1. **Identificación del problema**

**Contexto Problemático**

Un gran banco quiere implementar un programa que permita la capacidad de administrar la fila de clientes de una de sus sedes. Se busca que el software pueda procesar los turnos de los clientes, ubicándolos en una fila prioritaria si es el caso o en una fila de espera normal. Esto con el fin de realizar procesos estadísticos y mejorar el servicio de atención. Por lo tanto, el banco necesita solucionar este problema, por medio de un lenguaje de programación que tenga la habilidad de cumplir cada uno de los requerimientos solicitados.

**Identificación del problema**

De acuerdo con el enunciado, cuando un cliente ingresa a la sede bancaria a solicitar un turno, el banco registra su nombre y número de cedula. Inmediatamente busca que el cliente se encuentre en la base de datos del banco, si es así, procede a ubicar dicho cliente a una de las dos filas (prioritaria y normal). Una vez asignado el turno, el programa provee la función de atender al cliente permitiendo realizar operaciones a su cuenta bancaria como retiro y consignación de un monto a su cuenta de ahorros, entre muchas otras funciones. Además, la información de todos los usuarios presentes en la sede bancaria podrá ser visualizada en una tabla de datos y permitirá organizar a los clientes respecto a los cuatro parámetros más relevantes como su nombre o el monto que posee.

Entonces, de acuerdo con lo anterior, las especificaciones que debe cumplir el programa a desarrollar son:

1. Se le debe asignar al cliente un turno en la sede bancaria.
2. Cliente debe esperar dicho turno y ser atendido por el banco, permitiéndole realizar ciertas operaciones a sus cuentas bancarias.
3. Se debe mostrar la información de todos los clientes organizados respecto a cuatro parámetros de escogencia

Además, en caso de errores humanos el software debe proporcionar la función undo para poder deshacer las equivocaciones.

**2. Recopilación de la información necesaria**

A continuación, se presentará las estructuras y los algoritmos que nos pueden ayudar a resolver el problema.

**Queue**: Una cola (queue en ingles) es una estructura lineal que sigue un orden particular en el que se realizan las operaciones. El orden es Primero en entrar, primero en salir (FIFO). Esta estructura es perfecta para el problema presentado, debido a que ejemplifica a como es una fila de clientes, en donde se atiende al primero que llego.



Las principales operaciones que nos proporcionan esta estructura de datos lineal son las siguientes:

* Enqueue: agrega un elemento a la cola. Si la cola está llena, se dice que es una condición de desbordamiento.
* Dequeue: elimina un elemento de la cola. Los elementos se abren en el mismo orden en que se empujan. Si la cola está vacía, se dice que es una condición de desbordamiento.
* Front: Saca el elemento del frente de la cola.
* Rear: Obtiene el último elemento de la cola.

**Stack:** Consisten en una lista de elementos en la cual solo se tiene acceso al último

elemento insertado. La posición en donde se encuentra el último elemento se

reconoce como “top”. El tipo de orden en las entradas y salidas se conocen como

LIFO (Last in - First out).

* Push: agrega un elemento a la pila. Si la pila está llena, se dice que es una condición de desbordamiento.
* Pop: elimina un elemento de la pila. Los elementos aparecen en el orden inverso en el que se empujan. Si la pila está vacía, se dice que es una condición de desbordamiento.
* Peek or Top: Devuelve el elemento superior de la pila.
* isEmpty: Devuelve verdadero si la pila está vacía, de lo contrario es falso.

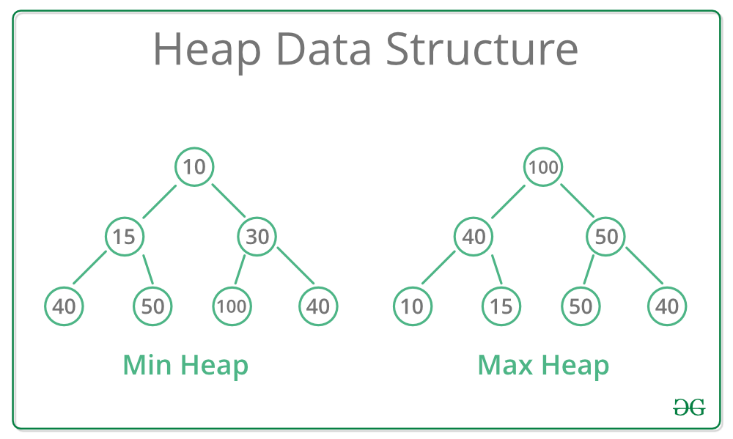
**ArrayList**: Una ArrayList, o una matriz de redimensionamiento dinámico, nos permite tener los beneficios de un arreglo a la vez que ofrece flexibilidad de tamaño. Nunca se quedará sin espacio en un ArrayList, ya que su capacidad aumentará a medida que inserte elementos.



ArrayList es parte de la librería de Java y pertenece al paquete java.util. Implementa a su vez todos los métodos de la estructura de datos lista. Utilizar ArrayList sera de gran ayuda al momento de ordenar los datos.

**Heap**: Un montículo es una estructura de datos especial basada en la estructura de datos árbol en la que el árbol es un árbol binario completo. Generalmente, los montículos pueden ser de dos tipos:

* Max-Heap: En un Max-Heap, la clave presente en el nodo raíz debe ser mayor entre las claves presentes en todos sus elementos secundarios. La misma propiedad debe ser recursivamente verdadera para todos los subárboles de ese árbol binario.
* Min-Heap: En un Min-Heap, la clave presente en el nodo raíz debe ser mínima entre las claves presentes en todos sus hijos. La misma propiedad debe ser recursivamente verdadera para todos los subárboles de ese árbol binario.



**Hash table:** El hash es una estructura de datos importante que está diseñada para usar una función especial llamada función Hash que se usa para mapear un valor dado con una clave particular para un acceso más rápido a los elementos. La eficiencia del mapeo depende de la eficiencia de la función hash utilizada.

Deje que una función hash H (x) mapea el valor x en el índice x% 10 en una matriz. Por ejemplo, si la lista de valores es [11,12,13,14,15], se almacenará en las posiciones {1,2,3,4,5} en la matriz o la tabla Hash, respectivamente. Esta estructura nos permitirá encontrar el cliente que deseemos buscar en la base de datos.

Imagen que contiene verde, firmar, grande, sostener

Descripción generada automáticamente

**Algoritmos de ordenamiento:** Los algoritmos de ordenamiento nos permite, como su nombre lo dice, ordenar información de una manera especial basándonos en un criterio de ordenamiento. Hay métodos muy simples de implementar que son útiles en los casos en dónde el número de elementos a ordenar no es muy grande (ej, menos de 500 elementos). Por otro lado, hay métodos sofisticados, más difíciles de implementar pero que son más eficientes en cuestión de tiempo de ejecución.

Los métodos sencillos por lo general requieren de aproximadamente n x n pasos para ordenar n elementos.

Los métodos simples son: insertion sort (o por inserción directa) selection sort, bubble sort, y shellsort, en dónde el último es una extensón al insertion sort, siendo más rápido. Los métodos más complejos son el quick-sort, el heap sort, radix y address-calculation sort. El ordenar un grupo de datos significa mover los datos o sus referencias para que queden en una secuencia tal que represente un orden, el cual puede ser numérico, alfabético o incluso alfanumérico, ascendente o descendente.

**3. Búsqueda de soluciones creativas**

Para este paso, aunque podemos pensar en soluciones propias, y como resulta que el problema de ordenamiento es uno de los clásicos en la ciencia de la computación, buscamos en textos especializados diversas estrategias a través de los cuales puedan ser ordenados los datos. Los métodos encontrados son los siguientes:

1. Primer requerimiento, asignar turno al cliente:

* Alternativa 1. Utilizar la estructura de datos stack para representar las dos filas de los clientes en la sede bancaria.
* Alternativa 2. Utilizar la estructura de datos queue para ejemplificar las dos filas de los clientes de la sede bancaria.
* Alternativa 3. Representar una fila por ejemplo la prioritaria con un stack y la otra fila de clientes normales implementarla con u queue.

2. Segundo requerimiento, atender al cliente:

* Alternativa 1: Diseñar una clase por cada producto financiero del cliente, es decir, una clase para la cuenta de ahorros y una clase para la tarjeta de crédito.
* Alternativa 2: Implementar las operaciones de los productos financieros en una sola clase llamada bank management.

3. Tercer requerimiento, organizar la información de los cientes.

* Alternativa 1. Burbuja (Bubble Sort): Complejidad
* Alternativa 2. Conteo (Counting Sort): Complejidad
* Alternativa 3. Montones (Heapsort): Complejidad
* Alternativa 4. Inserción (Insertion Sort): Complejidad
* Alternativa 5. Mezclas (Merge Sort): Complejidad
* Alternativa 6. Rápido (Quicksort): Complejidad
* Alternativa 7. Selección (Selection Sort): Complejidad

1. **Diseños preliminares**

**IDEAS NO VIABLES**

1. En el primer requerimiento se decidió descartar la alternativa 1. Debido a que la forma más cercana de modelar las filas de clientes es con una cola (queue), en donde el primer cliente en llegar es el primero al que se le atiende y el ultimo cliente en llegar es enviado al fondo de la fila. No tendría entonces sentido implementar una pila(stack) debido a que en esta el primer elemento eliminado es el último.

2. Para el requerimiento dos se decidió descartar la alternativa 2, debido a que cada producto financiero tiene sus propios atributos o propiedades y no tendría sentido mezclarlo todo en una clase.

3. Para el requerimiento tres, a los algoritmos se les realiza los respectivos análisis de complejidad temporal y espacial para decidir cual escoger.

**DISEÑO PRELIMINAR DE IDEAS VIABLES POR IMPLEMENTAR**

Después de realizar la respectiva evaluación para todas las ideas, se decidieron

descartar las ideas que se mencionaron anteriormente, dejando dos alternativas para el requerimiento uno y una alternativa para el requerimiento dos y a algunos algoritmos de ordenamiento se les realizo el análisis de temporalidad para comprobar que efectivamente demoran ese tiempo estimado.

Complejidad temporal del Selection sort

|  |  |
| --- | --- |
| Instrucción | Costo |
| Void selectionSort(int arr[]) |  |
| Int n = arr.length; | 1 |
| For(int i = 0; i < n-1; i++) | n-1 |
| Int min\_idx = i; | n-2 |
| For(int j = i+1; j<n; j++) |  |
| If(arr[j] < arr[min\_idx]) |  |
| Min\_idx = j; |  |
| Int temp = arr[min\_idx]; | n-1 |
| Arr[min\_idx] = arr[i]; | n-1 |
| Arr[i] = temo; | n-1 |

Complejidad del heapsort

|  |  |
| --- | --- |
| Instrucción | Costo |
| Void heapsort(int arr[], int n) |  |
| For(int i = n/2 – 1; i >= 0; i--) |  |
| Heapify(arr, n, i) | O(log n) – 1 |
| For(int i=n-1; i>0; i--) | n |
| Swap(arr[0], arr[i]) | n-1 |
| Heapify(arr, i , 0) | n-1 |

Complejidad temporal de los algoritmos

|  |  |
| --- | --- |
| Bubble Sort | O(n^2) |
| Counting Sort | O(n+k) |
| Heapsort | O(n log n ) |
| Insertion Sort | O(n^2) |
| Merge Sort | O(n log n). |
| Quicksort | O(n log n). |
| Selection Sort): | O(n^2). |

**5. Evaluación y selección de la mejor solución.**

A partir de las ideas que se realizaron anteriormente, se deben generar unos

criterios de evaluación que permita clasificar las ideas de acuerdo con su calificación.

Con base en el resultado obtenido después de la evaluación, se tomará la decisión

de cuál alternativa implementar. A continuación, se encuentran los criterios que se

escogieron. Cada uno tiene asociado varios valores numéricos para determinar el

puntaje de cada alternativa.

-Criterio A: Precisión de la solución. Es solucionado el requerimiento de manera:

● [2]Completa

● [1]Incompleta

-Criterio B: Eficiencia. El requerimiento es resuelto de un modo:

● [4] Muy eficiente

● [3] Eficiente

● [2] Ineficiente

● [1] Muy deficiente

-Criterio C: Facilidad de codificar. ¿A partir de lo planeado, la solución permite

desarrollar un algoritmo que cumpla con el requerimiento y que sea manejable a la

hora de llevarlo al código?

● [3] Manejable de llevar a la implementación en java

● [2] No es manejable, pero permite que sea llevado a la implementación en

java

● [1] No es manejable y tampoco se puede llevar a la implementación en java

Requerimiento 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *Criterio A* | *Criterio B* | *Criterio C* | *Total* |
| *Alternativa 2* | Completa [2] | *Muy eficiente [4]* | *Manejable [3]* | *9* |
| *Alternativa 3* | *Completa [2]* | Eficiente [3] | *Manejable [2]* | *7* |

Requerimiento 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Alternativas* | *Criterio A* | *Criterio B* | *Criterio C* | *Total* |
| *Alternativa 1* | 2 | 2 | 3 | 7 |
| *Alternativa 2* | 2 | 3 | 3 | 8 |
| *Alternativa 3* | 2 | 4 | 3 | 9 |
| *Alternativa 4* | 2 | 2 | 3 | 7 |
| *Alternativa 5* | 2 | 4 | 3 | 9 |
| *Alternativa 6* | 2 | 4 | 3 | 9 |
| *Alternativa 7* | 2 | 2 | 3 | 7 |

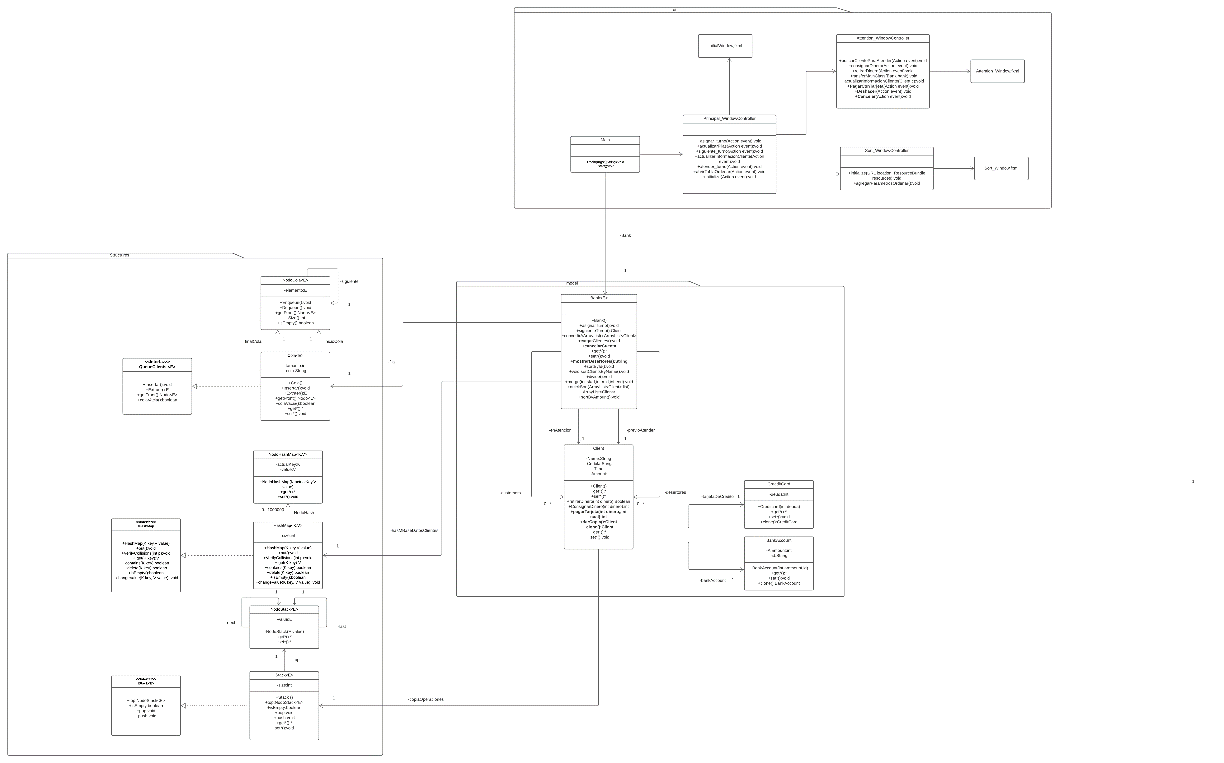
Para el primer requerimiento se decide escoger la alternativa 2 que utiliza dos colas o queue como las estructuras de datos que nos permitirán resolver el problema de las filas de clientes, cumple con los criterios de evaluación y además es manejable y se puede implementar en un lenguaje de programación como java. Además todos los datos deben estar a su ven en una estructura hash, para que puedan ser encontrados en la base de datos.

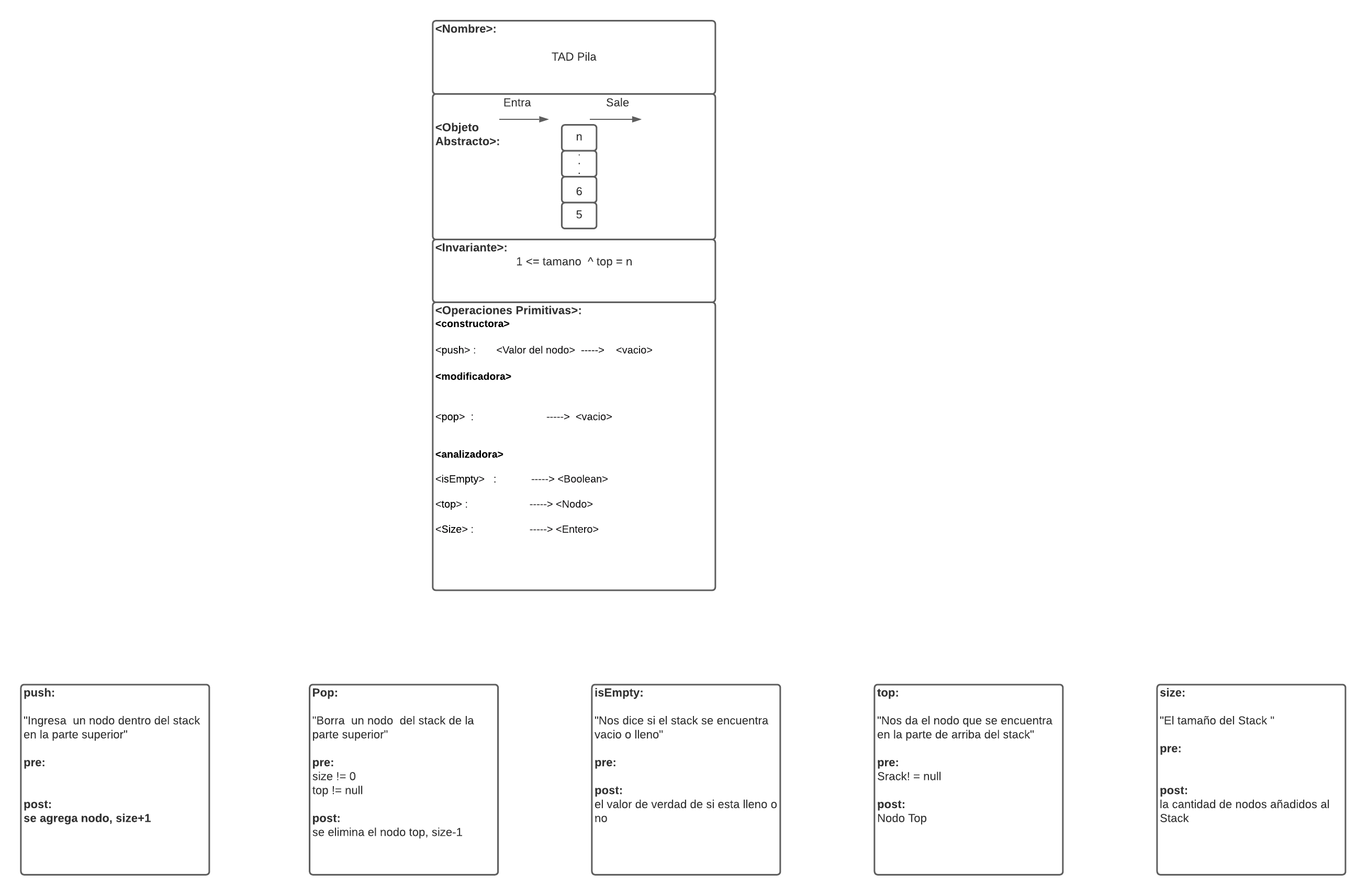
Para el segundo requerimiento se decide crear una clase por cada producto financiero, debido a que estos tienen sus propios atributos e independencia, no es lo mismo una tarjeta de credito a una cuenta de ahorros. Además, manejan diferentes funcionalidades.

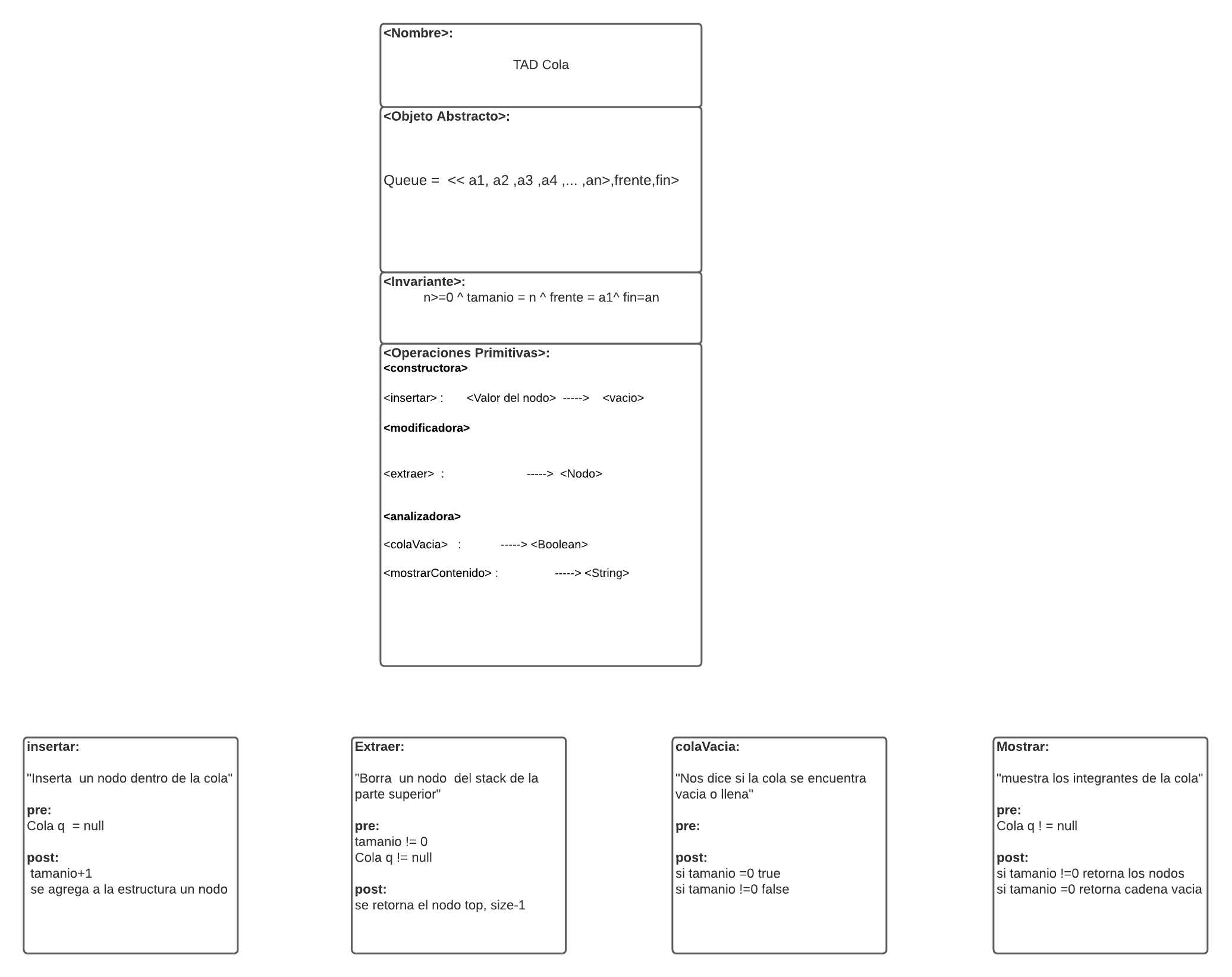
Por último, para el tercer requerimiento se eligen las alternativas 3, 5, 6 y 7 que corresponden a los algoritmos de ordenamiento de Heapsort, Merge Sort, Quicksort y Selection Sort.

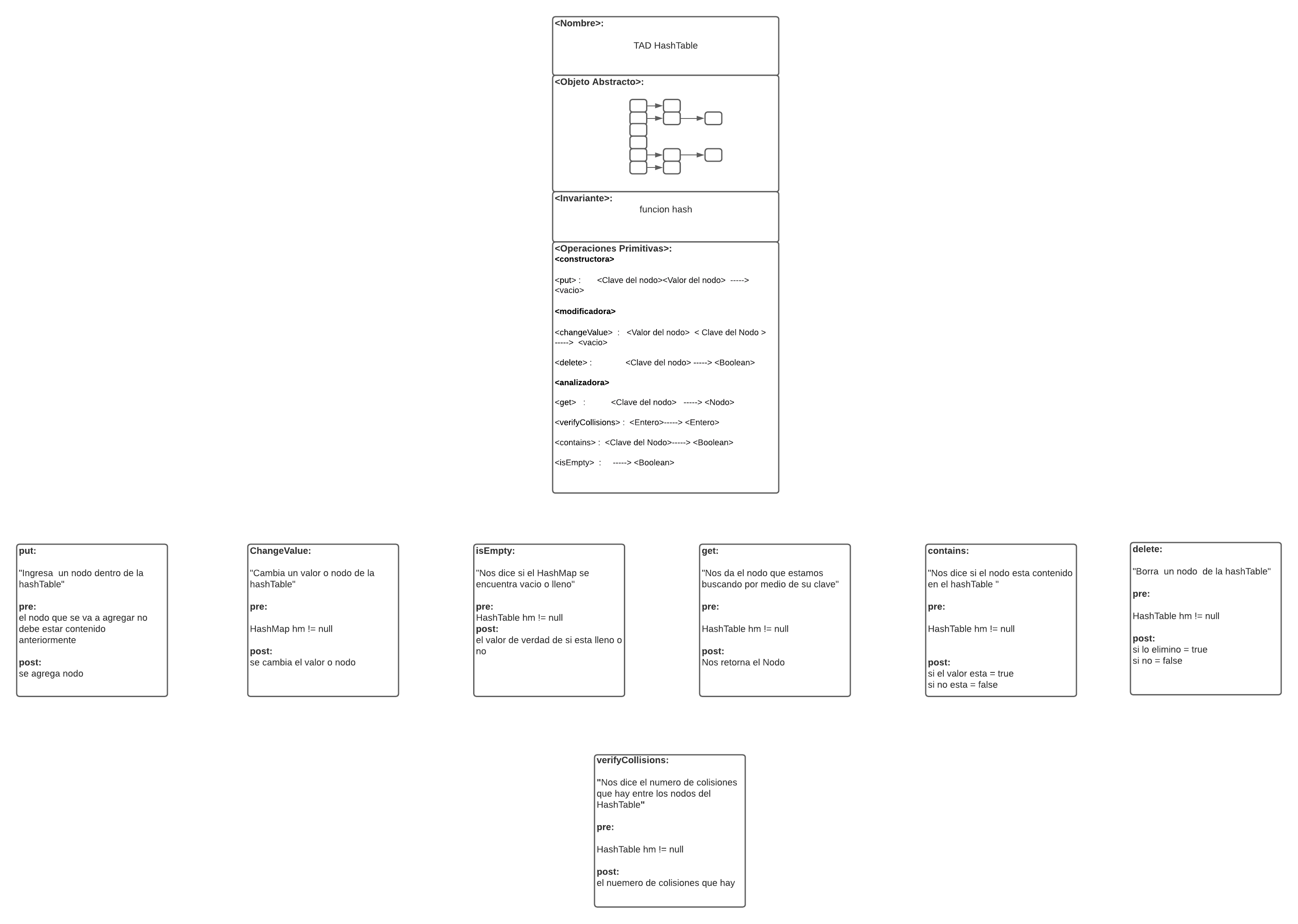
**6. Preparación de informes y especificaciones.**

**Diagrama de clases de la solución.**

****

****

****

****

**Configuración de los escenarios.**

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Clase** | **Escenario** |
| setupScenary1 | Cola |  |
| setupScenary2 | Cola | Hay una cola de tipo Integer con 4 elementos agregados en el siguiente orden. |

**Diseño de las pruebas.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** Verificar el correcto funcionamiento de método insertar() (enqueue) de la clase Cola, cuando:   1. La cola se encuentre vacía. 2. La cola tenga 5 elementos. | | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| Cola | insertar() | setupScenary1 | Un entero = 9 | Se ha agregado un entero a la cola en la primera posición exitosamente. |
| Cola | Insertar() | setupScenary2 | Un entero = 3 | Se agrega un entero a la cola y ha quedado en la ultima posición. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** Verificar que el método extraer() (dequeue) funcione correctamenete, cuando:   1. Hay elementos en la cola. 2. La cola se encuentre vacia. | | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| Cola | extraer() | setupScenary2 |  | Se elimina el primer elemento en entrar y se retorna. |
| Cola | extraer() | setupScenary1 |  | Se retorna un null, ya que no hay elementos para extraer. |

|  |  |  |
| --- | --- | --- |
| **Nombre** | **Clase** | **Escenario** |
| setupScenary1 | HashMap | Una estructura de datos lineal HashMap con un Integer como key y un String como valor ( HashMap<Integer,String> ) inicializada, sin ningún valor añadido. |
| setupScenary2 | HashMap | Una estructura de datos lineal HashMap con un Integer como key y un String como valor ( HashMap<Integer,String> ) inicializada, con 3 datos agregados:  Dato 1:  Key=14 , Value=”FA”  Dato 2:  Key=6 , Value=”MI”  Dato 3:  Key=2 , Value=”DO” |

**Diseño de las pruebas.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** Verificar el correcto funcionamiento de método put() de la clase HashMap, cuando:   1. El HashMap se encuentre vacío. | | | | |
| **Clase** | Método | Escenario | **Entradas** | **Resultado** |
| HashMap | put() | setupScenary1 | Un nuevo valor con Key=28, Value=”SOL” | Se ha agregado el valor “SOL” con key=28 al HashMap, por lo que ahora el HashMap contiene esta Key. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** Verificar el correcto funcionamiento de método contains() de la clase HashMap, cuando:   1. El HashMap contenga la llave indicada. 2. El HashMap no contenga la llave indicada. | | | | |
| **Clase** | Método | Escenario | **Entradas** | **Resultado** |
| HashMap | contains() | setupScenary1 | Un nuevo valor a insertar con KeyInsertar=28, ValueInsertar=”SOL” | (true) Se ha encontrado el valor que contiene la key=28. |
| HashMap | contains() | setupScenary1 | KeyQueNoEsta=56 | (false) No se ha encontrado el valor que contiene la key=56. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** Verificar el correcto funcionamiento de método get() de la clase HashMap, cuando:   1. El HashMap contenga la llave indicada. | | | | |
| **Clase** | Método | Escenario | **Entradas** | **Resultado** |
| HashMap | get() | setupScenary2 | Un nuevo valor a insertar con KeyInsertar=196, ValueInsertar=”CALI” | Se ha retornado el valor insertado previamente. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Objetivo de la prueba:** Verificar el correcto funcionamiento de método delete() de la clase HashMap, cuando:   1. El HashMap contenga el elemento indicado. | | | | |
| **Clase** | Método | Escenario | **Entradas** | **Resultado** |
| HashMap | delete() | setupScenary2 | Un nuevo valor a insertar con KeyInsertar=196, ValueInsertar=”CALI” (Para lograr ver que se agregó y después ya no estará) | El elemento que se encontraba en el hashMap después de usar el método delete ya no se encontró. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo: Poder sacar el dinero que el cliente posee en su cuenta de banco | | | | |
| Clase | Método | Escenario | Entradas | resultado |
| Client | retirarDinero() | Setup1() | -cantidad de dinero que desea sacar. | -valor de verdad sobre si se pudo sacar o no. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo: Poder agregar dinero al cliente en su cuenta de banco | | | | |
| Clase | Método | Escenario | Entradas | resultado |
| Client | ConsignarDinero() | Setup2() | -cantidad de dinero que desea sacar. | -se le agrega la suma de dinero que el cliente indique. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo: Poder abonar un dinero que el usuario debe por el uso de la tarjeta | | | | |
| Clase | Método | Escenario | Entradas | resultado |
| Client | PagarTarjeta() | Setup3() | -el dinero que quiere abonar a la deuda  -si es efectivo o si es por cuenta | -se le descuenta la deuda a la tarjeta  -se le descuenta la deuda a la tarjeta y se le reduce el monto de la cuenta de banco |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo: dar una copia del cliente antes de las operaciones realizadas | | | | |
| Clase | Método | Escenario | Entradas | resultado |
| Client | darCopia() | Setup4() |  | -una copia del cliente sin operaciones para deshacer sus cambios |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo: Poder agregar objetos al Stack | | | | |
| Clase | Método | Escenario | Entradas | resultado |
| Stack | push() | Setup1() | -El elemento o Nodo | -se agrega el nodo a la estructura |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo:Saber el tamaño del Stack o número de nodos | | | | |
| Clase | Método | Escenario | Entradas | resultado |
| Stack | size() | Setup2() |  | -la cantidad de nodos que hay en la estructura |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo: Borrar el elemento de arriba del stack | | | | |
| Clase | Método | Escenario | Entradas | resultado |
| Stack | Pop() | Setup3() |  | -se elimina el top de el stack |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo: Conocer si el arreglo tiene Nodos o esta vacio | | | | |
| Clase | Método | Escenario | Entradas | resultado |
| Stack | IsEmpty() | Setup4() |  | -un valor verdadero cuando esta vacío  -un valor negativo cuando esta lleno |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objetivo: Conocer el elemento tope del stack | | | | |
| Clase | Método | Escenario | Entradas | resultado |
| Stack | Top() | Setup5() |  | -Nos retorna el Nodo que se encuentra en el tope del stack |

**7. Implementación de la solución.**

**https://github.com/duvanovik/Tarea-integradora**