Práctica 2. Implementación de una lista dinámica mediante plantillas y operadores en C++

Sesiones de prácticas: 2

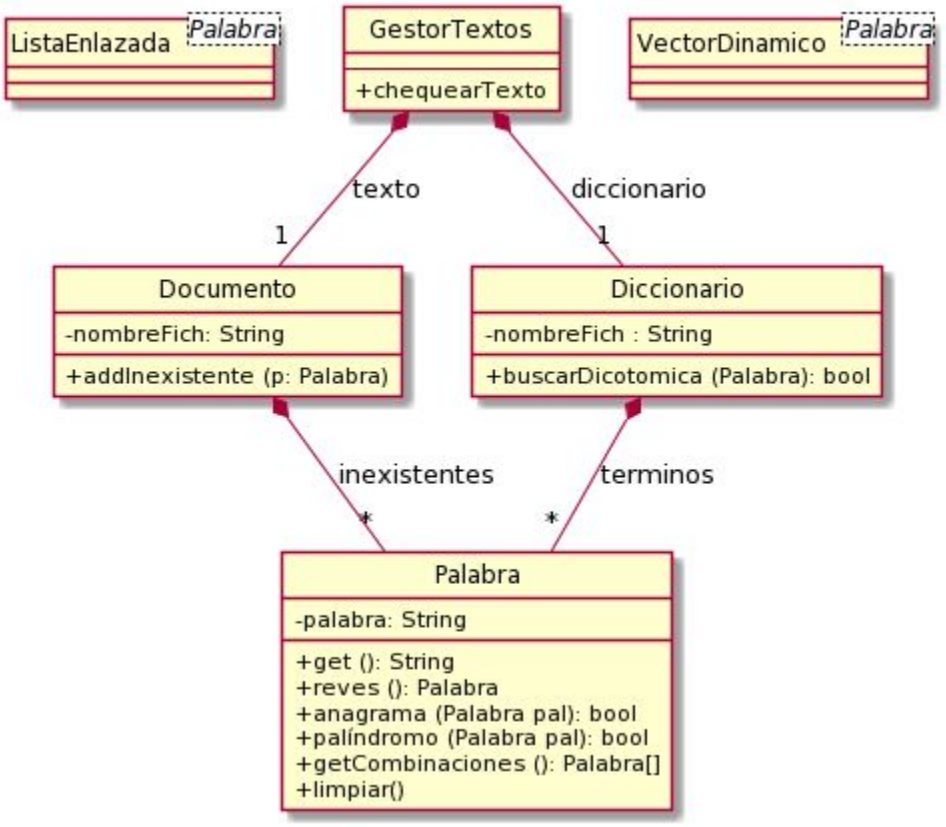
Objetivos

Implementar y utilizar la clase `ListaEnlazada<T>` y su clase auxiliar de tipo iterador ListaEnlazada<T>::`Iterador` utilizando **patrones de clase y excepciones**. Programa de prueba para comprobar su correcto funcionamiento.

Descripción de la EEDD

Implementar la clase `ListaEnlazada<T>` para que tenga toda la funcionalidad de una lista enlazada en memoria dinámica descrita en la Lección 7, utilizando patrones de clase y excepciones. Los métodos a implementar serán los siguientes:

- Constructor por defecto `ListaEnlazada<T>`
- Constructor copia `ListaEnlazada<T>(const ListaEnlazada<T>& origen)`.
- Operador de asignación (=)
- Obtener los elementos situados en los extremos de la lista: `T& inicio()` y `T& Fin()`
- Obtener un objeto iterador para iterar sobre una lista: `ListaEnlazada<T>::Iterador iterador ()`
- Insertar por ambos extremos de la lista, `void insertaInicio( T&dato)` y `void insertaFin(T& dato)`
- Insertar un dato en la posición anterior apuntada por un iterador: `void inserta(Iterador &i, T &dato)`
- Borrar el elemento situado en cualquiera de los extremos de la lista, `void borraInicio()` y `void borraFinal()`
- Borrar el elemento referenciado por un iterador: `void borra(Iterador &i)`
- Insertar un dato de forma ordenada, siempre y cuando la lista esté ordenada, sino se devuelve falso: `insertaOrdenado(T &dato): bool`
- `tam(): entero`, que devuelve de forma eficiente el número de elementos de la lista
- El destructor correspondiente.
- **Para los que trabajen en grupo:** implementar el método `buscar(T &dato, Iterador &it):bool`, que busca el *dato* de tipo *T* en la lista y devuelve *true* en caso de ser encontrado y el iterador a la posición localizada. En caso de no encontrarse, devuelve *false*.



Programa de prueba: crear un gestor de textos

El diagrama UML anterior describe un gestor de textos muy simple. Por el momento sólo nos va a indicar si el texto que le indicamos, por ejemplo el quijote.txt, tiene todas sus palabras en el diccionario, o si por el contrario existen palabras no clasificadas.

La clase *GestorTexto* será la clase principal y tendrá asociado por el momento un sólo documento y un diccionario. El diccionario representa al vector de palabras de la Práctica 1, pero encapsulado como clase *Diccionario*. La función +buscar() debe ser capaz de encontrar una palabra en el diccionario en tiempo logarítmico. La clase *Documento* representa a cada uno de los documentos a gestionar, por ejemplo el quijote.txt, cuyo nombre de fichero queda registrado. La función Documento::addInexistente() añade una palabra a la lista de palabras inexistentes.

La función GestorTextos::chequearTexto() se encarga de la lectura del fichero y de explorar cada una de las palabras del texto y comprobar si están o no en el diccionario. En caso negativo, se añade dicha palabra a su lista particular con la función Documento::addInexistente(). Como el documento original puede tener palabras pegadas a caracteres como puntos, comas, etc., la función Palabra::limpiar() debe eliminar dichos caracteres de la palabra en caso de que existan.

El diccionario no va a cambiar de tamaño, así que cada vez que una palabra del documento (quijote en este caso) no esté en el diccionario se debe añadir a una nueva lista de palabras “inexistentes”. Por tanto, será una lista de nombres propios, de verbos conjugados y de cualquier otro término no

existente en la versión que tenemos del diccionario. Se instanciará como una lista enlazada de palabras (ListaEnlazada<Palabra>) que deberá mantenerse ordenada.

Para probar esta práctica:

- Implementar la EEDD ListaEnlazada<T> con la funcionalidad señalada arriba y de acuerdo con la especificación de la Lección 7 .
- Definid el resto de clases tal y como indica el UML.
- Instanciar el diccionario como un vector de palabras, similar a como se hizo en la Práctica 1 usando el fichero *dicc-espanol.txt*
- Usar la lista enlazada para el conjunto de palabras del texto que no estén en el diccionario. El objeto de tipo *Documento* se instancia con el nombre de fichero *quijote.txt*
- Listar por pantalla todas (o unas 100) las palabras que no estén en el diccionario.
- Borrar de dicha lista aquellas palabras que sean nombres propios, es decir, que comiencen por mayúscula.

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.