Manual Técnico

Estructura de datos

proyecto 2

DESCRIPCIÓN DEL PROGRAMA

Este sistema de simulación de tráfico inteligente ha sido desarrollado con el propósito de gestionar eficientemente la circulación de vehículos en un entorno urbano simulado, priorizando aquellos vehículos de carácter especial (como ambulancias o bomberos) y aplicando algoritmos avanzados de análisis, búsqueda y ordenación.

Tecnologías usadas

Ide: Apache Neatbean
lenguaje: java

Estructuras usadas

- **Cola**
- **lista enlazada**
- **árbol binarios avl**
- **matriz ortogonal**
- **pila**

Algoritmos utilizados

matrizOrtogonal

- **método buscarOCrear:**

Cuando busca su complejidad sera de O(n), ya que depende de los elemento que contiene la lista de encabezados.

Cuando inserta de ser necesario su complejidad es de O(1).

- **Método insertarNodo:**

Cuando se inserta un nodo, cada while tiene una complejidad O(n), ya que cada while recorre un n cantidad antes de insertar el nodo.

- **Mostrar matriz:**

Este método tiene una complejidad de O(f*c), osea la multiplicación de filas por columnas.

- **Buscar nodo:**

Este método tiene una complejidad de O(f*c), osea la multiplicación de filas por columnas, porque va buscando las columnas de cada fila.

ListaEnlazadaPistas

- **método insertarPista:**

En el mejor caso cuando la lista este vacía, es O(1), pero cuando no el while recorre n nodos hasta llegar al final entonce es o(n).

- **ImprimirLista:**

El while imprime n nodos, por lo tanto es O(n).

EncabezadoCapa

- **buscarOCrear:**

El while busca n nodos, por lo tanto es O(n).
En la creación o devuelta es O(1).

ArbolAVL

- **insertar**

Este método inserta un nodo en el árbol AVL asegurando el balance mediante rotaciones.
En el peor de los casos, la complejidad es O(log n), ya que el árbol AVL se mantiene balanceado, y por lo tanto, las inserciones solo requieren recorrer la altura del árbol, que es logarítmica en relación al número de nodos.

- **insertarRec**

Este método realiza la inserción recursiva y se encarga de aplicar las rotaciones necesarias para mantener las propiedades del árbol AVL.
Complejidad:

Peor caso: $O(\log n)$, porque el árbol nunca se desequilibra más allá de una altura logarítmica, gracias a las rotaciones.

Mejor caso: $O(\log n)$, ya que incluso cuando no hay necesidad de rotar, se debe recorrer hasta encontrar la posición de inserción.

- **buscarPorID / buscarRec**

Este método busca un nodo en el árbol por su identificador.
Complejidad:

Peor caso: $O(n)$, porque aunque el árbol está balanceado, el método primero explora el subárbol izquierdo completamente antes de pasar al derecho, sin aprovechar la propiedad ordenada del AVL (es más una búsqueda exhaustiva).

Mejor caso: $O(1)$, si el nodo buscado está en la raíz.

- **imprimirArbol / imprimirRec**

Este método recorre el árbol para imprimirlo (o generar su imagen), utilizando un recorrido en orden para mostrar todos los nodos.
Complejidad:

Siempre: $O(n)$, porque visita cada nodo exactamente una vez.

- **generarImagen / dibujarArbol**

Este método genera una representación visual del árbol en formato de imagen.
Complejidad:

Siempre: O(n), ya que para dibujar cada nodo y sus conexiones, es necesario recorrer todo el árbol.

- **rebalancear / rebalancearRec**

  Este método recorre todo el árbol y asegura que cada nodo cumpla con las condiciones de balance, aplicando rotaciones cuando sea necesario.

Complejidad:

Peor caso: O(n log n), ya que para cada nodo (n), podría ser necesario calcular el balance y realizar operaciones logarítmicas (en altura).

Mejor caso: O(n), si el árbol ya está balanceado y solo requiere un recorrido sin aplicar cambios.

.