**1. PROPÓSITO GENERAL DEL SISTEMA**

Crear un sistema que **gestione estadías hogareñas**, con énfasis en:

* Administración de **alojamientos**, **reservaciones**, **anfitriones** y **huéspedes**.
* Control de **archivos de texto de acceso secuencial** como almacenamiento permanente.
* Diferenciación entre **reservaciones activas y reservaciones históricas**, según una **fecha de corte**.
* Funcionalidades diferenciadas según el **rol** de usuario (huésped o anfitrión).
* No se realiza el **registro de nuevos usuarios**, sino login desde archivo histórico.

**2. ENTIDADES CLAVE DEL DOMINIO**

**2.1. Alojamiento**

* Nombre
* Código identificador (único)
* Anfitrión responsable (referencia al documento del anfitrión)
* Departamento
* Municipio
* Tipo (casa o apartamento)
* Dirección
* Precio por noche
* Conjunto de amenidades (lista de características)
* Fechas futuras (hasta un año) que están reservadas

**Relación:**

* Un alojamiento tiene **cero o más reservaciones activas**.
* Está **administrado por un anfitrión** (uno a muchos).

**2.2. Reservación**

* Código único de reservación
* Fecha de entrada
* Duración (en noches)
* Código del alojamiento
* Documento del huésped
* Método de pago (PSE o tarjeta de crédito)
* Fecha del pago
* Monto del pago
* Anotaciones (hasta 1000 caracteres)

**Relación:**

* Una reservación está asociada a **un alojamiento** y a **un huésped**.
* Puede ser **activa** o **histórica** según la fecha de corte.

**2.3. Anfitrión**

* Documento (único)
* Antigüedad en la plataforma (meses)
* Puntuación (0 a 5)
* Alojamiento(s) que administra

**Relación:**

* Un anfitrión administra **uno o varios alojamientos**.

**2.4. Huésped**

* Documento (único)
* Antigüedad en la plataforma (meses)
* Puntuación (0 a 5)
* Reservaciones activas o no

**Restricción:**

* Un huésped puede tener **cero o más reservaciones**, pero **nunca se debe tener varias reservas para la misma fecha**.

**3. FUNCIONALIDADES REQUERIDAS**

**I. Carga / actualización de datos (NO visible para el usuario)**

* Se deben leer archivos de texto (secuencial).
* Archivos distintos para: **huéspedes**, **anfitriones**, **alojamientos**, **reservas activas**, **reservas históricas**.
* Requiere diseño de formatos simples y estructurados.
* Debe permitir **actualización** (reescritura completa del archivo cuando haya cambios).

**II. Ingreso a la plataforma por perfil**

* Inicio de sesión usando documento (clave única).
* El menú mostrado depende del tipo de usuario (huésped o anfitrión).
* No se realiza registro de nuevos usuarios.

**III. Reservar alojamiento**

Solo para huéspedes:

* Búsqueda por:
  + Municipio + fecha + cantidad de noches
  + Filtros opcionales: costo máximo por noche, puntuación mínima del anfitrión
  + Alternativa: búsqueda por código de alojamiento
* Validaciones:
  + No tener otra reserva que se cruce en fechas
  + Alojamiento disponible en esas fechas
* Generación de código único de reserva
* Mostrar comprobante en formato especial de fecha

**IV. Anular reservación**

* La puede hacer tanto el huésped como el anfitrión
* Elimina la reservación del archivo activo (requiere reescritura)

**V. Consultar reservaciones (anfitrión)**

* Mostrar todas las reservaciones **activas** de sus alojamientos en un **rango de fechas específico**

**VI. Actualizar histórico**

* Se solicita al anfitrión una **fecha de corte**
* Mueve todas las reservaciones anteriores al archivo histórico
* Modifica archivo de reservas activas
* Redefine el rango de fechas futuras (12 meses desde la nueva fecha de corte)

**VII. Medición de consumo de recursos**

* Al terminar cualquier funcionalidad:
  + Contar cantidad de iteraciones (medición de eficiencia)
  + Calcular memoria usada (número de objetos, tamaño fijo)

**6. CONSIDERACIONES DE DISEÑO**

**6.2. Fechas**

* Es crucial una estructura para **fechas** que permita:
  + Comparación (igual)
  + Cálculo de rango
  + Mostrar en formato solicitado (ej: "lunes, 14 de marzo del 2025")

**6.3. Validaciones críticas**

* No permitir reservas solapadas en el huésped (varias reservas para una misma fecha)
* Verificar disponibilidad total del alojamiento antes de reservar.
* Validar existencia de usuario antes de permitir acciones.
* El **control de fechas**, crucial para reservas activas e históricas.
* Las **relaciones entre entidades**, como alojamiento-huésped o anfitrión-alojamiento.

**RELACIONES ENTRE CLASES**

* + **Relación de pertenencia entre anfitrión y alojamiento:** un anfitrión puede tener 1 o N alojamientos. Si no existe el anfitrión, no existen los alojamientos. Anfitrión 1 ----🡪 N Alojamientos
  + **Relación de pertenencia entre huésped y reserva:** un huésped puede tener 0 o N reservaciones, siempre y cuando no haya superposición de fechas en cada reservación. Si no hay huésped, no se pueden hacer reservas. Huésped 0 -----🡪 N Reservas
  + **Relación de pertenencia entre reservación y alojamiento:** un alojamiento puede tener 0 o N reservas, siempre y cuando no se reserve el alojamiento para la misma fecha. Si no hay alojamientos, entonces no se pueden hacer reservaciones. Alojamiento 0 -----🡪 N Reservas

**ARCHIVOS DE TEXTO**

* 1. **Huesped.txt**

*Formato:*

documento; clave; antiguedad; puntuacion; infoReservas

*Ejemplo:*

1001470812; 1234; 5; 3.8; R3, R4

* 1. **Anfitrion.txt**

*Formato:*

documento; clave; antiguedad; puntuacion; alojamientos

*Ejemplo:*

1001651755; 2345; 3; 4.2; Al1

* 1. **Alojamientos.txt**

*Formato:*

codigoAloj; nombre; nombreAnfitrion; departamento; municipio; tipo; dirección; precioNoche; amenidades; fechasFuturas

*Ejemplo:*

Al1; Viña del mar; Camilo Gonzales; Antioquia; Necoclí; casa; calle La Linda; 50000; piscina, aire acondicionado, parqueadero; 23/05/2025, 08/08/2025

* 1. **ReservasActivas.txt**

*Formato:*

codigoReserva; codigoAloj; documentoHuesped; fechaEntrada; cantNoches; metodoPago; fechaPago; monto

*Ejemplo*:  
R3; Al1; 1001470812; 23/05/2025; 4; PSE; 22/05/2025; 200000

* 1. **Historico.txt**

*Formato:*

codigoReserva; codigoAloj; documentoHuesped; fechaEntrada; cantNoches; metodoPago; fechaPago; monto

*Ejemplo:*

R1; Al1; 34557673; 03/01/2025; 3; PSE; 01/01/2025; 150000

R2; Al2; 71116826; 15/03/2025; 6; PSE; 15/03/2025; 180000

Incluir documento y clave en archivos Huesped.txt y Anfitrion.txt: la información de cada entidad en un único archivo, sin tener que duplicar datos al usar otro archivo para login, lo cual permite cohesión.

Enlazar los alojamientos en Anfitrion.txt por código: evita la redundancia y duplicación de datos. Cada código (Al1, Al2) ya contiene todos los datos del alojamiento en su archivo. El anfitrión conoce qué alojamiento tiene, pero no los contiene directamente en su archivo.

Enlazar las reservas activas en Huesped.txt por código: evita la redundancia y duplicación de datos. El huésped conoce qué alojamientos tiene, pero no los contiene directamente en su archivo. Esto permite asociar fácilmente cada reserva con su huésped, sin necesidad de recorrer todos los huéspedes por cada reserva.

Separar ReservaActiva.txt de Historico.txt, de manera que el archivo reservas activas contiene solo las reservas actuales y futuras, lo que permite trabajar sobre un arreglo más pequeño.

**ESTRUCTURA DE DATOS**

Teniendo en cuenta que no se puede usar contenedores STL y struct, se plantea estructurar los datos de los archivos txt en arreglos dinámicos de punteros a objetos. Cada línea del archivo corresponde a un objeto de la clase.

* **Anfitrión (Anfitrion\*\* listaAnfitriones):** al leer el archivo anfitriones.txt, por cada línea se crea un nuevo objeto: listaAnfitriones[i] = new Anfitrion(…). Esto permite que el accedo sea directo, compatible con archivos secuencias, redimensionable manualmente y que se pueda realizar lectura y escritura línea a línea (usando ifstream.getline() y se puede usar un strtok() para extraer cada información del objeto n, si en el archivo de texto se encuentran separador por “;” o “,”. Con estas estructuras, se debe cuidar la liberación de la memoria dinámica.
* **Huésped (Huesped\*\* listaHuespedes):** puntero a huésped, muy similar al anterior. Además, al realizar recorridos secuenciales, el acceso por documento se simplifica.
* **Alojamiento (Alojamiento\*\* listaAlojamientos).**
* **Reservaciones actuales (Reservacion\*\* reservasActuales).**
* **Reservaciones históricas (Reservacion\*\* reservasHistoricas).**
* **Fechas**: aparecen en las reservas activas (fecha check in, número de noches, fecha alojamiento reservado), reservas históricas, alojamiento (fechas futuras) y actualización del histórico (fecha de corte).

Operaciones: se deben comparar fechas, verificar disponibilidad, generar la fecha de check out, validar que para una misma fecha un huésped no tenga 2 reservas. Para ello, hacer una clase fecha con formato DD/MM/AAAA y atributos de día, mes y año. Los métodos serían las operaciones que se realizan con las fechas: menor, igual, sumarDias.

**DEFINICIÓN DE CLASES**

* 1. **Clase Fecha:** representa una fecha y permite comparaciones y operaciones entre fechas

*Atributos*:

* + - int dia;
    - int mes;
    - int anio;
    - bool reservado;

*Métodos*:

* + - bool esIgual(Fecha f);
    - bool validarFecha(Fecha f);
    - void sumarDias(int n); // suma días a la fecha actual
    - void leerDesdeCadena(const char\* str); // formato "DD/MM/AAAA"
  1. **Anfitrión**: representa a una persona que ofrece alojamientos.

*Atributos*:

* + - char documento[15];
    - int antiguedad;
    - float puntuacion;
    - int cantidadAlojamientos;

*Métodos:*

* + - void cargarDesdeLinea(const char\* linea);
    - std::string aLinea(); //Convertir objeto en línea de texto para escribir en archivo
    - void actualizarPuntuacion(float nuevaPuntuacion);
  1. **Huésped**: representa a una persona que realiza reservaciones.

*Atributos*:

* + - char documento[15];
    - char nombre[50];
    - char amenidades[5][4]; // máximo 5 amenidades, codificadas con 3 letras + null
    - int cantidadAmenidades;

*Métodos:*

* + - void cargarDesdeLinea(const char\* linea);
    - std::string aLinea(); //Convertir objeto en línea de texto para escribir en archivo
    - bool deseaAmenidad(const char\* codigo);
  1. **Alojamiento**: representa un lugar ofrecido por un anfitrión.

*Atributos*:

* + - char codigo[10];
    - char nombre[50];
    - char ubicacion[30];
    - char tipo[15]; // casa, apartamento, cabaña, etc.
    - float precioPorNoche;
    - char documentoAnfitrion[15];
    - Fecha fechasReservadas[50]; // máximo 50 reservas por alojamiento
    - int cantNoches[50];
    - int cantidadReservas;

*Métodos:*

* + - void cargarDesdeLinea(const char\* linea);
    - std::string aLinea(); //Convertir objeto en línea de texto para escribir en archivo
    - bool estaDisponible(Fecha entrada, int noches);
    - void registrarReserva(Fecha entrada, int noches);
  1. **Reserva**: representa una reservación activa o histórica.

*Atributos*:

* + - char codigo[10];
    - char codAlojamiento[10];
    - char documentoHuesped[15];
    - Fecha fechaEntrada;
    - int cantNoches;
    - char metodoPago[10]; // tarjeta, efectivo, etc.
    - Fecha fechaReserva;
    - float costo;
    - char anotacion[1000];

*Métodos:*

* + - void cargarDesdeLinea(const char\* linea);
    - void convertirObjetoLineaTextoLinea(char\* buffer);
    - Fecha calcularFechaFin();
    - bool esAnteriorA(Fecha corte); // para pasar a histórico
  1. **Memoria:**

*Atributos*:

* + - char nombre[15] //Objeto al que se le contara la memoria
    - size\_t resultado

*Métodos:*

* + - void tamanioObjeto(objeto)
    - void getResultado()
    - Memoria ()
    - ~Memoria ()

**NOTA**: funciones auxiliares para leer y escribir archivos.

**¿Cuándo usar punteros (\*)?**

Usas punteros cuando:

* Quieres **modificar directamente el objeto original** (es decir, alterar su contenido en la memoria).
* Necesitas manejar **estructuras dinámicas o enlazadas**, como listas de vecinos, donde los elementos pueden cambiar, eliminarse o conectarse dinámicamente.
* Quieres que múltiples partes del programa compartan **el mismo objeto**, sin crear copias (como un router en varios mapas o listas).
* **Punteros (\*)** si necesitas manejar conexiones o estructuras que pueden crecer o cambiar, como relaciones entre objetos (por ejemplo, árboles, grafos o redes).

**¿Cuándo usar referencias (&)?**

Usas referencias cuando:

* Quieres **usar o leer un objeto sin modificarlo**, y sin copiarlo.
* Quieres modificar el objeto pero de una forma más segura (las referencias no pueden ser nulas).
* Trabajas con **estructuras grandes** (como mapas, vectores, etc.) y deseas evitar el costo de copia.
* **Referencias (&)** si solo necesitas acceder (y quizás modificar) los datos directamente.

Entonces, en estructuras dinámicas:

* Usa punteros si necesitas enlaces entre objetos, jerarquías, o flexibilidad.
* Usa referencias si solo quieres operar sobre la estructura directamente sin duplicarla.