Informe tp especial

Integrantes: Johnathan Katan,, Diego Bruno Cilla, Francisco Delgado, Joaquin Ormachea y Federico Mamone

**Introducción:**

El objetivo de este informe es en primer lugar presentar una descripción de las estructuras de datos utilizadas para el desarrollo de la Blockchain y del árbol AVL, explicando cómo están compuestas y los métodos implementados.

En segundo lugar, se presenta una justificación de las decisiones de diseño y de por qué se decidió implementar estas estructuras de datos, comparándolas con estructuras alternativas que se tuvieron en cuenta durante el desarrollo, pero que finalmente fueron descartadas.

También se explica cómo fueron implementados los métodos más relevantes, y por qué fueron implementados de esa manera.

En cuarto lugar, también se presentan las convenciones utilizadas para los métodos que involucran el pasaje de archivos, como el modify.

Finalmente, se presenta una conclusión acerca de las decisiones de diseño tomadas a lo largo del desarrollo del trabajo práctico.

**Instalación:**

Para instalar el trabajo práctico, se debe ejecutar el comando *ant* en la carpeta principal, donde está ubicado el archivo *build.xml* (tener en cuenta que hay que tener ant instalado). El target default de *build.xml* generará dos carpetas, una llamada *dist*, y otra llamada *build*. En la carpeta *build* estarán todos los archivos *.class*, y en la carpeta *dist* se encuentra el archivo ejecutable *.jar*. Si se desea limpiar las carpetas generadas al ejecutar *ant*, el archivo *build.xml* también contiene un target llamado *clean*, que borrará las carpetas *build* y *dist*, por lo que sólo sería necesaria ejecutar el comando *ant clean*.

**Clase Blockchain:**

Modelado de la clase:

La clase Blockchain fue creada para modelar la Blockchain sobre la que se mantiene el árbol AVL.

A continuación se presenta una descripción de cómo fue pensada e implementada esta clase, y una justificación de las decisiones tomadas.

En primer lugar, se decidió modelar la Blockchain como una lista simplemente encadenada con header, con la particularidad de que cada elemento tiene una referencia al anterior, por lo que la lista se empieza a recorrer a partir del último elemento. En esta parte de la implementación, es posible analizar el siguiente punto de discusión: ¿Por qué una lista simplemente encadenada donde cada elemento apunta al anterior, y no al siguiente, o por qué no una lista doblemente encadenada? ¿Por qué una lista simplemente encadenada y no un ArrayList?.

Respecto a la primer pregunta, como veremos más adelante, el hecho de que cada elemento apunte al anterior, al siguiente, o a ambos, no modifica la complejidad temporal de los métodos implementados, por lo que se decidió elegir el modelo que le resultó más cómodo al equipo.

Respecto a la segunda pregunta, se requirió un poco más de reflexión para responderla. Sabemos que en un ArrayList las operaciones de acceso a un elemento son de orden O(1), ya que los elementos están contiguos en memoria y se acceden mediante índices. Pero, esto provoca que de vez en cuando, al insertar un elemento, se llegue al límite del tamaño fijo y haya que realocar el espacio donde están los elementos, provocando una inserción de orden O(n) en el peor caso. En cambio, al utilizar una lista simplemente encadenada, las operaciones de inserción siempre serán de orden O(1), ya que los elementos no están contiguos en memoria. El problema de esto es que las operaciones de acceso son de orden O(n). Es por esto que se tuvo que realizar una decisión sobre a qué darle más importancia: ¿Operaciones de acceso a un bloque de la Blockchain, u operaciones de inserción de bloques en la Blockchain? El único momento donde se realiza un acceso a un índice particular de la Blockchain es cuando se ejecuta el comando *modify* para modificar un bloque en específico. Pero cuando se modifica un bloque, lo más probable es que la Blockchain quede inválida, y si la Blockchain es inválida, ya no se podrá seguir usando porque no es confiable. En cambio, los comando como *add* y *remove* se ejecutarán con mucha frecuencia, ya que la idea de esta Blockchain es poder realizar operaciones sobre un árbol AVL. Como se realizarán más operaciones de inserción de bloques que de búsqueda de bloques, se optó por implementar una lista simplemente encadenada en vez de un ArrayList.

Comentarios sobre métodos importantes:

Los métodos “*mine”* y “*valid* *ceros”* son explicados con profundidad en la sección “Calculando el Hash”. Respecto al método “*addBlock”*, es importante justificar por qué se decidió chequear la validez de la Blockchain cada vez que se agrega un elemento. En un primer lugar, se había implementado este método permitiendo que una vez que ya no sea válida la Blockchain, igual se pueda seguir agregando bloques. La idea de la Blockchain es que sea segura, y una vez inválida e insegura, se consideró que ya no se debería poder seguir usando. Una discusión que se tuvo fue si en vez de validar la Blockchain cada vez que se agrega un bloque, que sólo se chequee la validez una vez modificada la Blockchain con el método *modify*. Se llegó a la conclusión de que de todas maneras se seguiría validando en el método *addBlock*, ya que si por alguna razón de error de seguridad se llegase a modificar la Blockchain por otro lado que no es el método *modify*, no lo estaríamos verificando, y no se estarían presentando los estándares de confianza y seguridad que se esperan de una estructura como lo es la Blockchain. Una desventaja es que el método para chequear la validez de la Blockchain es de orden O(n) ya que hay que recorrerla, pero teniendo en cuenta que la Blockchain tiene que ser completamente segura, es un sacrificio que se consideró que era necesario tomar.

**Clase Block:**

La clase Block fue creada para modelar cada bloque de la Blockchain.

Como sabemos, cada bloque en una Blockchain tiene un índice, un nonce, un hash, una referencia al hash del bloque anterior, e información que se quiere almacenar en el bloque.

El índice y el nonce se decidió representar con la clase Integer. Para representar el hash se decidió usar un arreglo de bytes, la explicación del hash se ve más a fondo en la sección “Calculando el hash”.

También, se decidió en cada bloque agregar un campo nuevo, que sería un conjunto de nodos modificados por la operación almacenada en el bloque. El propósito de esto es facilitar la implementación del método *lookup*, que devuelve una lista de los índices de los bloques que modificaron a un nodo en particular.

Por último, se decidió crear la clase BlockData para representar el resto de la información. En esta clase se almacena por un lado la operación realizada sobre el árbol AVL en el bloque actual, y también una copia del árbol AVL al momento de haber agregado el bloque a la Blockchain. La operación realizada se almacena como un String, por ejemplo, “*add 4”*, el propósito de almacenarla como un String está explicado en la sección “Calculando el hash”.

**Calculando el Hash:**

El objetivo de esta sección del informe es explicar cómo se decidió representar y calcular el hash de cada bloque, y explicar los métodos implementados para lograr el minado de un bloque.

En primer lugar, creamos la clase abstracta HashUtilities, que contiene un conjunto de métodos estáticos que permiten realizar validaciones relacionadas al hash y al minado de un bloque. Se decidió utilizar el algoritmo SHA-256, ya que está provisto por la API de Java. Para calcular el hash, se tuvo en cuenta que se necesitaría una representación única de la información del bloque para no obtener hashes repetidos. Se consideró que la mejor manera de obtener esta representación única del bloque era mediante un String, concatenando cada campo del bloque obteniendo un String con todos los campos concatenados. Otra opción podía ser obteniendo un número único que represente al bloque y después aplicarle el hash, pero es igual de útil y más comprensible trabajar con un String, razón por la que se decidió usar eso.

El método provisto por la API de Java para calcular el SHA-256 devuelve un arreglo de bytes. En un primer momento, para calcular si el hash es válido se había decidido convertir ese arreglo de bytes a un String en hexadecimal, y luego hacer comparación con la cantidad de ceros para verificar que sea válido. El problema es que el nonce en numerosos casos llega hasta más de un millón, y por cada una de esas iteraciones se está haciendo la conversión de un arreglo de bytes a String. El algoritmo de minado podría ser más eficiente si en vez de convertir el arreglo de bytes en un String, se trabaja directamente con el arreglo de bytes, que es lo que se decidió hacer. En el método donde se verifican los ceros iniciales del hash, realizamos comparaciones bit a bit para evitar convertir el arreglo de bytes a un String.

**Convenciones:**

El objetivo de esta sección del informe es presentar la convención utilizada para leer el archivo que se le pasa al comando *modify*. Teniendo en cuenta esta convención de formato, el comando *modify* leerá correctamente el archivo y se ejecutará sin problemas.

El formato del archivo es el siguiente (la *x* seria un numero Integer):

nonce: *x*

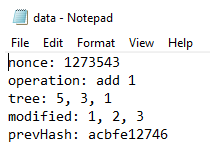
operation: (add|remove) *x*

tree: *x1, x2, x3, … , xn*

modified: x1, x2, x3, … , xn

prevHash: x

Notar que en los números separados por coma, después de cada coma viene un espacio. También, tener en cuenta que a prevHash sólo se le puede pasar un número en hexadecimal. Y en operation, solo se puede indicar o add o remove, pero no las dos.

Un ejemplo de un archivo válido:

**Conclusión:**