75.04/95.12 Algoritmos y Programación II
Trabajo práctico 1: algoritmos y estructuras de datos

Universidad de Buenos Aires - FIUBA
Segundo cuatrimestre de 2020

1. Objetivos

Ejercitar conceptos relacionados con estructuras de datos, diseño y análisis de algoritmos. Escribir un programa en C++ (y su correspondiente documentación) que resuelva el problema que presentaremos más abajo.

2. Alcance

Este Trabajo Práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá será entregado a través del campus virtual, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe de acuerdo con lo que mencionaremos en la Sección 5, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

4. Descripción

El propósito de este trabajo es continuar explorando los detalles técnicos de Bitcoin y blockchain, tomando como objeto de estudio nuestra versión simplificada de la blockchain introducida en el primer trabajo práctico: la Algochain.

En esta oportunidad, extenderemos el alcance de nuestros desarrollos y operaremos con cadenas de bloques completas. Para ello, nos apoyaremos en un protocolo sencillo que permite abstraer los aspectos técnicos de la Algochain. Al implementar este protocolo, nuestros programas podrán actuar como clientes transaccionales de la Algochain, simplificando la operativa de cara al usuario final.

4.1. Tareas a realizar

A continuación enumeramos las tareas que deberemos llevar a cabo. Cada una de estas será debidamente detallada más adelante:

1. Implementación de una interfaz operativa basada en un protocolo artificial para interactuar con la ALGOCHAIN.
2. Lectura, interpretación y pre-procesamiento de una ALGOCHAIN completa.
3. Nuevo algoritmo de cómputo del campo `txns_hash` basado en árboles de Merkle.

4.1.1. Protocolo operacional

El protocolo con el que trabajaremos consiste en una serie de **comandos** que permiten representar distintas operaciones sobre la ALGOCHAIN. Cada comando recibe una cantidad específica de parámetros, realiza cierta acción y devuelve un resultado al usuario final.

Conceptos preliminares Antes de detallar los comandos, es importante definir algunos conceptos preliminares:

- **Bloque génesis**: al igual que en blockchain, al primer bloque de toda ALGOCHAIN se lo conoce como *bloque génesis*. Esencialmente, este bloque introduce un saldo inicial para un usuario dado. Puesto que no existen bloques anteriores, el campo `prev_block` de un bloque genésis debe indicar un hash nulo (i.e., con todos los bytes en `0`). Este bloque debe también contar con un único *input* y un único *output*. De igual modo, el *input* debe referenciar un *outpoint* nulo, mientras que el *output* hace la asignación del saldo inicial respetando el formato usual.
- **Mempool**: el protocolo de este trabajo permitirá que nuestros programas operen como mineros de la ALGOCHAIN. Emulando el comportamiento de los mineros de Bitcoin, nuestros programas contarán con un espacio en memoria donde se alojarán las transacciones de los usuarios que aún no fueron confirmadas (i.e., que no se agregaron a la ALGOCHAIN). Este espacio se conoce como *mempool*.

Descripción de los comandos

- `init <user> <value> <bits>`

  Descripción. Genera un bloque génesis para inicializar la ALGOCHAIN. El bloque asignará un monto inicial `value` a la dirección del usuario `user`. El bloque deberá minarse con la dificultad `bits` indicada.

  Valor de retorno. El hash del bloque genésis. Observar que es posible realizar múltiples invocaciones a `init` (en tales casos, el programa debe descartar la información de la ALGOCHAIN anterior).

- `transfer <src> <dst1> <value1> ... <dstN> <valueN>`

Descripción. Genera una nueva transacción en la que el usuario `src` transferirá fondos a una cantidad `N` de usuarios (al `i`-ésimo usuario, `dsti`, se le transferirá un monto de `valuei`). Si el usuario origen no cuenta con la cantidad de fondos disponibles solicitada, la transacción debe considerarse inválida y no llevarse a cabo.

Consideraciones adicionales. Recordar que cada *input* de una transacción toma y utiliza la cantidad completa de fondos del *outpoint* correspondiente. En caso de que una de nuestras transacciones no utilice en sus *outputs* el saldo completo recibido en los *inputs*, la implementación de este comando debe generar un *output* adicional con el *vuelto* de la operación. Este vuelto debe asignarse a la dirección del usuario origen.

Valor de retorno. Hash de la transacción en caso de éxito; `FAIL` en caso de falla por invalidez.

- `mine <bits>`

  Descripción. Ensambla y agrega a la ALGOCHAIN un nuevo bloque a partir de todas las transacciones en la *mempool*. La dificultad del minado viene dada por el parámetro `bits`.

  Valor de retorno. Hash del bloque en caso de éxito; `FAIL` en caso de falla por invalidez.

- `balance <user>`

  Descripción. Consulta el saldo disponible en la dirección del usuario `user`. Las transacciones en la *mempool* deben contemplarse para responder esta consulta.

  Valor de retorno. Saldo disponible del usuario.

- `block <id>`

  Descripción. Consulta la información del bloque representado por el hash `id`.

  Valor de retorno. Los campos del bloque siguiendo el formato usual. Debe devolver `FAIL` en caso de recibir un hash inválido.

- `txn <id>`

  Descripción. Consulta la información de la transacción representada por el hash `id`.

  Valor de retorno. Los campos de la transacción siguiendo el formato usual. Debe devolver `FAIL` en caso de recibir un hash inválido.

- `load <filename>`

  Descripción. Lee la ALGOCHAIN serializada en el archivo pasado por parámetro.

  Valor de retorno. Hash del último bloque de la cadena en caso de éxito; `FAIL` en caso de falla por invalidez de algún bloque y/o transacción. Observar que es posible realizar múltiples invocaciones a `load` (en tales casos, el programa debe descartar la información de la ALGOCHAIN anterior).

- `save <filename>`

Descripción. Guarda una copia de la ALGOCHAIN en su estado actual al archivo indicado por el parámetro `filename`. Cada bloque debe serializarse siguiendo el formato usual. Los bloques deben aparecer en orden en el archivo, comenzando desde el génesis.

Valor de retorno. `OK` en caso de éxito; `FAIL` en caso de falla.

4.1.2. Lectura de la Algochain

Tal como se infiere del comando `load`, nuestros programas deberán tener la capacidad de leer e interpretar versiones completas de la ALGOCHAIN. Esto permitirá extender e interactuar con cadenas de bloques arbitrarias, permitiendo entre otras cosas el cruce de información entre grupos distintos.

En resumen, los programas deberán poder recibir una ALGOCHAIN serializada en un archivo de entrada y leer la información bloque a bloque, posiblemente organizando los datos en estructuras convenientes para facilitar las consultas y operaciones posteriores. El formato de entrada sigue los lineamientos detallados en el enunciado del trabajo práctico anterior: una ALGOCHAIN no es otra cosa que una concatenación ordenada de bloques.

4.1.3. Árboles de Merkle y hash de transacciones

Un *árbol de Merkle* [3] es un árbol binario completo en el que los nodos almacenan hashes criptográficos. Dada una secuencia de datos $L_1, \ldots, L_n$ sobre la que se desea obtener un hash, el árbol de Merkle se define computando primero los hashes $h(L_1), \ldots, h(L_n)$ y generando hojas a partir de estos valores. Cada par de hojas consecutivas es a su vez hasheado concatenando los respectivos hashes, lo cual origina un nuevo nodo interno del árbol. Este proceso se repite sucesivamente nivel tras nivel, llegando eventualmente a un único hash que corresponde a la raíz del árbol. Esto se ilustra en la Figura 1.

Una particularidad interesante de un hash basado en árboles de Merkle es que resulta muy eficiente comprobar que un dato dado forma parte del conjunto de datos representado por la raíz del árbol. Esta comprobación requiere computar un número de hashes proporcional al logaritmo del número de datos iniciales (cf. el costo lineal en esquemas secuenciales como el adoptado en el primer trabajo práctico).

Siguiendo los lineamientos del protocolo de Bitcoin, en este trabajo práctico computaremos los hashes de las transacciones de un bloque a partir de un árbol de Merkle. En otras palabras, el campo `txns_hash` del header de un bloque b arbitrario deberá contener el hash SHA256 correspondiente a la raíz del árbol del Merkle construido a partir de la secuencia de transacciones de b.

En caso de que la cantidad de transacciones no pueda agruparse de a pares, la última transacción debe agruparse consigo misma para generar los hashes del nivel superior del árbol. Esta estrategia debe repetirse en cada nivel sucesivo.

Ejemplo de cómputo Supongamos que queremos calcular el árbol de Merkle para una secuencia de tres cadenas de caracteres: $s_1 =$ `árbol`, $s_2 =$ `de` y $s_3 =$ `Merkle`. El cómputo debería seguir los siguientes pasos:

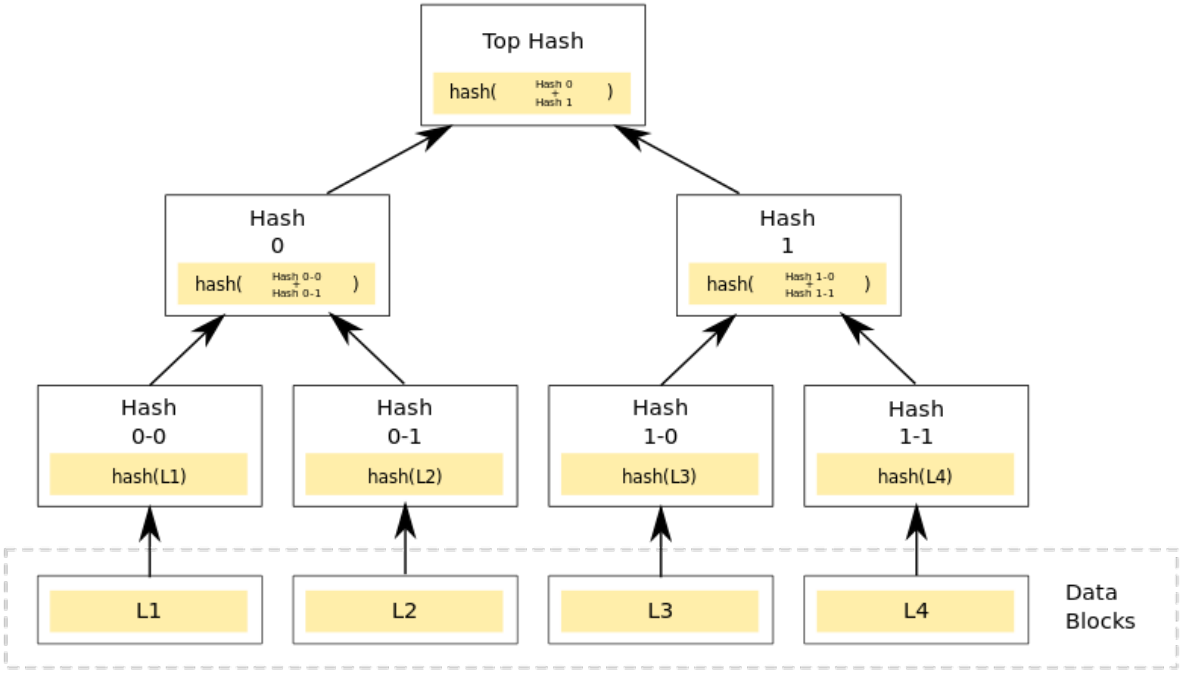

Figura 1: Esquema de un árbol de Merkle (cortesía Wikipedia). El operador +, en este contexto, indica concatenación de hashes y no suma numérica.

1. Para cada s_i, se calcula h_i, un doble hash SHA256 de dicha cadena.
2. Se agrupa h_1 con h_2 y h_3 consigo mismo. Esto da lugar a un nuevo nivel en el árbol formado por dos nuevas cadenas $s_{1,2} = h_1 + h_2$ y $s_{3,3} = h_3 + h_3$ con las respectivas concatenaciones de los hashes del nivel inferior.
3. Se vuelve a repetir el proceso anterior, en esta oportunidad a partir de los hashes $h_{1,2}$ y $h_{3,3}$ de $s_{1,2}$ y $s_{3,3}$, respectivamente.
4. De lo anterior surge un nuevo nivel del árbol con un único nodo, H. Este nodo es la raíz del árbol de Merkle para s_1, s_2 y s_3.

Los hashes anteriores son los siguientes:

- $h_1 =$ `a225a1d1a31ea0d7eca83bcfe582f915539f926526634a4a8e234a072b2cec23`
- $h_2 =$ `b2d04d58d202b5a4a7b74bc06dc86d663127518cfe9888ca0bb0e1a5d51e6f19`
- $h_3 =$ `b96c4732b691beb72b3a8f28c59897bd58f618dbac1c3b0119bcea85ada0212f`
- $h_{1,2} =$ `798f857ba2cdd63f03e22aa5aa52340f10da8fc8b5183dfe989ad366327d36fc`
- $h_{3,3} =$ `af2b866e8ef21130a6ca55776f256a002215e72e99a711978534772af767fbf8`
- $H =$ `abe24c1aeaf6f7358e1702009026c8ad146aa5321e91d36e1928bfc8e6e48896`

4.1.4. Consideraciones adicionales

- Los detalles técnicos de la blockchain y el formato de transacciones y bloques de la ALGOCHAIN fueron deliberadamente omitidos en este enunciado. Sugerimos remitirse al enunciado del primer trabajo práctico para revisar preventivamente todos estos conceptos.
- El cálculo de hashes SHA256 puede realizarse mediante la misma librería provista por la cátedra en la instancia anterior.
- Es importante remarcar que toda estructura de datos (e.g., listas, arreglos dinámicos, pilas o árboles) **debe ser implementada**. La única excepción permitida son las tablas de hash. En caso de necesitar utilizarlas, sugerimos revisar la clase `std::unordered_map` de la STL de C++ [4].

4.2. Interfaz de línea de comandos

Al igual que en el primer trabajo práctico, la interacción con nuestros programas se dará a través de la línea de comandos. Las opciones a implementar en este caso son las siguientes:

- `-i`, o `--input`, que permite controlar el stream de entrada de los comandos del protocolo detallado en la Sección 4.1.1. Si este argumento es "`-`", el programa deberá recibir los comandos por la entrada standard, `std::cin`. En otro caso, el argumento indicará el archivo de entrada conteniendo dichos comandos. Puede asumirse que cada comando aparece en una única línea dedicada.
- `-o`, o `--output`, que permite direccionar las respuestas del procesamiento de los comandos a un stream de salida. Si este argumento es "`-`", el programa deberá mostrar las respuestas de los comandos por la salida standard, `std::cout`. En otro caso, el argumento indicará el archivo de salida donde deberán guardarse estas respuestas.

4.3. Ejemplos

En lo que sigue mostraremos algunos ejemplos que ilustran el comportamiento básico del programa ante algunas entradas simples. Tener en cuenta las siguientes consideraciones:

- Los hashes mostrados podrían no coincidir con los computados por otras implementaciones, puesto que dependen entre otras cosas de la elección de los `nonces` al momento de minar los bloques.
- Al igual que en los ejemplos del trabajo práctico anterior, por conveniencia resumiremos algunos hashes con sus últimos 8 bytes. Las entradas y salidas de nuestros programas deben, naturalmente, trabajar con los hashes completos.

4.3.1. Ejemplo trivial: entrada vacía

Si no hay comandos para procesar (i.e., el stream de entrada es vacío), el programa no debe realizar ninguna acción:

```
$ ./tp1 -i /dev/null -o output.txt
$ cat output.txt
$
```

4.3.2. Múltiples `inits`

En los ejemplos subsiguientes, por claridad resaltaremos los comandos de entrada en color azul y con un símbolo > al comienzo.

En la invocación que se muestra más abajo, inicializamos primero una nueva cadena en la que el usuario `satoshi` dispone de 100 unidades de dinero. El correspondiente bloque génesis es minado con una dificultad de 10 bits. Luego de esto, reseteamos la cadena asignándole al usuario `lucas` una unidad de dinero:

```
$ ./tp1
> init satoshi 100 10
b983fdeb9cbe9426cc1df0ef057b44e583e5ea7531c90376eba518ecfb08a246
> balance satoshi
100.0
> balance lucas
0
> init lucas 1 10
b40495bf172be3c172a41a85f72d13e8b2e8e7e582fc7e14b05e614408b52667
> balance satoshi
0
> balance lucas
1.0
> block fb08a246
FAIL
> block 08b52667
0000000000000000000000000000000000000000000000000000000000000000
647bbe505403dca7a11d08269d02017c72eb0fc2e4398befe41cea620570e639
10
2535
1
1
00000000 0 00000000
1
1.0 f82e82dac113d37a21e2b3e0c37eab9e6fbc3657a38b0a8397d913abedab7605
$
```

Se observa lo siguiente:

- Luego del segundo `init`, los balances de los usuarios cambian. Puesto que este comando genera nuevas instancias de la ALGOCHAIN, es razonable que esto suceda.
- Al pedir el bloque con hash `fb08a246`, vemos que el programa informa una falla. Esta falla proviene de un hash de bloque inválido en la cadena actual: notar que dicho hash corresponde al bloque génesis de la primera cadena.

- El último comando solicita la información del bloque cuyo hash es `08b52667`. En este caso, dicho hash coincide con el del nuevo bloque génesis, por lo que la operación es ahora exitosa.

Podemos también realizar una operatoria en modo *batch* copiando todos estos comandos en un archivo e invocando luego al programa con este archivo como entrada:

```
$ cat commands.txt
init satoshi 100 10
balance satoshi
balance lucas
init lucas 1 10
balance satoshi
balance lucas
block fb08a246
block 08b52667
$ ./tp1 -i commands.txt -o output.txt
$ cat output.txt
b983fdeb9cbe9426cc1df0ef057b44e583e5ea7531c90376eba518ecfb08a246
100.0
0
b40495bf172be3c172a41a85f72d13e8b2e8e7e582fc7e14b05e614408b52667
0
1.0
FAIL
0000000000000000000000000000000000000000000000000000000000000000
647bbe505403dca7a11d08269d02017c72eb0fc2e4398befe41cea620570e639
10
2535
1
1
00000000 0 00000000
1
1.0 f82e82dac113d37a21e2b3e0c37eab9e6fbc3657a38b0a8397d913abedab7605
$
```

Prestar especial atención a la última línea de la salida: como todo *output*, debe finalizar con un caracter de salto de línea (sugerimos remitirse al formato de transacciones y bloques detallado en el enunciado del primer trabajo práctico en caso de dudas).

4.3.3. Transferencias

El próximo ejemplo utiliza el comando `transfer` para generar transacciones y mover dinero entre distintos usuarios:

```
$ ./tp1
> init satoshi 100 10
```

```
b983fdeb9cbe9426cc1df0ef057b44e583e5ea7531c90376eba518ecfb08a246
> transfer satoshi lucio 90
4ab0d8a4fdab846e9f28c1850fe06a73b446341ba7eab2cab8eae9948597e1e1
> transfer satoshi lucas 1
0a7e61b9b17c7e7e21aef8d5e65e3b036e949c7f398bd0692b5b704cf04e9b84
> balance lucio
90.0
> mine 10
5d4075e53f5cb51da5ffb3e68eef18046fc8c1327c4c4f787550b2e94e013806
> balance satoshi
9.0
> balance lucio
90.0
> balance lucas
1.0
> transaction f04e9b84
1
8597e1e1 1 ea55eb5c
2
1.0 f82e82dac113d37a21e2b3e0c37eab9e6fbc3657a38b0a8397d913abedab7605
9.0 5fe3f3a6faaef93165aff8d88e701f965b8b956ea77e3116c8c8b2cfea55eb5c

$
```

Es importante destacar lo siguiente:

- La primera invocación de `transfer` consume el UTXO del usuario `satoshi` en el bloque génesis. La transacción generada deposita 90 unidades de dinero en la dirección del usuario `lucio` y un vuelto de 10 unidades de dinero en la dirección de `satoshi`. El hash de esta transacción es `8597e1e1`.
- La segunda invocación de `transfer` debe, necesariamente, consumir el UTXO de `satoshi` correspondiente a la transacción anterior (observar que el primer *output* en el bloque génesis ya fue consumido y no puede volver a utilizarse). Esta vez, se generará una nueva transacción que deposita una unidad de dinero en la dirección de `lucas` y un vuelto de 9 unidades de dinero en la dirección de `satoshi`.
- La primera invocación de `balance` nos dice que `lucio` tiene 90 unidades de dinero disponibles. Este dinero está sujeto a ser confirmado puesto que la transacción todavía se encuentra en la *mempool*.
- Luego de minar el nuevo bloque a partir de las transacciones anteriores, el saldo de `lucio` aparece confirmado con la misma cantidad de dinero. Por otro lado, `satoshi` tiene un saldo de 9 unidades de dinero, mientras que `lucas` sólo dispone de una unidad de dinero.
- Por último, el comando `transaction` solicita información sobre la transacción con hash `f04e9b84`. Vemos que este hash corresponde a la transacción derivada del segundo uso de `transfer`. Allí puede verse el vuelto de 9 unidades de dinero a la dirección de satoshi (el segundo *output* de dicha transacción).

4.3.4. Lectura y escritura de cadenas

Finalmente, veamos cómo leer y escribir cadenas completas con nuestros programas. El siguiente ejemplo guarda una cadena de dos bloques al archivo `algochain.txt`:

```
$ ./tp1
> init satoshi 100 10
b983fdeb9cbe9426cc1df0ef057b44e583e5ea7531c90376eba518ecfb08a246
> transfer satoshi lucas 1 lucio 90
8b58f15e5c4408b30322daca6d14edd44ff3d067d8b1ea967dff89d5705f5ff3
> mine 10
3b44b8c5182097fa63c2e84aa27735f8cad40971c84266fb874d0bd993c15315
> save algochain.txt
OK
$
```

Notar que, esta vez, el comando `transfer` incluye múltiples destinatarios: la transacción depositará una unidad de dinero en la dirección de `lucas` y 90 unidades de dinero en la de `lucio` (esto se lleva a cabo definiendo dos *outputs* diferentes). Puesto que el saldo de `satoshi` consumido por la transacción es de 100 unidades de dinero, el vuelto que le corresponde es de 9 unidades.

En esta invocación posterior, cargamos la cadena anterior a partir del archivo generado. Observar que la información de la cadena inicial (la generada vía `init`) se descarta:

```
$ ./tp1
> init satoshi 100 10
b983fdeb9cbe9426cc1df0ef057b44e583e5ea7531c90376eba518ecfb08a246
> load algochain.txt
3b44b8c5182097fa63c2e84aa27735f8cad40971c84266fb874d0bd993c15315
> balance satoshi
9.0
> balance lucio
90.0
> balance lucas
1.0
$
```

4.4. Portabilidad

Es deseable que la implementación desarrollada provea un grado mínimo de portabilidad. Sugerimos verificar nuestros programas en alguna versión reciente de UNIX: BSD o Linux.

5. Informe

El informe deberá incluir, como mínimo:

- Una carátula que incluya los nombres de los integrantes y el listado de todas las entregas realizadas hasta ese momento, con sus respectivas fechas.
- Documentación relevante al diseño e implementación del programa.
- Documentación relevante a los algoritmos y estructuras de datos involucrados en la solución del trabajo.
- El análisis de las complejidades solicitado en la sección 4.
- Documentación relevante al proceso de compilación: cómo obtener el ejecutable a partir de los archivos fuente.
- Las corridas de prueba, con los comentarios pertinentes.
- El código fuente, en lenguaje C++.
- Este enunciado.

6. Fechas

La última fecha de entrega es el **jueves 3 de diciembre de 2020**.

Referencias

[1] Wikipedia, "Bitcoin Wiki." `https://en.bitcoin.it/wiki/Main_Page`.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009.

[3] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the theory and application of cryptographic techniques*, pp. 369–378, Springer, 1987.

[4] cplusplus.com, "Unordered Map." `https://www.cplusplus.com/reference/unordered_map/unordered_map/`.