PRÁCTICA TEMAS 2 Y 3
Algoritmos y Estructuras de Datos I
Curso: 2025/26

Autores:

Nombre completo	Subgrupo	DNI
María Almarcha Lorca	2.2	34332923H
Fadwa Abbaq El Gharras	2.2	30850969B

ÍNDICE

1. ENVÍOS A MOOSHAK

Nº de problema	Nº de envío a Mooshak	Propietario de la cuenta del envío	Usuario
001	6715	Almarcha Lorca, María	B16
002	6716	Almarcha Lorca, María	B16
003	6717	Almarcha Lorca, María	B16
004	6718	Almarcha Lorca, María	B16
005	6719	Almarcha Lorca, María	B16
006	6039	Almarcha Lorca, María	B16
200	6043	Almarcha Lorca, María	B16
300	6051	Almarcha Lorca, María	B16
301	6052	Almarcha Lorca, María	B16
302	6053	Almarcha Lorca, María	B16

2. CLASES DEFINIDAS Y DEPENDENCIAS

El sistema se estructura en torno a las siguientes clases principales, diseñadas para encapsular la información de los mensajes y la gestión de la estructura de datos:

- **Fecha:** Clase base independiente. Gestiona la lógica temporal (día, mes, año, hora) y encapsula las comparaciones.

- **Cuac:** Representa un mensaje individual. Depende de **Fecha** (Composición) para almacenar el momento de envío.

- **Par:** Clase auxiliar utilizada dentro de la tabla hash. Agrupa a un usuario con su lista de mensajes. Depende de **Cuac** (Agregación). Se ha declarado como class friend de **TablaHash**, para que esta pueda acceder a sus datos privados.

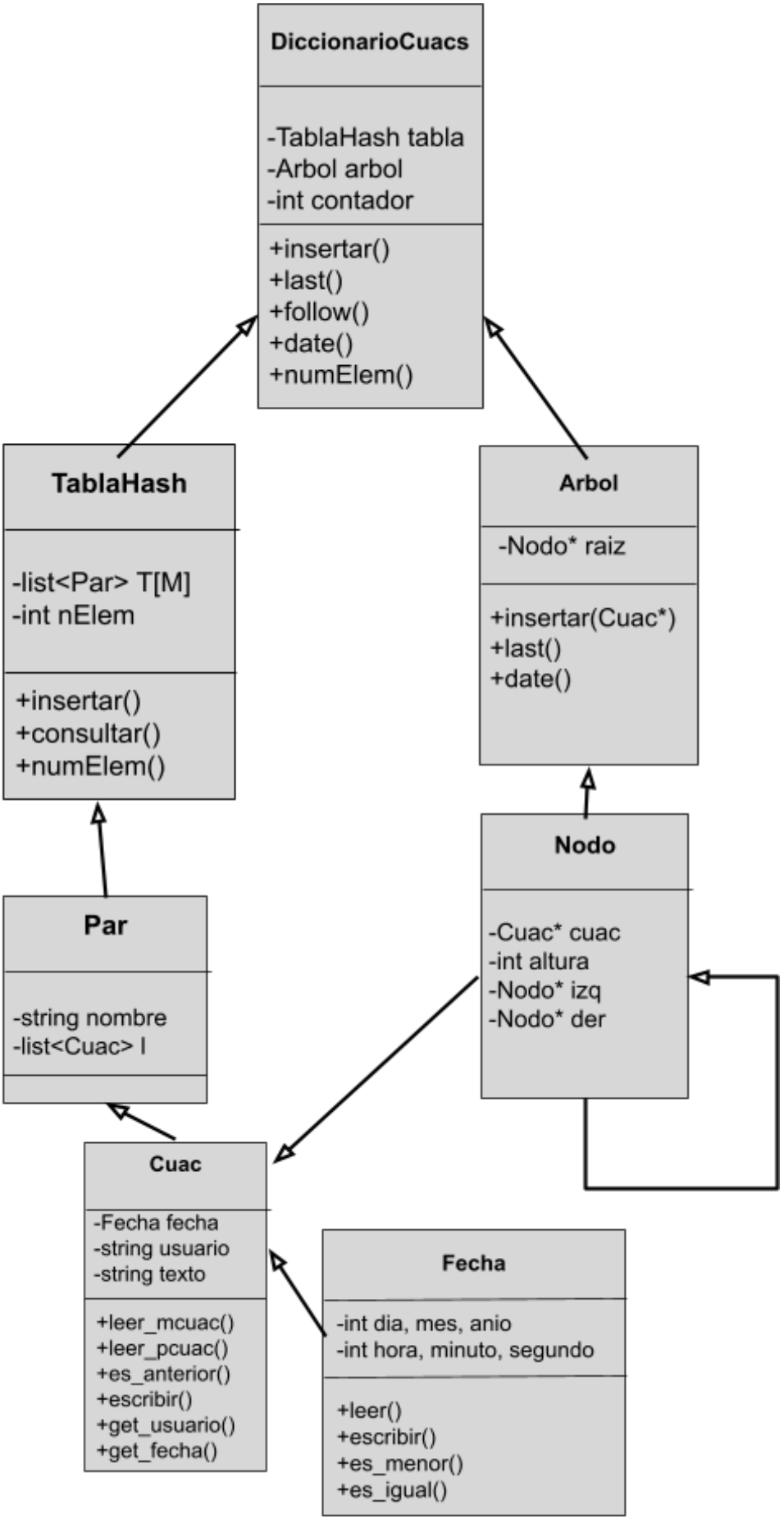
- **TablaHash:** Estructura de datos principal para indexar usuarios. Depende de **Par** (contiene un array de listas de Par).

- **Nodo:** Clase auxiliar para el árbol, contiene punteros a hijos y el dato. Se ha declarado como class friend de **Arbol**, para que esta pueda acceder a sus datos privados.

- **Arbol:** Clase que gestiona la estructura AVL. Depende de **nodo** y **Cuac.**

- **DiccionarioCuac:** Clase controladora que gestiona las operaciones de alto nivel. Depende de **TablaHash** y **Arbol** (Composición), delegando en ella el almacenamiento.

Diagrama de Clases:



3. MÓDULOS IMPLEMENTADOS, ARCHIVOS DE CADA MÓDULO

Siguiendo los principios de programación modular, el código se ha dividido en los siguientes ficheros para favorecer la encapsulación y facilitar la **compilación**:

MÓDULO	ARCHIVOS	Descripción y Dependencias
Fecha	**Fecha.h** **Fecha.cpp**	**Dominio Base:** Define la gestión del tiempo. **Dependencias**: Ninguna (biblioteca estándar).
Cuac	**Cuac.h** **Cuac.cpp**	**Dominio principal:** Define el mensaje, sus datos y su lógica de ordenación. **Dependencias**: Incluye Fecha.h .
TablaHash	**TablaHash.h** **TablaHash.cpp**	**Estructura de datos (Usuarios):** Implementa la tabla hash y define la clase auxiliar Par. **Dependencias**: Incluye Cuac.h (que a su vez incluye Fecha.h).
Nodo	**Nodo.h**	**Auxiliar:** Nodo del árbol AVL. **Dependencias:** Incluye Cuac.h.
Árbol	**Arbol.h** **Arbol.cpp**	**Estructura de datos (Fechas):** Árbol AVL para búsquedas por rango. **Dependencias**: Incluye Nodo.h (que a su vez incluye Cuac.h).
Diccionario	**Diccionario.h** **Diccionario.cpp**	**Controlador (Fachada):** Gestiona las estructuras. **Dependencias**: Incluye TablaHash.h, Arbol.h , Cuac.h y Fecha.h .
Main	**main.cpp**	**Interfaz:** Intérprete de comandos. **Dependencias**: Incluye Diccionario.h .

4. TABLA DE DISPERSIÓN UTILIZADA

Para resolver el problema de la gestión eficiente de usuarios (Problema 200) y optimizar la búsqueda por nombre (follow), se ha implementado una tabla de dispersión con las siguientes características:

4.1 Función de Dispersión

Se utiliza una función polinómica iterativa que procesa la cadena de caracteres del nombre de usuario:

- **Algoritmo**: La función recorre la cadena carácter a carácter, actualizando el valor hash mediante la fórmula: $hash = (hash * 67 + carácter) \, mod \, M$
- **Justificación**: El uso del multiplicador primo 67 ayuda a dispersar las claves y reducir colisiones en cadenas similares, además de hacer que la posición de cada letra de la cadena influya en el resultado.

4.2 Tamaño de la Tabla (M)

Se ha elegido **1009**.

- **Justificación**: Es un número primo suficientemente grande para los casos de prueba esperados, lo que minimiza las colisiones y asegura una distribución uniforme por los índices al aplicar la operación módulo. Asegurando un `O(1)`.

4.3 Resolución de Colisiones

Se ha optado por la **Dispersión Abierta** con **Encadenamiento Separado**

- **Implementación**: La tabla `T[M]` es un array de listas enlazadas (`std::list<Par>`), ya que cada posición de la tabla es una lista (cubeta).
- **Estrategia**: Cuando dos usuarios diferentes (Par) tienen el mismo valor hash, se almacenan en la misma lista en esa posición del array. Esto garantiza que nunca se rechacen inserciones por falta de espacio en la tabla.
- **Justificación frente a Dispersión Cerrada:** se descartó la dispersión cerrada, que almacena los elementos directamente en el array, porque su rendimiento se degrada cuando el factor de la tabla aumenta. Con la dispersión abierta, la tabla puede almacenar más de M elementos sin necesitar redimensionar ni sufrir problemas, además de no necesitar hacer redispersión.

5. ÁRBOL UTILIZADO

Para resolver los problemas de ordenación cronológica y búsqueda por rangos de fechas (Problemas 300, 301 y 302), se ha seleccionado e implementado la estructura del árbol.

5.1. Tipo de Árbol: AVL

Se ha optado por implementar un árbol AVL.

- **Justificación de Eficiencia:** Como los mensajes (`cuacs`) llegan ordenados cronológicamente, un árbol binario normal crecería solo hacia la derecha, convirtiendose en una lista lenta **`O(N)`**. Hemos elegido el **Árbol AVL** porque aplica **rotaciones automáticas** para evitar esto. Así, el árbol se mantiene siempre equilibrado, garantizando búsquedas y listados **`O(log N)`** sin importar el orden de llegada de los datos.
- **Implementación:** Se ha diseñado una clase Nodo que almacena un puntero al mensaje (`Cuac*`), la altura del subárbol y punteros a los hijos. La clase Arbol gestiona el equilibrio después de cada inserción.

5.2. Estrategia de Almacenamiento: Punteros para no duplicar

Una de las decisiones más importantes del diseño ha sido **no guardar copias** de los mensajes en el árbol.

- Los mensajes (`Cuac`) están como tal en la Tabla Hash.
- El **Árbol AVL** almacena solamente **punteros** (`Cuac*`) a esos objetos originales.
- **Justificación:** si no lo hiciésemos así, gastaríamos el doble de memoria (al guardar en la tabla y en el árbol). Con punteros, el árbol es mucho más rápido.

5.3. Criterio de Ordenación

Para cumplir con el requisito de listar siempre del más reciente al más antiguo, hemos organizado el árbol usando la función `Cuac::es_anterior`.

- **Subárbol Izquierdo:** Mensajes más recientes.
- **Subárbol Derecho:** Mensajes más antiguos.

Al recorrer el árbol en orden normal (Izquierda → Raíz → Derecha), los mensajes salen ordenados cronológicamente, sin tener que reordenarlos después.

5.4. Eficiencia de las Operaciones

- **Inserción (mcuac/pcuac):** Complejidad **`O(log N)`** gracias al balanceo AVL.
- **Consulta last N:** No recorremos todo el árbol. Empezamos por la izquierda (los recientes) y en cuanto hemos impreso los **N** mensajes que nos piden, paramos.
- **Consulta date (Poda):** Para no mirar todos los nodos:

- Si la fecha del nodo actual es posterior al fin del rango, no se busca en el subárbol izquierdo (mensajes aún más nuevos).
- Si la fecha del nodo actual es anterior al inicio del rango, no se busca en el subárbol derecho (mensajes aún más antiguos).
- Esto convierte una búsqueda que sería lenta `O(log N)`, en una búsqueda muy rápida `O(log N + K)`, porque solo visitamos los nodos que realmente nos sirven.

5.5 Alternativas consideradas y descartadas

La elección del árbol AVL se realizó tras descartar las siguientes estructuras:

- **Árbol Trie:** son buenos para claves que comparten prefijos, pero para nuestro caso era ineficiente. También sería ineficiente en memoria, por tener muchos nodos vacíos y la búsqueda por rango tampoco nos aportada una gran ventaja.
- **Árbol B:** dado que estos árboles destacan por optimizar el acceso a disco y agrupar muchas claves en un solo nodo, como nuestro Quacker se ejecuta en la memoria principal, nos suponía más trabajo que ventajas.

6. VARIABLES GLOBALES UTILIZADAS

Hemos definido 3 variables globales:

En **main.cpp:**

1. **`DiccionarioCuacs dir`:**
   Es el diccionario, guarda todos los usuarios y todos los mensajes. Se define globalmente para usarlo en el intérprete de comandos (`procesar_mcuac`, `procesar_follow`, etc) sin necesidad de pasarla constantemente por referencia.

2. **`Cuac actual`:**
   Sirve para guardar el último mensaje que hemos usado. Necesaria para `procesar_tag`.

En **TablaHash.h:**

3. **`const int M`:**
   Constante global que fija el tamaño de la tabla para todo el programa.

7. OTRAS DECISIONES DEL DISEÑO RELEVANTES

Estructura de la Tabla Hash:

Aunque para el árbol hemos usado nodos y punteros, para la tabla hemos decidido utilizar la clase `std::list<Cuac>` para gestionar colisiones y agrupar los mensajes de cada usuario.

- **Justificación**: esto nos ha facilitado gestionar el comando follow, permitiendo recorrer los mensajes de cada usuario de forma rápida y sencilla, al ser una lista doblemente enlazada.

Optimización de mensajes:

Para recorrer los mensajes predefinidos hemos usado un array **string puac[]**, para no tener que buscar entre todos los mensajes.

- **Justificación**: esto nos permite buscar los mensajes por índice **puac[n-1]**, consiguiendo un **`O(1)`**.

Friend class:

En el diseño del Árbol, hemos declarado la clase Nodo, para proteger la integridad de sus datos hemos declarado la clase Árbol dentro de Nodo como amiga, para poder acceder a sus variables privadas. También hemos usado esto en la Tablas Hash con Par.

- **Justificación**: en el Árbol, necesitamos los punteros de los nodos todo el tiempo, para usar izq, der… Haciéndolo así, el código queda más limpio y eficiente.

Destructores:

Al borrar el árbol, hemos implementado un destructor hacia la clase Nodo.

- **Justificación**: al ejecutar **`delete raíz`** en la clase Árbol, el destructor del nodo raíz llamada al destructor de la clase Nodo (delete de sus hijos (izq y der)), propagando la eliminación hasta las hojas.

Getters en Cuac:

Hemos implementado los métodos **`get_usuario()`** y **`get_fecha()`**.

- **Justificación**: para poder acceder a los valores del usuario y de la fecha del Cuac en la tabla y el Árbol sin ponerlos en público, hemos creado estas dos funciones que devuelven el valor de estas variables privadas sin necesidad de cambiar su visibilidad. (Lo hemos dado en POO).

Estrategia de Compilación (Makefile):

Se ha implementado un archivo Makefile para automatizar la construcción del proyecto.

- Se utiliza **compilación separada**: cada módulo se compila individualmente a código objeto (.o) y finalmente se juntan para generar el ejecutable.
- Esto optimiza el tiempo y permite ver de una forma mucho más clara las dependencias entre los módulos.

8. RESUMEN DEL TIEMPO DEDICADO A CADA APARTADO

Problemas básicos (001-004):
Para estos problemas iniciales no hubo tantas complicaciones y fluyó la mayor parte del tiempo sin problemas. El tiempo se dedicó principalmente al diseño correcto de las clases base (Fecha, Cuac) y a familiarizarnos un poco con el formato de entrada/salida del juez. Estimamos que tardamos alrededor de unas 6-8 horas o lo que es lo mismo, una media de **2 horas** por ejercicio.

Intérprete (005):
Para la estructura del main, el bucle de lectura de los comandos y las funciones del intérprete para procesar tardamos **2 horas**. No fue de los ejercicios más complicados, pero sí que nos supuso tiempo debido a su importancia en el proyecto.

Listas (006):
La implementación de la clase DiccionarioCuacs usando la lista generó bastante confusión, aquí comenzamos a entrar en la complejidad del proyecto, por el uso de las iteraciones para insertar en las posiciones correspondientes, recorriendo desde el principio hasta el final de la lista. Esto nos supuso unas **5 horas**.

Tabla Hash (200):
Diseño de la función de dispersión y gestión de las colisiones. Gran parte del tiempo dedicado a este ejercicio se dedicó a entender cómo funcionaban las tablas, elegir cual usar y ajustar el tamaño de esta. También se nos fue tiempo entendiendo el uso de los atributos de la clase Par en esta. En total, este apartado nos tomó **6 horas.**

Árbol AVL (300):
Sin ninguna duda, este apartado fue uno de los que nos supusieron un mayor reto en este proyecto. La implementación del Árbol AVL requirió numerosas horas tanto para entender y codificar las rotaciones simples y dobles (RSI, RSD, RDD, RDI), como para gestionar correctamente los punteros o implementar la lógica de poda y garantizar eficiencia. Por estos motivos, debimos demorarnos unas **8 horas** aproximadamente.

Función Last (301):
Este ejercicio fue más llevadero, pero no por ello sencillo. Teníamos que completar la función last que habíamos definido en el ejercicio anterior siendo cuidadosas con las condiciones (if) para las fechas, evitando así recorrer subárboles innecesarios para una mejor eficiencia. Tardamos alrededor de **1 hora**.

Prueba global (302):
Al llegar a este ejercicio nos dimos cuenta de que nos habíamos saltado un paso, separar el código en módulos desde el ejercicio 006. Este ejercicio nos sirvió para volver a entregar todos los ejercicios anteriores en módulos, actualizando cada uno de ellos con el siguiente.

Por suerte, no tuvimos que cambiar código, ya que desde el primer momento nos preocupamos por garantizar la mejor eficiencia posible del programa. Simplemente creamos los .h y los .cpp y el Makefile. Este último fue sencillo de entender, ya que usamos una estructura básica pero funcional.

Pusimos todos los `#include` correspondientes, que nos ayudaron a ver de manera mucho más clara la dependencia entre las clases implementadas. El tiempo total fue de **2 horas y media.**