Práctica 3. Árboles AVL

Sesiones de prácticas: 2

Objetivos

Implementar la clase `AVL<T>` utilizando **patrones de clase y excepciones**. Programa de prueba para comprobar su correcto funcionamiento.

Descripción de la EEDD

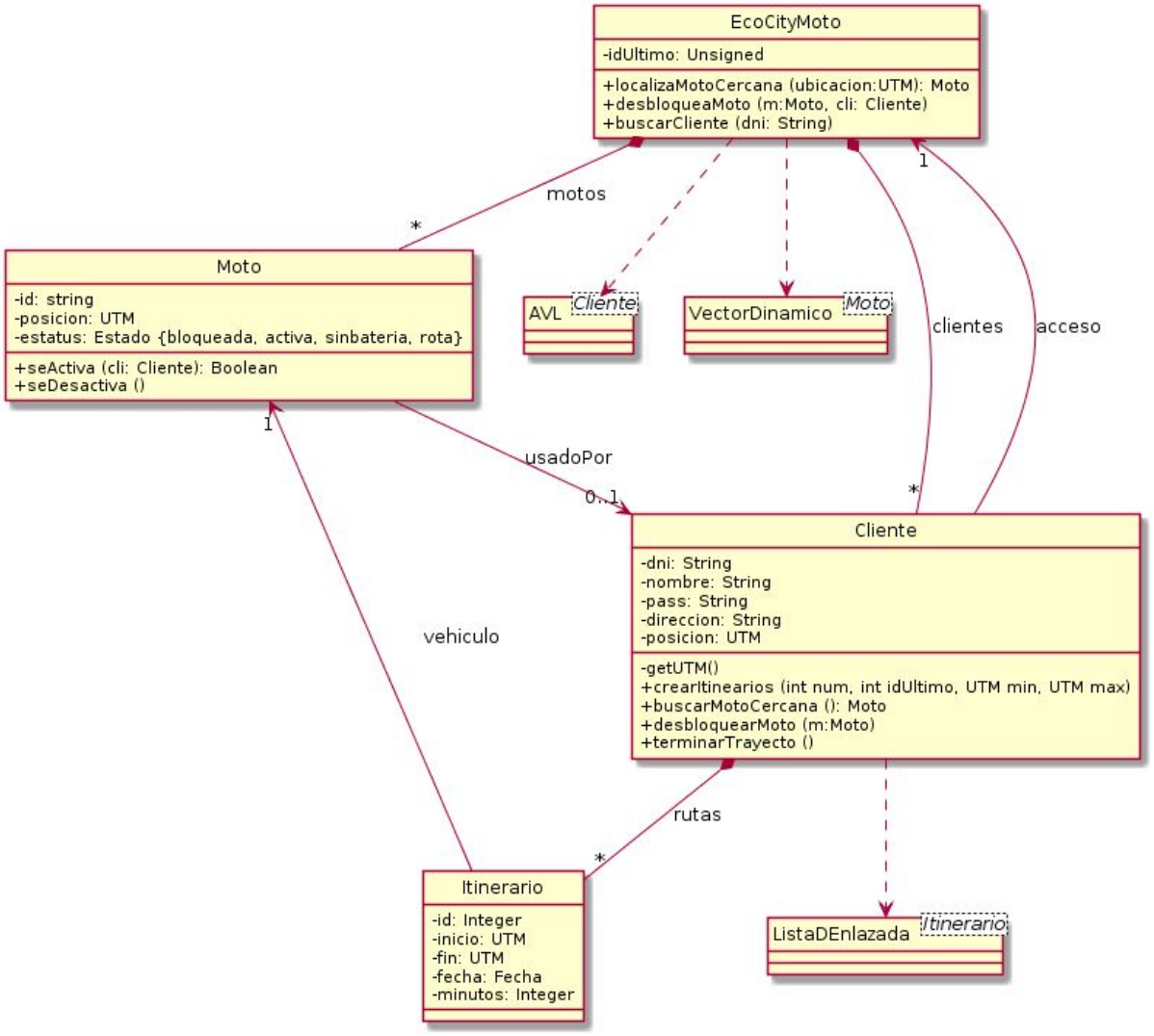
Implementar la clase `AVL<T>` para que tenga toda la funcionalidad de un árbol equilibrado AVL en memoria dinámica, tal y como se describe en la Lección 11, utilizando patrones de clase y excepciones. Los métodos a implementar serán los siguientes:

En concreto se usará:

- Constructor por defecto `AVL<T>`
- Constructor copia `AVL<T>(const AVL<T>& origen)`.
- Operador de asignación (=)
- Rotación a derechas dado un nodo `rotDer(Nodo<T>* &nodo)`
- Rotación a izquierdas dado un nodo `rotIzq(Nodo<T>* &nodo)`
- Operación de inserción `bool inserta(T& dato)`
- Operación de búsqueda recursiva `bool busca(T& dato, T& result)`
- Operación de búsqueda iterativa `bool buscaIt(T& dato, T& result)`
- Recorrido en inorden[1] `void recorreInorden()`
- Número de elementos del AVL `unsigned int numElementos()`
- Altura del AVL, `unsigned int altura()`
- Destructor correspondiente

Tanto el constructor copia como el operador de asignación deben crear copias idénticas. El contador de números de elementos puede realizarse mediante un atributo contador o mediante un recorrido en O(n).

[1] Para poder mostrar por pantalla los datos del arbol es necesario que la clase T haya implementado el operador << http://en.cppreference.com/w/cpp/language/operators



Diseño de la aplicación

La empresa EcoCityMoto es una empresa de alquiler de motos. Las motos están aparcadas en distintas localizaciones que vienen dadas por sus coordenadas UTM. Cuando un cliente solicita alquilar una moto a EcoCityMoto, ésta busca la moto más cercana al cliente que esté disponible, es decir, su estado es bloqueada (no se está usando), no está rota y tiene batería, para ello llama a *localizaMotoCercana(ubicacion:UTM): Moto*. Una vez que el cliente acepta alquilar la moto que le han ofrecido (la primera si hubiese varias), se procede a desbloquearla mediante *desbloquearMoto(id: Unsigned)*. En este momento la moto cambia de estado y se crea un nuevo itinerario comenzando por la posición donde se encuentra la moto aparcada. Una vez que el cliente recorre el trayecto deseado, se completa el itinerario y se llama a *terminarTrayecto()* que establece los minutos del trayecto (se puede tener en cuenta la hora del sistema para calcularlo) y su posición. La moto queda desactivada y dispuesta para ser utilizada de nuevo. Para saber la localización del cliente se utiliza su móvil, mediante el método getUTM(), que no se implementa.

La función *crearItinerarios(int num, int idUltimo, UTM min, UTM max)* no cambia con respecto a la Práctica 2 y sólo se encarga de crear los itinerarios históricos de forma aleatoria.

Para desarrollar esta práctica se usará el vector dinámico y la lista doblemente enlazada implementadas en las prácticas 1 y 2. Así el conjunto de motos que tiene la empresa se almacenan en un vector dinámico; los clientes se encuentran en un árbol AVL y los itinerarios realizados por los clientes se guardan en la lista doblemente enlazada.

Programa de prueba: Comprobación de la estructura de datos

Crear un programa de prueba con las siguientes indicaciones:

- Crear un árbol AVL con los clientes de la base de datos proporcionada en la práctica 1 (clientes_v2.csv). La clave ahora es el dni.
- Mostrar el árbol AVL en inorden.
- Mostrar la altura del árbol AVL
- Cargar las motos usando el fichero adjunto (motos.csv)
- Buscar un cliente en el árbol dado su DNI
- Simular que ese cliente quiere desplazarse, localiza la moto más cercana y hace un trayecto con un final aleatorio dentro del rango (37, 3) - (38,4) correspondiente a la zona de Jaén.

Estilo y requerimientos del código:

1. El código debe ser claro, tener un estilo definido y estar perfectamente indentado, para ello se pueden seguir algunos de los estilos preestablecidos para el lenguaje C++ (http://geosoft.no/development/cppstyle.html).

2. Deben comprobarse todas los posibles errores y situaciones de riesgo que puedan ocurrir (desbordamientos de memoria, parámetros con valores no válidos, etc.) y lanzar las excepciones correspondientes, siempre que tenga sentido. Leer el tutorial de excepciones disponible en el repositorio de la asignatura en docencia virtual.

3. Se valorará positivamente la calidad general del código: claridad, estilo, ausencia de redundancias, etc.