# MANUAL TECNICO

## INTRODUCCIÓN

Este manual técnico describe la implementación de una simulación de un aeropuerto, donde se gestiona la llegada y el estado de los aviones, Pilotos y sus rutas. El objetivo principal de esta simulación es proporcionar una herramienta que permita al usuario interactuar con diferentes aspectos operativos del aeropuerto, desde el control de los estados de los aviones hasta la administración del flujo de pilotos y rutas.

## CONTEXTO

El sistema de gestión de aeropuertos desarrollado previamente como parte de la práctica del curso de Estructuras de Datos permite gestionar vuelos, pasajeros, equipajes y otros aspectos clave del funcionamiento de un aeropuerto. Este sistema ha demostrado ser una herramienta eficaz para la administración de operaciones aeroportuarias, pero ahora se busca expandir y mejorar sus funcionalidades mediante la incorporación de nuevas estructuras de datos avanzadas.

El objetivo de esta ampliación es mejorar la eficiencia y la capacidad de manejo de datos del sistema, adaptándose a la creciente complejidad y volumen de información que debe gestionar un aeropuerto moderno. La inclusión de estructuras de datos como árboles binarios de búsqueda equilibrados, árboles B, matrices dispersas, tablas hash y grafos permitirá optimizar diversas operaciones, como la búsqueda, inserción, eliminación y organización de datos, así como la representación y análisis de redes y relaciones entre los diferentes elementos del sistema.

## OBJETIVOS

- Incorporar un árbol binario de búsqueda equilibrado (AVL o Red-Black Tree) para mejorar la eficiencia en la búsqueda y manipulación de datos relacionados con vuelos, pilotos y aviones.

- Implementar un árbol B para la gestión eficiente de grandes volúmenes de datos, como la información histórica de aviones.

- Utilizar una matriz dispersa para optimizar el almacenamiento y acceso a datos que tienen una gran cantidad de ceros, como las relaciones de conectividad entre diferentes aeropuertos.

- Implementar una tabla hash para acelerar las operaciones de búsqueda e inserción de datos, como el registro de pasajeros y equipajes.

- Emplear grafos para representar y analizar rutas de vuelo, optimizando la planificación y el análisis de conexiones entre diferentes destinos.

## ESTRUCTURAS DE DATOS

### Árbol Binario

```cpp
#ifndef BST_H
#define BST_H

#include "node.h"
#include "Pilotos.h"
#include <iostream>
#include <fstream>

class BST {
public:
    BST() : root(nullptr) {}

    void insert(const Pilotos& piloto) {
        root = insertRec(root, piloto);
    }

    void inorder() const {
        inorderRec(root);
        std::cout << std::endl;
    }

    void preorder() const {
        preorderRec(root);
        std::cout << std::endl;
    }

    void postorder() const {
        postorderRec(root);
        std::cout << std::endl;
    }

    bool search(int horas_de_vuelo) const {
        return searchRec(root, horas_de_vuelo);
    }
```

```
    void visualize() const {
        std::ofstream out("bst.dot");
        out << "digraph BST {" << std::endl;
        visualizeRec(out, root);
        out << "}";
        out.close();
        system("dot -Tpng bst.dot -o bst.png"); // Comando para generar el PNG usando Graphviz
        std::cout << "BST visualization generated as bst.png" << std::endl;
    }

    void remove(const std::string& numero_de_id) {
        root = removeRec(root, numero_de_id);
    }

private:
    Node* root;

    Node* insertRec(Node* node, const Pilotos& piloto) {
        if (node == nullptr) {
            return new Node(piloto);
        }
        if (piloto.horas_de_vuelo < node->data.horas_de_vuelo) {
            node->left = insertRec(node->left, piloto);
        } else if (piloto.horas_de_vuelo > node->data.horas_de_vuelo) {
            node->right = insertRec(node->right, piloto);
        }
        return node;
    }
```

```
    void inorderRec(Node* node) const {
        if (node != nullptr) {
            inorderRec(node->left);
            std::cout << node->data.nombre << " (" << node->data.horas_de_vuelo << " horas)" << "\n ";
            inorderRec(node->right);
        }
    }

    void preorderRec(Node* node) const {
        if (node != nullptr) {
            std::cout << node->data.nombre << " (" << node->data.horas_de_vuelo << " horas)" << "\n ";
            preorderRec(node->left);
            preorderRec(node->right);
        }
    }

    void postorderRec(Node* node) const {
        if (node != nullptr) {
            postorderRec(node->left);
            postorderRec(node->right);
            std::cout << node->data.nombre << " (" << node->data.horas_de_vuelo << " horas)" << "\n ";
        }
    }
```

```
    bool searchRec(Node* node, int horas_de_vuelo) const {
        if (node == nullptr) {
            return false;
        }
        if (node->data.horas_de_vuelo == horas_de_vuelo) {
            return true;
        }
        if (horas_de_vuelo < node->data.horas_de_vuelo) {
            return searchRec(node->left, horas_de_vuelo);
        } else {
            return searchRec(node->right, horas_de_vuelo);
        }
    }

    Node* removeRec(Node* node, const std::string& numero_de_id) {
        if (node == nullptr) {
            return node;
        }
        if (numero_de_id < node->data.numero_de_id) {
            node->left = removeRec(node->left, numero_de_id);
        } else if (numero_de_id > node->data.numero_de_id) {
            node->right = removeRec(node->right, numero_de_id);
        } else {
            if (node->left == nullptr) {
                Node* temp = node->right;
                delete node;
                return temp;
            } else if (node->right == nullptr) {
                Node* temp = node->left;
                delete node;
                return temp;
            }
            Node* temp = minValueNode(node->right);
            node->data = temp->data;
            node->right = removeRec(node->right, temp->data.numero_de_id);
        }
        return node;
    }
```

```
    Node* minValueNode(Node* node) {
        Node* current = node;
        while (current && current->left != nullptr) {
            current = current->left;
        }
        return current;
    }

    void visualizeRec(std::ofstream& out, Node* node) const {
        if (node != nullptr) {
            if (node->left != nullptr) {
                out << "\"" << node->data.nombre << " (" << node->data.horas_de_vuelo << " horas)" << "\" -> \"" << node->left->data.nombre << " (" << node->left->data.horas_de_
            }
            if (node->right != nullptr) {
                out << "\"" << node->data.nombre << " (" << node->data.horas_de_vuelo << " horas)" << "\" -> \"" << node->right->data.nombre << " (" << node->right->data.horas_d
            }
            visualizeRec([&]out, node->left);
            visualizeRec([&]out, node->right);
        }
    }
};

#endif // BST_H
```

## Árbol B de orden 5

```
#ifndef BTREE_H
#define BTREE_H

#include <iostream>
#include <fstream>
#include "BTreeNode.h"

using namespace std;

class BTree {
private:
    BTreeNode* root;

    bool setValue(Avion*, Avion*&, BTreeNode*&, BTreeNode*&);
    void insertNode(Avion*, int, BTreeNode*&, BTreeNode*);
    void splitNode(Avion*, Avion*&, int, BTreeNode*&, BTreeNode*, BTreeNode*&);
    BTreeNode* createNode(Avion*, BTreeNode*);
    void generateDotFile(BTreeNode*, ofstream&);
    void traversal(BTreeNode*);
    bool remove(const string&, BTreeNode*&, Avion*&);

public:
    BTree() : root(nullptr) {}

    void insert(Avion*);
    void traversal();
    void exportToGraphviz(const string&);
    Avion* deleteAvion(const string& numero_de_registro);
};

void BTree::insert(Avion* avion) {
    Avion* i = nullptr;
    BTreeNode* child = new BTreeNode();
    if (setValue(avion, i, root, child)) {
        root = createNode(i, child);
    }
}
```

```cpp
bool BTree::setValue(Avion* val, Avion*& pval, BTreeNode*& node, BTreeNode*& child) {
    int pos;
    bool res;
    BTreeNode* newnode = new BTreeNode();
    if (node == nullptr) {
        pval = val;
        child = nullptr;
        res = true;
        return res;
    }

    if (val->numero_de_registro < node->val[1]->numero_de_registro) {
        pos = 0;
    } else {
        pos = node->num;
        while (val->numero_de_registro < node->val[pos]->numero_de_registro && pos > 1) {
            pos--;
        }
        if (val->numero_de_registro == node->val[pos]->numero_de_registro) {
            cout << "Duplicates are not permitted" << endl;
            res = false;
            return res;
        }
    }

    if (setValue(val, [&]pval, node: [&]node->link[pos], [&]child)) {
        if (node->num < MAXI) {
            insertNode(pval, pos, [&]node, child);
        } else {
            splitNode(val: pval, [&]pval, pos, [&]node, child, [&]newnode);
            child = newnode;
            res = true;
            return res;
        }
    }

    res = false;
    return res;
}
```

```cpp
void BTree::insertNode(Avion* val, int pos, BTreeNode*& node, BTreeNode* child) {
    int j = node->num;
    while (j > pos) {
        node->val[j + 1] = node->val[j];
        node->link[j + 1] = node->link[j];
        j--;
    }
    node->val[j + 1] = val;
    node->link[j + 1] = child;
    node->num++;
}

void BTree::splitNode(Avion* val, Avion*& pval, int pos, BTreeNode*& node, BTreeNode* child, BTreeNode*& newnode) {
    int median, i, j;
    if (pos > 1) {
        median = 2;
    } else {
        median = 1;
    }
    newnode = new BTreeNode();
    j = median + 1;
    while (j <= MAXI) {
        newnode->val[j - median] = node->val[j];
        newnode->link[j - median] = node->link[j];
        j++;
    }
    node->num = median;
    newnode->num = MAXI - median;
    if (pos <= 1) {
        insertNode(val, pos, [&]node, child);
    } else {
        insertNode(val, pos - median, node: [&]newnode, child);
    }
    pval = node->val[node->num];
    newnode->link[0] = node->link[node->num];
    node->num--;
}
```

```
BTreeNode* BTree::createNode(Avion* val, BTreeNode* child) {
    BTreeNode* newNode = new BTreeNode();
    newNode->val[1] = val;
    newNode->num = 1;
    newNode->link[0] = root;
    newNode->link[1] = child;
    return newNode;
}

void BTree::traversal() {
    traversal(root);
    cout << endl;
}

void BTree::traversal(BTreeNode* myNode) {
    if (myNode != nullptr) {
        cout << " [ ";
        int i = 0;
        while (i < myNode->num) {
            cout << myNode->val[i + 1]->numero_de_registro << " ";
            i++;
        }
        for (int j = 0; j <= myNode->num; ++j) {
            traversal(myNode->link[j]);
        }
        cout << "] ";
    }
}
```

```
void BTree::generateDotFile(BTreeNode* myNode, ofstream& outFile) {
    if (myNode != nullptr) {
        outFile << "  node" << myNode << " [label=\"";
        for (int i = 1; i <= myNode->num; ++i) {
            outFile << myNode->val[i]->numero_de_registro ;
            if (i < myNode->num) {
                outFile << "|";
            }
        }
        outFile << "\"];" << endl;
        for (int i = 0; i <= myNode->num; ++i) {
            if (myNode->link[i] != nullptr) {
                outFile << "  node" << myNode << " -> node" << myNode->link[i] << ";" << endl;
                generateDotFile(myNode->link[i], [&]outFile);
            }
        }
    }
}

void BTree::exportToGraphviz(const string& filename) {
    ofstream outFile(filename);
    outFile << "digraph BTree {" << endl;
    outFile << "  node [shape=record];" << endl;
    generateDotFile(root, [&]outFile);
    outFile << "}" << endl;
    outFile.close();
    system( Command: "dot -Tpng btree.dot -o btree.png"); // Comando para generar el PNG usando Graphviz
    cout << "Graphviz file created: " << filename << endl;
}

Avion* BTree::deleteAvion(const string& numero_de_registro) {
    Avion* removedAvion = nullptr;
    if (!remove(numero_de_registro, [&]root, [&]removedAvion)) {
        cout << "Avion with numero_de_registro " << numero_de_registro << " not found." << endl;
    }
    return removedAvion;
}
```

# Tabla hash

```cpp
#ifndef HASHTABLEPILOTOS_H
#define HASHTABLEPILOTOS_H

#include "Pilotos.h"
#include <vector>
#include <list>
#include <iostream>
#include <fstream>
#include <cctype>
#include <algorithm>

// Función para comparar los pilotos ignorando la letra 'X' en el número de ID
bool comparePilotos(const Pilotos& a, const Pilotos& b) {
    return std::stoi(a.numero_de_id.substr(1)) < std::stoi(b.numero_de_id.substr(1));
}

class HashTablePilotos {
private:
    int M; // Tamaño de la tabla hash
    std::vector<std::list<Pilotos>> table; // La tabla hash es un vector de listas

    // Función para extraer y convertir los dígitos de numero_de_id a un número entero
    int extractDigits(const std::string& id) {
        std::string digits;
        for (char c : id) {
            if (isdigit(c)) {
                digits += c;
            }
        }
        return std::stoi(digits);
    }

    // Función de dispersión
    int hashFunction(const std::string& id) {
        int int_id = extractDigits(id);
        return int_id % M;
    }
```

```cpp
public:
    // Constructor
    HashTablePilotos(int size) : M(size), table(size) {}

    // Función para insertar un piloto en la tabla hash
    void insert(const Pilotos& piloto) {
        int index = hashFunction(piloto.numero_de_id);
        table[index].push_back(piloto);
    }

    // Función para buscar un piloto por su ID
    Pilotos* search(const std::string& id) {
        int index = hashFunction(id);
        for (auto& piloto Pilotos& : table[index]) {
            if (piloto.numero_de_id == id) {
                return &piloto;
            }
        }
        return nullptr; // Si no se encuentra el piloto
    }

    // Función para eliminar un piloto por su ID
    bool remove(const std::string& id) {
        int index = hashFunction(id);
        for (auto it = table[index].begin(); it != table[index].end(); ++it) {
            if (it->numero_de_id == id) {
                table[index].erase( it);
                return true; // Piloto eliminado exitosamente
            }
        }
        return false; // Si no se encuentra el piloto
    }

    // Función para mostrar la tabla hash
    void display() {
        for (int i = 0; i < M; ++i) {
            std::cout << "Index " << i << ": ";
            for (const auto& piloto const Pilotos& : table[i]) {
                std::cout << piloto.nombre << " (" << piloto.numero_de_id << ") ";
```

nTablePilotos › remove

## Grafo con lista de adyacencia

```cpp
#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <map>
#include <vector>
#include "Rutas.h"
#include <set>

// Estructura para representar una lista de adyacencia
struct AdjacencyList {
    Rutas ruta;
    AdjacencyList* next;
};

// Clase para representar un grafo
class Graph {
private:
    std::map<std::string, int> cityIndexMap; // Mapa de nombres de ciudades a índices
    std::vector<std::string> cities; // Vector de nombres de ciudades
    std::vector<AdjacencyList*> adjLists; // Vector de listas de adyacencia

    // Función para crear un nuevo nodo en la lista de adyacencia
    AdjacencyList* createNode(Rutas ruta) {
        AdjacencyList* newNode = new AdjacencyList;
        newNode->ruta = ruta;
        newNode->next = nullptr;
        return newNode;
    }

    // Función para obtener el índice de una ciudad, agregándola si es necesario
    int getCityIndex(const std::string& city) {
        if (cityIndexMap.find(city) == cityIndexMap.end()) {
            cityIndexMap[city] = cities.size();
            cities.push_back(city);
            adjLists.push_back(nullptr); // Agregar nueva lista de adyacencia
        }
```

```cpp
public:
    // Constructor
    Graph() {}

    // Función para agregar una arista al grafo
    void addEdge(const std::string& origen, const std::string& destino, int distancia) {
        int src = getCityIndex(origen);
        int dest = getCityIndex(destino);

        // Crear ruta
        Rutas ruta;
        ruta.origen = origen;
        ruta.destino = destino;
        ruta.distancia = distancia;

        // Agregar arista de src a dest
        AdjacencyList* newNode = createNode(ruta);
        newNode->next = adjLists[src];
        adjLists[src] = newNode;

        // No crear ruta inversa para mantener el grafo dirigido
    }

    // Función para imprimir el grafo
    void printGraph() {
        for (size_t i = 0; i < cities.size(); i++) {
            AdjacencyList* temp = adjLists[i];
            std::cout << "City " << cities[i] << ":";
            while (temp) {
                std::cout << " -> (" << temp->ruta.origen << ", " << temp->ruta.destino << ", " << temp->ruta.distancia << ")";
                temp = temp->next;
            }
            std::cout << std::endl;
        }
    }
```

```cpp
// Función para leer el archivo y agregar las aristas
void readFromFile(const std::string& filename) {
    std::ifstream file(filename);
    std::string line;

    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string origen, destino, distanciaStr;
        int distancia;

        std::getline(ss, origen, '/');
        std::getline(ss, destino, '/');
        std::getline(ss, distanciaStr, ';');
        distancia = std::stoi(distanciaStr);

        addEdge(origen, destino, distancia);
    }
}

// Función para exportar el grafo a un archivo DOT para Graphviz
void exportToGraphviz(const std::string& filename) {
    std::ofstream file(filename);
    if (!file) {
        std::cerr << "Error opening file " << filename << std::endl;
        return;
    }

    file << "digraph G {\n";

    std::set<std::pair<std::string, std::string>> edges;

    for (size_t i = 0; i < cities.size(); i++) {
        AdjacencyList* temp = adjLists[i];
        while (temp) {
            std::pair<std::string, std::string> edge(x: temp->ruta.origen, y: temp->ruta.destino);
            if (edges.find(edge) == edges.end()) {
                file << "    \"" << temp->ruta.origen << "\" -> \"" << temp->ruta.destino << "\" [label=\"" << temp->ruta.distancia << "\"];\n";
                edges.insert(edge);
            }
```