Tarea N°2:

Introducción a C++

Joaquín Ignacio Ormazábal Acevedo
Departamento de Ingeniería Eléctrica, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile
20 de Octubre, 2024

***Abstract*—Este informe presenta la solución implementada para un par de problemas relacionados con la construcción de un diccionario basado en la estructura de datos Trie, en el lenguaje *C++*. Los problemas incluyeron la implementación de un diccionario de traducciones del inglés al español y una estructura para almacenar información asociada a números RUT. A través de la construcción y modificación del Trie, se exploraron técnicas de manejo de memoria dinámica y optimización de búsqueda. Los resultados muestran una implementación exitosa que permite agregar, buscar y eliminar entradas en el Trie, con un enfoque en la eficiencia y la correcta gestión de recursos. Además, se describen los desafíos encontrados durante la implementación y las soluciones adoptadas para asegurar la fiabilidad y funcionalidad del programa, haciendo énfasis en la importancia del manejo adecuado de la memoria dinámica en *C++*.**

I. Introduction

Los Tries son estructuras de datos ampliamente utilizadas en aplicaciones que requieren acceso rápido y eficiente a cadenas de caracteres, como diccionarios, sistemas de autocompletado, y análisis de texto. Un Trie es un árbol cuyas ramas representan caracteres individuals de una cadena. Cada nodo en el Trie representa un prefijo de una cadena, y las palabras se almacenan a lo largo de un camino desde la raíz hasta un nodo final que marca el final de una palabra. Esto permite búsquedas en tiempo lineal respecto al tamaño de la cadena, ya que cada letra se busca en un nivel del árbol. Por ejemplo, si se desea almacenar las palabras ”casa” y ”carro”, se tendría una raíz común para las letras ”c” y ”a”, y luego ramas que diferencian ”sa” de ”rro”. De esta forma, el Trie aprovecha las partes comunes de las palabras para reducir el espacio y tiempo de búsqueda. En esta tarea, se utilizaron Tries para implementar un diccionario de traducciones del inglés al español y para almacenar información asociada a RUTs chilenos, que incluyen datos personales. Este informe analiza las soluciones propuestas para las preguntas 1A, 1B, 2A y 2B, enfocándose en el diseño del Trie, el manejo de la memoria dinámica, las mejoras realizadas respecto a la eficiencia, y las diferencias entre las versiones de los problemas..

II. Metodología

A. Pregunta 1: Implementación del Diccionario de Traducciones (1A y 1B)

La primera parte de la tarea consistió en la implementación de un diccionario de traducciones que permitiera almacenar palabras en inglés y sus correspondientes traducciones al español. El objetivo era permitir el agregado, búsqueda y eliminación de palabras y sus traducciones de manera eficiente utilizando un Trie. Por lo tanto, se implementó esta estructura donde cada nodo representa una letra de la palabra. Este Trie permite agregar múltiples traducciones para una palabra, almacenándolas en un vector de cadenas. Además, para optimizar el acceso a las traducciones, se utilizó la función qsort para ordenarlas lexicográficamente.

El código se encuentra adjunto a esta tarea. A continuación explicaré algunos extractos relevantes del código:

a) Función `agregar_palabra`*:* Esta función implementa la inserción de una palabra y sus traducciones al Trie. Primero, se recorren los caracteres de la palabra para crear los nodos correspondientes, si es que no existen. Luego, al llegar al nodo final que representa la última letra de la palabra, se almacena la traducción en un vector. Si la palabra ya tiene traducciones almacenadas, se agrega la nueva traducción y luego se ordena el vector usando `qsort`. A continuación se muestra el código:

Código 1: Función agregar_palabra para insertar palabras y traducciones en el Trie.

```
agregar_palabra_begin
// agregar_palabra: Agregar palabra y su traducci n al
    trie
void Trie::agregar_palabra(const char* palabra, const char*
    traduccion) {
    TrieNode* nodo_actual = raiz;
    int idx;
    // Navego por el trie creando nodos seg n sea
    necesario
    for (size_t i = 0; i < strlen(palabra); ++i) {
        idx = palabra[i] - 'a'; // Obtener el  ndice  de la
     letra en el alfabeto
        if (nodo_actual->hijos[idx] == nullptr) {
            nodo_actual->hijos[idx] = new TrieNode();
        }
        nodo_actual = nodo_actual->hijos[idx]; // Mover el
    puntero al hijo siguiente
    }
    // Si hay menos de MAX_TRANSLATIONS traducciones, se
    asigna memoria a una nueva traducci n y se a ade al
    vector traducciones
    if (nodo_actual->traducciones.size() < MAX_TRANSLATIONS
    ) {
        char* nueva_traduccion = new char[strlen(traduccion
    ) + 1]; //
        strcpy(nueva_traduccion, traduccion); // Copiar la
    traducci n a la nueva traducci n
        nodo_actual->traducciones.push_back(
    nueva_traduccion); // Agregar la nueva traducci n al
    vector de traducciones

        ordenar_traducciones(nodo_actual->traducciones); //
     Ordenar las traducciones lexicogr ficamente
    } else {
        cout << "N mero m ximo de traducciones alcanzado
    para la palabra" << endl;
    }
}
// agregar_palabra_end
```

En este fragmento, `nodo_actual` se mueve a través del Trie creando nodos si no existen. Cuando se llega al final de la palabra, se inserta la traducción y se llama a `ordenar_traducciones` para mantener el vector en orden lexicográfico.

b) Función `buscar_palabra`: Esta función permite buscar una palabra y devolver todas sus traducciones almacenadas. Se realiza una búsqueda iterativa por cada letra de la palabra. Si en algún punto no se encuentra el nodo correspondiente a un caracter, la palabra no está en el Trie y se devuelve un mensaje indicando que no se encontró (None). Si se encuentra el modo final, se devuelven las traducciones almacenadas en ese nodo, concatenadas en un string separadas por comas. A continuación se muestra el código:

Código 2: Función buscar_palabra para recuperar las traducciones almacenadas en el Trie.

```
buscar_palabra_begin
// buscar_palabra: Buscar palabra en el trie
string Trie::buscar_palabra(const char* palabra) {
    TrieNode* nodo_actual = raiz; // Comenzar en la ra z
    int idx;

    for (size_t i = 0; i < strlen(palabra); ++i) {
        idx = palabra[i] - 'a'; //  ndice  de la letra en el alfabeto (misma l gica que en agregar_palabra)
        if (nodo_actual->hijos[idx] == nullptr) {
            return "None"; // Si no hay un nodo hijo en la posici n idx (no existe la palabra en el trie), retornar "None"
        }
        nodo_actual = nodo_actual->hijos[idx]; // Mover el puntero al hijo siguiente si existe
    }

    if (!nodo_actual->traducciones.empty()) { // Comparar si el nodo actual tiene traducciones
        string resultado; // resultado se utilizar  para almacenar todas las traducciones concatenadas
        for (size_t i = 0; i < nodo_actual->traducciones.size(); ++i) {
            if (i > 0) {
                resultado += ","; // Si el nodo tiene traducciones, concatenar en cadena separadas por comas
            }
            resultado += nodo_actual->traducciones[i]; // Agregar la traducci n al resultado
        }
        return resultado; // Retornar todas las traducciones
    } else {
        return "None";
    }

}
// buscar_palabra_end
```

En este fragmento, `nodo_actual` recorre los hijos del Trie según los caracteres de la palabra. Si se encuentra el nodo final, se concatenan las traducciones para devolverlas como una cadena.

c) Función `borrar_nodo`: Esta función se encarga de eliminar una palabra y todas sus traducciones del Trie. Primero, navega a través del Trie para encontrar el nodo que representa la última letra de la palabra. Luego, libera la memoria asignada para las traducciones y elimina los nodos innecesarios si no tienen otros hijos. Esto asegura que el Trie no mantenga referencias a datos eliminados y se optimiza el uso de la memoria. A continuación se muestra el código:

Código 3: Función borrar_nodo para eliminar un nodo del Trie junto todos sus hijos.

```
borrar_nodo_begin
// borrar_nodo: Borrar nodo y todos sus hijos
void Trie::borrar_nodo(TrieNode* nodo) {
    if (!nodo) return; // Si el nodo es nullptr, no hacer nada
    for (int i = 0; i < ALPHABET_SIZE; ++i) {
        if (nodo->hijos[i]) {
            borrar_nodo(nodo->hijos[i]); // Recorrer los hijos del nodo recursivamente
        }
    }
    if (nodo->palabra) {
        delete[] nodo->palabra; // Liberar la memoria de la palabra si no es nullptr
    }
    for (auto& trad : nodo->traducciones) {
        delete[] trad; // Recorrer las traducciones (como en el desctructor de TrieNode) y liberar la memoria
    }
    delete nodo;
}
// borrar_nodo_end
```

B. Pregunta 2: Implementación de un Diccionario de RUTs (2A y 2B)

La segunda parte de la tarea fue modificar el Trie para almacenar RUTs y asociar información personal como el nombre del usuario, la dirección y la fecha de nacimiento. Esto implicó aumentar la complejidad de los nodos del Trie para incluir información adicional y manejar adecuadamente la memoria dinámica asociada. Para ello, se modificó la estructura del Trie para trabajar con dígitos en lugar de letras, ya que los RUTs son valores numéricos. Cada nodo del Trie tiene punteros que representan los dígitos del 0 al 9. Además, cada nodo final almacena punteros para la información del usuario. Esto requirió una gestión cuidadosa de la memoria para evitar fugas y garantizar la correcta liberación de los recursos cuando se eliminan nodos.

El código se encuentra adjunto a esta tarea. A continuación explicaré algunos extractos relevantes del código:

a) Función `numero_no_deudor`: Esta función permite insertar un RUT y marcarlo como no deudor, además de almacenar información personal del usuario. Por ejemplo, al agregar el RUT "12345678", se crean nodos para cada dígito del RUT, y una vez alcanzado el nodo final, se asignan los valores de nombre, dirección y fecha de nacimiento. A continuación se presenta el código:

Código 4: Función numero_no_deudor para insertar un RUT y su información asociada en el Trie.

```
numero_no_deudor_begin
// Agrega un RUT al trie y lo marca como no deudor
void Trie::numero_no_deudor(const char* rut, const string& nombre, const string& direccion, const string& fecha_nacimiento) {
    TrieNode* nodo_actual = raiz;
    int idx;

    // Navegamos por el trie creando nodos seg n sea necesario
    for (size_t i = 0; i < strlen(rut); ++i) {
        idx = rut[i] - '0';  // Obtenemos el  ndice  basado en el d gito actual (0-9)
        if (nodo_actual->hijos[idx] == nullptr) {
            nodo_actual->hijos[idx] = new TrieNode();
        }
        nodo_actual = nodo_actual->hijos[idx];
    }

    // Al llegar al nodo final, almacenamos la informaci n del RUT
    if (nodo_actual->rut) {
```

```
        delete[] nodo_actual->rut;
    }
    nodo_actual->rut = new char[strlen(rut) + 1];
    strcpy(nodo_actual->rut, rut);
    nodo_actual->nombre = nombre;
    nodo_actual->direccion = direccion;
    nodo_actual->fecha_nacimiento = fecha_nacimiento;
    nodo_actual->deudor_status = DeudorStatus::NO_DEUDOR;
     // Marcamos el RUT como no deudor
}
// numero_no_deudor_end
```

En este fragmento, la memoria se asigna dinámicamente para almacenar la información personal, asegurando que cada RUT tenga sus atributos únicos.

b) Función `buscar_RUT`*:* Esta función permite buscar un RUT en el Trie y recuperar la información del usuario. Se navega a través de cada dígito del RUT y, si se encuentra el nodo final, se devuelven los datos almacenados. Si el RUT no está en el Trie, se devuelve un mensaje indicando que no se encontró. A continuación se muestra el código:

Código 5: Función buscar_RUT para recuperar información personal asociada a un RUT en el Trie.

```
buscar_RUT_begin
// Busca la informaci n del RUT en el trie
string Trie::buscar_RUT(const char* rut) {
    TrieNode* nodo_actual = raiz;
    int idx;

    // Navegamos por el trie buscando el RUT
    for (size_t i = 0; i < strlen(rut); ++i) {
        idx = rut[i] - '0';
        if (nodo_actual->hijos[idx] == nullptr) {
            return "RUT no encontrado";
        }
        nodo_actual = nodo_actual->hijos[idx];
    }

    // Si el nodo contiene la informaci n del RUT,
     devolvemos la informaci n
    if (nodo_actual->rut) {
        string resultado = "Nombre: " + nodo_actual->nombre
      + "\n";
        resultado += "Direccion: " + nodo_actual->direccion
      + "\n";
        resultado += "Fecha de Nacimiento: " + nodo_actual
     ->fecha_nacimiento + "\n";
        resultado += "Deudor: " + string(nodo_actual->
     deudor_status == DeudorStatus::DEUDOR ? "S " : "No");
        return resultado;
    } else {
        return "RUT no encontrado";
    }
}
// buscar_RUT_end
```

En este fragmento, se accede a los atributos del nodo final y se concatenan para formar el resultado que se devuelve al usuario.

c) Función `borrar_RUT`*:* Esta función busca y elimina un RUT del Trie. Primero, navega a través del Trie para encontrar el nodo que representa el RUT ingresado. Luego, se elimina el RUT y se libera la memoria asignada a los atributos asociados, tales como el nombre, la dirección y la fecha de nacimiento. Esto asegura que no queden datos residuales y que la memoria se gestione de manera eficiente. A continuación se presenta el código:

Código 6: Función borrar_RUT para eliminar un RUT del Trie y liberar la memoria asociada.

```
borrar_RUT_begin
// Busca y elimina un RUT del trie
void Trie::borrar_RUT(const char* rut) {
    TrieNode* nodo_actual = raiz;
    int idx;

    // Navegamos por el trie buscando el RUT
    for (size_t i = 0; i < strlen(rut); ++i) {
        idx = rut[i] - '0';
        if (nodo_actual->hijos[idx] == nullptr) {
            cout << "RUT no encontrado." << endl;
            return;
        }
        nodo_actual = nodo_actual->hijos[idx];
    }

    // Si el nodo contiene el RUT, lo eliminamos
    if (nodo_actual->rut) {
        delete[] nodo_actual->rut;
        nodo_actual->rut = nullptr;
        nodo_actual->nombre.clear();
        nodo_actual->direccion.clear();
        nodo_actual->fecha_nacimiento.clear();
        nodo_actual->deudor_status = DeudorStatus::
     NO_DEUDOR;
        cout << "RUT eliminado." << endl;
    } else {
        cout << "RUT no encontrado." << endl;
    }
}
// borrar_RUT_end
```

En este fragmento, el nodo se recorre hasta llegar al nodo final correspondiente al RUT y se eliminan todos los datos asociados, asegurando la liberación correcta de los recursos.

El manejo de memoria dinámica en la implementación del Trie para los RUTs fue fundamental para evitar fugas de memoria. Cada vez que se agregaba un nodo, se utilizaba `new` para asignar memoria para los datos personales. Del mismo modo, cuando un RUT era eliminado, se llamaba a `delete` para liberar cada atributo. La complejidad de esta parte del código reside en asegurarse de que cada nodo, incluyendo los atributos adicionales, sea correctamente eliminado sin afectar otros nodos del Trie.

III. Resultados

A. Resultados de la Pregunta 1: Diccionario de Traducciones (1A y 1B)

En la primera prueba del diccionario de traducciones, los resultados obtenidos fueron los esperados. Se agregaron varias palabras junto con sus traducciones y se verificó que la búsqueda de dichas palabras devolviera las traducciones en orden lexicográfico. Las palabras agregadas incluyeron ”like”, ”apple” y ”banana”, con sus respectivas traducciones. Luego, se realizó una segunda prueba que consistía en ingresar palabras y sus traducciones manualmente a través de la consola, como ”cry” con ”llorar”, ”pretty” con ”bella”, y ”fingertips” con ”huellas”. Posteriormente, se realizó una búsqueda para verificar la existencia de estas palabras y sus traducciones, lo cual funcionó correctamente.

En general, el comportamiento del programa para ambas pruebas se ajustó a las expectativas, mostrando las traducciones agregadas de manera correcta y devolviendo ”None” en caso de no encontrar una palabra específica.

B. Resultados de la Pregunta 2: Diccionario de RUTs (2A y 2B)

Para la implementación del diccionario de RUTs, se realizaron pruebas de inserción, búsqueda y eliminación de RUTs junto con la información personal correspondiente. En

la primera prueba, se agregaron tres RUTs (correspondientes a John Lennon, Elizabeth Woolridge Grant y Thom Yorke) con sus respectivas direcciones, fechas de nacimiento y estados de deudor. Los resultados de las búsquedas fueron exitosos, devolviendo la información correcta de cada uno de los RUTs ingresados.

Luego, se intentó buscar un RUT que no había sido ingresado, lo cual resultó en el mensaje ”RUT no encontrado”. Posteriormente, se realizó la eliminación del RUT de John Lennon, y nuevamente se verificó que la búsqueda de dicho RUT devolviera el mensaje esperado de ”RUT no encontrado”, lo cual demuestra que la operación de eliminación se llevó a cabo correctamente.

En conclusión, los resultados obtenidos en la prueba de la implementación del diccionario de RUTs fueron los esperados, mostrando el correcto manejo de inserción, búsqueda y eliminación, así como la adecuada liberación de memoria para evitar fugas.

IV. Análisis

En la primera parte de la tarea, la implementación del Trie permitió comprender los conceptos básicos de esta estructra y su aplicación para almacenar y recuperar cadenas de texto de forma eficiente. La eficiencia del Trie radica en su capacidad para reducir el tiempo de búsqueda a la longitud de la palabra, lo cual es especialmente útil para aplicaciones que requieren muchas búsquedas, como un diccionario. La segunda parte incrementó la complejidad de la estructura al incluir información adicional en cada nodo del Trie, lo que requirió un manejo más detallado de la memoria dinámica. El uso de new y delete para gestionar atributos como nombre, dirección y fecha nacimiento implicó un mayor riesgo de fugas de memoria si no se liberaban correctamente todos los recursos al eliminar un nodo. Esta complejidad adicional también hizo que fuera crucial implementar y verificar rigurosamente la función de eliminación para garantizar que se liberara toda la memoria asignada.

Además, se destacó la importancia de un enfoque meticuloso en el manejo de memoria, especialmente en una estructura como el Trie, donde los nodos pueden tener múltiples hijos y almacenar información adicional. En cada operación de eliminación, se debía asegurar que cada hijo y cada atributo fueran correctamente liberados, para evitar cualquier tipo de fuga de memoria que pudiera comprometer la eficiencia del sistema. La implementación también destacó cómo la capacidad de asociar información adicional a los nodos del Trie puede hacerlo extremadamente versátil, pero a su vez requiere un diseño cuidadoso para manejar correctamente todos los recursos asociados.

V. Conclusiones Generales

La implementación de un Trie para un diccionario de traducciones y posteriormente para almacenar información asociada a RUTs demostró la versatilidad de esta estructura para resolver problemas complejos de almacenamiento y búsqueda de cadenas. La correcta gestión de la memoria dinámica es fundamental en la implementación de estructuras como el Trie, especialmente cuando los nodos almacenan información adicional que debe ser manejada con new y delete. Durante la implementación, se puso de manifiesto que una gestión inadecuada de los recursos de memoria puede resultar en fugas de memoria, lo cual destaca la importancia de tener un enfoque riguroso en la liberación de recursos.

El desarrollo de esta tarea permitió afianzar conocimientos en estructuras de datos, manejo de memoria y optimización de búsquedas en *C++*. La versatilidad del Trie quedó demostrada tanto en su capacidad para almacenar palabras con múltiples traducciones, como en su adaptación para gestionar RUTs y asociar información detallada. Además, se evidenció que, aunque el uso de memoria dinámica incrementa la flexibilidad de la estructura, también añade una capa adicional de complejidad que debe ser cuidadosamente gestionada. En aplicaciones reales, el uso de Tries seguramente podría extenderse a otros dominios que requieren búsquedas rápidas y eficientes, como sistemas de autenticación, almacenamiento de configuraciones, etc. Esta tarea sirvió como un ejercicio valioso para introducirse en el lenguaje *C++*, y explorar tanto los beneficios como los desafíos de usar estructuras de datos avanzadas en contextos donde el manejo eficiente de la memoria y la velocidad de acceso son cruciales.