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 `T* buscaRec(T& dato)`
- Operación de búsqueda iterativa `T* buscaIt(T& dato)`
- Recorrido en inorden[1] `VDinamico<T*> recorreInorden()`
- Número de elementos del AVL `unsigned int numElementos()`
- Altura del AVL, `unsigned int altura()`
- Destructor correspondiente

Tanto el constructor de copia como el operador de asignación deben crear copias idénticas. El constructor se crea mediante un recorrido en preorden y el destructor se hace en postorden.

El contador de números de elementos puede realizarse mediante un atributo contador o mediante un recorrido en O(n).

[1] Devuelve un vector dinámico de punteros a los elementos del AVL haciendo un recorrido en Inorden

Descripción de la práctica: gestión de aeropuertos

Como ya se explicó en la Práctica 2, VuelaFlight es un nuevo proyecto para gestionar todos los aeropuertos del mundo. En esta práctica trabajaremos con el vector dinámico de aeropuertos de la Práctica 1, la lista enlazada de rutas de la Práctica 2, y además se instanciará un árbol AVL para almacenar las aerolineas que realizan las rutas entre aeropuertos.

El siguiente UML representa el diseño de *VuelaFlight* como la clase gestora de aeropuertos, rutas y aerolíneas. Se mantiene el vector dinámico de objetos de tipo *Aeropuerto* y la lista enlazada de objetos de tipo *Ruta* como composición de la clase gestora *VuelaFlight.* Ahora además añadiremos otra composición con la clase *Aerolinea*, compañía que realiza rutas entre aeropuertos. Las aerolíneas se encuentran en el fichero adjunto "aerolineas_v1.csv". En este fichero, la primera columna representa un identificador interno (no utilizado), la segunda representa el código ICAO que identifica a la aerolínea, la tercera el nombre de la aerolínea, la cuarta su país y la quinta indica si esa aerolínea está activa. Las siglas de cada aerolínea siguen el código ICAO, que identifica con 3 letras la aerolínea que realiza las rutas. Por tanto, un objeto de tipo *Aerolínea* puede realizar muchas rutas.

Para inicializar los datos de dichas clases se deben seguir los siguientes pasos:

1. Leer el fichero *"aeropuertos_v2.csv"* y cargarlos en el vector dinámico *VuelaFlight:airports* del mismo modo que en la Práctica 2.
2. Leer el fichero "aerolineas_v1.csv" y cargarlos en el AVL en la relación VuelaFlight:work añadiendo todos los atributos menos Aerolinea::aeroroutes.
3. Leer el fichero "rutas_v1.csv" para rellenar la relación *VuelaFlight:routes.* Para cada objeto ruta que se cree:
    a. buscar en tiempo logarítmico en *VuelaFlight:airports* el identificador IATA del aeropuerto de origen y destino y enlazarlo en *Ruta::origin y Ruta::destination* (igual que en la Práctica 2).
    b. buscar en tiempo logarítmico la aerolínea que hace la ruta en VuelaFlight::work, obtener el objeto Aerolínea correspondiente y añadirlo a la relación Ruta::company.
    c. Insertar ahora el nuevo objeto ruta en VuelaFlight::routes.
    d. Obtener la dirección del objeto ruta recién insertado en la lista (en la última posición).
    e. Enlazar la aerolínea encontrada antes con la ruta anterior mediante Aerolinea::linkAerolRuta.

La funcionalidad de la clase Aerolínea es la siguiente:

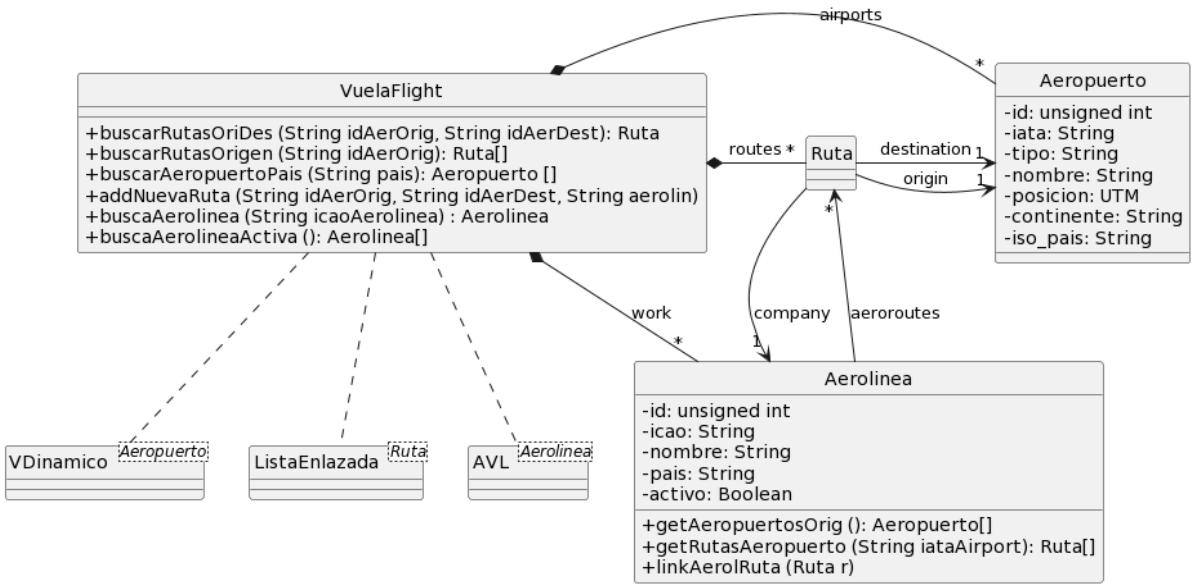
- *getAeropuertosOrig()*: Aeropuerto[], que devuelve todos los aeropuertos en los que la aerolínea funciona, para lo cual bastaría ver los aeropuertos origen.
- *getRutasAeropuerto(String iataAeropuerto)*: Ruta[], que devuelve todas las rutas cuyo aeropuerto origen o destino sea el indicado por parámetro.

La funcionalidad de la clase *VuelaFlight* se amplía en esta práctica con las siguientes búsquedas:

- *buscaAerolinea(String icaoAerolinea)*: Aerolinea, que devuelve la información de una aerolínea dado su código icao.

- *buscaAerolineasActivas()*: Aerolinea[], que devuelve todas las aerolíneas activas, es decir, que operan alguna ruta

IMPORTANTE: no devolváis objetos copia



Programa de prueba: probar la funcionalidad de VuelaFlight

1. Instanciar la clase *VuelaFlight* según el diseño UML, y rellenar el vector, la lista y el árbol AVL como se ha descrito anteriormente.
2. Visualiza toda la información de la aerolínea Ryanair, RYR
3. Muestra todas las aerolíneas activas.
4. Busca todos los aeropuertos (origen) en los que opera Iberia Airlines, con icao IBE
5. Busca todas las rutas operadas por Iberia Airlines con origen en el aeropuerto de Málaga (AGP).

Para los que trabajan por parejas:

Implementar el método *VuelaFlight::getAerolineasPais(String idPais): Aerolinea[]* que muestra todas las aerolíneas de un país dado como parámetro (utilizar el recorrido inorden). Para probarlo, muestra por pantalla la información de las aerolíneas que operan en España.

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.