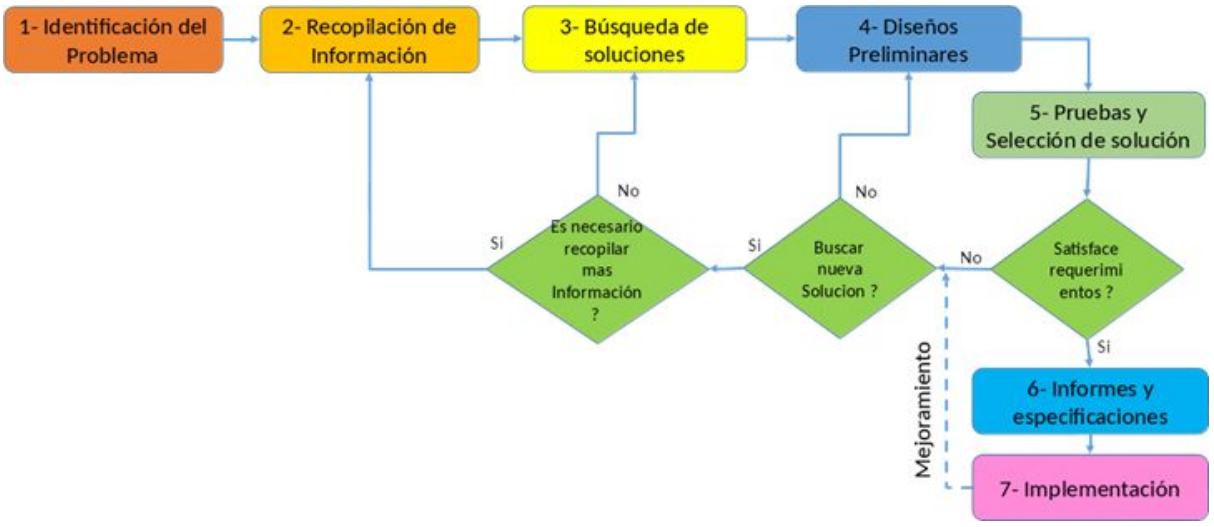
# MÉTODO DE LA INGENIERÍA

## Contexto problemático

Un banco necesita implementar un software para modelar el funcionamiento de una de sus sedes con mayor flujo de personas. El banco requiere un programa capaz de resolver todas la necesidades del cliente, entre las cuales se encuentran el proceso de turnos al momento del ingreso (fila para clientes y fila para personas con diferentes prioridades), manejo de tablas de datos, entre otros.

## Desarrollo de la Solución

Seguiremos estos pasos mostrados en el siguiente diagrama de flujo para llegar al desarrollo de la solución:




## Paso 1: Identificación del problema

*Identificación de necesidades y síntomas*

1. El banco necesita un software capaz de resolver las necesidades del cliente, además de llevar un buen manejo de la información de estos.
2. Se necesita un control en el sistema de turnos para poder clasificar a los clientes en dos filas.
3. Se busca que en lo posible, al ordenar la información de los clientes esta se haga de manera eficiente y consuma pocos recursos.
4. El banco desea poder revertir cambios hechos con la información del cliente, debido a que pueden haber errores al ingresar o cambiar estos datos.
5. El banco desea tener un buen manejo de su base de datos para poder buscar la información de manera eficiente.

*Definición del Problema*

El banco requiere un programa capaz de resolver todas la necesidades del cliente, entre las cuales se encuentran el proceso de turnos al momento del ingreso (fila para clientes y fila para personas con diferentes prioridades), manejo de tablas de datos, entre otros.

**Paso 2: Recopilación de la información**

Definiciones

**Retiro bancario:** El concepto de retiro bancario hace referencia a la acción de extraer dinero en efectivo de un banco. Para que este proceso sea posible, la persona debe contar con una cuenta en la entidad bancaria en cuestión y, a la vez, tener fondos disponibles en la misma.

**Consignación:** Consiste en la colocación de fondos en un banco u otras instituciones financieras para su custodia. Estas consignaciones se realizan al depositar en cuentas tales como cuentas de ahorro, cuentas corrientes y cuentas del mercado monetario.

**Cuenta de ahorros:** Una cuenta de ahorros es un producto financiero ofrecido por el Banco que permite ahorrar dinero de forma segura. El dinero depositado en una cuenta generará intereses de acuerdo a las políticas del banco y a las características de cada producto. Estas permiten disponer de tu dinero de forma rápida. Al abrir tu cuenta, el banco te entregará una tarjeta débito con la que podrás realizar retiros en los cajeros automáticos que dispone el banco.

**Complejidad de algoritmos recursivos:** Los algoritmos recursivos tienen asociada una ecuación de recurrencia conocida como el Teorema Maestro donde:

| Parámetro | Significado |
|---|---|
| a | Número de llamadas recursivas que se realizan en el caso recursivo |
| b | Factor por el cual se divide el tamaño del problema en cada llamada recursiva |
| k | Mayor exponente de las polinomios asociados a las complejidades del caso base y de la parte no recursiva de la rama recursiva del algoritmo, asumiendo que en ambos caso |

Fuentes:

https://definicion.de/

https://www.lifeder.com/

https://www.scotiabankcolpatria.com/personas/cuentas-e-inversion/mas-informacion/definicion-cuenta-ahorro

http://www.cime.cl/archivos/ILI134/9261_Complejidad_Algoritmos_Recursivos.pdf

https://ocw.unican.es/pluginfile.php/1246/course/section/1539/Recursivos.pdf

**Paso 3: Búsqueda de soluciones creativas**

**Opción 1:**

Determinar las acciones que se pueden realizar dentro de la cuenta como pilas de pares instanciando cada cambio que se realiza y añadiendolo dentro de las pilas, para a su vez, poder revertir las acciones que se realizan dentro del sistema.
El manejo de las filas de clientes y las filas de prioridad se realizarán con colas.
El cliente es un hashMap/hashTable donde tenga como key sus respectivas cédulas, y su valor sea el objeto Cliente.
La cola de clientes tendrá dos botones distintos para atender a los clientes de acuerdo a su prioridad o no.
Utilizar quicksort, mergesort, insertion, in order como métodos de ordenamiento para cada criterio diferente.

**Opción 2:**

- Utilizar Listview para mostrar el estado de las colas en interfaz gráfica.
- Utilizar Tableview para mostrar los datos de los clientes en las bases de datos.
- Implementar métodos de ordenamiento eficientes que permitan mostrar a los clientes de manera ascendente dado un criterio.
- Implementar una tablaHash basada en la técnica de encadenamiento para hacer búsquedas rápidas entre los clientes.
- Utilizar operaciones básicas para calcular cuando un cliente realiza un depósito o un retiro.
- Utilizar operaciones básicas para calcular cuando un cliente cancela el monto utilizado por su tarjeta de crédito.
- Utilizar un stack implementado como una lista doblemente enlazada para revertir la operación de cancelamiento de la cuenta de un cliente.

- Utilizar una cola implementada como una lista enlazada para almacenar los clientes corrientes y utilizar un heap de prioridad para almacenar los clientes con prioridades.
- Utilizar un escritor de archivos para guardar la información del banco.

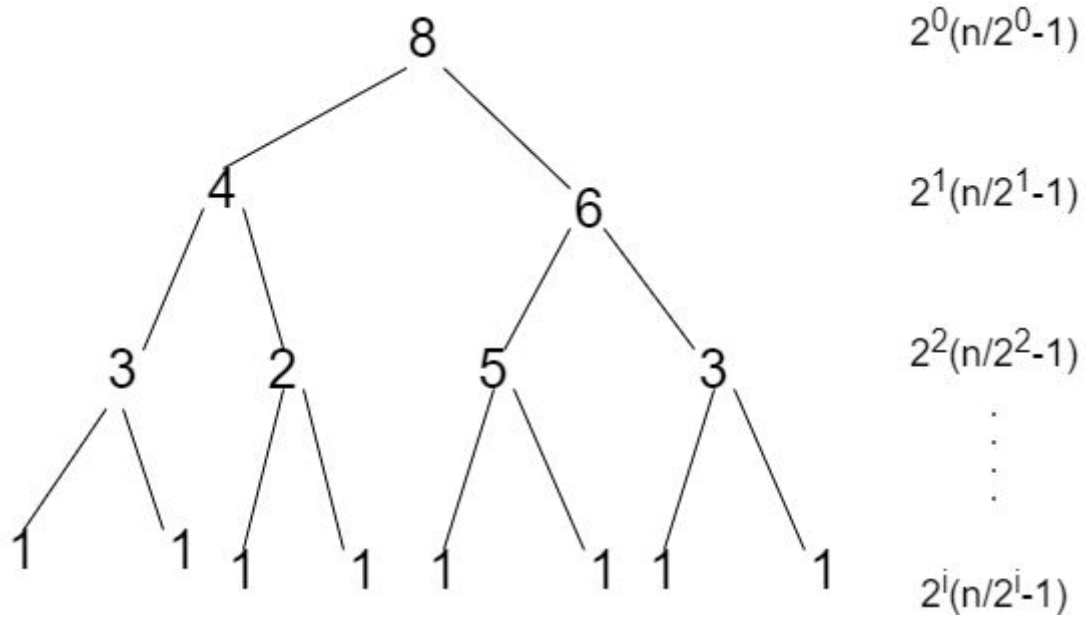
**Paso 4. Transición de las Ideas a los Diseños Preliminares**

**Opción 1:**

- Puede ser muy engorroso tener dos botones distintos para atender a las personas ya que implica tener dos acciones distintas y aumentar el espacio y quitarle practicalidad a la solución.
- Si se usa una HashTable la búsqueda será más eficiente.
- Insertion sort tiene una complejidad temporal de (n^2) en el peor de los casos, esta complejidad al no ser eficiente, haría que el problema no fuera lo más eficaz posible.
- Mergesort y quicksort tienen complejidad nlog(n) en el mejor de sus casos haciendo más eficiente los ordenamientos.

**COMPLEJIDAD TEMPORAL:**

Usar merge sort como método de ordenamiento resulta más eficiente debido a su complejidad temporal:




Lo puedo dividir 3 veces, y log(8)= 3
Recurrencia
n=8

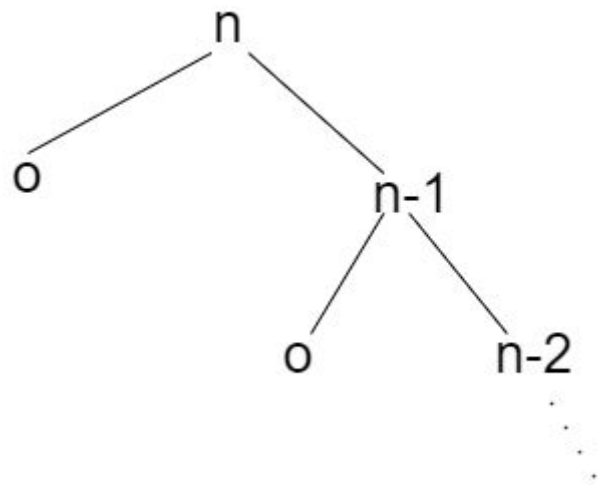
T(n)= T(n/2) + T(n/2) + (n-1)
En este caso n es igual a 8, por tanto
T(8)= T(8/2) + T(8/2) + (8-1)
T(8)= 2T(4) + 7
T(4)= T(4/2) + (4-1)
T(4)= 2T(2) + 3

T(2)= 2T(2/2) +(2-1)
T(2)= 2T(1) + 1 El caso base hace que T(1)=0

$2^i(n/2^i-1)$
$n- 2^i$
$\sum_{i=0}^{log(n)-1}$ = n( log(n)-1 -(0)) +1)

= n log(n)

Quicksort: en su mejor caso es:




T(n)= $\frac{n+(n-1)+(n-2)+1}{2}$ = $\frac{n*(n+1)}{2}$ = $n^2$

y el mejor caso

T(n)= T(n/2) + T(n/2) + (n-1)
$\sum_{i=0}^{log(n)-1}$ $2^i(n/2^i-1)$

$\sum_{i=0}^{log(n)-1}$ = n( log(n)-1 -(0)) +1)

= n log(n)

Insertion sort llega a ser ineficiente debido a la complejidad de su peor caso:

| public void sortClientsByTime() { | Best case | Worst case |
|---|---|---|
| for (int i = 1; i < clientList.size(); i++) { | n | |
| Client key = clientList.get(i); | n-1 | |
| int j = i-1; | n-1 | |
| while (j >= 0 && key.getMemberSinceDate().compareTo(clientList.get(j).getMemberSinceDate()) < 0) { | 1 | $\frac{n(n+1)}{2}$-1 |
| clientList.set(j+1, clientList.get(j)); | 0 | $\frac{n(n+1)}{2}$-n |
| j--; | 0 | $\frac{n(n+1)}{2}$-n |
| } | 0 | 0 |
| clientList.set(j+1, key); | n-1 | |
| } | 0 | |
| } | 0 | |
| Time complexity. Average case: $\theta(n^2)$ | $T(n) = 4n - 2 = \Omega(n)$ | $T(n) = \frac{3n^2}{2} + \frac{7x}{2} - 4 = O(n^2)$ |

inOrder sort:

| public void inOrder(Collection<V> collection) { | Best case | Worst case |
|---|---|---|
| if(left!=null) | 1 | |
| left.inOrder(collection); | $T\left(\frac{n}{2}\right) + 1$ | |
| collection.add(v); | 1 | |
| if(right!=null) | 1 | |
| right.inOrder(collection); | $T\left(\frac{n}{2}\right) + 1$ | |
| Time complexity. Average case: $\theta(n)$ | $T(n) = 2T\left(\frac{n}{2}\right) + 5 = \Omega(n)$ | $T(n) = 2T\left(\frac{n}{2}\right) + 5 = O(n)$ |
| | Apply Master theorem case | $c < \log_b a$ *where* $a = 2, b = 2, c = 0$ |

Teorema maestro: a*T(n/b)
f(n)=O(n^c) donde c < c_critico (c_critico es log_b(a))
T(n)=teta(n^(c_crit)) que es lo mismo que teta(n)

**COMPLEJIDAD ESPACIAL:**

| sortClientsByTime() | | |
|---|---|---|
| Input | clientList | n |
| Auxiliary | i | 1 |
| | key | 1 |
| | j | 1 |
| Output | N/A | |
| Space Complexity | | O(n) |

| InOrder | |
|---|---|
| If the bst happens to be balanced, the addresses are removed from the memory stack when returning. This space is re-used when making a new call from a level closer to the root. So the maximum numbers of memory addresses on the stack at the same time is the tree height: $\theta(logn)$=$\theta$(h) | |
| Space Complexity | |
| Avarage/Best Case | Worst Case |
| $\theta(logn)$ | O(n) |

Merge sort space Complexity:

O(n), debido a que cada partición que hace es n/2 creando sub arreglos, pero debido a que estos son arreglos temporales y se volverá al arreglo original, el espacio que ocupe el arreglo va a seguir siendo n.

Heapsort space complexity:
O(n log(n)) debido que al hacer las particiones, va guardandolas en la memoria (n) después de eso se llama a sí mismo recursivamente log(n)

**Opción 2:**

- Una hashtable basada en la técnica de encadenamiento requiere seguir punteros para buscar en cada lista enlazada.
- El método de ordenamiento insertion tiene complejidad cuadrática en el peor caso.
- Atender dos clientes a la vez no es muy práctico y puede generar confusión en la interfaz de usuario.

**Paso 5. Evaluación y Selección de la Mejor Solución**

**Criterios**

- **Criterio A:** Precisión de la solución. La alternativa entrega una solución:
  [2] Exacta
  [1] Aproximada

- **Criterio B:** Eficiencia. Se prefiere una solución con mejor eficiencia que las otras consideradas. La eficiencia puede ser:
  [4] Constante
  [3] Mayor a constante
  [2] Logarítmica
  [1] Lineal

- **Criterio C:** Completitud. El programa cumple con cuantos requerimientos:
  [3] Todas
  [2] Más de una si las hay, aunque no todas
  [1] Sólo una o ninguna

- **Criterio D:** Manejo de la tabla hash:
  [2] Se pueden colocar más de dos elementos en un mismo index.
  [1] No se pueden colocar elementos en un mismo index.

**Evaluación**

Opción 1:

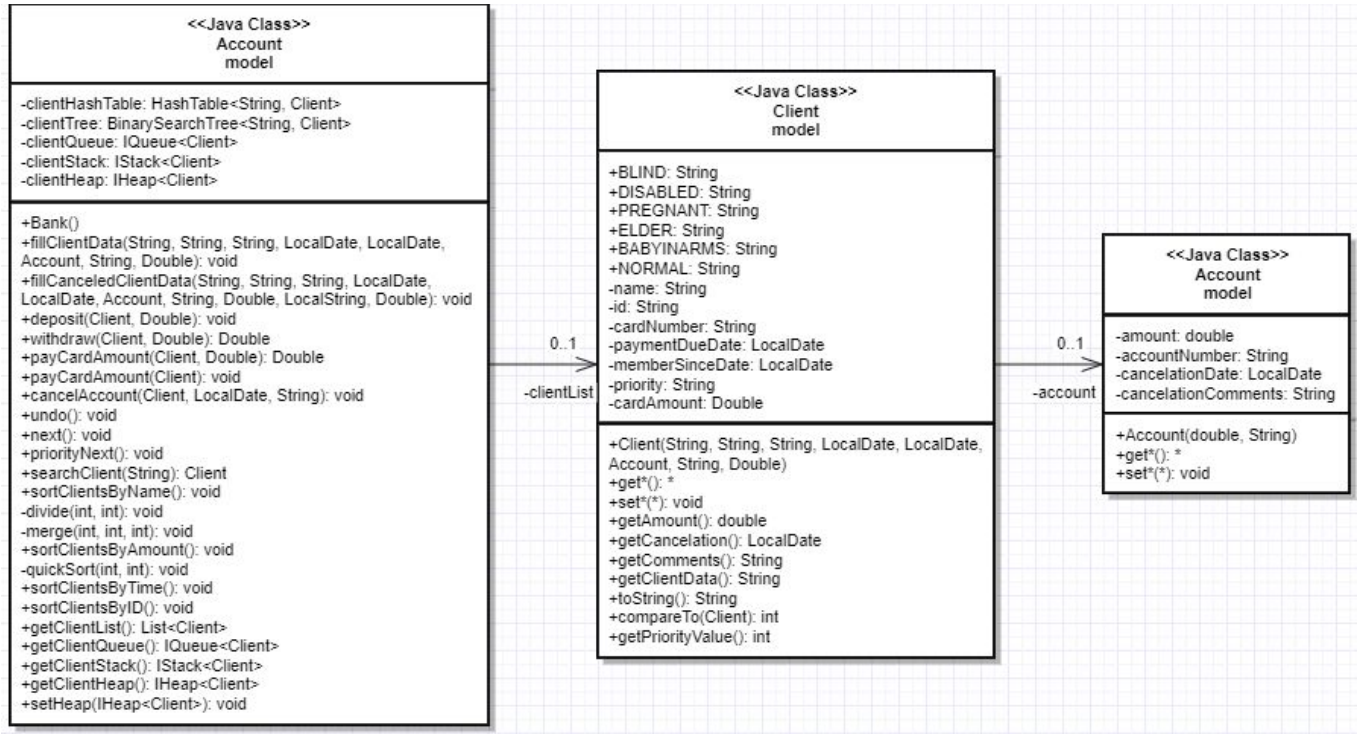
- ➔ A= 2
- ➔ B= 2

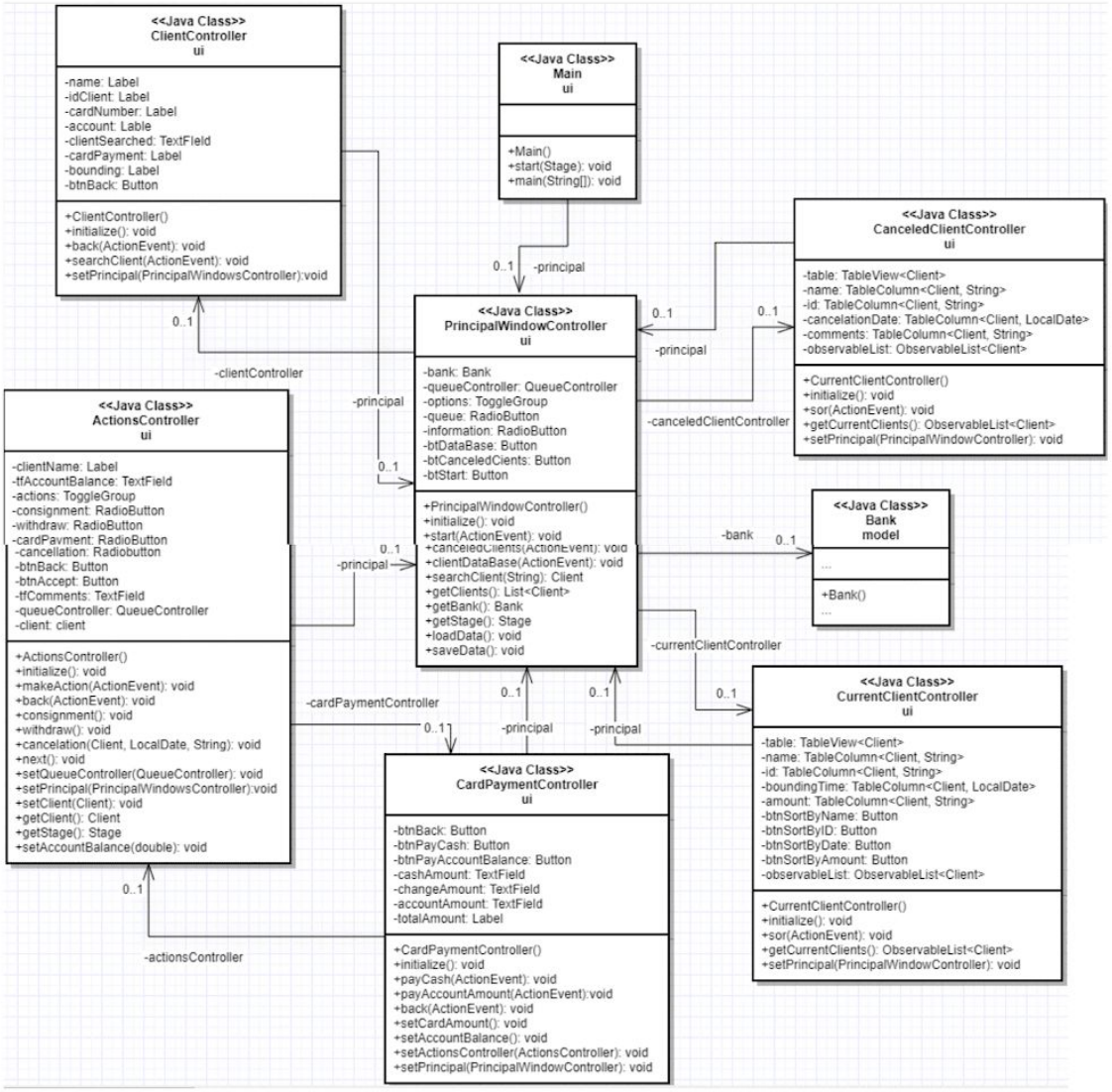
- ➔ C= 3
- ➔ D= 1

Opción 2:

- ➔ A= 2
- ➔ B= 2
- ➔ C= 3
- ➔ D= 2

**Paso 6: Preparación de informes y especificaciones**






**Las imágenes se pueden ver con mayor claridad en git/bank/docs.**

**Especificación del problema general:** Un banco requiere un sistema completo para poder atender a sus clientes de mejor manera.

**Consideraciones:**

1. Se debe de tener en cuenta que no va a ser posible atender a un cliente con prioridad y a un cliente normal a la misma vez, ya que esto produciría una colisión para ser atendidos.
2. El programa solamente deshace las acciones de cancelar cuenta, las demás acciones no pueden ser cambiadas.
3. Se debe tener muy en cuenta los montos y/o la cantidad de dinero disponible, ya que de no ser así, el sistema no le permitirá hacer parte de las acciones dentro del software.

**Paso 7: Implementación del diseño**

Implementando en el lenguaje de programación Java en el paradigma orientado a objetos.

Tareas a implementar:

a. Mostrar la interfaz de cola de usuarios.
b. Mostrar la interfaz de datos del usuario
c. Ordenar a los usuarios de acuerdo al criterio deseado.
d. Buscar información del usuario en el banco
e. Realizar consignaciones.
f. Realizar retiros.
g. Cancelar cuentas bancarias de clientes.
h. Pago de la tarjeta de crédito
i. Deshacer una acción.

**SUBRUTINAS:**

| Nombre | Mostrar cola de usuarios |
|---|---|
| Resumen | Muestra la interfaz en donde los clientes están esperando en 2 colas distintas, una es la de clientes y otra es la cola de clientes con prioridad |
| Entradas | Cliente |
| Salidas | Muestra en pantalla los clientes en sus respectivas filas y en su orden. |

```
public void getNormalQueue() {
        for (Client client : principal.getBank().getClientQueue()) {
                        normalQueue.getItems().add(client.getName());
                }
   }

public void getPriorityQueue() {
        IHeap<Client> h= new IHeap<Client>(100, true);
        h= principal.getBank().getClientHeap();
        while(!h.isEmpty()) {
                priorityQueue.getItems().add(h.extract().getName());
        }
   }
```

| Nombre | Mostrar datos de usuario |
|---|---|
| Resumen | Muestra la interfaz con el nombre, cédula, fecha en que se inscribió al banco y su monto actual. |
| Entradas | cedula |
| Salidas | Se muestra la información del cliente en su respectiva interfaz con sus datos. |

```
@FXML
    void searchClient(ActionEvent event) {
        try {
            Client client = principal.searchClient(clientSearched.getText());
            name.setText(client.getName());
            idClient.setText(client.getId());
            account.setText(client.getAccount().getAccountNumber());
            cardNumber.setText(client.getCardNumber());
            cardPayment.setText(client.getPaymentDueDate().toString());
            bounding.setText(client.getMemberSinceDate().toString());
        }catch(NullPointerException e) {
            JOptionPane.showMessageDialog(null, "Client not found.");
        }
    }
```

| Nombre | Ordenar lista de clientes |
|---|---|
| Resumen | Ordenar los usuarios de acuerdo a algún criterio específico, hay 4 criterios disponibles: nombre, cedula, monto, tiempo de vinculación del cliente |
| Entradas | criterio de ordenamiento |
| Salidas | Lista de clientes ordenados por el criterio deseado. |

- Ordenar por el nombre:

```
public void sortClientsByName() {
        divide(0, clientList.size()-1);
}

private void divide(int start,int end){
```

```
        if(start<end && (end-start)>=1){
                int mid = (end + start)/2;
                divide(start, mid);
                divide(mid+1, end);
                merge(start,mid,end);
         }
    }

    private void merge(int start,int mid,int end){
        List<Client> mergedSortedArray = new ArrayList<Client>();
        int left = start;
        int right = mid+1;
        while(left<=mid && right<=end){
        if(clientList.get(left).getName().compareTo(clientList.get(right).getName())<0){
                mergedSortedArray.add(clientList.get(left));
                left++;
        }else{
                mergedSortedArray.add(clientList.get(right));
                right++;
        }
        }
        while(left<=mid){
                mergedSortedArray.add(clientList.get(left));
                left++;
        }

        while(right<=end){
        mergedSortedArray.add(clientList.get(right));
        right++;
        }

        int i = 0;
        int j = start;

        while(i<mergedSortedArray.size()){
        clientList.set(j, mergedSortedArray.get(i++));
        j++;
        }
    }
```

- Ordenar por la cédula

```
    public void sortClientsByID() {
        clientList.clear();
        clientTree.inOrder(clientList);
```

```
}

public void inOrder(Collection<V> collection) {
        if(left!=null)
                left.inOrder(collection);
        collection.add(v);
        if(right!=null)
                right.inOrder(collection);
}
```

- Ordenar por el tiempo de vinculación

```
public void sortClientsByTime() {
        for (int i = 1; i < clientList.size(); i++){
                Client key = clientList.get(i);
                 int j = i-1;
                while (j >= 0 &&
        key.getMemberSinceDate().compareTo(clientList.get(j).getMemberSinceDate
        ()) < 0) {
                clientList.set(j+1, clientList.get(j));
                j--;
        }
                clientList.set(j+1, key);
        }
}
```

- Ordenar por el monto

```
public void sortClientsByAmount() {
        quickSort(0, clientList.size()-1);
}

protected void quickSort(int a, int b) {
        if(a<b) {
                Client pivot = clientList.get(b);
                int left = a;
                int right = b;

                while (left < right) {
                        while(clientList.get(left).getAccount().getAmount() <
                pivot.getAccount().getAmount())
                        left++;
                        while(clientList.get(right).getAccount().getAmount() >
                pivot.getAccount().getAmount())
                        right--;
                      if (right > left) {
                        Collections.swap(clientList, left, right);
```

```
                    }
                }
                quickSort(a, right-1);
                quickSort(right+1, b);
            }
        }
```

| Nombre | Buscar cliente |
|---|---|
| Resumen | Buscar la información del cliente en el banco con base en su ID |
| Entradas | cedula |
| Salidas | Muestra la información del usuario específico o null si no hay ningún cliente |

```
public Client searchClient(String iD) {
        return clientHashTable.get(iD);
}
```

| Nombre | Realizar consignación |
|---|---|
| Resumen | Guarda una consignación de un monto de dinero que haga un cliente al banco. |
| Entradas | cantidad a consignar |
| Salidas | La consignación se ha realizado y el dinero del cliente se ha guardado en el banco. |

```
public void deposit(Client client, Double deposit) {
        Double amount = client.getAccount().getAmount()+deposit;
        client.getAccount().setAmount(amount);
}
```

| Nombre | Realizar retiro |
|---|---|
| Resumen | Saca una cantidad de dinero específica del banco con base en el monto actual del cliente, y lo devuelve en efectivo al cliente. |
| Entradas | Cliente que va a realizar el retiro, y el monto a retirar |
| Salidas | El monto actual después del retiro. |

```
public void withdraw(Client client, Double withdrawal) throws RuntimeException {
	Double amount = client.getAccount().getAmount()-withdrawal;
	if(amount<0)
		throw new RuntimeException("Invalid operation: Cannot withdraw an amount
		greater than the account balance.");
	else
		client.getAccount().setAmount(amount);
}
```

| Nombre | Cancelar cuenta bancaria |
|---|---|
| Resumen | Cancela la cuenta creada para la persona, y la transfiere a la base de datos de cuentas canceladas. |
| Entradas | Cuenta del cliente a cancelar |
| Salidas | Los datos del cliente y su cuenta han sido retirados del sistema bancario y quedan en una base de datos reservada. |

```
public void cancelAccount(Client client, LocalDate cancelationDate, String
cancelationComments) throws IOException{
	client.getAccount().setCancelationDate(cancelationDate);
	client.getAccount().setCancelationComments(cancelationComments);
	clientStack.push(client);
	File tempFile = new File("resources\\tempFile.txt");
	File file1 = new File("resources\\canceledAccounts.txt");
	File file2 = new File("resources\\database.txt");
	BufferedWriter tempBw = new BufferedWriter(new FileWriter(tempFile));
	BufferedWriter bw = new BufferedWriter(new FileWriter(file1, true));
	BufferedReader br = new BufferedReader(new FileReader(file2));
	updateDataBase(client, tempBw, bw, br);
	Files.move(tempFile.toPath(),
	file2.toPath(),StandardCopyOption.REPLACE_EXISTING);
	clientList.remove(client);
	clientTree.deleteNode(client.getId());
	clientHashTable.delete(client.getId());
}
```

| Nombre | Pagar tarjetas de crédito |
|---|---|
| Resumen | Paga las cuentas de las tarjeta de crédito que la persona tenga, se puede pagar de dos formas: con dinero o con el monto |

| | |
|---|---|
| | actual disponible en el banco. |
| Entradas | Cliente, y la cantidad solamente cuando se va a pagar con efectivo. |
| Salidas | Las cuentas quedan pagas. |

- Pagar con  dinero

```
public double payCardAmount(Client client, Double amount) {
        Double cardAmount = client.getCardAmount()-amount;
        if(cardAmount>0)
                throw new RuntimeException("Insufficient amount: Please insert the
total amount spent using the credit card.");
        else
                client.setCardAmount(0.0);
        return Math.abs(cardAmount);
}
```

- Pagar con el monto de la cuenta

```
public void payCardAmount(Client client) {
        Double cardAmount =
client.getCardAmount()-client.getAccount().getAmount();

        if(cardAmount>0)
                throw new RuntimeException("Insufficient amount: Your account's
balance is not enough.");
        else {
                client.setCardAmount(0.0);
                client.getAccount().setAmount(Math.abs(cardAmount));
        }
}
```

| Nombre | Deshacer una acción |
|---|---|
| Resumen | Deshace la acción de cancelar la cuenta de un cliente para volver a ingresar sus datos a la base de datos de clientes del banco. |
| Entradas | |
| Salidas | Retorna el cliente de la base de datos de clientes retirados a la de activos. |

```
public void undo() throws Exception {
        Client client = clientStack.pop();
```

```
                client.getAccount().setCancelationDate(null);
                client.getAccount().setCancelationComments(null);
                clientList.add(client);
                clientTree.addNode(client.getId(), client);
                clientHashTable.delete(client.getId());
}
```

# PRUEBAS UNITARIAS

Configuración de los Escenarios de la clase Bank:

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage1 | GameTest | User name= “Fernanda”, nickname= “fernandarojas152”, gender=”femenine”, password= “fernanda” |

Diseño de Casos de Prueba:

| Objetivo de la Prueba:. El programa es capaz de verificar que existen usuarios registrados previamente dentro del usuario y no permite ingresar uno con el mismo apodo. | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| Game | addUser() | setUpStage1 | User2 name= “Fernanda”, nickname= “fernandarojas152”, gender=”femenine”, password= “fernanda” | UserAlreadyExistsException |

Configuración de los Escenarios de Stack:

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage1 | StackTest | Se instancia únicamente el stack. |

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage2 | StackTest | stack= new IStack<Integer>()<br>stack.push(12)<br>stack.push(432)<br>stack.push(14)<br>stack.push(65)<br>stack.push(76)<br>stack.push(67)<br>stack.push(19)<br>stack.push(1)<br>stack.push(23)<br>stack.push(34)<br>stack.push(45) |

Diseño de Casos de Prueba:

| Objetivo de la Prueba:. El programa es capaz de atrapar la excepción cuando se intenta eliminar un elemento en una pila ya vacía. | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IStack | stack.pop() | setUpStage1 | | EmptyStackException |

| Objetivo de la Prueba:. El programa es capaz de atrapar la excepción cuando se intenta mirar el top una pila ya vacía. | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| IStack | stack.peek() | setUpStage1 | | EmptyStackException |

| Objetivo de la Prueba:. El programa es capaz de mostrar el top de la pila. | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IStack | stack.peek() | setUpStage2 | | 45 |

| Objetivo de la Prueba:. El programa es capaz de retornar el tamaño actual de la pila | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IStack | stack.getSize() | setUpStage2 | | 11 |

| Objetivo de la Prueba:. El programa es capaz de identificar si la pila está vacía o no | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| IStack | stack.isEmpty() | setUpStage1 | | true |

| Objetivo de la Prueba:. El programa es capaz de agregar un elemento cuando la pila esta vacia | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IStack | stack.push() | setUpStage1 | stack.push(19) | 19 |

| Objetivo de la Prueba:. El programa es capaz de añadir un nuevo elemento a una pila con elementos previos | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IStack | stack.push() | setUpStage2 | stack.push(14) | 14 |

Configuración de los Escenarios de Heap:

| Nombre | Clase | Escenario |
|---|---|---|

| | | |
|---|---|---|
| setUpStage1 | HeapTest | Se instancia únicamente el heap con tamaño 100 y en maxHeap. |

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage2 | HeapTest | heap= new IHeap<Integer>(100, true);<br>heap.insert(7);<br>heap.insert(8);<br>heap.insert(6);<br>heap.insert(50);<br>heap.insert(86); |

Diseño de Casos de Prueba:

| Objetivo de la Prueba:. El programa es capaz de verificar si el heap está vacío o no | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IHeap | heap.isEmpty() | setUpStage1 | | true |

| Objetivo de la Prueba:. El programa es capaz de eliminar el maximo elemento en el heap | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| IHeap | stack.extract() | setUpStage2 | | 86 |

| Objetivo de la Prueba:. El programa es capaz de eliminar todos los elementos en el heap | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IHeap | stack.extract() | setUpStage2 | heap.extract();<br>heap.extract();<br>heap.extract();<br>heap.extract(); | 6 |

| Objetivo de la Prueba:. El programa es capaz de mostrar el valor máximo dentro del heap. | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IHeap | stack.max() | setUpStage2 | heap.extract(); | 50 |

| Objetivo de la Prueba:. El programa agrega un elemento al heap y mantiene la propiedad de max-heap | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| IHeap | stack.insert() | setUpStage1 | heap.insert(4);<br>heap.insert(14);<br>heap.insert(12); | 14 |

| Objetivo de la Prueba:. El programa puede verificar cual es el tamaño del heap. | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IHeap | stack.getHeap_Size() | setUpStage2 | | 5 |

Configuración de los Escenarios de Queue:

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage1 | QueueTest | Se instancia únicamente la cola |

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage2 | QueueTest | q= new IQueue<Integer>();<br>q.enqueue(76);<br>q.enqueue(54);<br>q.enqueue(22);<br>q.enqueue(32); |

| | | |
|---|---|---|
| | | heap.insert(86); |

Diseño de Casos de Prueba:

| Objetivo de la Prueba:. El programa es capaz de verificar si la cola está vacía o no | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IQueue | q.isEmpty() | setUpStage1 | | true |

| Objetivo de la Prueba:. El programa agrega un nuevo elemento a la cola. | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IQueue | stack.enqueue() | setUpStage1 | q.enqueue(14);<br>q.enqueue(33); | 14 |

| Objetivo de la Prueba:. El programa elimina un elemento de la cola | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| IQueue | stack.dequeue() | setUpStage2 | | 76 |

| Objetivo de la Prueba:. El programa es capaz de mostrar el tamaño actual de la cola. | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IQueue | stack.size() | setUpStage2 | | 4 |

| Objetivo de la Prueba:. El programa muestra la cola | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| IQueue | stack.getRear() | setUpStage2 | | 32 |

| Objetivo de la Prueba:. El programa muestra la cabeza | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| IQueue | stack.getFront() | setUpStage2 | | 76 |

Configuración de los Escenarios de Binary Search Tree:

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage1 | BSTTest | Se instancia únicamente el arbol binario |

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage2 | BSTTest | b= new BinarySearchTree<Integer, String>();<br>b.addNode(3, "Poe");<br>b.addNode(6, "Akutagawa");<br>b.addNode(2, "Conan Doyle");<br>b.addNode(1, "Woolf"); |

Diseño de Casos de Prueba:

| Objetivo de la Prueba:. El programa es capaz de agregar nuevos elementos al arbol binario | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| BinarySearchTree | b.addNode() | setUpStage1 | b.addNode(2, "Rimbaud"); | Rimbaud |

| Objetivo de la Prueba:. El programa no permite agregar un elemento a la misma llave cuando esta ya esta ocupada. | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| BinarySearchTree | b.addNode() | setUpStage1 | b.addNode(2, "Pizarnik");<br><br>b.addNode(1, "Verlaine");<br><br>b.addNode(6, "Edogawa");<br><br>b.addNode(1, "Baudelaire") | Exception |

| Objetivo de la Prueba:. El programa permite actualizar el valor de una llave dentro del arbol binario | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| BinarySearchTree | b.updateNode() | setUpStage2 | b.updateNode(1, "Osamu"); | “Osamu” |

| Objetivo de la Prueba:. El programa permite buscar una llave, y mostrar el valor de la llave o del valor establecido para esta. |
|---|

| Clase | Método | Escenario | Valores de Entrada | Resultado |
|---|---|---|---|---|
| BinarySearchTree | b.searchNode() | setUpStage2 | | “Poe” |

| Objetivo de la Prueba:. El programa permite eliminar una llave | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| BinarySearchTree | b.Delete() | setUpStage2 | | True |

| Objetivo de la Prueba:. El programa permite calcular el peso de un arbol binario | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| BinarySearchTree | b.Weight() | setUpStage2 | | 4 |

| Objetivo de la Prueba:. El programa permite calcular la altura de un arbol binario | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| BinarySearchTree | b.Height() | setUpStage2 | | 3 |

Configuración de los Escenarios de HashTable:

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage1 | HashTest | Se instancia únicamente el hashTable |

| Nombre | Clase | Escenario |
|---|---|---|
| setUpStage2 | HashTest | hash= new HashTable<Integer, String>();<br>hash.insert(2, "Pedro");<br>hash.insert(2, "Alejandra");<br>hash.insert(5, "Maria");<br>hash.insert(1, "Susana"); |

Diseño de Casos de Prueba:

| Objetivo de la Prueba:. El programa es capaz de buscar un elemento aunque tengan la misma llave | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| HashTable | hash.get) | setUpStage 2 | | “Alejandra” |

| Objetivo de la Prueba:. El programa es capaz de buscar un elemento | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| HashTable | hash.get) | setUpStage 2 | | “Susana” |

| Objetivo de la Prueba:. El programa es capaz de verificar si la tabla hash esta vacia o no | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| HashTable | hash.get Size() | setUpStage 1 | | true |

| Objetivo de la Prueba:. El programa es capaz de dar el tamaño de la hash | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| HashTable | hash.getSize() | setUpStage2 | | true |

| Objetivo de la Prueba:. El programa es capaz de verificar si la tabla hash añade un elemento | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| HashTable | hash.insert() | setUpStage1 | hash.insert(6, "Amanda"); | ‘‘Amanda’’ |

| Objetivo de la Prueba:. El programa es capaz de verificar si la tabla hash añade un elemento con la misma llave | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |
| HashTable | hash.insert() | setUpStage2 | hash.insert(1, "Amanda"); | ‘‘Amanda’’ |

| Objetivo de la Prueba:. El programa es capaz de eliminar un elemento | | | | |
|---|---|---|---|---|
| Clase | Método | Escenario | Valores de Entrada | Resultado |

| | | | | |
|---|---|---|---|---|
| HashTable | hash.delete() | setUpStage2 | hash.delete(1); | true |