ANALISIS Y DISEÑO DE ALGORITMOS I

Trabajo Final

- **Diego Iarussi**

- INTRODUCCION AL PROBLEMA :

El trabajo consiste en la especificación e implementación del **TDA TRIE**, con el objetivo de almacenar palabras para realizar búsquedas eficientes. Dicho TDA, debe poder realizar las siguientes tareas:
. Agregar una palabra
. Eliminar una palabra
. Dado un prefijo, devolver todas las palabras que contengan ese prefijo
. Decidir si una palabra es valida o no

Inicialmente, debemos elegir una estructura de datos adecuada para poder representar el TDA sugerido de la mejor manera posible. Y luego, centrarnos en el diseño e implementación de los algoritmos que nos van a permitir desarrollar la funcionalidad que deseamos para nuestro programa.
Una vez hecho todo esto, debemos realizar una interfaz gráfica que nos permita la entrada de datos y visualización de los resultados obtenidos.
Ademas, contamos con un archivo de texto que posee un listado de palabras(3000) en idioma castellano, el cual nos va a facilitar la entrada de datos a nuestro programa.

-TIPOS DE DATOS UTILIZADOS :

* **Tst :** Esta es la clase que representa al TDA TRIE, su nombre viene de la abreviatura de TERNARY SEARCH TRIE. Cada nodo posee una variable de tipo char para almacenar un carácter, una variable de tipo integer para guardar la clave de cada palabra en el ultimo carácter de la misma (por defecto, se van a encontrar en 0 todos los nodos). Ademas, tiene tres hijos, a los cuales se accede de la siguiente manera:
  - Nodo Izquierda : si el carácter que se desea almacenar es menor alfabéticamente que el actual.
  - Nodo Medio : si el carácter que se desea almacenar es igual al actual.
  - Nodo Derecha : si el carácter que se desea almacenar es mayor al actual.

  . insert : este procedimiento se encarga de almacenar una palabra en la estructura detallada anteriormente.
  . get : esta función devuelve la clave de la palabra que se quiere buscar. Si la palabra no existe, devuelve un 0.
  . getListWords : esta función devuelve todas las palabras almacenadas en la estructura. Util para mostrar por pantalla los resultados obtenidos.
  . contains : esta función devuelve un valor booleano (true o false) dependiendo si una palabra dada existe en la estructura o no.
  . depth : esta función devuelve un valor entero que representa la profundidad de la estructura.
  . remove : este procedimiento se encarga de eliminar una palabra almacenada en la estructura.
  . prefix : esta función devuelve todas las palabras que contienen un prefijo dado.
  . uploadFile : este procedimiento se encarga de cargar todas las palabras contenidas en el archivo de texto mencionado en la introducción, almacenándolas en la estructura.
  . saveAllWords : este procedimiento se encarga de guardar todas las palabras, contenidas en la estructura, en un archivo de texto para luego volver a utilizarlas, si es que así lo desea el usuario.

Para las siguientes clases se utilizo un IDE (entorno de desarrollo integrado) que se llama Qt Creator.
Qt Creator es un IDE multi plataforma programado en C++, JavaScript y QML creado por Trolltech el cual es parte de SDK para el desarrollo de

aplicaciones con Interfaces Gráficas de Usuario (GUI por sus siglas en inglés) con las bibliotecas Qt.
Qt Creator se centra en proporcionar características que ayudan a los nuevos usuarios de Qt a aprender, también aumentar la productividad.[3]

- Editor de código con soporte para C++, QML y ECMAscript.
- Herramientas para la rápida navegación del código.
- Resaltado de sintaxis y auto-completado de código.
- Control estático de código y estilo a medida que se escribe.
- Soporte para refactoring de código.
- Ayuda sensitiva al contexto.
- Plegado de código (code folding).
- Paréntesis coincidentes y modos de selección.

El depurador visual (visual debugger) para C++. Qt Creator muestra la información en bruto procedente de GDB de una manera clara y concisa.[3]

- Interrupción de la ejecución del programa.
- Ejecución línea por línea o instrucción a instrucción.
- Puntos de interrupción (breakpoints).
- Examinar el contenido de llamadas a la pila (stack), los observadores y de las variables locales y globales.

***MainWindows :** Es un widget especial, que conforma la ventana más completa y por tanto la principal de una aplicación gráfica, con menús, herramientas, status bar, dock, etc. En programación, un widget (o control) es un elemento de una interfaz (interfaz gráfica de usuario o GUI) que muestra información con la cual el usuario puede interactuar. Por ejemplo: ventanas, cajas de texto, checkboxs, listbox, entre otros. Un widget provee un punto de interacción con el usuario para la manipulación directa de un tipo de dato dado. En otras palabras, los widgets son bloques básicos y visuales de construcción que, combinados en una aplicación, permiten controlar los datos y la interacción con los mismos.
En todos los framework diseñados para desarrollar GUIs debemos tener un mecanismo para responder a los eventos producidos por los componente de la interface de usuario y también para emitir dichos eventos, en Qt contamos con los Signals & Slots para tal propósito, por

ejemplo, al presionar el botón Salir se genera un signal, si deseamos que dicho botón cierre la ventana entonces debemos establecer la función close() de la ventana como *slot*.
Signal : es emitido por un objeto cuando el mismo detecte un cambio de estado en alguna de sus propiedades.
Slot : es una función C++ que es invocada en respuesta a un signal, para que esto sea posible debemos conectarlos.

Private slots :

. on_actionOpen_triggered : este procedimiento se activa al presionar “Open” en la ventana principal de la aplicación, llama al procedimiento del Tst uploadFile y luego al getListWords (detallados anteriormente) para poder mostrar por pantalla la lista de palabras cargadas hasta el momento.

. on_actionSave_triggered : este procedimiento se activa al presionar “Save" en la ventana principal de la aplicación, llama al procedimiento del Tst saveAllWords y así queda guardado todo en un archivo de texto.

. on_actionNew_triggered : este procedimiento se activa al presionar “New” en la ventana principal de la aplicación, llama al destructor del Tst, crea uno nuevo y limpia toda la ventana principal para poder empezar de nuevo, sin necesidad de tener que abrir y cerrar la aplicación.

. on_PushBottom_clicked : este procedimiento se activa al apretar el botón “Accept” pero ademas, depende de lo que diga en el comboBox (el cual manipula el usuario):

- Insert : en este caso, se llama al procedimiento del Tst (insert) para agregar la palabra que se escribió en el lineEdit y se muestra por pantalla.
- Search : se llama a la función del Tst (contains) y se crea un QMessageBox con el valor obtenido y luego, se muestra por pantalla para saber si la palabra escrita en el lineEdit es valida o no.
- Remove : se llama al procedimiento del Tst (remove) para eliminar la palabra escrita en el lineEdit y se muestra por pantalla.
- Prefix : se llama a la función del Tst (prefix) y llama a un procedimiento (wordsPrefix) de la clase Dialog, la cual detallaremos a continuación. Luego, se muestra una ventana adicional, perteneciente a Dialog, con la lista de palabras con el prefijo escrito en el lineEdit.

Cabe destacar, que en el constructor de esta clase es donde se crea un puntero de tipo Tst y se inicia una variable integer llamada “val” en 1 que se modifica cada vez que se agrega una palabra en la estructura, de esta manera, podemos llevar un conteo de las palabras cargadas hasta el momento.

* **Dialog :** Es un widget especial, que conforma una ventana que además devuelve un valor siempre tras cerrarse (OK, Cancel, etc…).

Esta clase se utiliza únicamente para mostrar todas las palabras con un cierto prefijo de una manera mas ordenada, de tal modo que no aparezcan en la ventana principal en la que ya se están mostrando otros datos. Cuenta con un solo procedimiento:

. wordsPrefix : se encarga de agregar a la ventana todas las palabras con un cierto prefijo, escrito anteriormente en el lineEdit.

- DESCRIPCION Y COMPLEJIDAD TEMPORAL DE LOS ALGORITMOS UTILIZADOS :

N = cantidad de palabras.
L = tamaño de palabras.
K = lineas del archivo de texto.
P = cantidad de nodos.

. insert : llama a un procedimiento privado recursivo (insertWord) que es el que se encarga realmente de almacenar una palabra, carácter por carácter, en la estructura. Si el nodo actual es nulo, se crea uno y se le asigna el carácter. Si existe el nodo, se fija si el carácter actual es menor que el se debe almacenar, y se dirige hacia el nodo izquierdo. Si es igual, se dirige hacia el nodo medio, lo almacena y avanza al siguiente carácter. Y si es mayor, se dirige hacia el nodo derecho. Una vez que se llega al ultimo carácter a almacenar, se guarda con la clave de la palabra.
Complejidad Temporal: O(L + ln N).

. contains : llama a una función privada (getNode) que se encarga de buscar una palabra deseada, carácter por caracter, de la misma manera que el procedimiento de insert, solo que, si el nodo actual es nulo, se retorna NULL. Si no, devuelve el nodo que posee el carácter final, para que luego la función contains decida si devuelve true o false.
Complejidad temporal: O(L + ln N).

. remove : llama a un procedimiento privado (removeWord) que se encarga de eliminar la palabra deseada. Inicialmente, busca la palabra carácter por carácter, siguiendo la misma metodología de los procedimientos anteriores. Una vez posicionado en el final de la palabra, guarda la clave en una variable entera. Primero, pregunta si el nodo actual es un nodo hoja llamado a otra función privada (isFreeNode) que nos retornara un true o false. Si esto es así, lo elimina directamente, liberando el espacio de memoria. Si no es hoja, lo primero que se pregunta es si posee un hijo medio, esto le va a permitir darse cuenta si es parte de otra palabra. En este caso, el nodo no se elimina pero si se cambia la clave de la palabra por el valor por defecto (0), para que la palabra no sea mas considerada por si misma. Luego

se pregunta si el hijo izquierdo no es nulo, de ser así llama a un procedimiento privado (changeSuccessor) que se encarga de que, lo que apuntaba hacia el nodo actual, apunte al nodo mas derecho del actual (esto es, yendo al hijo derecho del hijo derecho y así sucesivamente hasta encontrar el ultimo), de esta manera, la estructura se mantiene ordenada siempre, con una complejidad de O(ln N). Por ultimo pregunta si el hijo derecho no es nulo, y en caso de no serlo, lo que apuntaba al nodo actual, ahora apunta hacia su hijo derecho, para después poder liberar el espacio de memoria sin pisar ningún dato.
Complejidad temporal: O(ln N + (L + ln N)) = O(L + ln N).

. prefix : primero busca el prefijo en la estructura y se posiciona en el carácter final, esto lo hace llamando a una función privada (getNode). Luego, carga todas las palabras, en orden, que poseen dicho prefijo en una QStringList invocando al procedimiento privado (getWords) y luego devuelve la lista de palabras.
Complejidad temporal: O(P).

. uploadFile : este procedimiento lee linea por linea el archivo de texto que contiene todas las palabras, y los almacena en la estructura llamando al procedimiento insert.
Complejidad temporal: O(K x (L + ln N)).

. saveAllWords : este procedimiento llama a la función privada (getWords) con un string vacio(“”), de manera que devuelva todas las palabras contenidas en la estructura. Y luego, las guarda en un archivo de texto, cada palabra en una linea del archivo.
Complejidad temporal: O(K + P).

- CONCLUSIONES :

R-way trie

- Búsqueda exitosa: Necesita examinar todos los L caracteres por igualdad.
- Búsqueda fallida: Pueda fallar en el primer carácter. Caso típico, examinar solo unos pocos caracteres.
- Espacio: R conexiones nulas en cada hoja.
- Conclusión: Búsquedas exitosas rápidas y fallidas aún más rápidas, pero desperdicia mucho espacio.

Ternary Search trie

- Búsqueda exitosa: El nodo en donde termina la búsqueda tiene un valor no nulo.
- Búsqueda fallida: Llegar a un conector nulo o que el nodo donde la búsqueda termina tenga un valor nulo.
- Observación: Se puede construir TSTs por medio de rotaciones para lograr garantizar un peor caso de L + log N.
- Conclusiones: TST es rápido como hashing (para claves de cadenas de caracteres), y eficiente en espacio.

<u>Hashing</u>
· Necesidad de examinar la clave completa.
· Búsqueda exitosa y fallida cuestan lo mismo.
· El rendimiento se basa en la función hash.
· No admite operaciones de tabla de símbolos ordenadas.
<u>TSTs</u>
· Funciona solo para claves de string.
· Solo examina los caracteres clave suficientes.
· La búsqueda fallida puede incluir solo unos pocos caracteres.
· Admite operaciones de tablas de símbolos ordenadas.
· Más rápido que hash (especialmente para errores de búsqueda).

Por lo tanto, fue una decisión correcta la de elegir a **Tst**.