



U N I V E R S I D A D
COMPLUTENSE
M A D R I D

Tema 4: Diseño e implementación de TADs

Miguel Gómez-Zamalloa
Estructuras de Datos y Algoritmos (EDA)
Grado en Desarrollo de Videojuegos
Facultad de Informática

- Análisis de eficiencia
 - ① Análisis de eficiencia de los algoritmos✓
- Algoritmos
 - ② Divide y vencerás (DV), o *Divide-and-Conquer*✓
 - ③ Vuelta atrás (VA), o *backtracking*
- Estructuras de datos
 - ④ **Diseño e implementación de TADs**
 - ⑤ Tipos de datos lineales
 - ⑥ Tipos de datos arborescentes
 - ⑦ Diccionarios
 - ⑧ Aplicación de TADs

Tema 4: Diseño e implementación de TADs

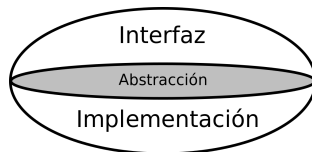
- Introducción
- Especificación de TADs
- Implementación de TADs en C++
- Ejemplo: El TAD Set

- Te plantean el siguiente problema: Dado un número x , se cogen sus dígitos y se suman sus cuadrados, para dar x_1 . Se realiza la misma operación, para dar x_2 , y así mucho rato hasta que ocurra una de las dos cosas siguientes:
 - se llega a 1, y se dice entonces que el número es “feliz”
 - nunca se llega a 1 (porque se entra en un ciclo que no incluye el 1), y se dice entonces que el número es “infeliz”
- Ejemplos:
 - el 7 es feliz: $7 \rightarrow 49 \rightarrow 97 \rightarrow 130 \rightarrow 10 \rightarrow 1$
 - el 38 es infeliz:
 $38 \rightarrow 73 \rightarrow 58 \rightarrow 89 \rightarrow 145 \rightarrow 42 \rightarrow 20 \rightarrow 4 \rightarrow 16 \rightarrow 37 \rightarrow 58$
- Deberíamos implementar (o importar) una clase *Conjunto* y plantear el algoritmo usándolo, centrándonos así en la lógica del problema.
- Los detalles de como implementar el conjunto quedarían encapsulados y abstraídos.

- La *abstracción* es un método de resolución de problemas.
 - Los problemas tienden a ser cada vez más complejos.
 - Es imposible considerar todos los detalles al mismo tiempo.
- Una abstracción es un modelo simplificado de un problema donde:
 - Se consideran los aspectos de un determinado nivel
 - y se ignoran los restantes.
- Ejemplos: construcción de edificio, receta albóndigas, internet, etc.
- Ventajas al razonar con abstracciones:
 - Se simplifica la resolución del problema
 - Soluciones claras \Rightarrow Razonamientos de corrección
 - Soluciones (o fragmentos de ellas) reutilizables
 - Permite la división de tareas

La Abstracción como Metodología de Programación

- La programación se beneficia del uso de abstracciones.
- Evolución de la programación:
 - Código máquina
 - Ensamblador
 - Lenguajes de alto nivel (funciones, módulos, TADS, POO, prog declarativa, etc)
 - Gestores de aplicaciones (hojas de cálculo, gestores de BBDD, generadores de interfaces, etc)
- En cada nivel tenemos:



- Una función o procedimiento es un conjunto de sentencias que realizan una determinada acción sobre unos datos.
- Una vez implementada se puede usar como si fuese una operación del lenguaje (abstracción).
- Es importante disponer de una especificación (formal o informal).

- El nivel de abstracción de datos ha ido aumentando:
 - Bits y bytes
 - Tipos predefinidos: `int`, `char`, `float`, etc \Rightarrow No debemos preocuparnos por su representación binaria.
 - Tipos definidos por el programador. E.g. Tipo Fecha con 3 `int`.
 - TADs definidos por el programador: Entidades donde se definen conjuntamente el tipo de datos + las operaciones que los manipulan.
- Observar la diferencia entre los tipos predefinidos y los definidos por el programador:
 - Un tipo predefinido es, en cierto modo, un TAD
 - Conjunto de valores + conjunto de operaciones.
 - El compilador chequea que no se asignen valores erróneos.
 - El programador solo necesita conocer el comportamiento de las operaciones.
 - En los tipos definidos por el programador, al tener acceso a la representación interna es posible realizar operaciones incorrectas. E.g. Fecha `f`; `f.dia = 31`;

- Un ejemplo clásico de TAD predefinido de C++ es la cadena de caracteres de su librería estándar: *std::string*
- Dada una cadena *s*, es posible saber su longitud (*s.length()* ó *s.size()*), concatenarle otras cadenas (*s += t*), sacar copias (*t = s*), o mostrarla por pantalla (*std::cout << s*), entre otras muchas operaciones.
- Todo ello sin necesidad de saber cómo reserva la memoria necesaria ni cómo codifica sus caracteres.

Un TAD se compone de:

- 1 *Interfaz*
(*visible*) \Rightarrow Especificación $\left\{ \begin{array}{l} \text{Nombre} \\ \text{Dominio del tipo} \\ \text{Operaciones y su comportamiento} \end{array} \right.$
- 2 Implementación
(*oculto*) $\left\{ \begin{array}{l} \text{Representación interna del tipo} \\ \text{Algoritmos} \Rightarrow \text{Funciones y procedimientos} \end{array} \right. \left\{ \begin{array}{l} \text{Tipos predefs} \\ \text{Tipos definidos} \\ \text{Otros TADS} \end{array} \right.$

Ejemplo: El TAD Date (de momento sin POO)

1) Especificación

- Nombre: Date
- Dominio: Las fechas desde el 1/1/1
- Operaciones + especificación informal:

```
/* Se devuelve un Date representando la fecha day/month/year.  
   day/month/year deben representar una fecha válida,  
   sino se lanzaría una excepción. */  
Date newDate(int day, int month, int year);  
  
// Se devuelve la fecha resultante al incrementar date en un día.  
Date incr(Date date);  
  
// Se devuelve la distancia en número de días entre date1 y date2.  
int diff(Date date1, Date date2);  
  
// Imprime por consola la fecha en el formato DD/MM/YYYY  
void print(Date date);
```

2) Implementación

- En general un TAD admite varias representaciones posibles.
- Normalmente cada representación facilita unas operaciones y dificulta otras.
- Alternativas para Date (ver Date1A y Date1B):
 - ❶ Registro con tres int \rightarrow day, month y year
 - ❷ Un solo int \rightarrow Número de días transcurridos desde el 1/1/1
- Con 1) se facilita `newDate` y `print`, y se dificulta `incr` y `diff`.
- Con 2) se facilita `incr` y `diff`, y se dificulta `newDate` y `print`.
- Otra alternativa sería llevar 1) + 2):
 - Al guardar más información se suelen poder implementar más fácilmente las operaciones *observadoras* (`diff` y `print`).
 - Pero la información hay que mantenerla! Se dificultan entonces las operaciones *constructoras* y *modificadoras* (`newDate` y `incr`).

Soporte para TADS en Lenguajes de Programación

- Un lenguaje ofrece soporte para la implementación de TADs si incluye mecanismos para separar la interfaz de la implementación.
 - Privacidad: La representación interna está oculta.
 - Protección: El tipo solo puede usarse a través de sus operaciones.
- **Convención:** Habitualmente se hacía un “acuerdo entre caballeros” entre usuario e implementador para no tocar la parte privada.
 - Ejemplo: Marca `_` en identificadores privados, escribiendo, e.g., `'_dia'`
 - Esto ya no es necesario en C++ pero sí lo era en C.
- Un lenguaje soporta el uso de TADs si permite elevar su uso al rango del de los tipos predefinidos del lenguaje.
- El grado de soporte varía mucho \Rightarrow Uso de módulos, POO.
- En este curso usaremos la POO de C++.
- Ejemplo: La clase `Date` (ver `Date2` y `Date3`).

Implementación de TADs en C++

- C++ es un lenguaje tremendamente flexible y potente (incluso demasiado ...).
- Hay más de una forma de hacer las cosas.
- Hay que entender las ventajas y desventajas de cada alternativa.
- Veremos las “mejores prácticas” reconocidas en la industria ⇒ Referencia para hacer tus propias implementaciones.

Clasificación de Operaciones

- Constructor** Crea una nueva instancia del tipo. En C++, un constructor se llama siempre como el tipo que construye. Se llaman automáticamente cuando se declara una nueva instancia del tipo.
- Mutador** Modifica la instancia actual del tipo. En C++, *no pueden* llevar el modificador *const* al final de su definición.
- Observador** No modifican la instancia actual del tipo. En C++, *deben* llevar el modificador *const* (aunque el compilador no genera errores si se omite).
- Destructor** Destruye una instancia del tipo, liberando cualquier recurso que se haya reservado en el momento de crearla (por ejemplo, cerrando ficheros, liberando memoria, o cerrando conexiones de red). Los destructores se invocan automáticamente cuando las instancias a las que se refieren salen de ámbito.

Mutadores vs. observadores

- Se debe usar el modificador **const** siempre que sea posible (i.e., para todas las operaciones observadoras)
- Hacer esta distinción tiene múltiples ventajas. Por ejemplo, una instancia de un tipo *immutable* (sin mutadores) se puede compartir sin que haya riesgo alguno de que se modifique el original.
- Es frecuente, durante el diseño de un TAD, poder elegir entre suministrar una misma operación como mutadora o como observadora.
- E.g., en el TAD *Fecha*, podríamos elegir entre suministrar una operación *suma(int dias)* que modifique la fecha actual sumándola *delta* días (mutadora), o que devuelva una *nueva* fecha *delta* días en el futuro (observadora, y por tanto *const*).

- Ciertas operaciones pueden ser erróneas; bien por definición o bien por limitaciones de la representación
- Ejemplos:
 - Acceso a la cima de una pila vacía
 - Acceso a un índice no válido en un vector
- Estas operaciones se denominan *parciales*
- Sus precondiciones para garantizar un comportamiento predecible deben especificarse debidamente
- Utilizaremos *excepciones* como mecanismo de tratamiento de errores

- Un TAD genérico es aquel en el que uno o más de los tipos que se usan se dejan sin identificar, permitiendo usar las mismas operaciones y estructuras con distintos tipos concretos
- Ejemplo: TAD Conjunto
 - No debería haber grandes diferencias entre almacenar enteros, punteros arbitrarios, o Rectángulos
- Hay varias formas de conseguir esta genericidad
 - 1 Plantillas
 - 2 Herencia
 - 3 Lenguajes dinámicos

1) Plantillas

- Permite declarar tipos como “de plantilla” (*templates*), que se resuelven en tiempo de compilación para producir todas las variantes concretas que se usan realmente
- Mantienen un tipado fuerte y transparente al programador

```
1 // requiere <stack>, un TAD generico 'pila' via plantillas
2     stack<bool> s;
3     s.push(true);           // s.push("ey") falla
4     bool b = s.top();      // b == true
```

2) Herencia

- Muy usado en Java y disponible en cualquier lenguaje con soporte OO.
- Requiere que todos los tipos concretos usados descendan de un tipo base que implemente las operaciones básicas que se le van a pedir

```
1 // java.util.Stack (TAD generico 'pila' via herencia)
2   Stack s = new Stack(); // contiene Object
3   s.push(Boolean.FALSE); // s.top() devuelve Object
4   s.push("ey");           // funciona sin problemas
5 // Java SE 5 en adelante: version semi-generica
6   Stack<Boolean> t = new Stack<Boolean>();
7   s.push(Boolean.FALSE); // s.top() devuelve Boolean
8   s.push("ey");           // error de compilacion
```

3) Lenguajes dinámicos

- JavaScript o Python son lenguajes que permiten a los tipos adquirir o cambiar sus operaciones en tiempo de ejecución
- En estos casos, basta con que los objetos de los tipos introducidos soporten las operaciones requeridas en tiempo de ejecución (pero se pierde la comprobación de tipos en compilación)

- En C++, se pueden definir TADs genéricos usando la sintaxis

1 **template** <**class** T_1 , ... **class** T_n > *contexto*

y refiriéndose a los T_i igual que se haría con cualquier otro tipo a partir de este momento

- Generalmente se escogen mayúsculas que hacen referencia a su uso
- Por ejemplo, para un tipo cualquiera se usaría T, para un elemento E; etc.

Ejemplo de TAD genérico: Pair

```
1 // en el .h
2 template <class A, class B>
3 class Pair {
4     // una pareja inmutable generica
5     A _a; B _b;
6 public:
7     // Generador (un constructor sencillo)
8     Pair(A a, B b) { _a=a; _b=b; } // cuerpos en el .h
9     // Observadores
10    // Pair(a, b).first() = a
11    A first() const { return _a; }
12    // Pair(a, b).second() = b
13    B second() const { return _b; }
14 };
```

Plantillas en C++

```
1 // en el main.cpp
2 #include <iostream>
3 #include <string>
4 using namespace std;
5 int main() {
6     Pair<int, string> p(4, "hola");
7     cout << p.first() << " " << p.second() << "\n";
8     return 0;
9 }
```

- En C++ se pueden definir los cuerpos de funciones en el .h en lugar de en el prototipo (tal y como se hace en el ejemplo)
- En el caso de TADs genéricos, esto es **obligatorio** (en caso contrario se producirán errores de enlazado)
- Aunque para implementaciones más grandes es preferible usar esta versión alternativa, que deja el tipo más despejado, a costa de repetir la declaración de los tipos de la plantilla para cada contexto en el que se usa:

Plantillas en C++

```
1 // en el. h
2 ...
3 // Generador (un constructor sencillo)
4 Pair(A a, B b);
5 // Observadores
6 A first() const;
7 B second() const;
8 };
9 template <class A, class B>
10 Pair<A,B>::Pair(A a, B b) { _a = a; _b = b; }
11 template <class A, class B>
12 A Pair<A,B>::first() const { return _a; }
13 template <class A, class B>
14 B Pair<A,B>::second() const { return _b; }
```

- Un TAD está formado por una colección de valores y un conjunto de operaciones sobre dichos valores
- Una **estructura de datos** es una estrategia de almacenamiento en memoria de la información que se desea guardar
- Muchos TADs se implementan utilizando estructuras de datos
 - E.g., los TADs Pila y Cola pueden implementarse usando la estructura de datos de las listas enlazadas
- Algunos TADs son de uso tan extendido y frecuente, que es de esperar que en cualquier lenguaje de programación de importancia exista una implementación. E.g.: vectores, tablas, grafos, etc.

Especificación de TADs

Especificaciones Informales

- Las especificaciones informales son, en general, imprecisas
 - E.g. Qué ocurre si hacemos `Date date; ... date += -5;?`
- Se podrían tratar de escribir todos los detalles textualmente ...
- Pero siempre habría ambigüedades e imprecisiones al no ser un lenguaje formal
- Aún así, las especificaciones informales son el escenario habitual
 - E.g. La *STL* de C++, la *API* de Java o la misma *JVM*

- Una especificación formal permite:
 - ① Verificación formal \Rightarrow Demostrar la corrección de los programas (automáticamente)
 - ② Testing formal \Rightarrow Generación automática de tests, testing automático, etc.
- Mas del 50 % de los costes en el desarrollo de software se dedica al testing!
- La verificación formal y el testing necesitan especificaciones formales
- Aspectos básicos de especificaciones formales:
 - ① Dominio del TAD \Rightarrow conjunto de posibles valores
 - ② Invariante de la representación
 - ③ Relación de equivalencia
 - ④ Especificaciones formales pre/post de cada operación

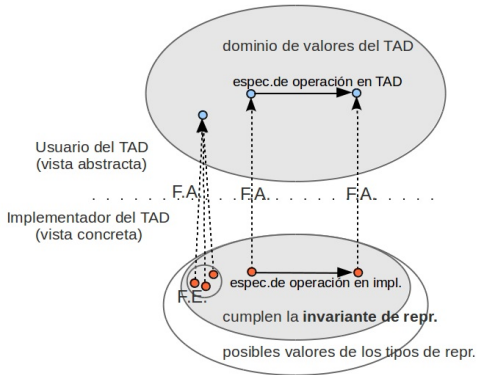
Invariante de la Representación

- Consiste en el *conjunto de condiciones que se tienen que cumplir para que una representación se considere como válida*
- Ejemplos:
 - TAD *Fecha* representado mediante tres enteros \Rightarrow Los tres enteros forman una fecha válida
 - TAD *Rectangulo* representado mediante punto origen y par de enteros para ancho y alto $\Rightarrow 0 \leq _ancho \wedge 0 \leq _alto$
- Proporcionaremos el invariante de la representación mediante lógica de primer orden (como comentario)

Relación de Equivalencia

- La **relación de equivalencia** indica cuándo dos valores del tipo implementador representan el *mismo valor* del TAD.
- En C++, una forma de implementarla es sobrecargando el operador '=='.

```
1 class Rectangulo {
2     // permite ver equivalencia de rectangulos mediante r1 == r2
3     bool operator==(const Rectangulo& r) const {
4         return (esVacio() && r.esVacio())
5             || (_alto == r._alto && _ancho == r._ancho
6                 && _origen == r._origen);
7     };
8 };
```



- El invariante de representación delimita los valores posibles del tipo representante *válidos*.
- La función de abstracción (F.A) los pone en correspondencia con términos del TAD.
- Varios valores del tipo representante pueden describir al mismo término del TAD – Función de equivalencia (F.E).

Ejemplo: El TAD Set

- Planteamos una versión sencilla con las operaciones más básicas: constructor, add, remove, contains, size y **operator**==.
- Está bien tener en mente posibles extensiones para más operaciones como: unión, intersección y diferencia.

```
template <class T>
class Set {
public:
    Set(); // Constructor

    void add(const T& e); // Inserta un elemento (mutadora)

    void remove(const T& e); // Elimina un elemento (mutadora)

    bool contains(const T& e) const; // Pertenencia (observadora)

    int size() const; // Devuelve la cardinalidad (observadora)

    bool operator==(Set<T>& other); // Equivalencia (observadora)
};
```

Ejemplo: El TAD Set

Vamos a plantear tres representaciones:

- 1 Array dinámico no ordenado y sin evitar duplicados.
- 2 Array dinámico no ordenado y **evitando duplicados**.
- 3 Array dinámico **ordenado** y **evitando duplicados**.

En los tres casos el tipo representante será igual. Lo que cambiará será la implementación de las operaciones (y el invariante de la repr.).

```
template <class T>
class Set {
protected:
    static const int TAM_INICIAL = 10; // tamaño inicial del array

    // número de elementos del array
    int nelems;

    // tamaño del array
    int capacidad;

    // puntero al array que contiene los datos (redimensionable)
    T* array;
```

Ejemplo: El TAD Set

- La construcción, destrucción, copia y asignación es igual con las tres representaciones.

```
public:
    // Constructor: conjunto vacío
    Set() : nelems(0), capacidad(TAM_INICIAL), array(new T[capacidad]) {}

    // Destructor
    ~Set() {
        libera();
    }

    // Constructor por copia
    Set(Set<T> const& other) {
        copia(other);
    }

    // Operador de asignación
    Set<T>& operator=(Set<T> const& other) {
        if (this != &other) {
            libera();
            copia(other);
        }
        return *this;
    }
}
```

Ejemplo: El TAD Set

protected:

```
void libera() {
    delete[] array;
}

// this está sin inicializar
void copia(Set const& other) {
    capacidad = other.nelems + TAM_INICIAL;
    nelems = other.nelems;
    array = new T[capacidad];
    for (int i = 0; i < nelems; ++i)
        array[i] = other.array[i];
}

void amplia() {
    T* viejo = array;
    capacidad *= 2;
    array = new T[capacidad];
    for (int i = 0; i < nelems; ++i)
        array[i] = std::move(viejo[i]);
    delete[] viejo;
}
```

Set1: Implementación con la 1ª repr.

- El resto de operaciones varía mucho dependiendo de la representación:

```
// Añadir un elemento. O(1), salvo cuando redimensiona
void add(T const& elem) {
    array[nelems] = elem;
    ++nelems;
    if (nelems == capacidad)
        amplia();
}
```

```
// Borrar elemento elem. O(n), n=nelems
void remove(T const& elem){
    bool found;
    int pos;
    linSearch(elem, 0, nelems, found, pos);
    while (found){
        --nelems;
        array[pos] = array[nelems];
        linSearch(elem, pos, nelems, found, pos);
    }
}
```

Set1: Implementación con la 1ª repr.

```
// Chequear pertenencia de e. O(n), n=nelems
bool contains(T const& e) const {
    bool found; int pos;
    linSearch(e,0,nelems,found,pos);
    return found;
}

// Consultar tamaño. O(n*log(n)), n = nelems
int size() {
    removeDups(); // O(n*log(n)) con n = nelems
    return nelems;
}

// Relación de equivalencia. O(n*log(n)), n = nelems
bool operator==(Set1<T>& other){
    this->removeDups();
    other.removeDups();
    if (nelems == other.nelems)
        return std::equal(array, array+nelems, other->array);
    else
        return false;
}
```

Set2: Implementación con la 2ª repr.

- Al no haber repeticiones size, remove y **operator==** quedarán más sencillas, pero add necesita recorrer.

```
// Añadir un elemento. O(n), n=nelems
void add(T const& elem) {
    if (!contains(elem)) {
        array[nelems] = elem;
        ++nelems;
        if (nelems == capacidad)
            amplia();
    }
}

// Borrar elemento elem. O(n), n=nelems
void remove(T const& elem){
    bool found; int pos;
    linSearch(elem, 0, nelems, found, pos);
    if (found){
        --nelems;
        array[pos] = array[nelems];
    }
}
```

Set2: Implementación con la 2ª repr.

```
// Chequear pertenencia de e. O(n), n=nelems
bool contains(T const& x) const {
    bool found; int pos;
    linSearch(x, 0, nelems, found, pos);
    return found;
}

// Consultar tamaño. O(1)
int size() const {
    return nelems;
}

// Relación de equivalencia. O(n*log(n)), n = nelems
bool operator==(Set2<T>& other){
    if (nelems == other.nelems) {
        sort(array, array + nelems); // O(n*log(n))
        sort(other.array, other.array + other.nelems);
        return std::equal(array, array + nelems, other->array);
    } else
        return false;
}
```


Set3: Implementación con la 3ª repr.

- Mantenemos el array ordenado. Se aprovecha siempre que haya que buscar. También simplifica el **operator==** al no tener que ordenar.

```
// Añadir un elemento. O(n), n=nelems
void add(T const& elem) {
    bool found; int pos;
    binSearch(elem, found, pos);
    if (!found){
        shiftRightFrom(pos+1);
        array[pos+1] = elem;
        ++nelems;
        if (nelems == capacidad) amplia();
    }
}
```

```
// Borrar elemento elem. O(n), n=nelems
void remove(T const& elem){
    bool found; int pos;
    binSearch(elem, found, pos);
    if (found) {
        shiftLeftFrom(pos);
        --nelems;
    }
}
```

Set3: Implementación con la 3ª repr.

```
// Chequear pertenencia de e.  $O(\log(n))$ ,  $n = \text{nelems}$ 
bool contains(T const& x) const {
    bool found; int pos;
    binSearch(x, found, pos);
    return found;
}

// Consultar tamaño.  $O(1)$ 
int size() const {
    return nelems;
}

// Relación de equivalencia.  $O(n)$ ,  $n = \text{nelems}$ 
bool operator==(Set3<T>& other){
    if (nelems == other.nelems)
        return std::equal(array, array + nelems, other->array);
    else
        return false;
}
```

Ejemplo: El TAD Set

Conclusiones

	Set1	Set2	Set3
add	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
remove	$\mathcal{O}(n')$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
contains	$\mathcal{O}(n')$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$
size	$\mathcal{O}(n' * \log(n'))$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
operator ==	$\mathcal{O}(n' * \log(n'))$	$\mathcal{O}(n * \log(n))$	$\mathcal{O}(n)$

- Observa que en *Set1* los recorridos son lineales en el número de elementos del array (denotado por n'), que al haber repeticiones podría ser mucho mayor que el número de elementos distintos!
- La implementación de la unión, intersección y diferencia también sería más eficiente en *Set3* ($\mathcal{O}(n)$ vs. al menos $\mathcal{O}(n * \log(n))$).
- Claramente nos quedamos con *Set3* (al que llamaremos *Set*). En el tema 6 veremos una representación que permitirá implementar también add y remove en $\mathcal{O}(\log(n))$.