TAREA INTEGRADORA NO.1

JHOAN DAVID FIAT RESTREPO

DAVID JHUN KIM

JULIAN ANDRES RIVERA CARRILLO

Método de la ingeniería.

Contexto del problema.

Un gran banco requiere de un software que sea capaz de manejar el funcionamiento de una de sus sedes con mayor flujo de personas.

Fase no.1: Identificación del problema.

Identificación de necesidades y síntomas.

- La sede de este banco requiere de un software que le permite manejar de manera eficiente el gran flujo de clientes.
- No existe un software en esta sede capaz de manejar la gran cantidad de clientes.
- Se requiere de algoritmos de búsqueda eficientes para el software.
- Se requiere de un manejo adecuado de los turnos para los clientes.
- Se requiere de un manejo eficiente de los clientes registrados y de las acciones que estos decidan realizar.
- Se requiere que la información de un usuario pueda ser visualizada.
- Se requiere de un botón en el software para deshacer cambios realizados.
- Se requiere que el software posea una interfaz gráfica de usuario.

Definición del problema.

La sede de este banco requiere de un software para el manejo de grandes cantidades de clientes.

Fase no.2: Recopilación de información.

Con el fin de dar una solución optima al problema, se ha recopilado información necesaria que permita el entendimiento de este. Para ello, se ha recopilado los requerimientos funcionales y no funcionales que podrían dar solución al problema.

Requerimientos funcionales.

Requerimiento no.1: Registrar un usuario en el programa. Permite registrar un usuario a través de su nombre y su número de identidad.

Requerimiento no.2: Modificar datos de un usuario registrado. Permite modificar el nombre, número de identidad, cuenta bancaria, tarjetas de débito o crédito, fechas de pago de las tarjetas y la fecha en la que se incorporó al banco el usuario.

Requerimiento no.3: Mostrar datos de un usuario registrado. Permite visualizar los datos de un usuario, los datos que deben mostrarse son el nombre, número de identidad, cuenta bancaria, tarjetas de débito o crédito, fechas de pago de las tarjetas y la fecha de incorporación al banco.

Requerimiento no.4: Asignar turno a un usuario. Permite asignar un turno al usuario una vez que este ha sido registrado en el programa.

Requerimiento no.5: Atender turno de usuario. Permite atender un turno a un usuario para que realice operaciones en el programa.

Requerimiento no.6: Buscar un usuario registrado. Permite buscar un usuario por medio de un número de identificación.

Requerimiento no.7: Modificar monto de dinero de un usuario. Permite modificar el monto de dinero de un usuario si este decide retirar o consignar dinero en su cuenta.

Requerimiento no.8: Cancelar la cuenta de un usuario. Permite cancelar la cuenta de un usuario registrado en el programa.

Requerimiento no.9: Registrar exusuarios que han cancelado su cuenta. Permite registrar a los usuarios que han eliminado su cuenta del programa. Además, se almacenará la fecha de la cancelación y el motivo por el cual se canceló la cuenta.

Requerimientos no funcionales.

Requerimiento no.1: Realizar un botón de “Undo” o “Deshacer”. El programa deberá contar con un botón para deshacer cambios o modificaciones realizadas durante el uso del programa.

Requerimiento no.2: Implementar algoritmos de ordenamiento. El programa debe contar con al menos un algoritmo de ordenamiento cuya complejidad temporal promedio sea de $O(n^2)$ y tres algoritmos de ordenamiento mucho más eficientes.

Fase no.3: Búsqueda de soluciones creativas.

Con el propósito de hallar la mejor solución para este problema, decidimos hacer uso de una de las técnicas operacionales más comunes para generar ideas creativas. En este caso, elegimos una Lluvia de ideas con el objetivo de reunir las

mejores alternativas posibles pensando en el uso de estructuras de datos y algoritmos de ordenamiento.

Posibles alternativas de solución.

- ArrayList.
- Lista enlazada.
- Lista doblemente enlazada.
- Hash table.
- Cola.
- Pila.
- Array.
- Árbol binario de búsqueda.
- Algoritmo de ordenamiento tipo burbuja.
- Algoritmo de ordenamiento tipo inserción.
- Algoritmo de ordenamiento tipo selección.
- Algoritmo de ordenamiento tipo Montones.
- Algoritmo de ordenamiento tipo Conteo.
- Algoritmo de ordenamiento tipo Mezclas.
- Algoritmo de ordenamiento tipo rápido.

Fase no.4: Transición de las ideas a los diseños preliminares.

En esta fase, hemos descartado las alternativas que no son factibles y las hemos asignado en dos grupos. Para las estructuras de datos, hemos descartado el uso de arrays y arboles binarios, debido a que, no presentan una buena aproximación para la solución del problema. Además, hemos descartado el uso de los algoritmos de ordenamiento tipo inserción y selección, porque se requiere al menos uno de estos en la implementación de la solución.

Alternativa no.1.

Implementar las estructuras de datos Arraylist, lista enlazada y lista doblemente enlazada y como algoritmos de ordenamiento, tipo burbuja, tipo por montones, tipo conteo y tipo mezclas.

- Las estructuras de datos ArrayList, lista enlazada y lista doblemente enlazada poseen la característica de ser dinámicos, lo cual puede ayudar a almacenar muchos datos.
- A pesar de ser estructuras de datos lineales (Arraylist, lista enlazada y lista doblemente enlazada), no tienen una representación adecuada de la problemática.
- Los algoritmos de ordenamiento de tipo burbuja, montones, conteo y mezcla son levemente ineficientes en comparación con la alternativa 2.

Alternativa no.2.

Implementar las estructuras de datos Hash table, cola y pila y como algoritmos de ordenamiento, tipo burbuja, tipo por montones, tipo conteo y tipo rápido.

- Las estructuras de datos Hash table, cola y pila poseen la característica de ser dinámicos, lo que contribuye a almacenar muchos datos.
- La estructura de datos Hash table, es un poco más eficiente en consumo de memoria comparado con las estructuras de datos de la alternativa 1.
- Las estructuras de datos Cola y Pila, presentan una mejor representación de la problemática. Por ejemplo, con esta estructura se puede modelar el funcionamiento de una fila de espera de un establecimiento.
- Los algoritmos de ordenamiento de tipo burbuja, tipo por montones, tipo conteo y tipo rápido; son más eficientes que los algoritmos de la alternativa 1.

Fase no.5: Evaluación y selección de la mejor solución.

Criterios de evaluación para las alternativas.

Para la selección de la mejor solución, decidimos asignar unos criterios con un respectivo valor para determinar cual de las alternativas era la más viable.

Criterio no.1. Precisión de la solución. La alternativa entrega una solución:

[2] Exacta.

[1] Aproximada.

Criterio no.2. Eficiencia de los algoritmos de ordenamiento. La alternativa presenta una eficiencia de los algoritmos:

[3] Buena.

[2] Medio.

[1] Mala.

Criterio no.3. Complejidad en la implementación de las estructuras de datos para el desarrollador. La alternativa presenta un nivel de implementación:

[3] No complicada.

[2] Medio.

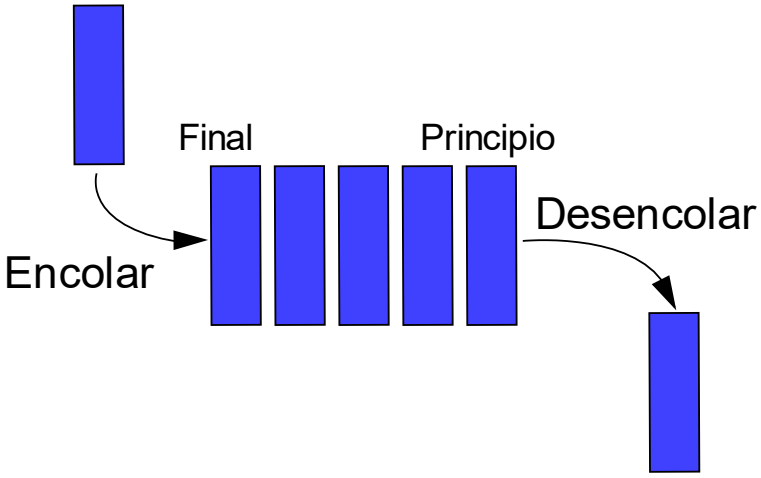
[1] Complicada.

Evaluación.

	Criterio no.1	Criterio no.2	Criterio no.3	Total

Alternativa no.1	Aproximada [1]	Medio [2]	No complicada [3]	6 puntos.
Alternativa no.2	Exacta [2]	Buena [3]	Medio [2]	7 puntos.

TAD'S ESTRUCTURAS DE DATOS

TAD Queue.

{inv:}
Operaciones primitivas: • Enqueue: Nodo → Queue • Dequeue: → Queue • Front: → Queue • IsEmpty: → Boolean • Size: → Entero

Operaciones modificadoras.

Enqueue ()
"Agrega un nodo a la cola" {pre: Nodo no está vacío} {post: Queue.Size() ≠ 0}

Dequeue ()

“Elimina y retorna el nodo que esta al principio de la cola”

{pre: Queue.Size() ≠ 0}

{post: Queue.Nodo & Queue.Size() – 1}

Operaciones analizadoras.

Front ()

“Retorna el nodo que esta al principio de la cola”

{pre: Queue.Size() ≠ 0}

{post: Queue}

IsEmpty ()

“Comprueba el estado de la cola”

{pre: TRUE}

{post: Retorna TRUE si la cola esta vacía. De lo contrario, retorna FALSE}

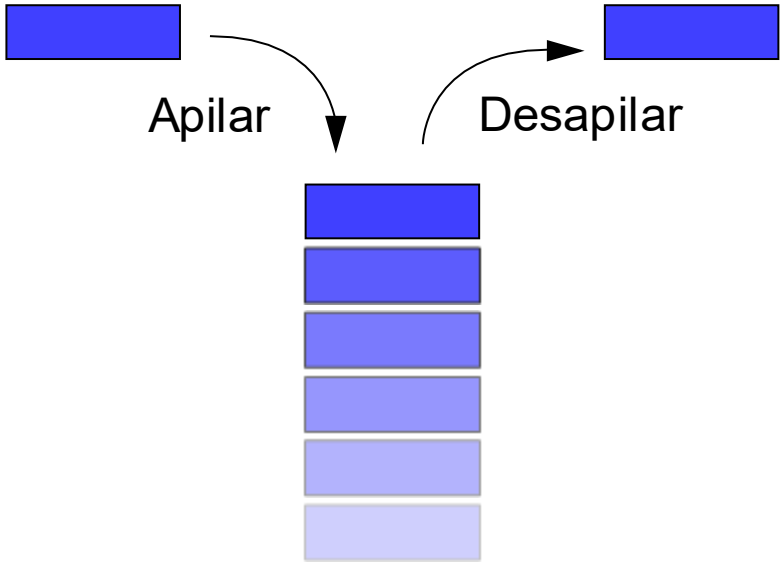
Size ()

“Retorna el número de elementos almacenados en la cola”

{pre: TRUE}

{post: Retorna un Z+ *n*. De lo contrario, retorna 0}

TAD Stack.



{inv:}

Operaciones primitivas:

- Push: Nodo → Stack
- Pop: → Stack
- Top: → Stack
- IsEmpty: → Boolean
- Size: → Entero

Operaciones modificadoras.

Push ()

"Agrega un nodo a la pila"

{pre: Nodo no está vacío}

{post: Stack.Size() ≠ 0}

Pop ()

"Retorna y elimina el nodo que se encuentra en la cima de la pila"

{pre: Stack.Size() ≠ 0}

{post: Stack.Nodo & Stack.Size() – 1}

Top ()

“Retorna el nodo que está en la cima de la pila”

{pre: Stack.Size() ≠ 0}

{post: Stack.Nodo}

Operaciones Analizadoras.

IsEmpty ()

“Comprueba el estado de la pila”

{pre: TRUE}

{post: Retorna TRUE si la pila esta vacía. De lo contrario, retorna FALSE}

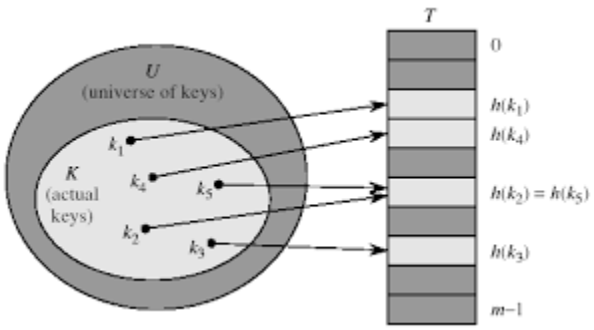
Size ()

“Retorna el número de elementos almacenados en la pila”

{pre: TRUE}

{post: Retorna un Z+ *n*. De lo contrario, retorna 0}

TAD Hash table.



{inv: HashTable.length > HashTable.size}

Operaciones primitivas:

- Add: Key, Value → HashTable
- Search: Key → HashTable
- Delete: Key → HashTable
- returnHash: → Lista

Operaciones modificadoras.

Add ()

“Agrega un nuevo nodo a la tabla Hash”

{pre: HashTable.length ≥ HashTable.size}

{post: HashTable.size + 1}

Search ()

“Busca un nodo en la tabla Hash con una clave ingresada”

{pre: HashTable.size ≠ 0}

{post: HashTable}

Delete ()

“Elimina y retorna un nodo de la tabla Hash con una clave ingresada”

{pre: HashTable.size ≠ 0}

{post: HashTable.size – 1 ∧ HashTable}

Operaciones analizadoras.

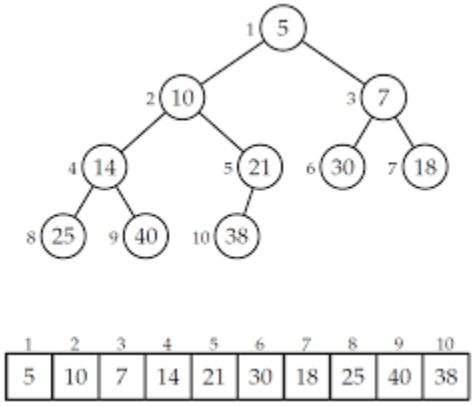
ReturnHash ()

“Retorna una lista de todos los elementos almacenados en la tabla Hash”

{pre: HashTable.size ≠ 0}

{post: Lista}

TAD Heap.



{inv:}

Operaciones primitivas:

- Heap: → Heap
- BuildHeap: → Heap
- Insert: Nodo → Heap
- Heapify: Entero → Heap
- GetHeapSize: → Entero
- GetElements: → Lista

Operaciones constructoras.

Heap ()

"Crea un heap sin elementos"

{pre: TRUE}

{post: Heap.size = 0}

Operaciones modificadoras.

BuildHeap ()

"Modela la estructura del Heap"

{pre: Heap.size ≠ 0}

{post: Heap}

Insert ()

"Agrega un nodo al Heap"

{pre: Heap.size ≥ 0}

{post: Heap.size + 1}

Heapify ()

“Intercambia el elemento más pequeño del Heap con la raíz”

{pre: Heap.size ≠ 0}

{post: Heap}

Operaciones analizadoras.

GetHeapSize ()

“Retorna el número de elementos almacenados en el Heap”

{pre: TRUE}

{post: Retorna un Z+ n. De lo contrario retorna 0}

GetElements ()

“Retorna una lista de todos los elementos almacenados en el Heap”

{pre: Heap.size ≠ 0}

{post: Lista}

Diseño de los casos de prueba.

Clase: HashTable.

Prueba: El método permite buscar a un cliente dentro de la Hash table.

Clase	Método probado	Escenario	Valores de entrada	Resultado

HashTable	Search (K key)	Setup (): Se crea una nueva tabla Hash a la cual se agregan 7 clientes con una clave asignada.	Una clave K la cual tiene asignado un usuario.	Verdadero.

Clase: HashTable.

Prueba: Comprobar que no pueden agregar elementos repetidos a la tabla Hash.

Clase	**Método probado**	**Escenario**	**Valores de entrada**	**Resultado**
HashTable	Add (K key, V value)	Setup (): Se crea una nueva tabla Hash a la cual se agregan 7 clientes con una clave asignada.	Un cliente con valor asignado “c” y un valor de “Violetta”	Verdadero.

Clase: HashTable.

Prueba: El método permite eliminar clientes almacenados en la Hash table.

Clase	**Método probado**	**Escenario**	**Valores de entrada**	**Resultado**
HashTable	Delete (K key)	Setup (): Se crea una nueva tabla Hash a la cual se agregan 7 clientes con una clave asignada.	Se ingresan diferentes claves con valor {“a”, “c”, “e”, “g”, “i”, “k”, “m”}	Verdadero.

Clase: Queue.

Prueba: El método permite obtener el primer elemento de la cola.

Clase	**Método probado**	**Escenario**	**Valores de entrada**	**Resultado**

Queue	Front ()	Setup1 (): Se crea una nueva cola vacía.	Ninguno.	Falso.
Queue	Front ()	Setup2 (): Se crea una nueva cola vacía y se agrega una cadena.	Ninguno.	Verdadero.

Clase: Queue.

Prueba: El método permite agregar un elemento a la cola.

Clase	**Método probado**	**Escenario**	**Valores de entrada**	**Resultado**
Queue	Enqueue ()	Setup2 (): Se crea una nueva cola vacía y se agrega una cadena.	Una cadena con valor “Davif Fiat”.	Verdadero.

Clase: Queue.

Prueba: El método permite eliminar el primer elemento de la cola.

Clase	**Método probado**	**Escenario**	**Valores de entrada**	**Resultado**
Queue	Dequeue ()	Setup3 (): Se crea una nueva cola vacía y se agregan 3 elementos.	Ninguno.	Verdadero.

Clase: Stack.

Prueba: El método permite retornar el último elemento de la pila.

Clase	**Método probado**	**Escenario**	**Valores de entrada**	**Resultado**
Stack	Top ()	Setup1 (): Se crea una nueva pila	Ninguno.	Verdadero.

		vacía y se agregan 3 elementos.		

Clase: Stack.

Prueba: El método permite agregar un elemento a la pila.

Clase	**Método probado**	**Escenario**	**Valores de entrada**	**Resultado**
Queue	Push ()	Setup1 (): Se crea una nueva pila vacía y se agregan 3 elementos.	Se agrega un elemento con valor “Camila Giraldo”	Verdadero.

Clase: Stack.

Prueba: El método permite eliminar el último elemento de la pila.

Clase	**Método probado**	**Escenario**	**Valores de entrada**	**Resultado**
Queue	Pop ()	Setup1 (): Se crea una nueva pila vacía y se agregan 3 elementos.	Ninguno.	Verdadero.