TRABAJO OBLIGATORIO DE ESTRUCTURAS DE DATOS Y ALGORITMOS

MANEJADOR DE VERSIONES - SEGUNDA PARTE

El obligatorio consiste en la construcción de un sistema para modelar un manejador de versiones para archivos de texto, que extiende el editor básico de la Primera Parte.

CARACTERÍSTICAS GENERALES - SEGUNDA PARTE:

Esta parte extiende el editor básico de la Primera Parte agregando funcionalidad de versiones:

1. El sistema permite crear múltiples versiones de un archivo de texto.
2. Las versiones se identifican de manera unívoca por una secuencia de 1 o más números separados por un punto. Estos números pueden ser de más de 1 dígito y no hay límite en la secuencia de números. Las versiones nunca empiezan ni terminan con un punto. Algunos ejemplos son: 4, 2.3, 4.12.3.21, 1.2.3.4.5.6.7.8.9.10.11
3. Cada versión puede tener 0, 1 ó más subversiones dependientes.
4. Una versión depende y tiene como padre una sola versión, excepto las versiones iniciales del archivo (correspondientes a los números: 1, 2, 3…).
5. Cada versión X de un archivo incorpora 0, 1 o más modificaciones a la versión padre (si existe) de X.
6. Una versión que tiene subversiones no puede ser modificada directamente. Solo se pueden insertar o eliminar líneas en versiones que no tengan subversiones dependientes.
7. En el sistema se distingue el uso de minúsculas y mayúsculas. Por ejemplo CASA ≠ Casa ≠ casa.

EJEMPLO:

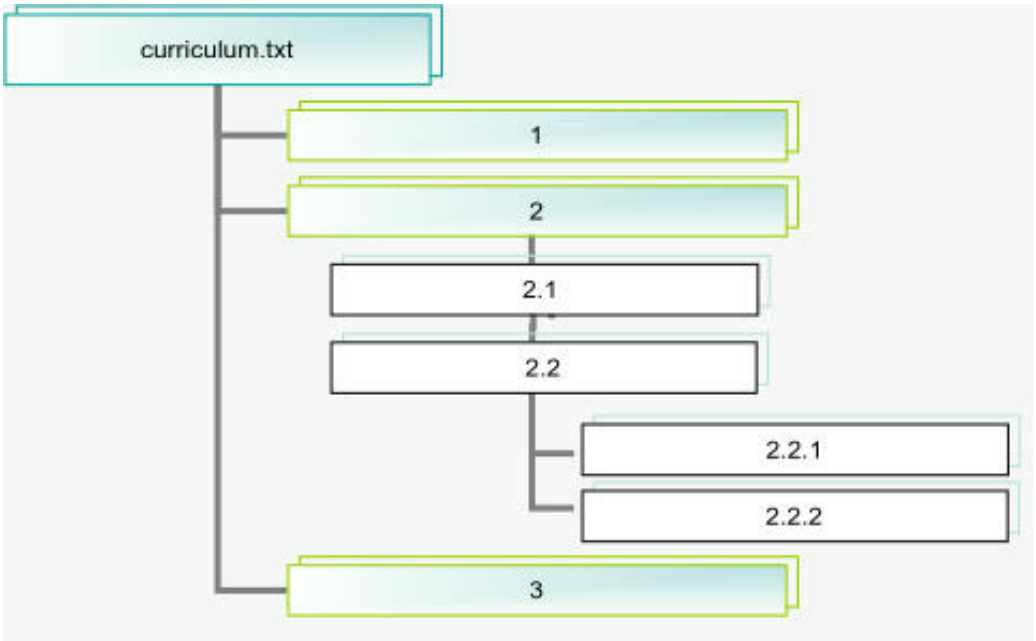

Figure 1: Estructura jerárquica de versiones del archivo curriculum.txt

En el ejemplo anterior, el esquema representa una posible estructura de versiones de un archivo con nombre “curriculum.txt”. Este archivo posee tres versiones (1, 2 y 3). A su vez, la versión 2 contiene 2 subversiones (2.1 y 2.2) y finalmente, la 2.2 tiene como subversiones a 2.2.1 y 2.2.2.

TIPOS DE DATOS A MANEJAR:

```
typedef enum {
  OK,
  ERROR,
  NO_IMPLEMENTADA
} TipoRet;

typedef struct _archivo* Archivo;
```

Pueden definirse tipos de datos (estructuras de datos) auxiliares.

El sistema debe permitir realizar las operaciones que se detallan a continuación. Cada operación, excepto CrearArchivo, puede ser ejecutada exitosamente, retornando OK, ó puede generar un error, retornando ERROR, e imprimiendo: ERROR: mensaje. Los mensajes posibles para cada operación del sistema serán indicados. En caso de producirse un error, la estructura que representa a las versiones de un archivo (de tipo Archivo) permanecerá inalterada.

En la descripción de cada operación se incluye al menos un ejemplo. En todas las operaciones, excepto en la primera, se retoma el resultado del ejemplo del caso anterior.

OPERACIONES DE TEXTO ACTUALIZADAS:

Las siguientes operaciones de la Primera Parte ahora trabajan con versiones específicas:

InsertarLinea

```
TipoRet InsertarLinea(Archivo &a, char * version, char * linea, unsigned int nroLinea);
```

Esta función inserta una línea de texto en la versión especificada del archivo en la posición nroLinea. Las mismas reglas de la Primera Parte se aplican, pero ahora se debe especificar la versión sobre la cual trabajar.

Restricción importante: Si la versión especificada tiene subversiones, no se pueden insertar líneas en ella. Solo se pueden modificar versiones que no tengan subversiones dependientes.

Retornos posibles:
- **OK** Si se pudo insertar la línea en la posición especificada del archivo.
- **ERROR** Si nroLinea no es válido.

Si la versión especificada tiene subversiones dependientes. Si la versión especificada no existe.
- **NO_IMPLEMENTADA** Cuando aún no se implementó. Es el tipo de retorno por defecto.

BorrarLinea

```
TipoRet BorrarLinea(Archivo &a, char * version, unsigned int nroLinea);
```

Esta función elimina una línea de texto de la versión especificada del archivo en la posición nroLinea. Las mismas reglas de la Primera Parte se aplican, pero ahora se debe especificar la versión sobre la cual trabajar.

Restricción importante: Si la versión especificada tiene subversiones, no se pueden eliminar líneas de ella. Solo se pueden modificar versiones que no tengan subversiones dependientes.

Retornos posibles:
- **OK** Si se pudo eliminar la línea con éxito.
- **ERROR** Si nroLinea no es válido.

Si la versión especificada tiene subversiones dependientes. Si la versión especificada no existe.
- **NO_IMPLEMENTADA** Cuando aún no se implementó. Es el tipo de retorno por defecto.

MostrarTexto

```
TipoRet MostrarTexto(Archivo a, char * version);
```

Esta función muestra el texto completo de la versión especificada, teniendo en cuenta los cambios realizados en dicha versión y en las versiones ancestras de la cual ella depende.

OPERACIONES RELATIVAS A LAS VERSIONES:

En las siguientes operaciones asumimos como precondición general que el archivo parámetro (de tipo puntero) está definido, en particular tiene nombre asignado (aunque su contenido sea eventualmente vacío y no tenga versiones).

1) Crear una nueva versión.

```
TipoRet CrearVersion(Archivo &a, char * version);
```

Crea una nueva versión del archivo si la versión especificada cumple con las siguientes reglas:

- El padre de la nueva versión a crear ya debe existir. Por ejemplo, si creo la versión 2.15.1, la versión 2.15 ya debe existir.

Las versiones del primer nivel no siguen esta regla, ya que no tienen versión padre.

- La versión especificada no debe existir previamente.

Cada subversión de una versión X de un archivo incorpora, eventualmente, cambios (inserción y/o supresión de líneas) a la versión X. Ver las operaciones relativas al texto que se describen más adelante en este documento.

Ejemplo:

```
CrearVersion(a, "1");
CrearVersion(a, "2");
CrearVersion(a, "2.1");
CrearVersion(a, "2.3");  // Se puede crear sin necesidad de 2.2
CrearVersion(a, "2.3.1");
MostrarVersiones(a);
```

Salida:

```
curriculum.txt

1
2
2.1
2.3
2.3.1
```

Retornos posibles:

- **OK** Si se pudo crear la nueva versión con éxito.
- **ERROR** Si la versión padre no existe (ver arriba).

Si la versión especificada ya existe.

- **NO_IMPLEMENTADA** Cuando aún no se implementó. Es el tipo de retorno por defecto.

Ejemplo de restricción de modificación: Una vez que se crean subversiones, la versión padre no puede ser modificada directamente:

```
CrearVersion(a, "1");
CrearVersion(a, "1.1");
InsertarLinea(a, "1", "Nueva línea", 1);  // ERROR: versión "1" tiene subversiones
BorrarLinea(a, "1", 1);                   // ERROR: versión "1" tiene subversiones
```

2) Borrar una versión, junto con sus hijos (subversiones), liberando toda la memoria involucrada.

```
TipoRet BorrarVersion(Archivo &a, char * version);
```

Elimina una versión del archivo si la version pasada por parámetro existe. En otro caso la operación quedará sin efecto. Si la versión a eliminar posee subversiones, éstas deberán ser eliminadas también, así como el texto asociado a cada una de las versiones. El texto será explicado más adelante.

No deben quedar números de versiones libres sin usar. Por lo tanto cuando se elimina una versión, las versiones hermanas que le siguen deben decrementar su numeración.

Si el archivo posee únicamente una versión (la 1), al eliminarla el archivo quedará vacío como cuando fue creado, es decir, únicamente con su nombre y contenido nulo.

Ejemplo: Eliminar una versión específica del archivo.

Retornos posibles:
- **OK** Si se pudo eliminar la versión con éxito.
- **ERROR** Si version no existe.
- **NO_IMPLEMENTADA** Cuando aún no se implementó. Es el tipo de retorno por defecto.

3) Mostrar todas las versiones de un archivo de forma jerárquica.

```
TipoRet MostrarVersiones(Archivo a);
```

FORMATO: En primer lugar muestra el nombre del archivo. Después de una línea en blanco lista todos las versiones del archivo ordenadas por nivel jerárquico e indentadas según muestra el siguiente ejemplo (cada nivel está indentado por un tabulador).

Ejemplo:

```
CrearVersion(a, "1.2");
CrearVersion(a, "2");
CrearVersion(a, "1.2.1");
MostrarVersiones(a);
```

Salida:

```
curriculum.txt

1
1.2
1.2.1
2
```

Si el archivo no contiene versiones se mostrará la siguiente salida.

Salida:

```
curriculum.txt

No hay versiones creadas
```

Retornos posibles:

- **OK** Siempre retorna OK.
- **ERROR** No existen errores posibles.
- **NO_IMPLEMENTADA** Cuando aún no se implementó. Es el tipo de retorno por defecto.

4) Chequear si dos versiones son iguales.

```
TipoRet Iguales(Archivo a, char * version1, char * version2, bool &iguales);
```

Esta función asigna al parámetro booleano (iguales) el valor true si ambas versiones (version1 y version2) del archivo tienen exactamente el mismo texto, y false en caso contrario.

Ejemplo: Comparar si dos versiones tienen el mismo contenido de texto.

Retornos posibles:
- **OK** Si se pudieron comparar las versiones.
- **ERROR** Si version1 o version2 no existen.
- **NO_IMPLEMENTADA** Cuando aún no se implementó. Es el tipo de retorno por defecto.

5) Contar el total de versiones en el archivo.

```
TipoRet ContarVersiones(Archivo a, int &cantidad);
```

Esta función asigna al parámetro cantidad el número total de versiones que existen en el archivo, incluyendo todas las subversiones.

Ejemplo: Si el archivo tiene las versiones 1, 1.1, 1.2, 2, 2.1, entonces cantidad será 5.

Retornos posibles:
- **OK** Si se pudo contar las versiones correctamente.
- **ERROR** No existen errores posibles.
- **NO_IMPLEMENTADA** Cuando aún no se implementó. Es el tipo de retorno por defecto.

CATEGORÍA DE OPERACIONES - SEGUNDA PARTE

TIPO 1	Operaciones **imprescindibles** para que el trabajo obligatorio sea corregido.
TIPO 2	Operaciones importantes. Estas serán probadas independientemente, siempre que estén correctamente implementadas las operaciones de TIPO 1.
TIPO 3	Funciones **opcionales**. Estas operaciones son adicionales y no son requeridas para la aprobación del obligatorio.

TIPO 1	TIPO 2	TIPO 3
1) CrearVersion	2) BorrarVersion	3) MostrarVersiones
2) Iguales		4) ContarVersiones

INFORMACIÓN IMPORTANTE:

- **Modalidad:** El obligatorio será de a dos personas.
- **Entrega:** El obligatorio deberá ser entregado mediante Moodle, con tiempo límite **17 de noviembre 23:59hs**.
- **Defensa:** El obligatorio tendrá una etapa de defensa el **19 de noviembre**.
- **Requisitos técnicos:** Los alumnos deberán entregar código que compile en Linux.
- **Material de apoyo:** Este documento va acompañado de sets de pruebas básicos junto con la salida esperada (ver archivos `.in.txt` y `.out.txt`). También se entrega un archivo `editor.cpp` con la interfaz de usuario (menú) para que el alumno se enfoque en la implementación del obligatorio.

CÓMO USAR LOS TESTS MANUALMENTE:

Para probar tu implementación, puedes usar los archivos de test de la siguiente manera:

Ejemplo básico:

```
# Compilar tu programa
g++ -o mi_editor editor.cpp mi_implementacion.c

# Ejecutar un test específico
./mi_editor < tests/test_basico.in.txt

# Comparar con la salida esperada
./mi_editor < tests/test_basico.in.txt > mi_salida.txt
diff tests/test_basico.out.txt mi_salida.txt
```

Si hay diferencias, el comando `diff` mostrará algo como:

```
$ diff tests/test_basico.out.txt mi_salida.txt
5a6
> ERROR: Función InsertarLinea no implementada.
8a10
> ERROR: Función BorrarLinea no implementada.
```