Algoritmos y estructura de datos

Curso: K1004

TRABAJO PRÁCTICO Nº: 1

PROFESOR:
Ing. Pablo D. Mendez

***TÍTULO*:**
Prototipo – Banco Interamericano

UNIDADES PRINCIPALES:
Archivos – Structs – Arrays

INTEGRANTES		
Rincón, Martín	174.347-8	martinrincon04@gmail.com
Bloemer, Eliel	174.521-9	elielbloemercorrea@gmail.com
Campi, Lucas Ezequiel	174.341-7	lucas.ezequiel001@gmail.com

Ciclo lectivo 2020

ÍNDICE:

- Introducción……(2
  Herramientas utilizadas.
  División del trabajo.

- Programa-Main……(4
  Structs - arrays.
  Procesamiento de información.
  Menú interactivo.

- Subprogramas principales……(8
  Búsqueda binaria.
  Creación de clientes.
  Ordenamientos.

- Cierres de programa y guardado de datos……(13
  Borrado lógico y físico
  Procesamiento de lotes y movimientos

Introducción

Herramientas utilizadas

Para poder llevar a cabo el desarrollo del código prototipo. Nos valimos de las herramientas Codeblocks y ATOM.

Codeblocks, fue utilizado mayormente para la adecuada compilación de código y creación de archivos del programa, siendo éste muy seguro para evitar errores durante la creación inicial y el trabajo continuo con los mismos.

ATOM, es un software, en conjunto con GitHub, que nos permite, por un lado, desarrollar código de distintos lenguajes de alto nivel (En nuestro caso C y C++), aportándonos un editor de texto y múltiples herramientas de compilado y ejecución de programas. A la hora de la escritura del código, utilizamos éste programa, ya que cuenta con la posibilidad de la apertura de un servidor, para que nosotros podamos trabajar en tiempo real y en conjunto.

Y por último, diagrams.net para diseñar los diagramas de Lindsay.

División del trabajo

Gracias a la herramienta ATOM, pudimos llevar a cabo una organización más enfocada en el trabajo en grupal y en la opinión directa con cada integrante del grupo.

Cada uno llevó a cabo el desarrollo de distintos subprogramas, con la posibilidad de que pueda ser rápidamente asistido por uno de los integrantes y así poder ahorrar tiempo en pruebas y correcciones.

En general, muchos subprogramas son similares. Fuimos creando cada tipo de subprograma (en base a lo necesitado) y luego por separado fuimos desarrollando el resto de los subprogramas necesitados basándonos en el primero. Esto nos permitió ganar más tiempo, entre otras cosas. Lo mismo ocurre con el desarrollo del main.

Programa - Main

Hemos dejado para el programa principal, la invocación de funciones en determinados **Case** de un switch el cual interactúa directamente con un menú.

Creamos dos structs, uno que contendrá los datos de los clientes:

```
struct Tarjeta
{
int TarjetaID[3] = {0,0,0};
int FechaCreacion;
bool  activa = false;
float Saldo=0;
int Nrocliente;
};
```

Algunas cuestiones a mencionar sobre el mismo:

- El máximo de cuentas que puede tener una persona (legalmente) en el banco, debería ser de tres. Cada cliente puede tener habilitada: Una cuenta corriente, una caja de ahorro y una cuenta en dólares. Las características de éstos tipos de cuenta no son desarrolladas en éste trabajo. Ésta cuestión, fue desarrollada por un **int TarjetaID[3]**, inicializada totalmente en 0, para que los subprogramas posteriores puedan desarrollar la determinada búsqueda de espacios/slots vacíos.
- Cada cuenta viene definida por una fecha de creación para aportar información a la misma. Utilizamos **int FechaCreacion.**
- El elemento **bool activa**, lo utilizamos para la creación de clientes. Durante el desarrollo, el usuario tendrá la posibilidad de creación de clientes y cuentas para cada cliente. Más adelante, explicaremos que durante la creación de clientes se van a habilitar

a los mismos y sus cuentas, en su totalidad, figurarán como ACTIVAS, luego el usuario tendrá la posibilidad de desactivar, y activar en el caso que sea necesario, cada cuenta para que finalmente sean retirados de la base de datos utilizada, si éstos se encuentran como clientes DESACTIVOS a la hora de cerrar el programa.

- Finalmente, un **float Saldo** que hace referencia al saldo en la totalidad (teniendo en cuenta las tres cuentas que puede tener cada cliente) y el número de cliente con el que podremos identificar a todos los mismos

Para el desarrollo de movimientos y procesamiento de lotes utilizamos:

```
struct Movimientos
{
int MovimientoID;
int FechaHora;
float Monto;
int TarjetaID[3] = {0,0,0};
};
```

Del cual podemos agregar el número de movimiento, la fecha y el monto del mismo.

De los mismos, utilizamos un vector del Struct Tarjeta que será el encargado de recibir la información del archivo, recibir modificaciones por parte del usuario y sobrescribir al archivo al final.

A su vez, creamos una variable única del mismo tipo, que le aportara ayuda al vector principal en algunas cuestiones que mencionaremos más adelante.

Procesamiento de información

Una de las cuestiones más importantes, para el procesamiento y apertura de archivos, era el cómo iban a ser cargados en el vector, siendo importante entender que, si bien ahora trabajamos con un prototipo, debemos pensar siempre al sistema como una complejidad y con tamaños aún mayores a los utilizados en el desarrollo de éste programa.

Si bien la utilización de un vector trae muchas complicaciones, debido en que en algunos casos podríamos excedernos de la capacidad que puede tener un array o que puede soportar una computadora cualquiera, muchos de los subprogramas que desarrollamos, nos facilitan más la navegación por los datos, ya sea para buscar un dato o para realizar los ordenamientos correspondientes.

Calculamos aproximadamente cuánta memoria podría consumir el programa durante su ejecución. Cada elemento del struct más pesado pesa 28 bytes, en un array de 10000 elementos, es decir, posiblemente no termine siendo un programa pesado en la ejecución y quizás se pueda extender a más de 10000 elementos, teniendo en cuenta al programa final (No el prototipo). Y este es el único caso donde utilizamos un vector, para los movimientos, por ejemplo, se lee el archivo necesario leyendo elemento por elemento

Siempre que se ejecuta el programa, automáticamente procesa los datos del archivo “Cuentas.BIC” y carga el vector con todos los clientes almacenados en él. Cuando no sea posible realizar el procesamiento de datos, se le informará al usuario y el programa finalizará.

Menú interactivo

Para poder desplazarnos por las distintas opciones del programa, desarrollamos un procedimiento cuyo objetivo es mostrar por pantalla las distintas opciones posibles. A su vez, el main en su mayoría está compuesto por un switch que irá recorriendo las opciones hasta que el programa se finalice, donde se va a reemplazar el archivo original con uno auxiliar que tendrá los datos actualizados y será renombrado como Cuentas.BIC.

También, una de las opciones importantes dentro del menú, es la del procesamiento de lotes con movimientos de cuentas. Cada lote realizará la modificación correspondiente a cada cuenta siempre y cuando ésta se encuentre activa, en el caso de que esto no ocurra, se le informará al usuario por pantalla el estado actual de la cuenta ACTIVADA o DESACTIVADA. El usuario podrá acceder, ente las opciones del menú, al estado de cada cliente, pudiendo activarla o desactivarla según sea necesario.

Subprogramas principales

Dentro de la mayoría de subprogramas, tenemos para mencionar los relacionados con la creación de clientes, búsqueda entre clientes por número de cliente o número de tarjeta, ordenar por un criterio a los clientes que figuran, etc.

Búsquedas binarias

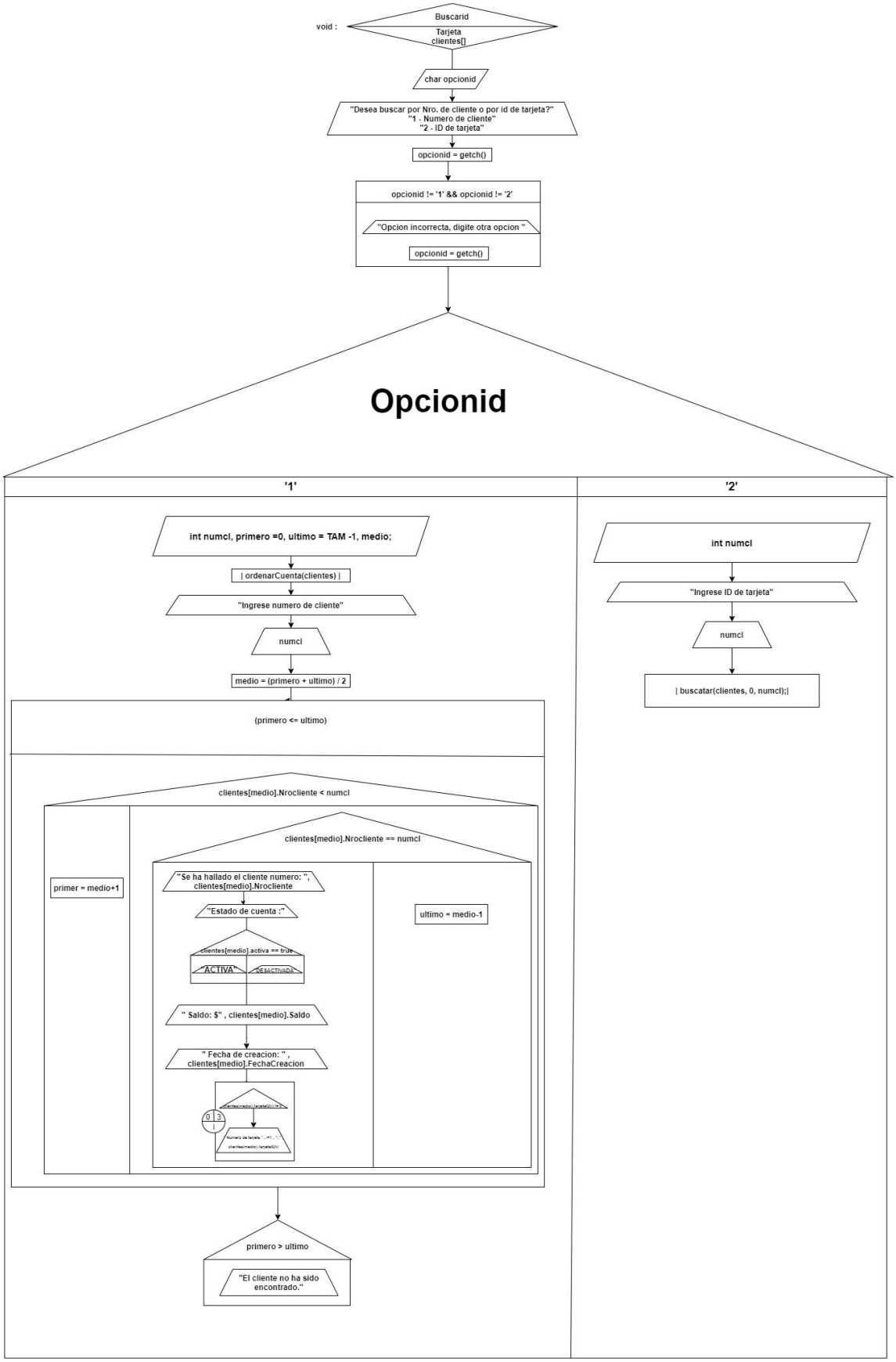
Aclaración: **Por el foro nos escribió la corrección de no utilizar búsqueda binaria de manera recursiva para la búsqueda de cuentas ya que consume más memoria, en general. No nos dio tiempo a corregir esto y ya tuvimos que entregar el TP. Pero consideramos que está bueno mencionarlo con esta aclaración.**

Para todo tipo de búsquedas necesarios durante el algoritmo, desarrollamos una búsqueda binaria como se podrá ver en el próximo Lindsay. El objetivo es ahorrar tiempo en búsqueda e informar si lo que se busca se encuentra en la base de datos o no.

Para la búsqueda de cuentas, desarrollamos el mismo algoritmo, pero de manera recursiva, es decir, como cada cliente puede tener un máximo de tres cuentas (esto se encuentra desarrollado con un vector de tamaño 3), el programa irá invocándose dentro de sí mismo hasta que encuentre en alguna de las posiciones a la cuenta buscada o el algoritmo llegue a su corte y este informe al usuario que la cuenta no ha sido encontrada.

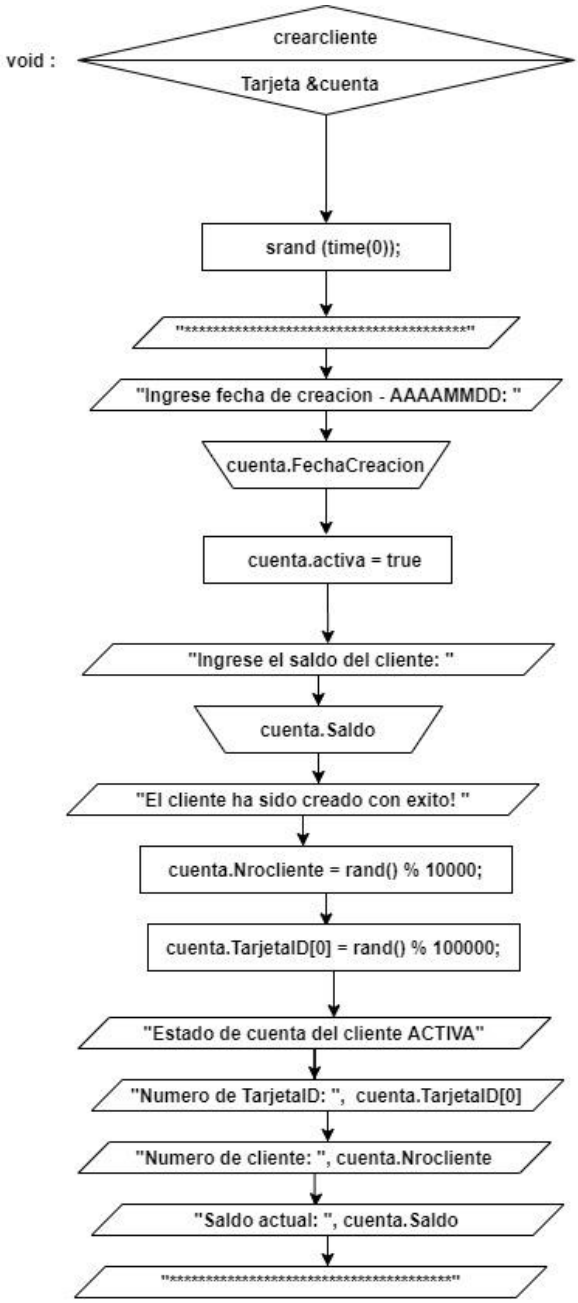
De la misma manera, también sirve para la desactivación u activación de cuentas. El programa pedirá el número de cliente correspondiente y desactivará la cuenta si esta se encuentra desactivada y viceversa.

A continuación, el Lindsay del procedimiento **buscarid**. La realidad es que no desarrollamos los Lindsay para los casos de desactivación y búsqueda de cuentas porque es el mismo algoritmo (adaptado a cada caso) dónde lo único que se cambia son los criterios y en el caso de la búsqueda de cuentas, que sea recursivo.


void :
Buscarid
Tarjeta
clientes[]
char opcionid
"Desea buscar por Nro. de cliente o por id de tarjeta?"
"1 - Numero de cliente"
"2 - ID de tarjeta"
opcionid = getch()
opcionid != '1' && opcionid != '2'
"Opcion incorrecta, digite otra opcion "
opcionid = getch()
Opcionid
'1'
'2'
int numcl, primero =0, ultimo = TAM -1, medio;
| ordenarCuenta(clientes) |
"Ingrese numero de cliente"
numcl
medio = (primero + ultimo) / 2
(primero <= ultimo)
clientes[medio].Nrocliente < numcl
clientes[medio].Nrocliente == numcl
primer = medio+1
"Se ha hallado el cliente numero: ", clientes[medio].Nrocliente
"Estado de cuenta :"
clientes[medio].activa == true
"ACTIVA"
"DESACTIVADA"
" Saldo: $" , clientes[medio].Saldo
" Fecha de creacion: " , clientes[medio].FechaCreacion
ultimo = medio-1
primero > ultimo
"El cliente no ha sido encontrado."
int numcl
"Ingrese ID de tarjeta"
numcl
| buscatar(clientes, 0, numcl);|

Creación de cuentas o clientes:

Es un procedimiento más corto, al invocarse, pide la fecha de creación y saldo inicial. Finalmente, se encarga de otorgarle al nuevo cliente un número de tarjeta, que completa el primer espacio posible de 3, y número de cliente aleatorio dado por el sistema. Además, activa la cuenta para su uso. Estos son informados al usuario para llevar un breve registro del mismo.

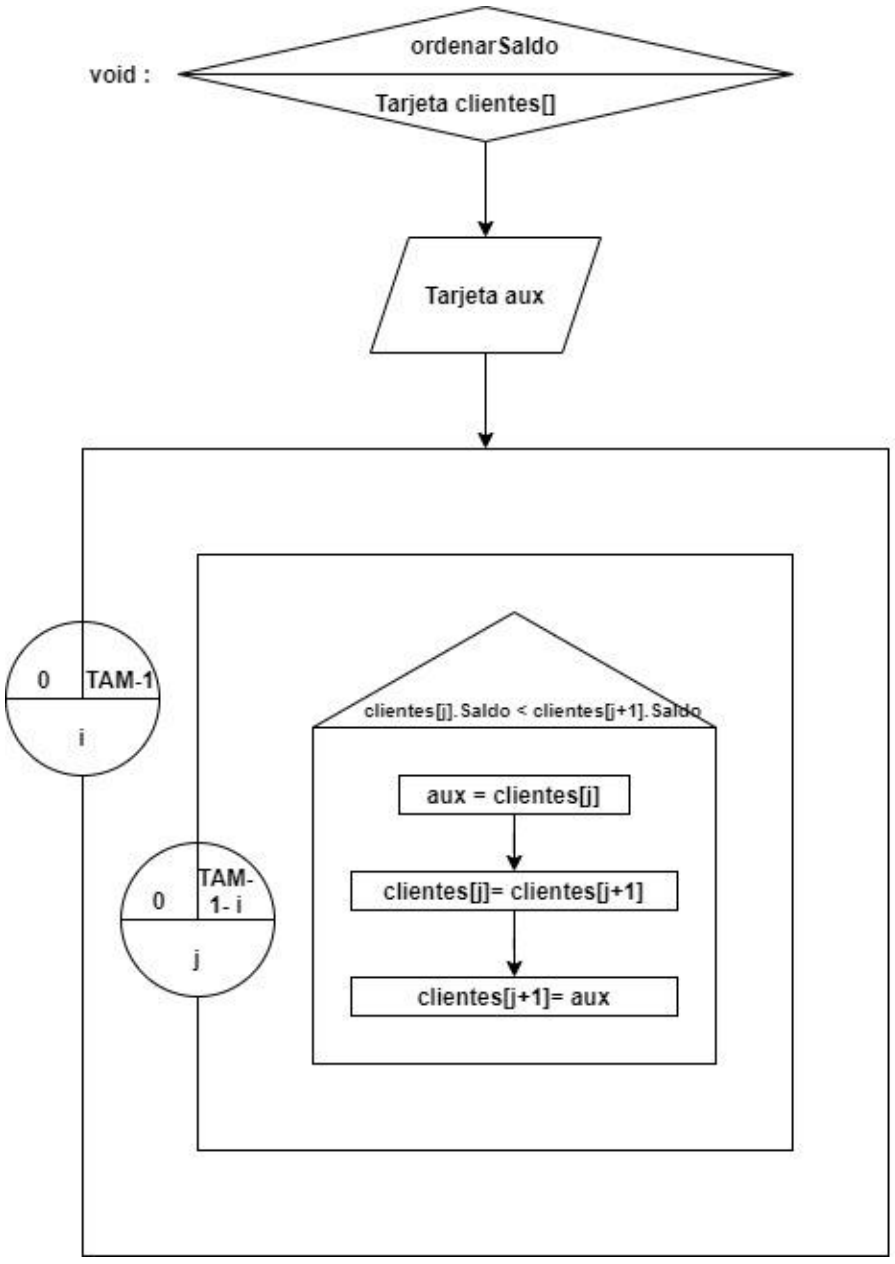


Para la creación de cuentas, el procedimiento es el mismo. El sistema pide el número de cliente a quien se le va a crear la cuenta nueva. Realiza una búsqueda binaria y completará los espacios posibles para las tarjetas, siempre que se encuentren vacíos. En el caso de que la capacidad máxima de cuentas sea alcanzada y el usuario quiera adicionar una más, entonces el sistema informará que no es posible crear una cuenta nueva.

Ordenamientos

Para realizar las búsquedas correspondientes en cada caso, era importante crear un ordenamiento según cada criterio pedido. Nos basamos en el método de burbuja mejorado. Se irán ordenando los structs completamente en base a lo necesitado.

Todos los ordenamientos son similares, mostramos a continuación el utilizado para listar los saldos de manera descendente.



Cierres de programa y guardado de datos

Borrado lógico y físico.

Fue una de las cuestiones más importantes y de lo que más nos ha ayudado con el vector.

Para el borrado lógico, nos basamos en la desactivación de cuentas. Cada vez que una cuenta es desactivada, no se puede interactuar con ella, es decir, por ejemplo, en el procesamiento de lotes de movimientos (informaremos más adelante), cuando se quiere cargar o extraer el saldo correspondiente a una cuenta desactivada, no se podrá realizar la operación.

Para el borrado físico, sobreescribimos el archivo original (modo wb) y cargando solo los datos de las cuentas que se encuentran activas.

El algoritmo:

```
ct = fopen("Cuentas.BIC","wb");
   for(int i = 0; i < TAM; i++)
   {
      if(clientes[i].activa == true )
      {
          cargadatos = clientes[i];
          fwrite(&cargadatos,sizeof(Tarjeta),1,ct);
      }
   }
   fclose(ct);
          cout << "El programa ha guardado y finalizado correctamente. " << endl;
```

Procesamiento de lotes y movimientos

Para el procesamiento, es necesario que se encuentre creado el archivo correspondiente a los movimientos, denominados “lote(numero). BIC”

Para este caso y para asegurarnos de que haya un buen procesamiento de datos no hemos creado un vector, principalmente porque puede haber muchos más movimientos que clientes activos a la hora del procesamiento. En este caso, creamos un buffer unitario que lee elemento por elemento y va informando al usuario los movimientos realizados posibles y los que no se pudieron realizar, a su vez, los movimientos realizados son guardados en un archivo llamado “procesados.BIC”, con el fin de llevar un registro de los mismos.

El algoritmo:

```
      fread(&procesados,sizeof(Movimientos),1,ct);
while(!feof(ct))
{
fread(&procesados,sizeof(Movimientos),1,ct);
pos = postar(clientes,0,procesados.TarjetaID);
if (clientes[pos].activa){
cout << "*****************************" << endl;
cout << "Fecha de operacion: " << procesados.Fecha << endl;
cout << "Numero de operacion: " << procesados.MovimientoID << endl;
cout << "Cuenta destino: " << procesados.TarjetaID << endl;
cout << "Saldo inicial: $" << clientes[pos].Saldo << endl;
      clientes[pos].Saldo += procesados.Monto;
cout << "Saldo final: $" << clientes[pos].Saldo << endl;
cout << "Valor de la operacion: $" << procesados.Monto << endl;
cout << "*****************************" << endl;
```

```
    cout << endl;

    fwrite(&procesados,sizeof(Movimientos),1,save);

    }

    else

        cout << " o el cliente " << clientes[pos].Nrocliente << " ha desactivado la
cuenta" << endl;

        cout << endl;

    }

    cout << endl;

    cout << "*******************************" << endl;

    cout << "Lote procesado exitosamente." << endl;

    cout << "*******************************" << endl;

    fclose(save);

    fclose(ct);
```