Trabajo Práctico 2

Teclado Predictivo

Programación en C

Tecnología Digital II

1. Introducción

Hace mucho tiempo en una galaxia muy, muy lejana donde las pantallas capacitivas eran solo un producto de la ciencia ficción, los precursores de los seres humanos actuales utilizaban en sus teléfonos celulares teclados con botones para mandar mensajes. Como el costo de hacer cada botón era muy caro, resultaba prohibitivo tener un teclado completo con una tecla por letra del alfabeto. Algunos desarrolladores lo intentaron, pero en el caso general solo podíamos contar con un teclado numérico de unas pocas teclas. Aun así, en estas pocas teclas se encontraba codificado todo el alfabeto. Para lograr escribir una letra había que presionar varias veces una misma tecla hasta llegar a la letra requerida.Esta titánica tarea favoreció significativamente el uso de acrónimos para reducir la cantidad de letras de los mensajes, haciéndolos casi incomprensibles para un lector poco avispado.

En este contexto fue lógico armar diccionarios que buscarán predecir qué palabra completa quiere escribir el usuario en base a las primeras letras que logró escribir en el rudimentario artilugio.Así es como aparecieron los teclados predictivos. Su objetivo era identificar rápidamente la palabra que el usuario quería escribir, y presentar sugerencias de cuál podría ser. Para hacer esto, almacena todas las palabras del diccionario y permite que sean accedidas rápidamente.

Luego la tecnología fue mejorando un poco más, utilizando como dato la frecuencia con la que una palabra era escrita, o identificando incluso errores y arreglando las faltas ortográficas. Actualmente tenemos teclados que prácticamente mirándolos saben lo que queremos escribir.

El objetivo de este trabajo práctico es implementar una estructura de datos que nos permita recorrer un diccionario de palabras de forma eficiente y con esto implementar funciones para predecir un conjunto posible de palabras en base a un prefijo. La estructura que implementaremos está diseñada de forma simple, para que sea fácil de implementar. En implementaciones más avanzadas es posible ser más eficiente, tanto en rendimiento de los algoritmos de búsqueda como de espacio ocupado. La tarea será completar algunas de las funciones más básicas de esta estructura de datos.

El trabajo práctico debe realizarse en grupos de tres personas. Tienen tres semanas para realizar la totalidad de los ejercicios. **La fecha de entrega límite es el 2 de noviembre hasta las 23:59; y la de re-entrega es el 23 de noviembre hasta las 23:59**
Se solicita no realizar consultas del trabajo práctico por canales públicos. Limitar las preguntas al foro privado creado para tal fin.

2. Tipos de datos: `keysPredict`

A partir de las siguientes estructuras se define una `keysPredict`:

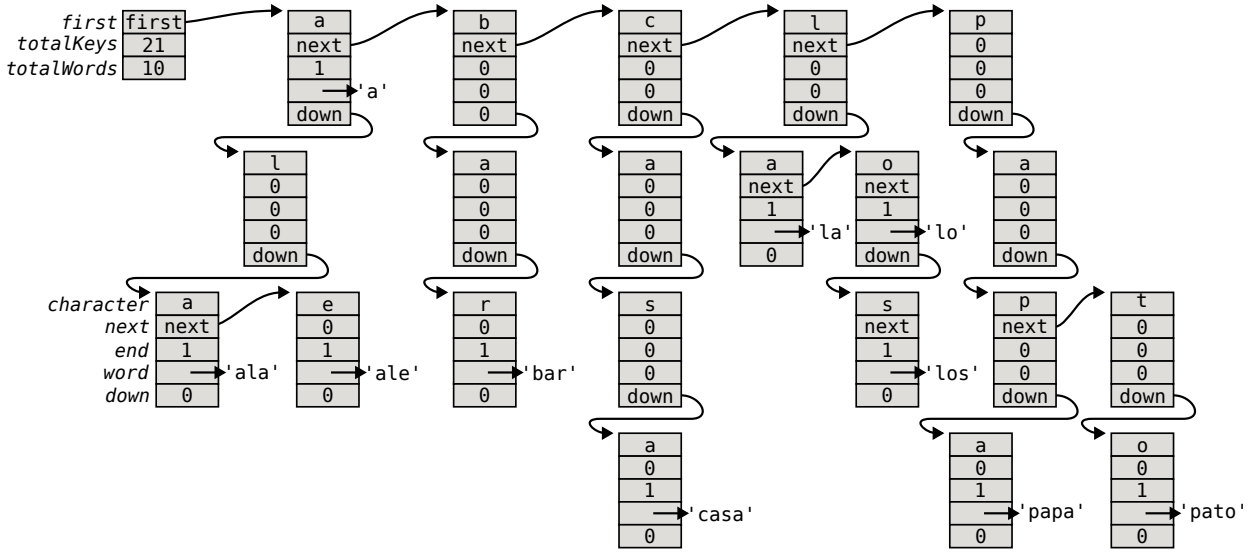
```
struct keysPredict {
    struct node* first;
    int totalKeys;
    int totalWords;
};

struct node {
    char character;
    struct node* next;
    int end;
    char* word;
    struct node* down;
};
```

La estructura `keysPredict`, contiene un puntero a una lista de listas de nodos de tipo `node`. Cada nodo representa una letra indicada por el campo `character`. El campo `next` apunta a la siguiente letra dentro del mismo nivel, mientras que el campo `down` permite llegar a la siguiente lista de letras en un nivel inferior.

Los campos `end` y `word` están relacionados, ya que indican la palabra completa formada por los nodos desde el primero al último. El campo `end` indica que ese nodo es nodo final de una palabra, mientras que `word` es un puntero a la palabra almacenada para ese nodo.

Dentro de la estructura principal, además del puntero `first`, existen dos campos más. El campo `totalKeys` indica la cantidad total de letras dentro de la estructura, mientras que `totalWords` corresponde a la cantidad total de palabras almacenadas.
A continuación se ilustra un ejemplo de la estructura:



En el ejemplo se puede observar como el primer nivel de la estructura está compuesto por las letras 'a', 'b', 'c', 'l' y 'p'. Estas siempre son almacenadas en orden alfabetico dentro de la lista. Luego los punteros `down` van apuntando a las siguientes listas, algunas pueden tener un solo elemento como la que parte del primer nodo con una 'a', o pueden tener más elementos, como el siguiente puntero. Esta primera parte del la estructura logra almacenar las palabras 'ala' y 'ale'. Siendo la última letra la única diferente.

Incluso, es posible tener palabras almacenadas en un nodo intermedio, como es el caso de la palabra 'lo', que un nodo más abajo construye la palabra 'los'.

3. Enunciado

Ejercicio 1

Implementar las siguientes funciones sobre *strings*.

- `int strLen(char* src)`
  Calcula la longitud de una *string* pasada por parámetro, indicando como respuesta la cantidad de caracteres de esta.
- `char* strDup(char* src)`
  Duplica un *string*. Debe contar la cantidad de caracteres totales de *src* y solicitar la memoria equivalente. Luego, debe copiar todos los caracteres a esta nueva área de memoria. Además, como valor de retorno se debe retornar el puntero al nuevo *string*. Esta función NO debe borrar la *string* original.

Ejercicio 2

Implementar las siguientes funciones auxiliares sobre listas y arreglos.

- `struct node* findNodeInLevel(struct node** list, char character)`
  Dada una lista de nodos, retorna un puntero al nodo que tiene el caracter pasado por parámetro.
- `struct node* addSortedNewNodeInLevel(struct node** list, char character)`
  Dada una lista de nodos, agrega un nuevo nodo a la lista de forma ordenada. Este nodo llevará el caracter pasado por parámetro y el resto de sus datos en cero.
- `void deleteArrayOfWords(char** words, int wordsCount)`
  Dado un puntero a un arreglo de punteros a strings y el tamaño del arreglo. Se encarga de borrar una a una las strings y además borrar el arreglo.

Ejercicio 3

Implementar las siguientes funciones sobre el tipo de datos `keysPredict*`.

- `struct keysPredict* keysPredictNew()`
  Construye una nueva estructura `keysPredict` con todos sus campos en cero.
- `void keysPredictAddWord(struct keysPredict* kt, char* word)`
  Agrega una nueva palabra a la estructura `keysPredict`. Para esto debe ir recorriendo cada lista y en cada nivel agregar cada letra de la palabra en caso de ser necesario. Al final, en el último nodo, correspondiente a la última letra de la palabra, debera hacer una copia de la palabra y agregarla en el nodo. Además deberá marcar este nodo como final, indicando un `1` en `end`.
- `void keysPredictRemoveWord(struct keysPredict* kt, char* word)`
  Busca el nodo correspondiente a la última letra de la palabra a borrar y borra la palabra almacenada en dicho nodo. Adicionalmente marca el nodo con un `0` en `end`. Notar que los nodos de las letras que correspondian a la palabra no son borrados, sino que solo se borra la palabra del último nodo. Es decir, la estructura va a quedar con nodos que podrían no existir para las palabras que realmente tiene la estructura.
- `struct node* keysPredictFind(struct keysPredict* kt, char* word)`
  Busca en la estructura el nodo correspondiente a la palabra indicada, de no encontrarlo retorna un cero. Para esto debe recorrer la estructura según cada letra de la palabra. El nodo retornado será válido si está marcado en `end` con `1` y el puntero `word` tiene la palabra buscada.
- `char** keysPredictRun(struct keysPredict* kt, char* prefix, int* wordsCount)`
  Dadas las primeras letras de una palabra, busca todas las palabras que contenga el prefijo indicado. Para esto debe recorrer la estructura hasta alcanzar la última letra del prefijo. A partir de ese nodo, debe recontruir todas las palabras que se puedan armar con la subestructura restante. Debe retornar estas palabras en una estructura de tipo arreglo de punteros a string (`char**`) como valor de retorno de la función. La cantidad de elementos de este arreglo de

punteros estará dada por las palabras encontradas y deberá retornarce en el puntero `wordsCount` pasado como parámetro.

- `char** keysPredictListAll(struct keysPredict* kt, int* wordsCount)`
  Esta función debe retornar todas las palabras almacenadas en la estructura. El resultado de la función respetará el mismo comportamiento que la función `keysPredictRun`.
- `void keysPredictDelete(struct keysPredict* kt)`
  Borra la estructura `keysPredict` completa. Para esto debe borrar todos los nodos, incluyendo las palabras almacenadas en cada nodo de ser necesario. Además debe borrar el nodo raíz.
- `void keysPredictPrint(struct keysPredict* kt)`
  Imprime en pantalla la estructura `keysPredict`. Imprime cada nodo de cada lista, yendo desde los nodos en la raíz hasta los nodos más lejanos en cada lista. EL formato de impresión en pantalla es el siguiente:

```
--- Predict --- Keys: 12 Words: 5
[a]
 |    c
 |    |    t
 |    |    |    o
 |    |    |    |   [r]
 |    |    |    u
 |    |    |    |    a
 |    |    |    |    |   [r]
 s
 |    o
 |    |   [l]
 |    |    |   [a]
```

  En el ejemplo se puede ver una estructura con las palabras '`a`', '`actor`', '`actuar`', '`sol`' y '`sola`'. Notar que las letras entre corchetes indican que la letra es final de una palabra. Las jerarquías por nivel corresponden a cada columna de letras, mientras que una columna corresponde a una lista de letras de un nivel o sub nivel específico.

Para facilitar el desarrollo, las funciones `keysPredictNew` y `keysPredictPrint` vienen dadas.

Ejercicio 4

A continuación, se enumera un conjunto mínimo de casos de test que deben implementar dentro del archivo `main`:

- `strLen`
  1. String vacío.
  2. String de un carácter.
  3. String que incluya todos los caracteres alfanumericos.
- `strDup`
  1. String vacío.
  2. String de un carácter.
  3. String que incluya todos los caracteres alfanumericos.
- `keysPredict` casos chicos.
  1. Armar un diccionar con las palabras “alfajor”, “canoa”, “rinoceronte”, “casa” y “rino”.
  2. Borrar la palabra “canoa” y agregar la palabra “pato”.
  3. Predecir a partir “c”, “ca”, “casa” y “casas”.

- `keysPredict` casos grandes.

    1. Armar un diccionario con 100 palabras cualesquiera.
    2. Borrar la mitad de las palabras del diccionario.
    3. Predecir todas las combinaciones posibles de prefijos dos letras.

Se recomienda agregar todos los casos de test que gusten, teniendo en cuenta casos borde de borrado y escritura de datos.

Ejercicio 5

Utilizando cualquier herramienta de inteligencia artificial, ya sea `Gemini`, `Copilot`, `ChatGPT`, `Claude` o `LLaMa`, responder las siguiente preguntas:

- Indique el porcentaje aproximado de líneas de código del trabajo práctico fueron realizadas con asistencia de una IA.
- ¿Cómo verificaron que las sugerencias de la IA eran correctas?
- ¿Se enfrentaron a alguna dificultad al utilizar las herramientas de IA? ¿Cómo las resolvieron?
- ¿Consideran que el uso de la IA les ha permitido desarrollar habilidades de programación en C? ¿Por qué?

Nota: *Estas preguntas fueron generadas con la asistencia de la IA, respondiendo al prompt: "Supone que sos un docente universitario y estás haciendo un trabajo práctico de programación en C. Tu intención es que tus estudiantes utilicen las herramientas de IA para favorecer el aprendizaje y no para que estas les resuelvan el trabajo. Que preguntas se les podrías hacer al final del tp para determinar si utilizaron correctamente las herramientas de IA."*

Entregable

Para este trabajo práctico no deberán entregar un informe. Sin embargo, deben agregar comentarios en el código que expliquen su solución. No deben comentar qué es lo que hace cada una de las instrucciones sino cuáles son las ideas principales del código implementado y por qué resuelve cada uno de los problemas.
La entrega debe contar con el mismo contenido que fue dado para realizarlo más lo que ustedes hayan agregado, habiendo modificado **solamente** los archivos `utils.c` y `main.c`. Es requisito para aprobar entregar el código **correctamente comentado**.

Lista de archivos provistos:

- `Makefile`: Script para compilar el trabajo práctico. Se ejecuta por medio del comando '`make`'
- `main.c`: Archivo donde completar la solución de los casos de test pedidos.
- `utils.c`: Archivo donde completar todo el código a implementar.
- `utils.h`: Encabezados, definiciones de estructuras y funciones.
- `predict.c`: Programa que carga un diccionario y permite predecir palabras por línea de comando.
- `dicc.txt`: Diccionario base del programa `predict.c`.