**Universidad Nacional Del Altiplano**

**Facultad De Ingeniería Mecánica Eléctrica, Electrónica Y Sistemas**

Imagen que contiene Interfaz de usuario gráfica

El contenido generado por IA puede ser incorrecto.**Escuela Profesional De Ingeniería De Sistemas**

**Practica Nº4-"Sistema de Gestión de Registros Académicos con Estructuras Dinámicas y Programación**

**Orientada a Objetos"Implementación avanzada de estructuras dinámicas con listas enlazadas personalizadas** **CURSO:**

Algoritmos y Estructuras de Datos

**DOCENTE:**

Mg. Aldo Hernan Zanabria Galvez.

**ESTUDIANTE:**

Yefferson Miranda Josec

**CODIGO:** 216984

**FECHA:** 28/04/2025

**SEMESTRE:**

IV

# I. OBJETIVO GENERAL

Implementar un sistema de gestión de registros basado en listas doblemente enlazadas o

circulares, haciendo uso de técnicas de programación orientada a objetos, con

almacenamiento y recuperación desde archivos.

# II. OBJETIVOS ESPECÍFICOS

• Aplicar conceptos de encapsulamiento, abstracción y modularidad en la

implementación de clases.

• Desarrollar funciones CRUD (crear, leer, actualizar, eliminar) sobre una estructura

dinámica.

• Integrar lectura y escritura de archivos en formato .txt o .csv.

• Analizar la complejidad computacional de cada operación implementada.

• Reflexionar sobre la aplicabilidad real de las estructuras implementadas.

# III. DESCRIPCIÓN DEL PROBLEMA

Se desea construir un sistema que permita gestionar registros de estudiantes, cada uno con los

siguientes atributos:

• ID (entero autoincremental)

• Nombre completo

• Correo electrónico

• Escuela profesional

• Año de ingreso

Los registros deben ser almacenados dinámicamente mediante una estructura de lista

enlazada, y deben implementarse como una clase de tipo Lista, Pila o Cola.

La información debe almacenarse además en archivos planos (.txt o .csv) para ser recuperada

al reiniciar el programa.

Se sugiere implementar esta práctica en C++, por ser el lenguaje base del curso, aunque

también puede explorarse la implementación paralela en Python

# III. DESCRIPCIÓN DEL PROBLEMA

Se desea construir un sistema que permita gestionar registros de estudiantes, cada uno con los

siguientes atributos:

• ID (entero autoincremental)

• Nombre completo

• Correo electrónico

• Escuela profesional

• Año de ingreso

Los registros deben ser almacenados dinámicamente mediante una estructura de lista

enlazada, y deben implementarse como una clase de tipo Lista, Pila o Cola.

La información debe almacenarse además en archivos planos (.txt o .csv) para ser recuperada

al reiniciar el programa.

Se sugiere implementar esta práctica en C++, por ser el lenguaje base del curso, aunque

también puede explorarse la implementación paralela en Python.

# IV. FUNCIONALIDADES REQUERIDAS

1. Agregar nuevo estudiante con ID automático.

2. Mostrar todos los estudiantes registrados desde el nodo inicial.

3. Buscar estudiante por ID o nombre.

4. Modificar datos de un estudiante encontrado.

5. Eliminar un estudiante dado su ID.

6. Guardar los datos en archivo .txt o .csv.

7. Cargar los datos desde archivo al iniciar el programa.

# V. CODIGO IMPLEMENTADO

<https://github.com/yefferson12355/Algoritmos-y-Estructuras-de-Datos>

#include <iostream>

#include <string>

#include <fstream>

**using** **namespace** std;

**class** **Nodo** {

**public:**

Nodo\* siguiente;

Nodo\* anterior;

**int** id;

string nombre;

string correo;

string escuela;

**int** anio;

Nodo(**int** id, string nombre, string correo, string escuela, **int** anio) {

**this**->id = id;

**this**->nombre = nombre;

**this**->correo = correo;

**this**->escuela = escuela;

**this**->anio = anio;

siguiente = nullptr;

anterior = nullptr;

}

};

// Clase ListaDoble con operaciones básicas

**class** **ListaDoble** {

**private:**

Nodo\* cabeza;

**int** contadorId;

**public:**

ListaDoble() {

cabeza = nullptr;

contadorId = **1**;

}

Nodo\* getCabeza() { **return** cabeza; }

**void** setCabeza(Nodo\* nuevoCabeza) { cabeza = nuevoCabeza; }

**void** agregar(string nombre, string correo, string escuela, **int** anio)

{

Nodo\* nuevo = **new** Nodo(contadorId++, nombre, correo, escuela, anio);

**if** (cabeza == nullptr) {

cabeza = nuevo;

} **else** {

Nodo\* actual = cabeza;

**while** (actual->siguiente != nullptr) {

actual = actual->siguiente;

}

actual->siguiente = nuevo;

nuevo->anterior = actual;

}

}

**void** mostrar() {

Nodo\* actual = cabeza;

**while** (actual != nullptr) {

cout << actual->id << " - " << actual->nombre << " - " <<

actual->correo << endl;

actual = actual->siguiente;

}

}

**void** buscar(**int** id) {

Nodo\* actual = cabeza;

**while** (actual != nullptr) {

**if** (actual->id == id) {

cout << "Encontrado: " << actual->nombre << endl;

**return**;

}

actual = actual->siguiente;

}

cout << "No encontrado." << endl;

}

**void** modificar(**int** id, string nuevoNombre, string nuevoCorreo, string nuevaEscuela, **int** nuevoAnio) {

Nodo\* actual = cabeza;

**while** (actual != nullptr) {

**if** (actual->id == id) {

actual->nombre = nuevoNombre;

actual->correo = nuevoCorreo;

actual->escuela = nuevaEscuela;

actual->anio = nuevoAnio;

cout << "Registro modificado." << endl;

**return**;

}

actual = actual->siguiente;

}

cout << "No encontrado." << endl;

}

**void** eliminar(**int** id) {

Nodo\* actual = cabeza;

**while** (actual != nullptr) {

**if** (actual->id == id) {

**if** (actual->anterior != nullptr) {

actual->anterior->siguiente = actual->siguiente;

} **else** {

cabeza = actual->siguiente; // Si es el primero

}

**if** (actual->siguiente != nullptr) {

actual->siguiente->anterior = actual->anterior;

}

**delete** actual;

cout << "Registro eliminado." << endl;

**return**;

}

actual = actual->siguiente;

}

cout << "No encontrado." << endl;

}

**void** guardar() {

ofstream archivo("datos.txt");

**if** (!archivo) {

cout << "Error al abrir el archivo para guardar." << endl;

**return**;

}

Nodo\* actual = cabeza;

**while** (actual != nullptr) {

archivo << actual->id << ";" << actual->nombre << ";" << actual->correo << ";"

<< actual->escuela << ";" << actual->anio << endl;

actual = actual->siguiente;

}

archivo.close();

cout << "Datos guardados correctamente." << endl;

}

**void** cargar() {

ifstream archivo("datos.txt");

**if** (!archivo) {

cout << "Archivo no encontrado. No se cargaron datos." << endl;

**return**;

}

string linea;

**while** (getline(archivo, linea)) {

**int** id, anio;

string nombre, correo, escuela;

**size\_t** pos = **0**;

// Parsear ID

pos = linea.find(";");

id = stoi(linea.substr(**0**, pos));

linea.erase(**0**, pos + **1**);

// Parsear nombre

pos = linea.find(";");

nombre = linea.substr(**0**, pos);

linea.erase(**0**, pos + **1**);

// Parsear correo

pos = linea.find(";");

correo = linea.substr(**0**, pos);

linea.erase(**0**, pos + **1**);

// Parsear escuela

pos = linea.find(";");

escuela = linea.substr(**0**, pos);

linea.erase(**0**, pos + **1**);

// Parsear año

anio = stoi(linea);

Nodo\* nuevo = **new** Nodo(id, nombre, correo, escuela, anio);

**if** (cabeza == nullptr) {

cabeza = nuevo;

} **else** {

Nodo\* actual = cabeza;

**while** (actual->siguiente != nullptr) {

actual = actual->siguiente;

}

actual->siguiente = nuevo;

nuevo->anterior = actual;

}

**if** (id >= contadorId) {

contadorId = id + **1**;

}

}

archivo.close();

cout << "Datos cargados correctamente." << endl;

}

};

**int** **main**() {

ListaDoble lista;

lista.cargar(); // Cargar datos al iniciar

**int** opcion;

**do** {

cout << "**\n**----- MENU PRINCIPAL -----**\n**";

cout << "1. Agregar nuevo estudiante**\n**";

cout << "2. Mostrar todos los estudiantes**\n**";

cout << "3. Buscar estudiante por ID o nombre**\n**";

cout << "4. Modificar estudiante**\n**";

cout << "5. Eliminar estudiante por ID**\n**";

cout << "6. Guardar datos en archivo**\n**";

cout << "0. Salir**\n**";

cout << "Seleccione una opcion: ";

cin >> opcion;

cin.ignore(); // Limpiar buffer

**if** (opcion == **1**) {

string nombre, correo, escuela;

**int** anio;

cout << "Nombre: ";

g etline(cin, nombre);

cout << "Correo: ";

getline(cin, correo);

cout << "Escuela: ";

getline(cin, escuela);

cout << "Anio de ingreso: ";

cin >> anio;

cin.ignore();

lista.agregar(nombre, correo, escuela, anio);

cout << "Estudiante agregado.**\n**";

} **else** **if** (opcion == **2**) {

lista.mostrar();

} **else** **if** (opcion == **3**) {

**int** subop;

cout << "Buscar por: 1. ID 2. Nombre**\n**";

cin >> subop;

cin.ignore();

**if** (subop == **1**) {

**int** id;

cout << "Ingrese ID: ";

cin >> id;

cin.ignore();

lista.buscar(id);

} **else** **if** (subop == **2**) {

string nombre;

cout << "Ingrese nombre: ";

getline(cin, nombre);

Nodo\* actual = lista.getCabeza(); // Accedemos al nodo cabeza

**bool** encontrado = false;

**while** (actual != nullptr) {

**if** (actual->nombre == nombre) {

cout << actual->id << " - " << actual->nombre << " - "

<< actual->correo << " - " << actual->escuela << " - " << actual->anio << endl;

encontrado = true;

}

actual = actual->siguiente;

}

**if** (!encontrado)

cout << "No encontrado." << endl;

}

} **else** **if** (opcion == **4**) {

**int** id, anio;

string nombre, correo, escuela;

cout << "ID del estudiante a modificar: ";

cin >> id;

cin.ignore();

cout << "Nuevo nombre: ";

getline(cin, nombre);

cout << "Nuevo correo: ";

getline(cin, correo);

cout << "Nueva escuela: ";

getline(cin, escuela);

cout << "Nuevo anio de ingreso: ";

cin >> anio;

cin.ignore();

lista.modificar(id, nombre, correo, escuela, anio);

} **else** **if** (opcion == **5**) {

**int** id;

cout << "ID del estudiante a eliminar: ";

cin >> id;

cin.ignore();

lista.eliminar(id);

} **else** **if** (opcion == **6**) {

lista.guardar();

} **else** **if** (opcion == **0**) {

lista.guardar(); // Guardar al salir

cout << "Saliendo y guardando datos...**\n**";

} **else** {

cout << "Opcion invalida.**\n**";

}

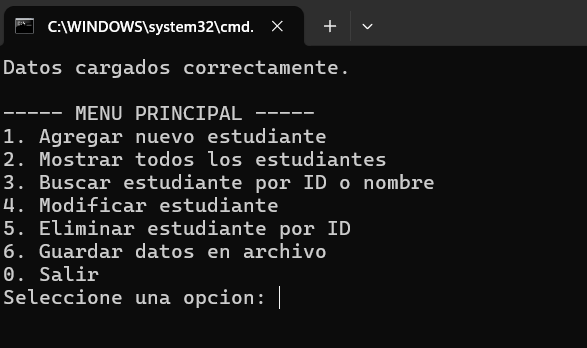
} **while** (opcion != **0**);

**return** **0**;

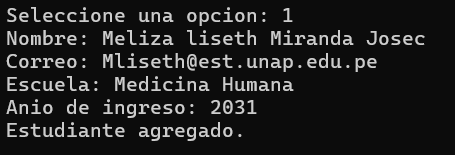
}

// Actividad N°4 - Listas Doblemente Enlazadas

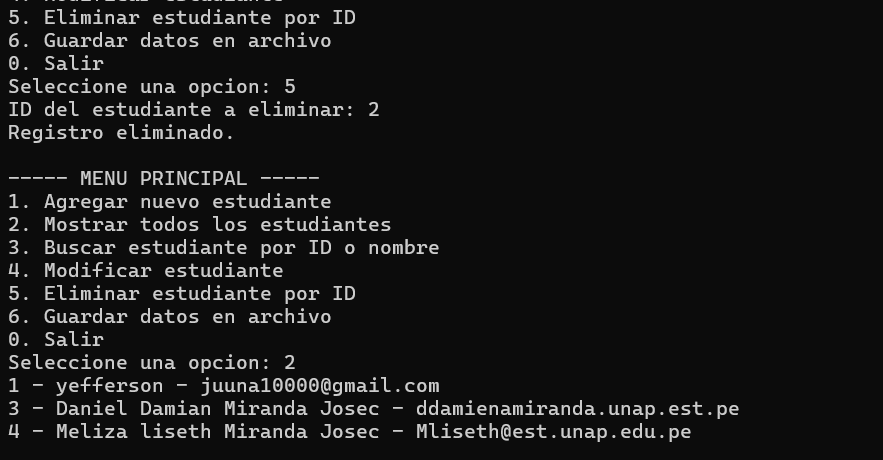
// Integrantes: yo y chat gpt



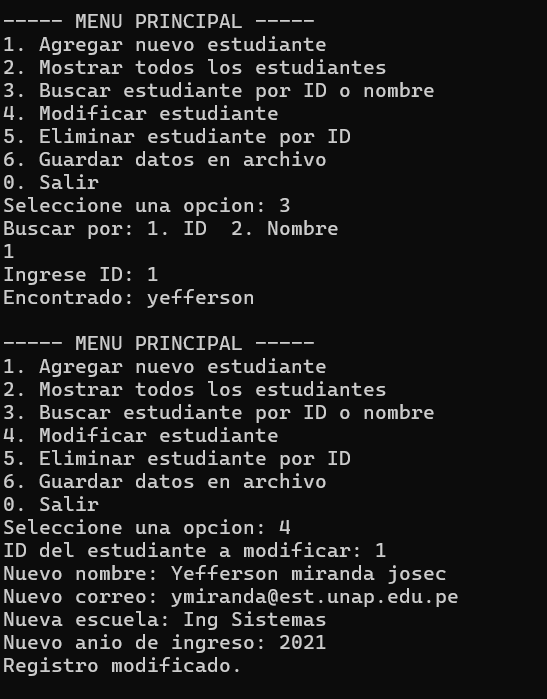
INSERCION



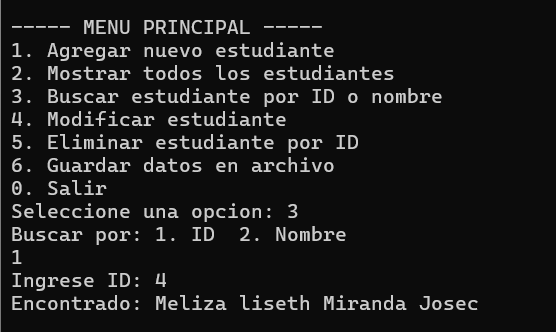
ElIMINACION



MODIFICACION



BUSQUEDA



**PYTON**

**class** **Nodo**:

**def** **\_\_init\_\_**(self, id, nombre, correo, escuela, anio):

self.id = id

self.nombre = nombre

self.correo = correo

self.escuela = escuela

self.anio = anio

self.siguiente = None

self.anterior = None

**class** **ListaDoble**:

**def** **\_\_init\_\_**(self):

self.cabeza = None

self.contador\_id = **1**

**def** **agregar**(self, nombre, correo, escuela, anio):

nuevo = Nodo(self.contador\_id, nombre, correo, escuela, anio)

self.contador\_id += **1**

**if** **not** self.cabeza:

self.cabeza = nuevo

**else**:

actual = self.cabeza

**while** actual.siguiente:

actual = actual.siguiente

actual.siguiente = nuevo

nuevo.anterior = actual

**def** **mostrar**(self):

actual = self.cabeza

**while** actual:

**print**(f"{actual.id} - {actual.nombre} - {actual.correo} - {actual.escuela} - {actual.anio}")

actual = actual.siguiente

**def** **buscar**(self, criterio):

actual = self.cabeza

**while** actual:

**if** str(actual.id) == str(criterio) **or** actual.nombre.lower() == str(criterio).lower():

**print**(f"Encontrado: {actual.id} - {actual.nombre}")

**return** actual

actual = actual.siguiente

**print**("No encontrado.")

**return** None

**def** **modificar**(self, id, nuevo\_nombre, nuevo\_correo, nueva\_escuela, nuevo\_anio):

nodo = self.buscar(id)

**if** nodo:

nodo.nombre = nuevo\_nombre

nodo.correo = nuevo\_correo

nodo.escuela = nueva\_escuela

nodo.anio = nuevo\_anio

**print**("Registro modificado.")

**def** **eliminar**(self, id):

actual = self.cabeza

**while** actual:

**if** actual.id == id:

**if** actual.anterior:

actual.anterior.siguiente = actual.siguiente

**else**:

self.cabeza = actual.siguiente

**if** actual.siguiente:

actual.siguiente.anterior = actual.anterior

**print**("Registro eliminado.")

**return**

actual = actual.siguiente

**print**("No encontrado.")

**def** **guardar**(self, archivo='estudiantes.txt'):

**with** open(archivo, 'w') **as** f:

actual = self.cabeza

**while** actual:

f.write(f"{actual.id},{actual.nombre},{actual.correo},{actual.escuela},{actual.anio}**\n**")

actual = actual.siguiente

**print**("Datos guardados.")

**def** **cargar**(self, archivo='estudiantes.txt'):

**try**:

**with** open(archivo, 'r') **as** f:

**for** linea **in** f:

partes = linea.strip().split(',')

**if** len(partes) == **5**:

self.agregar(partes[**1**], partes[**2**], partes[**3**], int(partes[**4**]))

**except** FileNotFoundError:

**print**("Archivo no encontrado, iniciando lista vacía.")

**def** **menu**():

lista = ListaDoble()

lista.cargar()

**while** True:

**print**("**\n**--- MENÚ ---")

**print**("1. Agregar estudiante")

**print**("2. Mostrar estudiantes")

**print**("3. Buscar estudiante por ID o nombre")

**print**("4. Modificar estudiante")

**print**("5. Eliminar estudiante")

**print**("6. Guardar en archivo")

**print**("0. Salir")

opcion = input("Opción: ")

**if** opcion == '1':

nombre = input("Nombre: ")

correo = input("Correo: ")

escuela = input("Escuela: ")

anio = int(input("Año: "))

lista.agregar(nombre, correo, escuela, anio)

**elif** opcion == '2':

lista.mostrar()

**elif** opcion == '3':

criterio = input("ID o nombre a buscar: ")

lista.buscar(criterio)

**elif** opcion == '4':

id\_mod = int(input("ID a modificar: "))

nombre = input("Nuevo nombre: ")

correo = input("Nuevo correo: ")

escuela = input("Nueva escuela: ")

anio = int(input("Nuevo año: "))

lista.modificar(id\_mod, nombre, correo, escuela, anio)

**elif** opcion == '5':

id\_elim = int(input("ID a eliminar: "))

lista.eliminar(id\_elim)

**elif** opcion == '6':

lista.guardar()

**elif** opcion == '0':

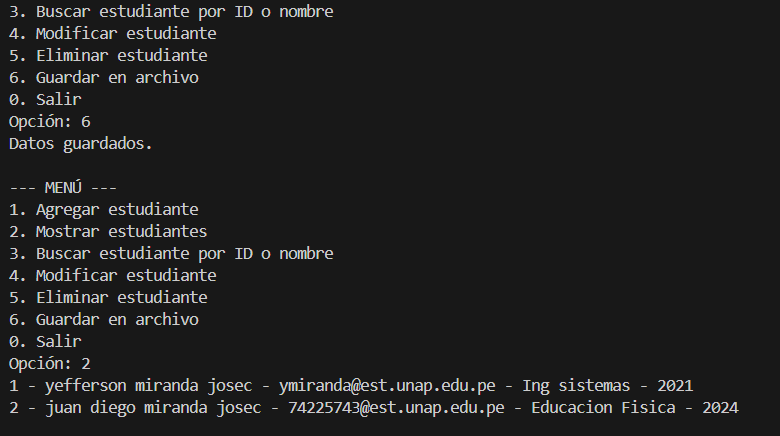
lista.guardar()

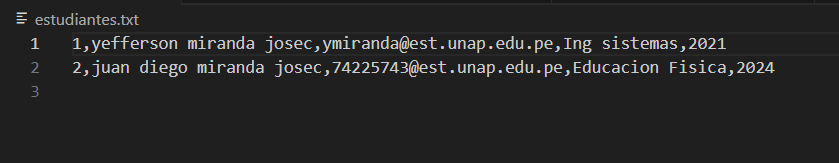
**break**

**else**:

**print**("Opción inválida.")

menu()

****

****

**VII. REQUERIMIENTO**

# ✅ ****1. Aplicar conceptos de encapsulamiento, abstracción y modularidad****

* **Encapsulamiento:** Se usa private y public para controlar el acceso a los datos dentro de la clase ListaDoble.
* **Abstracción:** El usuario del programa no necesita conocer cómo se implementan los nodos o cómo se maneja la lista internamente.
* **Modularidad:** Las operaciones (agregar, mostrar, buscar, modificar, eliminar, guardar, cargar) están separadas en funciones bien definidas, lo que mejora la legibilidad y el mantenimiento del código.

# ✅ ****2. Desarrollar funciones CRUD sobre una estructura dinámica****

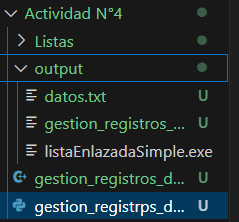
Tu clase ListaDoble implementa completamente las operaciones CRUD:

* **Create:** agregar
* **Read:** mostrar, buscar
* **Update:** modificar
* **Delete:** eliminar

Además, se usa una **estructura dinámica (lista doblemente enlazada)**, cumpliendo este requerimiento.

# ✅ ****3. Integrar lectura y escritura de archivos en .txt o .csv****

* Estás guardando los datos en un archivo .txt con guardar() y cargar(), lo cual permite persistencia de datos entre ejecuciones.
* <https://github.com/yefferson12355/Algoritmos-y-Estructuras-de-Datos>



# ✅ ****4. Analizar la complejidad computacional****

Aunque no está explícitamente documentada en comentarios o informe, **las operaciones tienen complejidades razonables para estructuras enlazadas**:

* **Agregar:** O(n), recorre hasta el final.
* **Buscar:** O(n), recorrido lineal.
* **Modificar y eliminar:** O(n), por búsqueda lineal.
* **Guardar y cargar:** O(n), procesamiento secuencial de nodos.

Puedes añadir comentarios al código o explicaciones en un informe si se requiere profundizar en este análisis.

# ✅ ****5. Reflexionar sobre la aplicabilidad real de las estructuras****

Tu implementación es un ejemplo claro de cómo una **lista doblemente enlazada** puede ser útil para:

* Gestionar información secuencial con posibilidad de recorrer en ambos sentidos.
* Operar sobre colecciones de datos sin usar estructuras estáticas como arrays.

Esta estructura podría usarse en sistemas reales de registro de estudiantes, agendas, etc.

# ****VI. ACTIVIDADES COMPLEMENTARIAS****

# ****1. Análisis de complejidad (Big-O)****

| **Operación** | **Descripción** | **Complejidad Big-O** |
| --- | --- | --- |
| agregar() | Recorre la lista hasta el final y agrega | O(n) |
| mostrar() | Recorre todos los nodos para imprimir | O(n) |
| buscar() | Búsqueda por ID o nombre | O(n) |
| modificar() | Busca un nodo por ID y lo actualiza | O(n) |
| eliminar() | Busca por ID y elimina el nodo | O(n) |
| guardar() | Recorre y guarda todos los nodos en archivo | O(n) |
| cargar() | Lee desde archivo e inserta en la lista | O(n) |

# ****2.** Comparación con arreglos dinámicos**

| **Característica** | **Listas Doblemente Enlazadas** | **Arreglos Dinámicos (vector)** |
| --- | --- | --- |
| Inserción al final | O(n) sin puntero al final | O(1) amortizado |
| Inserción en medio | O(n) | O(n) |
| Eliminación | O(n) | O(n) |
| Búsqueda | O(n) | O(n) (O(1) si está indexado) |
| Acceso por posición directa | O(n) | O(1) |
| Uso de memoria | Eficiente para cambios frecuentes | Uso eficiente si no hay muchas inserciones |
| Doble dirección | Sí | No |

# ****3. Casos reales de aplicación****

* **Sistema de atención por turnos**: Una lista doble puede representar una cola en la que se insertan y eliminan personas por orden de llegada, pero también permite priorizar pacientes críticos moviéndolos hacia adelante fácilmente.
* **Listas de reproducción multimedia**: Los usuarios pueden recorrer canciones hacia adelante o atrás, eliminar o insertar nuevas canciones en cualquier posición de la lista.

# **4**.- ****Explicación del código (estructura, clases y métodos)****

# 📘 ****Estructura general****

* El programa consta de **dos clases principales**:
  1. Nodo: representa un estudiante individual.
  2. ListaDoble: gestiona los nodos mediante una lista doblemente enlazada y contiene métodos CRUD.

# 🧩 ****Clases y métodos****

# 🔸 class Nodo

* Atributos:
  + id: ID único del estudiante.
  + nombre, correo, escuela, anio: datos del estudiante.
  + siguiente, anterior: punteros a los nodos siguiente y anterior.

# 🔸 class ListaDoble

* Atributos:
  + cabeza: primer nodo de la lista.
  + contador\_id: se autoincrementa para asignar IDs únicos.
* Métodos:
  + agregar(): Inserta un nuevo estudiante al final.
  + mostrar(): Imprime todos los nodos desde el primero.
  + buscar(): Busca un nodo por ID o nombre.
  + modificar(): Encuentra y actualiza un nodo existente.
  + eliminar(): Elimina un nodo por su ID.
  + guardar(): Escribe todos los nodos a un archivo .txt.
  + cargar(): Carga registros desde un archivo al iniciar el programa.

# 5.- Bibliografía (APA 7):

Malik, D. S. (2018). *C++ Programming: From Problem Analysis to Program Design* (8th ed.). Cengage Learning.  
Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.