**Algoritmos y Estructuras de datos**

**Laboratorio 2 Unidad 2**

**Método de la ingeniería**

**NICOLAS TABORDA HOYOS**

**LAURA HINCAPIE**

**JULIAN DAVID MABESOY**

**JAMES MONTEALEGRE**

## FASE 1 : Identificación del problema

### Contexto Problemático:

Las empresas Epic Games y People Can Fly, desarrolladores de videojuegos, y uno de estos Fornite, el videojuego del año, requieren las soluciones de los problemas, como, la búsqueda de jugadores (adversarios) con la misma destreza para jugar una partida. Otro de los problemas a resolver es, la exclusividad de poder ingresar a una partida, la cual será dictada por medio de la plataforma en la que el jugador se encuentra actualmente (PlayStation, Xbox, PC, etc.). Por último, se requiere implementar un modo de juego donde solamente se pueda usar la última arma que el jugador levantó, la cual solo será posible cambiar si se acaban las municiones o el jugador tome otra, en el caso que se le acaben las municiones esta desaparecerá y se equipara con la última arma que levantó, pero si esta es el arma por defecto (hacha) esta no podrá desaparecer.

### 1. Identificación del problema

Los jugadores de el videojuegos han presentado múltiples recomendaciones y reclamos por diferentes aspectos de este que podrían ser mejorados. Una de las principales problemáticas se ve a la hora de encontrar una partida, pues se asigna una partida a jugadores con diferentes skill, o destreza en el juego. Por lo que al llegar al mapa, muchos jugadores se encuentran con otros y mueren al instante.

El *primer requerimiento* específicamente, consiste en crear un ranking en donde se puedan clasificar a los jugadores según aspectos que se consideren pertinentes a tener en cuenta, para prevenir al máximo la desventaja entre los jugadores en una

partida. Con respecto a este requerimiento, se debe tener en cuenta, que este ranking, debe procurar no afectar el tiempo promedio de búsqueda para comenzar una partida de n jugadores.

*Como segundo requerimiento*, se debe proponer una solución para contrarrestar el problema de latencia o retardos que presenta cada usuario cuando se lleva a cabo la conexión con los servidores de **Amazon**, los cuales administran la infraestructura y soportan la aplicaciones de ***Epic Games*** en la nube.

Por otra parte, se ha identificado una manera de impulsar más el mercado y a las empresas que manejan la plataforma para el juego, por la cual sea posible realizar una partida con diferentes jugadores, dependiendo de la plataforma en la que se encuentren jugando. Por ende, *el tercer requerimiento* que se solicitó es implementar un modo plataforma para las partidas, en donde el tipo de plataforma sea un filtro para hacer parte de una.

*El cuarto requerimiento*, es una implementación especial del juego para San Valentín, se ha identificado para este, crear una TAD en donde los jugadores almacenen sus armas y solamente puedan utilizar la última arma que levantaron y no hay un límite de armas.

En este punto se ha identificado que el jugador entonces, cambiará de arma cuando:

- Levante una nueva arma
- Agote las municiones del arma actual

La empresa Epic Games y People Can Fly han desarrollado el videojuego de zombis Fornite, el cual está habilitado para jugarlo en diferentes plataformas. La última versión de este es Fornite Battle Royale, el cual se juega en Microsoft Windows, macOs, PlayStation 4 y Xbox One.

**FASE 2: Recopilación de la información**

Estructuras de datos: Las estructuras de datos las podemos ver como formas de representar información, las cuales tienen un comportamiento interno específico, se rigen por determinadas reglas y tienen algunas restricciones condicionadas por la forma en la que están construidas internamente.

Estas estructuras son muy útiles pues nos permiten en algunos casos, solucionar problemas de manera muy sencilla. Entre estas podemos identificar los arreglos,

que son estructuras de datos de acceso aleatorio y las listas enlazadas, las cuales con estructuras de datos de acceso secuencial. Otro tipo de estructuras de datos, son los subcasos de estas, tales como las colas y las pilas.

Cola: Son estructuras de datos que se implementan mediante listas enlazadas, en las cuales los elementos que se insertan siempre van al final de la lista, y los que se extraen siempre son los los del inicio. Este tipo de orden en las entradas y salidas se conocen como FIFO( First in - First out).

Estas se caracterizan por el hecho de que sólo podemos acceder al primer y al último elemento de la estructura. Así mismo, los elementos sólo se pueden eliminar por el principio y sólo se pueden añadir por el final de la cola. En la vida real, estos se asemejan a las líneas de espera.

Pila: Consisten en una lista de elementos en la cual solo se tiene acceso al último elemento insertado. La posición en donde se encuentra el último elemento se reconoce como “top”. El tipo de orden en las entradas y salidas se conocen como LIFO (Last in - First out).

Hash Tables: Estas son estructuras de datos, que usualmente se utilizan cuando se requiere almacenar un elevado número de datos sobre los que se necesitan operaciones de búsqueda e inserción muy eficientes. La operación principal de esta estructura de datos es la de búsqueda. Permite el acceso a los elementos a partir de una clave generada por la función hash.

Árboles binarios: Un árbol impone una estructura jerárquica sobre una colección de objetos, en el cual solo hay un punto de entrada y una serie de caminos que va a abriéndose en cada punto hacia sus sucesores.

Fornite: Recopilando información acerca del juego, con jugadores de el, ellos opinaron sobre ciertos aspectos por los que ellos consideran se podrían hacer partidas con jugadores del mismo nivel de skill, como lo son:

- Promedio de muertes por partida
- Ping (Calidad del internet en donde se encuentren)
- Geolocalización (Especialmente para que la comunicación por el idioma, no se dificulte)

- Total de victorias

Por otra parte, conocimos que hay ocho diferentes tipos de armas en el juego, las cuales son:

- Assault Rifles
- SMGs
- Pistols
- Shotguns
- Sniper Rifles
- Machine guns
- Explosives
- Grenades
- Traps

Estas a su vez, se clasifican por colores, según lo buenas que sean en cinco colores, enunciadas en orden, desde la mejor:

| |
|---|
| GOLD |
| ORANGE |
| PURPLE |
| BLUE |
| GREEN |
| GRAY |

Fuentes:
https://platzi.com/blog/estructuras-de-datos-que-son/
https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/TDA/
http://sedici.unlp.edu.ar/bitstream/handle/10915/33386/Documento_completo.pdf?sequence=3&isAllowed=y

**FASE 3: Búsqueda de soluciones creativas**

1. Primer requerimiento, organizar los jugadores en un ranking :

- Alternativa 1: Se podrían poner los jugadores ordenadamente en un árbol binario, y de esta forma recorrer el árbol buscando cada jugador y agregandolo a la partida correspondiente.
- Alternativa 2: Otra forma de poder resolver este requerimiento es definir un atributo que nos informe en qué nivel se encuentra el jugador, y conociendo este, poderlo agregar a una Arraylist con jugadores de la misma altura.
- Alternativa 3: También se puede utilizar el concepto de Hash Tables ya que, si se habla de eficiencia esta estructura de datos nos proporciona esto, podríamos agregar los jugadores de un mismo nivel al mismo espacio (slot) de la tabla hash, ya que cada espacio tiene una lista enlazada de elementos con la misma llave.

2. Segundo Requerimiento, disminuir la latencia entre los jugadores

- Alternativa 1: Crear un socket y un server socket con el cual se pueda simular los tiempos de subida y regreso cuando se genera la conexión entre un gamer y el servidor de Epic games.
- Alternativa 2: Garantizar que los gamers se enlazan con los servidores de Amazon (Servicio en la nube tercerizado - Amazon presta sus servicios de plataforma sobre la nube a Epic games) que se encuentran en el radio más cercano de distancia.
- Alternativa 3: Implementar un sistema de routing mediante el algoritmo de Dijkstra; en donde algunos nodos de esta estructura simulan a los servidores en la nube, y mediante las características del algoritmo determinar el camino más corto a dichos nodos.

3. Tercer Requerimiento, implementar el modelo plataforma

- Alternativa 1: Un atributo que indique la plataforma que se está utilizando y de esta forma almacenar los jugadores en diferentes ArrayList para clasificarlos y sacarlos directamente del ArrayList cuando se busque un jugador con el atributo de la plataforma deseada.
- Alternativa 2: Para empezar se utilizara un atributo que nos permitirá conocer la plataforma en la que está el jugador, después manejaremos cada jugador como un nodo, y lo iremos agregando a una lista de nodos doble a la cual pertenece, por último cuando hayamos clasificado a los jugadores, los agregaremos a una lista de tipo cola.

- Alternativa 3: El primer paso es utilizar un atributo para poder identificar el tipo de plataforma en la que se encuentra el jugador, el siguiente paso es ubicarlo en una Arraylist en el que solamente se encuentran jugadores de la misma plataforma, por último se agregará cada jugador a una lista de tipo cola.

4. Cuarto Requerimiento, Especial San Valentín

- Alternativa 1: Una idea para este requerimiento es la implementación de un arraylist donde se almacenen las armas recogidas, agregando las armas en la posición 1 del arraylist, de esta forma siempre el jugador va a utilizar la última arma recogida, y solo podrá cambiarla cuando se le acaben las balas o tome una nueva arma.
- Alternativa 2: Para esta función podremos hacer uso de la lista tipo pila, cada vez que el jugador recoja un arma, esta será la única que podrá usar hasta que se acabe la munición o tome otra, la única arma que no se podrá descartar es la que se encuentra en la posición cero de la lista.
- Alternativa 3: También para resolver esta parte, podremos hacer uso de los nodos dobles, podemos ir agregando cada arma que se recoja en la última posición, y cuando se acabe la munición de esta se le quitara el puntero y se quedará con el arma anterior, por otro lado si se toma otra arma esta quedará en la última posición y como arma actual.

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

**IDEAS NO VIABLES**

1. En el primer requerimiento se decidió que no se utilizará de ninguna forma, para almacenar los jugadores, los ArrayList ya en el peor de los casos se tendría que recorrer todo el ArrayList para buscar un jugador, y así para las veces que se necesitará buscar un jugador para agregar al juego, por lo tanto el tiempo de búsqueda sería muy grande y no sería nada eficiente
2. Para el segundo requerimiento se descartan las opciones 1 y 3 dado que, con la implementación de sockets, sería algo complicado la simulación de jugabilidad por el cuento de la asignación ip y en direccionamiento. Por otro lado la opción 3 se descarta dado que el routing lleva a cabo esta labor del

camino más corto y basta con hacer ping a los servidores cercanos para que esto de manera implícita llegue de manera precisa.

3. En el tercer requerimiento también se llegó a la la conclusión de descartar la implementación del almacenamiento de los jugadores de la misma plataforma en ArrayList (alternativa uno), ya que, como se había dicho anteriormente, la búsqueda en un ArrayList para el peor de los casos tendría que recorrer todo el ArrayList para encontrar un jugador con las características y la plataforma deseada.
4. En el cuarto requerimiento, como en los anteriores, se descarta de primera mano la implementación de un ArrayList para almacenar las armas ya que no es práctico utilizarlo habiendo más alternativas que conlleven menos tiempo de ejecución.

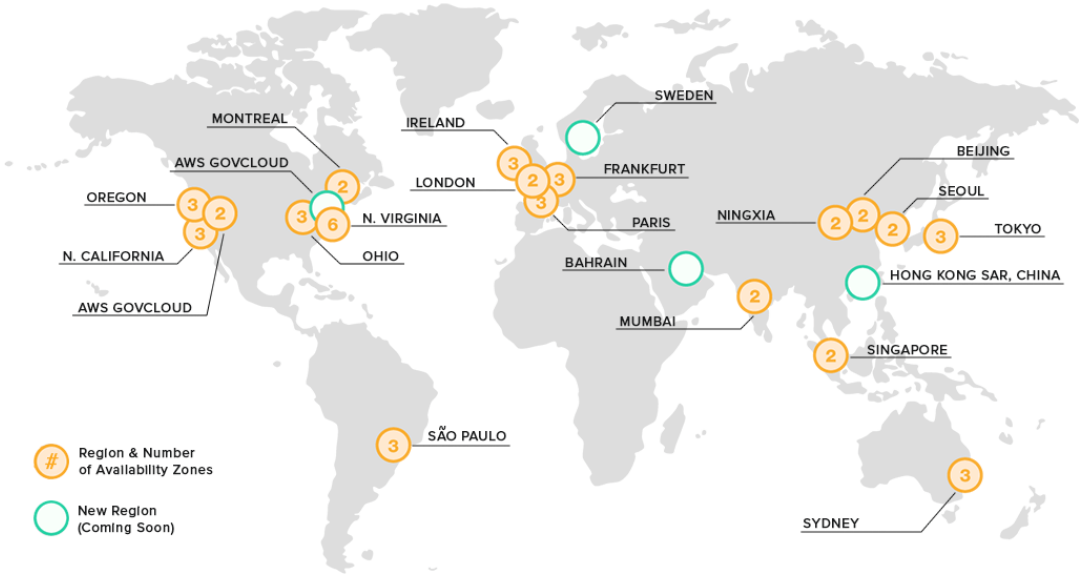
## DISEÑO PRELIMINAR DE IDEAS VIABLES POR IMPLEMENTAR

Después de realizar la respectiva evaluación para todas las ideas, se decidieron descartar las ideas que se mencionaron anteriormente, dejando dos alternativas para el desarrollo de cada requerimiento.

A partir de estas alternativas, se pretende llegar a la que más convenga a la hora de realizar las implementaciones y que cumple con los criterios que se establecerán más adelante.

De cada requerimiento nos quedan dos alternativas para la resolución de cada requerimiento, del primer requerimiento quedó la implementación de un árbol para realizar la respectiva clasificación de todos los jugadores, el árbol puede ser implementado de forma de que los jugadores se agreguen ordenadamente o de una manera arbitraria, siendo cada nodo del árbol un jugador con su respectivo “número” de clasificación y su puntaje. La segunda alternativa que queda para resolver el primer requerimiento es la implementación de una hashtable, las hashtables permiten almacenar objetos en sus slots de manera de que en cada slot se encuentren objetos enlazados entre sí, como listas enlazadas. Pensando en el requerimiento, la hashtable servirá para clasificar todos los jugadores de acuerdo a su puntaje en cada slot de la hashtable de manera eficiente.

En el segundo requerimiento se llevará a cabo un direccionamiento estratégico, es decir, se pedirán los recursos a los servidores próximos para contrarrestar los retardos en el establecimiento y ejecución de las partidas.




Esto quiere decir que si un usuario se encuentra en Colombia por ejemplo, los servidores indicados para una correcta conexión son **Virginia y Ohio**. Mientras que, para una persona en el sur del continente americano como Chile o Argentina, el servidor que proporciona menos retardos estaría en **Sao Paulo.** Es por esto que se propone que se identifique la región o continente de procedencia de cada gamer y asignar una conexión a un servidor contiguo a su ubicación.

Del tercer requerimiento quedaron las ideas de asignar a cada jugador un atributo de acuerdo a la plataforma en la cual iba a jugar, ya que como se mencionaba en la descripción del problema hay diferentes plataformas de las cuales pueden jugar diferentes personas, para poder implementar el requerimiento de modo plataforma y que cada jugador que juegue de esta manera, pueda jugar con personas que estén jugando en la misma plataforma de él, después de tener el atributo creado, cada jugador con ese atributo será agregado a una lista doblemente enlazada para que después de la clasificación y adición a la lista, los jugadores sean agregados a una lista de tipo cola y después puedan iniciar el juego. La otra alternativa que quedó después de la selección, fue la de implementar el mismo mecanismo que la alternativa anterior, solo que en lugar de agregar los jugadores a una lista doblemente enlazada se agregarían a un arraylist y después se adiciona a una lista de tipo cola para luego iniciar el juego.

Del cuarto requerimiento se quedó la alternativa de utilizar una lista tipo pila, para que cada jugador solo pueda tener acceso a la primer arma que recoja dejando como opción utilizarla hasta que se acaben las municiones, además solo puede utilizar otra arma si se acaba las municiones o toma otra. La otra alternativa es agregar las armas a una lista doblemente enlazada en la primera posición, para que cada que el jugador agregue o tome un arma esta quede en la posición cero y tenga que utilizarla como la alternativa anteriormente descrita.

**FASE 5: Evaluación y selección de la mejor solución**

A partir de las ideas que se realizaron anteriormente, se deben generar unos criterios de evaluación que permita clasificar las ideas de acuerdo a su calificación. Con base en el resultado obtenido después de la evaluación, se tomará la decisión de cuál alternativa implementar. A continuación se encuentran los criterios que se escogieron. Cada uno tiene asociado varios valores numéricos para determinar el puntaje de cada alternativa.

-*Criterio A:* Precisión de la solución. Es solucionado el requerimiento de manera:

- [2]Completa
- [1]Incompleta

-*Criterio B*: Eficiencia. El requerimiento es resuelto de un modo:

- [4] Muy eficiente
- [3] Eficiente
- [2] Ineficiente
- [1] Muy deficiente

-*Criterio C*: Facilidad de codificar. ¿A partir de lo planeado, la solución permite desarrollar un algoritmo que cumpla con el requerimiento y que sea manejable a la hora de llevarlo al código?

- [3] Manejable de llevar a la implementación en java
- [2] No es manejable, pero permite que sea llevado a la implementación en java
- [1] No es manejable y tampoco se puede llevar a la implementación en java

Requerimiento 1:

| | *Criterio A* | *Criterio B* | *Criterio C* | *Total* |
|---|---|---|---|---|
| *Alternativa 1* | *Completa[2]* | *Ineficiente[2]* | *Manejable[3]* | *7* |
| *Alternativa 3* | *Completa[2]* | *Muy eficiente[4]* | *No es manejable, pero permite su implementación [2]* | *8* |

Requerimiento 2:

| | *Criterio A* | *Criterio B* | *Criterio C* | *Total* |
|---|---|---|---|---|
| *Alternativa 1* | *Completa [2]* | *Muy Eficiente [4]* | *No es manejable, pero permite su implementación [2]* | *8* |
| *Alternativa 2* | *Completa [2]* | *Eficiente [3]* | *No es manejable, pero permite su implementación [2]* | |

Requerimiento 3:

| | *Criterio A* | *Criterio B* | *Criterio C* | *Total* |
|---|---|---|---|---|
| *Alternativa 2* | *Completa[2]* | *Eficiente[3]* | *Manejable[3]* | *8* |

| | | | | |
|---|---|---|---|---|
| *Alternativa 3* | *Completa[2]* | *Muy Deficiente[1]* | *Manejable[3]* | *6* |

Requerimiento 4:

| | *Criterio A* | *Criterio B* | *Criterio C* | *Total* |
|---|---|---|---|---|
| *Alternativa 2* | *Completa[2]* | *Muy eficiente[4]* | *Manejable[3]* | *9* |
| *Alternativa 3* | *Completa[2]* | *Eficiente[3]* | *Manejable[3]* | *8* |

Para el primer requerimiento, a pesar de que la implementación de un árbol binario para hacer una búsqueda podría llegar a ser bastante sencilla, la complejidad temporal para su búsqueda no favorece a la necesidad de que el tiempo de búsqueda no se vea afectado. Por otra parte, el no tener un árbol en el que la inserción se haga de manera ordenada (hacerlo así requeriría más tiempo y memoria), no es idea a la hora de acceder a él pues se debe recorrer todo el arreglo, a fin de encontrar jugadores con una skill similar. Por esto, consideramos que un árbol binario para hacer la búsqueda no es la mejor solución. Dejando como mejor solución la implementación de la hashtable, ya que cumple con todos los criterios de evaluación llegando a un mayor puntaje que la alternativa del árbol.

Para el tercer requerimiento a pesar de que se podría implementar un arraylist para el almacenamiento de los jugadores con el mismo modo de plataforma, no es eficiente para acceder y tiene un espacio de almacenamiento de objetos limitada. Por otro lado, la implementación de una lista doblemente enlazada nos da mejor acceso a los jugadores y permite almacenar más jugadores ya que cada jugador

está enlazado a otro y solo hay un jugador raíz. Dejando como mejor solución la implementación de la lista doblemente enlazada.

En el cuarto requerimiento a pesar de la posibilidad de implementar una lista doblemente enlazada para el almacenamiento de las armas, la lista tipo pila ofrece todos los requerimientos que se requieren, valga la redundancia, para cumplir con la totalidad del cuarto requerimiento. A pesar de estos, también, se llegó a la conclusión de que la implementación de la lista tipo pila cumplia con mas criterios y obtuvo mayor puntaje que la lista doblemente enlazada, de esta forma quedó como la que resolvería el requerimiento a la hora de la implementación.

Después de realizarse la respectiva calificación y clasificación de las alternativas se llegaron a 4 alternativas que serían las que se implementaran en el código.

1. Requerimiento 1: Implementación de HashTable para el ranking de los jugadores
2. Requerimiento 2:
3. Requerimiento 3: Implementación de lista doblemente enlazada y lista tipo cola.
4. Requerimiento 4: Implementación de lista tipo pila.

**FASE 6: Preparación de informes y especificaciones**

*Diseño de casos de pruebas unitarias*

*Prueba Estructura Pila*

| *Objetivo: Probar que el método push() funciona correctamente en diferente ocasiones* | | | | |
|---|---|---|---|---|
| *Clase: Stack* | | *Método: +push(<E> e):void* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | *Valores de entrada* | *Resultado* |

| | | | | |
|---|---|---|---|---|
| *1* | *Verifica que se agregue un elemento a la pila. Es decir, que el elemento se agregue en una nueva posición y que esta se convierta en la última* | *Stack.size()=3*<br>*Stack.peek()=e1* | *<E> e4* | *Stack.size()=4*<br>*Stack.peek()= e4* |
| *2* | *Verifica que cuando se agregue un elemento a la pila, el resto de elementos se conservan en el orden en el que estaban.* | *stack.size()=2;*<br>*stack().get(0)=e1;*<br>*stack().get(1)=e2* | *<E> e3* | *stack.size()=3;*<br>*stack.get(0)=e1;*<br>*stack.get(1)=e2;*<br>*stack.get(2)=e3;* |

| *Objetivo: Probar que el método pop() funciona correctamente para situaciones distintas.* | | | | |
|---|---|---|---|---|
| *Clase: Stack* | | *Método: +pop():<E>* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que cuando se desapila un elemento este se elimina de la estructura* | *Stack.size()=3*<br>*Stack.peek()= e1* | *<E> e = stack.pop();* | *e.equals(e1)* |

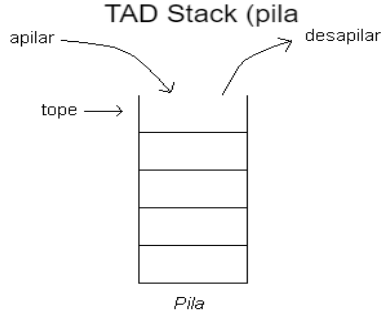
| | | | | |
|---|---|---|---|---|
| *2* | *Verifica que cuando se intente desapilar el último elemento de la pila esta no se tendrá éxito.* | *Stack.size()=1* | *<E> e = stack.pop();* | *Stack.size()=1 ;* |
| *3* | *Verifica que cuando se desapila un elemento, el resto de elementos se conserva en el orden en el que estaban.* | *Stack.size()=3 Stack.get(0)= e1; Stack.get(1)= e2; Stack.get(2)= e3;* | *<E> stack.pop()* | *Stack.size=2 Stack().get(0) =e1 Stack().get(1) =e2* |

| *Objetivo: Probar que el método top funciona correctamente para situaciones distintas.* | | | | |
|---|---|---|---|---|
| *Clase: Stack* | | *Método: +top(): <E>* | | |
| *Caso #* | *Descripción de la prueba* | *Escenario* | *Valores de entrada* | *Resultado* |
| *1* | *Verifica que cuando se observa un elemento, es es el que estaba de último de la estructura* | *Stack.size()=3 stack.peek()=e1* | *<E> e=stack.top() ;* | *e.equals(e1);* |
| *2* | *Verifica que cuando se observa un elemento, este no se elimina de la pila.* | *stack.size()=3; stack.peek()= e1* | *<E> e = stack.top();* | *stack.size(3); stack.top().equals( e);* |

| | | | | |
|---|---|---|---|---|
| *3* | *Verifica que cuando se observa un elemento, los elementos se conservan en el orden en el que estaban.* | *stack.size()=2;*<br>*stack().get(0)=e1;*<br>*stack().get(1)=e2;* | *<E> e = stack.peek();* | *stack.size()=2;*<br>*stack().get(0)=e1;*<br>*stack().get(1)=e2;* |

## *TAD PARA PILAS*

TAD Stack (pila




{inv tamanio>=1}

Operaciones Primitivas

| | | |
|---|---|---|
| createStack | | Stack |
| Top | Pila x Elemento | Arma |
| Pop | Pila x Arma | Stack |
| Push | Pila x Elemento | Stack |
| peek | Stack | Elemento |
| isEmpty | Stack | boolean |

**createStack()**

"Construye un Stack vacio"

{pre: }
{post: Stack s =0}

**Top()**

"Devuelve el elemento tope en la pila actualmente"

{pre:true}
{post: Arma={nombre=" "}}

**Pop()**

"Elimina el elemento tope en la pila actualmente"

{pre:true}
{post: Arma={nombre=" "}}

**Push()**

"Ingresa el elemento a la posicion tope de la pila y este devuelve el elemento agregado"

{pre:true}
{post: Arma={nombre=" "}}

**peek()**

"Recupera el valor del elemento que esta en el tope"

{pre:t Stack s != 0 es decir, s = <a1,a2,a3,...,an}
{post: Elemento an}

**isEmpty()**

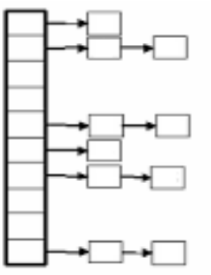
"informa si el Stack esta o no vacia"

{pre:t Stack s}
{post: Verdadero si s==0, falso si es !=0}

## *TAD para HashTables*

**TAD Hash Table(Diccionario)**

estructura de datos que utiliza una función hash para identificar datos mediante una llave o clave



{inv: función hash }

Operaciones primitivas

| | | |
|---|---|---|
| • createTable | | HashTable |
| • tableInset | HashTable x Element | HashTable |
| • tableDelate | HashTable x SearchKey | Element |
| • isEmpty | HashTable | boolean |
| • tableLength | HashTable | int |

**createTable()**

"Crea un HashTable vació"

{pre:}

{pos: HashTable h !=0}

**tableInsert()**

"Agrega a la TableHash h un nuevo elemento e"

{pre: el nuevo elemento a agregar no debe estar en la hash}

{pos: el nuevo elemento se arega con exito}

**tableDelate()**

"elimina un elemento según su código e"

{pre: el elemento existe en la hash}

{pos: el elemento se elimina de la hash con éxito}

**isEmpty()**

"verifica si esta vavio"

{pre: debe existir la tabla hash}

{pos: ejecuta la verificación correctamente}

**tableLength()**

"tamaño de la tabla hash"

{pre: debe existir la tabla y debe tener un tamaño definido}

{pos: retorna el tamaño de la tabla}

***TAD PARA COLAS***

| ***TAD Queue (cola)*** |
|---|
| *Queue =<<$e_1$, $e_2$, $e_3$, ..., $e_n$>, frente, final>* |
| *{inv: 0 ≤ n ∧ Tam(Queue) = n ∧ frente = $e_1$ ∧ final = $e_n$}* |
| *Operaciones primitivas:*<br>● *createQueue→ - →Queue*<br>● *add→Queue x Element → Queue*<br>● *poll→Queue → Element*<br>● *peek→Queue → Element*<br>● *isEmpty → Queue → boolean* |

| ***createQueue()*** |
|---|
| *"Crear un nuevo Queue vacío"* |
| *{pre: - }* |
| *{pos: Queue q = ∅}* |

| ***add()*** |
|---|
| *"Agrega al Queue q un nuevo elemento e"* |
| *{pre: Queue q = <$e_1$, $e_2$, $e_3$, ..., $e_n$> y elemento e ó q = ∅ y elemento e }* |

| |
|---|
| *{pos: Queue q = <$e_1$, $e_2$, $e_3$, ..., $e_n$, e> ó q = <e> }* |

| ***poll()*** |
|---|
| *"Saca del Queue q el elemento en su frente"* |
| *{pre: Queue q ≠ ∅ es decir q = <$e_1$, $e_2$, $e_3$, ..., $e_n$>}* |
| *{pos: Queue q = <$e_2$, $e_3$, $e_4$, ..., $e_n$> y Elemento $e_1$}* |

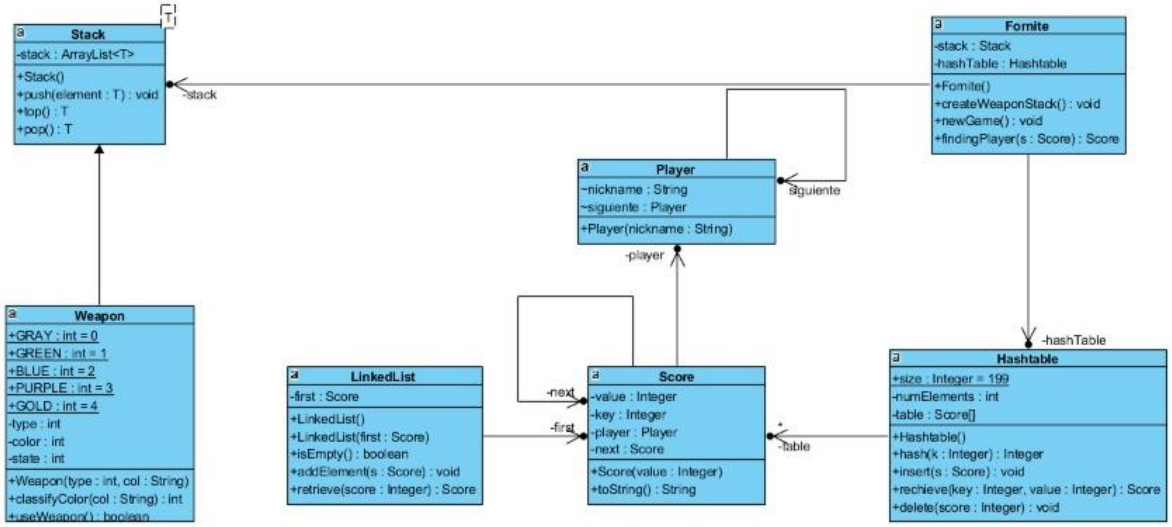
| ***peek()*** |
|---|
| *"Recupera el valor del elemento que está en el frente"* |
| *{pre: Queue q ≠ ∅ es decir q = <$e_1$, $e_2$, $e_3$, ..., $e_n$>}* |
| *{pos: Elemento $e_1$}* |

| ***isEmpty()*** |
|---|
| *"Informa si el Queue q está o no vacía"* |
| *{pre: Queue q}* |
| *{pos: Verdadero si q = ∅, Falso si q ≠ ∅}* |

| *TAD Fornite* |
| --- |
| *Fornite={ Stack = <Stack>, Ping = <ping> ,Hashtable = <hasTable>, Queue = <weaponsQueue}* |
| *{inv: }* |
| *Operaciones Primitivas:*<br>*· Fornite: Fornite*<br>*· findingPlayer Score x Texto Score*<br>*· getScores int x entero hashTable*<br>*· newGame*<br>*· getNextWeapon int x entero String*<br>*· createWeaponStack*<br>*· useWeapon String[]*<br>*· showWeaponsNames String[]*<br>*· catchWeapon*<br>*· getWeapons* |

| *Fornite()* |
| --- |
| *“Crea un nuevo Fornite”* |
| *{pre:-}* |
| *{pos: Fornite a = new Fornite()}* |

## Diagrama de clases Modelo




## Diagrama de clases de Interfaz