# Especificación e Implementación de TADs: Conjuntos

Cátedra AED

DC-UBA

2 cuat, 2023

# Outline

# Motivación

- Ver una implementación (simple) de un TAD muy usado
- Repasar conceptos
- Introducir Memoria Dinámica

# Valores Conceptuales

- Necesitamos un tipo Conceptual (matemático) para el/los observador/es
- En este caso, Conjunto de Naturales

# Presentación

TAD Set

- Obs set: Conj[$\mathbb{N}$]
  es el conjunto denotado
- **Proc EmptySet (): Set**
  asegura res.set = $\emptyset$
- **Proc Add (inout s: Set ,in d: $\mathbb{N}$)**
  asegura s.set = $s_0$.set $\cup\{d\}$
- **Proc In? (in s: Set, in d: $\mathbb{N}$): Bool**
  asegura res = True $\Leftrightarrow$ (d $\in$ s.set)
- **Proc Delete (inout s: Set ,in d: $\mathbb{N}$)**
  asegura s.set = $s_0$.set $\setminus\{d\}$

# Implementación de Conjuntos

- La implementación correcta y eficiente de este TAD y sus variantes, es uno de los principales temas de este curso
- Hoy vamos a explicar una forma, que nos va a permitir introducir un concepto fundamental para la gestión de la memoria en los lenguajes de programación: la Memoria Dinámica.

# Memoria Dinámica

- El espacio que va a requerir un programa para su ejecución, puede ser conocido de antemano, o no.
- La Memoria Dinámica es el mecanismo a través del cual los lenguajes imperativos nos proveen de primitivas para almacenar información cuando no sabemos de antemano cuánto espacio necesitamos, o incluso cuando lo sabemos pero la cota superior es muy grande (y por lo tanto la gestión estática de la memoria no resulta adecuada).
- Parte de una organización de la memoria dividida en dos partes:
  - El stack (pila) de memoria estática, y
  - El "heap" (montón) para la memoria dinámica

# Memoria Dinámica: el stack

- El stack se usa para la memoria estática.
- Cada vez que se entra a ejecutar una rutina (incluyendo el `main`), se asigna un "frame" o bloque, que contiene la memoria estática que pide el procedimiento, incluyendo espacio para los parámetros formales, que son cargados con los valores con los que se invocó (algunos de ellos, potencialmente referencias).
- Cuando finaliza la ejecución de la rutina, su "frame" se desapila y esa memoria se libera automáticamente. Los detalles son transparentes al programador (y a les estudiantes de AED)

# Memoria Dinámica: el heap

- El “Heap” (montón), es un espacio de memoria para almacenar objetos (estructuras y arreglos) a los que se accede por medio de referencias (o punteros)
- Estas elementos pueden contener, además de tipos básicos, referencias a otros elementos en el heap
- La memoria que es parte del Heap puede ser requerida de diversas maneras (por ejemplo, a través de la primitiva New y el tipo del dato que se quiere)
- Para dejar de utilizar la memoria dinámica, se procede según tres paradigmas principales

# Memoria Dinámica: devolución

- Los tres paradigmas principales respecto a la devolución de la memoria dinámica son:
  - Gestionada por programador ("a la C"): la gestión, incluyendo la liberación, es responsabilidad de le programador. Eficiente pero potencialmente "unsafe" o con "leaks"
  - Gestionada x Garbage Collector ("a la Java"). Hay "aliasing" pero la liberación es por medio del GC. Menos eficiente pero "safe", y no necesita gestión de eliminación por parte de le programador
  - Mecanismo de Ownership ("a la Rust"): el concepto de ownership y salida de scope implica liberación de memoria. Requiere seguir ciertas disciplinas de programación (pero menos carga para el programador). Es más eficiente (que Java) y safe (que C y Java)
- Vamos a seguir el paradigma de gestión de Java

# Memora Dinámica: primitivas que vamos a usar

- `New(Tipo)` crea un nuevo lugar en el Heap de tipo `Tipo` (para nosotros tipicamente una estructura ó un arreglo de algo) y devuleve una referencia al mismo para que esta se guarde en algún otro lugar (campo, slot, parámetro de salida, variable auxiliar, etc.). `NewArray<>(length)` es un caso particular
- Hay una constante de tipo referencia distinguida `null` que no denota ningún elemento del heap
- Si un referencia es distinta a `null` es válido acceder al elemento y manipularlo según el tipo correspondiente
- Así, por ejemplo, `r.f` / `r[i]`, denotan un campo/slot del elemento denotado por `r`. Estos se pueden leer ó asignar (con `:=`)
- No hay primitivas para devolver memoria. Esto ocurre cuando esa memoria no es referenciada desde ningún lugar y ya no se la puede acceder. Es tarea del Garbage Collector

# Definición de Tipos

Ya dijimos que vamos a tener estructuras y arreglos. Así vamos a poder introducir tipos de representación.
Por ejemplo:

`EstructuraConDatos = <dato:`$\mathbb{N}$`, arr:Array<DatoEstructurado>>`

`DatoEstructurado = <datointerno: Float, contador:` $\mathbb{N}$`>`

Y, lo que es buenísimo: ¡Vale hacer definiciones recursivas de tipos de representación!

Ejemplo:

`Nodo = Struct <dato:` $\mathbb{N}$`, pxmo: Nodo >`

# Ejemplo

```
Nodo = Struct <dato: ℕ, pxmo: Nodo >;
VAR valor: ℕ;
VAR aux: Nodo;
VAR sll: Nodo;
valor := 3; %hasta acá sólo se usó el stack
aux := New(Nodo); %Una referencia a un nuevo elemento de tipo
Nodo en el Heap
aux.dato := valor;
aux.pxmo := aux;
sll := aux; % en este punto hay infinitos alias al único
elemento de la Heap
sll:= null
```

# Memoria Dinámica: Aspectos del Modelado Formal

Heap se puede entender una función de referencias a tuplas y arreglos. La fórmula `r` $\neq$ `null` predica la no nulidad de la referencia `r` en el heap corriente mientras que `r.dato = d` formula que la estructura referenciada por `r` en el heap corriente tiene el valor `d` en el campo `dato`.

# Memoría dinámica: Importancia del razonamiento

Notas informativas
Particularmente, en el software de infraestructura (ej., drivers, sistemas operativos, hypervisores, middleware, frameworks, etc., etc.) errores lógicos en la manipulación de memoria dinámica constituyen una categoría importante de bugs que comprometen la seguridad, performance y funcionalidad de las componentes (junto con otros bugs como la ejecución incorrecta de system calls o bugs de concurrencia)

Hay propuestas basadas, por ejemplo, en conceptos como `separation logic` (ej. `Infer` de Meta) que automatizan el razonamiento sobre memoria dinámica. El operador más importante de esa lógica es el de separación (ej. `P*Q`) para predicar que la heap se divide en dos o más partes en donde valen ciertos predicados (en este caso una parte en dónde vale `P` y otra disjunta en dónde vale `Q`). Lo que vamos a ver es similar en espíritu pero usa la lógica de primer orden que estamos usando en la materia (más expresiva pero menos automatizable)

# Representación sobre listas encadenadas

Primero, un tipo que vamos a necesitar:

`Nodo = Struct <dato:` $\mathbb{N}$`, pxmo: Nodo >`

## SetUsingLinkedList (a.k.a. SLL)

`Módulo SLL implementa Set {`
`head: Nodo`

El módulo introduce un tipo SLL que es un struct que tiene un campo `head`. Además, un módulo introduce la implementación de las operaciones del TAD y documenta el invariante de representación y la función de abstracción

# Invariante de Representación

Definición Auxiliar de "shape":

`list?  (l,x)` $=_{def}$ `(l=<>⇔x=null)` $\wedge_l$ `(x≠null` $\Rightarrow_l$
`(x.dato=head(l)` $\wedge$ `list(tail(l),x.pxmo)))`

Notar que la recursión termina aunque tenga un ciclo la estuctura porque consume la secuencia

## Invariante de Representación de SLL

```
pred InvRep(sll:  SLL)
{sll≠null ∧l ∃l:Seq<ℕ>.list?(l,sll.head) }
```

Esto es algo que en la clase de BSBV no dijimos: (`bsbv≠null`)

A decir verdad en un lenguaje OO se puede asumir como precondición ya que el runtime va a quejarse si se invoca un método sobre un `null`. Dicho esto, sí nos deberíamos cuidar de no devolver un puntero a `null`. Notar que en BSBV indirectamente lo pedíamos en el `def(res.arr)` (que el array no sea un puntero nulo y que `res` tampoco)

# Función de Abstracción

Notar:
$\forall$`l,l':Seq<`$\mathbb{N}$`> list(l,x)`$\wedge$`list(l',x)`$\Rightarrow$`l=l'`

Por lo tanto, cuando valga a partir de un nodo `x` que existe un lista `l` tal que vale `list?(l,x)`, está definida la lista subyacente de la estructura (`list(x)`)

## Función de abstracción de SLL

$\alpha$`(sll:SLL):Set`
$\alpha$`(sll).set = SetOf(list(sll.head))`

Definida en términos de los observadores del TAD
dónde `SetOf` de una lista devuelve su conjunto subyacente

# Especificación Operaciones sobre la Estructura

Vamos a ver qué contratos debería respetar el implementador de estas operaciones si pretende cumplir con el contrato abstracto del TAD

**`Proc EmptySet (): SLL`**

`asegura InvRep(res)` $\wedge$ $\alpha$`(res)=`$\emptyset$

$\Leftarrow$ (x def. de $\alpha$ y InvRep...)

`asegura res`$\neq$`null` $\wedge_l$ `res.head=null`

Recordar que por más que en este nivel muestre la Pre y Post del código de implementación, el código que usa este módulo opera tratándolo como un Conjunto. No accede a la representación. Esto se conoce como **Information Hidding**

# Código Operaciones sobre la Estructura

```
Proc EmptySet ():  SLL

Var Aux:SLL;
Aux:=New(SLL);
Aux.head:=null;
RETURN Aux
```

asegura res≠null $\wedge_l$ res.head=null

Una observación: estamos asumiendo de ahora en más que hay espacio en memoria cada vez que hago New!

# Especificación Operaciones sobre la Estructura

**Proc Add (inout sll: SLL ,in d: $\mathbb{N}$)**

requiere InvRep(sll) $\wedge_l$ elems=$\alpha$(sll).set

$\Leftrightarrow$

requiere sll$\neq$null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$>.list?(l,sll.head) $\wedge_l$ $l_0$=list(sll.head) $\wedge$ elems=SetOf($l_0$)

asegura InvRep(sll) $\wedge_l$ $\alpha$(sll).set=elems$\cup${d}

$\Leftarrow$ uso defs y fortalezco con decisión de diseño (predico que el elemento queda en el frente de la lista)

asegura sll$\neq$null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$> list?(l,sll.head) $\wedge_l$ list(sll.head)=<d>++$l_0$

# Código Operaciones sobre la Estructura

**Proc Add (inout sll: SLL ,in d: $\mathbb{N}$)**

requiere sll≠null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$>.list?(l,sll.head) $\wedge_l$
$l_0$=list(sll.head) $\wedge$ elems=SetOf($l_0$)

```
VAR aux:Nodo
aux := New(Nodo)
aux.dato := d
aux.pxmo := sll.head
sll.head := aux
RETURN
```

asegura sll≠null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$> list?(l,sll.head) $\wedge_l$
list(sll.head)=<d>++$l_0$

# Especificación de Operaciones sobre la Estructura

**Proc In? (in sll: SLL:, in d: $\mathbb{N}$): Bool**

requiere InvRep(sll) $\wedge_l$ elems=$\alpha$(sll).set

$\Rightarrow$

requiere sll$\neq$null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$>.list?(l,sll.head) $\wedge_l$
$l_0$=list(sll.head) $\wedge$ elems=SetOf($l_0$)

asegura InvRep(sll) $\wedge_l$ res=d$\in$elems

$\Leftarrow$

asegura sll$\neq$null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$>list?(l,sll.head) $\wedge_l$
list(sll.head)=$l_0$ $\wedge_l$ (res=True$\Leftrightarrow$d$\in l_0$)

# Código de Operaciones sobre la Estructura

**Proc In? (in sll: SLL:, in d: $\mathbb{N}$): Bool**

requiere sll≠null $\wedge_l$ ∃l:Seq<$\mathbb{N}$>.list?(l,sll.head) $\wedge_l$
$l_0$=list(sll.head) $\wedge$ elems=SetOf($l_0$)

```
VAR Res: Bool; actual: Nodo
res:= False;
actual:= sll.head
WHILE actual != null ∧ res != True
 IF actual.dato = d THEN res := True ENDIF
 actual := actual.pxmo
```

Inv.ciclo: sll≠null $\wedge_l$ (∃l:Seq<$\mathbb{N}$>list?(l,sll.head)) $\wedge_l$
list(sll.head)=$l_0$ $\wedge$ (∃l:Seq<$\mathbb{N}$> list?(l,actual)) $\wedge_l$ ∃l':Seq<$\mathbb{N}$> (
$l_0$=l'++list(actual) $\wedge$ (res=True⇔d∈l')) ....Variante?

```
ENDWHILE
RETURN res
```

asegura sll≠null $\wedge_l$ ∃l:Seq<$\mathbb{N}$>list?(l,sll.head) $\wedge_l$
list(sll.head)=$l_0$ $\wedge_l$ (res=True⇔d∈$l_0$)

# Especificación Operaciones sobre la Estructura

**Proc Delete (inout sll: SLL ,in d: $\mathbb{N}$)**

requiere InvRep(sll) $\wedge_l$ elems=$\alpha$(sll).set

$\Leftrightarrow$

requiere sll$\neq$null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$>.list?(l,sll.head) $\wedge_l$
$l_0$=list(sll.head) $\wedge$ elems=SetOf($l_0$)

asegura InvRep(sll) $\wedge_l$ $\alpha$(sll).set=elems\{d}

$\Leftarrow$ (x def. de $\alpha$ e InvRep)

asegura sll$\neq$null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$>list?(l,sll.head) $\wedge_l$
list(sll.head)=Remove($l_0$,d)

# Código de Operaciones sobre la Estructura

**Proc Delete (inout sll: SLL ,in d: $\mathbb{N}$)**

requiere sll≠null $\wedge_l$ ∃l:Seq<$\mathbb{N}$>.list?(l,sll.head) $\wedge_l$ $l_0$=list(sll.head) $\wedge$ elems=SetOf($l_0$)

```
WHILE (sll.head != null && sll.head.dato = d) { sll.head :=
sll.head.pxmo;}
actual := sll.head;
WHILE (actual != null & & actual.pxmo != null)
  IF (actual.pxmo.dato = d) THEN
   actual.pxmo := actual.pxmo.pxmo;
  ELSE actual := actual.pxmo;
  ENDIF
ENDWHILE
RETURN
```

asegura sll≠null $\wedge_l$ ∃l:Seq<$\mathbb{N}$>list?(l,sll.head) $\wedge_l$ list(sll.head)=Remove($l_0$,d)

# Código de Operaciones sobre la Estructura (con invariantes) (pensar Variantes)

```
VAR actual:nodo;
WHILE (sll.head != null && sll.head.dato = d) {sll.head:=sll.head.pxmo;
```

sll≠null $\wedge_l$ ∃l:Seq<ℕ>list?(l,sll.head) $\wedge_d$
remove(list(sll.head),d)=remove($l_0$,d)

```
}
actual := sll.head;
WHILE (actual != null & & actual.pxmo != null)
  IF (actual.pxmo.dato = d) THEN
   actual.pxmo = actual.pxmo.pxmo;
  ELSE actual = actual.pxmo;
  ENDIF
```

sll≠null $\wedge_l$ ∃l:Seq<ℕ>list?(l,sll.head) $\wedge_l$
∃l:Seq<ℕ>list?(l,actual) $\wedge_l$ (∃l':Seq<ℕ>.$l_0$=l'++list(actual) $\wedge$
list(sll.head)=Remove(l',d)++list(actual))

```
ENDWHILE
RETURN
```