ORGANIZACIÓN DE COMPUTADORAS
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Segundo Cuatrimestre de 2017

Proyecto N° 1
Programación en lenguaje C

Propósito

El objetivo principal del proyecto, es implementar en lenguaje C un programa que registre la frecuencia de aparición de palabras que forman parte de un archivo de texto.

Con este objetivo se debe implementar:

- TDA Lista, para almacenar elementos de tipo genérico.
- TDA Lista Ordenada, el cual haciendo uso del TDA Lista deberá permitir almacenar elementos en forma ordenada. El orden de los elementos será determinado por una función diseñada específicamente para ese propósito.
- TDA Trie, para almacenar cadenas de caracteres junto con un contador asociado a cada cadena. Permite recuperar fácilmente tanto palabras como su correspondiente contador.
- Un programa principal, el cual debe tomar como argumento por línea de comandos el nombre de un archivo de texto, recorrer el mismo, calcular la frecuencia de aparición de las palabras que lo componen y ofrecer un conjunto de operaciones de consulta y manipulación del Trie.

1. TDA Lista

Implementar un TDA Lista en lenguaje C, cuyos elementos sean punteros genéricos. La lista debe ser simplemente enlazada sin centinelas, cuya implementación provea las siguientes operaciones:

1. `TLista crear_lista()` Crea y retorna una lista vacía.
2. `int l_insertar(TLista lista, TPosicion pos, TElemento elem)` Agrega el elemento `elem` en la posición anterior a `pos`, dentro de la lista. Si `pos` es `POS_NULA`, inserta el elemento en la primer posición de la lista. Retorna verdadero si procede con éxito, falso en caso contrario.
3. `int l_eliminar(TLista lista, TPosicion pos)` Elimina el elemento en la posición `pos`. Reacomoda la lista adecuadamente. Retorna verdadero si procede con éxito, falso en caso contrario. Si la posición no es válida retornar `LST_POS_INV`.
4. `TPosicion l_primera(TLista lista)` Retorna la primer posición de la lista. Si la lista es vacía retornar `POS_NULA`.
5. `TPosicion l_ultima(TLista lista)` Retorna la última posición de la lista. Si la lista es vacía retornar `POS_NULA`.

6. `TPosicion l_anterior(TLista lista, TPosicion pos)` Retorna la posición anterior a `pos` en la lista `lista`. Si `pos` es la primer posición de la lista, retornar `POS_NULA` .

7. `TPosicion l_siguiente(TLista lista, TPosicion pos)` Retorna la posición siguiente a `pos` en la lista `lista`. Si `pos` es la última posición de la lista o `POS_NULA`, retorna `POS_NULA`.

8. `TElemento l_recuperar(TLista lista, TPosicion pos)` Retorna el elemento correspondiente a la posición `pos`. Si la posición es `POS_NULA`, retornar `ELE_NULO`.

9. `int l_size(TLista lista)` Retorna la cantidad de elementos de la lista.

En los casos anteriormente indicados, sin considerar la operación `crear_lista`, si la lista parametrizada no está inicializada, se debe abortar con *exit status* `LST_NO_INI`.

Para la implementación, se debe considerar que los tipos `TLista`, `TCelda`, `TPosicion` y `TElemento`, están definidos de la siguiente manera:

```
typedef struct lista {
  unsigned int cantidad_elementos;
  TCelda primer_celda;
} * TLista;

typedef struct celda {
  TElemento elemento;
  struct celda * proxima_celda;
} * TCelda;

typedef struct celda * TPosicion;
typedef void * TElemento;
```

2. TDA Lista Ordenada

Implementar un TDA Lista Ordenada en lenguaje C, haciendo uso del TDA Lista, cuyos elementos sean punteros genéricos. El orden de los elementos en la lista se especifica al momento de la creación, a través de una función de comparación. La implementación debe proveer las operaciones:

1. `TListaOrdenada crear_lista_ordenada(int (*f)(void *,void *))` Crea y retorna una lista ordenada vacía. El orden de los elementos insertados estará dado por la función de comparación `int f(void *,void *)`. Se considera que la función `f` devuelve -1 si el orden del primer argumento es menor que el orden del segundo, 0 si el orden es el mismo, y 1 si el orden del primer argumento es mayor que el orden del segundo.

2. `int lo_insertar(TListaOrdenada lista, TElemento elem)` Agrega el elemento `elem` en la posición correspondiente de la lista, de modo que la misma quede siempre ordenada de forma ascendente. Retorna verdadero si procede con éxito, falso en caso contrario.

3. `int lo_eliminar(TListaOrdenada lista, TPosicion pos)` Elimina el elemento en la posición `pos`. Reacomoda la lista adecuadamente al eliminar en posiciones intermedias. Retorna verdadero si procede con éxito, falso en caso contrario.

4. `int lo_size(TListaOrdenada lista)` Retorna la cantidad de elementos de la lista.

5. `TPosicion lo_primera(TListaOrdenada lista)` Retorna la primer posición de la lista.

6. `TPosicion lo_ultima(TListaOrdenada lista)` Retorna la última posición de la lista.

7. `TPosicion lo_siguiente(TListaOrdenada lista, TPosicion pos)` Retorna la posición siguiente a `pos` en la lista.

En los casos anteriormente indicados, sin considerar la operación `crear_lista_ordenada`, si la lista parametrizada no está inicializada, se debe abortar con *exit status* `LST_NO_INI`.

Para la implementación, se debe considerar que `TListaOrdenada` está definido de la siguiente manera:

```
typedef struct lista_ordenada {
  unsigned int cantidad_elementos;
  TLista lista;
} * TListaOrdenada;
```

3. TDA Trie

Implementar un TDA Trie en lenguaje C, cuyos nodos tienen como rótulo un *caracter* (`char`), y almacena adicionalmente un contador de tipo *entero* (`int`). El trie debe implementarse manteniendo referencia a un nodo raíz, y considerando cada nodo del árbol como una estructura que mantiene referencia al padre y una lista ordenada de nodos como hijos. La implementación debe proveer las operaciones:

1. `TTrie crear_trie()` Retorna un nuevo trie vacío, esto es, con nodo raíz que mantiene rótulo nulo y contador en cero.

2. `int tr_insertar(TTrie tr, char* str)` Inserta el string `str` en el trie, inicializando el valor de contador asociado en uno. En caso de que el string ya se encuentre representado en el trie, aumenta el valor del contador asociado a dicho string en una unidad. Retorna verdadero si la inserción fue exitosa, falso en caso de que el string ya perteneciera al trie.

3. `int tr_pertenece(TTrie tr, char* str)` Retorna verdadero si el string `srt` pertenece al trie, falso en caso contrario.

4. `int tr_recuperar(TTrie tr, char* str)` Retorna el entero asociado al string `str`, dentro del trie. Si el string no pertenece al trie, retorna `STR_NO_PER`.

5. `int tr_size(TTrie tr)` Retorna la cantidad de palabras almacenadas en el trie.

6. `int tr_eliminar(TTrie tr, char* str)` Elimina el string `str` dentro del trie, liberando la memoria utilizada. Retorna verdadero en caso de operación exitosa, y falso en caso contrario.

En los casos anteriormente indicados, sin considerar la operación `crear_trie`, si el trie parametrizado no está inicializado, se debe abortar con *exit status* `TRI_NO_INI`.

Para la implementación, se deben considerar que los tipos `TTrie` y `TNodo` están definidos de la siguiente manera:

```
typedef struct trie {
  unsigned int cantidad_elementos;
  TNodo raiz;
} * TTrie;

typedef struct nodo {
  char rotulo;
  unsigned int contador;
  struct nodo * padre;
  TListaOrdenada hijos;
} * TNodo;
```

4. Programa Principal

Implementar una aplicación de consola que, recibiendo como argumento por línea de comandos el nombre de un archivo de texto, compuesto por todo tipo de caracteres, contabilice la cantidad de apariciones de cada palabra en el archivo. Se considerará como palabra toda secuencia de caracteres S, tal que:

$$S =< c_1, \ldots, c_n >, n > 0$$
$$c_i \in \{\text{a,}\ldots\text{,z}\} \cup \{\text{á,é,í,ó,ú}\}, \forall i(0 < i <= n).$$

El programa debe ofrecer un menú de operaciones, con las que el usuario luego puede consultar y manipular el estado del Trie:

1. **Mostrar palabras:** permite visualizar el listado de todas las palabras junto con la cantidad de apariciones de la misma.
2. **Consultar:** permite determinar si una palabra ingresada pertenece o no al archivo, y en consecuencia, cuántas veces esta se repite en el archivo.
3. **Comienzan con:** permite consultar cuántas palabras comienzan con una letra dada.
4. **Es prefijo:** permite consultar si una palabra ingresada es prefijo de otras almacenadas en el trie.
5. **Porcentaje prefijo:** dado un prefijo, indica el porcentaje de palabras del trie que comienzan con él.
6. **Salir:** permite salir del programa, liberando toda la memoria utilizada por el Trie.

El programa implementado, denominado `evaluador`, debe conformar la siguiente especificación al ser invocado desde la línea de comandos:

```
$ evaluador <archivo_texto>
```

El parámetro `archivo_texto`, indica el archivo a partir del cual se contabilizará la frecuencia de aparición de las palabras que lo componen. En caso de ingresar un parámetro erróneo, se debe mostrar un mensaje indicando el error, y finalizar la ejecución.

Sobre la implementación

- Los archivos fuente principales se deben denominar **lista.c**, **lista_ordenada.c, trie.c y evaluador.c** respectivamente. En el caso de las librerías, también se deben adjuntar los respectivos archivos de encabezados **lista.h, lista_ordenada.h y trie.h**, los cuales han de ser incluidos en los archivos fuente de los programas que hagan uso de las mismas.
- Es importante que durante la implementación del proyecto se haga un uso cuidadoso y eficiente de la memoria, tanto para la reservar (`malloc`), como para liberar (`free`) el espacio asociado a variables y estructuras.
- Se deben respetar con exactitud los nombres de tipos y encabezados de funciones especificados en el enunciado. Los proyectos que no cumplan esta condición quedarán automáticamente desaprobados.
- La compilación debe realizarse con el *flag* `-Wall` habilitado. El código debe compilar **sin advertencias** de ningún tipo.
- La copia o plagio del proyecto es una falta grave. Quien incurra en estos actos de deshonestidad académica, desaprobará automáticamente el proyecto.

Sobre el estilo de programación

- El código implementado debe reflejar la aplicación de las técnicas de programación modular estudiadas a lo largo de la carrera.
- En el código, entre eficiencia y claridad, se debe optar por la claridad. Toda decisión en este sentido debe constar en la documentación que acompaña al programa implementado.
- El código debe estar indentado, comentado, y debe reflejar el uso adecuado de nombres significativos para la definición de variables, funciones y parámetros.

Sobre la documentación

Los proyectos que no incluyan documentación estarán automáticamente desaprobados. La misma debe:

- Estar dirigida a usuarios finales y desarrolladores.
- Explicar detalladamente los programas realizados, incluyendo el diseño de la aplicación y el modelo de datos utilizado, así como toda decisión de diseño tomada, y toda observación que se considere pertinente.
- Incluir explicación de todas las funciones implementadas, indicando su prototipo y el uso de los parámetros de entrada y de salida (tanto dentro del código fuente como en la documentación del proyecto). Se espera que la explicación no sea una mera copia del código fuente, sino más bien una síntesis de lo implementado a través de diagramas, pseudocódigos, o cualquier representación que considere adecuada.
- En general, se deben respetar todas las consignas indicadas en la "Guía para la documentación de proyectos de software" entregada por la cátedra.

Sobre la entrega

Toda comisión que no cumpla con los requerimientos, estará automáticamente desaprobada. Los mismos son:

- Las comisiones estarán conformadas por 2 alumnos, y serán las que oportunamente registró y notificó la cátedra.
- La entrega del código fuente y la documentación se realizará a través de un archivo comprimido **zip** o **rar**, denominado ***PR1-Apellido1-Apellido2***, que debe incluir las siguientes carpetas:
  - **Fuentes**, donde se deben incorporar los archivos fuente ".c" y ".h" (ningún otro).
  - **Documentación**, donde se debe incorporar el informe del proyecto en formato PDF (ningún otro).
- El archivo comprimido debe enviarse por e-mail, respetando el siguiente formato:
  - **Para:** *gabriela.diaz@cs.uns.edu.ar*
  - **Asunto:** *OC :: PR1 :: COM XX :: Apellido1 - Apellido2*
  - **Cuerpo del e-mail:**
    Se adjunta Proyecto Nº 1, de la comisión XX:
    Apellido, Nombre 1 - LU 1
    Apellido, Nombre 2 - LU 2
- El e-mail debe ser enviado con anterioridad al día **Martes 10 de Octubre de 2017, 22:00 hs**. Se considerará como hora de ingreso, la registrada en el servidor de e-mail del DCIC.

Sobre la corrección

- La cátedra evaluará tanto el **diseño** e **implementación** como la **documentación** y **presentación** del proyecto, y el cumpliemiento de **todas** las condiciones de entrega.
- Tanto para compilar el proyecto, como para verificar su funcionamiento, se utilizará la máquina virtual "OCUNS" publicada en el sitio web de la cátedra.

Sobre las constantes a utilizar

Se considerarán los siguientes valores para las constantes definidas en las especificaciones de las operaciones del proyecto:

Constante	***Valor***	***Significado***
`FALSE`	`0`	Valor lógico falso.
`TRUE`	`1`	Valor lógico verdadero.
`LST_NO_INI`	`2`	Intento de acceso inválido sobre lista sin inicializar.
`LST_POS_INV`	`3`	Intento de acceso a posición inválida en lista.
`LST_VAC`	`4`	Intento de acceso inválido sobre lista vacía.
`TRI_NO_INI`	`5`	Intento de acceso inválido sobre Trie sin inicializar.
`STR_NO_PER`	`-1`	String no perteneciente a trie.
`POS_NULA`	`NULL`	Posición nula.
`ELE_NULO`	`NULL`	Elemento nulo.