TAREA INTEGRADORA 1: TIENDA DE VIDEOJUEGOS

JULIAN CAMILO BOLAÑOS

ARIEL EDUARDO PABÓN BOLAÑOS

JUAN FELIPE BLANCO TIGREROS

19 DE SEPTIEMBRE DE 2021

ALGORITMOS Y ESTRUCTURAS DE DATOS

UNIVERSIDAD ICESI

1. INFORME DEL MÉTODO DE LA INGENIERÍA

FASE 1: Identificación del problema.

Necesidades identificadas:

- Se necesita esclarecer el proceso de compra en la tienda de videojuegos. Esto implica que el cliente entienda todas las secciones en las que la tienda divide el proceso, partiendo desde que el cliente hace la búsqueda de sus juegos hasta la compra de estos con el cajero. Por tanto, se quiere encontrar una solución que muestre:
    A. La lista con los juegos disponibles y ordenados de acuerdo con la posición más cercana de estanterías tras haber hecho la búsqueda en la sección 2 (agilización de recolección de ejemplares).
    B. Cómo se apilan los juegos cada vez que el cliente los recoge en la sección 3.
    C. Cómo se va definir el turno de pago y la fila para cada cliente de acuerdo con el tiempo en que este llegue a la tienda, y con el tiempo que se tarda en recoger sus juegos (sección 4).
    D. Cómo se da el órden de empaque y cuál es el valor final de la compra.

Definición del problema:

- La solución puede ser un programa que simule el proceso de compra, permitiéndole al usuario observar los distintos comportamientos que se dan con diferentes condiciones.

FASE 2: Recopilación de información necesaria.

Para implementar la solución planteada, es necesario definir conceptos como la notación asintótica y las estructuras de datos de pilas, colas y tablas hash. La notación asintótica es el análisis matemático requerido para implementar una solución eficiente, correspondiendo a las definiciones de $O(n)$, $\Omega(n)$ y $\theta(n)$. Para modelar el funcionamiento de la tienda resulta útil referenciar la teoría de colas, ya que puede evidenciar la eficiencia del proceso manejado por la tienda. Por último, se asocian estos conceptos con las definiciones formales de las otras estructuras de datos mencionadas.

Notación asintótica:

Las notaciones que utilizamos para describir el tiempo de ejecución asintótico de un algoritmo se definen en términos de funciones cuyos dominios son el conjunto de números naturales N = {0, 1, 2, ...}. Estas notaciones son convenientes para describir la función de tiempo de ejecución en el peor caso T(n), que suele definirse sólo para tamaños de entrada enteros.

Θ(g(n)) = {f(n) : existen constantes positivas **c1**, **c2**, y **n0** tales que 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) para todo n ≥ n0}.

O(g(n)) = {f(n): existen constantes positivas **c** y **n0** tales que 0 ≤ f(n) ≤ cg(n) para todo n ≥ n0}.

Ω(g(n)) = {f(n): existen constantes positivas **c** y **n0** tales que 0 ≤ cg(n) ≤ f(n) para todo n ≥ n0}.

(Tomado de: Introduction to Algorithms, Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein).

Pilas:

Una pila es una estructura de datos lineal, cuya principal característica es la forma en la que se añaden y toman elementos del mismo, que se hace siguiendo la norma de "FILO" que hace referencia a "First in, last out"; puede pensarse como una baraja de cartas. Sus métodos más importantes son: *push*, *pop* e *isEmpty*. El *push* se encarga de añadir un elemento al tope de la pila mientras que el *pop* se encarga de removerlo. Por otro lado, el *isEmpty* se encarga de verificar que la pila esté vacía. Todos los métodos previamente mencionados poseen una complejidad de O(1).

Para poder implementar una pila se utiliza un arreglo con *n* posiciones y una referencia al índice del tope en el arreglo. Al hacer un push, se aumenta el tope en 1 y se añade el

elemento en esa posición. En caso de querer realizar un pop, se obtendrá el elemento en top y posteriormente se disminuirá este valor en 1 (no es necesario eliminar nada ya que al realizar push nuevamente se sobre escribirá los valores por encima del top actual). Finalmente, para conocer si la pila está vacía, se verifica que el índice del tope no esté por debajo del mínimo. Cabe resaltar que se deben hacer las validaciones necesarias en cada caso, de lo contrario se lanzan excepciones de pila llena o pila vacía para el push y pop respectivamente.

(Tomado de: GeekForGeeks & ThoughtCo)

Colas:

Una cola es una estructura de datos lineal, en la cual se conocen únicamente el primer y el último elemento. Estas se denominan como estructuras de orden FIFO (First In First Out), debido a la manera de añadir y eliminar elementos que tienen. Cada vez que se desee añadir un elemento a una cola, se referenciará el nuevo elemento como el último elemento de esta -es decir, el tope-. Para eliminar, siempre se removerá el primer elemento, haciendo que la referencia a este pase a ser el siguiente elemento que haya. Una cola eficientemente implementada en su método de inserción y eliminación tendrá una complejidad de O(1) (Tomado de: GeeksforGeeks).

Véase que existen diferentes maneras de implementar una cola (genérica), ya sea dinámicamente por medio de listas enlazadas o estáticamente por medio de arreglos. Igualmente, existen distintos tipos de estas, como las colas de prioridad, colas doblemente terminadas y colas circulares. El uso de cada una de estas estructuras dependerá de las necesidades del problema (Tomado de: Baeldung).

Tablas hash:

Una tabla hash es una estructura de datos que almacena pares ordenados en forma de una llave y un valor. Todas cuentan con dos elementos: 1) Una función hash y 2) Un arreglo para almacenar los datos.

1) Función hash: Determina el índice en el que se agrega el par con la llave y el valor, tomando el primero como parámetro y realizando una serie de operaciones aritméticas para obtener el resultado. La selección de una función hash adecuada es crucial para la elaboración de una buena tabla. Esta se puede hacer más efectiva teniendo una noción de las posibles llaves a utilizar. Comúnmente se denotan las tablas hash con el término h.

2) Arreglo: El arreglo asociado a una tabla hash es el espacio en el cual se almacenarán los valores deseados. El tamaño del arreglo deberá ser escogido acorde al número de datos que se quieran almacenar y a la función hash que se implemente.

En las tablas hash se pueden presentar colisiones. Estas ocurren cuando dos llaves k_1 y k_2 tales que $k_1 \neq k_2$ tiene un mismo resultado en la función hash $h(k_1) = h(k_2)$. Esto implica que los valores a almacenar en la posición $h(k_1)$ y $h(k_2)$ se verán modificados incorrectamente cada vez que se manipulen las llaves k_1 y k_2. Para solucionar esto existen distintos métodos: la técnica de exploración lineal, la de encadenamiento, la de redimensionamiento, entre otras.

- *Exploración lineal*: Si la posición $h(k)$ ya se encuentra ocupada, se busca linealmente el siguiente espacio vacío en la tabla.

- *Encadenamiento*: La tabla hash se define como un arreglo de listas enlazadas. Todos los objetos que coincidan en un mismo espacio serán añadidos a la lista enlazada de este. Esta última puede variar, pero es útil implementar una lista doblemente enlazada para conseguir una mayor eficiencia con respecto a algunos procesos.

- Redimensionamiento: El tamaño de la tabla hash puede incrementar para distribuir sus elementos de forma más eficiente -más espaciados y con menos colisiones-. Para implementar esta solución se requiere de un umbral, dicho umbral indicará el porcentaje que deberá ser ocupado de la tabla antes de redimensionar.

Las operaciones más comunes de las tablas de hash son la búsqueda de un valor de acuerdo a una llave, la inserción de nuevo valor a la tabla y la eliminación. Todos estos procesos en su caso promedio tienen una complejidad de $O(1)$ (Tomado de: Educative).

FASE 3: Búsquedas de soluciones creativas.

Para esto pensamos en los problemas dados y utilizamos la técnica de lluvia de ideas para formular las posibles soluciones. En este proceso definimos la consulta y el ordenamiento de la lista de cliente junto con la fila y el turno del cliente como los problemas a solucionar.

Consulta de juegos de la lista:

Alternativa 1:

Tomar la entrada dada por el usuario y recorrer una lista de juegos para buscarlos y agregarlos a la lista del cliente. Para esto los juegos deben contener el código, el cual se usa para verificar cada código dado hasta dar con el juego correcto. Recorrer esta lista tiene complejidad $O(n)$ en el peor de los casos.

Alternativa 2:

Tomar la entrada dada por el usuario y recorrer una lista de juegos ordenada por códigos para buscarlos y agregarlos a la lista del cliente. En este caso se utiliza búsqueda binaria para optimizar el proceso. Este algoritmo tiene una complejidad $O(n\ log(n))$.

Alternativa 3:

Tomar la entrada dada por el usuario y utilizar una tabla hash para buscarlos y agregarlos a la lista del cliente. Las llaves de la tabla hash serían los códigos de los juegos y los valores serían las instancias de estos. Este procedimiento tiene una complejidad $O(1)$.

Alternativa 4:

Tomar la entrada dada por el usuario y recorrer las estanterías de forma lineal para buscar los juegos y agregarlos a la lista del cliente. Este procedimiento tiene una complejidad $O(n^2)$ dado que implica recorrer un listado de listas.

Ordenamiento de lista (agilización de la recolección):

Alternativa 1:

Recorrer el catálogo de juegos de la tienda en busca de los juegos del cliente. El catálogo se representaría como una tabla hash que contiene estos últimos. Se tomarían los códigos de cada juego como las llaves y los valores como instancia de cada uno. Cada juego tiene una referencia que indica en qué estantería se encuentra. Al final del proceso se obtiene un listado con los juegos existentes y ordenados acorde con la ubicación -órden- de las estanterías. Para esto se utiliza un algoritmo de ordenamiento de complejidad $O(n^2)$.

Alternativa 2:

Recorrer las estanterías en busca de los juegos del cliente. La estantería se representaría como una lista que contiene estos últimos. Al final del proceso se obtiene un listado con los juegos existentes y ordenados acorde con la ubicación -órden- de las estanterías. Para esto se utiliza un algoritmo de ordenamiento de complejidad $O(n^2)$.

Determinar la fila y el turno de pago para cada cliente:

Alternativa 1:

Utilizar una lista en la que se posicionan los clientes de acuerdo al tiempo calculado por la tienda, recorrerla y añadir estos últimos a los cajeros disponibles para el proceso de pago.

Alternativa 2:

Utilizar una cola en la que se posicionan los clientes de acuerdo al tiempo calculado por la tienda. Luego, se van dejando pasar los clientes de tal forma que los primeros en la fila sean los primeros en pasar a los cajeros con el fin de realizar el proceso de pago.

FASE 4: Transición de la formulación de ideas a diseños preliminares.

Para esta fase, se busca descartar las ideas que no nos resultan factibles para la solución del problema. Con esto en mente, descartamos:

-La *Alternativa 2* para la **consulta de la lista**, ya que se desconoce el orden inicial de los juegos de cada cliente, por lo cual hacer una búsqueda binaria sería erróneo.

La revisión cuidadosa de las otras alternativas nos conduce a lo siguiente:

Consulta de juegos de la lista:

Alternativa 1. Búsqueda lineal

- Dado un listado de tamaño n de juegos, se tiene que recorrer toda la lista para encontrar el juego deseado. Lo anterior implica que se tiene que buscar desde el elemento 1 hasta el elemento n.
- Los juegos se buscan por medio de su respectivo código.
- Se agrega el juego en caso que se encuentre en el listado con la posición *i*. En caso contrario, no se añadirá nada a la lista de juegos una vez verificado el n-ésimo juego.

Alternativa 3. Búsqueda con hashmaps

- Para encontrar un juego en una tabla hash se necesita usar una función que retorna la posición del elemento buscado dado una key. En este caso, el juego se busca por medio de su respectivo código.
- Se agrega el juego al listado en caso que esté en la tabla hash. En caso contrario, se recorre lo que queda de esta última sin agregar nada.

Alternativa 4. Búsqueda lineal por estanterías

- En este caso, se recurre a un listado de estanterías que se toman como las listas de juegos de la tienda.
- Por cada estantería se buscan linealmente los juegos. Esto implica que en el peor de los casos se hace un recorrido desde el primer juego de la primera estantería hasta el último de la última estantería.
- Se agrega el juego en caso que se encuentre en la estantería *i* en la posición *j* del listado de dicha estantería. En caso contrario, no se añadirá nada a la lista de juegos una vez verificado el m-ésimo juego de la n-ésima estantería.

Ordenamiento de la lista:

Alternativa 1. Ordenar - Catálogo de juegos

- En este caso se toma el catálogo de juegos como una tabla hash con los códigos de los juegos como llaves son los códigos de los juegos y el valor asociado la instancia de cada uno.
- Cada juego tiene una referencia (nombre) de la estantería en la que se encuentra. Se obtiene dicha referencia y se genera una lista de referencias.
- Para ordenar la lista de referencia a estanterías (Nombres de estanterías) se utiliza un ordenamiento de burbuja. Simultáneamente se ordena en base a las referencias el listado que se le entregará al cliente.
- Dado que para la búsqueda del nombre estantería se usa una tabla hash, dicho proceso tiene una complejidad temporal de $O(1)$. Por su parte, el ordenamiento de burbuja tiene una complejidad temporal de $O(n^2)$. Con estos dos factores, se tiene que todo el procedimiento tiene una complejidad de $O(n^2 \cdot 1) = O(n^2)$.

Alternativa 2. Ordenar - Estanterías por listas

- En este caso se toman las estanterías como listas de juegos y se recorre siempre esta para obtener el nombre de la estantería en la que está cada juego.
- Una vez encontrado el juego, se añade el nombre de la estantería a un listado de nombres de estanterías y a medida que este listado de nombre se ordena, simultáneamente se ordena el listado final del cliente.
- Para ordenar las listas se utiliza un ordenamiento de burbuja.
- Dado que el proceso de búsqueda de nombres implica una búsqueda lineal, dicho proceso tiene una complejidad temporal de $O(n)$. Por su parte, el ordenamiento de burbuja tiene una complejidad de $O(n^2)$. En conclusión, esta alternativa tiene una complejidad de $O(n^2 \cdot n) = O(n^3)$.

Determinar la fila y el turno de pago del cliente:

Alternativa 1. Lista

- Se utiliza una lista para simular la fila de clientes y se ordena de acuerdo al tiempo dado por la tienda. Este tiempo se ve influido por el órden de llegada a la tienda y la cantidad de juegos recogidos.

- Cada vez que un cliente sale, se elimina el primer elemento de la fila y se mueve el resto hacia la izquierda.

Alternativa 2. Cola

- Se utiliza una implementación de colas para simular la fila de clientes y se ordena de acuerdo al tiempo dado por la tienda. Este tiempo se ve influido por el órden de llegada a la tienda y la cantidad de juegos recogidos.
- Cada vez que un cliente sale se elimina de la fila y el próximo queda como el primero en esta.

FASE 5: Evaluación y selección de la mejor solución.

Para corroborar lo dicho en la fase anterior se van a establecer unos criterios con el fin de encontrar la solución más adecuada para el problema descrito.

Criterios evaluativos:

Criterio A. Eficiencia. Se prefiere una solución con mejor eficiencia que las otras consideradas. Esta puede ser:

1. O(1)
2. O(n logn)
3. O(n)
4. O(n^2)

Criterio B. Semejanza. Se prefiere una solución que se asemeje más al proceso que se está realizando en la tienda. Esta puede ser:

1. Semejante
2. No semejante

Evaluación:

Consulta de la lista:

	Criterio A	**Criterio B**

Alternativa 1	O(n)	Semejante
Alternativa 3	O(1)	Semejante
Alternativa 4	O(n^2)	No semejante

Ordenamiento de la lista:

Alternativa 1	O(n^2)	Semejante
Alternativa 2	O(n^3)	No semejante

Determinar fila y turno de pago del cliente:

Alternativa 1	O(n)	No semejante
Alternativa 2	O(n)	Semejante

2. REQUERIMIENTOS FUNCIONALES

RF 1. Simular el proceso de compra de la tienda.

- **RF 1.1. Simular la sección 2 del proceso de compra.**
  - **RF 1.1.1 Ordenar lista del cliente.** El programa debe ordenar la lista acorde con la ubicación -órden- de las estanterías de la tienda. En el caso de que el elemento no exista, no se incluye en la lista. Para realizar esto, por cada cliente se recibe una listado de códigos y se entrega como salida el listado de códigos ordenados según la estantería (De menor a mayor A→Z).
- **RF 1.2. Simular la sección 3 del proceso de compra.**
  - **RF 1.2.1 Simular el cesto del cliente.** El programa debe apilar los juegos que va recogiendo el cliente en un cesto, de tal forma que los primeros juegos queden al fondo y los últimos al tope. Se requiere uso de pilas -o stacks- para implementar este requerimiento. A su vez, como entrada se pasa la cola de clientes de la tienda, en la cual, cada cliente tiene su lista de juegos ordenada acorde a las estanterías de cada juego. Esta simulación entrega un listado de clientes (Con el

mismo orden de la cola inicial), pero en el cual se ha incrementado el tiempo que lleva cada cliente en la tienda una unidad de tiempo por cada juego recogido. A su vez, los clientes de dicha lista también cuentan con la pila de juegos recogidos en el orden de su respectiva lista.

- **RF 1.3. Simular la sección 4 del proceso de compra.**
  - **RF 1.3.1 Ordenar la fila de clientes.** El programa debe ordenar la fila de clientes acorde al tiempo que pasaron en la tienda y a su orden de llegada. Para este proceso se requiere de un listado de clientes (listado generado en la sección 3 - RF 1.2.1) y se entrega como salida la cola de clientes ordenada acorde al tiempo que han pasado en la tienda y su orden de ingreso.
  - **RF 1.3.2. Simular la fila de ingreso a las cajas registradoras.** El programa debe ubicar a los clientes dentro de una fila acorde con el tiempo (RF 1.4.). Se requiere el uso de colas -o queues- para implementar este requerimiento. Como entrada se toma la cola de clientes ordenada acorde al tiempo de cada cliente en la tienda (RF 1.3.1). Se entregan como salida dos listados, uno con el listado de clientes en el orden que salían estos de los cajeros y el segundo listado tiene el precio pagado por cada cliente. El cliente *i* del listado de clientes pagó el valor en x en la posición *i* del listado de precios pagados.
- **RF 1.4. Calcular el tiempo que se toma cada cliente.** El programa debe calcular un tiempo para cada cliente, el cual es determinado por la cantidad de juegos que recoja y por el órden de llegada a la tienda. Para esto se requiere como entrada un cliente con un tiempo inicial y se retorna el cliente con el nuevo tiempo.

RF 2. Permitirle al usuario ingresar distintos valores relacionados con la tienda. El programa debe permitir al usuario ingresar el catálogo de juegos -código del juego, cantidad de ejemplares, estantería donde se ubica y precio-, cantidad de cajeros a utilizarse

durante la jornada, serie de cédulas de los clientes -en el órden que entran a la tienda-, y la lista de juegos por comprador -dado con los códigos-. Se recibe como entrada la ruta de un archivo .txt con el formato deseado y se retorna la tienda con los valores indicados en el archivo.

RF 3. Mostrar al usuario el resultado de la simulación. El programa debe informar sobre el órden de salida de los clientes, el valor de cada compra y el órden en que quedaron empacados los juegos. Se recibe como entrada la tienda configurada con los valores deseados (RF 2) y se retorna un archivo de texto en el cual se indica por cada cliente el valor pagado y el listado de juegos comprados. Igualmente, se entrega un archivo de texto en el cual se evidencia el avance (id, juegos y tiempo en la tienda) de los clientes en las secciones 2 y 3 (No se incluye la 4 debido a que la salida mencionada antes hace referencia al estado final tras esta sección).

3. REQUERIMIENTOS NO FUNCIONALES

RNF 1. Implementar estructuras de datos genéricas.

- **RNF 1.1. Implementar pilas -stacks- genéricas.**
- **RNF 1.2. Implementar colas -queues- genéricas.**
- **RNF 1.3. Implementar tablas hash genéricas.**

RNF 2. Implementar algoritmos de ordenamiento de $O(n^2)$**.**

RNF 3. El programa debe interactuar con el usuario a través de una interfaz gráfica.

4. DISEÑOS TAD

TAD Stack
Stack = {top = <*t*>}
{inv: t ∈ListNode ∨ t = null}

Operaciones Primitivas:

-	createStack		→ Stack
-	top	Stack	→ ListNode
-	push	Stack x Object	→ Stack
-	pop	Stack	→ Stack
-	isEmpty	Stack	→ Boolean

createStack()

Crea una pila vacía de ListNode.

{pre: True}
{post: Stack = top = null}

top(stack)

Retorna el elemento *t* que se encuentra en la parte superior de la pila.

{pre: stack ≠ null ∧ {t ∈ListNode ∨ t = null} }
{post: <t>}

push(stack, e)

Ingresa un nuevo elemento e al Stack en la posición top.

{pre: stack = top = <t> ∧ {t ∈ListNode ∨ t = null} ∧ e ≠ null }
{post: stack = top = <e>}

pop(stack)

Remueve el elemento top *t* que se encuentra en la parte superior de la pila.

{pre: stack = top = <t> ∧ t ≠ null }
{post: stack = top <b>, {b ∈ListNode ∨ b = null} }

isEmpty(stack)

Retorna verdadero si el top de la pila es igual a null (Pila vacía).

{pre: stack = top = $<t>$ ∧ {t ∈ListNode ∨ t = null} }
{post: True o False}

TAD Queue		
Queue= {front = $<f>$, rear = $<r>$}		
{inv: (f ∈ListNode ∧ r ∈ ListNode) ∨ (f = null ∧ r = null) }		
Operaciones Primitivas:		
- createQueue		→ Queue
- front	Queue	→ ListNode
- rear	Queue	→ ListNode
- enqueue	Queue x Object	→ Queue
- dequeue	Queue	→ Queue
- isEmpty	Queue	→ Boolean

createQueue()

Crea una cola de ListNode vacía.

{pre: True}
{post: queue = front = null, rear = null}

front(queue)

Retorna el elemento f que se encuentra en el frente de la cola.

{pre: queue ≠ null ∧ { f, r ∈ ListNode ∨ (f = null ∧ r = null) } }
{post: $<f>$}

rear(queue)

Retorna el elemento r que se encuentra al final de la cola.

{pre: queue ≠ null ∧ { f, r ∈ ListNode ∨ (f = null ∧ r = null) } }
{post: $<r>$}

enqueue(queue, e)

Añade al final de la cola $queue$ el elemento e.

{pre: queue ≠ null ∧ { f, r ∈ ListNode ∨ (f = null ∧ r = null) } }
{post: queue = front = <f>, rear = <e>}

dequeue(queue)

Remueve el primer elemento f de la cola. El nuevo front de la cola, en caso de que tenga, será el segundo elemento b de esta, sino, la cola quedará vacía.

{pre: queue ≠ null ∧ { f, r ∈ ListNode } ∧ {b = null ∨ b ∈ ListNode} }
{post: *Si queda vacía*: queue = front = <b>, rear = <b> ; b = null
Si tiene 2do elemento: queue = front = <b>, rear = <r>}

isEmpty(queue)

Retorna verdadero si el front de la cola es igual a null (Cola vacía).

{pre: queue = front = <f>, rear = <r> ∧ {f, r ∈ ListNode ∨(f = null ∧ r= null)} }
{post: True o False}

TAD HashTable

1. HashTable= {table = <t>, MAX_SIZE = <c>}
2. HashTable tiene asociada una función hash h, la cual va a recibir como entrada una llave key. $h(key)$ retorna la posición de la tabla en la cual se va a almacenar el valor junto con la llave asociada.
3. HashTable implementa exploración lineal para solucionar las colisiones que se lleguen a presentar en la tabla.

{inv: table =< T >; $T = \{\}$ ∧ $\{x | x \in T \rightarrow x \in$ HashNode}
MAX_SIZE = <c> ; $c \in Z^+$ } Nota: $T \subseteq$HashNode

Operaciones Primitivas:

-	createHashTable		→ HashTable
-	add	HashTable x Object(key) x Object (value)	→ HashTable
-	get	HashTable x Object (key)	→ Object

-	containsValue	HashTable x Object (value)	→ Boolean
-	containsKey	HashTable x Object (key)	→ Boolean
-	set	HashTable x Object(key) x Object (value)	→ HashTable

createHashTable()

Crea una tabla de hash vacía con una table T de HashNode de tamaño MAX_SIZE.

{pre: True }
{post: hashtable = {table = <T>, MAX_SIZE = <c>}}

add(hashTable, key, value)

Añade un nuevo valor $value$ a la table t de a hashTable. La tabla guardará tanto key como $value$ en la posición $h(key)$.

{pre: hashTable ≠ null ∧ key ≠ null ∧ $value$ ≠ null}
{post: hashTable = {table = <T>, MAX_SIZE = <c>}}

get(hashTable, key)

En caso de que la tabla tenga la llave key, se retorna el valor $value$ en la posición $h(key)$ de la tabla, tal que en dicha posición $h(key)$ también se almacene la llave key.

En caso de que no contenga la llave key, se retorna null.

{pre: hashTable ≠ null ∧ key ≠ null}
{post: <$value$> o null}

containsValue(hashTable, value)

Retorna True en caso de que en alguna posición de la tabla, se encuentre el valor $value$.

{pre: hashTable ≠ null ∧ $value$ ≠ null}
{post: True o False}

containsKey(hashTable, key)

Retorna True en caso de que en alguna posición de la tabla, se encuentre la llave key.

{pre: hashTable ≠ null ∧ key ≠ null}
{post: True o False}

set(hashTable, key, value)

En caso de que $key \in T$, establece el valor $value$ en la posición $h(key)$ donde se almacene la llave key.

{pre: hashTable ≠ null ∧ key ≠ null ∧ $value$ ≠ null}
{post: hashTable = {table = <T>, MAX_SIZE = <c>}}

TAD Node		
Node= {value = <v>}		
{inv: value ≠ null}		
Operaciones Primitivas:		
- createNode	Object	→ Node
- getValue	Node	→ Object
- setValue	Node x Object	→ Node

createNode(value)

Crea un nodo que contiene el valor $value$.

{pre: $value$ ≠ null}
{post: node= {value = <$value$>}}

getValue(node)

Retorna el valor $value$ que almacena el nodo.

{pre: node = {value = <$value$>}}
{post: <$value$>}

setValue(node, value)

Define $value$ como el nuevo valor que almacena el nodo.

{pre: node ≠ null}
{post: node = {value = <$value$>}}

TAD ListNode

ListNode= {value = <v>, next = <n>, previous = <p>}

{inv: value ≠ null ∧ next, previous ∈ ListNode}

Operaciones Primitivas:

- createListNode	Object	→ ListNode
- getNext	ListNode	→ ListNode
- getPrevious	ListNode	→ ListNode
- setNext	ListNode x ListNode	→ ListNode
- setPrevious	ListNode x ListNode	→ ListNode
- getValue	ListNode	→ Object
- setValue	ListNode x Object	→ ListNode

createListNode(value)

Crea un ListNode que almacena el valor $value$ y define next y previous como null.

{pre: $value$ ≠ null}
{post: listNode = {value = <$value$>, next = null, previous = null} }

getNext(listNode)

Retorna el elemento next (nextListNode) de $listNode$. nextListNode puede ser un ListNode o null.

{pre: $listNode$ ≠ null}
{post: nextListNode = {value = <$value$>, next = <n>, previous =<p>} o
nextListNode = null }

getPrevious(listNode)

Retorna el elemento previous(prevListNode) de $listNode$. prevListNode puede ser un ListNode o null.

{pre: $listNode \neq$ null}
{post: prevListNode= {value = <$value$>, next = <n>, previous =<p>} o
prevListNode= null }

setNext(listNode, nextListNode)

Cambia el valor actual de next en $listNode$. El nuevo valor de next será $nextListNode$.

{pre: $listNode \neq$ null $\wedge$ ($nextListNode \in ListNode \vee nextListNode = null$)}
{post: listNode = {value = <$value$>, next = <$nextListNode$>, previous =<p>} }

setPrevious(listNode, prevListNode)

Cambia el valor actual de previous en $listNode$. El nuevo valor de previous será $prevListNode$.

{pre: $listNode \neq$ null $\wedge$ ($prevListNode \in ListNode \vee prevListNode = null$)}
{post: listNode = {value = <$value$>, next = <n>, previous =<$prevListNode$>} }

getValue(listNode)

Retorna el valor $value$ que almacena $listNode$.

{pre: listNode= {value = <$value$>, next = <n>, previous =<p>}}
{post: <$value$>}

setValue(listNode, value)

Define $value$ como el nuevo valor que almacena $listNode$.

{pre: listNode≠ null}
{post: listNode= {value = <$value$>, next = <n>, previous =<p>}}}

TAD HashNode

HashNode= {value = <v>, key = <k>}

{inv: value ≠ null ∧ key ≠ null}

Operaciones Primitivas:

- createHashNode	Object x Object	→ HashNode
- getKey	HashNode	→ Object
- setKey	HashNode x Object	→ HashNode
- getValue	HashNode	→ Object
- setValue	HashNode x Object	→ HashNode

createHashNode(value, key)

Crea un HashNode que almacena el valor $value$ y la llave key.

{pre: $value$ ≠ nul ∧ key ≠ null}
{post: hashNode = {value = <$value$>, key = <key>} }

getKey(hashNode)

Retorna la llave key que almacena $hashNode$.

{pre: hashNode = {value = <$value$>, key = <key>}}
{post: <key>}

setKey(hashNode, newKey)

Define $newKey$ como el nuevo valor para key de $hashNode$.

{pre: hashNode = {value = <$value$>, key = <key>} ∧ newKey ≠ null}
{post: hashNode = {value = <$value$>, key = <$newKey$>}}

getValue(hashNode)

Retorna el valor $value$ que almacena $hashNode$.

{pre: hashNode = {value = <$value$>}}
{post: <$value$>}

setValue(hashNode, value)

Define $value$ como el nuevo valor que almacena $hashNode$. {pre: hashNode≠ null} {post: hashNode= {value = <$value$>}}

5. DIAGRAMAS DE CLASE

El diagrama de clase diseñado se encuentra en la carpeta docs.

6. DISEÑO DE LOS CASOS DE PRUEBA

Casos de prueba:

Nombre	Clase	Escenario
setup1HT	HashTable Test	No existe ningún elemento de tipo HashTable.
setup2HT	HashTable Test	Existe un objeto HashTable con todos sus espacios disponibles vacíos.
setup3HT	HashTable Test	Existe un objeto HashTable con un espacio lleno, con key 232 y valor 1.
setup4HT	HashTable Test	Existe un objeto HashTable sin espacios vacíos.

Nombre	Clase	Escenario
setup1Q	Queue Test	No existe un objeto de tipo Queue.
setup2Q	Queue Test	Existe un objeto de tipo Queue vacío, es decir que su elemento en *front* va a ser nulo.
setup3Q	Queue Test	Existe un objeto de tipo Queue con un *front* igual a un ListNode cuyo valor es 7 y un *rear* con otro ListNode cuyo valor es 9

Nombre	Clase	Escenario
setup1ST	Stack Test	No existe ningún elemento de tipo Stack.

setup2ST	Stack Test	Existe un elemento de tipo Stack vacío, es decir que su *top* es null.
setup3ST	Stack Test	Existe un elemento de tipo Stack con un *top* (ListNode): cuyo *valor* es 7 y su *next* es otro ListNode, con *valor* 9 y *next* igual a null.

Diseño de Casos de Prueba

Objetivo de la Prueba: Validar que al agregar una nueva llave y valor, se ubiquen correctamente en la tabla.

Clase	Método	Escenario	Valores de Entrada	Resultado
HashTable	add	setup2HT	key = 1 value = 5	La tabla hash contiene la key 1 y el valor 5
HashTable	add	setup3HT	key = 1 value = 5	La tabla hash contiene la key 1 y el valor 5
HashTable	add	setup3HT	key = 232 value = 500	La tabla hash no contiene el valor 500
HashTable	add	setup4HT	key = 1 value = 500	La tabla hash no contiene el valor 500

Objetivo de la Prueba: Validar que al buscar el contenido de una llave se retorna el valor correcto.

Clase	Método	Escenario	Valores de Entrada	Resultado
HashTable	get	setup2HT	key = 1	null la tabla hash no contiene la key 1 por lo que no tiene valor asociado
HashTable	get	setup3HT	key = 232	1 la tabla hash contiene la key 232 por lo que devuelve su valor el cual es 1
HashTable	get	setup4HT	key = 1	1 La tabla contiene la llave 1, por lo que devuelve el valor 1

Objetivo de la Prueba: Validar que al cambiar el valor del contenido de una llave, este se actualice correctamente.

Clase	Método	Escenario	Valores de Entrada	Resultado
HashTable	set	setup2HT	key = 1 value = 5	La tabla no contendrá el valor agregado, en este caso, 5
HashTable	set	setup3HT	key = 232 value = 5	key = 232 ahora contiene el valor 5 en vez de 1
HashTable	set	setup4HT	key = 1 value = 500	key = 1 ahora contiene el valor 500 en vez de 1

Objetivo de la Prueba: Validar que al buscar un valor dentro del conjunto de llaves se retorna el valor booleano adecuado.

Clase	Método	Escenario	Valores de Entrada	Resultado
HashTable	containsValue	setup2HT	Value = 2	false
HashTable	containsValue	setup3HT	value = 11	True

Objetivo de la Prueba: Validar que al buscar una llave dentro del conjunto de llaves se retorna el valor booleano adecuado.

Clase	Método	Escenario	Valores de Entrada	Resultado
HashTable	containsKey	setup2HT	key = 1	false
HashTable	containsKey	setup3HT	key = 232	true

Objetivo de la Prueba: Validar que el elemento front sea correcto.

Clase	Método	Escenario	Valores de Entrada	Resultado

Queue	front	setup2Q	Ninguna	Null Obtiene un elemento nulo ya que la cola se encuentra vacía
Queue	front	setup3Q	Ninguna	7 Obtiene el ListNode del valor en *front* cuyo valor es 7

Objetivo de la Prueba: Validar que el elemento rear sea correcto.

Clase	Método	Escenario	Valores de Entrada	Resultado
Queue	rear	setup2Q	Ninguna	Null Obtiene un elemento nulo ya que la cola se encuentra vacía
Queue	rear	setup3Q	Ninguna	9 Obtiene el ListNode del valor en *rear* cuyo valor es 9

Objetivo de la Prueba: Validar que al añadir un elemento a la cola, este sea añadido al final apropiadamente.

Clase	Método	Escenario	Valores de Entrada	Resultado
Queue	enqueue	setup2Q	15	Se agrega exitosamente el elemento con valor 15 al final de la fila, al estar vacía, tanto *front* como *rear* van a apuntar a este valor.
Queue	enqueue	setup3Q	15	Se agrega exitosamente el elemento con valor 15 al final de la fila, es decir que *rear* apuntará a 15, con un *next* cuyo valor es 9 y su *next* (*front*) es igual a 7.

Objetivo de la Prueba: Validar que al retirar un elemento de la fila (front), se remueva y re acomode adecuadamente.

Clase	Método	Escenario	Valores de Entrada	Resultado

Queue	dequeue	setup2Q	Ninguna	Se atrapa una NullPointerException debido a que se está tratando de sacar de una cola vacía.
Queue	dequeue	setup3Q	Ninguna	El elemento en rear y front harán referencia al mismo valor que en este caso es 7.

Objetivo de la Prueba: Verificar que la cola se encuentre vacía.

Clase	Método	Escenario	Valores de Entrada	Resultado
Queue	isEmpty	setup2Q	Ninguna	True La fila se encuentra vacía por lo que deberá retornar un valor verdadero
Queue	isEmpty	setup3Q	Ninguna	False La fila contiene dos elementos por lo que no se encuentra vacía

Objetivo de la Prueba: Validar que se obtiene el valor en la posición *top* del Stack, en el caso de estar vacío, regresará un valor nulo.

Clase	Método	Escenario	Valores de Entrada	Resultado
Stack	top	setup2ST	Ninguna	*null* La pila devuelve el elemento en la posición top, que en este caso está vacío
Stack	top	setup3ST	Ninguna	La pila retorna el valor del elemento en *top*, que en este caso es 7

Objetivo de la Prueba: Validar que al añadir un elemento al Stack, este elemento queda como el nuevo valor en *top*.

Clase	Método	Escenario	Valores de Entrada	Resultado
Stack	push	setup2ST	8	El nuevo elemento top de la pila será 8, es decir que la pila ya no se encontrará vacía

Stack	push	setup3ST	8	El nuevo elemento top de la pila será 8

Objetivo de la Prueba: Validar que al retirar un elemento del Stack, este tenga un elemento menos; en caso de estar vacío desde un principio, permanece igual.

Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Stack	pop	setup2ST	Ninguna	Se atrapa una NullPointerException debido a que se está tratando de sacar de una pila vacía.
Stack	pop	setup3ST	Ninguna	La lista queda apuntando a un elemento ListNode cuyo valor es 9

Objetivo de la Prueba: Verificar que la pila se encuentre vacía

Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Stack	isEmpty	setup2ST	Ninguna	True La pila efectivamente está vacía con un *top* cuyo valor es nulo.
Stack	isEmpty	setup3ST	Ninguna	False Es una pila que cuenta con un valor en *top* distinto a nulo

7. ANÁLISIS DE LA COMPLEJIDAD TEMPORAL

A continuación se presenta el análisis de complejidad temporal para los algoritmos de ordenamiento implementados. *orderGamesByShelf* (Bubble Sort) y *sortClientList* (Insertion Sort).

orderGamesByShelf: Donde n es igual al tamaño de availableGames y aGames.

orderGamesByShelf(ArrayList<String> availableGames) {	
String l = "";	1
ArrayList<String> finalLsit = new ArrayList<>();	1
for (int i = 0; i < availableGames.size(); i++) {	n + 1

l += games.get(availableGames.get(i)).getShelf().getName(); }	n
char[] aGames = l.toCharArray();	1
int m = aGames.length;	1
for (int i = 0; i < m-1; i++){	n
for (int j = 0; j < m-i-1; j++) {	$\frac{n(n+1)}{2} - 1$
if (aGames[j] > aGames[j+1]) {	$\frac{n(n+1)}{2} - n$
char temp = aGames[j];	$\frac{n(n+1)}{2} - n$
aGames[j] = aGames[j+1];	$\frac{n(n+1)}{2} - n$
aGames[j+1] = temp;	$\frac{n(n+1)}{2} - n$
String tempCode = availableGames.get(j);	$\frac{n(n+1)}{2} - n$
availableGames.set(j, availableGames.get(j + 1));	$\frac{n(n+1)}{2} - n$

availableGames.set(j + 1, tempCode);
}
}
}
	$\frac{n(n+1)}{2} - n$

for (int i = 0; i < aGames.length; i++) {	n + 1
finalLsit.add(aGames[i]+"");	n
}	
return availableGames; }	1

Para el algoritmo anterior se tiene la complejidad $T(n)$ como:

$$T(n) = 1 + 1 + (n + 1) + n + 1 + 1 + n + \frac{n(n+1)}{2} - 1 + 7(\frac{n(n+1)}{2} - n) + (n$$

$$T(n) = 6 + 4n + \frac{n(n+1)}{2} + \frac{7n(n+1)}{2} - 7n$$

$$T(n) = 6 - 3n + \frac{8n(n+1)}{2}$$

$$T(n) = 6 - 3n + 4n(n+1)$$

$$T(n) = 6 - 3n + 4n^2 + 4n$$

$$T(n) = 4n^2 + n + 6$$

En base al análisis anterior y a la función $T(n)$ nos es posible afirmar que la complejidad temporal del algoritmo de ordenamiento implementado en *orderGamesByShelf* es de $O(n^2)$.

sortClientList: Donde n es igual al tamaño de passedClients.

public static void sortClientList(ArrayList<Client> passedClients) {	
for(int i = 1; i < passedClients.size(); i++) {	n
Client key = passedClients.get(i);	n-1
for(int j = i-1; j>=0 && key.getTime() < passedClients.get(j).getTime(); j--) {	$\frac{n(n+1)}{2}$ —
Client temp = passedClients.get(j);	$\frac{n(n+1)}{2}$ —
passedClients.set(j, key);	$\frac{n(n+1)}{2}$ —

passedClients.set(i, temp);
}
}
}
	$\frac{n(n+1)}{2}$ —

Para el algoritmo anterior se tiene la complejidad $T(n)$ como:

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

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

$$T(n) = -\frac{4}{2} + \frac{n^2+n}{2} + \frac{n^2+n}{2} + \frac{n^2+n}{2} + \frac{n^2+n}{2}$$

$$T(n) = \frac{4n^2+4n-4}{2}$$

$$T(n) = 2(n^2 + n - 1)$$

$$T(n) = 2n^2 + 2n - 2$$

En base al análisis anterior y a la función $T(n)$ nos es posible afirmar que la complejidad temporal del algoritmo de ordenamiento implementado en *sortClientList* es de $O(n^2)$.

8. ANÁLISIS DE LA COMPLEJIDAD ESPACIAL

En esta sección se presenta el análisis de complejidad espacial de los algoritmos de ordenamiento implementados en los métodos *orderGamesByShelf* (Bubble Sort) y *sortClientList* (Insertion Sort), analizados en la sección anterior.

Tipo	Variable	Cantidad de Valores Atómicos
Entrada y Salida	availableGames	n
Auxiliar	l	1
	finalList	n
	i	1
	j	1
	m	1
	aGames	n
	temp	1
	tempCode	1

Sumando los valores de la tercera columna podemos hallar la complejidad espacial del algoritmo.

Complejidad Espacial Total = n + 1 + n + 1 + 1 + 1 + n + 1 + 1 + 1 = Θ(n)

Complejidad Espacial Auxiliar = 1 + n + 1 + 1 + 1 + n + 1 + 1 + 1 = Θ(n)

Complejidad Espacial Auxiliar + Salida = n + 1 + n + 1 + 1 + 1 + n + 1 + 1 + 1= Θ(n)

Tipo	Variable	Cantidad de Valores Atómicos
Entrada	passedClients	n

Salida	-	0
Auxiliar	i	1
	key	1
	j	1
	temp	1

Sumando los valores de la tercera columna podemos hallar la complejidad espacial del algoritmo.

Complejidad Espacial Total = n + 0 + 1 + 1 + 1 + 1 = n + 4 = Θ(n)

Complejidad Espacial Auxiliar = 4 = Θ(1)

Complejidad Espacial Auxiliar + Salida = 4 + 0 = Θ(1)