**UNIVERSIDAD CATOLICA ANDRES BELLO**.

**FACULTAD DE INGENIERIA.**

**ESCUELA DE INGENIERIA INFORMATICA**

**PUERTO ORDAZ – ESTADO BOLIVAR.**

**CATEDRA: ALGORITMOS Y PROGRAMACION II.**

**SEMESTRE: III**

Proyecto 3:

Diccionario de sinónimos y antónimos implementado con Tries

ANGEL PEÑA

XAVIER BECKLES

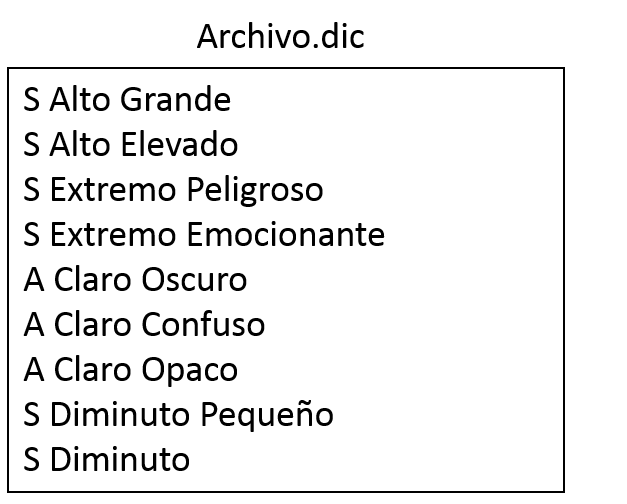
PROF. JESUS LAREZ

CIUDAD GUAYANA, ENERO DE 2015

Indice

Premisas

Se debe hacer un diccionario de sinónimos y antónimos para la terminal de Linux. La estructura que datos que contiene los sinónimos y antónimos debe ser un Trie el cual será construido a partir de un archivo con extensión ‘.dic’ con una lista de dos palabras y un indicador que determina si son sinónimos o antónimos entre sí. Dicho archivo es cargado por el usuario y puede ser cargado mediante el uso del modo interactivo y el comando ‘cargar’. Dicho archivo tiene el siguiente formato:



La letra ‘s’ y ‘a’ definen si la segunda palabra es sinónimo o antónimo de la primera (y viceversa), y los pares de palabras no tienen que estar necesariamente ordenados. Se supone que las palabras están compuestas por letras, pero de todas formas se valida en caso de encontrarse con un numero o signo.

El diccionario debe trabajar mediante dos modalidades distintas. En una basta con introducir (en la terminal de Linux) el nombre del programa a ejecutar, un indicados ‘s’ o ‘a’, y una palabra, seguido de esto se mostraran todos los sinónimos o antónimos de la misma, dependiendo del indicador. La segunda modalidad es un diccionario interactivo, en el que después de cargar un archivo, se pueden revisar sinónimos y antónimos de palabras completas o palabras que comiencen con un expresión. Esta modalidad debe iterar hasta que el usuario desee salir del diccionario. Los comandos en este modo son:

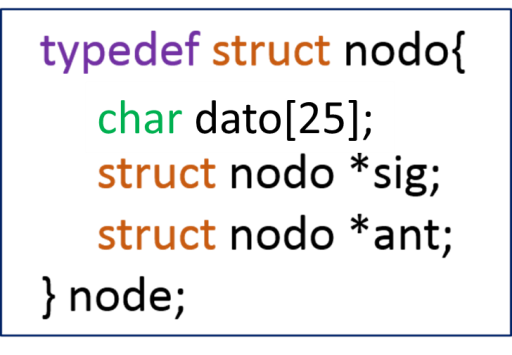
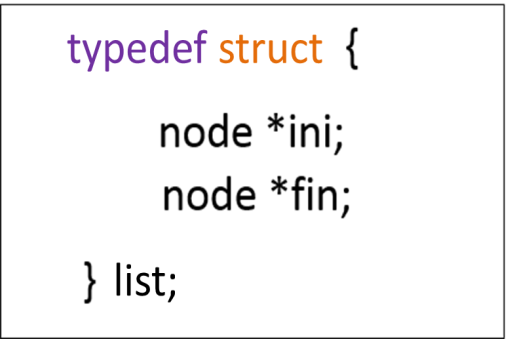
* Cargar palabra: carga el archivo palabra.dic.
* s palabra : muestra los sinónimos de la palabra.
* a palabra: muestra los antónimos de la palabra.
* e expresión: muestra los sinónimos y antónimos de las palabras que comiencen con dicha expresión.
* ayuda . : muestra el menú de ayuda, en donde se describen todos los comandos.
* salir . : sale del modo interactivo.

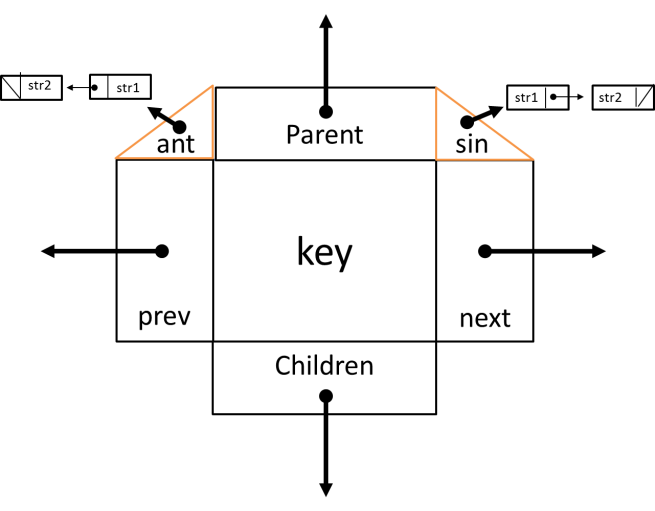
Marco Teórico

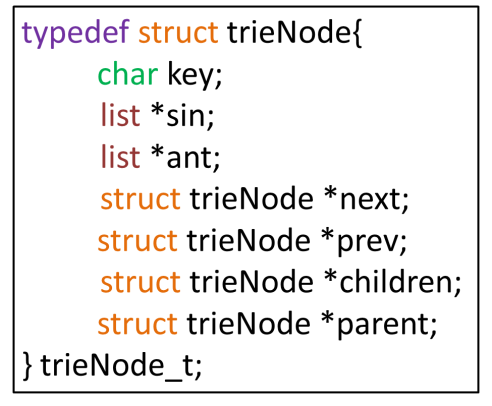
Estructuras de datos Utilizadas

Para la implementación del diccionario, se usaron principalmente dos estructuras de datos las listas y los tries. En primer lugar, para guardar los sinónimos y antónimos de cada palabra, se usaron listas doblemente enlazadas, contenidas dentro de la estructura de cada nodo del trie.

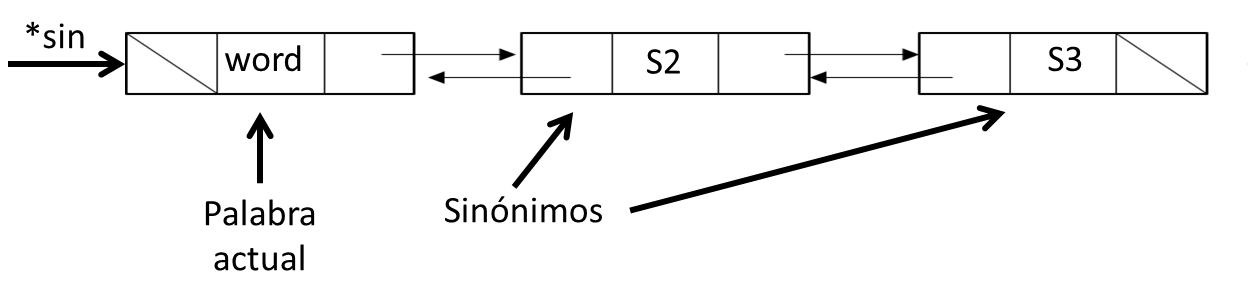
Estructura node Estructura Lista

 Para el almacenamiento de las palabras y la búsqueda de las mismas se usó un Trie, el cual es una estructura de datos eficiente al momento de guardar y recuperar cadenas de caracteres, ya que en vez de contener una palabra en cada nodo, contiene una letra de una palabra en cada uno. Esto permite buscar una cadena dentro de la estructura en orden O(h), siendo h la longitud de la palabra.

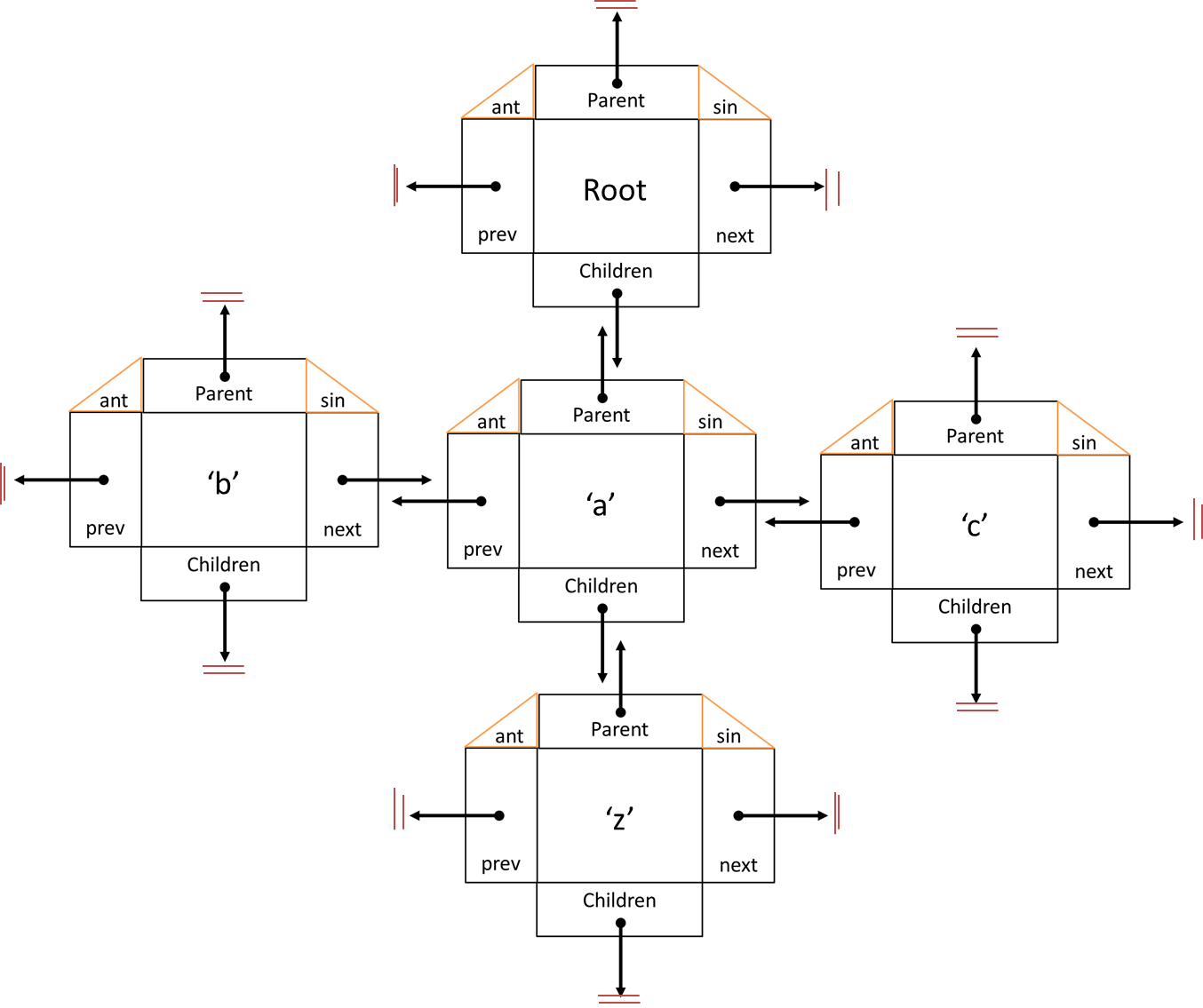


Cada nodo posee un puntero a uno de sus hijos (\*children), También tiene dos punteros a nodos en su mismo nivel, uno para cada vecino adyacente (\*prev y \*next), y uno a su nodo padre ( \*parent). El campo ‘key’ guarda la letra que se le asigna al nodo. Las listas ‘sin’ y ‘ant’ son la estructura que se usa para almacenar los sinónimos y antónimos de una palabra.

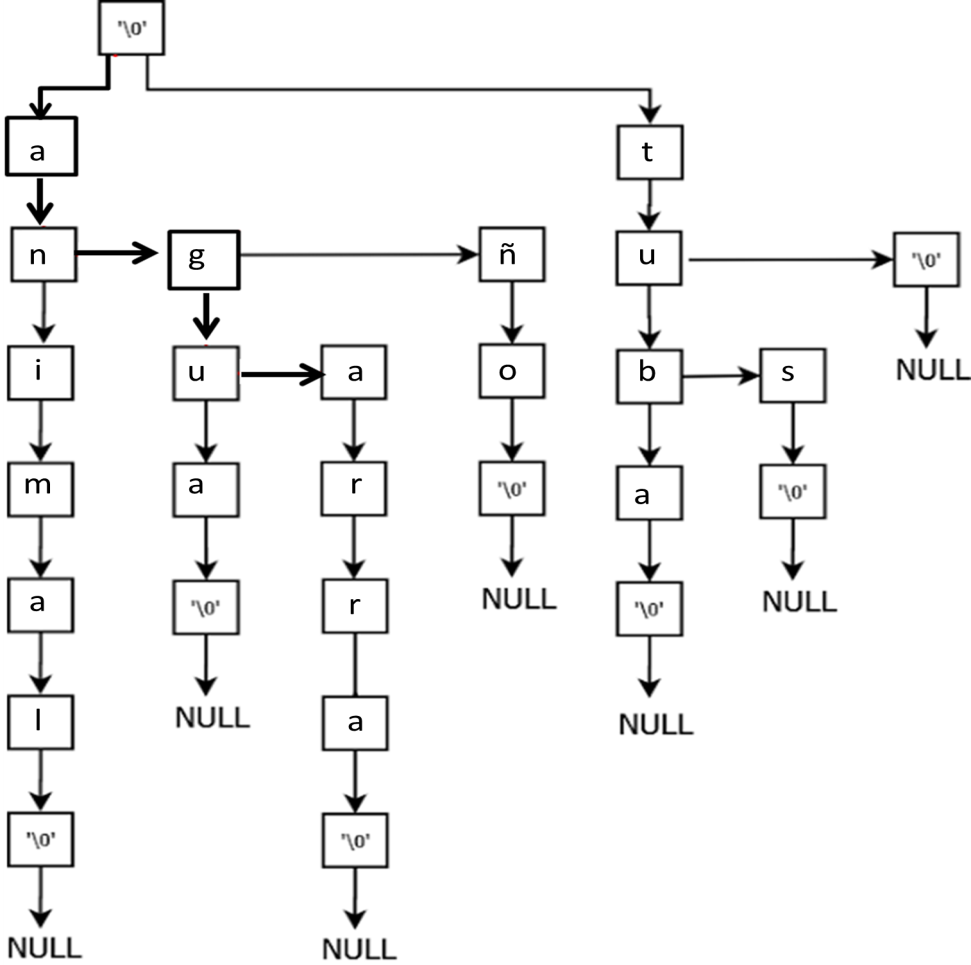


El primer elemento de la lista es la palabra actual, y los siguientes son los sinónimos (o antónimos) de la misma. Tener la palabra como primer elemento facilita la tarea de mostrar los resultados de la búsqueda por expresión en el modo interactivo, pero al momento de mostrar los resultados de una búsqueda de sinónimos o antónimos, solo se imprimen los elementos a partir de la segunda posición de la lista. Las listas que contienen los sinónimos de la palabra se encuentran solo en el nodo que contiene el carácter de fin de palabra ‘\0’, en los otros nodos con las demás letras de la cadena, los punteros a lista están vacíos.

La estructura de datos ‘Trie’ se crea a partir de estos nodos, enlazándolos letra por letra para crear palabras y familias de palabras, emparentadas por sufijos.



Por ejemplo, obtenemos el siguiente Trie al insertar las siguientes palabras: animal, agua, agarra, año, tuba, tus y tú. Cada nodo hijo representa la siguiente letra de una palabra, y cada nodo con la llave ‘\0’ representa el final de una palabra y contiene las listas con los sinónimos y antónimos de la misma. Si una palabra se encuentra contenida dentro de otra en el árbol, pero no se encuentra su fin de palabra, esa palabra no existe dentro de la estructura. Tomemos por ejemplo la palabra ‘anima’, esta se encuentra dentro de ‘animal’, pero no existe en el árbol debido a que no está el nodo con su fin de palabra.



Nótese que cada nivel del trie representa una posición de una letra en una cadena. Es decir los nodos en el nivel 1 (en este caso ‘a’ y ‘t’) representan todas las primeras letras de todas las palabras del diccionario. Los nodos en el segundo nivel, representan todas las segundas letras del diccionario y así sucesivamente. Por lo que si un nodo tiene 27 vecinos, esos representan 27 caminos diferentes que puede tomar la palabra formada por todos sus padres.

La búsqueda en esta estructura es eficiente debido a que para buscar una palabra, se recorre el trie nivel por nivel en búsqueda de la letra que corresponde a ese nivel, y como cada uno representa una letra en la palabra, el tiempo de búsqueda depende solamente de la longitud de la misma.

Funciones de la cabecera “trie.h”

Todas las funciones usadas para manejar la estructura de datos principal, los tries, están contenidas dentro de la cabecera “trie.h”.

***void LowerCase ( char str[] ):*** recibe una cadena de letras y la transforma en una cadena formada únicamente por letras minúsculas. Se usa para evitar que dos cadenas iguales arrojen resultados diferentes únicamente por el hecho de una contener mayúsculas y la otra no.

***trieNode\_t \*TrieCreateNode(char key****):* reserva memoria para un nuevo nodo, asigna el carácter dado al campo ‘key’ del mismo, e inicializa todos sus punteros a NULL. Además, crea dos nuevas listas y las asigna a sus campos ‘sin’ y ‘ant’.

***trieNode\_t\* TrieSearch(trieNode\_t \*root, const char \*key):*** Busca dentro del trie la palabra dada. De conseguirla, retorna un puntero a su último nodo, el cual es el fin de la palabra (\0) y contiene las listas de sinónimos y antónimos. En caso contrario, retorna NULL.

***void TrieAdd(trieNode\_t \*\*root, char \*palabra, char original[], char \*palabra2, char estado ):*** Agrega al trie la palabra pasada como el parámetro ‘palabra’. Cuando termina de insertarla, agrega la palabra original a las listas que se encuentran dentro de su último nodo (\0). Además agrega la palabra dada como parámetro ‘palabra2’ a su lista de sinónimos o a su lista de antónimos, dependiendo de si el carácter ‘estado’ es “s” o “a”.

***trieNode\_t\* TrieExpSearch(trieNode\_t \*root, const char \*key):***  Se usa para buscar una expresión dentro del Trie. Es idéntica a la función ‘TrieSearch()’, con la excepción de que esta busca una palabra incompleta (la expresión).

***void MostrarExpresion ( trieNode\_t \*root ):*** Recorre recursivamente todo el sub-arbol dado como parámetro y muestra todas las palabras encontradas, sus sinónimos y antónimos. Se usa para mostrar las palabras contenidas en el sub-arbol que tiene como raíz la última letra de una expresión.

***void BuscarExpresion( trieNode\_t \*root , char exp[] ):*** Mediante la función ‘TrieExpSearch()’ busca la expresión dada como parámetro dentro del Trie y muestra todas las palabras, sus sinónimos y sus antónimos usando la función ‘MostrarExpresion()’.

***int validarPalabra ( char s[] ):*** Se usa para verificar si la cadena pasada como parámetro contiene un carácter invalido, ya sea un numero o un símbolo. Retorna 1 si no se encuentra un error, de lo contrario retorna 0.

***void cargar\_datos(char archivo[], trieNode\_t \*root):*** Utiliza las funciones anteriores para cargar los datos desde el archivo .dic, e introducirlas en la estructura de datos principal (el Trie). Además, verifica que las palabras dentro del archivo sean válidas, es decir, que no contengan un carácter no permitido. De encontrar un error al momento de cargar una palabra, mandara un mensaje de error al usuario y no cargara dicha palabra al Trie.

***trieNode\_t \*LiberarTrie( trieNode\_t \*root ):*** Recorre recursivamente el Trie, liberando la memoria reservada para cada nodo y cada lista contenida dentro de ellos, retorna un puntero a NULL.

Tipo Abstracto de datos Lista

Para el almacenamiento de los sinónimos y antónimos de las palabras contenidas dentro de los nodos del Trie, se usó el tipo abstracto de dato listas, las cuales permiten las operaciones de insertar al final, buscar y mostrar, adecuadas para esta tarea. Este TAD fue implementado mediante listas doblemente enlazadas y sus funciones se encuentran en la cabecera “ListasDobles.h”.

***list \*nueva\_lista ( void ):*** asigna memoria para una nueva lista, e inicializa sus dos punteros (inicio y fin) en NULL.

***void insertar\_fin ( list \*lp, char palabra[] ):*** Crea un nodo que contiene la palabra dada como parámetro y lo inserta al final de la lista dada.

***int BuscarEnLista( list \*lp , char str[]):*** Traversa una lista, buscando una palabra en cada uno de sus nodo. De encontrarla retorna 1, de lo contrario retorna 0.

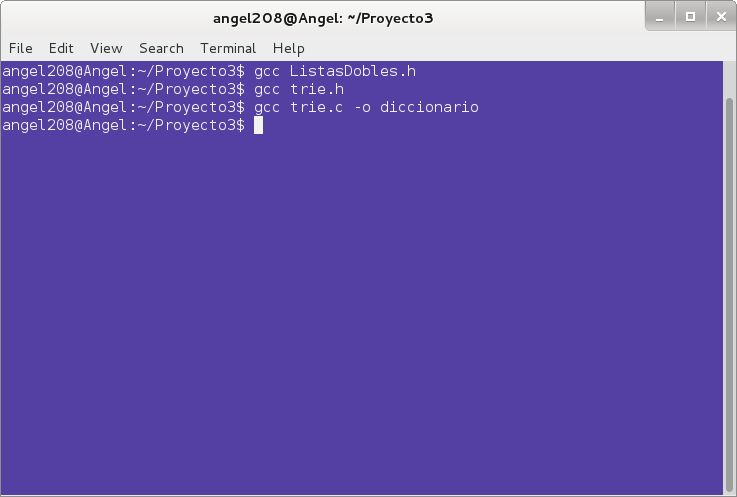
***void free\_all ( list \*lp ):*** Traversa una lista, liberando la memoria asignada a cada uno de sus nodos, para luego liberar la memoria asignada para la lista en sí.

**Imprimir una lista:** Para mostrar los resultados de las búsquedas en el programa se requiere mostrar las listas que contienen los sinónimos y los antónimos de las palabras. Se usan dos funciones para este fin y las dos difieren solamente en el formato en el cual se imprimen los resultados en consola:

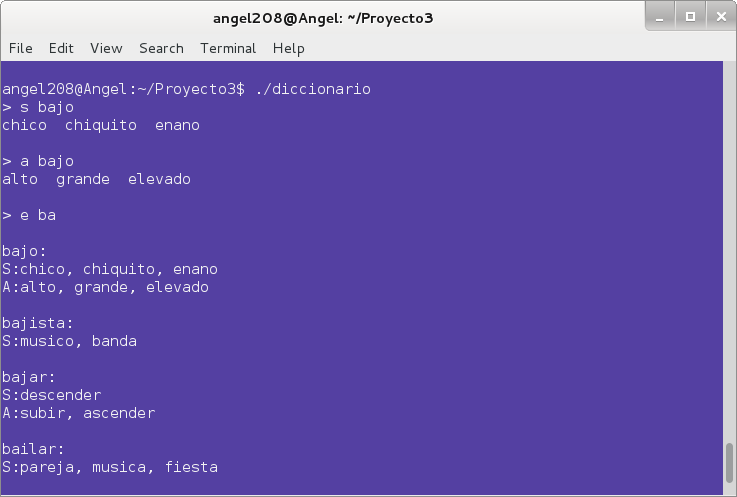
***void mostrarExp( list \*lp ):*** Se usa al momento de mostrar los resultados de la búsqueda de una expresión en el modo Iterativo

***void mostrar( list \*lp ):*** Se usa para mostrar los resultados de la búsqueda de sinónimos o antónimos tanto en el modo iterativo como en el modo comando.

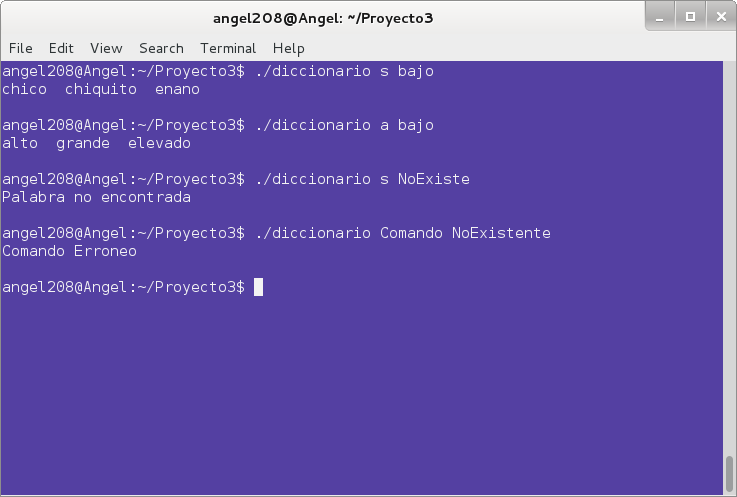
Casos de Prueba



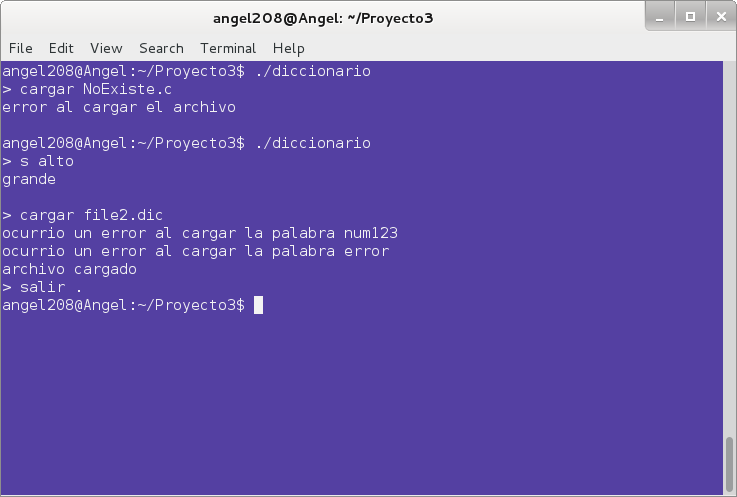
*Compilación Por Separado*

**

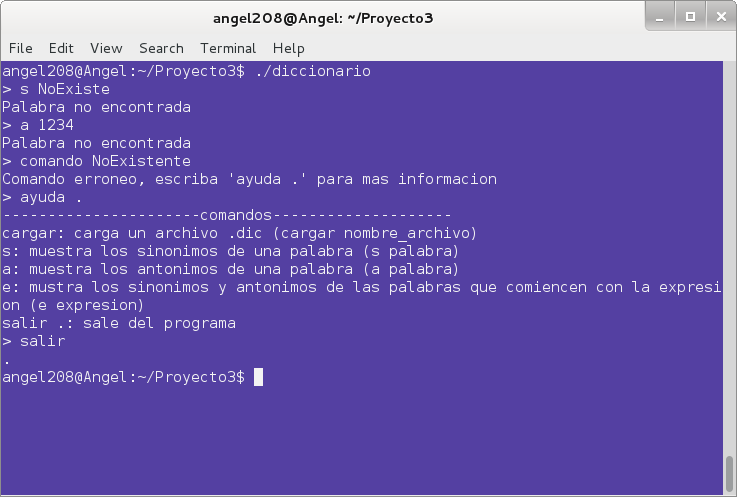
*Modo interactivo*

**

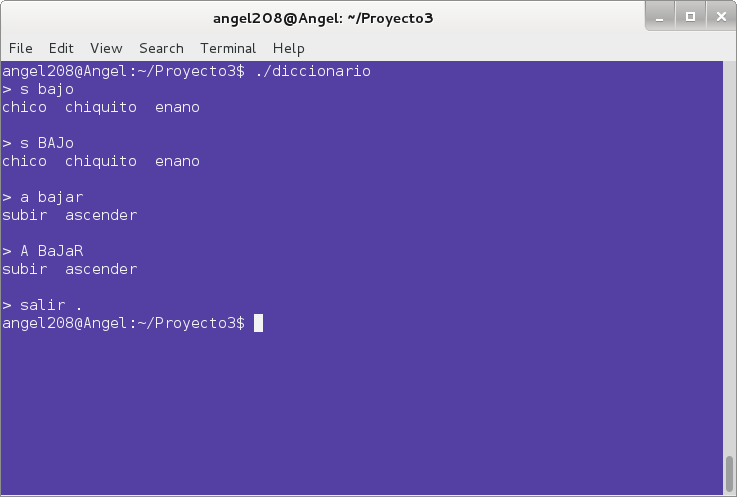
*Modo Comandos*

**

*Carga de archivos*

**

*Validaciones y menú de ayuda*

**

*Validación de Mayúsculas*

Conclusiones

Referencias

Fuentes del Programa