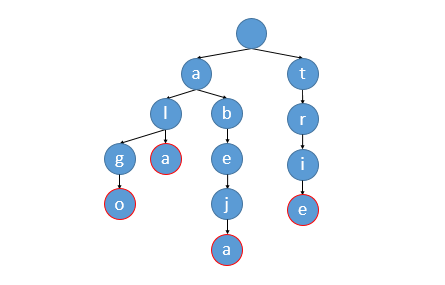
Integrante:

Stalin Franco 27.922.928

Un árbol es una estructura auto-referenciada, por lo que lógicamente necesita de tener en su estructura uno o mas apuntadores a si mismo, en el caso de tener uno seria solo una lista enlazada, con dos en cambio se puede hacer un árbol, en el que cada uno de los apuntadores son dos hijos del nodo actual. Ahora, teniendo en cuenta estas características el árbol trie puede tener una cantidad grande de apuntadores, donde cada uno de ellos representa una parte de la clave inicial.

Los arboles trie en el caso de nuestro diccionario puede tener dos implementaciones principales, una con listas enlazadas con apuntadores y otra con arreglos, cada una con sus ventajas particulares. Este es un esquema de un arbol trie.



**Planteamiento del problema:**

Se desea implementar un diccionario de palabras en ingles haciendo uso de árboles trie, los trie son arboles de búsqueda, donde la clave de búsqueda es seccionada en claves mas pequeñas, de forma que estas sean usadas para armar la estructura del árbol de búsqueda.

Este método es particularmente conveniente con un diccionario, ya que las palabras se pueden seccionar en letras y cada una de ellas se puede usar de indice para crear una estructura.

El diccionario se debe construir a partir un archivo con extensión .dic, que posee un formato texto y donde cada línea del mismo contiene la palabra seguida de su significado, pudiendo existir múltiples lineas para el mismo pero comenzando con carácter +.

**Analisis:**

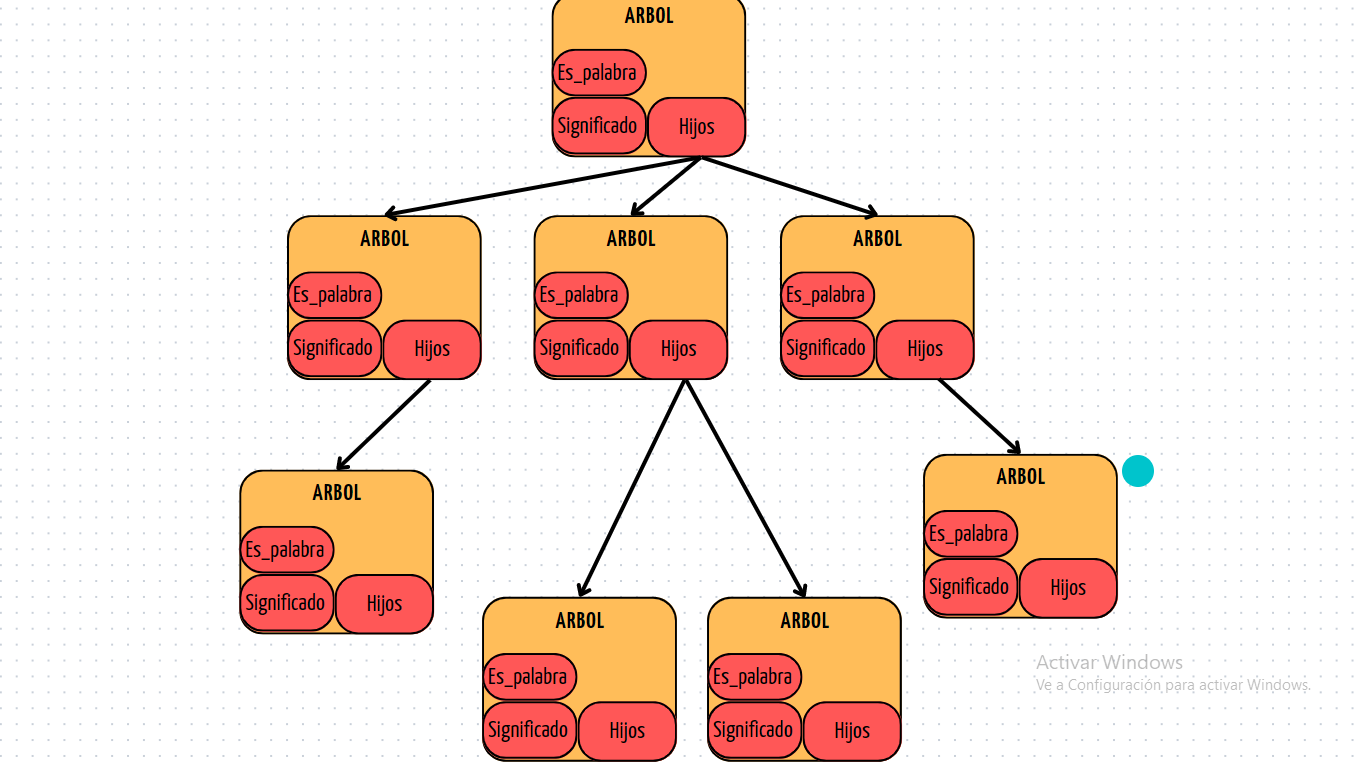
En implementación con arreglos el acceso a memoria es mas rápido, lógicamente por la característica de los arreglos, en cambio con listas es al contrario, mayor eficiencia con respecto al ahorro de memoria, pero lógicamente el acceso de forma secuencial, que lo hará mas lento.

El punto a considerar es que tanta memoria habrá de diferencia entre ambas, teniendo en cuenta solo un nodo, en la estructura con arreglos se reservan 26 espacios de apuntadores, cada puntero en 64 bits ocupa 8 bytes, teniendo 208 bytes de memoria solo en apuntadores, por otro lado los significados considerando 200 caracteres serian 200 bytes por significado, teniendo al rededor de unos 400 bytes en memoria, también siendo esta implementación la mas sencilla y fácil de corregir

Por otro lado la estructura basada en listas no tendría problemas con la memoria, ya que cada nodo solo tendría dos apuntadores, uno a la lista de los demás elementos y otro con los hijos, siendo estos 16 bytes, mas el significado que son 200 bytes, así que el ahorro de memoria es de la mitad, la velocidad de acceso en vez de ser de O(n) en la implementación de arreglos, pasa a ser mayor a O(n) por la naturaleza de las listas enlazadas.

**Diseño:**

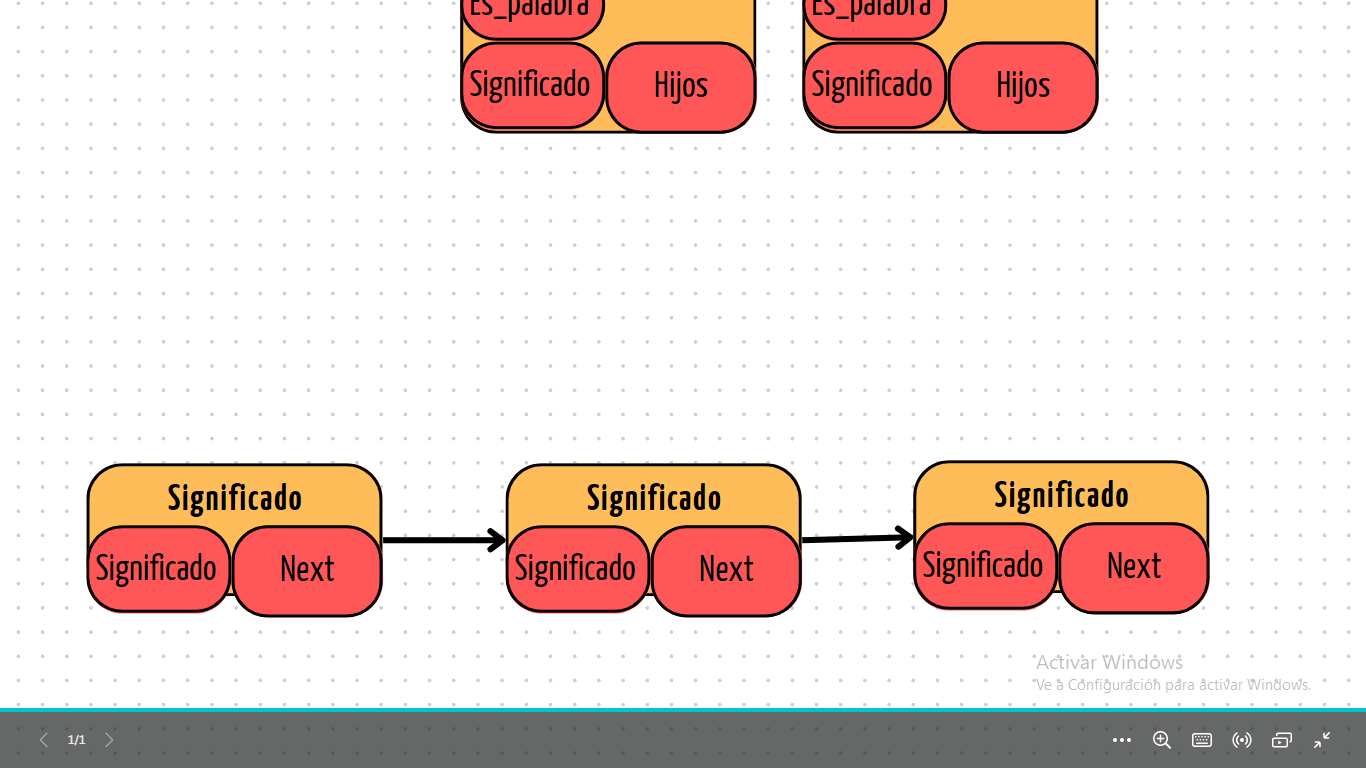
Ambas implementaciones tienen sus ventajas y desventajas, se optara por la implementación con arreglos por su mayor velocidad de acceso a costa de la memoria. Para indicar el final de una palabra se usa una bandera, un booleano o entero que indique verdadero o 1 cuando se haya llegado al final de una palabra. En el siguiente esquema se expresa la estructura general, en el que campo de los hijos es un arreglo de apuntadores con 26 elementos.



Para poder organizar los nodos con arreglos, es necesario determinar la posición de las claves en el mismo, para eso solamente convertiremos cada carácter en minúscula y le restaremos el carácter ‘a’, mientras la entrada sean letras dentro de A-Z y a-z, los indices siempre serán de 0-26.

Por ultimo, al tener la posibilidad de que una palabra tenga múltiples significados, simplemente se guardara la palabra anterior antes de seguir con la siguiente, se confirmara que la palabra sea igual a el carácter ‘+’, si es así se asigna el significado ingresado a la palabra anterior, de lo contrario se asigna a la nueva palabra.

Para hacer mas sencilla las operaciones con los significados, se hace con una lista simplemente enlazada, y los nuevos significados se van añadiendo al principio, de esta forma al buscar una palabra, o varias, solo se recorrerá la lista simplemente enlazada imprimiendo sus significados. El siguiente esquema representa la lista de significados.



**Implementación:**

**Empecemos por el programa centra llamado diccionario.c:**

#include "arbol.h"

Node \*dic;

char opcion[20];

int main(){

while (1)

{

printf("Escriba un comando con su argumento:\n");

fgets(opcion,20, stdin);

/\*Esta funcion se puede usar para separar una cadena por un delimitador

en este caso solo la usaremos para eliminar el salto de linea del texto entrante

ya que esto pudiera causa problemas al leer archivos y buscar palabras\*/

strtok(opcion,"\n");

/\*se verifica la primera letra, que es la del comando a recibir

en cada caso se llama a la función necesaria\*/

switch (opcion[0])

{

case 'q':

return 0;

case 'l':

dic=Cargar(dic,&opcion[2]);

break;

case 's':

Palabra(dic,&opcion[2]);

break;

case 'p':

Prefijo(dic,&opcion[2]);

break;

case 'h':

help();

default:

break;

}

}

return 0;

}

**Archivo de cabecera llamado arbol.h:**

#ifndef \_ARBOL\_H

#define \_ARBOL\_H

/\*Se usa ifndef y define para confirmar si la libreria ya ha sido incluida en el archivo

y no genere problemas, a la hora de compilar el compilador solo la incluira una vez\*/

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

/\*Estructura de la lista enlazada que contendra los significado\*/

typedef struct significado{

char significado[200];

struct significado \*next;

}Significado;

/\*Estructura general del arbol trie\*/

typedef struct nodo{

struct nodo\* hijos[26];

bool es\_palabra;

Significado \*significado;

}Node;

/\*Funciones publicas a usar en el programa principal\*/

/\*Funcion que recibe el nombre del archivo y crea el TAD diccionario

con la informacion del archivo\*/

Node\* Cargar(Node\* dic, char\* N\_archivo);

/\*Funcion que imprime el significado si la palabra

se encuentra almacenada, de lo contrario imprime que no se encuentra\*/

void Palabra(Node\* dic,char \*palabra);

/\*Funcion que imprime un mensaje de ayuda\*/

void help();

/\*Imprime todos los significados de palabras que tengan de prefijo el argumento ingresado\*/

void Prefijo(Node\* dic, char \*prefijo);

#endif

**Archivo .c de la librería llamado arbol.c:**

#include "arbol.h"

/\*Macro que sera el mensaje de ayuda\*/

#define ayuda "Uso:[opcion] [argumento]\n Opciones validas:\n\

l Carga el archivo desde el nombre indicado como argumento con su extension\n\

s Busca el significado de la palabra indicada como argumento\n\

p Imprime todos los significados de las palabras que comienzan con el prefijo\n\

h Muestra este mensaje de ayuda\n\

q Sale de la aplicacion"

/\*Se define los tamaños maximo\*/

#define max\_palabra 15

#define max\_definicion 200

/\*macro para calcular el indice de las claves para construir el TAD\*/

#define indice(a) ((int)tolower (a) - 'a')

//Funcion que crea un nodo para el diccionario

static Node\* CreaNodo(){

Node \*newp;

if ((newp=(Node\*)malloc(sizeof(Node)))==NULL){

printf("error malloc");

return NULL

}

// asigna NULL a todos los apuntadores del arreglo

for(int i=0;i<26;i++)

newp->hijos[i]=NULL;

newp->es\_palabra=false;

newp->significado=NULL;

return newp;

}

// Crea un nodo de la estructura que contiene los significados

static Significado\* CreaSignificado(char \*significado){

Significado \*newp;

if ((newp=(Significado\*)malloc(sizeof(Significado)))==NULL){

printf("Error en malloc");

return NULL;

}

newp->next=NULL;

//copia el significado dentro de la estructura

strcpy(newp->significado,significado);

return newp;

}

//Añade al inicio de la lista de significados un nuevo significado

static Significado \*AñadeSignificado(char \*significado, Significado\* list){

Significado \*newp=CreaSignificado(significado);

newp->next=list;

return newp;

}

//Añade palabras en la estructura

static Node\* AñadirPalabra(char\* palabra, char\* significado, Node\* dic){

Node\* p;

int indice;

//Si el diccionario no existe, se crea

if(!dic)

dic=CreaNodo();

// se usa un apuntador para moverse en la estructura, comenzando por la raiz

p=dic;

// avanza con un contador hasta el final de el largo de la palabra a ingresar

for(int i=0;i<(int)strlen(palabra);i++){

//calcula el indice del caracter

indice=indice(palabra[i]);

//verifica si el apuntador es nullo, e caso de ser verdadero lo crea y avanza

if (p->hijos[indice] == NULL){

p->hijos[indice]=CreaNodo();

//en caso de que no sea nulo significa que la clave ya existe, por lo que solo avanza

p=p->hijos[indice];

};

//al salir del ciclo es que es el final de la palabra

//se indica que es la palbra con la bandera y se añade el significado

p->es\_palabra=true;

p->significado=AñadeSignificado(significado,p->significado);

return dic;

}

//Funcion que hace free a todos los nodos de los significados

static void EliminaSignificado(Significado\* sig){

if(!sig)

return;

EliminaSignificado(sig->next);

free(sig->significado);

return;

}

//funcion que hace free a todos los nodos del diccionario

static void Elimina(Node\* dic){

if(!dic){

return;

}

//Si es una palabra, se eliminan los significados

if(dic->es\_palabra){

EliminaSignificado(dic->significado);

}

//se recorren todos los elementos del arreglo, si no es nulo, avanza con

//la funcion elimina haciendo free al nodo

for(int i=0;i<26;i++){

if(dic->hijos[i]!=NULL)

Elimina(dic->hijos[i]);

}

free(dic);

dic=NULL;

return;

}

//Recibe el nombre del archivo, lo abre y manipula para crear la estructura

Node\* Cargar(Node\* dic, char \*N\_archivo){

//se crea el archivo y variables necesarias para las operaciones

char\* file;

char palabra[max\_palabra],significado[max\_definicion],anterior[max\_palabra];

//Si el diccionario existe, se borra

if(dic){

Elimina(dic);

dic=NULL;

}

FILE \*Archivo;

//Se abre el archivo

Archivo=fopen(N\_archivo,"r");

//En caso de que el archivo sea nulo, se regresa un error

if(Archivo==NULL){

printf("\nError al abrir archivo\n");

return NULL;

}

//Se lee la primera palabra, fscanf leera la entrada hasta el primer espacio en blanco

//esta es la palabra a ingresar

while (fscanf(Archivo,"%s",palabra)!=EOF)

{

//fgets leera hasta el primer salto de linea, este es el significado

fgets(significado,max\_definicion,Archivo);

//si la palabra leida es el caracter '+' significa que se añade el significado a la palabra anterior

if(palabra[0]=='+'){

dic=AñadirPalabra(anterior,significado+1,dic);

}else{

//de lo contrario se añade la nueva palabra y se copia en la variable 'anterior'

dic=AñadirPalabra(palabra,significado+1,dic);

strcpy(anterior,palabra);

}

}

//se devuelve el diccionario

return dic;

}

//Fiuncion que busca la lista de significados y la regresa

static Significado \*ObtenerSignificados(Node\* dic, char \*palabra){

Node\* p=dic;

int indice;

//recorre de la misma forma que en el asignar

for(int i=0;i<(int)strlen(palabra);i++){

indice=indice(palabra[i]);

//en caso de que el arreglo con el indice indique a null

//significa que la palabra no se encuentra en el diccionario

if (p->hijos[indice] == NULL){

printf("No se encuentra en el diccionario\n");

return NULL;

}else

p=p->hijos[indice];

}

return p->significado;

}

//Imprime los significados obtenidos de la funcion ObtieneSignificado

void Palabra(Node\* dic,char \*palabra){

//la lista de significados es obtenida

Significado \*p=ObtenerSignificados(dic,palabra);

//En caso de que haya significados, se imprime la palabra buscada al principio

if(p)

printf("%s: ",palabra);

//Se recorre la lista hasta llegar al final imprimiendo los significados

while (p)

{

printf("%s", p->significado);

p=p->next;

}

}

//Funcion auxiliar que recorrera toda la estructura, imprimiendo cada significado que encuentre

static void recorre(Node\* dic){

Significado\* list;

//confirma si el nodo actual es el final de una palabra, si es correcto, recorrela la lista

if(dic->es\_palabra){

list=dic->significado;

while (list)

{

printf("%s",list->significado);

list=list->next;

}

}

//recorre todos los nodos desde el primero hasta el ultimo

//en caso de encontrar un nodo con hijos, lo recorre tambien

for(int i=0;i<26;i++){

if(dic->hijos[i])

recorre(dic->hijos[i]);

}

}

//Funcion que imprime el mensaje de ayuda escrito al inicio del programa

void help(){

printf(ayuda);

}

//Funcion que avanza al nodo que contiene el prefijo indicado

void Prefijo(Node\* dic, char \*prefijo){

Node\* p;

Significado\* list;

int indice;

//Recorre la estructura de la misma forma que al crear y buscar una palabra

for(int i=0;i<(int)strlen(prefijo);i++){

indice=indice(prefijo[i]);

//En caso de que alguno de los nodos buscados no exista, no existe ninguna palabra con ese prefijo

if (dic->hijos[indice] == NULL){

printf("No se encuentra en el diccionario\n");

return ;

}else

dic=dic->hijos[indice];

}

p=dic;

//Se recorren e imprimen todos los hijos del nodo

recorre(dic);

}

**Makefile:**

all: diccionario diccionario.o arbol.o

arbol.o: arbol.c

gcc -c arbol.c

diccionario.o: diccionario.c

gcc -c diccionario.c

diccionario: arbol.o diccionario.o

gcc -o main diccionario.o arbol.o

rm \*.o

**Casos de prueba:**

**Conclusiones:**

Es una estructura muy útil al momento de organizar datos a partir de seccionarlos en partes, usando sus partes por sus claves, la única desventaja aparente con respecto a la implementación con listas es el hecho de tener que recorrer los arreglos de forma secuencial para recorrerlos todos, que en el caso de las listas también aplicaría, pero omitirían los que en los arreglos son NULL porque directamente no existen, para buscar palabras puntuales los arreglos son mas eficientes, pero requeriría comparar con una implementación con listas el recorrer todo el arbol, de esta forma identificar que tanta diferencia en velocidad existe entre ambas.

**Referencias:**

"Implementación Trie en C – Insertar, Buscar y Eliminar". Techie Delight | Ace your Coding Interview. https://www.techiedelight.com/es/trie-implementation-insert-search-delete/

"Trie | (Insert and Search) - GeeksforGeeks". GeeksforGeeks. https://www.geeksforgeeks.org/trie-insert-and-search/