INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Estructura de Datos

Práctica 3

Prof.: Franco Martínez Edgardo Adrián

Integrantes:

Mendoza Parra Sergio.

Salcedo Barrón Rubén Osmair.

Tejeda Martínez José Miguel.



MEXICO, D.F. a 6 de diciembre de 2015

# INTRODUCCIÓN

Con la implementación del TAD lista realizar la implementación de una tabla hash abierta, capaz de soportar el almacenamiento de palabras y sus definiciones (Diccionario de palabras). [1]

La función hash a usar deberá ser analizada, definida e implementada, teniendo en el programa dos opciones disponibles.

# DESCRIPCION DEL PROBLEMA

Como requisitos del programa, es necesario que se muestren las siguientes opciones:

1.- Cargar un archivo de definiciones: Realiza la carga en la tabla hash de un archivo que contiene una palabra y su definición en cada línea del archivo.

2.- Agregar una palabra y su definición: Permite agregar una palabra nueva al diccionario y su definición.

3.- Modificar una definición: Permite que se busque una palabra y modifique su definición.

4.- Eliminar una palabra: Permite buscar una palabra y eliminarla.

5.- Salir

Nota: Se deberá de realizar hashing abierto, determinando el tamaño de la tabla.

# DISEÑO Y FUNCIONAMIENTO DE LA SOLUCIÓN

Con base a la descripción del problema planteado, se implementaron las funciones necesarias para realizar las acciones que se necesitaban. Los modos de implementación que se utilizaron (dos tipos de implementación); una basada en listas por orden alfabético, y la segunda usando un hashing abierto a partir de una llave (key).

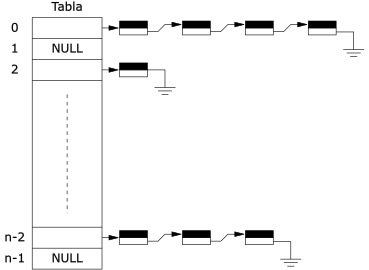
Para la primera opción, el almacenamiento en listas se realizó de tal manera que existiera una lista disponible por cada letra del alfabeto (A - Z), de tal modo que el hasheo se realizara con base al código ASCII de la primera letra de la palabra. Ejemplo:

|  |  |  |  |
| --- | --- | --- | --- |
| A 🡪 |  | Abrir | Almohada… |
| B 🡪 |  | Burro | Bondadoso… |
| C 🡪 |  | Campaña | Color… |
| D 🡪 |  | Diablo | Dormir… |
| E 🡪 |  | Estadio | Emoción… |
| … |  |  |  |
| Y 🡪 |  | Yema | Yuca… |
| Z 🡪 |  | Zapote | Zoológico… |

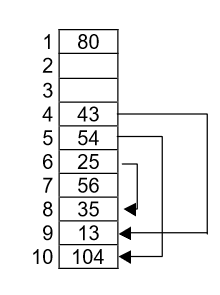
En el caso correspondiente al almacenamiento a partir de una llave, se considera un tamaño de tabla para almacenar los elementos. La llave obtenida mediante una función hash se limita a ubicarse dentro de esos valores y los elementos son insertados en su posición.

Cuando existen colisiones de almacenamiento pueden considerarse diferentes opciones:

1. Almacenamiento en paralelo, considerando a cada posición de la tabla principal como una sub-lista.



1. Realizar saltos hasta encontrar un espacio vacío donde poder almacenar el elemento.



En el programa, después de cargar el archivo correspondiente el usuario es capaz de elegir entre el método de almacenamiento alfabético o un almacenamiento por llave hash, donde en el último el método de solución es el correspondiente a realizar saltos en la tabla buscando espacios vacíos.

# IMPLEMENTACION DE LA SOLUCIÓN

Tomando en cuenta los métodos de almacenamiento en tablas hash, fueron desarrolladas dos opciones para el usuario:

1. Almacenamiento de los elementos de manera alfabética.

En esta opción, considerando las letras del alfabeto de la A hasta la Z, omitiendo la letra Ñ, se inicializó una lista para cada letra, comenzando desde el elemento 0 hasta el 25 (en total 26 letras a considerar).

Declaración del arreglo de listas en la función main(). El tipo de dato “lista” corresponde al struct declarado en la librería “TADListaDoblementeLigada.h”.

lista DiccionarioAlf[25]; //Arreglo de listas (0-25) a usar para almacenar los elementos (palabras).

Inicialización de las listas para dejarlas a disposición del programa.

for(i=0;i<26;i++) Initialize(&DiccionarioAlf[i]);

La forma de obtener el número ASCII correspondiente a las palabras se realiza por medio de “casting”, almacenando en un entero el valor correspondiente del mismo tipo proveniente de un carácter, el cual de forma general se describe de la siguiente manera:

ascii=(int)dicc[i].palabra[0];

ascii=ascii-65;

if(ascii<0||ascii>25)

ascii=25; //Si el codigo ASCII no se encuentra entre A & Z se envía por default a la ultima lista

Add(&D[ascii],dicc[i]); //Agrega el elemento en la lista correspondiente

Como podemos observar, por cada palabra se obtiene el numero ASCII correspondiente a su primera letra, después se resta un valor de 65 para corresponder a las listas inicializadas de 0 a 25; finalmente, cuando la primer letra de un elemento no se encuentra entre los valores ASCII considerados, se asocia el valor ASCII correspondiente a la última lista (25). Al tener listos los elementos son agregados en su lista asociada.

Las funciones que hacen este manejo son las siguientes:

int CargarDiccionarioAlf(lista \*D,elemento \*dicc,unsigned int n);

int eliminar\_palabra(lista \*L1,lista \*L2);

int AgregarPalabra(lista \*L1,lista \*L2);

int ConsultarDefinicion(lista \*L1,lista \*L2);

int ModificarDefinicion(lista \*L1,lista \*L2);

1. Almacenamiento de los elementos mediante llave hash.

En esta opción, únicamente fue inicializada una lista con un tamaño fijo (llamado tamaño de tabla), el cual corresponde a una lista con 4 veces la cantidad de palabras que son leídas por archivo.

Si la cantidad de palabras leídas fuese de 300, la función de tamaño de lista correspondería a 1200:

unsigned int tamanio\_tabla;

tamanio\_tabla=Size(&D);

printf(“%d”,tamanio\_tabla);

, de tal manera que al ejecutar el programa, printf nos mostraría 1200.

Utilizar este tamaño de tabla fue determinado debido a la función hash implementada, ya que al generar una llave puede ser igual a la de otra palabra. La solución propuesta en nuestro caso es: cuando sucede una colisión sobre la lista se salta hacia el siguiente elemento, de modo que el elemento a almacenar proceda en un espacio de lista vacío.

Los procedimientos implementados comienzan con la inicialización de la lista o tabla hash desde la función main().

lista DiccionarioHash;

Initialize(&DiccionarioHash);

Las llaves hash para cada palabra se encuentran determinadas por la siguiente función:

unsigned int hashing(char palabra[],int n)

{

char car;

unsigned int ascii[50],key;

unsigned int i,acu=0,pal;

pal=strlen(palabra);

for(i=0;i<pal;i++)

{

car=palabra[i];

ascii[i]=(int)car;

acu=acu+ascii[i];

}

acu=acu\*pal;

key=acu%(n\*4);

return key;

}

La función hashing devuelve el la llave correspondiente a los valores de una palabra, es decir, se capturan los valores ASCII para cada letra de la palabra, los cuales son almacenados en un acumulados; posteriormente la suma acumulada es multiplicada por lo largo de la palabra, dando como resultado de salida el valor acumulado en el módulo de 4 veces “n”, donde “n” es la cantidad de palabras leídas del archivo previamente por la función leeArchivo().

Las funciones que se basan en los valores de llave de palabra son las siguientes:

int CargarDiccionarioHash(lista \*D,elemento \*dicc,unsigned int n);

int eliminar\_palabra(lista \*L1,lista \*L2);

int AgregarPalabra(lista \*L1,lista \*L2);

int ConsultarDefinicion(lista \*L1,lista \*L2);

int MidificarDefinicion(lista \*L1,lista \*L2);

Los saltos que se registran en esta función son guardados en el contador global de carga o consulta, el cual cuenta los saltos realizados cuando se detecta una colisión. El método de detección de los saltos se basa en los saltos después de recorrer la lista hasta el valor de la llave correspondiente y no corresponde con la palabra sobre la cual se va a aplicar la operación:

p=First(L1);

e1=Position(L1,p);

count\_jump\_charg++;

while(strcmp(e1.palabra,"0")!=0){//Inicia la busqueda p=Following(L1,p);

e1=Position(L1,p);

count\_jump\_charg++;

}

Nótese que la condición de la instrucción while depende de lo que se trabaje, es decir, cuando se desea buscar un espacio vacío se utiliza el operador ==, en caso contrario usamos una diferencia ¡= sobre el parámetro del elemento de lista que deseemos trabajar.

Como aspectos generales, el código presenta distintas particularidades:

1. Se declaró un tipo de estructura llamada “línea”, la cual será usada para almacenar las líneas que se leen mediante la función leeArchivo().

//struct linea

typedef struct linea

{

char lineaArchivo[300];//Arreglo donde se guarda cada una de las lineas que se leen del archivo

}linea;

1. El máximo de palabras a leer por archivo fue delimitado a 3000.

#define MAX\_Pal 3000

1. Se usaron variables globales para los contadores, las opciones de hasheo y la cantidad de palabras leídas.

//Variables globales (contadores)

unsigned int opcHash=3,n,count\_jump\_charg,count\_jump\_query;

1. El método de separación de las palabras se determinó mediante el uso de la función strok(), contenida en la librería string.h.

for(i=0;i<n;i++)

{

aux\_palabra=strtok(e[i].lineaArchivo,":"); //Funcion que separa las palabras de la definicion.

strcpy(dicc[i].palabra,aux\_palabra); //Copiar la palabra en en elemento de lista

while( (aux\_palabra = strtok( NULL,":")) != NULL ) // Posteriores llamadas para hasta que el apuntador ya no pueda apuntar a mas cadenas

strcpy(dicc[i].definicion,aux\_palabra); //Copiar la definicion en el elemento de lista

}

1. Se implementó una función que permite guardar los cambios realizados en un archivo de texto plano llamado “Diccionario”, el cual si se cargó información al programa, se guardará en la carpeta de ejecución del mismo.

int GuardarArchivo(lista \*L1,lista \*L2)

{

FILE \*archivo;

posicion p;

elemento e;

unsigned int i,j;

if(opcHash==1) //Para almacenamiento alfabetico

{

archivo=fopen("Diccionario.txt","w"); //Archivo de salida

for(i=0;i<26;i++) //Busqueda por lista correspondiente a las letras de

A a Z

{

p=First(&L2[i]);

for(j=0;j<Size(&L2[i]);j++){ //Busqueda por elemento de cada

lista

e=Position(&L2[i],p);

if(strcmp(e.palabra,"0")) //Validacion de elemento vacio

fprintf(archivo,"%s:%s",e.palabra,e.definicion);

//Almacenamiento de palabra en el archivo

p=Following(&L2[i],p);

}

}

fclose(archivo);

}

else if(opcHash==2)

{

archivo=fopen("Diccionario.txt","w");

p=First(L1);

for(i=0;i<Size(L1);i++)

{

e=Position(L1,p);

if(strcmp(e.palabra,"0"))//Validacion de elemento vacio

fprintf(archivo,"%s:%s",e.palabra,e.definicion);

//Almacenamiento de palabra en el archivo

p=Following(L1,p);

}

fclose(archivo);

}

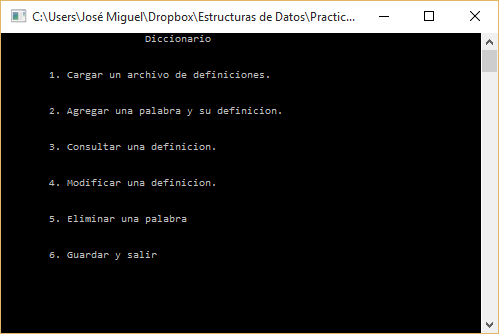
else; //No se guardan cambios en caso de no haber sido usado un metodo.

return 0;

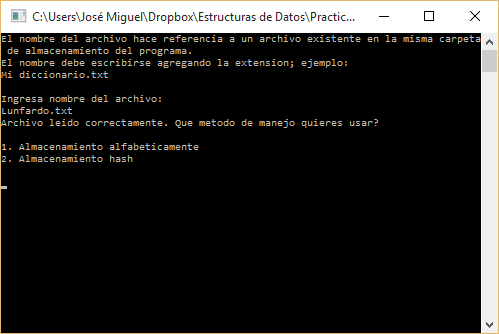
}

# FUNCIONAMIENTO

1. Pantalla de inicio

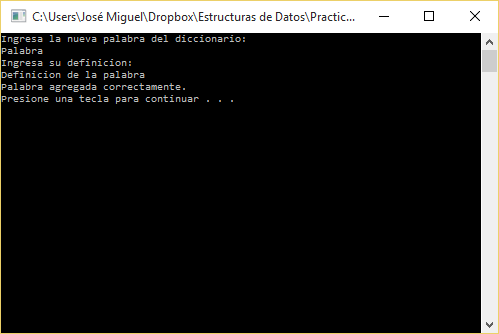


1. Carga del archivo



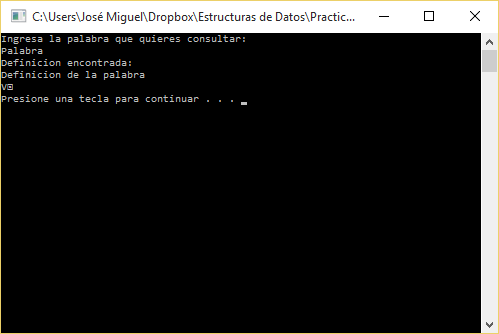
Como esta sección lo especifica, es necesario agregar la extensión del archivo al momento de cargarlo a la memoria de ejecución del programa. Después de esto, se muestran las dos opciones disponibles para el manejo del diccionario. El usuario puede seleccionar la que más le guste y así observar las diferencias en tiempo y saltos de búsqueda.

1. Agregar palabra

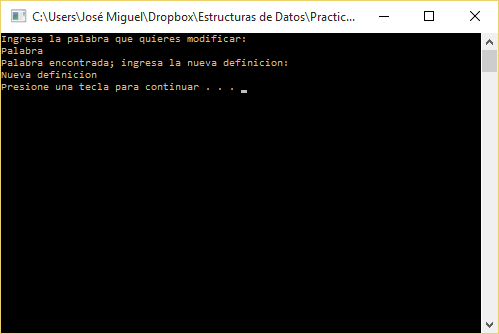


La función de agregar verifica la opción de manejo seleccionada, siendo así como hace uso de las listas correspondientes.

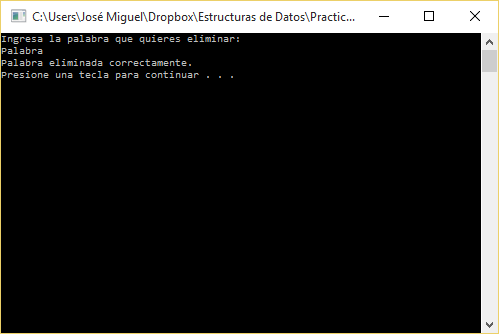
1. Consultar una definición



1. Modificar definición

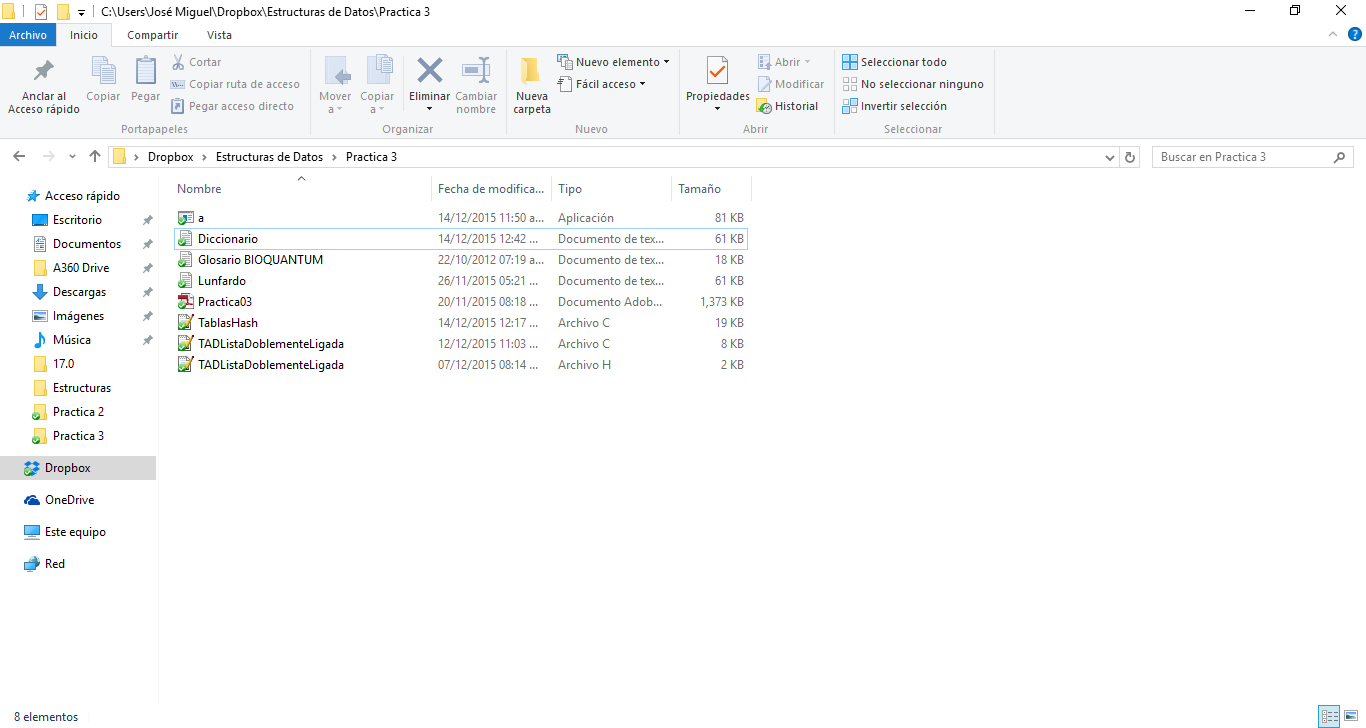


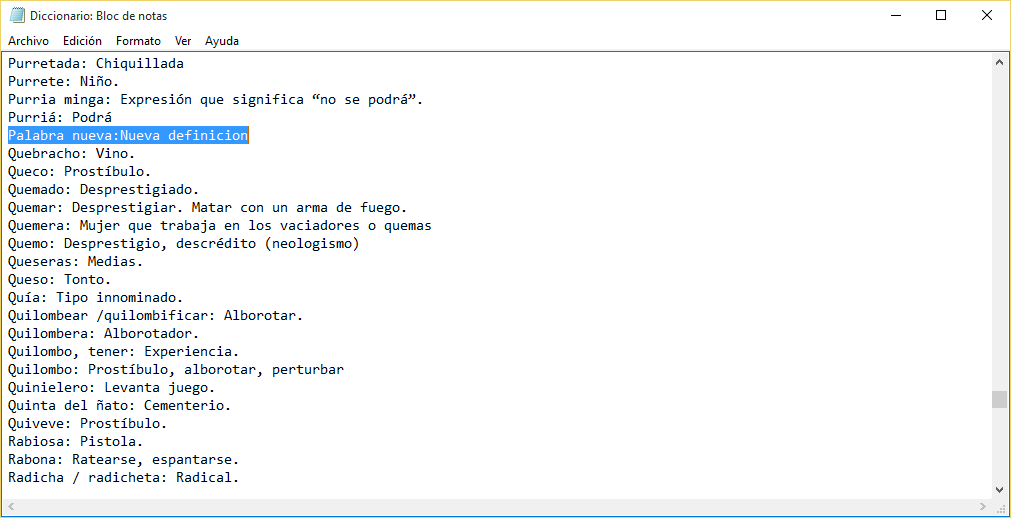
1. Eliminar palabra



1. Guardar y salir

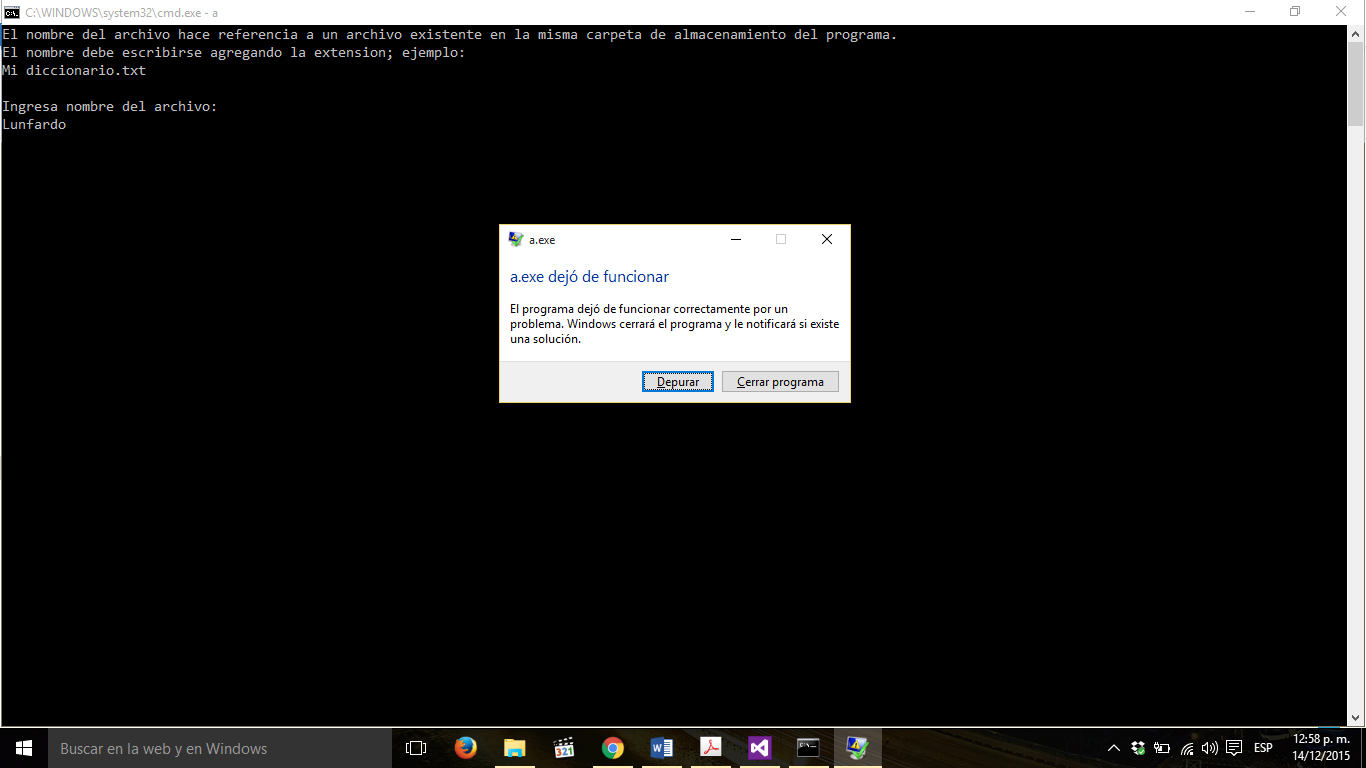
Al seleccionar la opción para salir del programa, si hay algún archivo cargado en la memoria de ejecución, se guardan los cambios realizados, es decir, se almacenan las palabras agregadas o el diccionario sin las palabras eliminadas.



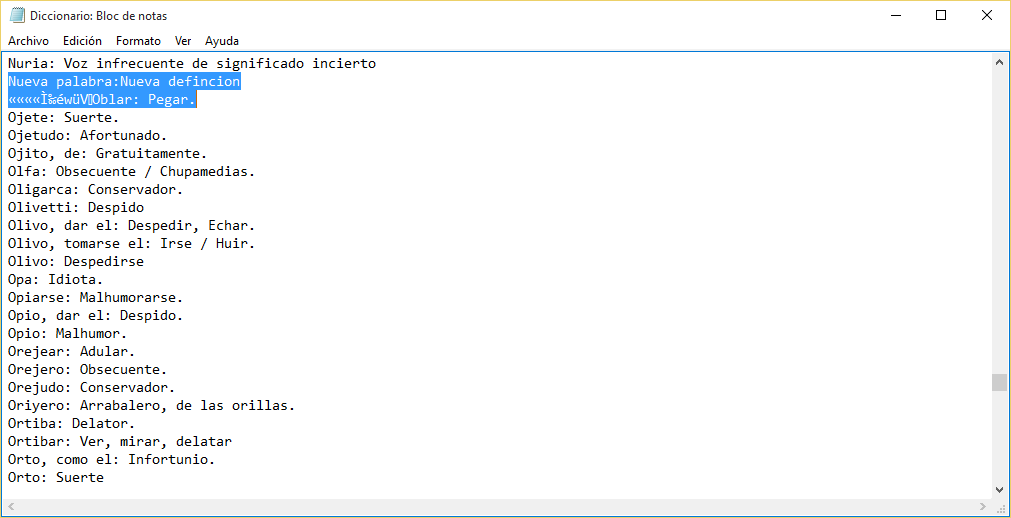


# ERRORES DETECTADOS

1. En la función de leer archivo, en caso de no abrir el archivo correctamente el programa deja de responder. Esto sucede cuando se ingresa incorrectamente el nombre del archivo o no se le añade su extensión.



1. Cuando se agregan palabras en ocasiones al comienzo de la que sigue aparecen datos basura, probablemente por la información almacenada en la entrada estándar.



1. En la implementación del TAD Lisa Doblemente Ligada la función elemento Element(lista \*l, unsigned long n); “n” se toma como elemento consecutivo a como se fue agregando cada palabra al diccionario, de tal forma que aunque la función hashing devuelva un valor de, por ejemplo “739”, si se busca la posición del elemento “739” pasándolo como parámetro en “n”, la posición corresponde al número consecutivo de cómo fue agregada la palabra.

# POSIBLES MEJORAS

La mejora propuesta para la implementación del TAD Lista Doblemente Ligada esta en agregar un contador, clave o id en struct elemento, de tal modo que la función Elemento, en su parámetro “n” haga referencia a esa parte del elemento; esto garantiza que este identificador sea inamovible de acuerdo al valor que se le asigne, y no sea generado de acuerdo a como se van añadiendo los elementos a la lista.

# CONCLUSIONES

# Salcedo Barrón Ruben: En esta práctica pudimos implementar varios conceptos de la materia como archivos listas, colas y pilas para poder hacer la simulación de un diccionario el cual podía hacer varias cosas pero la más importante fue la de búsqueda la cual tuvo que ser casi perfecta para que fuera rápido y efectivo el cual lo pudimos lograr con el hasheo que implementamos y en todas las búsquedas lo máximo que se tardaba era un salto por lo cual fue la mejor opción para lograr esta práctica. En lo personal no se me hizo difícil esta práctica solo era dedicarle tiempo.

Mendoza Parra Sergio: En ésta práctica se pudo realizar la parte del hashing en un archivo lo cual como se vio en el programa se pudo realizar en la parte de tipo alfanumérico que era dependiendo de la primera letra de las palabras y el de tipo hash en el cual con una fórmula que obtuvimos fue como las posiciones en las que estaba cada palabra de tal manera que si se agregaba o eliminaba alguna palabra o definición se recorrían las palabras de tal manera que no hubiera espacios en blanco. En mi punto de vista fue una práctica algo laboriosa ya que tuvimos que implementar varias formulas de tal manera que encontramos una fórmula que se apegó a lo que se necesitaba y ya al final solo se creó un fichero tipo .txt en el cual se agregaban todos los cambios que se hicieron.

# ANEXO

Código fuente

/\*

DICCIONARIO HASH

AUTORES: Mendoza Parra Sergio, Salcedo Barrón Rubén Osmair, Tejeda Martínez José Miguel (C) Noviembre 2015

VERSIÓN: 1.0

DESCRIPCIÓN: Con la implementación del TAD lista

realizar la implementación de una tabla

hash abierta, capaz de soportar el

almacenamiento de palabras y sus definiciones

(Diccionario de palabras).

OBSERVACIONES: El programa permite poder agregar nuevas

palabras, consultar el significado de alguna,

modificar el significado y eliminar palabras

del diccionario hash.

EL CODIGO QUE IMPLEMENTA LAS FUNCIONES DE LIBRERIA ES EL ARCHIVO: TADListaDoblementeLigada.c

COMPILACION: gcc TablasHash.c TADListaDoblementeLigada.c -o "Nombre del ejecutable"

EJECUCION: ./"Nombre del ejecutable" (Linux)

"Nombre del ejecutable".exe (Windows)

\*/

//Librerias

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include "TADListaDoblementeLigada.h" //Libreria que contiene las funciones definidas en TADListaDoblementeLigada.c

//Definiciones

#define MAX\_Pal 3000

#define PAUSE system("pause") //Cambiar por getchar() en Linux

#define CLEAN system("cls") //Cambiar por system("clear") en Linux

//struct linea

typedef struct linea

{

char lineaArchivo[300];//Arreglo donde se guarda cada una de las lineas que se leen del archivo

}linea;

//Declaraciones de funciones

//Funciones

unsigned int leeArchivo(linea \*e,char ruta[]);

int eliminar\_palabra(lista \*L1,lista \*L2);

int CargarArchivo(linea \*e,elemento \*dicc,lista \*L1,lista \*L2);

int CargarDiccionarioAlf(lista \*D,elemento \*dicc,unsigned int n);

int CargarDiccionarioHash(lista \*D,elemento \*dicc,unsigned int n);

unsigned int hashing(char palabra[],int n);

int AgregarPalabra(lista \*L1,lista \*L2);

int ConsultarDefinicion(lista \*L1,lista \*L2);

int MidificarDefinicion(lista \*L1,lista \*L2);

int GuardarArchivo(lista \*L1,lista \*L2);

//Variables globales (contadores)

unsigned int opcHash=3,n,count\_jump\_charg,count\_jump\_query,add\_pal;

int main(void)

{

elemento diccionario[MAX\_Pal]; //Diccionario en tiempo de ejecucion

linea palabras[MAX\_Pal]; //Arreglo para guardar lineas del archivo de entrada

lista DiccionarioHash,DiccionarioAlf[26]; //Listas a usar para almacenar los elementos (palabras)

int i,x;

for(i=0;i<26;i++) Initialize(&DiccionarioAlf[i]);

Initialize(&DiccionarioHash);

do{

fflush(stdin);

puts("\t\t\tDiccionario");

puts("\n\n\t1. Cargar un archivo de definiciones.");

puts("\n\n\t2. Agregar una palabra y su definicion.");

puts("\n\n\t3. Consultar una definicion.");

puts("\n\n\t4. Modificar una definicion.");

puts("\n\n\t5. Eliminar una palabra");

puts("\n\n\t6. Guardar y salir");

scanf("%d",&x);

CLEAN;

switch(x){

case 1:

CargarArchivo(palabras,diccionario,DiccionarioAlf,&DiccionarioHash);

break;

case 2:

AgregarPalabra(&DiccionarioHash,DiccionarioAlf);

break;

case 3:

ConsultarDefinicion(&DiccionarioHash,DiccionarioAlf);

break;

case 4:

ModificarDefinicion(&DiccionarioHash,DiccionarioAlf);

break;

case 5: eliminar\_palabra(&DiccionarioHash,DiccionarioAlf);

break;

case 6:

GuardarArchivo(&DiccionarioHash,DiccionarioAlf); //Guarda cambios mandando los resultados en el archivo "Diccionario.txt"

for(i=0;i<26;i++) Destroy(&DiccionarioAlf[i]);

Destroy(&DiccionarioHash);

PAUSE;

exit(0);

break;

}

}while(x!=6);

return 0;

}

/\*

unsigned int leeArchivo(linea \*e,char ruta[]);

Descripcion: Abre un archivo (diccionario de palabras) y lo carga en la memoria de ejecucion del programa.

Recibe: Arreglo para guardar la linea de datos del archivo (linea \*e), ruta del archivo a consultar (char ruta[]).

Devuelve: 0 si no se pudo abrir el archivo, en caso contrario devuelve la cantidad de palabras leidas.

Observaciones: La ruta del archivo debe escribirse junto con la extension correspondiente.

\*/

unsigned int leeArchivo(linea \*e,char ruta[]){

unsigned int i=0;

FILE \*archivo;

char linea[300]; //auxiliar que obtiene lo que se va leyendo linea por linea del archivo

archivo = fopen(ruta, "r");

if(fopen == NULL) return 0; //Si no se pudo abrir el archivo

else if(fopen != NULL){

while(!feof(archivo))//Lectura hasta llegar al final del archivo

{

fflush(stdin);

fgets(linea, 300, archivo); //Lecuta de linea

strcpy(e[i].lineaArchivo,linea); //Almacenamiento de linea

i++;

}

fclose(archivo);

}

return i;

}

/\*

unsigned int hashing(char palabra[],int n);

Descripcion: Funcion que permite obtener una llave a partir de una cadena.

Recibe: Palabra para obtener codigo hash(char palabra[]), tamaño de la tabla o elementos totales como parámetro secundario(int n).

Devuelve: Codigo hash (unsigned int key).

Observaciones: La llave depende de la cantidad de elementos totales!!!

\*/

unsigned int hashing(char palabra[],int n)

{

char car;

unsigned int ascii[50],key;

unsigned int i,acu=0,pal;

pal=strlen(palabra);

for(i=0;i<pal;i++)

{

car=palabra[i];

ascii[i]=(int)car;

acu=acu+ascii[i];

}

acu=acu\*pal;

key=acu%(n\*4);

return key;

}

/\*

int CargarDiccionarioAlf(lista \*D,elemento \*dicc,unsigned int n);

Descripcion: Funcion que almacena en memoria del programa un diccionario en forma alfabética.

Recibe: Diccionario de palabras en forma de elementos(elemento \*dicc),

tamaño de la tabla o cantidad total de palabras(int n).

apuntador al arreglo de listas donde se almacenarán elementos(lista \*D).

Devuelve:

Observaciones: Las listas deben haber sido previamente inicializadas.

\*/

int CargarDiccionarioAlf(lista \*D,elemento \*dicc,unsigned int n)

{

unsigned int i;

int ascii; //Numero ASCII de la primera letra de la palabra a almacenar

for(i=0;i<n;i++)

{

ascii=(int)dicc[i].palabra[0];

ascii=ascii-65;

if(ascii<0||ascii>25)

ascii=25; //Si el codigo ASCII no se encuentra entre A & Z se envía por default a la ultima lista

Add(&D[ascii],dicc[i]); //Agrega el elemento en la lista correspondiente

}

return 0;

}

/\*

int CargarDiccionarioHash(lista \*D,elemento \*dicc,unsigned int n);

Descripcion: Funcion que almacena elementos en una lista de hashing abierto.

Recibe: Lista donde se almacenarán los elementos(lista \*D),

arreglo de elementos que contienen palabra y definicion(elemento \*dicc),

tamaño de tabla(unsigned int n).

Devuelve: Promedio de saltos por cada elemento agregado.

Observaciones: La lista (\*D) debe haber sido inicializada.

\*/

int CargarDiccionarioHash(lista \*D,elemento \*dicc,unsigned int n)

{

unsigned int keys[MAX\_Pal],i; //keys[] - arreglo de llaves correspondientes a cada elemento.

elemento e1,e;

posicion p;

strcpy(e1.palabra,"0"); //

strcpy(e1.definicion,"0"); //Elementos vacios.

count\_jump\_charg=0; //Contador de saltos

for(i=0;i<(n\*4);i++) Add(D,e1); //Inicializa la lista con espacios vacios (la lista se inicializa con un tamaño 4 veces mayor al total de elementos)

p=First(D);

for(i=0;i<n;i++)

{

keys[i]=hashing(dicc[i].palabra,n); //Obtencion de la clave de cada palabra

e=Position(D,p);

dicc[i].key=keys[i]; //Almacenamiento de la clave en el elemento

if(!strcmp(e.palabra,"0")) //Si esta vacia la posicion

{

Replace(D,p,dicc[i]); //Almacenamiento de elemento

count\_jump\_charg++;

p=Following(D,p);

}

else if(strcmp(e.palabra,"0")) //Si no es vacía la posicion (colapso)

{

while(strcmp(e.palabra,"0")){ //Buscar espacio vacio

p=Following(D,p);

e=Position(D,p);

count\_jump\_charg++;

}

Replace(D,p,dicc[i]);

count\_jump\_charg++;

p=Following(D,p);

}

}

printf("Promedio de saltos al cargar: %d saltos por palabra\n",count\_jump\_charg/n);

return 0;

}

/\*

int CargarArchivo(linea \*e ,elemento \*dicc,lista \*L1,lista \*L2);

Descripcion: Funcion que carga un archivo a memoria de programa, obteniendo la lectura y la separacion de palabras almacenandolas en elementos de lista

Recibe: Arreglo de lineas de archivo(linea \*e),

arreglo de elementos donde se almacenarán las palabras separadas por nombre y definicion,

listas correspondientes a la forma de almacenamiento: L1 - Hash, L2 - Alfabetico.

Devuelve:

Observaciones: Las listas a usar deben haber sido previamente inicializadas.

\*/

int CargarArchivo(linea \*e ,elemento \*dicc,lista \*L1,lista \*L2)

{

unsigned int i;

char \*aux\_palabra,ruta[100];

puts("El nombre del archivo hace referencia a un archivo existente en la misma carpeta de almacenamiento del programa.\nEl nombre debe escribirse agregando la extension; ejemplo:");

puts("Mi diccionario.txt\n");

puts("Ingresa nombre del archivo:");

fflush(stdin);

gets(ruta);

n=0;

n=leeArchivo(e,ruta); //n=cantidad de palabras leidas.

if(n!=0){ //Si se leyeron palabras...

for(i=0;i<n;i++)

{

aux\_palabra=strtok(e[i].lineaArchivo,":"); //Funcion que separa las palabras de la definicion.

strcpy(dicc[i].palabra,aux\_palabra); //Copiar la palabra en en elemento de lista

while( (aux\_palabra = strtok( NULL,":")) != NULL ) // Posteriores llamadas para hasta que el apuntador ya no pueda apuntar a mas cadenas

strcpy(dicc[i].definicion,aux\_palabra); //Copiar la definicion en el elemento de lista

}

puts("Archivo leido correctamente. Que metodo de manejo quieres usar?\n");

puts("1. Almacenamiento alfabeticamente\n2. Almacenamiento hash\n");

scanf("%d",&opcHash); //Seleccion del metodo a usar

if(opcHash==1) CargarDiccionarioAlf(L1,dicc,n);

else if(opcHash==2) CargarDiccionarioHash(L2,dicc,n);

}

else //Si no se pudo leer el archivo

puts("Error al leer archivo\n");

PAUSE;

CLEAN;

return 0;

}

/\*

int eliminar\_palabra(lista \*L1,lista \*L2);

Descripcion: Elimina una palabra de la memoria de ejecucion del programa.

Recibe: Listas hash y alfabeticas (L1 y L2).

Devuelve:

Observaciones: Si no ha sido seleccionado un método de almacenamiento, es mostrado un mensaje de error.

\*/

int eliminar\_palabra(lista \*L1,lista \*L2)

{

char palabra[50];

elemento e,aux;

posicion p;

unsigned int i;

int ascii;

strcpy(aux.palabra,"0");//

strcpy(aux.definicion,"0");//Elemento vacio

puts("Ingresa la palabra que quieres eliminar: ");

fflush(stdin);

gets(palabra);

if(opcHash==1) //Si el metodo usado es alfabetico

{

ascii=(int)palabra[0];

ascii=ascii-65; //Codigo ASCII de la primera letra de la palabra.

if(ascii<0||ascii>25) puts("Palabra no encontrada.");

else

{

p=First(&L2[ascii]); //La busqueda comienza desde el inicio de la lista correspondiente

for(i=0;i<Size(&L2[ascii]);i++)

{

e=Position(&L2[ascii],p);

if(strcmp(e.palabra,palabra)==0) //Si se encuentra la paabra

{

Replace(&L2[ascii],p,aux); //Se reemplaza el elemento de la lista por uno vacío.

i=i+MAX\_Pal;

}

p=Following(&L2[ascii],p);

}

puts("Palabra eliminada correctamente.");//Si se llega al final de la lista y no se encuentra el elemento.

}

}

else if(opcHash==2){ //Si el metodo usado es hash

p=First(L1);

e=Position(L1,p);

while(p!=Final(L1)) //Inicia la busqueda hasta el final de la lista hash

{

if(strcmp(e.palabra,palabra)==0) //si se encuentra la palabra...

{

Replace(L1,p,aux); //Se reemplaza el elemento de lista con uno vacío.

puts("Palabra eliminada correctamente\n");

PAUSE;

CLEAN;

return 0;

}

else

{

p=Following(L1,p);

e=Position(L1,p);

}

}

puts("No se encontro la palabra en la lista"); //Si se llega al final de la lista

}

else

puts("No se han capturado datos\n");

PAUSE;

CLEAN;

return 0;

}

/\*

int AgregarPalabra(lista \*L1,lista \*L2);

Descripcion: Agrega una palabra a la lista correspondiente.

Recibe: Listas hash y alfabeticas (L1 y L2).

Devuelve:

Observaciones: Si no ha sido seleccionado un método de almacenamiento, es mostrado un mensaje de error.

\*/

int AgregarPalabra(lista \*L1,lista \*L2)

{

unsigned int key;

int ascii;

char palabra[50],definicion[250];

elemento e1,e2;

posicion p;

strcpy(e1.palabra,"0");//

strcpy(e1.definicion,"0");//Elemento vacio

puts("Ingresa la nueva palabra del diccionario:");

fflush(stdin);

gets(palabra);

puts("Ingresa su definicion:");

fflush(stdin);

gets(definicion);

definicion[strlen(definicion)]='\n'; //Salto de linea al final de la definicion.

strcpy(e2.palabra,palabra);//

strcpy(e2.definicion,definicion);//Copia de la palabra con definicion a un elemento de lista.

key=hashing(palabra,n); //Obtencion de la llave de palabra.

e2.key=key; //Copia de la llave al elemento de lista.

count\_jump\_charg=0;

if(opcHash==1)

{

ascii=(int)palabra[0];

ascii=ascii-65; //Codigo ASCII de la primera letra de la palabra

if(ascii<0||ascii>25) ascii=25;

Add(&L2[ascii],e2); //Agregar a la lista correspondiente

puts("Palabra agregada correctamente.");

}

else if(opcHash==2)

{

p=First(L1);

e1=Position(L1,p);

count\_jump\_charg++;

while(strcmp(e1.palabra,"0")!=0){//Inicia la busqueda en el inicio de lista

p=Following(L1,p);

e1=Position(L1,p);

count\_jump\_charg++;

}

Replace(L1,p,e2); //Almacenamiento de la palabra en un lugar vacio

printf("Palabra agregada correctamente en %d saltos.\n",count\_jump\_charg);

}

else

puts("No se pudo agregar la palabra.\nNo hay diccionario de referencia.");

PAUSE;

CLEAN;

return 0;

}

/\*

int ConsultarDefinicion(lista \*L1,lista \*L2);

Descripcion: Consulta la definicion de una palabra.

Recibe: Listas hash y alfabeticas (L1 y L2).

Devuelve:

Observaciones: Si no ha sido seleccionado un método de almacenamiento, es mostrado un mensaje de error.

\*/

int ConsultarDefinicion(lista \*L1,lista \*L2)

{

unsigned int a=0,i,key;

char palabra[50];

int ascii;

posicion p;

elemento e;

strcpy(e.palabra,"0");//

strcpy(e.definicion,"0");//Elemento vacio

puts("Ingresa la palabra que quieres consultar: ");

fflush(stdin);

gets(palabra);

count\_jump\_query=0;

if(opcHash==1)//Para almacenamiento alfabetico

{

ascii=(int)palabra[0];

ascii=ascii-65; //ASCII de la primera letra de la palabra

if(ascii<0||ascii>25) ascii=25;

p=First(&L2[ascii]);

count\_jump\_query++;

for(i=0;i<Size(&L2[ascii]);i++,count\_jump\_query++)//Inicia la nusqueda en la lista correspondiente

{

e=Position(&L2[ascii],p);

if(!strcmp(e.palabra,palabra)){//Si coincide la palabra

printf("Definicion encontrada:\n%s\n",e.definicion);

PAUSE;

CLEAN;

return 0;

}

p=Following(&L2[ascii],p);//Si no coincide pasa al elemento siguiente.

}

printf("Definicion no encontrada en %d saltos.",count\_jump\_query);

}

else if(opcHash==2){//Para almacenamiento hash

key=hashing(palabra,n);//Llave hash de la palabra

p=First(L1);

for(i=0;i<Size(L1);i++) //Inicio de la busqueda desde el comienzo

{

e=Position(L1,p);

if(e.key==key){//Si coinside la clave

if(!strcmp(e.palabra,palabra))//Si se encuentra la palabra

{

count\_jump\_query++;

i=Size(L1);

}

else //Colision

{

p=Following(L1,p);

count\_jump\_query++;

}

}

else

p=Following(L1,p);

}

if(!strcmp(palabra,e.palabra))//Si se encontro la palabra

{

printf("Definicion encontrada.\n");

printf("%s\n",e.definicion);

printf("\nSaltos de busqueda: %d\n",count\_jump\_query);

}

else //Si no fue encontrada

{

puts("Palabra no encontrada.");

}

}

else

puts("No se han capturado datos.\n");

PAUSE;

CLEAN;

return 0;

}

/\*

int ModificarDefinicion(lista \*L1,lista \*L2);

Descripcion: Modifica la definicion de una palabra.

Recibe: Listas hash y alfabeticas (L1 y L2).

Devuelve:

Observaciones: Si no ha sido seleccionado un método de almacenamiento, es mostrado un mensaje de error.

\*/

int ModificarDefinicion(lista \*L1,lista \*L2)

{

unsigned int a=0,i,key;

char palabra[50],definicion[250];

int ascii;

posicion p;

elemento e;

puts("Ingresa la palabra que quieres modificar: ");

fflush(stdin);

gets(palabra);

count\_jump\_query=0;

if(opcHash==1)

{

ascii=(int)palabra[0];

ascii=ascii-65; //ASCII de la primera letra de la palabra

if(ascii<0||ascii>25) ascii=25;

p=First(&L2[ascii]);

for(i=0;i<Size(&L2[ascii]);i++) //Inicia la busqueda desde el inico de la lista correspondiente

{

e=Position(&L2[ascii],p);

if(!strcmp(e.palabra,palabra)){ //Si es encontrada la palabra

printf("Palabra encontrada; ingresa la nueva definicion:\n");

fflush(stdin);

gets(definicion);

definicion[strlen(definicion)]='\n';

strcpy(e.definicion,definicion); //Se copia la lueva definicion

Replace(&L2[ascii],p,e); //Actualiza el elemento de lista

PAUSE;

CLEAN;

return 0;

}

p=Following(&L2[ascii],p);

}

puts("Definicion no encontrada. No se pudo modificar");

}

else if(opcHash==2){

p=First(L1);

key=hashing(palabra,n); //Llave hash de la palabra

for(i=0;i<Size(L1);i++)

{

e=Position(L1,p);

if(e.key==key){ //Si coincide la llave

if(!strcmp(e.palabra,palabra)) //Si es la palabra correspondiente

{

count\_jump\_query++;

i=Size(L1); //Salida del ciclo

}

else //Colision!

{

p=Following(L1,p);

count\_jump\_query++;

}

}

else

p=Following(L1,p);

}

if(!strcmp(palabra,e.palabra))//Si se encontro la palabra

{

printf("Palabra encontrada.\nIngresa la nueva definicion.\n");

fflush(stdin);

gets(definicion);

definicion[strlen(definicion)]='\n'; //Salto de linea

strcpy(e.definicion,definicion);//Ingreso de la nueva definicion

Replace(L1,p,e); //Actualizacion edl elemento de lista

puts("Definicion modificada correctamente");

printf("\nSaltos de busqueda: %d\n",count\_jump\_query);

}

else

{

puts("Palabra no encontrada.");

}

}

else

puts("No se han capturado datos.\n");

PAUSE;

CLEAN;

return 0;

}

/\*

int GuardarArchivo(lista \*L1,lista \*L2);

Descripcion:

Recibe:

Devuelve:

Observaciones:

\*/

int GuardarArchivo(lista \*L1,lista \*L2)

{

FILE \*archivo;

posicion p;

elemento e;

unsigned int i,j;

if(opcHash==1) //Para almacenamiento alfabetico

{

archivo=fopen("Diccionario.txt","w"); //Archivo de salida

for(i=0;i<26;i++)//Busqueda por lista correspondiente a las letras de A a Z

{

p=First(&L2[i]);

for(j=0;j<Size(&L2[i]);j++){ //Busqueda por elemento de cada lista

e=Position(&L2[i],p);

if(strcmp(e.palabra,"0")) //Validacion de elemento vacio

fprintf(archivo,"%s:%s",e.palabra,e.definicion); //Almacenamiento de palabra en el archivo

p=Following(&L2[i],p);

}

}

fclose(archivo);

}

else if(opcHash==2)

{

archivo=fopen("Diccionario.txt","w");

p=First(L1);

for(i=0;i<Size(L1);i++)

{

e=Position(L1,p);

if(strcmp(e.palabra,"0"))//Validacion de elemento vacio

fprintf(archivo,"%s:%s",e.palabra,e.definicion); //Almacenamiento de palabra en el archivo

p=Following(L1,p);

}

fclose(archivo);

}

else; //No se guardan cambios en caso de no haber sido usado un metodo.

return 0;

}

# BIBLIOGRAFÍA

[1] Edgardo Adrián Franco Martínez, Practica 03: "Diccionario con hashing abierto", Noviembre de 2015. [En línea]. Disponible en: <http://eafranco.com/docencia/estructurasdedatos/files/practicas/Practica03.pdf>