MÉTODO DE LA INGENIERÍA

Juan David Vera Usman (A00293816)

FASE 1: IDENTIFICACIÓN DEL PROBLEMA

- Descripción del contexto problemático

Los desarrolladores de Minecraft detectaron un problema de eficiencia con el sistema de recolección de bloques en el inventario, consume mas tiempo del necesario saber si un ítem recogido por el usuario va a parar a un nuevo slot o aumenta en un stack del mismo tipo de ítem en el inventario. Además, se desea implementar una nueva característica al menú de acceso rápido del juego, esto permitirá a los jugadores almacenar en dicha barra todos los stacks pertenecientes al tipo de ítem que hayan elegido. Adicional a esto desean contar con varias de estas nuevas barras de acceso rápido para mejorar su productividad en el juego.

- Identificación del problema

Optimizar el sistema de detección y asignación para los ítems recogidos por el usuario para optimizar el juego y agregar un conjunto de características en el menú de acceso rápido para satisfacer la petición de los usuarios de Minecraft.

- Requerimientos funcionales y no funcionales

El programa debe estar en la capacidad de:

RF1 – Mejorar la velocidad de comprobación para saber si el ítem recogido esta ya en el inventario.

- Entrada: un ítem - salida: ninguna

<pos> ha sido aumentado en una unidad el stack del mismo tipo del ítem recibido o añadido a un nuevo slot del inventario.

RF2 – Identificar todos los ítems en el inventario igual a uno seleccionado por el usuario y remplazar lo que se encuentre en el menú de acceso rápido por esta selección.

- Entrada: un ítem - Salida: ninguna

<pos> la barra de acceso rápido a remplazado sus ítems con la selección por el usuario

RF3 – Crear tantas barras de acceso rápido con la característica mencionada en el requisito funcional dos como el usuario desee.

<pre> la barra de acceso rápido cumple con el requisito funcional dos y el usuario ya decidió el tipo de ítem que contendrá

- Entrada: una barra de acceso rápido -Salida: ninguna

<pos> Habrá mas de una barra de acceso rápido

RNF1 – El tiempo que tome agregar un ítem al inventario del jugador deberá ser mejor que el actual.

FASE 2: RECOPILACIÓN DE LA INFORMACIÓN NECESARIA

- Estructuras de datos

Las estructuras de datos pueden ofrecernos alternativas a la hora de resolver problemas con grupos de datos en este caso para los tres requerimientos es necesario manejar varios conjuntos de datos y revisar conjuntos de datos también. Por eso se tomarán en cuenta las siguientes estructuras de datos:

- Hash Table

Una Hash Table es una estructura de datos que usa un par de valores llave/valor. Esta usa una función hash para calcular un índice dentro de un arreglo en el cual un elemento será guardado o buscado. Al usar una buena función hash, estas operaciones de diccionario pueden trabajar muy bien. Bajo asunciones razonables, el tiempo de búsqueda para un elemento en una Hash Table es O(1).

La lustración uno nos muestra cómo se vería una tabla hash con elementos almacenado dentro de ella una vez calculados los índices para las claves de los elementos almacenados en ella.

Index	
0	
1	
-	
-	
-	
11	defabc
12	

Ilustración 1

- Queue

Una Cola es una estructura de datos que permite almacenar elementos en un orden particular, su característica más útil es la manera en que son consultados y agregados los elementos. Los elementos ingresan por un extremo de la estructura y salen por el otro. Al consultar un elemento de la cola este sale de la misma, por lo tanto, si se quiere conservar el elemento deberá ser ingresado de nuevo a la cola.

Ilustración 2

En la ilustración dos podemos observar tres elementos encolados. Si desease consultar un elemento sacaría de la cola el número 3, si desease continuar sacando elementos el siguiente sería el número 4 y así sucesivamente. Si por lo contrario desease agregar un número este estaría detrás del 2 y seria la cola, el último de la fila.

- Stack

Una Pila es una estructura de datos que permite almacenar elementos en un orden particular, su característica más útil es la manera en que son consultados y agregados los elementos. Los elementos que ingresan a la estructura empiezan a apilarse uno sobre el otro, a diferencia de la cola esta estructura de datos solo tiene una entrada la cual es la misma salida de los elementos. Al consultar un elemento de la pila este sale de la misma, por lo tanto, si se quiere conservar el elemento deberá ser ingresado de nuevo a la pila.

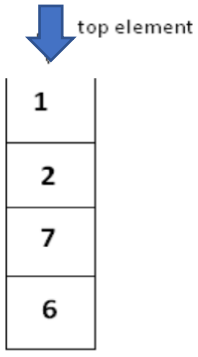

Ilustración 3

En la ilustración tres podemos observar cuatro elementos apilados. Si desease consultar un elemento de la pila sacaría el número 1, si continuase sacaría el número 2, así sucesivamente. Si desease agregar un elemento este estaría encima del número 1 y sería el nuevo elemento en el tope.

FASE 3: BÚSQUEDA DE SOLUCIONES CREATIVAS

Para la búsqueda de alternativas que su puedan utilizar para la solución del problema realice una lluvia de ideas llevando así a cabo la selección de la mejores. Entre las siguientes ideas se encuentran las mejores ideas para satisfacer los tres requisitos funcionales.

Alternativa 1. Realizar una búsqueda binaria en un arreglo que contenga los ítems del inventario

Para esta alternativa realizare sobre un arreglo una búsqueda binaria para encontrar más rápido si el elemento recogido por el usuario esta ya en su inventario. Después será agregado su stack correspondiente o en un nuevo espacio en el arreglo.

Alternativa 2. Emplear una hash table para almacenar los ítems del inventario con resolución de colisiones por encadenamiento

Esta alternativa me permitiría guardar los ítems recogidos por el jugador de manera rápida y consultar los de igual forma. La resolución por colisiones es la parte importante, esta me ayudaría a distribuir bien los ítems. Esta resolución de colisiones me permite almacenar los valores con un mismo tipo de llave en una lista enlazada.

Alternativa 3. Emplear una hash table para almacenar los ítems del inventario con resolución de colisiones por direccionamiento abierto

Esta alternativa me permitiría guardar los ítems recogidos por el jugador de manera rápida y consultar los de igual forma. La resolución por colisiones es la parte importante, esta me ayudaría a distribuir bien los ítems. Esta resolución de colisiones me permite almacenar en diferentes slots de la hash table valores con el mismo tipo de llave.

Alternativa 4. Emplear una cola para guardar los tipos del ítem que selecciona el jugador

Para esta alternativa usaría una cola para guardar la mayor cantidad posible o que posea de stacks del mismo tipo del ítem que el jugador haya elegido. Se vería de la siguiente manera:

Alternativa 5. Emplear una pila para guardar los tipos del ítem que selecciona el jugador

Para esta alternativa usaría una pila para guardar la mayor cantidad posible o que posea de stacks del mismo tipo del ítem que el jugador haya elegido. Se vería de la siguiente manera:

Alternativa 6. Emplear una cola para guardar menús de acceso rápido

Para esta alternativa usaría una cola para guardar los menús de acceso rápido que el jugador desee tener. Se vería de la siguiente forma:

Alternativa 7. Emplear una pila para guardar menús de acceso rápido

Para esta alternativa usaría una pila para guardar los menús de acceso rápido que el jugador desee tener. Se vería de la siguiente forma:

FASE 4: TRANSICIÓN DE LA FORMULACIÓN DE IDEAS A LOS DISEÑOS PRELIMINARES

- **Descarte de ideas no factibles**

Se descartan las siguientes ideas:
El realizar una búsqueda binaria mejoraría bastante la aproximación actual para resolver el problema de saber si un tipo de ítem ya se encuentra en el inventario, el tiempo que tarda actualmente es aproximadamente O(n) donde n es la cantidad de ítems que tenga el inventario. La búsqueda binaria podría dividir este tiempo a la mitad, pero existen mejores estrategias para satisfacer este requisito.

Para el requisito funcional 2 el requisito funcional 1 es clave por lo tanto lo mejor seria guardar en una hash table listas del mismo tipo de ítem. Se descarta la hash table con resolución de colisiones por encadenamiento ya que tendría si dos ítems diferentes dan la misma clave después del hashing podrían presentarse situaciones que harían tardar mucho el tiempo de búsqueda, como por ejemplo:

	Roca		Lana
	Arena		
	Nieve		

En esta hash table ocurrió lo siguiente: el usuario recogió roca y después lana, como después del hashing tuvieron claves diferentes no ocurrió colisión. A continuación, recogió mas roca, lleno su 1er stack como lo lleno se enlaza más roca en el mismo slot. Ahora ocurre lo siguiente, recogió arena y al tener el mismo slot después del hashing se agrega a la lista enlazada. Esto provocara que al agregar arena tarde mas de lo necesario y ni imaginar cuando nieve sea agregada.

Se descarta la alternativa 4, el jugador tendrá en su barra de acceso rápido lo que necesita y le será más conveniente usar lo que tenga de primero, hasta aquí no hay problema con la cola pero, si él recoge materiales le seria más útil que se incremente la cantidad en su selección actual que estaría restringida por practicidad al primer elemento en su barra de acceso rápido, en lugar de incrementar su cantidad en el último stack que podría estar lleno y ocuparle un slot mas en su barra de acceso rápido o ir a su inventario.

Se descarta la alternativa 7, ya que cuando el usuario llegue a su ultima barra de acceso rápido cambiara a su primera barra de acceso rápido en una pila la implementación seria engorrosa y requeriría de mas estructuras de datos para operar de tal forma lo que consumiría más espacio del necesario.

- **Diseños preliminares**

<u>Análisis de alternativa 3</u>

A continuación, procedemos a analizar la alternativa tres: al ser una hash table nos permitirá realizar consultas y agregar elementos en $O(1 + \alpha)$ donde α es el factor de carga de la has table. Al tener resolución de colisiones por direccionamiento abierto todo ítem que recoja el usuario estará bien distribuido.

<u>Análisis de alternativa 5</u>

A continuación, procedemos a analizar la alternativa cinco: dado que este tipo de barras de acceso rápido se emplearán mayormente para construcción al jugador le beneficia no estar moviéndose sobre la barra para seleccionar un nuevo stack sino que lo mejor seria que de primero me quedar un nuevo stack lleno y que lo que recoja se incremente en mi stack principal (el seleccionado) la mejor opción es guardar todos los tipos del mismo ítem en una pila.

<u>Análisis de alternativa 6</u>

A continuación, procedemos a analizar la alternativa seis: para este caso debe comportarse de manera cíclica el deslazamiento entre barras de acceso rápido, para esto lo mejor es usar una cola que almacene las pilas que representan las barras de acceso rápido.

FASE 5: EVALUACIÓN Y SELECCIÓN DE LA MEJOR SOLUCIÓN

En este caso, se escogieron las alternativas tres, cinco y seis como mejores soluciones ya que estas son las que mejor se acomodan a las especificaciones del problema y ofrecen una practicidad y manipulación de los ítems del inventario muy eficiente. Por lo tanto, la alternativa tres resuelve con satisfacción el requisito funcional uno, la alternativa cinco resuelve adecuadamente el requisito funcional dos y por último la alternativa seis resuelve muy bien el requisito funcional tres.

Las alternativas cinco y seis tiene una demostración grafica de su funcionalidad la tres no por ello se muestra a continuación.

Esta aproximación nos muestra que ítems de diferente tipo se guardaran en slots diferentes y al ser llenado un stack se enlazara el siguiente stack del mismo tipo de ítem encima de este, así el agregar siempre será muy cercano a O(1) y tendré los stacks del mismo tipo en un lugar para hacer mas eficiente los requisitos dos y tres.

FASE 6: PREPARACION DE INFORME Y ESPECIFICACIONES

Diagrama de clases:

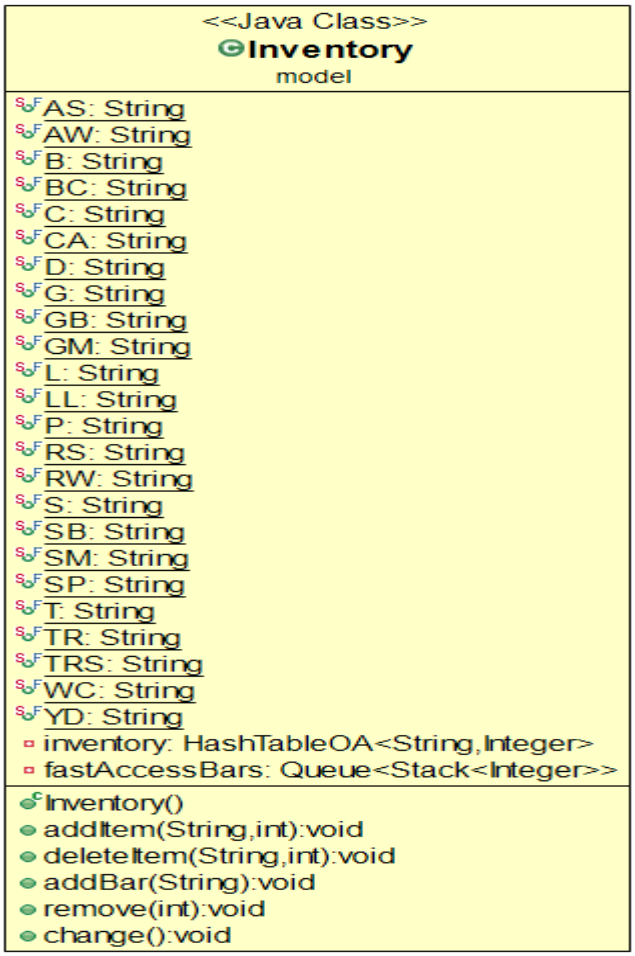

Diagrama de objetos:

UserInventory : Inventory
inventory = <String, Integer> fastAccessBars = Queue<Stack <Integer>>

Representación de las estructuras de datos creadas, con contenido dentro:

inventory:

key = “lana” key = “piedra”

fastAccessBars:

Diseños de los casos de pruebas:

Objetivo: verificar que los items que se agregen entren efectivamente a la tabla hash y los stacks no excedan 64 unidades.

Clase: Agregar addItem(String key, int cantidad)

caso #	Descripcion de la prueba	Escenario	Valores de entrada	Salida
1	Se agregan varios items que no esten en el inventario	1	{key = "roca", value = 34} {key = "lana", value = 2} {key = "tierra", value = 60}	la cantidad de slots ocupados es 3
2	se agregan items que esten en la tabla, los suficientes para que se llene un stack y verificar que se distribuya en el inventario	1	{key = "roca", value = 60} {key = "tierra", value = 28}	la cantidad de slots ocupados es 5
3	se agrega una cantidad menor a un stack para verificar que se acumulan los items apropiadamente	1	{key = "lana", value = 18}	la cantidad de slots ocupados es 5

Clase: Eliminar deleteItem(String key, int cantidad)

caso #	Descripcion de la prueba	Escenario	Valores de entrada	Salida
1	Se eliminan items para disminuir un stack	2	{key = "roca", value = 20} {key = "tierra", value = 30}	el total de roca es 32, el total de tierra es 8
2	Se eliminan items para eliminar un stack	2	{key = "roca", value = 20} {key = "tierra", value = 30}	el total de roca es 32, el total de tierra es 9

Objetivo: verificar que se creen las barras de acceso rapido

Clase: Agregar addBar(String key)

caso #	Descripcion de la prueba	Escenario	Valores de entrada	Salida
1	se agregan 2 barras de acceso rapido a la cola	3	{madera = (64,32)} {piedra = (64,64,64,32)}	la barra actual es madera

Objetivo: verificar que se cambie a la siguiente barra de acceso rapido

Clase: consulta change()

caso #	Descripcion de la prueba	Escenario	Valores de entrada	Salida
1	se cambia a la 3era barra en la cola	4		la barra actual es roca

Referencias

Ref. 1: https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/

Ref. 2: https://www.hackerearth.com/practice/data-structures/queues/basics-of-queues/tutorial/

Ref. 3: https://www.hackerearth.com/practice/data-structures/stacks/basics-of-stacks/tutorial/