

Carrera

Ingeniería en Computación

Sistemas de créditos

Materia

Algoritmo y Programación 3

Alumno

Gomez Agustin – gomez.agustin.30602@gmail.com

Docentes:

Claudio, Aciti

Martin, Rodríguez

**Resumen**

*La abstracción es una capacidad intelectual que poseemos los seres humanos que nos permite concentrarnos en los aspectos importantes de un problema, dejando de lado los detalles irrelevantes. En este trabajo se explicará cómo se desarrolla un sistemas de créditos, en lenguaje c, utilizando tipos de datos abstractos conocidos como TDA*

**Índice General**

**1. Introducción** [**4-5**](https://docs.google.com/document/d/1ZnIB4JToPiIjkNvpF8HcJGiZh6SGW79rklTaqf8q7Ac/edit#heading=h.gjdgxs)

**2. Desarrollo experimental 6-8**

**3. Resultados y análisis 8-9**

**4. Conclusiones 9**

**5. Referencias 10**

## 

## **1.Introducción**

La abstracción es una capacidad intelectual que poseemos los seres humanos que nos permite concentrarnos en los aspectos importantes de un problema, dejando de lado los detalles irrelevantes. Esta capacidad nos permite, asimismo, dividir el problema en partes más pequeñas y, en consecuencia, más fácilmente tratables, para luego integrar todo en una solución. Es decir, es una capacidad humana que nos permite lidiar con la complejidad.

Cuando programamos resolvemos un problema y creamos un programa que es una abstracción de la realidad; en consecuencia, es importante contar con mecanismos de programación que favorezcan la abstracción. Si analizamos los lenguajes de programación convencionales, podemos ver que cuentan con un mecanismo que ayuda a la abstracción: la modularidad. Así, cuando hacemos uso de un módulo (por ejemplo, una función en C) en un programa, uno sólo se debe preocupar por conocer qué es lo que este módulo hace, es decir, qué funcionalidad provee, sin preocuparse por saber cómo lo hace. Además, la modularidad nos proporciona un medio para descomponer una tarea en partes más pequeñas. Es decir, la modularidad en programación, está bastante cerca de capturar el significado de abstracción. Sin embargo, este tipo de abstracción, referida como abstracción procedural, no es suficiente. En programación, existe también lo que se conoce como abstracción de datos, que nos lleva al concepto de Tipo de Dato Abstracto (TDA). Un TDA define datos de forma abstracta caracterizados por las operaciones que se pueden llevar a cabo sobre ellos. Esto significa que podemos decir que un TDA puede ser definido definiendo las operaciones que lo caracterizan. [1]

Quien usa el TDA debería poder interactuar solamente con las características públicas del TDA, sus operaciones características, y no tener acceso a la implementación de este. Para separar la interfaz del TDA, es decir aquellas funciones y tipos que el programador "usuario" utilizará, de los detalles de implementación, vamos a utilizar los archivos headers de C. Es decir, los .h. Por lo tanto:

la interfaz pública de utilización será declarada en el archivo .h, mientras que la implementación interna de los objetos de datos será declarada en el archivo .c

, y el programa de prueba o cliente deberá simplemente incluir el .h, por ejemplo

#**include** "lista.h"

Dentro del .h vamos a definir:

* Los tipos: **typedef** y **structs**
* Los "prototipos" de las **funciones** (es decir, las firmas, pero no la implementación de las funciones).

En él .h:

* Se define un "**struct**" vacío
* Se define un "**typedef**" de puntero a **struct**
* Todas las funciones van a hacer referencia al tipo "**typedef**"

Luego él .c va a definir un nuevo **struct** con el mismo nombre que el definido en el .h de esta forma se va a "sobrescribir" el original vacío por el nuevo que sí tendrá los datos. Por ejemplo:

en él .h

**struct** ListaStruct;

**typedef struct** ListaStruct\* Lista;

// funciones

y en él .c

**struct** ListaStruct {

**int** longitud;

**int**\* vector;

};

Como vemos ambos **structs** se llaman iguales. Solo que el segundo tiene realmente datos, cuando el primero está vacío. Como los usuarios del TDA compilan contra él .h, solo verían el struct vacío. [2]

Lo que se hará en este trabajo será desarrollar un sistema de Créditos, con TDA.

Se debe permitir:

\*Dar de alta y listar clientes

\*Buscar clientes por nombre

\*Buscar clientes por rango de edad

\*Cada cliente tiene una referencia que puede ser otro cliente

\*Cada cliente puede solicitar un crédito con determinado monto

\*Dar de alta y listar créditos

\*Pagar créditos

\*Cancelar crédito.

El sistema debe poder soportar un gran listado de clientes y créditos. Dichos datos deben ser guardados en un archivo.

**2.Desarrollo experimental**

Para desarrollar el programa se necesitó de los siguientes archivos en formato TDA:

**\*lista**

**\*lista\_de\_creditos**

**\*Clientes**

**\*Creditos**

, cuyos métodos son:

**\* lista.h**

**typedef struct** Nodo {

Clientes cliente;

**struct** Nodo\* siguiente;

} Nodo;

**typedef struct** lista {

Nodo\* inicio;

Nodo\* fin;

**int** tam;

} lista;

* **void inicializacion (lista\* lista):** inicializa la lista de clientes
* **int ins\_inicio\_lista (lista\* lista, Clientes\* cliente)**: inserta un cliente al inicio de la lista.
* **void buscar\_clientes\_por\_nombre (lista\* a, lista\*b, char\* dato)**: busca clientes por el nombre y los guarda en una lista.
* **void buscar\_clientes\_por\_rango (lista\* a, lista\*b, int edad1, int edad2)**: buscar clientes por rango de edad y los guarda en una lista.
* **int sup\_inicio (lista\* lista)**: elimina el primer cliente al inicio de la lista.
* **int sup\_en\_lista (lista\* lista, int pos)**: elimina un cliente de la lista en la posición que no sea el inicio de esta.
* **void destruir (lista\* lista)**: elimina la lista de clientes.
* **void ver\_lista\_clientes (lista\* lista)**: imprime la lista de clientes.
* **void agregar\_credito\_a\_cliente (lista\*lista, char\* nombre, char\* apellido, Credito\* credito):** agrega crédito a un cliente, buscando dicho cliente por nombre y apellido.
* **void agregar\_credito\_a\_cliente2(lista\*lista, char\* nombre, Credito\* credito):**

agrega crédito a aquellos clientes que tengan el mismo nombre.

* **void agregar\_credito\_a\_cliente3(lista\*lista, int edad1, int edad2, Credito\* credito):**

agrega crédito a aquellos clientes que están en el rango de edad.

* **void pagar\_credito\_del\_cliente (lista \* lista, char\* nombre, char\* apellido, char\* motivo, double dinero)**: paga el crédito de un cliente.
* **void cancelar\_credito\_del\_cliente (lista \* lista, char\* nombre, char\* apellido, char\* motivo)** : cancela el crédito de un cliente.
* **void eliminar\_cliente\_de\_lista (lista\* lista, char\* nombre, char\* apellido)**:elimina un cliente de la lista.
* **void guardar\_lista\_de\_clientes (lista\* lista, char\* archivo):** Te guarda la lista de clientes en un archivo, mayormente de texto.

**\*lista\_de\_creditos.h**

typedef struct Nodo2 {

Credito credito;

**struct** Nodo2\* siguiente;

} Nodo2;

**typedef struct** lista2 {

Nodo2\* inicio;

Nodo2\* fin;

**int** tam;

} lista2;

* **void iniciar\_lista (lista2\* lista)** : inicializa la lista de créditos del cliente.
* **int ingresar\_credito (lista2\* lista, Credito\* credito)** : inserta un crédito al inicio de la lista del cliente.
* i**nt sup\_credito (lista2\* lista)**: elimina el credito ubicado al inicio de la lista del cliente.
* **int sup\_credito\_en (lista2 \* lista, int pos)**: elimina el credito de la lista del cliente, en cualquier posición que no sea al inicio.
* **void destruir\_lista (lista2\* lista)**: elimina la lista de créditos del cliente.

**\*Clientes.h**

**typedef struct \_Clientes** {

**char\*** nombre;

**char\*** apellido;

**int** edad;

**double** monto;

**double** total;

lista2 lista;

}Clientes;

* **void cliente\_init (Clientes\* this, char\* nombre, char\* apellido, int edad)** : inicializa un cliente
* **void eliminar\_cliente (Clientes\* this)** : elimina un cliente, o sea, libera la memoria de la variable cliente.
* **void imprimir\_cliente (Clientes\* this)**: imprime los datos del cliente.
* **void ver\_lista\_cre (lista2\* lista)**: imprime la lista de créditos del cliente.
* **void guardar\_cliente\_en\_archivo (FILE \*file, Clientes cliente)**: guarda los datos del cliente en un archivo de texto.

**\*Credito.h**

**typedef struct** \_Credito {

bool estado;

**double** prestamo;

**char**\* motivo;

} Credito;

* **void credito\_init (Credito\* credito, double prestamo,char\* motivo)**: inicializa un crédito.
* **void cancelar\_credito (Credito\* this)**: elimina un credito, ósea libera la memoria de la variable crédito.
* **void pagar\_credito (Credito\* this, double monto)**: paga el valor
* **void imprimir\_credito (Credito\* this)**: imprime los datos del crédito.
* **void guardar\_credito (FILE\* file, Credito\* this)**: guarda los datos del crédito en un archivo.

## **3.Resultados y análisis**

Se utiliza un archivo llamado **main.c** para acceder a los métodos de los archivos en formato TDA, logrando así el modelo del sistema de créditos. El modelo consiste en un menú con múltiples opciones. La idea del menú se optó para que el sistema sea más fácil de manejar y sea más ordenado, debido a que cada opción es una combinación de métodos.

El menú consta de 9 opciones:

1.- agregar cliente a la lista

2.- eliminar cliente de la lista

3.- agregar crédito al cliente

4.- pagar crédito del cliente

5.- cancelar crédito del cliente

6.- imprimir lista

7.- guardar lista

8. -borrar lista

9. -salir

Al iniciar el programa, se declara y se inicializa automáticamente una lista, está se pone a disposición para albergar a los clientes. Dicha lista está en forma de pila, haciendo más fácil agregar más clientes en vez de darle un tope de capacidad con un array.

Con la primera opción, se agrega un cliente en la lista ya inicializada, guardando los datos del cliente que serían su nombre, apellido y edad. La segunda, recorre la lista de clientes buscando un cliente por medio de su nombre y apellido. Una vez encontrado, lo elimina de la lista. La tercera, agrega un crédito en la lista de un cliente, guardando los datos de este que serían estado, monto del préstamo y el motivo de dicho crédito.

Con la cuarta, se recorre la lista de clientes, buscando el cliente, por nombre y apellido. Una vez encontrado, recorre la lista de créditos buscando el crédito a pagar por medio del motivo, al encontrarlo deposita el dinero descontando el monto del crédito. El dinero depositado para pagar tiene que ser menor o igual al monto del crédito, si por casualidad es mayor al monto del crédito, no se podrá pagar dicho crédito. Una vez que el crédito se pagó completamente, es decir el valor del monto de crédito es cero, se cambia su estado de “no pagado” a “pagado”. La quinta, por medio del motivo, se recorre la lista de créditos del cliente, buscando el crédito que coincida con el motivo y que el estado esté en “pagado”. Una vez encontrado, lo elimina de la lista.

Con la sexta, se imprime la lista de clientes. Esta opción puede imprimir la lista de 3 formas: la 1era es imprimir la lista completa, la 2da es imprimir la lista de clientes que tengan el mismo nombre y la 3era es una lista de clientes que estén en un rango de edad. La séptima, por medio de un nombre, te crea un archivo de texto en el cual te deposita, la lista de clientes con sus respectivas listas de créditos. Si utilizamos el nombre de un archivo que contenía una lista de clientes para guardar otra lista de clientes, lo que va a hacer es sobre escribir el archivo guardando la lista debajo de la otra.

Con la octava, se borra la lista de clientes de forma completa. Una vez borrada la lista, es inicializada nuevamente, para volver a agregar clientes. Y, para terminar, la última opción finaliza el programa.

## **4.Conclusiones**

Por motivos de tiempo y de cómo desarrollarlo, no pude hacer que los clientes tengan una referencia con otros clientes. Por lo que si, por ejemplo, quiero agregar un cliente a la lista y dicho cliente tiene los mismos datos (nombre, apellido y edad) de otro cliente que está en la lista, entonces el programa lo agregara a la lista como si los dos clientes fuera distintos.

El programa cumple en su mayoría con lo que pide el enunciado, funciona correctamente, incluso en algunos casos específicos como, por ejemplo:

* si la lista posee un número grande de clientes, en vez de agregarles créditos uno por uno, se optó por agregar en la opción 3) dos formas de acreditar clientes por mayor. Una por nombre, acredita a todos los clientes de la lista que tenga el mismo nombre, y otra por rango de edad.
* para poder agregar más créditos al cliente, hay que saber que el monto del cliente es la suma de todos los montos(prestamos) de cada crédito, que llega hasta los 100000. Si el monto del cliente es 100000 entonces, hay que pagar algunos créditos de mismo.
* cuando un crédito tiene de estado “pagado” no se puede volver a usar. Opte por dejar los créditos pagados para notar con mayor facilidad los diferentes tipos de créditos que hay en la lista. Si deseo eliminar los “pagados” de la lista solo tendría que usar la opción 5).

En conclusión, se pudo hacer un Sistema de Créditos con TDA.

**Referencias**

[1]<http://proguno.unsl.edu.ar/t4_TDA.pdf>

[2]<https://sites.google.com/site/programacioniiuno/temario/unidad-2---tipo-abstracto-de-dato/implementacin-de-un-tda-en-c>