TAD Fundamentales

Lección 5

Tipos Abstractos de Datos

Un Tipo Abstracto de Datos es un conjunto de valores y de operaciones definidos mediante una especificación independiente de cualquier representación

TAD = valores + operaciones

- La especificación de un TAD consiste en establecer las propiedades que lo definen:
  - el conjunto de ***valores***, el conjunto de ***operaciones***, y todas las ***propiedades*** que los definen
    - Todo ello debe tenerse en cuenta tanto al seleccionar un TAD entre los disponibles, como al definir nuevos TADs reutilizables
    - TADs similares pueden aparentar tener el mismo conjunto de operaciones y sin embargo diferenciarse en las propiedades de dichas operaciones o valores
  - Dado un TAD todas sus posibles implementaciones deben cumplir con su especificación garantizando dichas propiedades

Esquema

- TAD fundamentales
  - Contenedores
    - Multiconjuntos
    - Conjuntos
    - Diccionarios

  →Iteradores
    - Listas, árboles, grafos, …
- *Standard Template Library* (STL)

TAD fundamentales

- TAD fundamentales y algoritmos de uso frecuente, que todo futuro Ingeniero Informático debe **conocer,** debe **saber implementar** y **utilizar,** para poder **diseñar** soluciones en los nuevos contextos o problemas
- Estudiaremos algunos TAD e implementaciones que se utilizan muy frecuentemente al resolver problemas:
  - de forma aislada o combinándose para crear otros TAD y estructuras más complejas
- Situaciones habituales al resolver problemas:
  - TAD que actúan como **contenedores** de datos y que nos permitan gestionarlos y manipularlos de forma eficiente
  - TAD que permitan reflejar las relaciones o interacciones entre los datos que se dan en la realidad y permiten plantear soluciones sencillas y eficientes. Ejemplos:
    - Colas de clientes delante de una taquilla
    - Listas de tareas pendientes
    - Árboles que representan jerarquías o clasificaciones
    - Grafos de dependencias,
    - etc.…

TAD fundamentales

- Contenedores:
  - TAD que actúan como contenedores de grupos de elementos de un mismo tipo
  - Cuando no es preciso almacenar relaciones(*) entre los elementos:
    - Multiconjunto (o colección, bolsa, saco)
    - Conjunto
    - Diccionario (o mapa)
  - Para almacenar relaciones(*) entre los elementos:
    - Tipos lineales o Listas
    - Tipos no lineales

(*) *Relaciones de orden, secuencia, jerarquías, o de otro tipo*

Contenedores

- **Multiconjunto (o colección, bolsa, saco):**
  - Colección de 0 ó más elementos
  - Puede contener elementos repetidos
  - Operaciones típicas:
    - **crear**: crea un multiconjunto sin elementos
    - **añadir**: añade un elemento al multiconjunto
    - **pertenece?**: dado un elemento, comprueba si se encuentra en el multiconjunto
    - **quitar**: dado un elemento, si existe en el multiconjunto, elimina una de sus apariciones
    - **cardinal**: devuelve el número de elementos en el multiconjunto
    - **esVacío?**: comprueba si el multiconjunto está vacío (no contiene ningún elemento)
    - eliminarTodos: elimina todos los elementos del multiconjunto dejándolo vacío
    - …

    Situaciones de error en general:
      - Si se utilizan elementos de tipo distinto al de los del multiconjunto, …
  - Otras operaciones frecuentes: unión, diferencia, intersección entre multiconjuntos

Contenedores

- **Conjunto:**
  - Colección de 0 ó más elementos
  - No se admiten elementos repetidos
  - Operaciones típicas:
    - **crear**: crea un conjunto sin elementos
    - **añadir**: dado un elemento, si no pertenece al conjunto lo añade
    - **pertenece?**: dado un elemento comprueba si se encuentra en el conjunto
    - **quitar**: dado un elemento, si pertenece al conjunto lo elimina
    - **cardinal**: devuelve el número de elementos en el conjunto
    - **esVacío?**: comprueba si el conjunto esta vacío (no contiene ningún elemento)
    - eliminarTodos: elimina todos los elementos del conjunto dejándolo vacío
    - …

    Situaciones de error en general:
    - Si se utilizan elementos de tipo distinto al de los del conjunto
    - …
  - Otras operaciones frecuentes: unión, diferencia, intersección entre conjuntos

Contenedores

- **Diccionario (o mapa):**
  - Conjunto de 0 ó más elementos formados como *pares* (clave, valor)
    - DEFINICIÓN: función de elementos de tipo clave en elementos de tipo valor
    - cada clave tiene asociado un valor
  - No se permiten claves repetidas pero varias claves pueden corresponderse con el mismo valor
  - Las claves no pueden cambiar, los valores si
  - Operaciones de acceso y manipulación *por clave*
  - Operaciones típicas:
    - **crear**: crea un diccionario sin elementos
    - **añadir**: dada una clave y un valor, asigna el valor a la clave en el diccionario
    - **pertenece?**: dada una clave devuelve un booleano indicando si se encuentra en el diccionario
    - **obtenerValor**: dada una clave devuelve el valor asociado a ella
    - **quitar**: dada una clave la borra del diccionario junto con su valor
    - **cardinal**: devuelve el número de elementos en el diccionario
    - **esVacío?**: comprueba si el diccionario esta vacío (no contiene ningún elemento)
    - …

    Situaciones de error en general:
    - Si se utilizan claves o valores de tipo distinto al de los del diccionario
    - Si se intentan obtener o quitar para claves no existentes en el diccionario

Contenedores

- Con las operaciones típicas en los contenedores (*multiconjunto, conjunto,* y *diccionario)*:

  crear, añadir, esVacío?, quitar, pertenece, cardinal

  cómo podemos (los usuarios del contenedor) :
  - Mostrar todos los elementos por pantalla?
  - Mostrar todos los elementos que cumplan cierta condición?
- Falta poder acceder a todos y cada uno de los elementos, de forma eficiente:
  - Sin acceder a un elemento más de una vez
  - Sin dejarse elementos sin acceder

  → Iteradores

¡Sin exponer los detalles de la implementación (deben quedar protegidos y ocultos)!

Iteradores

- Un iterador definido sobre un contenedor (multiconjunto, conjunto o diccionario) permite visitar todos sus elementos, uno por uno
  - En orden indeterminado
  - Permiten implementar **algoritmos de recorrido o búsqueda** (es decir algoritmos de consulta, pero no modificaciones)

¡Sin exponer los detalles de la implementación (deben quedar protegidos y ocultos)!

- El iterador se puede ver como una forma de recorrer todos los elementos de forma secuencial, disponiendo de un *cursor* o índice
  - El *cursor* está entre el elemento *previo* (último visitado) y el *siguiente* (el siguiente a visitar), dentro del recorrido
  - Inicialmente no hay elemento *previo* (el siguiente es el primero a visitar)
  - Cuando se ha visitado el último elemento, no hay *siguiente*

Iteradores

- Un iterador se utiliza para visitar todos los elementos de una colección una única vez:
  - Sin dejarse ninguno, ni visitar ninguno más de una vez
- El comportamiento/funcionamiento de **un iterador** se define en base a **varias operaciones**:
  - ***iniciarIterador***: *prepara el iterador* para que el siguiente elemento a visitar sea el primero (situación de no haber visitado ningún elemento).
  - ***existeSiguiente?***: devuelve falso si ya se ha visitado el último elemento, devuelve verdad en caso contrario.
  - ***siguiente***: devuelve el siguiente elemento a visitar.
    Parcial: la operación no está definida si ya se ha visitado el último elemento (es decir, *no existeSiguiente?*).
  - ***avanza***: *prepara el iterador* para que se pueda visitar otro elemento.
    Parcial: la operación no está definida si ya se ha visitado el último elemento (es decir, *no existeSiguiente?*).

Sus **4 operaciones** están **diseñadas para ser utilizadas de una forma concreta**…

Iteradores

- Esquema básico del uso de un iterador para visitar (recorrer) **todos** los elementos del contenedor:

```
-- Para iniciar el recorrido, realizar 1 llamada a la operación:
iniciarIterador
Mientras que existeSiguiente? hacer
    -- Obtener el elemento que toca tratar realizando al menos 1 llamada a la
       operación siguiente
    -- "Tratar" dicho elemento como corresponda (escribirlo en pantalla,
       comprobar si cumple cierta condición, etc.)
       ...
    -- Realizar 1 llamada operación avanza para prepararse para poder tratar
       otro elemento
fmq          { sin utilizar ninguna operación que modifique el contenedor }
```

- No se deberían intentar utilizar operaciones parciales, en este caso **avanza** o **siguiente**, sin comprobar antes si podrán devolver un resultado (son parciales: no definidas si no existeSiguiente?)
  - *En el esquema anterior, está comprobado antes de usarlas*

Iteradores

- *Iterador: interno versus externo*
  - ***Iterador externo:*** *es un dato de tipo auxiliar con acceso limitado a la implementación del contenedor, de tal forma que proporciona mecanismos para recorrerlo sin que éstos estén integrados en el contenedor*
    - ***El contenedor ofrece una operación para obtener un iterador para recorrerlo, pero el iterador y sus operaciones NO forman parte del contenedor***
    - *Desventaja: puede requerir exponer, aunque sea parcialmente, la implementación del contenedor*
    - *Ventaja: es posible utilizar simultáneamente varios iteradores sobre el mismo contenedor, cada uno en distinto estado*
    - ***No los vamos a utilizar en esta asignatura, se estudiarán más adelante*** *(patrón iterator)*
  - ***Iterador interno:*** ***el iterador y sus operaciones se añaden al contenedor,*** *sumándose a sus operaciones pero* ***sin modificar las propiedades del resto de las operaciones del contenedor***
    - *Ventaja: el iterador se implementa de forma sencilla, eficiente, y sin exponer la implementación del contenedor*
    - *Desventaja: solo se tiene un iterador para el contenedor*
    - ***Serán los únicos que utilizaremos en esta asignatura***

Ejemplo iterador en PRÁCTICA 1: TAD Agenda

espec agendas
  usa contactos, booleanos
  género agenda *{Los valores del TAD representan colecciones de contactos a las que se pueden añadir elementos de tipo contacto, y de las que se pueden eliminar sus contactos de uno en uno, eliminándose siempre el último contacto añadido de todos los que contenga la agenda}*

operaciones

iniciar: → agenda
{Devuelve una agenda vacía, sin contactos}

añadir: agenda a, contacto c → agenda
{Devuelve la agenda igual a la resultante de añadir un contacto c a la agenda a.}

vacía: agenda a → booleano
{Devuelve verdad si y sólo si la agenda a está vacía}
...

Ejemplo iterador en PRÁCTICA 1: TAD Agenda

borrarUltimo: agenda a → agenda
{Si a no está vacía, devuelve la agenda igual a la resultante de eliminar de a el último contacto añadido a ella. Si a está vacía, devuelve la agenda vacía}

está: agenda a, contacto c → booleano
{Dada una agenda a y un contacto c, devuelve verdad si y sólo si en a hay algún contacto igual a c (en el sentido de la operación iguales del TAD contacto), falso en caso contrario}

. . . ***{le añadimos las operaciones del iterador…}***

Especificación de Agenda con iterador

. . . ***{operaciones del iterador para agenda:}***

iniciarIterador: agenda a → agenda
{Prepara el iterador para que el siguiente contacto a visitar sea el primero (situación de no haber visitado ningún contacto)}
existeSiguiente?: agenda a → booleano
{Devuelve verdad si queda algún contacto por visitar, devuelve falso si ya se ha visitado el último contacto}
parcial siguiente: agenda a → *contacto*
{Devuelve el siguiente contacto a visitar.
Parcial: la operación no está definida si no quedan contactos por visitar (no existeSiguiente?(a)}
parcial avanza: agenda a → agenda
{Prepara el iterador para que se pueda visitar el siguiente contacto.
Parcial: la operación no está definida si no quedan contactos por visitar (no existeSiguiente?(a)}

fespec

Observaciones

- Especificación:

```
parcial siguiente: agenda a → contacto
{Devuelve el siguiente elemento a visitar.
 Parcial: la operación no está definida si no quedan elementos
 por visitar ( no existeSiguiente?(a)}

parcial avanza: agenda a → agenda
{Prepara el iterador para que se pueda visitar el siguiente
 elemento.
 Parcial: la operación no está definida si no quedan elementos
 por visitar ( no existeSiguiente?(a)}
```

- Implementación C++: ambas operaciones juntas

```
bool siguienteYavanza (agenda& a, contacto& e);
{Si existe algún contacto pendiente de visitar, modifica e
con el siguiente contacto a visitar, y además después avanza
el iterador para que a continuación se pueda visitar otro
contacto, y devuelve true. Si no quedaban contactos
pendientes por visitar, devuelve false.}
```

Implementación: fichero agenda.hpp

```
#ifndef AGENDA_HPP
#define AGENDA_HPP
#include "contacto.hpp"
// Inicio interfaz del TAD agenda. Pre-declaración:
. . .
void iniciarIterador (agenda& a); //...
bool existeSiguiente (const agenda& a); //...
bool siguienteYavanza (agenda & a, contacto& c); //...
// Fin interfaz del TAD agenda.
// Declaración:
struct agenda{
    . . .
 friend void iniciarIterador (agenda& a);
 friend bool existeSiguiente (const agenda& a);
 friend bool siguienteYavanza (agenda & a, contacto& c);
 private: // declaración de la representación interna del tipo:
     contacto datos[MAX_AGENDA];
     int total;
     . . . .


};

#endif
```

Añadiremos algo, a la **representación interna** del tipo, **para** gestionar cuál es el ***estado del iterador*** **:**

- Lo que le añadamos, **únicamente lo usarán las operaciones que implementan el iterador** (y **no** deberán usarlo las otras operaciones del contenedor)
- Las operaciones del contenedor **no usarán a las operaciones del iterador** (ni siquiera las que no modifican el contenedor)

Ejemplo de uso del iterador: PRÁCTICA 1

- Las operaciones del iterador **se usarán fuera de la implementación del TAD**
- **Nunca** se debe modificar el contenedor (en este caso, la agenda) mientras se recorre con las operaciones de un iterador

```
// Por ejemplo, en el main de P1:
agenda miagenda;
iniciar(miagenda);
. . . . //crear contactos y añadirlos a la agenda …
contacto c; bool ok;
//Recorrer todos los contactos de la agenda:
//Inicio recorrido: a partir de aquí NO se debe modificar miagenda
iniciarIterador(miagenda);
while (existeSiguiente(miagenda)) {
    ok = siguienteYavanza(miagenda,c);
   //tratar el contacto siguiente obtenido en c,
   //o tratar el error
}
//Fin del recorrido: se puede volver a modificar miagenda
. . .
```

Iteradores

Validez de un iterador:

- *Un iterador tiene validez garantizada* ***mientras no se modifique el contenedor*** *al que referencia*
  - **Sirve para recorrer un contenedor en un estado determinado**
  - El uso de operaciones que modifiquen el contenedor mientras está siendo recorrido, tendrá efectos imprevisibles(*) en el iterador y en general lo harán incorrecto
  - *El que un iterador quede invalidado o no cuando se modifica el contenedor que referencia, depende del tipo de contenedor, de la alteración realizada, y de sus implementaciones.*
    - *Detalles que no debe necesitar conocer el usuario del contenedor o del iterador*
- Los iteradores sirven para implementar (desde fuera del contenedor) **algoritmos de recorrido o búsqueda en el contenedor, pero NO algoritmos que modifiquen el contenedor**

(*) *A menos que se trate de un contenedor especificado de tal forma que todas sus operaciones contemplen el estado del iterador y especifiquen su efecto sobre el.*

Ejemplo de repaso

- Ver ejemplo de TAD conjunto de caracteres (con iterador) en el material de clase:
  - Especificación algebraica
  - Especificación no formal
  - Implementación en:
    - Pseudocódigo
    - C++

Otros Contenedores

- TAD que **permiten almacenar relaciones** entre los elementos:
  - Tipos lineales o Listas:

    Secuencias de elementos de un cierto tipo dispuestos en una dimensión

    e1 e2 e3 e4 e5 e6 e7 … eN
    A C X B A D P … E

    Toda secuencia refleja un orden en sus elementos, resultado de colocarlos en una dimensión

    - Mucha variedad de posibles TAD:
      - Con o sin elementos repetidos
      - Criterios de orden en la secuencia:
        - respecto a los propios datos en los elementos (Ej.: por orden alfabético),
        - respecto a criterios externos a los datos (Ej.: por orden de llegada o inserción)
      - Operaciones típicas: *crear, añadir, esVacía?, quitar, pertenece, tamaño, …*
      - Pueden ser operaciones:
        - guiadas por el orden de los datos o de la secuencia
        - limitadas a determinados extremos de la secuencia
        - relativas al último elemento accedido o insertado
        - relativas a la posición en la secuencia, …
      - Operaciones de acceso y recorrido (iteradores) → según el orden en la secuencia
        - primero, siguiente, esÚltimo?, último, esPrimero?, anterior, …
  - Tipos no lineales: árboles, grafos, etc.

Standard Template Library (STL)

- La Standard Template Library (STL) es una biblioteca de componentes genéricos para C++ en la que se han implementado TADs, estructuras de datos y algoritmos de uso frecuente:
  - Ejemplo de definición e implementación de TADs reutilizables
  - Gran cantidad de componentes genéricos reutilizables que corresponden a TADs fundamentales y avanzados
    - No siempre encontraremos precisamente lo que necesitemos
      - Ejemplo: por tener código general y reutilizable se pueden estar sacrificando más eficiencia y recursos de los aceptables
  - No existe una biblioteca equivalente disponible en todos los lenguajes de programación

https://en.wikipedia.org/wiki/Standard_Template_Library

Standard Template Library (STL)

- Un fragmento de la organización de la STL :

  (extraído de http://www.cplusplus.com/reference/stl/)

Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

Objetivos de la asignatura

- *"El alumno aprenderá a diseñar e implementar los TAD para que sean reutilizables, eficientes y robustos, y a implementarlos garantizando dichas propiedades"*
  - El objetivo de la asignatura NO ES conocer y utilizar la *Standard Template Library*, ni otras bibliotecas similares
  - Salvo que se indique explícitamente lo contrario, para la implementación de TAD **NO** se permitirá utilizar las disponibles en la STL o en bibliotecas similares

- Obviamente, en ninguna prueba de evaluación (prácticas, exámenes, …) se permite entregar implementaciones o trabajo ajeno → FRAUDE ACADÉMICO (PLAGIO)