Tarea Integradora I: Tienda de videojuegos

Integrantes:

Diana Sofia Olano Montaño (A00369468)

Angélica Corrales Quevedo (A00367954)

Keren López Córdoba (A00368902)

Profesor: Johnatan Garzón

Algoritmos y estructuras de datos, grupo 1

Universidad Icesi

Cali, 28 de septiembre del 2021

Método de la ingeniería

Contexto de la problemática

Una tienda de videojuegos que se ubicará en la ciudad de Cali quiere prestar sus servicios de una manera innovadora, por lo que desea realizar un programa que le permita conocer a sus clientes cómo sería el proceso de compra por medio de una simulación.

Fase 1: Identificación del problema

Identificación de necesidades y síntomas:

- La solución al problema debe tener en cuenta las secciones en las que se divide el funcionamiento del proceso de compra de la tienda de videojuegos.
- La solución al problema debe incluir una interfaz gráfica con el fin de que la interacción entre el usuario y el programa sea más amigable.
- Los usuarios de la tienda de videojuegos requieren acceder a un catálogo digital del negocio.
- Los usuarios de la tienda de videojuegos requieren crear una lista de compras que incluya un código que les permita ingresar a la tienda.
- Los usuarios de la tienda de videojuegos requieren realizar una búsqueda eficiente de su lista de videojuegos en las estanterías del establecimiento, asignándole la mejor ruta.
- Los usuarios de la tienda de videojuegos requieren recoger las copias físicas de su listado de juegos siguiendo el orden suministrado en el punto anterior.
- Los usuarios de la tienda de videojuegos requieren pagar su listado de videojuegos realizando una única fila que se forma teniendo en cuenta el tiempo de llegada (tomado desde el ingreso a la tienda hasta finalizar la recolección de todos los juegos de las estanterías).
- La tienda no tiene un sistema de software que le permita conocer el proceso de compra a sus clientes.

Definición del Problema:
La tienda de videojuegos requiere el desarrollo de un programa de software que permita simular el proceso de compra de sus clientes.

Especificación de requerimientos funcionales

R1. Agregar cajeros. Indicar la cantidad de cajeros a utilizarse durante toda la jornada.

R2. Agregar estanterías. Indicar la cantidad de estanterías disponibles dentro del establecimiento. Los indicadores serán asignados por el programa, en este caso son las letras del abecedario (A-Z). No se permite que una estantería tenga el mismo identificador que otra.

R3. Agregar un juego al catálogo de juegos. Se debe ingresar el código del juego (el cual consta solo de 3 dígitos), la cantidad de ejemplares disponibles, la estantería donde está

ubicado y el precio de dicho juego. No se podrá añadir un juego con el mismo código que otro.

R4. Asignar un código o registrar la cédula de los clientes en el orden en que entraron a la tienda.

R5. Agregar juegos al listado del cliente. Los juegos serán se podrán seleccionar de los que han sido añadidos previamente al programa. No se permite que un cliente agregue un juego más de una vez al listado.

R6. Ordenar el listado de los juegos acorde con la ubicación de las estanterías de tal manera que el comprador siga la mejor ruta (si el juego se encuentra agotado, su código no aparecerá en la lista ordenada final). En esta sección 2, el cliente puede utilizar dos algoritmos distintos de ordenamiento para cumplir dicha tarea, los cuales son Insertion y Selection Sort. Cada cliente sale en el orden de llegada al establecimiento con una unidad de tiempo de diferencia entre ellos.

R7. Asignar un cesto automatizado para ubicar los juegos que va encontrando el cliente, colocándolos uno encima del otro y, siguiendo el orden de los juegos suministrado en la sección 2. Este proceso corresponde a la sección 3. Cada cliente se demora 1 unidad de tiempo en recoger un juego, por tal, el tiempo total al salir de esta sección será el valor previo proveniente de la sección 2 más la cantidad que se toma en recoger los ejemplares.

R8. Establecer el orden con el que los clientes ingresan a la única fila de cajas existente, teniendo en cuenta desde el tiempo en el que entraron a la tienda, más lo que se tomaron recogiendo los juegos de cada estantería. Si dos usuarios se toman el mismo tiempo, va primero aquel que estaba antes a la salida de la sección 2. Además, en este caso, pasan tantos clientes como puntos disponibles haya para ser atendidos uno a uno en cada uno de ellos. El orden de salida de los clientes dependerá del tiempo que se demore la atención de cada uno en el punto de pago, es decir, por la cantidad de juegos que se vayan a comprar.

R9. Facturar los juegos. Se debe indicar el valor de compra a pagar por el cliente. El último juego añadido al cesto será el primero en ser facturado.

R10. Empacar los juegos. Se debe indicar el orden en que quedaron empacados los juegos. Hay que tener en cuenta que, como los juegos vienen en un cesto, el último juego añadido será el primero en ser empacado.

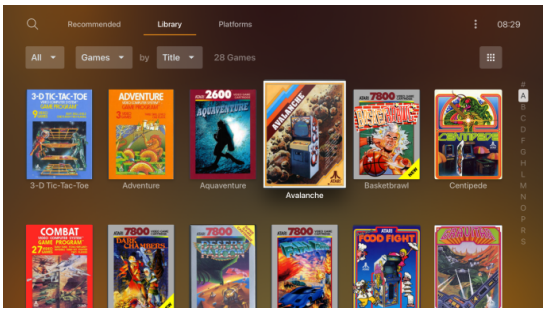
Fase 2: Recopilación de la información necesaria

Definiciones

- **Teoría de las colas:** Tiene como objetivo fundamental predecir las longitudes de las filas y los tiempos de espera.

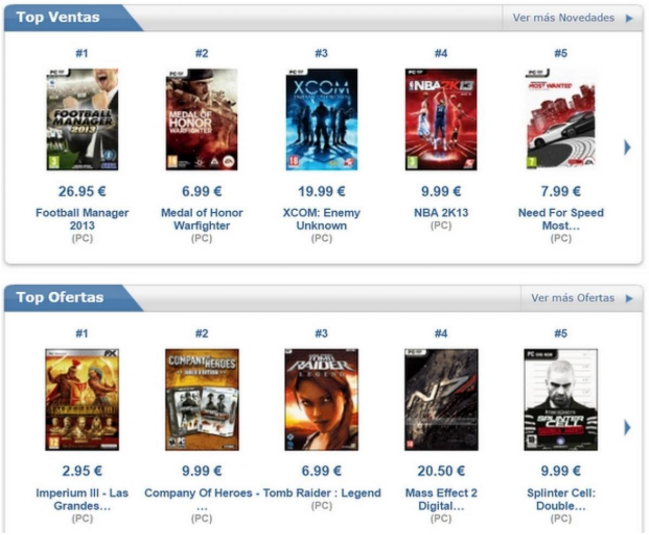
  Derivandodx. (2017). Teoría de colas | ¿Cuál es la fila más rápida del supermercado? Recuperado de https://www.derivandodx.com/cual-es-la-fila-mas-rapida-del-super-teoria-de-colas/

- **Ejemplos de catálogos de videojuegos que tengan similitud**



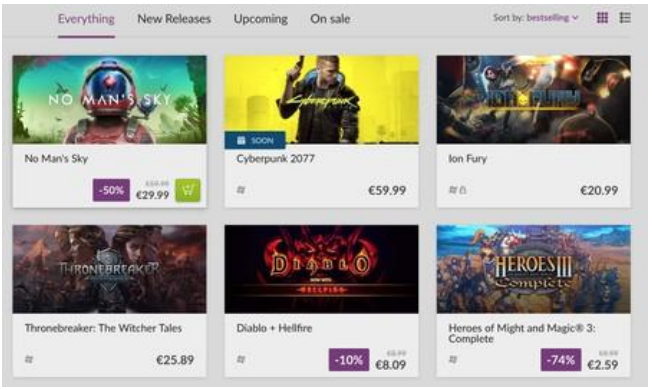
Plataforma web de videojuegos retro

Gamer, N. (2021, 28 enero). *Plex Arcade: una nueva plataforma de streaming de videojuegos retro*. MARCA. Recuperado de https://www.marca.com/claro-mx/esports/2021/01/28/6012125546163f216a8b459c.html



Plataforma web de videojuegos de acción

Pascual, J. A. (2016, 17 marzo). *Las 10 mejores tiendas de juegos digitales para PC*. ComputerHoy. Recuperado de https://computerhoy.com/listas/zona-gaming/10-mejores-tiendas-juegos-digitales-pc-40501



Plataforma web de videojuegos de acción

Sabán, A. (2019, 21 agosto). *Porque no todo es Steam: nueve plataformas donde también puedes comprar juegos para PC digitales*. Xataka. https://www.xataka.com/videojuegos/porque-no-todo-steam-nueve-plataformas-donde-tambien-puedes-comprar-juegos-para-pc-digitales

- **Stacks**
  Representa una pila de objetos en la que el último en entrar es el primero en salir (LIFO). Se hace uso de las operaciones de apilar (push) y desapilar (pop), así como un método para mirar el elemento superior de la pila (top), un método para comprobar si la pila está vacía (isEmpty).

  Oracle. (2020). Stack (Java Platform SE 7). Recuperado de https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html

- **Queues**
  Representa una estructura de datos diseñada para que se inserten elementos al final de la cola y se eliminen elementos del principio de la cola. Esto es similar a cómo funciona una cola en un supermercado. La idea es ordenar los elementos de manera FIFO (First-In-First-Out), es decir, el primer elemento se elimina de primero y el último elemento se elimina al final.

  Jenkov, J. (s. f.). *Java Queue*. Jenkov.Com. Recuperado de http://tutorials.jenkov.com/java-collections/queue.html

Java Queue and PriorityQue - javatpoint. (s. f.). Recuperado de https://www.javatpoint.com/java-priorityqueue

- **Algoritmos de ordenamiento (bubble, selection and insertion sort)**

  - Bubble sort:
    Es un método caracterizado por la comparación e intercambio de pares de elementos hasta que todos los elementos estén ordenados. En cada iteración se coloca el elemento más pequeño (orden ascendente) en su lugar correcto, cambiándose además la posición de los demás elementos de la lista. La complejidad del algoritmo es $O(n^2)$

    Universidad de Cantabria. (s. f.). Algoritmos de ordenación. Recuperado de https://personales.unican.es/corcuerp/VB/Slides/Ordenacion.pdf

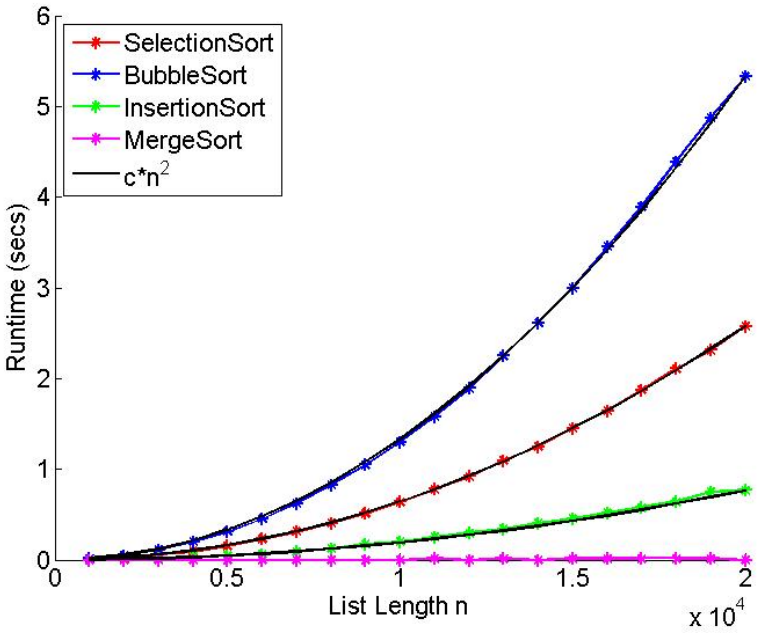
  - Selection sort:
    Mejora el ordenamiento burbuja haciendo un sólo intercambio por cada pasada a través de la lista. Para hacer esto, un ordenamiento por selección busca el valor mayor a medida que hace una pasada y, después de completar la pasada, lo pone en la ubicación correcta. Al igual que con un ordenamiento burbuja, después de la primera pasada, el ítem mayor está en la ubicación correcta. Después de la segunda pasada, el siguiente mayor está en su ubicación. La complejidad del algoritmo es $O(n^2)$

    5.8. El ordenamiento por selección — Solución de problemas con algoritmos y estructuras de datos. (s. f.). Solución de problemas con algoritmos y estructuras de datos. Recuperado de https://runestone.academy/runestone/static/pythoned/SortSearch/ElOrdenamientoPorSeleccion.html

    Universidad de Cantabria. (s. f.). Algoritmos de ordenación. Recuperado de https://personales.unican.es/corcuerp/VB/Slides/Ordenacion.pdf

  - Insertion sort:
    Es un sencillo algoritmo de ordenación que funciona de forma similar a como se ordenan las cartas en las manos. El arreglo es divido en una parte ordenada y otra sin ordenar. Los valores de la parte no ordenada se eligen y se colocan en la posición correcta en la parte ordenada. La complejidad de este algoritmo es $O(n^2)$.

    GeeksforGeeks. (2021, 8 julio). *Insertion Sort*. Recuperado de https://www.geeksforgeeks.org/insertion-sort/



http://www.cs.toronto.edu/~jepson/csc148/2007F/notes/sortTimes.jpg

- **Generics**
  Son tipos parametrizados. La idea es permitir que los tipos (Integer, String, ... etc, y los tipos definidos por el usuario) sean un parámetro para los métodos, clases e interfaces. Usando Generics, es posible crear clases que trabajen con diferentes tipos de datos. Una entidad como clase, interfaz o método que opera con un tipo parametrizado se llama entidad genérica.

  GeeksforGeeks. (2021b, julio 22). *Queue Interface In Java*. Recuperado de https://www.geeksforgeeks.org/queue-interface-java/

- **Tabla hash**
  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 almacenados a partir de una clave generada. Funciona transformando la clave con una función hash en un hash, un número que identifica la posición donde la tabla hash localiza el valor deseado.

  Gomez, E. (s. f.). *Tablas Hash - Estructuras de Datos en Java*. Estructuras de Datos en Java. Recuperado de https://sites.google.com/a/espe.edu.ec/programacion-ii/home/tablas-hash

- **Notación big O**

Es una herramienta muy funcional para determinar la complejidad de un algoritmo que estemos utilizando, permitiéndonos medir su rendimiento en cuanto a uso de espacio en disco, recursos (memoria y ciclos del reloj del CPU) y tiempo de ejecución, entre otras, ayudándonos a **identificar el peor escenario donde el algoritmo llegue a su más alto punto de exigencia**.

Fuentes, C. (2021, 22 julio). *Notación Big O: Guía para principiantes - Carlos Fuentes*. Medium. Recuperado de https://medium.com/@charlie_fuentes/notacion-big-0-para-principiantes-f9cbb4b6bec8

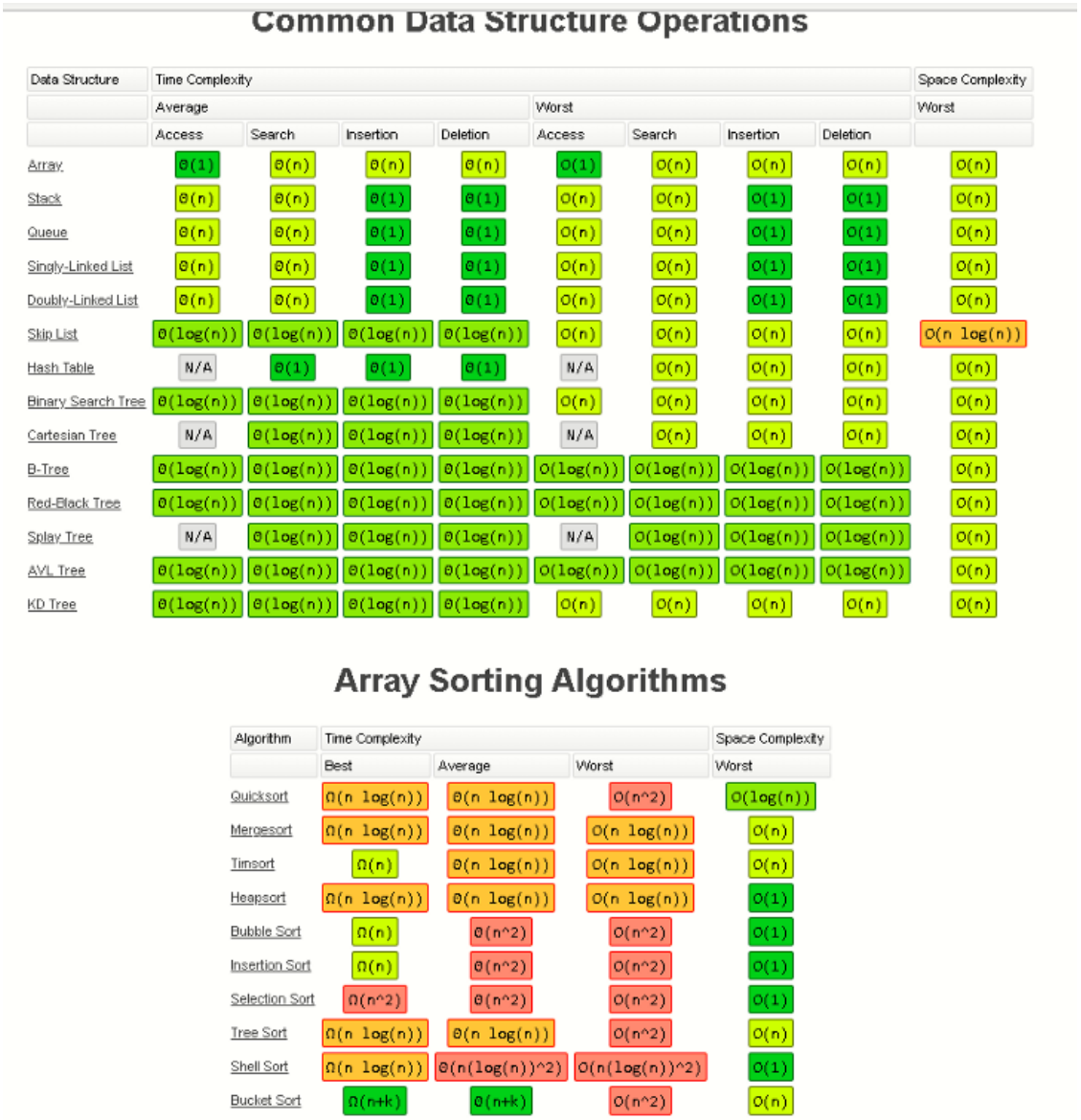
Los términos de complejidad Big O más utilizados son:

O(1)	Orden constante
O(log n)	Orden logarítmico
O(n)	Orden lineal
O(n log n)	Orden cuasi-lineal
$O(n^2)$	Orden cuadrático
$O(n^3)$	Orden cúbico
$O(n^a)$	Orden polinómico
$O(2^n)$	Orden exponencial
O(n!)	Orden factorial

https://medium.com/@joseguillermo_/qu%C3%A9-es-la-complejidad-algor%C3%ADtmica-y-con-qu%C3%A9-se-come-2638e7fd9e8c

"Assuming the graph has *n* vertices, the time complexity to build such a matrix is $O(n^2)$. The space complexity is also$O(n^2)$ **.** Given a graph, to build the adjacency matrix, we need to create a square *n x n* matrix and fill its values with 0 and 1. It costs us $O(n^2)$space".

https://www.baeldung.com/cs/adjacency-matrix-list-complexity



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)

https://miro.medium.com/max/724/1*DCdQiB6XqBJCrFRz12BwqA.png

Fase 3: Búsqueda de soluciones creativas

Técnica empleada: Lluvia de ideas

Es una técnica de pensamiento creativo utilizada para estimular la producción de un elevado número de ideas, por parte de un grupo, acerca de un problema y de sus soluciones o, en general, sobre un tema que requiere de ideas originales.

Consultores, A. (2019, 16 septiembre). *Tormenta de Ideas: Creatividad para la Mejora*. Aiteco Consultores. Recuperado de https://www.aiteco.com/tormenta-de-ideas/

Alternativa 1: Implementar Arreglo, arraylist y lista doblemente enlazada.

La lista doblemente enlazada puede usarse para simular la fila para el cajero, el arraylist para guardar los juegos que el cliente escoge (como el cesto), y el arreglo se utilizará para simular las estanterías donde se ubican los juegos.

Alternativa 2: Implementar árbol binario, matriz y lista enlazada simple.

La matriz puede usarse para simular las estanterías en donde se van a ubicar los juegos, la lista enlazada para guardar los juegos que el cliente escoge (que sirva como un cesto), y el árbol binario con el fin de representar la fila para pagar.

Alternativa 3: Implementar tablas hash, stack y queue.

Existiría una tabla hash que contenga a las estanterías en cada uno de sus slots. La key para un slot sería el identificador de una estantería (letra del abecedario) y el value sería una tabla hash en donde cada uno de sus slots guarda juegos. Para esta última tabla hash, la key para un slot sería el código del juego y el value sería el objeto juego con sus respectivos atributos.

Por otro lado, la estructura stack nos serviría para simular el cesto donde el cliente guarda los juegos que escogió de las estanterías, mientras que las queue representan la única fila formada para pasar a cada uno de los cajeros existentes.

Fase 4: Transición de las ideas a los diseños preliminares.

La alternativa 2 la descartamos, ya que el uso de una estructura como lo es el árbol binario no es adecuada, teniendo en cuenta que si lo que queremos es simular la fila de pago hacia los cajeros, al final resultaríamos formando una estructura diferente, en este caso lineal, como una lista enlazada. De igual manera, encontramos que con las matrices sería compleja la búsqueda de los juegos, puesto que tardaríamos más tiempo en encontrar la estantería a la que corresponden para así obtenerlos. Por último, con la implementación de una lista enlazada simple, gastaríamos más recursos con el fin de eliminar cierto juego del cesto de cada cliente.

Si revisamos cuidadosamente las demás alternativas obtenemos lo siguiente:

- **Alternativa 1:** Implementar Arreglo, arraylist y lista doblemente enlazada.

  Esta alternativa se puede adaptar a la situación, pero sería necesaria la implementación de métodos adicionales para adecuar el comportamiento de cada estructura mencionada. Por ejemplo, la lista doblemente enlazada se debería programar de tal forma que funcione como una fila, el arraylist se debe ajustar con el fin de que los juegos se puedan apilar en el cesto, y el arreglo se deberá acomodar de tal manera que sea posible realizar la búsqueda e inserción de los juegos en cada una de las estanterías.

- **<u>Alternativa 3:</u>** Implementar tablas hash, stack y queue.

  Esta alternativa permite simular de mejor manera el comportamiento de la situación presentada. En primer lugar, las tablas hash modelan las estanterías, ya que permiten asignar una clave y almacenar un valor, los cuales serían los identificadores (letras del abecedario) y los juegos respectivamente. Además la búsqueda de un juego sería más sencilla. Con respecto al stack, gracias a su comportamiento LIFO (last-in-first-out) podemos recrear la forma en que el cliente apila en su cesto cada uno de los juegos que escoge. En cuanto al uso de la estructura queue, encontramos que su comportamiento FIFO (first-in-first-out) es similar a como funciona una fila, y por ello es apropiada para el ejercicio.

Fase 5: Evaluación y selección de la mejor solución.
Se definen los criterios que permitirán evaluar las alternativas de solución y con base en este resultado elegir la solución que mejor satisface las necesidades del problema planteado. Los criterios que escogimos en este caso son los que enumeramos a continuación. Al lado de cada uno se ha establecido un valor numérico con el objetivo de establecer un peso que indique cuáles de los valores posibles de cada criterio tienen más peso (i.e., son más deseables).

<u>Criterios</u>

- **Criterio A: Complejidad espacial de las estructuras de datos a usar.**
  [1]$O(n^2)$
  [2]$O(n)$
  [3]$O(1)$

- **Criterio B: Adaptación de las estructuras escogidas con respecto al contexto del problema.**
  [1] No se adapta directamente
  [2] Se adapta directamente

<u>Evaluación</u>

Alternativa	Criterio A	Criterio B	Total
1. Implementar Arreglo, arraylist y lista doblemente enlazada.	[1] $O(n^2)$	[1] No se adapta directamente	2

3. Implementar tablas hash, stack y queue.	[2] $O(n)$	[2] Se adapta directamente	4

Selección
De acuerdo con la evaluación anterior se debe seleccionar la **Alternativa 3**, ya que obtuvo la mayor puntuación de acuerdo con los criterios definidos.

Fase 6: Preparación de informes y especificaciones.

Especificación del problema:

- **Problema:** Simulación del proceso de compra de los clientes de una tienda de videojuegos.

- **Entradas:** Catálogo de juegos (código del juego, cantidad de ejemplares, estantería donde está ubicado, precio del juego), cantidad de cajeros a utilizarse durante la jornada, serie de códigos o cédulas que representan a los clientes (en el orden en que entraron a la tienda) y lista de juegos por comprador (sus códigos).

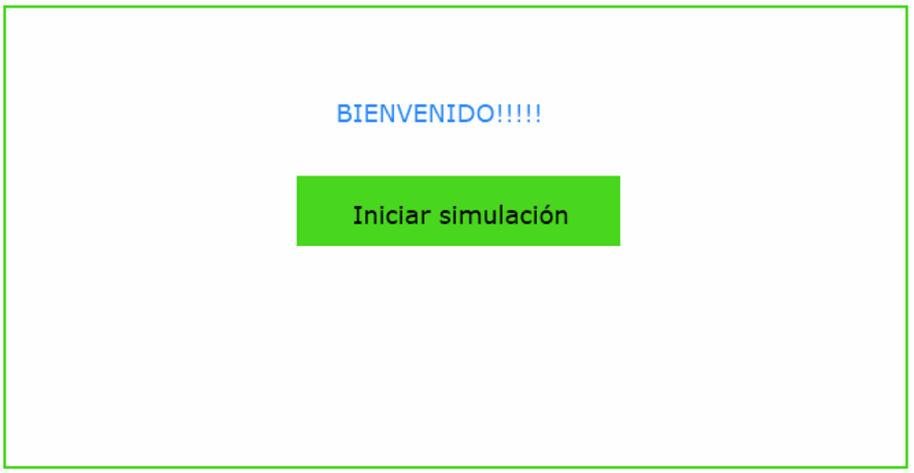
- **Salidas:** Orden de salida de los clientes, el valor de cada compra realizada y el orden en que quedaron empacados sus juegos.

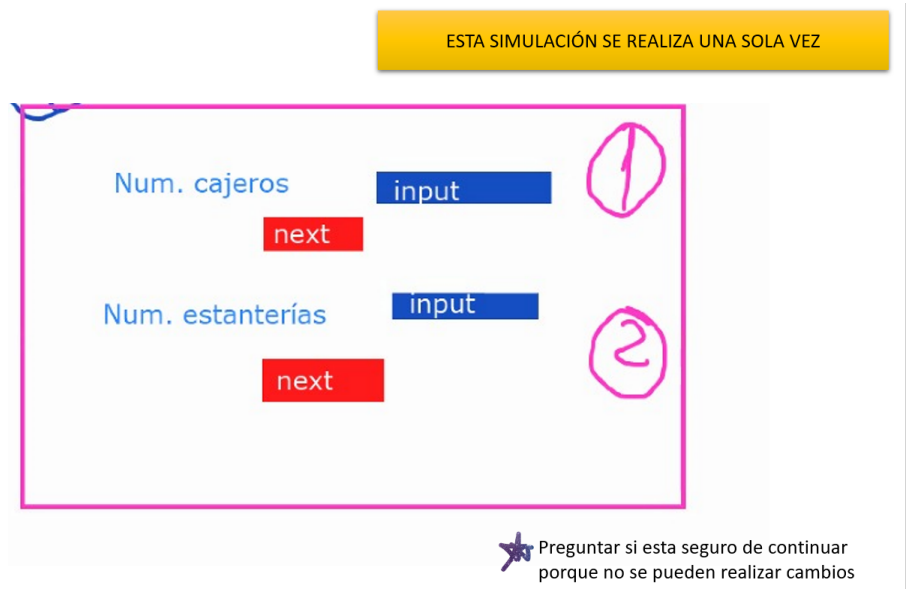
Consideraciones: Se deben tener en cuenta los siguientes casos al momento de resolver el problema:

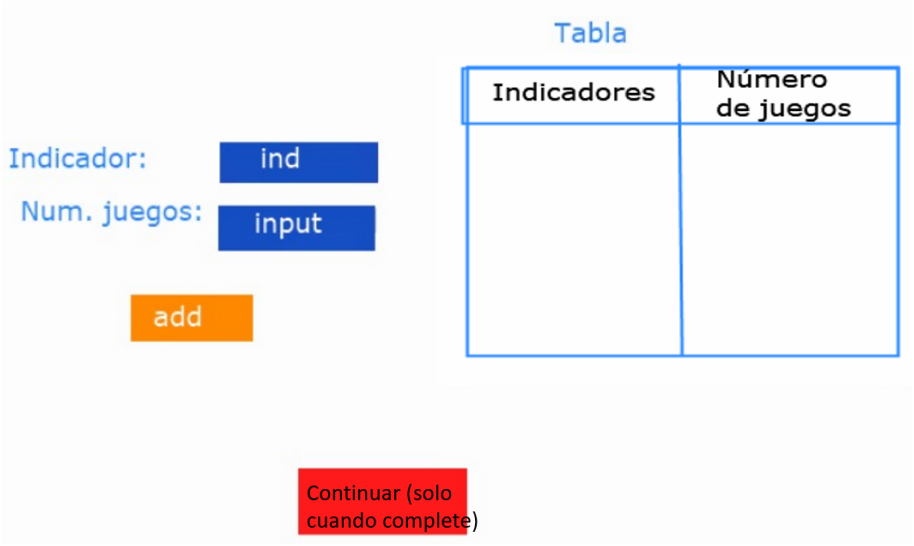
1. Si el cliente no tiene un código generado al crear una lista de juegos no puede ingresar al establecimiento.
2. Cada cliente sale de la sección 2 en el orden suministrado con una unidad de tiempo de diferencia entre ellos.
3. Si un juego que se encuentra en su lista está agotado, el código de dicho juego no aparecerá en la lista ordenada final.
4. El orden con el que un cliente ingresa a la única fila de cajas se establece teniendo en cuenta desde el tiempo en que ha entrado a la tienda, más lo que se ha tomado recogiendo los juegos de cada estantería.
5. El orden de salida de un cliente del establecimiento depende de la cantidad de juegos que va a comprar.
6. El orden en que se empacan los juegos de los clientes se da teniendo en cuenta el montón que tiene el cajero a su lado. Aquel que esté en el tope obviamente será el primero en empacarse y así sucesivamente.
7. El cliente debe elegir entre dos algoritmos de ordenamiento que se le ofrecerán en la sección 2. Se cuenta con la restricción de que ambos deben tener complejidad temporal $O(n^2)$.

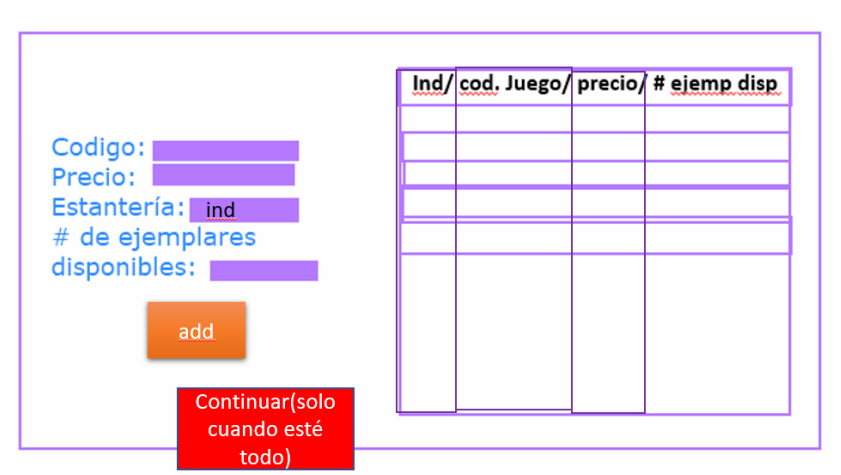
Fase 7: Implementación del diseño.

Mockups preliminares









PANTALLA DE LOS CLIENTES AL SALIR DE LA SEC. 2

Cédula	Listado de juegos	Tiempo

next

PANTALLA DE LOS CLIENTES AL SALIR DE LA SEC. 3

Cédula	Listado de juegos	Tiempo	Cesto

next

PANTALLA DE LOS CLIENTES SEC. 4

Cédula	Bolsa con juegos	Valor de compra

TABLA CON LOS CLIENTES ORDENADOS, CON SU VALOR DE COMPRA, Y SU BOLSA ORDENADA

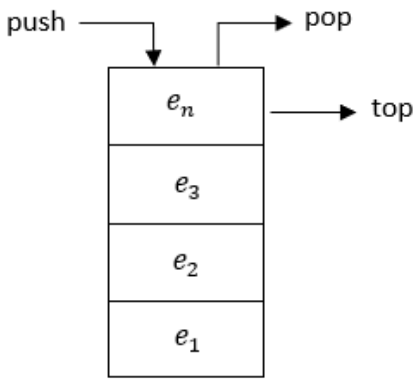
next

Fin del programa

Diseño del TAD para cada estructura de datos requerida

TAD Stack

$Stack = << e_1, e_2, e_3, ..., e_n >, top >$



$$e_i = elemento \in Stack$$

$$e_1 = el\ primer\ elemento\ añadido \wedge e_n = elemento\ top$$

$\{inv: 0 <= n \wedge size(Stack) = n \wedge top = e\}$

Operaciones primitivas:

- $Stack\ (Constructora)$: $\rightarrow$ $Stack$
- $push\ (Modificadora)$: $Stack\ x\ elemento \rightarrow Stack$
- $pop\ (Modificadora)$: $Stack \rightarrow Stack\ x\ elemento$
- $top\ (Analizadora)$: $Stack \rightarrow Elemento$
- $IsEmpty\ (Analizadora)$: $Stack \rightarrow Booleano$

Stack()
"Construye un nuevo stack"
{pre: true}
{pos: Un nuevo stack vacío}

push(stack, elemento)
"Añade un nuevo elemento al stack"
{pre: El stack debe estar vacío o lleno de elementos. Asimismo, el elemento a insertar no es nulo}
{pos: El stack ha aumentado su tamaño en 1 y contiene el elemento}

pop(stack)
"Retorna y remueve el último elemento insertado en el stack"
{pre: El stack no puede estar vacío}

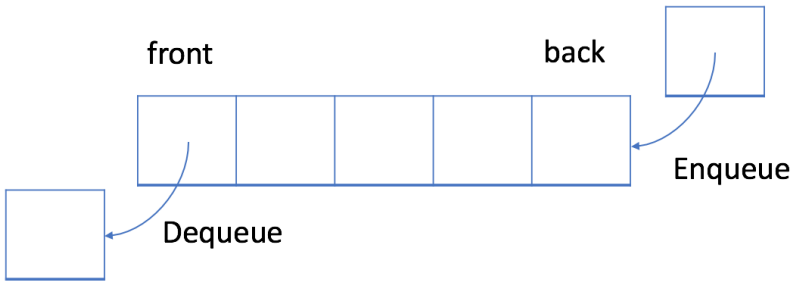
{pos: El stack ya no contiene el elemento que se había insertado últimamente y por lo tanto la estructura ha disminuido su tamaño en 1}

top(stack)
"Retorna el primer elemento del stack sin removerlo de la estructura"
{pre: El stack no puede estar vacío}
{pos: El primer elemento del stack fue obtenido}

IsEmpty(stack)
"Verifica si el stack se encuentra o no vacío"
{pre: El stack debe existir (haberse inicializado previamente)}
{pos: true si el stack está vacío y false si el stack no se encuentra vacío}

TAD Queue

$Queue = << e_1, e_2, e_3, ..., e_n >, front, back >$



$\{inv: 0 \leq n \wedge size(queue) = n \wedge front = e_1 \wedge back = e_n\}$

Operaciones primitivas:

- Queue (Constructora): → Queue
- enqueue (Modificadora): Queue x Elemento → Queue
- dequeue (Modificadora): Queue → Queue
- front(Analizadora): Queue → Elemento
- isEmpty(Analizadora): Queue → Booleano
- back(Analizadora): Queue → Elemento

Queue()
"Crea una nueva queue vacía"
{pre: true }
{post: queue vacía creada}

enqueue(queue, elemento)
"Inserta un nuevo elemento al final (back) de la queue"
{pre: la queue está inicializada, elemento es diferente de nulo }
{post: queue modificada}

dequeue(queue)
"Remueve y retorna el elemento que está en la cabeza (front) de la queue"
{pre: la queue está inicializada, la queue no está vacía}
{post: queue modificada}

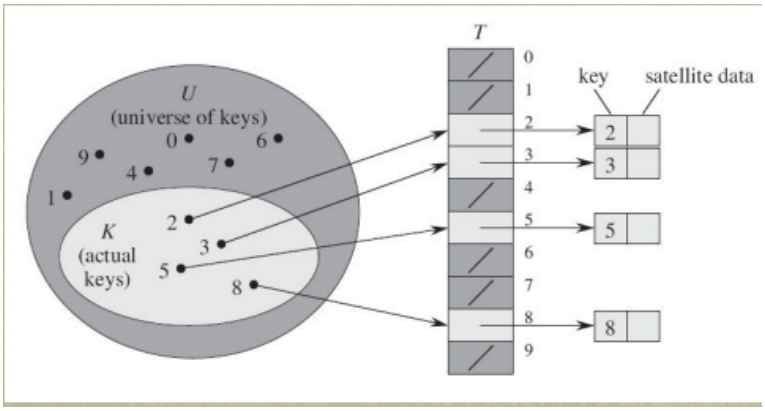
front(queue)
"Devuelve el elemento que está en la cabeza (front) de la queue"
{pre: la queue está inicializada}
{post: elemento obtenido}

isEmpty(queue)
"Devuelve un valor de verdad, dependiendo si la queue está vacía o no"
{pre: la queue está inicializada}
{post: true si la queue está vacía, false si la queue no está vacía}

back**(queue)**
"Devuelve el elemento que está al final(back) de la queue"
{pre: la queue está inicializada}
{post: elemento obtenido}

TAD Tabla Hash

$TablaHash$: $\{k_1,\ k_2 \ldots\ k_n\}\{v_1, v_2, \ldots v_n\}$



Donde k_n corresponden a las llaves y v_n son los valores asociados a dichas llaves.

$\{inv: k_1 \neq k_2 \neq ... \neq k_n\}$

Operaciones Primitivas:

- HashTable (Constructora): Tamaño → HashTable
- search (Analizadora): HashTable x key → valor
- add (Modificadora): HashTable x key x valor → HashTable
- delete(Modificadora): HashTable x key → booleano
- replace(Modificadora): Hashtable x key x valor → HashTable
- elements(Analizadora): Hashtable → Elementos

HashTable(Tamaño)
"Crea una tabla hash vacía, del tamaño dado"
{pre: TRUE}
{pos: se crea la tabla hash sin ningún elemento}

search(hashTable, key)
"Retorna el elemento que esté asociado a la llave en la tabla hash"
{pre: la tabla hash debe estar inicializada previamente}
{pos: se retorna el elemento si existe en la tabla hash, si no existe se retorna null}

add(hashTable, key, valor)
"Añade el elemento con su respectiva clave en la tabla hash"
{pre: la tabla hash debe estar inicializada previamente}
{pos: se añade exitosamente el elemento con su clave en la tabla hash}

delete(hashTable, key)

"Elimina la clave y lo que contiene de la tabla hash"
{pre: la tabla hash debe estar inicializada previamente}
{pos: si se elimina la llave su condición es verdadera, si no es así su condición es falsa}

replace(hashTable, key, valor)
"Reemplaza el valor que está asociado a la clave con un nuevo valor"
{pre: la tabla hash debe estar inicializada previamente}
{pos: hashTable modificada}

elements(hashTable)
"Retorna los elementos que se han agregado a la hashTable"
{pre: la tabla hash debe estar inicializada previamente}
{pos: elementos obtenidos}

Análisis de complejidad temporal de cada uno de los algoritmos de ordenamiento.

Insertion sort

Instrucción	**# veces que se ejecuta cada instrucción**
sortedList.addAll(gameList);	1
for(int i = 1;i<sortedList.size();i++) {	n+1
for(int j=i;j>0 && Character.compare(sortedList.get(j-1).getShelf(), sortedList.get(j).getShelf())>0;j--) {	[n(n+1)/2]+n
if(sortedList.get(j).getAmount()>0) {	n(n+1)/2
Videogame temp = sortedList.get(j);	n(n+1)/2
sortedList.set(j, sortedList.get(j-1));	n(n+1)/2

sortedList.set(j-1, temp);
}
}
}
	n(n+1)/2

$T(n) = (1) + (n+1) + [(n(n+1)/2) + n] + [n(n+1)/2] \cdot 4$
$T(n) = (5/2)n^2 + (9/2)n + 2$

$T(n) = O(n^2)$

Selection sort

Instrucción	**# veces que se ejecuta cada instrucción**
sortedList.addAll(gameList);	1
for(int i=0;i<sortedList.size();i++) {	n+1
char min = sortedList.get(i).getShelf();	n
for(int j=i+1;j<sortedList.size();j++) {	[n(n+1)/2]+n
if(Character.compare(sortedList.get(j).getShelf(), min)<0 && sortedList.get(j).getAmount()>0) {	n(n+1)/2
Videogame temp = sortedList.get(j);	n(n+1)/2
sortedList.set(j, sortedList.get(i));	n(n+1)/2

sortedList.set(i,temp);
}
}
	n(n+1)/2

sortedList.set(i,sortedList.get(i)); }	n

$T(n) =$
$1 + (n + 1) + n + [[n(n + 1)/2] + n] + [n(n + 1)/2] + [n(n + 1)/2] + [n(n + 1)/2] + [n(n$
T(n) = $1 + (n + 1) + n + [[n(n + 1)/2] + n] + [n(n + 1)/2] * 4 + n$
T(n)= $2 + 4n + [(n^2 + n)/2] + [(n^2 + n)/2] * 4$
T(n)= $2 + 4n + [(n^2 + n)/2] * 5$
T(n)= $2 + 4n + [(5n^2 + 5n)/2]$
T(n)= $(5n^2 + 13n + 4)/2$
T(n)= $O(n^2)$

Análisis de complejidad espacial de cada uno de sus algoritmos de ordenamiento.

Insertion sort

Tipo	**Variable**	**Cantidad de valores atómicos**
Auxiliar	i	1

	j temp gameList	1 1 n
Salida	sortedList	n

$S(n) = 1 + 1 + 1 + n + n = 3 + 2n$
$S(n) = O(n)$

Selection sort

Tipo	**Variable**	**Cantidad de valores atómicos**
Auxiliar	i j temp min gameList	1 1 1 1 n
Salida	sortedList	n

$S(n) = 1 + 1 + 1 + 1 + n + n = 4 + 2n$
$S(n) = O(n)$

Diseño de pruebas unitarias

Configuración de los escenarios:

Nombre	**Clase**	**Escenario**
setupScenary1	HashTable	HashTable (con key Integer y value Videogame) inicializada con tamaño 7.
setupScenary2	HashTable	HashTable (con key Integer y value Videogame) inicializada con tamaño 3. En sus slots tiene lo siguiente: ● key= 777 value=videogame1: {shelf='A', code=777, price= 12000,amount= 27} ● key= 123

		value=videogame2:{shelf='A', code=123, price= 5000, amount= 4} • key= 321 value=videogame3:{shelf='A', code=321, price= 15000, amount= 14}
setupScenary1	Stack	Un stack de tipo Videogame inicializado como Stack<Videogame> de tamaño 0. Sin ningún elemento, por lo que su nodo top va a ser nulo.
setupScenary2	Stack	Un stack de tipo Videogame inicializado como Stack<Videogame> con 1 elemento de la clase Videogame añadido. Elemento Videogame (top): code = 150 price = 2500 shelf = 'C' amount = 3
setupScenary3	Stack	Un stack de tipo Videogame inicializado como Stack<Videogame> con dos elementos de la clase Videogame añadidos. Elemento Videogame #1: code = 800 price = 2000 shelf = 'A' amount = 2 Elemento Videogame #2 (top): code = 801 price = 3000 shelf = 'B' amount =2
setupScenary1	Queue	Un queue de tipo Videogame inicializado como Queue<Client> de tamaño 0. Sin ningún elemento, por lo que sus nodos front y back van a ser nulos.
setupScenary2	Queue	Un queue de tipo Videogame inicializado como Queue<Client> con dos elementos de la clase Client añadidos. Elemento Client #1 (front): id = "52295812" sort = "INSERTION" Elemento Client #2 (back): id = "36984517" sort = "SELECTION"

setupScenary1	VideogameStore	Objeto de la clase controladora VideogameStore inicializado.

Diseño de los casos de prueba

Objetivo de la Prueba: Validar que se agrega correctamente un videojuego en la HashTable				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
HashTable	add	setupScenary1	videogame1: shelf=’A’ code=321 price= 15000 amount= 14 .	El videojuego se ha añadido a la HashTable
HashTable	add	setupScenary2	videogame4 shelf=’A’ code=322 price= 15000 amount= 3 .	El videojuego no se ha añadido a la HashTable
HashTable	add	setupScenary1	videogame1 shelf=’A’ code=321 price= 15000 amount= 14 videogame2 shelf=’A’ code=333 price= 4000 amount= 2 .	Los videojuegos se han añadido a la HashTable

Objetivo de la Prueba: Validar que se busca correctamente un videojuego en la HashTable

Clase	Método	Escenario	Valores de Entrada	Resultado
HashTable	search	setupScenary2	key= 777 .	El videojuego se ha encontrado en la HashTable
HashTable	search	setupScenary2	.key= 888	El videojuego no se ha encontrado en la HashTable
HashTable	search	setupScenary2	key= 321	El videojuego se ha encontrado en la HashTable

Objetivo de la Prueba: Validar que se elimina correctamente un videojuego en la HashTable				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
HashTable	delete	setupScenary2	key= 777 .	El videojuego se ha eliminado de la HashTable
HashTable	delete	setupScenary2	key= 888	El videojuego no se ha eliminado de la HashTable
HashTable	delete	setupScenary2	key= 321	El videojuego se ha eliminado de la HashTable

Objetivo de la Prueba: Validar que se reemplaza correctamente un videojuego en la HashTable				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
HashTable	replace	setupScenary2	key= 777 value=videogame4: {shelf='A', code=777, price= 77000,amount= 8} .	El videojuego se ha reemplazado en la HashTable, ahora la key 777 está asociada con el videojuego con atributos shelf='A', code=777, price= 77000,amount= 8
HashTable	replace	setupScenary2	key= 888 value=videogame4: {shelf='A', code=777, price= 77000,amount= 8}	El videojuego no se ha reemplazado en la HashTable, ya que no hay un juego asociado a la key dada

HashTable	replace	setupScenary2	key= 321 value=videogame4: {shelf=’A’, code=321, price= 77000,amount= 8}	El videojuego se ha reemplazado en la HashTable, ahora la key 321 está asociada con el videojuego con atributos shelf=’A’, code=321, price= 77000,amount= 8

Objetivo de la Prueba: Validar que se retorna correctamente los elementos videojuego de la HashTable				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
HashTable	elements	setupScenary1	.	Se devuelve una lista vacía de videojuegos
HashTable	elements	setupScenary2		Se devuelve una lista de videojuegos, de tamaño 3

Objetivo de la Prueba: Validar que se agrega correctamente un videojuego a un stack, específicamente como elemento top de la estructura.				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Stack	push	setupScenary1	videogame1: shelf= ‘A’ code=321 price= 15000 amount= 14	El videojuego se ha añadido al stack. Ahora el tamaño de la estructura es 1 y videogame1 es el top.
Stack	push	setupScenary2	videogame2: shelf= ‘B’ code=620 price= 4000 amount= 2	El videojuego se ha añadido al stack. Ahora el tamaño de la estructura es 2 y videogame2 es el elemento top.
Stack	push	setupScenary3	videogame3: shelf= ‘A’ code=322	El videojuego se ha añadido al stack. Ahora el tamaño de la estructura es 3 y videogame3 es el elemento top.

			price= 15000 amount= 3	

Objetivo de la Prueba: Validar que se retorna y remueve correctamente el último elemento insertado en un stack (el elemento top).				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Stack	pop	setupScenary1		Se retorna un elemento nulo, ya que el stack no tiene ningún elemento añadido a su estructura. Asimismo, su tamaño sigue siendo 0.
Stack	pop	setupScenary2		Se retorna y remueve el elemento Videogame {code = 150, price = 2500, shelf = 'C', amount = 3}. Ahora el elemento top del stack creado es nulo y el tamaño de la estructura es 0.
Stack	pop	setupScenary3		Se retorna y remueve el elemento Videogame {code = 801, price = 3000, shelf = 'B', amount = 2}. Ahora el elemento top del stack creado es el objeto Videogame {code = 800, price = 2000, shelf = 'A', amount = 2} y el tamaño de la estructura es 1.

Objetivo de la Prueba: Validar que se retorna correctamente el primer elemento del stack sin removerlo del stack.				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Stack	top	setupScenary1	.	Se retorna un elemento nulo, ya que el stack no tiene ningún elemento añadido a su estructura. Asimismo, su tamaño sigue siendo 0.
Stack	top	setupScenary2		Se retorna el elemento Videogame {code = 150, price = 2500, shelf = 'C', amount = 3}. Asimismo, el tamaño de la estructura continúa siendo 1.

Stack	top	setupScenary3		Se retorna el elemento Videogame {code = 801, price = 3000, shelf = ‘B’, amount = 2}. Asimismo, el tamaño de la estructura continúa siendo 2.

Objetivo de la Prueba: Validar que se verifica correctamente el estado del stack (si se encuentra vacío o no).				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Stack	isEmpty	setupScenary1		True. El stack no contiene ningún elemento y su tamaño es 0.
Stack	isEmpty	setupScenary3		False El stack tiene 2 elementos, por lo que no se encuentra vacío.

Objetivo de la Prueba: Validar que se agrega correctamente un elemento a la estructura Queue.				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Queue	enqueue	setupScenary1	client1: id = “25967841” sort = “INSERTION”	El cliente se ha añadido al queue. Ahora el tamaño de la estructura es 1 y client1 es el elemento front y back de la estructura.
Queue	enqueue	setupScenary2	client3: id = “4596382” sort = “SELECTION”	El cliente se ha añadido al queue. Ahora el tamaño de la estructura es 3 y client3 es el elemento back de la estructura. El elemento front sería el objeto Client con atributos: {id = “52295812”, sort = “INSERTION”}

Objetivo de la Prueba: Validar que se retorna y remueve correctamente el elemento que está en la cabeza (front) de la Queue.				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**

Queue	dequeue	setupScenary1		Se retorna un elemento nulo, ya que el queue no tiene ningún elemento añadido a su estructura. Asimismo, su tamaño sigue siendo 0.
Queue	dequeue	setupScenary2		Se retorna y remueve el elemento Client {id = “52295812”, sort = “INSERTION”}. Ahora el elemento tanto front como back del queue creado es Client{id = “36984517”, sort = “SELECTION”} y el tamaño de la estructura es 1.

Objetivo de la Prueba: Validar que se retorna correctamente el primer elemento del stack sin removerlo del stack.				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Queue	front	setupScenary1	.	Se retorna un elemento nulo, ya que el queue no tiene ningún elemento añadido a su estructura. Asimismo, su tamaño sigue siendo 0.
Queue	front	setupScenary2		Se retorna el elemento Client {id = “52295812”, sort = “INSERTION”}. Asimismo, el tamaño de la estructura continúa siendo 2.

Objetivo de la Prueba: Validar que se retorna correctamente el primer elemento del stack sin removerlo del stack.				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Queue	back	setupScenary1		Se retorna un elemento nulo, ya que el queue no tiene ningún elemento añadido a su estructura. Asimismo, su tamaño sigue siendo 0.

Queue	back	setupScenary2		Se retorna el elemento Client {id = “36984517”, sort = “SELECTION”}. Asimismo, el tamaño de la estructura continúa siendo 2.

Objetivo de la Prueba: Validar que se verifica correctamente el estado del stack (si se encuentra vacío o no).				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Queue	isEmpty	setupScenary1		True. El queue no contiene ningún elemento y su tamaño es 0.
Queue	isEmpty	setupScenary2		False El queue tiene 2 elementos, por lo que no se encuentra vacío.

Objetivo de la Prueba: Validar que el resultado final de la aplicación sea correcto. Con el orden de salida de los clientes, el valor de cada compra y el orden en que quedaron empacados sus juegos.				
Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
VideogameStore		setupScenary 1	Cantidad de cajeros disponibles= 3 Cantidad de estanterías= 3 Número de juegos para la estantería A= 4 Número de juegos para la estantería B= 5 Número de juegos para la estantería C= 2 Juego para la estantería A con código 331,	El resultado final de la aplicación corresponde al esperado, que es el siguiente, en orden: ● CLIENTE:3219 Valor de compra:65000 Juegos: 287 ● CLIENTE:1627 Valor de compra: 145000 Juegos: 612, 287 ● CLIENTE:2100 Valor de compra: 17000 Juegos: 331 ● CLIENTE:3456 Valor de compra: 248000 Juegos:612, 465, 333, 287

			precio 17000, y cantidad de ejemplares 3 Juego para la estantería A con código 465, precio 60000, y cantidad de ejemplares 6 Juego para la estantería A con código 612, precio 80000, y cantidad de ejemplares 2 Juego para la estantería A con código 971, precio 70000, y cantidad de ejemplares 6 Juego para la estantería B con código 441, precio 30000, y cantidad de ejemplares 3 Juego para la estantería B con código 112, precio 22000, y cantidad de ejemplares 6 Juego para la estantería B con código 229, precio 28000, y cantidad de ejemplares 6 Juego para la estantería B con código 281, precio 38000, y cantidad de ejemplares 2 Juego para la estantería B con código 333, precio 43000, y cantidad de ejemplares 6 Juego para la estantería C con código 767, precio 40000, y cantidad de ejemplares 2	• CLIENTE:3311 Valor de compra: 203000 Juegos: 971, 229, 767, 287

			Juego para la estantería C con código 287, precio 65000, y cantidad de ejemplares 6 Cliente con cédula 1627, con juegos a comprar 287, 612 Cliente con cédula 3456, con juegos a comprar 612, 333 287, 465 Cliente con cédula 3219, con juegos a comprar 287 Cliente con cédula 3311, con juegos a comprar 767, 287, 229, 971 Cliente con cédula 2100, con juegos a comprar 331	

Diagrama de clases