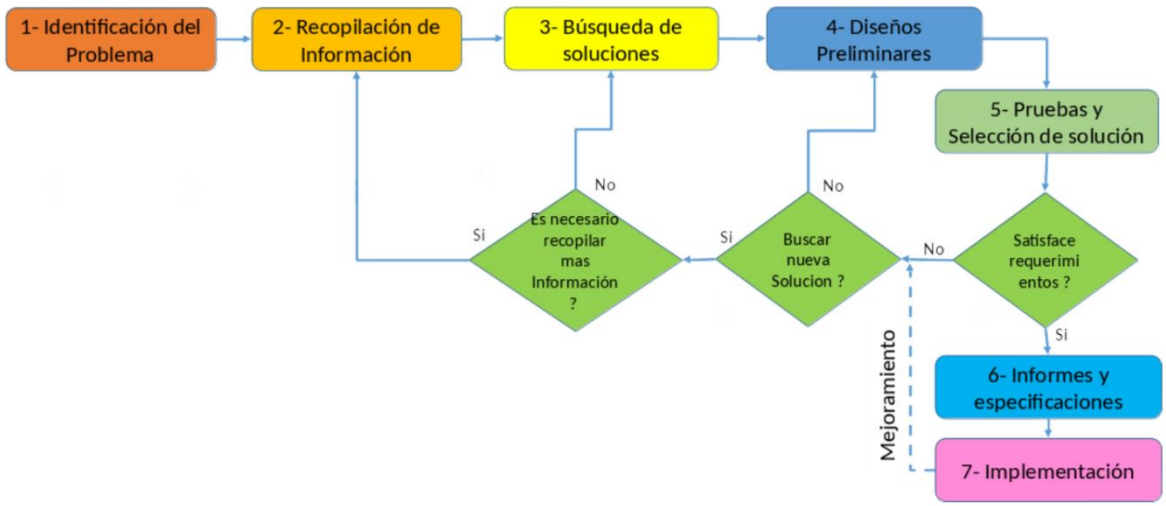

FASE 1: IDENTIFICACIÓN DEL PROBLEMA

Definición del problema

Una sucursal bancaria requiere el desarrollo de un módulo de software que permita gestionar las operaciones e información de los clientes y usuarios.

Identificación de necesidades y síntomas

-La sucursal requiere poder gestionar la recepción y atención de sus usuarios

-La solución al problema debe ser eficiente para que el servicio pueda mejorar sin importar la cantidad de clientes y usuarios que tenga la sucursal

-El acceso efectivo a su base de datos y la realización de optima de sus operaciones financieras.

FASE 2: RECOPILACIÓN DE LA INFORMACIÓN NECESARIA

Requerimientos para la solución del problema

-Buscar un cliente por el parámetro de elección

-Eliminar cuenta bancaria de un cliente por el parámetro de elección

-Deshacer la última acción financiera en caso de que exista un error al realizarla

-Crear cuenta bancaria para un cliente con los respectivos datos y pagar tarjeta de crédito del respectivo cliente.

Restricciones para la solución del problema

-Implementar interfaz gráfica con información de los clientes del banco a través de una tabla de datos.

-Implementación de métodos de ordenamiento, tres métodos de ordenamiento eficiente y uno que sea regular.

-Implementar interfaz gráfica que represente el estado actual de las filas de usuarios.

Definiciones

Sucursal Bancaria

Sede descentralizada de una entidad financiera.

Módulo Caja

Este es un módulo operacional que se encarga de gestionar las funcionalidades los usuarios, tales como registrar a un nuevo cliente, realizar operaciones sobre la cuenta de un cliente dado y efectuar pagos.

Módulo Fila

Es un módulo operacional que realiza la asignación de turnos de manera prioritaria a los usuarios que llegan a la sucursal.

Usuario

Persona que requiere un servicio financiero y posee un número de cédula y un nombre.

Cliente

Tipo de usuario que tiene una cuenta registrada en el banco, es decir, que tiene una cuenta para poder efectuar operaciones financieras.

Cuenta bancaria

Deposito donde se guarda el dinero que tiene o adeuda un cliente

Fila

Conjunto de usuarios a la espera de recibir un servicio bancario.

FASE 3: BÚSQUEDA DE SOLUCIONES CREATIVAS

Para este paso, pensamos soluciones propias dado que es un problema dirigido a un sector muy específico con características especiales y no se encuentran soluciones en fuentes bibliográficas

Solución asignación de turnos

Se propone modelar la asignación de turnos mediante características especiales de personas que requieren una atención prioritaria. Dichas filas se modelarían mediante estructuras de datos regida por los conceptos LIFO y FIFO, Stacks y Queue específicamente. Se implementará dos filas, una sin restricción alguna y otra para recibir a las personas en condición especial.

Solución a la gestión de la información de clientes.

Se implementaría el modelamiento de los clientes mediante distintas estructuras de datos con características especiales que permitan una inserción, búsqueda, ordenamiento modificación y eliminación de los mismos, las cuales podrían ser Hash Table, Heaps, BST, LinkedList, ArrayList, entre otros.

Solución consulta de información de los clientes

Se propone modelar una tabla tipo hoja de cálculo donde se visualice la información de los usuarios ordenando por medio de algoritmos como Insertion, QuickSort, MergeSort, BucketSort, BinaryTree Sort los parámetros de datos por nombre, cédula, tiempo de vinculación del cliente y monto de dinero, brindándonos mayor eficiencia al momento de consultar la información de los clientes por medio de la interfaz

Definición de las entidades del mundo del problema:

Usuario

Representación de una persona que requiere un servicio bancario, dicha persona posee un número único de identificación y un nombre.

Cliente

Representación de un usuario que está registrado en la base de datos del banco, el cual posee los mismos atributos del usuario y, además, una cuenta bancaria, tarjeta de crédito y débito, fecha de pago de la tarjeta crédito y fecha en la que se incorporó al banco.

Modulo caja

Representación de los comportamientos del cliente tales como agregar un cliente a la fila, realiza operaciones bancarias del cliente tales como realizar un pago, crear una cuenta bancaria, eliminar una cuenta bancaria, además conoce la persona que está siendo atendida actualmente y la siguiente que espera por atención

Modulo fila

Representación de las filas de espera, el cual contendrá un conjunto de usuarios en espera de atención. El módulo fila debe indicar el siguiente usuario a atender por su orden de llegada o prioridad de atención, además debe generar una representación gráfica del estado de las dos filas.

FASE 4: TRANSICIÓN DE LA FORMULACIÓN DE IDEAS A LOS DISEÑOS PRELIMINARES

Alternativas estructuras de datos

Alternativa 1: BST

Alternativa 2: Queue

Alternativa 3: Stack

Alternativa 4: LinkedList

Alternativa 5: Hash Table

Alternativa 6: Heaps

Alternativas implementaciones de ordenamiento

Alternativa 1: MergeSort

Alternativa 2: Quick Sort

Alternativa 3: Insertion Sort

Alternativa 4: Bucket Sort

Alternativa 5: HeapSort

Alternativa 6: Direct Hash Sorting

Alternativa 7: BinaryTree Sort

FASE 5: EVALUACIÓN Y SELECCIÓN DE LA MEJOR SOLUCIÓN

Criterios estructuración de los datos

- Criterio A: por efectividad en su complejidad temporal al momento de realizar operaciones sobre las mismas estructuras de datos
- Criterio B: por su eficiencia al momento guardar la cantidad de clientes que necesitáramos
- Criterio C: por la eficiencia al permitirnos simular una fila de clientes que se ordenaran según las condiciones especiales establecidas

Criterios implementación de ordenamientos

- Criterio A: por la eficiencia en su complejidad temporal al momento de realizar los ordenamientos
- Criterio B: por el tipo de estructura utilizada, si no se está implementando la estructura a la que pertenece el ordenamiento no se podrá utilizar.

Selección de alternativa para estructuración de los datos

Se optó por implementar todas las alternativas, menos la tercera de la siguiente manera: Se realizó el particionamiento de todos los clientes en cuatro subgrupos, los cuales se definieron con base a cuatro rangos de sus respectivos números de identificación. No se realiza la evaluación de los criterios anteriores dado que el problema nos permite elegir todas las opciones anteriores

Selección de alternativas para implementación de ordenamientos

De acuerdo a los criterios anteriores, se concluyó que se implementarían las alternativas de ordenamiento 3, 5, 6, 7

	Criterio A	Criterio B	Total
Alternativa 1	4	3	7
Alternativa 2	2	3	5
Alternativa 3	3	5	8
Alternativa 4	2	1	3
Alternativa 5	5	5	10
Alternativa 6	5	5	10
Alternativa 7	5	5	10

FASE 6: PREPARACIÓN DE INFORMES Y ESPECIFICACIONES

Especificación del problema

Problema: Desarrollar un módulo de software que permita gestionar las operaciones e información de los clientes y usuarios de la sucursal

Entradas: Usuarios o clientes con sus diferentes atributos

Salidas: Módulo de software eficiente que cumple con todos los requerimientos solicitados

Diagrama de clases

Class Diagram2

model

CashierModule

MyBSTTest
-myBst : MyBST<Client>
+setUp1() : void
+setUp2() : void
+addNodeTest1() : void
+addNodeTest2() : void
+removeNodeTest() : void
+searchNodeTest() : void
+searchNodeTest2() : void
+isEmptyTest() : void

ABBLibrary

-myBst

MyBST
+size : int
-arrayL : ArrayList<T>
<<Property>> -root : NodeBST<T>
+MyBST()
+addNode(dato : T) : void
+removeNode(dato : T) : void
+searchNode(dato : T) : NodeBST<T>
+getHeight() : int
+getWeight() : int
+isEmpty() : boolean
+inOrden(n : NodeBST<T>) : void
+preOrden(n : NodeBST<T>) : void
+postOrden(n : NodeBST<T>) : void
+generateArrayList() : ArrayList<T>

T : Comparable

root

NodeBST
<<Property>> +data : T
<<Property>> +repetitions : int
<<Property>> +right : NodeBST<T>
<<Property>> -left : NodeBST<T>
+NodeBST(t : T)
+isLeaf() : boolean
+getHeight() : int
+compareTo(n : Object) : int
+getWeight() : int
+insertNode(nuevo : NodeBST<T>) : void
+getLeast() : NodeBST<T>
+getBiggest() : NodeBST<T>
+search(datoBuscado : T) : NodeBST<T>
+remove(unDato : T) : NodeBST<T>
+inorden(listaAmount : ArrayList<T>) : void

left

right

<<Interface>>
IMyBST
<<Property>> +height : int
<<Property>> +weight : int
<<Property>> +empty : boolean
<<Property>> +root : NodeBST<T>
+addNode(dato : T) : void
+removeNode(dato : T) : void
+searchNode(dato : T) : NodeBST<T>
+inOrden(n : NodeBST<T>) : void
+preOrden(n : NodeBST<T>) : void
+postOrden(n : NodeBST<T>) : void
+generateArrayList() : ArrayList<T>

-dataBasePartitionA

HeapSortLibrary

HeapSort
V : Comparable<V>
+n : int
-theArray : GenericArray<V>
-sort() : void
-heapify(a : GenericArray<V>, n : int, i : int) : void
+sortArray(theArray : GenericArray<V>) : GenericArray<V>

-theArray

myHeapSort

GenericArray
V
GenericArray
-obj_array : Object[]
+length : int
+GenericArray(length : int)
+get(i : int) : t
+set(i : int, e : t) : void
+toString() : String

myGenericArray

HeapSortTest
-myHeapSort : HeapSort<Client>
-myGenericArray : GenericArray<Client>
+setUp1() : void
+setUp2() : void
+setUp3() : void
+sortTest1() : void
+sortTest2() : void
+sortTest3() : void
+sortTest4() : void

CashierModuleTest
-cashierModule : CashierModule
+setUp1() : void
+setUp2() : void
+signUpClientTest() : void
+unifyClientTest() : void
+searchTest() : void
+deleteClientAccountTest() : void

-cashierModule

CashierModule
<<Property>> -current : User
-dataBasePartitionA : IMyHashtable<Integer, Client>
-dataBasePartitionB : IMyBST<Client>
-dataBasePartitionC : IMyDoublyLinkedList<Client>
-dataBasePartitionD : Client
<<Property>> -desertersClients : Client
<<Property>> -allClients : Client
+CashierModule()
+signUpClient(theNew : User, balance : double, creditQuota : double, today : String, specialCondition : int) : void
-addClientToPartitionA(theNewClient : Client) : void
-addClientToPartitionB(theNewClient : Client) : void
-addClientToPartitionC(theNewClient : Client) : void
-addClientToPartitionD(theNewClient : Client) : void
+unifyClients() : ArrayList<Client>
+searchClient(id : int) : Client
+insertionSort() : void
+deleteClientAccount(id : int) : boolean
+doPay(idUser : int, amount : double) : void

CashierModule

QueueModule

QueueLibrary

Element
V
<<Property>> -v : V
<<Property>> -nextElement : Element<V>
+Element()

-nextElement

-first

-last

Queue
V
<<Property>> +size : int
-first : Element<V>
<<Property>> -last : Element<V>
+Queue()
+isEmpty() : boolean
+enqueue(item : V) : void
+front() : Element<V>
+dequeue() : Element<V>

<<Interface>>
IQueue
V
<<Property>> +size : int
<<Property>> +empty : boolean
<<Property>> +last : Element<V>
+enqueue(item : V) : void
+front() : Element<V>
+dequeue() : Element<V>

+simpleQueue

PriorityQueueHeapTest
-myPriorityQueueHeap : PriorityQueueHeap
+setUp1() : void
+setUp2() : void
+setUp3() : void
+enqueueTest1() : void
+enqueueTest2() : void
+dequeueTest1() : void
+dequeueTest2() : void

PriorityQueue

-myPriorityQueueHeap

PriorityQueueHeap
+lastIndex : int
<<Property>> -heap : User[]
+PriorityQueueHeap(size : Integer)
+enqueue(value : User) : void
-goUp() : void
-goDown() : void
-parent(index : int) : int
-left(index : int) : int
-right(index : int) : int
-swap(index1 : int, index2 : int) : void
+dequeue() : User

priorityQueue

QueueModule
+countSimpleQueue : int
+countPriorityQueue : int
<<Property>> +simpleQueue : IQueue<User>
+priorityQueue : PriorityQueueHeap
+QueueModule()
+receivePerson(p : User) : boolean
+getCurrentState() : String[]
+returnQueue(opcion : int) : String
+getCurrentWithPriority() : User
+getCurrentWithPriority2() : User
+getCurrentWithoutPriority() : User
+getCurrentWithoutPriority2() : User
-dequeueCurrentWithoutPriority() : void
-dequeueCurrentWithPriority() : void

qm

-queueModule

QueueModuleTest
-qm : QueueModule
+setUp1() : void
+setUp2() : void
+receivePersonTest() : void

BankController
-clientsTV : TableView<Client>
-nameTC : TableColumn<Client, String>
-idTC : TableColumn<Client, String>
-genderTC : TableColumn<Client, String>
-balanceTC : TableColumn<Client, String>
-creditCuotaTC : TableColumn<Client, String>
-openingDateTC : TableColumn<Client, String>
-specialConditionTC : TableColumn<Client, String>
-nameUserLbl : TextField
-idUserLbl1 : TextField
-norSpecialCRB : RadioButton
-specialCondicions : ToggleGroup
-ancientSpecialCRB : RadioButton
-discapacitySpecialCRB : RadioButton
-pregnancySpecialCRB : RadioButton
-priorityLabel : Label
-queueLabel : Label
-mainPane : Pane
-girlRB : RadioButton
-gender : ToggleGroup
-boyRB : RadioButton
-currentNameTF : TextField
-idCurrentTF : TextField
-amountTF : TextField
-currentDateTF : TextField
-limitCreditCardTF : TextField
-genderCurrentTF : TextField
-createAccountButtonID : Button
-idToDeleteTF : TextField
-deleteAccountFinishButton : Button
-nameToDeleteTF : TextField
-amountToDeleteTF : TextField
-dateCreationToDeleteTF : TextField
-undoDeleteAccountButton : Button
-currentUserLbl : Label
-valuePay : TextField
-msgQueue : String
-msgPriority : String
-seleccion : int
-idToDoPayTF : TextField
-accountToDoPayTF : TextField
-doPayButton : Button
-doPayCreditCardButton : Button
-IDToPayCreditCardTF : TextField
-AccountToPayCreditCardTF : TextField
-nameSearchTF : TextField
-idToSearchTF : TextField
-genderSearchTF : TextField
-balanceSearchTF : TextField
-creditSearchTF : TextField
-creationSearchTF : TextField
-specialConditionSearchTF : TextField
-userInformation : Pane
-nameCurrentUser : TextField
-idCurrentUser : TextField
-specialCurrentUser : TextField
-genderCurrentUser : TextField
-nameNextUser : TextField
-idNextUser : TextField
-specialNextUser : TextField
-genderNextUser : TextField
-queueType : int
-t : String
-amount : double
-limitGenerated : double
-today : String
-date : Date
-queueModule : QueueModule
-cashierModule : CashierModule
-current : User
-toDelete : Client
+initializeTableView() : void
+BankController()
-nextUserSimple(event : ActionEvent) : void
-nextUserPriority(event : ActionEvent) : void
-verifyID(event : ActionEvent) : void
-createAccountAccount(event : ActionEvent) : void
-deleteClientAccount(event : ActionEvent) : void
-doPayOption(event : ActionEvent) : void
-payCreditCard(event : ActionEvent) : void
-searchAClient(event : ActionEvent) : void
-seeClientsDataBase(event : ActionEvent) : void
-assignTurn(event : ActionEvent) : void
~userInformationQueue(event : ActionEvent) : void
-enqueueUserBtn(event : ActionEvent) : void
-loadAmountButton(event : ActionEvent) : void
-createAccountButton(event : ActionEvent) : void
-deleteAccountFinishButton(event : ActionEvent) : void
~undoDeleteAccountButton(event : ActionEvent) : void
-searchClientButton(event : ActionEvent) : void
-doPayButton(event : ActionEvent) : void
-verifyIDtoDoPay(event : ActionEvent) : void
-doPayCreditCardAccion(event : ActionEvent) : void
-verifyIDToPayCreditCard(event : ActionEvent) : void
-searchClientButtonAction(event : ActionEvent) : void

QueueTest
-myQueue : User
+setUp1() : void
+setUp2() : void
+isEmptyTest() : void
+sizeTest() : void
+dequeueTestLimits() : void
+dequeueTestCaseBase() : void
+enqueueTestInteresting Case() : void
+enqueueLastTest() : void

-myQueue

User
<<Property>> -name : String
<<Property>> -id : int
<<Property>> -specialCondition : int
<<Property>> -specialConditionTableView : String
<<Property>> -genderTableView : String
<<Property>> -gender : boolean
+User(name : String, id : int, gender : boolean, specialCondition : int)
+isSpecialCondition() : int
+compareTo2(u : User) : int

heap

-currents

-current

Main
-bc : BankController
+Main()
+main(args : String[]) : void
+start(primaryStage : Stage) : void

-bc

-toDelete

Client
<<Property>> -balance : double
<<Property>> -creditQuota : double
-limit : double
<<Property>> -registrationDate : String
+Client(name : String, id : int, gender : boolean, balance : double, creditQuota : double, today : String, specialCondition : int)
+compareTo(u : Client) : int

-desertersClients

-allClients

LinkedListLibrary

-dataBasePartitionC

<<Interface>>
IMyDoublyLinkedList
E : Comparable
<<Property>> +empty : boolean
+size() : int
+add(index : int, element : E) : void
+add(element : E) : boolean
+get(index : int) : E
+remove(index : int) : E
+set(index : int, element : E) : E
+generateArrayList() : ArrayList<E>

-dataBasePartitionD

LLNode
E : Comparable
-data : E
-prev : LLNode<E>
<<Property>> -next : LLNode<E>
+LLNode(e : E)
+linkAfter(theNew : LLNode<E>) : void
+compareTo(n : Object) : int

next

prev

head

tail

emptyList

MyDoublyLinkedList
E : Comparable
+size : int
-head : LLNode<E>
-tail : LLNode<E>
+MyDoublyLinkedList()
+add(element : E) : boolean
+get(index : int) : E
+add(index : int, element : E) : void
+size() : int
+remove(index : int) : E
+set(index : int, element : E) : E
+isEmpty() : boolean
+generateArrayList() : ArrayList<E>

MyDoublyLinkedListTester
LONG_LIST_LENGTH : int = 30
-shortList : MyDoublyLinkedList<Client>
-emptyList : MyDoublyLinkedList<Client>
-longerList : MyDoublyLinkedList<Client>
-list1 : MyDoublyLinkedList<Client>
+setUp() : void
+testGet() : void
+testRemove() : void
+testAddEnd() : void
+testSize() : void

longerList

shortList

list1

HashTableTest
-myHashTable : MyHashtable<Integer, Client>
+setUp1() : void
+setUp2() : void
+setUp3() : void
+isEmptyTest() : void
+isEmptyTest2() : void
+getTest() : void
+getTestInteresting Case() : void
+removeTestSpecialCase() : void
+removeTestBaseCase() : void
+addTest() : void
+addTestInteresting Case() : void

HashTableLibrary

myHashTable

-dataBasePartitionB

Element
K
V
<<Property>> -key : K
<<Property>> -value : V
<<Property>> -next : Element<K, V>
+Element(key : K, value : V)

-next

slots

MyHashtable
-maxs : int
<<Property>> -size : int
-slots : Element<K, V>[]
+MyHashtable()
+isEmpty() : boolean
+getSlotIndex(key : K) : int
+get(key : K) : V
+remove(key : K) : V
+add(key : K, value : V) : void
+generateArrayList() : ArrayList<Client>

<<Interface>>
IMyHashtable
<<Property>> +size : int
<<Property>> +empty : boolean
+getSlotIndex(key : K) : int
+get(key : K) : V
+remove(key : K) : V
+add(key : K, value : V) : void
+generateArrayList() : ArrayList<Client>

Lista de tareas a implementar

1. Una interfaz gráfica de usuario donde se pueda visualizar el estado actual de las 2 filas de espera.
2. Una interfaz gráfica de usuario que le permita visualizar y ordenar una tabla tipo hoja de cálculo por los siguientes criterios:
    a. Nombre del cliente
    b. Cédula del cliente
    c. Tiempo de vinculación del cliente
    d. Monto

    De los 4 algoritmos de ordenamiento solo uno puede ser de complejidad temporal promedio de $O(n^2)$, los 3 restantes deben ser de mejor complejidad.

3. Una interfaz gráfica de usuario que le permita buscar la información bancaria de un cliente a partir de su cédula y realizar las siguientes acciones:

a. Retiro: Aumentar el monto de la cuenta de ahorros
b. Consignación
c. Cancelación de cuenta
d. Pago de la tarjeta
e. *Undo* de una de las anteriores acciones

Functional requirements

Name:	R. #1. Assign turns at the time of admission
Summary:	Assign turns to users with name and ID data
Input:	String name, int id
Results:	Turn assigned to user
	Incomplete data, turn not assigned

Name:	R. #2. Place a user in the queue

Summary:	Place a user in the queue. There are two queue, the customer and the priority
Input:	
Results:	User located in queue

Name:	R. #3. Search a user in data base
Summary:	Search efficiently a user in data base to get information about user before it reaches the office where it will be attended
Input:	int id
Results:	User was found
	User was not found

Name:	R. #4. Change the amount of the customer's saving account
Summary:	Change the amount of the customer's saving account when requesting a withdrawal or consignment

Input:	int id
Results:	Changed the amount of the customer
	User does not have a saving account

Name:	R. #5. Cancellation of a customer's account
Summary:	Delate client data from database and add it to an exclusive datbase for those who desert
Input:	
Results:	Changed the amount of the customer
	User does not have a saving account

Name:	R. #6. Card payment
Summary:	User can pay amount used by credit card until now. Payment can be by cash or saving account

Input:	int id
Results:	Paid card
	User does not have debts

Name:	R. #7. Undo functionality
Summary:	Undo functionality works to fix mistakes, even after save those
Input:	int id
Results:	Paid card
	User does not have debts

Non-functional requirements

- Implement a ordering method for each of the chosen parameters, with the restriction that only one of them can have averge temporal complexity of n^2
- Users information must be displayed in a spreadsheet-type table
- Must have a graphical interface to display users information and make operations, spreadsheet-type table and status of queues