UNIVERSIDAD DE GRANADA

Práctica 3

Implementación de TDAs sobre contenedores STL.

Estructura de Datos

Curso 2025/2026

Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

30 de noviembre de 2025

Índice

1 Introducción

En esta práctica vamos a desarrollar un sistema de gestión de contactos personales que permitirá almacenar información relevante sobre cada persona (nombre, teléfonos, correo electrónico, etiquetas, etc.) y consultarla de manera estructurada y eficiente. El objetivo fundamental es diseñar dos tipos de datos abstractos (T.D.A.): uno que represente un contacto individual con sus distintos atributos, y otro que gestione un conjunto de contactos permitiendo realizar operaciones avanzadas de búsqueda, inserción, eliminación y filtrado por criterios.

A diferencia de la práctica anterior, en esta ocasión se hará uso de los contenedores de la **librería estándar de C++ (STL)**, específicamente de `map`, `multimap` y `set`. Estos contenedores permitirán trabajar con información ordenada automáticamente, buscar contactos mediante diferentes claves y gestionar relaciones del tipo uno-a-varios (por ejemplo, varias personas asociadas a la misma etiqueta).

El sistema que se desarrollará permitirá representar una agenda de contactos personales, profesionales o académicos. Se pretende que el estudiante practique los conceptos fundamentales de diseño de T.D.A., así como el uso avanzado de los contenedores asociativos de la STL para gestionar información compleja. Se prestará especial atención a la documentación, a la función de abstracción, al invariante de representación y al uso adecuado de los tipos plantilla (*templates*) proporcionados por la STL.

En esta práctica se reforzarán los conceptos de abstracción de datos, especificación modular, encapsulamiento e independencia entre representación e interfaz. Además, se introduce el diseño de tipos de datos que hacen uso de estructuras dinámicas ya implementadas, priorizando la correcta integración y utilización de los contenedores de la STL frente a la implementación manual de estructuras. **En esta práctica está permitido y recomendado el uso de `map`, `multimap` y `set`, así como otros contenedores de la STL cuando sea necesario**.

2 Objetivos y requisitos

Los objetivos de este guion de prácticas son los siguientes:

1. Comprender y aplicar los conceptos fundamentales de abstracción de datos utilizando los contenedores de la STL.
2. Diseñar y documentar un tipo de dato abstracto (T.D.A.) que represente entidades del mundo real (contactos personales).
3. Practicar el uso de los contenedores asociativos de la STL, en particular `map`, `multimap` y `set`, aplicándolos a un problema real.
4. Profundizar en los conceptos relacionados con la especificación, representación, función de abstracción e invariante de representación de un T.D.A. construido sobre estructuras dinámicas ya existentes.
5. Aplicar estos conocimientos al desarrollo de un sistema de gestión de contactos, incorporando búsquedas eficientes, filtrado por etiquetas y organización de la información mediante claves múltiples.
6. Familiarizarse con la correcta documentación del código mediante `doxygen`, siguiendo las convenciones de especificación de operaciones.

Los requisitos previos para poder realizar esta práctica son:

1. Haber completado la Práctica 2 y comprender los conceptos de abstracción, especificación, representación y diseño modular de T.D.A.

2. Tener conocimientos básicos del uso de plantillas (*templates*) en C++.
3. Haber estudiado el seminario impartido en prácticas: STL en C++.
4. Conocer el funcionamiento de los contenedores asociativos de la STL y sus operaciones fundamentales: `insert`, `find`, `erase`, `equal_range`, `lower_bound` y `upper_bound`.
5. Conocer las nociones básicas de iteradores y su uso en bucles para recorrer contenedores.

3 Documentación

El objetivo fundamental de un programador es que los tipos de datos abstractos (T.D.A.) que desarrolle puedan ser reutilizados por él mismo o por otros programadores. Para que esto sea posible, los módulos donde se implementan los T.D.A. deben estar correctamente documentados. Esta documentación debe distinguir entre dos niveles claramente diferenciados:

1. **Especificación**: documento que describe la parte pública del T.D.A., es decir, las operaciones disponibles, su sintaxis y su comportamiento. Permite utilizar el tipo abstracto sin conocer los detalles internos de su implementación.
2. **Implementación**: documento que recoge los detalles internos del T.D.A., su representación concreta y las decisiones de diseño asociadas. Resulta útil para realizar mantenimiento, ampliaciones o detección de errores.

3.1 Definición del T.D.A.

La definición de un T.D.A. debe presentarse en lenguaje natural, de forma clara y precisa. En ella se describe qué representa cada instancia del tipo, qué información encapsula y cuál es su dominio. En esta práctica, por ejemplo, se deberá especificar qué es un `Contacto` y qué es una `AgendaContactos`, indicando qué representa cada uno y qué información maneja.

Se recordará que la definición debe centrarse en el *qué* y no en el *cómo*: el usuario del T.D.A. no necesita conocer su representación interna, únicamente su comportamiento externo.

3.2 Operaciones

Cada operación definida en un T.D.A. representa una función, y su especificación debe documentarse adecuadamente. Se recomienda utilizar el sistema de anotaciones `doxygen` para describir cada una de ellas. La documentación de una operación debe incluir:

1. **@brief:** Breve descripción de la funcionalidad.
2. **@param:** Explicación de cada parámetro, indicando si es de entrada, salida o entrada/salida.
3. **@pre:** Condiciones que deben cumplirse antes de ejecutar la operación.
4. **@return:** Valor devuelto por la función (si lo hubiera).
5. **@post:** Condiciones garantizadas tras la ejecución de la operación.

Estas anotaciones deberán incluirse en todas las funciones públicas de los T.D.A. `Contacto` y `AgendaContactos`, así como en cualquier operación auxiliar que se considere relevante.

3.3 Implementación del T.D.A.

Para implementar un T.D.A. es necesario escoger primero una representación interna adecuada. En esta práctica, dicha representación podrá apoyarse en los contenedores de la STL, como `map`, `multimap` y `set`, respetando los criterios de encapsulamiento y ocultación de la información.

La representación seleccionada constituye el **tipo de representación** (*tipo rep*), y deberá estar acompañada de:

1. **Invariante de representación**: conjunto de condiciones que deben cumplir los valores del tipo rep para ser considerados válidos. En esta práctica, por ejemplo, deberá garantizarse que no existan contactos duplicados, que las etiquetas se almacenen sin repeticiones, etc.

2. **Función de abstracción**: describe cómo se interpreta el tipo rep para obtener el valor abstracto que representa. Indica, por ejemplo, cómo un `map<string, Contacto>` representa conceptualmente una agenda ordenada de contactos.

La función de abstracción debe relacionar claramente el nivel interno (contenedores y estructuras internas) con el nivel abstracto (concepto de agenda). Asimismo, el invariante debe asegurar que todas las operaciones mantienen la consistencia del T.D.A. tras su ejecución.

Toda esta información deberá incluirse en la documentación interna de los módulos `contacto.cpp` y `agendacontactos.cpp`, siguiendo las convenciones explicadas en el guion de la asignatura.

4 Ejercicio

En esta práctica el alumno deberá desarrollar un programa que gestione información mediante tipos de datos abstractos. Para ello, se propone la implementación de dos T.D.A. que trabajen de forma conjunta: uno para representar contactos individuales y otro para gestionarlos de manera estructurada utilizando los contenedores de la STL.

Los pasos a realizar son los siguientes:

1. Diseñar un tipo de dato abstracto que represente un contacto, con varios atributos relevantes (nombre, teléfonos, correos y etiquetas).

2. Diseñar un segundo tipo abstracto que almacene múltiples contactos del tipo anterior y permita su gestión: inserción, eliminación, búsqueda por nombre, filtrado por etiquetas, etc.

3. Proporcionar operaciones adicionales que permitan analizar y consultar información derivada (por ejemplo, listar contactos por etiqueta, buscar coincidencias, exportar información, etc.).

Para cada tipo de dato abstracto, el alumno deberá:

1. Establecer una **definición** clara en lenguaje natural.

2. Determinar y justificar diferentes opciones posibles para el **tipo de representación** (*tipo rep*), indicando qué contenedores de la STL podrían utilizarse.

3. Escoger una representación y justificar su elección.

4. Establecer el **invariante de representación** y la **función de abstracción**.

5. Implementar las operaciones necesarias de forma modular.

6. Documentar adecuadamente todo el código con `doxygen`.

Además, se deberá desarrollar un programa principal que pruebe todos los módulos y permita validar el funcionamiento de los T.D.A. implementados. Este programa debe construirse de forma flexible, permitiendo modificar los datos de entrada sin cambiar el código fuente.

4.1 Módulos a desarrollar

El desarrollo de esta práctica deberá estructurarse en varios módulos claramente diferenciados. Como mínimo, el alumno deberá implementar:

- Un módulo para el primer tipo de dato abstracto (`contacto.h` y `contacto.cpp`), que contendrá la definición y funcionalidad de los contactos individuales.
- Un módulo para el segundo tipo de dato abstracto (`agendacontactos.h` y `agendacontactos.cpp`), que gestionará un conjunto de objetos del tipo anterior mediante los contenedores de la STL.
- Un programa principal (`main.cpp`) que permita probar las funcionalidades desarrolladas.

Cada módulo deberá tener su propia cabecera (`.h`) y su implementación (`.cpp`). Es posible añadir módulos adicionales si se considera necesario para mejorar la organización del código o incorporar funcionalidades extra.

Además, el proyecto deberá incluir un archivo `Makefile` para facilitar la compilación de todos los componentes de forma automatizada. La estructura del proyecto debe seguir el mismo modelo mostrado en clase.

Todos los ficheros fuente, incluyendo cabeceras, implementaciones, pruebas y documentación, deberán entregarse de forma organizada y limpia. No se incluirán ficheros objeto (`.o`), ejecutables ni carpetas innecesarias.

4.2 Compilación y Makefile

El proyecto deberá incluir un archivo `Makefile` que permita compilar automáticamente el código fuente y generar la documentación del proyecto. Las reglas mínimas exigidas son:

- `make`: compila el programa principal.
- `make clean`: elimina ficheros objeto y ejecutables.
- `make doc`: genera la documentación con `doxygen`, usando el fichero `doc/Doxyfile`.

El `Makefile` deberá compilar el proyecto respetando la estructura de carpetas indicada, generando los archivos objeto en la carpeta `build/` y utilizando la carpeta `include/` para las cabeceras. El alumno puede organizar sus reglas como considere adecuado, siempre que cumpla con las funcionalidades anteriores.

5 Entrega de la práctica

La entrega de la práctica se realizará a través de la plataforma de la asignatura, dentro de la actividad habilitada específicamente para ello. El alumno deberá comprimir todo el contenido del proyecto en un único fichero con formato `.zip`, con el nombre:

`apellidos_nombre_p3.zip`

No se admitirán prácticas que no respeten esta estructura de entrega, ni aquellas que incluyan ficheros generados automáticamente como ejecutables, objetos o temporales.

Importante: Asegúrese de que la práctica se puede compilar correctamente ejecutando el comando:

```
make
```

y que se ejecuta sin errores con:

```
./programa
```

La práctica deberá estar correctamente documentada con `doxygen` y acompañada del correspondiente archivo `Makefile`.

5.1 Estructura de carpetas

El proyecto deberá organizarse siguiendo la siguiente estructura:

```
apellidos_nombre_p3/
|
|-- include/
|   |-- contacto.h
|   |-- agendacontactos.h
|
|-- src/
|   |-- contacto.cpp
|   |-- agendacontactos.cpp
|   |-- main.cpp
|
|-- build/              (carpeta generada automáticamente, no se entrega)
|
|-- datos/
|   |-- agenda_contactos.txt
|
|-- doc/
|   |-- Doxyfile
|   |-- html/           (generado automáticamente por doxygen)
|
|-- Makefile
|-- README.md (opcional)
```

- El código fuente debe organizarse en las carpetas `include/` y `src/`.
- La carpeta `datos/` debe incluir los ficheros de entrada necesarios para probar la práctica.
- La carpeta `doc/` debe contener el archivo de configuración `Doxyfile` y la documentación generada por `doxygen`.
- El `Makefile` debe compilar el proyecto desde esa estructura y permitir generar la documentación con `make doc`.
- Todos los archivos objeto (`.o`) se generarán en la carpeta `build/`. Estos archivos no deberán entregarse, y deberán eliminarse con la regla `make clean`.

6 Rúbrica de evaluación

La práctica se calificará sobre un total de **10 puntos**, distribuidos según los siguientes criterios:

Criterio	Puntuación	Descripción
Implementación del T.D.A. Contacto	2 puntos	- Correcta representación del TDA individual. - Gestión adecuada de teléfonos, correos y etiquetas. - Métodos bien diseñados y funcionales.
Implementación del T.D.A. AgendaContactos	2 puntos	- Uso apropiado de `map`, `multimap` y `set`. - Gestión correcta de colecciones de contactos. - Métodos de búsqueda, inserción, eliminación y filtrado bien implementados.
Organización y estructura del código	1.5 puntos	- Separación en módulos `include/`, `src/`, etc. - Código legible, estructurado y limpio. - Uso correcto del `Makefile`.
Documentación técnica del código y T.D.A.	2 puntos	- Uso correcto de `doxygen` (`@brief`, `@param`, etc.). - Inclusión del invariante de representación y función de abstracción. - Generación de documentación en HTML.
Programa principal de pruebas	1 punto	- Incluye un `main.cpp` funcional. - Permite cargar datos, probar métodos y mostrar resultados.
Estructura de proyecto y entrega	0,5 puntos	- Proyecto organizado correctamente. - Entregado comprimido en un único fichero `.zip`. - Estructura de carpetas respetada.
Calidad técnica y restricciones	1 punto	- Uso adecuado y justificado de contenedores STL. - Código eficiente, sin errores de compilación o ejecución. - Mantenimiento del invariante y consistencia de datos.

Cuadro 1: Rúbrica de evaluación de la práctica