Ejercicio 1

Diseño con Árboles Binarios

Ejercicio 1. Implementamos un **Árbol Binario** (AB) con

```
 Nodo<T>es Struct<dato: T, izq: Nodo, der: Nodo >

Modulo ArbolBinario<T> implementa Árbol Binario<T>{
      var raíz: Nodo<T> // "puntero" a la raíz del árbol

         ...
}
```

1. Escriba en castellano el invariante de representación para este módulo
2. Escriba en lógica el invariante de representación para este módulo usando predicados recursivos.
3. Escriba los algoritmos para los siguientes procs y, de ser posible, calcule su complejidad
   a) `altura(in ab: ArbolBinario<T>): int` // *devuelve la distancia entre la raíz y la hoja más lejana*
   b) `cantidadHojas(in ab: ArbolBinario<T>): bool`
   c) `está(in ab: ArbolBinario<T>, int t: T): bool` // *devuelve true si el elemento está en el árbol*
   d) `cantidadApariciones(in ab: ArbolBinario<T>, int t: T): int`

Figura 1: Enunciado Problema 1

Anexo: TAD ArbolBinario

```
TAD ArbolBinario<T> {
      obs vacio: bool
      obs dato: T
      obs izq: ArbolBinario<T>
      obs der: ArbolBinario<T>

      proc nuevoArbolVacío(): ArbolBinario<T>
            asegura {ret.vacio = true}

      proc nuevoArbol(in l: ArbolBinario<T>, in e: T, in r: ArbolBinario<T>): ArbolBinario<T>
            asegura {ret.vacio = false}
            asegura {ret.dato = e}
            asegura {ret.izq = l}
```



```
            asegura {ret.der = d}

      proc obtenerIzq(in ab: ArbolBinario<T>): ArbolBinario<T>
            requiere {ab.vacio = false}
            asegura {ret = ab.izq}

      proc obtenerDer(in ab: ArbolBinario<T>): ArbolBinario<T>
            requiere {ab.vacio = false}
            asegura {ret = a.der}

      proc estaVacio(in ab: ArbolBinario<T>): bool
            asegura {ret = ab.vacio}

      proc obtenerDato(in ab: ArbolBinario<T>): T
            requiere {ab.vacio = false}
            asegura {ret = a.dato}

      pred estaPred(ab: ArbolBinario<T>, e: T)
         {ab.vacio = false ∧_L (ab.dato = e ∨ estaPred(ab.izq, e) ∨ estaPred(ab.der, e))}

      pred esHoja(in ab: ArbolBinario<T>)
         {ab.raiz.izquierda = null ∧ ab.raiz.derecha = null}
}
```

Figura 2: TAD Arbol Binario¡T¿

```
T extends Comparable<T>
Nodo<T> = Struct⟨valor: T, izq: Nodo<T>, der: Nodo<T>⟩

Modulo ArbolBinarioImpl<T> implements ArbolBinario<T> {
    var raiz: Nodo<T>

    pred equivalentes (raiz: Nodo<T>, ab': ArbolBinario<T>) {
        (raiz = null ∧ ¬def(ab')) ∨
        (
            raiz ≠ null ∧ def(ab') ∧_L raiz.valor = ab'.dato ∧
            equivalentes(raiz.izq, ab'.izq) ∧ equivalentes(raiz.der, ab'.der)
        )
    }
    pred valoresDefinindos (raiz: Nodo<T>) {
        raiz = Null ∨_L (raiz.valor ≠ Null ∧ valoresDefinindos(raiz.izq) ∧ valoresDefinindos(raiz.der))
    }
aux nodos (raiz: Nodo<T>) : conj⟨Nodo<T>⟩ = IfThenElse(
    raiz = null,
    ∅,
    {raiz} ∪ nodos(raiz.izq) ∪ nodos(raiz.der),
);
aux cantidadPadres (raiz: Nodo<T>, nodo: Nodo<T>) : ℤ =
    IfThenElse(
        raiz = null,
        0,
        ifThenElse(
            raiz = nodo,
            1,
            cantidadPadres(raiz.izq, nodo) + cantidadPadres(raiz.der, nodo),
        )
    );
pred tieneUnicoPadre (raiz: Nodo<T>, nodo: Nodo<T>) {
    cantidadPadres(raiz, nodo) = 1
}
pred sinCiclos (nodo: Nodo<T>, visitados: conj⟨Nodo<T>⟩) {
    nodo = Null ∨_L (
        nodo ∉ visitados ∧_L
        sinCiclos(nodo.izq, visitados ∪ {nodo}) ∧
        sinCiclos(nodo.der, visitados ∪ {nodo})
    )
}
% Asumo que ab' no tiene ciclos %
pred abs (ab: ArbolBinarioImpl<T>, ab': ArbolBinario<T>) {
    (ab'.vacio == true ⇔ ab.raiz == Null) ∧ equivalentes(ab.raiz, ab')
}
pred invRep (ab: ArbolBinarioImpl<T>) {
    sinCiclos(ab.raiz) ∧_L valoresDefinindos(ab.raiz) ∧
    (∀nodo : Nodo<T>) (nodo ∈ nodos(raiz) ⟶ tieneUnicoPadre(raiz, nodo))
}

% Aux %
proc auxMax (in a: int, in b: int) : int
```

```
if a > b
  return a
endif

return b
```

```
proc auxEsHoja (in nodo: Nodo<T>) : int
```

```
return nodo.izq == null && nodo.der == null
```

```
% Fin Aux %
proc alturaSubArbol (in ab: ArbolBinarioImpl<T>, in nodo: Nodo<T>) : int
```

```
if nodo == null
  return 0
endif

return 1 + ab.auxMax(
  ab.alturaSubArbol(ab, nodo.izq),
  ab.alturaSubArbol(ab, nodo.der)
)
```

```
proc altura (in ab: ArbolBinarioImpl<T>) : int
```

```
return ab.alturaSubArbol(ab, ab.raiz)
```

```
proc cantidadHojasSubArbol (in ab: ArbolBinarioImpl<T>, in nodo: Nodo<T>) : int
```

```
if nodo == null then
  return 0
endif

if ab.auxEsHoja(nodo) then
  return 1
endif

return
  ab.cantidadHojasSubArbol(ab, nodo.izq) +
  ab.cantidadHojasSubArbol(ab, nodo.der)
```

```
proc cantidadHojas (in ab: ArbolBinarioImpl<T>) : int
```

```
return ab.cantidadHojasSubArbol(ab, ab.raiz)
```

proc estáEnSubArbol (in ab: ArbolBinarioImpl<T>, in nodo: Nodo<T>, in t: T) : bool

```
if nodo == null then
  return false
endif

return nodo.valor.compareTo(t) == 0
  || ab.estáEnSubArbol(ab, nodo.izq, t)
  || ab.estáEnSubArbol(ab, nodo.der, t)
```

proc está (in ab: ArbolBinarioImpl<T>, in t: T) : bool

```
return ab.estáEnSubArbol(ab, ab.raiz, t)
```

proc cantidadAparicionesEnSubArbol (in ab: ArbolBinarioImpl<T>, in nodo: Nodo<T>, in t: T) : int

```
if nodo == null then
  return 0
endif

var cantidadAparicionesEnSubArboles: int
    cantidadAparicionesEnSubArboles:=
      ab.cantidadAparicionesEnSubArbol(ab, nodo.izq, t) +
      ab.cantidadAparicionesEnSubArbol(ab, nodo.der, t)

if nodo.valor.compareTo(t) == 0 then
  return 1 + cantidadAparicionesEnSubArboles
endif

return cantidadAparicionesEnSubArboles
```

proc cantidadApariciones (in ab: ArbolBinarioImpl<T>, in t: T) : int

```
return ab.cantidadAparicionesEnSubArbol(ab, ab.raiz, t)
```

}

Ejercicio 2. Un **Árbol Binario de Búsqueda** (ABB) es un árbol binario que cumple que para cualquier nodo N, todos los elementos del árbol a la izquierda son menores o iguales al valor del nodo y todos los elementos del árbol a la derecha son mayores al valor del nodo, es decir

```
pred esABB (a: ArbolBinario⟨T⟩) {
    a = Nil ∨ (
        (∀e : T)(e ∈ elems(a.Izq) → e ≤ a.dato) ∧ (∀e : T)(e ∈ elems(a.Der) → e > a.dato) ∧
        esABB(d.Izq) ∧ esABB(d.Der)
    )
}
aux elems (a: ArbolBinario⟨T⟩) : conj⟨T⟩{
    IfThenElseFi(a = Nil, ∅, {a.dato} ∪ elems(a.Izq) ∪ elems(a.Der))
}
```

- Implemente los algoritmos para los siguientes procs y calcule su complejidad en mejor y peor caso
  1. `está(in ab: ABB<T>, int t: T): bool` // *devuelve true si el elemento está en el árbol*
  2. `cantidadApariciones(in ab: ABB<T>, int t: T): int`
  3. `insertar(inout ab: ABB<T>, int t: T)`
  4. `eliminar(inout ab: ABB<T>, int t: T)`
  5. `inOrder(in ab: ABB<T>) : Array<T>` // *devuelve todos los elementos del árbol en una secuencia ordenada*
- Asumiendo que el árbol está balanceado, recalcule, si es necesario, las complejidades en peor caso de los algoritmos del ítem anterior
- ¿Qué pasa en un ABB cuando se insertan valores repetidos? Proponga una modificación del módulo que resuelva este problema

Figura 3: Ejercicio 2

```
T extends Comparable<T>
Nodo<T> = Struct⟨valor: T, izq: Nodo<T>, der: Nodo<T>⟩

Modulo ABB<T> implements ArbolBinario<T> {
    var raiz: Nodo<T>

    % Aux %

    proc auxTieneDosHijos (in nodo: Nodo<T>) : int
        return nodo.izq != null && nodo.der != null

    proc auxMaximoSubArbol (in nodo: Nodo<T>) : T
        if nodo.der != null then
          return auxMaximoSubArbol(nodo.der)
        endif

        return nodo.valor

    proc auxColaAArray (in cola: ColaSobreLista<T>, in tamañoCola: int) : Array<T>
        res:= new Array<T>(tamañoCola)

        var i: int
            i:= 0

        while (!cola.colaVacía()) do
          res[i]:= cola.desencolar()
          i:= i + 1
        endwhile

        return res

    % Fin Aux %

    proc estáEnSubArbol (in abb: ABB<T>, in nodo: Nodo<T>, in t: T) : bool
        if nodo == null then
          return false
        endif

        if nodo.valor.compareTo(t) < 0 then
          return abb.estáEnSubArbol(abb, nodo.izq, t)
        endif

        if nodo.valor.compareTo(t) > 0 then
          return abb.estáEnSubArbol(abb, nodo.der, t)
        endif

        return true

    proc está (in abb: ABB<T>, in t: T) : bool
        return abb.estáEnSubArbol(abb, abb.raiz, t)


    proc cantidadAparicionesEnSubArbol (in abb: ABB<T>, in nodo: Nodo<T>, in t: T) : int
        if nodo == null then
          return 0
        endif

        if nodo.valor.compareTo(t) <= 0 then
          if nodo.valor.compareTo(t) == 0 then
            return 1 + abb.cantidadAparicionesEnSubArbol(abb, nodo.izq, t)
          endif

          return abb.cantidadAparicionesEnSubArbol(abb, nodo.izq, t)
        endif

        return abb.cantidadAparicionesEnSubArbol(abb, nodo.der, t)

    proc cantidadApariciones (in abb: ABB<T>, in t: T) : int
        return abb.cantidadAparicionesEnSubArbol(abb, abb.raiz, t)


    proc insertarEnSubArbol (inout abb: ABB<T>, in nodo: Nodo<T>, in t: T) : Nodo<T>
        if nodo == null then
          var nodo: Nodo<T>
              nodo:= new Nodo()

          nodo.valor:= t

          return nodo
        endif

        if nodo.valor.compareTo(t) >= 0 then
          nodo.izq:= abb.insertarEnSubArbol(abb, nodo.izq, t)
        else
          nodo.der:= abb.insertarEnSubArbol(abb, nodo.der, t)
        endif

        return nodo

    proc insertar (inout abb: ABB<T>, in t: T)
        abb.raiz:= abb.insertarEnSubArbol(abb, abb.raiz, t)

        return


    proc eliminarDelSubArbol (inout abb: ABB<T>, in nodo: Nodo<T>, in t: T) : Nodo<T>
        if nodo.valor.compareTo(t) > 0 then
          nodo.izq:= eliminarDelSubArbol(abb, nodo.izq, t)
        else if nodo.valor.compareTo(t) < 0 then
          nodo.der:= eliminarDelSubArbol(abb, nodo.der, t)
        else
          // Acá se encontró un potencial nodo a eliminar
          if abb.auxTieneDosHijos(nodo) then
            nodo.valor:= abb.auxMaximoSubArbol(nodo.izq)
            nodo.izq:= eliminarDelSubArbol(abb, nodo.izq, nodo.valor)
          else
            // Acá puedo tener 3 casos: a) Hijo izq; b) Hijo der; c) Hijos null
            return nodo.izq != null ? nodo.izq : nodo.der
          endif
        endif

        return nodo

    proc eliminar (inout abb: ABB<T>, in t: T)
        abb.raiz:= eliminarDelSubArbol(abb, abb.raiz, t)

        return


    proc inorderSubArbol (inout cola: ColaSobreLista<T>, in abb: ABB<T>, in nodo: Nodo<T>) : int
        if nodo == null then
          return 0
        endif

        var tamañoIzq: int
        var tamañoDer: int

        tamañoIzq:= inorderSubArbol(cola, abb, nodo.izq)
        cola.encolar(nodo.valor)
        tamañoDer:= inorderSubArbol(cola, abb, nodo.der)

        return 1 + tamañoIzq + tamañoDer
```

proc inorder (in abb: ABB<T>) : $Array<T>$

```
var cola: ColaSobreLista<T>
    cola:= new colaVacía()

var tamañoCola: int
    tamañoCola:= abb.inorderSubArbol(cola, abb, abb.raiz)

return abb.auxColaAArray(cola, tamañoCola)
```

}

Ejercicio 3. Implementar los siguientes TADs sobre ABB. Calcule las complejidades de los procs en mejor y peor caso

1. Conjunto$\langle T \rangle$
2. Diccionario$\langle K, V \rangle$
3. ColaDePrioridad$\langle T \rangle$

Recalcule, si es necesario, las complejidades en peor caso de los algoritmos de los TADs considerando que se implementan sobre AVL en vez de ABB.

Figura 4: Ejercicio 3

Éste ej. directamente lo voy a hacer usando ConjuntoLog porque sino no se termina más.
Implementarlo sobre ABB no balanceado tiene complejidades temporales lineales y habría que implementar todos los procs, que es básicamente lo que hicimos en el ej. anterior sólo que sin repetidos.

```
Modulo ConjuntoSobreAVL<T> implements Conjunto<T> {
    var datos: ConjuntoLog<T>

    proc conjuntoVacío () : ConjuntoSobreAVL<T>
        1 res.datos:= new conjVacío<T>()
        2
        3 return res

    proc tamaño (in c: ConjuntoSobreAVL<T>) : int
        1 // Llamo al proc de Conjunto<T>... pues ConjuntoLog<T> implementa Conjunto<T>
        2 return res.datos.tamaño(res.datos)

    % Y así con todos los procs... %
}
```

En este ejercicio, si quisiéramos hacerlo con ABB, tendríamos que crear la siguiente estructura:
`ParClaveValor<K, V>` = `Struct`$\langle$`clave:` K`, valor:` $V\rangle$
Ese será el valor de cada nodo en nuestro árbol, y no admitiremos repetidos por clave.
El tipo "K"debe ser comparable.

```
Modulo DiccionarioSobreAVL<K, V> implements Diccionario<K, V> {
    var datos: DiccionarioLog<K, V>

    proc diccionarioVacío () : DiccionarioSobreAVL<K, V>
        1 res.datos:= new diccionarioVacío<K, V>()
        2
        3 return res

    proc está (in d: DiccionarioSobreAVL<K, V>, in clave: K) : bool
        1 // Llamo al proc de Diccionario<K, V>
        2 // ... pues DiccionarioLog<K, V> implementa Diccionario<K, V>
        3 return d.datos.está(clave)

    % Y así con todos los procs... %
}
```

Ejercicio 4

Muy divertido.. vamos a implementar un AVL, al menos parcialmente, ya que para mantener complejidades en tiempos logarítmicas vamos a tener que garantizar que el árbol esté balanceado.

En el proc agregar vamos a tener que balancear el árbol luego de cada inserción, para que el invRep siga valiendo. Una cosa poco intuitiva a la hora de implementar un AVL, es que éste, a diferencia del ABB, no puede tener repetidos. Los repetidos son todo un tema, pero supongamos que tenemos un AVL vacío e insertamos 3 veces el mismo elemento: insertar(2), insertar(2), insertar(2). El árbol ya no podrá balancearse.

Ésa es la verdadera razón por la cual el ejercicio nos pide que implementemos el TAD Conjunto, pero intuitivamente un AVL siempre es un conjunto.

Vamos a condensar un poco lo que nos piden en el ejercicio. Para calcular la cantidad de elementos en rango debemos realizar la siguiente cuenta $\#\{elems \mid desde \leq elem < hasta\} = \#\{elems \mid elem \leq hasta\} - \#\{elems \mid elem < desde\}$.

Para lograr eso en un ABB vamos a tener que ubicar el primer nodo que sea mayor igual a hasta, y contar la cantidad de nodos de ese subarbol.

Complejidad de esto: $O(n)$ para encontrar el nodo, y $O(m)$ para contar la cantidad de nodos; m es la cantidad de nodos del subarbol, y n es la altura del árbol.

Dicho esto, la complejidad es $O(n+m)$ pero simplemente la podemos acotar como $O(n)$, dónde n es la cantidad total de nodos del arbol.

Rápidamente vemos que ésto está muy lejos de la complejidad que nos solicita el ejercicio así que para arreglarlo vamos a implementar un AVL pero seguimos con el mismo problema, la complejidad para calcular los menores iguales de hasta, será $O(log(n)+n)$, no nos sirve.

Debemos alterar la estructura de nuestro árbol para transformar esa variable lineal en algo constante. Y solo podemos lograr eso si en cada nodo guardamos cuántos nodos tiene ese subarbol. Esto lo podemos hacer de manera eficiente.

Cada vez que insertemos un nodo vamos a actualizar la cantidad de nodos para cada nodo desde el nodo donde insertamos hasta la raíz.

Notemos que no tenemos que implementar eliminar, y la complejidad para insertar un elemento en nuestro AVL seguirá siendo $O(log(n))$

$\texttt{Nodo} = \texttt{Struct}\langle \texttt{izq: } Nodo\texttt{, der: } Nodo\texttt{, valor: } int\texttt{, altura: } int\texttt{, cardinal: } int\rangle$

```
Modulo ArbolBinarioBalanceado implements Conjunto<ℤ> {
    var raiz: Nodo
    var cardinal: int

    aux cantidadPadres (raiz: Nodo<T>, nodo: Nodo<T>) : ℤ =
        IfThenElse(
            raiz = null,
            0,
            ifThenElse(
                raiz = nodo,
                1,
                cantidadPadres(raiz.izq, nodo) + cantidadPadres(raiz.der, nodo),
            )
        );
    aux nodos (nodo: Nodo) : conj⟨Nodo⟩ = IfThenElse(
        nodo = null,
        ∅,
        {nodo} ∪ nodos(nodo.izq) ∪ nodos(nodo.der),
    );
    aux cardinal (nodo: Nodo) : ℤ =
        IfThenElse(
            raiz = null,
            0,
            1 + cardinal(nodo.izq) + cardinal(nodo.der)
        );
    aux altura (nodo: Nodo) : ℤ = IfThenElse(nodo = null, 0, nodo.altura);
    pred sinCiclos (nodo: Nodo, visitados: conj⟨Nodo⟩) {
        nodo = Null ∨_L (
            nodo ∉ visitados ∧_L
            sinCiclos(nodo.izq, visitados ∪ {nodo}) ∧
            sinCiclos(nodo.der, visitados ∪ {nodo})
        )
    }
    pred sinRepetidos (raiz: Nodo) {
        |elementos(raiz)| = cardinal(raiz)
    }
    pred sePuedeLlegarPor (raiz: Nodo, nodo: Nodo) {
        raiz = nodo ∨_L (
            raiz ≠ null ∧ nodo ≠ null ∧
            ((sePuedeLlegarPor(raiz.izq, nodo) ∧ ¬sePuedeLlegarPor(raiz.der, nodo)) ∨
            (sePuedeLlegarPor(raiz.der, nodo) ∧ ¬sePuedeLeggarPor(raiz.izq, nodo)))
        )
    }
    pred tieneUnicoPadre (raiz: Nodo<T>, nodo: Nodo<T>) {
        cantidadPadres(raiz, nodo) = 1
    }
    pred alturasOK (nodo: Nodo) {
        nodo = null ∨_L
        (nodo.altura = 1 + max(altura(nodo.izq), altura(nodo.der)) ∧ alturasOK(nodo.izq) ∧ alturasOK(nodo.der))
    }
    pred cardinalesOK (nodo: Nodo) {
        nodo = null ∨_L (nodo.cardinal = cardinal(nodo) ∧ cardinalesOK(nodo.izq) ∧ cardinalesOK(nodo.der))
    }
    pred estáBalanceado (raiz: Nodo) {
        (∀nodo : Nodo<T>) (nodo ∈ nodos(raiz) ⟶ −1 ≤ (nodo.izq.altura − nodo.der.altura) ≤ 1)
    }
    pred invRep (abb: ArbolBinarioBalanceado) {
        abb.cardinal = cardinal(abb.raiz) ∧ sinCiclos(abb.raiz) ∧_L sinRepetidos(abb.raiz) ∧
        (∀nodo : Nodo<T>) (nodo ∈ nodos(raiz) ⟶ tieneUnicoPadre(raiz, nodo)) ∧_L
        alturasOK(abb.raiz) ∧ cardinalesOK(abb.raiz) ∧ estáBalanceado(abb.raiz)
    }
    aux elementos (nodo: Nodo) : conj⟨ℤ⟩ = IfThenElse(
        nodo = null,
        ∅,
        {nodo.valor} ∪ elementos(nodo.izq) ∪ elementos(nodo.der),
    );
    pred abs (abb: ArbolBinarioBalanceado, c': Conjunto<ℤ>) {
        (|c'.elems| = 0 ↔ abb.raiz = null) ∧_L c'.elems = elemenos(abb.raiz)
    }
```

proc cardinalNodo (in nodo: Nodo) : int

```
return nodo == null ? 0 : nodo.cardinal
```

proc alturaNodo (in nodo: Nodo) : int

```
return nodo == null ? 0 : nodo.altura
```

proc recalcularAltura (in nodo: Nodo) : int

```
return 1 + Math.max(alturaNodo(nodo.izq), alturaNodo(nodo.der))
```

proc factorDeBalanceo (in nodo: Nodo) : int

```
return alturaNodo(nodo.izq) - alturaNodo(nodo.der)
```

proc buscarPrimerNodoMayor (in abb: ArbolBinarioBalanceado, in nodo: Nodo, in valor: int, in estricto: bool) : Nodo

```
if nodo == null then
  return null
endif

var comparacion: bool
    comparacion:= estricto ? nodo.valor > valor : nodo.valor >= valor

if comparacion then
  var nodoIzquierdo: Nodo
      nodoIzquierdo:= abb.buscarPrimerNodoMayor(abb, nodo.izq, valor)

  return nodoIzquierdo != null ? nodoIzquierdo : nodo
else
  return abb.buscarPrimerNodoMayor(abb, nodo.der, valor)
endif

```

proc cardinalElementos (in abb: ArbolBinarioBalanceado, in valor: int, in estricto: bool) : int

```
var nodoPorIzquierda: Nodo
var nodoPorIzquierda: Nodo

nodoPorIzquierda:= abb.buscarPrimerNodoMayor(abb, abb.raiz.izq, valor, estricto)
nodoPorDerecha:= abb.buscarPrimerNodoMayor(abb, abb.raiz.der, valor, estricto)

var cardinal: int
    cardinal:= cardinalNodo(nodoPorIzquierda) + cardinalNodo(nodoPorDerecha)

return abb.raiz.cardinal - cardinal
```

proc cantidadElementosEnRango (in abb: ArbolBinarioBalanceado, in desde: int, in hasta: int) : int

```
if abb.raiz == null then
  return 0
endif

return abb.cardinalElementos(abb, hasta, false) - abb.cardinalDesde(abb, desde, true)
```

proc insertar (inout abb: ArbolBinarioBalanceado, in valor: int)

```
abb.raiz:= insertarEnSubArbol(abb.raiz, valor)
```

proc insertarEnSubArbol (in raiz: Nodo, in valor: int) : Nodo

```
if raiz == null then
  var nuevoNodo: Nodo
      nuevoNodo:= new Nodo()
      nuevoNodo.valor:= valor
      nuevoNodo.altura:= 1
      nuevoNodo.cardinal:= 1
  return nuevoNodo
endif

var cardinalActual: int
    cardinalActual:= abb.cardinal

if raiz.valor > valor then
  raiz.izquierda:= insertarEnSubArbol(raiz.izquierda, valor)
elseif raiz.valor < valor then
  raiz.derecha:= insertarEnSubArbol(raiz.derecha, valor)

if abb.cardinal > cardinalActual then
  raiz.altura:= recalcularAltura(raiz)
  raiz.cardinal:= raiz.cardinal + 1
endif

return balancear(raiz)
```

% Faltaría actualizar el cardinal al rotar %

proc rotacionDerecha (in y: Nodo) : Nodo

```
var x: Nodo

x:= y.izquierda;
y.izquierda:= x.derecha;
x.derecha:= y;

y.altura:= recalcularAltura(y);
x.altura:= recalcularAltura(x);

return x
```

% Faltaría actualizar el cardinal al rotar %

proc rotacionIzquierda (in x: Nodo) : Nodo

```
var y: Nodo

y:= x.derecha;
x.derecha:= y.izquierda;
y.izquierda:= x;

x.altura:= recalcularAltura(x);
y.altura:= recalcularAltura(y);

return y
```

proc balancear (in raiz: Nodo) : Nodo

```
var fb: int
    fb:= factorDeBalanceo(raiz)

if fb < -1 && factorDeBalanceo(raiz.izquierda) <= 0 then
    raiz:= rotacionDerecha(raiz);
elseif fb > 1 && factorDeBalanceo(raiz.derecha) >= 0
    raiz:= rotacionIzquierda(raiz);
elseif fb < -1 && factorDeBalanceo(raiz.izquierda) > 0
    raiz.izquierda:= rotacionIzquierda(raiz.izquierda);
    raiz:= rotacionDerecha(raiz);
elseif fb > 1 && factorDeBalanceo(raiz.derecha) < 0
    raiz.derecha:= rotacionDerecha(raiz.derecha);
    raiz:= rotacionIzquierda(raiz);
endif

raiz.altura:= recalcularAltura(raiz);

return raiz;
```

}