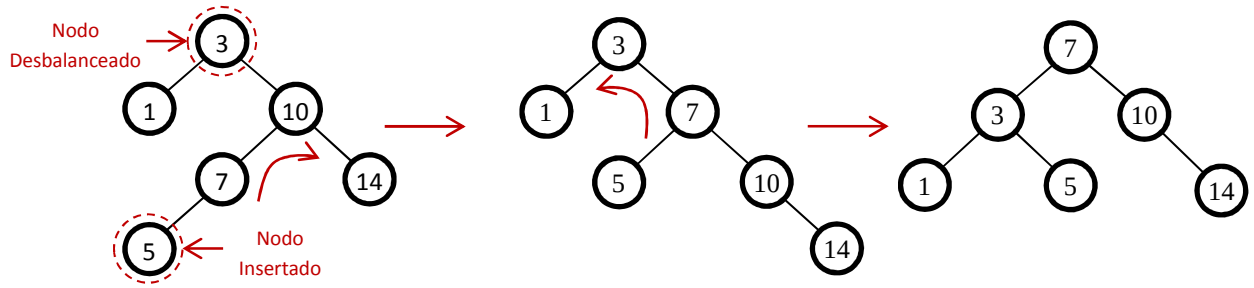



Figura 21. Caso c: Rotación doble hacia la izquierda.

**Algoritmos para la Rotación**

*Rotación Simple a la Derecha*

```
estático func AVL::rotacionSimpleDerecha(Apuntador a NodoAB: ptrRaiz): Apuntador a NodoAB
var
        Apuntador a NodoAB: aux
inicio
        aux ← ptrRaiz
        ptrRaiz ← ptrRaiz↑.getHijoIzquierdo()
        aux↑.setHijoIzquierdo(ptrRaiz↑.getHijoDerecho())
        ptrRaiz↑.setHijoDerecho(aux)
        retornar(ptrRaiz)
ffunc
```

*Rotación Doble a la Derecha (Rotación Izquierda-Derecha)*

```
estático func AVL::rotacionDobleDerecha(Apuntador a NodoAB: ptrRaiz): Apuntador a NodoAB
inicio
        ptrRaiz↑.setHijoIzquierdo(rotacionSimpleIzquierda(ptrRaiz↑.getHijoIzquierdo()))
        retornar(rotacionSimpleDerecha(ptrRaiz))
ffunc
```

**Ejercicios:**

1. Realice el algoritmo de rotación simple hacia la izquierda.
2. Realice el algoritmo de rotación doble hacia la izquierda.

## 8. Referencias


- Aho, A. V., Sethi, R. y Ullman, J. D. (1998). Compiladores: Principios, Técnicas y Herramientas. México: Addison Wesley Iberoamericana S.A.

- Gupta, P., Agarwal, V. y Varshney, M. (2008). Design and Analysis of Algorithms. New Delhi, India: PHI Learning Private Limited.

- Lee, K. D. (2008). Programming Languages. An active Learning Approach. New York, USA: Springer Science + Business Media.

- Martínez, A. A. y Rosquete, D. H. (2009). NASPI: Una Notación Algorítmica Estándar para la Programación Imperativa. Télématique. 8(3). 55 – 74.

- Ottogalli, K., Martínez, A. y León L. (2011). NASPOO: Una Notación Algorítmica Estándar para la Programación Orientada a Objetos. Télématique. 10(1). 81 – 102.