Entrega 2: Segundo Reto de Informática II

1st Daniela Escobar Velandia
Dpto. de Ingeniería Electrónica y Telecomunicaciones
Universidad de Antioquia
Medellín, Colombia
daniela.escobarv@udea.edu.co

2nd Yonathan López Mejía
Dpto. de Ingeniería Electrónica y Telecomunicaciones
Universidad de Antioquia
Medellin, Colombia
harley.lopez@udea.edu.co

***Abstract*—Este informe presenta la segunda fase del desarrollo de UdeAStay, una plataforma digital orientada a la gestión de estadías hogareñas en el departamento de Antioquia. A partir del análisis del problema y el diseño estructurado del sistema, se implementan cinco clases principales que modelan las interacciones clave entre los actores del sistema.**

La solución se construye desarrollando estructuras de datos propias como listas simplemente enlazadas y tablas hash con resolución por encadenamiento, empleando el algoritmo djb2. También se integran mecanismos de lectura desde archivos externos y validación de datos para asegurar la integridad de las relaciones entre entidades.

El documento incluye la descripción de las estructuras, funciones y métodos creados, los archivos utilizados como entrada, ejemplos de salida del sistema, y resultados de complejidad computacional. El sistema es una implementación funcional, organizada y escalable que sienta las bases para futuras ampliaciones del proyecto.

***Keywords*—Gestión de reservas, estructuras de datos personalizadas, listas enlazadas, tablas hash.**

I. Introducción

El crecimiento del turismo local ha impulsado la creación de nuevas alternativas de alojamiento que promueven una experiencia más cercana con la cultura regional. En este contexto surge UdeAStay, una propuesta de plataforma digital orientada a la gestión de estadías hogareñas en el departamento de Antioquia. El presente documento aborda la segunda fase de desarrollo del sistema: que incluye el análisis del problema, el diseño de la solución estructurada, la implementación del sistema que inlcluye las funciones y métodos creados, y finalmente los resultados de las iteraciones de una función y su complejidad computacional.

En esta propuesta se modelan cinco clases principales —Anfitrion, Huesped, Alojamiento, Reserva y Fecha— con atributos y comportamientos definidos que permiten representar las interacciones reales entre usuarios. Uno de los principales desafíos del desarrollo es la restricción de no utilizar herencia ni bibliotecas estándar, lo que obliga a un diseño cuidadoso, enfocado en la eficiencia del manejo de memoria y la representación precisa de relaciones entre objetos.

Para ello, se emplean estructuras de datos propias, como listas simplemente enlazadas para almacenar alojamientos o reservas de cada entidad, y tablas hash implementadas desde cero para indexar rápidamente a anfitriones y huéspedes mediante el algoritmo djb2 y resolución por encadenamiento.

Además del desarrollo de estas estructuras, se integran mecanismos de lectura desde archivos externos, con procesos rigurosos de validación de entradas para garantizar la integridad de las relaciones entre objetos. Se valida, por ejemplo, que un alojamiento pertenezca a un anfitrión existente, o que una reserva refiera a un huésped y alojamiento válidos. El diseño incluye el uso de un diagrama UML simplificado y decisiones técnicas orientadas a garantizar un sistema funcional, organizado y sostenible, que sirva como base para su posterior implementación.

El cuerpo del documento presenta una descripción de las estructuras utilizadas y su implementación, las funciones principales del sistema, organizadas por entidad y propósito, el contenido y propósito de los archivos de entrada (`alojamientos.txt`, `reservas.txt`, etc.), la estructura de las tablas hash para anfitriones y huéspedes, con su rendimiento esperado, ejemplos de salida del sistema ante diferentes operaciones, como listar alojamientos, consultar reservas, o generar reportes.

En conjunto, esta entrega demuestra la materialización del diseño lógico en una solución estructurada que permite escalar a futuro la aplicación mediante nuevas funcionalidades, como la posibilidad de realizar reservas desde consola, filtrar alojamientos disponibles, o implementar una interfaz gráfica.

II. Propuesta de solución

A. Condiciones y restricciones

Para la realización del reto se asume lo siguiente:

- Se tienen 5 clases: Afitrion, Huesped, Alojamiento, Reserva y Fecha.
- No se permite el uso de herencia.
- Se agregó un campo de contraseña para que el usuario inicie sesión.
- Los datos en los archivos de texto provienen de una fuente que garantiza su integridad y que responden al formato que se definirá más adelante.
- Las fechas siguen el siguiente formato de 10 caracteres: dd/mm/aaaa.
- Las estructuras de datos a utilizar son: mapas hash y listas simples enlazadas de implementación propia.

- Los códigos de los alojamientos y las reservas son numéricos e incrementales. Así: el alojamiento con código 1 se agregó al sistema antes que el alojamiento de código 2. Igualmente en el caso de las reservaciones.
- La identificación del huésped es un número de cédula colombiano u otro código exclusivamente numérico.
- Una reservación cancelada no se agrega al histórico.
- Se agregan al histórico únicamente aquellas reservaciones que a la fecha de corte se hayan completado.
- Los alojamientos están asociados siempre a un anfitrión que existe en el archivo de anfitriones.
- Las reservas están asociadas a un huésped y un alojamiento que existen en el archivo de huéspedes y alojamientos.
- Si bien las reservas canceladas no se agregan al archivo de históricos sí se agregan a un nuevo archivo de texto llamado **cancelaciones.txt**
- Respecto a la entrega 1 se agrega a los huéspedes un campo de nombre para mostrarlo cuando se hagan las reservaciones.

B. Formato de los archivos de texto

Con respecto a los archivos definidos en la primera entrega, se han agregado cabeceras con el objetivo de mantener información sobre la cantidad de líneas que contiene cada uno. Esto permite crear los mapas hash con un tamaño suficientemente grande para almacenar los datos, minimizando así la cantidad de colisiones posibles.

- **anfitriones.txt**: Contiene los administradores de los alojamientos. En cada línea se almacenan los datos del anfitrión en el siguiente formato: `[cedula] [password] [antigüedad] [calificación]`. Cada campo está separado por espacios y no deben existir dos anfitriones con la misma cédula. La primera línea del archivo indica el número total de anfitriones (es decir, la cantidad de líneas siguientes).
- **huespedes.txt**: Contiene los clientes del sistema. Cada línea almacena los datos del huésped en el siguiente formato: `[cedula];[nombreconapellidos];[password];[antiguedad];[calificacion]`. Los campos están separados por punto y coma. No deben existir dos huéspedes con la misma cédula. La primera línea del archivo indica la cantidad total de huéspedes.
- **reservaciones.txt**: Almacena las reservaciones realizadas en el sistema. Cada línea contiene la información de una reservación con el siguiente formato: `[fecha_inicio];[duracion_noches];[codigo_reservacion];[codigo_alojamiento];[codigo_huesped];[tipo_pago];[fecha_pago];[precio];[anotaciones]`. No habrá dos reservaciones con el mismo código. Los campos están separados por punto y coma. La cabecera del archivo contiene dos valores separados por espacio: la cantidad total de reservaciones actualmente en el archivo y el código de la última reservación.
- **alojamientos.txt**: Incluye las casas o apartamentos que administra un anfitrión y que pueden ser reservados por los huéspedes. Cada línea contiene la información del alojamiento en el siguiente formato: `[nombre];[codigo_alojamiento];[anfitrion_responsable];[departamento];[municipio];[tipo1:Casa,tipo2:Apartamento];[direccion];[precio];[amenidades]`. No habrá dos alojamientos con el mismo código. Se asume que todos los alojamientos están asociados a un anfitrión existente. La cabecera del archivo indica la cantidad total de alojamientos.
- **historico.txt**: Contiene las reservaciones que han pasado a formar parte del histórico. Cada línea sigue el mismo formato que en el archivo de reservaciones, pero no incluye cabecera. Las reservaciones están ordenadas en orden descendente según su momento de finalización.
- **cancelaciones.txt**: Archivo adicional que almacena las reservaciones anuladas. Estas no se consideran parte del histórico, ya que no finalizan de forma regular, sino por causas fortuitas. En lugar de eliminarlas, se almacenan de forma persistente en este archivo separado.

ESTRUCTURAS DE DATOS

Las estructuras de datos empleadas deben garantizar tiempos de acceso rápidos, flexibilidad ante el crecimiento del volumen de datos y la posibilidad de almacenamiento en memoria no contigua.

Para cumplir con estos requisitos, se eligieron e implementaron dos tipos principales: mapas no ordenados y listas simplemente enlazadas. Los mapas se utilizan para almacenar punteros a las instancias de las clases creadas, permitiendo búsquedas eficientes por clave.

Por otro lado, las listas simplemente enlazadas, aunque menos eficientes para operaciones de búsqueda o eliminación, se emplean para mantener el rastreo de elementos asociados a una entidad, como las reservaciones de cada huésped o alojamiento, y los alojamientos correspondientes a cada anfitrión.

Aunque sobre estas listas se realizan operaciones de búsqueda y eliminación, se asume que el tamaño de las mismas será moderadamente grande. Por ejemplo, es poco probable que un huésped tenga más de 500 reservaciones, o que un anfitrión administre más de 500 alojamientos. En tales casos, donde el volumen de datos supera estos límites, sería más apropiado almacenar la información en sistemas de bases de datos relacionales en lugar de archivos de texto.

De esta manera, la posible pérdida de eficiencia en la consulta se compensa con la flexibilidad que brindan estas estructuras en cuanto a la gestión de memoria no contigua.

`Unordered_Map`

Un `unordered_map` es una estructura de datos que almacena pares *clave-valor* y permite acceder a los valores de manera eficiente a partir de su clave, sin importar el orden de inserción.

Características principales:

- Basado en una *tabla hash* que distribuye las claves en diferentes *buckets* o cubetas, usando una función hash.
- La función hash transforma la clave en un índice numérico que indica en qué bucket se debe almacenar el par.
- Para manejar *colisiones* (cuando dos claves diferentes tienen el mismo índice), se utiliza la técnica de *encadenamiento*, donde cada bucket contiene una lista enlazada de elementos.
- Las operaciones típicas (inserción, búsqueda y eliminación) tienen un tiempo promedio cercano a $O(1)$.

Funcionamiento interno

Para insertar un nuevo par `(clave, valor)`:

- Se calcula el índice aplicando la función hash a la clave.
- Se revisa el bucket correspondiente en la tabla.
- Si la clave ya existe, se actualiza su valor.
- Si no, se añade un nuevo nodo al inicio de la lista enlazada en ese bucket.

Para buscar un valor dado su clave:

- Se calcula el índice con la función hash.
- Se recorre la lista enlazada del bucket hasta encontrar la clave o llegar al final. Note que si no han existido colisiones la función hash mapea al bucket exacto donde está el elemento y la búsqueda es O(1).

Para eliminar un par:

- Se busca el nodo con la clave.
- Se elimina el nodo ajustando los punteros de la lista enlazada.
- Se devuelve el valor eliminado para que el usuario pueda gestionarlo (liberarlo, reutilizarlo, etc.). La idea detrás de esto es que el usuario tenga la capacidad de almacenar en el mapa punteros crudos a memoria dinámica o no. La acción de borrado elimina el contenido del bucket pero la responsabilidad de liberar o no la memoria recae sobre el caller. Nuevamente, si no existieron colisiones la operación es O(1).

Consideraciones importantes

- La eficiencia depende en gran medida de la calidad de la función hash y del factor de carga (relación entre el número de elementos y el tamaño de la tabla).
- La función hash utilizada es *djb2*, un algoritmo simple y efectivo para tipos básicos, propuesto en 1991.
- Usualmente un mapa hash ocupa más espacio en memoria que por ejemplo un arreglo, pues debe almacenar referencias a nodos, crear una tabla sobredimensionada para evitar colisiones, aún así, en la mayoría de los casos se justifica su uso por la flexibilidad de poder usar claves de muchas naturalezas para almacenar datos y acceder a ellos rápidamente.
- No se realiza *rehashing* automático. Por ello, se reserva un tamaño inicial suficiente para evitar una alta carga. Esto justifica la existencia de las cabeceras con el número de elementos: cada mapa se sobredimensiona de forma que, al agregar elementos, se procure no superar el 75 % de su capacidad.
- El *rehashing* no se contempla porque implica crear una nueva estructura de memoria más grande, volver a aplicar la función hash a los elementos antiguos y nuevos, y copiar los datos y liberar la estructura anterior. Este proceso conlleva un costo computacional que, en esta aplicación, no se considera justificado. De hecho, si se requiriera un *rehashing* frecuente, lo más apropiado sería optar por estructuras como árboles balanceados.
- La memoria utilizada incluye tanto la tabla de *buckets* como los nodos de las listas enlazadas.
- Las claves solamente pueden ser tipos básicos como int, char, float, para tipos compuestos se deberá extender su funcionalidad o utilizar la STL (recomendado).

Algorithm 1: Función hash `djb2`

Require: Cadena de caracteres str
Ensure: Valor hash entero sin signo
1: $hash \leftarrow 5381$
2: **for** cada carácter c en str **do**
3: $hash \leftarrow ((hash \ll 5) + hash) + \text{ASCII}(c)$ // equivalente a $hash \times 33 + c$
4: **end for**
5: **return** $hash$

El algoritmo anterior muestra la implementación básica de la función hash djb2, el entero es un magic number que funciona bien disminuyendo las colisiones y tiene la característica de ser un número primo. La cadena de caracteres es simplemente una forma de generalizar el hecho de que se aplica sobre cada byte de información de la llave.

Esta estructura permite un acceso rápido y flexible a los datos mediante claves personalizadas, siendo muy útil para implementar mapas y diccionarios en C++. La STL tiene implementaciones mucho más robustas, no obstante, esta exploración es útil para implementar las estructuras en lenguajes como C.

`Lista simple enlazada`

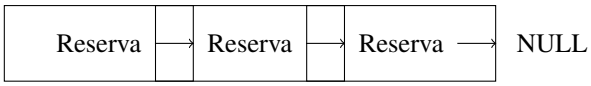
En este proyecto se implementa una **lista simplemente enlazada** como estructura de datos personalizada, sin utilizar la STL (Standard Template Library). Esta lista permite almacenar punteros a objetos de distintas clases, tales como `Reserva` o `Alojamiento`, y es utilizada para representar colecciones dinámicas dentro del sistema.

Una lista simplemente enlazada consiste en una secuencia de nodos, donde cada nodo contiene un puntero a un objeto y un puntero al siguiente nodo. Esta estructura permite una gestión dinámica de memoria eficiente, especialmente útil cuando la cantidad de elementos varía durante la ejecución del programa pues permite crecer la estructura sin realizar copias y no almacena datos en memoria contigüa.

Cada objeto de tipo `Huésped` contiene un puntero a una lista enlazada con sus reservas, si es un `Anfitrión` es una lista enlazada a sus alojamientos. Por otra parte, un `Alojamiento` tiene una lista enlazada a las reservaciones que tiene activas.

La implementación de esta estructura permite realizar operaciones como inserción, eliminación y recorrido de manera iterativa, utilizando únicamente punteros crudos, lo que asegura un control explícito sobre la memoria.

A continuación, se ilustra gráficamente el concepto de una lista simplemente enlazada, por ejemplo, para tener acceso a las reservas de un huésped:



La principal desventaja es que en el peor caso buscar en una lista es O(n) con n el número de elementos, al igual que en una búsqueda. Las inserciones son O(1) si se agregan nodos siempre al inicio. Esto no deberá ser un problema teniendo en cuenta que un huésped o anfitrión no debería tener demasiados alojamientos o reservaciones asociadas a la vez. Mientras que un alojamiento no debería estar a la vez presente en muchas reservaciones.

CLASES

El programa hace uso de cinco clases principales: *Anfitrión*, *Huésped*, *Alojamiento*, *Reserva* y *Fecha*. El diagrama de clases correspondiente se muestra en la Figura 1.

Clase: Anfitrión

Esta clase representa a los administradores responsables de los alojamientos. Cada anfitrión mantiene una lista de referencias a sus alojamientos y tiene la capacidad de eliminar reservas asociadas a ellos. Inicialmente se contempló una clase genérica *Usuario* para englobar tanto a huéspedes como a anfitriones, pero se optó por mantenerlas separadas. Esta decisión se basa en que, en contextos más complejos o industriales, ambas clases podrían evolucionar con funcionalidades y atributos específicos que justifican su independencia, sin recurrir a herencia en esta etapa del proyecto.

Clase: Huésped

Esta clase representa a las personas registradas en la aplicación que pueden realizar reservas. Los huéspedes almacenan sus reservas activas en una lista simple y únicamente tienen acceso a sus propias reservas, sin visibilidad ni control sobre las de otros huéspedes o alojamientos. La creación y gestión de reservas es responsabilidad de la lógica principal del sistema, que otorga a cada huésped una referencia para acceder o cancelar su propia reserva.

Clase: Alojamiento

Esta clase modela los espacios disponibles para reservar. Cada alojamiento está asociado a un anfitrión y mantiene una lista de sus reservas activas. Proporciona métodos para gestionar dichas reservas, incluyendo su eliminación, verificación de estado y visualización.

Clase: Fecha

Esta clase facilita la definición y manipulación de fechas a partir de valores numéricos o cadenas con formato `dd/mm/aaaa`. Cada instancia de fecha se vincula a una reserva; aunque múltiples reservas pueden compartir una misma fecha, para los propósitos de este proyecto, esta asociación simple resulta suficiente.

Los detalles sobre atributos, métodos y relaciones entre las clases se presentan en el diagrama de la Figura 1, generado con PlantUML. En dicho diagrama, los métodos públicos están indicados con puntos verdes, mientras que los atributos y métodos privados aparecen con cuadros rojos.

MEDICIÓN DE CONSUMO

Una de las solicitudes al momento de realizar el reto es medir el consumo de memoria por parte de los objetos creados y la cantidad de ciclos realizados para realizar alguna función en el programa. Así fue necesario crear una serie de variables de 32 bits tener control sobre lo pedido. El enfoque fue el siguiente

1) Variables globales y declaración `extern`: Dado que el proyecto se dividió en múltiples archivos fuente, fue necesario compartir ciertas variables entre ellos. Para lograrlo, se definieron variables globales en el archivo `app.c`, y se accedió a ellas desde otros módulos mediante la declaración `extern`.

Si bien el uso de variables globales suele considerarse una mala práctica en programación estructurada, existen contextos donde su utilización está justificada. Por ejemplo, en sistemas embebidos es común declarar variables globales para controlar el estado del sistema, como banderas de interrupción, registros de estado de periféricos, o variables de depuración accesibles desde distintas rutinas.

En este proyecto, las variables globales permitieron, entre otras cosas, realizar mediciones de rendimiento (como el conteo de ciclos, llamadas a funciones estándar, o consumo de memoria) sin alterar innecesariamente las firmas de las funciones. Si se hubiese optado por pasar estas variables como parámetros, cada función debería gestionar información adicional que no forma parte de su propósito principal, lo cual introduciría una sobrecarga innecesaria y dificultaría la lectura del código.

Además, como estas métricas son solo relevantes durante la etapa de pruebas o desarrollo, deben ser eliminadas antes de la entrega final o puesta en producción. El uso de variables globales permite centralizar y aislar este tipo de lógica, facilitando su remoción posterior sin tener que refactorizar múltiples prototipos y llamadas a funciones.

Por estas razones, el uso de variables globales en este contexto se considera razonable y justificado.

C. ¿Qué se considera un objeto en memoria?

En el contexto del análisis de uso de memoria, se hace una distinción entre objetos efímeros y objetos persistentes. Por ejemplo, supóngase que para implementar una funcionalidad `foo()` se invocan internamente tres funciones: `foo1()`,

Fig. 1. Diagrama UML.

`foo2()` y `foo3()`. Durante la ejecución de `foo2()`, se instancia un objeto de la clase `Class0`, que solo es utilizado dentro de `foo2()` y `foo3()`, y es destruido automáticamente al finalizar la ejecución de estas funciones.

Dado que este objeto no se almacena en estructuras persistentes (como listas, mapas o referencias compartidas), ni se retorna hacia funciones de nivel superior, su ciclo de vida es corto y limitado al alcance de la funcionalidad interna. Por lo tanto, en este proyecto no se contabiliza su impacto en el uso de memoria general. Su existencia temporal se considera irrelevante en comparación con los objetos de mayor duración, como las instancias de `Reserva`, `Alojamiento`, `Huésped` o `Anfitrión`, que permanecen activos a lo largo de toda o gran parte de la ejecución del programa.

En resumen, únicamente se consideran para el análisis de consumo de memoria aquellos objetos cuya vida útil sea lo suficientemente prolongada como para tener un impacto representativo en el uso global de recursos.

D. ¿Qué pasa con las funciones de la librería estándar?

Cuando se utilizan funciones de la biblioteca estándar, como `strlen()`, `memcpy()`, `string.length()`, entre otras, se lleva un registro de la cantidad de veces que cada una ha sido invocada durante la ejecución de las funcionalidades que las emplean. Al finalizar dichas funcionalidades, se imprime en la terminal el número de llamadas realizadas, junto con la complejidad computacional estimada de cada función.

Este monitoreo permite tener una mejor comprensión del costo computacional asociado al uso de funciones auxiliares que, aunque externas al código fuente del proyecto, influyen en el desempeño general y el consumo de recursos.

E. ¿Existen alternativas?

Si bien el objetivo principal de esta funcionalidad es que los estudiantes comprendan el costo computacional asociado a determinadas decisiones de diseño, en el ámbito profesional existen herramientas ampliamente utilizadas —algunas integradas en compiladores modernos— que permiten estimar el impacto en el rendimiento de una aplicación de forma automatizada.

Entre estas herramientas se encuentran los analizadores de rendimiento (*profilers*) como `gprof`, `valgrind` o `perf`, así como opciones de compilación avanzadas que generan reportes detallados sobre el consumo de tiempo, uso de memoria, número de llamadas a funciones, entre otros aspectos. Estas herramientas resultan fundamentales en contextos profesionales, ya que permiten identificar cuellos de botella y optimizar el comportamiento del software sin necesidad de implementar manualmente el código fuente.

Para futuras ediciones del curso, podría considerarse la introducción de este tipo de herramientas con enfoque profesional, lo cual enriquecería la formación del estudiante y resultaría especialmente útil en cursos avanzados relacionados con sistemas embebidos o circuitos digitales.

III. Implementación

A continuación se hace un resumen de algunas funciones y métodos relevantes:

El archivo **`app.c`** contiene la lógica para gestionar todos los objetos, permite crear reservas y agregarlas a los anfitriones y alojamientos involucrados. Permite anular las reservas y llamar a los métodos dentro de los alojamientos y los huéspedes para eliminarlas. Además, gestiona la creación del histórico a partir de la orden que da un anfitrión. A continuación se tienen algunas funciones relevantes:

Función `zona_huesped`: Esta función gestiona la interacción del huésped con el sistema durante una sesión.

Primero, intenta iniciar sesión un huésped y, si falla, informa el error y muestra estadísticas de ciclos y memoria utilizados. Luego carga los datos necesarios: anfitriones, alojamientos y reservas, manejando errores en cada paso y mostrando estadísticas de uso de memoria y ciclos de procesamiento.

El huésped puede seleccionar entre anular una reservación existente, crear una nueva reservación o salir del sistema. Para anular, se verifica la existencia de la reserva y se actualizan las estructuras de datos, además de registrar la cancelación. Para crear, se invoca un menú que genera la nueva reserva y la inserta en la estructura correspondiente.

Al salir, la función libera toda la memoria asignada dinámicamente, guarda los datos actualizados si fue necesario y muestra reportes de recursos consumidos. Finalmente, se libera la memoria del objeto huésped y se informa que los objetos fueron eliminados correctamente.

Función `zona_anfitrion`: Esta función administra la sesión y las operaciones que un anfitrión puede realizar dentro del sistema.

Primero, intenta iniciar sesión el anfitrión; si falla, se reporta el error y se muestra información sobre ciclos y memoria consumida. Luego, carga en memoria los alojamientos y reservas correspondientes al anfitrión, verificando errores en la carga y mostrando estadísticas de uso de recursos.

El anfitrión puede elegir entre varias opciones:

- Consultar reservaciones activas dentro de un rango de fechas ingresado.
- Anular una reservación, verificando la existencia y actualizando las estructuras de datos y archivos relacionados.
- Crear un histórico de reservas, exportándolo a un archivo.
- Cambiar la fecha del sistema, para simular diferentes escenarios o fechas de consulta.
- Guardar los cambios y salir.

Al salir, se liberan todos los recursos dinámicos utilizados, se actualizan los archivos de reserva si hubo cambios y se informa sobre el consumo de recursos. Finalmente, se libera la memoria del objeto anfitrión.

Función `obtener_fecha_actual`: Esta función obtiene la fecha actual del sistema y la formatea como una cadena en el formato `dd/mm/yyyy`. Para ello, utiliza la función `std::time` para obtener el tiempo actual en segundos desde la época, luego convierte este valor a la hora local mediante `std::localtime`, y finalmente formatea

la fecha en el buffer provisto usando `std::strftime`. Las llamadas a `std::time` y `std::localtime` tienen complejidad constante $O(1)$, mientras que `std::strftime` tiene complejidad lineal $O(n)$ respecto al tamaño del buffer.

Función `dividir_linea`: Esta función recibe una línea de texto y la divide en campos separados por el carácter punto y coma (';'). Los campos extraídos se almacenan en un arreglo de cadenas `campos` con un límite máximo de campos indicado por `max_campos`. La función recorre la línea buscando cada delimitador ';' mediante `std::string::find`, extrayendo subcadenas con `std::string::substr` desde la posición inicial hasta el delimitador o el final de la línea. Retorna el número de campos encontrados. Durante la ejecución se incrementan contadores globales para medir llamadas a `find` y `substr`, así como ciclos procesados.

Función `leer_reservas`: Esta función lee un archivo de texto que contiene datos de reservas y los almacena en una estructura de tipo `Unordered_Map<uint32_t, Reserva>`. Recibe como parámetros el nombre del archivo, un mapa de alojamientos para relacionar reservas con alojamientos, un puntero a un huésped para asignar reservas específicas, y referencias para devolver el número total de reservas y el código de la última reserva.

La función abre el archivo y lee inicialmente la cantidad de reservas y el código de reserva inicial. Luego reabre el archivo para procesar línea por línea, dividiendo cada línea en campos mediante la función `dividir_linea`. Cada línea representa una reserva cuyos campos se convierten a los tipos adecuados (fechas, números, caracteres y cadenas).

Se crean objetos `Fecha` para manejar las fechas de inicio, pago y final de la reserva, y se instancia un objeto `Reserva` con estos datos. La reserva se inserta en el mapa de reservas, y si es posible, se asocia también con el alojamiento correspondiente y con el huésped indicado.

La función maneja errores de conversión con excepciones para evitar la interrupción del proceso ante datos incorrectos. Finalmente, retorna un puntero al mapa con todas las reservas leídas.

Función `cargar_anfitriones`: Esta función lee un archivo de texto que contiene datos de anfitriones y los carga en un mapa hash `Unordered_Map<uint64_t, Anfitrion>`.

Primero, se abre el archivo y se lee el número total de anfitriones a cargar. Se reserva memoria para el mapa con un tamaño basado en esta cantidad. Luego, en un bucle, se leen los datos de cada anfitrión: documento (identificador único), contraseña, antigüedad y puntuación. Por cada línea leída, se crea un objeto `Anfitrion` y se inserta en el mapa con el documento como clave.

Durante la ejecución, se actualizan contadores globales para seguimiento de tamaño, longitud de cadenas y ciclos de procesamiento. Al finalizar, se cierra el archivo y se retorna un puntero al mapa con todos los anfitriones cargados.

Función `cargar_anfitriones`: Esta función se encarga de cargar en memoria los datos de los anfitriones desde un archivo de texto.

Primero, intenta abrir el archivo especificado; si no lo logra, retorna `nullptr` e imprime un mensaje de error. Luego, lee la cantidad de anfitriones que contiene el archivo y crea un mapa hash dinámico `Unordered_Map` para almacenar los anfitriones.

Por cada registro, lee el documento, la contraseña, la antigüedad y la puntuación, crea un objeto `Anfitrion` y lo inserta en el mapa. Durante este proceso, actualiza contadores globales de memoria y ciclos para el seguimiento del rendimiento.

Finalmente, cierra el archivo y retorna un puntero al mapa con todos los anfitriones cargados, o `nullptr` en caso de error.

Función `crear_reservacion`: Esta función permite a un huésped crear una nueva reservación en el sistema usando filtros.

Primero, solicita al usuario la fecha de inicio de la reservación. Si es válida, se pide la duración para calcular la fecha de finalización. El sistema verifica que el huésped no tenga ya una reserva en ese rango de fechas. Si todo es correcto, solicita el municipio como filtro inicial.

Luego, se recorre el mapa de alojamientos con un `callback` para determinar cuáles están disponibles en ese municipio y en las fechas seleccionadas. Si no hay alojamientos disponibles, se reporta al usuario.

A continuación, se permite aplicar filtros opcionales como el precio máximo por noche y la calificación mínima del anfitrión. Se muestran los alojamientos que cumplen con todos los criterios y se permite al usuario seleccionar uno ingresando su código.

Finalmente, se crea una instancia de `Reserva` con los datos proporcionados, se muestra al usuario y se retorna el puntero a la reserva creada. Si en cualquier punto ocurre un error (como fechas inválidas o selección incorrecta), la función libera los recursos dinámicos utilizados y retorna `nullptr`.

Función `crear_reservacion_codigo`: Esta función permite a un huésped crear una reservación ingresando directamente el código de un alojamiento específico.

Primero, solicita al usuario el código del alojamiento y verifica su existencia en el sistema. Si no existe, se informa el error. Si existe, se muestra su información y se solicita la fecha de inicio de la reserva. Luego, se obtiene la duración de la estadía para calcular la fecha de finalización.

A continuación, se valida que el huésped no tenga otra reserva que coincida con el rango de fechas especificado. También se comprueba que el alojamiento esté disponible durante ese periodo. Si alguna validación falla, se reporta el error y se liberan los recursos dinámicos utilizados.

Si todo es correcto, se procede a crear la instancia de `Reserva` con los datos suministrados y se muestra al usuario. Finalmente, se retorna un puntero a la reserva creada.

Función `escribir_reservas`: Esta función se encarga de guardar todas las reservaciones almacenadas en memoria dentro de un archivo de texto.

Primero, intenta abrir el archivo especificado para escritura. Si no se puede abrir, muestra un mensaje de error y termina

la ejecución. En caso contrario, escribe en la primera línea del archivo el número total de reservas y el último código de reserva utilizado.

Luego, utilizando una función de orden superior, recorre todas las reservas en la estructura tipo `Unordered_Map` y aplica el callback `escribir_reserva_callback` para escribir los datos de cada reserva individualmente en el archivo. Esta implementación aprovecha características de programación funcional en C++ para mejorar la legibilidad del código y reducir la complejidad. Finalmente, se cierra el archivo.

Función `validar_alojamientos`: Esta función es utilizada como callback para filtrar alojamientos que cumplen con los criterios básicos de una reserva (fechas y municipio).

Recibe un puntero a un objeto de tipo `Alojamiento` y un puntero genérico a los parámetros requeridos para la validación, que incluyen la fecha de inicio y fin, así como el municipio solicitado. Si el alojamiento es candidato válido para la reserva, se añade al frente de la lista de alojamientos disponibles.

Función `mostrar_alojamientos_disponibles`: Esta función aplica filtros adicionales sobre una lista de alojamientos previamente validados, considerando el precio máximo por noche y la puntuación mínima del anfitrión.

Recorre la lista de alojamientos candidatos y, para cada uno, consulta la puntuación del anfitrión desde el mapa correspondiente. Si el alojamiento cumple los criterios especificados, se añade a la lista final de disponibles y se muestra su información por pantalla, incluyendo la calificación del anfitrión. También incrementa los contadores globales de ciclos durante la iteración.

Devuelve `true` si se encontró al menos un alojamiento que cumpla con los criterios, o `false` en caso contrario.

Función `existe_alojamiento`: Verifica si existe un alojamiento específico, identificado por su código, dentro de una lista enlazada de alojamientos.

Recorre la lista comparando el código de cada alojamiento con el proporcionado. Si encuentra una coincidencia, devuelve un puntero al objeto correspondiente. En caso contrario, devuelve `nullptr`. Esta función también incrementa el contador global de ciclos durante la búsqueda.

Métodos relevantes de la clase `Huesped`

Método `get_obj_size`: Calcula el tamaño total en memoria que ocupa un objeto de tipo `Huesped`, incluyendo la longitud de la contraseña almacenada dinámicamente. Este método es útil para estimaciones de uso de memoria.

Método `eliminar_reserva`: Elimina una reserva asociada al huésped, siempre que esta exista en su lista de reservas. Si la operación es exitosa, devuelve `true`; en caso contrario, `false`. Utiliza macros de registro como `LOG_SUCCESS` y `LOG_ERROR` para reportar el resultado de la operación.

Método `set_reserva`: Añade una nueva reserva al huésped, insertándola al inicio de su lista de reservas. Si la reserva es nula, se registra un error y no se realiza la operación.

Método `tengo_reservas`: Verifica si el huésped ya tiene una reserva activa dentro de un intervalo de fechas dado. Recorre todas las reservas actuales y compara los rangos de fechas. Devuelve `true` si encuentra alguna superposición, y `false` en caso contrario. Además, incrementa un contador global de ciclos durante la iteración.

Método `mostrar_reserva_huesped`: Muestra por consola los detalles de una reserva específica asociada al huésped, incluyendo su código, nombre, ID del alojamiento, fecha de entrada y fecha de salida, con formato legible.

Métodos relevantes de la clase `Anfitrión`

Método `get_obj_size`: Calcula el tamaño total en memoria que ocupa el objeto `Anfitrion`, incluyendo el tamaño dinámico de la contraseña. Además, incrementa un contador global `g_strlen_cnt` cada vez que se calcula la longitud de la cadena, para fines estadísticos.

Método `set_alojamiento`: Agrega un nuevo alojamiento a la lista de alojamientos del anfitrión. Si el puntero recibido es nulo, se registra un error y se retorna `nullptr`. En caso contrario, se inserta el alojamiento al frente de la lista y se retorna el puntero al mismo.

Método `eliminar_reserva`: Intenta eliminar una reserva específica asociada a uno de los alojamientos del anfitrión. Recorre su lista de alojamientos buscando coincidencia por código. Si se encuentra, se llama al método `eliminar_reserva` del alojamiento correspondiente. El resultado de la operación se registra mediante macros de log (`LOG_SUCCESS`, `LOG_ERROR`).

Método `mostrar_alojamientos`: Muestra los alojamientos del anfitrión que tienen actividad en un rango de fechas especificado. Para cada alojamiento en la lista, se llama a su método `mostrar_reservas`, filtrando por el intervalo definido entre `desde` y `hasta`.

Métodos relevantes de la clase `Alojamiento`

Método `set_reserva`: Inserta una reserva en la lista de reservas del alojamiento. Si el puntero recibido es nulo, se registra un error y se retorna `nullptr`. En caso contrario, se agrega al frente de la lista y se retorna el puntero.

Método `mostrar_reservas()`: Muestra todas las reservas activas del alojamiento sin aplicar ningún filtro por fecha.

Método `es_candidato_reserva(const Fecha&, const Fecha&)`: Verifica si el alojamiento está disponible en el rango de fechas especificado. Retorna `false` si hay una superposición con alguna reserva existente; `true` en caso contrario.

Método `es_candidato_reserva(const Fecha&, const Fecha&, const std::string&)`: Variante del método anterior que además verifica si el municipio del alojamiento coincide con el solicitado. Si hay reservas que se cruzan con las fechas dadas, retorna `false`. En caso de estar libre y coincidir el municipio, retorna `true`.

Método `mostrar_reservas(Fecha&, Fecha&)`: Muestra las reservas activas que se encuentren dentro del rango de fechas proporcionado. Incluye el nombre del alojamiento y los detalles de cada reserva mostrada.

Método `mostrar_alojamiento`: Muestra en consola todos los detalles del alojamiento, incluyendo nombre, código del anfitrión, dirección, ubicación, tipo, precio y amenidades.

Métodos de la clase `Fecha`

Constructor `Fecha()`: Inicializa la fecha con valores por defecto (día, mes y año definidos como constantes).

Constructor `Fecha(uint8_t d, uint8_t m, int16_t a)`: Intenta establecer la fecha con los valores proporcionados. Si los valores no representan una fecha válida, se asignan los valores por defecto.

Método `bool set_fecha(uint8_t d, uint8_t m, int16_t a)`: Valida y asigna una nueva fecha si es válida. Retorna `true` si la fecha fue asignada correctamente, o `false` si no.

Método `bool cargar_desde_cadena(const char* cadena)`: Carga una fecha desde una cadena con formato `"dd/mm/aaaa"`. Valida tanto el formato como la fecha. Retorna `true` si es válida.

Método `char* a_cadena(char* destino) const`: Convierte la fecha a una cadena con formato `"dd/mm/aaaa"`. El buffer `destino` debe tener al menos 11 bytes.

Método `bool es_bisiesto(int16_t anio) const`: Determina si un año dado es bisiesto. Utiliza las reglas del calendario gregoriano.

Método `uint8_t dias_en_mes(uint8_t mes, int16_t anio) const`: Retorna la cantidad de días del mes indicado, teniendo en cuenta los años bisiestos.

Método `int32_t comparar(const Fecha& otra) const`: Compara la instancia actual con otra fecha. Retorna:

- `-1` si la instancia actual es menor,
- `0` si son iguales,
- `1` si es mayor.

Operadores `==`, `!=`, `<`, `<=`, `>`, `>=`: Sobrecargas de operadores de comparación entre fechas, utilizando la lógica del método `comparar`.

Método `void mostrar_fecha(const Fecha& fecha)`: Imprime por consola la fecha proporcionada en formato `"dd/mm/aaaa"`.

Método `Fecha* sumar_noches(uint16_t noches) const`: Devuelve un puntero a una nueva instancia de `Fecha` que representa la fecha actual más un número dado de noches. El usuario debe liberar la memoria.

Método `Fecha* agregar_anios(uint8_t anios) const`: Retorna una nueva fecha con los años sumados al año actual. El usuario debe liberar la memoria.

Método `void formato_legible() const`: Imprime la fecha en un formato legible por humanos (por ejemplo: `"martes, 23 de abril del 2024"`).

Método `uint8_t dia_semana() const`: Devuelve el día de la semana (0 para domingo, 1 para lunes, etc.), calculado con la fórmula de Zeller.

Métodos de la clase `Unordered_Map`

Constructor `Unordered_Map(size_t size)`: Crea una tabla hash con un tamaño ajustado para que su ocupación máxima no supere el 75%. Inicializa todas las posiciones a `nullptr`. En caso de fallo de asignación de memoria, lanza una excepción `std::bad_alloc`.

Destructor `~Unordered_Map()`: Libera la memoria reservada por todos los elementos almacenados en la tabla hash, así como la propia tabla.

Método `clear_values()`: Elimina únicamente los valores almacenados en la tabla hash. La responsabilidad de liberar esta memoria recae en el llamador.

Método `insert(const Key&, Value*)`: Inserta un nuevo par clave-valor en la tabla. Si la clave ya existe, se reemplaza el valor anterior. El manejo de colisiones se realiza mediante encadenamiento.

Método `find(const Key&)`: Busca un valor asociado a una clave determinada. Retorna un puntero al valor si lo encuentra, o `nullptr` en caso contrario.

Método `erase(const Key&)`: Elimina un par clave-valor de la tabla. Retorna un puntero al valor eliminado si la clave existe; en caso contrario, retorna `nullptr`. Es responsabilidad del llamador liberar la memoria del valor retornado.

Método `hash_fuction(const Key&) const`: Calcula el índice hash utilizando el algoritmo `djb2` aplicado a los bytes de la clave. Devuelve un valor en el rango `[0, m_size)`.

Método `info_map() const`: Retorna el tamaño en bytes de la estructura `Unordered_Map`, sin incluir la memoria ocupada por los elementos.

Método `for_each(void(*callback)(Key, Value*, void*), void*)`: Aplica una función callback a todos los elementos de la tabla hash, pasando la clave, el valor y un puntero a datos adicionales como argumento.

Problemas y evolución de la solución

Los dos principales retos en la implementación fueron definir el tipo de estructuras de datos a utilizar, pues debían ser robustas, tener tiempos de consulta, inserción y eliminación razonables y, en la medida de lo posible, utilizar únicamente los elementos estrictamente necesarios en memoria contigua. Para almacenar otros tipos de referencias, las listas simples resultaron ser una herramienta bastante útil. Por otro lado, los mapas hash, aunque por su naturaleza utilizan memoria contigua, esta solo almacena punteros y permite accesos, inserciones y eliminaciones en tiempo aproximadamente constante, por lo que fueron elegidos como las estructuras donde se almacenan las referencias a todos los objetos creados.

El siguiente reto fue establecer las relaciones. Para esto, en lugar de crear una maraña de interdependencias entre clases, se optó por mantener cada clase lo más independiente posible de las demás, y que solo tuvieran acceso directo a aquello que les pertenece. Por ejemplo, un huésped solo puede ver las referencias a sus reservaciones; sin embargo, a partir de ellas y haciendo cruces con los alojamientos, es posible ampliar las posibilidades de consulta.

En cuanto a la evolución de la solución, inicialmente se contempló una cantidad relativamente pequeña de métodos para las clases, lo que habría resultado en métodos complejos y muy dependientes de otras clases. Por ejemplo, se decidió que realizar una reservación no sería un método de la clase huésped, sino una funcionalidad de la aplicación principal que crea un objeto reservación y agrega la referencia a la lista de reservaciones del huésped, mientras esta se almacena junto a las demás reservaciones, permitiendo un control centralizado de todas ellas. Esto hizo que el diagrama de clases cambiara sustancialmente en algunos elementos. Aún así, se entiende que los diagramas y las soluciones iniciales no son una camisa de fuerza, sino parte del proceso iterativo para lograr una solución adecuada.

Resultados

A continuación se muestran los resultados en iteraciones, memoria consumida para algunas funcionalidades relevantes del sistema: cargar todos los datos en memoria, agregar una reservación con filtros, anular una reservación o hacer un histórico. Estos datos se hacen sobre un vector de pruebas con 5 huespedes, 5 anfitriones, 10 alojamientos y 15 reservas. Es una cantidad pequeña pero manejable para crear los datos.

Tabla I
Resumen de llamadas a funciones y métricas durante la funcionalidad *Cargar datos* en Huésped

Función	Complejidad	Veces llamada
strlen	O(n)	175
memcpy	O(n)	70
memcmp	O(n)	0
isdigit	O(1)	240
getline	O(n)	26
string::find	O(n)	226
string::substr	O(n)	226
std::npos	O(1)	0
c_str	O(1)	95
stoi	O(n)	55
stof	O(n)	15
stoull	O(n)	25
strcmp	O(n)	0
sprintf	O(n)	0
string::length	O(1)	0
Ciclos totales para cargar datos en memoria		965
Memoria ocupada por objetos creados (bytes)		5432

Tabla II
Resumen de llamadas a funciones y métricas durante la funcionalidad *Cargar datos en memoria* en Huésped

Función	Complejidad	Veces llamada
strlen	O(n)	157
memcpy	O(n)	26
memcmp	O(n)	5
isdigit	O(1)	240
getline	O(n)	27
string::find	O(n)	227
string::substr	O(n)	227
std::npos	O(1)	0
c_str	O(1)	95
stoi	O(n)	65
stof	O(n)	15
stoull	O(n)	25
strcmp	O(n)	0
sprintf	O(n)	0
string::length	O(1)	0
Ciclos totales para cargar datos en memoria		747
Memoria ocupada por objetos creados (bytes)		3045

Tabla III
Resumen de llamadas a funciones y métricas durante la funcionalidad *Crear reservación* en Huésped

Función	Complejidad	Veces llamada
strlen	O(n)	4
memcpy	O(n)	1
memcmp	O(n)	0
isdigit	O(1)	16
getline	O(n)	2
string::find	O(n)	0
string::substr	O(n)	0
std::npos	O(1)	0
c_str	O(1)	2
stoi	O(n)	0
stof	O(n)	0
stoull	O(n)	0
strcmp	O(n)	1
sprintf	O(n)	0
string::length	O(1)	2
Ciclos totales para crear una reservación		71
Memoria ocupada por objetos creados (bytes)		5608

Tabla IV
RESUMEN DE LLAMADAS A FUNCIONES Y MÉTRICAS DURANTE LA FUNCIONALIDAD *Anular reservación* EN HUÉSPED

Función	**Complejidad**	**Veces llamada**
strlen	O(n)	1
memcpy	O(n)	0
memcmp	O(n)	0
isdigit	O(1)	0
getline	O(n)	0
string::find	O(n)	0
string::substr	O(n)	0
std::npos	O(1)	0
c_str	O(1)	0
stoi	O(n)	0
stof	O(n)	0
stoull	O(n)	0
strcmp	O(n)	0
sprintf	O(n)	2
string::length	O(1)	0
Ciclos totales para anular una reservación		0
Memoria ocupada por objetos creados (bytes)		5412

Tabla V
RESUMEN DE LLAMADAS A FUNCIONES Y MÉTRICAS DURANTE LA FUNCIONALIDAD *Crear histórico de reservas* EN HUÉSPED

Función	**Complejidad**	**Veces llamada**
strlen	O(n)	6
memcpy	O(n)	0
memcmp	O(n)	0
isdigit	O(1)	0
getline	O(n)	0
string::find	O(n)	0
string::substr	O(n)	0
std::npos	O(1)	0
c_str	O(1)	0
stoi	O(n)	0
stof	O(n)	0
stoull	O(n)	0
strcmp	O(n)	0
sprintf	O(n)	12
string::length	O(1)	0
Ciclos totales para crear el histórico		72
Memoria ocupada por objetos creados (bytes)		2191

DISCUSIÓN

Los datos obtenidos son claros y esperables: la mayoría de las llamadas a funciones de la biblioteca estándar se concentran en los momentos de lectura de datos y creación de objetos. Esto se debe a que la lectura desde archivos de texto requiere transformar los datos, buscando patrones, tokens u otra información relevante que permita separar líneas y alimentar los constructores de las clases para generar las instancias necesarias.

Posteriormente, la frecuencia de llamadas a dichas funciones disminuye, ya que no es necesario escribir en los archivos hasta que el usuario decide guardar los cambios realizados. Esta estrategia se adoptó para evitar escrituras constantes, ya que las llamadas recurrentes al sistema para modificar archivos resultan costosas. De este modo, si un usuario realiza cambios pero no los guarda explícitamente, dichos cambios no se conservarán.

En cuanto al uso de memoria, las cantidades observadas son bajas, lo cual tiene sentido considerando el escaso volumen de datos procesados. Esto pone de manifiesto una observación importante: si se requiriera procesar un mayor volumen de datos, el consumo de memoria aumentaría. Este problema suele resolverse de dos maneras: utilizando bases de datos que se consultan desde un servidor para reducir la carga sobre el sistema principal, o aprovechando características del sistema operativo, como el uso de mmap en Linux, que permite mapear porciones de archivos en memoria sin cargar su totalidad, ya que muchas veces solo se necesita acceder a fragmentos específicos.

En este proyecto se buscó implementar una aproximación similar. Por ello, la cantidad de archivos que se cargan al actuar como anfitrión o como huésped difiere: las funcionalidades asociadas a los anfitriones requieren menos datos que las de los huéspedes, lo que se refleja en una diferencia en el uso de recursos.

Por otro lado, en un sistema embebido típico no es común procesar datos desde archivos, ya que estos dispositivos suelen carecer de soporte para sistemas de archivos tradicionales. Incluso cuando disponen de dicho soporte (por ejemplo, mediante tarjetas SD y bibliotecas como FATFS), el manejo de archivos suele ser limitado y se evita en la medida de lo posible. En escenarios más complejos, las tareas de procesamiento intensivo o consulta de bases de datos se delegan a sistemas más potentes, como servidores o plataformas en la nube, que luego envían los resultados al sistema embebido. Un ejemplo representativo de este enfoque es un microcontrolador que utiliza el protocolo MQTT para enviar una solicitud sobre la disponibilidad de un alojamiento en una fecha específica, y que recibe la respuesta desde un servidor remoto, reduciendo así la carga de procesamiento local.

Finalmente, durante la ejecución de la funcionalidad de anulación de reservas, la prueba consistió en anular la reserva recién creada. Se observó una leve diferencia de 20 bytes menos en la memoria utilizada en comparación con el estado posterior a la carga completa de datos. Este comportamiento parece deberse a un bug, probablemente relacionado con una doble liberación o resta asociada a algún dato o arreglo, cuya causa exacta no ha podido ser determinada al momento de redactar este informe. No obstante, la discrepancia es mínima y no afecta de manera significativa el funcionamiento del sistema ni el valor global de las métricas obtenidas.

CONCLUSIONES

El uso de archivos de texto puede parecer simple pero es una decisión práctica que permite mantener un control claro sobre la información y facilita futuras ampliaciones del sistema.

La implementación propia de listas simplemente enlazadas y tablas hash demuestra que es posible desarrollar mecanismos eficientes para la gestión dinámica de datos sin depender de bibliotecas estándar. El uso del algoritmo djb2 como función hash y del encadenamiento para resolución de colisiones favorece el acceso en tiempo constante promedio.

El uso de estructuras dinámicas como listas enlazadas permite que el sistema pueda escalar con facilidad, adaptándose al crecimiento del número de alojamientos o reservas, sin comprometer la eficiencia ni el uso de memoria.

Si bien se reconoce que las listas enlazadas no son óptimas para búsquedas frecuentes, su uso en contextos donde el número de elementos es limitado (como las reservas por usuario o los alojamientos por anfitrión) permite mantener un equilibrio entre simplicidad estructural y eficiencia operativa. Esta decisión refleja una buena lectura de la relación entre el dominio del problema y las herramientas disponibles.

Uno de los principales logros del proyecto fue lograr la correcta integración entre las entidades del sistema, mediante validaciones consistentes y relaciones bien definidas. Además, la modularidad de las funciones y la organización del código favorecen su mantenibilidad y expansión futura. El análisis de la complejidad de funciones clave también permitió reforzar la toma de decisiones algorítmicas.