Borrador análisis del desafío II de Informática II  
Alejandro Naranjo Naranjo – 1041440121   
  
A) Analisis del problema y consideraciones para la solución propuesta:  
1) Contextualización del problema:  
Lo que tenemos que hacer se basa en un sistema de UdeAStay, una plataforma orientada en el mercado de estadías hogareñas en Antioquia.   
Similar a servicios como Airbnb, UdeAStay permite a anfitriones ofrecer alojamientos y a huéspedes reservarlos por períodos definidos. El sistema debe gestionar eficientemente **alojamientos, reservaciones, anfitriones y huéspedes**, mediante un menú interactivo, utilizando **Programación Orientada a Objetos en C++**, sin el uso de la STL y con manejo de memoria dinámica.

El sistema debe permitir:

* Iniciar sesión como anfitrión o huésped
* Reservar y anular alojamientos
* Consultar reservaciones
* Filtrar búsquedas por ciudad, precio, puntuación
* Mantener un histórico (historial)
* Medir recursos usados (memoria e iteraciones)

2) Identificación de identidades y sus relaciones:  
**Clases principales:**

* Usuario (base)
  + Anfitrion (hereda de Usuario)
  + Huesped (hereda de Usuario)
* Alojamiento
* Reservacion
* Amenidad (podría ser un arreglo o enum dentro de Alojamiento)
* Sistema o UdeAStay (gestiona todo)

**Relaciones:**

* Un **anfitrión** puede tener múltiples alojamientos.
* Un **huésped** puede tener varias reservaciones, sin que se crucen fechas.
* Un **alojamiento** puede tener varias reservaciones.
* Cada **reservación** está asociada a un huésped y a un alojamiento.

vamos a empezar con una declaracion temprana de las clases, y ya lo vamos ajustando según lo veamos conveniente, haciendo diferentes cambios:  
  
Clase Usuario (Base)  
Atributos:  
 char\* documento;

int antiguedad; // en meses

float puntuacion; // de 0.0 a 5.0

Metodos:  
 Constructor (por defecto y con parámetros)

Constructor de copia

Destructor

getters y setters

virtual void mostrarMenu() = 0;

Relacion:  
Clase base de Huesped y Anfitrion

Clase Huesped (hereda de usuario)  
Atributos:  
 Reservacion\*\* reservaciones;

int cantidadReservas;

Metodos:  
 hacerReservacion(...)

anularReservacion(...)

mostrarReservaciones()

bool tieneConflictoFechas(fechaInicio, duracion)

void mostrarMenu() override;  
  
Relacion:  
Tiene una lista de Reservacion\*

Clase Anfitrion (hereda de usurio)  
Atributos:  
 Alojamiento\*\* alojamientos;

int cantidadAlojamientos;  
  
Metodos:  
 consultarReservasRango(...)

anularReservacion(...)

void mostrarMenu() override;  
  
Relacion:  
Tiene una lista de Alojamiento\*  
  
Clase Alojamiento  
Metodos:  
 char\* nombre;

char\* codigo;

char\* direccion;

char\* municipio;

char\* departamento;

char tipo; // 'C' casa, 'A' apartamento

float precioPorNoche;

char\*\* amenidades;

int cantidadAmenidades;

Reservacion\*\* reservas;

int cantidadReservas;

Anfitrion\* anfitrion;

Atributos:  
 bool estaDisponible(fechaInicio, duracion)

void agregarReservacion(Reservacion\*)

void eliminarReservacionPorCodigo(char\* cod)

void mostrarDetalles()  
  
Relacion:  
Pertenece a un Anfitrion

Tiene varias Reservacion\*  
  
  
Clase Reservacion:

Atributos:  
 char\* codigo;

char\* fechaEntrada; // formato: YYYY-MM-DD

int duracion;

char\* metodoPago;

char\* fechaPago;

float monto;

char\* anotacion; // hasta 1000 caracteres

Huesped\* huesped;

Alojamiento\* alojamiento;

Metodos:  
 void mostrarComprobante();

bool intersectaCon(fecha, duracion);

bool esAnteriorA(fechaCorte);

Relacion:  
  
Tiene un Huesped\* y un Alojamiento\*

Clase Sistema:  
Atribitos:  
 Huesped\*\* huespedes;

int cantidadHuespedes;

Anfitrion\*\* anfitriones;

int cantidadAnfitriones;

Alojamiento\*\* alojamientos;

int cantidadAlojamientos;

Reservacion\*\* historicoReservas;

int cantidadHistorico;

int contadorIteraciones;

int totalMemoriaConsumida;  
  
Metodos:  
 void cargarDatos();

void guardarDatos();

Usuario\* login(char\* documento);

void reservarAlojamiento(...)

void anularReservacion(...)

void consultarReservacionesAnfitrion(...)

void actualizarHistorico(char\* fechaCorte);

void medirRecursos();

void mostrarMenuPrincipal();

Relacion:  
Tiene acceso a todos los usuarios, alojamientos y reservaciones  
  
porque se eligieron estas clases y demás?  
  
primero, se arranco por las clases mas obvias o por decirlo así mas evidentes, como lo era la clase usuario, con el huésped y el anfitrión, esto favorece mucho lo que sería la reutilización del código, y nos permite usar lo que seria el **polimorfismo** con métodos como mostrarMenu()  
  
por que usar punteros dobles y memoria dinámica?  
  
dado que no conocemos la capacidad como tal de lo que son por ejemplo los alojamientos, reservas, etc habrá, se eligió usar como tal memoria dinámica, primero, porque como tal es un requisito del desafío, y porque este nos va a permitir usar tanta memoria en tiempo real como necesitemos.  
La elección de usar doble puntero, en lugar de por ejemplo optar por una solución mas simple como un puntero a un arreglo dinámico ya que es una forma mucho mas flexible, que nos va a permitir, por ejemplo, con los alojamientos, tratarlos de manera independiente uno por uno, que es justamente lo que se necesita, y poder por ejemplo agregar o eliminar el que queramos sin ningún tipo de problema.   
  
Eficiencia:  
  
Yo considero, que bajo, lo que es, por ejemplo, los punteros dobles, se hace un buen uso de lo que sería la memoria dinámica y por tanto la eficiencia como tal.  
Se evita lo que son operaciones innecesarias como lo serias duplicaciones.   
No hay operaciones que impliquen un costo en la eficiencia como por ejemplo reordenamientos ni estructuras genéricas que se usen de manera innecesaria.  
  
Al usar referencias en los punteros, entre entidades, se evita lo que seria el duplicado de la información que se maneje.  
  
B) Diagrama de clases de la solución planteada:  
1 Que clases hay

2 Que representa cada una

3 Como se relacionan entre si

4 Que tipo de relación tienen (herencia, composición, asociación, etc)

|  |  |  |
| --- | --- | --- |
| Clase | Tipo | Representa a |
| Usuario | Clase base | Cualquier persona en la plataforma |
| Huesped | Derivada | Persona que hace las reservas |
| Anfitrion | Derivada | Persona que ofrece los alojamientos |
| Alojamiento | Compuesta | Lugar que se puede reservar |
| Reservacion | Asociada | La reserva hecha por un huésped |
| Sistema | Controladora | Administra todo el funcionamiento del sistema |

Diagrama de clases: (borrador)

|  |
| --- |
| Usuario |
| - documento: char\*  - antiguedad: int  - puntuacion: float |
| + mostrarMenu(): virtual |

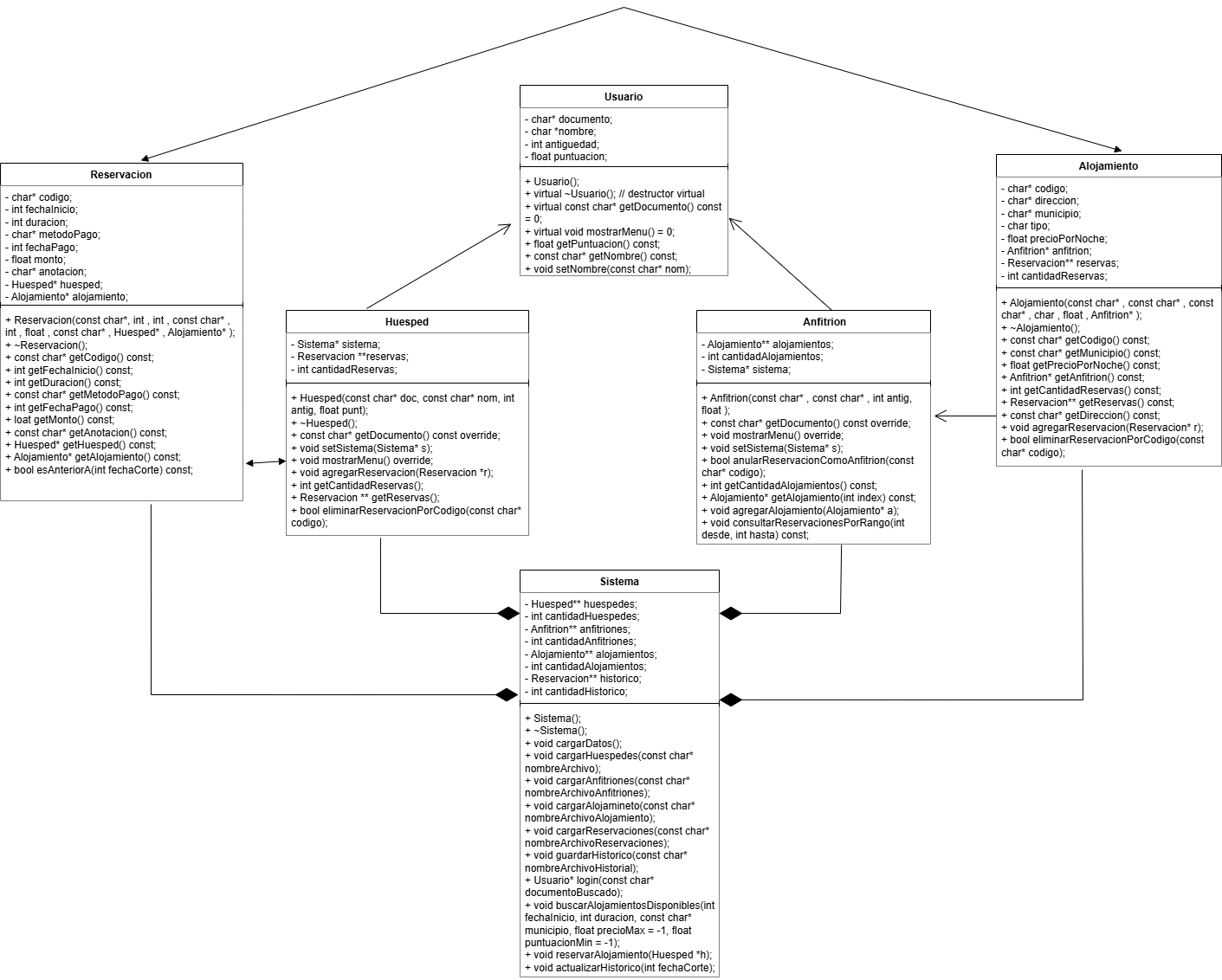
|  |
| --- |
| Anfitrion |
| - alojamientos: Alojamiento  - cantAlojamientos: int |
| + mostrarMenu(): virtual |

|  |
| --- |
| Huesped |
| - reservas: Reservacion\*\*│  - cantReservas: int |
|  |

|  |
| --- |
| Reservacion |
| - codigo: char\*  - fechaInicio: Fecha  - duracion: int  - metodoPago: char\*  - monto: float  - anotacion: char\*  - huesped: Huesped\*  - alojamiento: Alojamiento\* |
|  |

|  |
| --- |
| Alojamiento |
| - codigo: char\*  - direccion: char\*  - municipio: char\*  - tipo: char  - precioNoche: float  - amenidades: char\*\*  - anfitrion: Anfitrion\*  - reservas: Reservacion\*\*  - cantReservas: int |
|  |

|  |
| --- |
| Sistema |
| - huespedes: Huesped\*\*  - anfitriones: Anfitrion\*\*  - alojamientos: Alojamiento\*\*  - historico: Reservacion\*\*  - contadorIteraciones: int  - memoriaUsada: int |
| + login()  + reservarAlojamiento()  + anularReservacion()  + consultarPorRango()  + actualizarHistorico() |

Diagrama de clases:  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
C Descripción en alto nivel de la lógica de tareas no triviales   
  
Una de las tareas no triviales, tiene que ver con las fechas de las reservaciones, es decir, asegurarse de que no se crucen fechas entre las distintas reservaciones para un huésped o un alojamiento.  
  
Hay que verificarlo en dos lados:

Que el alojamiento esté libre en esas fechas.  
Que el huésped no tenga otra reservación en ese mismo rango.  
  
Esta debe hacerse para cada reservación que se tenga activa en el momento, lo que puede crecer bastante si hay muchos datos.  
  
Ejemplo de un caso conflictivo:  
Supón que un huésped ya tiene una reserva así:

* Fecha de entrada: 2025-06-10
* Duración: 4 noches → termina en 2025-06-14

Ahora quiere reservar otro alojamiento desde 2025-06-12 por 2 noches.  
  
Aquí tenemos un conflicto, porque 2025-06-12 ya esta ocupada por otra reservación del mismo huésped  
  
ahora, como lo resolvemos, primero tenemos que tener en cuenta que las fechas serán ingresadas, por ejemplo, asi: 20250612, ya con esto claro, tenemos el código:  
  
bool esBisiesto(int anio) {

return (anio % 4 == 0 && anio % 100 != 0) || (anio % 400 == 0);

}

int diasEnMes(int mes, int anio) {

int diasMes[] = {31, 28, 31, 30, 31, 30,

31, 31, 30, 31, 30, 31};

if (mes == 2 && esBisiesto(anio)) return 29;

return diasMes[mes - 1];

}

// Convierte AAAAMMDD en año, mes, día

void descomponerFecha(int fecha, int& anio, int& mes, int& dia) {

anio = fecha / 10000;

mes = (fecha % 10000) / 100;

dia = fecha % 100;

}

// Suma 'n' días a una fecha (formato AAAAMMDD)

int sumarDias(int fecha, int n) {

int anio, mes, dia;

descomponerFecha(fecha, anio, mes, dia);

while (n > 0) {

int diasMes = diasEnMes(mes, anio);

if (dia + n <= diasMes) {

dia += n;

break;

} else {

n -= (diasMes - dia + 1);

dia = 1;

mes++;

if (mes > 12) {

mes = 1;

anio++;

}

}

}

return anio \* 10000 + mes \* 100 + dia;

}   
  
luego, se hace la comparativa de lo que serian como tal las fechas:  
  
bool fechasSeCruzan(int inicio1, int duracion1, int inicio2, int duracion2) {

int fin1 = sumarDias(inicio1, duracion1);

int fin2 = sumarDias(inicio2, duracion2);

// No se cruzan si una termina antes o empieza después de la otra

if (fin1 <= inicio2 || fin2 <= inicio1) {

return false;

}

return true;

}  
  
tenemos como tal lo que seria un ejemplo del uso de esta:  
  
int main() {

int r1\_inicio = 20250610;

int r1\_duracion = 4; // 10 al 14

int r2\_inicio = 20250612;

int r2\_duracion = 2; // 12 al 14

if (fechasSeCruzan(r1\_inicio, r1\_duracion, r2\_inicio, r2\_duracion)) {

cout << “Las fechas se cruzan" << endl;

} else {

cout << "Las fechas están libres" << endl;

}

return 0;

}  
  
D Algoritmos implementados debidamente intra-documentados

E Problemas de desarrollo que se afrontó:  
uno de los problemas que se afronto fue el dilema por el uso de la memoria dinámica, en sí, de lo que seria el uso de punteros.  
Al final se opto por el uso de punteros dobles ya que es más flexible y cómodo a la hora de trabajar con ellos, el cual nos va a permitir manejar cada, por ejemplo, reservación, de manera independiente, cosa que no se nos permitía con un puntero a lo que seria un arreglo dinámico.  
  
También se tuvo problemas con lo que fue el manejo de lo que son los archivos, lo que es la lectura y escritura de los mismos   
  
De igual forma, mantener la independencia entre entidades sin hacer duplicaciones o copias de los datos   
  
F Evolución de la solución y consideraciones para la implementación   
  
Etapa inicial:  
Bueno, como tal solo se ha planteado la idea y el análisis como tal de lo que sería la solución del desafío II, lógicamente, ya con el pasar de los días se procederá con lo que es, en sí, lo más importante, por lo que este documento puede presentar diferentes cambios.   
  
Etapa de diseño:  
Con el borrador del diseño del diagrama de clases, se valido la viabilidad de la solución.

Etapa de codificación:  
  
Se comenzó con las clases tempranas, lo que serían entonces usuario, huésped, anfitrión, alojamiento, reservación y sistema   
  
Futuras mejoras:

Manejo de fechas con una clase Fecha.

Sustitución de char\* por una clase ligera tipo Cadena para evitar errores con cadenas C.