Introducción a la Computación (Matemática)

Primer Cuatrimestre de 2019

Tipos Abstractos de Datos

Problema: Agenda

Queremos programar una agenda de contactos.
De cada persona nos interesa guardar:

- Nombre
- Teléfono
- Dirección

¿Cómo representamos los datos de las personas?

- Nombres: lista de strings.
- Teléfonos: lista de strings.
- Direcciones: lista de strings.

Tales que la *i*-ésima posición de los 3 listas correspondan a la misma persona. Esta representación *funciona* (cumple el objetivo), pero tiene serios problemas...

Problema: Agenda

Mucho mejor sería contar con un tipo Persona, que **encapsule** todos los datos relevantes a una persona para nuestra agenda.

Tipo Abstracto de Datos (TAD) Persona
Operaciones:

- CrearPersona(nom, tel, dir) $\rightarrow$ Persona: Crea una persona nueva con el nombre, teléfono y dirección especificados.
- Nombre(p) $\rightarrow$ String: Devuelve el nombre de la persona p.
- Teléfono(p) $\rightarrow$ String: Devuelve el teléfono de la persona p.
- Dirección(p) $\rightarrow$ String: Devuelve la dirección de la persona p.

Así, podemos representar la agenda con una **lista de Personas**.

¿Cómo se implementa el TAD? Como usuarios, **no nos importa**.

Problema: Contar días entre 2 fechas

```
# Dice si el año a es bisiesto.
def bisiesto(a):
  return a%4==0 and (a%100!=0 or a%400==0)

# Devuelve la cantidad de dias del mes m, año a.
def diasEnMes(m, a):
  r=0
  if m==1 or m==3 or m==5 or m==7 or \
     m==8 or m==10 or m==12:
    r = 31
  if m==4 or m==6 or m==9 or m==11:
    r = 30
  if m==2:
    if bisiesto(a):
      r = 29
    else:
      r = 28
  return r

# Cuenta los dias entre los meses m1 y m2
# inclusive, en el año a.
def diasEntreMeses(m1, m2, a):
  r = 0
  m = m1
  while m <= m2:
    r = r + diasEnMes(m, a)
    m = m + 1
  return r
```

```
# Cuenta los dias entre el d1/m1/a1 y el d2/m2/a2.
def contarDias(d1, m1, a1, d2, m2, a2):
  r = 0

  if a1 == a2 and m1 == m2:
    r = d2 - d1

  if a1 == a2 and m1 < m2:
    r = diasEnMes(m1, a1) - d1
    r = r + diasEntreMeses(m1+1, m2-1, a1)
    r = r + d2

  if a1 < a2:
    r = diasEnMes(m1, a1) - d1
    r = r + diasEntreMeses(m1+1, 12, a1)
    a = a1 + 1
    while a < a2:
      r = r + 365
      if bisiesto(a):
        r = r + 1
      a = a + 1
    r = r + diasEntreMeses(1, m2-1, a2)
    r = r + d2

  return r
```

Problema: Contar días entre 2 fechas

Imaginemos que contamos con un tipo Fecha que nos ofrece estas operaciones (entre otras):

- CrearFecha$(d, m, a) \rightarrow$ *Fecha*
- FechaSiguiente$(f) \rightarrow$ *Fecha*: Devuelve la fecha siguiente a la fecha dada. (Ej: al 31/12/1999 le sigue el 1/1/2000.)
- Menor$(f_1, f_2) \rightarrow \mathbb{B}$: Devuelve TRUE si la fecha f_1 es anterior que la fecha f_2, y FALSE en caso contrario.

Usando esto, un algoritmo para contar días podría ser:

ContarDías$(f_1, f_2) \rightarrow \mathbb{Z}$:

```
RV ← 0
while (Menor(f1, f2)) {
        RV ← RV + 1
        f1 ← FechaSiguiente(f1)
}
```

Modularidad y encapsulamiento

La clave está en **encapsular** los datos y sus operaciones.

Al programar, definimos funciones (ej: Primo(n)) para generar código más simple y claro (código *modular*).
Encapsulamos un algoritmo para poder reusarlo muchas veces.

Ahora generalizamos este concepto, y encapsulamos datos (ej: personas, fechas) y sus operaciones (ej: FechaSiguiente(f)) en Tipos Abstractos de Datos (TAD).

Para **usar** un TAD, el programador sólo necesita conocer el nombre del TAD y la especificación de sus operaciones (y quizá sus órdenes de complejidad).

Un TAD puede estar implementado de muchas formas distintas.
Esto debe ser **transparente** para el usuario.

Partes de un Tipo Abstracto de Datos

- Parte pública: Disponible para el usuario externo.
  - Nombre y tipos paramétricos (ej: Fecha, Lista(ELEM)).
  - Operaciones, sus especificaciones y posiblemente sus órdenes de complejidad temporal.

- Parte privada: Sólo accesible desde dentro del TAD. ¡El usuario externo **nunca** debe ver ni meterse en esto!
  - Próxima clase...

TAD Fecha

Operaciones públicas (para f, f_1, f_2 : Fecha; $d, m, a : \mathbb{Z}$):

- CrearFecha$(d, m, a) \rightarrow$ Fecha
- Día$(f) \rightarrow \mathbb{Z}$
- Mes$(f) \rightarrow \mathbb{Z}$
- Año$(f) \rightarrow \mathbb{Z}$
- Menor$(f_1, f_2) \rightarrow \mathbb{B}$
- FechaSiguiente$(f) \rightarrow$ Fecha

Como usuarios del TAD, podemos programar algo como:

ContarDías$(f_1, f_2) \rightarrow \mathbb{Z}$:

```
RV ← 0
while (Menor(f1, f2)) {
    RV ← RV + 1
    f1 ← FechaSiguiente(f1)
}
```

¿Cómo se implementa? Como usuarios, **no nos importa**.

Supongamos que trabajamos con un lenguaje de programación que no tiene arreglos, ni listas, ni nada parecido.

Vamos a crear nuestro propio tipo Lista(ELEM).

Primero definamos qué operaciones querríamos que tenga.

Después veremos una posible implementación.

TAD Lista(ELEM)

Operaciones:

- CrearLista() $\rightarrow$ Lista(ELEM): Crea una lista vacía.
- L.Agregar(x): Inserta el elemento x al final de L.
- L.Longitud() $\rightarrow \mathbb{Z}$: Devuelve la cantidad de elementos de L.
- L.Iésimo(i) $\rightarrow$ *ELEM*: Devuelve el i-ésimo elemento de L.
  Precondición: $0 \leq i < L$.Longitud().
- L.Cantidad(x) $\rightarrow \mathbb{Z}$: Devuelve la cantidad de veces que aparece el elemento x en L.

donde L : Lista(ELEM); $i : \mathbb{Z}$; x : ELEM (entero, char, etc.).

TAD Lista(ELEM)

Operaciones (cont.):

- L.Insertar(i, x): Inserta el elemento x en la posición i de L, pasando los elementos de la posición i y siguientes a la posición inmediata superior. Pre: $0 \leq i \leq L$.Longitud().
  (Si $L = a_0, .., a_{n-1}$, se convierte en $L = a_0, .., a_{i-1}, x, a_i, .., a_{n-1}$.)
- L.Indice$(x) \rightarrow \mathbb{Z}$: Devuelve el índice $(0 .. L.\text{Longitud}() - 1)$ de la primera aparición de x en L. Pre: L.Cantidad$(x) > 0$.
- L.Borrarlésimo(i): Borra el i-ésimo elemento de L
  Precondición: $0 \leq i < L$.Longitud().
  (Si $L = a_0, .., a_{n-1}$, se convierte en $L = a_0, .., a_{i-1}, a_{i+1}, .., a_{n-1}$.)
- ...

donde L : Lista(ELEM); $i : \mathbb{Z}$; x : ELEM (entero, char, etc.).

TAD Lista(ELEM) - Ejemplo de uso

Encabezado: $Máximo : L \in Lista(\mathbb{Z}) \to \mathbb{Z}$

Precondición: $\{L = L_0 \wedge L.Longitud() > 0\}$

Poscondición: $\{(\forall i)\ 0 \leq i < L_0.Longitud() \Rightarrow L_0.Iésimo(i) \leq RV$
$\wedge\ \ L_0.Cantidad(RV) > 0\}$

$m \leftarrow L.Iésimo(0)$
$i \leftarrow 1$
while $(i < L.Longitud())$ {
    if $(L.Iésimo(i) > m)$ {
        $m \leftarrow L.Iésimo(i)$
    }
    $i \leftarrow i + 1$
}
$RV \leftarrow m$

TADs: Uso vs. implementación

Hasta ahora, nos enfocamos en el **uso** de los TADs, que podemos haber creado nosotros o alguien más:

- TAD Persona
- TAD Fecha
- TAD Lista(ELEM)

Desde el punto de vista del usuario, los detalles de implementación (la parte privada) son irrelevantes.

Ahora vamos a ver cómo **implementar** estos TADs.

TAD Fecha - Posible implementación

Primero elegimos una estructura de representación para el TAD Fecha (que es **privada**).

$$\text{Fecha} == \langle \mathit{día}\colon \mathbb{Z},\ \mathit{mes}\colon \mathbb{Z},\ \mathit{año}\colon \mathbb{Z} \rangle$$

$\langle ... \rangle$ define una tupla.

Después describimos el invariante de representación de esta estructura.

$$\mathit{DMAVálidos}(\mathit{día},\ \mathit{mes},\ \mathit{año})$$

donde:

$$\begin{aligned}
\mathit{DMAVálidos}(d, m, a) \equiv\ & \big(1 \leq d \leq 31 \ \wedge\ (m = 1 \vee m = 3 \vee \dots)\big) \vee \\
& \big(1 \leq d \leq 30 \ \wedge\ (m = 4 \vee m = 6 \vee \dots)\big) \vee \\
& \big(1 \leq d \leq 29 \ \wedge\ m = 2 \wedge \mathit{Bisiesto}(a)\big) \vee \\
& \big(1 \leq d \leq 28 \ \wedge\ m = 2 \wedge \neg \mathit{Bisiesto}(a)\big)
\end{aligned}$$

$$\mathit{Bisiesto}(a) \equiv \big(a \text{ div } 4 = 0 \wedge (a \text{ div } 100 \neq 0 \vee a \text{ div } 400 = 0)\big)$$

TAD Fecha - Posible implementación

$$\text{Fecha} == \langle \textit{día}\colon \mathbb{Z},\ \textit{mes}\colon \mathbb{Z},\ \textit{año}\colon \mathbb{Z} \rangle$$

Por último, damos los algoritmos de las operaciones del TAD Fecha para la estructura de representación elegida:

CrearFecha(d, m, a) $\rightarrow$ Fecha: (Pre: $\textit{DMAVálidos}(d, m, a)$)

$RV.\textit{día} \leftarrow d$

$RV.\textit{mes} \leftarrow m$

$RV.\textit{año} \leftarrow a$

donde $d, m, a : \mathbb{Z}$.

Operaciones como esta se conocen como constructores.
Permiten armar elementos del TAD.

TAD Fecha - Posible implementación

$$\text{Fecha} == \langle \textit{día}\colon \mathbb{Z},\ \textit{mes}\colon \mathbb{Z},\ \textit{año}\colon \mathbb{Z} \rangle$$

Más operaciones del TAD Fecha (para f: Fecha):

f.Día() $\to \mathbb{Z}$:

$RV \leftarrow f.\textit{día}$

f.Mes() $\to \mathbb{Z}$:

$RV \leftarrow f.\textit{mes}$

f.Año() $\to \mathbb{Z}$:

$RV \leftarrow f.\textit{año}$

Estas operaciones se conocen como proyectores.
Permiten inspeccionar elementos de un TAD.

TAD Fecha - Posible implementación

$$\text{Fecha} == \langle \textit{día}\colon \mathbb{Z},\ \textit{mes}\colon \mathbb{Z},\ \textit{año}\colon \mathbb{Z} \rangle$$

Más operaciones del TAD Fecha (para f_1, f_2: Fecha):

f_1.Menor(f_2) $\rightarrow \mathbb{B}$:

```
if (f1.Año() < f2.Año()) {
    RV ← TRUE
}
elsif (f1.Año() = f2.Año() ∧ f1.Mes() < f2.Mes()) {
    RV ← TRUE
}
elsif (f1.Año() = f2.Año() ∧ f1.Mes() = f2.Mes() ∧ f1.Día() < f2.Día()) {
    RV ← TRUE
}
else {
    RV ← FALSE
}
```

TAD Fecha - Posible implementación

$$\text{Fecha} == \langle \textit{día}\colon \mathbb{Z},\ \textit{mes}\colon \mathbb{Z},\ \textit{año}\colon \mathbb{Z} \rangle$$

Más operaciones del TAD Fecha (para f: Fecha):

f.FechaSiguiente() $\rightarrow$ Fecha:

```
d  ← f.Día() + 1
m ← f.Mes()
a  ← f.Año()
if (d > CantidadDeDías(m, a)) {
    d ← 1
    m ← m + 1
    if (m > 12) {
       m ← 1
       a ← a + 1
    }
}
RV ← CrearFecha(d, m, a)
```

TAD Fecha - Posible implementación

$$\text{Fecha} == \langle \textit{día}\colon \mathbb{Z},\ \textit{mes}\colon \mathbb{Z},\ \textit{año}\colon \mathbb{Z} \rangle$$

Más operaciones del TAD Fecha (para f: Fecha; $m, a : \mathbb{Z}$):

CantidadDeDías(m, a) $\to \mathbb{Z}$: (Pre: $1 \leq m \leq 12$)

```
if (m = 1 ∨ m = 3 ∨ m = 5 ∨ m = 7 ∨ m = 8 ∨ m = 10 ∨ m = 12) {
    RV ← 31
} elsif (m = 4 ∨ m = 6 ∨ m = 9 ∨ m = 11) {
    RV ← 30
} elsif (m = 2 ∧ EsBisiesto(a)) {
    RV ← 29
} else {
    RV ← 28
}
```

Podemos elegir que CantidadDeDías y EsBisiesto sean operaciones privadas: no accesibles para usuarios del TAD Fecha.

Partes de un Tipo Abstracto de Datos

- Parte pública: Disponible para el usuario externo.
  - Nombre y tipos paramétricos (ej: Fecha, Lista(ELEM)).
  - Operaciones, sus especificaciones y posiblemente sus órdenes de complejidad temporal.

- Parte privada: Sólo accesible desde dentro del TAD. ¡El usuario externo **nunca** debe ver ni meterse en esto!
  - **Estructura de representación** del TAD.
  - **Invariante de representación**: qué propiedades debe cumplir la estructura elegida para que tenga sentido como la representación de una instancia del TAD.
  - **Algoritmos de las operaciones** para la estructura de representación elegida.

TAD Lista(ELEM)

- CrearLista() $\rightarrow$ Lista(ELEM)
- L.Agregar(x)
- L.Longitud() $\rightarrow \mathbb{Z}$
- L.Ésimo(i) $\rightarrow$ *ELEM*
- L.Cantidad(x) $\rightarrow \mathbb{Z}$
- L.Insertar(i, x)
- L.Borrarlésimo(i)
- L.Indice(x) $\rightarrow \mathbb{Z}$

donde L : Lista(ELEM), $i : \mathbb{Z}$, x : ELEM (entero, char, etc.).

TAD Lista(ELEM) - Posible implementación

Vamos a representar una lista como una cadena de nodos.
Definamos primero qué es un nodo.

Nodo(ELEM) : $\langle \mathit{valor} : \text{ELEM}, \ \mathit{siguiente} : \text{Ref(Nodo)} \rangle$

$\langle ... \rangle$ define una tupla.

Ref(Nodo) es una referencia a una instancia de tipo Nodo. Puede tomar el valor especial *None* ("referencia a nada").

Construcción de un nuevo Nodo:

- $n \leftarrow \text{Nodo}(x, r)$

Acceso a los campos de un nodo n:

- $n.\mathit{valor}$ devuelve el campo *valor*.
- $n.\mathit{siguiente}$ devuelve el campo *siguiente*.

TAD Lista(ELEM) - Posible implementación

Primero elegimos una estructura de representación del TAD Lista(ELEM) (que es **privada**).

$$\text{Lista(ELEM)} == \langle \textit{cabeza} : \text{Ref(Nodo)},\ \textit{longitud} : \mathbb{Z} \rangle$$

Después describimos el invariante de representación de esta estructura. En este caso *longitud* siempre debe ser igual a la cantidad de nodos encadenados a partir de *cabeza*, y en la cadena de nodos no deben formarse ciclos.

Por último, damos los algoritmos de las operaciones del TAD Lista(ELEM) para la estructura de representación elegida:

CrearLista() $\rightarrow$ Lista(ELEM):
    $RV.cabeza \leftarrow None$
    $RV.longitud \leftarrow 0$

TAD Lista(ELEM) - Posible implementación

```
L.Agregar(x):
    nuevo ← Nodo(x, None)
    if (L.cabeza = None) {
        L.cabeza ← nuevo
    }
    else {
        aux ← L.cabeza
        while (aux.siguiente ≠ None) {
            aux ← aux.siguiente
        }
        aux.siguiente ← nuevo
    }
    L.longitud ← L.longitud + 1
```

TAD Lista(ELEM) - Posible implementación

L.Longitud() $\rightarrow \mathbb{Z}$:
    $RV \leftarrow L.longitud$

L.lésimo(i) $\rightarrow ELEM$: (Pre: $0 \leq i < L$.Longitud())
    $aux \leftarrow L.cabeza$
    while ($i > 0$) {
        $aux \leftarrow aux.siguiente$
        $i \leftarrow i - 1$
    }
    $RV \leftarrow aux.valor$

Y así con las otras operaciones del TAD Lista(ELEM): Cantidad, Insertar, Indice y Borrarlésimo

TAD Lista(ELEM) - Posible implementación

Complejidad temporal de los algoritmos vistos para esta estructura de representación.

- CrearLista() $\rightarrow$ Lista(ELEM) $O(1)$
- L.Agregar(x) $O(n)$
- L.Longitud() $\rightarrow \mathbb{Z}$ $O(1)$
- L.Iésimo(i) $\rightarrow$ *ELEM* $O(n)$
- L.Cantidad(x) $\rightarrow \mathbb{Z}$ $O(n)$
- L.Insertar(i, x) $O(n)$
- L.Borrariésimo(i) $O(n)$
- L.Indice(x) $\rightarrow \mathbb{Z}$ $O(n)$

donde L : Lista(ELEM), $i : \mathbb{Z}$, x : ELEM (entero, char, etc.).

Repaso de la clase de hoy

Tipos Abstractos de Datos

- Parte pública: Disponible para el usuario externo.
  - Nombre y tipos paramétricos (ej: Fecha, Lista(ELEM)).
  - Operaciones, sus especificaciones y posiblemente sus órdenes de complejidad temporal.

- Parte privada: Sólo accesible desde dentro del TAD. ¡El usuario externo **nunca** debe ver ni meterse en esto!
  - **Estructura de representación** del TAD.
  - **Invariante de representación**: qué propiedades debe cumplir la estructura elegida para que tenga sentido como la representación de una instancia del TAD.
  - **Algoritmos de las operaciones** para la estructura de representación elegida.