Práctica 3. Implementación de una lista dinámica mediante plantillas y operadores en C++

Sesiones de prácticas: 2

Objetivos

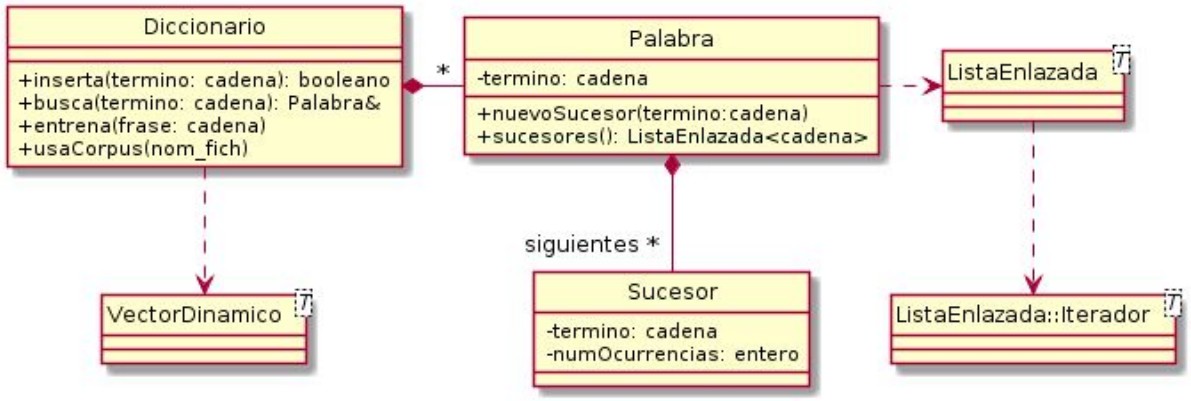
Implementar la clase `ListaEnlazada<T>` y su clase auxiliar de tipo iterador ListaEnlazada<T>::`Iterador`[1] utilizando **patrones de clase y excepciones**. Programa de prueba para comprobar su correcto funcionamiento.

Descripción de la EEDD

Implementar la clase `ListaEnlazada<T>` para que tenga toda la funcionalidad de una lista simplemente enlazada en memoria dinámica descrita en la Lección 6, 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 con orden de eficiencia *O(1)*, `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)`[2]
- El destructor correspondiente.

[1] Si definimos la clase Iterador dentro del espacio de nombres de la clase ListaEnlazada, en su sección public, los objetos creados de tipo `ListaEnlazada<T>::Iterador` pueden acceder a elementos privados de objetos del tipo lista (sin necesidad de definir relaciones de tipo *friend*)
[2] ¿Qué eficiencia considera que tienen los métodos inserta y borra a partir de un iterador?



Programa de prueba: creación de una lista de posibles términos que pueden aparecer a continuación de cada palabra del diccionario y obtención de los mismos

Con la nueva plantilla de clase `ListaEnlazada`, añadiremos a la clase Palabra un atributo de tipo lista de objetos `Sucesor`. Un sucesor contendrá un posible término que puede preceder a dicha palabra, además de disponer de información sobre la frecuencia con que suele aparecer junto a ella.

A continuación, deberemos entrenar el diccionario con frases habituales del lenguaje vinculado para conocer los sucesores habituales de cada palabra. Para ello, crearemos un nuevo método `Diccionario::entrena(frase)` que a partir de la frase proporcionada analizará las palabras que contiene y, para cada una de ellas, observará el término que la precede y actualizará con él su lista de sucesores en el diccionario. Si una palabra de la frase no aparece en el diccionario, se añadirá como nueva al diccionario además de como sucesor de la palabra en cuestión.

Para actualizar un sucesor de una palabra, debemos buscar, en primer lugar, la palabra en el diccionario y utilizar su nuevo método `Palabra::nuevoSucesor(termino)` que se encargará de localizarla en su lista de posibles sucesores para incrementar su número de ocurrencias o, si no estuviera, de insertarla.

Para facilitar el proceso de entrenamiento del diccionario, también podemos obtener las frases desde un corpus almacenado en un fichero de texto. Para ello haremos uso de un nuevo método `Diccionario::usaCorpus(nom_fich_corpus)`

Para comprobar su funcionamiento se creará un objeto `Diccionario` usando el constructor de la práctica anterior `Diccionario(const std::string& ruta)`, el cual añadirá las palabras del fichero proporcionado (por ahora seguimos suponiendo que las palabras del fichero están ordenadas por lo que simplemente se irán añadiendo a la derecha del vector). A continuación, se entrenará al diccionario con una frase determinada. Para comprobar que el proceso ha tenido éxito, se buscarán algunas de las palabras de la frase en el diccionario y se visualizarán los sucesores que se obtengan para cada una mostrando los datos devueltos por el método `Palabra::sucesores()`.

Si los resultados anteriores son correctos, probar a entrenar el diccionario a partir del fichero proporcionado `corpus_spanish.txt`[3] que contiene un corpus de frases del español.

[3] En el corpus proporcionado se han eliminado los acentos, por lo que es necesario que también se utilice como fichero de diccionario la versión correspondiente sin acentos.

Posteriormente, solicitar palabras sueltas al usuario y, para cada una de ellas, mostrar las diferentes sugerencias que ofrece el sistema.

Se puede mejorar la interactividad con el usuario haciendo que las sugerencias se muestren asociadas a valores numéricos. De esta forma, si el usuario en vez de introducir un nuevo término, introduce el valor numérico de una sugerencia mostrada, el sistema repetirá el proceso automáticamente contemplando como nueva palabra a buscar la sugerencia seleccionada.

Lectura de palabras en una frase

Ya vimos en las prácticas anteriores que las palabras se pueden leer fácilmente de los flujos de texto utilizando el operador >>, que se encarga de descartar los caracteres que no forman parte de las mismas como espacios, tabuladores y retornos de carro. Sin embargo, en ocasiones, como ocurre con el método `Diccionario::entrena(frase)`, las palabras a obtener están en una cadena de caracteres. Pues bien, utilizando la clase `std::stringstream`[4], se puede transformar una cadena de caracteres en un flujo con el que podemos utilizar las operaciones habituales de flujos.

Por lo tanto, a partir de una frase almacenada en una cadena, podemos descomponerla en sus palabras constituyentes de la siguiente forma:

```
std::string palabra, frase = "El veloz murciélago hindú comía feliz cardillo y kiwi";
std::stringstream ss;
int total = 0;
ss << frase;                    //enviamos la cadena al stream
while (!ss.eof())  {
        ss >> palabra;   //leemos la siguiente palabra
        if (palabra!="") {
                std::cout << ++total << " " << palabra << std::endl;
                palabra="";
        }
}
std::cout << "TOTAL PALABRAS: "<< total << std::endl;
```

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.

[4] Disponible en <sstream>. Consultar http://www.cplusplus.com/reference/sstream/stringstream/