Universidad Continental

FINPROC: SIMULADOR DE PROCESOS DE ATENCIÓN FINANCIERA BANCARIA

Informe Técnico – Entregable 1: Planificación y Diseño

Curso: Estructura de Datos
Docente: Yesenia Concha Ramos
Integrantes:

- *Jhon gustavo Ccarita Velasquez (100%)*
- *Rodrigo Sevillanos Tinco(100%)*
- *Andre Sebastian Espinosa Zea(100%)*

Tema: Entregable 1
NRC: 59098

2025-02
Cusco-Perú

Índice General

Formación de equipos y distribución de roles

El equipo de desarrollo del proyecto FINPROC: Simulador de Procesos de Atención Financiera Bancaria está conformado por tres integrantes, quienes asumen funciones específicas de acuerdo con sus habilidades y responsabilidades dentro del proceso de desarrollo.

Integrantes y roles asignados:

1. **Gustavo Jhon Ccarita Velásquez – Scrum Master y desarrollador principal**

   Responsable de dirigir la planificación y asignación de tareas mediante la metodología ágil **Scrum**, utilizando **Obsidian** con los complementos *Kanban* y *Tasks* para la gestión visual del flujo de trabajo. Supervisa el cumplimiento de los objetivos semanales, fomenta la colaboración y participa activamente en el desarrollo del código y la integración de los módulos del sistema.

2. **Rodrigo Sevillanos Tinco – Analista y diseñador de algoritmos**

   Encargado de realizar el análisis funcional del sistema, diseñar los diagramas de flujo y elaborar los algoritmos principales asociados a las estructuras de datos (lista enlazada, cola y pila). Participa en la documentación técnica y asegura la coherencia entre el diseño lógico y la implementación del sistema.

3. **André Sebastián Espinoza Zea – Tester y responsable de documentación**

   Responsable de la validación funcional del sistema, realizando pruebas exhaustivas de cada módulo. Además, gestiona la documentación técnica, recopila las evidencias del trabajo colaborativo y mantiene actualizado el informe del proyecto, incluyendo avances, actas de reunión y resultados de validación.

Creación del repositorio para el desarrollo del trabajo:

https://github.com/HustavoJhon/FinProc

Planificación de tareas y responsabilidades por cada integrantes

Metodología de Trabajo: Scrum Adaptado

El equipo adoptará una **metodología ágil basada en Scrum**, ajustada a la escala académica del proyecto.

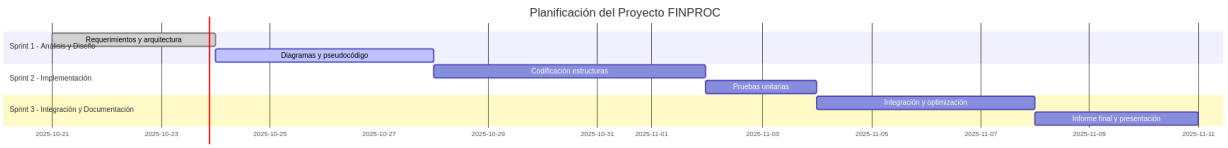
El desarrollo se organizará en **tres sprints semanales**, cada uno con un conjunto definido de objetivos y tareas.

- **Sprint Planning (día 1):** reunión inicial de cada semana para asignar tareas y estimar tiempos.
- **Daily Scrum:** reuniones breves 2 veces por semana (lunes y jueves) para revisar avances y obstáculos.
- **Sprint Review (día 7):** revisión del trabajo finalizado y documentación de resultados.
- **Sprint Retrospective:** ajustes de metodología y redistribución de tareas según desempeño.

Adaptado Planificación de Sprints (Gantt)

semana	sprints	Objetivos Principales	Tareas Clave	Reuniones
Semana 1 (Del 21 al 27 oct)	Sprint 1: Análisis y diseño	Definir requerimientos y arquitectura del sistema	Definir estructura, elaborará diagrama y pseudocódigo	Lunes(Planificación), Jueves (daily
Semana 2 (Del 28 oct al 3 nov)	Sprint 2: Implementación	Codificación de estructura (lista, pila, cola) y pruebas unitarias	Programacion en C++, validación de inserciones, eliminación y búsqueda búsquedas	Lunes (daily), Viernes (review)
Semana 3 (Del 4 al 10 nov)	Spirit 3: Integración y Documentación	Integrar módulos, optimización, documentar y preparar presentación	Ensayo de sustentación, commits finales y revisión del informe	Martes (daily) , Domingo (retrospectiva)

(Figura: Diagrama Gantt de los sprints)



CAPÍTULO 1

1.-Análisis del Problema

1.1. Descripción del problema

En las entidades financieras, la atención de clientes suele presentar problemas de demora y desorganización, especialmente cuando se manejan múltiples tipos de operaciones como depósitos, retiros o solicitudes de préstamos. El objetivo de este proyecto es simular el funcionamiento de un sistema de atención bancaria que gestione de forma ordenada los procesos de atención y registro de transacciones, aplicando estructuras de datos dinámicas lineales (listas enlazadas, colas y pilas) para mejorar la eficiencia y organización.

El **contexto operativo** en las entidades financieras se define por la **concurrencia de múltiples procesos transaccionales** (e.g., depósitos, retiros) en el flujo de atención al cliente. Esta dinámica a menudo resulta en **ineficiencias operativas**, evidenciadas en **elevados tiempos de espera** para el usuario y deficiencias en la organización interna y el registro de operaciones. La problemática central radica en la ausencia de un **mecanismo algorítmico robusto** capaz de gestionar eficientemente los procesos

simultáneos de registro, verificación y ejecución, lo que conlleva a errores operacionales y a una pérdida de trazabilidad.

Objetivo e Implementación Técnica en C++

El objetivo primordial de este proyecto es **modelar y simular** un **Sistema de Gestión de Atención Bancaria** mediante la aplicación rigurosa de **estructuras de datos dinámicas lineales**, aprovechando los contenedores y el control de memoria que ofrece **C++**. Esta implementación técnica está enfocada en **optimizar la secuenciación de los procesos de atención** y el **registro estructurado de transacciones**, buscando un incremento tangible en la eficiencia, organización y trazabilidad operacional del servicio.

La solución se articulará mediante la implementación de los siguientes componentes, utilizando la **Librería de Plantillas Estándar (STL)** de C++ o estructuras de datos personalizadas basadas en punteros:

- **Colas (std::queue o std::priority_queue):** Se utilizarán para gestionar el **flujo de clientes** con lógica **FIFO (First-In, First-Out)** o mediante **esquemas de prioridad** definidos.
- **Pilas (std::stack):** Se implementarán para el **control de procesos con lógica LIFO**, adecuadas para la **validación jerárquica** o la gestión del estado de una sesión.
- **Listas Enlazadas (std::list o nodos y punteros):** Servirán para la **administración dinámica del historial de transacciones** o del maestro de clientes.

Esta aplicación de estructuras de datos busca **analizar y validar empíricamente** cómo

estos componentes permiten **optimizar el orden de servicio**, **manipular eficientemente la información** y **administrar de forma estructurada** el historial operacional. La **simulación desarrollada operará como un banco de pruebas (*testbed*)** en C++, esencial para validar la *performance algorítmica* (complejidad temporal y uso de memoria) y establecer la viabilidad técnica de su implementación en entornos operativos reales de alto rendimiento.

1.2. Requerimientos del sistema

Requerimientos funcionales

- Registrar nuevos clientes y almacenarlos en una lista enlazada.
- Gestionar una cola de atención priorizada según el tipo de cliente o servicio.
- Registrar las transacciones realizadas (retiros, depósitos, préstamos).

Diagramas de las estructuras de datos a implementar

- Guardar y cargar los datos desde archivos (persistencia de información).
- Permitir visualizar el estado actual de los procesos y la memoria.

Requerimientos no funcionales

- El sistema debe ser desarrollado en Dev-C++ y ejecutarse por consola.
- El código debe estar modularizado, comentado y documentado.

- Se debe garantizar una ejecución eficiente en tiempo y memoria.
- Interfaz sencilla, amigable y clara para el usuario.

1.3. Estructuras de datos propuestas

La modelización del sistema de gestión se basará en la implementación estratégica de **estructuras de datos dinámicas lineales**, aprovechando los contenedores de la **Librería de Plantillas Estándar (STL) de C++** para asegurar la eficiencia algorítmica y la flexibilidad del sistema:

1. Lista Enlazada (Linked List)

- **Función Operacional: Registro Maestro de Clientes.** Esta estructura será utilizada para almacenar, administrar y mantener el conjunto dinámico de todos los perfiles de clientes activos.
- **Implementación en C++:** Se utilizará el contenedor **std::list<T>** o una implementación de lista enlazada con **punteros** (Node*).
- **Justificación Técnica:** Ofrece una **complejidad temporal de $O(1)$** para las operaciones de inserción y eliminación de nodos (clientes) en el registro, lo cual es esencial para una gestión eficiente de la memoria y evitar las penalizaciones de redimensionamiento de los *arrays* estáticos.

2. Cola (Queue)

- **Función Operacional: Mecanismo de Despacho de Servicio y Priorización de Atención.** Gestiona el flujo de solicitudes de atención.
- **Implementación en C++:** Se empleará **std::priority_queue<T>** o **std::queue<T>** (dependiendo del esquema de prioridades).
- **Justificación Técnica:** Permite estructurar la atención con lógica **FIFO (First-In, First-Out)** o implementar **algoritmos de *scheduling*** basados en la **prioridad**

(urgencia o tipo de transacción), garantizando una secuenciación lógica y óptima del servicio.

3. Pila (Stack)

- **Función Operacional: Control de Trazabilidad y Reversión de Operaciones.** Administra la secuencia de acciones realizadas por un usuario o agente.
- **Implementación en C++:** Se utilizará el contenedor **std::stack<T>**.
- **Justificación Técnica:** Su principio **LIFO (Last-In, First-Out)** es el más adecuado para la **gestión de la memoria de estado** y la implementación de funcionalidades de **"rollback"** (deshacer), ya que permite acceder y anular la última transacción registrada de forma inmediata y eficiente.

1.4. Justificación de la elección

El **diseño arquitectónico** del sistema de gestión bancaria se fundamenta en la **explotación estratégica de estructuras de datos dinámicas lineales**, aprovechando las capacidades de manipulación de bajo nivel y la librería estándar (**STL**) del lenguaje **C++** para lograr una **modelización precisa y eficiente** de los procesos transaccionales.

La elección de estas estructuras optimiza el rendimiento y la flexibilidad del sistema:

- Listas Enlazadas (std::list o implementación con punteros): Registro Maestro de Clientes.

  Se emplean para el Registro Maestro de Entidades, proporcionando la flexibilidad y escalabilidad necesarias para la gestión dinámica del pool de clientes. En C++, esto permite realizar inserciones y eliminaciones de nodos (clientes) con una complejidad temporal de $O(1)$ (cuando se conoce la

posición), un factor crítico para el manejo eficiente de grandes volúmenes de datos sin las penalizaciones de redimensionamiento de los arrays estáticos.

- Colas (std::queue o std::priority_queue): Mecanismo de Despacho de Servicio.

  Son esenciales para el Mecanismo de Despacho de Servicio, estructurando la atención bajo principios FIFO (First-In, First-Out) o esquemas de prioridad definidos. La implementación con std::priority_queue en C++ facilita el desarrollo de algoritmos de scheduling complejos que asignan recursos de atención basándose en la urgencia o el tipo de transacción, lo cual asegura una secuenciación lógica y optimizada del throughput de solicitudes.

- Pilas (std::stack): Control de Trazabilidad y Reversión.

  Facilitan el Control de Trazabilidad y Reversión de Operaciones. Utilizando su lógica LIFO (Last-In, First-Out), el contenedor std::stack administra el historial de comandos transaccionales. Esto no solo simplifica el seguimiento del estado de una sesión, sino que es fundamental para implementar funcionalidades de rollback eficientes, permitiendo deshacer la última acción de manera inmediata.

CAPÍTULO 2

---

2. Diseño de la Solución

2.1 Descripción general del diseño

El sistema FINPROC se ha diseñado siguiendo una arquitectura modular, donde cada unidad funcional trabaja de manera independiente, pero coordinada para resolver el flujo completo de atención bancaria. Para cumplir con los requerimientos del proyecto, se implementan **estructuras de datos dinámicas lineales**, específicamente una **lista enlazada**, una **cola de prioridad** y una **pila**, cada una destinada a resolver un proceso específico dentro del sistema.

A continuación, se describe detalladamente el funcionamiento de cada módulo, los datos que almacena, cómo se procesan y cómo interactúan entre sí.

2.2 Módulo 1: Gestor de Clientes (Lista Enlazada)

Objetivo del módulo:

Administrar el registro general de clientes del banco, permitiendo almacenar nuevos clientes, buscar sus datos y mantener un listado actualizado.

Estructura de datos utilizada:

Lista enlazada simple, donde cada nodo representa un cliente.

Cada nodo contiene los siguientes campos:

- **Nombre completo**
- **DNI** (campo clave para búsqueda)
- **Fecha de nacimiento**
- **Tipo de cliente**: Regular, Preferencial, VIP, etc.
- **Número de cuenta**
- **Puntero al siguiente nodo** (siguiente cliente)

Flujo técnico del proceso de registro de clientes

1. El usuario ingresa los datos obligatorios del cliente (nombre, DNI, tipo, etc.).
2. El sistema verifica que el DNI no esté duplicado en la lista.
3. Se crea un nuevo nodo de cliente con toda la información ingresada.
4. Si la lista está vacía, el nuevo cliente pasa a ser el primer nodo.
5. Si existe al menos un cliente, la lista se recorre hasta el último nodo.
6. El nuevo nodo se enlaza al final de la lista.
7. El sistema confirma el registro.

Búsqueda de clientes

Para localizar a un cliente por su DNI:

1. La lista se recorre desde el primer nodo.
2. Se compara el DNI registrado con el DNI buscado.
3. Si coinciden, se devuelve al cliente.
4. Si no se encuentra, se informa que el cliente no está registrado.

Este proceso tiene tiempo lineal, lo cual es aceptable en una simulación académica.

2.3 Módulo 2: Gestor de Atención (Cola de Prioridad)

Objetivo del módulo:

Organizar el orden de atención de los clientes que se encuentran esperando ventanilla, simulando el flujo real de un banco.

Estructura de datos utilizada:

Cola de prioridad, donde el criterio de prioridad es:

1. Tipo de cliente (VIP > Preferencial > Regular)
2. Orden de llegada en caso de empate

Cada elemento en la cola guarda:

- **DNI del cliente**
- **Tipo de cliente (para ordenar por prioridad)**
- **Número de turno o posición de llegada**

Flujo técnico del proceso de encolamiento

1. Se solicita el DNI del cliente a encolar.
2. Se verifica que el cliente exista en el registro (lista enlazada).
3. El sistema determina la prioridad según el tipo de cliente.
4. El cliente se inserta en la cola respetando el orden de prioridad:
    - Los clientes VIP quedan al inicio.
    - Los regulares se insertan detrás.
5. Se notifica que el cliente quedó registrado en la cola de atención.

Atención de clientes

1. Se verifica si la cola tiene elementos.

2. Se toma el primer elemento (cliente de mayor prioridad).
3. Se retira de la cola.
4. Se envía el cliente al módulo de transacciones para registrar la operación realizada.

2.4 Módulo 3: Gestor de Transacciones (Pila)

Objetivo del módulo:

Registrar las operaciones realizadas por cada cliente durante su atención (depósitos, retiros, pagos, etc.) y mantener un historial ordenado.

Estructura de datos utilizada:

Pila por cliente, donde la operación reciente queda en la parte superior.

Cada transacción almacena:

- **Tipo de operación** (depósito, retiro, consulta, etc.)
- **Monto involucrado**
- **Fecha y hora**
- **Estado** (realizada o deshecha)

Flujo técnico del proceso de registro de transacciones

1. El cliente es atendido en ventanilla.
2. Se solicita el tipo de operación a realizar.
3. Se registra el monto (si corresponde).
4. Se crea un registro de transacción.
5. La transacción se inserta en la pila del cliente (se apila).
6. La transacción queda registrada como la más reciente.

7. El sistema confirma que la operación fue realizada.

Función adicional: deshacer operación (Undo)

(Dependiendo de si lo incluirán o no)

Para deshacer la última transacción del cliente:

1. Se verifica si su pila tiene operaciones.
2. Se extrae la operación del tope.
3. Se marca como “revertida”.
4. (Opcional) Se ajusta el saldo simulado.

Esto es útil para simular errores humanos o cancelaciones.

2.5 Integración entre módulos

El sistema funciona como una cadena:

1. **Gestor de Clientes** → registra información personal del cliente.
2. **Gestor de Atención** → encola y determina el orden de atención.
3. **Gestor de Transacciones** → registra las operaciones realizadas.

Ejemplo del flujo real:

1. El cliente se registra en la lista enlazada.
2. Va a la cola según su prioridad.
3. Es atendido en ventanilla.
4. Se registra una transacción asociada.

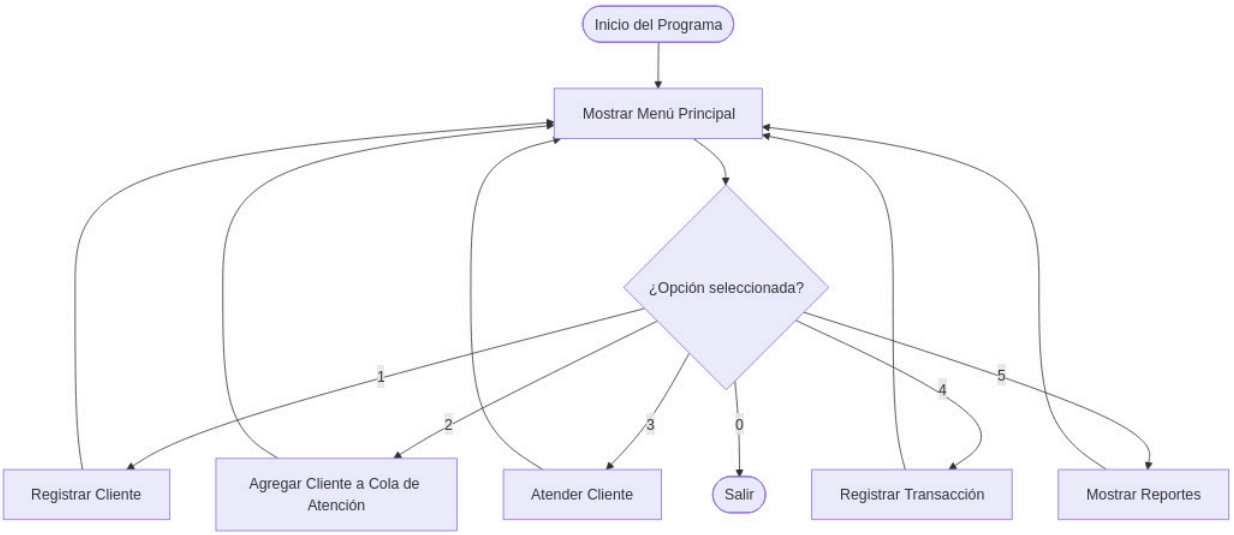
Esto permite simular el funcionamiento real de un banco utilizando estructuras de datos básicas.

2.6 Consideraciones finales del diseño

- La **lista enlazada** permite registrar clientes de forma flexible sin necesidad de una cantidad fija.
- La **cola de prioridad** reproduce el orden de atención respetando prioridades reales.
- La **pila** permite modelar un historial donde la última operación siempre está accesible.
- Los tres módulos trabajan de forma integrada, permitiendo una simulación clara y ordenada.

Diagramas de las estructuras de datos a implementar

1. **DIAGRAMA 1 — Flujo General del Programa (Main Menu)**



2. DIAGRAMA 2 — Registrar Cliente (Lista Enlazada)

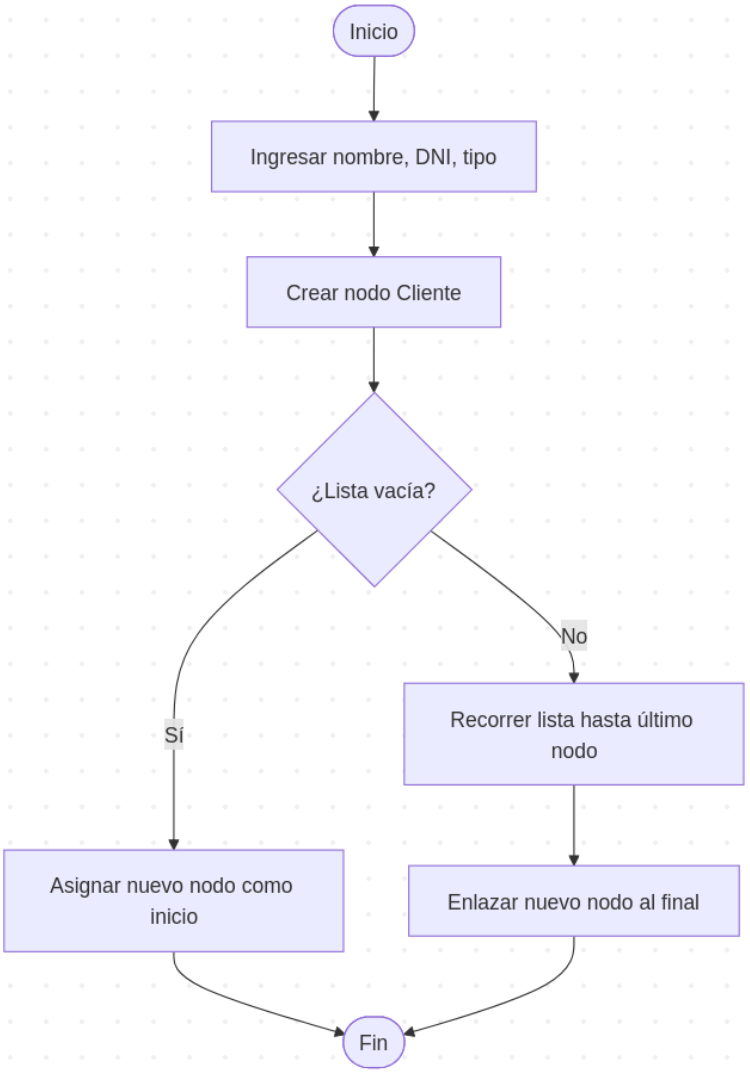


Figura B1. *Diagrama de la Lista Enlazada del Registro de Clientes*
Descripción: El diagrama representa la estructura dinámica utilizada para almacenar el maestro de clientes. Cada nodo contiene los datos personales del cliente y un puntero hacia el siguiente nodo, lo que permite inserciones y búsquedas eficientes.
Relevancia: Esta estructura es fundamental para gestionar clientes de forma flexible durante la simulación

3. DIAGRAMA 3 — Encolar Cliente (Cola de Atención)

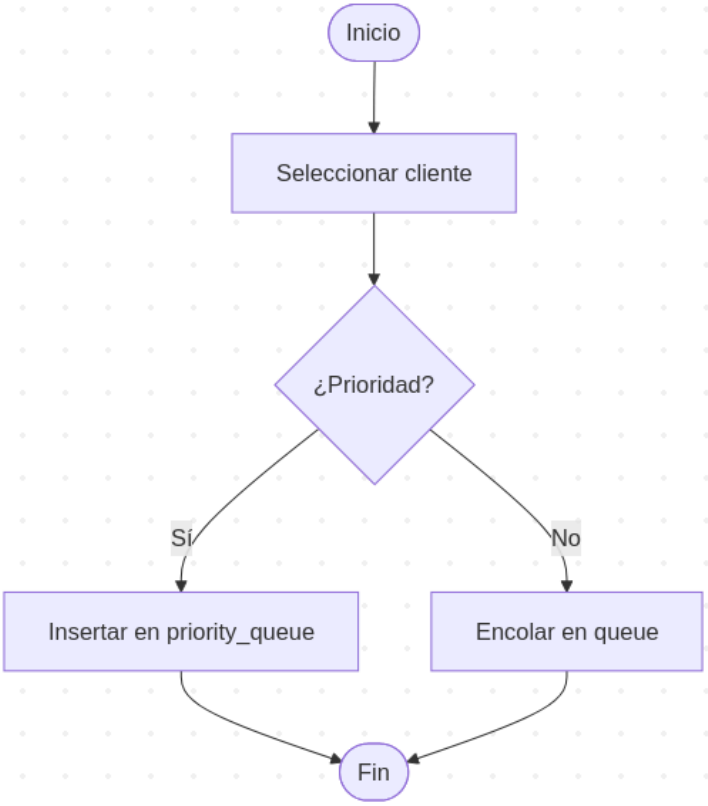
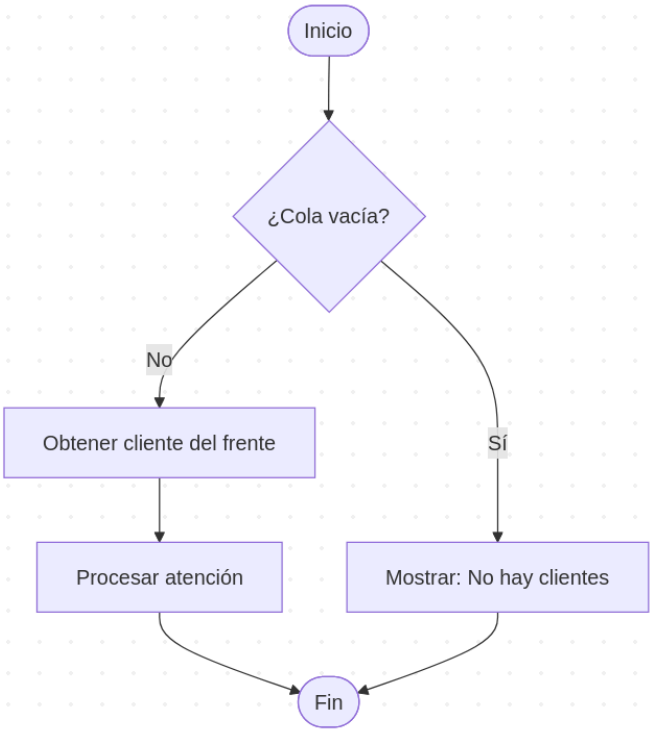


Figura B2. Diagrama de la Cola de Atención Bancaria
Descripción: *En la imagen se visualiza el modelo FIFO o por prioridad empleado para gestionar el orden de atención. Cada elemento en la cola representa un cliente esperando ser atendido.*
Relevancia: *Este mecanismo simula el flujo real de atención en ventanilla.*

4. DIAGRAMA 4 — Atender Cliente en Ventanilla



5. DIAGRAMA 5 — Registrar Transacción (Pila)

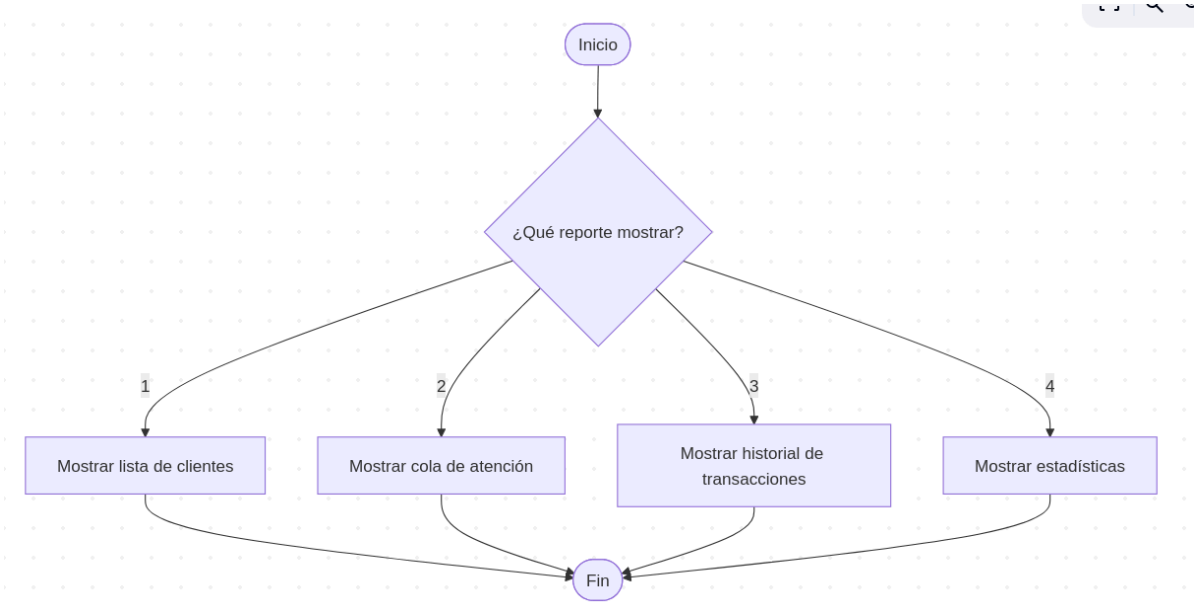
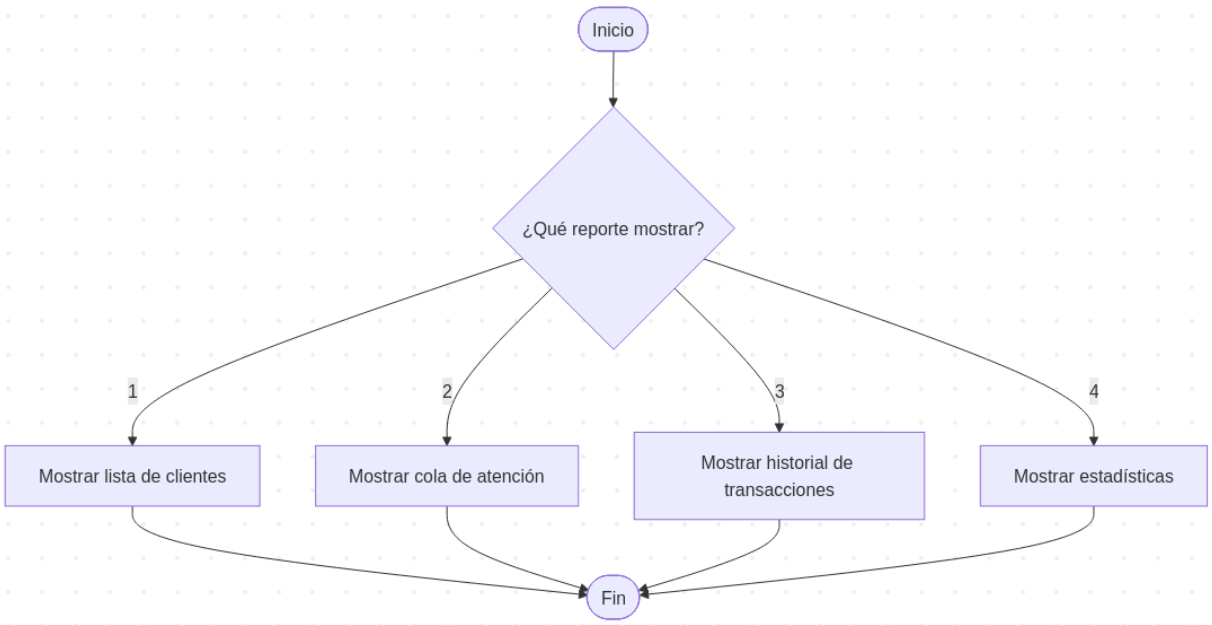
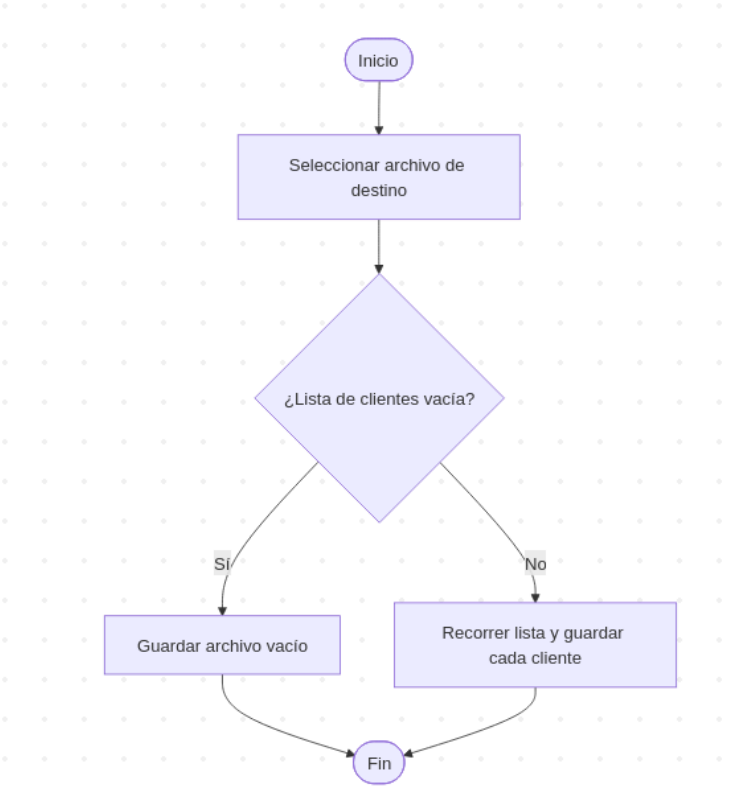


Figura B3. Diagrama de la Pila de Transacciones
Descripción: *La figura muestra la pila utilizada para almacenar las operaciones realizadas por cada cliente. La estructura LIFO permite mantener un historial ordenado y posibilita funciones como deshacer la última operación.*
Relevancia: *Esta estructura asegura la trazabilidad de las acciones en el sistema.*

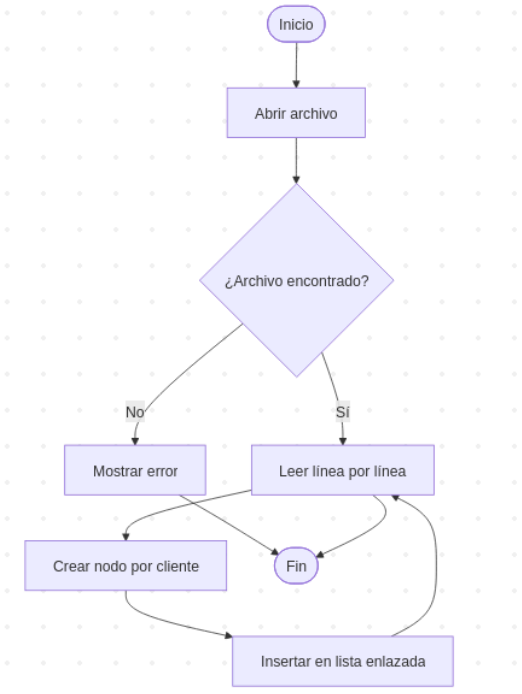
6. DIAGRAMA 6 — Mostrar Reportes del Sistema



7. DIAGRAMA 7 — Guardar Datos en Archivos



8. DIAGRAMA 8 — Cargar Datos desde Archivos



9. Diagramas de Flujo del Sistema

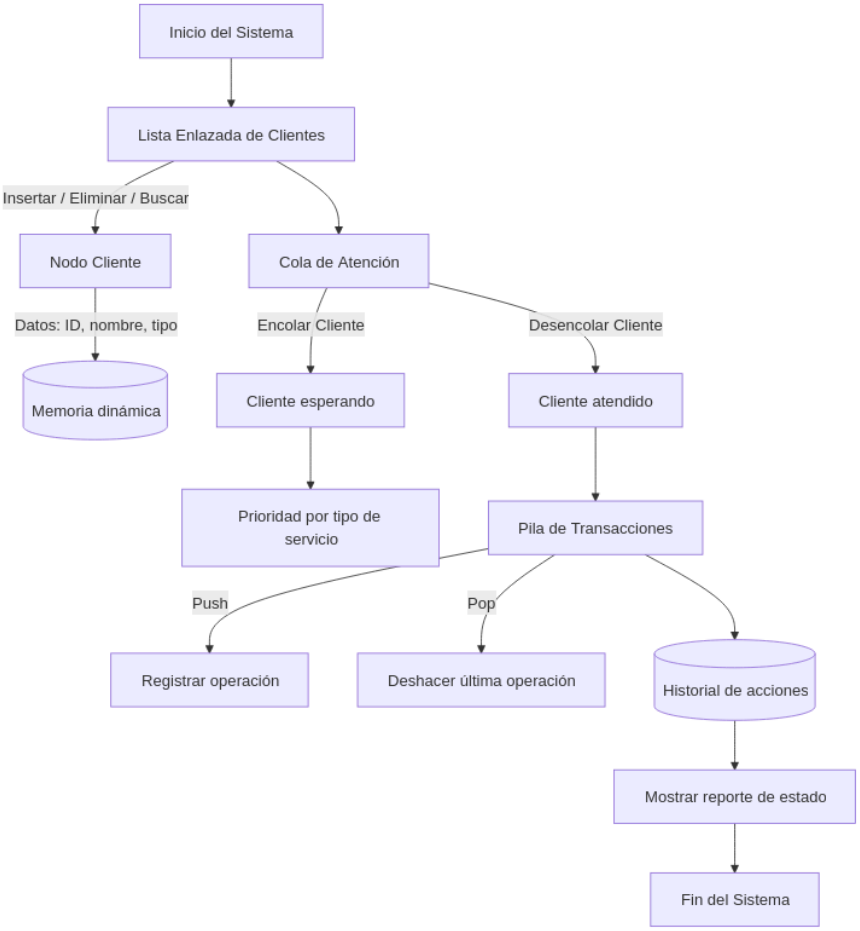


Figura C1. Flujo general del programa FINPROC
Descripción: *El diagrama ilustra la lógica principal del sistema, mostrando el menú inicial y los módulos disponibles para gestión de clientes, atención, transacciones y reportes.*
Relevancia: *Este flujo facilita la comprensión del funcionamiento integral del software.*

2.7 Algoritmos principales

Pseudocódigo:

Algoritmo 1: Agregar proceso (cliente nuevo)

```
ALGORITMO AgregarCliente
// Inserta un nuevo cliente en la lista enlazada de clientes

ENTRADAS:
    datosCliente ← información del nuevo cliente

VARIABLES:
    nuevoNodo, actual

INICIO
    nuevoNodo ← CrearNodo(datosCliente)

    SI (listaClientes está vacía) ENTONCES
        listaClientes.inicio ← nuevoNodo
    SINO
        actual ← listaClientes.inicio
        MIENTRAS (actual.siguiente ≠ NULO) HACER
            actual ← actual.siguiente
        FIN MIENTRAS
        actual.siguiente ← nuevoNodo
    FIN SI

    MOSTRAR("Cliente agregado correctamente")
FIN
```

Figura PX. Pseudocódigo de Registro de Cliente

Descripción:

El pseudocódigo muestra el proceso para registrar un nuevo cliente en la lista enlazada. Incluye la lectura de datos, validación de DNI existente y la inserción del nodo al inicio o al final según corresponda.

Algoritmo 2: Cambiar estado de atención

```
ALGORITMO AtenderCliente
// Desencola un cliente y registra su atención

VARIABLES:
    clienteActual

INICIO
    SI (colaAtencion está vacía) ENTONCES
        MOSTRAR("No hay clientes en espera")
    SINO
        clienteActual ← Desencolar(colaAtencion)
        MOSTRAR("Atendiendo a: ", clienteActual.nombre)

        // Registrar transacción en la pila de historial
        Push(pilaHistorial, clienteActual.operacion)

        MOSTRAR("Operación registrada con éxito")
    FIN SI
FIN
```

FGigura PW. Pseudocódigo de Registro de Transacciones

Descripción:

El pseudocódigo explica cómo se registra una transacción para un cliente. Se ingresan los datos de la operación y luego se agrega a la pila de historial mediante una operación push.

Algoritmo 3: Registrar transacción

```
ALGORITMO RegistrarTransaccion
// Guarda la operación realizada por un cliente en la pila

ENTRADAS:
    idCliente, tipoOperacion, monto

VARIABLES:
    nuevaTransaccion

INICIO
    nuevaTransaccion ← CrearTransaccion(idCliente, tipoOperacion, monto)
    Push(pilaTransacciones, nuevaTransaccion)
    MOSTRAR("Transacción registrada para el cliente ", idCliente)
FIN
```

Descripción:

El pseudocódigo describe cómo un cliente es incorporado a la cola de atención del sistema. Se identifica su prioridad y se inserta en una cola FIFO o una cola de prioridad según el tipo definido.

2.1. Justificación del diseño

El diseño del sistema se adhiere al principio de **modularidad** y **bajo acoplamiento**, lo cual resulta fundamental para la **comprensión, escalabilidad y mantenimiento** del *software*. La arquitectura modular permite la **separación de intereses (*Separation of Concerns*)** al encapsular funcionalidades específicas (e.g., gestión de clientes, *scheduling* de peticiones) en clases o módulos independientes, facilitando las futuras **modificaciones y pruebas unitarias**.

Optimización del Rendimiento y Fidelidad Operacional

El uso de **estructuras de datos dinámicas lineales** (Listas Enlazadas, Colas y Pilas) es una decisión técnica clave para la **optimización del rendimiento algorítmico**. Esta elección asegura una **reducción significativa de la complejidad temporal** ($\mathbf{O(1)}$ en casos óptimos) para las operaciones fundamentales de **inserción y eliminación de datos**, superando la ineficiencia que presentan los *arrays* estáticos al requerir reasignación o desplazamiento masivo de elementos.

Además, el enfoque de simulación busca una **alta fidelidad** con los procesos de atención reales en un entorno bancario. Al modelar las colas de espera y el historial de transacciones con precisión estructural, la simulación actúa como un **gemelo digital funcional**, lo que permite **validar empíricamente** la eficacia de los algoritmos de *scheduling* propuestos antes de una implementación en producción.