UNIVERSIDAD DEL VALLE DE GUATEMALA

Proyecto 1 / Fase 1

Jose Carlos Ceferino Fabian 251043

Axel Antonio Xitumul Chen 25783

Oscar Rodrigo Melchor Estrada 25216

Catedrático:

Ing. Santiago Solorzano Padilla

Algoritmos y Estructuras de datos

Sección 30

Fundamentos Script

Script es un minilenguaje de programación, el cual usa un sistema de bloqueo para proteger salidas en operaciones de bitcoin. Para cada operación se coloca un script de bloqueo (ScriptPubKey), al cual se le debe dar un script de desbloqueo (ScriptSig). Si ambas partes son válidas, la salida se desbloquea y se puede usar la moneda. El script de bloqueo se encuentra en un blockchain público de internet, mientras que el script de desbloqueo es provisto por la entidad que quiere acceder a la moneda

En sí, el lenguaje tiene únicamente dos tipos de dato: los opcodes (funciones con identificador) y los datos (como llaves públicas y firmas). Estos datos se manejan con un stack, el cual es operado con los opcodes dependiendo de cuántos bytes se quieren manipular. Nota: es recomendable hacer la operación de push más pequeña posible cuando deben ingresarse datos al stack

Flujo de validación

Los Scripts se leen de izquierda a derecha, y se consideran válidos si hay 1 elemento diferente de 0 en el stack al momento de terminar de leerse. Es decir, el Script es invalido si:

- El stack final está vacío
- El último elemento en el Stack es 0 (OP_0)
- Hay más de un elemento en el stack final
- El script realiza una salida antes de terminar de leerse

En general, se provee de primero el script de bloqueo y después el script de desbloqueo. Sin embargo, el script de desbloqueo (que contiene llaves) se ejecuta primero.

Estructuras de JCF

Java Collections Framework (JCF) se refiere a varias clases e interfaces que proveen estructuras de datos para almacenar y manipular conjuntos de datos. Dichas clases e interfaces poseen estándares de uso, con lo que hacen el código reusable y fácil de mantener.

Todas las clases e interfaces heredan de Collection, aunque con propósitos diferentes según lo que necesiten.

Interfaz
Clase

Collection
- List
- Set
- Queue
- Map<K, V>

List: es una colección ordenada de elementos, con acceso aleatorio para la inserción y obtención de elementos dentro de ella. Esta estructura se caracteriza por permitir elementos duplicados dentro de ella, al igual que elementos de tipo “null”. Esta clase provee un iterador propio llamado “ListIterator”, el cual es recomendado de usar cuando se necesita recorrer la lista. En cuestion de busqueda de datos, esta clase provee dos metodos diferentes de orden linear (O(n)), aunque es efectiva en la insercion y eliminacion de multiples elementos en cualquier posicion.

Set: es una colección sin orden que no contiene elementos duplicados, permitiendo a lo sumo 1 valor null. Un set no puede contenerse a sí mismo, y hay que tener cuidado si se usan objetos mutables dentro de un set, ya que en caso de que repita otro valor del set esta estructura tiene un comportamiento errático.

Queue: es una colección ordenada de elementos, cuyo propósito es mantener datos previo a que estos sean procesados. Sus elementos pueden tratarse con varios ordenes, como FIFO o LIFO, pudiendo crearse ordenaciones especializadas según lo que se necesite. Esta clase cuenta con métodos especializados de add, remove o element, los cuales regresan un valor especial en vez de dar una excepción si no pueden completarse:

	Da excepción	Da valor especial
Insertar	add(e)	offer(e)
Remover	remove()	poll()
Examinar	element()	peek()

No todas las implementaciones de Queue permiten ingresar null, y ninguna tiene método de comparación (equals) propio (usan el de Object).

Map: Es una colección de valores con orden opcional (depende de la implementación), la cual consiste en pares de valores (llave - valor). Las llaves no pueden duplicarse, y cada mapa puede dar 3 tipos diferentes para ver sus valores: un set de llaves, una colección de valores, o un set de llave - valor. Un mapa puede tenerse a sí mismo como valor (no como llave), aunque esto lleva a comportamientos erráticos. Cada mapa tiene dos constructores principales: uno vacio, y uno que tenga otro objeto de Map, lo que permite hacer una copia del Map original hacia la nueva estructura de Map. Algunos mapas tienen restricciones en los tipos de llaves que usan, y los metodos que modifican mapas no funcionan para todos los mapas (lanzan una excepción)

En base a las anteriores definiciones, consideramos que una clase que implemente Query sería la más adecuada para manejar bitScript. Esto se debe a que puede tener un orden de tipo LIFO para sus elementos, lo cual refleja el funcionamiento de datos en el lenguaje bitScript. Además de ello, nos asegura que los elementos no puedan ser accedidos de forma aleatoria y que no se puedan tener valores nulos.

- Queue
    - Deque
    - BlockingQueue
    - AbstractQueue

Deque: es una colección linear de elementos que permite su manejo al inicio y al fin. Al igual que Queue, tiene metodos especiales para la insercion, eliminacion y acceso a los elementos en la cabeza y la cola de la estructura. Dichos métodos retornan valores especiales en caso de falla, en vez de dar una excepción. La interfaz de Deque puede usarse con comportamiento FIFO (si se usa como Queue) o LIFO (si se usa como stack), teniendo dos metodos para remover elementos dentro de él: removeFirstOccurance() y removeLastOccurance(). Las implementaciones de Deque pueden usar elementos null (aunque se recomienda exhaustivamente que no lo haga, por los métodos especiales), y no cuentan con acceso aleatorio a sus elementos

BlockingQueue: es un Queue cuyo comportamiento permite congelar o bloquear un hilo hasta que una operación pueda completarse. Tiene una capacidad máxima de objetos, y cuenta con los siguientes métodos:

	Da excepción	Da valor especial	Bloquea hilo hasta completarse	Bloquea hilo durante X tiempo máximo
Insertar	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remover	remove()	poll()	take()	poll(time, unit)
Examinar	element()	peek()	N/A	N/A

La clase tiene acceso a los métodos de Collection, pero no es recomendado usarlos debido a su mal rendimiento en tiempo y memoria al ejecutarse.

AbstractQueue: esta clase provee las operaciones base que cualquier Queue debe ser capaz de realizar. Dichas operaciones deben sobreescribirse según el fin que se necesite cumplir. Es recomendado usar esta clase cuando la implementación base no permite elementos de tipo null.

De estas 3 estructuras, consideramos que la más apropiada es AbstractQueue. Esto se debe a que en el programa no usaremos multiples hilos, por lo que BlockingQueue no es adecuada. Asimismo, queremos mantener un solo orden de recorrido para los elementos guardados, lo que elimina a Deque como una opcion apropiada.

- AbstractQueue
    - ArrayBlockingQueue
    - ConcurrentLinkedQueue

- DelayQueue
- LinkedBlockingDeque
- LinkedBlockingQueue
- LinkedTransferQueue
- PriorityBlockingQueue
- PriorityQueue
- SynchonousQueue

ArrayBlockingQueue: es un Queue que usa un array como estructura interna. Los elementos tienen un orden de FIFO, siendo la cabeza el elemento con más tiempo en la estructura de datos y la cola el elemento con menos tiempo. Debido a que se usa un array, este Queue tiene un tamaño fijo, bloqueando sus datos en caso de que la capacidad quiera superarse.

ConcurrentLinkedQueue: es un Queue que usa nodos simplemente enlazados, teniendo un orden de FIFO. Esta estructura está diseñada para ser usada por múltiples hilos, teniendo en su cabeza al elemento de mayor tiempo en la estructura y en su cola el elemento de menor tiempo. Sus iteradores regresan los elementos del Queue una vez (desde el momento de la creacion del iterador), y la clase cuenta con todos los métodos de Queue y Iterator. Algo importante a resaltar es que el método de size() tiene un tiempo de ejecución inconsistente, lo que puede afectar en el rendimiento del programa.

DelayQueue: es un Queue que maneja elementos que implementan Delayed. Dichos elementos solo pueden usarse después de que su tiempo de espera se acabe. La cabeza del Queue es el elemento que tenga más tiempo desde que expiró, y la cola es el elemento con más tiempo de espera. Algo importante a notar es que no pueden retirarse elementos que no hayan expirado, y en caso de que ningun elemento haya expirado se regresa null al llamar a poll() y remove(). Por lo demás, los elementos se tratan con normalidad.

LinkedBlockingDeque: es un Queue que combina un Deque con un BlockingQueue. Los elementos se almacenan en base a nodos enlazados, los cuales se crean dinamicamente al insertar elementos. Esta estructura puede tener una capacidad máxima (opcional), con lo que se previene la expansión innecesaria de nodos en almacenamiento. La mayoria de operaciones tienen un tiempo constante en su ejecucion, a excepción de remove() y contains(), los cuales tienen tiempo linear.

LinkedBlockingQueue: es lo mismo que un LinkedBlockingDeque, aunque no implementa las funcionalidades de un Deque. Debido a esto, sus operaciones se hacen en un solo orden (FIFO), siendo la cabeza el elemento con más tiempo de creación y la cola el elemento con menos tiempo.

LinkedTransferQueue: funciona de la misma manera que ConcurrentLinkedQueue, aunque puede tomar en consideración varios “suppliers” y ordenar elementos en base a su creador.

PriorityQueue: es un Queue que se ordena automáticamente cuando se le añaden nuevos elementos. El orden se hace en base a un Comparator o al orden natural de los elementos, por lo que no pueden agregarse elementos sin Comparable. La cabeza del PriorityQueue es el elemento de menor orden, y en caso de empate estos se resuelven arbitrariamente.

Internamente, se usa un array para guardar los elementos, el cual se redimensiona conforme sea necesario. Por otra parte, los métodos de esta clase dan tiempo O(log(n)) para los métodos de queue y deque, O(n) para remove(E) y contains(E) y O(k) para peek(), element() y size().

PriorityBlockingQueue: es una combinación de PriorityQueue y BlockingQueue, lo que permite tener un orden de elementos con prioridad mientras se usan varios hilos de procesamiento.

SynchronousQueue: es un BlockingQueue con ciertas capacidades especiales. No cuenta con almacenamiento interno y no puede realizar operaciones como peek(), tratándose como si fuese una colección vacía. Esto se debe a que la clase esta diseñada para usarse con varios hilos, en donde un hilo debe pedir un elemento del Queue para que otro pueda insertarlo, y viceversa. Las operaciones deben resultar en la transferencia de datos entre dos hilos simultaneos, ya que de lo contrario se da valor de Null para cualquier operación.

Basándonos en el objetivo del programa, consideramos que la clase más adecuada a implementar para la pila es la de LinkedBlockingDeque. Esto se debe a que sus operaciones tienen un tiempo constante de ejecución (excepto por remove()), lo que nos ayuda en el performance de la aplicación. Sin embargo, esta clase puede tener dos métodos de orden (LIFO o FIFO), lo cual podría causar problemas al momento de procesar los Scripts. Para evitar esto, modificaremos el comportamiento de la clase de forma que solo tenga orden LIFO (el mismo de un Stack). Al momento de usarse en código, el objeto se creará como una instancia de LinkedBlockingDeque de tipo AbstractQueue.

Referencias:

- https://docs.oracle.com/javase/8/docs/api/java/util/List.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Set.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

- https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html
- https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html

- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html
- https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html
- https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html