Algoritmos y Estructuras de Datos

Laboratorio 2 Unidad 1

Nelson López

Carlos Lizalda.

Santiago Chasqui.

1. Identificación del problema: La empresa Epic Games solicita el desarrollo de ciertas mejoras a su juego bandera conocido como Fortnite, las cuales deberán estar incluidas en la nueva versión del software y que serán descritas con mayor detalle a continuación:

R1	Categorizar jugadores por su destreza de juego.
Resumen	Crea un ranking de jugadores categorizándolo por la destreza de cada jugador
Entrada	Jugadores.
Salida	Ranking de Jugadores.

R2	Crear partidas exclusivas.
Resumen	Crea una partida exclusiva de acuerdo con el tipo de plataforma a la que pertenezca el usuario.
Entradas	Jugadores
Salida	Partida exclusiva de jugadores

Falta especificar el requerimiento sobre la latencia

R3	Crear partida especial
Resumen	Se crea una tipo de partida específica para el evento de San Valentin con una jugabilidad distinta a la utilizada comúnmente.
Entradas	Jugadores
Salidas	Partida especial.

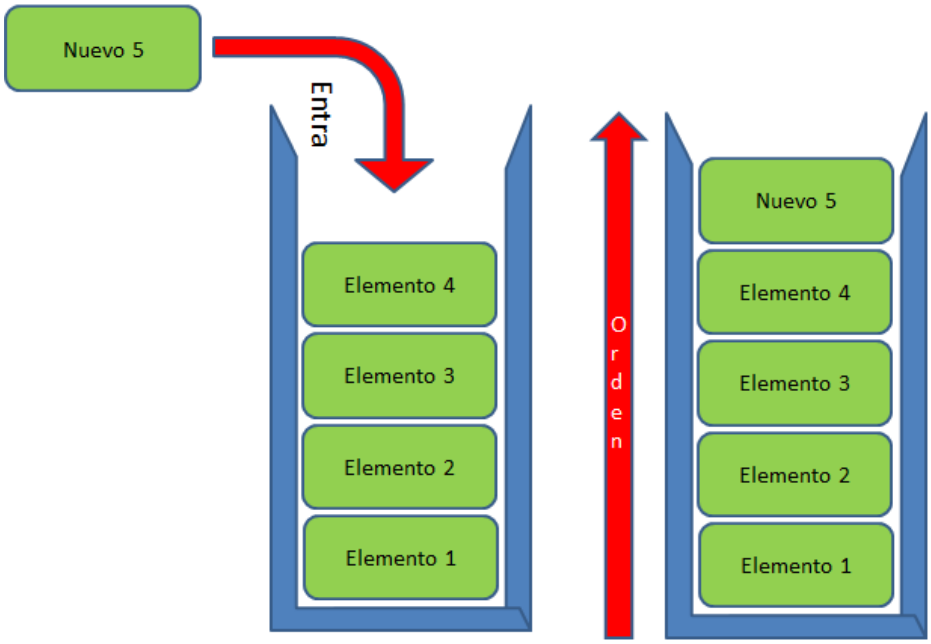
Requerimientos no funcionales:

- Los jugadores de una misma partida deben tener una latencia muy cercana.

2. Recopilación de la información: A continuación se describirán los conceptos formales necesarios para abordar el problema de manera pertinente:

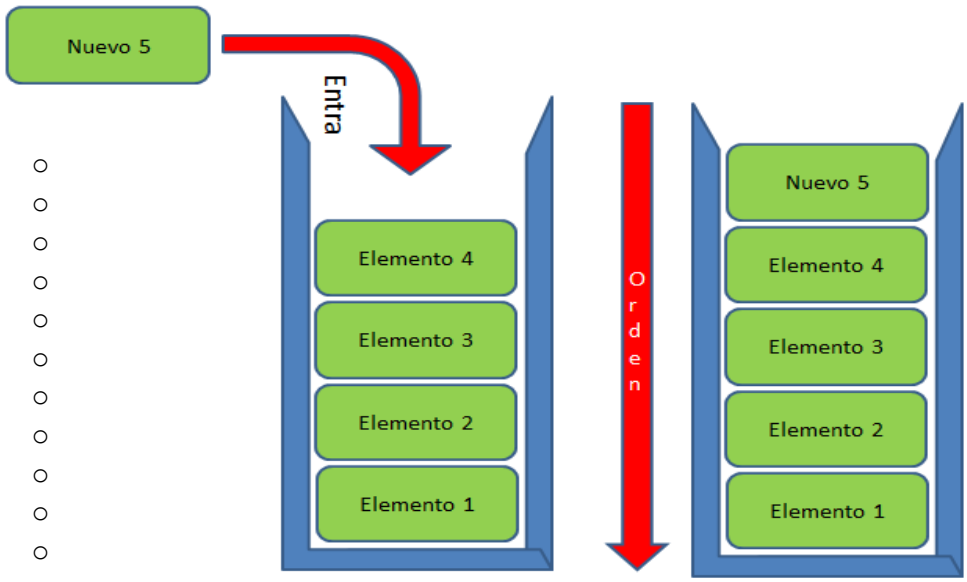
- **Estructura de Datos:** En términos sencillos, una estructura de datos una forma de organizar un conjunto de datos con el objetivo de facilitar su manipulación. Entre las estructuras de datos más conocidas se encuentran:

    - **Pilas:** Una pila es una colección ordenada de datos a los que solo se puede acceder por un extremo, denominado tope o cima de la pila. Esta es una estructura de datos que se caracteriza porque el último elemento en entrar es el primero que debe salir. Este tipo de estructura se denomina LIFO (Last-In-First-Out). La siguiente es una representación gráfica de esta estructura de datos:



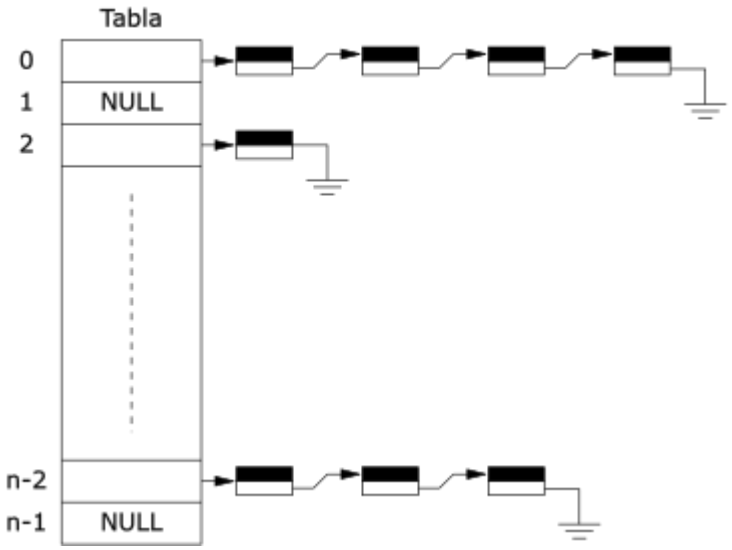
(Blanco, 2017)

- **Colas:** Es una estructura de datos que se caracteriza porque el primer datos en entrar es el primero en salir. Este tipo de estructuras se conocen como estructuras FIFO. La siguiente es una representación gráfica de esta estructura de datos:



(Blanco, 2017)

- **Tablas Hash:** "Es una estructura de datos no lineal cuyo propósito final se centra en llevar a cabo las acciones básicas (inserción, eliminación y búsqueda de elementos) en el menor tiempo posible, mejorando las cotas de rendimiento respecto a un gran número de estructuras." (Heileman, 1994). La siguiente es una representación gráfica de esta estructura de datos:



(uc3m, 2017)

- **Interfaz:** Se compone de un conjunto de declaraciones de signaturas de métodos sin implementar que especifican un comportamiento específico para una o varias clases con la ventaja de que una clase puede implementar diferentes interfaces.

- **Generics:** Permiten asignar parámetros a clases, interfaces o métodos de forma que sólo admitan tipos de objetos ya especificados. El siguiente es un ejemplo de implementación de generics:

Before-after generics

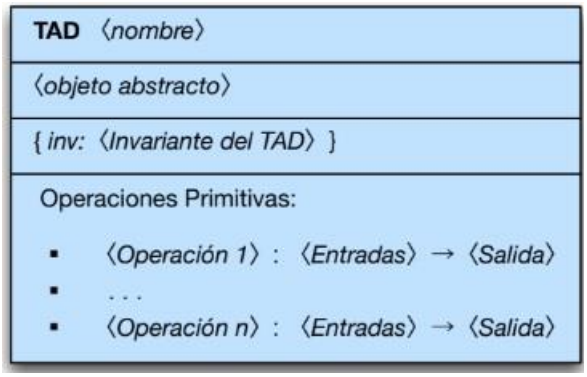
```
// before generics
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1))
assert s.equals("Hello world!");


// with generics
List<String> words = new ArrayList<String>();
words.add("Hello ");
words.add("world!");
String s = words.get(0)+words.get(1); // no explicit casts
assert s.equals("Hello world!");
```

«since generics are implemented by erasure
at bytecode level, two sources above will be identical»

(slideshare, 2012)

- **Tipos de Datos Abstractos:** Se diferencian de los tipos de datos en el sentido de que los datos abstractos son especificados de manera precisas y diseñados independientemente de cualquier implementación, es decir que pueden ser implementados en cualquier lenguaje de programación ya que su forma de definirlos proporciona la información necesaria para hacerlo. El siguiente es un ejemplo de cómo se representa un tipo de dato abstracto de manera formal:



(Villalobos, 1996).

- **Pruebas Unitarias:** Una prueba unitaria es un mecanismo que permite comprobar la eficacia de determinado software. Esto se lleva a cabo creando diferentes casos de prueba que, al ejecutarlos en el programa, corroboran el correcto funcionamiento del mismo. El siguiente es un diseño formal de pruebas unitarias:

Clase	**Método**	**Escenario**	**Valores de Entrada**	**Resultado**
Tablero	testTraducirNotacion()	escenario1	jugada =a7a6 convertida =6858	Verdadero. La notación se tradujo en coordenada de fila y columna
Tablero	testMoverFicha()	escenario1	jugada= g8h6	Verdadero. La ficha se movió a la casilla de llegada.
Tablero	testNotacionInvalidaExcepcion()	escenario1	jugada= sdfsdfsd	Verdadero. El mensaje de excepción corresponde al de notación invalida

Basado en el Libro J. Villalobos en [Villalobos, 1996].

TADS VA EN ETAPA 6.

3. Búsqueda de soluciones creativas:

Existen numerosas estructuras de datos que se encargan de almacenar información de una manera particular, haciendo que cada una de ellas tenga diferentes características que pueden adaptarse de mejor manera a cierto tipo de problemas dependiendo de las condiciones y necesidades que proporcione el mismo.

Teniendo en cuenta lo anterior se procederá a hacer uso de una técnica conocida como lluvia de ideas para enlistar algunas de las opciones que a primera vista son factibles para resolver de manera pertinente el problema abordado:

Alternativa 1: Tabla Hash

Esta es una estructura de datos que se caracteriza porque funciones como buscar o insertar son notablemente rápidos, haciendo que sea una excelente alternativa para solucionar algunos aspectos del problema ya que se requiere un nivel de eficiencia aceptable en los algoritmos usados en la solución del problema.

Alternativa 2: Pilas

Una pila es una estructura de datos útil en situaciones en las que se necesita acceder de manera particular a la información guardada debido a que el último dato almacenado es el primer dato que deberá salir. Esta alternativa podría ser usada para almacenar las armas pertenecientes a un jugador teniendo en cuenta que la última arma recolectada será la primera en usar cuando el jugador solicite esta opción.

Alternativa 3: Colas

Una cola es una estructura de datos útil en situaciones en las que se necesita acceder de manera particular a la información guardada debido a que el primer dato almacenado es el primer dato que deberá salir. Esta alternativa podría ser usada tentativamente en el ranking de jugadores pero deberá ser una opción a analizar más a fondo para determinar si es la estructura más eficiente para solucionar este problema.

Alternativa 4: Montículos

"La estructura de datos conocida como montículo es un arreglo de objetos que podemos ver como un árbol binario casi completo. Cada nodo del árbol corresponde a un elemento del arreglo" (Thomas H. Cormen, 2009)**.** Esta estructura de datos también podría incluirse en el grupo de candidatos a suplir la funcionalidad de crear un ranking para los jugadores.

Alternativa 5: Cola de Prioridad

“Una cola de prioridad es una estructura de datos que mantiene un conjunto de S elementos, cada uno de ellos con un valor asociado llamado clave” (Thomas H. Cormen, 2009). Esta estructura de datos es una de las más opcionadas para ser utilizada en la implementación del ranking de jugadores, debido a que, al ser un array, es posible llevar a cabo sus principales procesos en un corto lapso de tiempo comparado con los procesos de otras estructuras.

Alternativa 6: Lista

“Una lista es una estructura de datos lineal que se puede representar simbólicamente como un conjunto de nodos enlazados entre sí.” (Castrillon, 2015)

Alternativa 7: Arbol Binario

“Un árbol binario es un conjunto finito de elementos, el cual está vacío o dividido en tres subconjuntos separados: El primer subconjunto contiene un elemento único llamado raíz del árbol. El segundo subconjunto es en sí mismo un árbol binario y se le conoce como subárbol izquierdo del árbol original. El tercer subconjunto es también un árbol binario y se le conoce como subárbol derecho del árbol original“ (Serrano, 2013). Este tipo de estructura puede usarse para soportar el ranking de jugadores, pero se deberá tener en cuenta la complejidad temporal del mismo y comprarla con otras estructuras para así elegir la más eficiente.

Alternativa 8: Árbol n-ario

“Un árbol n-ario es una estructura recursiva, en la cual cada elemento tiene un número cualquiera de árboles n-arios asociados. Estos árboles corresponden a la generalización de un árbol binario. La diferencia radica en que esta estructura puede manejar múltiples subárboles asociados a cada elemento, y no solamente 2, como en el caso de los árboles binarios.” (Blanco, 2017). Esta estructura de datos también puede tomarse en cuenta para crear el ranking de jugadores debido que, al existir jugadores con el mismo puntaje resultaría más conveniente utilizar este tipo de estructura.

Alternativa 9: Grafos

“Los grafos no son más que la versión general de un árbol, es decir, cualquier nodo de un grafo puede apuntar a cualquier otro nodo de éste (incluso a él mismo).” (Asencio, 2011).

Esta estructura de datos puede tenerse en cuenta para soportar el modo multiplataforma del juego, aunque esta alternativa se deberá analizar con detalle para concluir si esta estructura tan

“robusta” debe implementarse necesariamente o si otras estructuras pueden solventar el mismo problema y a un menor costo.

4. Transición de las ideas a los diseños preliminares

A continuación se evaluarán individualmente cada una de las alternativas planteadas en el inciso anterior y se procederá a descartar las propuestas menos factibles teniendo en cuenta los requerimientos del problema abordado, es decir, sus funcionalidades y condiciones de eficiencia. Para ello se procederá a aplicar la técnica de revisión selectiva que consiste en describir los aspectos relevantes de la información abordada dividiéndolos en dos secciones conocidas como pros y contras:

Alternativa	**Ventajas**	**Desventajas**
1. Tabla Hash	Los algoritmos que manipulan este tipo de estructura son muy eficientes dado que tienen una complejidad temporal O(1).	El proceso de redimensionamiento de la tabla resulta una maniobra costosa de implementar.
2. Pilas	Su funcionamiento puede emparejarse directamente con situaciones de la vida real. Los procesos de manipulación de la estructura son relativamente rápidos	Solo es posible acceder al primer elemento de la estructura y además se debe definir su tamaño antes de ser creado.
3. Colas	Su funcionamiento puede emparejarse directamente con situaciones de la vida real. Los procesos de manipulación de la estructura son relativamente rápidos	Solo es posible acceder al último elemento de la lista y también se debe definir su tamaño antes de ser creado.
4. Montículos	Los objetos guardados pueden ser organizados de manera ascendente o descendente. La ordenación se realiza con una complejidad temporal relativamente óptima.	Se debe definir su tamaño en el momento de crear la estructura.
5. Colas de Prioridad	Los objetos que se encuentren dentro de esta estructura pueden ordenarse prioritariamente definiendo un criterio especifico.	Debe definirse su tamaño en el momento de crear la estructura.

6. Lista	La inserción y extracción de elementos tiene un costo independiente del tamaño de la lista.	Para acceder a un elemento de la lista es necesario acceder a todos los elementos que se encuentren antes de él.
7. Árbol Binario	Su funcionamiento puede emparejarse directamente con situaciones de la vida real. La información puede guardarse de manera ordenada. Puede redimensionarse sin mayor problema	Si un árbol no está equilibrado, sus funcionalidades pueden no ser eficientes. No es posible usarlo para datos que no requieran un orden.
8. Árbol n-ario	Su funcionamiento puede emparejarse directamente con situaciones de la vida real. Puede redimensionarse sin mayor problema.	No es posible guardar información de manera ordenada, aunque si de manera jerárquica.
9. Grafos	Su funcionamiento puede emparejarse directamente con situaciones de la vida real. Todos sus nodos pueden apuntar a otro nodo entre sí, incluso puede apuntar al mismo. Puede redimensionarse fácilmente	La complejidad espacial de esta estructura suele ser elevada.

Teniendo en cuenta el análisis anterior, resulta necesario descartar las estructuras de datos cuyo funcionamiento no proporciona una solución óptima al problema planteado. Este es el caso de los grafos y el árbol binario debido a que son estructuras robustas que no permiten establecer un criterio de ordenamiento sencillo y fácil de manipular y que pueden ser reemplazadas por otras estructuras que pueden cumplir con este tipo de criterios y a un bajo costo.

5. Evaluación y Selección de la Mejor Solución:

A continuación se expondrán los criterios mediante los cuales se elegirán los algoritmos de ordenamiento que permitan solucionar de manera óptima el problema planteado.

Criterio 1: Complejidad temporal de los métodos que manipulan la estructura de datos.

Complejidad Temporal	**Puntuación**
O (n logn)	2
O (n+k)	4
O (1)	8

Criterio 2: Tipo de estructura de datos clasificada de acuerdo a su forma de almacenamiento en memoria.

Tipo de Estructura	**Puntuación**
Estática	3
Dinámica	6

Criterio 3: Similitud entre el funcionamiento de la estructura de datos y la dinámica del problema a resolver. Por ejemplo, es muy común el hecho de que se usen Hash Tables para implementar diccionarios debido a que su operatividad se asemeja de manera notoria.

Tipo de Estructura	**Puntuación**
Misma operatividad	10
Diferente operatividad	5

En el siguiente recuadro se mostraran las diferentes alternativas que, luego de evaluarlas con los criterios descritos, se elegirán las que cumplan con las condiciones establecidas con mayor rigurosidad.

	Criterio 1	**Criterio 2**	**Criterio 3**	**Total**
Alternativa 1. Tabla Hash	8	3	10	21
Alternativa 2. Pilas	8	6	10	24
Alternativa 3: Colas	8	6	10	24
Alternativa 4. Montículos	4	3	5	12
Alternativa 5. Colas de Prioridad	4	6	10	14
Alternativa 6. Listas	4	6	10	20
Alternativa 7: Árbol binario.	2	3	5	10

Selección definitiva: Teniendo en cuenta el método de selección utilizado anteriormente, se llegó a la conclusión de que las estructuras que deberán ser usadas en las mejoras del juego son las siguientes: Pilas, colas, colas de prioridad, tabla hash y listas. Las pilas serán usadas para implementar la funcionalidad de la recolección de armas, dado que la última arma recogida será la primera que estará disponible. Las colas de prioridad serán usadas en el ranking de jugadores clasificándolos de manera descendente de acuerdo al puntaje respecto de su skill. Las tablas hash se utilizaran para desarrollar el modo multiplataforma debido a este tipo de estructura permitirá agrupar de manera más eficiente los objetos de tipo jugador teniendo en cuenta que la búsqueda de cualquier elemento en un slot es de complejidad temporal O(1).

6. Preparación de informes y especificaciones:

Especificación del problema:

- **Problema:** Realizar mejoras al juego Fortnite.
- **Entrada:** Información de los jugadores.
- **Salidas:** Ranking de jugadores, partidas especiales.

Como parte de las especificaciones de las estructuras de datos utilizadas para resolver el problema se definieron los TADs de cada una de ellas:

TAD Hash Table
$(< k_1, V_1 >, < k_2, V_2 >, \dots, < k_n, V_n >)$
$(V_i \in X \mid 0 \leq i \leq n)$ X es el conjunto domino de los valores con un tipo de dato particular $(k_i \in U \mid 0 \leq i \leq n)$ U es el universo de llaves
• CrearHashTable(entero) -> <Hash Table> • Insertar (U, X) -> <Hash Table> • Buscar (U) -> <Hash Table, X> • Eliminar (U) -> <Hash Table>

CrearHashTable( )
*Crea un nuevo hash table del tamaño indicado como parámetro. {pre: TRUE} {post: hashTable. Se agregó un elemento a la tabla en la posición indicada por la función hash}

Insertar (U, X)
*Inserta un nuevo elemento en la tabla en la posición indicada por la función hash. {pre: $\mid 0 \leq U \leq n$ } {post: hashTable. Se agregó un elemento a la tabla en la posición indicada por la función hash}

Buscar (U)
*Busca el elemento en la posición U (llave) de la hash table y lo retorna. {pre: $\mid 0 \leq U \leq n$ }

{post: hashTable. Se agregó un elemento a la tabla en la posición indicada por la función hash}

Eliminar (U)

*Elimina el elemento en la posición U (llave) de la hash table.
{pre: | $0 \leq U \leq n$ }

{post: hashTable. Se eliminó el elemento de la tabla hash}

EstáVacío ( )

*Retorna un valor de verdad que indica si la tabla hash está vacía o no.
{pre: | $TRUE$ }

{post: Determina si la tabla está vacía o no}

TAD Lista

$\{k_1, k_2, k_3, k_4, \ldots , k_n\}$

$\{inv: \forall k_i.\ k_i \in X \mid 0 \leq i \leq n\}$ Dónde X es un tipo de dato no primitivo

- CrearLista: < > → <List>
- Insertar (Índice, X) → <List>
- Eliminar (Índice) → <List>
- EstáVacía ( ) → <List>

crearLista ( )

*Crea una lista sin ningún elemento.
{pre: | $TRUE$ }

{post: Crea una lista vacía}

agregar (índice, ítem)

*Agrega el ítem en la posición indicada por el índice.
{pre: | $1 \leq indice \leq tamaño + 1$ }

{post: Inserta el ítem en la posición de la lista indicada por el índice
//Los ítems en la posición índice+1 son movidos una posición a la derecha}

Eliminar (índice)

*Elimina el elemento de la lista que esté en el índice que entra como parámetro.
{pre: $|0 \leq indice \leq tamaño - 1$ }

{post: Determina si la tabla está vacía o no}

EstáVacío ( )

*Retorna un valor de verdad que indica si la tabla hash está vacía o no.
{pre: $| TRUE$ }

{post: Determina si la lista está vacía o no}

Obtener (índice)

*Retorna el valor que se encuentra en la posición indicada por el índice.
{pre: $| 0 \leq indice \leq tamaño - 1$}

{post: Se ha encontrado el elemento en la posición del índice}

Tamaño ( )

*Retorna un número entero que indica el tamaño de la lista
{pre: $| TRUE$ }

{post: Determina el tamaño de la lista}

TAD Cola

$\{k_1, k_2, k_3, k_4, \ \dots \ , k_n\}$

$\{inv: \forall k_i.\ k_i \in X \mid 0 \leq i \leq n\}$ Dónde X es un tipo de dato no primitivo

- CrearCola ( ) → <Cola>
- encolar (Ítem) → <Cola >
- desencolar ( ) → < Cola >
- EstáVacía ( ) → < Cola >

crearCola ( )

*Crea una cola sin ningún elemento.
{pre: $| TRUE$ }

{post: Crea una cola vacía}

encolar (ítem)

*Agrega el ítem al final de la cola.
{pre: | $la\ cola\ existe$ }

{post: Se ha agregado el ítem al final de la cola.}

desencolar ()

*Elimina el elemento que está en el frente de la cola y lo retorna.
{pre: |$La\ cola\ no\ está\ vacía$ }

{post: Se ha eliminado el último elemento de la cola}

EstáVacío ( )

*Retorna un valor de verdad que indica si cola está vacía o no.
{pre: | $La\ cola\ existe$ }

{post: Determina si la lista está vacía o no}

TAD Pila

$\{k_1, k_2, k_3, k_{4,} \ \dots \ , k_n\}$

$\{inv: \forall k_i.\ k_i \in X \mid 0 \leq i \leq n\}$ Dónde X es un tipo de dato no primitivo

- CrearPila: < > → <Pila>
- Empilar (Índice, X) → <Pila>
- Eliminar (Índice) → <Pila>
- EstáVacía ( ) → <Pila>

crearPila ( )

*Crea una lista sin ningún elemento.
{pre: | $TRUE$ }

{post: Crea una pila vacía}

push (ítem)

*Agrega el ítem en la cima de la pila
{pre: | $La\ pila\ no\ está\ vacía$}}

{post: Se ha agregado el ítem en la pila}

pop ()

*Elimina el elemento que está en la cima de la pila.
{pre: |$La\ pila\ no\ está\ vacía$ }

{post: Determina si la tabla está vacía o no}

EstáVacío ( )

*Retorna un valor de verdad que indica si la pila está vacía o no.
{pre: | $TRUE$ }

{post: Determina si la lista está vacía o no}

TAD Heap

$\{k_1, k_2, k_3, \dots, k_{i-1}, k_i, \dots, k_n\}$

$\{inv: \forall k_i.\ k_i \in X \mid 0 \leq i \leq n\ \}$

- ✓ $Dónde\ X\ es\ un\ tipo\ de\ dato\ no\ primitivo.$
- ✓ $i\ es\ el\ tamaño\ del\ heap.$
- ✓ $n\ es\ el\ tamaño\ del\ heap.$

- CrearMonticulo: < > → <Montículo>
- agregar (X) → <u>≤</u>Montículo>
- Eliminar (Índice) → <Montículo>
- EstáVacío ( ) → <Montículo>
- Maximo ( ) → <Montículo>
- Ordenar ( ) → <Montículo>

crearMonticulo ( )

*Crea una lista sin ningún elemento.
{pre: | $TRUE$ }

{post: Crea una pila vacía}

agregar (ítem)
*Agregar el ítem en la cima de la pila {pre: | $La\ pila\ no\ está\ vacía$}} {post: Se ha agregado el ítem en la pila}
Eliminar ()
*Elimina el elemento que está en la cima de la pila. {pre: |$La\ pila\ no\ está\ vacía\ $ } {post: Determina si la tabla está vacía o no}

EstáVacío ( )
*Retorna un valor de verdad que indica si la pila está vacía o no. {pre: | $TRUE\ $ } {post: Determina si la lista está vacía o no}

Maximo ( )
*Retorna un valor de verdad que indica si la pila está vacía o no. {pre: | $TRUE\ $ } {post: Determina si la lista está vacía o no}

Ordenar ( )
*Retorna un valor de verdad que indica si la pila está vacía o no. {pre: | $TRUE\ $ } {post: Determina si la lista está vacía o no}

Adicional a ello se elaboró una representación gráfica de las mejoras aplicadas al software denominado diagrama de clases, con el fin de tener una visión más clara de lo que va a ser el juego luego de añadir las modificaciones desarrolladas.

7. Implementación del Diseño:

A continuación se presentara una lista de las tareas a implementar por el programa:

A. Crear un ranking de jugadores de acuerdo a su destreza.
B. Crear un modo multiplataforma.
C. Crear un modo de partida especial.

<table>
<tr><th>Especificación de subrutinas</th><th>Construcción</th></tr>
<tr><td>
<table>
<tr><td>Nombre</td><td>getSkill</td></tr>
<tr><td>Descripción</td><td>Este es un método que devuelve el valor de la skill de un jugador calculándolo de acuerdo a su desempeño en el juego.</td></tr>
<tr><td>Entrada</td><td>Número de muertes, puntaje en ping, tiempo jugado, tiempo en el top.</td></tr>
<tr><td>Retorno</td><td>Valor de la skill.</td></tr>
</table>
</td><td>

```
public int getSkill() {
      return (int) (((((double) kills /

deads)*25) + (headShoots*10) +

(averageSurvivalTime * 5) + (TimesInTopTen *

20) + (TimePlayed * 5) - (ping * 30))/100);

}
```

</td></tr>
</table>

Nombre	getRanking
Descripción	Este es un método que se encarga de devolver el ranking de jugadores.
Entrada	Nombre y valor de la Skill de cada jugador
Retorno	Ranking de jugadores organizado de manera descendente.

```
public Player[] getRanking() {
		rank.heapSort();
		Player[] ranking = new Player[rank.size()];
		for (int i = ranking.length-1; i >=0 ; i--) {
			ranking[i] = rank.getMin();
		}
		return ranking;
	}
```

Nombre	addPlayer
Descripción	Este es un método que se encarga de añadir un nuevo jugador a la lista de jugadores.
Entrada	Datos del jugador.
Retorno	Nuevo jugador agregado a la lista de jugadores.

```
public void addPlayer(Player p) {
		rank.add(p);
	}
```

Nombre	set
Descripción	Este método se encarga de modificar un objeto específico almacenado en un slot particular en la Hash Table.
Entrada	Clave y valor de la Hash Table
Retorno	El objeto en el slot indicado ha sido modificado

```
public void set(K key, T value) {
		if (array[hash(key)] == null) {
			array[hash(key)] = new
CList<HashNode<T, K>>();
			array[hash(key)].add(new HashNode<T,
K>(value, key));
		} else {
			array[hash(key)].add(new HashNode<T,
K>(value, key));
		}
		numberOfElements++;
		alpha = numberOfElements/array.length;
		if (alpha > 0.7) {
			rehashing();
		}
	}
```

Nombre	get
Descripción	Este método devuelve el objeto que se encuentre en el slot al que haga referencia la clave dada como parametro
Entrada	Clave
Retorno	Objeto en el slot correspondiente a la clave dada.

```
	public T get(K key) {
		T value = null;
		List<HashNode<T, K>> list = array[hash(key)];
		if (list != null) {
			for (int i = 0; i < list.size(); i++) {
				if
(list.get(i).getKey().equals(key)) {
					value =
list.get(i).getData();
				}
			}
		}
		return value;
	}
```

Nombre	remove
Descripción	Este método se encarga de eliminar el objeto ubicado en el slot correspondiente a la clave.
Entrada	Clave.
Retorno	Objeto eliminado en el slot indicado.

```
	public T remove(K key) {
		T value = null;
		int index = -1;
		List<HashNode<T, K>> list = array[hash(key)];
		if (list != null) {
			for (int i = 0; i < list.size(); i++) {
				if
(list.get(i).getKey().equals(key)) {
					value =
list.get(i).getData();
					index = i;
					numberOfElements--;
					alpha =
numberOfElements/array.length;
				}
			}
		}
		if (index != -1) {
			array[hash(key)].remove(index);
		}
		return value;
	}
```

Nombre	maxHeapify
Descripción	Este método se encarga de mantener la condición de que el elemento de máximo valor siempre deberá ser el padre.
Entrada	Jugador
Retorno	Montículo ordenado bajo el criterio definido.

```
	public void maxHeapify(T arr[], int i) {
		int largest = i; // Initialize largest as
root
		int l = 2*i + 1; // left = 2*i + 1
		int r = 2*i + 2; // right = 2*i + 2

		// If left child is larger than roots
		if (l < heapLast &&
arr[l].compareTo((T)arr[largest]) > 0)
			largest = l;
		else largest = i;
		// If right child is larger than largest so
far
		if ( r < heapLast &&
arr[r].compareTo((T)arr[largest])>0)
			largest = r;

		// If largest is not root
		if (largest != i)
		{
			T swap = arr[i];
			arr[i] = arr[largest];
			arr[largest] = swap;

			// Recursively heapify the affected
sub-tree
			maxHeapify(arr, largest);
		}
	}
```

Nombre	swap
Descripción	Intercambia los elementos que se encuentran en disposiciones distintas en un arreglo
Entrada	Indice
Retorno	Sin salida

```
public void swap(int index, int parent) {
T temp = array[index];
	array[index] = array[parent];
	array[index] = temp;

}
```

Nombre	heapSort
Descripción	Se encarga de ordenar el arreglo que contiene el heap, para ello va obteniendo el máximo valor (con maxHeapify) y lo va dejando al principio del arreglo que contiene la estructura.
Entrada	Jugadores
Retorno	Montículo ordenado.

```
public void heapSort() {
	// TODO Auto-generated method stub
	int n = array.length;

	// One by one extract an element from heap
	for (int i=n-1; i>=0; i--)
	{
		// Move current root to end
		T temp = array[0];
		array[0] = array[i];
		array[i] = temp;

		// call max heapify on the reduced heap
		maxHeapify(array, i, 0);
	}


}
```

Nombre	buildMaxHeap
Descripción	Este método se encarga de construir pequeños montículos ordenados con el fin de juntarlos ordenadamente al final de este proceso
Entrada	Jugadores
Retorno	Montículo de jugadores.

```
public T[] buildMaxHeap(T[] arrayN){
	T[] arrayMax = arrayN;
	int n = arrayN.length;

	for (int i = n / 2 - 1; i >= 0; i--)
		maxHeapify(arrayMax, n, i);

	return arrayMax;
}
```

Nombre	enqueue
Descripción	Este método se encarga de añadir un nuevo elemento al final de la cola
Entrada	Nuevo elemento
Retorno	Cola con el nuevo elemento añadido

```
	public void enqueue(T item){
		if (first == null) {
			first = new CNode<T>(item);
			last = first;
		}else {
			Node<T> temp = new CNode<T>(item);
			last.setNext(temp);
			last = temp;
		}
		size++;
	}
```

Nombre	dequeue
Descripción	Este método remueve un elemento al frente de la cola
Entrada	Sin entrada
Retorno	Cola sin el elemento al frente de la cola

```
	public T dequeue() throws EmptyQueueExeption{
		if (isEmpty()) {
			throw new EmptyQueueExeption("the queue
is empty");
		}
		Node<T> temp = first;
		first = first.getNext();
		size--;
		return temp.getItem();
	}
```

Nombre	front
Descripción	Este método devuelve el elemento que se encuentra al frente de la cola
Entrada	Sin entrada
Retorno	Elemento que se encuentra al frente de la cola

```
	public T front() throws EmptyQueueExeption{
		if (isEmpty()) {
			throw new EmptyQueueExeption("the queue
is empty");
		}
		return first.getItem();
	}
```

Nombre	push
Descripción	Este método inserta un nuevo elemento al frente de la pila
Entrada	Nuevo elemento a insertar
Retorno	Pila con el nuevo elemento insertado.

```
	public void push(T newItem) {
		Node<T> temp = new CNode<T>(newItem);
		temp.setNext(top);
		top = temp;
		size++;
	}
```

Nombre	top
Descripción	Este método se encarga de devolver el elemento que se encuentra al frente de la pila.
Entrada	Sin entradas
Retorno	Elemento al frente de la pila.

```
public T top() throws StackException {
		if (isEmpty()) {
			throw new StackException("the stack is
empy");
		}
		return top.getItem();
	}
```

Bibliografía

Asencio, A. (2011). *iuma.* Obtenido de http://www.iuma.ulpgc.es

Blanco, O. (2 de 12 de 2017). *oscarblancarteblog.* Obtenido de https://www.oscarblancarteblog.com/

Castrillon, C. (2015). *ocw.* Obtenido de ocw.opm.es

Heileman, G. L. (1994). *HeEstructuras de Datos, Algoritmos y Programación Orientada a Objetos.* Madrid: McGraw-Hill.

Serrano, M. (2013). Obtenido de https://www.infor.uva.es/~mserrano/EDI/cap5.pdf

Thomas H. Cormen, C. E. (2009). *Introduction to Alogrithms.* Londres: Massachusetts Institute of Technology.

uc3m. (02 de 06 de 2017). Obtenido de http://www.it.uc3m.es/