



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

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

Módulo ListaEnlazada<T> implementa Secuencia<T> <
  var primero: NodoLista<T> // "puntero" al primer elemento
  var último: NodoLista<T> // "puntero" al primer elemento
  var longitud: int // cantidad total de elementos

proc NuevaListaVacía ( ): ListaEnlazada<T>
  res := new ListaEnlazada<T>
  res.primeros := null
  res.último := null
  res.longitud := 0
  return res

proc AgregarAdelante (inout l: ListaEnlazada<T>, in e: T):
  nodo := new NodoLista<T>
  nodo.value := e
  nodo.siguiente := null

  if l.longitud == 0 then
    l.primeros := nodo
    l.último := nodo
  else
    nodo.siguiente := l.primeros
    l.primeros := nodo
  end if
  l.longitud := l.longitud + 1

proc Pertenece (in l: ListaEnlazada<T>, in e: T): bool
  res := false
  actual := l.primeros
  while actual ≠ null do
    if actual.valor == e then
      res := true
    end if
    actual := actual.siguiente
  end while
  return res
>
```

- Escriba los algoritmos para los siguientes procs y calcule su complejidad
 - proc agregarAtras(inout l : ListaEnlazada<T>, in t : T)
 - proc obtener(in l : ListaEnlazada<T>, in i : \mathbb{Z}) : T
 - proc eliminar(inout l : ListaEnlazada<T>, in i : \mathbb{Z})
 - proc concatenar(inout l1 : ListaEnlazada<T>, in l2 : ListaEnlazada<T>)
- 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-lo:

```

pred InvRep (l: ListaEnlazada⟨T⟩) {
    accesible(l.primerO, l.ultimo) ∧ largoOK(l.primerO, l.longitud)
}
pred largoOK (n: NodoLista⟨T⟩, largo: ℤ) {
    (n = null ∧ largo = 0) ∨ (largoOK(n.siguiente, largo - 1))
}
pred accesible (n0: NodoLista⟨T⟩, n1: NodoLista⟨T⟩) {
    n1 = n0 ∨ (n0.siguiente ≠ null ∧L accesible(n0.siguiente, n1))
}

```

Ejercicio 2. Implemente el TAD `ConjuntoAcotado<T>` (definido en el apunte de TADs) usando la siguiente estructura.

```

Módulo ConjuntoArr<T> implementa ConjuntoAcotado<T> <
    var datos: Array<T>
    var tamaño: int
>

```

- Escriba el invariante de representación y la función de abstracción.
- Escriba los algoritmos para las operaciones `conjVacío` y `pertence`
- Escriba el algoritmo para la operación `agregar`
- Escriba los algoritmos para las operaciones `unir` e `intersecar`.
- Escriba el algoritmo para la operación `sacar`.
- Calcule la complejidad de cada una de estas operaciones
- Qué cambios haría en su implementación si se quiere que la operación `agregar` sea lo más rápida posible? Y si se quiere acelerar la operación `buscar`? Indique los cambios en la estructura, el invariante de representación, la función de abstracción y los algoritmos.

Ejercicio 3. Implementar el TAD `Conjunto⟨T⟩` (definido en el apunte de TADs) usando la siguiente estructura

```

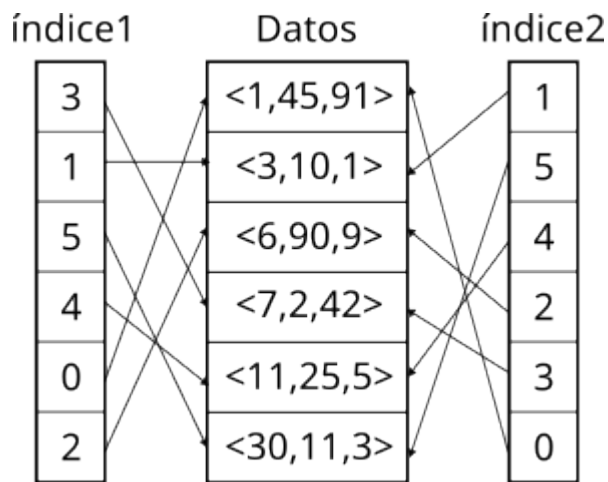
Módulo ConjuntoLista<T> implementa Conjunto⟨T⟩ <
    var datos: ListaEnlazada<T>
    var tamaño: int
>

```

- Escriba el invariante de representación y la función de abstracción.
- Escriba los algoritmos para las operaciones `conjVacío` y `pertence`
- Escriba el algoritmo para la operación `agregar`, `agregarRápido` y `sacar`
- Escriba los algoritmos para las operaciones `unir` e `intersecar`.
- Calcule la complejidad de cada una de estas operaciones

Ejercicio 4. Un *índice* es una estructura secundaria que permite acceder más rápidamente a los datos a partir de un determinado criterio. Básicamente un índice guarda *posiciones* o *punteros* a los elementos en un orden en particular, diferente al orden original.

Imagine una secuencia de tuplas con varias componentes, ordenada por su primer componente. Algunas veces vamos a querer buscar (rápido) por las demás componentes. Podríamos guardar los datos en sí en un arreglo y tener arreglos con las posiciones ordenadas por las demás componentes. A estos arreglos se los denomina índices.

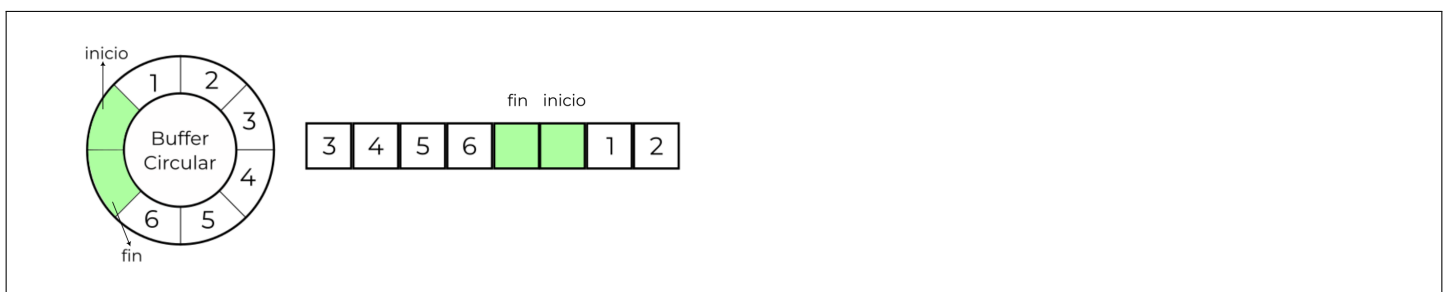


En la figura, si recorremos los datos en el orden en el que están guardados, obtenemos:
 [<1, 45, 91>, <3, 10, 1>, <6, 90, 9>, <7, 2, 42>, <11, 25, 5>, <30, 11, 3>]

Si lo recorremos usando el índice 1 (que apunta a los elementos en función de la segunda componente) obtenemos:
 [<7, 2, 42>, <3, 10, 1>, <30, 11, 3>, <11, 25, 5>, <1, 45, 91>, <6, 90, 9>]

- Escriba la estructura propuesta
- Escriba el invariante de representación y la función de abstracción, en castellano y en lógica para el TAD Conjunto(Tupla(\mathbb{Z} , \mathbb{Z} , \mathbb{Z}))
- Escriba el algoritmo de **BuscarPor** que busca por alguna componente
- Escriba los algoritmos de **agregar** y **sacar**

Ejercicio 5. Una forma eficiente de implementar el TAD Cola en su versión acotada (con una cantidad máxima de elementos predefinida), es mediante un *buffer circular*. Esta estructura está formada por un array del tamaño máximo de la cola (n) y dos índices (*inicio* y *fin*), para indicar adonde empieza y adonde termina la cola. El chiste de esta estructura es que, al llegar al final del arreglo, si los elementos del principio ya fueron consumidos, se puede reusar dichas posiciones.



- Elija una estructura de representación
- Escriba el invariante de representación y la función de abstracción
- Escriba los algoritmos de las operaciones **encolar** y **desencolar**
- ¿Por qué tiene sentido utilizar un buffer circular para una cola y no para una pila?

Ejercicio 6. Implementar los siguientes TADs (cuyas especificaciones están en el apunte) sobre arreglo y sobre lista enlazada. Calcule las complejidades de las operaciones

- Pila<T>
- Diccionario<K,V>