

Documentación de código.

Primera fase.

el programa permite al usuario interactuar con un diccionario multilingüe, buscando palabras, agregando nuevas palabras con traducciones, eliminando palabras y traduciendo palabras a diferentes idiomas. Utiliza un árbol binario de búsqueda para almacenar y organizar las palabras de manera eficiente.

Estructuras de Datos Utilizadas:

Nodo: Representa un nodo en el árbol binario de búsqueda. Cada nodo contiene una palabra, un mapa de traducciones a otros idiomas, y punteros a sus nodos hijo izquierdo y derecho.

Funciones Principales:

1. `main()`: La función principal del programa. Carga el diccionario desde un archivo, muestra un menú de opciones al usuario y maneja las interacciones del usuario con el diccionario hasta que elige salir.
2. `mostrarMenu()`: Muestra un menú de opciones al usuario.
3. `*cargarDesdeArchivo(Nodo& raiz, const std::string& nombreArchivo)**`: Lee el diccionario desde un archivo de texto y lo carga en la estructura de datos del árbol binario.
4. `*insertar(Nodo& raiz, std::string palabra, std::map<std::string, std::string> traducciones)**`: Inserta una nueva palabra con sus traducciones en el árbol binario.
5. `*eliminar(Nodo raiz, std::string palabra)**`: Elimina una palabra y sus traducciones del árbol binario.
6. `*imprimirTraducciones(Nodo raiz, std::string palabra)**`: Imprime las traducciones disponibles para una palabra dada y permite al usuario seleccionar un idioma para traducir la palabra.
7. `*traducirPalabra(Nodo nodo, std::string idioma)**`: Traduce una palabra a un idioma específico y la reproduce utilizando el comando de sistema `espeak`.
8. `reproducirTraduccion(const std::string& palabra)`: Reproduce la traducción de una palabra utilizando el comando de sistema `espeak`.

Otras Funciones:

`std::string comando = "espeak \"" + palabra + "\"";`: Construye un comando para reproducir la traducción de una palabra utilizando el comando de sistema `espeak`.

Librerías Utilizadas:

- `iostream`: Para entrada y salida estándar.
- `fstream`: Para manejar archivos de texto.
- `sstream`: Para leer líneas del archivo de texto.
- `string`: Para manejar cadenas de caracteres.

- windows.h: Para utilizar funciones del sistema operativo Windows.
- vector: Para almacenar los elementos de los nodos.
- map: Para almacenar las traducciones de las palabras.
- cstdlib: Para utilizar la función system().

Compilación de código.

```
C:\Users\ddere\Dropbox\PC\I x + v
Bienvenido al diccionario multilingue
1. Buscar traducciones de una palabra
2. Agregar una nueva palabra y sus traducciones
3. Eliminar una palabra y sus traducciones
4. Salir
Ingrese su opcion: |
```

```
C:\Users\ddere\Dropbox\PC\I x + v
Bienvenido al diccionario multilingue
1. Buscar traducciones de una palabra
2. Agregar una nueva palabra y sus traducciones
3. Eliminar una palabra y sus traducciones
4. Salir
Ingrese su opcion: 1
Ingrese la palabra a buscar: Direccion
Traducciones disponibles:
espanol
frances
ingles
intaliano
Ingrese el idioma al que desea traducir:
```

Imágenes del código:

```
1  #include <iostream>
2  #include <fstream>
3  #include <sstream>
4  #include <string>
5  #include <windows.h>
6  #include <vector>
7  #include <map>
8  #include <cstdlib>
9
10 struct Nodo {
11     std::string palabra;
12     std::map<std::string, std::string> traducciones;
13     Nodo* izquierda;
14     Nodo* derecha;
15
16     Nodo(std::string palabra) : palabra(palabra), izquierda(nullptr), derecha(nullptr) {}
17 };
18
19 // Prototipos de funciones
20 void reproducirTraduccion(const std::string& palabra);
21 void insertar(Nodo*& raiz, std::string palabra, std::map<std::string, std::string> traducciones);
22 void cargarDesdeArchivo(Nodo*& raiz, const std::string& nombreArchivo);
23 void imprimirTraducciones(Nodo* raiz, std::string palabra);
24 void traducirPalabra(Nodo* actual, std::string idioma);
25 Nodo* eliminar(Nodo* raiz, std::string palabra);
26 void mostrarMenu();
27
28 int main() {
29     Nodo* raiz = nullptr;
30     cargarDesdeArchivo(raiz, "archivo.txt");
31
32     int opcion;
33     do {
34         mostrarMenu();
35         std::cin >> opcion;
36         std::cin.ignore();
37
38         switch (opcion) {
39             case 1: {
40                 std::string palabraBuscar;
41                 std::cout << "Ingrese la palabra a buscar: ";
42                 std::getline(std::cin, palabraBuscar);
43                 imprimirTraducciones(raiz, palabraBuscar);
44             }
```

```

38 switch (opcion) {
39     case 1: {
40         std::string palabraBuscar;
41         std::cout << "Ingrese la palabra a buscar: ";
42         std::getline(std::cin, palabraBuscar);
43         imprimirTraducciones(raiz, palabraBuscar);
44         break;
45     }
46     case 2: {
47         std::string nuevaPalabra;
48         std::cout << "Ingrese la nueva palabra: ";
49         std::getline(std::cin, nuevaPalabra);
50
51         std::map<std::string, std::string> nuevasTraducciones;
52         std::string idioma;
53         std::string traduccion;
54         do {
55             std::cout << "Ingrese el idioma y la traduccion (o 'fin' para terminar): ";
56             std::cin >> idioma;
57             if (idioma != "fin") {
58                 std::cin >> traduccion;
59                 nuevasTraducciones[idioma] = traduccion;
60             }
61         } while (idioma != "fin");
62
63         insertar(raiz, nuevaPalabra, nuevasTraducciones);
64         break;
65     }
66     case 3: {
67         std::string palabraEliminar;
68         std::cout << "Ingrese la palabra a eliminar: ";
69         std::getline(std::cin, palabraEliminar);
70         raiz = eliminar(raiz, palabraEliminar);
71         break;
72     }
73     case 4:
74         std::cout << "Saliendo del programa..." << std::endl;
75         break;
76     default:
77         std::cout << "Opcion no valida. Por favor, ingrese una opción valida." << std::endl;
78         break;
79 }

```

```

80 } while (opcion != 4);
81
82 // Liberar memoria
83 // Implementar una función para liberar la memoria ocupada por los nodos del árbol
84
85 return 0;
86 }
87
88 void reproducirTraduccion(const std::string& palabra) {
89     std::string comando = "espeak \"" + palabra + "\"";
90     system(comando.c_str());
91 }
92
93 void insertar(Nodo*& raiz, std::string palabra, std::map<std::string, std::string> traducciones) {
94     if (raiz == nullptr) {
95         raiz = new Nodo(palabra);
96         raiz->traducciones = traducciones;
97     } else {
98         if (palabra < raiz->palabra) {
99             insertar(raiz->izquierda, palabra, traducciones);
100         } else {
101             insertar(raiz->derecha, palabra, traducciones);
102         }
103     }
104 }
105
106 void cargarDesdeArchivo(Nodo*& raiz, const std::string& nombreArchivo) {
107     std::ifstream archivo(nombreArchivo);
108     if (!archivo.is_open()) {
109         std::cout << "Error al abrir el archivo " << nombreArchivo << std::endl;
110         return;
111     }
112 }

```

```

113     std::string linea;
114     while (std::getline(archivo, linea)) {
115         std::istringstream iss(linea);
116         std::string palabra;
117         std::map<std::string, std::string> traducciones;
118
119         if (iss >> palabra) {
120             std::string idioma;
121             std::string traduccion;
122             while (iss >> idioma >> traduccion) {
123                 traducciones[idioma] = traduccion;
124             }
125             insertar(raiz, palabra, traducciones);
126         }
127     }
128
129     archivo.close();
130 }
131
132 void imprimirTraducciones(Nodo* raiz, std::string palabra) {
133     if (raiz == nullptr) {
134         std::cout << "Palabra no encontrada" << std::endl;
135         return;
136     }
137
138     Nodo* actual = raiz;
139     while (actual != nullptr) {
140         if (palabra == actual->palabra) {
141             std::cout << "Traducciones disponibles:" << std::endl;
142             for (const auto& par : actual->traducciones) {
143                 std::cout << par.first << std::endl;
144             }
145
146             std::string idioma;
147             std::cout << "Ingresa el idioma al que desea traducir: ";
148             std::cin >> idioma;
149
150             traducirPalabra(actual, idioma);
151             return;
152         } else if (palabra < actual->palabra) {
153             actual = actual->izquierda;
154         } else {
155             actual = actual->derecha;
156         }
157     }
158 }

```

```

154         } else {
155             actual = actual->derecha;
156         }
157     }
158
159     std::cout << "Palabra no encontrada" << std::endl;
160 }
161
162 void traducirPalabra(Nodo* nodo, std::string idioma) {
163     if (nodo->traducciones.find(idioma) != nodo->traducciones.end()) {
164         std::cout << "Traduccion a " << idioma << ": " << nodo->traducciones[idioma] << std::endl;
165         reproducirTraduccion(nodo->traducciones[idioma]);
166     } else {
167         std::cout << "Traduccion a " << idioma << " no encontrada" << std::endl;
168     }
169 }
170
171
172
173 Nodo* eliminar(Nodo* raiz, std::string palabra) {
174     if (raiz == nullptr) {
175         return raiz;
176     }
177
178     if (palabra < raiz->palabra) {
179         raiz->izquierda = eliminar(raiz->izquierda, palabra);
180     } else if (palabra > raiz->palabra) {
181         raiz->derecha = eliminar(raiz->derecha, palabra);
182     } else {
183         if (raiz->izquierda == nullptr) {
184             Nodo* temp = raiz->derecha;
185             delete raiz;
186             return temp;
187         } else if (raiz->derecha == nullptr) {
188             Nodo* temp = raiz->izquierda;
189             delete raiz;
190             return temp;
191         }

```

```

178     if (palabra < raiz->palabra) {
179         raiz->izquierda = eliminar(raiz->izquierda, palabra);
180     } else if (palabra > raiz->palabra) {
181         raiz->derecha = eliminar(raiz->derecha, palabra);
182     } else {
183         if (raiz->izquierda == nullptr) {
184             Nodo* temp = raiz->derecha;
185             delete raiz;
186             return temp;
187         } else if (raiz->derecha == nullptr) {
188             Nodo* temp = raiz->izquierda;
189             delete raiz;
190             return temp;
191         }
192
193         Nodo* temp = raiz->derecha;
194         while (temp->izquierda != nullptr) {
195             temp = temp->izquierda;
196         }
197
198         raiz->palabra = temp->palabra;
199         raiz->traducciones = temp->traducciones;
200         raiz->derecha = eliminar(raiz->derecha, temp->palabra);
201     }
202
203     return raiz;
204 }
205
206 void mostrarMenu() {
207     std::cout << "Bienvenido al diccionario multilingue" << std::endl;
208     std::cout << "1. Buscar traducciones de una palabra" << std::endl;
209     std::cout << "2. Agregar una nueva palabra y sus traducciones" << std::endl;
210     std::cout << "3. Eliminar una palabra y sus traducciones" << std::endl;
211     std::cout << "4. Salir" << std::endl;
212     std::cout << "Ingresa su opcion: ";
213 }
214

```

Segunda Fase

La propuesta consiste en integrar un árbol AVL en el proceso de carga y búsqueda de palabras en un archivo, con el objetivo de mejorar el rendimiento y la eficiencia del sistema. Además, se plantea la implementación de un mecanismo para encriptar las palabras buscadas por los usuarios y almacenar su historial de búsqueda, de manera que posteriormente se les pueda sugerir el top de palabras más buscadas por cada uno.

Componentes principales:

- **Árbol AVL:** Un árbol AVL es una estructura de datos autoequilibrada que permite almacenar y recuperar información de manera eficiente. En este caso, el árbol AVL se utilizará para almacenar las palabras del archivo.

```
struct NodoAVL {
    string palabra;
    map<string, string> traducciones;
    NodoAVL* izquierda;
    NodoAVL* derecha;
    int altura;

    NodoAVL(const string& palabra, const map<string, string>& traducciones)
        : palabra(palabra), traducciones(traducciones), izquierda(nullptr), derecha(nullptr), altura(1) {}
};
```

```
54 int altura(NodoAVL* nodo) {
55     return nodo ? nodo->altura : 0;
56 }
57
58 int maximo(int a, int b) {
59     return (a > b) ? a : b;
60 }
61
62 NodoAVL* rotacionDerecha(NodoAVL* y) {
63     NodoAVL* x = y->izquierda;
64     NodoAVL* T = x->derecha;
65
66     x->derecha = y;
67     y->izquierda = T;
68
69     y->altura = maximo(altura(y->izquierda), altura(y->derecha)) + 1;
70     x->altura = maximo(altura(x->izquierda), altura(x->derecha)) + 1;
71
72     return x;
73 }
74
75 NodoAVL* rotacionIzquierda(NodoAVL* x) {
76     NodoAVL* y = x->derecha;
77     NodoAVL* T = y->izquierda;
78
79     y->izquierda = x;
80     x->derecha = T;
81
82     x->altura = maximo(altura(x->izquierda), altura(x->derecha)) + 1;
83     y->altura = maximo(altura(y->izquierda), altura(y->derecha)) + 1;
84
85     return y;
86 }
87
```

```

int obtenerBalance(NodoAVL* nodo) {
    return nodo ? altura(nodo->izquierda) - altura(nodo->derecha) : 0;
}

NodoAVL* insertar(NodoAVL* raiz, const string& palabra, const map<string, string>& traducciones) {
    if (raiz == nullptr)
        return new NodoAVL(palabra, traducciones);

    if (palabra < raiz->palabra)
        raiz->izquierda = insertar(raiz->izquierda, palabra, traducciones);
    else if (palabra > raiz->palabra)
        raiz->derecha = insertar(raiz->derecha, palabra, traducciones);
    else
        return raiz; // La palabra ya está en el árbol

    raiz->altura = 1 + maximo(altura(raiz->izquierda), altura(raiz->derecha));

    int balance = obtenerBalance(raiz);

    // Casos de desbalance
    if (balance > 1 && palabra < raiz->izquierda->palabra)
        return rotacionDerecha(raiz);

    if (balance < -1 && palabra > raiz->derecha->palabra)
        return rotacionIzquierda(raiz);

    if (balance > 1 && palabra > raiz->izquierda->palabra) {
        raiz->izquierda = rotacionIzquierda(raiz->izquierda);
        return rotacionDerecha(raiz);
    }
}

```

```

119 if (balance < -1 && palabra < raiz->derecha->palabra) {
120     raiz->derecha = rotacionDerecha(raiz->derecha);
121     return rotacionIzquierda(raiz);
122 }
123
124 return raiz;
125 }
126
127 NodoAVL* nodoMinimoValor(NodoAVL* nodo) {
128     NodoAVL* actual = nodo;
129     while (actual->izquierda != nullptr)
130         actual = actual->izquierda;
131     return actual;
132 }
133

```

```

133
134 NodoAVL* eliminar(NodoAVL* raiz, const string& palabra) {
135     if (raiz == nullptr)
136         return raiz;
137
138     if (palabra < raiz->palabra)
139         raiz->izquierda = eliminar(raiz->izquierda, palabra);
140     else if (palabra > raiz->palabra)
141         raiz->derecha = eliminar(raiz->derecha, palabra);
142     else {
143         if (raiz->izquierda == nullptr || raiz->derecha == nullptr) {
144             NodoAVL* temp = raiz->izquierda ? raiz->izquierda : raiz->derecha;
145
146             if (temp == nullptr) {
147                 temp = raiz;
148                 raiz = nullptr;
149             } else
150                 *raiz = *temp;
151
152             delete temp;
153         } else {
154             NodoAVL* temp = nodoMinimoValor(raiz->derecha);
155             raiz->palabra = temp->palabra;
156             raiz->traducciones = temp->traducciones;
157             raiz->derecha = eliminar(raiz->derecha, temp->palabra);
158         }
159     }
160 }

```



```

161     if (raiz == nullptr)
162         return raiz;
163
164     raiz->altura = 1 + maximo(altura(raiz->izquierda), altura(raiz->derecha));
165
166     int balance = obtenerBalance(raiz);
167
168     if (balance > 1 && obtenerBalance(raiz->izquierda) >= 0)
169         return rotacionDerecha(raiz);
170
171     if (balance > 1 && obtenerBalance(raiz->izquierda) < 0) {
172         raiz->izquierda = rotacionIzquierda(raiz->izquierda);
173         return rotacionDerecha(raiz);
174     }
175
176     if (balance < -1 && obtenerBalance(raiz->derecha) <= 0)
177         return rotacionIzquierda(raiz);
178
179     if (balance < -1 && obtenerBalance(raiz->derecha) > 0) {
180         raiz->derecha = rotacionDerecha(raiz->derecha);
181         return rotacionIzquierda(raiz);
182     }
183
184     return raiz;
185 }

```

- **Encriptación:** Se propone utilizar un esquema de encriptación simple que reemplaza las vocales por la letra "U", las letras minúsculas por la letra "m" y las letras mayúsculas por la letra "g". Este esquema permite mantener la privacidad de las palabras buscadas por los usuarios.

```

378 string encriptar(string& palabra, NodoAVL*& raiz) {
379     string resultado;
380     for (char c : palabra) {
381         if (isupper(c)) {
382             if (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U') {
383                 resultado += 'U';
384             } else {
385                 resultado += 'g';
386             }
387         } else if (islower(c)) {
388             if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
389                 resultado += 'U';
390             } else {
391                 resultado += 'm';
392             }
393         } else {
394             resultado += c;
395         }
396     }
397
398     // Eliminar la palabra del diccionario
399     raiz = eliminar(raiz, palabra);
400
401     return resultado;
402 }
403

```

```

405 string descriptar(const string& palabra, NodoAVL*& raiz) {
406     string resultado;
407     for (char c : palabra) {
408         if (c == 'U') {
409             resultado += 'a'; // Suponiendo 'a' como vocal de ejemplo
410         } else if (c == 'm') {
411             resultado += 'b'; // Suponiendo 'b' como consonante minúscula de ejemplo
412         } else if (c == 'g') {
413             resultado += 'B'; // Suponiendo 'B' como consonante mayúscula de ejemplo
414         } else {
415             resultado += c;
416         }
417     }
418
419     // Agregar la palabra descriptada de nuevo al diccionario
420     map<string, string> traducciones;
421     string idioma, traduccion;
422     char deseaAgregarMas;
423     cout << "Ingrese el idioma de la traduccion: ";
424     cin >> idioma;
425     cout << "Ingrese la traduccion en " << idioma << ": ";
426     cin >> traduccion;
427     traducciones[idioma] = traduccion;
428     raiz = insertar(raiz, resultado, traducciones);
429
430     return resultado;
431 }
432

```

```

435 void registrarEncriptacion(const string& usuario, const string& palabraOriginal, const string& palabraEncriptada, bool encriptar) {
436     ofstream archivo("encriptacion.txt", ios::app);
437     if (archivo.is_open()) {
438         archivo << (encriptar ? "Encriptado por" : "Desencriptado por") << usuario << " : " << palabraOriginal << " -> " << palabraEncriptada << endl;
439         archivo.close();
440     } else {
441         cout << "Error al abrir el archivo de encriptaciones" << endl;
442     }
443 }
444

```

```

522 int main() {
523     NodoAVL* raiz = nullptr;
524     cargarDesdeArchivo(raiz, "diccionario.txt");
525     cargarUsuariosDesdeArchivo(usuarios, "usuarios.txt");
526
527     gestionarUsuarios();
528
529     int opcion;
530     do {
531         mostrarMenu();
532         cin >> opcion;
533         procesarOpcion(opcion, raiz);
534     } while (opcion != 7);
535
536     return 0;
537 }

```

- **Historial de búsqueda:** Se almacenará el historial de búsqueda de cada usuario, asociando cada palabra buscada con la frecuencia con la que ha sido buscada.

```

30 struct Historial {
31     vector<tuple<string, string, string>> traduccionesRealizadas; // palabra original, idioma, traducción
32
33     void registrarTraduccion(const string& palabra, const string& idioma, const string& traduccion) {
34         traduccionesRealizadas.push_back(make_tuple(palabra, idioma, traduccion));
35     }
36
37     void guardarEnArchivo(const string& nombreArchivo) {
38         ofstream archivo(nombreArchivo);
39         if (archivo.is_open()) {
40             for (const auto& entrada : traduccionesRealizadas) {
41                 archivo << get<0>(entrada) << ":" << get<1>(entrada) << ":" << get<2>(entrada) << endl;
42             }
43             archivo.close();
44         } else {
45             cout << "Error al abrir el archivo " << nombreArchivo << endl;
46         }
47     }
48 };

```

Funcionamiento:

1. **Carga del archivo:** Al cargar un archivo nuevo, las palabras del archivo se leerán y se encriptarán utilizando el esquema propuesto.
2. **Inserción en el árbol AVL:** Cada palabra encriptada se insertará en el árbol AVL.
3. **Búsqueda de palabras:** Cuando un usuario busca una palabra, se encripta primero utilizando el mismo esquema y luego se busca en el árbol AVL. Si la palabra se encuentra en el árbol, se recupera su frecuencia de búsqueda y se actualiza el historial del usuario.

Tercera fase.

Esta estructura almacena la información de un usuario.

```
struct Usuario {  
    string nombre;  
    string contrasena;  
    Usuario(const string& nombre, const string& contrasena) : nombre(nombre), contrasena(contrasena) {}  
};
```

Esta estructura se encarga de registrar y almacenar el historial de traducciones realizadas.

```
struct Historial {  
    vector<tuple<string, string, string>> traduccionesRealizadas;  
  
    void registrarTraduccion(const string& palabra, const string& idioma, const string& traduccion) {  
        traduccionesRealizadas.push_back(make_tuple(palabra, idioma, traduccion));  
    }  
  
    void guardarEnArchivo(const string& nombreArchivo) {  
        ofstream archivo(nombreArchivo);  
        if (archivo.is_open()) {  
            for (const auto& entrada : traduccionesRealizadas) {  
                archivo << get<0>(entrada) << ":" << get<1>(entrada) << ":" << get<2>(entrada) << endl;  
            }  
            archivo.close();  
        } else {  
            cout << "Error al abrir el archivo " << nombreArchivo << endl;  
        }  
    }  
};
```

Variables Globales

- vector<Usuario> usuarios: Almacena la lista de usuarios.
- Usuario usuarioActual("", ""): Almacena el usuario que ha iniciado sesión actualmente.
- Historial historial: Almacena el historial de traducciones realizadas.

Funciones de Gestión de Usuarios

- cargarUsuariosDesdeArchivo
- Carga los usuarios desde un archivo y los almacena en el vector usuarios.

```
void cargarUsuariosDesdeArchivo(vector<Usuario>& usuarios, const string& nombreArchivo) {  
    ifstream archivo(nombreArchivo);  
    if (archivo.is_open()) {  
        string linea;  
        while (getline(archivo, linea)) {  
            istringstream iss(linea);  
            string nombre, contrasena;  
            getline(iss, nombre, ',');  
            getline(iss, contrasena, ',');  
            usuarios.emplace_back(nombre, contrasena);  
        }  
        archivo.close();  
    }  
}
```

GuardarUsuariosEnArchivo

Guarda los usuarios del vector usuarios en un archivo.

```
void guardarUsuariosEnArchivo(const vector<Usuario>& usuarios, const string& nombreArchivo) {
    ofstream archivo(nombreArchivo);
    if (archivo.is_open()) {
        for (const auto& usuario : usuarios) {
            archivo << usuario.nombre << "," << usuario.contrasena << endl;
        }
        archivo.close();
    }
}
```

crearUsuario

Permite la creación de un nuevo usuario y lo guarda en el archivo.

```
void crearUsuario(vector<Usuario>& usuarios) {
    string nombre, contrasena;
    cout << "Ingrese nombre de usuario: ";
    cin >> nombre;
    cout << "Ingrese contraseña: ";
    cin >> contrasena;
    usuarios.emplace_back(nombre, contrasena);
    guardarUsuariosEnArchivo(usuarios, "usuarios.txt");
}
```

iniciarSesion

Permite a un usuario iniciar sesión verificando su nombre y contraseña.

```
bool iniciarSesion(vector<Usuario>& usuarios, Usuario& usuarioActual) {
    string nombre, contrasena;
    cout << "Ingrese nombre de usuario: ";
    cin >> nombre;
    cout << "Ingrese contraseña: ";
    cin >> contrasena;
    for (const auto& usuario : usuarios) {
        if (usuario.nombre == nombre && usuario.contrasena == contrasena) {
            usuarioActual = usuario;
            return true;
        }
    }
    return false;
}
```

gestionarUsuarios

Función principal para gestionar usuarios, permite crear un usuario nuevo o iniciar sesión.

```
void gestionarUsuarios() {
    int opcion;
    do {
        cout << "1. Crear usuario\n2. Iniciar sesion\nSeleccione una opcion: ";
        cin >> opcion;
        switch (opcion) {
            case 1:
                crearUsuario(usuarios);
                break;
            case 2:
                if (iniciarSesion(usuarios, usuarioActual)) {
                    cout << "Inicio de sesion exitoso\n";
                    return;
                } else {
                    cout << "Usuario o contraseña incorrectos\n";
                }
                break;
            default:
                cout << "Opción no valida. Intente nuevamente.\n";
        }
    } while (opcion != 2);
}
```

Flujo de Ejecución

- Carga de Usuarios y Diccionario:
- Al iniciar el programa, se cargan los usuarios desde el archivo usuarios.txt y el diccionario desde diccionario.txt.

```
NodoAVL* raiz = nullptr;
cargarDesdeArchivo(raiz, "diccionario.txt");
cargarUsuariosDesdeArchivo(usuarios, "usuarios.txt");
```

Gestión de Usuarios:

Se llama a la función gestionarUsuarios para permitir al usuario crear una cuenta o iniciar sesión.

gestionarUsuarios();

Interacción del Usuario:

Una vez iniciado sesión, el programa muestra un menú y permite al usuario realizar diversas operaciones como buscar traducciones, agregar nuevas palabras, eliminar palabras, etc.

```
int opcion;  
do {  
    mostrarMenu();  
    cin >> opcion;  
    procesarOpcion(opcion, raiz);  
} while (opcion != 7);
```

Guardar Datos:

Al cerrar sesión, se guarda el diccionario actualizado y los datos de los usuarios.

```
guardarEnArchivo(raiz, "diccionario.txt");  
cout << "Gracias por usar el diccionario multilingue. Hasta luego!" << endl;  
exit(0);
```

Este flujo asegura que los datos de los usuarios y las traducciones se manejen de manera persistente y segura, permitiendo una interacción eficiente y efectiva con el diccionario multilingüe.

Diagrama del Código

