TD3: Algoritmos y Estructuras de Datos

Trabajo Práctico "Resaltador de Sintaxis"

15 de noviembre

- El TP debe realizarse en grupos de 3 personas.
- El plazo de entrega es hasta el Domingo 1 de Diciembre inclusive.
- Se evaluará no sólo la corrección técnica de la solución propuesta sino también la claridad del código escrito.

Descripción del problema

Se necesita proveer la implementación para el tipo de datos `Resaltador` que modela un resaltador de sintaxis (syntax highligter). El `Resaltador` trabaja en conjunto con una clase `EditorSintaxis`, que modela un editor de documentos de texto. El resaltador se encarga de resaltar con color distintas categorias de palabras. La interfaz pública de las clases `Resaltador` y `EditorSintaxis` estará dada, mientras que las estructuras de representación y sus implementaciones deberán ser completadas, respetando las complejidades temporales para mantener al resaltador y al editor eficientes.

Cada categoria tiene asociada un color y un conjunto de palabras que deben ser resaltadas. Por ejemplo, si hicieramos un editor de C++ podría tener la categoría keywords, resaltando en rojo palabras como "if", "for", "while", y otras palabras importantes. En otro contexto, podríamos tener una nota periodística y querer resaltar nombres propios en azul, por ejemplo "Leonel", "Messi".

El resaltador debe resolver de manera eficiente a qué palabra asignarle qué color y el manejo general de categorías y sus palabras asociadas. Cada operación de su interfaz pública tendrá un límite de complejidad temporal asociado.

El editor de texto debe utilizar el resaltador para implementar el coloreo y una funcionalidad extra: la de conocer eficientemente qué posiciones del texto pertenecen a una categoría dada definida en su resaltador. También debe permitir en cualquier momento cargar una nueva sintaxis en su resaltador y aplicarla al texto actual.

Consigna

1. Definir estructuras de representación para ambas clases definidas en los archivos `resaltador.h` y `editor_texto.h` que permitan cumplir los Requerimientos de complejidad.
   - Utilizar clases provistas por la *Biblioteca Estándar de C++* (`std`) teniendo en cuenta sus órdenes de complejidad.
   - No es necesario (**y no se recomienda**) diseñar estructuras manejando memoria dinámica de manera manual (`new`, `delete`).
2. Escribir en un comentario en `resaltador.h` y en `editor_texto.h` los **invariantes de representación** de las clases (es decir, las condiciones que debe cumplir la estructura para ser válida) de **dos maneras**:

- en castellano;
- en lógica formal como predicado *Rep(e: estr)*[1]

3. Escribir en los archivos `resaltador.cpp` y `editor_texto.cpp` la implementación de los métodos respetando los Requerimientos de complejidad y el **invariante de representación** especificado. No está permitido modificar la interfaz pública (sección `public`) de la clase. Está permitido definir funciones auxiliares en la parte `private` de la clase.

4. Comentar claramente 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 *O(...)* de cada línea y agregando al pie del algoritmo la cuenta de complejidad total (aplicando álgebra de órdenes) y cualquier justificación/aclaración extra que sea necesaria para las complejidades anotadas línea a línea. **No se pide ninguna justificación formal extra**.

Interfaz de la clase `Resaltador`

```
class Resaltador {
    public:
        Resaltador();
        bool es_resaltable(const string & palabra) const;
        string categoria_de_palabra(const string & palabra) const;
        int color_categoria(const string & categoria) const;
        void cargar_sintaxis(const string & archivo_sintaxis);
        void definir_categoria(const string & categoria, int color);
        void borrar_categoria(const string & categoria);
        void asignar_categoria(const string & palabra, const string& categoria);
        void desasignar_palabra(const string & palabra);
        const set<string> & categorias() const;
        const set<string> & palabras_resaltables() const;
        int color_resaltado(const string & palabra) const;
};
```

Interfaz de la clase `EditorSintaxis`

```
class EditorSintaxis {
    public:
        EditorSintaxis();
        static EditorSintaxis con_texto(const string& texto);
        int longitud() const;
        const string& palabra_en(int pos) const;
        const Resaltador & resaltador() const;
        void cargar_texto(const string& archivo_texto);
        void cargar_sintaxis(const string& archivo_sintaxis);
        void insertar_palabra(const string& palabra, int pos);
        void borrar_palabra(int pos);
        const set<int> & posiciones_de_categoria(const string & categoria) const;
};
```

Notar:

[1] Se recomienda utilizar `forall/exists/&&/||` para denotar $\forall/\ \exists/\wedge/\vee$, respectivamente. También, a modo de ejemplo, puede utilizar `sum{i=k}{n}{i}` para denotar $\sum_{i=k}^{n} i$.

- Se pueden agregar las **funciones auxiliares** que crea necesarias en la sección `private:` de las clases `Resaltador` y `EditorSintaxis`.

Requerimientos de complejidad

Dadas las siguientes magnitudes asociadas al editor:

P	cantidad de palabras con categorias asignadas
N	cantidad de palabras totales
C	cantidad de categorias

Se pide respetar las siguientes cotas de complejidad en el peor caso:

`Resaltador()`	$O(1)$
`es_resaltable(...)`	$O(\log(P))$
`categoria_de_palabra(...)`	$O(\log(P))$
`color_categoria(...)`	$O(\log(C))$
`cargar_sintaxis(...)`	sin requerimiento
`definir_categoria(...)`	$O(\log(C))$
`borrar_categoria(...)`	sin requerimiento
`asignar_categoria(...)`	sin requerimiento
`desasignar_palabra(...)`	$O(\log(P) + \log(C))$
`categorias(...)`	$O(1)$
`palabras_resaltables(...)`	$O(1)$
`color_resaltado(...)`	$O(\log(P) + \log(C))$
`EditorSintaxis()`	$O(1)$
`con_texto(...)`	sin requerimiento
`longitud()`	$O(1)$
`palabra_en(...)`	$O(1)$
`resaltador()`	$O(1)$
`cargar_texto(...)`	sin requerimiento
`cargar_sintaxis(...)`	sin requerimiento
`insertar_palabra(...)`	sin requerimiento
`borrar_palabra(...)`	sin requerimiento
`posiciones_de_categoria(...)`	$O(\log(C))$

Descripción detallada de las operaciones

Clase `Resaltador`

- `Resaltador()`
  - Pre: Verdadero
  - Post: Construye un resaltador vacío.
- `bool es_resaltable(const string &palabra) const`
  - Pre: Verdadero
  - Post: devuelve `true` si y solo si `palabra` está en alguna categoría.
- `string categoria_de_palabra(const string &palabra) const`
  - Pre: `es_resaltable(palabra)` es `true`.
  - Post: devuelve la categoria asociada a `palabra`.

- `int color_categoria(const string & categoria) const`
  - Pre: `categoria` es una categoria del resaltador.
  - Post: se devuelve el color asignado a esa categoría.
- `void cargar_sintaxis(const string& archivo_sintaxis)`
  - Pre: `archivo_sintaxis` es el nombre de un archivo bien formateado (*).
  - Post: se construyen las categorias y se asignan las palabras correspondientes a cada una.
- `void definir_categoria(string categoria, int color)`
  - Pre: `color` es un número entre 1 y 16.
  - Post: se define la categoria y se le asigna el color correspondiente.
- `void borrar_categoria(string categoria)`
  - Pre: `categoria` es una categoria del resaltador.
  - Post: se borra la categoria, y todas las palabras que tenía quedan sin categoría.
- `void asignar_categoria(const string& palabra, const string& categoria)`
  - Pre: `categoria` es una categoria del resaltador.
  - Post: se asigna o reasigna la categoria de palabra.
- `void desasignar(const string& palabra)`
  - Pre: `palabra` esta en una categoria del resaltador.
  - Post: se desasigna la categoria actual a palabra.
- `const set<string> & categorias() const`.
  - Pre: Verdadero.
  - Post: se devuelve por referencia no modificable el conjunto de categorías definidas.
- `const set<string> & palabras_resaltables() const`
  - Pre: Verdadero.
  - Post: se devuelve el conjunto de palabras resaltables.
- `int color_resaltado(const string &palabra) const`
  - Pre: `es_resaltable(palabra)` es `true`.
  - Post: Devuelve el color de resaltado.

Clase `EditorSintaxis`

- `EditorSintaxis()`
  - Pre: Verdadero
  - Post: crea un editor de sintaxis sin texto ni sintaxis
- `static EditorSintaxis con_texto(const string& texto)`
  - Pre: Verdadero
  - Post: devuelve un editor inicializado con el string `texto` partido en palabras
- `int longitud() const`
  - Pre: Verdadero
  - Post: devuelve la cantidad de palabras del texto
- `const string& palabra_en(int pos) const`
  - Pre: 0 <= pos < longitud()
  - Post: devuelve la palabra ubicada en la posición `pos`
- `const Resaltador & resaltador() const`
  - Pre: Verdadero
  - Post: devuelve el resaltador actual del editor
- `void cargar_texto(const string& archivo_texto)`
  - Pre: Verdadero
  - Post: borra el texto viejo y carga el nuevo texto del archivo `archivo_texto`, aplicando la sintaxis definida
- `void cargar_sintaxis(const string& archivo_sintaxis)`
  - Pre: el archivo `archivo_sintaxis` contiene una sintaxis correctamente escrita como se indica en [Formato de archivo de sintaxis (*)]
  - Post: se carga la nueva sintaxis y se aplica al texto actual

- `void insertar_palabra(const string& palabra, int pos)`
  - Pre: 0 <= pos <= longitud()
  - Post: se extiende la longitud del texto en 1 y se inserta la palabra nueva en la posición `pos`, desplazando un lugar hacia adelante a todas las palabras que estaban en posiciones iguales o mayores a `pos`
- `void borrar_palabra(int pos)`
  - Pre: 0 <= pos < longitud()
  - Post: elimina la palabra de la posición `pos`, desplazando un lugar hacia atrás a todas las palabras posteriores
- `const set<int> & posiciones_de_categoria(const string & categoria) const`
  - Pre: Verdadero
  - Post: se devuelve el conjunto de posiciones del texto que contienen palabras que pertenecen a `categoria` para el resaltador actual

Formato de archivo de sintaxis (*)

El método `cargar_sintaxis` debe ser capaz de leer archivos con el siguiente formato:

```
C
categoria_1 color_1 N_1
palabra_1 palabra_2 ... palabra_N_1
categoria_2 color_2 N_2
palabra_1 palabra_2 ... palabra_N_2
...
categoria_C color_C N_C
palabra_1 palabra_2 ... palabra_N_C
```

El número `C` indica la cantidad de categorias a leer. Luego vienen `C` categorías. Cada categoría tiene un encabezado formado por `categoria_X` (el nombre de la categoria como una string), `color_X` (el color asociado a la categoría) y un número X con la cantidad de palabras de esa categoría. Después viene una línea con X palabras, que pertenecen a esa categoría.

Por ejemplo, el siguiente archivo:

```
2
keywords 2 4
if then else for
otros 4 3
hola como va
```

Tiene 2 categorías: “keywords” y “otros”.

- “keywords” tiene el color 2 y contiene 4 palabras: “if”, “then”, “else”, “for”.
- “otros” tiene el color 4 y contiene 3 palabras: “hola”, “como”, “va”.

Si se usa la UI para ejecutar el proyecto, se tiene la siguiente tabla de conversión de colores:

Black	0	GrayDark	8
Red	1	RedLight	9
Green	2	GreenLight	10
Yellow	3	YellowLight	11
Blue	4	BlueLight	12
Magenta	5	MagentaLight	13
Cyan	6	CyanLight	14
GrayLight	7	White	15

Entorno de desarrollo

El código base contiene la configuración necesaria para cargar el proyecto en VSCode dentro de un container. Al seleccionar el directorio mediante *Open Folder* asegurarse de elegir `tp-codigo`, ya que en caso contrario el paso de configuración de CMake podría fallar de forma poco clara.

Una vez elegido el directorio y abierto el container, se debe configurar el proyecto con CMake tras lo cual el editor detectará tres *targets*:

- `editor-texto`: el ejecutable del editor, con interfaz gráfica interactiva (experimental)
- `editor-tests`: tests para verificar el comportamiento del editor
- `resaltador-tests`: tests para verificar el comportamiento del resaltador

Sugerencias:

- Implementar primero `Resaltador`.
- Enfocarse inicialmente en tests, ya que la interfaz gráfica no funcionará hasta que las clases `Resaltador` y `EditorSintaxis` estén 100 % implementadas.
- Escribir tests adicionales si desean probar otros casos de borde no cubiertos o hacer pruebas.

Entrega

Este trabajo debe resolverse modificando únicamente los archivos `resaltador.h`, `resaltador.cpp`, `editor_texto.h` y `editor_texto.cpp` los cuales deberán entregarse a través del campus virtual antes de que finalize el plazo de entrega.

Anexo: *Rep* formal de la clase `EditorSintaxis`

En la estructura de representación de la clase `EditorSintaxis` seguramente van a necesitar almacenar una instancia de la clase `Resaltador`. Esto significa que, en el *Rep* de `EditorSintaxis`, van a necesitar operaciones de `Resaltador` en el lenguaje formal, porque pueden tener que validar condiciones de consistencia entre alguna parte de la estructura del `EditorSintaxis` y el `Resaltador` que almacena.

Pueden asumir que, en el lenguaje formal, una instancia `r` del tipo `Resaltador` tiene las siguientes operaciones:

- *esResaltable(r:* `Resaltador`*, s:* `string`*):* `bool`, que devuelve `true` sii la palabra `s` es resaltable en el resaltador r;
- *colorResaltado(r:* `Resaltador`*, s:* `string`*):* `int` (*), devuelve el color de resaltado de `s`;
- *categoriaDePalabra(r:* `Resaltador`*, s:* `string`*):* `string` (*), que devuelve a qué categoría corresponde la palabra `s`;
- *categorias(r:* `Resaltador`*):* `set<string>`, que devuelve el conjunto de categorías definidas en el resaltador `r`.

(*): sólo definidas si *esResaltable(r,s)=*`true`

Notar que las operaciones dadas en este inciso se corresponden con algunos observadores y otras operaciones de la interfaz pública de `Resaltador`. Se dan con una sintaxis levemente distinta para remarcar que son operaciones del *mundo lógico* y para no confundirlas con las funciones de C++ de nombre parecido.

Versiones de este documento

- 15/11: Versión inicial.