**Informe Requerimientos Funcionales.**

**Problema:** Simular un proceso de compra en un BookStore.

**Requerimientos Funcionales.**

**El sistema debe permitir:**

**R1**: Ingresar de estanterías de libros.

**R2**: cantidad de cajeros a utilizarse durante la jornada.

**R3**: ingresar cantidad de clientes de la jornada.

**R4**: Mostrar el orden de salida de los clientes.

**R5**: Mostrar el valor de cada compra.

**R6**: Mostrar el orden en que quedaron empacados los libros.

**Requerimientos No Funcionales.**

Rendimiento: En el momento que él se ingrese una estantería de libros muy grande o una cantidad de usuarios, el programa tendrá problemas para procesar el algoritmo dependiendo de la capacidad del equipo.

Eficiencia: el tiempo de salida de los clientes dependerá:

La cantidad de libros que el cliente compre, y la cantidad de cajas que estén disponible en la jornada si la cantidad de clientes es demasiada y las cajas de la jornada son pocas tardara mucho tiempo en salir los clientes

Usabilidad: el usuario solo contará en la interfaz los espacios correspondientes para cargar los datos que tienen que tener la información correspondiente al programa para que funcione correctamente de lo contrario cualquier tipo de error al cargar los datos no se podrá hacer las operaciones para calcular el tiempo de salida y el orden.

**Recopilación de información**:

Para resolver el problema de cómo simular el proceso de compras en un BookStore se tuvieron encuneta varios aspectos para que la aplicación sea lo más eficaz posible ya que el proceso de compra tiene diferentes procesos en los cuales vamos a utilizar diferentes estructuras de datos que van a modelar diferentes partes del problema.

**Estructura de Datos:**

En ciencias de la computación, una estructura de datos es una forma particular de organizar datos en una computadora para que puedan ser utilizados de manera eficiente.

Fuente:

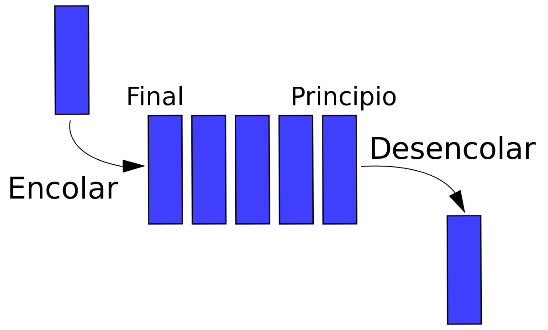
<https://es.wikipedia.org/wiki/Estructura_de_datos>

**Tipos de estructuras de datos**

**Cola:**

Una cola (también llamada fila) es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción Push se realiza por un extremo y la operación de extracción pop por el otro. También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.

Ejemplo:



Procedimiento Elemento peek() //Retorna el último elemento añadido a la cola.

Si queueEstaVacia entonces arroja excepción Queue underflow

Fin si

Retorna primero.elemento.

FinProceso

Procedimiento enqueue(Elemento elementoAgregar) //Encola un elemento.

UltimoNodoElemento <- ultimo

ultimo <- nuevoNodo

ultimo.elemento <- elementoAgregar

ultimo.siguiente <- nil

Si queueEstaVacia entonces primero <- ultimo

Fin si

De lo contrario entonces UltimoNodoElemento.siguiente <- ultimo

FinProceso

Procedimiento Elemento dequeue() //Desencola un elemento

Si queueEstaVacia entonces arroja excepción Queue underflow

Fin si

Elemento <- primero.elemento

Primero <- primero.siguiente

Si queueEstaVacia entonces ultimo <- nil

Retorna Elemento

Fin si

Fuente:

<https://es.wikipedia.org/wiki/Cola_(inform%C3%A1tica)>

**Pila:**

Una pila (Stack en inglés) es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»). Esta estructura se aplica en multitud de supuestos en el área de informática debido a su simplicidad y capacidad de dar respuesta a numerosos procesos.

Ejemplo:



Procedimiento push (Elemento elementoAgregar) //Agrega un elemento a la pila.

UltimoNodoEntrar <- primero

primero <- NuevoNodo

primero.elemento <- elementoAgregar

primero.siguiente <- PrimerNodo

FinProceso.

Procedimiento Elemento pop() //Retorna y elimina el último elemento agregado a la pila.

Si stackEstaVacia entonces arroja excepción Stack underflow

Fin si

De lo contrario

Elemento <- primero. elemento

Primero <- primero.siguiente

Retorna Elemento

FinProceso.

Procedimiento Elemento peek() //Retorna el último elemento de la pila.

Si stackEstaVacia entonces arroja excepción Stack underflow

Fin si

Retorna primero.elemento.

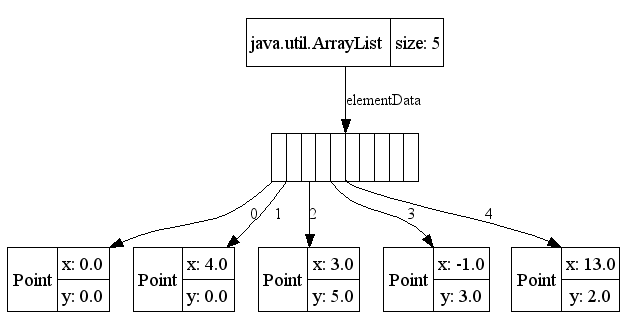
FinProceso

Fuente:

<https://es.wikipedia.org/wiki/Pila_(inform%C3%A1tica)>

**ArrayList:**

La clase ArrayList en Java, es una clase que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el número de elementos que almacena, lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño como pasa con los Arrays. Para ejemplificar, se tomó un ejemplo de internet donde podemos ver la estructura del ArrayList:



Pseudocodigo:

Procedimiento CrearArray.

Definir num como entero;

Dimension num[5];

FinProceso

Proceso AgregarValoresArray

Definir num como entero;

Dimension num[5];

num[1] <- 5;

num[2] <- 10;

num[3] <- 15;

num[4] <- 20;

num[5] <- 25;

FinProceso

Procedimiento verDatosArray

Definir num como entero;

Dimension num[5];

num[1] <- 5;

num[2] <- 10;

num[3] <- 15;

num[4] <- 20;

num[5] <- 25;

Para i<-1 Hasta 5 Con Paso 1 Hacer

leer num[i];

Fin Para

Para i<-1 Hasta 5 Con Paso 1 Hacer

escribir num[i];

Fin Para

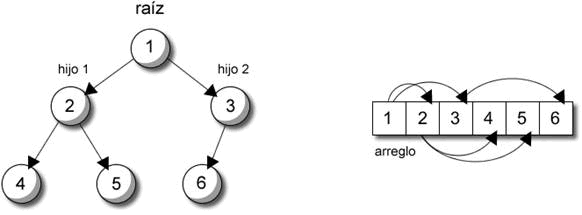
FinProceso

Fuente:

https://es.wikipedia.org/wiki/Array\_din%C3%A1mico

**Montículos:**

En computación, un montículo (o Heap en inglés) es una estructura de datos del tipo árbol con información perteneciente a un conjunto ordenado. Los montículos máximos tienen la característica de que cada nodo padre tiene un valor mayor que el de cualquiera de sus nodos hijos, mientras que, en los montículos mínimos, el valor del nodo padre es siempre menor al de sus nodos hijos.



Fuente:

<https://es.wikipedia.org/wiki/Mont%C3%ADculo_(inform%C3%A1tica)>

<https://altenwald.org/2012/02/12/monticulos/>

**Tabla Hash:**

Una tabla hash, matriz asociativa, Hashing, mapa hash, tabla de dispersión o tabla fragmentada es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos (teléfono y dirección, por ejemplo) almacenados a partir de una clave generada (usando el nombre o número de cuenta, por ejemplo). Funciona transformando la clave con una función hash en un hash, un número que identifica la posición (casilla o cubeta) donde la tabla hash localiza el valor deseado.

Ejemplo:



Fuente:

<http://lineadecodigo.com/java/usar-una-hashtable-java/>

**Clasificación**

Las Estructuras de datos pueden clasificarse en lineales y no lineales. Una Estructura de datos es lineal si sus elementos forman una secuencia o, en otras palabras, una lista lineal.

***Operaciones con estructuras de datos***

**Recorrido**

Implica el acceder a cada registro una única vez, aunque uno o más ítems del registro sean procesados. (Este acceso o procesamiento también se denomina a veces por el término «visitar» el registro).

**Búsqueda**

Implica la localización de un registro caracterizado por una determinada clave o también el acceso a todos los registros que cumplan una o más condiciones.

**Inserción**

Cuando añadimos nuevos registros a la estructura.

**Eliminación**

Operación de borrado de un registro de la estructura.

**Ordenación**

Es la operación de clasificar los registros conforme a un orden lógico determinado (por ejemplo, alfabéticamente, de acuerdo a una clave de nombre, o numérica, de acuerdo a alguna clave de número, tal como número de Seguridad Social o de inventario).

**Mezcla**

Es la operación de combinar dos archivos previamente ordenados en uno único que también lo está.

**Búsqueda de ideas y soluciones creativas:**

Soluciones creativas para la interfaz.

* Crear una interfaz donde se visualice las tres entradas dos donde se van a cargar los datos de las estanterías, número de usuarios y la otra donde se ingresará el número de cajas para la jornada
* Crear una interfaz donde el usuario pase por cada uno de los puntos e ir simulando el proceso gráficamente desde que escoge los libros en el punto uno hasta donde hace la fila y paga por la compra
* Crear una interfaz donde el usuario cargarte un archivo de texto con toda la información a procesar

**Transición de las ideas a los diseños preliminares:**

Se han escogido las 3 mejores estructuras, las cuales fueron seleccionados debido a que cumplen con las especificaciones que de nuestros respectivos requerimientos, una de ellas fue una cola para ordenar la fila de los clientes a la hora de pasar a pagar los libros, posteriormente para las plataformas se seleccionó una Hash Table y por último para ordenar los libros que el cliente desea compra se seleccionó una pila.

1. **TAD Biblioteca:**

|  |
| --- |
| **TAD BILIOTETA** |
| Biblioteca = { AmounCashRegister<amounCashRegister>,AmountClients< AmountClients >, AmountShelves< AmountShelves >} |
| {Inv: no se puede crear usuarios repetidos, el usuario es un usuario único} |
| **Operaciones primitivas**  - CrearBibliote  - getAmountCashRegisterBiblioteca Biblioteca ->int  - getAmountShelvesBiblioteca Biblioteca ->int  - getClientsBiblioteca Biblioteca ->int  - setAmountCashRegisterBiblioteca int ->Biblioteca  - setAmountClientsBiblioteca int ->Biblioteca  - setAmountShelvesBiblioteca int ->Biblioteca |

|  |
| --- |
| CrearBibliote ()  “Crear una biblioteca con los datos vacíos”  {pre: TRUE}  {post: biblioteca = {AmounCashRegister =” ”, AmountClients=” “,AmountShelves=” ”}} |

|  |
| --- |
| getAmountCashRegister ()  “Retorna si está registrado ”  {pre: TRUE}  {post: biblioteca = {AmounCashRegister =” ”, AmountClients=” “,AmountShelves=” ”}} |

|  |
| --- |
| getAmountShelves ()  “Crear una biblioteca con los datos vacíos”  {pre: TRUE}  {post: biblioteca = {AmounCashRegister =” ”, AmountClients=” “,AmountShelves=” ”}} |

|  |
| --- |
| getClients ()  “Crear una biblioteca con los datos vacíos”  {pre: TRUE}  {post: biblioteca = {AmounCashRegister =” ”, AmountClients=” “,AmountShelves=” ”}} |

|  |
| --- |
| setAmountCash ()  “Crear una biblioteca con los datos vacíos”  {pre: TRUE}  {post: biblioteca = {AmounCashRegister =” ”, AmountClients=” “,AmountShelves=” ”}} |

|  |
| --- |
| setAmountClients ()  “Crear una biblioteca con los datos vacíos”  {pre: TRUE}  {post: biblioteca = {AmounCashRegister =” ”, AmountClients=” “,AmountShelves=” ”}} |

|  |
| --- |
| setAmountShelves ()  “Crear una biblioteca con los datos vacíos”  {pre: TRUE}  {post: biblioteca = {AmounCashRegister =” ”, AmountClients=” “,AmountShelves=” ”}} |

1. **TAD Cola:**

|  |
| --- |
| **TAD Cola** |
| Resultado de imagen para cola en java |
| {Inv: Puedo modificar el último elemento añadido y puedo extraer únicamente el ultimo y primer elemento de la cola } |
| **Operaciones primitivas**  - CrearCola  - isEmpty - Cola -> boolean  - size - Cola -> int  -peek - Cola -> Cola  - enqueue - Cola Elemento -> Cola  - dequeue - Cola -> Cola |

|  |
| --- |
| CrearCola ()  “Crear una cola vacía”  {pre: TRUE}  {post: cola = {Encolar =””, Descolar=””}} |

|  |
| --- |
| isEmpty ()  “Retorna si está o no llena la cola”  {pre: TRUE}  {post: boolean} |

|  |
| --- |
| size ()  “Retorna si está o no llena la cola”  {pre: TRUE}  {post: Entero} |

|  |
| --- |
| Peek()  “Retorna el último elemento añadido a la cola”  {pre: TRUE}  {post: Cola-peek = <Elemento>} |

|  |
| --- |
| enqueue (Elemento elemento)  “Agregar un elemento al final de la cola”  {pre: TRUE}  {post: Cola.enqueue = <elemento>} |

|  |
| --- |
| dequeue ()  “Retorna y elimina el último elemento de la cola”  {pre: TRUE}  {post: Cola.dequeue = <elemento>} |

**Nombre**: TAD PILA libros:

|  |
| --- |
| **TAD Pila** |
| Imagen relacionada |
| {Inv: Solo puede tener referente del primer objeto de la pila. } |
| **Operaciones primitivas**  - CrearPila  - isEmpty - Pila -> boolean  - size - Pila -> int  - push - Cola -> Pila  - pop - Pila -> Pila  - top - Pila -> Pila |

|  |
| --- |
| CrearPila ()  “Crear una pila vacía”  {pre: TRUE}  {post: cola = {Apilar =””, Despilar=””}} |

|  |
| --- |
| isEmpty ()  “Retorna si está o no llena la pila”  {pre: TRUE}  {post: boolean} |

|  |
| --- |
| Size ()  “Retorna si está o no llena la pila”  {pre: TRUE}  {post: Entero} |

|  |
| --- |
| Push(Elemento elemento)  “Agregar un elemento al final de la pila”  {pre: TRUE}  {post: Cola.push = “elemento”} |

|  |
| --- |
| pop()  “Retorna el último elemento y a su vez lo elimina”  {pre: TRUE}  {post: pila.pop = <Elemento>} |

|  |
| --- |
| Top ()  “Retorna y el ultimo elemento”  {pre: TRUE}  {post: Cola.top= <Elemento>} |

**NOMBRE:** TAD Tabla Hash.

|  |
| --- |
| **TAD HASH TABLE** |
|  |
| {Inv: Para cada llave k hay un index en el array b tal que hash (k) =b. } |
| **Operaciones primitivas**  - CrearHashTable  - clear - HashTable  - contains - HashTable Elemento -> boolean  - containsValue - HashTable Elemento -> boolean  - elements - HashTable -> int  - put - HashTable Elemento,llave -> HashTable  - equals - HashTable Elemento -> boolean  - isEmpty - HashTable -> boolean  - remove - HashTable Elemento -> HashTable  - size - HashTable -> int |

Complejidad espacial:

Para la complejidad espacial de la Hash table con respecto a la que usa Java, está dada entre O (1) y O (n) en el peor de los casos.

Complejidad temporal:

Los métodos de buscar están dados en O (1) los demás están dado en O (n log n) y en el peor del caso está dado en O (n), en este caso los cálculos no serán dados, ya que se está usando un hash table por parte de Java

**Evaluación y selección de la mejor solución**

|  |  |  |  |
| --- | --- | --- | --- |
| **Criterios** | | **SI** | **NO** |
| A | El algoritmo es estable. | 2 | 1 |
| B | El algoritmo es fácil de entender. | 3 | 1 |
| C | El algoritmo tiene complejidad lineal | 0 | 4 |
| D | ¿El algoritmo tiene la misma complejidad temporal y espacial? | 3 | 1 |
| E | El algoritmo tarda mucho tiempo en terminarse de ejecutar con una misma cantidad de datos. | 0 | 3 |
| F | El algoritmo es no comparativo | 3 | 1 |
| G | El algoritmo en general es de complejidad O(1) o menor a O() | 3 | 1 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Estructura/Criterio** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **TOTAL** |
| **Arrays** | **2** | **3** | **0** | **3** | **0** | **1** | **1** | **10** |
| **Pila** | **2** | **3** | **4** | **4** | **3** | **3** | **3** | **21** |
| **Cola** | **2** | **3** | **1** | **1** | **3** | **3** | **3** | **18** |
| **Tabla Hash** | **2** | **3** | **4** | **3** | **3** | **3** | **3** | **21** |
| **Montículo** | **2** | **1** | **4** | **1** | **3** | **3** | **3** | **17** |

**Preparación informe y especificaciones**

Para concluir, después de realizar un amplio estudio y análisis de cada una de las estructuras presentadas anteriormente, y escogidas bajo unos criterios especificados en el punto anterior; seleccionamos las 3(tres) siguientes estructuras: pila, colas y tabla hash. Con estas se realizará la implementación de la solución, para así satisfacer los requerimientos del problema a tratar, es por lo anterior que ahora en adelante podemos comenzar a implementar todo lo aquí expresado.