Tarea N°2:

Arboles Trie

Tomás Maximiliano Barros Barrios
<Departamento de Ingeniería Eléctrica>, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile
17, Octubre, 2024

I. Introducción

El objetivo del trabajo es crear un diccionario ingles-español, y crear un sistema que permita ver la información personal de alguien a través del rut, todo utilizando arboles Trie en C++.

II. Diccionario Ingles-Español

A. Parte A

La siguiente sección tiene como objetivo implementar una clase tipo árbol Trie, para ser utilizada en la creación de un diccionario Ingles-Español. La estructura de la clase consiste de nodos, donde cada nodo almacena una o más traducciones de la palabra en el nodo y, además, almacena un arreglo de punteros a sus nodos hijos. Cada palabra puede tener una o más traducciones al español. La clase debe tener funciones que permitan agregar palabras y su traducción, buscar palabras y que entregue la traducción, un constructor de clase que asigne memoria para nuevos nodos, y un destructor de clase que elimine la memoria de un nodo y todos sus hijos. La estructura del arbol Trie es la siguiente:

```cpp
#include <string>
const int letras = 26;

struct ArbolTrie {
    struct ArbolTrie* HijosNodo[letras];
    bool FinPalabra;
    std::string traducciones;
    ArbolTrie();
};
```

Código 1: Código estructura clase diccionario en C++

Donde se utilizó 'std::string', pues posee más métodos y facilidades de uso que 'const char*' ([1] y [2]). La estructura del Trie consiste en un nodo con 26 punteros para sus hijos (cada uno corresponde a 1 de las 26 letras del alfabeto), un bool llamado FinPalabra, el cual es verdadero si el nodo representa el fin de una palabra guardada en el diccionario previamente, y falso si no. Y un 'std::string' para almacenar las traducciones de las palabras. Finalmente está el constructor para inicializar el arbol Trie. El constructor tiene la siguiente forma:

```cpp
ArbolTrie::ArbolTrie() : FinPalabra(false),
    traducciones("") {
    for (int i = 0; i < letras; i++) {
        HijosNodo[i] = nullptr;
    }
}
```

Código 2: Código estructura clase diccionario en C++

Donde FinPalabra comienza como 'false', pues aun no se agrega ninguna palabra, y traducciones comienza como un arreglo vacío para su uso posterior. Además, cada nodo hijo inicializa con puntero nullptr.

Una vez hecha la estructura, la clase Trie tiene las siguientes funciones:

```cpp
class Trie {
public:
    Trie();
    ~Trie();

    void agregar_palabra(const std::string& palabra,
     const std::string& traduccion);
    std::string buscar_palabra(const std::string&
    palabra);

private:
    ArbolTrie* raiz;

    void borrar_nodo(struct ArbolTrie* nodo);  //
    Libera memoria
};
```

Código 3: Código estructura clase diccionario en C++

Donde Trie() es un constructor que asigna memoria utilizando new() para un nuevo nodo y Trie() es un destructor que elimina a un nodo y a sus hijos de la memoria. Por otro lado, 'agregar_palabra' es una función tipo void (es decir, no retorna nada), cuyo objetivo es incluir a una palabra y su traducción. Si la palabra ya estaba en el diccionario, entonces incluye otra traducción para esta. La función 'buscar_palabra' es del tipo 'std::string', pues retorna la traducción de la palabra buscada (que es tipo std::string). Estas 2 funciones son para uso del usuario, y por tanto están en la sección 'public'. Para liberar memoria, está la función 'borrar_nodo' en la parte 'private', la cual utilizará delete() de forma recursiva para eliminar al nodo y todos sus nodos.
A continuación se verán las funciones que contiene la clase anterior individualmente.

1) ***Trie****:* La función Trie() utiliza a new() para asignar memoria a un nuevo nodo.

```cpp
Trie::Trie() {
    raiz = new ArbolTrie();
}
```

Código 4: Código constructor C++

2) ***- Trie****:* La funcion -Trie() llama a la función borrar_nodo, la cual se verá más adelante.

```cpp
Trie::~Trie() {
    borrar_nodo(raiz);
```

```
}
```

Código 5: Código destructor C++

*3) **agregar_palabra**:* Esta función tiene como argumentos 'std::string', que serán la palabra a agregar y su traducción. El código de la función es mostrado en el código 6.

- **Linea 2:** Inicializa puntero en la raíz del Trie para navegarlo.
- **Linea 3:** Bucle for para recorrer todos los caracteres de 'palabra' a lo largo del Trie. Si no hay caracter, se agregará uno nuevo.
- **Linea 4:** índice para acceder a la posición correspondiente del nodo hijo. Utilizando el valor ASCII obtenido en [3], si tenemos 26 letras, entonces tenemos 26 posiciones correspondientes para cada letra en los nodos hijo. Donde la posición 0 corresponde al caracter 'a', la posición 1 corresponde a 'b' y así. Entonces, si definimos 'indice' = c - 'a', si 'c' corresponde al caracter 'a', estamos obteniendo como indice el valor 0, y podemos acceder al caracter correspondiente (en este ejemplo 'a').
- **Linea 5,6,7:** Estas lineas de código dicen que, si no existe un nodo que contenga al caracter 'c' de 'palabra' (es decir, si el nodo es nullptr), entonces se genera un nuevo nodo Trie para continuar ingresando a 'palabra' al diccionario.
- **Linea 8,9:** Esta linea dice que en caso de existir el nodo que contenga al caracter 'c' de 'palabra', entonces se avanza en el nodo para seguir recorriendo el Trie. Finalmente la linea 9 cierra el ciclo for.
- **Linea 10:** Esta linea ocurre luego del ciclo for (es decir, cuando ya se llego al final de la palabra). Entonces se marca FinPalabra como True, para indicar que ya estamos en una palabra con definición en ese nodo.
- **Linea 11,12,13:** Se verifica si 'traducciones' está vació. En caso de no estar vacío (es decir, la palabra ya tenia traducción), entonces se agrega una coma y un espacio, para seguir agregando traducciones.
- **Linea 14:** Esta ultima linea agrega la traducción de la palabra agregada a 'traducciones', terminando el código,

```
void Trie::agregar_palabra(const std::string&
    palabra, const std::string& traduccion) {
    ArbolTrie* nodo = raiz;
    for (char c : palabra) {
        int index = c - 'a';
        if (!nodo->HijosNodo[index]) {
            nodo->HijosNodo[index] = new ArbolTrie()
    ;
        }
        nodo = nodo->HijosNodo[index];
    }
    nodo->FinPalabra= true;
    if (!nodo->traducciones.empty()) {
        nodo->traducciones += ",";
    }
    nodo->traducciones += traduccion;
}
```

Código 6: Código función para agregar palabras C++

*4) **buscar_palabra**:* Esta función busca recibe como argumento un 'std::string' correspondiente a una palabra. Si la palabra está guardada en el diccionario, entrega sus traducciones, en caso contrario, entrega el mensaje de que no se encontró la traducción. El código es mostrado en el código 7.

- **Linea 2:** Inicializa la búsqueda en la raíz del Trie.
- **Linea 3:** Bucle for para recorrer la palabra buscada en el Trie.
- **Linea 4:** Accede a la posición correspondiente del carácter en el nodo hijo.
- **Linea 5,6,7,8,9:** En caso de no encontrar los caracteres en el diccionario, entonces la palabra no está en este, y retorna un mensaje diciendo que no tiene traducción. En caso de encontrar el caracter, entonces sigue recorriendo el árbol Trie. Finalmente la linea 9 termina el bucle for (pues ya se recorrió toda la palabra).
- **Linea 10,11,12:** Este código indica que, si el nodo es distinto de nullptr (es decir, existe algo en dicho nodo) y que FinPalabra es True (es decir, dicho nodo representa una palabra completa desde la raíz hasta el nodo), entonces retorna la traducción asociada.
- **Linea 13,14:** Si ocurre el caso contrario a lo visto en las lineas 10, 11 y 12, entonces no hay traducción para esa palabra o esa palabra no está en el diccionario.

```
std::string Trie::buscar_palabra(const std::string&
    palabra) {
    ArbolTrie* nodo = raiz;
    for (char c : palabra) {
        int index = c - 'a';
        if (!nodo->HijosNodo[index]) {
            return "No se encontro traducci n";
        }
        nodo = nodo->HijosNodo[index];
    }
    if (nodo != nullptr and nodo->FinPalabra) {
        return nodo->traducciones;
    }
    return "No se encontro traducci n";
}
```

Código 7: Código función para buscar palabras en C++

*5) **borrar_palabra**:* Esta función toma como argumento un puntero a algún nodo del arbol Trie, liberando la memoria asociada a este nodo y todos sus descendientes de forma recursiva, utilizando delete(). El código es mostrado en el código 8.

- **Linea 2:** Inicia bucle for, que recorre los 26 hijos del nodo a eliminar.
- **Linea 3,4,5,6:** Si existe un elemento en el nodo hijo, entonces se llama a la función nuevamente para el nodo hijo, para eliminar todo lo que tenga ese nodo en cuestión.
- **Linea 7:** Una vez recorrido todo el nodo y sus descendientes, utiliza delete para liberar la memoria de ese nodo, terminando la función. La linea 5 termina el if, mientras que la linea 6 termina el bucle for.

```
void Trie::borrar_nodo(struct ArbolTrie* nodo) {
    for (int i = 0; i < letras; i++) {
        if (nodo->HijosNodo[i]) {
            borrar_nodo(nodo->HijosNodo[i]);
        }
    }
    delete nodo;
}
```

Código 8: Código estructura en C++

Una vez hecho esto, falta probar el funcionamiento del código. Para esto, se programa un mainP1A() que utilizará las funciones agregar_palabra y buscar_palabra. La prueba consiste en agregar 4 palabras y sus traducciones, donde a una de esas palabras se le agregará una traducción adicional, para comprobar el funcionamiento de múltiples traducciones para una sola palabra. Finalmente se llama a la función buscar_palabra, donde se buscarán las 4 palabras agregadas y una palabra adicional no agregada previamente, donde se busca comprobar el funcionamiento en caso de que exista la palabra y en caso de que no. El código 9 muestra a mainP1A().

```cpp
#include "diccionario.h"
#include <iostream>
int main() {
    Trie diccionario;

    // Insertar palabras y sus traducciones
    diccionario.agregar_palabra("like", "amar");
    diccionario.agregar_palabra("like", "gustar");
    diccionario.agregar_palabra("egg", "huevo");
    diccionario.agregar_palabra("steak", "filete");
    diccionario.agregar_palabra("cheese", "queso");

    // Probar las b squedas
    std::cout << "Traducci n de 'like': " <<
    diccionario.buscar_palabra("like") << std::endl;
    std::cout << "Traducci n de 'egg': " <<
    diccionario.buscar_palabra("egg") << std::endl;
    std::cout << "Traducci n de 'steak': " <<
    diccionario.buscar_palabra("steak") << std::endl
    ;
    std::cout << "Traducci n de 'cheese': " <<
    diccionario.buscar_palabra("cheese") << std::
    endl;
    std::cout << "Traduccion de 'lion': " <<
    diccionario.buscar_palabra("lion") << std::endl;
    return 0;
}
```

Código 9: Código de mainP1A() para comprobar el programa en C++

Para ejecutar el código, es necesario ir a powershell y utilizar los comandos (de manera individual) mostrados en 10.

```
cd C:\<Lugar donde se guardaron los codigos
    anteriores>

g++ -o Diccionario mainP1A.cpp diccionario.cpp

.\Diccionario
```

Código 10: Comandos para ejecutar código mainP1A() en Powershell

Donde la linea 1 es para abrir la carpeta donde se guardaron los códigos de la clase y main(). La linea 3 es para compilar el programa en un archivo.exe, y finalmente la linea 5 es para ejecutar el programa en la terminal. Hecho esto, los resultados son los mostrados en la figura 1, donde se evidencia el correcto funcionamiento de 'agregar_palabras' y 'buscar_palabras', pues la primera es capaz de agregar más de una traducción a la misma palabra, mientras que la segunda entrega las traducciones, y en caso de no haber una, entrega el mensaje de 'No se encontró traducción'.

```
a_2> .\Diccionario
Traducci├│n de 'like': amar, gustar
Traducci├│n de 'egg': huevo
Traducci├│n de 'steak': filete
Traducci├│n de 'cheese': queso
Traduccion de 'lion': No se encontro traducci├│n
```

Fig. 1: Resultado de ejecución de mainP1A() en powershell

B. *Parte B*

El objetivo de esta sección es añadir un orden lexicográfico a las traducciones agregadas. Para esto, se modificará la función agregar_palabra para que ordene automáticamente las traducciones agregadas utilizando quicksort. [4] muestra que C++ tiene quicksort incluido de la forma 'std::sort'. Por tanto se utilizará este para el ordenamiento.

Es importante mencionar que 'traducciones' es una cadena de strings 'std::string', donde las traducciones de la palabra van separadas por comas ','. Por lo tanto, para ordenar las traducciones de manera lexicográfica, la idea es programar una función que reciba como argumento un string con todas las traducciones separadas por comas. Esta función reconocerá las comas en el string y armará un vector de strings con las traducciones separadas. Utilizando este vector, simplemente queda aplicar std::sort a este y se obtendrá un vector ordenado de strings. Finalmente se juntan nuevamente todas las traducciones en un solo string y se retorna. Esta función irá en la sección 'private' del código, y será llamada dentro de la función 'agregar_palabra' para que cada vez que se utilice, ordene automáticamente todas las traducciones agregadas. El código 11 muestra como se agregó la nueva función a 'private'. El código 12 muestra como se programó esta nueva función, y el código 13 muestra como se implementó en la función 'agregar_palabra'.

```cpp
class Trie {
public:
    Trie();
    ~Trie();

    void agregar_palabra(const std::string& palabra,
     const std::string& traduccion);
    std::string buscar_palabra(const std::string&
    palabra);

private:
    ArbolTrie* raiz;

    void borrar_nodo(struct ArbolTrie* nodo); //
    Libera memoria
    std::string ordenatelordeno(const std::string&
    traducciones);  // Funcion para ordenar
};
```

Código 11: Código estructura clase diccionario en C++

```cpp
std::string Trie::ordenatelordeno(const std::string&
     traducciones) {

    std::vector<std::string> vectorTraducciones;
    std::stringstream Traducciones(traducciones);
    std::string guardado;

    while (getline(Traducciones, guardado, ',')) {
        if (!guardado.empty()) {
            vectorTraducciones.push_back(guardado);
```

```
        }
    }
    std::sort(vectorTraducciones.begin(),
    vectorTraducciones.end());
    std::string traduccionesOrdenadas;
    for (int i = 0; i < vectorTraducciones.size();
    ++i) {
        if (i > 0) traduccionesOrdenadas += ",";
        traduccionesOrdenadas += vectorTraducciones[
    i];
    }
    return traduccionesOrdenadas;
}
```

Código 12: Código de ordenamiento lexicográfico C++

```
void Trie::agregar_palabra(const std::string&
    palabra, const std::string& traduccion) {
    ArbolTrie* nodo = raiz;
    for (char c : palabra) {
        int index = c - 'a';
        if (!nodo->HijosNodo[index]) {
            nodo->HijosNodo[index] = new ArbolTrie()
    ;
        }
        nodo = nodo->HijosNodo[index];
    }
    nodo->FinPalabra= true;

    if (!nodo->traducciones.empty()) {
        nodo->traducciones += ",";
    }
    nodo->traducciones += traduccion;
    nodo->traducciones = ordenatelordeno(nodo->
    traducciones); //Aqui se llama a la funcion
    nueva
}
```

Código 13: Código agregar_palabra con ordenamiento C++

A continuación se realizará una explicación de la función que ordena las traducciones mostrada en código 12.

1) ***ordenatelordeno****:*

- **Linea 3:** Crea un vector de strings llamado 'vectorTraducciones' para ir almacenando las traducciones que serán separadas.
- **Linea 4:** stringstream facilita la extracción de los elementos, pues permite el uso de la función getline(), que tiene un caracter delimitador que puede ser especificado (visto en [5] y [6]). En este caso, se utilizará la coma ',' para ir separando los elementos del string.
- **Linea 5:** Crea un string 'guardado' cuyo objetivo es almacenar temporalmente las traducciones extraídas para luego colocarla en el vector. De esta forma, se extrae la traducción y se guarda temporalmente en 'guardado', para luego guardarse en el vector de traducciones. Luego se redefine nuevamente el string para extraer la siguiente traducción.
- **Linea 7,8,9:** Ejecuta ciclo while para la función getline(), con argumento delimitador ','. De esta forma, se recorrerá el string Traducciones hasta cada coma encontrada y se extraerá en 'guardado'. Luego, si 'guardado' es distinto de vacío, entonces se añade la traducción al vector de traducciones con push_back().
- **Linea 12:** Aplica quicksort al vector de traducciones, ordenándolo.
- **Linea 13:** Crea un nuevo string llamado 'traduccionesOrdenadas', cuyo proposito es ir recibiendo las traducciones del vector y concatenandolas en un solo string, separandolas por comas.
- **Linea 14,15,16:** Ciclo for cuyo objetivo es concatenar todas las traducciones nuevamente en un solo string separadas por comas. La linea 14 define que el bucle for será hasta recorrer todo el vector de traducciones. La linea 15 indica que si no es el primer elemento del vector (i¿0), entonces se añade una coma para separarlo. La linea 16 finalmente añade el elemento i del vector al string concatenado 'traduccionesOrdenadas'.
- **Linea 18:** Se retorna un string de todas las traducciones ordenadas y separadas por comas.

Para las pruebas de este código, se utiliza el código 14. Este código agrega la palabra 'like' y 3 definiciones desordenadas lexicográficamente. De esta forma, si al utilizar buscar_palabra se retornan las traducciones ordenadas, entonces se consiguió el objetivo. Para ejecutarlo, basta con ejecutar el código 15, donde solo se cambia mainP1A.cpp por mainP1B.cpp a diferencia de la primera parte.

```
#include "diccionario.h"
#include <iostream>

int main() {
    Trie diccionario;

    // Insertar palabras y sus traducciones
    diccionario.agregar_palabra("like", "gustar");
    diccionario.agregar_palabra("like", "amar");
    diccionario.agregar_palabra("like", "como");

    // Probar las b squedas
    std::cout << "Traducciones de 'like': " <<
    diccionario.buscar_palabra("like") << std::endl;
    return 0;
}
```

Código 14: Código main() para testeo en C++

```
cd C:\<Lugar donde se guardaron los codigos
    anteriores>

g++ -o Diccionario mainP1B.cpp diccionario.cpp

.\Diccionario
```

Código 15: Comandos para ejecutar código mainP1B() en Powershell

Los resultados son mostrados en la figura 2, donde se observa el correcto funcionamiento del código.

```
a_2> .\Diccionario
Traducciones de 'like': amar,como,gustar
```

Fig. 2: Resultado de ejecución de main() en powershell

III. Lookup inverso

La siguiente sección tiene como propósito reutilizar y modificar el código anterior para que el Trie implemente ruts con la información de la persona en cuestión. Para esto, se modifica la cantidad de hijos por nodo a 11 (números del 0 al 9, más el K), además de agregar strings que estén asociados a la información personal del usuario.

A. Parte A

El código 16 muestra la estructura de la clase rut. Las lineas 7, 8 y 9 definen la estructura 'enum' para definir si un usuario tiene deuda o no. Las lineas 15, 16, 17 y 18 definen las nuevas variables que deberán acompañar al rut (nombre, dirección, fecha de nacimiento, deudor/no deudor). La linea 28 modifica la función de agregar_palabra para el caso de los ruts, donde ahora tiene más argumentos, mientras que la linea 30 agrega la nueva función para borrar ruts. La linea 31 agrega la función para calificar a un rut como no deudor y finalmente la linea 37 agrega una función cuyo objetivo es ser utilizada en borrar rut para comprobar si los hijos de un nodo son vacíos (será útil decidir si borrar por completo el nodo o solo borrar información asociada).

```
1  #ifndef Tarea2Tomy
2  #define Tarea2Tomy
3
4  #include <string>
5  const int num = 11;  // Rut tiene numeros del 0 al 9
       y k => 11
6
7  enum Deudoroso {
8      No_deudor,
9      Deudor
10 };
11
12 struct ArbolTrie {
13     struct ArbolTrie* HijosNodo[num]; // Puntero a
       los hijos (11 posibilidades)
14     bool FinRut;// Indica si es el final del rut (
       verdadero) o no (falso)
15     Deudoroso deudor; // Indica si es deudor o no
16     std::string name; // Nombre del usuario
17     std::string dir; // Direccion del usuario (la
       casa supongo)
18     std::string birthdate; // Fecha de nacimiento
19
20     ArbolTrie();  // Constructor para inicializar el
        nodo
21 };
22
23 class Trie {
24 public:
25     Trie();
26     ~Trie();
27
28     void agregar_rut(const std::string& rut, const
       std::string& name, const std::string& direccion
       ,, const std::string& birthdate);
29     std::string buscar_rut(const std::string& rut);
30     void borrar_rut(const std::string& rut);
31     void numero_no_deudor(const std::string& rut);
32
33 private:
34     ArbolTrie* raiz;
35
36     void borrar_nodo(struct ArbolTrie* nodo);
37     bool nodo_vacio(struct ArbolTrie* nodo);
38 };
39
40 #endif
```

Código 16: Código estructura clase rut C++

Una vez vista la estructura, se procederá a ver las funciones una por una al igual que en la sección anterior. Para las funciones modificadas, solo se mencionarán y explicarán las lineas que fueron cambiadas, mientras que las funciones que no fueron cambiadas no serán mencionadas.

1) ***Estructura****:* El código 17 muestra la inicialización de la clase. Se inicializa la clase con los nombres, direcciones y fechas de nacimiento como vacías. Además se asigna a todos los usuarios como deudores (para darle sentido a la función no deudor, es necesario que todos sean deudores, pues si este no es el caso, entonces carecería de sentido definir esta función en primer lugar).

```
1 ArbolTrie::ArbolTrie() : FinRut(false), deudor(
      Deudor), name(""), dir(""), birthdate("") {
2     for (int i = 0; i < num; i++) {
3         HijosNodo[i] = nullptr;
4     }
5 }
```

Código 17: Código estructura clase rut C++

2) ***agregar_rut****:*

- **Lineas 2 a 8:** Se cambia como se trabaja con el recorrido del árbol. Ahora se tienen números del 0 al 9, además de las letras 'k' o 'K' al final del rut. Para trabajar esto, el índice toma el valor de 10 si es 'k' o 'K' (es decir, estas letras son el ultimo hijo del nodo, lineas 4 y 5), o toma los valores de c - '0' (linea 7), donde se sigue la misma lógica que en la función definida para el diccionario.
- **Lineas 18 a 30:** Estas lineas de código guardan la información del usuario. En caso de que ya haya información asociada al nodo (linea 18), a diferencia del diccionario donde se acumulaban las traducciones, aquí se elimina la información (se hace nuevamente un string vacío, linea 19), y se procede a guardar la información nueva del usuario (linea 21). Este procedimiento se realiza para el nombre, la dirección y la fecha de nacimiento del usuario.

```
1     ArbolTrie* nodo = raiz;
2     for (char c : rut) {
3         int index;
4         if (c == 'k' or c == 'K') {
5             index = 10;
6         } else {
7             index = c - '0';
8         }
9
10        if (!nodo->HijosNodo[index]) {
11            nodo->HijosNodo[index] = new ArbolTrie()
      ;
12        }
13        nodo = nodo->HijosNodo[index];
14    }
15
16    nodo->FinRut= true;
17
18    if (!nodo->name.empty()) {
19        nodo->name = "";
20    }
21    nodo->name += name;
22    if (!nodo->dir.empty()) {
23        nodo->dir = "";
24    }
25    nodo->dir += direccion;
```

```
    if (!nodo->birthdate.empty()) {
        nodo->birthdate = "";
    }
    nodo->birthdate += birthdate;
}
```

Código 18: Código función agregar_rut C++

3) ***buscar_rut****:*

- **Lineas 2 a 8:** Definición del índice para recorrer el árbol Trie.
- **Lineas 16 a 23:** Si se encuentra al usuario, se define un nuevo string llamado 'info' (linea 16), el cual tiene como objetivo recopilar la información del usuario y entrégala. El string 'info' comienza con la información del nombre, además del separador '/n', el cual separará la información. Las lineas 17 y 18 agregan la información de la dirección y la fecha de nacimiento. La linea 19 define un nuevo string llamado 'deuda', cuyo objetivo es tomar la información del 'enum' asociado a la deuda y convertirla en string. La linea 20 agrega al string 'deuda' a 'info', y finalmente la linea 21 retorna la información del usuario.

```
1   ArbolTrie* nodo = raiz;
2   for (char c : rut) {
3       int index;
4       if (c == 'k' or c == 'K') {
5           index = 10;
6       } else {
7           index = c - '0';
8       }
9
10      if (!nodo->HijosNodo[index]) {
11          return "No se encontro usuario asociado
    al RUT";
12      }
13      nodo = nodo->HijosNodo[index];
14  }
15  if (nodo != nullptr and nodo->FinRut) {
16      std::string info = "Nombre: " + nodo->name +
     "\n";
17      info += "Direcci n: " + nodo->dir + "\n";
18      info += "Fecha de Nacimiento: " + nodo->
    birthdate + "\n";
19      std::string deuda = (nodo->deudor ==
    No_deudor) ? "No deudor" : "Deudor";
20      info += "Estado de deuda: " + deuda + "\n";
21      return info;
22  }
23  return "No se encontro usuario asociado al RUT";
24 }
```

Código 19: Código función buscar_rut C++

4) ***borrar_rut****:*

- **Lineas 2 a 8:** Definición del índice para recorrer el árbol Trie.
- **Lineas 16 a 26:** Si el usuario fue encontrado (linea 16), entonces se procede a comprobar si los hijos del nodo son vacíos o no. Si son vacíos (linea 17), entonces se procede a borrar la memoria del nodo (linea 18) y se asigna 'FinRut' como falso (linea 19), pues ya no hay usuario asociado a ese rut. En caso de que los nodos hijos no sean vacíos, entonces se procede a eliminar por completo la información del usuario y se deja el nodo con el formato de inicialización por defecto (lineas 21 a 24), pero no se elimina, pues que los hijos del nodo sean no vacíos indican que hay otro usuario asociado a un rut similar o de valor superior al eliminado.

```
1   ArbolTrie* nodo = raiz;
2   for (char c : rut) {
3       int index;
4       if (c == 'k' or c == 'K') {
5           index = 10;
6       } else {
7           index = c - '0';
8       }
9       if (!nodo->HijosNodo[index]) {
10          std::cout << "No se encontro usuario
    asociado al RUT";
11          return;
12      }
13      nodo = nodo->HijosNodo[index];
14  }
15
16  if (nodo != nullptr and nodo->FinRut) {
17      if (nodo_vacio(nodo) == true) {
18          borrar_nodo(nodo);
19          nodo->FinRut = false;
20      } else {
21          nodo->name="";
22          nodo->dir="";
23          nodo->birthdate="";
24          nodo->FinRut = false;
25      }
26  } else {
27      std::cout << "No se encontro usuario
    asociado al RUT";
28      return;
29  }
30 }
```

Código 20: Código función borrar_rut C++

5) ***numero_no_deudor****:*

- **Lineas 2 a 8:** Definición del índice para recorrer el árbol Trie.
- **Lineas 10 a 13:** Si el usuario no fue encontrado (linea 10), se procede a agregar al usuario sin información de su nombre, dirección o fecha de nacimiento (linea 11). Posteriormente se llama nuevamente a la función numero_no_deudor (linea 12), pues ahora encontrará al usuario y será capaz de calificarlo como 'no deudor'. Luego de llamar a numero_no_deudor, se procede a finalizar la función ahí con un return (linea 13) para no caer en loops.
- **Linea 17:** En caso de ser encontrado el usuario, se cambia su estado de 'deudor' a 'no deudor'.

```
1   ArbolTrie* nodo = raiz;
2
3   for (char c : rut) {
4       int index;
5       if (c == 'k' or c == 'K') {
6           index = 10;
7       } else {
8           index = c - '0';
9       }
10      if (!nodo->HijosNodo[index]) {
11          agregar_rut(rut, "", "", "");  // Agrega
     el RUT si no existe (no hay datos del usuario,
    asi que se agerga vacio)
12          numero_no_deudor(rut);  // RUT agregado
13          return;  // Sale del ciclo porque el
    trie ahora contiene el RUT completo
14      }
15      nodo = nodo->HijosNodo[index];
16  }
```

```
    nodo->deudor=No_deudor;
}
```

Código 21: Código función numero_no_deudor C++

*6) **nodo_vacio**:*

- **Linea 1:** Inicia ciclo for para recorrer a todos los hijos del nodo.
- **Linea 2,3:** Si alguno de los nodos hijos presenta elementos (linea 2), retorna falso, indicando que no es nodo vacío.
- **Linea 4:** Si no presenta elementos, retorna true, indicando que es un nodo sin hijos.

```
    for (int i = 0; i < num; i++)
        if (nodo->HijosNodo[i])
            return false;
    return true;
}
```

Código 22: Código función nodo_vacio C++

Finalmente, el código para las pruebas será el mostrado en el código 23. Para ejecutar el código, basta con utilizar lo mostrado en el código 24. Las pruebas consisten en agregar 3 usuarios distintos, para luego buscar a los 3 usuarios, además de buscar a un usuario no existente. De esta manera, comprobamos el funcionamiento de agregar_rut y buscar_rut, además del comportamiento de esta ultima en caso de no encontrar al usuario. Posteriormente se comprueba el funcionamiento de numero_no_deudor y de borrar_rut, además del comportamiento de la primera en caso de no haber agregado al usuario antes.

```
#include "rut.h"
#include <iostream>
int main() {
    Trie rut;

    // Insertar rut e info
    rut.agregar_rut("208168320", "Tomas Barros","
    Padre Hurtado","17/10/2001");
    rut.agregar_rut("20816832k", "Lionel Messi","
    Miami","24/06/1987");
    rut.agregar_rut("208168321", "Estoy odiando","C
    ++","20/10/2024");

    // Probar las b squedas
    std::cout << "Pruebas para agregar y buscar rut:
     \n"<< std::endl;
    std::cout << "RUT: 20.816.832-0: " << rut.
    buscar_rut("208168320") << std::endl;
    std::cout << "RUT: 20.816.832-k: " << rut.
    buscar_rut("20816832k") << std::endl;
    std::cout << "RUT: 20.816.832-1: " << rut.
    buscar_rut("208168321") << std::endl;
    std::cout << "RUT: 20.816.832-2 (no agregado):"
    << rut.buscar_rut("208168322") << std::endl;

    std::cout << " " << std::endl;
    std::cout << " " << std::endl;

    std::cout << "Pruebas para deuda y borrar (se
    usa un rut agregado yel rut no agregado:) \n"
    << std::endl;
    rut.numero_no_deudor("208168320");
    rut.numero_no_deudor("208168322");
    rut.borrar_rut("208168321");
    std::cout << "Una vez usada la funcion, se llama
     a buscar rut nuevamente) \n" << std::endl;
    std::cout << "RUT: 20.816.832-0: " << rut.
    buscar_rut("208168320") << std::endl;
    std::cout << "RUT: 20.816.832-k: " << rut.
    buscar_rut("20816832k") << std::endl;
    std::cout << "RUT: 20.816.832-1 (usuario borrado
    ): " << rut.buscar_rut("208168321") << std::endl
    ;
    std::cout << "RUT: 20.816.832-2 (agregado con
    funcion numero_no_deudor):" << rut.buscar_rut("
    208168322") << std::endl;

    return 0;
}
```

Código 23: Código de pruebas para la clase rut C++

```
label=codigo-23]
cd C:\<Lugar donde se guardaron los codigos
    anteriores>

g++ -o Rut mainP2A.cpp rut.cpp

.\Rut
```

Código 24: Código de ejecución para pruebas de la clase rut C++

Los resultados muestran el correcto funcionamiento de todas las funciones, donde 3 muestra que agregar_rut agrega correctamente a los usuarios (dejándolos a todos como deudores, por lo discutido en la estructura del código 17), y buscar_rut es capaz de entregar correctamente la información de los usuarios agregados, además del mensaje en caso de no encontrar al usuario. Por otro lado, 4 muestra que numero_no_deudor es capaz de calificar a usuarios agregados y no agregados como no deudores, mientras que borrar_rut es capaz de borrar al usuario correctamente (buscar_usuario muestra al rut 20.816.832-1 como no encontrado, pues fue el usuario eliminado).

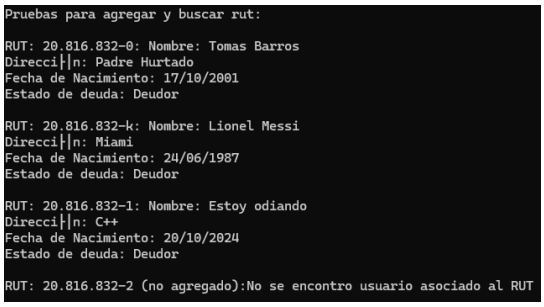

Fig. 3: Resultado de ejecución de mainP2A() en powershell

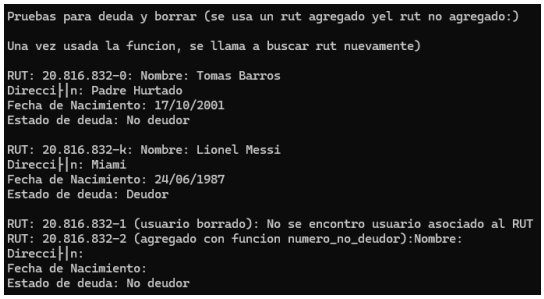

Fig. 4: Resultado de ejecución de mainP2A() en powershell

B. Parte B

Esta sección tiene como propósito guardar en un archivo la información del árbol, implementando una función a la clase.

IV. Referencias

Referencias

[1] C. de StackOverflow, "When to use std::string vs char*," https://stackoverflow.com/questions/10937767/when-to-use-stdstring-vs-char, 2012, escoger entre std::string y char*.
[2] cplusplus.com, "std::string," https://cplusplus.com/reference/string/string/, 2020, funciones que trae std::string.
[3] learncpp.com, "Chars," https://www.learncpp.com/cpp-tutorial/chars/, 2020, número asignado a las variables para trabajar índices.
[4] en.cppreference.com, "std::sort," https://en.cppreference.com/w/cpp/algorithm/sort, 2020, quicksort en C++.
[5] cplusplus.com, "std::stringstream," https://cplusplus.com/reference/sstream/stringstream/, 2020, funciones que trae std::stringstream.
[6] ——, "std::getline," https://cplusplus.com/reference/istream/istream/getline/, 2020, funcion getline y argumentos.

https://www.quora.com/How-do-I-store-a-class-object-into-a-file-in-C

https://www.tutorialspoint.com/read-write-class-objects-from-to-file-in-cplusplus