Diseño de TADs

Algoritmos y Estructuras de Datos

¿Qué es un TAD?

- Un TAD (tipo abstracto de datos) es una abstracción que describe una parte de un problema.
- Describe el **qué** y no el **cómo**.
- Tiene estado.
- Se manipula a través de operaciones, que describimos mediante un lenguaje de especificación (lógica) con pre y postcondiciones.

Diseño de TADs

- Un diseño de un TAD es una estructura de datos y una serie de algoritmos (en algún lenguaje de programación, real o simplificado) que nos indica cómo se representa y se codifica una implementación del TAD.
  - Tendremos que elegir una estructurade representación con tipos de datos
  - Tendremos que escribir algoritmospara todas las operaciones
  - Los algoritmos deberán respetar la especificación del TAD
- Miren lo que está en azulen esta diapo ;-)

Múltiples Diseños de un TAD

- ¡Puede haber muchos diseños para un TAD!
  - Porque dos personas lo pensaron de diferentes maneras
  - Porque hay requerimientos de eficiencia (memoria, tiempo de ejecución: complejidad)
  - Perchè mi piace
- Mientras respeten la especificación, quien las use podrá elegir uno u otro diseño sin cambiar sus programas (modularidad).

Ocultando Información

- Ventajas del ocultamiento, abstracción y encapsulamiento:
  - La implementación se puede cambiar sin afectar su uso.
  - Ayuda a modularizar.
  - Facilita la comprensión
  - Favorece el reuso.
  - Los módulos son más fáciles de entender
  - y programar.
  - El sistema es más resistente a cambios.

Ocultando Información

- Abstracción: "Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details." [Ghezzi et al, 1991]
- Information hiding: "The [...] decomposition was made using 'information hiding' [...] as a criterion. [...] Every module [...] is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little aspossible about its inner workings." [Parnas, 1972b]
- "[...] the purpose of hiding is to make inaccessible certain details that should not affect other parts of a system." [Ross et al, 1975]
- Encapsulamiento: "[...] A consumer has full visibility to the procedures offered by an object, and no visibility to its data. From a consumer's point of view, an object is a seamless capsule that offers a number of services, with no visibility as to how these services are implemented [...] The technical term for this is encapsulation." [Cox, 1986]

Diseño de un TAD

```
TAD Punto {
      obs x: ℝ
      obs y: ℝ
}
```

```
Modulo PuntoImpl implementa Punto {
      rho: float
      theta: float
}
```

Notar que especificamos con cartesianas y diseñamos con polares

¿Podríamos hacer al revés?

- En la especificación nos referíamos a los valores del tipo a partir de los observadores.
- En la implementación tenemos que definir los valores explícitamente a partir de una estructura.
- Los tipos de las variables de la estructura son tipos de implementación:
  - int, float, char, ...
  - tupla / struct (tupla con nombres)
  - array$< T >$ ¡¡TAMAÑO FIJO!! (no son seq ni conj)
  - Módulos de otros TADs
  - ¡y veremos muchísimas más!

Invariante de Representación

```
Modulo PuntoImpl implementa Punto {
      rho: float
      theta: float
}
```

- ¿Cómo va a funcionar nuestro módulo en el almacenamiento de los ángulos?
  - Normalizado (entre 0 y 2π entre $-\pi$ y π)
  - Desnormalizado (cualquier valor real)
- Podemos elegir cualquier, pero tenemos que tener consistencia

No todos los posibles valores de las variables de estado representan un estado de Punto válido.

Tenemos restricciones

Invariante de Representación

Invariante de representación: es un predicado que nos indica qué conjuntos de valores son instancias válidas de la implementación.

- Se tiene que cumplir siempre al entrar y al salir de todas las operaciones
- Generalmente lo denotamos como InvRep
- Para cualquier `proc x()` del módulo, se tiene que poder verificar la siguiente tripla de Hoare:

  $\{InvRep(p)\} \quad procx(p, ...) \quad \{InvRep(p)\}$
- Además de las que involucran la pre y la post del proc
- Se escribe en lógica, haciendo referencia a la estructura de implementación

```
Modulo PuntoImpl implementa Punto {
      rho: float
      theta: float
      pred invRep(p': PuntoImpl)
            {−π <= p'.theta < π} }
```

Función de Abstracción

```
TAD Punto {
      obs x: ℝ
      obs y: ℝ
}
```

```
Modulo PuntoImpl implementa Punto {
      rho: float
      theta: float
}
```

¿Cómo relacionamos el TAD con el módulo?

- **Función de abstracción:** nos va a indicar, dada una instancia de implementación, a qué instancia del TAD corresponde/representa, de qué instancia del TAD "es su abstracción"
- predAbs toma como parámetro una instancia del módulo y del TAD y evalua si son iguales
- Para definirla, se puede suponer que vale el invariante de representación

Función de Abstracción

A pesar de que se llama Función de Abstracción, lo vamos a escribir en formato de predicado:

```
Modulo PuntoImpl implementa Punto {
      rho: float
      theta: float

      pred invRep(p: PuntoImpl)
            {−π <= p.theta < π}

      pred predAbs(p: PuntoImpl, p':Punto)
            {p'.x = p.rho ∗ cos(p.theta)∧
             p'.y = p.rho ∗ sin(p.theta)}
}
```

Ahora sólo quedaría escribir los algoritmos.

Pero volvamos a los ejemplos de conjuntos que son más interesantes

Conjunto Acotado sobre Array de Bools

```
TAD ConjAcotado<ℕ> {
  obs set: conj<ℕ>
  obs cota: ℕ

  proc conjVacío(in cota: ℕ):
ConjAcotado<ℕ>
    asegura{res.set = {} ∧ res.cota = cota}

  proc pertenece(in c: ConjAcotado<ℕ>,
in e: ℕ): bool
    asegura{res = true ↔ e ∈ c.set}

  proc agregar(inout c: ConjAcotado<ℕ>,
in e: ℕ)
    requiere{c = C0 ∧ e ≤ c.cota}
    asegura{c.set = C0.set ∪ ⟨e⟩}
}
```

```
Modulo ConjActadoSobreBitArray<Bool> implementa
ConjAcotado<ℕ> {
  elems: Array<Bool>

  proc conjVacío(in cota: Int): CASBA<Int>
    Creo un Array de cota elementos, todos en False

  proc pertenece(in c: CASBA, in e: Int): bool
    Me fijo el estado de la posición e.

  proc agregar(inout c: CASBA, in e: Int)
    Me fijo el estado de la posición e.
    Si no lo está (False), lo cambio.

}
```

- ¿Qué restricciones tenemos para esta representación de conjunto acotado?

  → Tenemos que hacer el invRep
- ¿Cómo se relaciona la implementación con el TAD?

  → Tenemos que hacer el predAbs

Conjunto Acotado sobre Array de Bools

Invariante de Representación:

- No tenemos restricciones importantes.
- Solo que las posiciones del Array estén definidas
- Más adelante vamos a dejar de pedir esto

```
pred invRep(c: CASBA){
          (∀i : ℤ)(0 ≤ i < length(c.elems) →L def(c.elems[i]))}
```

Función de Abstracción:

- Tenemos que relacionar la variable *elems* del módulo con los observadores del TAD

```
pred abs(c:CASBA, c':ConjuntoAcotado)
          {c'.cota = length(c.elems)-1 ∧
          (∀n : ℕ)( n∈c'.set ⇔
            (n≤length(c.elems)-1 ∧L c.elems[n] = True) }
```

Y con esto ya podemos empezar a programar los procs

Conjunto Acotado sobre Array de Bools

```
TAD ConjAcotado<ℕ> {
  obs set: conj<ℕ>
  obs cota: ℕ

  proc conjVacío(in cota: ℕ):
ConjAcotado<ℕ>
    asegura{res.set = {} ∧ res.cota = cota}

  proc pertenece(in c: ConjAcotado<ℕ>,
in e: ℕ): bool
    asegura{res = true ↔ e ∈ c.set}

  proc agregar(inout c: ConjAcotado<ℕ>,
in e: ℕ)
    requiere{c = C0 ∧ e ≤ c.cota}
    asegura{c.set = C0.set ∪ ⟨e⟩}
}
```

```
Modulo ConjActadoSobreBitArray<Bool> implementa
ConjAcotado<ℕ> {
  elems: Array<Bool>

  pred invRep(c: CASBA){
    (∀i : ℤ)(0 ≤ i < length(c.elems) →L def(c.elems[i]))}
  pred abs(c:CASBA, c':ConjuntoAcotado)
    {c'.cota = length(c.elems)-1 ∧
    (∀n : ℕ)( n∈ c'.set ⇔
      (n≤length(c.elems)-1 ∧L c.elems[n] = true) }
  proc conjVacío(in cota: Int): CASBA
    res.elems = NewArray[cota+1][False];
    return res

  proc pertenece(in c: CASBA, in e: Int): bool
    if e<length(c.elems):
      res := c.elems[e]
    else:
      res := False
    return res

  proc agregar(inout c: CASBA, in e: Int)
    c.elems[e] := true
    return

}
```

- ¿Cómo impactan los requiere y asegura del TAD en los códigos del TAD?
- Cada proc del módulo tiene que cumplir con los requiere y asegura del TAD
- Y el invRep
- Para eso hay que "traducirlos" al "lenguaje" del módulo
- Se hace via la función de abstracción

Conjunto Acotado sobre Array de Bools

```
proc conjVacío(in cota: ℕ):
ConjAcotado<ℕ>
  asegura{res.set = {} ∧ res.cota = cota}
```

```
pred invRep(c: CASBA){
    (∀i : ℤ)(0 ≤ i < length(c.elems) →L def(c.elems[i]))}
pred abs(c:CASBA, c':ConjuntoAcotado)
    {c'.cota = length(c.elems)-1 ∧
    (∀n : ℕ)( n∈ c'.set ⇔
      (n≤length(c.elems)-1 ∧L c.elems[n] = true) }
```

```
proc conjVacío(in cota: Int): CASBA
      requiere {True}
      res.elems = NewArray[cota+1][False];
      return res
      asegura {length(res)= cota+1 ∧
      (∀i : ℤ)(0≤i<length(res) →L res[i]=False)}
```

Notar que es obvio que True ⇒ Wp(asegura, código) por semántica axiomática asumida del New de Array (crea un arreglo de length establecida y lo setea todo en el valor establecido). Eso sí, esto es caro en términos de tiempo: O(cota).

Conjunto Acotado sobre Array de Bools

```
proc pertenece(in c: ConjAcotado<ℕ>,
in e: ℕ): bool
  asegura{res = true ↔ e ∈ c.set}
```

```
pred invRep(c: CASBA){
    (∀i : ℤ)(0 ≤ i < length(c.elems) →L def(c.elems[i]))}
pred abs(c:CASBA, c':ConjuntoAcotado)
    {c'.cota = length(c.elems)-1 ∧
    (∀n : ℕ)( n∈ c'.set ⇔
      (n≤length(c.elems)-1 ∧L c.elems[n] = true) }
```

```
proc pertenece(in c: CASBA, in e: Int): bool
      requiere { invRep(c) }
      if e<length(c.elems):
          res := c.elems[e]
      else:
          res := False
      return res
```

Tenemos que pasar el asegura del TAD al Módulo: (res = True ↔ (e∈ c'.set))
asegura {invRep(c) ∧L (res = True ↔ (e<length(c.elems)
∧L c.elems[e]=True))}

Siempre vamos a utilizar el Abs para sacarnos todas las apariciones del TAD (cén este caso) y reemplazarlas por lo que corresponda del módulo

Conjunto Acotado sobre Array de Bools

```
proc agregar(inout c: ConjAcotado<ℕ>,
in e: ℕ)
  requiere{c = C0 ∧ e ≤ c.cota}
  asegura{c.set = C0.set ∪ ⟨e⟩}
```

```
pred invRep(c: CASBA){
    (∀i : ℤ)(0 ≤ i < length(c.elems) →L def(c.elems[i]))}
pred abs(c:CASBA, c':ConjuntoAcotado)
    {c'.cota = length(c.elems)-1 ∧
    (∀n : ℕ)( n∈ c'.set ⇔
      (n≤length(c.elems)-1 ∧L c.elems[n] = true) }
```

```
proc agregar(inout c: CASBA, in e: Int)
      requiere { InvRep(c) ∧L e<length(bsbv.elems) ∧ c=C0 }
      c[e] := true
      return
      asegura {InvRep(c) ∧L (∀i : ℤ)(0 ≤ i < length(c.elems) →L
       (i ≠ e ∧ c.elems[i] = C0.elems[i]) ∨ (i = e ∧ c.elems = True)}
```

- Asumimos que los parámetros *in* no son modificados durante la ejecución
- Para asegurarlo formalmente habría que inicializar metavariables en el requiere y usarlas en el asegura

Algunas observaciones

En resumen:

- Escribir el invRep permite que nuestra estructura de representación no se rompa
- El invRep se tiene que cumplor siempre antes y despues de cada proc
- Escribir el predAbs nos permite conectar con el TAD
- En base al invRep, el predAbs, el TAD y nuestras implementaciones vamos a tener **requiere** y **asegura**
- Los **asegura** de los procs del módulo pueden ser más fuertes que los de la especificación.
- Van a reflejar las decisiones de diseño
- Por esto es importante no sobreespecificar!!

Conjunto sobre Listas enlazadas (memoria dinámica)

¿Y si volvemos al TAD conjunto (no acotado?

```
TAD Conjunto<T> {
      obs elems: conj<T>

      proc conjVacío(): Conjunto<T>
            asegura {res.elems = ⟨⟩}

      proc pertenece(in c: Conjunto<T>, in e: T): bool
            asegura {res = true ↔ e ∈ c.elems}

      proc agregar(inout c: Conjunto<T>, in e: T)
            requiere {c = C0}
            asegura {c.elems = C0.elems ∪ ⟨e⟩}
}
```

Recordatorio: Acá teniamos el problema de la memoria estática.

¡¡Usemos Memoria Dinámica!!

Conjunto sobre Listas enlazadas (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

Conjunto sobre Listas enlazadas (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)

Conjunto sobre Listas enlazadas (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

Conjunto sobre Listas enlazadas (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

Conjunto sobre Listas enlazadas (memoria 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

Conjunto sobre Listas enlazadas (memoria dinámica)

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 >`

Conjunto sobre Listas enlazadas (memoria dinámica)

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
```


Conjunto sobre Listas enlazadas (memoria dinámica)

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`.

Conjunto sobre Listas enlazadas (memoria dinámica)

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)

Conjunto sobre Listas enlazadas (memoria dinámica)

Representación sobre listas encadenadas

Primero, un tipo que vamos a necesitar:

`Nodo = Struct <dato: ℕ, pxmo: Nodo >`

SetUsingLinkedList (a.k.a. SLL)

```
Módulo SLL implementa Conjunto {
     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

Conjunto sobre Listas enlazadas (memoria dinámica)

Invariante de Representación

Definición Auxiliar de "shape":
`list? (l:seq< T >,x:Nodo) { (l=<> ⇔ x=null) ∧L (x ≠ null ⇒L (x.dato=head(l) ∧ 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 ∧I (∃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)

Conjunto sobre Listas enlazadas (memoria dinámica)

Función de Abstracción

Notar:
`(∀l,l':Seq< ℕ >)( list(l,x)∧list(l',x)⇒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

```
pred abs(sll:SLL,c':Set)
     {c'.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)` $\wedge$ α`(res)=`$\emptyset$ $\}$

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

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

`asegura sll`$\neq$`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=α(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$\neq$null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$>.list?(l,sll.head) $\wedge_l$
l_0=list(sll.head) $\wedge$ elems=SetOf(l_0)
VAR Res: Bool; actual: Nodo
res:= False;
actual:= sll.head
WHILE actual != null $\wedge$ res != True
 IF actual.dato = d THEN res := True ENDIF
 actual := actual.pxmo

Inv.ciclo: sll$\neq$null $\wedge_l$ ($\exists$l:Seq<$\mathbb{N}$>list?(l,sll.head)) $\wedge_l$
list(sll.head)=l_0 $\wedge$ ($\exists$l:Seq<$\mathbb{N}$> list?(l,actual)) $\wedge_l$
$\exists$l':Seq<$\mathbb{N}$> (l_0=l'++list(actual) $\wedge$ (res=True$\Leftrightarrow$d$\in$l'))
....Variante?
ENDWHILE
RETURN res

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$\in l_0$)

Especificación Operaciones sobre la Estructura

Proc Delete (inout sll: SLL ,in d: $\mathbb{N}$)
requiere InvRep(sll) $\wedge_l$ elems=α(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) $\wedge_l$ α(sll).set=elems\{d}
$\Leftarrow$ (x def. de α 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$\neq$null $\wedge_l$ $\exists$l:Seq<$\mathbb{N}$>.list?(l,sll.head) $\wedge_l$
l_0=list(sll.head) $\wedge$ elems=SetOf(l_0)
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$\neq$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 (con invariantes) (pensar Variantes)

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

sll≠null ∧I ∃l:Seq<ℕ>list?(l,sll.head) ∧d
remove(list(sll.head),d)=remove(l0,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 ∧I ∃l:Seq<ℕ>list?(l,sll.head) ∧I
∃l:Seq<ℕ>list?(l,actual) ∧I
(∃l':Seq<ℕ>.l0=l'++list(actual) ∧
list(sll.head)=Remove(l',d)++list(actual))
ENDWHILE
RETURN
```


Bibliografía

1 Berard, E. V. .Abstraction, Encapsulation, and Information Hiding".

2 Parnas, D.L. .On the Criteria To Be Used in Decomposing Systems Into Modules", 1972.

3 Ghezzi, C., Jazayeri, M., Mandrioli, D. "Fundamentals of Software Engineering", 1991.

4 Ross, D.T., Goodenough, J.B., Irvine, C.A. "Software Engineering: Process, Principles, and Goals", 1975.

5 Cox, B.J. .Object-Oriented Programming: An Evolutionary Approach", 1986.