

Trabajo Práctico - Algoritmos y Estructuras de Datos

Licenciatura en Tecnologías Digitales, UTDT

Primer Semestre 2023

- El TP se debe realizar en grupos de 3 personas.
- La fecha de entrega es hasta el miércoles 21 de junio inclusive.
- Se evaluará no solo la correctitud técnica de la solución propuesta sino también la claridad del código escrito.

Descripción del problema

Se desea implementar un tipo de datos Editor que modela un editor de documentos de texto. Además de las funcionalidades básicas de ingresar texto nuevo y eliminar texto existente, el editor debe proveer la funcionalidad extra de definir un conjunto inicial de palabras denominadas *conectores* que no serán tenidas en cuenta a la hora de calcular el *conteo de palabras* del documento. Además, el editor debe resolver de manera eficiente las operaciones de *búsqueda* y *reemplazo* de palabras individuales.

Consigna

1. Definir una estructura de representación en el archivo `Editor.h` que permita satisfacer los **requerimientos de complejidad**.
2. Escribir, en español, como comentario en `Editor.h` las condiciones que debe cumplir la estructura para ser válida (el invariante de representación)
 - Dar un ejemplo de valores para la estructura que cumpla el invariante.
 - Dar un ejemplo de valores para la estructura que **NO** cumpla el invariante.
3. Escribir en lógica formal el invariante de representación $Rep(e : estr)$. Pueden escribirlo como comentario en `Editor.h` usando palabras como `forall`, `exists`, `sum` para denotar los símbolos del lenguaje formal de especificación.
4. Escribir en el archivo `Editor.cpp` la implementación de los métodos respetando los **requerimientos de complejidad** y el invariante de representación. No está permitido modificar la interfaz pública de la clase.
5. Comentar en el código las complejidades **de peor caso** de los métodos implementados, incluyendo los métodos que no tienen requisito de complejidad. Pueden hacerlo comentando la complejidad de cada línea y agregando al pie del algoritmo la cuenta de complejidad total (usando álgebra de órdenes) y cualquier justificación/aclaración (en castellano) que sea necesaria sobre las complejidades anotadas. **No se pide ninguna justificación formal.**

Sugerencias para la estructura de representación:

- Utilizar clases provistas por la *Biblioteca Estándar* de C++, aprovechando sus órdenes de complejidad.
- No es necesario diseñar estructuras manejando memoria dinámica de manera explícita.

Interfaz de la clase

NOTA: Se pueden agregar las **funciones auxiliares** que crea necesarias en la parte **privada** de la clase Editor.

```

1  class Editor{
2      // Constructor
3      Editor(const set<string> & conectivos);
4
5      // Observadores
6      string texto() const;
7      const set<string>& conectivos() const;
8
9      // Otras operaciones
10     const set<string>& vocabulario() const;
11     int conteo_palabras() const;
12     const set<int> & buscar_palabra(const string& palabra) const;
13
14     // Modificadores
15     void insertar_palabras (const string& oracion, int posicion);
16     void borrar_posicion (int posicion);
17     int borrar_palabra (const string& palabra);
18     void agregar_atras (const string& oracion);
19     void reemplazar_palabra(const string& palabra, const string& reemplazo);
20
21     private:
22         /* ... */
23 };

```

Los métodos vocabulario(), conectivos() y buscar_palabra(const string& palabra) devuelven un contenedor *por referencia*. Esto significa que se debe devolver por referencia información ya almacenada y, cuando la función devuelve el contenedor, no se computa costo de copiarlo.

Requerimientos de complejidad

Editor(const set<string>& conectivos)	$O(conectivos)$
texto()	$O(N)$
conectivos()	$O(1)$
vocabulario()	$O(1)$
conteo_palabras()	$O(1)$
longitud()	$O(1)$
buscar_palabra(const string& palabra)	$O(\log M)$
insertar_palabras(const string& oracion, int pos)	sin requerimiento
borrar_posicion(int pos)	sin requerimiento
borrar_palabra(const string& s)	sin requerimiento
agregar_atras(const string& oracion)	$O(oracion * \log(MP))$
reemplazar_palabra(const string& p1, const string& p2)	$O(\log M + P \log P)$

Donde

- N es la cantidad de palabras totales (incluyendo repetidos) del texto,
- M es la cantidad de palabras *diferentes* escritas en el texto, y

- P es la cantidad máxima de repeticiones de una palabra en el texto.

Para los requerimientos de complejidad pueden asumir que el costo de copiar un string es constante $O(1)$, y que la cantidad de *conectivos* registrados está acotada por un valor constante.

Descripción detallada de las operaciones

Aclaración: se entenderán como dos apariciones de la misma palabra si y sólo si los strings correspondientes son iguales. Por ejemplo, "hola" y "Hola" son dos palabras distintas, y "hola" y "holá" también son dos palabras distintas, pero "hola" y "hola" son la misma palabra.

- `Editor(const set<string>& conectivos);`
Pre: Los strings de conectivos son palabras sin espacios ni signos de puntuación.
Post: Construye un Editor vacío con un conjunto de conectivos dado.
- `const set<string> & conectivos() const;`
Pre: Verdadero
Post: Devuelve *por referencia* el conjunto de palabras registradas como conectivos.
- `const set<string> & vocabulario() const;`
Pre: Verdadero
Post: Devuelve *por referencia* el conjunto de todas las palabras que aparecen alguna vez en el texto y que no están en conectivos().
- `int conteo_palabras() const`
Pre: Verdadero
Post: Devuelve la cantidad total de palabras que aparecen en el texto (contando repeticiones), excluyendo las que estén en conectivos().
- `int longitud() const`
Pre: Verdadero
Post: Devuelve la cantidad total de palabras del texto (contando repeticiones).

Aclaración: para las siguientes operaciones, se asume que se indexa desde 0 hasta `longitud()-1` donde cada posición corresponde a una palabra. Por ejemplo, si el texto es "hola mundo", la palabra "hola" está en la posición 0 y la palabra "mundo" está en la posición 1.

- `const set<int> & buscar_palabra(const string& palabra) const`
Pre: El string palabra no tiene espacios ni signos de puntuación.
Post: Devuelve el conjunto de posiciones del texto donde aparece la palabra buscada.
- `void insertar_palabras(const string& oracion, int posicion)`
Pre: $0 \leq \text{posicion} \leq \text{longitud()}-1$ y oracion es una secuencia de palabras separadas por un espacio, sin signos de puntuación y sin espacios al principio/final.
Post: Se inserta en orden cada una de las palabras de la oracion, a partir de la posicion indicada.
- `void borrar_posicion(int posicion)`
Pre: $0 \leq \text{posicion} \leq \text{longitud()}-1$
Post: Se elimina la palabra ubicada en esa posición del texto.
- `int borrar_palabra(const string& palabra)`
Pre: El string palabra no tiene espacios ni signos de puntuación.
Post: Se elimina la palabra indicada de todo el texto, y se devuelve la cantidad de palabras eliminadas.

- **void** agregar_atras(**const** string& oracion)
Pre: El string oracion es una secuencia de palabras separadas por un espacio, sin signos de puntuación y sin espacios al principio/final.
Post: Se agregan todas las palabras de oracion al final del texto.
- **void** reemplazar_palabra(**const** string& palabra1, **const** string& palabra2)
Pre: Los strings palabra1 y palabra2 no tienen espacios ni signos de puntuación.
Post: Se reemplazan todas las ocurrencias en el texto de palabra1 por palabra2.

Versiones de este documento

7/6 – versión inicial

13/6 – Corrección de complejidad de agregar_atras. Agregado de cálculo de complejidades en la consigna.