ND 2015-XX
Lecturas en Ciencias de la Computación
ISSN 1316 – 6239

# Algoritmos y Estructuras de Datos

Esmitt Ramírez J.
Marzo 2015

## Parte I

# Árboles

Una vez estudiada las estructuras de datos lineales como pilas y colas y con experiencia en recursión, estudiaremos el tipo de dato llamado Tree (árbol). Los árboles son empleados en muchas áreas de las Ciencias de la Computación, que se incluyen sistemas operativos, gráficos, sistemas de base de datos, y redes de computadoras. Las estructuras de datos Tree tiene mucho en común con su equivalente en botánica. Un tree tiene raíz, ramas y hojas. La principal diferencia radica en que la raíz se encuentra en la parte superior y las hojas en la parte inferior.

Conceptualmente, el árbol es una estructura de datos jerárquica que puede ser definida recursivamente como una colección de nodos, y cada nodo tiene un valor junto con una lista que referencia a un conjunto de nodos. Del mismo modo, los árboles tienen como restricción que no existen elementos duplicados y ningún elemento apunta a la raíz. Pueden existir árboles vacíos y conjuntos de árboles (llamados bosques).

Un ejemplo se puede ver a continuación:

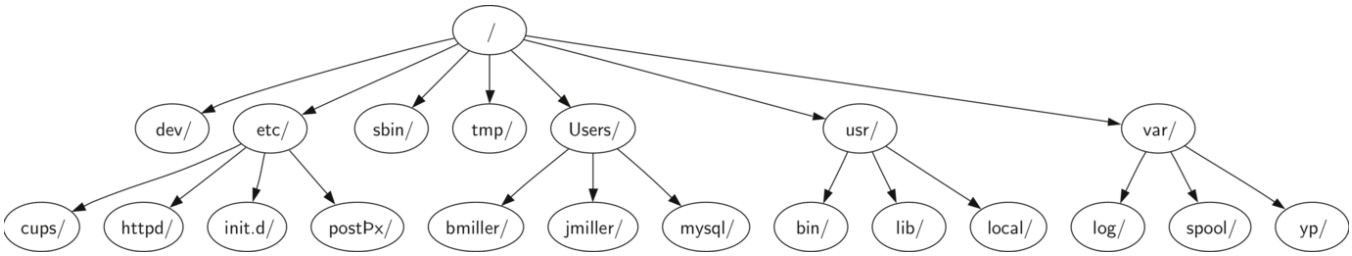


**Figura 1:** *El árbol representa el sistema de archivos en Unix. En un sistema de archivos existen archivos y directorios como una estructura de árbol.*

Nótese que empezando desde la raíz, se puede recorrer hasta la parte inferior del árbol en un único camino. Igualmente, se puede ver que todos los hijos de un nodo son independientes de los hijos de otro nodo. Así, cada nodo de la parte inferior (nodo hoja) es único, es decir, existe un solo camino para llegar a éste.

Como es sabido, es posible mover un directorio completo de un lugar del sistema de archivos a otros. Cuando se realiza este proceso, todos los hijos asociados q dicho nodo se moverán (a este conjunto de nodos se denomina subárbol). Por ejemplo, es posible mover el sub-arbol que empieza en /etc/ (quitarlo del nodo /) y colocarlo por debajo de /usr. Así, la ruta de acceso a httpd cambiará, quedando /usr/etc/http/ sin afectar el contenido o a cualquier directorio hijo de http.

Otro ejemplo basado en un sistema de árbol es una página Web escrita en HTML. Para el siguiente código en HTML:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>Premio Nobel</title>
</head>
<body>
<h1>Los candidatos al premio son:</h1>
<ul>
    <li>Homer J. Simpson</li>
    <li>Peter Griffin</li>
</ul>
<h2><a href="http://www.fakepage.org/voting">Consultar las votaciones</a><h2>
</body>
</html>
```

Su árbol asociado se muestra en la Fig. 21.

## 1 Definiciones

Formalmente, un árbol es un grafo conexo acíclico cuyos nodos se relacionan mediante una jerarquía.

En su implementación, un árbol de tipo $T$ es una estructura homogénea producto de la concatenación de un elemento de tipo $T$ junto con un número finito de árboles disjuntos (subárboles). Una forma particular de un árbol es una estructura vacía. Un árbol puede ser representado por una estructura estática (arreglos/registros/clases) ó dinámica (apuntadores/listas).

De forma recursiva, un árbol es una colección de nodos $T_1, T_2, \ldots, T_k$ del mismo tipo tal que:

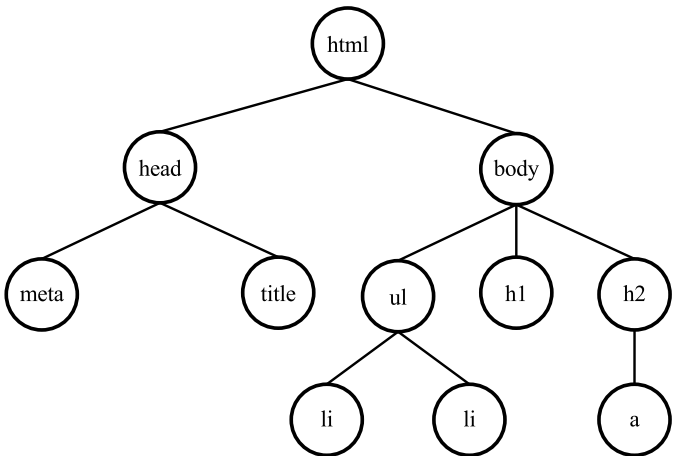


**Figura 2:** *Representación en forma de árbol del ejemplo de código en HTML.*

- Si $k == 0$, entonces el árbol es vacío
- Si $k > 0$, entonces existe un nodo especial llamado raíz (generalmente el primero de la definición, i.e. $T_1$) y los demás nodos forman parte de $n \geq 0$ conjuntos disjuntos que a su vez son árboles. Estos árboles se denominan subárboles del nodo raíz

**Conceptos asociados en la estructura Tree**

**Nodo -** Es la parte fundamental del árbol. Es posible que contenga un identificador (llamado key) e información adicional (*payload*)

**Enlace**– Algunas veces llamadas ramas, es otra parte fundamental del árbol, un enlace conecta dos nodos y muestra la relación que existe entre éstos. Cada nodo, con excepción de la raíz, está conectado con exactamente un enlace entrante desde otro nodo. Cada nodo puede tener muchos enlaces salientes

**Raíz** – La raíz de un árbol es el único nodo del árbol que no tiene enlaces entrantes. Por ejemplo, el nodo *html*

**Hijos** – El conjunto de nodos $c$ que tienen enlaces entrantes desde un mismo nodo $n$ se denominan hijos de $n$. Por ejemplo, los nodos *meta* y *title* son hijos del nodo *head*.

**Padre** – El concepto inverso a Hijo.

**Hermanos** – Nodos con el mismo padre.

**Camino** – Un camino es una lista ordenada de nodos conectadas por enlaces. Por ejemplo *html* $\rightarrow$ *body* $\rightarrow$ *h1*

**Longitud de un camino** – Es el número de veces que se debe aplicar la relación padre-hijos entre dos nodos que forman un camino

**Descendiente** – Un nodo alcanzable por un proceso repetitivo empleando los enlaces desde padres a sus hijos (un camino).

**Ancestro -** A veces llamado antecesor, es un nodo alcanzable por un proceso repetitivo empleando los enlaces desde los hijos a sus padres (un camino)

**Subárbol** – Un subárbol es un conjunto de nodos y enlaces formados por un padre y todos los descendientes de ese padre.

**Nodo Hoja** – También llamado nodo terminal ó externo. Es un nodo que no tiene hijos.

**Nodo Interno** – También llamado nodo no-terminal. Es un nodo con al menos un hijo (no es hoja).

**Grado** – Es el número de subárboles de un nodo.

**Grado de un árbol** – Es el máximo grado de todos los nodos del árbol, i.e. la cantidad máxima de hijos que soporta cada nodo

**Nivel** – Es la longitud del camino desde la raíz hasta un nodo. Si un árbol solo contiene la raíz, su nivel es 0.

**Altura** – La altura de un árbol se cuenta como el número de enlaces desde el nodo raíz hasta la hoja más lejana, es decir, el máximo nivel de cualquier nodo en el árbol.

**Profundidad** – La longitud máxima del camino desde un nodo a cualquier de sus descendientes.

**Peso** – El peso de un árbol se refiere al número de nodos que contiene el árbol.

**Bosque** – Un bosque es un conjunto de $n \geq 0$ árboles disjuntos

## 2 General Tree

Una estructura General Tree o árbol general $T$ es un conjunto finito de uno o más nodos donde existe un nodo designado $r$ llamado raíz de $T$, y los nodos restantes son particionados en $n \geq 0$ conjuntos disjuntos $T_1, T_2, \ldots, T_k$ donde cada uno es un árbol y cuyas raíces $r_1, r_2, \ldots, r_k$ son hijos de $r$. En la Fig. 22 se muestra un ejemplo de árbol general de altura 3, grado 4, peso 9, con nodo raíz 2, conteniendo 6 hojas/nodos externos/nodos terminales y 3 nodos internos/no-terminales.

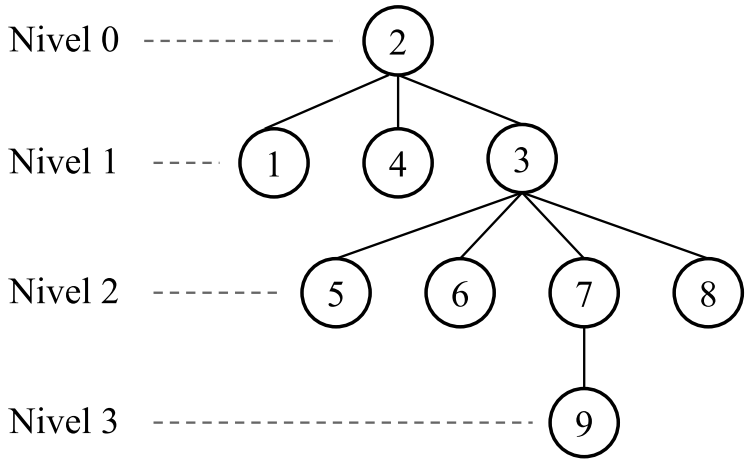


**Figura 3:** *Ejemplo de un árbol general de grado 4 (cuaternario) y altura 3.*

Nótese que el árbol general de grado $g$ puede desde 0 hijos (nodo terminal) a $g$ hijos (nodo no-terminal) para cada nodo.

### 2.1 Especificación

Se muestra a continuación la especificación de la clase GenTree que representa a un árbol general.

```
class GenTree <T>
public:
  Constructor GenTree()      //crea un árbol vacío
  Destructor GenTree()       //destruye el árbol
  function GetRoot () : Node  //retorna el nodo correspondiente a la raíz
  function GetChild (Node idNode) : Node  //retorna el nodo del 1er hijo de idNode (según convención, ↩
      el más a la izq)
  function GetRBrother(Node idNode) : Node  //retorna la raíz del 1er subárbol a la der de idNode con ↩
      el mismo padre
  function ExistRBrother(Node idNode) : Boolean //indica si tiene hermano derecho
  function GetValue(Node idNode) : T  //retorna el contenido del nodo idNode
  function GetParent(Node idNode) : Node  //retorna el padre de idNode
  function isLeaf(Node idNode) : Boolean  //retorna verdad/falso si idNode es hoja
  function isEmpty() : Boolean  //retorna verdad/falso si el árbol no contiene nodos
  function Insert(Node idNode)  //inserta el nodo idNode al árbol (no se especifica dónde)
  function Delete(Node idNode)  //elimina el nodo idNode del árbol
end
```

Un ejemplo para el cálculo del nivel en un GenTree es como sigue:

```
function Level(Node<T> idNode) : Integer
  Integer iLevel = 0
  Node<T> nTemp = idNode
  while nTemp  != EMPTY do
    nTemp = GetParent(nTemp)
    iLevel = iLevel + 1
  end
  return iLevel
end
```

Dado que no se conoce la implementación aún del árbol general, se emplea idNode el cual representa un identificador de un nodo del árbol. La idea es algoritmo es ir subiendo hasta la raíz (usando el padre) e ir contando hasta que no se pueda más (nodo raíz). Nótese que si se invoca la función con el nodo más "profundo" (el de mayor nivel) es equivalente a la altura del árbol.

### 2.2 Implementación

Para la implementación del tipo GenTree se requiere de una estructura tipo Node que almacene el valor de la información del árbol (el tipo T) e información de los hijos de dicho nodo. Es importante destacar que cada nodo representa a un árbol, o subárbol, por lo que tiene 0 o más hijos. Una primera implementación se puede definir como un arreglo de punteros al tipo Node.

```
class Node<T>
public:
  T tInfo
  Array aChild of Node<T>* [1..N]
end

class GenTree<T>
private:
  Node<T>* pRoot
public:
  ... //the public functions
end
```

En dicha implementación, se requiere definir el número máximo de hijos que puede tener un nodo. Es decir, si se conoce el grado del árbol, entonces el número máximo de dicho árbol corresponde con el grado del árbol. Sin embargo, es posible que en un momento dado una gran parte de los nodos tenga un número de nodos menor al grado del árbol, implicando que se desperdicie memoria en dicha implementación. Así, una mejor implementación requiere solo crear/reservar espacio en memoria de los nodos que son creados. Para ello, una implementación en donde los hijos de un nodo se construyen como una lista basada en apuntadores resulta eficiente. A continuación se muestra un ejemplo de ello.

```
class Node<T>
public:
  T tInfo
  List<T> L
end

class GenTree<T>
private:
  Node<T>* pRoot
public:
  ... //the public functions
end
```

### 2.3 Recorridos

El proceso de recorrido de un árbol consiste en visitar/recorrer/consultar o realizar una acción en cada nodo tal que no modifique la estructura del árbol (e.g. imprimir, contar, comparar). Esta visita se realiza bajo cierto orden y la acción se efectúa una sola vez por nodo. Básicamente se pueden definir dos recorridos para árboles generales: preorder o postorder.

El recorrido en preorder consiste en recorrer primeramente el nodo raíz, y luego los nodos que contienen a los hijos desde el nodo más a la izquierda hasta el nodo más a la derecha. El recorrido postorder primero recorre los nodos hijos de derecha a izquierda y luego el nodo raíz.

En el ejemplo mostrado en la Fig. 22 sus respectivos recorridos son:

Preorder: 2, 1, 4, 3, 5, 6, 7, 9, 8

Postorder: 1, 4, 5, 6, 9, 7, 8, 3, 2

## 3 Binary Tree - BT

Es posible representar diversos comportamientos empleando decisiones de aceptar/rechazar, si/no, fuera/dentro, etc. En Ciencias de la Computación, se suele emplear estructuras de datos que permitan manejar dichos comportamientos para solucionar diversos problemas. Por ejemplo, el lanzamiento de una moneda (i.e. moneda ideal) solo tiene dos posibilidades: cara (H) o sello (T). Así, el evento de lanzamiento de una moneda se puede representar como un árbol de decisión. Por ejemplo el lanzamiento de una moneda se puede representar gráficamente como se muestra en la Fig. 23

Por cada lanzamiento es posible obtener H o T, entonces partiendo desde la raíz se muestran las 8 posibles combinaciones (de izquierda a derecha): HHT, HHT, HTH, HTT, THH, THT, TTH y TTT.

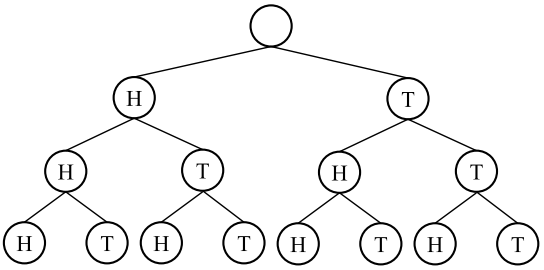


**Figura 4:** *Ejemplo del lanzamiento de una moneda 3 veces seguidas.*

### 3.1 Definiciones

En la teoría de árboles, un árbol es llamado k-ario si cada nodo contiene como máximo k hijos. Un caso particular es el árbol binario donde $k = 2$. Si todos los nodos del árbol, a excepción de las hojas, posee exactamente $k$ hijos entonces dicho árbol es completo.

### 3.2 Recorridos

Un recorrido en un árbol binario implica recorrer todos sus nodos de forma sistemática y asegurando que se aplica una operación sobre cada uno de éstos solo una vez. Existen tres tipos básicos de recorridos: Preorder, Inorder (también llamado Simétrico), y Postorder.

El recorrido preorder (pre-orden) consiste en visitar/evaluar el nodo raíz del árbol o subárbol, luego su subárbol izquierdo y finalmente su subárbol derecho. Claramente, el proceso de visitar/evaluar los subárboles consiste en aplicar la misma operación mientras el nodo no sea nulo o vacío. Una posible implementación es:

```
void Preorder (IdNode root)
  if root == EMPTY then
    Visit (root)
    Preorder (Left(root))
    Preorder (Right(root))
  end
end
```

El recorrido inorder (in-orden o simétrico) consiste en visitar/evaluar primero el subárbol izquierdo del nodo, luego el valor del nodo raíz del árbol o subárbol, y finalmente su subárbol derecho. Al igual que el recorrido en preorder, el proceso de visitar/evaluar los subárboles consiste en aplicar la misma operación mientras el nodo no sea nulo o vacío. Una posible implementación es:

```
void Inorder (IdNode root)
  if root == EMPTY then
    Inorder (Left(root))
    Visit (root)
    Inorder (Right(root))
  end
end
```

El recorrido postorder (post-orden) consiste en visitar/evaluar primero los subárboles izquierdo y derecho del nodo (en ese orden) y luego el valor del nodo raíz del árbol o subárbol. Al igual que los recorridos anteriores el proceso de visitar/evaluar los subárboles consiste en aplicar la misma operación mientras el nodo no sea nulo o vacío. Una posible implementación es:

```
void Postorder (IdNode root)
  if root == EMPTY then
    Postorder (Left(root))
    Postorder (Right(root))
    Visit (root)
  end
end
```

En la Fig. 24 se observa un ejemplo de árbol binario para representar una expresión aritmética. Al ejecutar los 3 recorridos sobre el árbol queda:

Preorder: + - A * B C D (expresión prefija)
Inorder: A - B * C + D (expresión infija)

Postorder: + - A * B C D (expresión postfija)

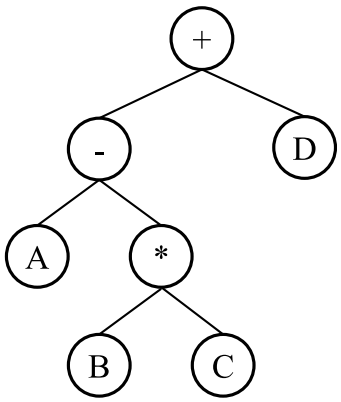


**Figura 5:** *Ejemplo de la representación de un BST para almacenar expresiones aritméticas.*

Adicionalmente, existe otro recorrido llamado por niveles (*level order*) que consiste en evaluar los nodos de izquierda a derecha empezando desde el nivel 0 del árbol hasta alcanzar su altura. Para el caso de la Fig. 24 el recorrido por niveles es: + - D A * B C.

Del mismo modo, es posible aplicar los recorridos antes mencionados de forma inversa. Por ejemplo, el recorrido inorder es LVR (Left-Value-Right) y el recorrido inorder inverso es RVL (Right-Value-Left).

### 3.3 Implementación

**Implementación basada en arreglos** La implementación de un BST empleando arreglos debe conocer el número máximo de nodos o estimarlo (estática o dinámica) tal que en cada posición se almacene un nodo. La idea es almacenar los hijos izquierdo y derecho, en ese orden, de un nodo $k$ en las posiciones $2 \times k + 1$ y $2 \times k + 2$ respectivamente. En la Fig. 25 se muestra un ejemplo de la implementación basada en arreglos.

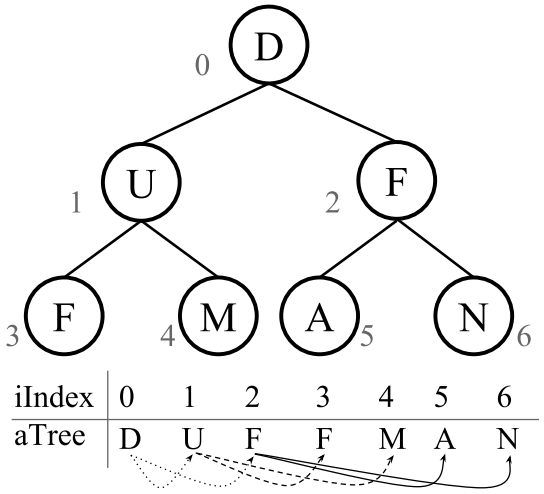


**Figura 6:** *Representación de un árbol completo del tipo Char almacenado dentro de un arreglo.*

El árbol *aTree* tiene 7 nodos con de altura 2, y la variable iIndex representa su posición a ser almacenada en dicho arreglo. La numeración es de arriba hacia abajo y de izquierda a derecha.

Esta representación es muy buena para árboles completos, pero muy mala para árboles degenerados.

**Implementación basada en apuntadores** La idea detrás de la implementación de un árbol binario basado en apuntadores es la creación de un nodo que contenga un puntero al hijo izquierdo, al hijo derecho y a la información/datos a almacenar. El puntero a la raíz apunta al nodo más arriba en el árbol. Los punteros derecho e izquierdo apuntan recursivamente a cada subárbol de cada lado, tal como se muestra en la figura 26. Se puede observar que en la raíz está el valor de 500 y como hijo

derecho al subárbol cuyo nodo raíz contiene al valor 569, y como hijo izquierdo el nodo (subárbol) que contiene el valor de 300. Los nodos hoja tienen los punteros a ambos hijos a nulo.

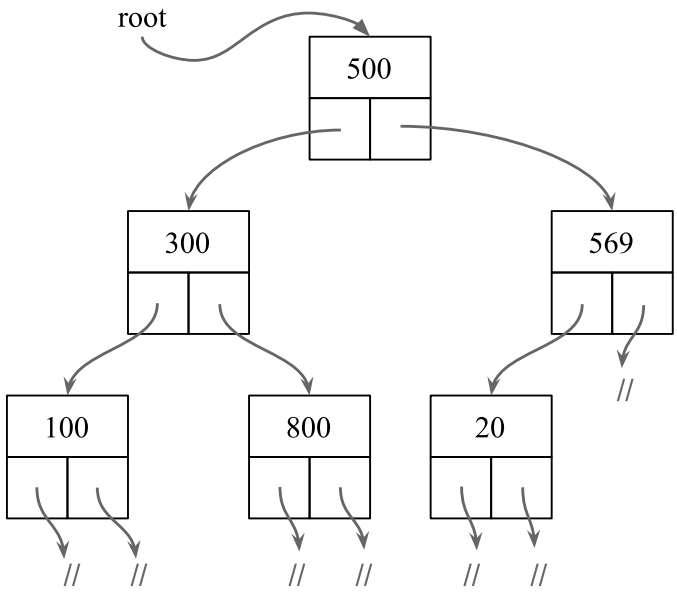


**Figura 7:** *Representación de un árbol empleando apuntadores. Cada nodo contiene un apuntador a su hijo derecho e izquierdo.*

Así es posible definir un árbol binario como una estructura formada por un nodo vacío o por un nodo simple donde los punteros izquierdo y derecho apuntan a un árbol binario. Entonces, solamente con una variable del tipo Node es posible construir el árbol. Su definición es:

```
class Node<T>
public:
  T tInfo
  Node<T>* pLeft
  Node<T>* pRight
end
```

En los lenguajes de programación que tienen soporte a apuntadores el valor de éstos pueden estar inicializados con NIL al momento de su creación o en caso contrario se le debe asignar dicho valor. Así, es posible construir una clase un poco más "segura" y en el enfoque orientado a objetos (*mutators* y *observers*) como:

```
class Node<T>
private:
  T tInfo
  Node<T>* pLeft
  Node<T>* pRight
public:
  Constructor Node()
    pLeft = NIL
    pRight = NIL
  end

  Constructor Node(T value)
    tInfo = value
  end

  Constructor Node(Node<T>* pRight, pLeft, T tInfo)
    this.pLeft = pLeft
    this.pRight = pRight
    this.tInfo = tInfo
  end

  Destructor Node()
    if pLeft != NULL then
      delete pLeft
    end
    if pRight != NULL then
      delete pRight
    end
```

```
  end

  void setRightChild(Node<T>* pRight)
    this.pRight = pRight
  end

  void setLeftChild(Node<T>* pLeft)
    this.pLeft = pLeft
  end

  void setInfo(T tInfo)
    this.tInfo = tInfo
  end

  Node<T>* getRightChild()
    return pRight
  end

  Node<T>* getLeftChild()
    return pLeft
  end

  void getInfo(T tInfo)
    return tInfo
  end

end
```

## 3.4 Algoritmos

A continuación se presentan una serie de algoritmos sencillos empleando árboles binarios.

### 3.4.1 Contar el número de nodos

Dado un árbol binario, determinar el número de nodos (su peso) que contiene. Para ello, el algoritmo planteado es muy simple basado en la idea que:

1. Si el árbol está vacío, contiene 0 nodos
2. Si el árbol no está vacío, contiene 1 nodo (el actual) más la cantidad de nodos que contenga su subárbol derecho y su subárbol izquierdo

El algoritmo se puede escribir como sigue:

```
function Count (Node<T>* pNode) : Integer
  if pNode == NIL then
    return 0
  else
    return 1 + Count(*pNode.pRight) + Count(*pNode.pLeft)
  end
end
```

La función retorna el número de nodos del árbol, o siendo equivalente, la cantidad de nodos que no fueron NIL.

### 3.4.2 Buscar el elemento mínimo

El problema de buscar el elemento mínimo consiste en encontrar el nodo con el valor de $tInfo$ mínimo. Dependiendo del tipo de dato de $tInfo$, para realizar este algoritmo se debe garantiza que exista la función de comparación "menor que".

Para este problema, se crean 2 funciones adicionales: una para obtener el mínimo entre dos valores, y otra para determinar si un nodo es hoja o terminal. Asumiendo que el tipo de dato del árbol es del tipo Integer, se puede escribir el algoritmo como sigue:

```
function Min(Integer a, Integer b) : Integer
  if a > b the
    return b
  end
  return a
end

function isLeaf(Node<Integer>* pNode) : Boolean
  return *pNode.pRight == NIL and *pNode.pLeft
end
```

```
function Min (Node<Integer>* pNode) : Integer
  if isLeaf(pNode) then
    return *pNode.tInfo
  else
    return Min(*pNode.tInfo, Min(Min(*pNode.pRight), Min(*pNode.pLeft)))
  end
end
```

Es importante destacar que debido al polimorfismo, es posible tener dos funciones de nombre $Min$ pero que ambas tienen parámetros distintos.

Así, la idea del algoritmo es básicamente obtener el valor mínimo entre el nodo actual (nodo no-terminal) y el valor mínimo de su subárbol derecho e izquierdo. La unidad mínima de un subárbol es cuando contiene un solo nodo, es decir, cuando es un nodo hoja.

### 3.4.3 Calcular la profundidad

Como se define en la sección 1, la profundidad es La longitud máxima del camino desde un nodo a cualquier de sus descendientes. Entonces, el algoritmo se en la idea de calcular la profundidad máxima entre el hijo derecho e hijo izquierdo. Entonces, el valor de la profundidad se define como el número de invocaciones recursivas empleando la relación de jerarquía hasta que se llegue a un nodo vacío (i.e. NIL).

```
function MaxDepth (Node<T>* pNode) : Integer
  if pNode == NIL then
    return 0
  else
    Integer iLDepth = MaxDepth(*pNode.pLeft)
    Integer iRDepth = MaxDepth(*pNode.pRight)
    if iLDepth > iRDepth then
      return iLDepth + 1
    else
      return iRDepth + 1
  end
end
```

En vez de utilizar el condicional, es posible implementar una función $Max$ que determine el máximo valor entre dos números.

### 3.4.4 Árboles binarios iguales

Determinar que dos árboles binarios son exactamente igual, en cuanto a valores y su estructura, es decir, que cada nodo tenga el mismo número de hijos/enlace en cada uno de sus niveles. El algoritmo verifica entonces la igualdad del valor de los nodos y la existencia del subárbol derecho e izquierdo para un mismo nodo.

```
function Equals (Node<T>* pNodeA, pNodeB) : Integer
  if pNodeA == NIL and pNodeB == NIL then
    return true
  elseif pNodeA != NIL and pNodeB != NIL then
    return *pNodeA.tInfo == *pNodeB.tInfo and Equals(*pNodeA.pLeft, *pNodeB.pLeft) and Equals(*pNodeA.↩
        pRight, *pNodeB.pRight)
  else
    return false
  end
end
```

Se destaca en el algoritmo que la comparación exacta se logra por el uso del comparador lógico *and*.

### 3.4.5 Suma de los nodos

Dado un BST de valores del tipo Integer, se quiere construir una función que sume todos los elementos de los nodos y lo retorne. Una posible función es como sigue:

```
function SumAll (Node<Integer>* pNode) : Integer
  if pNode == NIL then
    return 0
  else
    return *pNode.tInfo + SumAll(*pNode.pRight) + SumAll(*pNode.pLeft)
end
```

La función verifica en su caso base si es NIL o no. En la parte recursiva, invoca a la función con los nodos derecho e izquierdo. Ahora, si un nodo del árbol no tiene hijos, o solo tiene hijo derecho, o solo tiene hijo izquierdo entonces cuando se genere el

ambiente recursivo habrá de retornar 0 por el caso base (debido a que no aporta en la suma). Este enfoque puede resultar ineficiente debido a que si una de las ramas del nodo es NIL, entonces no es necesario invocar a la función. Por ejemplo, para un árbol completo de altura $h$ se realizarán $2^{h+1}$ invocaciones que siempre retornarán 0 (i.e. los nodos hojas).

Entonces, sería ideal primero verificar el número de hijos que posee un nodo para no hacer invocaciones que no aporten al cómputo (la suma final). A continuación se muestra una nueva versión donde se toma en cuenta dichos aspectos, siendo más eficientes en solo generar las invocaciones necesarias para el cálculo.

```
function isLeaf(Node<Integer>* pNode) : Boolean
  return *pNode.pRight == NIL and *pNode.pLeft
end

function SumAll (Node<Integer>* pNode) : Integer
  if pNode != NIL then
    if isLeaf(pNode) then
      return *pNode.tInfo
    elseif *pNode.pRight == NIL then
      return *pNode.tInfo + SumAll(*pNode.pLeft)
    elseif *pNode.pLeft == NIL then
      return *pNode.tInfo + SumAll(*pNode.pRight)
    else
      return *pNode.tInfo + SumAll(*pNode.pRight) + SumAll(*pNode.pLeft)
  end
  return 0  //solo se debería invocar si el árbol es vacío
end
```

## 4 Binary Search Tree - BST

Un tipo Binary Search Tree (BST), árbol binario de búsqueda, es un árbol binario con la característica de que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo $k$ son menores al valor del elemento almacenado en $k$, y que todos los elementos almacenados en el subárbol derecho de $k$ son mayores que el valor del elemento almacenado en $k$. La Fig. 27 muestra un ejemplo de dos BST empleando un mismo conjunto de números.

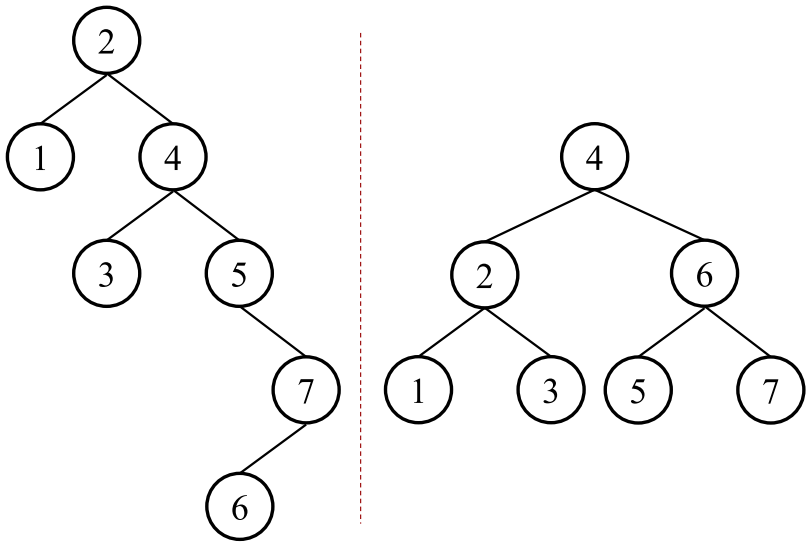


**Figura 8:** *Ejemplo de dos BST para un mismo conjunto de números.*

En un BST, para todo nodo existe una clave única que lo identifica, es decir, no hay repeticiones en el valor discriminante para la comparación. Un nodo puede almacenar diversos valores propios de la estructura de datos, pero requiere solo un valor que sirva de clave (key) y pueda ser aplicar el operador de comparación (mayor que, menor que). De esta forma existe una relación de orden total en el tipo asociado al key. Así por definición, para cada nodo las claves de los nodos de su subárbol izquierdo siempre son menores y las del subárbol derecho siempre son mayores.

El recorrido en inorder de un BST produce la secuencia de claves en orden ascendente. Por ejemplo, para los dos árboles de la Fig. 27 se produce la secuencia $1, 2, 3, 4, 5, 6, 7$. Por su lado, el recorrido inorder en reverso produce la secuencia en orden descendente.

La diferencia observada en los árboles del ejemplo se debe a que su forma viene dada por el orden en que aparecen las claves de la secuencia (del 1 al 7) al momento de construir el BST. A pesar de que los árboles tengan las mismas claves y el mismo

orden en recorrido inorder, ambos tuvieron una secuencia de inserción distinta. Para el primer árbol (izquierda) el primer valor de la secuencia fue el 2, para el segundo (derecha) el primer valor fue el 4.

## 4.1 Especificación

Las operaciones de Insert (inserción) y Lookup (búsqueda) en un árbol binario de búsqueda suelen ser rápidas. En promedio, un algoritmo de búsqueda en un árbol binario puede localizar un nodo en un árbol de $n$ nodos en un orden de complejidad de $log(N)$ (logaritmo base 2). Por lo tanto, un BST son estructuras ideales para problemas de **diccionario** donde un código/id es insertado y se busca su información asociada de forma eficiente. El comportamiento logarítmico es para el caso promedio, es posible que para un árbol en particular sea más lento (dependiendo de su forma).

A continuación se muestra una posible especificación para la definición de la clase BST.

```
class Node<T>
public:
  T tInfo //key
  Node<T>* pLeft
  Node<T>* pRight
end

class BST<T>
private:
  Node<T>* pRoot
public:
  Constructor BST()
  Destructor BST()
  function GetRoot() : Node<T> *
  function IsEmpty() : Boolean
  function Lookup(Node<T>* pNode, T tInfo) : Boolean      //puede retornar un Node<T>*
  function Insert (Node<T>* pNode, T tInfo) : Node<T> *   //puede retornar un Boolean
  void Delete(Node<T>* pNode, T tInfo)
end
```

Es posible que la operación de Lookup retorne un tipo $Node < T > *$ en vez de un tipo Boolean (igual con la función Insert). De esa forma permitirá localizar un elemento dentro del BST, o retornar NIL que indica que el elemento no existe.

## 4.2 Implementación

A continuación se muestra la implementación de la clase BST. Se mostrará las funciones con una breve explicación de cada una.

La implementación inicial de la clase BST consiste en el constructor, destructor, la función que retorna la raíz y verificar si el árbol es vacío o no. Todas estas funciones son muy similares en todas las estructuras dinámicas.

```
class BST<T>
public:

  Constructor BST()
    pRoot = NIL
  end

  Destructor BST()
  end

  function GetRoot() : Node<T> *
    return pRoot
  end

  function IsEmpty() : Boolean
    return pRoot == NIL
  end
```

Dado un BST y un valor se quiere verificar si dicho valor existe o no en el árbol. El patrón básico de la función Lookup ocurre de forma recursiva (como es usual en los algoritmos de árboles):

1. Tratar con el caso base donde el árbol es vacío
2. Tratar con el nodo actual y emplear la recursión para tratar con sus subárboles

Dado que el árbol es un BST, se tiene un orden que se evalúa mediante operaciones relacionales y decide si trata del árbol derecho o el árbol izquierdo.

```
function Lookup (Node<T>* pNode, T tInfo) : Boolean
  if pNode == NIL then
    return false
  else
    select
      tInfo < *pNode.tInfo: return Lookup (*pNode.pLeft, tInfo)
      tInfo > *pNode.tInfo: return Lookup (*pNode.pRight, tInfo)
      tInfo == *pNode.tInfo: return true
    end
  end
end
```

El proceso de inserción consiste en dado el BST y una clave/información colocarla en el lugar correcto. Se retorna el apuntador de la posición donde fue insertado. Para ello se realizan los siguientes pasos:

- Si el BST es vacío, se crea el nodo raíz del árbol
- Si el BST tenía al menos la raíz, se verifica dónde se debe insertar el nodo nuevo recorriendo el árbol desde la raíz, y descendiendo por la rama izquierda si el valor a insertar es menor o por la derecha si es mayor que el nodo raíz actual en cada invocación recursiva.
- Si la clave/información a insertar ya se encuentra en el árbol se debe verificar de alguna forma (i.e. mensaje de error, retorna NIL, etc). En la implementación mostrada se asume que no habrá claves repetidas pero **no inserta** un nodo nuevo.
- Si la clave/información no existe en el árbol, se crea un nodo nuevo como el paso 1 y se inserta siempre como una hoja hija derecha o izquierda del nodo hallado en el paso 2.

La Fig. 28 muestra de forma gráfica el proceso de insertar los elementos 11, 6, 8, 19, 4, 10, 5, 17 (en ese orden) en un BST.

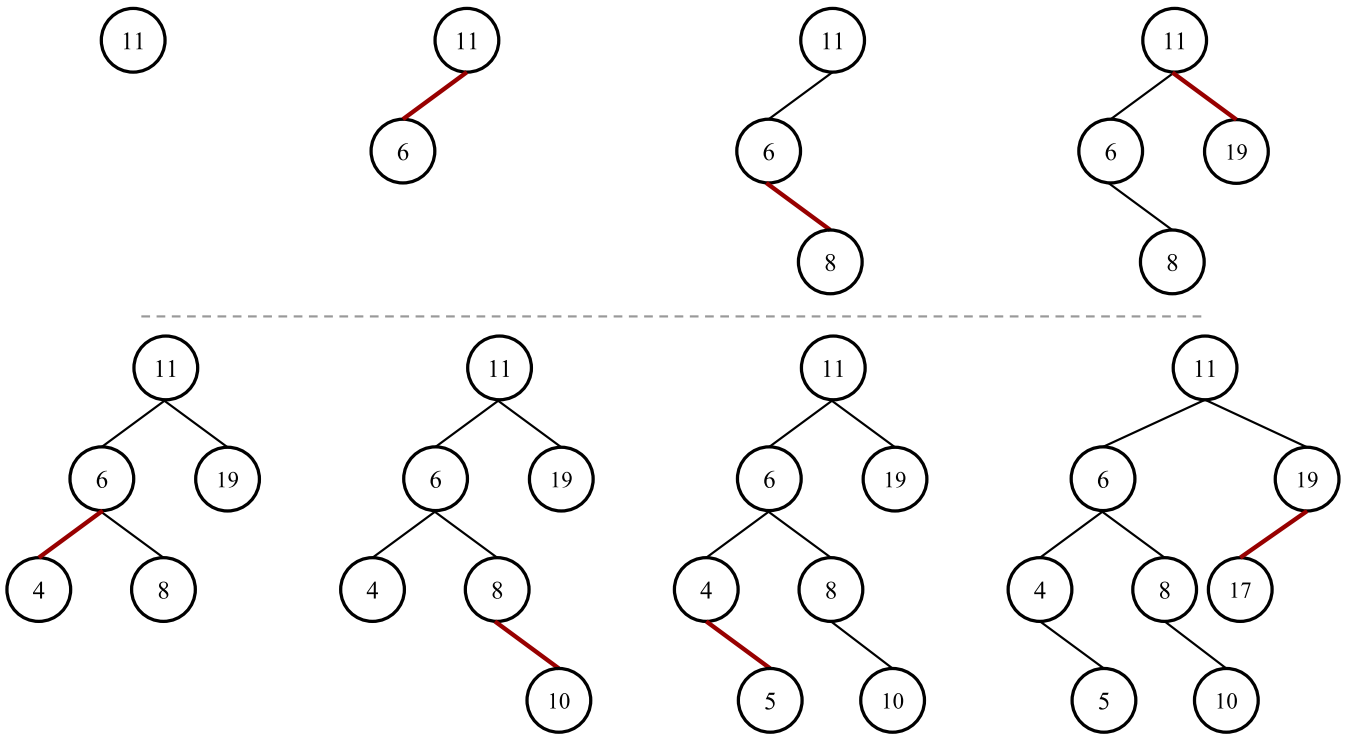


**Figura 9:** *Proceso de inserción de los elementos 11, 6, 8, 19, 4, 10, 5, 17 en un BST.*

Es importante destacar que la forma del árbol depende del orden en el cual los nodos son insertados.

```
function Insert (Node<T>* pNode, T tInfo) : Node<T> *
  if pNode == NIL then
    Node<T>* pNew = new Node
    *pNew.tInfo = tInfo
    *pNew.pLeft = *pNew.pRight = NIL
    return pNew
  else
    select
```

```
      tInfo < *pNode.tInfo:
        *pNode.pLeft = Insert(pNode->pLeft, tInfo)
      tInfo > *pNode.tInfo:
        *pNode.pRight = Insert(pNode->pRight, tInfo)
      tInfo == *pNode.tInfo:
        return pNode  //se asume que no hay claves repetidas en el BST
    end
    return pNode
  end
end
```

Para la operación Delete en un BST se deben considerar 3 casos posibles en cuanto a la ubicación de un nodo:

- El nodo a eliminar es hoja (no tiene hijos): En este caso se elimina el nodo sin mayores problemas.
- El nodo a eliminar tiene solo un hijo o subárbol (izquierdo o derecho): Se reenlaza al padre del nodo a eliminar con el hijo existente.
- El nodo a eliminar tiene dos hijos: En este caso se debe buscar:
  1. El nodo de mayor clave en su subárbol izquierdo. En la Fig. 29 corresponde al segundo árbol, el nodo más a la derecha del subárbol.
  2. El nodo de menor clave en su subárbol derecho. En la Fig. 29 corresponde al tercer árbol, el nodo más a la izquierda del subárbol. La clave de este nodo se le asigna al nodo parámetro de la función y posteriormente se elimina el nodo.

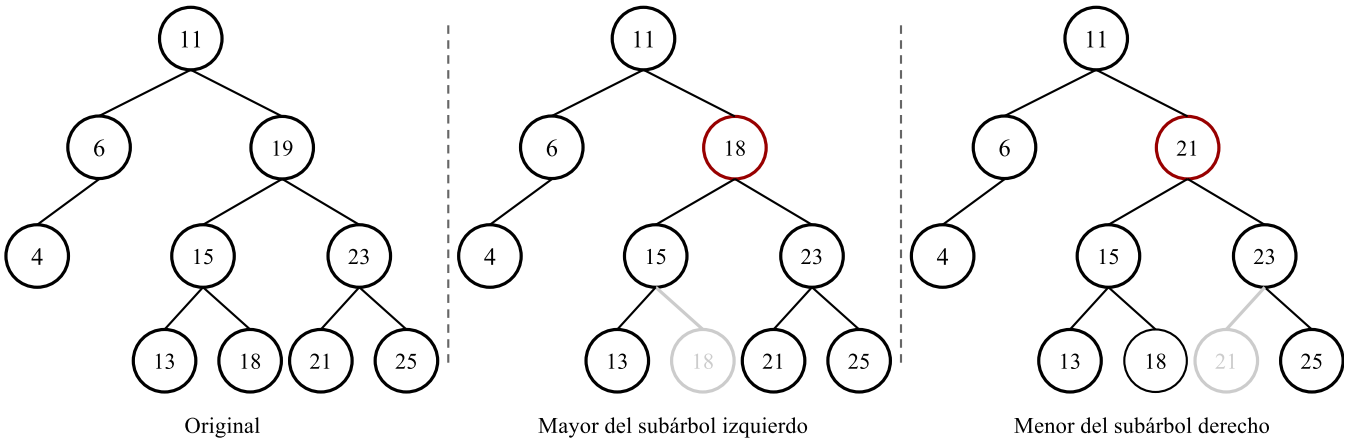


**Figura 10:** *Ejemplo del proceso de eliminación de un nodo en un BST. De izquierda a derecha: árbol original, se elimina el nodo con clave 19 y se busca el mayor de los menores, se elimina el nodo con clave 19 y se busca el menor de los mayores*

Como se explicó anteriormente, cuando el nodo a eliminar tiene dos hijos se elimina el menor de mayores (en cuando al valor de su clave) desde dicho nodo, o el mayor de los menores de éste. Para ello se puede emplear la función $DelMax$ o $DelMin$ mostradas a continuación.

```
void DelMax(ref Node<T>* pNode, ref T tInfo)
  if *pNode.pRight != NIL then
    DelMax(ref *pNode.pRight, ref tInfo)
  else
    Node<T>* pTemp = pNode
    tInfo = *pNode.tInfo
    pNode = *pNode.pLeft
    delete pTemp
  end
end
```

```
void DelMin(ref Node<T>* pNode, ref T tInfo)
  if *pNode.pLeft != NIL then
    DelMin(ref *pNode.pLeft, ref tInfo)
  else
    Node<T>* pTemp = pNode
    tInfo = *pNode.tInfo
    pNode = *pNode.pRight
    delete pTemp
  end
```

```
  end
```

La operación *Delete* consiste entonces en determinar en cuál caso se encuentra para el nodo a eliminar y **siempre** asegurar que el apuntador al nodo que se va a intercambiar desde su padre no quede con algún valor. Un ejemplo se observa en la Fig. 29 donde en el caso del mayor del subárbol izquierdo, el apuntador al subárbol derecho del nodo con clave 15 **debe** quedar apuntando a un valor de NIL. Igualmente sucede en el caso del menor del subárbol derecho, donde el apuntador del subárbol izquierdo del nodo con clave 23 debe estar en NIL.

Una implementación de dicha operación se muestra a continuación:

```
  void Delete(Node<T>* pNode, T tInfo)
    if pNode != NIL then
      select
        tInfo < *pNode.tInfo:
          Delete(*pNode.pLeft, tInfo)
        tInfo > *pNode.tInfo:
          Delete(*pNode.pRight, tInfo)
        tInfo == *pNode.tInfo:
          Node<T>* pTemp = pNode
          select
            *pNode.pLeft == NIL and *pNode.pRight == NIL: //no tiene hijos
              pNode = NIL
              delete pTemp
            *pNode.pLeft == NIL and *pNode.pRight != NIL: //tiene hijo derecho
              pNode = *pNode.pRight
              delete pTemp
            *pNode.pLeft != NIL and *pNode.pRight == NIL: //tiene hijo izquierdo
              pNode = *pNode.pLeft
              delete pTemp
            *pNode.pLeft != NIL and *pNode.pRight != NIL: //tiene ambos hijos
              DelMax(ref pNode, ref *pNode.tInfo)  //o DelMin
              *pNode.tInfo = tInfo
          end
      end
    end
  end
end
```

Una vez definida las operaciones de un BST es posible entonces aplicar diversos algoritmos sobre su estructura. Por ejemplo, dado un árbol binario de búsqueda no vacío, se quiere retornar el mínimo valor dentro del árbol.

Es importante destacar que dada la estructura del BST no es necesario recorrer todo el árbol para encontrar el valor mínimo, o de hecho, un valor cualquiera dado su orden implícito. El valor mínimo corresponde entonces al nodo que se encuentra más a la izquierda del árbol:

```
function MinValue (Node<T>* pNode) : Integer
  Node<T>* pTemp = pNode
  while (*pTemp.pLeft != NIL) do
    pTemp = *pTemp.pLeft
  end
  return *pTemp.tInfo
end
```

De forma general, se ha estudiado que los algoritmos de árboles resultan más convenientes si son escritos de forma recursiva por la naturaleza de la estructura de datos. Sin embargo, en este caso, un ciclo permite realizar el proceso de forma simple. También se puede emplear recursión tal como se hizo para la operación Delete.

### 4.3 Problema de Desequilibrio

El orden de inserción en un BST determina la forma estructural del árbol. La Fig. 30 muestra un ejemplo de distintos tipos de inserción en un BST del tipo Char, siendo insertados de izquierda a derecha.

Dado este aspecto, si los nodos de un BST son insertados en orden creciente (3er caso de la Fig. fig:BSTDesequilibrado) el árbol crecerá solo hacia el lado derecho como una lista simplemente enlazada, y todos los apuntadores izquierdos serán NIL. Del mismo modo, si son insertados en orden decreciente ('U','T','P','O','M','C','A'). La forma de una lista enlazada "degrada" la complejidad logarítimica del árbol.

Por el motivo anterior, es ideal construir un mecanismo que no permita construir árboles de búsqueda que sean degradados sino conseguir árboles tan equilibrados como sea posible. Para ello se muestra brevemente los árboles AVL y Red-Black.

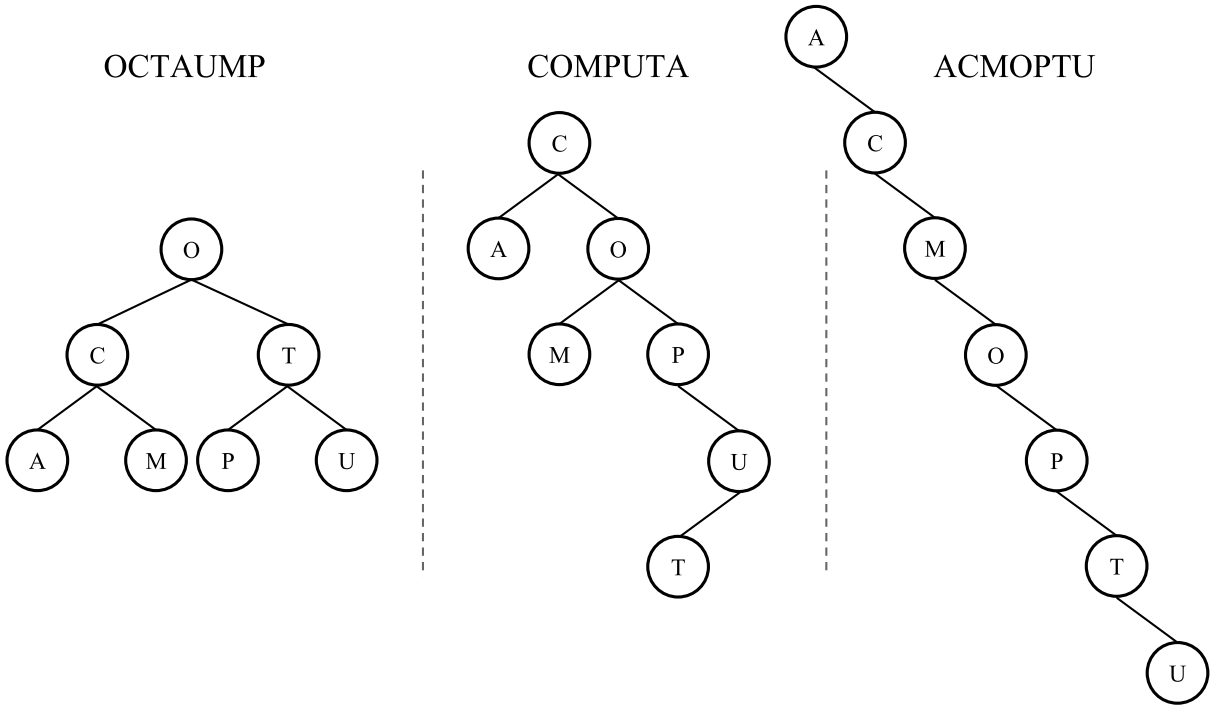


**Figura 11:** *Ejemplo de 3 tipos de BST generados con el mismo conjunto de datos en diferente orden.*

## 5 AVL

Un árbol AVL, llamado así por el nombre de sus inventores Georgy **A**delson-**V**elsky y Evgenii **L**andis, es un BST que se "auto" equilibra, es decir, cuando se inserta o remueve un nodo se aplican operaciones que tratan de mantener el árbol equilibrado. En un árbol AVL, las alturas de los subárboles de cualquier nodo difiere a lo sumo en 1, y en caso de no cumplirse esto se debe re-equilibrar para mantener dicha propiedad. Entonces, en la condición de los árboles AVL se considera un factor de equilibrio (*balance*): $balanceFactor = height(left) - height(right)$

En la Fig. 31 se muestra un ejemplo de 3 árboles AVL. El valor dentro de cada nodo (en esta ilustración) representa el factor de equilibrio de cada nodo, donde los valores positivos indican que la altura del subárbol izquierdo es mayor a la altura del subárbol derecho, y nos negativos el caso contrario. El factor de equilibrio 0 significa que ambos subárboles tienen la misma altura.

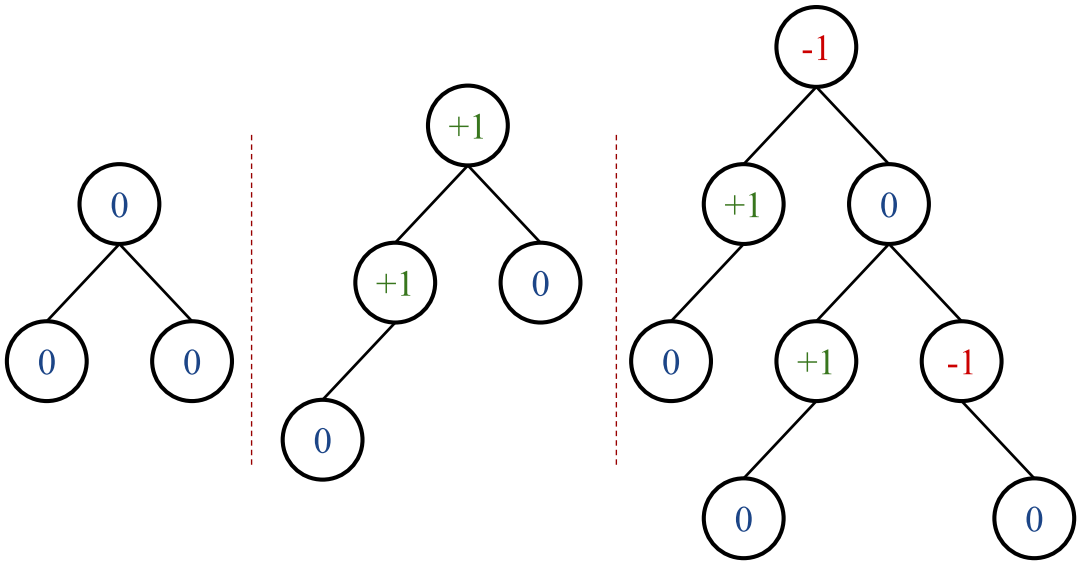


**Figura 12:** *Ejemplo de tres árboles equilibrados AVL mostrando su factor de equilibrio.*

Por su parte, la Fig. 32 muestra dos árboles binarios de búsqueda con su factor de equilibrio (AVL) donde se puede observar que los valor $+2$ y $-2$ representan una diferencia en las alturas de los nodos que contienen dicho valor (desequilibrio).

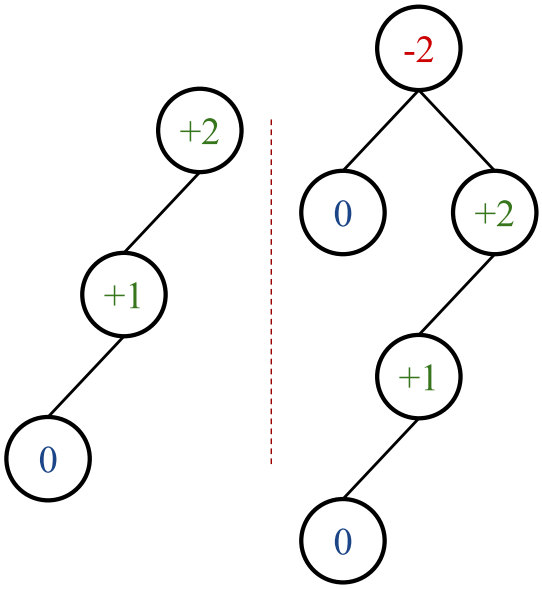


**Figura 13:** *Ejemplo de dos árboles desequilibrados AVL mostrando su factor de equilibrio.*

Las operaciones de un árbol AVL son las mismas que un BST, pero se añade un conjunto de operaciones llamadas rotaciones que sirven para mantener el equilibrio del árbol.

La operación de búsqueda para una clave en un AVL se hace de la misma forma que la búsqueda en un árbol de binario de búsqueda - BST. En un BST la búsqueda (y la inserción o eliminación) puede llegar a ser de $O(n)$ donde $n$ representa la altura del árbol. Este caso ocurre cuando el árbol está desequilibrado/degenerado como se mostró en la sección 4.3.

El mejor caso de un BST es cuando es de altura mínima, es decir, que en todos sus $h-1$ primeros niveles estás casi completos y solo varían en los nodos terminales, donde $h$ representa la altura del árbol. En dicho caso, el número de nodos viene dado por $n = 2^{h+1} - 1$ lo cual implica una complejidad de $O(log_2 n)$. La búsqueda en AVL representa la búsqueda en un árbol BST equilibrado, entonces en el peor de los casos la complejidad de la operación de búsqueda es $O(log_2 n)$.

Un recorrido en un AVL para todos los nodos del árbol hace que cada enlace se visite exactamente 2 veces: uno de entrada al nodo desde el subárbol al cual pertenece dicho nodo, y otro para dejar dicho nodo de ese subárbol que ha sido explorado. Entonces, si hay $n-1$ enlaces en el árbol, el corto amortizado es $2 \times (n-1)$

Para las operaciones de inserción y eliminación se utilizará la misma idea de un BST pero manteniendo siempre tras el equilibrio luego de cada inserción o eliminación. Este equilibrio se soluciona con las reajustes locales llamados rotaciones. Para ello se plantea solo el caso de las inserciones, donde para eliminar un nodo se opera de forma similar.

## 5.1 Inserción en un AVL

Luego de insertar un nodo, es necesario verificar la consistencia de la estructura del árbol tal que no existan desequilibrios. Para ello se cuenta con el factor de equilibrio (*balance*) que se calcula como:

$$balance = height(leftSubTree) - height(rightSubTree)$$

Entonces, en la implementación del nodo del AVL se requiere un campo adicional que contiene dicha diferencia. Los tres valores posibles para dicho campo son:

| *balance* | Relación de altura | Descripción |
|---|---|---|
| +1 | hijo Izq. > hijo Der. | La altura del subárbol izquierdo es mayor que la altura del subárbol derecho (uno más) |
| 0 | hijo Izq. = hijo Der. | La altura del subárbol izquierdo es igual que la altura del subárbol derecho |
| -1 | hijo Izq. < hijo Der. | La altura del subárbol izquierdo es menor que la altura del subárbol derecho (uno menos) |

El algoritmo de inserción en un AVL realiza los siguientes pasos:

1. Buscar en el árbol el lugar donde se insertará el nuevo nodo (nodo terminal). Para ello se recorre un camino desde la raíz hasta el lugar de inserción. La complejidad de este paso es $O(log_2 n)$ dado que la altura de un árbol AVL es $O(log_2 n)$, siendo $n$ el número de nodos.

2. Regresar por el camino recorrido en el paso 1 y se ajustan los factores de equilibrio. Si se insertó en el subárbol izquierdo se resta uno al valor del campo *balance* en cada nodo. Si se insertó por el subárbol derecho se suma uno al valor del campo *balance*. En cada paso de este retorno se pasa información acerca de la variación de la altura del árbol en caso que se incremente. Este paso consume un tiempo de ejecución $O(log_2 n)$.

3. Si existe algún desequilibrio (el campo *balance* de algún toma el valor $-2$ o $+2$) se debe reorganizar el subárbol que tiene como raíz dicho nodo llamado pivote (el que ocasiona el desequilibrio). La operación de re-equilibrar se efectúa mediante una secuencia de re-asignación de apuntadores que determinan una o dos rotaciones de dos o tres nodos, además de la actualización de los factores de equilibrio en cada nodo. Este paso es de $O(1)$. Los posibles casos una vez insertado un valor para el campo *balance* son:

| Campo balance (antes de Insertar) | Campo balance (Insertar Izquierda) | Campo balance (Insertar Derecha) |
|---|---|---|
| +1 | +2 | 0 |
| 0 | +1 | -1 |
| -1 | 0 | -2 |

Las celdas marcadas representan aquellos casos donde se debe re-equilibrar el subárbol partiendo del hecho que el nodo ocasiona el desequilibrio. Para ello se emplearán operaciones conocidas como rotaciones que se explican a continuación.

## 5.2 Rotaciones

Una operación de Rotación se emplea para restablecer el equilibrio de un AVL cuando se pierde debido a una operación de inserción o eliminación. Para conocer el tipo de Rotación a aplicar se debe identificar primero el caso donde se realiza un desequilibrio. Hay básicamente 4 casos posibles: Left-Right, Left-Left, Right-Left y Right-Right. Ya el nombre de cada caso identifica la ubicación de los nodos desequilibrados.

Cuando el factor de equilibrio de un nodo es +2 se puede observar en la Fig. 33 particularmente en el nodo A. Cuando el árbol tiene un desequilibrio positivo entonces está en el caso Left-Left o Left-Right. El caso Left-Left es cuando el árbol no se inclina hacia la derecha, y se puede hacer una rotación de todo el árbol hacia la derecha desde el nodo $C$, es decir, el elemento se insertó en el subárbol izquierdo del hijo izquierdo del ascendiente más cercano con factor +2.

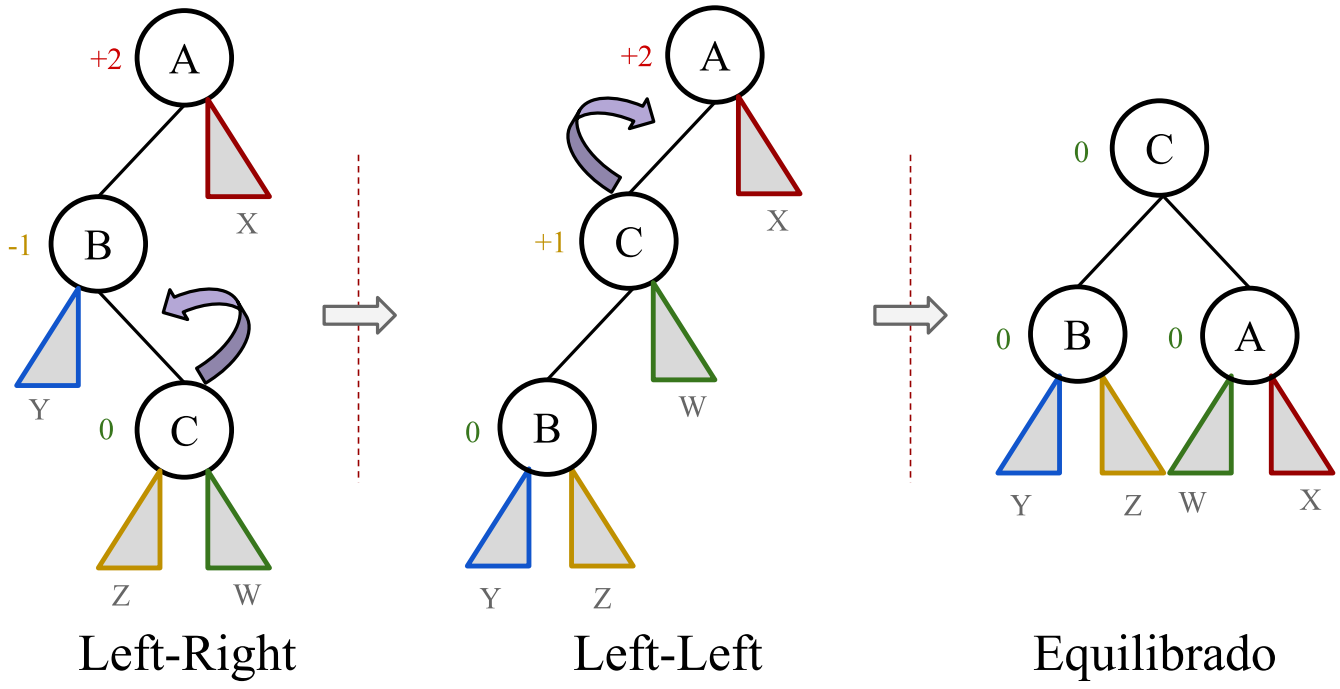


**Figura 14:** *De izquierda a derecha: casos de desequilibrio Left-Right, Left-Left, y el resultado del árbol ya equilibrado.*

En la Fig. 33, el caso Left-Left se refiere al árbol central, donde desde el nodo $C$ se aplica una rotación hacia la derecha para conseguir el árbol más a la derecha (equilibrado). Ahora, cuando el elemento es insertado como subárbol derecho del hijo izquierdo del ascendiente más cercano con factor +2, se está en presencia del caso Left-Right (árbol más a la izquierda de la figura). En dicho caso, primero se hace una rotación hacia la izquierda desde $C$ y luego se estará en el caso Left-Left donde se hace una rotación hacia la derecha de todo el árbol desde $C$.

Cuando el factor de equilibrio de un nodo es $-2$ se puede observar en la Fig. 34 particularmente en el nodo A. Cuando el árbol tiene un desequilibrio negativo entonces está en el caso Right-Right o Right-Left. El caso Right-Right es cuando el árbol no se inclina hacia la izquierda, y se puede hacer una rotación de todo el árbol hacia la izquierda desde el nodo $C$, es decir, el elemento se insertó en el subárbol derecho del hijo derecho del ascendiente más cercano con factor $-2$.

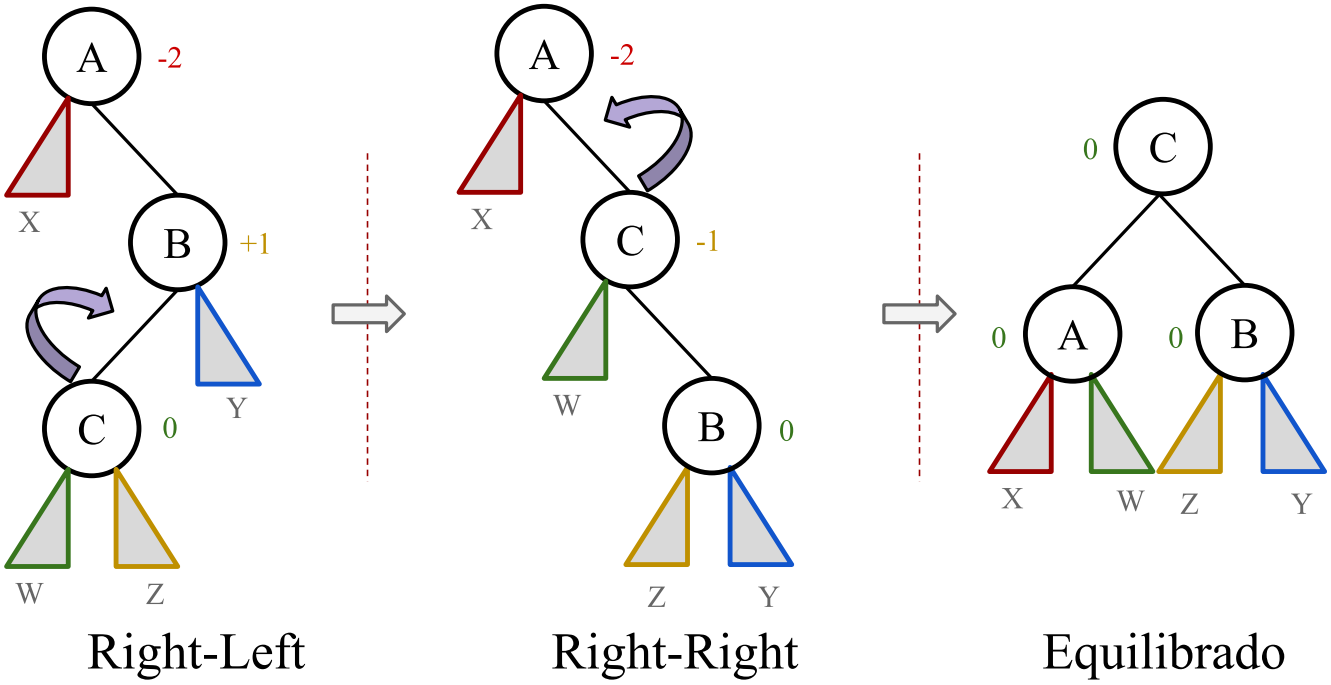


**Figura 15:** *De izquierda a derecha: casos de desequilibrio Right-Left, Right-Right, y el resultado del árbol ya equilibrado.*

En la Fig. 34, el caso Right-Right se refiere al árbol central, donde desde el nodo $C$ se aplica una rotación hacia la izquierda para conseguir el árbol más a la derecha (equilibrado). Ahora, cuando el elemento es insertado como subárbol izquierdo del hijo derecho del ascendiente más cercano con factor $-2$, se está en presencia del caso Right-Left (árbol más a la izquierda de la figura). En dicho caso, primero se hace una rotación hacia la derecha desde $C$ y luego se estará en el caso Right-Right donde se hace una rotación hacia la izquierda de todo el árbol desde $C$.

A nivel de implementación se suele manejar 3 apuntadores de la siguiente forma:

**Pointer A:** Apuntador del nodo con factor de equilibrio no permitido $\pm 2$.

**Pointer B:** Apuntador al hijo izquierdo o derecho de A, y se ubica de acuerdo a las siguientes reglas:

- Si A.balance $= +2$, entonces B es el hijo izquierdo de A.
- Si A.balance $= -2$, entonces B es el hijo derecho de A.

**Pointer C:** Apuntador al hijo del hijo ("nieto") de A. Este nodo es afectado si se está en el caso Left-Right o Right-Left, y se ubica de acuerdo a las siguientes reglas:

- Si B es el hijo izquierdo de A, entonces C es el hijo derecho
- Si B es el hijo derecho de A, entonces C es el hijo izquierdo

Para el caso de la Fig. 33 y 34, los apuntadores A, B y C corresponden con cada uno de los nodos asociados a esos valores.

Es importante destacar que cuando el caso de Left-Left o Right-Right, el número de nodos afectados es 2 (A y B). Ahora, cuando el caso es Left-Right o Right-Left el número de nodos afectados es 3 (A, B y C).

### 5.3 Ejemplo de Insertar

Para ver un poco los distintos casos para la inserción, veamos un ejemplo gráfico del proceso en insertar los nodos en un AVL con nodos que almacenan valores del tipo String.

El orden de inserción es: "SE", "PA", "AL", "BA", "OR", "LE", "CA", "LU", "GR".

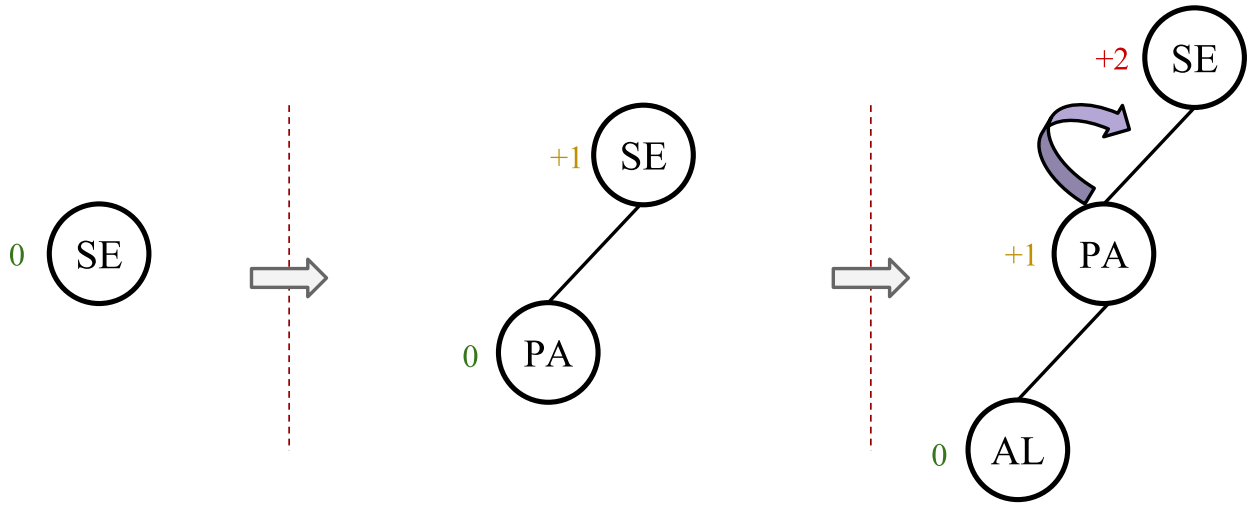


**Figura 16:** *De izq. a der.: inserción de "SE"; "PA"; y "AL". El nodo "AL" causa un desequilibrio +2 en el nodo raíz, teniendo así un caso Left-Left.*

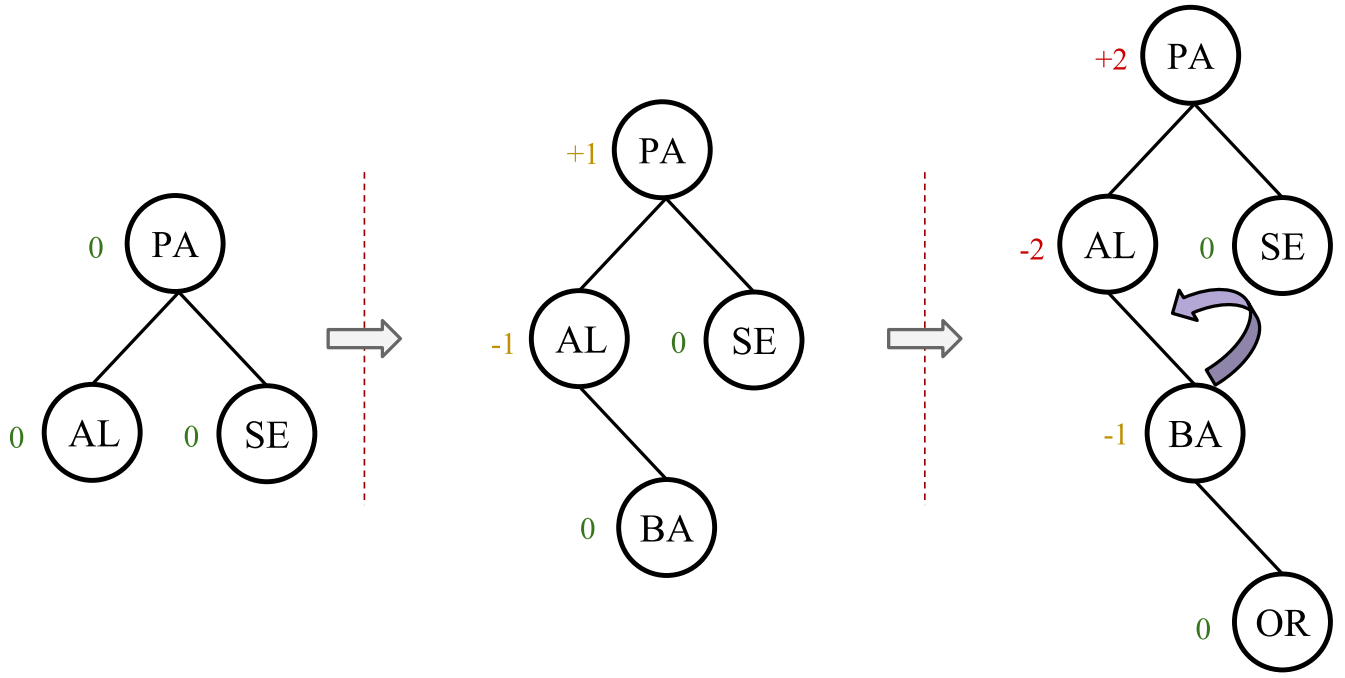


**Figura 17:** *De izq. a der.: luego de insertar "AL", se equilibra el árbol con una rotación hacia la derecha; se inserta "BA"; y "OR". El nodo "OR" causa un desequilibrio −2 en el nodo "AL", teniendo así un caso Right-Right.*

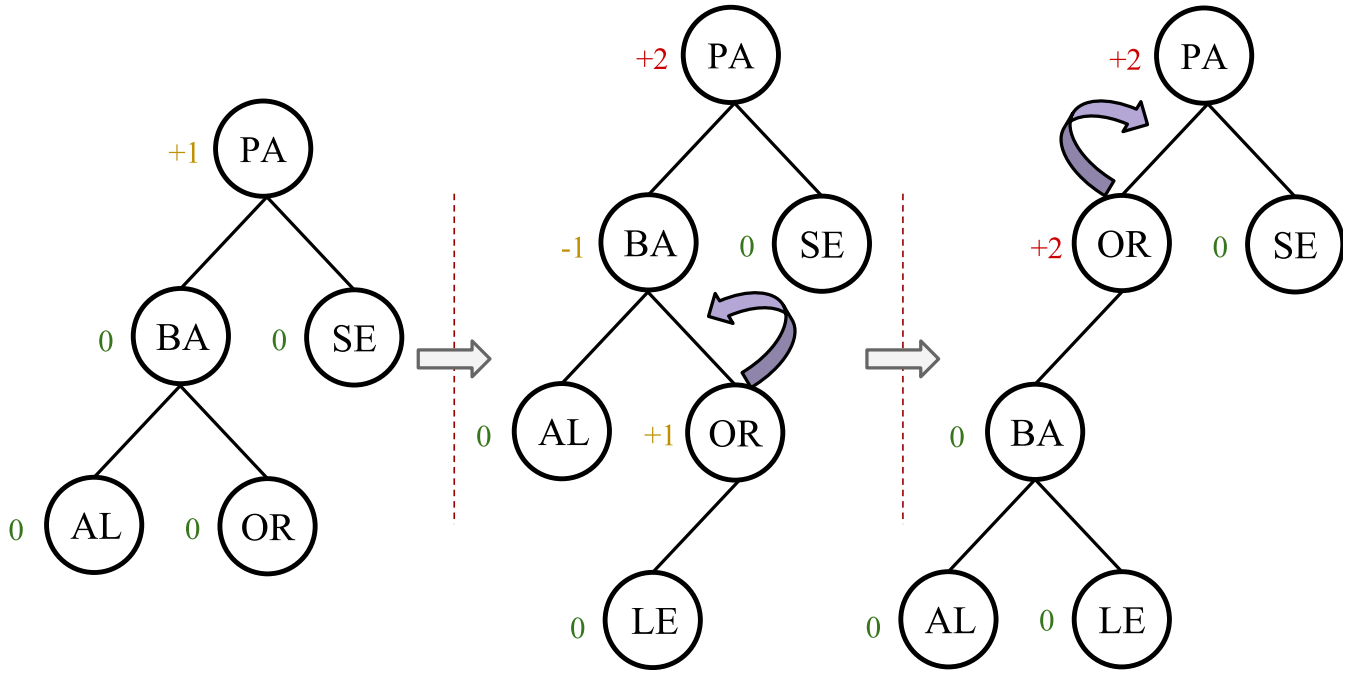


**Figura 18:** *De izq. a der.: luego de insertar "OR", se equilibra el árbol con una rotación hacia la izquierda; se inserta "LE"; y el nodo "LE" causa un desequilibrio +2 al nodo raíz. Se está frente a un caso Left-Right, teniendo que rotar hacia la izquierda para estar en un caso Left-Left.*

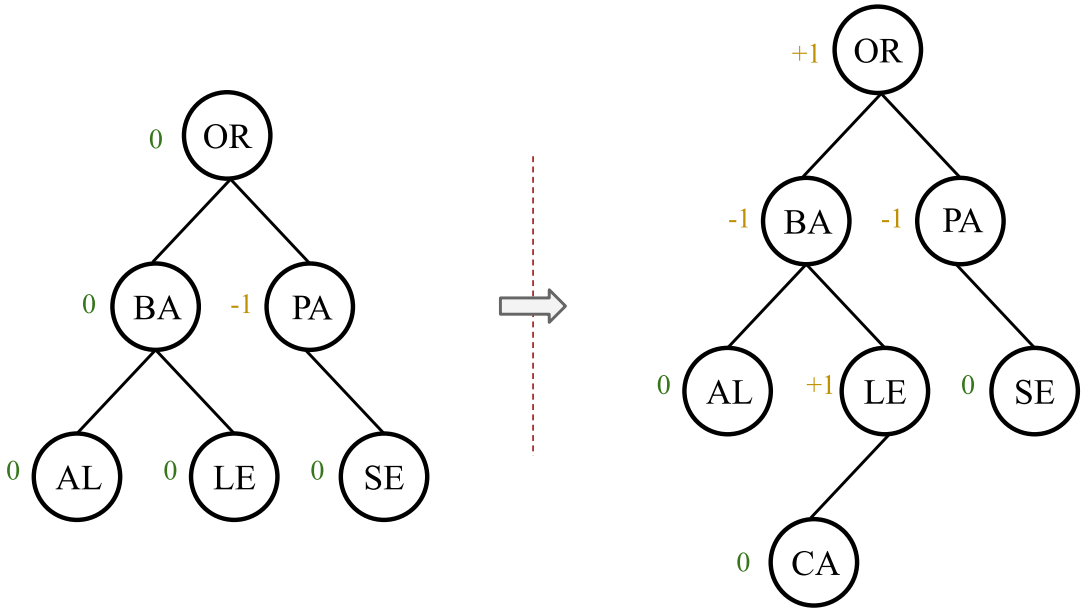


**Figura 19:** *De izq. a der.: estando en un caso Left-Left, se hace una rotación hacia la derecha y se equilibra el árbol; con una rotación hacia la izquierda; se inserta "LE"; y se inserta el nodo "CA".*

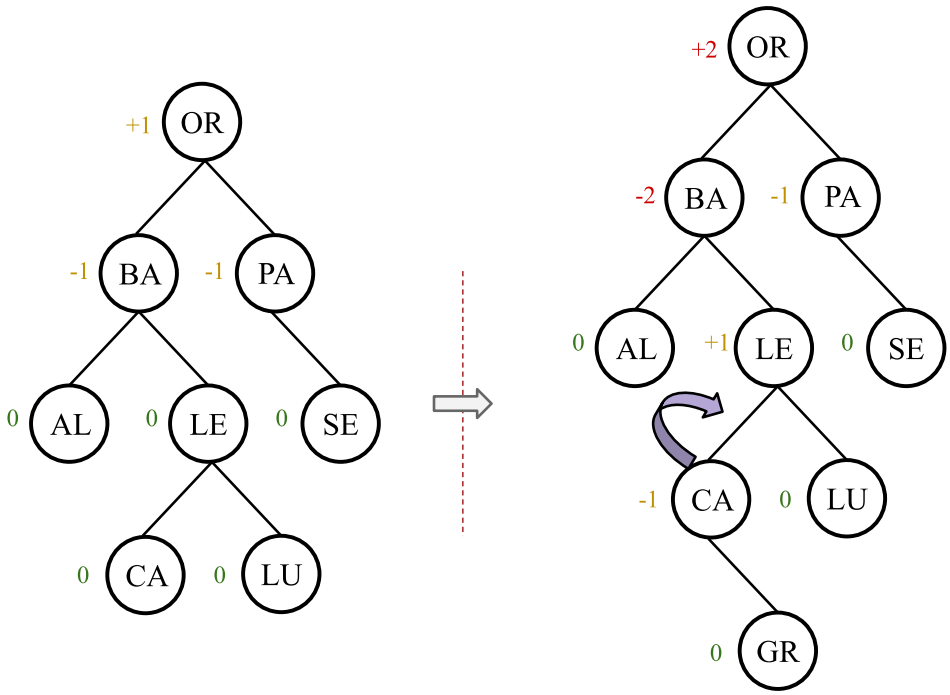


**Figura 20:** *De izq. a der.: se inserta el nodo "LU"; y el nodo "GR". El nodo "GR" causa un desequilibrio $-2$ en el nodo "BA", teniendo así un caso Right-Left.*

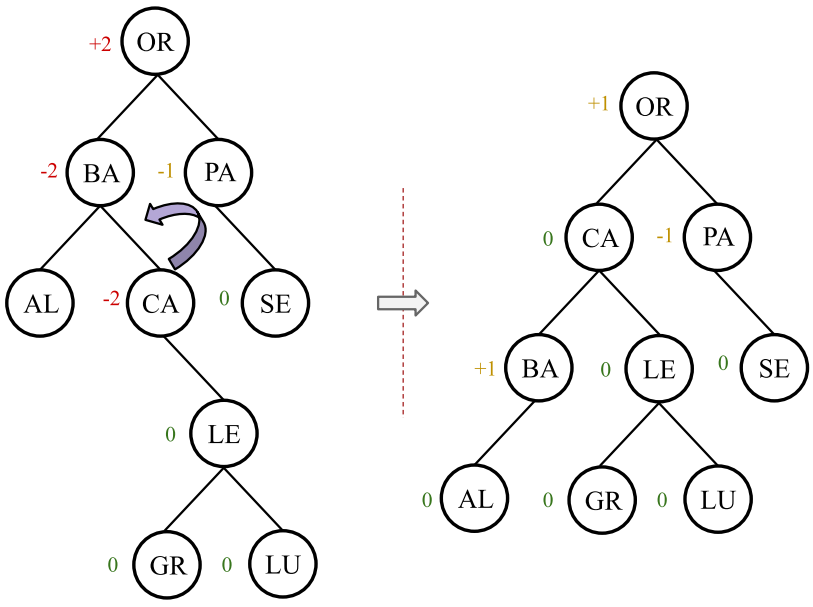


**Figura 21:** *De izq. a der.: insertado el nodo "GR" se hace una rotación a la derecha y se está en un caso Right-Right. Luego se aplica una rotación hacia la izquierda para equilibrar el árbol.*

# 6 Red-Black

Al igual que los árboles AVL, un árbol red-black (también llamado rojinegro) es un árbol binario de búsqueda que se auto-equilibra. Para ello, cada nodo tiene un atributo extra llamado color, el cual puede ser negro (*black*) o rojo (*red*). El equilibrio se da pintando cada uno de los nodos del árbol con uno de los dos colores de manera tal que cumpla ciertas propiedades.

Cuando el árbol se modifica, el nuevo árbol es acomodado y sus nodos son pintados nuevamente forma tal que se cumplas las propiedades de los árboles red-black. En la Fig. 41 se muestra un ejemplo de como se conforma un árbol red-black.

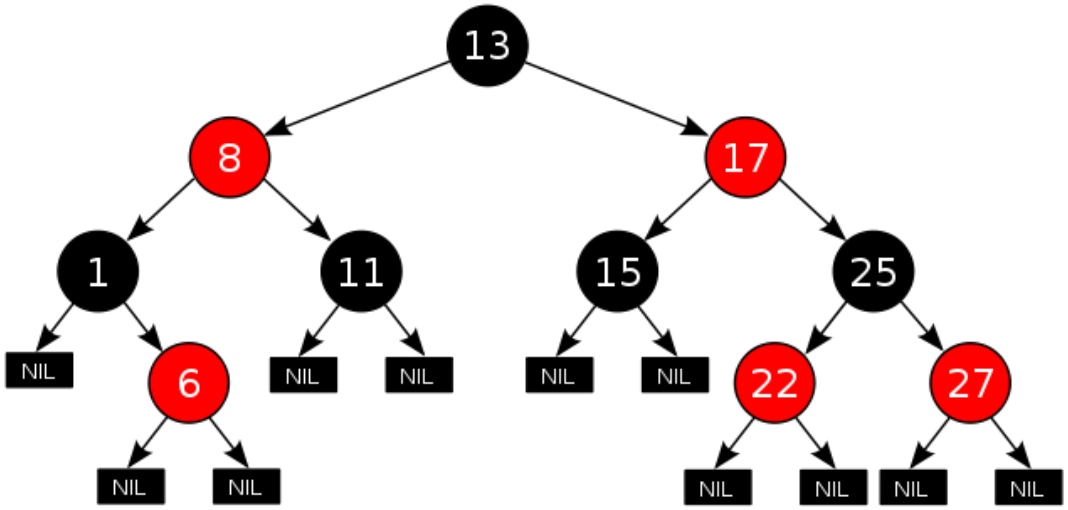


**Figura 22:** *Ejemplo de un árbol red-black.*

Las operaciones de búsqueda, inserción y eliminación es $O(logn)$. Por otro lado, el atributo de color de cada nodo puede ser almacenado en un solo bit.

## 6.1 Propiedades

Adicionalmente a las características de un BST, un árbol red-black debe cumplir las siguientes propiedades:

1. Un nodo es de color rojo o negro
2. La raíz siempre es color negro
3. Todo nodo terminal (hoja) es color negro, teniendo el valor de NIL. Todas las hojas tienen el mismo color que la raíz
4. Todo nodo color rojo debe tener dos hijos color negro
5. Todo camino desde un nodo cualquiera a cualquier de sus hojas descendientes contiene el mismo número de nodos negros. Al número de nodos negros de un camino se le denomina "altura negra".

Estas propiedades permiten asegurar que el camino más largo desde la raíz hasta una hoja no es más largo que dos veces el camino más corto desde la raíz a un nodo terminal. Con esto, el árbol trata de estar lo más equilibrado posible.

## 6.2 Operaciones

En el peor de los casos, las operaciones de insertar, eliminar y buscar tienen un tiempo de ejecución proporcional a la altura del árbol, $O(logn)$, siendo una mejor opción que un BST. Esto se debe por la propiedad #4 que hace que ningún camino pueda tener dos nodos rojos seguidos. Así, el camino más corto posible tiene todos sus nodos negros, y el más largo alterna entre nodos rojos y negros. Dado que todos los caminos máximos tienen el mismo número de nodos negros por la propiedad #5, no hay ningún camino que pueda tener longitud mayor que el doble de la longitud de otro camino.

Los árboles red-black son muy empleados en lenguajes con soporte a programación funcional, donde son una de las estructuras de datos persistentes más comúnmente utilizadas en la construcción de arreglos asociativos y conjuntos que pueden retener versiones previas tras mutaciones.

# 7 Heap

El Heap (o montículo) es un árbol binario complejo o esencialmente completo: todos los niveles están llenos, excepto posiblemente el último, en donde los nodos se encuentran en las posiciones más a la izquierda. Básicamente cuando se insertan

los nodos en un heap, los nodos internos se han subido en el árbol lo más posible, con los nodos internos del último nivel empujados hacia la izquierda.

Un heap satisface cierta propiedad de ordenamiento: el elemento almacenado en cada nodo es mayor o igual a los elementos almacenados en sus descendientes. En la Fig. 42 se puede observar un ejemplo de un Heap.

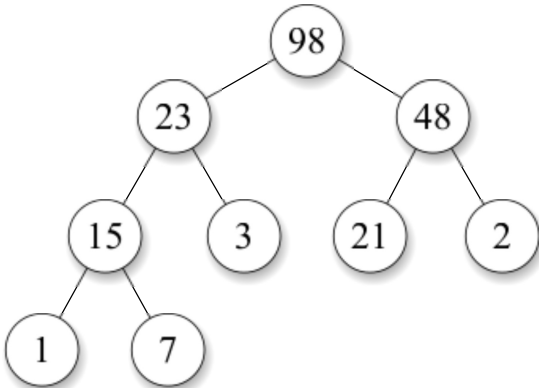


**Figura 23:** *Ejemplo de un heap representado un árbol binario esencialmente completo.*

## 7.1 Operaciones

Las operaciones básicas de un Heap de pueden resumir como:

- Construir un Heap vacío
- Comprobar si un Heap es vacío
- Obtener el mayor elemento
- Eliminar el mayor elemento
- Insertar un elemento

Una buena de representación del heap es empleando un arreglo, donde en cada posición se guarda un elemento basado en la numeración de los nodos del árbol por niveles. El esquema estructural del heap es como se muestra en la Fig. 25 de la sección 3.3, la única diferencia radica en que dicha imagen no cumple la propiedad de ordenamiento.

En dicho esquema, se almacena el i-ésimo elemento en la i-ésima posición del arreglo. El hijo izquierdo de $i$ en la posición $2 \times i$, el hijo derecho de $i$ en $2 \times i + 1$, y el padre de $i$ en $\frac{i}{2}$.

### 7.1.1 Implementación

Solo se emplea un arreglo y una variable que indica el número de elementos presentes:

```
class Heap<T>
private:
  const Integer MAX_SIZE = ...
  Array aHeap of T[1..MAX_SIZE]
  Integer iSize
public:
  Constructor Heap()
  Destructor Heap()
  function IsEmpty() : Boolean
  function GetRoot() : T
  void Delete()
  void Insert(T x)
end
```

La función constructora, la función *IsEmpty* y *GetRoot* resultan muy simples con está implementación:

```
Constructor Heap()
  iSize = 0
end

function IsEmpty() : Boolean
  return iSize == o
```

```
end

function GetRoot() : T
  return aHeap[1]
```

Para ilustrar la operación de eliminar, veamos el árbol de la Fig. 43.

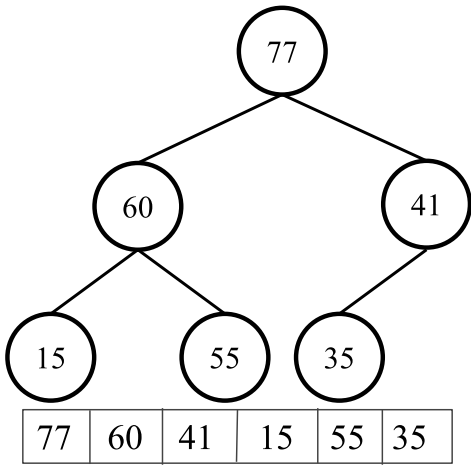


**Figura 24:** *Heap con seis valores almacenados en un arreglo.*

Eliminar un elemento consiste en eliminar el máximo elemento del Heap, es decir, el elemento del tope del Heap (la raíz del árbol), valor 77. El procedimiento consiste en extraer el nodo de la última posición del arreglo y colocarlo en la raíz. El último elemento del arreglo corresponde al nodo en el nivel más abajo a la derecha.

En el caso de la Fig. 43, dicho valor es 35. La Fig. 43 muestra este proceso y así se obtiene un nuevo Heap.

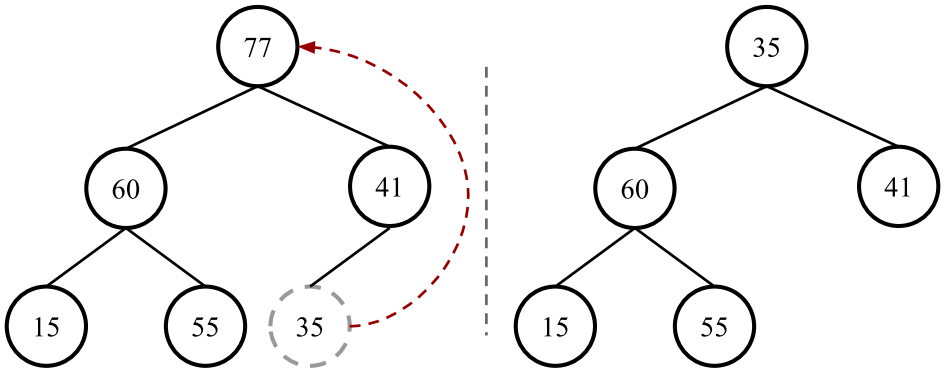


**Figura 25:** *Proceso de eliminar el valor de la raíz o el mayor elemento del árbol.*

Ahora, este nuevo árbol es un "semi-Heap" porque es esencialmente completo y ambos subárboles de la raíz son Heaps. Sin embargo, la raíz es el único elemento mal ubicado. Por ello se debe cambiar dicho elemento por el mayor de sus hijos. En el ejemplo, el cambio se debe realizar con el valor del subárbol izquierdo → 60. La Fig. 45 ilustra este proceso.

La nueva raíz, valor de 60, es ahora mayor que sus dos hijos, pero no se sabe si el subárbol izquierdo es un Heap. Si es un Heap, entonces se da por finalizada la operación de eliminación. En caso contrario, es necesario volver a repetir el mismo procedimiento. Dicho procedimiento se conoce como down-heap, sift-down o heapify-down (llamado también hundir). Escribiendo la función DownHeap que corresponde con el contenido de Delete se tiene:

```
void DownHeap(Integer iPos)
  Integer iC = 2 * iPos
  Boolean bFlag = false
  while iC <= iSize and not bFlag do
    if (iC < iSize and aHeap[iC] < aHeap[iC + 1] then
      iC = iC + 1
    end
    if aHeap[iPos] < aHeap[iC] then
      swap(ref aHeap[iPos], ref aHeap[iC])
```

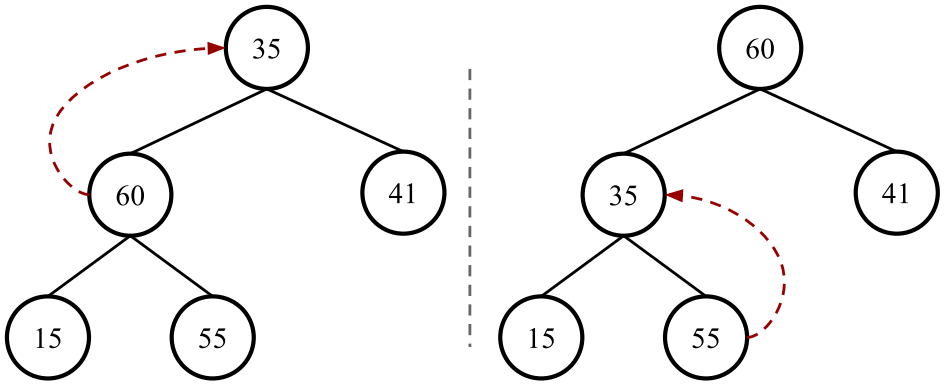


**Figura 26:** *Proceso de eliminar el valor de la raíz o el mayor elemento del árbol.*

```
      iPos = iC
      iC = 2 * iC
    else
      bFlag = true
    end
  end
  iSize = iSize - 1
end
```

Esta implementación es muy simple, existen otras versiones un poco más simplificadas. Lo importante es que la operación $DownHeap$ es de complejidad $O(log_2 n)$.

Por su parte, la operación de Insertar aplica el $DownHeap$ en sentido inverso llamado up-heap, sift-up o heapify-up (llamado también flotar). La función de complejidad $O(log_2 n)$ se puede escribir como:

```
void UpHeap(Integer x)
  iSize = iSize + 1
  aHeap[iSize] = x
  Integer iPos = iSize
  Integer iParent = iPos div 2
  while (iParent >= 1 and aHeap[iPos] > aHeap[iParent]) do
    swap(ref aHeap[iPos], ref aHeap[iParent])
  iPos = iParent
    iParent = iPos div 2
  end
end
```

Además, estas operaciones permiten construir un algoritmo de ordenamiento basado en comparaciones llamado Heapsort. El Heapsort se puede ver como una mejora a Selection Sort donde la entrada se divide en una región ordenada y otra desordenada, y se itera sobre la región no ordenada extrayendo siempre el máximo elemento y moviéndolo hacia la región ordenada. Esta operación en un tiempo de $O(nlogn)$.

## 8 Ideas Finales

- Un árbol es una estructura de datos que puede ser definida de forma recursiva como una colección de nodos (desde la raíz) empleando una relación de jerarquía
- Dependiendo del número de relaciones de jerarquía que tenga cada nodo del árbol, se dice que un árbol k-ario es aquel que tiene un máximo de $k$ enlaces/relaciones por nodo
- Cuando $k = 2$ se trata de un árbol binario, en donde cada nodo puede tener hasta dos hijos (derecho e izquierdo).
- Un BST o árbol binario ordenado o de búsqueda permite almacenar valores ordenados por una clave lo cual permite su búsqueda recorriendo desde la raíz hacia las hojas realizando comparaciones simples
- Tanto el árbol AVL como el red-black son árboles binarios de búsqueda auto-equilibrados con operaciones muy similares por las propiedades de equilibrio que imponen las cuales son distintas. Los árboles AVL están más rígidamente equilibrados que los del tipo red-black
- Además del árbol AVL y red-black existen otros tipos de árboles "auto-equilibrados" como los árboles 2-3, AA, Scapegoat, Splay, Treap y otros más.

# 9 Problemas

1. Usando las primitivas de árboles generales, implemente la operación PodarHojas, la cual consiste en eliminar todos aquellos nodos que son hojas en un árbol.
2. Se tiene un lenguaje que no permite la recursión, y para simularla emplea una pila en la cual almacena en el caso de los árboles, el camino recorrido desde la raíz hasta el nodo visitado. Se quiere que Ud. realice el algoritmo que permita recorrer un árbol binario (Sin utilizar recursión) en Simétrico.
3. Defina Árbol Binario de Búsqueda y Árbol Binario de Búsqueda Balanceado (AVL). Establezca similitudes y diferencias desde el punto de vista de Costo en Memoria y Complejidad en Tiempo de las operaciones de acceso y búsqueda.
4. Para un árbol AVL inicialmente vacío indique detalladamente el estado del árbol a medida que se insertan los valores: 10, 5, 9, 11 y 13. Sobre el árbol final indique su altura y qué se imprime usando un recorrido en Preorder.