**SISTEMA DE RESERVAS DE ALOJAMIENTOS**

*Adriana Camila Erazo Mora – 1006908110*

*María Angélica Osorio Rincón - 1001651755*

1. **ANÁLISIS DEL PROBLEMA Y CONSIDERACIONES PARA LA ALTERNATIVA DE SOLUCIÓN PROPUESTA**
   * **Análisis del problema**

El objetivo principal del proyecto es diseñar e implementar un sistema de gestión de reservas para alojamientos turísticos, que permita a anfitriones administrar sus propiedades y a huéspedes realizar reservas de manera eficiente y confiable. El sistema debe gestionar información de varias entidades claves: anfitriones, alojamientos, huéspedes, reservas y fechas asociadas a cada reserva.

El problema implica gestionar relaciones complejas y restricciones entre estas entidades, como:

* Un anfitrión puede tener múltiples alojamientos, pero un alojamiento pertenece a un único anfitrión.
* Un huésped puede tener múltiples reservas, pero no debe existir superposición de fechas entre sus reservas.
* Cada reserva está asociada a un alojamiento, que a su vez puede tener múltiples reservas, sin solapamiento en fechas.
* Una reserva puede abarcar múltiples fechas (por ejemplo, varias noches consecutivas).

Además, se debe considerar la persistencia de datos a través de archivos de texto, con formatos claros que permitan cargar y guardar la información del sistema sin redundancias innecesarias.

* + **Consideraciones para la alternativa de solución propuesta**

**1.2.1. Modelado de entidades y relaciones.**

Se propuso un modelo orientado a objetos, que refleja las entidades del dominio (anfitrión, huésped, alojamiento, reserva, fechas) y sus relaciones:

* Relaciones de pertenencia:
  + - Anfitriones “poseen” alojamientos (1 a N), sin alojamientos sin anfitrión.
    - Huéspedes “poseen” reservas (0 a N), sin reservas sin huésped.
    - Reservas “poseen” fechas (1 a N), fechas que no se solapan dentro de la misma reserva.
* Relación de asociación:
  + Las reservas están asociadas con alojamientos (1 a 0..N), asegurando que un alojamiento no esté reservado para las mismas fechas más de una vez.

Este modelado favorece la navegación eficiente entre objetos relacionados, la integridad de datos y una lógica clara para evitar solapamientos y redundancias.

**1.2.2. Persistencia y manejo de archivos.**

Se optó por usar archivos de texto separados para cada tipo de entidad (huespedes.txt, anfitriones.txt, alojamientos.txt, ReservasActivas.txt) con formatos específicos que incluyen identificadores y claves para login, además de referencias cruzadas mediante códigos (por ejemplo, códigos de reservas en huéspedes o códigos de alojamientos en anfitriones). Esto permite:

* + Mantener la cohesión al agrupar toda la información de una entidad en un único archivo.
  + Evitar la duplicación de datos (p. ej., no repetir datos completos de alojamientos en anfitriones, sino sólo referencias por código).
  + Facilitar la carga y asociación dinámica en memoria mediante listas de punteros a objetos, que se crean a partir de la lectura secuencial de archivos.
  + Manejar reservas activas separadamente de un histórico, permitiendo trabajar con un conjunto reducido de datos para las operaciones frecuentes.

**1.2.3. Estrategia de implementación.**

La solución implementa:

* Estructuras de datos dinámicas en memoria usando arreglos de punteros a objetos, para listas globales de anfitriones, alojamientos, huéspedes y reservas activas.
* Métodos estáticos en cada clase para cargar y guardar los datos desde y hacia archivos.
* Métodos de asociación que recorren listas para enlazar objetos relacionados basados en los códigos almacenados, creando punteros directos que facilitan la navegación y las operaciones posteriores.
* Control de acceso mediante login basado en documento y clave, almacenados en archivos, para permitir distintos menús y acciones según el tipo de usuario (anfitrión o huésped).
* Validaciones para evitar solapamientos de fechas en reservas y para garantizar integridad de las asociaciones.

1. **DIAGRAMA DE CLASES DE LA SOLUCIÓN PLANTEADA**

El sistema está compuesto por cinco clases principales: Anfitrion, Alojamiento, Huesped, Reservas y Fecha. Las relaciones entre estas clases se establecen conforme al modelo de dominio del problema y reflejadas en el código fuente.

* Clase Anfitrion:

Representa a los propietarios de los alojamientos. Cada anfitrión tiene una cédula, una clave de acceso, una antigüedad, una puntuación y los códigos de los alojamientos que posee. La relación más importante de esta clase es con la clase Alojamiento:

* + Relación de pertenencia con Alojamiento (1 a 1..N):
    - Un Anfitrion puede poseer varios alojamientos (Alojamiento\*).
    - Esta relación se refleja mediante un arreglo dinámico de punteros a alojamientos que le pertenecen, enlazado en tiempo de ejecución con el método asociarAlojamientos.
* Clase Alojamiento:

Describe un lugar disponible para ser reservado. Contiene información como código, nombre, dirección, precio por noche, ubicación (departamento, municipio), tipo de alojamiento y amenidades.

* + Relación de asociación con Resevas (1 a 0..N):
    - Un alojamiento puede tener cero o muchas reservas, pero no mantiene una lista interna de reservas.
    - Las reservas, en cambio, contienen el código del alojamiento, y se vinculan mediante el método enlazarAlojamiento.
* Clase Huesped:

Representa a los usuarios que realizan reservas. Contiene datos como documento, clave, antigüedad, puntuación y los códigos de las reservas que ha reservado.

* + Relación de pertenencia con Reservas (1 a 0..N):
    - Un Huesped puede tener cero o muchas reservas activas
    - Cada huésped almacena internamente una lista de punteros a objetos Reservas, los cuales se enlazan dinámicamente en el método asociarReservas según los códigos leídos desde el archivo huespedes.txt.
* Clase Reservas:

Es la clase central que conecta a Huesped, Alojamiento y Fecha. Contiene información como: código de reserva, fecha de entrada, cantidad de noches, código de alojamiento, documento del huésped, método de pago, monto y anotaciones.

* + Relación de pertenencia con Fecha (1 a 1..N):
    - Cada reserva tiene un conjunto de fechas reservadas representadas por objetos de la clase Fecha.
    - Estas fechas se calculan a partir de la fecha de entrada y la cantidad de noches, usando el método asociarFechasReservadas, que genera dinámicamente un arreglo de punteros a Fecha.
* Clase Fecha:

Una clase auxiliar que representa una fecha específica. Está diseñada para facilitar operaciones como sumar días, comparar fechas y verificar superposición de rangos.

* + Relación de pertenencia con Reservas:
    - Solo existen fechas en tanto existan reservas. Por eso, cada reserva genera y destruye sus fechas asociadas.

*Nota sobre los archivos auxiliares (memoria.h, liberar\_memoria.h, utilidades.h y utilidades.cpp)*

Es importante mencionar que los archivos memoria.h, liberar\_memoria.h, utilidades.h y utilidades.cpp no se incluyen en el diagrama de clases porque no contienen clases definidas. Estos archivos son de carácter auxiliar y están destinados a proporcionar funciones y utilidades que apoyan el funcionamiento del sistema, pero no representan entidades del dominio que justifiquen su inclusión en un diagrama de clases.

* **memoria.h** y **liberar\_memoria.h**: se encargan de la gestión de memoria dinámica, proporcionando funciones para la asignación y liberación de memoria. No se incluyen como clases ya que son simplemente funciones y macros que no representan objetos con atributos y comportamientos propios.
* **utilidades.h y utilidades.cpp**: contienen funciones auxiliares que realizan tareas generales como la validación de datos, manejo de fechas, o conversión de tipos. Estas funciones son de uso transversal en el sistema, pero no forman parte de las entidades del modelo de dominio, por lo que tampoco se representan como clases.

Estas implementaciones están orientadas a apoyar la lógica de negocio, pero no contribuyen a la estructura de clases del sistema ni forman parte de los objetos clave del dominio (Anfitrion, Alojamiento, Huesped, Reservas y Fecha).

1. **DESCRIPCIÓN EN ALTO NIVEL DE LA LÓGICA DE SUBPROGRAMAS NO TRIVIALES**

En esta sección se explican los métodos y funciones más importantes del sistema, cuya lógica de implementación va más allá de simples operaciones de entrada/salida o acceso directo a atributos. Estos subprogramas implementan procesos de carga, asociación entre objetos, validaciones complejas y navegación entre entidades.

* + asociarAlojamientos() – Clase Anfitrion:

Relacionar dinámicamente a cada Anfitrion con los objetos Alojamiento que le pertenecen, en base a los códigos de alojamiento listados en el archivo anfitriones.txt.

* + Lógica:
    - Al cargar un anfitrión desde el archivo anfitriones.txt, se guarda una lista de códigos de alojamientos en un arreglo interno de strings.
    - Luego, en tiempo de ejecución, se llama al método asociarAlojamientos, que:
      * Recorre el arreglo global Alojamiento\*\* alojamientos de tamaño totalAlojamientos.
      * Por cada código de alojamiento guardado en el anfitrión, compara contra el getCodigoAlojamiento() de cada alojamiento.
      * Si hay coincidencia, guarda el **puntero** al objeto Alojamiento dentro del anfitrión.
    - Esto permite que el anfitrión tenga acceso directo a sus alojamientos sin duplicar información.
  + asociarReservas() – Clase Huesped:

Vincular a cada huésped con las reservas activas que le pertenecen, utilizando los códigos listados en el archivo huespedes.txt.

* + Lógica:
    - Similar al caso anterior, en el archivo huespedes.txt se listan los códigos de reservas activas por cada huésped.
    - Al ejecutar asociarReservas:
      * Se recorre el arreglo global Reservas\*\* reservaciones de tamaño totalReservas.
      * Para cada código de reserva asociado al huésped, se busca coincidencia con el getCodigoReserva() de las reservas activas.
      * Si coinciden, se guarda el puntero de la reserva dentro del arreglo interno del huésped.
    - Así se puede acceder desde un huésped directamente a los objetos Reserva que le corresponden.
  + AsociarFechasReservadas() – Clase Reservas

Generar el conjunto de fechas que comprende la reserva, a partir de la fecha de entrada y la cantidad de noches.

* + Lógica:
    - Se usa la clase Fecha, que permite operar con días (sumar días, comparar fechas, etc.).
    - En la función:
      * Se toma la fecha de entrada.
      * Se crea un arreglo dinámico de punteros a Fecha, con tamaño igual a la cantidad de noches.
      * Por cada noche:
        + Se genera una nueva fecha con new Fecha(dia, mes, anio).
        + Se almacena esa fecha en el arreglo.
        + Se incrementa la fecha en un día para la siguiente iteración.
    - Este arreglo queda como atributo interno de la reserva.
  + reservarAlojamiento() – Clase Huesped

Permitir que el huésped cree una nueva reserva en un alojamiento disponible.

* + Lógica:
    - Se muestran al usuario los alojamientos disponibles.
    - El huésped ingresa el código del alojamiento, la fecha de entrada y la cantidad de noches.
    - Se crea un nuevo objeto Reservas con new, y se configuran sus atributos.
    - Se llama internamente a:
      * asociarFechasReservadas() para calcular las fechas de estadía.
      * enlazarAlojamiento() para asociar la reserva con su alojamiento.
    - Luego:
      * Se agrega el puntero a la nueva reserva al arreglo Reservas\*\* reservaciones (realocando el arreglo si es necesario).
      * Se guarda también el puntero dentro del huésped.
    - Se verifica que no existan solapamientos de fechas con reservas ya existentes del mismo alojamiento.
    - Si hay conflicto, se cancela la reserva y se libera memoria.
  + anularReservacion() – Clases Huesped y Anfitrion

Eliminar una reserva del sistema, ya sea desde el punto de vista del huésped o del anfitrión.

* + Lógica:
    - Se solicita al usuario el código de la reserva a eliminar.
    - Se recorre el arreglo Reservas\*\* reservaciones.
    - Si el código coincide:
      * Se libera la memoria del objeto reserva (delete y nullptr).
      * Se compacta el arreglo manualmente, desplazando las reservas restantes una posición a la izquierda.
      * Se actualiza el total de reservas.
      * En la clase Huesped, se actualiza también su arreglo de reservas asociadas.
  + enlazarAlojamiento() – Clase Reservas

Asociar la reserva con el objeto Alojamiento al que pertenece.

* + Lógica:
    - Se recorre el arreglo Alojamiento\*\* alojamientos.
    - Se compara el código del alojamiento de la reserva con el de cada alojamiento.
    - Al encontrar coincidencia, se guarda el puntero al alojamiento dentro del objeto Reservas.
* actualizarHistorico()

Mueve las reservas activas cuya fecha de entrada es anterior o igual a una fecha de corte al archivo histórico, actualizando también el estado de los huéspedes y liberando memoria innecesaria.

* + Lógica:
    - Se crea un arreglo dinámico temporal reservasHistorico para almacenar las reservas que deben pasar al archivo histórico.
    - Se recorre el arreglo reservasActivas para identificar las reservas cuya fecha de entrada es menor o igual a fechaCorte.
      * Si se cumple la condición:
        + La reserva se guarda en reservasHistorico.
        + Se obtiene la cédula del huésped y el código de la reserva.
        + Se busca al huésped correspondiente en el arreglo huespedes y se elimina esa reserva de su historial activo mediante eliminarReservaHistorico.
        + La posición en reservasActivas se marca como nullptr para indicar que ya no está activa.
    - Luego, se abre el archivo historico.txt en modo adjuntar (ios::app):
      * Si no se puede abrir, se eliminan las reservas temporalmente almacenadas y se libera la memoria usada.
      * Si se abre correctamente:
        + Se escriben en el archivo todas las reservas movidas, una por línea y separadas por punto y coma.
        + Se elimina cada objeto reserva del histórico para liberar memoria.
    - Después de mover las reservas, se compacta el arreglo reservasActivas para eliminar los nullptr, reacomodando las reservas aún activas al principio del arreglo.
    - Se actualiza el valor de totalReservas con la nueva cantidad de reservas activas.
    - Finalmente, se guardan los datos actualizados de reservasActivas y huespedes en sus respectivos archivos (ReservasActivas.txt y huespedes.txt) usando funciones auxiliares.

1. **ALGORITMOS IMPLEMENTADOS DEBIDAMENTE INTRADOCUMENTADOS**

A continuación, se describen los algoritmos clave desarrollados en el sistema. Dado que el proyecto fue implementado completamente en C++ utilizando Qt Creator, los detalles completos de implementación y declaraciones se encuentran en los archivos .h y .cpp incluidos en el proyecto, principalmente en las clases Anfitrion, Huesped, Reservas, Alojamiento, Fecha, y el archivo utilidades (utilidades.cpp).

* + Validación de datos de entrada:

Se desarrollaron funciones utilitarias para asegurar que los datos ingresados por el usuario cumplan con los rangos y tipos esperados:

* rangeValidation(int, int, int): Verifica si un número se encuentra dentro de un rango específico.
* intValidation(int, int): Solicita al usuario un número entero dentro de un rango válido, gestionando errores de entrada con cin.
* charValidation(string): Solicita al usuario un carácter y limpia el buffer para prevenir entradas múltiples o erróneas.

Estas funciones se encuentran en utilidades.cpp y se usan ampliamente en los menús de interacción.

* + Asociación entre entidades:

Una vez cargados los objetos desde los archivos .txt, se establece la conexión lógica entre las entidades del sistema mediante punteros, por medio de los métodos asociarAlojamientos(), asociarReservas(), enlazarAlojamiento() y asociarFechasReservadas(), los cuales se encuentran implementados en sus respectivas clases. Estas asociaciones permiten una navegación eficiente y directa entre objetos relacionados.

* + Manejo de memoria dinámica:

El sistema trabaja exclusivamente con arreglos dinámicos de punteros a objetos. Para garantizar una gestión adecuada de la memoria:

* + Se implementó la función plantilla liberarArregloDePunteros() en utilidades.h, que:
    - Recorre el arreglo dinámico.
    - Libera cada objeto apuntado (delete arreglo[i]).
    - Libera el arreglo de punteros completo (delete[] arreglo).
    - Asigna los punteros a nullptr y reinicia el contador a 0.

Además, cada objeto Reservas también libera internamente su propio arreglo dinámico de fechas reservadas (Fecha\*\*) en su destructor. Este proceso se lleva a cabo en el método:

* + ~Reservas(): Recorre fechasReservadas, elimina cada Fecha\* y luego el arreglo completo.

Por otra parte, cuando se anula una reserva (ya sea desde Huesped o Anfitrion), se elimina explícitamente el objeto Reservas correspondiente con delete, lo que a su vez activa el destructor y garantiza la liberación completa de memoria asociada a la reserva y sus fechas.

* + Registro y anulación de reservas:

El proceso de crear o eliminar una reserva involucra múltiples pasos no triviales:

* + Reservar alojamiento:

El algoritmo de reserva de alojamiento, implementado en el método reservarAlojamiento(), contempla dos formas distintas de realizar una búsqueda antes de confirmar una reserva:

* + - Búsqueda por filtros:
      * El huésped ingresa fecha de entrada, cantidad de noches y municipio.
      * Puede aplicar filtros opcionales: costo máximo por noche y puntuación mínima del anfitrión.
      * El sistema verifica disponibilidad del alojamiento para esas fechas y cumple con los filtros.
      * Muestra las opciones válidas para que el huésped elija y realice la reserva.
    - Reserva directa por código:
      * El huésped ingresa el código del alojamiento deseado.
      * Se valida que el código exista y el alojamiento esté disponible en las fechas solicitadas.
      * Si es válido, se crea un nuevo objeto Reservas.
      * La reserva se enlaza con el huésped y el alojamiento correspondientes.
      * La reserva se añade al arreglo global de reservas activas.
  + Anulación de reserva:
    - Se verifica que la reserva pertenezca al huésped o al anfitrión.
    - Se elimina la reserva del arreglo global de reservas.
    - Se elimina también del arreglo de reservas del huésped.
    - Se invoca delete sobre el objeto Reservas, desencadenando el borrado completo de sus datos y fechas.

1. **PROBLEMAS DE DESARROLLO AFRONTADOS**
   * Análisis inicial confuso:

Uno de los principales obstáculos fue no acotar correctamente el problema desde el inicio. Se intentó considerar demasiados escenarios y posibilidades al mismo tiempo, lo cual dificultó la definición clara de las entidades del sistema y su interacción.

* + Dificultades en la definición de las clases:

Al principio costó establecer los límites y responsabilidades de cada clase, así como las relaciones entre ellas. Por ejemplo, entender qué datos debían almacenarse directamente en cada clase y cuáles debían ser enlazados mediante punteros (como en el caso de reservas con huéspedes o alojamientos) tomó tiempo y varias revisiones.

* + Diseño de la estructura de datos:

Elegir cómo representar las colecciones de objetos (huespedes, anfitriones, alojamientos y reservas) fue uno de los mayores desafíos. Inicialmente hubo intentos con estructuras poco adecuadas o incompletas. Finalmente se optó por implementar punteros a arreglos de punteros a objetos dinámicamente creados, lo que resolvió los problemas de acceso y permitió asociar eficientemente las entidades. No obstante, esta decisión exigió un manejo cuidadoso de la memoria, especialmente al momento de liberar recursos o eliminar reservas.

1. **EVOLUCIÓN DE LA SOLUCIÓN Y CONSIDERACIONES PARA TENER EN CUENTA EN LA IMPLEMENTACIÓN**

A lo largo del desarrollo del proyecto, la solución evolucionó de manera significativa tanto en su enfoque lógico como en su implementación estructural.

* Replanteamiento del diseño a partir de las relaciones entre entidades:

Inicialmente, la relación entre las clases no estaba claramente definida. Sin embargo, a medida que se entendió mejor el dominio del problema, se establecieron relaciones de pertenencia y asociación sólidas entre las clases Huesped, Anfitrion, Alojamiento, Reservas y Fecha. Este modelo relacional permitió una codificación más coherente y facilitó la navegación entre objetos.

* Modularización progresiva:

Al principio, muchas operaciones estaban fuertemente acopladas dentro del main, lo cual complicaba el mantenimiento y el crecimiento del sistema. Posteriormente, se trasladaron esas funcionalidades a subprogramas y métodos de clases, organizando el flujo lógico de manera más limpia y permitiendo una reutilización eficiente del código.

* Implementación centrada en archivos de texto como única fuente de persistencia:

Se tomó la decisión de trabajar con archivos .txt para todas las entidades (huespedes.txt, anfitriones.txt, alojamientos.txt, ReservasActivas.txt), lo cual facilitó pruebas manuales y mantenimiento del sistema. Cada archivo mantiene una estructura consistente, con los elementos separados por punto y coma (;) y relaciones expresadas mediante códigos, evitando duplicidad de datos.

* Separación entre reservas activas e históricas:

Se implementó una distinción entre reservas activas y reservas pasadas (histórico) con el objetivo de optimizar la carga de datos y enfocarse solo en las reservas relevantes para las operaciones del sistema en tiempo de ejecución.

* Validación robusta de entradas:

Se incluyó un conjunto de funciones utilitarias (en utilidades.cpp) para validar entradas numéricas, de caracteres, y rangos, minimizando errores en la interacción con el usuario y mejorando la experiencia durante la ejecución del programa.

* Estrategia clara de manejo de memoria dinámica:

A diferencia de soluciones basadas en contenedores estándar (std::vector, etc.), se implementó de manera consistente una estrategia de manejo manual de memoria dinámica. Esto exigió diseñar cuidadosamente funciones para la liberación de arreglos de punteros y la eliminación de objetos individuales (por ejemplo, al anular una reserva), asegurando la integridad y el correcto cierre del programa.

* Enlace lógico entre objetos post-carga:

Una vez que todos los datos se cargaban desde los archivos, se ejecutaban rutinas de asociación entre entidades relacionadas (por ejemplo, un Huesped enlazaba sus reservas mediante sus códigos), lo cual mantenía el sistema sincronizado y permitía un acceso directo a las dependencias sin necesidad de buscar repetidamente por identificación o clave.

1. **FORMATOS DE ARCHIVOS**

El sistema trabaja exclusivamente con archivos de texto plano (.txt) para persistir la información relacionada con cada una de las entidades del dominio. Estos archivos fueron diseñados de manera que:

* Mantengan la cohesión de los datos.
* Sean fácilmente editables manualmente para pruebas o depuración.
* Eviten la duplicación de información entre archivos, utilizando referencias por código o documento.

Acontinuación, se describen los formatos empleados:

* huespedes.txt: contiene la información básica de cada huésped, junto con los códigos de sus reservas activas.
* Formato por línea:

documentoHuesped;clave;antiguedad;puntuacion;codigoReserva1,codigoReserva2,...

El campo de códigos de reserva puede estar vacío si el huésped no tiene reservas activas.

* anfitriones.txt: almacena la información de los anfitriones y los códigos de alojamientos que tiene a su cargo.
* Formato por línea:

documentoAnfitrion;clave;antiguedad;puntuacion;codigoAloj1,codigoAloj2,...

* + Cada anfitrión puede tener uno o varios alojamientos, pero siempre referenciados por su código.
* alojamientos.txt: contiene el listado completo de alojamientos registrados en el sistema.
* Formato por línea:

codigoAlojamiento;nombreAlojamiento;nombreAnfitrion;departamento;municipio;tipo;direccion;precioNoche;amenidades

* + Las amenidades se almacenan como una única cadena.
* ReservasActivas.txt: registra las reservas en curso o futuras.
* Formato por línea:

codigoReserva;codigoAlojamiento;documentoHuesped;fechaEntrada;cantidadNoches;metodoPago;fechaPago;monto;anotaciones

* + Las fechas están en formato "DD/MM/AAAA".
  + Las anotaciones pueden contener observaciones adicionales de la reserva.
  + Las fechas asociadas a la reserva se reconstruyen a partir de la fecha de entrada y la cantidad de noches.

1. **ESTRUCTURA DE DATOS**

Después de leer cada archivo de texto se crea una lista de objetos correspondiente a cada clase. Esta lista se implementa como un puntero a un arreglo de punteros a objetos. Es decir, para cada tipo de entidad (anfitriones, alojamientos, huéspedes y reservas), se tiene una estructura donde:

* Un puntero principal apunta a la dirección de memoria de un arreglo.
* Cada posición del arreglo contiene un puntero que apunta a un objeto creado dinámicamente con new.
* Así se tiene una lista global de todos los anfitriones, todos los alojamientos, todos los huéspedes y todas las reservas activas, según la clase correspondiente.

Además, una vez cargados todos los objetos, se realiza un proceso de enlace entre entidades relacionadas, con base en las relaciones de pertenencia definidas en el dominio del problema. Por ejemplo, cada huésped contiene internamente un listado de códigos de reservas que le pertenecen; el programa recorre la lista global de reservas y enlaza (asigna) a cada huésped punteros directos a los objetos Reservas que coincidan con esos códigos. De esta manera, los objetos quedan interconectados según su relación: un huésped no solo conoce los identificadores de sus reservas, sino que también tiene acceso directo a los objetos que las representan. Este mismo patrón de asociación se aplica de forma general entre entidades que están lógicamente relacionadas (como reservas con alojamientos, o alojamientos con anfitriones), permitiendo una navegación estructurada y eficiente dentro del sistema.

A diagram of a class

AI-generated content may be incorrect.  
Figura 1. Representación gráfica de las estructuras de datos para las clases Anfitrion, Huesped, Reservas y Alojamientos

1. **ANÁLISIS DE EFICIENCIA: USO DE MEMORIA Y EJECUCIÓN DE FUNCIONALIDADES**

El proyecto fue desarrollado priorizando una estructura modular clara y un manejo responsable de la memoria dinámica. Desde esta perspectiva, el análisis de eficiencia se realiza considerando dos factores principales: uso de memoria y tiempo de ejecución de las operaciones críticas.

**9.1. Uso de memoria**

9.1.1. Asignación dinámica controlada: cada entidad (huespedes, anfitriones, alojamientos y reservas) es almacenada como un arreglo dinámico de punteros a objetos. Esta estrategia permite gestionar el uso de memoria en tiempo de ejecución, solo reservando memoria para la cantidad exacta de elementos cargados desde los archivos.

9.1.2. Liberación manual de memoria: al final del programa se libera explícitamente cada objeto y cada arreglo de punteros usando la función liberarArregloDePunteros. Además, cuando se anula una reserva, se elimina manualmente el objeto Reserva afectado y se reajusta el arreglo para evitar fugas de memoria. Esto garantiza ausencia de memoria residual al finalizar el programa.

9.1.3. Sin duplicación de información: el modelo evita la duplicación de objetos entre clases relacionadas. Por ejemplo, un Huesped no contiene copias de las reservas, sino punteros a los objetos Reservas ya existentes en el sistema global. Esto mantiene un uso de memoria compacto y eficiente.

**9.2. Eficiencia en tiempo de ejecución**

9.2.1. Operaciones basadas en recorridos lineales: dado el uso de arreglos dinámicos, muchas operaciones (como búsquedas de reservas, anfitriones o alojamientos) implican recorridos secuenciales (de orden *O(n)*).

9.2.2. Filtrado de alojamientos eficiente: la lógica para reservar alojamiento aplica filtros por municipio, fecha, cantidad de noches, puntuación del anfitrión y costo máximo por noche. Esta operación también es lineal, pero está cuidadosamente estructurada para detenerse temprano cuando los criterios no se cumplen, lo que mejora el rendimiento práctico.

9.2.3. Eficiencia en el acceso cruzado entre entidades: gracias a las asociaciones por punteros entre huéspedes, reservas, alojamientos y anfitriones, una vez realizado el proceso de enlazado en la carga, los accesos entre objetos relacionados son directos (*O(1)* por puntero), lo cual favorece una navegación eficiente en tiempo de ejecución.

**9.3. Complejidad computaciones de los métodos utilizados de la clase std::string**

* .empty(): Verifica si la cadena está vacía. Complejidad: O(1).
* .length() / .size(): Devuelve el número de caracteres de la cadena. Complejidad: O(1).
* .find(): Busca una subcadena dentro de la cadena. Complejidad en el peor caso: O(n × m), donde n es el tamaño de la cadena y m el de la subcadena buscada.
* .pop\_back(): Elimina el último carácter de la cadena. Complejidad: O(1).
* .front(): Accede al primer carácter de la cadena. Complejidad: O(1).
* .substr(pos, len): Obtiene una subcadena a partir de la posición pos con longitud len. Complejidad: O(1) + O(len) (creación constante y copia lineal).
* .stoi(): Convierte una subcadena a entero. Complejidad: O(n), donde n es la longitud de la parte numérica.
* .stof(): Convierte una subcadena a número en punto flotante (float). Complejidad: O(n).
* .to\_string(): Convierte un número a cadena. Complejidad: O(n), siendo n el número de dígitos del número convertido.
* std::cin.clear(): Limpia los indicadores de error del flujo de entrada. Complejidad: O(1).
* std::cin.ignore(): Descarta caracteres del flujo de entrada. Complejidad: O(n), dependiendo de la cantidad de caracteres a ignorar.
* std::cin.fail(): Verifica si el flujo de entrada está en estado de error. Complejidad: O(1).
* std::cin.seekg(): Mueve el puntero de lectura del flujo de entrada. Complejidad: O(1), aunque puede variar ligeramente según el sistema.
* std::getline(): Lee caracteres desde el flujo hasta un salto de línea (\n) o el final del archivo. Complejidad: O(n), donde n es el número de caracteres leídos.