Informe Desafío II

Por:

Isabela Aguirre Ceballos

Juan Felipe Higuita Perez

Informática II

Universidad de Antioquia

2025

Análisis del problema

En este desafío se plantea el desarrollo de una aplicación llamada **UdeAStay**, orientada a facilitar el alquiler temporal de alojamientos hogareños entre usuarios, de manera similar al funcionamiento de plataformas existentes como Airbnb. El sistema debe permitir que dos tipos de usuarios interactúen: los **huéspedes**, quienes buscan un lugar para alojarse, y los **anfitriones**, quienes ofrecen propiedades en arriendo.

Cada tipo de usuario accede a funcionalidades específicas mediante un menú adaptado a su perfil. Los huéspedes pueden buscar alojamientos según criterios como la fecha de entrada, el municipio de interés y la cantidad de noches que desean hospedarse. Opcionalmente, pueden aplicar filtros como precio máximo por noche o puntuación mínima del anfitrión, y posteriormente realizar una reserva si encuentran una opción que se ajuste a sus preferencias.

Por su parte, los anfitriones pueden visualizar todas las reservaciones activas de sus alojamientos y administrar el histórico de reservaciones a través de una fecha de corte, que les permite mover las reservas pasadas a un archivo histórico para mantener el sistema organizado y eficiente.

El sistema debe permitir guardar la información más importante como los datos de usuarios, alojamientos y reservaciones de forma permanente, usando archivos externos simples. Estos archivos deben tener un formato claro y fácil de leer y escribir, para que el programa pueda recuperar los datos al iniciar y actualizar los cambios cuando sea necesario. Aunque no se busca construir una plataforma profesional, esta solución básica simula las funciones esenciales de un sistema de reservas, con el fin de aplicar los conceptos de programación orientada a objetos en un contexto cercano a la vida real.

Consideraciones para la alternativa de solución propuesta

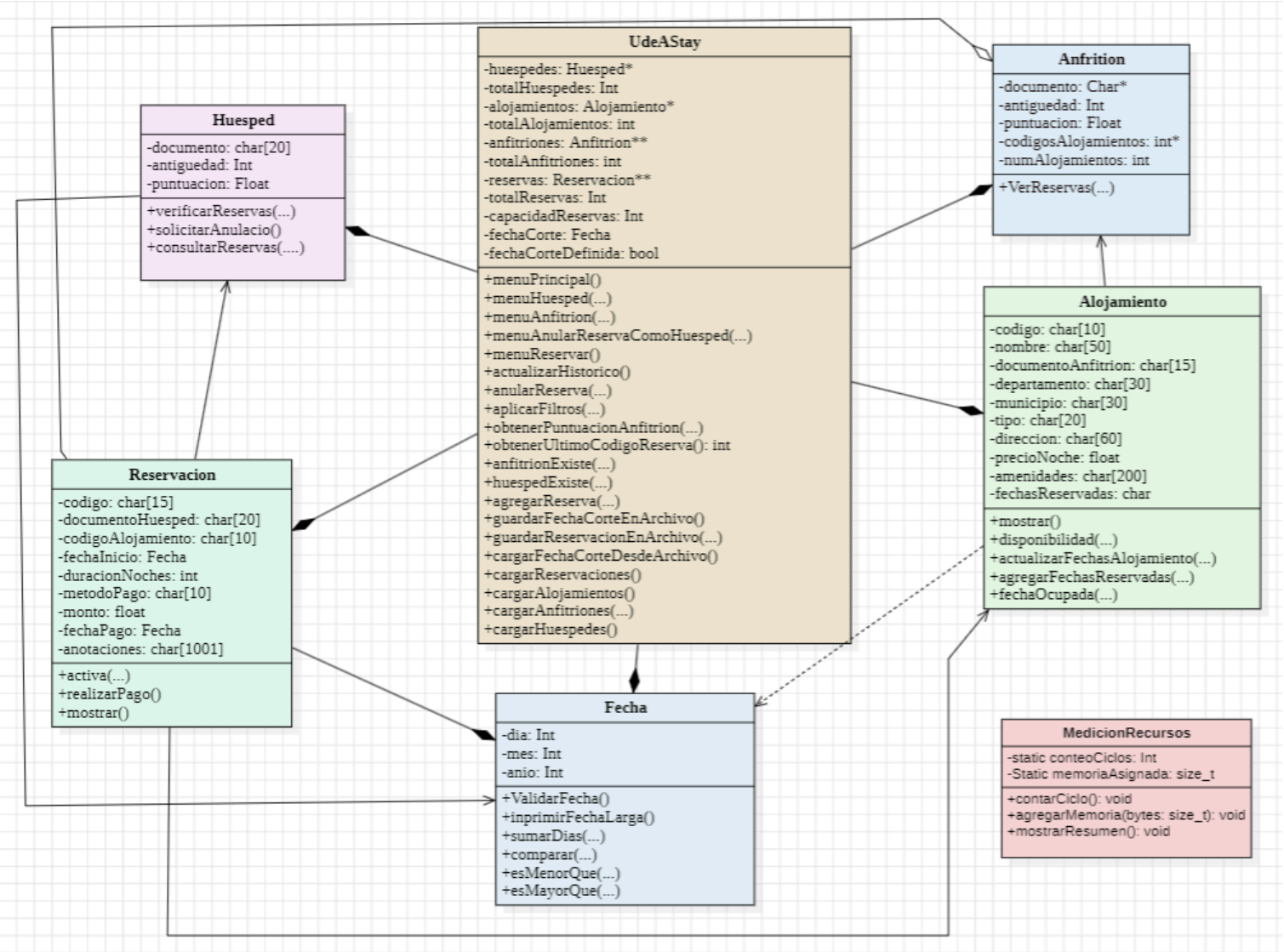
La solución propuesta para este desafío es desarrollar una aplicación en C ++ usando programación orientada a objetos (POO).

Se crearán varias clases que representen las partes principales del sistema: **huéspedes, anfitriones, alojamientos, reservaciones**, y también clases de apoyo como **Fecha** para organizar mejor la información y facilitar las validaciones. Cada clase tendrá sus propios atributos y funciones. Todas estarán diseñadas usando encapsulación, de modo que los datos se mantengan protegidos y solo se pueda acceder a ellos a través de funciones públicas.

- La clase **Huesped** tendrá un documento y una lista de códigos de reservaciones hechas.
- La clase **Anfitrion** almacenará el documento y una lista de códigos de alojamientos que administra.
- La clase **Alojamiento** tendrá su propio código, tipo (casa o apartamento), ubicación, precio por noche, amenidades y un registro de fechas reservadas.
- La clase **Reservacion** incluirá el código de reserva, fecha de inicio, duración, código del alojamiento, documento del huésped, método de pago, monto y anotaciones de un máximo de 1000 caracteres.
- La clase **Fecha** permitirá representar una fecha con día, mes y año, además de incluir funciones para comparar fechas, sumar días y validar rangos.

Finalmente, se incluirá una clase **UdeAStay** encargada de controlar el menú del sistema, realizar las validaciones necesarias, coordinar las funcionalidades según el rol del usuario que ingrese a la aplicación (Huésped o Anfitrión), y manejar la lectura y escritura de archivos para guardar los datos de manera permanente.

Diagrama de clases



Descripción en alto nivel de la lógica de tareas no triviales

menuReservar()

Esta función guía al usuario huésped en el proceso de reservar un alojamiento:

- Verifica la fecha de corte y los filtros de búsqueda.
- Busca alojamientos disponibles por municipio y filtros o por código de alojamiento.
- Valida si el huésped ya tiene reservas en esa fecha.
- Genera una reserva si se cumple todo y registra el pago.

actualizarHistorico()

Se encarga de:

- **V**erificar si cada reserva ya ha finalizado (comparando con la fecha de corte).
- Mover esas reservas a Historico.txt.
- Eliminar de memoria (reservas[]).
- Eliminar del archivo Reservaciones_activas.txt.
- Actualizar las fechas reservadas en Alojamientos.txt borrando solo las fechas vencidas.

anularReserva()

Esta función permite que el usuario anule una reservación activa. Su lógica es la siguiente:

- Recorre el archivo Reservaciones_activas.txt y copia todas las líneas a un archivo temporal, excepto la línea con el código de la reserva a anular.
- Extrae los datos clave de la reserva eliminada: código de alojamiento, fecha de inicio y duración.
- Elimina esa reserva del arreglo en memoria dinámica (reservas[]) para liberar espacio.
- Llama a actualizarFechasAlojamiento() para borrar esas fechas del alojamiento afectado.

- Renombra temp.txt como Reservaciones_activas.txt para conservar los cambios.Usa MedicionRecursos::contarCiclo() para registrar el uso de ciclos de procesamiento

Algoritmos implementados intra-documentados

Durante el desarrollo del programa se implementaron varios algoritmos bastante importantes para el buen funcionamiento del sistema. Para lograr una buena comunicación y el que el compañero supiera que hacía cada línea comentamos un poco por encima su funcionamiento. A continuación, se describirán todos o al menos la gran mayoría:

Funciones de cargar .txt:

En las funciones cargarHuespedes, cargarAnfitriones, cargarAlojamientos y cargarReservaciones, se implementaron algoritmos que abren los archivos, leen línea por línea, y separan los datos por ; con strtok para así almacenarlos de acuerdo al tipo de dato que corresponden de acuerdo al .txt del que provengan y así poder usarlos en un futuro del programa. A continuación se mostrará el código de una de estas funciones comentado con algunas líneas importantes de su función.

```
void UdeAStay::cargarHuespedes() {
    FILE* archivo = fopen("Huespedes.txt", "r"); //Abre el archivo Huespedes.txt en modo lectura.
    if (archivo == NULL) {
        cout << "No se pudo abrir el archivo huespedes.txt" << endl;
        return; //Si no se pudo abrir el archivo, se muestra un mensaje de error y se sale de la función.
    }
    char linea[256];
    int contador = 0;
    while (fgets(linea, 256, archivo)) {
        MedicionRecursos::contarCiclo(); //Se registra un ciclo de procesamiento.
        if (strlen(linea) > 1) contador++; //Lee línea por línea para contar cuántos huéspedes válidos hay.
    }
    if (contador == 0) {
        cout << "Archivo vacio o sin lineas validas." << endl;
        fclose(archivo);
        return; //Si no hay líneas válidas, se muestra un mensaje y se termina la función cerrando el archivo.
    }
    rewind(archivo); //Se regresa el puntero al inicio del archivo.
    huespedes = new Huesped[contador]; //Se reserva memoria dinámica para almacenar los huéspedes.
    MedicionRecursos::agregarMemoria(contador*sizeof(huespedes)); //Se hace registro de la memoria usada al hacer lo anterior
    totalHuespedes = 0;
    while (fgets(linea, 256, archivo)) { //Lee cada línea del archivo eliminando caracteres de fin de línea.
        MedicionRecursos::contarCiclo(); //Se hace registro del bucle
        linea[strcspn(linea, "\r\n")] = 0; // Elimina salto de línea

        char* token = strtok(linea, ";");
        if (token == NULL) continue;

        char doc[20];
        strcpy(doc, token); //Extrae y copia el documento del huésped.

        token = strtok(NULL, ";");
        if (token == NULL) continue;
        int ant = atoi(token); //Extrae la antigüedad como entero.
```

```
        token = strtok(NULL, ";");
        if (token == NULL) continue;
        float punt = atof(token); //Extrae la puntuación como flotante.

        if (ant < 0 || punt < 0.0f || punt > 5.0f) {
            cout << "Linea invalida: " << doc << endl;
            continue; //Valida todos los datos y si hay errores los ignora.
        }
        huespedes[totalHuespedes] = Huesped(doc, ant, punt);
        totalHuespedes++; //Crea y guarda el huésped en el arreglo
    }
    fclose(archivo); //Cierra el archivo.
}
```

Alojamiento::disponiblidad

En esta función se recorre la lista de reservas y se verifica que no haya alguna reserva ya ocupada dentro de la fecha en la que se quiere hacer la nueva reserva del alojamiento deseado siendo que para este proceso nos ayuda la clase fecha para comparar ambas fechas si es que es son iguales en caso de haya una reserva en el alojamiento.

```
bool Alojamiento::disponibilidad(const Fecha& nuevaEntrada, int duracion, Reservacion** reservas, int cantidadReservas) const {
    for (int i = 0; i < duracion; i++) {// Verifica si alguno de los días solicitados ya está ocupado manualmente
        MedicionRecursos::contarCiclo(); // Cuenta el ciclo para fines de medición de recursos
        Fecha dia = nuevaEntrada.sumarDias(i); // Calcula cada uno de los días del período solicitado
        if (fechaOcupada(dia)) { // Verifica si ese día ya está marcado como ocupado
            return false; // Si algún día está ocupado, no hay disponibilidad
        }
    }

    for (int i = 0; i < cantidadReservas; i++) {// Verifica conflictos con reservaciones existentes
        MedicionRecursos::contarCiclo(); // Cuenta el ciclo de procesamiento
        Reservacion* res = reservas[i]; // Obtiene la reservación actual

        if (strcmp(res->getCodigoAlojamiento(), codigo) == 0) { // Verifica si la reserva corresponde al mismo alojamiento
            Fecha resInicio = res->getFechaInicio(); // Fecha de inicio de la reserva actual
            Fecha resSalida = resInicio.sumarDias(res->getDuracion() - 1); // Fecha de salida de la reserva actual
            Fecha nuevaSalida = nuevaEntrada.sumarDias(duracion - 1); // Fecha de salida de la nueva solicitud

            bool noSeSolapan = nuevaSalida.esMenorQue(resInicio) || resSalida.esMenorQue(nuevaEntrada); // Verifica si las fechas no se juntan en algun momento

            if (!noSeSolapan) {
                return false; // Si se juntan o cruzan en alguna momento
            }
        }
    }

    return true; // Si no hay días ocupados ni reservas que se crucen, el alojamiento está disponible
}
```

Medición De Recursos

Para saber la cantidad de recursos que consume el programa se realizó un algoritmo el cual cuenta el total de iteraciones que hay durante todo el tiempo que el programa está activo contando todos los bucles que se pasan internamente, además de poseer otra función para que cada vez se cree un nuevo espacio en la memoria con new se calcule cuántos bytes se consume esta para así al final de programa muestre un resumen de todo el total consumido.

Durante la implementación del sistema UdeAStay, se detectaron múltiples invocaciones a funciones de la biblioteca estándar del lenguaje. Estas funciones se ejecutan muchas veces dentro de ciclos.

cargarHuespedes :.
22 iteraciones + 22 de fgets + 11 de strlen + 11 de strcspn + 33 de strtok + 11 de strcpy + 11 de atoi + 11 de atof
Memoria consumida: 352 bytes (11 × sizeof(Huesped)

cargarAnfitriones: 20 iteraciones + 20 de fgets + 20 de strcspn + 110 de strtok + 50 de atoi + 20 de atof

Memoria consumida: 960 bytes (20 × sizeof(Anfitrion))

cargarAlojamientos: 40 iteraciones + 40 de fgets + 20 de strlen + 20 de strcspn + 100 de strtok + 90 de strncpy + 20 de atof
Memoria consumida: 1600 bytes (20 × sizeof(Alojamiento))

cargarReservaciones:5 iteraciones + 5 de fgets + 5 de strcspn + 45 de strtok + 35 de strncpy + 10 de sscanf + 5 de atoi + 5 de atof
Memoria consumida: 320 bytes (5 × sizeof(Reservacion))

anularReserva: 5 iteraciones del ciclo while para leer líneas + 5 de fgets + 5 de strcspn + 15 de strtok + 5 de strcmp + 5 de strcpy + 5 de sscanf + 5 de atoi + 4 de fputs (una por línea que no es la reservación a eliminar)
5 iteraciones del primer ciclo for (para buscar y eliminar la reserva en memoria) + 5 de strcmp + 4 iteraciones del segundo ciclo for (para actualizar alojamiento) + 4 de strcmp

Memoria consumida:
Solo elimina 1 Reservacion (sizeof(Reservacion) bytes liberados). No se añade memoria nueva, solo gestión de memoria al borrar.

actualizarHistorico: 5 iteraciones + 5 de fgets + 5 de strcpy + 5 de strcspn + 30 de strtok + 5 de sscanf + 5 de atoi + 3 de fputs + 8 de contarCiclo + 3 llamadas a anularReserva**Memoria consumida**: No asigna memoria nueva directamente (usa memoria ya gestionada)

Problemas de desarrollo que afrontamos

Durante el desarrollo tuvimos algunas dudas que pudimos resolver después:

- **Fecha de corte compartida**: No sabíamos si debía ser por anfitrión o global. Se optó por una fecha de corte global para simplificar la validación y permitir mantener el sistema más limpio esta fecha se cambia cuando un anfitrión ingresa una nueva y no se puedan hacer reservas mientras no haya una.
- **Modo de apertura de archivos**: Un pequeño problema que hubo es que algunas veces cuando se intentaba pasar las reservas activas al histórico y eliminar las fechas de reservas del alojamiento se borraba todas las reservas activas que habian en el txt

En varias funciones se perdía información por abrir archivos en modo W. Se corrigió usando r, a, w o archivos temporales temp.txt.

- **Control de fechas**: asegurar que no se solaparan fechas fue complejo, pero se resolvió con comparaciones entre fechas de entrada y salida.

Evolución de la solución y consideraciones para la implementación

Desde el inicio, se planteó una estructura clara para organizar los datos en archivos .txt, lo que facilitó tanto la lectura como la escritura de la información en el sistema:

- **Huespedes.txt**: contiene el documento de identidad del huésped, su antigüedad (en meses) y la puntuación que ha recibido.
- **Anfitriones.txt**: almacena el documento, la antigüedad, la puntuación y los códigos de los alojamientos que administra cada anfitrión.
- **Alojamientos.txt**: guarda código de alojamiento, nombre de alojamiento, documento del anfitrión al que pertenece, departamento, municipio, tipo de alojamiento, dirección, precio de la noche, sus amenidades y fechas que han sido reservadas si hay alguna reserva
- **Reservaciones_activas.txt**: contiene los datos de cada reserva activa: código, documento del huésped, código del alojamiento, fecha de inicio, duración, método de pago, monto total, fecha de pago y anotaciones.
- **Historico.txt**: almacena las reservas ya finalizadas con la misma estructura que las reservas activas.

Inicialmente, nos enfocamos en diseñar la estructura general del sistema: las clases principales y su interacción. Posteriormente, desarrollamos las funciones básicas como carga de datos, validaciones y menús de navegación.

A medida que avanzamos, agregamos funcionalidades más completas:

- Visualización de reservas activas para anfitriones.
- Actualización del histórico, conservando datos pero eliminando solo fechas vencidas del alojamiento.
- Sistema de reservas con validaciones por disponibilidad, fechas y filtros como precio y puntuación.
- Anulación de reservas, asegurando la eliminación en memoria, archivos y fechas de los alojamientos.
- Validaciones sólidas y manejo de archivos temporales para evitar pérdidas de información.

Finalmente, se integró la funcionalidad de **medición de recursos**, que permite monitorear el comportamiento del sistema en tiempo real. Esto incluye:

- El número de ciclos ejecutados en operaciones clave, para tener una idea de la complejidad de cada tarea.
- El total de memoria dinámica consumida por objetos creados con **new**, lo que permite evaluar la eficiencia de la solución.