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

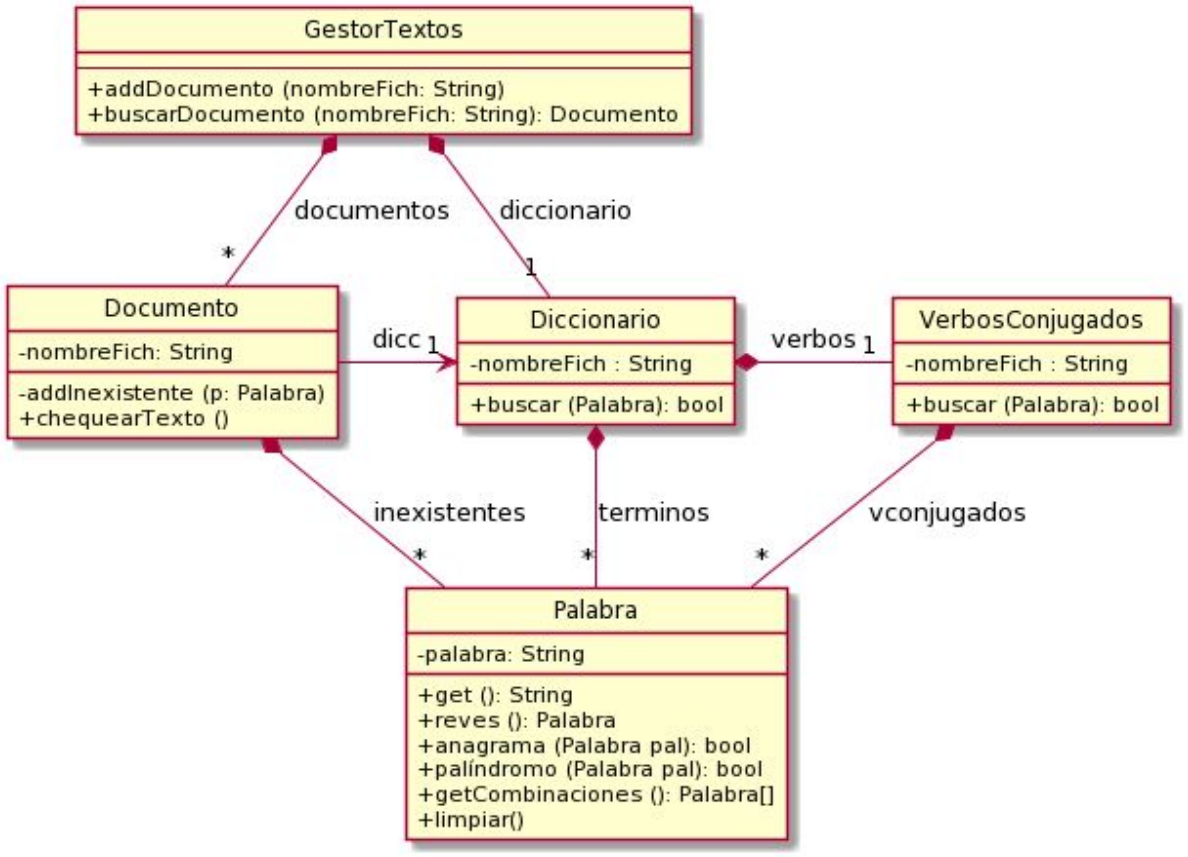
Implementar la clase `AVL<T>` para que tenga toda la funcionalidad de un árbol equilibrado AVL en memoria dinámica, tal y como se describe en la Lección 11, utilizando patrones de clase y excepciones. Los métodos a implementar serán los siguientes:

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 buscaNR(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



Programa de prueba: gestor de documentos

El gestor de archivos ha aumentado su funcionalidad y ahora es capaz de detectar palabras que no estaban en el diccionario porque son verbos conjugados, además de gestionar más de un archivo, en caso de que sea necesario.

La clase *GestorTextos* ahora puede gestionar más de un texto. Se añaden nuevos documentos con *addDocumento()*. La estructura de datos para almacenar los objetos de tipo *Documento* puede ser un vector dinámico.

En la clase *Documento* se encuentra ahora la función *chequearTexto()*, encargada de leer y chequear su propio documento. Se enlaza con *Diccionario* para poder consultar las palabras. Cada documento mantendrá también ahora una lista de palabras inexistentes, pero previsiblemente será un conjunto más pequeño, puesto que ahora tendremos en cuenta los verbos conjugados.

El *Diccionario* general no cambia, se implementa mediante un vector con las palabras de "dicc-espanol". Pero esta vez el diccionario se relaciona además con la clase *VerbosConjugados*, que mantiene el conjunto de verbos conjugados, permitiendo localizar así las palabras que no estén en dicho diccionario. Por tanto se considera que *Diccionario::buscar()* da sólo falso si dicha palabra no está ni en el diccionario principal ni en el verbos conjugados.

La clase *VerbosConjugados* se crea a partir del fichero adjunto (verbos_conjugados_desordenados) y se implementa sobre un AVL<Palabra>. Utilizamos mejor la versión desordenada para que se den todas las circunstancias posibles a la hora de insertar.

Programa de prueba: Comprobación de la estructura de datos

Crear un programa de prueba con las siguientes indicaciones:

Crear un árbol AVL con los verbos conjugados a partir del fichero adjunto (verbos_conjugados_desordenados).

- Implementar las clases anteriores de acuerdo al diagrama UML.
- Indicar la altura del árbol AVL de verbos conjugados.
- Realizar el mismo proceso de prueba que en la Práctica 2, es decir, chaquear el documento en busca de palabras inexistentes, pero ahora teniendo en cuenta los verbos conjugados antes de considerar esa palabra como inexistente. Comprobar la diferencia de tamaño de los dos conjuntos de términos inexistentes con respecto a la Práctica 2.
- Eliminar de la lista de palabras inexistentes los términos considerados nombres propios igual que se hizo en la práctica 2 y contar cuantos se eliminan.
- Mostrar por consola: el número de palabras inexistentes en comparación con la práctica 2 y el tiempo de cómputo empleado en las búsquedas de cada una de las EEDD mencionadas anteriormente.

Nota: Para los que trabajan en parejas:

- Añadir el método `void recorrePreorden()`. Mostrar por consola el recorrido del árbol con el método indicado.
- Realizar todos los apartados anteriores con otro de los libros adjuntos (divina comedia, tia tula, etc.), midiendo igualmente tiempos y haciendo un balance de palabras no encontradas, nombres propios, etc.

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.