



UNIVERSIDAD DE GRANADA

**Departamento de Ciencias de la
Computación e Inteligencia Artificial**

Práctica : Contenedores no lineales

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Estructuras de Datos

Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Matemáticas
Doble Grado en Ingeniería Informática y ADE

1.- Introducción

1.1 Objetivo.

El objetivo de esta prácticas es trabajar con las clases set y map.

1.2 Tipos de datos abstractos asociativos

Los TDA no lineales, de forma intuitiva, son aquellos donde no tendría sentido recorrerlo de forma indexada. Entre ellos, los contenedores asociativos son TDAs que permiten almacenar una colección de pares, en la que cada par se conforma de una clave y un valor. No nos importa cómo almacenan los datos sino las capacidades que tienen. Las operaciones más frecuentes de estos contenedores son:

- Añadir un par a la colección
- Eliminar un par de la colección
- Modificar un par existente
- Buscar un valor asociado con una determinada clave.

Las estructuras de datos asociativas que vamos a utilizar y forman parte de la STL son:

- Set
- Map

1.3 Set

La clase set representa un conjunto de elementos que se disponen de manera ordenada y en el que no se repiten elementos. En este caso y a diferencia del TDA map, en la pareja (clave, valor), el valor es siempre nulo. Los datos que insertamos en el set se llaman **claves**.

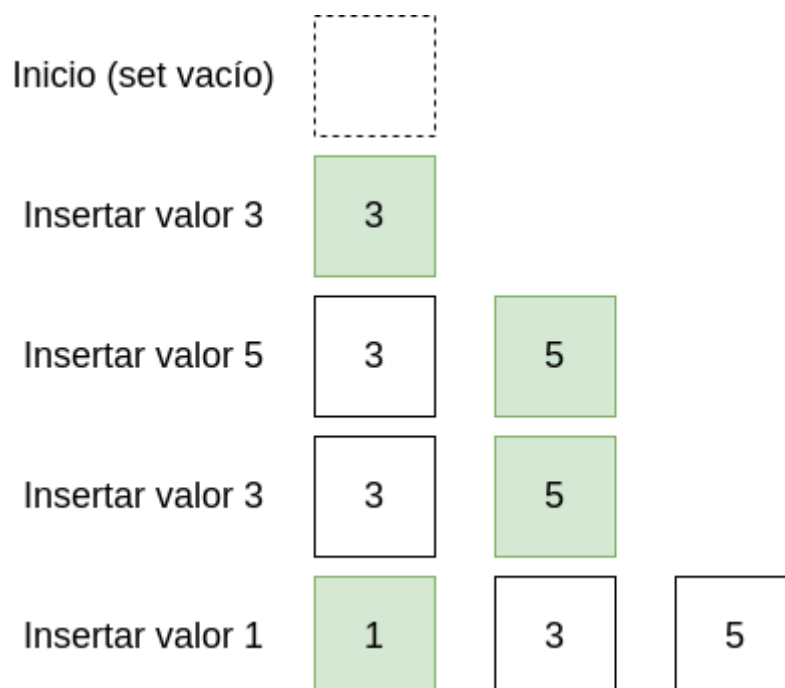


Figura 1: Procedimiento de inserción de valores en un set

Una estructura de tipo set deberá contar, como mínimo, con los siguientes métodos para su correcto funcionamiento:

- Tamaño: size()
- Vacío: empty()
- Borrar: erase()
- Vaciar: clear()
- Insertar: insert()
- Buscar: find()
- Iterador inicio: begin()
- Iterador fin: end()

Se puede consultar la documentación del TDA Set, disponible en la librería standard de C++ (<https://www.cplusplus.com/reference/set/set/>), para ver la funcionalidad básica de un set (aunque dicho TDA implementa adicionalmente algunas funciones que, a pesar de no ser imprescindibles, facilitan el uso de la estructura en muchos contextos).

1.4 Map

Un map está formado por parejas de valores: al primero se lo conoce como **clave**, y al segundo como el **valor** asociado a dicha clave. No permite valores de clave repetidos y se ordena según su clave. Podemos acceder, de forma directa, al valor asociado a la clave a través de la clave, pero no al revés.



Figura 2: Funcionamiento del TDA Map

Una estructura de tipo map deberá contar, como mínimo, con los siguientes métodos para su correcto funcionamiento:

- Tamaño: size()
- Vacío: empty()
- Borrar: erase()
- Vaciar: clear()
- Insertar: insert()
- Buscar: find()
- Iterador inicio: begin()
- Iterador fin: end()
- Obtención de valor dada una clave: operator[]

Se puede consultar la documentación del TDA Map, disponible en la librería standard de C++ (<https://www.cplusplus.com/reference/map/map/>), para ver la funcionalidad básica de un map (aunque dicho TDA implementa adicionalmente algunas funciones

que, a pesar de no ser imprescindibles, facilitan el uso de la estructura en muchos contextos).

1.5 Iteradores

Los iteradores son un T.D.A. que nos permite acceder a los elementos de distintos contenedores de forma secuencial, abstrayéndonos de la representación interna o **estructura de datos** subyacente. Los pasos a seguir para trabajar con iteradores son:

- 1 Iniciar el iterador a la primera posición del contenedor (función begin()).
- 2 Acceder al elemento que apunta (*it, donde it es de tipo iterador)
- 3 Avanzar el iterador al siguiente elemento del contenedor (++it)
- 4 Saber cuando hemos recorrido todos los elementos del contenedor (función end()).

2.- T.D.A Dictionary

El TDA que implementaremos será el TDA Dictionary. Este TDA nos permitirá mantener en nuestro programa un conjunto de palabras. Como no estaremos interesados en almacenar las definiciones de nuestras palabras, sólo los términos, necesitaremos una estructura que nos permita almacenar strings, sin necesidad de almacenar información más compleja. Además, nos interesará que nuestro conjunto tenga dos propiedades específicas sobre sus elementos:

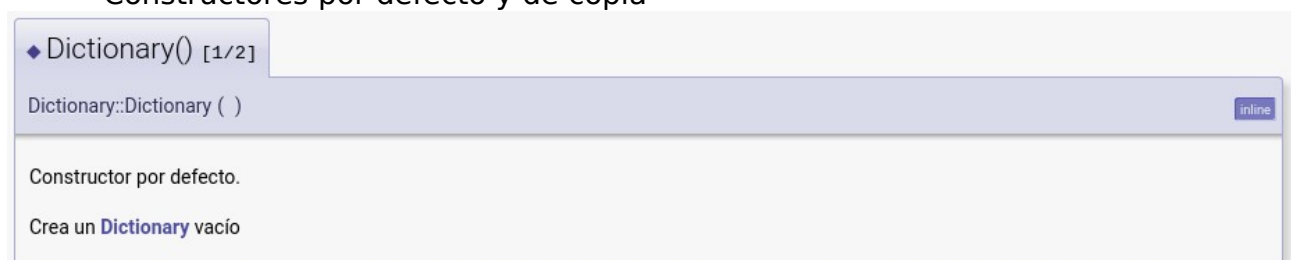
- 1 Los elementos deben estar ordenados por orden alfabético. Tiene mucho sentido que, en un diccionario, los elementos se ordenen de esta manera, y no en otro orden, independientemente del orden en que se hayan insertado
- 2 Los elementos no se deben repetir. Estamos manteniendo una lista de palabras de un idioma, no nos interesa tener palabras repetidas, si no saber simplemente qué palabras pertenecen al mismo.

La estructura de datos set es muy adecuada para lo que necesitamos, teniendo en cuenta las condiciones que hemos impuesto, así que será el contenedor que utilizaremos para nuestro TDA. Tendremos, por tanto, la siguiente información almacenada:

```
class Dictionary {
private:
    set <string> words;
public:
    ...
}
```

E implementaremos la siguiente funcionalidad en nuestro TDA:

- Constructores por defecto y de copia



The screenshot shows an IDE window with the following content:

- A tab at the top labeled "Dictionary() [1/2]".
- A header bar with the text "Dictionary::Dictionary ()" and an "inline" button on the right.
- The main editor area contains the text: "Constructor por defecto." followed by "Crea un Dictionary vacío".

◆ Dictionary() [2/2]

Dictionary::Dictionary (const Dictionary & other) inline

Constructor de copia.

Crea un Dictionary con el mismo contenido que el que se pasa como argumento

Parameters

other Dictionary que se quiere copiar

- Métodos para insertar, consultar la presencia, y borrar un elemento

◆ exists()

bool Dictionary::exists (const string & val) const inline

Indica si una palabra esta en el diccionario o no.

Este método comprueba si una determinada palabra se encuentra o no en el diccionario

Parameters

palabra la palabra que se quiere buscar.

Returns

Booleano indicando si la palabra existe o no en el diccionario

◆ insert()

bool Dictionary::insert (const string & val) inline

Inserta una palabra en el diccionario.

Parameters

val palabra a insertar en el diccionario

Returns

Booleano que indica si la inserción ha tenido éxito. Una palabra se inserta con éxito si no existía previamente en el diccionario

◆ erase()

bool Dictionary::erase (const string & val) inline

Elimina una palabra del diccionario.

Parameters

val Palabra a borrar del diccionario

Returns

Booleano que indica si la palabra se ha borrado del diccionario

- Métodos para consultar el tamaño, consultar si el diccionario está vacío, y limpiar el diccionario

◆ clear()

void Dictionary::clear ()

inline

Limpia el **Dictionary**.

Elimina todas las palabras contenidas en el conjunto

◆ empty()

bool Dictionary::empty () const

inline

Comprueba si el diccionario está vacío.

Returns

true si el diccionario está vacío, false en caso contrario

◆ size()

unsigned int Dictionary::size () const

inline

Tamaño del diccionario.

Returns

Número de palabras guardadas en el diccionario

- Las palabras de una determinada longitud.

◆ wordsOfLength()

vector< string > Dictionary::wordsOfLength (int **length**)

Devuelve las palabras en el diccionario con una longitud dada.

Parameters

length Longitud de las palabras buscadas

Returns

Vector de palabras con la longitud desdeada

- Numero de veces que aparece un determinado carácter en el diccionario.

◆ getOccurrences()

int Dictionary::getOccurrences (const char **c**)

Indica el numero de apariciones de una letra.

Parameters

c letra a buscar.

Returns

Un entero indicando el numero de apariciones.

- Añadir un metodo *void anade(const Dictionary &dic)* a la clase Dictionary que añade al diccionario actual el que se pasa como argumento.

Programas de prueba - palabras_longitud.cpp, apariciones.cpp y union.cpp.

Para probar el funcionamiento de este TDA, implementaremos tres programas de prueba distintos.

El primero de ellos, palabras_longitud.cpp, recibe dos argumentos

- 1 Un fichero con las palabras de un diccionario
- 2 Un entero con la longitud de las palabras que buscamos

Construye un Dictionary con el fichero de las palabras, extrae de dicho diccionario las palabras de la longitud que buscamos y las imprime por pantalla.

El segundo de ellos, apariciones.cpp, recibe dos argumentos

- 1 Un fichero con las palabras de un diccionario
- 2 Un caracter.

Devuelve el numero de veces que ese carácter aparece en el diccionario.

El tercero union.cpp recibe dos argumentos

- 1 Un fichero con las palabras de un diccionario.
- 2 Otro fichero de diccionario.

Calcula la unión y la muestra por pantalla.

3. TDA Guía de teléfonos usando el tipo map de la STL

Se trata de crear el TDA **Guia_Tlf** basándose en el tipo **map** de la STL. La guia de teléfonos contiene como tipo base parejas <nombre, num_teléfono> ambos de tipo string. En los ficheros include/guiatlf.h src/guatlf.cpp se da la clase Guia_Tlf. Se pide completar los comentarios doxygen de todos los métodos y añadir una clase iteradora y tres metodos para:

-Hacer la interseccion de la guia apuntada por this y otra pasada como argumento.

-Modificar el telefono asociado a un nombre.

-Devolver una guia con los telefonos de aquellos nombres que empiecen por un string determinado.

Para probar el funcionamiento de estos tres métodos implementaremos tres programas de prueba.

El primero de ellos, inter.cpp, recibe dos argumentos

- 1 Un fichero con una guía.
- 2 Otro nombre de fichero con otra guía de teléfonos.

Debe construir una nueva guía con la intersección de las dos pasadas por argumento e imprimirla por pantalla.

El siguiente programa de prueba se llama modificar.cpp que modifica el teléfono asociado a un nombre. Recibe tres argumentos.

- 1 Un fichero con una guía.
- 2 El nombre al que le vamos a modificar el teléfono.
- 3 El nuevo teléfono.

Debe buscar el nombre en la guía y modificar el teléfono por el nuevo introducido como argumento. Finalmente debe imprimir la nueva guía por pantalla.

El tercero de los programas de prueba se llama filtro.cpp que recibe dos argumentos:

- 1 El fichero con una guía.
- 2 Un string que es el comienzo de un nombre.

Debe mostrar por pantalla todos los nombres que comiencen por ese string con sus respectivos telefonos.

3.- Práctica a Realizar

3.- Ejercicio 1: Construcción del TDA Dictionary

En el primer ejercicio de la práctica, se propone la implementación del **tipo de dato Dictionary**, que implementa el diccionario de letras que hemos descrito en los apartados anteriores. Para ello, será necesario:

- 1 Dar una especificación del T.D.A Dictionary.
- 2 Definir el conjunto de operaciones necesarias para el funcionamiento del diccionario junto con sus especificaciones.
- 3 Implementar el TDA Dictionary **utilizando como contenedor subyacente un set**. Se podrá utilizar el set que viene implementado en la STL (set). Teniendo en cuenta los principios de **ocultamiento de información**, se recomienda separar la declaración y la implementación de dicha clase (se sugieren los nombres de archivo dictionary.h y dictionary.cpp, situados en la carpeta que les corresponda)
- 4 Se deberán implementar iteradores para poder iterar sobre los diferentes elementos del TDA Dictionary.
- 5 Hacer los programas de prueba palabras_longitud.cpp, apariciones.cpp y union.cpp

3.- Ejercicio 2: TDA Guia_Tlf

En el segundo ejercicio de la práctica, se da la implementación del **tipo de dato Guia_Tlf**, que implementa la guía de teléfonos que hemos descrito en los apartados anteriores. Será necesario:

- 6 Repasar los comentarios doxygen de todos los metodos.
- 7 Realizar los tres métodos indicados anteriormente para hacer la intersección, la modificación de un telefono, y el filtrado de una guía .
- 8 Hacer los correspondientes tres programas de prueba para probar cada uno de los tres métodos añadidos a este TDA.
- 9 Se deberán implementar iteradores para poder iterar sobre los diferentes elementos del TDA Guia_Tlf.

3.- Ejercicio 3: Hacer la página principal de la documentación común para ambos TDAs,

4.- Documentación y entrega

Toda la documentación de la práctica se incluirá en el propio Doxygen generado, para ello se utilizarán tanto las directivas Doxygen de los archivos .h y .cpp como los archivos .dox incluidos en la carpeta doc.

1. La entrega no es obligatoria pero si se desea entregar hacer un .zip con lo siguiente:

- Se dispone de la siguiente estructura de ficheros:
 - /

- estudiante/ (Aquí irá todo lo que desarrolle el alumno)
 - doc/ (Imágenes y documentos extra para Doxygen)
 - include/ (Archivos cabecera)
 - src/ (Archivos fuente)
- data/ (Datos de ejemplo)
- CMakeList.txt (Instrucciones CMake)
- Doxyfile.in (Archivo de configuración de Doxygen)

2. Elaboración y puntuación

- La práctica se realizará POR PAREJAS. Los nombres completos se incluirán en la descripción de la entrega en Prado. Cualquiera de los integrantes de la pareja puede subir el archivo a Prado, pero SOLO UNO.
- Se trata de una práctica voluntaria NO PUNTUABLE.
- La fecha límite de entrega aparecerá en la subida a Prado.