Introducción a la Computación (Matemática)

Primer Cuatrimestre de 2018

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?

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.

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.

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

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.

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.

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?

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

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.

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

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

$RV \leftarrow 0$
while (Menor(f_1, f_2)) {
    $RV \leftarrow RV + 1$
    $f_1 \leftarrow$ FechaSiguiente(f_1)
}

Modularidad y encapsulamiento

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

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.

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).

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.

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

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

$RV \leftarrow 0$
while (Menor(f_1, f_2)) {
    $RV \leftarrow RV + 1$
    $f_1 \leftarrow$ FechaSiguiente(f_1)
}

¿Cómo se implementa?

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

$RV \leftarrow 0$
while (Menor(f_1, f_2)) {
    $RV \leftarrow RV + 1$
    $f_1 \leftarrow$ FechaSiguiente(f_1)
}

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

Arreglos y Listas

En C++ teníamos el tipo `array`.

- Ejemplo:

```
array<int, 2> a;
a[0] = 10;
a[1] = a[0] + 1;
cout << a.size();
```

Arreglos y Listas

En C++ teníamos el tipo `array`.

- Ejemplo:

```
array<int, 2> a;
a[0] = 10;
a[1] = a[0] + 1;
cout << a.size();
```

- Los arreglos son muy eficientes.
- Pero proveen muy poca funcionalidad, y no pueden cambiar de tamaño.

Arreglos y Listas

En C++ teníamos el tipo `array`.

- Ejemplo:

```
array<int, 2> a;
a[0] = 10;
a[1] = a[0] + 1;
cout << a.size();
```

- Los arreglos son muy eficientes.
- Pero proveen muy poca funcionalidad, y no pueden cambiar de tamaño.

Las listas (p.ej., en Python) son bastante más útiles. Proveen las operaciones de arreglos, y muchas otras.

TAD Lista(ELEM)

TAD Lista(ELEM)

Operaciones:

- CrearLista() $\rightarrow$ Lista(ELEM): Crea una lista vacía.

TAD Lista(ELEM)

Operaciones:

- CrearLista() $\rightarrow$ Lista(ELEM): Crea una lista vacía.
- L.Agregar(x): Inserta el elemento x al final de L.

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

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.

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

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().

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

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 $L = a_0, .., a_{i-1}, x, a_i, .., a_{n-1}$.)

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.Longitud$() - 1$) de la primera aparición de x en L. Pre: L.Cantidad$(x) > 0$.

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.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: $\mathit{Máximo} : L \in Lista(\mathbb{Z}) \rightarrow \mathbb{Z}$

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

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

TAD Lista(ELEM) - Ejemplo de uso

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

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 $(L.Iésimo(i) > m)$ {
        $m \leftarrow L.Iésimo(i)$
    }
    $i \leftarrow i + 1$
}
$RV \leftarrow m$

TAD Lista(ELEM) - Implementación

La clase que viene vamos a ver formas de implementar Listas y otros TADs.

En esta clase nos enfocamos en el **uso** de los TADs, que podemos haber creado nosotros o alguien más.

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

TAD Pila(ELEM)

Operaciones:

- CrearPila() $\rightarrow$ Pila(ELEM): Crea una pila vacía.

TAD Pila(ELEM)

Operaciones:

- CrearPila() $\rightarrow$ Pila(ELEM): Crea una pila vacía.
- P.Apilar(x): Inserta el elem. x sobre el tope de la pila P.

TAD Pila(ELEM)

Operaciones:

- CrearPila() $\rightarrow$ Pila(ELEM): Crea una pila vacía.
- P.Apilar(x): Inserta el elem. x sobre el tope de la pila P.
- P.Vacía?(P) $\rightarrow \mathbb{B}$: Dice si la pila P está vacía.

TAD Pila(ELEM)

Operaciones:

- CrearPila() $\rightarrow$ Pila(ELEM): Crea una pila vacía.
- P.Apilar(x): Inserta el elem. x sobre el tope de la pila P.
- P.Vacía?(P) $\rightarrow \mathbb{B}$: Dice si la pila P está vacía.
- P.Tope() $\rightarrow ELEM$: Devuelve el elemento del tope de P.
  Precondición: $\neg P$.Vacía?().

TAD Pila(ELEM)

Operaciones:

- CrearPila() $\rightarrow$ Pila(ELEM): Crea una pila vacía.
- P.Apilar(x): Inserta el elem. x sobre el tope de la pila P.
- P.Vacía?(P) $\rightarrow \mathbb{B}$: Dice si la pila P está vacía.
- P.Tope() $\rightarrow ELEM$: Devuelve el elemento del tope de P.
  Precondición: $\neg P$.Vacía?().
- P.Desapilar(): Borra el elemento del tope de P.
  Precondición: $\neg P$.Vacía?().

donde P : Pila(ELEM), x : ELEM (entero, char, etc.).

Las pilas tienen estrategia LIFO (*last in, first out*).

Problema: Paréntesis balanceados

Dado un string, determinar si los caracteres { }, [ ], ( ) están balanceados correctamente.

Ejemplos:

- "{a(b)x[()]}" está balanceado.
- "}", "a(b))" y "[[})" no están balanceados.

¿Se les ocurre una solución? Sugerencia: usar el tipo Pila.

TAD Cola(ELEM)

Operaciones:

- CrearCola() $\rightarrow$ Cola(ELEM): Crea una cola vacía.
- C.Encolar(x): Agrega el elemento x al final de la cola C.
- C.Vacía?() $\rightarrow \mathbb{B}$: Dice si la cola C está vacía.
- C.Primero() $\rightarrow ELEM$: Devuelve el primer elemento de C.
  Precondición: $\neg C$.Vacía?().
- C.Desencolar(): Borra el primer elemento de C.
  Precondición: $\neg C$.Vacía?().

donde C : Cola(ELEM), x : ELEM (entero, char, etc.).

Las colas tienen estrategia FIFO (*first in, first out*).

Otro ejemplo

Queremos representar conjuntos de especies animales.
Podemos hacerlo con listas, por ejemplo:

- Felinos = [león, gato, tigre, guepardo, pantera, puma]
- Cánidos = [lobo, coyote, chacal, dingo, zorro]
- Cetáceos = [delfín, ballena, orca, narval, cachalote]

Otro ejemplo

Queremos representar conjuntos de especies animales.
Podemos hacerlo con listas, por ejemplo:

- Felinos = [león, gato, tigre, guepardo, pantera, puma]
- Cánidos = [lobo, coyote, chacal, dingo, zorro]
- Cetáceos = [delfín, ballena, orca, narval, cachalote]

¿Las listas son una buena forma de representar conjuntos?

Otro ejemplo

Queremos representar conjuntos de especies animales.
Podemos hacerlo con listas, por ejemplo:

- Felinos = [león, gato, tigre, guepardo, pantera, puma]
- Cánidos = [lobo, coyote, chacal, dingo, zorro]
- Cetáceos = [delfín, ballena, orca, narval, cachalote]

¿Las listas son una buena forma de representar conjuntos?

- Orden: [lobo, coyote] = [coyote, lobo] ?
- Repetidos: [delfín, delfín] ?

Otro ejemplo

Queremos representar conjuntos de especies animales.
Podemos hacerlo con listas, por ejemplo:

- Felinos = [león, gato, tigre, guepardo, pantera, puma]
- Cánidos = [lobo, coyote, chacal, dingo, zorro]
- Cetáceos = [delfín, ballena, orca, narval, cachalote]

¿Las listas son una buena forma de representar conjuntos?

- Orden: [lobo, coyote] = [coyote, lobo] ?
- Repetidos: [delfín, delfín] ?

Mejor usar un TAD Conjunto que nos evite estos problemas.

TAD Conjunto(ELEM)

Operaciones:

TAD Conjunto(ELEM)

Operaciones:

- CrearConjunto() $\rightarrow$ Conjunto(ELEM): Crea un conjunto vacío.

TAD Conjunto(ELEM)

Operaciones:

- CrearConjunto() $\rightarrow$ Conjunto(ELEM): Crea un conjunto vacío.
- C.Agregar(x): Agrega el elemento x al conjunto C.

TAD Conjunto(ELEM)

Operaciones:

- CrearConjunto() $\rightarrow$ Conjunto(ELEM): Crea un conjunto vacío.
- C.Agregar(x): Agrega el elemento x al conjunto C.
- C.Pertenece?(x) $\rightarrow \mathbb{B}$: Dice si el elemento x está en C.

TAD Conjunto(ELEM)

Operaciones:

- CrearConjunto() $\rightarrow$ Conjunto(ELEM): Crea un conjunto vacío.
- C.Agregar(x): Agrega el elemento x al conjunto C.
- C.Pertenece?(x) $\rightarrow \mathbb{B}$: Dice si el elemento x está en C.
- C.Eliminar(x): Elimina el elemento x de C.

TAD Conjunto(ELEM)

Operaciones:

- CrearConjunto() $\rightarrow$ Conjunto(ELEM): Crea un conjunto vacío.
- C.Agregar(x): Agrega el elemento x al conjunto C.
- C.Pertenece?(x) $\rightarrow \mathbb{B}$: Dice si el elemento x está en C.
- C.Eliminar(x): Elimina el elemento x de C.
- C.Tamaño() $\rightarrow \mathbb{Z}$: Devuelve la cantidad de elementos de C.

TAD Conjunto(ELEM)

Operaciones:

- CrearConjunto() $\rightarrow$ Conjunto(ELEM): Crea un conjunto vacío.
- C.Agregar(x): Agrega el elemento x al conjunto C.
- C.Pertenece?(x) $\rightarrow \mathbb{B}$: Dice si el elemento x está en C.
- C.Eliminar(x): Elimina el elemento x de C.
- C.Tamaño() $\rightarrow \mathbb{Z}$: Devuelve la cantidad de elementos de C.
- C_1.Igual?(C_2) $\rightarrow \mathbb{B}$: Dice si los dos conjuntos son iguales.

donde C, C_1, C_2 : Conjunto(ELEM), x : ELEM.

TAD Conjunto(ELEM)

Operaciones (cont.):

- C_1.Unión(C_2) $\rightarrow$ Conjunto(ELEM): Devuelve un nuevo conjunto con $C_1 \cup C_2$.
- C_1.Intersección(C_2) $\rightarrow$ Conjunto(ELEM): Devuelve un nuevo conjunto con $C_1 \cap C_2$.
- C_1.Diferencia(C_2) $\rightarrow$ Conjunto(ELEM): Devuelve un nuevo conjunto con $C_1 \backslash C_2$.

TAD Conjunto(ELEM)

Operaciones (cont.):

- C_1.Unión(C_2) $\rightarrow$ Conjunto(ELEM): Devuelve un nuevo conjunto con $C_1 \cup C_2$.
- C_1.Intersección(C_2) $\rightarrow$ Conjunto(ELEM): Devuelve un nuevo conjunto con $C_1 \cap C_2$.
- C_1.Diferencia(C_2) $\rightarrow$ Conjunto(ELEM): Devuelve un nuevo conjunto con $C_1 \backslash C_2$.
- C.ListarElementos() $\rightarrow$ Lista(ELEM): Devuelve una lista de los elementos de C, en cualquier orden.

TAD Conjunto(ELEM)

Operaciones (cont.):

- C_1.Unión(C_2) $\rightarrow$ Conjunto(ELEM): Devuelve un nuevo conjunto con $C_1 \cup C_2$.
- C_1.Intersección(C_2) $\rightarrow$ Conjunto(ELEM): Devuelve un nuevo conjunto con $C_1 \cap C_2$.
- C_1.Diferencia(C_2) $\rightarrow$ Conjunto(ELEM): Devuelve un nuevo conjunto con $C_1 \backslash C_2$.
- C.ListarElementos() $\rightarrow$ Lista(ELEM): Devuelve una lista de los elementos de C, en cualquier orden.
- C.AgregarTodos(L): Agrega todos los elementos de L en C.

donde C, C_1, C_2: Conjunto(ELEM), x: ELEM, L: Lista(ELEM).

Repaso de la clase de hoy

- Tipos Abstractos de Datos.
  - Parte pública: nombre + especificación de operaciones.
  - Parte privada (implementación): clase que viene.
- Parte pública de:
  - TAD Fecha.
  - TAD Lista(ELEM).
  - TADs Pila(ELEM), Cola(ELEM), Conjunto(ELEM).