Estructura de datos
Y algoritmos

INFORME

TPE 1 EDA

Segundo Cuatrimestre 2017

Integrantes:

Della Sala, Rocío
Giorgi, María Florencia
Heimann, Matías
Rodríguez, Ariel Andrés
Santoflaminio, Alejandro

AVL:

Un árbol AVL es un árbol binario de búsqueda que cumple con una condición particular que lo diferencia de los BST: la diferencia, en valor absoluto, entre las alturas de los subárboles de cada uno de sus nodos es como mucho 1.

Métodos:

Si bien los métodos del AVL son varios, los dividiremos y explicaremos en 2 grupos, “Avl” y “Prueba”.

Métodos Avl:

Los métodos “Avl” que se detallarán en esta sección son:

- *add*
- *remove*
- *balanceTree*
- *lookUp*

Los métodos *add* y *remove*, funcionan de manera casi idéntica a cualquier función de *add* y *remove* de un BTS.

El método *balanceTree*, el cual es llamado en *add* y *remove*, es característico del AVL puesto que revisa si el nodo está desbalanceado y en caso afirmativo, decide qué tipo de rotación debe realizar para su respectivo balanceo

El método *lookUp* hace una búsqueda binaria como en cualquier BTS y retorna el nodo, el cual en su interior posee qué bloques de la blockchain realizaron modificaciones en él.

Métodos Prueba:

Los métodos “Prueba” que se detallarán en esta sección son:

- *isBTS*
- *isAVL*

La función *isBTS* revisa únicamente, que el árbol siga correctamente la definición de un árbol BTS.

La función *isAVL* revisa el balance de todos los nodos, para revisar que el árbol sea ABL.

Nota: En esta sección se detalló a grandes rasgos los métodos más importantes involucrados en el AVL. Cabe aclarar, que los métodos “Prueba” no forman parte del funcionamiento del AVL; sólo se usaron para verificar que el funcionamiento fuera el esperado en main’s externos a la clase.

Problemas encontrados y justificaciones:

Para la realización del método *lookUp*, se debieron tomar 2 decisiones importantes, en las cuales cada una tenía 2 opciones posibles que el grupo pensó que eran las más favorables para la implementación y la eficiencia del AVL. Cada una de las decisiones se detallará a continuación y se justificará por qué se eligió cada una.

Primera justificación:

Tuvimos que decidir entre guardar una lista con las modificaciones al árbol en cada bloque, o guardar una lista en cada nodo con los bloques que lo modificaron.

Se decidió guardar una lista en cada nodo por 2 razones:

La primera, porque el árbol se guardaba una única vez en la Blockchain, de esta manera, ambas opciones imponían un déficit en la complejidad espacial muy parecido.

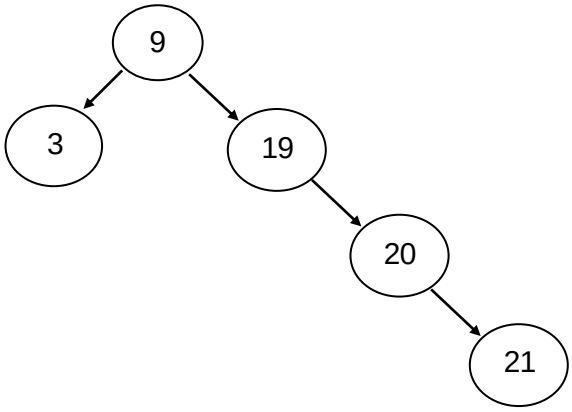
La segunda, y más importante, suponiendo el caso de que se guardaran las modificaciones en cada bloque, la función *lookUp* tendría una peor complejidad temporal. Esto es así, puesto que primero habría que buscar el nodo en el árbol y luego recorrer todos los bloques hasta llegar a aquel que lo agregó. De la otra forma, se debería buscar únicamente buscar el nodo en el árbol y retornar la lista que contiene los bloques que lo modificaron.

Segunda justificación:

Un nodo se ve modificado si es agregado, participa en alguna rotación o si alguno de sus hijos se modifica. Para llevar acabo correctamente ese

seguimiento debimos decidir si cada nodo tenía un campo “padre” o si en la ida de la recursión de *add* y *remove* guardábamos 2 variables “left-son” y “right-son” para que en la vuelta de la recursión se chequee si los hijos eran los mismos.

Como ya se dijo anteriormente, no se encontró una forma para poder decidir que bloques modificaban a cada nodo correctamente.
¿Por qué? Porque, si bien podía definirse correctamente si un nodo participaba en una rotación, si se agregaba o si se cambiaba alguno de sus hijos, había un problema en el caso de las rotaciones.



Suponiendo que es el bloque 5 y se agregó el “21”:
Como vemos en el ejemplo de la izquierda, al hacer la rotación podemos marcar fácilmente que “19”, “20” y “21” forman parte de la rotación, por lo cual se ven modificados por el bloque 5. Pero no hay ninguna forma de avisarle a “9” que su hijo “19” no será el mismo luego de la rotación. La única manera es tener un campo padre que revise eso, o que en la ida de la recursión se guarden los 2 hijos del nodo y en la vuelta se chequee que sean los mismos

La decisión tomada fue que en la ida de la recursión se guarden los 2 hijos y en la vuelta se chequee que sean los mismos.
Si bien esto afecta al orden espacial de los métodos *add* y *remove* puesto que pasan a ser O(Log(N)) (y más específicamente a tener magnitud 2Log(N) en el peor caso) siendo N la cantidad de nodos es mucho mejor para la ejecución del programa en general, en términos de espacio, que tener un campo más por cada nodo que sea un campo padre. Esto queda demostrado en la siguiente tabla de comparación

Casos:	**Punteros adicionales con campo padre**	**Punteros adicionales en la ejecución de add o remove**
Número de nodos:		
16 nodos	16 punteros	4 punteros
64 nodos	64 punteros	6 punteros
1024 nodos	1024 punteros	10 punteros
16384 nodos	16384 punteros	14 punteros
262144 nodos	262144 punteros	18 punteros
4194304 nodos	4194304 punteros	22 punteros

Tabla 1

Como se ve claramente en la Tabla 1, si bien se modifica el orden espacial de los métodos *add* y *remove*, claramente es mejor en términos de eficiencia espacial del programa en general la decisión tomada.

Blockchain:

Una blockchain (cadena de bloques) es una base de datos distribuida que registra bloques de información, entrelazándolos para facilitar la recuperación de la información y la verificación de que ésta no haya sido modificada.

Métodos:

Los métodos que se detallarán en esta sección son:

- *add*
- *remove*
- *lookup*
- *modify*
- *validate*

Los métodos *add* y *remove* modifican el AVL solo si el comando es válido, es decir en el caso de *add* si el elemento a insertar no existe en el AVL y en el caso de *remove* si el elemento existe en el AVL. Tanto en el caso que el comando sea válido como inválido, se genera un nuevo bloque en la blockchain y se detalla el estado de la operación por salida estándar.
El método *lookUp* no modifica al AVL, ni tampoco genera un nuevo bloque en la blockchain, independientemente de la validez del comando. Si el comando es

válido, es decir el elemento a buscar existe en el AVL, retorna todos los índices de bloques que modificaron el nodo. Si el comando es inválido, solo muestra un mensaje indicando que el elemento no existe en el AVL. Para su utilización fue necesario crear la clase List que guarda los índices de bloques que modificaron dicho nodo, para luego iterarla y mostrarlos por salida estándar.

El método *modify* no modifica al AVL, ni tampoco genera un nuevo bloque en la blockchain, independientemente de la validez del comando. Si el comando es válido, modify debe recibir dos argumentos: el primero un índice que indique el número de bloque a modificar y el segundo el path de un archivo. Lo que hace es cambiar la operación con el contenido del archivo e invalida la cadena.

El método *validate* no modifica al AVL, ni tampoco genera un nuevo bloque en la blockchain. Lo que hace esta operación es indicar por salida estándar si la blockchain está "rota". ¿A que me refiero con "rota"? Si la misma fue modificada con un *modify* previamente.

Nota: Nos referimos a comandos inválidos también a aquellos mal escritos, que no reciban el/los parámetro/s adecuado/s para operar o menos o más parámetros de los que necesita. En el video se detallan estos ejemplos con claridad.

Problemas encontrados y justificaciones:

Para la realización de la clase BlockChain, se debieron tomar las siguientes decisiones:

Primera justificación:

Tuvimos que decidir entre guardar el árbol AVL en la blockchain o guardarlo en cada bloque. Se decidió guardar el árbol AVL en la blockchain.

La complejidad espacial que requiere guardar en cada bloque el AVL es mucho mayor a guardarlo una sola vez en la blockchain. Además, si podemos guardar una representación del árbol (hashTree) en cada bloque no se justifica gastar grandes cantidades de memoria para el otro caso.

Por otro lado, existe un "trade-off" entre la complejidad espacial y temporal (si le dedico más espacio a un programa probablemente me lleve menos tiempo y

viceversa). No tener guardado el árbol conlleva generar su representación cada vez que agrego un bloque a la blockchain, pero esto tiene O(N).

Como indico en la sección *Tabla de comparaciones*, la complejidad temporal se ve sumamente afectada por la cantidad de ceros que tiene el hash (y esto es independiente de donde guardo el arbol AVL). Es decir que si hubiéramos guardado en cada bloque el arbol AVL el programa se vería afectado tanto espacial como temporalmente, lo que haría al programa ineficiente e ineficaz.

Haciendo un balance entre la complejidad temporal y la espacial que ambos casos tenían, preferimos darle prioridad a la espacial.

Segunda justificación:

El funcionamiento del método *validate* está basado en la BlockChain Demo de Anders[1] y no en los requerimientos de el enunciado del TPE.

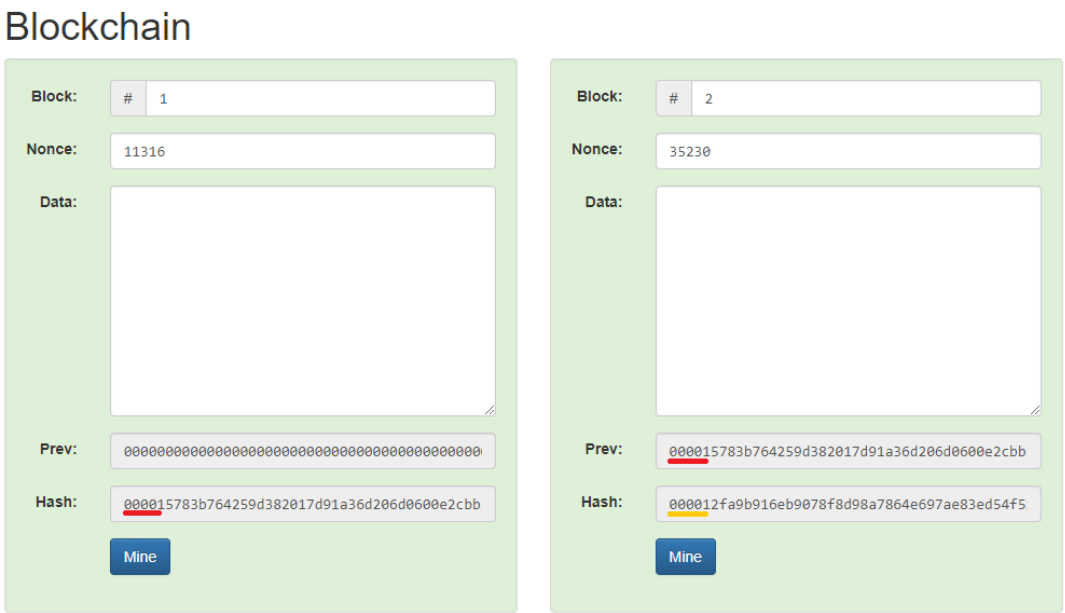


Figura 1

[1] https://anders.com/blockchain/blockchain.html

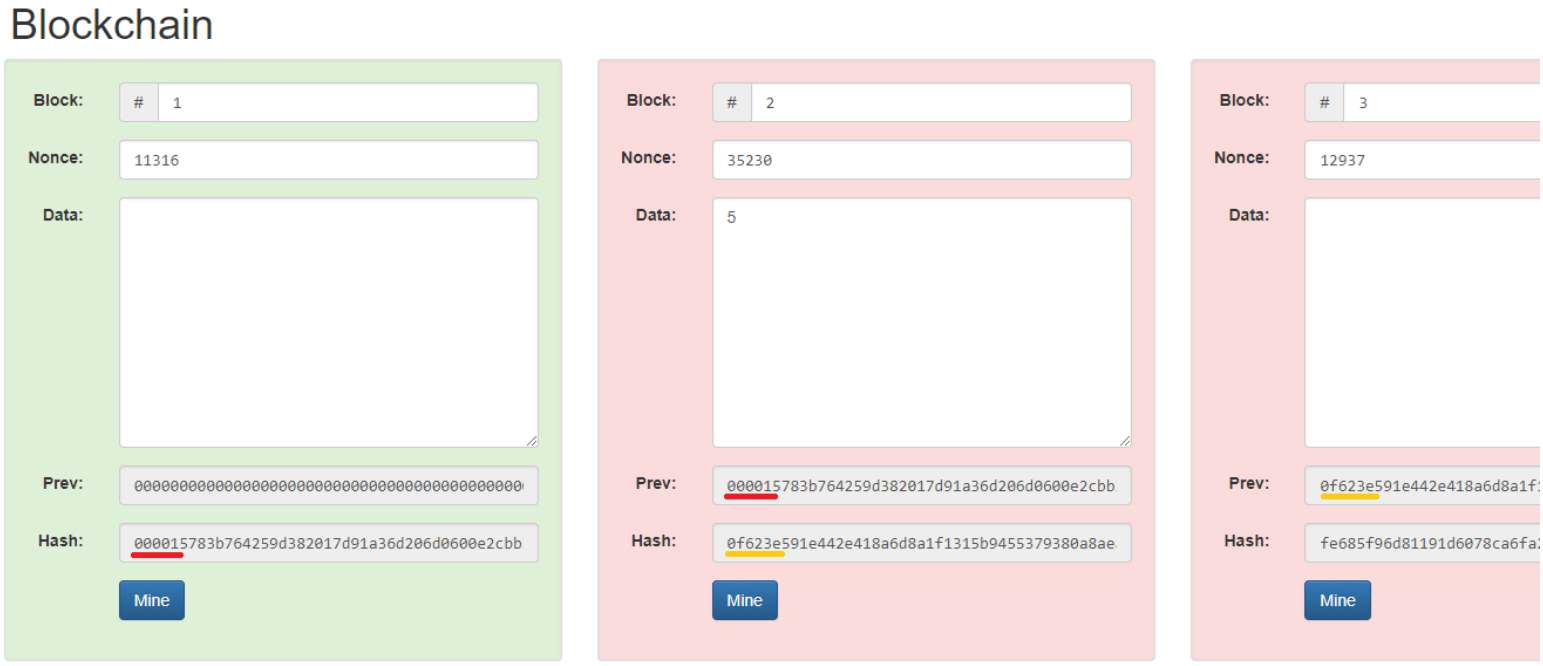


Figura 2

En la Figura 2 podemos ver que al invalidar la blockchain se genera un nuevo hash desde el bloque invalidado hasta el último, sin tener en cuenta los ceros como en la Figura 1. Notar que la referencia al hash previo en cada bloque siempre es correcta.

Para esto tenemos un método recursivo *modifyRecursive* en el cual nos movemos en la blockchain hasta el bloque N pedido, cuando llegamos al mismo modificamos su hash y retornamos, modificando todos los hashes hasta el último bloque.

Tercera justificación:

El método *modify* no modifica el arbol, solo invalida la blockchain y cambia la operación (data) correspondiente al bloque indicado. Tampoco validamos si lo que se levanta de file es una operación o no. Como la blockchain va a quedar invalidada, es decir luego de aplicar *modify* no podemos aplicar ninguna otra operación, y la catedra aclaro que no era un requisito minarla, decidimos que no

era necesario tener un AVL que se corresponda con la blockchain luego de *modify*.

Cuarta justificación:

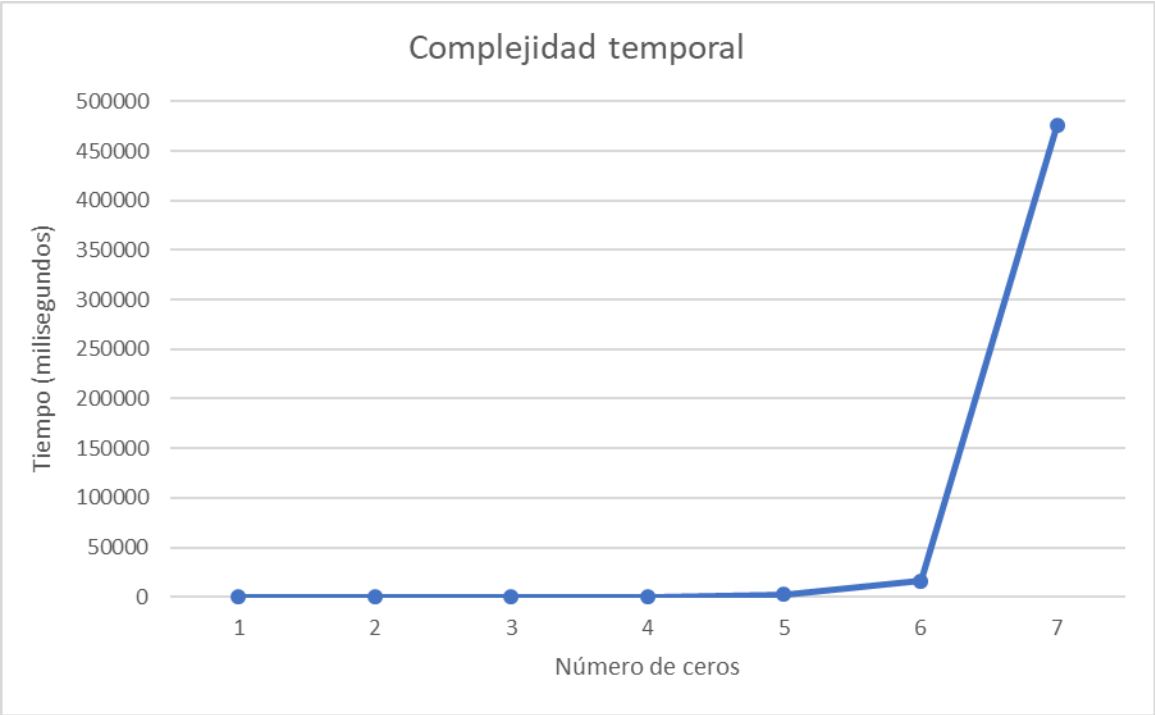
La blockchain posee una referencia al último bloque y no al primero. Como la inserción de bloques en la blockchain siempre se hace al final de la misma, por eso decidimos apuntar al último bloque. Por ende, para movernos por la blockchain era necesario tener en cada bloque una referencia al anterior. Además, por definición de blockchain, cada bloque de la misma hace referencia al hash del bloque anterior y esto indica implícitamente cómo será su estructura.

Tabla de comparaciones:

La siguiente tabla compara el tiempo que tarda en generar un hash para un bloque según la cantidad de ceros:

Cantidad de ceros	Tiempo (milisegundos)					
	Prueba 1	**Prueba 2**	**Prueba 3**	**Prueba 4**	**Prueba 5**	**Promedio**
1	8	14	5	2	2	6.2
2	29	20	8	3	3	12.6
3	55	20	15	19	3	22.4
4	224	155	20	72	36	101.4
5	233	4448	713	6558	423	2475
6	5054	20124	22020	3644	30790	16326.4
7	1045576	126401	691381	283460	229440	475251.6

Tabla 2



Los métodos *add* y *remove* son aquellos que generan un hash para un bloque. Pero no dependen únicamente de esto, sino también del tamaño del arbol AVL. Cabe destacar que sucede con aquellos métodos que no generan un hash para un bloque: sus tiempos son independientes de la cantidad de ceros, pero no son constantes.
Para el caso del método *lookUp* este dependerá solamente del tamaño del AVL y de la cantidad de modificaciones que tenga el elemento a buscar. En el caso del *modify* se genera un hash sin ceros y solo depende del N ingresado (es decir, lo que tarda en recorrer la blockchain). Por último, en el caso de *validate* solo depende del tamaño de la blockchain.

Conclusiones:

Este trabajo nos permitió trabajar con dos estructuras que los integrantes del grupo no estábamos familiarizados.
En el caso del arbol AVL, aunque ya conocíamos cómo funcionaba a nivel teórico, fue la primera vez que implementamos uno. Como la estructura del AVL es la misma que la de un BST decidir cómo iba a ser fue fácil, ya que en el taller de la materia implementamos un BST. La dificultad estuvo en hacer los métodos

add y remove ya que al insertar o remover un nodo, la altura de los subárboles se podía ver afectada. Lo que más costo fue darnos cuenta de los 8 casos distintos que tenemos al remover un nodo.
En el caso de la blockchain, aunque todos coincidimos en haber escuchado alguna vez lo que era (más que nada por su uso en el campo de las criptomonedas) nunca habíamos pensado como implementar una. Este caso fue contrario al AVL, nos costó más pensar la estructura de la blockchain y cómo iba a funcionar con los requisitos pedidos, pero una vez determinados estos aspectos, su implementación fue más sencilla.
Para ambos casos, una vez implementado cada uno, nos concentramos en modularizar su funcionamiento lo máximo posible y trabajar sobre la programación defensiva del programa.
Con respecto a las decisiones detalladas en el informe, cada una fue tomada luego de hacer una puesta en común con el resto de los integrantes sobre las diferentes posibilidades que teníamos para resolver un problema. La idea con la que la mayor parte de los integrantes estaba de acuerdo era la elegida, siempre priorizamos aquellas que más se correspondían con lo aprendido en la materia.