

Estructuras de Datos y Algoritmos

Implementación de TAD

LECCIÓN 3

© All wrongs reversed – bajo licencia CC-BY-NC-SA 4.0



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos

Índice

- 1 Implementación de un TAD
- 2 Implementación en pseudocódigo
- 3 Implementación en C++

Índice

- 1 Implementación de un TAD
- 2 Implementación en pseudocódigo
- 3 Implementación en C++

Implementación de un TAD

De la especificación a la implementación

- Aquí **usaremos Programación Modular** (pseudocódigo y C++)
- Otras opciones (*asignaturas posteriores*): programación orientada a objetos (C++, Java, etc.)

Implementación de un TAD

Qué es y características

- **Determinar una representación** para los valores del tipo
- **Implementar sus operaciones**, a partir de esa representación

Implementación de un TAD

Qué es y características

- **Determinar una representación** para los valores del tipo
- **Implementar sus operaciones**, a partir de esa representación

Características

- **Estructurada** (facilita el desarrollo)
- **Eficiente** (optimiza el uso de recursos: tiempo, espacio)
- **Legible** (facilita su modificación y mantenimiento)
- **Segura y robusta** (programación defensiva)
- **Correcta, verificable y fácil de usar**
- Garantiza la **encapsulación**:
 - Privacidad de la representación
 - Protección del tipo

Implementación de un TAD

- **Lenguaje de programación abstracto (pseudocódigo) para implementación de TADs**
 - Bajo el paradigma de programación modular
 - [Resumen de la sintaxis en el material de la asignatura](#)

Pasos en la implementación de TAD

- 1 **Definir lo que aparecerá en la parte pública (o *interfaz*) del módulo**
 - Identificadores válidos
 - Perfiles (o cabeceras) de cada operación
 - Parámetros de entrada, salida, entrada-salida
 - Comunicación de situaciones de error

Nota: *Este primer paso es requisito previo para distribuir el trabajo de programación en equipo*

Pasos en la implementación de TAD

2 Decidir la representación del tipo de datos a definir

- Basándose en tipos básicos predefinidos, tipos estructurados básicos (vectores, registros), y/u otros tipos definidos previamente → representación interna concreta
- Ha de permitir la **implementación de las operaciones definidas de forma eficiente** (en tiempo y espacio)
- Ha de **permanecer oculta**
 - Restringir el uso del tipo a operaciones definidas en la interfaz
 - Aporta **encapsulación** (aporta privacidad y protección)

Pasos en la implementación de TAD

2 Decidir la representación del tipo de datos a definir

- Basándose en tipos básicos predefinidos, tipos estructurados básicos (vectores, registros), y/u otros tipos definidos previamente → **representación interna** concreta
- Ha de permitir la **implementación de las operaciones definidas de forma eficiente** (en tiempo y espacio)
- Ha de **permanecer oculta**
 - Restringir el uso del tipo a operaciones definidas en la interfaz
 - Aporta **encapsulación** (aporta privacidad y protección)

3 Implementar cada operación de la interfaz, así como las operaciones auxiliares que sean de interés/utilidad

- Según la representación interna definida
- Las operaciones de la interfaz serán accesibles para los usuarios del TAD
- El resto de operaciones sólo serán accesibles en la implementación del TAD

Implementación de un TAD

Guía para paso de especificación a implementación

- Las operaciones 0-arias (constantes)
 - Se implementarán como **constantes** o como **procedimientos o funciones sin parámetros**
- Las demás operaciones, como **procedimientos o funciones**
 - Varias operaciones con el mismo dominio y distinto rango podrán implementarse como un procedimiento que devuelva varios resultados
 - Especialmente si se van a usar a menudo de forma conjunta
 - Importante si de esa forma el coste de obtener los resultados se reduce
- **En las operaciones con situaciones de error** (parciales):
 - **Añadir mecanismos de protección frente a errores** dependientes del lenguaje de programación a utilizar
 - Opciones:
 - Manejo de excepciones: identificar o definir excepciones a utilizar
 - Parámetros de salida de error en cada operación

Índice

- 1 Implementación de un TAD
- 2 Implementación en pseudocódigo**
- 3 Implementación en C++

Implementación en pseudocódigo

Operaciones: procedimientos y funciones

- **Encapsulan un bloque de acciones (código) reutilizable**
- **Mejora la legibilidad** de los programas
- *Implementación de una operación como ¿procedimiento o función?*
 - Sólo hay un dato resultado a devolver → **procedimiento o función**
 - Hay 0, 2, o más resultados a devolver → **procedimiento**

Implementación en pseudocódigo

Operaciones: procedimientos y funciones

Procedimientos

{Un procedimiento es una acción o instrucción virtual, que una vez definido se puede utilizar como otra instrucción más.

Un procedimiento puede definirse con 0 o más parámetros, cada uno de ellos podrá ser de:

- Entrada (ent): dato que el procedimiento recibe para ser utilizado en sus acciones*
- Salida (sal): dato que el procedimiento comunica como resultado de sus acciones*
- Entrada y Salida (e/s): dato que el procedimiento recibe para ser utilizado en sus acciones y cuyo valor actualizado comunica como resultado de sus acciones. }*

```
procedimiento <nombre>(ent <parámetros_1>:<tipo_1>;  
                      sal <parámetros_2>:<tipo_2>;  
                      e/s <parámetros_3>:<tipo_3> ...)  
{declaraciones locales de constantes, tipos de datos, variables, ... }  
principio  
  <secuencia de acciones>  
fin
```

Implementación en pseudocódigo

Operaciones: procedimientos y funciones

Funciones

{Una función es un valor virtual, es decir, se puede utilizar dentro de una expresión y el resultado es un valor. Puede tener 0 o más parámetros, todos son de entrada, y sólo puede devolver un resultado.}

```
función <nombre>(<parám_1>:<tipo_1>; <parám_2>:<tipo_2> ...) devuelve <tipo_fun>  
{declaraciones locales de constantes, tipos de datos, variables, ...}
```

```
principio
```

```
  <secuencia de acciones>
```

```
  devuelve <valor_de_tipo_fun>
```

```
{tras devolver el valor, la función termina}
```

```
fin
```

Implementación en pseudocódigo

Uso de procedimientos y funciones

Uso de procedimientos

```
contador := 0;  
nombre(2.5, valorMedio, contador, ...);  
escribir(valorMedio); escribir(contador); ...
```

Uso de funciones

```
cálculo := 23.5 + nombre(2.7, x, y)
```

Decisiones de implementación sobre los resultados

■ Actualización de uno de los parámetros del dominio

- Parámetro del dominio y resultado será un único parámetro (e/s)
- Sólo posible con procedimientos (**las funciones sólo tienen parámetros de entrada**)
- Con esto se evita:
 - Ocupar nueva memoria para los datos resultado
 - Tiempo de copiar todo lo que no resulta modificado
- Creación de nuevas copias separadas de los datos (valor previo y posterior a la modificación) si se combina su uso con una operación *copiar* (o *duplicar*)

■ Copia distinta en memoria del parámetro del dominio

- El parámetro del dominio será de entrada y el resultado será de salida (o el valor devuelto por una función)
- Se ocupa memoria adicional (independiente) para valor y resultado
- Sustitución del valor original (en memoria) si se combina su uso con una operación *copiar* (o *duplicar*)

Implementación en pseudocódigo

Ejemplo: especificación no formal del TAD fecha

espec fechas

usa enteros, booleanos

género fecha

{(Descripción del TAD:) Los valores del TAD fechas representan fechas válidas según las reglas del calendario gregoriano}

operaciones

parcial crear: entero d, entero m, entero a -> fecha

{Dados los tres valores enteros, se obtiene una fecha compuesta con los tres valores dados usados como día, mes y año respectivamente.}

Parcial: $1 \leq d \leq 31$, $1 \leq m \leq 12$, $1582 \leq a$ y además deben formar una fecha válida según el calendario gregoriano.}

día: fecha f -> entero

{Dada una fecha f, se obtiene el entero que corresponde al día en la fecha f}

mes: fecha f -> entero

{Dada una fecha f, se obtiene el entero que corresponde al mes en la fecha f}

año: fecha f -> entero

{Dada una fecha f, se obtiene el entero que corresponde al año en la fecha f}

...

Implementación en pseudocódigo

Ejemplo: especificación no formal del TAD fecha

...

iguales: fecha f1 , fecha f2 -> booleano

{Dadas dos fechas f1 y f2, se obtiene un booleano con valor verdad si y solo si la fecha f1 es igual que la fecha f2, es decir, corresponden al mismo día, mes y año.}

anterior: fecha f1 , fecha f2 -> booleano

{Dadas dos fechas f1 y f2, se obtiene un booleano con valor verdad si y solo si la fecha f1 es cronológicamente anterior a la fecha f2.}

posterior: fecha f1 , fecha f2 -> booleano

{Dadas dos fechas f1 y f2, se obtiene un booleano con valor verdad si y solo si la fecha f1 es cronológicamente posterior a la fecha f2.}

fespec

Implementación en pseudocódigo

Ejemplo: implementación de la especificación fechas

- Lo que el módulo exporta constituye la parte visible o *interfaz*
- La implementación queda oculta, no es visible desde el exterior

módulo fechas

exporta

```
tipo fecha {Aquí se incluye la descripción de los valores del TAD}
procedimiento crear(ent d, m, a: entero; sal f: fecha; sal error: booleano)
{aquí me faltarían todas las especificaciones de cada operación ...}
función día(f: fecha) devuelve entero
función mes(f: fecha) devuelve entero
función año(f: fecha) devuelve entero
función iguales(f1, f2: fecha) devuelve booleano
función anterior(f1, f2: fecha) devuelve booleano
función posterior(f1, f2: fecha) devuelve booleano
```

implementación

```
tipo fecha = registro
    eldía, elmes, elaño: entero
freg
```

...

Tratamiento de caso de
error (operación parcial)

```

...
procedimiento crear(ent d, m, a: entero; sal f: fecha; sal error: booleano)
principio
  si d < 1 or d > 31 or m < 1 or m > 12 or a < 1582 or
    (d=31 and (m=2 or m=4 or m=6 or m=9 or m=11)) or (m=2 and d=30) entonces
      {1582=año del inicio de la adopción del calendario gregoriano}
      error := verdad
  sino
    si m=2 and d=29 and ((a mod 4/=0) or
      (a mod 4=0 and a mod 100=0 and a mod 400/=0)) entonces
      error := verdad
    sino
      f.eldía := d; f.elmes := m; f.elaño := a; error := falso
    fsi
  fsi
fin

función día(f: fecha) devuelve entero
principio
  devuelve f.eldía
fin

función mes(f: fecha) devuelve entero
principio
  devuelve f.elmes
fin

función año(f: fecha) devuelve entero
principio
  devuelve f.elaño
fin

```

...

```
función iguales(f1, f2: fecha) devuelve booleano
principio
    { devuelve f1=f2 }
    devuelve ((f1.elaño = f2.elaño) and (f1.elmes = f2.elmes)
              and (f1.eldia = f2.eldia))
fin

función anterior(f1, f2: fecha) devuelve booleano
principio
    devuelve (f1.elaño < f2.elaño) or
              ((f1.elaño = f2.elaño) and (f1.elmes < f2.elmes)) or
              ((f1.elaño = f2.elaño) and (f1.elmes = f2.elmes)
              and (f1.eldia < f2.eldia))
fin

función posterior(f1, f2: fecha) devuelve booleano
principio
    devuelve not (iguales(f1, f2) or anterior(f1, f2))
fin

fin { módulo fechas }
```

Implementación en pseudocódigo

Relación entre especificación e implementación

- Dada una especificación de TAD, hay muchas implementaciones válidas
- Cambios de implementación transparentes a los programas que lo usan

Implementación en pseudocódigo

Basura y confusión

Una implementación es una interpretación de la especificación

- Ha de mantener las propiedades del TAD, **sin introducir *basura ni confusión* en la representación elegida** (si es posible)
 - Son representables más valores de los especificados (llamados *basura*)
 - Varios de los valores especificados tienen una misma representación (*confusión*)
- **Si no es posible, han de estar documentadas** para que quien use la implementación del TAD sepa exactamente qué se le está ofreciendo

Implementación en pseudocódigo

Basura y confusión

■ Ejemplos de basura y confusión para el TAD fechas:

- **Basura**: que un dato fecha pueda tomar valores de fechas no válidas
 - Ejemplos: 31-2-2011, 2-15-2011, ...
- **Confusión**: que varios valores válidos se representen exactamente igual y por tanto sean indistinguible
 - Ejemplos: ¿qué fecha es 31-1-20? ¿31-1-1920? ¿31-1-2020?

Implementación en pseudocódigo

Basura y confusión

■ Ejemplos de basura y confusión para el TAD fechas:

- **Basura:** que un dato fecha pueda tomar valores de fechas no válidas
 - Ejemplos: 31-2-2011, 2-15-2011, ...
- **Confusión:** que varios valores válidos se representen exactamente igual y por tanto sean indistinguible
 - Ejemplos: ¿qué fecha es 31-1-20? ¿31-1-1920? ¿31-1-2020?

■ Ejemplos de limitaciones:

- Rango no infinito de valores posibles (ejemplo típico: números enteros)
- Capacidad de almacenamiento limitada por la cantidad de memoria disponible en el ordenador o por la capacidad máxima de la estructura de memoria utilizada en la implementación (ejemplo: implementación que limita el tamaño del TAD tabla)

Implementación en pseudocódigo

Especificación no formal del TAD tabla

espec tablas

usa naturales, enteros {supondremos que el 0 está en los naturales}

género tabla

{{(DESCRIPCION:)}

Los valores del TAD tablas de frecuencia representan colecciones de números enteros tales que:

- no se almacenan enteros repetidos, pero si se registra cuántas veces se ha introducido cada entero (su frecuencia)*
- las operaciones permiten obtener la información de un entero o su frecuencia*

según su puesto en el orden decreciente por valores de frecuencia}

operaciones

inicializar: -> tabla

{Devuelve una tabla vacía, es decir, que no contiene datos para ningún número entero}

añadir: tabla t, entero e -> tabla

{Si e no está t, devuelve la tabla resultante de añadir e a t con número de apariciones igual a 1; si e está en t, devuelve la tabla resultante de incrementar en 1 el número de apariciones de e (su frecuencia) en t}

...

...

total: tabla t -> natural

{Devuelve el número total de enteros para los que t contiene información}

parcial infoEnt: tabla t , natural n -> entero

{Devuelve el entero que corresponde al n-ésimo entero en t según n el orden en número de apariciones decreciente.

Parcial: la operación no está definida si n=0 OR n>total(t)}

parcial infoFrec: tabla t , natural n -> natural

{Devuelve el natural que corresponde al número de apariciones del n-ésimo entero en la tabla t según el orden en número de apariciones decreciente.

Parcial: la operación no está definida si n=0 OR n>total(t)}

fespec

Implementación en pseudocódigo

módulo tablas

exporta

tipo tabla

{tabla de frecuencias de enteros (etc) Implementación limitada a tablas con un tamaño máximo de 1000 números enteros distintos. }

procedimiento inicializar(sal t: tabla)

{Crea una tabla vacía t de frecuencias}

Tratamiento de caso de error (operación parcial)

procedimiento añadir(e/s t: tabla; ent n: entero; sal error: booleano)

{Modifica t incrementando en 1 la frecuencia de n.

La implementación limita a 1000 el nº de datos distintos, por tanto, si n no cabe en la tabla, devuelve error=verdad. }

función total(t: tabla) devuelve natural

{Devuelve el nº de enteros distintos en la tabla t.}

Tratamiento de caso de error (operación parcial)

procedimiento info(ent t: tabla; ent i: entero; sal n: entero;

sal frec: natural; sal error: booleano)

{Devuelve en n el entero que ocupa el i-ésimo lugar en la tabla t, en orden de frecuencias decrecientes, y frec es su frecuencia.

(Error:) Si no existe el entero i-ésimo, devuelve 0 en n y en frec, y error=verdad.}

implementación

...

fin

[ver módulo completo en el material de clase]

Implementación en pseudocódigo

Ejemplo de uso del módulo tablas

procedimiento estadística

{Lee una secuencia de enteros de un fichero y escribe en pantalla cada entero distinto leído junto con su frecuencia de aparición, en orden de frecuencias decrecientes. Si la tabla se llena se muestra un error. }

importa tablas, cadenas, ficheros

variables

f: fichero de entero;
nombre: cadena;
t: tabla;
dato: entero;
orden, frec: natural;
error: booleano := falso;

principio

escribir("Nombre del fichero: "); leer(nombre);
asociar(f, nombre); iniciarlectura(f);
...

```
...
inicializar(t);
mientrasQue not finFichero(f) and not error hacer
    leer(f, dato);
    añadir(t, dato, error)
fmq;
disociar(f);

si error entonces
    escribir("Error por saturación de la capacidad de la tabla utilizada.")
sino
    para orden:=1 hasta total(t) hacer
        info(t, orden, dato, frec, error);
        escribir("entero: ", dato, " frecuencia: ", frec)
    fpara
fsi
fin
```

Índice

- 1 Implementación de un TAD
- 2 Implementación en pseudocódigo
- 3 Implementación en C++**

Implementación en C++

- Para el TAD, usaremos registros (`struct`)
- Sobre la representación:
 - Privada (`private`)
 - Las operaciones del TAD **para acceder a la representación serán funciones amigas** (`friend`)
- Operaciones de tipo procedimientos: **funciones void**
- Operaciones de tipo funciones: **funciones que devuelvan un dato**
- **Parámetros de entrada:**
 - Parámetros de entrada (por valor) (útiles si ocupan poca memoria)
 - Parámetros constantes por referencia (útiles si ocupan mucha memoria)
- **Parámetros de salida o de entrada/salida:**
 - Parámetros por referencia

Implementación en C++

Ejemplo de fechas: archivo de cabecera («fecha.hpp»)

```
#ifndef _FECHA_HPP
#define _FECHA_HPP

// Interfaz del TAD fecha. Pre-declaraciones:

/* Los valores del TAD fecha representan fechas válidas
 * según las reglas del calendario gregoriano (adoptado en 1582) */
struct Fecha;

/* Dados los tres valores enteros dia, mes y anyo, se devuelve en f
 * la fecha compuesta por ellos.
 * Parcial: se precisa que 1<=dia<=31, 1<=mes<=12, 1582<=anyo, y además
 * que dia, mes y anyo formen una fecha válida según el calendario
 * gregoriano; de lo contrario, error devuelve el valor verdad */
void crear(int dia, int mes, int anyo, Fecha& f, bool& error);

/* Devuelve el dia de la fecha */
int dia(const Fecha& f);

/* Devuelve el mes de la fecha */
int mes(const Fecha& f);

/* Devuelve el año de la fecha */
int anyo(const Fecha& f);

/* Devuelve verdad si y sólo si f1 y f2 son la misma fecha */
bool iguales(const Fecha& f1, const Fecha& f2);

/* Devuelve verdad si y sólo si la fecha f1 es cronológicamente
 * anterior a la fecha f2 */
bool anterior(const Fecha& f1, const Fecha& f2);

/* Devuelve verdad si y sólo si la fecha f1 es cronológicamente
 * posterior a la fecha f2 */
bool posterior(const Fecha& f1, const Fecha& f2);
...
```

Tratamiento de caso de error (operación parcial)

```
...
// Declaración

struct Fecha {
    friend void crear(int dia, int mes, int anyo, Fecha& f, bool& error);
    friend int dia(const Fecha& f);
    friend int mes(const Fecha& f);
    friend int anyo(const Fecha& f);
    friend bool iguales(const Fecha& f1, const Fecha& f2);
    friend bool anterior(const Fecha& f1, const Fecha& f2);
    friend bool posterior(const Fecha& f1, const Fecha& f2);

    private: ← private aporta encapsulación
    // Representación de los valores del TAD.
    int elDia;
    int elMes;
    int elAnyo;
};

#endif
```

Implementación en C++

Ejemplo de fechas: archivo de implementación («fecha.cpp»)

```
#include "fecha.hpp"
```

```
// Implementación de las operaciones del TAD fecha.
```

```
    // Operaciones auxiliares sobre enteros.
```

```
    // Devuelve verdad si y sólo si el año a es bisiesto.
```

```
bool esBisiesto(int a) {  
    return ((a % 4 == 0 && a % 100 != 0) || a % 400 == 0);  
}
```

```
    // Devuelve verdad si y solo si (d,m,a) representan una fecha válida.
```

```
bool esFechaValida(int d, int m, int a) {  
    ...  
}
```

```
    // Operaciones del TAD
```

```
void crear(int dia, int mes, int anyo, Fecha& f, bool& error) {  
    ...  
}  
...
```

[ver implementación completa en el material de clase]

```
int dia(const Fecha& f) {
    return f.elDia;
}

int mes(const Fecha& f) {
    return f.elMes;
}

int anyo(const Fecha& f) {
    return f.elAnyo;
}

bool iguales(const Fecha& f1, const Fecha& f2) {
    return f1.elDia == f2.elDia && f1.elMes == f2.elMes
        && f1.elAnyo == f2.elAnyo;
}

bool anterior(const Fecha& f1, const Fecha& f2) {
    ...
}

bool posterior(const Fecha& f1, const Fecha& f2) {
    ...
}
```

[ver implementación completa en el material de clase]

Implementación en C++

Ejemplo de tablas: archivo de cabecera («tabla_frec.hpp»)

```
#ifndef _TABLA_FREC_HPP
#define _TABLA_FREC_HPP

// Constantes y tipos previos

const int MAX_NUM_DATOS = 1000;

// Interfaz del TAD tabla de frecuencias. Pre-declaraciones:

struct Tabla;
void inicializar(Tabla& t);
bool anyadir(Tabla& t, int n);
int total(const Tabla& t);
int infoEnt(const Tabla& t, int n);
int infoFrec(const Tabla& t, int n);
...

void info(const Tabla& t, int n, int& entero, int& frec, bool& error);
```

Falta incluir el tratamiento de caso de error

```
// Declaración
```

```
struct Tabla {  
    friend void inicializar(Tabla& t);  
    friend bool anyadir(Tabla& t, int n);  
    friend int total(const Tabla& t);  
    friend int infoEnt(const Tabla& t, int n);  
    friend int infoFrec(const Tabla& t, int n);
```

```
private:
```

```
// Representación interna de los valores del TAD
```

```
    struct Frecuencia {  
        int numero;  
        int frec;  
    };
```

```
    Frecuencia elementos [MAX_NUM_DATOS];  
    int numElementos;
```

```
};
```

```
#endif
```

Implementación en C++

Ejemplo de tablas: archivo de implementación («tabla_frec.cpp»)

```
#include "tabla_frec.hpp"

void inicializar(Tabla& t){
    t.numElementos = 0;
}

bool anyadir(Tabla& t, int n){
    ...
}

int total(const Tabla& t) {
    ...
}

int infoEnt(const Tabla& t, int n) {
    ...
}

int infoFrec(const Tabla& t, int n) {
    ...
}
```

[ver implementación completa en el material de clase]

Estructuras de Datos y Algoritmos

Implementación de TAD

LECCIÓN 3

© All wrongs reversed – bajo licencia CC-BY-NC-SA 4.0



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA

Aula 0.04, Edificio Agustín de Betancourt

