Manual Técnico

Maria Paola Guadalupe Dávila Valenzuela
Estructura de Datos
Sección A
02/07/2023

La totalidad del código de este programa ha sido cuidadosamente elaborado en el lenguaje de programación C++. Se ha optado por hacer un extenso uso de Visual Studio Code, destacando la programación orientada a objetos y la implementación de un estructuras de datos. Estas elecciones han sido importantes para alcanzar exitosamente los objetivos trazados en este proyecto.

Estructuras de Datos Implementadas

1. Estructura de Datos Arbol B (BTree)

- **Descripción:** Implementación de un árbol B para almacenar y gestionar aviones.
- **Clase:** BTree
- **Métodos:**
  - traverse(): Recorre el árbol en orden.
  - search(const string& k): Busca un avión por número de vuelo.
  - insert(const Avion& k): Inserta un avión en el árbol.
  - remove(const string& k): Elimina un avión del árbol.
  - exportToDOT(const string& archivo): Exporta la estructura del árbol a un archivo DOT para visualización.

```
class BTreeNode {
public:
    bool isLeaf;
    vector<Avion> keys;
    vector<BTreeNode*> children;
    int t; // Grado mínimo
    int m; // Grado máximo
    BTreeNode(int _t, int _m, bool _isLeaf);
    void traverse();
    BTreeNode* search(const string& k);
    Avion* getAvionVuelo(const string& k);
    Avion* getAvionDestino(const string& k);
    Avion* getAvionRegistro(const string& k);
    void insertNonFull(const Avion& k);
    void splitChild(int i, BTreeNode* y);
    void remove(const string& k);
    void removeFromLeaf(int idx);
    void removeFromNonLeaf(int idx);
    Avion getPredecessor(int idx);
    Avion getSuccessor(int idx);
    void fill(int idx);
    void borrowFromPrev(int idx);
    void borrowFromNext(int idx);
    void merge(int idx);

    friend class BTree;
};
```

```
class BTree {
public:
    BTreeNode* root;
    int t; // Grado mínimo
    int m; //grado maximo
    BTree(int _m) {
        root = nullptr;
        t = _m/2;
        m = _m - 1;
    }
    void traverse() {
        if (root != nullptr) root->traverse();
    }
    BTreeNode* search(const string& k) {
        return (root == nullptr) ? nullptr : root->search(k);
    }
    void insert(const Avion& k);
    void remove(const string& k);
    void exportToDOT(const string& archivo);
    void exportToDOTHelper(ofstream& file, BTreeNode* node, int& nodeId);
};
```

2. Lista Circular Doble (ListaCircularDoble)

- **Descripción:** Implementación de una lista circular doble para almacenar aviones.
- **Clase:** ListaCircularDoble
- **Métodos:**
    - insertar(Avion avion): Inserta un avión en la lista.
    - exportToDOT(const string &ruta): Exporta la lista a un archivo DOT para visualización.
    - buscarAvion(string numero_de_registro): Busca un avión por número de registro.
    - eliminar(string numero_de_registro): Elimina un avión de la lista.

```
struct ListaCircularDobleNodo {
    Avion avion;
    ListaCircularDobleNodo* ant;
    ListaCircularDobleNodo* sig;
};
```

```
class ListaCircularDoble {
public:
    ListaCircularDobleNodo* head;
    ListaCircularDoble() : head(nullptr) {}
    void insertar(Avion avion);
    void exportToDOT(const string &ruta);
    Avion* buscarAvion(string numero_de_registro);
    void eliminar(string numero_de_registro);
};
```

3. Árbol Binario de Búsqueda (Arbol)

- **Descripción:** Implementación de un árbol binario para almacenar información de pilotos.
- **Clase:** Arbol
- **Métodos:**
  - preOrder(Node* root): Recorre el árbol en preorden.
  - inOrder(Node* root): Recorre el árbol en inorden.
  - postOrder(Node* root): Recorre el árbol en postorden.
  - eliminar(Node* root, string numero_de_id): Elimina un piloto del árbol.
  - exportToDOT(const string& archivo): Exporta la estructura del árbol a un archivo DOT para visualización.

```
struct Node {
    Piloto piloto;
    Node* left;
    Node* right;
};
```

```
class Arbol {
public:
    Node* root;
    Node* insert(Node* root, Piloto piloto);
    Arbol() : root(nullptr) {}
    void preOrder(Node* root);
    void inOrder(Node* root);
    void postOrder(Node* root);
    void eliminar(Node* root, string numero_de_id);
    void exportToDOT(const string& archivo);
    Piloto* getPiloto(Node* root, string vuelo);
    Piloto* getPilotoId(Node* root, string numero_de_id);
private:
    int i;
    void exportToDOTHelper(ofstream& file, Node* root);
    Node* findMin(Node* root);
};
```

4. Tabla de Hashing (HashTable)

- **Descripción:** Implementación de una tabla de hashing para almacenar pilotos.
- **Clase:** HashTable
- **Métodos:**
  - insert(Piloto piloto): Inserta un piloto en la tabla.
  - eliminar(Piloto* piloto): Elimina un piloto de la tabla.
  - exportToDOT(const string& archivo): Exporta la estructura de la tabla a un archivo DOT para visualización.

```
struct HashNode {
    Piloto piloto;
    HashNode* next;

    HashNode(Piloto p) : piloto(p), next(nullptr) {}
};
```

```
class HashTable {
private:
    HashNode** table;
    int size;
    int hashFunction(const std::string& key);

public:
    HashTable(int size);
    ~HashTable();
    bool insert(Piloto piloto);
    void eliminar(Piloto* piloto);
    void exportToDOT(const std::string& archivo);
};
```

5. Grafo Dirigido (Grafo)

- **Descripción:** Implementación de un grafo dirigido para representar rutas entre lugares.
- **Clase:** Grafo
- **Métodos:**
    - insertarLugar(string lugar): Inserta un lugar en el grafo.
    - insertarLugarDestino(string lugar, string destino, int distancia): Inserta una ruta entre dos lugares con una distancia.
    - exportToDOT(const string& archivo): Exporta la estructura del grafo a un archivo DOT para visualización.
    - encontrarRuta(Lugar* origen, Lugar* destino): Encuentra la mejor ruta entre dos lugares.

```
struct Lugar;
struct Destino {
    Lugar* lugar;
    int distancia;
    Destino* siguiente;
};
struct Lugar {
    int id;
    string nombre;
    Destino* destinos;
    Lugar* siguiente;
};
struct Ruta {
    int distancia;
    string ruta;
};
```

```
class Grafo {
private:
    Lugar* insertarLugar(string lugar);
public:
    Lugar* root;
    int indice = 0;
    Ruta mejorRuta;
    int visitados[0];
    Grafo() : root(nullptr) {}
    Lugar* buscarLugar(string lugar);
    void insertarLugarDestino(string lugar, string destino, int distancia);
    void exportToDOT(const string& archivo);
    Ruta encontrarRuta(Lugar* origen, Lugar* destino);
};
```

6. Matriz Dispersa (MatrizDispersa)

- **Descripción:** Implementación de una matriz dispersa para organizar datos relacionados con vuelos, pilotos y destinos.
- **Clase:** MatrizDispersa
- **Métodos:**
    - Métodos para insertar y eliminar vuelos, pilotos y destinos.
    - exportToDOT(const string& archivo): Exporta la estructura de la matriz a un archivo DOT para visualización.

```
class Celda {
public:
    int fila, columna;
    Piloto* piloto;
    Avion* vuelo;
    Lugar* destino;
    Celda* arriba, *abajo, *derecha, *izquierda;
    Celda() : arriba(nullptr), abajo(nullptr), derecha(nullptr), izquierda(nullptr),
};
```

```
class MatrizDispersa {
private:
    Celda* head;
    Grafo* grafoRutas;
    Arbol* arbolPilotos;
    BTree* arbolAviones;
    void insertarCelda(Celda* fila, Celda* columna, Celda* celda);
    Celda* getColumna(string destino);
    Celda* getFila(string vuelo);
    Celda* getCelda(Piloto* piloto);
    bool existeAvion(BTreeNode arbol, const std::string& k);
    void insertarAvion(BTreeNode arbol, const std::string& k);
public:
    MatrizDispersa(Grafo* grafoRutas, Arbol* arbolPilotos, BTree* arbolAviones) :
        head(new Celda()), grafoRutas(grafoRutas), arbolPilotos(arbolPilotos), arbolAvione
    void insertarVuelo(Avion* vuelo);
    void eliminarVuelo(Avion* vuelo);
    void insertarPiloto(Piloto* piloto);
    void eliminarPiloto(Piloto* piloto);
    void insertarDestino(Lugar* destino);
    void eliminarDestino(Lugar* destino);
    void exportToDOT(const string& archivo);
};
```