Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Ingeniería en Ciencias y Sistemas
Catedrático: Ing. Rene Ornelis
Auxiliar: Daniel Monterroso
Curso: Estructura de datos
Sección: “A”

Proyecto

Nombre:
Alexis José Trujillo Vásquez

Carnet:
202401884

Manual Técnico-Sistema de Gestión de Aeropuerto

1. Requerimientos del sistema:

- Windows 10/11 (64 bits) - Recomendado
- Linux (Ubuntu 20.04 o superior)
- macOS (Big Sur o superior)

2. Dependencias y herramientas:

2.1 Compilador de C++:

Requerido: Compilador compatible con C++11 o superior.

Windows:

- **MinGW-w64** (g++ 8.1.0 o superior)
- **MSYS2** con g++
- **Visual Studio 2019/2022** con soporte C++.

2.2 Graphviz

Versión: 2.38 o superior

3. ARQUITECTURA DEL SISTEMA

ListaCircularDoble.h:

```h
#ifndef LISTA_CIRCULAR_DOBLE_H
#define LISTA_CIRCULAR_DOBLE_H

#include "NodoListaCircular.h"
#include <fstream>

class ListaCircularDoble {
private:
    NodoListaCircular* cabeza;  // Apuntador al primer nodo
    int tamanio;                // Contador de elementos

public:
    ListaCircularDoble() : cabeza(nullptr), tamanio(0) {}

    /**
     * Libera toda la memoria de los nodos
     * Debe romper el ciclo antes de eliminar para evitar bucle infinito
     */
    ~ListaCircularDoble() {
        if (cabeza == nullptr) return;

        NodoListaCircular* actual = cabeza;
        do {
            NodoListaCircular* temp = actual;
            actual = actual->siguiente;
            delete temp;  // Liberar memoria de cada nodo
        } while (actual != cabeza);
    }

    /**
     * Inserta un nuevo avión al final de la lista circular
     */
    void insertar(const Avion& avion) {
        NodoListaCircular* nuevo = new NodoListaCircular(avion);

        // Caso 1: Lista vacía
        if (cabeza == nullptr) {
            cabeza = nuevo;
            nuevo->siguiente = nuevo;  // Apunta a sí mismo
            nuevo->anterior = nuevo;   // Apunta a sí mismo
        }
```

```h
        // Caso 2: Lista con elementos
        else {
            NodoListaCircular* ultimo = cabeza->anterior;  // Último nodo

            // Insertar al final y mantener circularidad
            nuevo->siguiente = cabeza;
            nuevo->anterior = ultimo;
            ultimo->siguiente = nuevo;
            cabeza->anterior = nuevo;
        }
        tamanio++;
    }

    /**
     * Elimina un avión de la lista por su número de registro
     * Retorna true si se eliminó, false si no se encontró
     */
    bool eliminar(const char* registro) {
        if (cabeza == nullptr) return false;

        NodoListaCircular* actual = cabeza;
        do {
            // Comparar registro del avión
            if (strcmp(actual->avion.registro, registro) == 0) {
                // Caso especial: Único nodo
                if (actual == cabeza && actual->siguiente == cabeza) {
                    delete actual;
                    cabeza = nullptr;
                }
                // Caso general: Múltiples nodos
                else {
                    // Reconectar nodos adyacentes
                    actual->anterior->siguiente = actual->siguiente;
                    actual->siguiente->anterior = actual->anterior;

                    // Si se elimina la cabeza, actualizarla
                    if (actual == cabeza) {
                        cabeza = actual->siguiente;
```

Concepto General

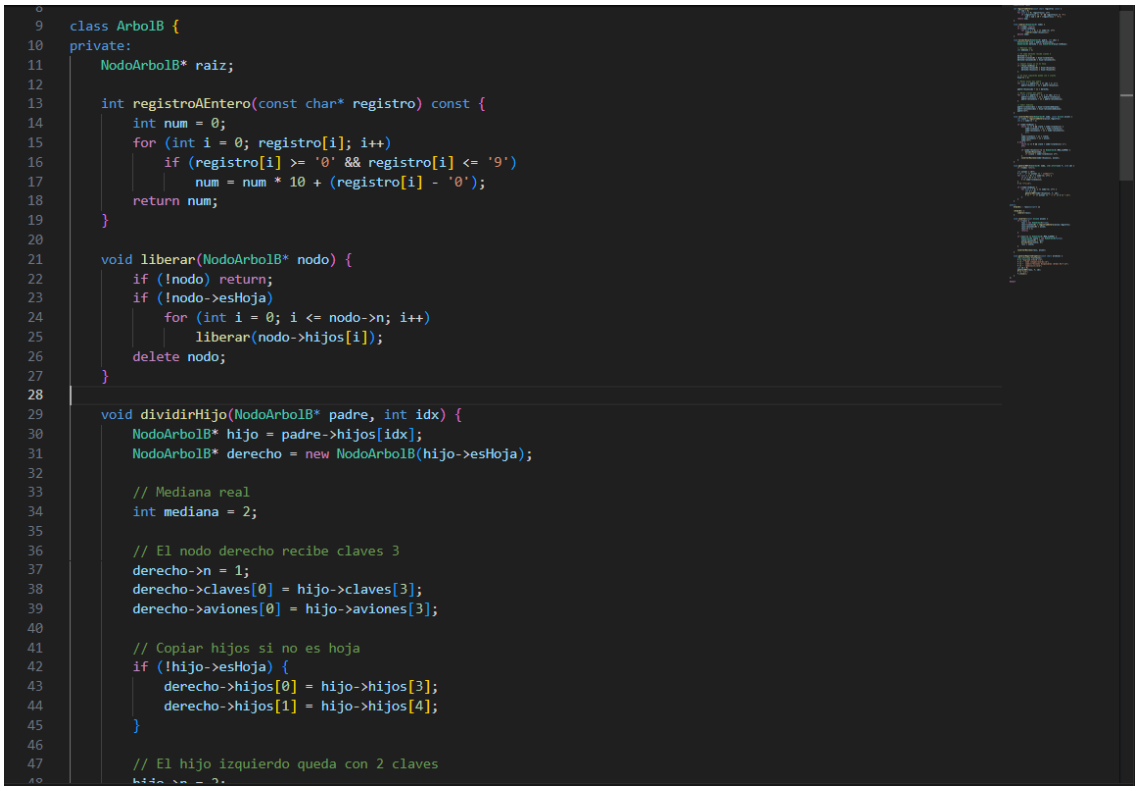
Es una lista doblemente enlazada y circular para gestionar aviones. Combina dos características:

- Doble enlace: Cada nodo apunta al siguiente Y al anterior
- Circular: El último nodo apunta al primero (y viceversa), formando un ciclo

Uso en el Proyecto

- avionesDisponibles: Almacena aviones con estado "Disponible"
- avionesMantenimiento: Almacena aviones con estado "Mantenimiento"

ArbolB.h



```
class ArbolB {
private:
    NodoArbolB* raiz;

    int registroAEntero(const char* registro) const {
        int num = 0;
        for (int i = 0; registro[i]; i++)
            if (registro[i] >= '0' && registro[i] <= '9')
                num = num * 10 + (registro[i] - '0');
        return num;
    }

    void liberar(NodoArbolB* nodo) {
        if (!nodo) return;
        if (!nodo->esHoja)
            for (int i = 0; i <= nodo->n; i++)
                liberar(nodo->hijos[i]);
        delete nodo;
    }

    void dividirHijo(NodoArbolB* padre, int idx) {
        NodoArbolB* hijo = padre->hijos[idx];
        NodoArbolB* derecho = new NodoArbolB(hijo->esHoja);

        // Mediana real
        int mediana = 2;

        // El nodo derecho recibe claves 3
        derecho->n = 1;
        derecho->claves[0] = hijo->claves[3];
        derecho->aviones[0] = hijo->aviones[3];

        // Copiar hijos si no es hoja
        if (!hijo->esHoja) {
            derecho->hijos[0] = hijo->hijos[3];
            derecho->hijos[1] = hijo->hijos[4];
        }

        // El hijo izquierdo queda con 2 claves
```

```
        hijo->n = 2;

        // Mover hijos del padre
        for (int j = padre->n; j >= idx + 1; j--)
            padre->hijos[j + 1] = padre->hijos[j];

        padre->hijos[idx + 1] = derecho;

        // Mover claves del padre
        for (int j = padre->n - 1; j >= idx; j--) {
            padre->claves[j + 1] = padre->claves[j];
            padre->aviones[j + 1] = padre->aviones[j];
        }

        // Subir mediana
        padre->claves[idx] = hijo->claves[mediana];
        padre->aviones[idx] = hijo->aviones[mediana];
        padre->n++;
    }

    void insertarNoLleno(NodoArbolB* nodo, const Avion& avion) {
        int clave = registroAEntero(avion.registro);
        int i = nodo->n - 1;

        if (nodo->esHoja) {
            while (i >= 0 && clave < nodo->claves[i]) {
                nodo->claves[i + 1] = nodo->claves[i];
                nodo->aviones[i + 1] = nodo->aviones[i];
                i--;
            }
            nodo->claves[i + 1] = clave;
            nodo->aviones[i + 1] = avion;
            nodo->n++;
        } else {
            while (i >= 0 && clave < nodo->claves[i]) i--;
            i++;

            if (nodo->hijos[i]->n == NodoArbolB::MAX_CLAVES) {
```

Es un Árbol B de orden 5 auto-balanceado para indexar aviones disponibles. Garantiza que todas las hojas están al mismo nivel y permite búsquedas eficientes.

ArbolBinarioBusqueda.h

```
class ArbolBinarioBusqueda {

    // Insertar recursivamente comparando horas de vuelo para decidir si va a izquierda o derecha
    NodoArbolBinario* insertarRecursivo(NodoArbolBinario* nodo, const Piloto& piloto) {
        if (nodo == nullptr) {
            return new NodoArbolBinario(piloto);  // Encontramos el lugar, crear nodo aquí
        }

        if (piloto.horas_de_vuelo < nodo->piloto.horas_de_vuelo) {
            nodo->izquierdo = insertarRecursivo(nodo->izquierdo, piloto);  // Menos horas, ir a izquierda
        } else if (piloto.horas_de_vuelo > nodo->piloto.horas_de_vuelo) {
            nodo->derecho = insertarRecursivo(nodo->derecho, piloto);  // Más horas, ir a derecha
        } else {
            nodo->derecho = insertarRecursivo(nodo->derecho, piloto);  // Mismas horas, meter a la derecha
        }

        return nodo;
    }

    // Buscar por horas de vuelo navegando el árbol según el valor
    NodoArbolBinario* buscarRecursivo(NodoArbolBinario* nodo, int horas) {
        if (nodo == nullptr || nodo->piloto.horas_de_vuelo == horas) {
            return nodo;  // Lo encontramos o no existe
        }

        if (horas < nodo->piloto.horas_de_vuelo) {
            return buscarRecursivo(nodo->izquierdo, horas);  // Buscar en rama izquierda
        }

        return buscarRecursivo(nodo->derecho, horas);  // Buscar en rama derecha
    }

    // Encontrar el mínimo es ir siempre a la izquierda hasta el final
    NodoArbolBinario* encontrarMinimo(NodoArbolBinario* nodo) {
        while (nodo && nodo->izquierdo != nullptr) {
            nodo = nodo->izquierdo;
        }
        return nodo;
    }
```

```cpp
NodoArbolBinario* eliminarRecursivo(NodoArbolBinario* nodo, const char* numero_id) {
    if (nodo == nullptr) {
        return nullptr;
    }

    // Comparar numero_de_id
    int comparacion = strcmp(numero_id, nodo->piloto.numero_de_id);

    if (comparacion < 0) {
        nodo->izquierdo = eliminarRecursivo(nodo->izquierdo, numero_id);
    } else if (comparacion > 0) {
        nodo->derecho = eliminarRecursivo(nodo->derecho, numero_id);
    } else {
        // Nodo encontrado, revisar cuántos hijos tiene
        if (nodo->izquierdo == nullptr) {
            NodoArbolBinario* temp = nodo->derecho;  // Solo hijo derecho
            delete nodo;
            return temp;
        } else if (nodo->derecho == nullptr) {
            NodoArbolBinario* temp = nodo->izquierdo;  // Solo hijo izquierdo
            delete nodo;
            return temp;
        }

        // Tiene dos hijos: copiar el sucesor inorden y eliminar ese sucesor
        NodoArbolBinario* temp = encontrarMinimo(nodo->derecho);
        nodo->piloto = temp->piloto;
        nodo->derecho = eliminarRecursivo(nodo->derecho, temp->piloto.numero_de_id);
    }

    return nodo;
}

// Preorden visita raíz primero, luego izquierda, luego derecha
void preordenRecursivo(NodoArbolBinario* nodo) {
    if (nodo != nullptr) {
        std::cout << "Piloto: " << nodo->piloto.nombre
                  << " | Horas: " << nodo->piloto.horas_de_vuelo << std::endl;
        preordenRecursivo(nodo->izquierdo);
```

Es un Árbol Binario de Búsqueda (ABB) clásico que organiza pilotos según sus horas de vuelo. Cada nodo tiene máximo 2 hijos: izquierdo (menos horas) y derecho (más horas).

TablaHash.h

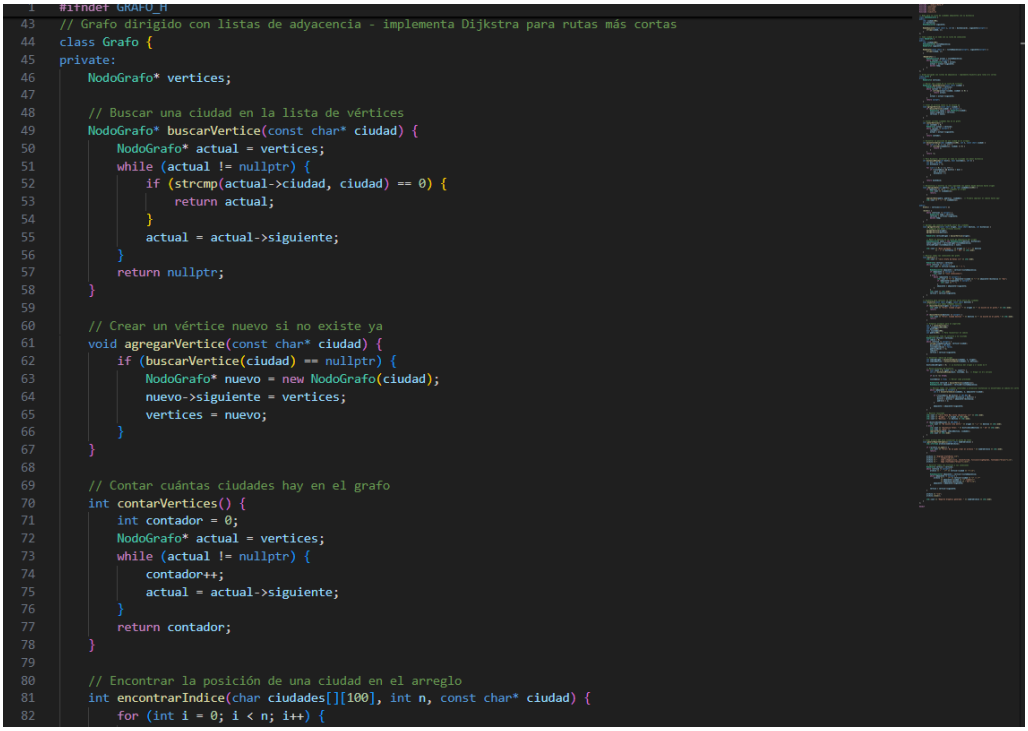
```cpp
class TablaHash {
private:
    static const int M = 19; // Tamaño fijo de la tabla (19 posiciones)
    NodoHash* tabla[M];      // Arreglo de punteros a listas encadenadas

    // Calcular posición en la tabla: ASCII de la letra + valores numéricos de los dígitos, luego módulo 19
    // Ejemplo: "X10000123" -> (88+1+0+0+0+0+1+2+3) % 19 = 95 % 19 = 0
    int funcionHash(const char* numero_id) {
        int suma = 0;
        for (int i = 0; numero_id[i] != '\0'; i++) {
            if (numero_id[i] >= '0' && numero_id[i] <= '9') {
                suma += (numero_id[i] - '0');  // Convertir char a su valor numérico
            } else {
                suma += (int)numero_id[i];  // Letras u otros caracteres: usar ASCII
            }
        }
        return suma % M;  // Aplicar módulo para obtener posición (0-18)
    }

public:
    TablaHash() {
        for (int i = 0; i < M; i++) {
            tabla[i] = nullptr;  // Todas las posiciones vacías al inicio
        }
    }

    ~TablaHash() {
        // Recorrer cada posición de la tabla y liberar sus cadenas
        for (int i = 0; i < M; i++) {
            NodoHash* actual = tabla[i];
            while (actual != nullptr) {
                NodoHash* siguiente = actual->siguiente;
                delete actual;  // Liberar nodo
                actual = siguiente;
            }
        }
    }

    // Insertar un piloto calculando su posición hash
```

```
class TablaHash {
    void insertar(const Piloto& piloto) {
        // Verificar que no exista ya (recorrer la cadena de esa posicion)
        NodoHash* actual = tabla[indice];
        while (actual != nullptr) {
            if (strcmp(actual->piloto.numero_de_id, piloto.numero_de_id) == 0) {  // Ya existe
                std::cout << "Advertencia: Piloto con ID " << piloto.numero_de_id
                          << " ya existe en la tabla." << std::endl;
                return;
            }
            actual = actual->siguiente;
        }

        // No existe, insertar al final de la cadena (respeta orden de llegada)
        NodoHash* nuevo = new NodoHash(piloto);

        if (tabla[indice] == nullptr) {
            // La cadena está vacía, el nuevo es el primero
            tabla[indice] = nuevo;
        } else {
            // Recorrer hasta el final de la cadena
            NodoHash* temp = tabla[indice];
            while (temp->siguiente != nullptr) {
                temp = temp->siguiente;
            }
            temp->siguiente = nuevo;  // Agregar al final
        }

        std::cout << "Piloto " << piloto.nombre << " insertado en posicion "
                  << indice << std::endl;
    }

    // Buscar un piloto por ID calculando su hash y recorriendo su cadena
    Piloto* buscar(const char* numero_id) {
        int indice = funcionHash(numero_id);  // Calcular en qué posición debería estar
        NodoHash* actual = tabla[indice];

        // Recorrer la cadena de esa posición buscando el ID
        while (actual != nullptr) {
            if (strcmp(actual->piloto.numero_de_id, numero_id) == 0) {  // Lo encontramos
```

Es una Tabla Hash de tamaño fijo (M=19) que usa encadenamiento para manejar colisiones. Permite búsqueda ultra rápida de pilotos por su número de ID.

Grafo.h

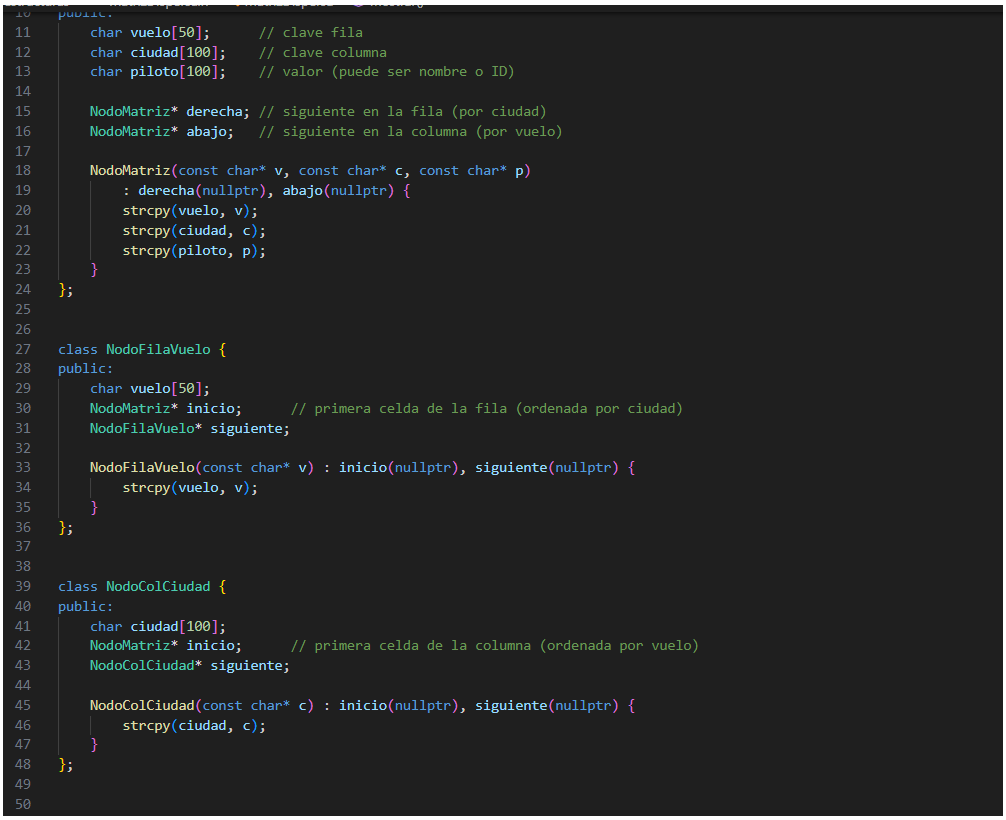
```

// Nodo para la lista de ciudades adyacentes con su distancia
class NodoAdyacente {
public:
    char ciudad[100];
    int distancia;
    NodoAdyacente* siguiente;

    NodoAdyacente(const char* c, int d) : distancia(d), siguiente(nullptr) {
        strcpy(ciudad, c);
    }
};

// Cada ciudad es un nodo con su lista de conexiones
class NodoGrafo {
public:
    char ciudad[100];
    NodoAdyacente* listaAdyacencia;
    NodoGrafo* siguiente;

    NodoGrafo(const char* c) : listaAdyacencia(nullptr), siguiente(nullptr) {
        strcpy(ciudad, c);
    }

    ~NodoGrafo() {
        NodoAdyacente* actual = listaAdyacencia;
        while (actual != nullptr) {
            NodoAdyacente* temp = actual;
            actual = actual->siguiente;
            delete temp;
        }
    }
};

// Grafo dirigido con listas de adyacencia - implementa Dijkstra para rutas más cortas
class Grafo {
private:
    NodoGrafo* vertices;
```



```
#ifndef GRAFO_H
// Grafo dirigido con listas de adyacencia - implementa Dijkstra para rutas más cortas
class Grafo {
private:
    NodoGrafo* vertices;

    // Buscar una ciudad en la lista de vértices
    NodoGrafo* buscarVertice(const char* ciudad) {
        NodoGrafo* actual = vertices;
        while (actual != nullptr) {
            if (strcmp(actual->ciudad, ciudad) == 0) {
                return actual;
            }
            actual = actual->siguiente;
        }
        return nullptr;
    }

    // Crear un vértice nuevo si no existe ya
    void agregarVertice(const char* ciudad) {
        if (buscarVertice(ciudad) == nullptr) {
            NodoGrafo* nuevo = new NodoGrafo(ciudad);
            nuevo->siguiente = vertices;
            vertices = nuevo;
        }
    }

    // Contar cuántas ciudades hay en el grafo
    int contarVertices() {
        int contador = 0;
        NodoGrafo* actual = vertices;
        while (actual != nullptr) {
            contador++;
            actual = actual->siguiente;
        }
        return contador;
    }

    // Encontrar la posición de una ciudad en el arreglo
    int encontrarIndice(char ciudades[][100], int n, const char* ciudad) {
        for (int i = 0; i < n; i++) {
```

Es un Grafo Dirigido Ponderado implementado con listas de adyacencia para representar rutas entre ciudades. Incluye el Algoritmo de Dijkstra para calcular la ruta más corta.

MatrizDispersa.h



```
public:
    char vuelo[50];      // clave fila
    char ciudad[100];    // clave columna
    char piloto[100];    // valor (puede ser nombre o ID)

    NodoMatriz* derecha; // siguiente en la fila (por ciudad)
    NodoMatriz* abajo;   // siguiente en la columna (por vuelo)

    NodoMatriz(const char* v, const char* c, const char* p)
        : derecha(nullptr), abajo(nullptr) {
        strcpy(vuelo, v);
        strcpy(ciudad, c);
        strcpy(piloto, p);
    }
};


class NodoFilaVuelo {
public:
    char vuelo[50];
    NodoMatriz* inicio;      // primera celda de la fila (ordenada por ciudad)
    NodoFilaVuelo* siguiente;

    NodoFilaVuelo(const char* v) : inicio(nullptr), siguiente(nullptr) {
        strcpy(vuelo, v);
    }
};


class NodoColCiudad {
public:
    char ciudad[100];
    NodoMatriz* inicio;      // primera celda de la columna (ordenada por vuelo)
    NodoColCiudad* siguiente;

    NodoColCiudad(const char* c) : inicio(nullptr), siguiente(nullptr) {
        strcpy(ciudad, c);
    }
};
```

```
    NodoFilaVuelo* ant = nullptr;
    NodoFilaVuelo* act = filas;

    while (act && strcmp(act->vuelo, vuelo) < 0) {
        ant = act;
        act = act->siguiente;
    }

    if (act && strcmp(act->vuelo, vuelo) == 0) return act;

    NodoFilaVuelo* nueva = new NodoFilaVuelo(vuelo);
    if (!ant) {
        nueva->siguiente = filas;
        filas = nueva;
    } else {
        nueva->siguiente = act;
        ant->siguiente = nueva;
    }
    return nueva;
}

// ---------- Crear/Buscar columna (ciudad) ordenada ----------
NodoColCiudad* buscarOCrearColumna(const char* ciudad) {
    NodoColCiudad* ant = nullptr;
    NodoColCiudad* act = columnas;

    while (act && strcmp(act->ciudad, ciudad) < 0) {
        ant = act;
        act = act->siguiente;
    }

    if (act && strcmp(act->ciudad, ciudad) == 0) return act;

    NodoColCiudad* nueva = new NodoColCiudad(ciudad);
    if (!ant) {
        nueva->siguiente = columnas;
        columnas = nueva;
    } else {
        nueva->siguiente = act;
        ant->siguiente = nueva;
```

Es una Matriz Dispersa Ortogonal que representa asignaciones de Piloto-Vuelo-Ciudad. Usa listas doblemente enlazadas (horizontal y vertical) para ahorrar memoria en matrices con muchas celdas vacías.

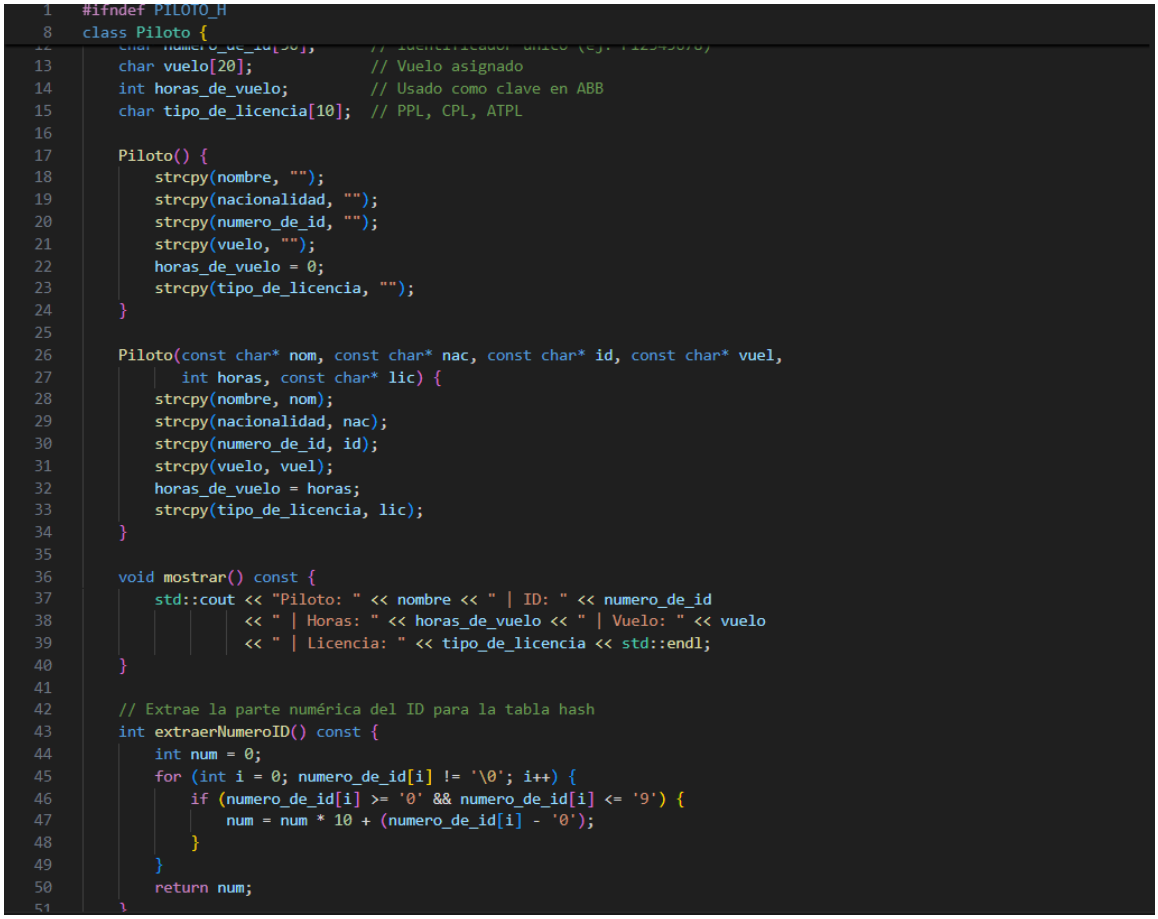
Avion.h

```
    char registro[20];
    char modelo[50];
    char fabricante[50];
    int capacidad;
    int peso_max_despegue;
    char aerolinea[50];
    char ciudad_destino[100];
    char estado[20]; // "Disponible" o "Mantenimiento"

    Avion() {
        strcpy(vuelo, "");
        strcpy(registro, "");
        strcpy(modelo, "");
        strcpy(fabricante, "");
        capacidad = 0;
        peso_max_despegue = 0;
        strcpy(aerolinea, "");
        strcpy(ciudad_destino, "");
        strcpy(estado, "Disponible");
    }

    Avion(const char* reg, const char* mod, const char* fab, int cap, int peso, const char* aero, const char* est) {
        strcpy(vuelo, "");
        strcpy(registro, reg);
        strcpy(modelo, mod);
        strcpy(fabricante, fab);
        capacidad = cap;
        peso_max_despegue = peso;
        strcpy(aerolinea, aero);
        strcpy(ciudad_destino, "");
        strcpy(estado, est);
    }

    void mostrar() const {
        std::cout << "Vuelo: " << vuelo << " | Registro: " << registro << " | Modelo: " << modelo
                  << " | Capacidad: " << capacidad << " | Aerolinea: " << aerolinea
                  << " | Ciudad Destino: " << ciudad_destino << " | Estado: " << estado << std::endl;
```

Es una clase modelo (DTO - Data Transfer Object) que representa un avión con toda su información técnica y operativa. Es la estructura de datos básica almacenada en las listas circulares y árbol B.

Piloto.h

Es una clase modelo que representa un piloto con su información profesional. A diferencia de Avion, esta clase tiene un método de utilidad (extraerNumeroID()) y se usa como clave en múltiples estructuras.



```
#ifndef PILOTO_H
class Piloto {
    char vuelo[20];             // Vuelo asignado
    int horas_de_vuelo;         // Usado como clave en ABB
    char tipo_de_licencia[10];  // PPL, CPL, ATPL

    Piloto() {
        strcpy(nombre, "");
        strcpy(nacionalidad, "");
        strcpy(numero_de_id, "");
        strcpy(vuelo, "");
        horas_de_vuelo = 0;
        strcpy(tipo_de_licencia, "");
    }

    Piloto(const char* nom, const char* nac, const char* id, const char* vuel,
           int horas, const char* lic) {
        strcpy(nombre, nom);
        strcpy(nacionalidad, nac);
        strcpy(numero_de_id, id);
        strcpy(vuelo, vuel);
        horas_de_vuelo = horas;
        strcpy(tipo_de_licencia, lic);
    }

    void mostrar() const {
        std::cout << "Piloto: " << nombre << " | ID: " << numero_de_id
                  << " | Horas: " << horas_de_vuelo << " | Vuelo: " << vuelo
                  << " | Licencia: " << tipo_de_licencia << std::endl;
    }

    // Extrae la parte numérica del ID para la tabla hash
    int extraerNumeroID() const {
        int num = 0;
        for (int i = 0; numero_de_id[i] != '\0'; i++) {
            if (numero_de_id[i] >= '0' && numero_de_id[i] <= '9') {
                num = num * 10 + (numero_de_id[i] - '0');
            }
        }
        return num;
    }
```