Guia 7

May 2024

1. Ejercicio 1

NodoLista< T > es struct <
    val: T,
    siguiente: NodoLista< T >
>

Modulo ListaEnlazada< T > **Implementa** Secuencia< T >
    var primero: NodoLista< T >
    var ultimo: NodoLista< T >
    var longitud: $\mathbb{Z}$

`proc nuevaListaVacia` () : ListaEnlazada< T >

```
1 var res: ListaEnlazada<T>
2 res.primero := null
3 res.ultimo := null
4 res.longitud := 0
5 return res
```

- **Complejidad O(1)**

`proc agregarAdelante` (inout l:ListaEnlazada< T >, in t:T)

```
1  var nuevo: NodoLista<T>
2  nuevo.val := t
3  nuevo.siguiente := l.primero
4
5  if (l.longitud == 0) do
6      l.primero := nuevo
7      l.ultimo := nuevo
8  else
9      l.primero := nuevo
10 endif
11 l.longitud += 1
```

- **Complejidad O(1)**

`proc agregarAtras` (inout l:ListaEnlazada< T >, in t:T)

```
var nuevo: NodoLista<T>
nuevo.val := t
nuevo.siguiente := null

if (l.longitud == 0) do
    l.primero := nuevo
    l.ultimo := nuevo
else
    l.ultimo.siguiente := nuevo
    l.ultimo := nuevo
endif
l.longitud += 1
```

- **Complejidad O(1)**

`proc eliminar` (inout l:ListaEnlazada< T >, in i:$\mathbb{Z}$)

```
var j: int
var actual: NodoLista<T>
var anterior: NodoLista<T>

j := 0
actual := l.primero

while (j < i) do
        anterior := actual
        actual := actual.siguiente
        j := j + 1
endwhile

if (actual == l.primero) do
    l.primero := actual.siguiente
    actual.siguiente := null
else if (actual == l.ultimo) do
     anterior.siguiente := null
     l.ultimo := anterior
else
     anterior.siguiente := actual.siguiente
     actual.siguiente := null
```

- **Complejidad O(n)**

`proc pertenece` (in l:ListaEnlazada< T >,in t:T) : Bool

```
var actual: NodoLista<T>
var i: int

i := 0
actual := l.primero
```

```
6
7  while (i < l.longitud) do
8        if (actual.val == t)
9           return true
10       else
11          actual := actual.siguiente
12       i := i + 1
13 endwhile
14
15 if (actual != null) do
16    return actual.val == t
17 else
18    return false
```

- **Complejidad O(n)**

proc obtener (in l:ListaEnlazada< T >, in i:$\mathbb{Z}$) : T

```
1  var j: int
2  var actual: NodoLista<T>
3
4  j := 0
5  actual := l.primero
6
7  while (j < i) do
8        actual := actual.siguiente
9        j := j + 1
10 endwhile
11
12 return actual.val
```

- **Complejidad O(n)**

proc concatenar (inout l1:ListaEnlazada< T >, in l2:ListaEnlazada< T >)

```
1  l1.ultimo.siguiente := l2.primero
2  l1.ultimo := l2.ultimo
```

- **Complejidad O(1)** pero se crea aliasing (para evitarlo deberia copiar todo pero la complejidad seria O(m))

proc sublista (in l:ListaEnlazada< T >, in inicio:$\mathbb{Z}$,in fin:$\mathbb{Z}$) : ListaEnlazada< T >

```
1  var res: ListaEnlazada<T>
2  var i: int
3  var f: int
4  var actual: NodoLista<T>
5
```

```
6  i := 0
7  f := 0
8  primero := l.primero
9  ultimo := l.primero
10
11 while (i < inicio) do
12         primero := primero.siguiente
13         i := i + 1
14 endwhile
15
16 while (f < fin) do
17         ultimo := ultimo.siguiente
18         f := f + 1
19 endwhile
20
21 res.primero := primero
22 res.ultimo := ultimo
23 res.longitud := |fin − inicio| + 1
```

- **Complejidad O(n)**

2. Ejercicio 2

- Con el primer nodo debo poder recorrer toda la lista hasta llegar al ultimo.
- La longitud debe ser igual a la cantidad de Nodos.
- El siguiente de ultimo apunta siempre a null.

`pred invRep` (l:ListaEnlazada< T >) {
$accesible(l.primero, l.ultimo) \ \wedge \ largoOk(l.primero, l.longitud) \ \wedge \ (l.ultimo \neq null \longrightarrow_L l.ultimo.siguiente = null)$
}

3. Ejercicio 3

NodoLista< T > es struct <
val: T,
siguiente: NodoLista< T >
anterior: NodoLista< T >
>

Modulo ListaDobleEnlazada< T > **Implementa** Secuencia< T >
var primero: NodoLista< T >
var ultimo: NodoLista< T >
var longitud: $\mathbb{Z}$

- Con el primer Nodo debo poder llegar al ultimo y viceversa.
- La longitud debe indicar la cantidad de nodos.
- El anterior de primero debe ser null y el siguiente de ultimo debe ser null, nadie tiene como siguiente al primero y el nadie tiene como anterior a Ultimo.

`proc agregarAdelante` (inout l:ListaDobleEnlazada< T >, in t:T)

```
1  var nuevo: NodoLista<T>
2
3  nuevo := new NodoLista()
4  nevo.val := t
5
6  if (l.longitud == 0) do
7      l.primero := nuevo
8      l.ultimo := nuevo
9      l.longitud := 1
10
11 else
12     nuevo.siguiente := l.primero
13     l.primero.anterior := nuevo
14     l.primero := nuevo
15     l.longitud := l.longitud + 1
16 endif
```

- **Complejidad O(1)**

`proc agregarAtras` (inout l:ListaDobleEnlazada< T >, in t:T)

```
1  var nuevo: NodoLista<T>
2
3  nuevo := new NodoLista<T>()
4  nuevo.val := t
5
6  if (l.longitud == 0) do
7      l.primero := nuevo
8      l.ultimo := nuevo
9      l.longitud := 1
10
11 else
12     l.ultimo.siguiente := nuevo
13     nuevo.anterior := l.ultimo
14     l.ultimo := nuevo
15     l.longitud := l.longitud + 1
16 endif
```

- **Complejidad O(1)**

`proc eliminar` (inout l:ListaDobleEnlazada< T >, in e:$\mathbb{Z}$)

```
var i: int
var actual := NodoLista<T>

i := 0
actual := new NodoLista<T>()
actual := l.primero

while (i < e) do
        actual := actual.siguiente
        i := i + 1
endwhile

if (actual == l.primero) do
    l.primero := l.primero.siguiente
    l.primero.anterior := null
    l.longitud := l.longitud − 1

else if (actual == l.ultimo) do
    l.ultimo := l.ultimo.anterior
    l.ultimo.siguiente := null
    l.longitud := l.longitud − 1

else
    actual.siguiente.anterior := actual.anterior
    actual.anterior.siguiente := actual.siguiente
    l.longitud := l.longitud − 1
endif
```

- **Complejidad O(1)**

- Obtener y Pertenece es igual que Lista Simple Enlazada con la misma complejidad.

`proc sublista` (in l:ListaDobleEnlazada$< T >$, in inicio:$\mathbb{Z}$,in fin:$\mathbb{Z}$) : ListaDobleEnlazada$< T >$

```
var res: ListaDobleEnlazada<T>

res := new nuevaListaVacia()

var i: int
var actual: NodoLista<T>

i := 0
actual := new NodoLista<T>()
actual := l.primero

while (i < inicio) do
        actual := actual.siguiente
        i := i + 1
endwhile

```

```
17 res.primero := actual
18 res.primero.anterior := null
19
20 while (i < fin ) do
21       actual := actual.siguiente
22       i := i + 1
23 endwhile
24
25 res.ultimo := actual
26 res.ultimo.siguiente := null
27
28 res.longitud := fin − inicio + 1
29
30 return res
```

- **Complejidad O(n)**

`proc concatenar` (inout l1:ListaDobleEnlazada< T >, in l2:ListaDobleEnlazada< T >)

```
1  if (l1.longitud == 0) do
2     l1 := l2
3
4  else if (l2.longitud == 0) do
5     skip
6
7  else
8     l1.ultimo.siguiente := l2.primero
9     l2.primero.anterior := l1.ultimo
10    l1.ultimo := l2.ultimo
11    l1.longitud := l1.longitud + l2.longitud
12 endif
```

- **Complejidad O(1)** Pero!! genera aliasing , si quiero evitar aliasing deberia copiar los elementos de l2 a l1 pero la complejidad cambiaria a O(n)

Modulo ListaCircular< T > **Implementa** Secuencia< T >
var primero: NodoLista< T >
var ultimo: NodoLista< T >
var longitud: $\mathbb{Z}$

- El invariante queda igual solo que ahora el siguiente de ultimo debe ser el primer Nodo.

`proc nuevaListaVacia` () : ListaCircular< T >

```
1 var res: ListaCircular<T>
2 res := new ListaCircular <T>
3
4 res.primero := null
```

```
5  res.ultimo := null
6  res.longitud := 0
7
8  return res
```

- **Complejidad O(1)**

`proc agregarAdelante` (inout l:ListaCircular< T >, in t:T)

```
1  var nuevo: NodoLista<T>
2
3  nuevo := new NodoLista<T>
4  nuevo.val := t
5
6  if (l.longitud == 0) do
7      l.primero := nuevo
8      l.ultimo  := nuevo
9      l.utlimo.siguiente := nuevo
10     l.longitud := 1
11
12 else
13     nuevo.siguiente := l.primero
14     l.primero := nuevo
15     l.ultimo.siguiente := nuevo
16     l.longitud := l.longitud + 1
17
18 endif
```

- **Complejidad O(1)**

`proc agregarAtras` (inout l:ListaCircular< T >, in t:T)

```
1  var nuevo: NodoLista<T>
2
3  nuevo := new NodoLista<T>
4  nuevo.val := t
5
6  if (l.longitud == 0) do
7      l.primero := nuevo
8      l.ultimo  := nuevo
9      l.ultimo.siguiente := nuevo
10     l.longitud := 1
11
12 else
13     l.ultimo.siguiente := nuevo
14     nuevo.siguiente := l.primero
15     l.ultimo := nuevo
16     l.longitud := l.longitud + 1
17 endif
```

- **Complejidad O(1)**

`proc concatenar` (inout l1:ListaCircular< T >, in l2:ListaCircular< T >)

```
1
2  if (l1.longitud == 0) do
3      l1.primero := l2.primero
4      l1.ultimo := l2.ultimo
5      l1.longitud := l2.longitud
6
7  else if (l2.longitud != 0) do
8      l1.ultimo.siguiente := l2.primero
9      l1.ultimo := l2.ultimo
10     l1.ultimo.siguiente := l1.primero
11     l.longitud := l1.longitud + l2.longitud
12
13 else
14    skip
15 endif
```

- **Complejidad O(1)** Pero se crea aliasing , bla bla :P

`proc subsecuencia` (in l1:ListaCircular< T >, in inicio,in fin) : ListaCircular< T >

```
1  var i: int
2  var res: ListaCircular<T>
3  var actual: NodoLista<T>
4
5  i := 0
6  actual := l.primero
7
8  while (i < inicio) do
9        actual := actual.siguiente
10       i := i + 1
11 endwhile
12
13 res.primero := actual
14
15 while (i < fin) do
16       actual := actual.siguiente
17       i := i + 1
18 endwhile
19
20 res.ultimo := actual
21 res.ultimo.siguiente := res.primero
22 res.longitud := fin - inicio + 1
23
24 return res
```

- **Complejidad O(n)**

NodoLista< T > **es struct** {
    datos: Array< T >
    sig: NodoLista< T >
    ant: NodoLista< T >
    largo: $\mathbb{Z}$
}

- Agregue la variable largo para ahorrarme escribir dos variables en listaArr que me digan que posiciones estan usadas en los array y voy a establecer el largo de los array en 10 pero podria hacerlo mas dinamico.

Modulo ListaArr< T > **Implementa** Secuencia< T >
    var primero: NodoLista< T >
    var ultimo: NodoLista< T >
    var total: $\mathbb{Z}$

`proc nuevaListaVacia` () : ListaArr< T >

```
var res: ListaArr<T>

res := new ListaArr<T>()
res.primero := null
res.ultimo := null
res.total := 0

return res
```

`proc agregarAdelante` (inout l:ListaArr< T >, in t:T)

```
if (l.total == 0) do
    var nuevo: NodoLista<T>

    nuevo := new NodoLista<T>(10)
    nuevo.datos[0] := t

    l.primero := nuevo
    l.ultimo := nuevo
    l.primero.largo := 1

else if (l.primero.largo == 10) do
    var nuevo: NodoLista<T>

    nuevo := new NodoLista<T>(10)
    nuevo.datos[0] := t

    l.primero.anterior := nuevo
    nuevo.siguiente := l.primero
    l.primero := nuevo
    l.primero.largo := 1
```

```
21
22 else
23     var i: int
24
25     i := l.primero.largo - 1
26     while (i >= 0) do
27         l.primero.datos[i + 1] := l.primero.datos[i]
28         i := i - 1
29     endwhile
30
31     l.primero.datos[0] := t
32     l.primero.largo := l.primero.largo + 1
33
34 endif
35 l.total := l.total + 1
```

proc agregarAtras (inout l:ListaArr< T >, in t:T)

```
1  if (l.total == 0) do
2      var nuevo: NodoLista<T>
3
4      nuevo := new NodoLista<T>(10)
5      nuevo.datos[0] := t
6
7      l.primero := nuevo
8      l.ultimo := nuevo
9      l.primero.largo := 1
10
11 else if (l.primero == l.ultimo) do
12
13     if (l.primero.largo < 10) do
14         l.primero.datos[l.primero.largo] := t
15         l.primero.largo := l.primero.largo + 1
16     else
17         var nuevo: NodoLista<T>
18
19         nuevo := new NodoLista<T>(10)
20         nuevo.datos[0] := t
21
22         l.primero.siguiente := nuevo
23         nuevo.anterior := l.primero
24         l.ultimo := nuevo
25         l.ultimo.largo := 1
26      endif
27 else
28     if (l.ultimo.largo == 10) do
29         var nuevo: NodoLista<T>
30
31         nuevo := new NodoLista<T>(10)
32         nuevo.datos[0] := t
```

```
33
34          l.ultimo.siguiente := nuevo
35          nuevo.anterior := l.ultimo
36          l.ultimo := nuevo
37          l.ultimo.largo := 1
38
39      else
40          l.ultimo.datos[l.ultimo.largo] := t
41          l.ultimo.largo := l.ultimo.largo + 1
42  endif
43  l.total := l.total + 1
```

`proc eliminar` (inout l:ListaArr< T >, in e:$\mathbb{Z}$)

```
1  eliminarNodo(l.primero,e)
```

`proc eliminarNodo` (inout actual:NodoLista< T >, in e:$\mathbb{Z}$)

```
1  if (e < actual.largo) do
2      if (actual.datos[e] != null) do
3          actual.datos[e] := actual.datos[actual.largo - 1]
4
5          var i: int
6
7          i := e
8          while (i < actual.largo - 1) do
9                  var valor: int
10
11                 valor := actual[i]
12                 actual[i] := actual[i+1]
13                 actual[i+1] := valor
14                 i := i + 1
15         endwhile
16
17     else
18         eliminarNodo(actual.siguiente,e - actual.largo)
19     endif
20  else
21      eliminarNodo(actual.siguiente,e - 10)
22  endif
```

`proc pertenece` (in l:ListaArr< T >, in t:T) : Bool

```
1  perteneceNodo(l.primero,t)
```

`proc perteneceNodo` (in actual:NodoLista< T >, in t:T) : Bool

```
1  if (actual == null) do
2      return false
3  else
4      var i: int
```

```
5
6     i := 0
7     while (i < actual.largo) do
8           if (actual.datos[i] == t) do
9              return true
10          else
11             skip
12          endif
13          i := i + 1
14     endwhile
15 endif
16 return perteneceNodo(actual.siguiente, t)
```

`proc obtener` (in l:ListaArr< T >, in e:$\mathbb{Z}$) : T

```
1 obtenerElem(l.primero, t)
```

`proc obtenerElem` (in actual: NodoLista< T >, in e:$\mathbb{Z}$) : T

```
1 if (e < actual.largo) do
2     if (actual.datos[e] != null) do
3        return actual.datos[e]
4     else
5        return obtenerElem(actual.siguiente, e − actual.largo)
6 else
7     return obtenerElem(actual.siguiente, e − 10)
```

4. Ejercicio 4

- a) Usaria el modulo ListaDeArr ya que como las secuencias no son acotadas quiero que agregar tenga costo O(1).
- b) **Modulo** ColaImpl< T > **implementa** Cola< T >

  var datos: ListaCircular< T >

  var longitud: $\mathbb{Z}$

  `proc colaVacia` () : ColaImpl< T >

```
1 res: ListaCirclar<T>
2
3 res := new nuevaListaVacia()
4 res.longitud := 0
5
6 return res
```

  `proc vacia` (in c:ColaImpl< T >) : Bool

```
1 return c.longitud == 0
```

`proc encolar` (inout c:ColaImpl< T >, in t:T)

```
c.datos.agregarAtras(t)
c.longitud := c.longitud + 1
```

`proc desencolar` (inout c:ColaImpl< T >) : T

```
var res: T

res := c.datos.obtener(0)
c.datos.eliminar(0)
c.longitud := c.longitud − 1

return res
```

`proc proximo` (in c:ColaImpl< T >) : T

```
var res: T

res := c.datos.obtener(0)

return res
```

- c) **Modulo** ConjImpl< T >**implementa** Conjunto< T >
  var datos: ListaEnlazada< T >
  var longitudo: $\mathbb{Z}$

`proc conjVacio` () : ConjImpl

```
var res: ConjuntoImpl

res.datos := nuevaListaVacia()
res.longitud := 0

return res
```

`proc pertenece` (in c:ConjImpl, in t:T) : Bool

```
return c.datos.pertenece(t)
```

`proc agregar` (inout c:ConjImpl, in t:T)

```
if (c.datos.pertenece(t) == false) do
    c.datos.agregarAdelante(t)
    c.longitud := c.longitud + 1
else
    skip
endif
```

`proc sacar` (inout c:ConjImpl, in e:T)

```
c.datos.eliminar(e)
c.longitud := c.longitud − 1
```

proc unir (inout c1:ConjImpl, in c2:ConjImpl)

```
if (c1.longitud == 0) do
    c1.datos := c2.datos
    c1.longitud := c2.longitud

else if (c2.longitud != 0) do
    var i: int

    i := 0
    while (i < c2.longitud) do
          if (c1.datos.pertenece(c2.datos.obtener(i)) == false) do
             c1.datos.agregarAdelante(c2.datos.obtener(i))
             c1.longitud := c1.longitud + 1
          else
             skip
          endif
          i := i + 1
     endwhile
else
    skip
endif
```

proc restar (inout c1:ConjImpl, in c2:ConjImpl)

```
var i: int

i := 0
while (i < c1.longitud) do
       if (c2.datos.pertenece(c1.datos.obtener(i)) == true) do
           c1.datos.eliminar(i)
           c1.longitud := c1.longitud − 1
       else
           skip
       endif
          i := i + 1
    endwhile
```

proc intersecar (inout c1:ConjImpl, in c2:ConjImpl)

```
if (c1.datos.longitud == 0 || c2.datos.longitud == 0) do
    c1.datos := c.datos.nuevaListaVacia()
    c1.longitud := 0

else
    var i: int

    i := 0
    while (i < c1.longitud) do

          if (c2.datos.pertenece(c1.datos.obtener(i)) do

```

```
            c1.datos.eliminar(i)
            c1.longitud := c1.longitud − 1
        else
            skip
        endif
        i := i + 1
    endwhile
endif
```

`proc agregarRapido` (inout c:ConjImpl, in t:T)

```
c.datos.agregarAdelante(t)
```

`proc tamanio` (in c:ConjImpl) : $\mathbb{Z}$

```
var res: int

res := c.datos.longitud

return res
```

- **Modulo** DiccImpl$< K, V >$ **Implementa** Diccionario$< K, V >$

  var datos: ListaEnlazadaSimple$< (K, V) >$

  var longitud: $\mathbb{Z}$

`proc diccionarioVacio` () : DiccImpl

```
var res: DiccImpl

res.datos := nuevaListaVacia()

return res
```

`proc esta` (in d:DiccImpl, in k:K) : Bool

```
var i: int
var actual: Tupla (K,V)

i := 0

while (i < d.longitud) do
        actual := d.obtener(i)
        if (actual[0] == k) do
            return true

        else
            skip
        endif
        i := i + 1
```

```
endwhile
return false
```

proc definir (inout d:DiccImpl, in k:K, in v:V)

```
if (!d.datos.esta(k)) do
    var nuevo: (K,V)

    nuevo := (k,v)
    d.datos.agregarAdelante(nuevo)
    d.longitud := d.longitud + 1

else
    var i: int
    var actual: (K,V)

    i := 0
    while (i < d.longitud) do
          actual := d.datos.obtener(i)
          if (actual[0] == k) do
             actual[1] := v
          else
             skip
          endif
          i := i + 1
    endwhile
    d.longitud := d.longitud + 1
endif
```

proc obtener (in d:DiccImpl, in k:K) : V

```
var i: int
var actual: (K,V)

i := 0
while (i < d.longitud) do
      actual := d.datos.obtener(i)
      if (actual[0] == k) do
         return actual[1]
      else
         skip
      endif
      i := i + 1
endwhile
```

proc borrar (inout d:DiccImpl, in k:K)

```
var i: int
```

```
var actual: (K,V)

i := 0
while (i < d.longitud) do
        actual := d.datos.obtener(i)
        if (actual[0] == k) do
            d.datos.eliminar(i)
        else
            skip
        endif
        i := i + 1
endwhile
```

`proc definirRapido` (inout d:DiccImpl, in k:K, in v:V)

```
var nuevo: (K,V)

nuevo := (k,v)

d.datos.agregarAdelante(nuevo)
```

- MultiConjunto usaria lista Simple y los procs serian los mismos que el modulo ConjImpl solo que sacaria el proc agregar y solo dejaria agregarRapido , y agregaria un proc multiplicidad que recorra la lista con un while contando cuantas veces aparece el elemento.

5. Ejercicio 5

Modulo PilaImpl< T > **implementa** Pila< T >
    var data: ListaArr< T >
    var tope: $\mathbb{Z}$

`proc pilaVacia` () : PilaImpl< T >

```
var res: PilaArr<T>

res.data := new nuevaListaVacia()
res.tope := 0

return res
```

`proc apilar` (inout p:PilaImpl< T >, in e:T)

```
p.data.agregarAtras(e)
p.tope := p.tope + 1
```

`proc desapilar` (inout p:PilaImpl< T >) : T

```
var res: T

res := p.data.obtener(p.tope − 1)
p.data.eliminar(p.tope − 1)
p.tope := p.tope − 1

return res
```

6. Ejercicio 6

Vagon **es** String
Tren **es** ListaEnlazadaDoble< $Vagon$ >

Modulo PlayaDeManiobrasImpl **implementa** PlayaDeManiobras
var trenes: Array< $Tren$ >

`proc abrirPlaya` (in capacidad:ℤ) : PlayaDeManiobrasImpl

```
var res: PlayaDeManiobrasImpl

res.datos := new Array<Tren>(capacidad)
res.via := 0

return res
```

`proc recibirTren` (inout pdm: PlayaDeManiobrasImpl, in t:Tren)

```
var res: int
var v: int

v := 0
while ( pdm.datos[v] != null ) do
    v := v + 1
endwhile

res := v
pdm.datos[v] := t

return res
```

`proc despacharTren` (inout pdm:PlayaDeManiobrasImpl, in v:ℤ)

```
pdm.datos[v] := pdm.datos[v].nuevaListaVacia()
```

`proc unirTrenes` (inout pdm:PlayaDeManiobras, in via1:ℤ, in via2:ent)

```
pdm.datos[via1].concatenar(pdm.datos[via2])
pdm.despacharTren(via2)
```

proc moverVagon (inout pdm:PlayaDeManiobras, in v:Vagon, in viaDestino)

```
var via: int

via := 0
while (pdm.datos[via].pertenece(v) == false) do
        via := via + 1
endwhile

pdm.datos[via].eliminar(v)
pdm.datos[viaDestino].agregarAtras(v)
```

7. Ejercicio 7

- En el invariante debo asegurarme que cada nodo apunte a izquierda o derecha , no puede apuntar hacia arriba ni hacia otros punteros en su mismo nivel.
- Desde la raiz debo poder llegar hacia todos los Nodos.

proc altura (in ab:ArbolBinario< T >) : $\mathbb{Z}$

```
var res: int

res := alturaRaiz(ab.raiz)

return res
```

proc alturaRaiz (in nodo:NodoAB< T >) : $\mathbb{Z}$

```
if (nodo == null) do
    return 0
else
    return 1 + max(alturaRaiz(nodo.der),alturaRaiz(nodo.izq))
```

proc cantidadHojas (in ab:ArbolBinario< T >) : $\mathbb{Z}$

```
var res: int

res := hojas(ab.raiz)

return res
```

proc hojas (in nodo:NodoAB< T >) : $\mathbb{Z}$

```
if (nodo.der == null && nodo.izq == null) do
    return 1

else if (nodo == null) do
    return 0

```

```
7 | else
8 |     return hojas(nodo.der) + hojas(nodo.izq)
9 | endif
```

`proc esta` (in ab:ArbolBinario< T >, in e:T) : bool

```
1 | var res: bool
2 |
3 | res := estaNodo(ab.raiz)
4 |
5 | return res
```

`proc estaNodo` (in nodo:NodoAB< T >, in e:T) : bool

```
1 | if (nodo == null) do
2 |     return false
3 |
4 | else if (nodo.val == e) do
5 |     return true
6 |
7 | else
8 |     return estaNodo(nodo.der,e) || estaNodo(nodo.izq,e)
9 | endif
```

`proc cantidadApariciones` (in ab:ArbolBinario< T >, in e:T) : $\mathbb{Z}$

```
1 | var res: int
2 |
3 | res := cantidad(ab.raiz,e)
4 |
5 | return res
```

`proc cantidad` (in nodo:NodoAB< T >,in e:T) : $\mathbb{Z}$

```
1 | if (nodo == null) do
2 |     return 0
3 |
4 | else if (nodo.val == e) do
5 |     return 1 + cantidad(nodo.izq,e) + cantidad(nodo.der,e)
6 | else
7 |     return cantidad(nodo.der,e) + cantidad(nodo.izq,e)
```

`proc ultimoNivelCompleto` (in ab:ArbolBinario< T >) : $\mathbb{Z}$

```
1 | var res: int
2 |
3 | res := ultimoNivel(ab.raiz,i)
4 |
5 | return res + 1
```

proc ultimoNivel (in nodo:NodoAB< T >) : $\mathbb{Z}$

```
1  if (nodo.der == null || nodo.izq == null)
2      return 0
3
4  else
5      return 1 + min(ultimoNivel(nodo.der),ultimoNivel(nodo.izq))
```

- Todas tienen complejidad **O(n)**.

8. Ejercicio 8

proc esta (in ab:ABB< T >, in e:T) : bool

```
1  var res: bool
2
3  res := estaABB(ab.raiz,e)
4
5  return res
```

proc estaABB (in nodo:NodoAB< T >, in e:T) : bool

```
1  if (nodo == null) do
2      return false
3
4  else if (nodo.val == e) do
5      return true
6
7  else
8      if (nodo.val <= e) do
9          return estaABB(nodo.izq,e)
10     else
11         return estaABB(nodo.der,e)
12 endif
```

- En peor caso tiene complejidad **O(n)**.
- En mejor caso **O(1)**.

proc cantidadApariciones (in ab:ABB< T >, in e:T) : $\mathbb{Z}$

```
1  var res: int
2
3  res := cantidad(ab.raiz,e)
4
5  return res
```

proc cantidad (in nodo:NodoAB< T >, in e:T) : $\mathbb{Z}$

```
1  if (nodo == null) do
2      return 0
```

```
3
4  else if (nodo.val == e) do
5      return 1 + cantidad(nodo.izq,e)
6
7  else
8      if (nodo.val <= e) do
9          return cantidad(nodo.izq,e)
10     else
11         return cantidad(nodo.der,e)
12 endif
```

- En peor y mejor caso tiene complejidad **O(n)**.

`proc insertar` (inout ab:ABB< T >, in e:T)

```
1  insertarABB(ab.raiz,e)
```

`proc insertarABB` (inout nodo:NodoAb< T >, in e:T)

```
1  if (nodo == null) do
2      nodo.val := e
3      nodo.izq := null
4      nodo.der := null
5
6  else if (nodo.val >= e && nodo.izq == null) do
7      var nuevo: NodoAB<T>
8
9      nuevo := new NodoAB<T>()
10     nuevo.val := e
11     nodo.izq := nuevo
12
13 else if (nodo.val < e && nodo.der == null) do
14     var nuevo: NodoAB<T>
15
16     nuevo := new NodoAB<T>
17     nuevo.val := e
18     nodo.der := nuevo
19
20 else if (nodo.val >= e) do
21     return insertarABB(nodo.izq,e)
22
23 else
24     return insertarABB(nodo.der,e)
25
26 endif
```

- En peor caso tiene complejidad **O(n)**.
- En mejor caso tiene complejidad **O(1)**.

```
proc eliminar (inout ab:ABB< T >, in e:T)
1 | eliminarABB(nodo)

proc eliminarABB (inout nodo: NodoAB< T >, in e:T)
1  | var actual: NodoAB<T>
2  | var anterior: NodoAB<T>
3  |
4  | actual := nodo
5  | anterior := null
6  |
7  | while (actual.val != e) do
8  |       if (actual.val > e) do
9  |          anterior := actual
10 |          actual := actual.izq
11 |
12 |       else
13 |          anterior := actual
14 |          actual := actual.der
15 |       endif
16 | endwhile
17 |
18 | if (actual.izq == null && actual.der == null) do
19 |    if (anterior == null) do
20 |       actual := null
21 |
22 |    else if (anterior.der == actual) do
23 |       anterior.der := null
24 |
25 |    else
26 |       anterior.izq := null
27 |    endif
28 |
29 | else if (actual.der == null) do
30 |    if (anterior == null) do
31 |       actual := actual.izq
32 |
33 |    else if (anterior.der == actual) do
34 |       anterior.der := actual.izq
35 |
36 |    else
37 |       anterior.izq := actual.izq
38 |    endif
39 |
40 | else if (actual.izq := null) do
41 |    if (anterior == null) do
42 |       actual := actual.der
43 |
44 |    else if (anterior.der == actual) do
45 |       anterior.der := actual.der
46 |
```

```
47      else
48          anterior.izq := actual.der
49      endif
50
51  else
52      cambiarSucesor(actual)
53  endif
```

`proc cambiarSucesor` (in actual:NodoAB< T >)

```
1  var sucesor: Nodo<T>
2
3  sucesor := buscarMinimo(actual.der)
4
5  actual.val := sucesor.val
6
7  if (sucesor.der == null) do
8      sucesor := null
9  else
10     sucesor := sucesor.der
11 endif
```

`proc buscarMinimo` (in nodo:NodoAB< T >) : NodoAB< T >

```
1  var actual: NodoAB<T>
2
3  actual := nodo
4
5  while(actual.izq != null) do
6        actual := actual.izq
7  endwhile
8  return actual
```

- En peor caso tiene complejidad **O(n)**.
- En mejor caso **O(1)**.

9. Ejercicio 9

- **esta** cambia a **O(log(n))**
- **cantApariciones** cambia a **O(log(n))**
- **insertar** cambia a **O(log(n))**
- **eliminar** cambia a **O(log(n))**

10. Ejercicio 10

- A completar.

11. Ejercicio 11