Glosario Trabajo Práctico Final

- El programa está dividido en 3 módulos, uno para todo lo que tiene que ver con la creación y lectura de los archivos llamado, Diccionario, otro para la creación del Árbol, que contiene el árbol de listas en el cual se basa el buscador, y el ultimo modulo seria el del Usuario, con todas las funciones pedidas por el trabajo práctico.

Funciones del diccionario (primer módulo):

int filtrarCarater(char caracter);
- Función encargada de filtrar solo caracteres desde los archivos de texto, sin espacios ni comas ni puntos, ni números.

int contarPalabras(char *nombre, int iD);
- Retorna la cantidad de palabras, para poder crear el arreglo con la cantidad justa necesaria de celdas (validos).

int openDIR();
- Función encargada de abrir un directorio de archivos, sin importar el nombre, todos los archivos que se encuentren dentro de la carpeta DIR los va a leer, y va a filtrar caracter por caracter pasandolos a un arreglo de términos.

void cargarVector(char *nombre, int iD, termino *array, int *j);
- Esta función carga el arreglo de términos, con toda la información de las palabras.

void cargarDiccionario(termino *arreglo, int validos);
- Carga/crea un archivo de nombre Diccionario.dat, con la información de las palabras teniendo en cuenta la posición, el ID Documento y la palabra en sí.

void mostrararchivoDiccionario();
- Función simple para mostrar diccionario.dat

void cargarNuevoArchivo();
- Creada por nosotros, con la intención de que el usuario no tenga que meterse en la carpeta directorio ni crear un archivo a mano. Esta función le pide al usuario que nombre le quiere adjudicar al archivo, y luego le pide ingresar un texto de hasta 5000 caracteres, luego se vuelve a llamar al main y se vuelve a cargar el arreglo con las nuevas palabras, para actualizar el árbol con el nuevo archivo.

Grupo : Ignacio Pavone, Bruno Starita, Franco Ferro

Funciones del motor (segundo módulo):

void insertarenArbol(nodoA **arbol, termino info);
- Carga el Árbol de listas, ordenado por palabras, y sumando la frecuencias de las palabras cada vez que las inserta, para obtener la cantidad de veces que se repite una palabra ingresada al árbol.

void cargarMotor(nodoA **motor);
- Mientras se hace una lectura del archivo Diccionario.dat creado en el primer módulo, leyendo palabras de estructura Termino, se llama a la función insertarenArbol(), pasandole el término por parametros, para ir guardando cada palabra con toda su información en el árbol de listas.

void mostrarSublista(nodoT *ocurrencias);
- Función encargada de mostrar la lista de ocurrencias con la posición y el id documento de cada palabra.

void mostrarMotor(nodoA *arbol);
- Mostrando el motor de forma “Inorder”, para poder visualizar si fue cargado correctamente.

Funciones del usuario (tercer módulo):

void buscarPalabra(nodoA *motor, char palabra[20]);
- Primer ejercicio del trabajo práctico, recibiendo una palabra por parámetro la función se va a encargar de recorrer el árbol para derecha y para izquierda buscando por todos nodos la palabra hasta que la encuentre recursivamente, una vez encontrada se imprimirá por pantalla toda su información.

void comprobarpalabraenDOCS(nodoA *motor, char palabra[20], int idDOC1, int idDOC2);
- Función principal utilizada por el usuario para el ejercicio 2 que busca en 2 id docs que el usuario ingrese una palabra para verificar su existencia en los mismos. Esta función, primero busca si la palabra existe en el Árbol, si esto es así, va a recorrer la sublista “ocurrencias”, verificando que esta palabra esté en uno de esos documentos, si esto es así retorna el ID doc para luego compararlo. Luego en la función principal se hace esta verificación para los dos IDS, si los 2 retornan 1, entonces la palabra existe en ambos documentos.

int buscarPalabraArchivos(nodoA *motor, char palabra[20], int idDOC);
- Si la palabra existe, y el id doc es igual al ingresado por el usuario retorna 1.

int recorrerSublista(nodoT *ocurrencias, int id);
- Recorre sublista y retorna el IDdoc si en las ocurrencias aparece el ID doc.

void buscardosTerminos(nodoA *motor, char palabra1[20], char palabra2[20], int id);
- Función principal del tercer ejercicio, que utilizando la función buscarPalabraID, tendrá una doble condición de que si las 2 palabras pasadas por parámetro, existen en los documentos ingresados por el usuario.

int buscarPalabraID(nodoA *motor, char palabra[20], int id);
- Función que recorre el árbol, y retorna 1 si la palabra existe, y además si en su lista de ocurrencias también existe el id doc pasado por parámetro.

void buscarFrase(nodoA *motor, char frase[500]);
- Función del ejercicio 4, desde una frase ingresada por el usuario y luego separadas en palabras con la función int scanearFrase(char frase[500], char array[][500], se procede a cargar un Arreglo, con la función cargarArraydePOS(), que guarda todas las posiciones de la primera palabra si es que existe y se retorna la cantidad de las mismas (válidos). Luego se hace un for hasta válidos comparando las posiciones, con la siguiente +1, para ver si la siguiente posición es continua con la primera, verificando que el ID y la posición exista con la función buscarPosicionesTermino().
Esto se hace buscando en todos los documentos que la frase exista.

int buscarFrecuencias(nodoT *ocurrencias, int id);
- Función auxiliar del ejercicio 5, que guarda la frecuencia de las palabras de un determinado documento.

void maximaFrecuencia(nodoA *motor, int id, palabraAux *array, int *validos);
- Función del ejercicio 5, que va guardando todas las frecuencias, y con la ayuda de una estructura auxiliar, cargo cada palabra con su frecuencia en un arreglo.

void buscandoPalabraMaximaFreq(palabraAux array[], int validos1);
- Función principal del ejercicio 5, la cual con los validos de la función anterior, y recorriendo la estructura auxiliar es capaz de buscar cual es la palabra con la máxima frecuencia en un ID particular.

void distanciaLevenshtein(nodoA *motor, char palabra[20]);
- Función principal del ejercicio 6, que utilizando la distancia de *Levenshtein,* recorre el árbol, para buscar la distancia con todas las palabras en existencia con la condición de que la distancia sea <3, y luego sugerir similares.

int Levenshtein(char *s1, char *s2);
- Función que devuelve la distancia que existe entre una palabra y otra “número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra”.

Funciones Auxiliares (extras):

int mostrarPalabrasdeID(int id);
- Muestra todas las palabras de un documento en particular.

int cantidadPalabrasTotal();
- Para saber cuántas palabras hay en total en nuestro motor de búsqueda.

int cantidadDocumentos();
- Cantidad de documentos totales que existen en el directorio “/DIR.

int buscarFrecuenciasTodos(nodoT *ocurrencias);
void maximaFrecuenciaTodos(nodoA *motor, palabraAux *array, int *validos);
void buscandoPalabraMaximaFreqTodos(palabraAux array[], int validos1);
- 3 funciones auxiliares, para saber cual es la palabra que más se repite en todo nuestro motor.