Proyecto EDD Vacaciones 2024

Manual Tecnico

Creado por: Juan Pablo Samayoa Ruiz

USAC TRICENTENARIA
Universidad de San Carlos de Guatemala

Proyecto Objetivos

General:
Aplicar los conocimientos de Estructuras de Datos en la creación de soluciones de software.

Específicos:

- Hacer uso correcto de memoria dinámica y apuntadores en el lenguaje de programación C++.
- Aplicar los conocimientos adquiridos sobre estructuras de datos no lineales.
- Utilizar la herramienta graphviz para generar reportes.

Descripción general:

En este proyecto se le solicita, implementar nuevas funcionalidades al sistema de gestión de aeropuerto que se desarrolló como parte de la práctica. Se mantendrán las mismas funciones, como la gestión de vuelos, pasajeros, equipajes y algunas otras características más. Este sistema utilizará diversas estructuras de datos adicionales a las que ya se tenían. Las estructuras a considerar son: árbol binario de búsqueda equilibrado, árbol B, matriz dispersa, tabla hash y grafos.

Estructuras a utilizar:

- **Lista simple:** Es una lista enlazada de nodos, donde cada nodo tiene un campo único de enlace.
- **Lista circular Doble:** Lista enlazada donde el último nodo apunta al primer, formando un bucle. Esto permite recorrer la lista indefinidamente sin llegar al final.
- **Árbol b(Orden 5):** Árbol cutos nodos pueden tener un número múltiple de hijos. En el caso de un árbol b de orden 5 el numero máximo de llaves es de:
  5-1 = 4 llaves máximas
  5 = Número de hijos máximos
- **Árbol Binario de búsqueda:** Árbol binario tal que el valor de cada nodo es mayor que los valores de su subárbol izquierdo y es menor que los valores de su subárbol derecho y, además, ambos subárboles son árboles binarios de búsqueda.
- **Tabla hash:** Estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda que permite

el acceso a los elementos almacenados a partir de una clave generada usando alguno de los datos almacenados.

- **Lista de adyacencia:** Representación de todas las aristas o arcos de un grafo mediante una lista. Si el grado es no dirigido, cada entrada es un conjunto o multiconjunto de dos vértices conteniendo los dos extremos de la arista correspondiente.
- **Grafo dirigido:** Grafo cuyas aristas tienen un sentido definido, a diferencia del grafo no dirigido, en el cual las aristas son relaciones simétricas y no apuntan en ningún sentido.
- **Matriz dispersa:** Es una matriz de gran tamaño en la que la mayor parte de sus elementos son cero.

Herramientas a utilizar:

- **Jetbrains-Clion:** IDE recomendado para el trabajo del lenguaje C++
- **Json:** Formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, debido a su amplia adopción como alternativa a XML, se considera un formato independiente del lenguaje.
- **Json nlhoman:** Librería externa de C++ para poder hacer lectura de archivos tipo Json.
- **Graphviz:** Conjunto de herramientas de software para el diseño de diagramas definido en el lenguaje descriptivo DOT.

Flujo:

Main: Menú inicial de la aplicación, la cual ayudará a moverse entre diferentes funciones.

```
int main() {
    int opcion;
    bool salir = false;

    while(!salir){

        cout << "*------------Menu principal-----------*" << endl;
        cout << "| 1. Carga de aviones                |" << endl;
        cout << "* 2. Carga de pilotos                *" << endl;
        cout << "| 3. Carga de rutas                  |" << endl;
        cout << "* 4. Carga de comandos               *" << endl;
        cout << "| 5. Consultar hora de vuelo(PILOTO) |" << endl;
        cout << "* 6. Recomendar ruta                 *" << endl;
        cout << "| 7. Visualizar reportes             |" << endl;
        cout << "*------------- 8.Salir ---------------*" << endl;
        cout << "Ingrese una opcion: ";
        cin >> opcion;
```

Utilizar funciones para poder declarar a que parte del programa se quiere dirigir el usuario.

```
void opcion1() {
    CargaAviones::OpcionesAviones();
}
```

```
switch (subOpcion){
    case 1:
        CargaAviones::ArbolAviones();
    break;
```

Los include dependerán de las funciones que realice el proyecto.

```
#include <iostream>
#include "CargaAviones.h"
#include "CargaComandos.h"
#include "CargaPilotos.h"
#include "CargaRutas.h"
#include "MatrizDisp.h"
using  namespace std;
```

Aviones:

- **Carga de aviones:** Se cargan los datos de los aviones con un archivo Json, y con la ayuda de la librería Json nlhoman se hará la lectura de dicho Json para así poder obtener los datos del constructo de aviones.

```
struct Avion {
    std::string vuelo;
    std::string numero_de_registro;
    std::string modelo;
    int capacidad;
    std::string aerolinea;
    std::string ciudad_destino;
    std::string estado;
};
```

```
void CargaAviones::cargarDesdeArchivo(std::string nombre_archivo) {
    std::ifstream archivo(nombre_archivo);
    if(!archivo.is_open()){
        std::cout << "No se pudo abrir el archivo" << std::endl;
        return;
    }

    json j;
    archivo >> j;

    for (const auto& item :const basic_json<>& : j) {
        Avion avion;
        avion.vuelo = item["vuelo"];
        avion.numero_de_registro = item["numero_de_registro"];
        avion.modelo = item["modelo"];
        avion.capacidad = item["capacidad"];
        avion.aerolinea = item["aerolinea"];
        avion.ciudad_destino = item["ciudad_destino"];
        avion.estado = item["estado"];
```

Además de esto se agregaran sentencias if para determinar en que parte se tienen que ingresar los arboles: Disponibles-Arbol B(5), No disponibles-Lista circular Doble

```
ListaCircularMan ListaAM;
BTree Arbol_Aviones(t:3);
```

```
if(avion.estado == "Disponible") {
    Arbol_Aviones.insert(avion);
    Arbol_Aviones.traverse();
    std::cout << " " << std::endl;
    std::cout << "Avion disponible insertado" << std::endl;
}else if(avion.estado == "Mantenimiento") {
    ListaAM.insertarFinal(avion);
    std::cout << " " << std::endl;
    std::cout << "Avion en mantenimiento insertado" << std::endl;
    std::cout << " " << std::endl;
}else {
    std::cout << "El estado no es valido" << std::endl;
}

addAvion(avion);
```

Pilotos: De igual forma que los aviones estos se cargaran desde un archivo tipo Json, y los datos leídos se cargaran en Un árbol de búsqueda binaria y una tabla hash, utilizando la primera letra de su numero de id en ASCII para poder crear su respectiva llave, además de tener un tamaño M = 18

```
void CargaPilotos::cargarDesdeArchivo(std::string nombre_archivo) {
    std::ifstream archivo(nombre_archivo);
    if(!archivo.is_open()){
        std::cout << "No se pudo abrir el archivo" << std::endl;
        return;
    }

    json j;
    archivo >> j;

    for (const auto& item :const basic_json<>& : j) {
        Piloto piloto;
        piloto.nombre = item["nombre"];
        piloto.nacionalidad = item["nacionalidad"];
        piloto.numero_de_id = item["numero_de_id"];
        piloto.vuelo = item["vuelo"];
        piloto.horas_de_vuelo = item["horas_de_vuelo"];
        piloto.tipo_de_licencia = item["tipo_de_licencia"];

        THash.insertar(piloto);
        arbol_bb.insertar(piloto);

        pilotos.push_back(piloto);
    }
    arbol_bb.generarReporte();
    THash.generarReporte();
```

```
TablaHash THash(tam:18);
ArbolBB arbol_bb;
```

```cpp
struct Piloto {
    std::string nombre;
    std::string nacionalidad;
    std::string numero_de_id;
    std::string vuelo;
    int horas_de_vuelo;
    std::string tipo_de_licencia;
};
```

Rutas: En este caso las rutas vendrán en un archivo .txt, por lo tanto, no se necesitará la Liberia Json, y estas rutas se cargaran a un grafo dirigido el cual utilizará una lista de adyacencia.

```cpp
struct Ruta {
    std::string origen;
    std::string destino;
    int distancia;
};
```

```cpp
GrafoDirigido grafo;

void CargaRutas::cargarDesdeArchivo(std::string nombre_archivo) {
    std::ifstream archivo(nombre_archivo);
    if(!archivo.is_open()){
        std::cout << "No se pudo abrir el archivo" << std::endl;
        return;
    }

    std::string linea;
    while (std::getline([&]archivo, [&]linea)) {
        std::istringstream iss(linea);
        std::string origen, destino, distancia_str;
        if (std::getline([&]iss, [&]origen, delim: '/') && std::getline([&]iss, [&]destino, delim: '/') && std::getline([&]i
            Ruta ruta;
            ruta.origen = origen;
            ruta.destino = destino;
            try {
                ruta.distancia = std::stoi(distancia_str);
                rutas.push_back(ruta);
            } catch (const std::invalid_argument& e) {
                std::cerr << "Error al convertir la distancia a número: " << distancia_str << std::endl;
            }
        }
    }
    archivo.close();
}
```

Comandos: Los comandos servirán para definir que elementos salen de las estructuras, de igual forma que las rutas estas vendrán en un archvio .txt

```
class CargaComandos {
public:
    CargaComandos();
    ~CargaComandos();
    void cargarComandos(const std::string& nombreArchivo);
    static void OpcionesComandos();
};
```

```
extern TablaHash THash;
extern ArbolBB arbol_bb;
```

```
void CargaComandos::cargarComandos(const std::string& nombreArchivo) {
    std::ifstream file(nombreArchivo);
    std::string linea;

    if (!file.is_open()) {
        std::cerr << "No se pudo abrir el archivo\n";
        return;
    }

    while (getline([&]file, [&]linea)) {
        std::stringstream ss(linea);
        std::string comando;
        getline([&]ss, [&]comando, delim:'(');

        // Imprimir el comando que se está procesando
        std::cout << "Procesando comando: " << linea << std::endl;

        if (comando == "DarDeBaja") {
            std::string numero_de_id;
            getline([&]ss, [&]numero_de_id, delim:')');
            THash.eliminar( numero_de_id);
            arbol_bb.eliminarPorId(numero_de_id);
        }
        // Agrega aquí otros comandos según sea necesario
    }
```

Recomendación de ruta: En base al algortimo de dijsktra se realizaran recomendaciones de rutas utilizando los datos cargados en el grafo dirigido, se utiliza vector y limits.

```
void GrafoDirigido::FindShortcut(const std::string& origen, const std::string&
destino) const {
    // Inicialización
    std::vector<std::string> vertices;
    std::vector<int> distancias;
    std::vector<std::string> anteriores;
    std::vector<bool> visitados;

    NodoGrafo* actual = nodos;
    while (actual != nullptr) {
        vertices.push_back(actual->origen);
        distancias.push_back(std::numeric_limits<int>::max());
        anteriores.push_back("");
        visitados.push_back(false);
        actual = actual->siguiente;
    }

    int idxOrigen = -1;
    for (size_t i = 0; i < vertices.size(); ++i) {
        if (vertices[i] == origen) {
            idxOrigen = i;
            break;
        }
    }

    if (idxOrigen == -1) {
        std::cerr << "La ciudad origen no se encuentra en el grafo." << std::endl;
        return;
    }

    distancias[idxOrigen] = 0;

    while (true) {
        // Encuentra el vértice no visitado con la menor distancia
        int minDistancia = std::numeric_limits<int>::max();
        int u = -1;
        for (size_t i = 0; i < vertices.size(); ++i) {
            if (!visitados[i] && distancias[i] < minDistancia) {
                minDistancia = distancias[i];
                u = i;
            }
        }

        if (u == -1 || vertices[u] == destino) break;

        visitados[u] = true;

        // Actualiza las distancias de los vértices adyacentes
        NodoGrafo* nodo = encontrarNodo(vertices[u]);
        if (nodo) {
            NodoAdyacencia* adyacente = nodo->adyacencias;
            while (adyacente != nullptr) {
                int v = -1;
                for (size_t i = 0; i < vertices.size(); ++i) {
                    if (vertices[i] == adyacente->destino) {
                        v = i;
                        break;
```

```
                    }
                }

                if (v != -1 && !visitados[v] && distancias[u] + adyacente-
>distancia < distancias[v]) {
                    distancias[v] = distancias[u] + adyacente->distancia;
                    anteriores[v] = vertices[u];
                }
                adyacente = adyacente->siguiente;
            }
        }
    }

    // Mostrar la ruta más corta
    int idxDestino = -1;
    for (size_t i = 0; i < vertices.size(); ++i) {
        if (vertices[i] == destino) {
            idxDestino = i;
            break;
        }
    }

    if (idxDestino == -1) {
        std::cerr << "La ciudad destino no se encuentra en el grafo." << std::endl;
        return;
    }

    if (distancias[idxDestino] == std::numeric_limits<int>::max()) {
        std::cout << "No hay ruta disponible desde " << origen << " hasta " <<
destino << "." << std::endl;
        return;
    }

    std::vector<std::string> ruta;
    for (std::string v = destino; !v.empty(); ) {
        ruta.push_back(v);
        int idx = -1;
        for (size_t i = 0; i < vertices.size(); ++i) {
            if (vertices[i] == v) {
                idx = i;
                break;
            }
        }
        v = anteriores[idx];
    }

    for (int i = 0; i < ruta.size() / 2; ++i) {
        std::swap(ruta[i], ruta[ruta.size() - 1 - i]);
    }

    std::cout << "La ruta más corta desde " << origen << " hasta " << destino << "
es:" << std::endl;
    for (const auto& ciudad : ruta) {
        std::cout << ciudad << " ";
    }
    std::cout << "\nDistancia total: " << distancias[idxDestino] << std::endl;
}
```

Arbol B (Orden):

```
BTree Arbol_Aviones(3);
#ifndef BTREE_H
#define BTREE_H
#include <iostream>
#include <fstream>
#include <vector>
#include <cstdlib>
#include "CargaAviones.h"

class BTreeNode {
public:
    std::vector<Avion> keys;     // Vector de claves
    std::vector<BTreeNode*> children; // Vector de punteros a hijos
    bool isLeaf;                 // Indicador de si es una hoja

    BTreeNode(bool leaf);

    void insertNonFull(const Avion& avion, int t);
    void splitChild(int i, BTreeNode* y, int t);
    void traverse();
    void generateGraphviz(std::ostream& out, int& index) const;

    BTreeNode* search(const std::string& NV);

    friend class BTree;
};

class BTree {
public:
    BTreeNode* root; // Puntero a la raíz
    int t;           // Grado mínimo

    BTree(int t);

    void traverse() {
        if (root != nullptr) root->traverse();
    }

    BTreeNode* search(const std::string& NV) {
        return (root == nullptr) ? nullptr : root->search(NV);
    }

    void insert(const Avion& avion);

    void generateGraphviz(std::ostream& out) const {
        out << "digraph G {\n";
        out << "node [shape=record];\n";
        out << "rank=same;\n";
        int index = 0;
        if (root != nullptr) root->generateGraphviz(out, index);
        out << "}\n";
    }

    void visualizar() const {
        std::ofstream outFile("btree.dot");
        if (outFile.is_open()) {
            generateGraphviz(outFile);
            outFile.close();
            std::system("dot -Tpng btree.dot -o btree.png");
            std::system("btree.png");
        } else {
```

```
            std::cerr << "No se pudo abrir el archivo para escribir\n";
        }
    }
};

BTreeNode::BTreeNode(bool leaf) {
    isLeaf = leaf;
}

void BTreeNode::traverse() {
    int i;
    for (i = 0; i < keys.size(); i++) {
        if (!isLeaf) {
            children[i]->traverse();
        }
        std::cout << " " << keys[i].numero_de_registro;
    }

    if (!isLeaf) {
        children[i]->traverse();
    }
}

BTreeNode* BTreeNode::search(const std::string& NV) {
    int i = 0;
    while (i < keys.size() && NV > keys[i].numero_de_registro) {
        i++;
    }

    if (i < keys.size() && keys[i].numero_de_registro == NV) {
        return this;
    }

    if (isLeaf) {
        return nullptr;
    }

    return children[i]->search(NV);
}

BTree::BTree(int t) {
    this->t = t;
    root = nullptr;
}

void BTree::insert(const Avion& avion) {
    if (root == nullptr) {
        root = new BTreeNode(true);
        root->keys.push_back(avion);
    } else {
        if (root->keys.size() == 2 * t - 1) {
            BTreeNode* s = new BTreeNode(false);
            s->children.push_back(root);
            s->splitChild(0, root, t);
            int i = 0;
            if (s->keys[0].numero_de_registro < avion.numero_de_registro) {
                i++;
            }
            s->children[i]->insertNonFull(avion, t);
            root = s;
        } else {
            root->insertNonFull(avion, t);
        }
```

```
    }
}

void BTreeNode::insertNonFull(const Avion& avion, int t) {
    int i = keys.size() - 1;
    if (isLeaf) {
        keys.resize(keys.size() + 1);
        while (i >= 0 && keys[i].numero_de_registro > avion.numero_de_registro) {
            keys[i + 1] = keys[i];
            i--;
        }
        keys[i + 1] = avion;
    } else {
        while (i >= 0 && keys[i].numero_de_registro > avion.numero_de_registro) {
            i--;
        }
        i++;
        if (children[i]->keys.size() == 2 * t - 1) {
            splitChild(i, children[i], t);
            if (keys[i].numero_de_registro < avion.numero_de_registro) {
                i++;
            }
        }
        children[i]->insertNonFull(avion, t);
    }
}

void BTreeNode::splitChild(int i, BTreeNode* y, int t) {
    BTreeNode* z = new BTreeNode(y->isLeaf);
    z->keys.resize(t - 1);

    for (int j = 0; j < t - 1; j++) {
        z->keys[j] = y->keys[j + t];
    }

    if (!y->isLeaf) {
        z->children.resize(t);
        for (int j = 0; j < t; j++) {
            z->children[j] = y->children[j + t];
        }
    }

    y->keys.resize(t - 1);

    children.insert(children.begin() + i + 1, z);
    keys.insert(keys.begin() + i, y->keys[t - 1]);

    //y->keys.resize(t - 1);
}

void BTreeNode::generateGraphviz(std::ostream& out, int& index) const {
    int current_index = index++;
    out << "node" << current_index << " [label=\"";
    for (int i = 0; i < keys.size(); ++i) {
        if (i > 0) out << " | ";
        out << "<f" << i << "> " << keys[i].numero_de_registro;
    }
    out << "\"];\n";

    for (int i = 0; i < children.size(); ++i) {
        int child_index = index;
        if (!isLeaf) {
            children[i]->generateGraphviz(out, index);  // Mover la generación del
```

```
hijo antes de la conexión
            //int key_index = i;
            //if( key_index < keys.size() ) {
               // out << "node" << current_index << ":f" << key_index << " -> node"
<< child_index << ";\n";
            //}else {
                //out << "node" << current_index << ":f" << (keys.size() - 1) << "
-> node" << child_index << ";\n";
            //}
            out << "node" << current_index <<" -> node" << child_index << ";\n";
        }
    }
}

#endif // BTREE_H
```

Lista circular Doble:

```
//
// Created by jpsam on 13/06/2024.
//

#ifndef CIRCULARDOBLEMAN_H
#define CIRCULARDOBLEMAN_H
#include "nodoAv.h"

class ListaCircularMan
{
private:
    Nodo* primero;
    Nodo* ultimo;
public:
    ListaCircularMan();
    bool estaVacia();
    void insertarInicio(Avion dato);
    void insertarFinal(Avion dato);
    void eliminarInicio();
    void eliminarFinal();
    void visualizarLista();
    void generarReporte();
    ~ListaCircularMan();
};

ListaCircularMan::ListaCircularMan()
{
    primero = nullptr;
    ultimo = nullptr;
}

bool ListaCircularMan::estaVacia()
{
    return (primero == nullptr) && (ultimo == nullptr);
}

void ListaCircularMan::insertarInicio(Avion dato)
{
    Nodo *nuevo = new Nodo(dato);
    if (estaVacia())
    {
        nuevo->setSiguiente(nuevo);
        nuevo->setAnterior(nuevo);
        primero = ultimo = nuevo;
```

```
    }
    else
    {
        nuevo->setSiguiente(primero);
        nuevo->setAnterior(ultimo);
        primero->setAnterior(nuevo);
        ultimo->setSiguiente(nuevo);
        primero = nuevo;
    }
}

void ListaCircularMan::insertarFinal(Avion dato)
{
    Nodo *nuevo = new Nodo(dato);
    if (estaVacia())
    {
        nuevo->setSiguiente(nuevo);
        nuevo->setAnterior(nuevo);
        primero = ultimo = nuevo;
    }
    else
    {
        nuevo->setSiguiente(primero);
        nuevo->setAnterior(ultimo);
        primero->setAnterior(nuevo);
        ultimo->setSiguiente(nuevo);
        ultimo = nuevo;
    }
}

void ListaCircularMan::eliminarInicio()
{
    if (estaVacia())
    {
        std::cout << "La lista está vacía" << std::endl;
    }
    else
    {
        if (primero == ultimo)
        {
            delete primero;
            primero = ultimo = nullptr;
        }
        else
        {
            Nodo *segundo = primero->getSiguiente();
            segundo->setAnterior(ultimo);
            ultimo->setSiguiente(segundo);
            delete primero;
            primero = segundo;
        }
    }
}

void ListaCircularMan::eliminarFinal()
{
    if (estaVacia())
    {
        std::cout << "La lista está vacía" << std::endl;
    }
    else
    {
        if (primero == ultimo)
```

```
        {
            delete primero;
            primero = ultimo = nullptr;
        }
        else
        {
            Nodo *temporal = primero;
            while (temporal->getSiguiente() != ultimo)
            {
                temporal = temporal->getSiguiente();
            }
            temporal->setSiguiente(primero);
            primero->setAnterior(temporal);
            delete ultimo;
            ultimo = temporal;
        }
    }
}

void ListaCircularMan::visualizarLista()
{
    if (estaVacia())
    {
        std::cout << "La lista está vacía" << std::endl;
    }
    else
    {
        Avion nodoDato;
        Nodo *actual = primero;
        do
        {
            nodoDato = actual->getDato();
            std::cout << nodoDato.numero_de_registro << std::endl; // Imprime el
número de vuelo
            actual = actual->getSiguiente();
        } while (actual != primero);
    }
}

void ListaCircularMan::generarReporte(){
    if (estaVacia()){
        std::cout << "La lista está vacía\n" << std::endl;
    }
    else
    {
        std::ofstream archivo;
        archivo.open("grafica_LC_Mantenimiento.dot", std::ios::out);
        archivo << "digraph G { rankdir=LR; node [shape=oval];" << std::endl;

        Avion nodoDato;
        Nodo *actual = primero;
        int nodeCount = 0;
        do
        {
            nodoDato = actual->getDato();
            archivo << "node" << nodeCount << " [label=\"" <<
nodoDato.numero_de_registro << "\"];" << std::endl; // Escribe el número de vuelo
            actual = actual->getSiguiente();
            nodeCount++;
        } while (actual != primero);

        actual = primero;
        int currentNode = 0;
```

```
        do
        {
            int nextNode = (currentNode + 1) % nodeCount;
            archivo << "node" << currentNode << " -> node" << nextNode << ";" <<
std::endl;
            archivo << "node" << nextNode << " -> node" << currentNode << ";" <<
std::endl;
            actual = actual->getSiguiente();
            currentNode++;
        } while (actual != primero);

        archivo << " }";
        archivo.close();
        system("dot -Tpng grafica_LC_Mantenimiento.dot -o
grafica_LC_Mantenimiento.png");
        system("start grafica_LC_Mantenimiento.png");
    }
}

ListaCircularMan::~ListaCircularMan()
{
    while (!estaVacia()) {
        eliminarInicio();
    }
}

#endif //CIRCULARDOBLEMAN_H
```

Arbol BB:

```
#include "ArbolBB.h"

ArbolBB::ArbolBB() {
    raiz = nullptr;
}

bool ArbolBB::estaVacio() {
    return (raiz == nullptr);
}

void ArbolBB::insertar(Piloto dato) {
    raiz = insertarNodo(dato, raiz);
}

NodoPilotos* ArbolBB::insertarNodo(Piloto dato, NodoPilotos* nodoPtr) {
    if (nodoPtr == nullptr) {
        NodoPilotos* nuevo = new NodoPilotos(dato);
        nodoPtr = nuevo;
    } else if (dato.horas_de_vuelo < nodoPtr->getPiloto().horas_de_vuelo) {
        nodoPtr->setSiguiente(insertarNodo(dato, nodoPtr->getSiguiente()));
    } else if (dato.horas_de_vuelo > nodoPtr->getPiloto().horas_de_vuelo) {
        nodoPtr->setAnterior(insertarNodo(dato, nodoPtr->getAnterior()));
    } else {
        std::cout << "Nodo duplicado\n";
    }
    return nodoPtr;
}

void ArbolBB::buscar(Piloto dato) {
    std::cout << "Recorrido del nodo encontrado: " << buscarNodo(dato,
```

```
raiz).horas_de_vuelo << ", " << recorrido;
}

Piloto ArbolBB::buscarNodo(Piloto dato, NodoPilotos* nodoPtr) {
    if (nodoPtr == nullptr) {
        std::cout << "Nodo no encontrado\n";
        return Piloto(); // Devuelve un piloto vacío si el nodo no se encuentra
    } else if (dato.horas_de_vuelo == nodoPtr->getPiloto().horas_de_vuelo) {
        recorrido++;
        return nodoPtr->getPiloto();
    } else if (dato.horas_de_vuelo < nodoPtr->getPiloto().horas_de_vuelo) {
        recorrido++;
        return buscarNodo(dato, nodoPtr->getSiguiente());
    } else if (dato.horas_de_vuelo > nodoPtr->getPiloto().horas_de_vuelo) {
        recorrido++;
        return buscarNodo(dato, nodoPtr->getAnterior());
    }
}

void ArbolBB::eliminarPorId(const std::string& numero_de_id) {
    NodoPilotos* nodo = buscarNodoPorId(raiz, numero_de_id);
    if (nodo != nullptr) {
        int horas_de_vuelo = nodo->getPiloto().horas_de_vuelo;
        raiz = eliminarNodo(raiz, horas_de_vuelo);
    }
}

NodoPilotos* ArbolBB::eliminarNodo(NodoPilotos* nodoPtr, int horas_de_vuelo) {
    if (nodoPtr == nullptr) {
        return nodoPtr;
    }

    if (horas_de_vuelo < nodoPtr->getPiloto().horas_de_vuelo) {
        nodoPtr->setSiguiente(eliminarNodo(nodoPtr->getSiguiente(),
horas_de_vuelo));
    } else if (horas_de_vuelo > nodoPtr->getPiloto().horas_de_vuelo) {
        nodoPtr->setAnterior(eliminarNodo(nodoPtr->getAnterior(), horas_de_vuelo));
    } else {
        // Nodo con solo un hijo o sin hijos
        if (nodoPtr->getSiguiente() == nullptr) {
            NodoPilotos* temp = nodoPtr->getAnterior();
            delete nodoPtr;
            return temp;
        } else if (nodoPtr->getAnterior() == nullptr) {
            NodoPilotos* temp = nodoPtr->getSiguiente();
            delete nodoPtr;
            return temp;
        }

        // Nodo con dos hijos: Obtener el sucesor inorden (el más pequeño en el
subárbol derecho)
        NodoPilotos* temp = encontrarMinimo(nodoPtr->getSiguiente());

        // Copiar el contenido del sucesor inorden a este nodo
        nodoPtr->setPiloto(temp->getPiloto());

        // Eliminar el sucesor inorden
        nodoPtr->setSiguiente(eliminarNodo(nodoPtr->getSiguiente(), temp-
>getPiloto().horas_de_vuelo));
    }
    return nodoPtr;
}
```

```
NodoPilotos* ArbolBB::encontrarMinimo(NodoPilotos* nodoPtr) {
    NodoPilotos* actual = nodoPtr;
    while (actual && actual->getAnterior() != nullptr) {
        actual = actual->getAnterior();
    }
    return actual;
}

NodoPilotos* ArbolBB::encontrarMaximo(NodoPilotos* nodoPtr) {
    NodoPilotos* actual = nodoPtr;
    while (actual && actual->getSiguiente() != nullptr) {
        actual = actual->getSiguiente();
    }
    return actual;
}

NodoPilotos* ArbolBB::buscarNodoPorId(NodoPilotos* nodoPtr, const std::string&
numero_de_id) {
    if (nodoPtr == nullptr) {
        return nullptr;
    }

    if (numero_de_id == nodoPtr->getPiloto().numero_de_id) {
        return nodoPtr;
    }

    NodoPilotos* encontrado = buscarNodoPorId(nodoPtr->getSiguiente(),
numero_de_id);
    if (encontrado != nullptr) {
        return encontrado;
    }
    return buscarNodoPorId(nodoPtr->getAnterior(), numero_de_id);
}

void ArbolBB::imprimirPreorden() {
    std::cout << "Recorrido Preorden: " << std::endl;
    RecorridoPreorden(raiz);
    std::cout << std::endl;
}

void ArbolBB::imprimirInorden() {
    std::cout << "Recorrido Inorden: " << std::endl;
    RecorridoInorden(raiz);
    std::cout << std::endl;
}

void ArbolBB::imprimirPostorden() {
    std::cout << "Recorrido Postorden: " << std::endl;
    RecorridoPostorden(raiz);
    std::cout << std::endl;
}

void ArbolBB::RecorridoPreorden(NodoPilotos* nodoPtr) {
    if (nodoPtr != nullptr) {
        std::cout << nodoPtr->getPiloto().horas_de_vuelo << std::endl;
        RecorridoPreorden(nodoPtr->getSiguiente());
        RecorridoPreorden(nodoPtr->getAnterior());
    }
}

void ArbolBB::RecorridoInorden(NodoPilotos* nodoPtr) {
    if (nodoPtr != nullptr) {
        RecorridoInorden(nodoPtr->getSiguiente());
```

```
        std::cout << nodoPtr->getPiloto().horas_de_vuelo << std::endl;
        RecorridoInorden(nodoPtr->getAnterior());
    }
}

void ArbolBB::RecorridoPostorden(NodoPilotos* nodoPtr) {
    if (nodoPtr != nullptr) {
        RecorridoPostorden(nodoPtr->getSiguiente());
        RecorridoPostorden(nodoPtr->getAnterior());
        std::cout << nodoPtr->getPiloto().horas_de_vuelo << std::endl;
    }
}

void ArbolBB::generarReporte() {
    if (ArbolBB::estaVacio()) {} else {
        archivo.open("grafica_arbol.dot", std::ios::out);
        archivo << "digraph G { " << std::endl;

        imprimirNodo(raiz);

        archivo << " }";
        archivo.close();
        system("dot -Tpng grafica_arbol.dot -o grafica_arbol.png");
        system("start grafica_arbol.png");
    }
}

void ArbolBB::imprimirNodo(NodoPilotos* nodoPtr) {
    if (nodoPtr == nullptr) {
        return;
    }
    if (nodoPtr->getSiguiente() != nullptr) {
        nodoDato = nodoPtr->getPiloto();
        archivo << nodoDato.horas_de_vuelo;
        archivo << "->";
        nodoDato = nodoPtr->getSiguiente()->getPiloto();
        archivo << nodoDato.horas_de_vuelo;
        archivo << ";";
    }
    imprimirNodo(nodoPtr->getSiguiente());

    if (nodoPtr->getAnterior() != nullptr) {
        nodoDato = nodoPtr->getPiloto();
        archivo << nodoDato.horas_de_vuelo;
        archivo << "->";
        nodoDato = nodoPtr->getAnterior()->getPiloto();
        archivo << nodoDato.horas_de_vuelo;
        archivo << ";";
    }
    imprimirNodo(nodoPtr->getAnterior());
}

ArbolBB::~ArbolBB() {}
```

Tabla Hash:

```
TablaHash THash(18);
```

```
//
// Created by jpsam on 29/06/2024.
//

#ifndef THASH_H
#define THASH_H
#include "ListaSimple.h"

class TablaHash {
private:
    int M;  // tamaño de la tabla
    ListaSimple* tabla;  // array de listas simples

public:
    TablaHash(int tam) : M(tam) {
        tabla = new ListaSimple[M];
    }

    ~TablaHash() {
        delete[] tabla;
    }

    int funcionHash(std::string llave) {
        int suma = static_cast<int>(llave[0]);
        for (size_t i = 1; i < llave.length(); ++i) {
            suma += (llave[i] - '0');
        }
        return suma % M;
    }

    void insertar(Piloto p) {
        int indice = funcionHash(p.numero_de_id);
        tabla[indice].insertar(p);
    }

    Piloto buscar(std::string llave) {
        int indice = funcionHash(llave);
        return tabla[indice].buscar(llave);
    }

    void eliminar(std::string llave) {
        int indice = funcionHash(llave);
        tabla[indice].eliminar(llave);
    }

    void imprimir() {
        for (int i = 0; i < M; i++) {
            tabla[i].imprimir(i);
        }
    }

    void generarReporte() {
        std::ofstream archivo("hash_reporte.dot");
        if (!archivo.is_open()) {
            throw std::runtime_error("No se pudo abrir el archivo");
        }

        archivo << "digraph G {\n";
        archivo << "rankdir=LR;\n";  // Cambia la dirección del grafo de izquierda
a derecha
        archivo << "node [shape=record];\n";

        // Crear los nodos para los índices de la tabla
```

```
        for (int i = 0; i < M; i++) {
            archivo << "index" << i << " [label=\"" << i << "\", shape=box,
style=filled, fillcolor=orange];\n";
        }

        // Crear los nodos para los pilotos y las conexiones
        for (int i = 0; i < M; i++) {
            tabla[i].generarReporte(archivo, i);
        }

        archivo << "}\n";
        archivo.close();
        system("dot -Tpng hash_reporte.dot -o hash_reporte.png");
        system("start hash_reporte.png");
    }
};

#endif //THASH_H
```

Lista simple:

```
//
// Created by jpsam on 30/06/2024.
//

#ifndef LISTASIMPLE_H
#define LISTASIMPLE_H

#include <iostream>
#include <fstream>

#include "NodoListaSimple.h"

class ListaSimple {
private:
    NodoLista* cabeza;

public:
    ListaSimple() : cabeza(nullptr) {}

    ~ListaSimple() {
        NodoLista* actual = cabeza;
        while (actual) {
            NodoLista* temp = actual;
            actual = actual->siguiente;
            delete temp;
        }
    }

    void insertar(Piloto p) {
        NodoLista* nuevo = new NodoLista(p);
        nuevo->siguiente = cabeza;
        cabeza = nuevo;
    }

    Piloto buscar(std::string llave) {
        NodoLista* actual = cabeza;
        while (actual) {
            if (actual->piloto.numero_de_id == llave) {
                return actual->piloto;
            }
            actual = actual->siguiente;
```

```
        }
        throw std::runtime_error("Piloto no encontrado");
    }

    void eliminar(std::string numero_de_id) {
        NodoLista* actual = cabeza;
        NodoLista* anterior = nullptr;

        while (actual != nullptr && actual->piloto.numero_de_id != numero_de_id) {
            anterior = actual;
            actual = actual->siguiente;
        }

        if (actual == nullptr) return;

        if (anterior == nullptr) {
            cabeza = actual->siguiente;
        } else {
            anterior->siguiente = actual->siguiente;
        }

        delete actual;
    }

    void imprimir(int indice) {
        NodoLista* actual = cabeza;
        while (actual) {
            std::cout << "Indice: " << indice << std::endl;
            std::cout << "Numero de ID: " << actual->piloto.numero_de_id <<
std::endl;
            std::cout << "-------------------------" << std::endl;
            actual = actual->siguiente;
        }
    }

    void generarReporte(std::ofstream& archivo, int indice) {
        NodoLista* actual = cabeza;
        int nodo_id = 0;
        std::string prev_node = "index" + std::to_string(indice);
        while (actual) {
            std::string current_node = "nodo" + std::to_string(indice) + "_" +
std::to_string(nodo_id);
            archivo << current_node << " [label=\"{" << actual->piloto.numero_de_id
<< "}\", shape=record, style=filled, fillcolor=lightblue];\n";
            archivo << prev_node << " -> " << current_node << ";\n";
            prev_node = current_node;
            actual = actual->siguiente;
            nodo_id++;
        }
    }
};
#endif //LISTASIMPLE_H
```

Lista de adyacencia:

```
#ifndef LISTAADYACENCIA_H
#define LISTAADYACENCIA_H
#include <string>

struct NodoAdyacencia {
    std::string destino;
    int distancia;
```

```
    NodoAdyacencia* siguiente;

    NodoAdyacencia(const std::string& dest, int dist)
        : destino(dest), distancia(dist), siguiente(nullptr) {}
};

struct NodoGrafo {
    std::string origen;
    NodoAdyacencia* adyacencias;
    NodoGrafo* siguiente;

    NodoGrafo(const std::string& orig)
        : origen(orig), adyacencias(nullptr), siguiente(nullptr) {}
};
#endif //LISTAADYACENCIA_H
```

Grafo dirigido

```
#include "GrafoDirigido.h"
#include <limits>

GrafoDirigido::GrafoDirigido() : nodos(nullptr) {}

GrafoDirigido::~GrafoDirigido() {
    NodoGrafo* actual = nodos;
    while (actual != nullptr) {
        NodoGrafo* siguienteNodo = actual->siguiente;
        eliminarNodosYAdyacencias(actual);
        actual = siguienteNodo;
    }
}

void GrafoDirigido::eliminarNodosYAdyacencias(NodoGrafo* nodo) {
    NodoAdyacencia* actualAdyacencia = nodo->adyacencias;
    while (actualAdyacencia != nullptr) {
        NodoAdyacencia* siguienteAdyacencia = actualAdyacencia->siguiente;
        delete actualAdyacencia;
        actualAdyacencia = siguienteAdyacencia;
    }
    delete nodo;
}

NodoGrafo* GrafoDirigido::encontrarNodo(const std::string& origen) const {
    NodoGrafo* actual = nodos;
    while (actual != nullptr) {
        if (actual->origen == origen) {
            return actual;
        }
        actual = actual->siguiente;
    }
    return nullptr;
}

void GrafoDirigido::agregarArista(const std::string& origen, const std::string&
destino, int distancia) {
    NodoGrafo* nodoOrigen = encontrarNodo(origen);
    if (nodoOrigen == nullptr) {
        nodoOrigen = new NodoGrafo(origen);
        nodoOrigen->siguiente = nodos;
        nodos = nodoOrigen;
    }
```

```
    NodoAdyacencia* nuevaAdyacencia = new NodoAdyacencia(destino, distancia);
    nuevaAdyacencia->siguiente = nodoOrigen->adyacencias;
    nodoOrigen->adyacencias = nuevaAdyacencia;
}

void GrafoDirigido::mostrarGrafo() const {
    NodoGrafo* actual = nodos;
    while (actual != nullptr) {
        std::cout << "Origen: " << actual->origen << std::endl;
        NodoAdyacencia* actualAdyacencia = actual->adyacencias;
        while (actualAdyacencia != nullptr) {
            std::cout << "  -> " << actualAdyacencia->destino << " (Distancia: " <<
actualAdyacencia->distancia << ")" << std::endl;
            actualAdyacencia = actualAdyacencia->siguiente;
        }
        actual = actual->siguiente;
    }
}

void GrafoDirigido::generarArchivoDOT(const std::string& nombreDOT) const {
    std::ofstream archivo(nombreDOT);
    if (!archivo.is_open()) {
        std::cerr << "No se pudo abrir el archivo " << nombreDOT << std::endl;
        return;
    }

    archivo << "digraph G {\n";
    NodoGrafo* actual = nodos;
    while (actual != nullptr) {
        NodoAdyacencia* actualAdyacencia = actual->adyacencias;
        while (actualAdyacencia != nullptr) {
            archivo << "    \"" << actual->origen << "\" -> \"" <<
actualAdyacencia->destino << "\" [label=\"" << actualAdyacencia->distancia <<
"\"];\n";
            actualAdyacencia = actualAdyacencia->siguiente;
        }
        actual = actual->siguiente;
    }
    archivo << "}\n";
    archivo.close();

    // Llamar a Graphviz para generar la imagen
    std::string comando = "dot -Tpng " + nombreDOT + " -o grafo.png";
    system(comando.c_str());
    system("start grafo.png");
}

void GrafoDirigido::FindShortcut(const std::string& origen, const std::string&
destino) const {
    // Inicialización
    std::vector<std::string> vertices;
    std::vector<int> distancias;
    std::vector<std::string> anteriores;
    std::vector<bool> visitados;

    NodoGrafo* actual = nodos;
    while (actual != nullptr) {
        vertices.push_back(actual->origen);
        distancias.push_back(std::numeric_limits<int>::max());
        anteriores.push_back("");
        visitados.push_back(false);
        actual = actual->siguiente;
    }
```

```
    int idxOrigen = -1;
    for (size_t i = 0; i < vertices.size(); ++i) {
        if (vertices[i] == origen) {
            idxOrigen = i;
            break;
        }
    }

    if (idxOrigen == -1) {
        std::cerr << "La ciudad origen no se encuentra en el grafo." << std::endl;
        return;
    }

    distancias[idxOrigen] = 0;

    while (true) {
        // Encuentra el vértice no visitado con la menor distancia
        int minDistancia = std::numeric_limits<int>::max();
        int u = -1;
        for (size_t i = 0; i < vertices.size(); ++i) {
            if (!visitados[i] && distancias[i] < minDistancia) {
                minDistancia = distancias[i];
                u = i;
            }
        }

        if (u == -1 || vertices[u] == destino) break;

        visitados[u] = true;

        // Actualiza las distancias de los vértices adyacentes
        NodoGrafo* nodo = encontrarNodo(vertices[u]);
        if (nodo) {
            NodoAdyacencia* adyacente = nodo->adyacencias;
            while (adyacente != nullptr) {
                int v = -1;
                for (size_t i = 0; i < vertices.size(); ++i) {
                    if (vertices[i] == adyacente->destino) {
                        v = i;
                        break;
                    }
                }

                if (v != -1 && !visitados[v] && distancias[u] + adyacente-
>distancia < distancias[v]) {
                    distancias[v] = distancias[u] + adyacente->distancia;
                    anteriores[v] = vertices[u];
                }
                adyacente = adyacente->siguiente;
            }
        }
    }

    // Mostrar la ruta más corta
    int idxDestino = -1;
    for (size_t i = 0; i < vertices.size(); ++i) {
        if (vertices[i] == destino) {
            idxDestino = i;
            break;
        }
    }
```

```
    if (idxDestino == -1) {
        std::cerr << "La ciudad destino no se encuentra en el grafo." << std::endl;
        return;
    }

    if (distancias[idxDestino] == std::numeric_limits<int>::max()) {
        std::cout << "No hay ruta disponible desde " << origen << " hasta " <<
destino << "." << std::endl;
        return;
    }

    std::vector<std::string> ruta;
    for (std::string v = destino; !v.empty(); ) {
        ruta.push_back(v);
        int idx = -1;
        for (size_t i = 0; i < vertices.size(); ++i) {
            if (vertices[i] == v) {
                idx = i;
                break;
            }
        }
        v = anteriores[idx];
    }

    for (int i = 0; i < ruta.size() / 2; ++i) {
        std::swap(ruta[i], ruta[ruta.size() - 1 - i]);
    }

    std::cout << "La ruta más corta desde " << origen << " hasta " << destino << "
es:" << std::endl;
    for (const auto& ciudad : ruta) {
        std::cout << ciudad << " ";
    }
    std::cout << "\nDistancia total: " << distancias[idxDestino] << std::endl;
}
```