INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS (TAD´S)

ESTRUCTURAS DE DATOS

Profesora: Mª José Domínguez Alda

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

OBJETIVO GENERAL:

- Entender la importancia de realizar la definición de los tipos de datos de forma abstracta, independientemente de su posterior implementación.

OBJETIVOS ESPECÍFICOS:

- Estudiar una técnica formal para realizar la especificación de un TAD.
- Estudiar las especificaciones de los TAD´s fundamentales.
- Estudiar los aspectos relativos a la programación con TAD´s.

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS (2)

1. Concepto de abstracción.
2. La abstracción en la historia de la programación.
3. ¿Qué es un tipo abstracto de datos?.
4. La programación con tipos abstractos de datos.
5. Especificación formal de tipos abstractos de datos.
6. Pseudocódigo.
7. Notación Asintótica.
8. Ejemplos.

CONCEPTO DE ABSTRACCIÓN

- La **abstracción** es un mecanismo de la mente humana que resulta fundamental para la comprensión de fenómenos o situaciones que incluyen gran cantidad de detalles.
- El proceso de abstracción comprende dos aspectos que son complementarios:
  1. Destacar los detalles relevantes del objeto, sistema,... en estudio.
  2. Ignorar los detalles irrelevantes, por supuesto en este nivel de abstracción.
- La **abstracción** permite tratar la complejidad, entendida como exceso en el número de detalles.
- Generalmente se trabaja en sentido descendente, de menor a mayor nivel de detalle.

LA ABSTRACCIÓN EN LA HISTORIA DE LA PROGRAMACIÓN

ABSTRACCIÓN EN EL DISEÑO DE PROGRAMAS:

- Ensamblador: *mnemoténicos* y *macros*.
- Lenguajes de alto nivel: las *instrucciones* y los *programas*.
- Programación estructurada: *abstracciones de control* y *abstracción funcional o procedimental.*
- Programación modular: *división* en *parte pública* y *privada*.
- Programación funcional y lógica: *abstracción de la secuencia de instrucciones* que sigue la máquina al ejecutar el programa.
- Programación orientada a objetos: proceso de *abstracción* entre lo *que puede hacer* un objeto y *cómo* lo hace.

LA ABSTRACCIÓN EN LA HISTORIA DE LA PROGRAMACIÓN (2)

ABSTRACCIÓN EN LOS DATOS

- *Tipos de datos* en lenguajes de alto nivel: *los datos* dejan de tratarse como secuencias de bits manipuladas por el programador y *pasan a ser independientes de la máquina.*
- Con los *tipos definidos por el programador* se logra un mayor nivel de abstracción. *Se definen los valores concretos* del tipo y se utilizan mediante *operaciones ya predefinidas* para su manipulación.
- *Tipos de datos estructurados*: el lenguaje proporciona *constructores genéricos de tipos (*el programador debe "concretar" el parámetro formal) *y operaciones predefinidas* para manipular los valores del tipo.
- ***Tipos de datos abstractos (TAD's): permiten el uso de los datos y operaciones (encapsulación) sin conocer su representación ni la implementación de las operaciones (ocultación).***
- *Clases de Objetos: Soportan además herencia y polimorfismo.*

LA ABSTRACCIÓN EN LA HISTORIA DE LA PROGRAMACIÓN (3)

PROBLEMAS CON TIPOS DE DATOS ESTRUCTURADOS:

Considerar la implementación siguiente...

tipo fecha = **Registro**

dia: 1..31

mes: 1..12

año: 1..9999

finRegistro

- Se puede crear una función auxiliar **`festivo: fecha → boolean`**, pero no hay nada que impida al programador hacer cosas como…

  `f1.mes ← f2.dia*f2.año`

  aunque no tenga sentido.

TIPOS ABSTRACTOS DE DATOS

- El concepto de **Tipo Abstracto de Datos** fue introducido por John Guttag (1974).

- Definiciones:
  - "Un Tipo Abstracto de Datos es una *colección de valores* y *operaciones* que se definen mediante una *especificación* que es *independiente* de cualquier representación". [Ricardo Peña, 99]
  - "Un Tipo Abstracto de Datos es un *modelo matemático* con unas *operaciones* definidas sobre él". [Alfred Aho, 88]

- Objetivo:
  - Trabajar con los tipos abstractos de datos definidos por el programador de igual forma que con los tipos de datos de un lenguaje de programación.

TIPOS ABSTRACTOS DE DATOS (2)

- El programador debe **establecer la *interface*** del Tipo Abstracto de Datos, es decir *especificar* los *valores* y las *operaciones* que van a formar parte del mismo. La interface debe ser *pública*.

- Posteriormente, el programador deberá ***elegir* y *realizar la implementación*** o representación de los *datos* y *operaciones* del tipo especificado.

- La implementación debe hacerse con *ámbito de declaraciones* que no sea accesible desde fuera (*ámbito privado*). Así cualquier modificación o actualización que se realice en dicha implementación no afectará a los programas que la utilicen.

- La implementación de las operaciones se *"encapsula"* junto con la *representación* del tipo de datos. De esta forma, se facilita la localización para posteriores modificaciones o actualizaciones.

TIPOS ABSTRACTOS DE DATOS (3)

ESPECIFICACIÓN:
VALORES
+
OPERACIONES

"INTERFACE" **PÚBLICA**
DEL TAD´S

IMPLEMENTACIÓN

REPRESENTACION **PRIVADA**
DEL TAD´S:
"ENCAPSULAMIENTO"
+
"OCULTACIÓN"

TIPOS ABSTRACTOS DE DATOS (4)

CARACTERÍSTICAS:

- Compartidas con los tipos de datos de los lenguajes de programación: ***privacidad*** de la representación interna de los datos y las operaciones, y ***protección***: los valores del tipo sólo se tratan con las operaciones previstas en la especificación.

- Compartidas con los subprogramas de la programación estructurada: ***generalización*** de los tipos de datos (los procedimientos implementan las operaciones) y ***encapsulación*** de la información (la representación de los datos se declara junto con la implementación de las operaciones).

TIPOS ABSTRACTOS DE DATOS (5)

UNA NUEVA FORMA PARA LAS FECHAS

Considerar la interfaz

fun crear (día, mes, año: natural) **dev** (f: fecha)

fun incrementar (fini: fecha; num_días:entero) **dev** (ffin: fecha)

fun reducir (fini: fecha; num_dias: entero) **dev** (ffin: fecha)

fun distancia (fini, ffin: fecha) **dev** (numdías: entero)

fun día_semana (f: fecha) **dev** (dia: 1..7)

TIPOS ABSTRACTOS DE DATOS (6)

UNA NUEVA FORMA PARA LAS FECHAS

Ventajas de esta interfaz:

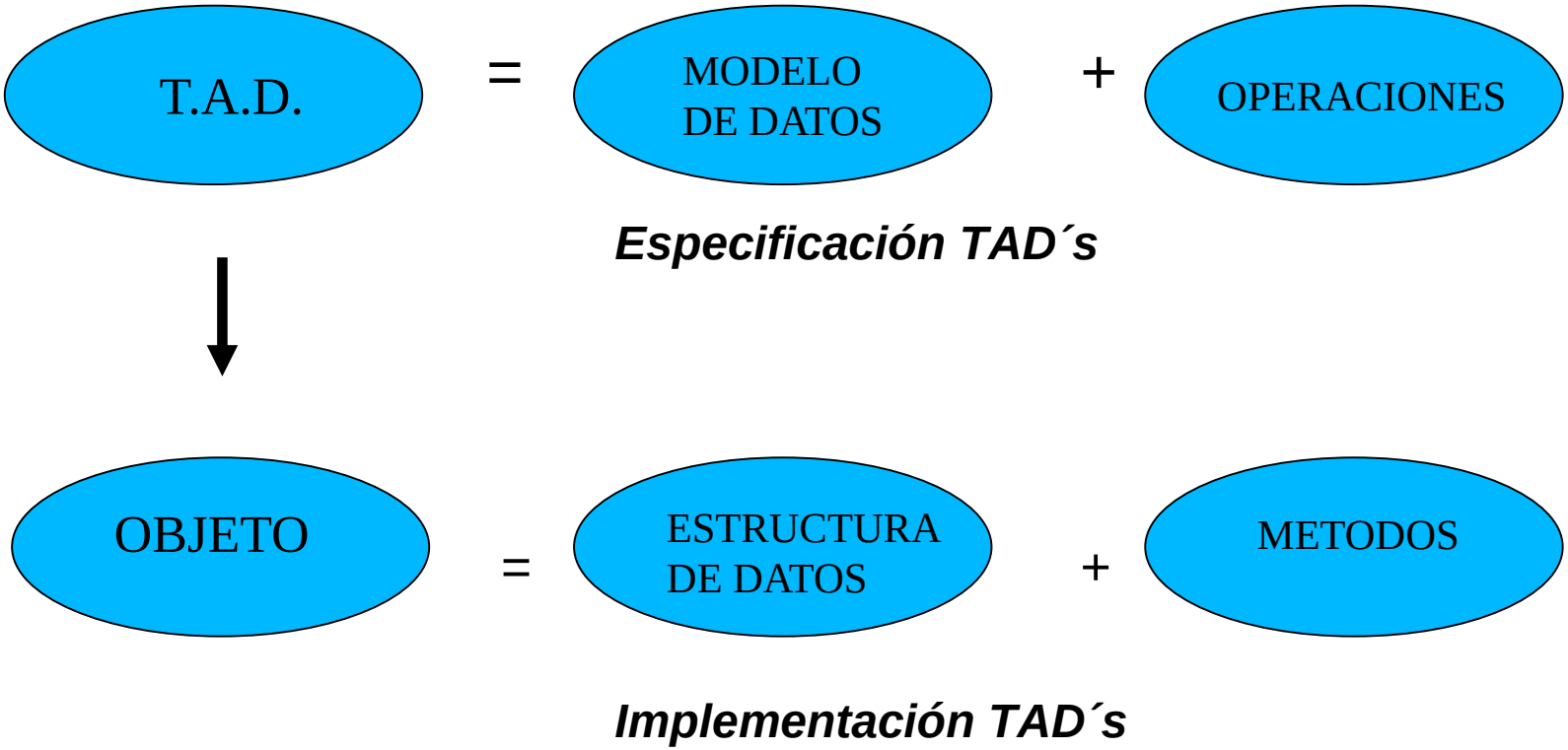
- Es independiente de la representación interna de fecha.
- Privacidad: el usuario desconoce los detalles de la representación. Solo se necesita el nombre del tipo y la especificación de las operaciones para utilizarlo.
- Protección: sólo se pueden usar las operaciones previstas por la especificación. Pueden crearse funciones auxiliares pero no definitorias del tipo.

TIPOS ABSTRACTOS DE DATOS (7)

- Destacan la importancia tanto de los **datos** como de sus **operaciones**.
- Los valores de un TAD se generan a partir de las operaciones definidas para éste.
- Es necesario crear dos “documentos” diferentes:
  a) *Especificación* del TAD: consta de *sintaxis* (nombre de las operaciones, tipos de parámetros y resultados) y *semántica* (descripción de lo que hacen estas operaciones).
  b) La *implementación* de los datos del TAD´s y de las operaciones.
- Un TAD representa una abstracción:
  a) Se *destacan* los detalles del comportamiento observable (especificación). Fijos durante la vida del programa.
  b) Se *ocultan* los detalles de la implementación. Pueden cambiar.

LA PROGRAMACIÓN CON TIPOS ABSTRACTOS DE DATOS

Tipos Abstractos de Datos y Programación Orientada a Objetos



LA PROGRAMACIÓN CON TIPOS ABSTRACTOS DE DATOS (2)

1. Descripción en lenguaje informal de los datos, tipos y operaciones.
2. Especificación del TAD:
   - Especificación formal (lenguaje algebraico).
   - Pseudocódigo (puede no ser necesaria).
3. Implementación/es del TAD:
   - Definir los tipos de datos necesarios para representar los datos en memoria de una manera concreta.
   - Especificar en lenguaje algorítmico las operaciones sobre los datos (pseudocódigo).

- Debe elegirse la implementación más adecuada teniendo en cuenta las operaciones más frecuentes.

  Estudiar la ***eficiencia*** (notación asintótica) de los algoritmos utilizados en la implementación concreta realizada.

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS

IMPORTANCIA:

1- La **especificación formal** de un TAD´S *proporciona la información necesaria* para la *posterior implementación*, facilitando esta.

2- *Permite razonar sobre la* ***corrección*** de dicha *implementación*.

3- Permite una *interpretación* ***unánime*** de los diferentes usuarios del tipo de datos especificado.

4- Es posible, a partir de la especificación, ***deducir*** *de forma automática las* ***propiedades*** *que debe satisfacer cualquier representación.*

5- Una vez especificadas las operaciones necesarias, se *analiza la* ***eficiencia*** de las diferentes posibles *implementaciones* de estas, utilizando la más adecuada para cada caso.

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS (2)

- La **especificación formal** de un tipo abstracto de datos consta de *sintaxis* (descripción de operaciones, tipos de parámetros y resultados) y *semántica* (descripción de "lo que hacen" o significado de cada operación).
- La **especificación algebraica** es una *técnica formal* que tiene el objetivo de *definir*, de forma *no ambigua e independiente* de cualquier implementación, un tipo de datos: valores del mismo y efecto de cada operación.
- Al realizar la definición de estas especificaciones se sigue una *técnica modular*, desde abajo hacia arriba, comenzando por los tipos más básicos para continuar por los más complejos.

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS (3)

SINTAXIS DE UNA ESPECIFICACIÓN ALGEBRAICA:

espec NOMBRE_ESPECIFICACIÓN

{Nombre o identificador de la especificación, en mayúsculas}

usa TIPO1, TIPO2,...

{Nombre de otros TAD´s ya especificados, en mayúsculas}

parámetro formal

{Especificación algebraica del parámetro formal, de existir}

fparámetro

géneros nombre_tipo

{Nombre de los valores del TAD que se define, en minúsculas}

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS (4)

SINTAXIS DE UNA ESPECIFICACIÓN ALGEBRAICA:

operaciones

operación: parámetro1 parámetro2 ....→ resultado

{Nombre de cada operación en minúsculas, seguido de los tipos de los parámetros y del resultado}

{Las operaciones se declaran prefijas, pero puede indicarse la ubicación de los parámetros usando guiones bajos "_"}

- Las operaciones son *funciones* con cualquier cantidad de parámetros y *un único resultado*.

ecuaciones

ti = tj

{ti y tj son valores del TAD obtenidos usando operaciones}

- Realmente cada ecuación es una *fórmula lógica*.

fespec

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS (5)

EJEMPLO: LOS NÚMEROS NATURALES

```
espec NATURALES
   géneros natural
   operaciones
        0: → natural {función constante, no tiene parámetros}
        suc: natural → natural
        _ + _: natural natural → natural
fespec
```

Ejemplos de valores del TAD:

0, suc(0), 0 + 0, suc(suc(0)), suc(0) + suc(0)…

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS (6)

SEMÁNTICA DE UNA ESPECIFICACIÓN ALGEBRAICA:

1. Cada **término sintácticamente correcto** es un valor del tipo.
   - Ejemplo: 0 + suc(0), 0 + 0 + suc(0), suc(suc (0)),...

2. **Sólo pertenecen al tipo** de datos especificado los valores generados por **términos sintácticamente correctos**. Dos términos sintácticamente distintos son distintos.
   - Ejemplo: “0 + suc(0)” y “suc(0)” son valores distintos debido a que son generados por términos diferentes.

3. Cada **ecuación expresa la igualdad entre dos términos** sintácticamente diferentes. El orden en que aparecen escritas las ecuaciones es irrelevante.

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS (7)

ASPECTOS QUE DEBEN TENERSE EN CUENTA AL DEFINIR UNA ESPECIFICACIÓN ALGEBRAICA:

a) En la definición de las operaciones y ecuaciones debe **evitarse "destruir"** las **especificaciones** realizadas para los tipos de datos, generalmente más básicos, utilizados. Es posible que se generen nuevos términos que no sean congruentes con los ya existentes o que se escriban ecuaciones que no son válidas según la especificación previa.
   - Ejemplo: una especificación en la que se utiliza el tipo boolean, especificado previamente, y se obtiene T=F ("corrupción" del tipo) o se genera un término diferente de T y F ("contaminación" del tipo)

b) Es difícil **determinar** el **número de ecuaciones** necesarias para la especificación correcta del tipo.

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS (8)

OPERACIONES DE LA ESPECIFICACIÓN ALGEBRAICA:

a) **Operaciones constructoras**: su resultado es del tipo especificado. Se pueden dividir en: ***operaciones generadoras***, que son el *mínimo conjunto de operaciones a partir de las cuales se pueden obtener todos los valores* del tipo de datos, y ***operaciones modificadoras***, que son el resto de las operaciones constructoras que no forman parte del conjunto de operaciones generadoras.

b) **Operaciones observadoras**: al menos uno de los parámetros de la operación es del tipo de interés y el resultado es de otro tipo distinto.

Si cada término obtenido a partir de las operaciones generadoras es un valor diferente del tipo especificado se dice que el conjunto de *operaciones generadoras* es un **conjunto libre**. En este caso, para cada valor del tipo de datos existe un único *término canónico*.

EJEMPLO: LOS NÚMEROS NATURALES

```
espec NATURALES
   géneros natural
   operaciones
        0: → natural
        suc: natural → natural
        _ + _: natural natural → natural
fespec
```

Conjunto *libre* de operaciones generadoras: {0, suc}

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS (10)

EJEMPLO: BOOLEANOS

```
espec BOOLEANOS
 géneros bool
 operaciones
   T: → bool
   F: → bool
   ¬: bool → bool
   _ ∧ _: bool bool →  bool
   _ ∨ _: bool bool →  bool
 fespec
```

Algunos conjuntos posibles de operaciones generadoras:

{T, F} *Conjunto libre*

{T, ¬} *Conjunto no libre:* "¬(¬(T))" debe ser el mismo valor que "T"

ESPECIFICACIÓN FORMAL DE TIPOS ABSTRACTOS DE DATOS (11)

ECUACIONES DE LA ESPECIFICACIÓN ALGEBRÁICA

a) **Ecuaciones entre generadoras:** son necesarias si es posible obtener, a partir de las operaciones generadoras, *términos sintácticamente distintos que representan el mismo valor del tipo*.

b) Para cada **operación modificadora** se escriben las ecuaciones necesarias para garantizar que *cada nuevo término obtenido puede expresarse usando las operaciones generadoras*.

c) Para cada **operación observadora** se escriben las ecuaciones necesarias para garantizar que los *términos del tipo de datos de su resultado coincide con alguno de los posibles valores del TAD* que se especifica.

EJEMPLO: LOS NÚMEROS NATURALES

```
espec NATURALES
   géneros natural
   operaciones
        0: → natural
        suc: natural → natural
        _ + _: natural natural → natural
   var x,y: natural
   ecuaciones
        x + 0 = x
        x + suc(y) = suc(x + y)
fespec
```

EJEMPLO: BOOLEANOS

espec BOOLEANOS

géneros bool

operaciones

T: → bool

F: → bool

¬: bool → bool

_ ∧ _: bool bool → bool

_ ∨ _: bool bool → bool

var x: bool

ecuaciones *{usando como generadoras T y F}*

{not}	¬(T) = F	¬(F) = T
{and}	T ∧ x = x	F ∧ x = F
{or}	T ∨ x = T	F ∨ x = x

fespec

EJEMPLO: BOOLEANOS

espec BOOLEANOS_2

géneros bool

operaciones

T: → bool

F: → bool

¬: bool → bool

_ ∧ _: bool bool → bool

_ ∨ _: bool bool → bool

var x: bool

ecuaciones *{usando como generadoras T y ¬}*

	¬(¬(x)) = x	F = ¬(T)
{and}	T ∧ x = x	F ∧ x = F
{or}	T ∨ x = T	F ∨ x = x

fespec

PSEUDOCÓDIGO

- Comentarios: entre llaves *{comentario}*
- Asignación: símbolo ←
- Sentencias selectivas:

Casos

$c_1 \rightarrow P_1$

$c_2 \rightarrow P_2$

$c_3 \rightarrow P_3$

…

fcasos

c_i son condiciones, y P_i es la sentencia a realizar (ya sea simple o compuesta) cuando se cumple c_i

PSEUDOCÓDIGO(2)

- Condicionales

  si cond *entonces* P_1

  si no P_2

  fsi

- Bucles

  Mientras cond *hacer*

  P1

  fmientras

  Desde var ← vini *hasta* vfin *paso* p *hacer*

  P

  fdesde

- Sentencias básicas:

  leer() *escribir*() *error*() *Devolver*

PSEUDOCÓDIGO(3)

- Tipos de datos primitivos:
  - bool (booleanos)
  - nat (naturales), ent (enteros), real (reales)
  - car (caracteres)
  - Tipos enumerados [v_1, … , v_k]
  - Rangos: i..j (e.g. index:1..10)
  - vectores: v[i,j]
  - Registros

    reg

    campo$_1$: tipo

    …

    campo$_n$: tipo

    freg

PSEUDOCÓDIGO(4)

- Funciones

  fun nombre(arg_1:$tipo_1$, …, arg_n:$tipo_n$) dev s:tipo

  var x_1:tipo, …, x_n:tipo

  P {codigo, incluyendo al menos una instrucción *Devolver*}

  ffun

- Procedimiento

  proc nombre (E arg_1:$tipo_1$,…, E arg_n:$tipo_n$,

  E/S arg_1:$tipo_1$,…,E/S arg_n:$tipo_n$)

  var x_1:tipo, …, x_n:tipo

  P {codigo}

  fproc

PSEUDOCÓDIGO(5)

- Punteros

  Declaración de un puntero:

  p: ^tipo

  p: puntero a tipo

  Reserva de memoria:

  reservar(p)

  Liberación de memoria:

  liberar(p)

  Acceso al valor al que apunta un puntero

  p^

EJEMPLO: BOOLEANOS

```
espec BOOLEANOS
ecuaciones
   fun not (b:bool):bool
     Si b entonces Devolver F
          si no Devolver T
   ffun
   fun and (b1, b2:bool):bool
     Si b1 entonces Devolver b2
           si no Devolver F
   ffun
   fun or (b1, b2:bool):bool
     Si b1 entonces Devolver T
           si no Devolver b2
   ffun
   fespec
```

NOTACIÓN ASINTÓTICA O

- Es una función del tamaño de la entrada de datos (n) utilizada para hacer referencia al tiempo máximo de ejecución T(n) de un programa para una entrada de tamaño n.
  - Ejemplo: El tiempo de ejecución de un programa es $O(n^2)$: existen dos valores n_0 y c de forma que para tamaños de entrada $n > n_0$, se cumple que $T(n) < cn^2$

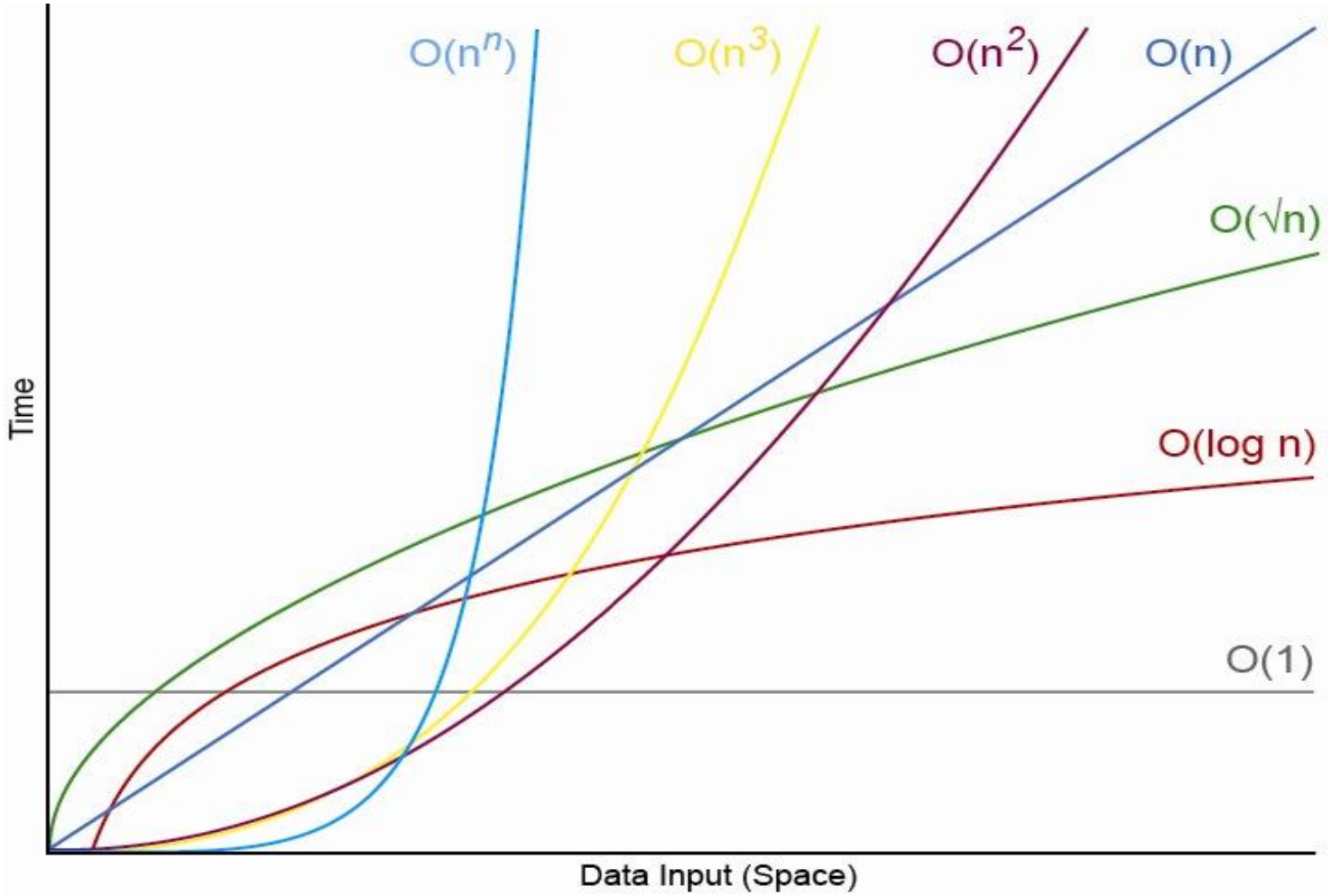
- Suponemos que es posible evaluar programas comparando sus funciones de tiempo de ejecución.
  - Ejemplo: ¿Un programa $O(n^2)$ es mejor que otro $O(n^3)$?

NOTACIÓN ASINTÓTICA O (2)

- No podemos olvidar la importancia de la velocidad de crecimiento de la función y, por tanto, la influencia del tamaño de las entradas en el tiempo de ejecución.
  - En el ejemplo anterior si consideramos $n<20$ entonces $5n^3<100n^2$, aunque la función n^3 sea peor que la función n^2

- La elección del algoritmo puede basarse en el comportamiento de la función: debemos por tanto elegir programas con velocidades de crecimiento pequeñas?

NOTACIÓN ASINTÓTICA O (3)



NOTACIÓN ASINTÓTICA O (4)

- Reglas generales para el calculo del tiempo de ejecución:
  1. El tiempo de ejecución de una sentencia o instrucción simple (lectura, escritura, asignación) es O(1).
  2. El tiempo de ejecución de una composición de sentencias o instrucciones es el mayor de los tiempos de ejecución de dichas sentencias.
  3. El tiempo de ejecución de una sentencia o instrucción condicional es el tiempo de ejecución máximo entre ambas ramas (si - entonces/sino) o cada una de las sentencias de los casos (en un case) más el tiempo de evaluación de la condición (suele ser O(1)).
  4. El tiempo de ejecución de un bucle es la suma sobre todas las iteraciones del tiempo de ejecución del cuerpo más el tiempo de evaluar la condición de fin de bucle (suele ser O(1)).

IMPLEMENTACIÓN DE TIPOS ABSTRACTOS DE DATOS

- No es única: debe elegirse la implementación más adecuada teniendo en cuenta qué operaciones son las más frecuentes.

¡Hay que buscar la eficiencia de los algoritmos utilizados en la implementación!

- Implica:
  - Representar los datos en memoria de una manera concreta, es decir, definir los tipos de datos necesarios.
  - Especificar en lenguaje algorítmico las operaciones sobre los datos ➔pseudocódigo

EJEMPLO

{vector es un array con elementos en las posiciones 1..n}

```
fun MaximoVector(v:vector) dev integer;
    maxActual ← v[1]
    desde i ← 2 hasta n hacer
        si (a[i] > maxActual) entonces maxActual ← a[i]
        fsi
    fdesde
    Devolver maxActual
ffun
```

O(n)

EJEMPLO (2)

```
fun OrdIns(v:vector) dev vector
    i ← 2
    mientras (i < n) hacer                                  O(n)
        x ← v[i]
        j ← i - 1
        mientras ((j > 0) y (v[j] > x)) hacer                O(n)
            v[j+1] ← v[j]
            j ← j - 1
        fmientras
        v[j+1] ← x
        i ← i + 1
    fmientras
    Devolver v
Ffun                                                        O(n²)
```

EJEMPLO (3)

```
fun Fac(n:integer) dev integer;
   si (n=0) entonces Devolver 1
            si no Devolver n*Fac(n-1)
   fsi
ffun
```

–El caso base posee un coste constante: $T_{Fac}(0)=1$

–Para los casos recurrentes:

$T_{Fac}(n)=1+T_{Fac}(n-1)=$

$T_{Fac}(n)=1+1+T_{Fac}(n-2)=$

.....

$T_{Fac}(n)=1+1+....+1+T_{Fac}(0) = n+1 \in O(n)$

NOTACIÓN ASINTÓTICA O

- Regla para realizar el calculo de forma intuitiva:
  - $O(n^k)$ donde k es el número de bucles anidados en la *peor* rama del programa,
  - O(log n) si el programa divide el tamaño de la entrada por un número.

EJEMPLO: EL TAD CONJUNTO

DEFINICIÓN

"Un **conjunto** es una colección de elementos de un mismo tipo base, *dominio* del conjunto, distintos entre sí y con una relación de orden lineal entre ellos."

- La relación básica es la de *pertenencia*.
- El orden no tiene importancia: {2,3,4,5} y {3,4,2,5} son el mismo conjunto.
- Relación lineal:
  - ✓ Si a y b pertenecen a un conjunto dado C, o bien a < b o bien b < a o bien a = b.
  - ✓ Si a, b y c pertenecen a un conjunto dado C y a < b y b < c entonces a < c

EJEMPLO: EL TAD CONJUNTO (2)

ESPECIFICACIÓN FORMAL

espec CONJUNTOS[ELEMENTO]

usa BOOLEANOS

parámetro formal

espec elemento

operaciones

igual: elemento elemento → bool

fparámetro

géneros conjunto

operaciones *{operaciones generadoras: conjunto_vacío, insertar}*

∅: → conjunto

insertar: elemento conjunto → conjunto

es_vacio?: conjunto → booleano

pertenece?: elemento conjunto → booleano

eliminar: elemento conjunto → conjunto

EJEMPLO: EL TAD CONJUNTO (3)

ESPECIFICACIÓN FORMAL

var c: conjunto; x,y: elemento

ecuaciones

insertar(x, insertar(x, c)) = insertar(x, c)

insertar(y, insertar(x, c)) = insertar(x, insertar(y, c))

es_vacio?(∅) = T

es_vacio?(insertar(x, c)) = F

pertenece?(x, ∅) = F

pertenece?(y, insertar(x, c)) = igual(x, y) V pertenece?(y, c)

eliminar(x, ∅) = ∅

igual(x, y) = T ⇒ eliminar(y, insertar(x, c)) = c

igual(x, y) = F ⇒ eliminar(y, insertar(x, c)) = insertar(x, eliminar(y, c))

fespec

EJEMPLO: EL TAD CONJUNTO (4)

IMPLEMENTACIONES:

- Vector de Bits o vector de Booleanos.
- Lista enlazada.
- …

EJEMPLO: EL TAD CONJUNTO (5)

IMPLEMENTACION: Vector de Booleanos.

- Las posiciones del vector representan cada uno de los elementos del dominio del conjunto.
- Si la posición i-ésima del vector de Booleanos tiene el valor *True* entonces el elemento i pertenece al conjunto.

Ejemplo:

– representación de conjuntos cuyo *dominio* son las *letras mayúsculas* del alfabeto inglés:

tipo *conjunto= vector[A..Z] de Boolean;*

– el conjunto {A, B, C, F, G, Z}

estaría almacenado como [T,T,T,F,F,T,T,F,F,F.....,T]

EJEMPLO: EL TAD CONJUNTO (6)

IMPLEMENTACION: Vector de Booleanos.

```
const N =
tipo conjunto = vector [1..N] de boolean
operaciones
    proc conjunto_vacio (E/S c:conjunto)
        desde i ← 1 hasta N hacer c[i] ← F fdesde
    fproc
    fun es_vacio (c:conjunto):boolean;
        mientras ((i<N) y no (c[i])) hacer i ← i+1 fmientras
        si (no c[i]) entonces Devolver T
                        si no Devolver F
        fsi
    ffun
```

EJEMPLO: EL TAD CONJUNTO (7)

IMPLEMENTACION: Vector de Booleanos.

operaciones

fun pertenece? (e: elemento, c:conjunto) : boolean

Devolver(c[e])

ffun

proc insertar (E e: elemento, E/S c:conjunto)

c[e] ← T

fproc

proc eliminar (E e: elemento, E/S c:conjunto)

c[e] ← F

fproc

EJEMPLO: EL TAD CONJUNTO (8)

IMPLEMENTACION: Vector de Booleanos.

VENTAJAS

- Las operaciones básicas del TAD, *insertar*, *eliminar* y *pertenece*?, se realizan en tiempo constante.
- Las operaciones *es_vacio?* y *conjunto_vacío* se realizan en tiempo proporcional al tamaño del *dominio* del conjunto.

INCONVENIENTES

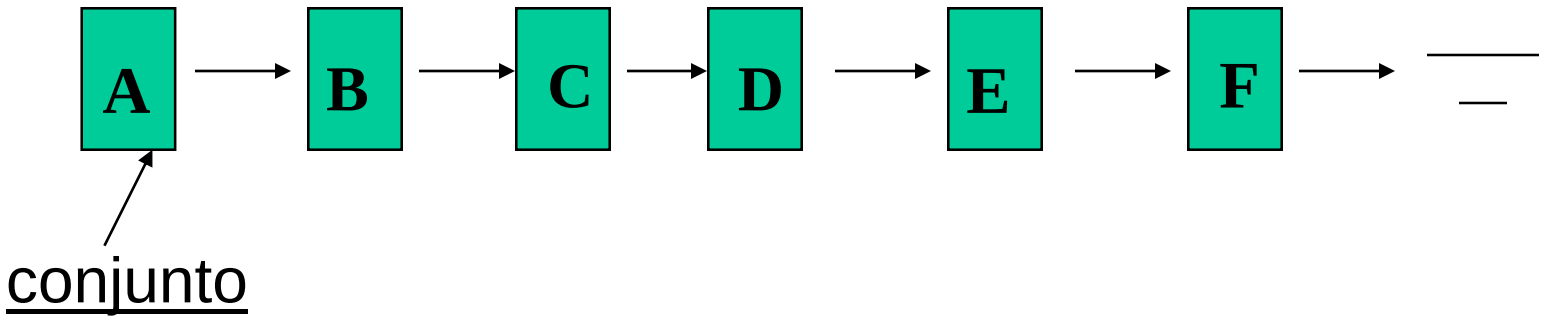
- El espacio utilizado para almacenar la estructura es constante, proporcional al tamaño del *dominio*.
- Debido a las **restricciones** que imponen los lenguajes de programación para el trabajo con variables de tipo vector, sólo es posible utilizar esta implementación cuando el *dominio* de los elementos del conjunto es un subconjunto finito de los números naturales o es posible establecer una correspondencia entre ambos.

EJEMPLO: EL TAD CONJUNTO (9)

IMPLEMENTACION: Lista enlazada.

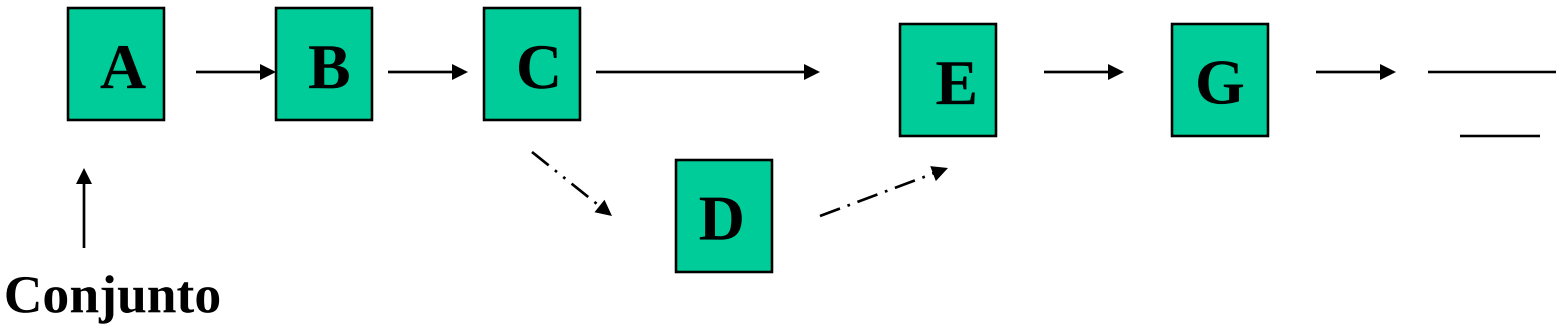
- Una *lista enlazada* es una estructura dinámica de datos en la que cada elemento, *nodo*, almacena un dato y la dirección del siguiente dato en la estructura.
- Cuando es necesario almacenar un nuevo dato se le asigna una posición nueva de memoria y se incorpora un nuevo nodo a la estructura con dicho dato.

representación del conjunto {A, B, C, D, E, F}



EJEMPLO: EL TAD CONJUNTO (10)

IMPLEMENTACION: Lista enlazada.



Ejemplo: Insertar D en el conjunto {A, B, C, E, G}

EJEMPLO: EL TAD CONJUNTO (11)

IMPLEMENTACION: Lista enlazada.

VENTAJAS

- Esta representación es más general que la anterior, no impone restricciones respecto al *dominio* del conjunto que se representa.
- El espacio utilizado es proporcional al cardinal del conjunto representado, no de su *dominio*.

INCONVENIENTES

- Debido a que se realiza una búsqueda lineal en la estructura, el tiempo utilizado por las operaciones *pertenece?*, *insertar* y *eliminar* es, en el peor de los casos, proporcional al cardinal del conjunto que se almacena.

TIPOS ABSTRACTOS DE DATOS

CONCLUSIÓN:

- Es importante realizar una definición formal de los tipos de datos al diseñar un programa. En concreto, la especificación algebraica de un TAD permite, por un lado, **definir formalmente** los **valores** y **operaciones** de los tipos de datos que se van a utilizar de forma independiente de la implementación y, por otro, **analizar diferentes implementaciones** y realizar la más adecuada.

BIBLIOGRAFÍA

- Estructuras de datos. Especificación, diseño e implementación Autor: Xavier Franch Gutiérrez Editorial: Ediciones UPC, 1999 Págs. 19-65
- Diseño de programas. Formalismo y abstracción Autor: Ricardo Peña Marí Editorial : Prentice-Hall, 1999 Págs. 155-204