



## Diseño con Listas Enlazadas

**Ejercicio 1.** Implementamos el TAD Secuencia sobre una **lista simplemente enlazada** usando

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

Modulo ListaEnlazada<T> implementa Secuencia<T>{
    var primero: NodoLista<T> // "puntero" al primer elemento
    var último: NodoLista<T> // "puntero" al primer elemento
    var longitud: ℤ // cantidad total de elementos
    ...
}
```

Escriba los algoritmos para los siguientes procs y calcule su complejidad

- nuevaListaVacía(): ListaEnlazada<T>
- agregarAdelante(inout l: ListaEnlazada<T>, in t: T)
- agregarAtras(inout l: ListaEnlazada<T>, in t: T)
- eliminar(inout l: ListaEnlazada<T>, in i: int)
- pertenece(in l: ListaEnlazada<T>, in t: T) : bool
- obtener(in l: ListaEnlazada<T>, in i: int) : T
- concatenar(inout l1: ListaEnlazada<T>, in l2: ListaEnlazada<T>)
- sublista(in l: ListaEnlazada<T>, in inicio : int, in fin: int): ListaEnlazada<T>

**Ejercicio 2.** Para el módulo ListaEnlazada definido en el ejercicio anterior

- Escriba el invariante de representación para este módulo en castellano
- Dado el siguiente invariante de representación, indique si es correcto. En caso de no serlo, corrija:

```
pred InvRep(l: ListaEnlazada<T>)
    {accesible(l.primero, l.ultimo) ∧ largoOK(l.primero, l.longitud)}

pred accesible(n0: NodoLista<T>, n1: NodoLista<T>)
    {n1 = n0 ∨ (n0.siguiente ≠ null ∧L accesible(n0.siguiente, n1))}

pred largoOK(n: NodoLista<T>, largo: ℤ)
    {(n = null ∧ largo = 0) ∨ (largoOK(n.siguiete, largo - 1))}
```

**Ejercicio 3.** Defina el módulo de las siguientes alternativas a una lista simplemente enlazada. Para eso será necesario:

- Elegir las variables de la estructura
- Escribir el invariante de representación en castellano
- En caso de ser necesario, reescribir los algoritmos de las operaciones implementadas en el ejercicio anterior
- Calcular las complejidades de las operaciones

- a) Lista **doblemente enlazada** que usa la siguiente definición del módulo `NodoLista`

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

- b) Lista **circular** (simple o doblemente enlazada) donde el “último” nodo está conectado con el primero

- c) Lista **de arreglos**: una lista doblemente enlazada cuyos valores son arreglos de tamaño fijo, es decir que el nodo es

```
NodoLista<T> es struct<
  data: Array<T>,
  siguiente: NodoLista<T>,
  anterior: NodoLista<T>,
>
```

**Ejercicio 4.** Implementar los siguientes TADs (cuyas especificaciones están en el apunte) sobre alguna forma de `ListaEnlazada` (simple, doble, etc.), explicando por qué eligió esa variante. Calcule las complejidades de los procs

- a) `Secuencia<T>`
- b) `Cola<T>`
- c) `Pila<T>`
- d) `Conjunto<T>`
- e) `Diccionario<K,V>`
- f) `Multiconjunto<T>`

**Ejercicio 5.** Implementar la pila no acotada (ejercicio 2 de la práctica 5) usando una lista de arreglos (ejercicio 3 inciso 3 de esta práctica)

**Ejercicio 6.** Considere la siguiente especificación

```
Vagón es string
Tren es seq<Vagón>

TAD PlayaDeManiobras {
  obs trenes: seq<Tren>

  proc abrirPlaya(in capacidad: ℤ) : PlayaDeManiobras
    requiere { capacidad > 1 }
    asegura { |ret.trenes| = capacidad ∧ (∀i: ℤ)(0 ≤ i < capacidad) →L (ret.trenes[i] = []) }
  proc recibirTren(inout pdm: PlayaDeManiobras, in t: Tren) : ℤ
    requiere { (∃v: ℤ)(0 ≤ v < |pdm.trenes|) ∧L (pdm.trenes[v] = []) ∧ pdm = pdm0 }
    requiere { TodosLosVagonesSonDistintos(pdm, t) }
    asegura { (∃v)(0 ≤ v < |pdm0.trenes|) ∧L
      (pdm0.trenes[v] = [] ∧ pdm.trenes = setAt(pdm0.trenes, v, t)) ∧ ret = v }

  proc despacharTren(inout pdm: PlayaDeManiobras, in v: ℤ)
    requiere { (0 ≤ v < |pdm.trenes|) ∧L (pdm[v] ≠ []) ∧ pdm = pdm0 }
    asegura { pdm.trenes = setAt(pdm0.trenes, v, []) }
```

```

proc unirTrenes(inout pdm: PlayaDeManiobras, in via1:  $\mathbb{Z}$ , in via2:  $\mathbb{Z}$ )
  requiere {  $(0 \leq via1 < |pdm.trenes|) \wedge 0 \leq via2 < |pdm.trenes|$  }
  requiere {  $pdm.trenes[via1] \neq [] \wedge pdm.trenes[via2] \neq []$  }
  requiere {  $via1 \neq via2$  }
  requiere {  $pdm = pdm_0$  }
  asegura {  $|pdm| = |pdm_0|$  }
  asegura {  $pdm.trenes[via1] = concat(pdm_0.trenes[via1], pdm_0.trenes[via2])$  }
  asegura {  $pdm.trenes[via2] = []$  }
  asegura {  $(\forall v : \mathbb{Z})(0 \leq v < |pdm.trenes| \wedge v \neq via1 \wedge v \neq via2) \longrightarrow_L$ 
     $(pdm.trenes[v] = pdm_0.trenes[v])$  }

proc moverVagon(inout pdm: PlayaDeManiobras, in vagon: Vagon, in viaDestino:  $\mathbb{Z}$ )
  requiere {  $0 \leq viaDestino < |pdm.trenes|$  }
  requiere {  $(\exists v : \mathbb{Z})(0 \leq v < |pdm.trenes|) \wedge_L (vagon \in pdm.trenes[v])$  }
  requiere {  $pdm = pdm_0$  }
  asegura {  $|pdm| = |pdm_0|$  }
  asegura {  $(\exists v : \mathbb{Z})(0 \leq v < |pdm_0.trenes|) \wedge_L (vagon \in pdm_0.trenes[v] \wedge vagon \notin pdm.trenes[v])$  }
  asegura {  $vagon \in pdm.trenes[viaDestino]$  }
  asegura {  $(\forall v : \mathbb{Z})(0 \leq v < |pdm.trenes| \wedge v \neq viaDestino \wedge vagon \notin pdm_0.trenes[v]) \longrightarrow_L$ 
     $(pdm.trenes[v] = pdm_0.trenes[v])$  }

pred TodosLosVagonesSonDistintos(pdm: PlayaDeManiobras, t: Tren)
  {  $(\forall vi : \mathbb{Z})(0 \leq vi < |pdm.trenes| \longrightarrow_L (\forall vg : Vagon)$ 
     $(vg \in pdm.trenes[vi] \longrightarrow vg \notin t) \wedge (vg \in t \longrightarrow vg \notin pdm.trenes[vi]))$  }
}

```

- Implementar el TAD PlayaDeManiobras usando listas enlazadas.
- Calcular la complejidad de las operaciones en peor caso en función de la cantidad de vías  $v$  y el largo del tren más largo  $t$
- Si la complejidad calculada para las operación `moverVagon()` es mayor a  $O(t)$  y/o la de `unirTrenes()` es mayor a  $O(1)$ , modifique la estructura para lograr estas complejidades.

## Diseño con Árboles binarios

**Ejercicio 7.** Implementamos un **Árbol Binario** (AB) con

```

NodoAB<T>es struct<
  val: T,
  izquierda: NodoAB<T>,
  derecha: NodoAB<T>,
>

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

  ...
}

```

(El TAD está definido en el anexo al final de la práctica)

- Escriba en castellano el invariante de representación para este módulo
- Escriba los algoritmos para los siguientes procs y, de ser posible, calcule su complejidad
  - `altura(in ab: ArbolBinario<T>): int` // devuelve la distancia entre la raíz y la hoja más lejana
  - `cantidadHojas(in ab: ArbolBinario<T>): bool`

- (3) `está(in ab: ArbolBinario<T>, int t: T): bool` // devuelve true si el elemento está en el árbol
- (4) `cantidadApariciones(in ab: ArbolBinario<T>, int t: T): int`
- (5) `últimoNivelCompleto(in ab: ArbolBinario<T>): int` // devuelve el número del último nivel que está completo (es decir, que tiene todos los nodos posibles). Considere la raíz como nivel 1

**Ejercicio 8.** 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

$$(\forall i : \mathbb{Z})((\text{estáPred}(N.\text{izq}, i) \longrightarrow i \leq N.\text{val}) \wedge (\text{estáPred}(N.\text{der}, i) \longrightarrow i > N.\text{val}))$$

Implemente los algoritmos para los siguientes procs y calcule su complejidad en mejor y peor caso

- a) `está(in ab: ABB<T>, int t: T): bool` // devuelve true si el elemento está en el árbol
- b) `cantidadApariciones(in ab: ABB<T>, int t: T): int`
- c) `insertar(inout ab: ABB<T>, int t: T)`
- d) `eliminar(inout ab: ABB<T>, int t: T)`
- e) `inOrder(in ab: ABB<T>) : Array<T>` // devuelve todos los elementos del árbol en una secuencia ordenada

**Ejercicio 9.** Asumiendo que el árbol está balanceado, recalcule, si es necesario, las complejidades en peor caso de los algoritmos del ejercicio 8.

**Ejercicio 10.** ¿Qué pasa en un ABB cuando se insertan valores repetidos? Proponga una modificación del módulo que resuelva este problema

**Ejercicio 11.** Implementar los siguientes TADs (cuyas especificaciones están en el apunte) sobre ABB. Calcule las complejidades de los procs en mejor y peor caso

- a) `Conjunto<T>`
- b) `Diccionario<K,V>`
- c) `ColaDePrioridad<T>`
- d) `Multiconjunto<T>`

**Ejercicio 12.** Recalcule, si es necesario, las complejidades en peor caso de los algoritmos de los TADs `Conjunto<T>` y `Diccionario<T>` implementados en el ejercicio 11, considerando que se implementan sobre AVL en vez de ABB.

## Anexo: TAD ArbolBinario

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

  proc nuevoArbolVacio(): 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 = r}
```

```

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 = ab.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 = ab.dato}

pred estaPred(ab: ArbolBinario<T>, e: T)
  {ab.vacio = false  $\wedge_L$  (ab.dato = e  $\vee$  estaPred(ab.izq, e)  $\vee$  estaPred(ab.der, e))}

pred esHoja(in ab: ArbolBinario<T>)
  {ab.raiz.izquierda = null  $\wedge$  ab.raiz.derecha = null}
}

```