Universidad de Buenos Aires
Facultad de Ingeniería
Año 2020 - 2.^{do} cuatrimestre

95.12 Algoritmos y Programación II
Trabajo práctico N.º 1 - Algoritmos y estructuras de datos

Estudiantes:

Ochagavia, Lara	100637
lochagavia@fi.uba.ar	
Pintos, Gastón Maximiliano	99711
gmpintos@fi.uba.ar	
Ferreyra, Lucas Ariel	97957
lferreyra@fi.uba.ar	

Profesores:
CALVO, Patricia Mabel
SANTI, Lucio
SANTI, Leandro

Fecha de entrega: 3 de Diciembre del año 2020

Índice

1. Objetivo del proyecto

El objetivo del siguiente trabajo práctico es desarrollar un programa que sea capaz de generar una cadena *Algochain*, con la cual se podrá interactuar, como *stream* de salida a partir de comandos ingresados como *stream* de entrada, utilizando como herramienta el lenguaje C++. Tanto el *stream* de entrada como el de salida son identificados por el usuario por linea de comando de argumento al momento de ejecutar el programa.

El siguiente informe detalla las tareas a realizar por los alumnos con una descripción de los comandos utilizados para interactuar con la *Algochain* permitiendo una mayor comprensión del trabajo. Están detalladas en el informe las estrategias de programación utilizadas como así también el uso de distintas herramientas propias del lenguaje, como son los *Templates* y el paradigma de programación orientada a objetos. Por último, se presentan las corridas de prueba del proyecto junto con las conclusiones y observaciones desarrolladas durante la realización del mismo.

2. Descripción del proyecto

El presente trabajo práctico refiere al uso de una estructura denominada *Algochain*. Dicha estructura se basa en la lista enlazada de bloques que contienen información sobre transacciones de criptomonedas *bitcoin* denominada *Blockchain*. La *Algochain* se compone de bloques que agrupan transacciones. A su vez, las transacciones constan de una secuencia de *inputs* y otra de *outputs*.

El programa que se desarrolla para este trabajo se basa en un protocolo que permite abstraer conceptos de la llamada *Algochain* para que dicho programa pueda actuar como cliente transaccional de esta cadena de bloques. El usuario dispone de una serie de comandos para interactuar con la cadena de bloques. Cada uno realiza una tarea especifica que afecta la conformación de las estructuras que forman la *Algochain*. Dependiendo de lo indicado en la linea de comandos al momento de la ejecución, el usuario tiene la posibilidad de ingresar la serie de comandos en un archivo de entrada o bien, ingresar los comandos de forma interactiva.

Para los campos de identificación de cada *input* y *output* de cada transacción dentro de cada bloque son utilizadas funciones de *hash* criptográficas que identifican unívocamente a las partes involucradas por medio de claves. Para el desarrollo de este trabajo practico se adopta la función *SHA256* provista por la cátedra para generar dichos *hashes*. Se realizaran *hashes* también a las transacciones, a los bloques y a determinados campos dentro de estas últimas estructuras mencionadas. Para el campo *txns_hash* del *header* del bloque se utiliza un árbol de *Merkle* para poder calcular el *hash* de las transacciones del bloque. Se presenta en la figura 2.1 la estructura de la cadena *Algochain* para la mejor comprensión de los campos mencionados en este apartado.

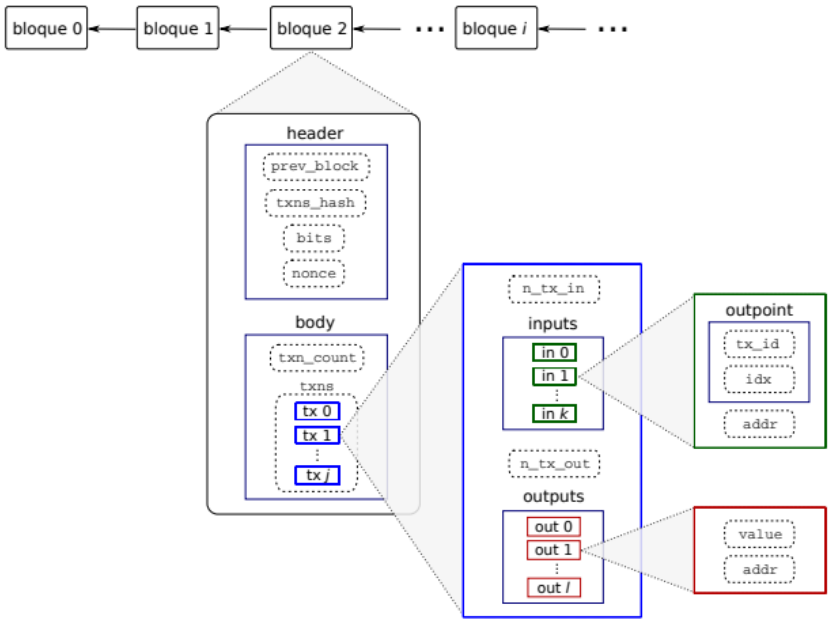

Figura 2.1: Esquema de la estructura Algochain.

3. Desarrollo

3.1. Consideraciones previas:

El programa debe de cumplir con el paradigma de programación orientado a objetos. Toda la información tratada a lo largo del programa se maneja a través de objetos pertenecientes a clases. La definición de clases, y la instanciación de objetos, permite manejar la información de manera prolija, eficiente y segura al cumplir con el ocultamiento y encapsulamiento de la información perteneciente a cada objeto.

Para los campos cuya información era resultado de una función de *hash* se implementa el uso de cadenas de caracteres definidas en la clase *string*. Esto es necesario ya que cada *hash* cuenta con 64 caracteres, cada uno representando un número hexadecimal, por lo que el manejo de esta información era mas simple en formato de *string*. El alcance de la definición de un número entero no es suficiente para definir un *hash* y era necesaria la implementación de modificadores y nuevas estructuras que hubieran dificultado el manejo del código.

Los datos que conforman a cada estructura de la *Algochain* respetan lo enunciado en el trabajo práctico anterior (figura 2.1).

Los errores detectados, de haberlos, se informan por el flujo de errores estándar *cerr*. Los mensajes de error especifican el campo de información corrupta o el tipo de error detectado.

La invocación del programa es por linea de argumentos de comando según el siguiente formato: se utilizan los nombres largos de cada opción y se especifican los archivos para el flujo de entrada y salida, se utilizan los nombres cortos de las opciones con el valor por defecto de los flujos de entrada y salida o bien una combinación de todo lo mencionado previamente. Si la opción seleccionada es que el flujo de entrada sea un archivo de texto, este archivo debe contener los comandos para interactuar con la *Algochain*. Si el flujo de entrada seleccionado para la ejecución del programa es el estándar, se ingresan por consola de manera interactiva los comandos deseados.

```
./programa --input Archivo.txt --output Archivo.txt

./programa -i - -o -

./programa --input Archivo.txt -o -
```

3.2. Representación de datos en la ALGOCHAIN

Las estructuras de datos implementadas para poder procesar los datos y almacenarlos de manera que se conforme la cadena de la figura 2.1 fueron descritas en el trabajo practico anterior.

Para poder desarrollar la ***Algochain*** se utiliza un contenedor de datos implementado con la clase ***List*** que contiene en sus nodos bloques de la clase ***Blocks***. Esta lista es doblemente enlazada, y dentro de su definición cuenta con la clase ***Iterator*** que permite recorrer y acceder de forma dinámica a los elementos de la lista. La lista de bloques *Algochain* fue declarada como variable global a todas las funciones para que estas puedan acceder y modificarla según su funcionamiento.

A su vez, se declara otra lista global denominada ***mempool***, compuesta por nodos que contienen como información las transacciones, objetos de la clase ***Transaction*** , generadas por el usuario. Una vez que se mine el bloque, el contenido de la lista *mempool* es utilizado para generar el vector de transacciones del bloque que se inserta en la *Algochain* en caso de ser válido.

Se utilizan dos listas globales para almacenar los *hashes* que identifican a los bloques y a las transacciones generadas en el programa. Con esto se busca tener una referencia de la existencia de los bloques presentes de la *Algochain* y las transacciones contenidas en la *mempool* y las ya confirmadas en la cadena. Los *hashes* de transacciones son almacenados en la lista ***hashes_transactions***. Los hashes de los bloques son almacenadas en la lista ***hashes_blocks***.

Con el fin de calcular el campo *txns_hash* se implementa una clase denominada ***MerkleTree***. Esta clase desarrolla un árbol binario cuyas hojas se conforman de los *hashes* de cada transacción presente en cada bloque y su raíz es la función de *hash* de la concatenación de sus hijos. La clase *MerkleTree* cuenta con un puntero a la clase ***TreeNode*** como atributo, con un método constructor que recibe un vector de los *hashes* que funcionan como hojas del árbol y un método destructor que invoca a un método privado encargado de liberar la memoria correspondiente y eliminar todos los nodos del árbol de manera recursiva. Se ve representada la estructura del árbol en la figura a continuación:

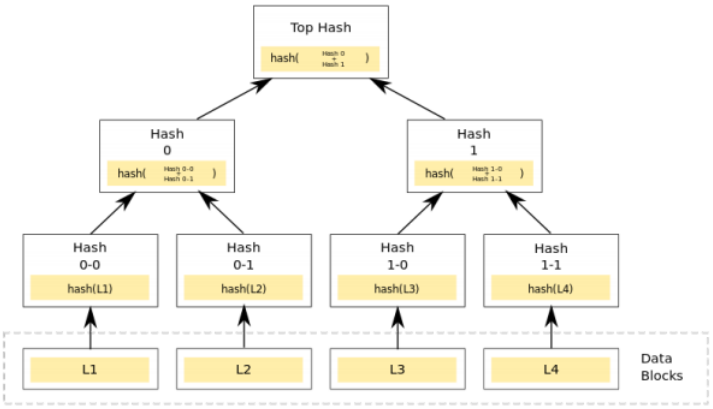

Figura 3.1: Esquema de un árbol de Merkle. Imagen obtenida de Wikipedia

El desarrollo del programa para este trabajo implica realizar múltiples validaciones sobre los datos que conforman a la *Algochain*. Una de las principales validaciones a tener en cuenta es la validez de las transacciones. Para ello, se crea un diccionario, utilizando el contenedor *unordered_map* de la clase estándar del lenguaje c++ . Con este diccionario, al que se llamó ***wallet***, se asocia un cierto valor a una clave. En el programa implementado la clave es el *hash* de cada usuario y el valor asociado a dicha clave es un vector de tres elementos de la clase ***Vector*** que contiene el saldo total del usuario en cuestión, el *hash* de la última transacción en la que estuvo involucrado y por último, la posición en la que apareció como *output* en esa transacción. Al tener esta información asociada a cada usuario presente en la *Algochain* se permite validar si una transacción es válida por distintos aspectos: Si el usuario que quiere transferir existe y, de existir, si la cantidad que desea transferir es menor o igual a los fondos totales con los que cuenta el usuario.

La implementación de la *wallet* se realizó de esta manera porque se interpretó que todo usuario podía hacer uso del total sus fondos sin hacer referencia a cada transacción en la que recibió la suma correspondiente al monto que desee transferir, sino que alcanzaba con sólo hacer referencia a la última transacción en la que estuvo involucrado. En otras palabras, se interpretó que cada transferencia contaría con un único *input* en el cual se indique la información que contienen los últimos dos campos del vector de la *wallet* asociado al usuario que está realizando la transferencia.

3.3. Comandos:

El usuario interactúa con el programa a partir del uso de comandos. Para procesar la entrada de estos comandos se implementa la clase ***Command***, la cual cuenta con un método constructor que recibe una arreglo de estructuras en donde se especifican cada uno de los comandos y el puntero a función asociado, y una función denominada *Command::parse()* que se encarga de configurar el programa para ejecutar el comando ingresado a partir de su puntero a función.

La función *Command::parse()* guarda toda la línea ingresada por el flujo de entrada como un *string* al cual separa en dos partes: se queda con todos los caracteres de esa línea hasta encontrar el delimitador especificado como un espacio, y los guarda como un nuevo *string* llamado *line_name* y el resto de la cadena de caracteres original se guarda en una nueva cadena auxiliar llamada *line_args* donde se supone quedan guardados los parámetros específicos a cada comando. En caso de coincidir la cadena *line_name* con alguno de los comandos indicados en un arreglo definido al comienzo de la función *Command::parse()*, se invoca a la función correspondiente a dicho comando y se le pasa como argumento *line_args*.

Todas las funciones de comandos reciben como argumento una variable de tipo *string* pasada por referencia, en la que se encuentran todos los parámetros necesarios para su funcionamiento. Dentro de cada función se invoca a las funciones de validación correspondientes para los parámetros que se esperan recibir por parte del usuario. En caso de que la función necesite separar los parámetros incluidos en la cadena única recibida por argumento, se invoca a una función que se encarga de separar cada parámetro en cadenas de caracteres separadas y almacenarlas en un arreglo, para que luego la validación de cada parámetro resulte más cómoda.

A continuación se presentan todos los comandos que dispone el usuario junto con una breve descripción de como son implementados y sus resultados en el programa:

3.3.1. Comando init:

$$init \quad \langle user \rangle \quad \langle value \rangle \quad \langle bits \rangle$$

Este comando es utilizado por el usuario para generar el Primer bloque de la *Algochain* llamado Génesis. La función realiza la validación de los parámetros al obtenerlos a partir de la cadena de caracteres recibida por referencia con la función *split_string()*. En caso de haber una cadena de bloques previa, se elimina toda la información contenida en las variables globales y se genera el bloque con su información validada. Este bloque tiene tres particularidades que lo identifican: su campo *prev_block* esta formado por un *hash* nulo y dentro de su única transacción, los campos del input son todos nulos. Para instanciar un bloque de estas características se creo el método *Block::load_Genesis()*. Una vez creado el bloque Génesis, se inserta en la *Algochain* como primer elemento. También se calcula el *hash* del bloque con el método *Block::hash_Block()* y se inserta en la lista *hashes_blocks*. Por ultimo, esta función devuelve por *stream* de salida el *hash* del bloque Génesis generado en caso de no encontrar fallas.

3.3.2. Comando transfer:

$$transfer \quad \langle src \rangle \quad \langle dst1 \rangle \quad \langle value1 \rangle \quad ... \quad \langle dstN \rangle \quad \langle valueN \rangle$$

Este comando es utilizado por el usuario para generar una transacción a ser insertada en la *mempool*. En la función *command_transfer()* se realiza la validación de los parámetros obtenidos de la cadena de caracteres pasada por referencia. Se verifica si el usuario identificado como *source* se encuentra en la *wallet*, si se lo encuentra se determina su saldo y se comprueba si tiene los fondos suficientes para poder realizar la transacción. En caso de no cumplirse alguna de estas premisas, se declara como inválida la transacción y se pasa a leer el siguiente comando. Si ambas son válidas, con esta información se completan los campos de un objeto *Transaction* con el método *Transaction::load_transaction_for_mempool()*. Se actualiza la *wallet* con esta nueva transacción, y se inserta la transacción como dato de un nuevo nodo en la *mempool*. Se realiza el *hash* de la transacción con el método *Trasaction::hash_transaction()* y se lo inserta en la lista *hashes_transactions*. Por ultimo, esta función imprime por *stream* de salida el *hash* de la transacción generada en caso de no encontrar fallas.

3.3.3. Comando mine:

$$mine \quad \langle bits \rangle$$

Este comando es utilizado por el usuario para generar un nuevo bloque utilizando todas las transacciones que se encuentren en la *mempool* según la dificultad de minado indicada. Para realizar esta tarea primero se verifica que la lista *mempool* no se encuentre vacía ya que no seria posible realizar un bloque sin transacciones. Tampoco sería válido hacerlo si el primer elemento de la lista *Algochain* no es el bloque Génesis o bien si esta lista se encuentra vacía, por lo que se realiza esta validación. Luego se instancia un elemento de la clase *Block* y se completan sus campos utilizando las transacciones contenidas en la *mempool* con el método *Block::load_block()* que recibe la dificultad de minado como argumento. Una vez generado el bloque, se inserta en la ultima posición de la *Algochain*, se inserta el *hash* del bloque generado con el método *Block::hash_Block()* y se borra todo el contenido de la *mempool*. Por ultimo, esta función devuelve por *stream* de salida el *hash* del bloque generado en caso de no encontrar fallas.

3.3.4. Comando balance:

$$balance \quad \langle user \rangle$$

Este comando es utilizado por el usuario para imprimir por el *stream* de salida el saldo total (actual) de un usuario de la *Algochain*. Para realizar esta tarea se busca la ocurrencia del usuario en la *wallet* según el *hash* de su nombre. En caso de encontrar al usuario en la *wallet* se imprime su saldo actual. En caso contrario se imprime un "0".

3.3.5. Comando txn:

$$txn \quad \langle id \rangle$$

Este comando es utilizado por el usuario para imprimir por *stream* de salida la transacción deseada a partir de su *hash*. Para realizar esta tarea primero se verifica que el *hash* ingresado como argumento este contenido dentro de la lista *hashes_transactions*. En caso contrario, esta transacción no se encuentra contenida en la *Algochain* y se informa por el *stream* de salida de errores. Luego, se utiliza un objeto de la clase *Iterator* para recorrer cada bloque la lista *Algochain* realizando el *hash* de cada transferencia contenida en esos bloques hasta encontrar una coincidencia con la pasada como parámetro. Si no se encontró en la *Algochain*, se recorre la *mempool* de la misma manera, realizando el *hash* a cada transacción contenida hasta hallar una coincidencia. En caso de hallar una coincidencia en alguna iteración se imprime la transacción por *stream* de salida. En caso contrario se imprime un mensaje de error al usuario.

3.3.6. Comando block:

$$block \quad \langle id \rangle$$

Este comando es utilizado por el usuario para imprimir por *stream* de salida el bloque deseado a partir del parámetro ingresado correspondiente al *hash* de un bloque. Para realizar esta tarea primero se verifica que el *hash* ingresado como argumento este contenido dentro de la lista *hashes_blocks*. En caso de no estarlo, este bloque no se encuentra contenido en la *Algochain*, se imprime un mensaje por el *stream* de salida de errores indicando que no se lo encontró y se procede a leer el próximo comando del *stream* de entrada. En caso de encontrarlo, se utiliza la clase *Iterator* para recorrer cada bloque la lista *Algochain* realizando el *hash* de cada bloque hasta encontrar una coincidencia. En caso de hallar una coincidencia en se imprime el bloque por *stream* de salida.

3.3.7. Comando load:

$$load \quad \langle filename \rangle$$

Este comando es utilizado por el usuario para ingresar una *Algchain* serializada en un archivo de texto con el fin de cargarla al programa. Para realizar esta tarea se abre el archivo pasado por argumento en formato de lectura. Se eliminan los datos contenidos en las variables globales si no hubo errores previos. Luego se utiliza el operador sobrecargado *Block::operator>>()* para completar los campos del bloque a partir de la lectura del archivo como flujo de entrada, sometiendo a todos los campos a la propia validación. Una vez cargado el bloque, se inserta en la ultima posición de la *Algochain* y se inserta el *hash* del bloque generado con el método *Block::hash_Block()*. Esta tarea se realiza hasta detectar el fin del archivo. Se valida que el primer bloque en la *Algochain* debe ser el Génesis. Por ultimo, esta función imprime por *stream* de salida el *hash* del último bloque generado en caso de no encontrar fallas.

3.3.8. Comando save:

$$save \quad \langle filename \rangle$$

Este comando es utilizado por el usuario para imprimir todo el contenido de la *Algochain* en un archivo especificado por parámetro. Se abre el archivo en modo escritura y se imprime cada bloque de la lista recorriendo iterativamente todos sus elementos. Esta función imprime por flujo de salida el mensaje 'OK' en caso en caso de no encontrar fallas.

3.4. Implementación del programa:

El programa funciona de la siguiente manera: La función principal denominada *main()* recibe como parámetros la linea de comandos ingresada por el usuario. Lo primero que se realiza es la instanciación de un objeto *cmdlne* para poder procesar las opciones con la función *cmdline::parse()*. Dentro del programa se utilizan variables globales en donde se almacenan los flujos de entrada y salida que fueron seleccionados según las opciones ingresadas.

Una vez inicializados los flujos de entrada y salida que serán utilizados en el programa, se procede a instancias un objeto de la clase *Command* para poder procesar los comandos con la función *Command::parse()*. El programa está configurado para poder recibir un archivo de entrada con cada comando a realizar o bien, para recibir cada comando de a uno por vez a medida que se ingresan por la terminal o consola. Para que este último caso sea posible, se realiza un ciclo *while* que itera infinitamente hasta que se ingrese el caracter de *EOF*, que en *LINUX* se ingresa como 'Ctrl+d'.

A medida que se ingresan los comandos, las funciones correspondientes se ejecutan para poder efectuar cada tarea específica.

3.5. Compilación del programa:

Para compilar el proyecto se utiliza la herramienta *make* para lo cual se genera un archivo *Makefile*. El compilador utilizado es el g++ en su versión 9.3.0. Los *flags* especificados en la línea de compilación son: -g, -Wall, -ansi y -std=c++11.

En el directorio */TP1_Grupo14* donde se encuentra el archivo *Makefile*, se encuentra la carpeta *src* que contiene todos los archivos .h y .cc utilizados en el proyecto. Estos archivos son:

- main.cc
- Block.cc
- Transaction.cc
- Command.cc
- Utilities.cc
- MerkleTree.cc
- cmdline.cc
- main.h
- Block.h
- Transaction.h
- Command.h
- Utilities.h
- MerkleTree.h
- cmdline.h
- List.h
- TreeNode.h
- sha256.cc
- Vector.h
- sha256.h

Para compilar, se ejecuta la herramienta *make* desde el directorio */TP1_Grupo14*. El nombre del ejecutable será TP1_G14. Se muestra a continuación el archivo *Makefile* utilizado

3.5.1. Makefile

```
1
2  #CC specifies which compiler we're using
3  CC = g++
4
5  #COMPILER_FLAGS specifies the additional compilation options we're using
6  # -w suppresses all warnings
7  COMPILER_FLAGS = -g -Wall -ansi -std=c++11
8
9  #This is the target that compiles our executable
10
11 SRC_DIR := src
12 OBJ_NAME := TP1_G14
13
14 all :
15     $(CC) $(wildcard $(SRC_DIR)/*.cc) $(COMPILER_FLAGS) -o $(OBJ_NAME)
```

3.6. Ejecución del programa

Para ejecutar el programa, el usuario debe situarse en la carpeta donde se encuentre el ejecutable llamado TP1_G14.

3.6.1. Corridas de prueba del programa:

A continuación se exhibirán distintas corridas del programa que funcionaron como prueba para evaluar el correcto funcionamiento del mismo.

- **Corrida del programa utilizando las opciones por defecto:**
  En este caso el usuario deberá ingresar las transacciones por el flujo de entrada estándar. Para indicar el corte de información de entrada, deberá ingresarse EOF, que, desde la terminal de

Ubuntu se consigue con Ctrl+d.
Linea de comando:

```
./TP1_G14 -i - -o -
```

Resultado:

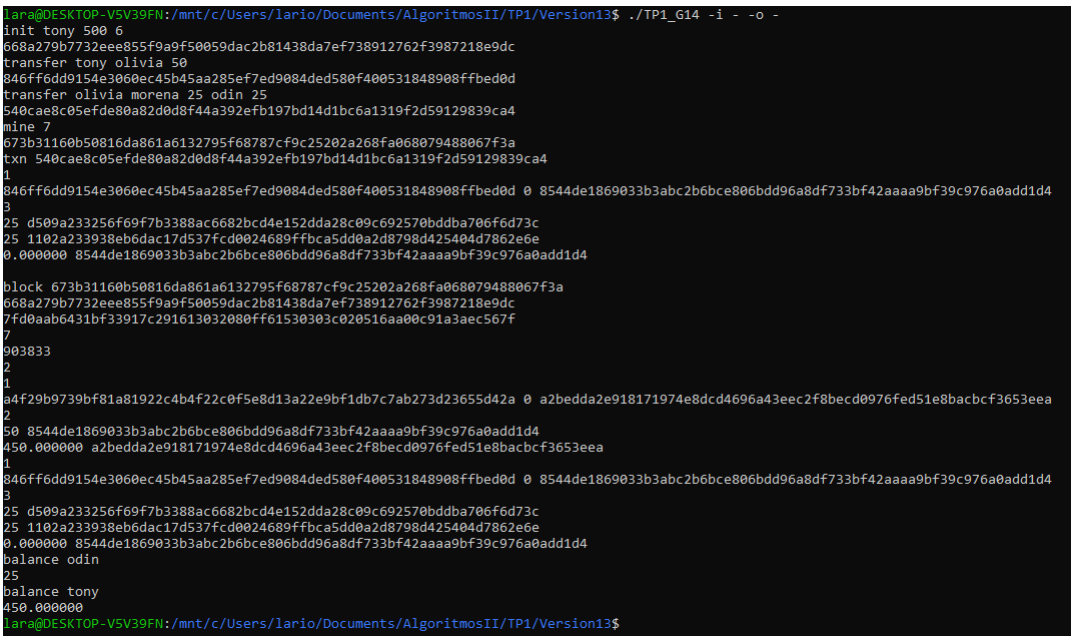

Figura 3.2: Resultado de haber corrido el programa con la linea de comando ./TP0_G14 -i - -o -

- **Corrida del programa utilizando archivos fuente:**
  Se usa como flujo de entrada el archivo commands.txt que contiene distintos comandos para realizarse por el programa y se imprimen los resultados en el archivo program.txt.
  Linea de comando:

```
./TP1_G14 -i commands.txt -o program.txt
```

Resultado:

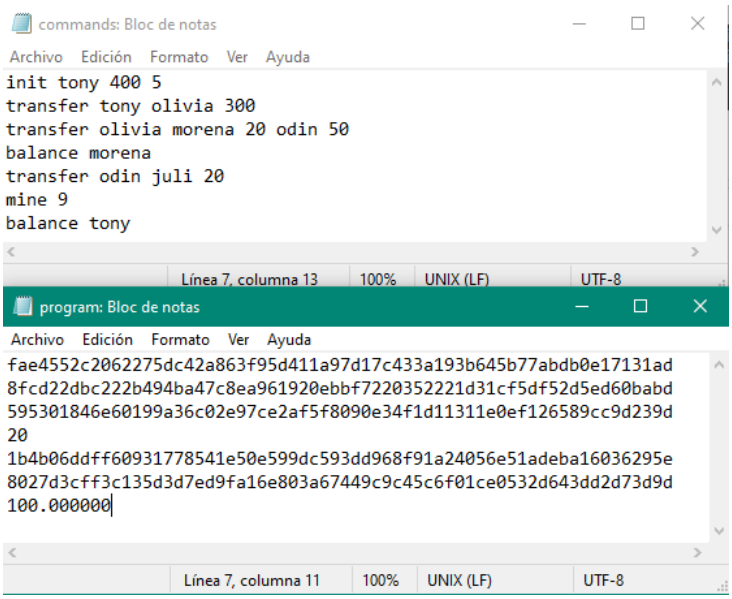

Figura 3.3: Archivo de entrada 'commands.txt' y archivo resultante 'program.txt'

- **Corrida del programa para un archivo de entrada vacío:**
  empty.txt es un archivo vacío. En este caso se imprime por flujo de salida estándar. Como se puede ver en la figura 3.4, el programa no realiza ninguna acción, cumpliendo con lo requerido por la consigna.
  Linea de comando:

  ```
  ./TP1_G14 -i empty.txt -o -
  ```

  Resultado:

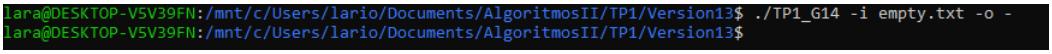

Figura 3.4: Resultado impreso por la terminal ante la entrada de un archivo vacío

- **Corrida del programa con comandos save y load:**
  Se ingresan diversos comandos para generar una cadena *Algochain* y guardarla en el archivo algochain.txt al ingresar el comando *save*. Una vez guardada en el archivo la *Algochain* serializada, se utiliza el comando *load* para cargar la cadena de bloques en el programa.
  Linea de comando:

  ```
  ./TP1_G14 -i - -o -
  ```

  Resultado:

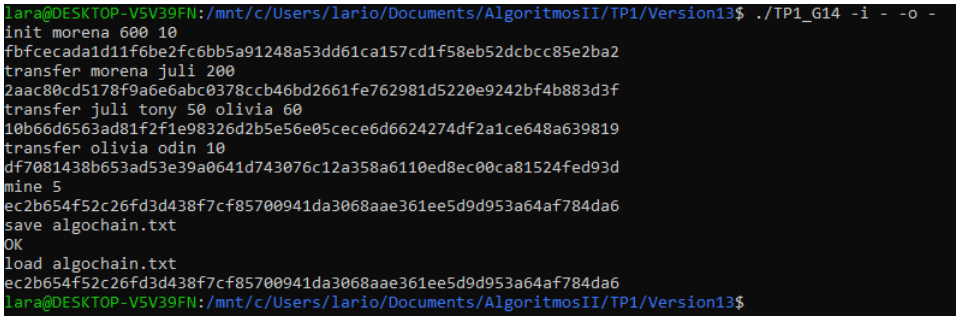

Figura 3.5: Resultados del programa al ingresar los comandos save y load.

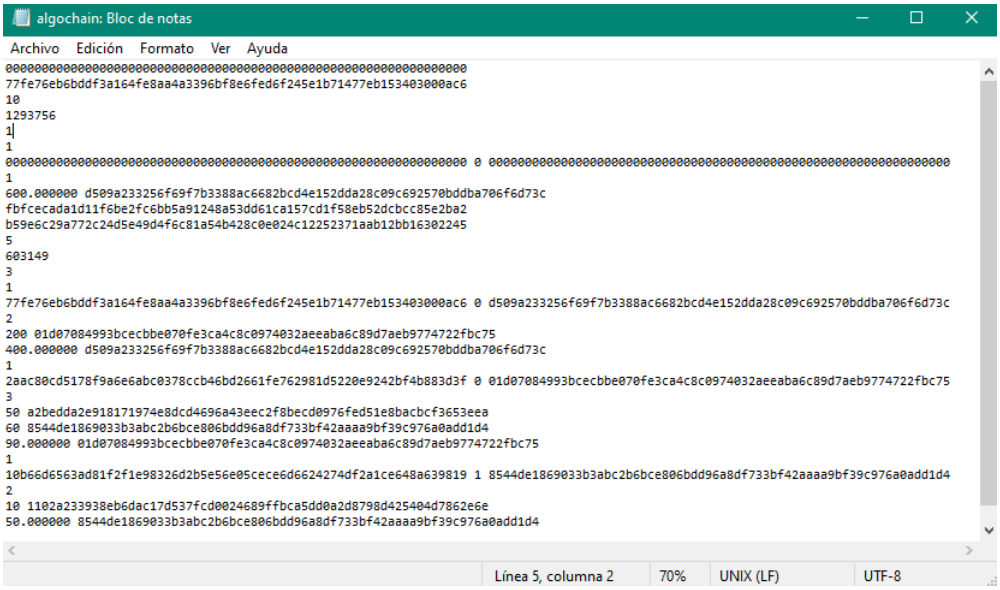

Figura 3.6: Archivo algochain.txt resultante al ingresar el comando save algochain.txt

- **Corrida del programa poniendo a prueba validaciones:**
  Para la segunda imagen presentada en este caso se utiliza el archivo algochain.txt, cuya dificultad es 12, pero el *hash* del *header* de el bloque que contiene el archivo no cumple con la dificultad del minado. La tercer imagen presenta otras validaciones del comando *load* que detectaron fallas en el archivo algochain.txt.
  Linea de comando:

  ```
  ./TP1_G14 --input - --output -
  ```

  Resultado:

```
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 --input - --output -
init morena 300 2
1f7a01d872f53f07fc918e626310e86d15259eb37bbfe7e018e5c5cf5b0b3f5e
transfer tony olivia 40
FAIL: This user does not belong to the Algochain tony.
transfer morena tony 302
FAIL: not sufficient funds.
balance tony
0
mine 2
FAIL: Cannot mine with no transactions.
block 1f7a01d872f53f07fc918e626310e86d15259eb37bbfe7e018e5c5cf5b0b3f5r
FAIL:Block not found 1f7a01d872f53f07fc918e626310e86d15259eb37bbfe7e018e5c5cf5b0b3f5r.
txn b59e6c29a772c24d5e49d4f6c81a54b428c0e024c12252371aab12bb16302245
FAIL:transaction not found.
tranfer morena juli 30
Command not found: --tranfer
```

Figura 3.7: Resultado obtenido al utilizar comandos con información inválida

```
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 --input - --output -
load algochain.txt
FAIL: Difficulty does not verify.
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$
```

algochain: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
0000000000000000000000000000000000000000000000000000000000000000
77fe76eb6bddf3a164fe8aa4a3396bf8e6fed6f245e1b71477eb153403000ac6
12
1293756
1
1
0000000000000000000000000000000000000000000000000000000000000000 0 0000000000000000000000000000000000000000000000000000000000000000
1
600.000000 d509a233256f69f7b3388ac6682bcd4e152dda28c09c692570bddba706f6d73c
```

Línea 9, columna 76 | 70% | UNIX (LF) | UTF-8

Figura 3.8: Resultado obtenido al utilizar el comando load con un archivo corrupto

```
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -
load algochain.txt
a0178ecfe4ae2a43a5c4ad6760b7b35ca6c8de956e228fe107d961c3c3101e83
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -
load algochain.txt
Invalid block, txns_hash does not match transaction's hash.
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -
load algochain.txt
Invalid tx_id format: 0000000000000000000000000000000000000000000000000000000000000000.
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -
load algochain.txt
Invalid input information from command load
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -
load algochain.txt
Invalid addr format: D509a233256f69f7b3388ac6682bcd4e152dda28c09c692570bddba706f6d73c.
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -
load algochain.txt
Invalid transaction, not sufficient funds: 600.
```

Figura 3.9: Distintos tipos de fallas ante validaciones con el comando load y un archivo corrupto

- **Corrida del programa poniendo a prueba validaciones de la función Command::parse():**
  Todos los comandos esperan al menos un parámetro, por lo que cuando no se especifica ninguno, se finaliza al programa.
  Linea de comando:

  ```
  ./TP1_G14 -i - -o -
  ```

  Resultado:

```
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -
init
Command requires argument: --init
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -
mine
Command requires argument: --mine
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -
balance
Command requires argument: --balance
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ ./TP1_G14 -i - -o -

Empty line in commands
```

Figura 3.10: Resultado obtenido al utilizar comandos sin argumentos

- **Corridas del programa con valgrind:**
  Se utiliza el archivo commands.txt para ambas pruebas. Para la primera, se escriben en el archivo comandos válidos, mientras que para la segunda, el archivo está corrupto.
  Linea de comando:

  ```
  valgrind --tool=memchek ./TP0_G14 -i commands.txt -o -
  ```

  Resultado:

```
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ valgrind --tool=memcheck ./TP1_G14 -i commands.txt -o -
==152== Memcheck, a memory error detector
==152== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==152== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==152== Command: ./TP1_G14 -i commands.txt -o -
==152==
==152== error calling PR_SET_PTRACER, vgdb might block
3881c54ba70798de71c90da390402e347007b5e79fe5e5d0cbb09dc15f78fd93
8fcd22dbc222b494ba47c8ea961920ebbf7220352221d31cf5df52d5ed60babd
595301846e60199a36c02e97ce2af5f8090e34f1d11311e0ef126589cc9d239d
20
1b4b06ddff60931778541e50e599dc593dd968f91a24056e51adeba16036295e
dba399986892da7b461394711bc42f8a516c3564a5b0053335ab31c62e01ac5b
100.000000
OK
==152==
==152== HEAP SUMMARY:
==152==     in use at exit: 0 bytes in 0 blocks
==152==   total heap usage: 12,664 allocs, 12,664 frees, 19,126,157 bytes allocated
==152==
==152== All heap blocks were freed -- no leaks are possible
==152==
==152== For lists of detected and suppressed errors, rerun with: -s
==152== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 3.11: Resultado con la linea de comando utilizando el archivo commands.txt sin fallas

```
lara@DESKTOP-V5V39FN:/mnt/c/Users/lario/Documents/AlgoritmosII/TP1/Version13$ valgrind --tool=memcheck ./TP1_G14 -i commands.txt -o -
==153== Memcheck, a memory error detector
==153== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==153== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==153== Command: ./TP1_G14 -i commands.txt -o -
==153==
==153== error calling PR_SET_PTRACER, vgdb might block
cba58d84df90dd5bd5c6d1d6a65f8dfd794644231d479937c3b560c4d22eadc9
8fcd22dbc222b494ba47c8ea961920ebbf7220352221d31cf5df52d5ed60babd
595301846e60199a36c02e97ce2af5f8090e34f1d11311e0ef126589cc9d239d
20
Command not found: --tranfer
==153==
==153== HEAP SUMMARY:
==153==     in use at exit: 1,645,029 bytes in 236 blocks
==153==   total heap usage: 1,424 allocs, 1,188 frees, 8,276,599 bytes allocated
==153==
==153== LEAK SUMMARY:
==153==    definitely lost: 0 bytes in 0 blocks
==153==    indirectly lost: 0 bytes in 0 blocks
==153==      possibly lost: 0 bytes in 0 blocks
==153==    still reachable: 1,645,029 bytes in 236 blocks
==153==                       of which reachable via heuristic:
==153==                         newarray           : 1,643,240 bytes in 205 blocks
==153==         suppressed: 0 bytes in 0 blocks
==153== Rerun with --leak-check=full to see details of leaked memory
==153==
==153== For lists of detected and suppressed errors, rerun with: -s
==153== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 3.12: Resultado con la linea de comando utilizando el archivo commands.txt con fallas

Como puede verse en las imágenes de las figuras 3.11 y 3.12, los resultados obtenidos no son los mismos. Esta diferencia se debe a que cuando se detecta un archivo corrupto, se finaliza al programa con la función *exit(1)* la cual no invoca a los destructores de los objetos del programa. No obstante, aunque esa memoria podría considerarse perdida, el proceso finaliza y, consecuentemente, toda la memoria que tenía es reclamada por el sistema operativo. De este modo puede considerarse que la memoria 'no se pierde'.

4. Conclusiones

Se puede concluir que se han logrado los objetivos propuestos al comienzo del informe, dada la interpretación de la consigna. Se ha podido realizar con éxito la generación de la cadena de bloques *Algochain*, utilizando conceptos claves como POO, modularización, memoria dinámica y *templates*, entre otros. La estrategia utilizada por el grupo consistió en diagramar previamente las distintas clases y estructuras a utilizar. Esto resultó de vital importancia ya que al momento de implementar el código se tuvo una clara visión de como llevarlo a cabo.

Se hizo especial hincapié en la utilización de memoria dinámica. Por ejemplo, para generar los vectores *inputs* y *outputs* con su tamaño adecuado, para generar las listas globales y para instanciar objetos de las clases que se valieran de memoria dinámica.

Se realiza una amplia variedad de validaciones sobre aspectos de formato y sobre cuestiones propias al concepto de la *Algochain* en cada función correspondiente a cada comando a ser utilizado por el usuario del programa.

5. Código Fuente

5.1. main.cc

```
1  #include "main.h"
2
3  static void opt_input(string const &);
4  static void opt_output(string const &);
5
6  istream *iss = 0;    // Input Stream (clase para manejo de los flujos de entrada
       )
7  ostream *oss = 0;    // Output Stream (clase para manejo de los flujos de salida
       )
8  fstream ifs;         // Input File Stream (derivada de la clase ifstream que
       deriva de istream para el manejo de archivos)
```

```
9   fstream ofs;        // Output File Stream (derivada de la clase ofstream que
        deriva de ostream para el manejo de archivos)
10
11  void command_init(string &);
12  void command_transfer(string &);
13  void command_mine(string &);
14  void command_balance(string &);
15  void command_transaction(string &);
16  void command_block(string &);
17  void command_load(string &);
18  void command_save(string &);
19  void clear_global_variables();
20
21  /*************** Elementos globales *****************/
22  static option_t options[] =
23  {
24      {1, "o", "output", "-", opt_output, OPT_DEFAULT},
25      {1, "i", "input", "-", opt_input, OPT_DEFAULT},
26      {0, },
27  };
28
29  static command_t commands[]=
30  {
31   {1, "init", command_init},
32   {1, "balance", command_balance},
33   {1, "transfer", command_transfer},
34   {1, "mine", command_mine},
35   {1, "load", command_load},
36   {1, "save", command_save},
37   {1, "block", command_block},
38   {1, "txn", command_transaction},
39   {0, },
40  };
41
42  unordered_map < string, Vector<string> > wallet;
43
44  list<Block> *Algochain = new list<Block>;
45  list<Transaction> *mempool = new list<Transaction>;
46  list<string> *hashes_transactions = new list<string>;
47  list<string> *hashes_blocks = new list<string>;
48
49  /*****************************************************/
50  static void
51  opt_input(string const &arg)
52  {
53      /* Si el nombre del archivos es "-", usaremos la entrada
54         estandar. De lo contrario, abrimos un archivo en modo
55         de lectura.*/
56      if (arg == "-")
57      {
58          iss = &cin;
59      }
60
61
62      else
63      {
64          ifs.open(arg.c_str(), ios::in); /*c_str(): Returns a pointer to an
                array that contains a null-terminated
65                                          sequence of characters (i.e., a C-
                                                string) representing
66                                          the current value of the string
                                                object.*/
67          iss = &ifs;
```

```
68      }
69
70      // Verificamos que el stream este OK.
71      if (!iss->good()) {
72          cerr << "cannot open file "
73               << arg
74               << "."
75               << endl;
76          exit(1);
77      }
78  }
79
80  static void
81  opt_output(string const &arg)
82  {
83      /* Si el nombre del archivos es "-", usaremos la salida
84         estandar. De lo contrario, abrimos un archivo en modo
85         de escritura.
86      */
87      if (arg == "-") {
88          oss = &cout;     // Establezco la salida estandar cout como flujo de
                salida
89      } else {
90          ofs.open(arg.c_str(), ios::out);
91          oss = &ofs;
92      }
93
94      // Verificamos que el stream este OK.
95      if (!oss->good()) {
96          cerr << "cannot open "
97               << arg
98               << "."
99               << endl;
100         exit(1);         // EXIT: Terminacion del programa en su totalidad
101     }
102 }
103
104 //Genera el primer bloque ( genesis ) de la Algochain
105 //Recibe como argumento el nombre del usuario, el monto inicial, y la
        dificultad de minado
106 //Imprime el hash del bloque genesis o error terminal en caso de falla
107 void command_init(string & args)
108 {
109     Vector <string> inits;
110     string delim = " ";
111
112     inits = split_string(args, delim);
113
114     valid_init_args(inits);
115
116     clear_global_variables();
117
118     Block *block_Genesis = new Block;
119
120     Block Gen(*block_Genesis);
121
122     Gen.load_Genesis(inits[0], stof(inits[1]), stoi(inits[2]));
123
124     Algochain->insert_after(Gen, Algochain->last());
125
126     hashes_blocks->insert(Gen.hash_Block());
127
128     *oss << Gen.hash_Block()<<endl;
```

```
129
130     delete block_Genesis;
131 }
132
133 //Genera una transaccion a ser insertada en la mempool
134 //Recibe como argumento el nombre ususario fuente, el nombre del usuario
        destino y el monto de la transferencia
135 //Imprime el hash de la transaccion o error en caso de falla
136 void command_transfer(string & args)
137 {
138     string source_address, src_arg, src_value,output_pos, source_txn, delim = "
            ";
139     Vector <string> transfers;
140     Transaction txn;
141     float total_value = 0;
142
143     transfers = split_string(args, delim);
144
145     valid_transfer_args_amount(transfers);
146
147     source_address=sha256(sha256(transfers[0]));
148
149     if( wallet.count(source_address) == 1 )
150     {
151         src_value =  (( (wallet.find(source_address)) )->second)[0];
152         source_txn = (( (wallet.find(source_address)) )->second)[1];
153         output_pos = (( (wallet.find(source_address)) )->second)[2];
154     }
155     else
156     {
157         cerr << "FAIL: This user does not belong to the Algochain "
158              << transfers[0]
159              << "."
160              << endl;
161         return;
162     }
163
164     valid_transfer_args(transfers, total_value, src_value);
165     if(total_value > stof(src_value))
166         return;
167
168     // Inserto la transferencia en la mempool
169     txn.load_transaction_for_mempool(transfers, total_value,  stof(src_value),
            source_txn, stoi(output_pos) );
170
171     Block bloque_aux;
172     bloque_aux.load_wallet(txn);
173
174     mempool->insert_after(txn, mempool->last());
175
176     hashes_transactions->insert_after(txn.hash_transaction(),
            hashes_transactions->last());
177     *oss << txn.hash_transaction() << endl;
178 }
179
180 //Genera un bloque con todas las transacciones que se encuentren en la mempool
181 //Recibe como argumento la dificultad de minado del bloque
182 //Imprime el hash del bloque o error en caso de falla
183 void command_mine(string & args)
184 {
185     //Validacion contra dificultad negativa, que sea un int y un maximo de 256.
186     if (is_int(args) == false || stoi(args) > MAXIMUM_DIFFICULTY)
187     {
```

```
188             cerr << "non-valid difficulty: "
189                  << args
190                  << ". It must be a positive integer between 0 and 256."
191                  << endl;
192             exit(1);
193         }
194         //Chequeamos si la mempool esta vacia, porque necesitamos los
                transferencias para armar el bloque
195         if (mempool->empty() == true)
196         {
197             cerr << "FAIL: Cannot mine with no transactions. "
198                  << endl;
199             return;
200         }
201 
202         //Verifico que exista un bloque genesis como primer elemento de la
                Algochain:
203         if( Algochain->empty() == true || Algochain->first().data().is_genesis() ==
                 false)
204         {
205             cerr << "FAIL: Cannot mine with no genesis block. "
206                  << endl;
207             return;
208         }
209 
210         //Cargo el bloque con todas las transacciones de la mempool
211         Block *block = new Block;
212         block->load_block(args);
213 
214         Block bl(*block);
215 
216         //Cargo el bloque en la Algochain:
217         Algochain->insert_after(bl, Algochain->last());
218 
219         hashes_blocks->insert_after(bl.hash_Block(), hashes_blocks->last());
220 
221         delete mempool;
222         mempool = new list<Transaction>;
223 
224         *oss << bl.hash_Block() << endl;
225         delete block;
226     }
227 
228     //Verifica el balance de un usuario en la WALLET
229     //Recibe como argumento el nombre del usuario
230     //Imprime el balance total del usuario o 0 si no se encuentra en la WALLET
231     void command_balance(string & args)
232     {
233         //Chequeo el formato para el nombre del usuario:
234         if (is_alphabetic(args) == false)
235         {
236             cerr << "non-valid user: "
237                  << args
238                  << "."
239                  << endl;
240             exit(1);
241         }
242 
243         if( wallet.count( sha256(sha256(args)) )==1 )
244         {
245             *oss<< (wallet.find(sha256(sha256(args)))->second)[0]<<endl;
246         }
247         else
```

```
248         {
249             *oss<<"0"<<endl;
250         }
251     }
252
253     //Consulta la informacion contenida en la transferencia
254     //Recibe el hash de la transaccion
255     //Imprime los campos de la transaccion o FAIL por hash invalido.
256     void command_transaction(string &args)
257     {
258         //Chequeo el formato para el hash de la transaccion:
259         if (is_alphanumeric(args) == false || args.length() != HASH_LENGTH )
260         {
261             cerr << "non-valid transaction: "
262                  << args
263                  << "."
264                  << endl;
265             exit(1);
266         }
267         //Chequeo si existe la transaccion del hash:
268         if(hashes_transactions->contains(args) == false )
269         {
270             cerr << "FAIL:transaction not found. "
271                  << endl;
272             return;
273         }
274         //Busco en la Algochain
275         for(list<Block>::iterator iter_list(Algochain->first());  iter_list.end()==
                false; iter_list.go_forward())
276         {
277             for(int i=0; i< (iter_list.data()).get_txns_count(); i++)
278             {
279                 if( ((iter_list.data().get_txns())[i]).hash_transaction() == args )
280                 {
281                     *oss<<((iter_list.data().get_txns())[i])<<endl;
282                     return;
283                 }
284             }
285         }//Busco en la Mempool:
286         for(list<Transaction>::iterator iter_list (mempool->first());  iter_list.
                end()==false; iter_list.go_forward())
287         {
288             if( ((iter_list).data()).hash_transaction() == args )
289             {
290                 *oss<<((iter_list).data())<<endl;
291                 return;
292             }
293         }
294
295         cerr << "FAIL:transaction not found "<< args<< "." << endl;
296         return;
297     }
298
299     //Consulta la informacion contenida en un bloque
300     //Recibe el hash del bloque
301     //Imprime todos los campos del bloque o FAIL por hash invalido.
302     void command_block(string & args)
303     {
304         //Chequeo el formato para el hash del block:
305         if (is_alphanumeric(args) == false || args.length() != HASH_LENGTH )
306         {
307             cerr << "non-valid argument for command block: "
308                  << args
```

```
309                 << "."
310                 << endl;
311             exit(1);
312         }
313         //Chequeo si existe el hash del bloque:
314         if( hashes_blocks->contains(args) == false )
315         {
316             cerr << "FAIL:Block not found "
317                 << args
318                 << "."
319                 << endl;
320             return;
321         }
322 
323         for(list<Block>::iterator iter_list(Algochain->first());  iter_list.end()==
                false; iter_list.go_forward())
324         {
325             if(  !(args.compare( ((iter_list).data()).hash_Block()) ) )
326             {
327                 *oss<<((iter_list).data());
328                 return;
329             }
330         }
331     }
332 
333     //Lee la Algochain serializada en el archivo pasado por parametro.
334     //Recibe un archivo que incluye los bloques de la Algochain
335     //Imprime el hash del ultimo bloque de la cadena o FAIL por bloque o/y
            transaccion invalido.
336     void command_load(string & args)
337     {
338         static istream *input_stream = 0;
339         static fstream input_file;
340 
341         //Chequeo el nombre del archivo
342         if (is_alphanumeric_point(args) == false )
343         {
344             cerr << "non-valid file name: "
345                 << args
346                 << "."
347                 << endl;
348             exit(1);
349         }
350         // Se abre un archivo en forma de lectura que contiene bloques
351         input_file.open(args.c_str(), ios::in);
352         input_stream = &input_file;
353 
354         // Verificamos que el stream este OK.
355         if (!input_stream->good()) {
356             cerr << "FAIL:cannot open file "
357                 << args
358                 << "."
359                 << endl;
360             return;
361         }
362 
363         //Validar si el archivo esta vacio
364         if( input_stream->peek() == EOF)
365         {
366             cerr << "Empty file detected, no information to load from "
367                 << args
368                 << "."
369                 << endl;
```

```
370             exit(1);
371         }
372
373         //Borrar algochain anterior
374         clear_global_variables();
375
376         // Cargamos el primer bloque en la Algochain
377         Block *block = new Block;
378
379         //Cargo el bloque con la informacion del archivo input_stream
380         *input_stream>>*block;
381
382         Block bl(*block);
383
384         Algochain->insert_after(bl, Algochain->last());
385
386         hashes_blocks->insert_after(bl.hash_Block(), hashes_blocks->last());
387
388         // Chequeamos que el primer bloque sea el genesis:
389         if( Algochain->first().data().is_genesis() == false )
390         {
391             cerr << "FAIL: First block must be genesis."<< endl;
392             exit(1);
393         }
394
395         // Seguimos cargando los bloques hasta el final del archivo:
396         while(input_stream->eof() == false)
397         {
398             //Cargo el bloque con la informacion del archivo input_stream
399             *input_stream>>*block;
400             Block bl(*block);
401
402             Algochain->insert_after(bl, Algochain->last());
403             hashes_blocks->insert_after(bl.hash_Block(), hashes_blocks->last());
404         }
405         *oss<< hashes_blocks->last().data()<<endl;
406         delete block;
407     }
408
409     //Lee la Algochain serializada e imprime en el archivo pasado por parametro
410     //Recibe un archivo
411     //Imprime OK en exito o FAIL en FALLA.
412     void command_save(string & args)
413     {
414         fstream output_file;
415         ostream *output_stream;
416
417         //Chequeo el nombre del archivo
418         if (is_alphanumeric_point(args) == false )
419         {
420             cerr << "non-valid file name: "
421                  << args
422                  << "."
423                  << endl;
424             exit(1);
425         }
426
427         //Abro el archivo en modo escritura:
428         output_file.open(args.c_str(), ios::out);
429         output_stream=&output_file;
430
431         // Verificamos que el stream este OK.
432         if (!output_stream->good()){
```

```
433             cerr << "FAIL:cannot open file "
434                  << args
435                  << "."
436                  << endl;
437             return;
438         }
439
440         for(list<Block>::iterator iter_list(Algochain->first()); iter_list.end()==
                false ; iter_list.go_forward())
441         {
442             output_file<<((iter_list).data());
443         }
444         *oss<<"OK"<<endl;
445     }
446
447     //Funcion para borrar la informacion de las variables globales
448     void clear_global_variables()
449     {
450         //Borrar algochain anterior
451         if (Algochain->empty() == false)
452         {
453             //borra los bloques de la algochain
454             delete Algochain;
455             Algochain = new list<Block>;
456
457             //borra la wallet
458             wallet.clear();
459
460             //borra los bloques de la hashes_txn
461             delete hashes_transactions;
462             hashes_transactions = new list<string>;
463
464             //borra los bloques de la hashes_block
465             delete hashes_blocks;
466             hashes_blocks = new list<string>;
467
468             //borra los bloques de la mempool
469             delete mempool;
470             mempool = new list<Transaction>;
471         }
472     }
473
474     int
475     main(int argc, char * const argv[])
476     {
477         string line;
478
479         cmdline cmdl(options);  // Objeto con parametro tipo option_t (struct)
                declarado globalmente.
480         cmdl.parse(argc, argv); // Metodo de parseo de la clase cmdline
481
482         Command command(commands);
483
484         if (iss == &cin)
485         {
486             while(1)
487             {
488                 if(iss->eof() == true)
489                     return -1;
490                 command.parse(iss);
491             }
492         }
493         else
```

```
494     {
495         command.parse(iss);
496     }
497 
498     delete Algochain;
499     delete mempool;
500     delete hashes_transactions;
501     delete hashes_blocks;
502 }
```

5.2. main.h

```
1  #ifndef MAIN_INCLUDED
2  #define MAIN_INCLUDED
3  
4  #include "Block.h"
5  #include "cmdline.h"
6  #include "list.h"
7  #include "Command.h"
8  #include "Transaction.h"
9  
10 #include <string>
11 #include <string.h>
12 #include <fstream>
13 #include <iomanip>
14 #include <iostream>
15 #include <sstream>
16 #include <cstdlib>
17 #include <cstring>
18 #include <stdio.h>
19 
20 #define DELIM_LENGTH 1
21 #define AMOUNT_COMMANDS 8
22 #define MAXIMUM_DIFFICULTY 256
23 
24 using namespace std;
25 
26 #endif
```

5.3. Vector.h

```
1  #ifndef VECTOR_H_INCLUDED_
2  #define VECTOR_H_INCLUDED_
3  #include <iostream>
4  #include <string>
5  
6  #define DEFECT_SIZE 100
7  
8  using namespace std;
9  
10 template <typename T>
11 class Vector
12 {
13 private:
14     int size; // Cuanto va a tener de largo el Array
15     T *ptr; // Puntero al vector dinamico que contiene los datos de tipo T
16 
17 public:
```

```
18      Vector();
19      Vector(int);
20      Vector(const Vector <T> &);
21      Vector(int, const Vector <T> & );
22      ~Vector();
23
24      Vector<T>& operator=(const Vector <T> &);
25      bool operator==(const Vector <T> &);
26      T & operator[](int) const;
27
28      int get_size() const;
29      int search(const T &);
30      int position(const T &);
31      void clear();
32  };
33
34  template <typename T>
35  Vector<T>::Vector()
36  {
37      size = DEFECT_SIZE;
38      ptr = new T[size];
39  }
40
41  template <typename T>
42  Vector<T>::Vector(int x)
43  {
44      size = x;
45
46      if(x!=0)
47          ptr = new T[size];
48      else
49          ptr = 0;
50  }
51
52  template <typename T>
53  Vector<T>::Vector(const Vector <T> & v)
54  {
55      size = v.size;
56
57      ptr = new T[size];
58
59      for (int i=0; i < size ; i++)
60          ptr[i] = v.ptr[i];
61  }
62
63  template <typename T>
64  Vector<T>::Vector(int len,const Vector <T> & v)
65  {
66      size=len;
67
68      if(len !=0)
69      {
70          ptr = new T[size];
71
72          for (int i=0; i < v.size ; i++)
73              ptr[i] = v.ptr[i];
74      }
75  }
76
77  template <typename T>
78  Vector<T>::~Vector()
79  {
80      if (ptr) //si el puntero NO apunta a NULL-> libera la memoria pedida
```

```
81          delete [] ptr;
82  }
83
84  template <typename T>
85  Vector<T>& Vector<T>::operator=(const Vector<T> & v2) // v1 = v2
86  {
87      if (this != &v2 )
88      {
89          if (size != v2.size)
90          {
91              T * aux;
92              size = v2.size;
93              delete [] ptr;
94              aux = new T[size];
95              ptr = aux;
96          }
97          for (int i = 0 ; i < size; i++)
98              ptr[i] = v2.ptr[i];
99          return *this; //devuelve el vector de <T> de la clase Vector por
                referencia
100     }
101     return *this;
102 }
103
104 template <typename T>
105 bool Vector<T>::operator==(const Vector<T> & v2)//v1 == v2
106 {
107     if (size != v2.size)
108         return false;
109     else
110     {
111         for (int i = 0 ; i < size ; i++)
112         {
113             if (ptr[i] != v2.ptr[i])
114                 return false;
115         }
116         return true;
117     }
118 }
119
120 template <typename T>
121 T & Vector<T>::operator[](int pos) const// v[pos]
122 {
123     if( pos < size)
124         return ptr[pos];
125     cout << "Out of range"<< endl;
126     exit(1);
127 }
128
129 template <typename T>
130 int Vector<T>::get_size() const
131 {
132     return size;
133 }
134
135 template <typename T>
136 int Vector<T>::search(const T & data)
137 {
138     int j = 0;
139     for (int i = 0 ; i < size ; i++)
140     {
141         if (ptr[i] == data)
142             j++;
```

```
143         }
144         return j;
145     }
146
147     template <typename T>
148     int Vector<T>::position(const T & data)
149     {
150         for (int i = 0 ; i < size ; i++)
151         {
152             if (ptr[i] == data)
153                 return i;
154         }
155         return -1;
156     }
157
158     template <typename T>
159     void Vector<T>::clear()
160     {
161         if(ptr)
162             delete [] ptr;
163         size = 0;
164     }
165
166     #endif //VECTOR_H_INCLUDED_
```

5.4. Block.h

```
1   #ifndef _BLOCK_H_INCLUDED_
2   #define _BLOCK_H_INCLUDED_
3
4   #include "Transaction.h"
5   #include "Vector.h"
6   #include "sha256.h"
7   #include "cmdline.h"
8   #include "Utilities.h"
9   #include "list.h"
10  #include "main.h"
11
12  #include "MerkleTree.h"
13
14  #include <cstddef>
15  #include <bitset>
16  #include <time.h>
17  #include <string>
18  #include <string.h>
19  #include <iostream>
20  #include <fstream>
21  #include <sstream>
22  #include <cstdlib>
23  #include <cstring>
24  #include <cmath>
25  #include <unordered_map>
26
27  using namespace std;
28
29  #define PREV_BLOCK_GENESIS "
        0000000000000000000000000000000000000000000000000000000000000000"
30  #define NONCE_MIN 10000
31  #define NONCE_MAX 2000000
32  #define HASH_LENGTH 64
33
```

```
34  struct header_t
35  {
36      string prev_block;
37      string txns_hash;
38      int bits; // Dificultad
39      int nonce; // Para el hash
40  };
41
42  struct body_t
43  {
44      int txns_count;
45      Vector<Transaction> txns;
46  };
47
48  class Block
49  {
50  private:
51      header_t * header;
52      body_t * body;
53
54  public:
55      Block();
56      Block(const Block &);
57      ~Block();
58      void load_block(string);
59      void load_Genesis(string &,float, int);
60      bool is_genesis();
61
62      string get_prev_block()const { return header->prev_block;};
63      string get_txns_hash()const {return header->txns_hash;};
64      int get_bits()const {return header->bits;};
65      int get_nonce()const {return header->nonce;};
66      int get_txns_count()const {return body->txns_count;};
67      Vector<Transaction> get_txns()const {return body->txns;};
68
69      void set_prev_block(string value){header->prev_block=value;};
70      void set_txns_hash(string value){header->txns_hash=value;};
71      void set_bits(int value){header->bits=value;};
72      void set_nonce(int value){header->nonce=value;};
73
74      void set_txns_count(int value){body->txns_count=value;};
75      void set_txns(Vector<Transaction> value){body->txns=value;};
76
77      bool is_valid_hash_header(int dif);
78      void check_hash_header(int dif);
79      string hash_Block();
80      void load_wallet(Transaction &);
81
82      friend ostream & operator<<( ostream &, Block &);
83      friend istream & operator>>( istream &, Block &);
84  };
85
86  extern list<Block> *Algochain;
87  extern list<Transaction> *mempool;
88  extern list<string> *hashes_transactions;
89  extern list<string> *hashes_blocks;
90
91  extern istream *iss;     // Input Stream (clase para manejo de los flujos de
        entrada)
92  extern ostream *oss;     // Output Stream (clase para manejo de los flujos de
        salida)
93  extern fstream ifs;      // Input File Stream (derivada de la clase ifstream que
         deriva de istream para el manejo de archivos)
```

```
94  extern fstream ofs;     // Output File Stream (derivada de la clase ofstream
        que deriva de ostream para el manejo de archivos)
95
96  #endif
```

5.5. Block.cc

```
1   #include "Block.h"
2
3   Block::Block()
4   {
5       header = new header_t;
6       body = new body_t;
7   }
8
9   Block::Block(const Block &bl)
10  {
11      header = new header_t;
12      body = new body_t;
13
14      *header = *bl.header;
15      *body = *bl.body;
16  }
17
18
19  Block::~Block()
20  {
21      delete header;
22      delete body;
23  }
24
25  bool Block::is_genesis()
26  {
27      if(header->prev_block == PREV_BLOCK_GENESIS)
28      {
29          if( (body->txns)[0].get_input(0).tx_id !=PREV_BLOCK_GENESIS)
30              return false;
31
32          if( (body->txns)[0].get_input(0).idx != "0")
33              return false;
34
35          if( (body->txns)[0].get_input(0).addr != PREV_BLOCK_GENESIS)
36              return false;
37
38          return true;
39      }
40      return false;
41  }
42
43
44  bool Block::is_valid_hash_header(int dif)
45  {
46      string hash_header;
47      hash_header = header->prev_block + "\n"+ header->txns_hash + "\n" + std::
            to_string(header->bits)+ "\n" + std::to_string(header->nonce) + "\n";
48      string h = sha256(sha256(hash_header));
49
50      //Parte de division:
51      int position=dif/4;     // Nro de caracteres -----> Valor de dividendo
52      int rest=dif%4;
53
```

```
54      srand(time(NULL));  //Se usa para generar una semilla para generar numeros
            aleatorios
55
56      string test(h); //Genera una cadena auxiliar con el valor de h
57
58      test.resize(position+1);    //Genera una cadena de n bits a testear donde
            cada caracter determinado por position son 4 bits
59
60      test=Hex2Bin(test ); //Queda guardado el numero en binario
61
62      string  string_zeros(position*4+rest, '0'); /*Forma una cadena de "0" igual
             a la cantidad de bits cero que necesita
63                                                      Position*4 porque
                                                            position lo que cuenta
                                                             son 4 bits
64                                                      Rest son bits*/
65
66      int comparation=(test.compare(0, string_zeros.length(), string_zeros));
67
68      if(comparation != 0)
69          return false;
70      else
71          return true;
72  }
73
74  void Block::check_hash_header(int dif)
75  {
76      string hash_header;
77      hash_header = header->prev_block + "\n"+ header->txns_hash + "\n" + std::
            to_string(header->bits)+ "\n" + std::to_string(header->nonce)+ "\n";
78      string h = sha256(sha256(hash_header));
79
80      //Parte de division:
81      int position=dif/4;     // Nro de caracteres -----> Valor de dividendo
82      int rest=dif%4;
83
84      srand(time(NULL));  //Se usa para generar una semilla para generar numeros
            aleatorios
85
86      std::string test(h);    //Genera una cadena auxiliar con el valor de h
87
88      test.resize(position+1);    //Genera una cadena de n bits a testear donde
            cada caracter determinado por position son 4 bits
89
90      test=Hex2Bin(test ); //Queda guardado el numero en binario
91
92      std::string string_zeros(position*4+rest, '0'); /*Forma una cadena de "0"
            igual a la cantidad de bits cero que necesita
93                                                      Position*4 porque
                                                            position lo que cuenta
                                                             son 4 bits
94                                                      Rest son bits*/
95
96      int comparation=(test.compare(0, string_zeros.length(), string_zeros));
97
98      while( comparation != 0)
99      {
100         header->nonce = rand() % NONCE_MAX + NONCE_MIN;     //Genera un nonce
                aleatorio
101
102         hash_header = header->prev_block + "\n"+ header->txns_hash+ "\n"+ std::
                to_string(header->bits)+ "\n" + std::to_string(header->nonce)+ "\n";
103         h = sha256(sha256(hash_header));
```

```
104
105             string test(h);
106             test.resize(position+1);
107
108             test=Hex2Bin(test);
109
110             comparation=(test.compare(0, string_zeros.length(), string_zeros));
111         }
112     }
113
114     void Block::load_block(string dif)
115     {
116         //Carga el body:
117         body->txns_count = mempool->size();
118         list<Transaction>::iterator mem = mempool->first();
119
120         Vector <TreeNode *> txns_hashes(body->txns_count); //Vector de hojas del
                arbol de Merkle
121         TreeNode *p = NULL;
122         //Se itera sobre cada transaccion de la mempool
123         for (int i = 0 ; i < body->txns_count ; i++)
124         {
125             //Se duplica memoria del vector transacciones si no hay mas espacio
126             if (body->txns_count == (body->txns).get_size())
127             {
128                 Vector <Transaction> aux(2*((body->txns).get_size()), body->txns);
129                 body->txns = aux;
130             }
131
132             //Se incluye la transaccion en el vector de transacciones
133             (body->txns)[i] = (mem.data());
134
135             txns_hashes[i] = new TreeNode((body->txns)[i].hash_transaction());
136             txns_hashes[i]->set_left(p);
137             txns_hashes[i]->set_rigth(p);
138
139             mem.go_forward();
140         }
141
142         header->prev_block = ((Algochain->last()).data()).hash_Block();
143
144         MerkleTree *Tree = new MerkleTree(txns_hashes);
145         header->txns_hash = (Tree->get_root())->get_hash();
146
147         delete Tree;
148
149         header->bits = stoi(dif);
150
151         header->nonce = NONCE_MIN;
152
153         check_hash_header(stoi(dif));
154     }
155
156     void Block::load_Genesis(string & user, float value, int dif)
157     {
158         body->txns_count = 1;
159         (body->txns)[0].set_n_tx_in("1");
160         (body->txns)[0].set_n_tx_out("1");
161
162         (body->txns)[0].set_input(PREV_BLOCK_GENESIS, "0", PREV_BLOCK_GENESIS, 0);
163         (body->txns)[0].set_output(to_string(value), sha256(sha256(user)), 0);
164
165         header->prev_block = PREV_BLOCK_GENESIS;
```

```
166
167         Vector <TreeNode*> txns_hashes(body->txns_count); //Vector de hojas del
                arbol de Merkle
168         txns_hashes[0] = new TreeNode((body->txns)[0].hash_transaction());//Carga
                las hojas
169
170         TreeNode * p = NULL;
171         txns_hashes[0]->set_left(p);
172         txns_hashes[0]->set_rigth(p);
173
174
175         MerkleTree * hashTree = new MerkleTree(txns_hashes);
176         header->txns_hash = (hashTree->get_root())->get_hash();
177
178         delete hashTree;
179
180         header->bits = dif;
181         header->nonce = NONCE_MIN;
182
183         check_hash_header(dif);
184
185         //Se carga el Vector<string> de la wallet con el value, hash transferencia
                y el idx
186         Vector<string> aux(3);
187         aux[0] = to_string(value);
188         aux[1] = (body->txns)[0].hash_transaction();
189         aux[2] = "0";
190
191         wallet.insert( {sha256(sha256(user)), aux} );
192     }
193
194
195     string Block::hash_Block()
196     {
197         string hash_block = "";
198         string header_txnshash = "";
199
200         for (int i = 0 ; i < body->txns_count ; i++)
201         {
202             header_txnshash += (body->txns)[i].get_n_tx_in() + "\n";
203
204             for(int j = 0; j < (stoi((body->txns)[i].get_n_tx_in())); j++)//Cant de
                    inputs
205             {
206                 header_txnshash += ((body->txns)[i]).get_input(j).tx_id + " ";
207                 header_txnshash += ((body->txns)[i]).get_input(j).idx + " ";
208                 header_txnshash += ((body->txns)[i]).get_input(j).addr + "\n";
209             }
210
211             header_txnshash += (body->txns)[i].get_n_tx_out() + "\n";
212
213             for(int j = 0; j < (stoi((body->txns)[i].get_n_tx_out())); j++)//Cant
                    de outputs
214             {
215                 header_txnshash += ((body->txns)[i]).get_output(j).value + " ";
216                 header_txnshash += ((body->txns)[i]).get_output(j).addr + "\n";
217             }
218         }
219
220         hash_block = header->prev_block +"\n"+ header->txns_hash +"\n"+ to_string(
                header->bits) +"\n"+ to_string(header->nonce) +"\n"+ to_string(body->
                txns_count) +"\n"+ header_txnshash +"\n";
221
```

```
222         return sha256(sha256(hash_block));
223     }
224
225
226     void Block::load_wallet(Transaction & txn)
227     {
228         //Al usuario que aparece en el INPUT de la transaccion: se le tiene que
                poner el valor del ultimo output de esta transaccion
229
230         // Cuando aparece en el input hay que dejarle el vuelto en el value
231         (wallet.find( (txn.get_input(0)).addr )->second)[0] = txn.get_output(stoi(
                txn.get_n_tx_out() ) -1 ).value;
232
233         //Actualizar la segunda posicion del vector con el hash de la transaccion
                actual
234         (wallet.find( (txn.get_input(0)).addr )->second)[1] = txn.hash_transaction
                ();
235
236         //Actualizar la ultima posicion del vector con la posicion del ULTIMO
                output de esta transaccion
237         (wallet.find( (txn.get_input(0)).addr )->second)[2] =  to_string( stoi(txn.
                get_n_tx_out()) - 1 );
238
239         //Cargar en la wallet los fondos que se correspondan con los user que
                aparecen solo en las outputs:
240         //Se itera sobre los outputs de esta transaccion
241         //Se recorre hasta el anteultimo output porque el ultimo tiene el vuelto
                del user que aparece en el input
242         float total_value;
243
244         for (int j = 0; j < stoi(txn.get_n_tx_out()) -1 ; j++)
245         {
246             if( (wallet.count( (txn.get_output(j)).addr ) == 1) )
247             {
248                 //Ya esta en la wallet
249                 total_value = stof((wallet.find( (txn.get_output(j)).addr )->second
                        )[0])+ stof( txn.get_output(j).value );
250
251                 (wallet.find( (txn.get_output(j)).addr )->second)[0] = to_string(
                        total_value);
252
253                 (wallet.find( (txn.get_output(j)).addr )->second)[1] = txn.
                        hash_transaction();
254
255                 (wallet.find( (txn.get_output(j)).addr )->second)[2] = to_string(j)
                        ;
256             }
257             else
258             {
259                 //No esta en la wallet
260                 //Se carga el Vector<string> de la wallet con el value, hash
                        transferencia y el idx
261                 Vector<string> aux(3);
262                 aux[0] = txn.get_output(j).value;
263                 aux[1] = txn.hash_transaction();
264                 aux[2] = to_string(j);
265
266                 pair <string, Vector<string> > mywallet (txn.get_output(j).addr ,
                        aux);
267                 wallet.insert(mywallet);
268             }
269         }
270     }
```

```
271
272 //Operador para imprimir en output
273 ostream & operator<<( ostream &output,  Block &block)
274 {
275     output<<block.get_prev_block()<<"\n";
276     output<<block.get_txns_hash()<<"\n";
277     output<<block.get_bits()<<"\n";
278     output<<block.get_nonce()<<"\n";
279     output<<block.get_txns_count()<<"\n";
280
281     int amount_outputs = block.get_txns_count();
282     Vector<Transaction> vector_txns = block.get_txns();
283
284     for(int i = 0; i < amount_outputs; i++)//Cant de Transacciones
285     {
286         output<<(vector_txns)[i];
287     }
288     return output;
289 }
290
291 //Operador para cargar un bloque a partir de un input
292 istream & operator>>( istream & input, Block & block)
293 {
294     string aux_load;
295     // Carga el header del bloque:
296     getline(input, aux_load);
297     if(input.eof() == true)
298         return input;
299     if (aux_load.length() != HASH_LENGTH || is_alphanumeric(aux_load) == false
        || aux_load.empty() == true)
300     {
301         cerr << "Invalid input information from command load"<< endl;
302         exit(1);
303     }
304     block.set_prev_block(aux_load);
305
306     getline(input, aux_load);
307     if (aux_load.length() != HASH_LENGTH || is_alphanumeric(aux_load) == false
        || aux_load.empty() == true)
308     {
309         cerr << "Invalid input information from command load"<< endl;
310         exit(1);
311     }
312     block.set_txns_hash(aux_load);
313
314     getline(input, aux_load);
315     if ( is_int(aux_load) == false || stoi(aux_load) > MAXIMUM_DIFFICULTY ||
        aux_load.empty() == true )
316     {
317         cerr << "Invalid input information from command load. "<< endl;
318         exit(1);
319     }
320     block.set_bits(stoi(aux_load));
321
322     getline(input,aux_load);
323     if ( is_int(aux_load) == false || aux_load.empty() == true)
324     {
325         cerr << "Invalid input information from command load. "<< endl;
326         exit(1);
327     }
328     block.set_nonce(stoi(aux_load));
329
330     //Carga el Body:
```

```
331         getline(input,aux_load);
332         if ( is_int(aux_load) == false || aux_load.empty() == true)
333         {
334             cerr << "Invalid input information from command load. "<< endl;
335             exit(1);
336         }
337         block.set_txns_count(stoi(aux_load));
338 
339 
340         Vector <Transaction> aux_tnxs(block.get_txns_count());
341         Vector <TreeNode *> leaves_hashes(block.get_txns_count()); //Vector de
                hojas del arbol de Merkle
342         TreeNode *p = NULL;
343 
344         float total_txn_value = 0;
345         for(int i = 0; i < block.get_txns_count(); i++)
346         {
347             //Se carga cada transaccion
348             aux_tnxs[i].load_Transaction(&input, total_txn_value);
349 
350             if(wallet.empty()==false)
351             {
352                 //Se valida que la input de la transccion exista en la wallet
353                 if( wallet.count(aux_tnxs[i].get_input(0).addr) == 0)
354                 {
355                     cerr << "FAIL: Non valid source for transaction: "
356                          << aux_tnxs[i].get_input(0).addr
357                          << "."
358                          << endl;
359                     exit(1);
360                 }
361                 if(total_txn_value > stof((wallet.find(aux_tnxs[i].get_input(0).
                        addr)->second)[0]) )
362                 {
363                     cerr << "Invalid transaction, not sufficient funds: "
364                          << stof((wallet.find(aux_tnxs[i].get_input(0).addr)->
                                second)[0])
365                          << ". "
366                          << endl;
367                     exit(1);
368                 }
369 
370                 block.load_wallet( aux_tnxs[i] );
371             }
372             else
373             {
374                 Vector <string> aux_genesis(3);
375                 aux_genesis[0] = (aux_tnxs)[0].get_output(0).value;
376                 aux_genesis[1] = (aux_tnxs)[0].hash_transaction();
377                 aux_genesis[2] = "0";
378 
379                 wallet.insert( {(aux_tnxs)[i].get_output(0).addr, aux_genesis} );
380             }
381 
382             hashes_transactions->insert_after((aux_tnxs)[i].hash_transaction(),
                    hashes_transactions->last());
383 
384             leaves_hashes[i] = new TreeNode((aux_tnxs)[i].hash_transaction());
385             leaves_hashes[i]->set_left(p);
386             leaves_hashes[i]->set_rigth(p);
387         }
388 
389         MerkleTree *Tree = new MerkleTree(leaves_hashes);
```

```
390         string expected_txns_hash = (Tree->get_root())->get_hash();
391         delete Tree;
392 
393         if(expected_txns_hash != block.get_txns_hash())
394         {
395             cerr << "Invalid block, txns_hash does not match transaction's hash."
396                  << endl;
397             exit(1);
398         }
399         block.set_txns(aux_tnxs);
400 
401         //Verifica que el hash del header cumpla con la dificultad pasada
402         if(block.is_valid_hash_header(block.get_bits()) == false)
403         {
404             cerr << "FAIL: Difficulty does not verify."
405                  << endl;
406             exit(1);
407         }
408         return input;
409     }
```

5.6. Transaction.h

```
1   #ifndef _TRANSACTION_H_INCLUDED_
2   #define _TRANSACTION_H_INCLUDED_
3   #include "Vector.h"
4   #include "cmdline.h"
5   #include "Utilities.h"
6   #include "sha256.h"
7 
8   #include <cstring>
9   #include <ostream>
10  #include <string>
11  #include <string.h>
12  #include <string_view>
13  #include <iostream>
14  #include <istream>
15  #include <fstream>
16  #include <iomanip>
17  #include <sstream>
18  #include <cstdlib>
19 
20  #define HASH_LENGTH 64
21 
22  using namespace std;
23 
24  struct input_t {
25 
26      string tx_id;
27      string idx;
28      string addr;
29  };
30 
31  struct output_t {
32 
33      string value;
34      string addr;
35  };
36 
37 
38  class Transaction
```

```
39  {
40  private:
41      string n_tx_in;
42      string n_tx_out;
43
44      Vector <input_t> inputs;
45      Vector <output_t> outputs;
46
47  public:
48      Transaction();
49      ~Transaction();
50      void load_transaction_for_mempool(Vector <string> &, float , float, string,
            int);
51      void load_Transaction(istream *, float &);
52      string get_n_tx_in()const{return n_tx_in;};
53      string get_n_tx_out()const{return n_tx_out;};
54      input_t get_input(int)const;
55      output_t get_output(int)const;
56
57      void set_n_tx_in(string value){n_tx_in = value;};
58      void set_n_tx_out(string value){n_tx_out = value;};
59      void set_inputs(Vector <input_t> value){inputs=value;};
60      void set_outputs(Vector <output_t> value){outputs=value;};
61      void set_input(string txid_, string idx_, string addr_, int pos){inputs[pos
            ].tx_id = txid_; inputs[pos].idx = idx_ ;inputs[pos].addr = addr_;};
62      void set_output(string val, string addr_, int pos){outputs[pos].value = val
            ; outputs[pos].addr = addr_ ; };
63      string hash_transaction();
64
65      friend ostream & operator<<( ostream &, const Transaction &txn);
66      friend istream & operator>>( istream &, Transaction &);
67  };
68
69  #include <unordered_map>
70  extern unordered_map < string, Vector<string> > wallet;
71
72  #endif
```

5.7. List.h

```
1   #ifndef _LIST_H_INCLUDED_
2   #define _LIST_H_INCLUDED_
3
4   #include <cstdlib>
5   #include "Block.h"
6
7
8   using namespace std;
9
10  template<typename T>
11  class list
12  {
13      class node
14      {
15          friend class iterator;
16          friend class list;
17          node *next_;
18          node *prev_;
19          T data_;
20
21      public:
```

```
22          node(T const&);
23          ~node();
24      };
25      node *first_;
26      node *last_;
27      size_t siz_;
28
29  public:
30      class iterator
31      {
32          friend class list;
33
34          node *actual_;
35          iterator(node*);
36
37      public:
38          iterator();
39          iterator(list<T> const &);
40          iterator(iterator const &);
41          ~iterator();
42
43          T& data();
44          T const &data() const;
45          iterator &go_forward();
46          iterator &move_backwards();
47          bool end() const;
48
49          bool operator==(const iterator &) const;
50          bool operator!=(const iterator &) const;
51          iterator const &operator=(iterator const &);
52      };
53
54      typedef T t_data;
55      typedef node t_node;
56      typedef iterator t_iter;
57
58      list();
59      list(const list &);
60      ~list();
61
62      //Metodos de list.
63      size_t size() const;
64      bool contains(const T &) const;
65      bool empty() const;
66      void insert(const T &);
67      void insert_before(const T &, iterator const &);
68      void insert_after(const T &, iterator const &);
69      void erase(const T &);
70      void destroy(iterator);
71      list const &operator=(list const &);
72
73      iterator first() const;
74      iterator last() const;
75
76  };
77
78  template<typename T>
79  list<T>::iterator::iterator() : actual_(0)
80  {
81  }
82
83  template<typename T>
84  list<T>::iterator::iterator(node *actual) : actual_(actual)
```

```
85  {
86
87  }
88
89  template<typename T>
90  list<T>::iterator::iterator(list<T> const &l) : actual_(l.first_)
91  {
92  }
93
94  template<typename T>
95  list<T>::iterator::iterator(iterator const &it) : actual_(it.actual_)
96  {
97  }
98
99  template<typename T>
100 list<T>::iterator::~iterator()
101 {
102 }
103
104 template<typename T>
105 T & list<T>::iterator::data()
106 {
107     return actual_->data_;
108 }
109
110 template<typename T>
111 T const &list<T>::iterator::data() const
112 {
113     return actual_->data_;
114 }
115
116 template<typename T>
117 typename list<T>::iterator &list<T>::iterator::go_forward()
118 {
119     if (actual_)
120         actual_ = actual_->next_;
121     return *this;
122 }
123
124 template<typename T>
125 typename list<T>::iterator &list<T>::iterator::move_backwards()
126 {
127     if (actual_)
128         actual_ = actual_->prev_;
129     return *this;
130 }
131
132 template<typename T>
133 bool list<T>::iterator::end() const
134 {
135     return actual_ == 0 ? true : false;
136 }
137
138
139 template<typename T>
140 bool list<T>::iterator::operator==(const typename list<T>::iterator &it2) const
141 {
142     return actual_ == it2.actual_;
143 }
144
145 template<typename T>
146 bool list<T>::iterator::operator!=(const typename list<T>::iterator &it2) const
147 {
```

```
148         return actual_ != it2.actual_;
149 }
150
151 template<typename T>
152 typename list<T>::iterator const &list<T>::iterator::operator=(iterator const &
        orig)
153 {
154         if (this != &orig)
155         actual_ = orig.actual_;
156     return *this;
157 }
158
159 template<typename T>
160 list<T>::node::node(const T &t) : next_(0), prev_(0), data_(t)
161 {
162 }
163
164 template<typename T>
165 list<T>::node::~node()
166 {
167 }
168
169 template<typename T>
170 list<T>::list() : first_(0), last_(0), siz_(0)
171 {
172 }
173
174 template<typename T>
175 list<T>::list(const list &orig) : first_(0), last_(0), siz_(orig.siz_)
176 {
177     node *iter;
178     node *ant;
179
180     for (iter = orig.first_, ant = 0; iter != 0; iter = iter->next_)
181     {
182         node *nuevo = new node(iter->data_);
183         nuevo->prev_ = ant;
184         nuevo->next_ = 0;
185         if (ant != 0)
186             ant->next_ = nuevo;
187
188         if (first_ == 0)
189             first_ = nuevo;
190         ant = nuevo;
191     }
192
193     last_ = ant;
194 }
195
196 template<typename T>
197 list<T>::~list()
198 {
199     for (node *p = first_; p; )
200     {
201         node *q = p->next_;
202         delete p;
203         p = q;
204     }
205 }
206
207 template<typename T>
208 size_t list<T>::size() const
209 {
```

```
210     return siz_;
211 }
212
213 template<typename T>
214 bool list<T>::contains(const T &elem) const
215 {
216         node *iter;
217
218     for (iter = first_; iter != 0; iter = iter->next_)
219         if (elem == iter->data_)
220             return true;
221     return false;
222 }
223
224 template<typename T>
225 bool list<T>::empty() const
226 {
227     return first_ ? false : true;
228 }
229
230 template<typename T>
231 void list<T>::insert(const T &t)
232 {
233     node *p = new node(t);
234     p->next_ = first_;
235     p->prev_ = 0;
236
237     if (first_)
238         first_->prev_ = p;
239     first_ = p;
240     if (!last_)
241         last_ = p;
242
243     siz_++;
244 }
245
246 template<typename T>
247 void list<T>::insert_after(const T &t, iterator const &it)
248 {
249     node *nuevo = new node(t);
250     node *actual = it.actual_;
251
252     if (actual == 0)
253     {
254         if (first_ != 0)
255             std::abort();
256         first_ = nuevo;
257         last_ = nuevo;
258     }
259     else
260     {
261         if (actual->next_ != 0)
262             actual->next_->prev_ = nuevo;
263         nuevo->next_ = actual->next_;
264         nuevo->prev_ = actual;
265         actual->next_ = nuevo;
266         if (last_ == actual)
267             last_ = nuevo;
268     }
269
270     siz_++;
271 }
272
```

```
273  template<typename T>
274  void list<T>::insert_before(const T &t, iterator const &it)
275  {
276          node *nuevo = new node(t);
277          node *actual = it.actual_;
278
279          if (actual == 0)
280          {
281                  if (first_ != 0)
282                          std::abort();
283                  first_ = nuevo;
284                  last_ = nuevo;
285          }
286          else
287          {
288                  if (actual->prev_ != 0)
289                          actual->prev_->next_ = nuevo;
290                  nuevo->next_ = actual;
291                  nuevo->prev_ = actual->prev_;
292                  actual->prev_ = nuevo;
293                  if (first_ == actual)
294                          first_ = nuevo;
295          }
296
297      siz_++;
298  }
299
300  template<typename T>
301  void list<T>::erase(const T &t)
302  {
303
304          node *iter, *sig=0;
305
306          for (iter = first_; iter != 0; iter = sig)
307      {
308          sig = iter->next_;
309          if (t == iter->data_)
310          {
311              node *ant = iter->prev_;
312              if (ant == 0)
313                  first_ = sig;
314              else
315                  ant->next_ = sig;
316              if (sig == 0)
317                  last_ = ant;
318              else
319                  sig->prev_ = ant;
320              delete iter;
321
322              siz_--;
323          }
324      }
325  }
326
327  template<typename T>
328  void list<T>::destroy(iterator q)
329  {
330     if (q.actual_)
331      {
332          node* paux = q.actual_;
333          q = q.go_forward();
334          delete paux;
335          destroy (q);
```

```
336         }
337     }
338
339
340     template<typename T>
341     typename list<T>::iterator list<T>::last() const
342     {
343         return typename list<T>::iterator(last_);
344     }
345
346     template<typename T>
347     typename list<T>::iterator list<T>::first() const
348     {
349             return typename list<T>::iterator(first_);
350     }
351
352     template<typename T>
353     list<T> const &list<T>::operator=(list const &orig)
354     {
355         node *iter;
356         node *sig;
357         node *ant;
358
359         if (this != &orig)
360         {
361             for (iter = first_; iter != 0; )
362             {
363                 sig = iter->next_;
364                 delete iter;
365                 iter = sig;
366             }
367
368             first_ = 0;
369             last_ = 0;
370
371             for (iter = orig.first_, ant = 0; iter != 0; iter = iter->next_)
372             {
373                 node *nuevo = new node(iter->data_);
374                 nuevo->prev_ = ant;
375                 nuevo->next_ = 0;
376                 if (ant != 0)
377                     ant->next_ = nuevo;
378                 if (first_ == 0)
379                     first_ = nuevo;
380                 ant = nuevo;
381             }
382             last_ = ant;
383             siz_ = orig.siz_;
384         }
385
386         return *this;
387     }
388
389     #endif
```

5.8. Transaction.cc

```
1   #include "Transaction.h"
2
3   Transaction::Transaction()
4   {
```

```
5       n_tx_in = "0";
6       n_tx_out = "0";
7   }
8
9   Transaction::~Transaction()
10  {
11  }
12
13  input_t Transaction::get_input(int position) const
14  {
15
16      return inputs[position];
17  }
18
19  output_t Transaction::get_output(int position) const
20  {
21
22      return outputs[position];
23  }
24
25  string Transaction::hash_transaction()
26  {
27      string hash = n_tx_in + "\n";
28
29      for (int i = 0 ; i < stoi (n_tx_in) ; i++)
30          hash += inputs[i].tx_id + " " + inputs[i].idx + " " + inputs[i].addr +
                "\n";
31
32      hash += n_tx_out + "\n";
33
34      int j;
35      for (j = 0 ; j < stoi (n_tx_out) -1; j++)
36          hash += outputs[j].value + " " + outputs[j].addr + "\n";
37
38      hash += outputs[j].value + " " + outputs[j].addr + "\n";
39
40      return sha256(sha256(hash));
41  }
42
43  //Recibe un Vector<string> que contiene nombre de destinatario y monto de la
        transaccion
44  void Transaction::load_transaction_for_mempool(Vector <string> & transfers,
        float total_value, float src_value, string tx_id_ , int idx_)
45  {
46      n_tx_in="1";
47
48      n_tx_out = to_string(((transfers.position(END_OF_ARGS)-1)/2)+1);
49      inputs[0].tx_id = tx_id_; //cargue tx_id de la transaccion
50      inputs[0].idx = to_string(idx_);
51      inputs[0].addr = sha256(sha256(transfers[0]));
52
53      int i, j;
54      for (i = 0, j=0 ; j < stoi (n_tx_out)-1 ; i = i + 2, j++) //iterando sobre
            el vector de outputs
55      {
56          outputs[j].addr = sha256(sha256(transfers[i+1]));
57          outputs[j].value = transfers[i+2];
58      }
59      //Para el ultimo Output:
60      outputs[j].addr = inputs[0].addr;
61      outputs[j].value = to_string(src_value-total_value);
62  }
63
```

```
64  //Operador para imprimir en output
65  ostream & operator<<( ostream &output, const Transaction &txn)
66  {
67      output<<txn.n_tx_in<<"\n";
68      for (int i = 0; i < std::stoi (txn.n_tx_in); i++)
69      {
70          output<<txn.inputs[i].tx_id<<" ";
71          output<<txn.inputs[i].idx<<" ";
72          output<<txn.inputs[i].addr<<"\n";
73      }
74      output<<txn.n_tx_out<<"\n";
75      for (int i = 0; i < std::stoi (txn.n_tx_out); i++)
76      {
77          output<<txn.outputs[i].value<<" ";
78          output<<txn.outputs[i].addr<<"\n";
79      }
80      return output;
81  }
82
83  //Operador para cargar una transaccion desde input
84  istream & operator>>( istream &input, Transaction &txn)
85  {
86      char delim = ' ';
87      string aux_str;
88      getline(input,aux_str);
89
90      if (is_int(aux_str) == false)
91      {
92          cerr << "Invalid n_tx_in format: "
93               << aux_str
94               << ". "
95               << endl;
96          exit(1);
97      }
98      txn.set_n_tx_in(aux_str);
99
100     // Carga los inputs:
101     Vector <input_t> aux_inputs(stoi(txn.get_n_tx_in()));
102
103     for (int i = 0; i < stoi(txn.get_n_tx_in()); i++)
104     {
105         cout << "Entre al for de cargar inputs en transacciones" << endl;
106         getline(input, aux_inputs[i].tx_id, delim);
107
108         if (is_alphanumeric(aux_inputs[i].tx_id) == false || (aux_inputs[i].
                tx_id).length() != HASH_LENGTH)
109         {
110             cerr << "Invalid tx_id format: "
111                 << aux_inputs[i].tx_id
112                 << ". "
113                  << endl;
114             exit(1);
115         }
116
117         getline(input, aux_inputs[i].idx, delim);
118
119         if (is_int(aux_inputs[i].idx) == false)
120         {
121             cerr << "Invalid idx format: "
122                 << aux_inputs[i].idx
123                 << ". "
124                  << endl;
125             exit(1);
```

```
126             }
127
128             getline(input, aux_inputs[i].addr);
129
130             if (is_alphanumeric(aux_inputs[i].addr) == false || (aux_inputs[i].addr
                    ).length() != HASH_LENGTH)
131             {
132                 cerr << "Invalid addr format: "
133                    << aux_inputs[i].addr
134                    << ". "
135                     << endl;
136                 exit(1);
137             }
138         }
139         txn.set_inputs(aux_inputs);
140         // Carga los outputs:
141         getline(input,aux_str);
142         if (is_int(aux_str) == false)
143         {
144             cerr << "Invalid n_tx_out format: "
145                  << aux_str
146                  << ". "
147                  << endl;
148             exit(1);
149         }
150         txn.set_n_tx_out(aux_str);
151         // Carga el vector de outputs:
152         Vector <output_t> aux_outputs(stoi(txn.get_n_tx_out()));
153
154         float total_txn_value=0;
155         for (int j = 0; j < stoi (txn.get_n_tx_out()); j++)
156         {
157             getline(input, aux_outputs[j].value, delim);
158
159             if (is_float_or_double(aux_outputs[j].value) == false)
160             {
161                 cerr << "Invalid value format: "
162                      << aux_outputs[j].value
163                     << ". "
164                     << endl;
165                 exit(1);
166             }
167             total_txn_value+=stof(aux_outputs[j].value);
168
169             getline(input, aux_outputs[j].addr);
170             if ( is_alphanumeric(aux_outputs[j].addr) == false||(aux_outputs[j].
                    addr).length() != HASH_LENGTH)
171             {
172                 cerr << "Invalid addr format: "
173                      << aux_outputs[j].addr
174                      << ". "
175                      << endl;
176                 exit(1);
177             }
178         }
179         txn.set_outputs(aux_outputs);
180
181         return input;
182     }
183
184     void Transaction::load_Transaction(istream * is, float & total_txn_value)
185     {
186         char delim = ' ';
```

```
187
188         getline(*is,n_tx_in);//Carga de cantidad de inputs
189         if (is_int(n_tx_in) == false)
190         {
191             cerr << "Invalid n_tx_in format: "
192                  << n_tx_in
193                  << ". "
194                  << endl;
195             exit(1);
196         }
197
198         Vector <input_t> aux(stoi(n_tx_in));
199         inputs = aux;
200         for (int i = 0; i < stoi (n_tx_in); i++)//Carga de los inputs y validacion
                de formato
201         {
202             getline(*is, inputs[i].tx_id, delim);
203             if (is_alphanumeric(inputs[i].tx_id) == false || (inputs[i].tx_id).
                    length() != HASH_LENGTH)
204             {
205                 cerr << "Invalid tx_id format: "
206                    << inputs[i].tx_id
207                    << ". "
208                     << endl;
209                 exit(1);
210             }
211
212             getline(*is, inputs[i].idx, delim);
213             if (is_int(inputs[i].idx) == false)
214             {
215                 cerr << "Invalid idx format: "
216                    << inputs[i].idx
217                    << ". "
218                     << endl;
219                 exit(1);
220             }
221
222             getline(*is, inputs[i].addr);
223             if (is_alphanumeric(inputs[i].addr) == false || (inputs[i].addr).length
                    () != HASH_LENGTH)
224             {
225                 cerr << "Invalid addr format: "
226                    << inputs[i].addr
227                    << ". "
228                     << endl;
229                 exit(1);
230             }
231         }
232
233         getline(*is,n_tx_out); //Carga de cantidad de outputs
234         if (is_int(n_tx_out) == false)
235         {
236             cerr << "Invalid n_tx_out format: "
237                  << n_tx_out
238                  << ". "
239                  << endl;
240             exit(1);
241         }
242
243         Vector <output_t> aux1(stoi(n_tx_out));
244         outputs = aux1;
245
246         total_txn_value=0;
```

```
247     for (int j = 0; j < stoi (n_tx_out); j++)//Carga de los outputs y
            validacion de formato
248     {
249         getline(*is, outputs[j].value, delim);
250         if (is_float_or_double(outputs[j].value) == false)
251         {
252             cerr << "Invalid value format: "
253                  << outputs[j].value
254                 << ". "
255                 << endl;
256             exit(1);
257         }
258
259         total_txn_value+=stof(outputs[j].value);
260
261         getline(*is, outputs[j].addr);
262         if ( is_alphanumeric(outputs[j].addr) == false||(outputs[j].addr).
                length() != HASH_LENGTH)
263         {
264             cerr << "Invalid addr format: "
265                  << outputs[j].addr
266                  << ". "
267                  << endl;
268             exit(1);
269         }
270     }
271 }
```

5.9. Utilities.cc

```
1  #include "Utilities.h"
2
3  string Hex2Bin(const string& s)//transforma de hexa a binario una string
4  {
5      string bin_number="";
6
7      for(size_t i=0; i< s.length(); i++)
8      {
9          stringstream ss;
10         unsigned n;
11         ss << hex << s[i];
12         ss >> n;
13
14         bitset<4> b(n);
15         bin_number.append( b.to_string() );
16     }
17
18     return bin_number;
19 }
20
21 bool is_float_or_double(const string &str)
22 {
23
24     return str.find_first_not_of("0123456789.") == string::npos;
25 }
26
27 bool is_alphanumeric(const string &str)
28 {
29
30     return str.find_first_not_of("0123456789qwertyuiopasdfghjklzxcvbnm") ==
           string::npos;
```

```
31  }
32
33  bool is_alphabetic(const string &str)
34  {
35
36      return str.find_first_not_of("qwertyuiopasdfghjklzxcvbnm") == string::npos;
37  }
38
39  bool is_alphanumeric_point(const string &str)
40  {
41
42      return str.find_first_not_of("0123456789qwertyuiopasdfghjklzxcvbnm._-") ==
            string::npos;
43  }
44
45  bool is_int(const string &str)
46  {
47
48      return str.find_first_not_of("0123456789") == string::npos;
49  }
50
51  bool is_point(const string &str)
52  {
53
54      return str.find_first_not_of(".") == string::npos;
55  }
56
57  int count_points(string s)
58  {
59      int count = 0;
60
61      for (size_t i = 0; i < s.length(); i++)
62      {
63          if (s[i] == '.')
64              count++;
65      }
66
67    return count;
68  }
69
70  void valid_init_args(Vector<string> & inits)
71  {
72      if (inits[3]!=END_OF_ARGS) //Los argumentos son <user> <value> <bits>
73      {
74          cerr << "Invalid amount of arguments for command init. "
75               << endl;
76          exit(1);
77      }
78
79      if (is_alphabetic(inits[0]) == false) //inits[0] = user
80      {
81          cerr << "non-valid user: "
82               << inits[0]
83               << "."
84               << endl;
85          exit(1);
86      }
87
88      if (is_float_or_double(inits[1]) == false) //inits[1] = value
89      {
90          cerr << "non-valid value: "
91               << inits[1]
92               << "."
```

```
93                  << endl;
94              exit(1);
95          }
96
97          if (is_int(inits[2]) == false) //inits[2] = dificultad; // no se admite
                dificultad negativa
98          {
99              cerr << "non-valid difficulty: "
100                 << inits[2]
101                 << "."
102                 << endl;
103             exit(1);
104         }
105     }
106
107     void valid_transfer_args_amount(Vector <string> & transfers)
108     {
109         // Validacion contra impar y contra minima cantidad de arguemntos
110         if ( transfers.position(END_OF_ARGS) % 2 == 0 || transfers.position(
                END_OF_ARGS) < MIN_AMOUNT_TRANSFER_ARGS)
111         {
112             cerr << "Invalid amount of arguments for transfers."
113                 << endl;
114             exit(1);
115         }
116         if (is_alphabetic(transfers[0]) == false)
117         {
118             cerr << "non-valid source : "
119                 << transfers[0]
120                 << "."
121                 << endl;
122             exit(1);
123         }
124     }
125
126     void valid_transfer_args( Vector <string> transfers, float & total_value,
            string & src_value)
127     {
128         //Se chequea DESTINO y VALUE, ademas se tomam la suma de los values para
                despues ver si tiene saldo suficiente
129         //Se valida el formato de los argumentos, la doble inclusion de un usuario
                en una misma transferencia
130         // y que el saldo de la transferencia no supere al balance total.
131         for (int i = 1; i < transfers.position(END_OF_ARGS) ; i = i + 2)
132         {
133             // Validacion ante doble-inclusion
134             if (transfers.search(transfers[i]) != 1)
135             {
136                 cerr << "FAIL: Repeated destination"
137                     << transfers[i]
138                     << "."
139                     << endl;
140                 return;
141             }
142
143             if (is_alphabetic(transfers[i]) == false)
144             {
145                 cerr << "non-valid destination : "
146                     << transfers[i]
147                     << "."
148                     << endl;
149                 exit(1);
150             }
```

```
151
152         if (is_float_or_double(transfers[i+1]) == false || is_point(transfers[i
                +1]) == true  || count_points(transfers[i+1]) > 1)
153         {
154             cout << "non-valid value: "
155                  << transfers[i+1]
156                  << endl;
157             exit(1);
158         }
159
160         total_value+=stof(transfers[i+1]);
161
162         if(total_value > stof(src_value))
163         {
164             cerr << "FAIL: not sufficient funds. "
165                  << endl;
166             return;
167         }
168     }
169 }
170
171 Vector <string> split_string(string & args, string & delimiter)
172 {
173     size_t pos = 0;
174     int i = 0;
175     string token;
176     Vector <string> transfers;
177
178     while ((pos = args.find(delimiter)) != string::npos)
179     {
180         token = args.substr(0, pos);
181         if (i == transfers.get_size())
182         {
183             Vector <string> aux(2*(transfers.get_size()) + 1, transfers);
184             transfers = aux;
185         }
186         transfers[i] = token;
187         i++;
188         args.erase(0, pos + delimiter.length());
189     }
190     transfers[i] = args;
191     transfers[i+1] = END_OF_ARGS;
192
193     return transfers;
194 }
```

5.10. Utilities.h

```
1  #ifndef UTILITIES_H_INCLUDED_
2  #define UTILITIES_H_INCLUDED_
3
4  #include "Vector.h"
5
6  #include <cstring>
7  #include <ostream>
8  #include <string>
9  #include <string.h>
10 #include <string_view>
11 #include <iostream>
12 #include <istream>
13 #include <fstream>
```

```
14  #include <iomanip>
15  #include <sstream>
16  #include <cstdlib>
17  #include <bitset>
18  #include <time.h>
19
20  #define END_OF_ARGS "E.n.D.o.F.a.R.g.S."
21  #define MIN_AMOUNT_TRANSFER_ARGS 3
22
23  using namespace std;
24
25  string Hex2Bin(const string&);
26
27  bool is_float_or_double(const string &);
28  bool is_alphanumeric(const string &);
29  bool is_alphabetic(const string &);
30  bool is_alphanumeric_point(const string &);
31  bool is_int(const string &);
32  bool is_point(const string &);
33  int count_points(string);
34  void valid_init_args(Vector<string> &);
35  void valid_transfer_args_amount(Vector <string> &);
36  void valid_transfer_args( Vector <string>, float &, string &);
37  Vector <string> split_string(string &, string &);
38
39  #endif
```

5.11. Command.cc

```
1   #include "Command.h"
2
3   Command::Command(command_t * comm):command_table(comm)
4   {
5   }
6
7   void Command::parse(istream *is)
8   {
9       string commands_names[] = {"init", "transfer", "mine", "balance", "block",
            "txn", "load", "save"};
10      string line, line_name, line_args, delim = " ";
11      int found=0;
12
13      while (getline(*is,line).eof() == false)
14      {
15          if (line.empty() == true)
16          {
17              cout << "Empty line in commands \n";
18              continue;
19          }
20
21          line_name = line.substr(0,line.find(delim));
22          found = 0;
23          for (int i = 0; i < AMOUNT_COMMANDS ; i++)
24          {
25              if (line_name == commands_names[i])
26                  found++;
27          }
28
29          if(found == 0)
30          {
31              cerr << "Command not found: "
```

```
32                  << "--"
33                  << line_name
34                  << "\n";
35              exit(1);
36          }
37
38          line_args = line.erase(0, line.find(delim) + DELIM_LENGTH);
39
40          for(command_t *com = command_table; com->name !=0; ++com)
41          {
42              if (line_name == string(com->name))
43              {
44                  if (line_args.length() == 0 || line_args == line_name)
45                  {
46                      cerr << "Command requires argument: "
47                           << "--"
48                           << com->name
49                           << "\n";
50                      exit(1);
51                  }
52                  com->parse(line_args);//Pasaje mediante puntero a funcion
53                  break;
54              }
55
56          }
57      }
58
59      if (line.empty() == false)
60      {
61
62          line_name = line.substr(0,line.find(delim));
63
64          for (int i = 0; i < AMOUNT_COMMANDS ; i++)
65          {
66              if (line_name == commands_names[i])
67                  found++;
68          }
69
70          if(found == 0)
71          {
72              cerr << "Command not found: "
73                   << "--"
74                   << line_name
75                   << "\n";
76              exit(1);
77          }
78
79          line_args = line.erase(0, line.find(delim) + DELIM_LENGTH);
80
81          for(command_t *com = command_table; com->name !=0; ++com)
82          {
83              if (line_name == string(com->name))
84              {
85                  if (line_args.length() == 0)
86                  {
87                      cerr << "Command requires argument: "
88                           << "--"
89                           << com->name
90                           << "\n";
91                      exit(1);
92                  }
93                  com->parse(line_args);//Pasaje mediante puntero a funcion
94                  break;
```

```
95              }
96
97          }
98      }
99
100     return;
101 }
```

5.12. Command.h

```
1   #ifndef COMMAND_H_INCLUDED
2   #define COMMAND_H_INCLUDED
3   #include "Block.h"
4   #include "Transaction.h"
5   #include "main.h"
6   #include <string>
7   #include <cstdlib>
8   #include <iostream>
9   #include <cstring>
10
11  using namespace std;
12
13  struct command_t
14  {
15   int has_arg;
16   const char *name;
17   void (*parse)(string &); // Puntero a funcion de opciones
18  };
19
20  class Command
21  {
22  private:
23      command_t * command_table;
24
25  public:
26      Command(command_t *);
27
28      void parse(istream *is);
29
30  };
31
32  #endif
```

5.13. MerkleTree.cc

```
1   #include "MerkleTree.h"
2
3   MerkleTree::MerkleTree(Vector <TreeNode *> leaves)
4   {
5       //leaves es el Vector que contiene los punteros a TreeNode que contienen
            los hashes de las transacciones
6       if(leaves.get_size() == 0)
7       {
8           cerr << "FAIL: Not able to calculate txns_hash."
9                << "\n";
10          exit(1);
11      }
12      TreeNode * p = NULL;
```

```
13      while(leaves.get_size() != 1)
14      {
15          Vector <TreeNode *> nodes(leaves.get_size()/2 + leaves.get_size()%2);
                //Se crea el nivel de arriba del arbol
16
17          for( int i = 0, n = 0; i < leaves.get_size(); i = i+2, n++)
18          {
19              if( i != leaves.get_size() -1 )//Se fija si quedan elementos por
                    analizar en el Vector por si es impar
20              {
21                  nodes[n] = new TreeNode( sha256(sha256( leaves[i]->get_hash() +
                        leaves[i+1]->get_hash() )));
22                  nodes[n]->set_left(leaves[i]);
23                  nodes[n]->set_rigth(leaves[i+1]);
24              }
25              else //En caso de quedar un elemento solo, se hace la suma del hash
                     de ese elemento consigo mismo para cargar al padre
26              {
27                  nodes[n] = new TreeNode(sha256(sha256( leaves[i]->get_hash() +
                        leaves[i]->get_hash())));
28                  nodes[n]->set_left(leaves[i]);
29                  nodes[n]->set_rigth(p);
30              }
31          }
32          leaves = nodes;
33      }
34      root = leaves[0];
35  }
36
37  MerkleTree::~MerkleTree()
38  {
39
40      deleteMerkleTree(root);
41  }
42
43  void MerkleTree::deleteMerkleTree(TreeNode * father)
44  {
45      if (father)//Si es distinto de nulo entra
46      {
47          deleteMerkleTree(father->get_left());
48          deleteMerkleTree(father->get_rigth());
49          delete father;
50          father = NULL;
51      }
52  }
```

5.14. MerkleTree.h

```
1   #ifndef MERKLE_TREE_INCLUDED
2   #define MERKLE_TREE_INCLUDED
3
4   #include <string>
5   #include <iostream>
6
7   #include "Vector.h"
8   #include "TreeNode.h"
9   #include "sha256.h"
10
11  using namespace std;
12
13  class MerkleTree
```

```
14  {
15  private:
16      TreeNode * root;
17      void deleteMerkleTree(TreeNode *);
18
19  public:
20      MerkleTree(Vector <TreeNode *>);
21      ~MerkleTree();
22      TreeNode *get_root(){return root;};
23  };
24
25  #endif
```

5.15. TreeNode.h

```
1   #ifndef TREENODE_INCLUDED
2   #define TREENODE_INCLUDED
3
4   #include "MerkleTree.h"
5   #include "Vector.h"
6
7   using namespace std;
8
9   class TreeNode
10  {
11      string hash;
12      TreeNode * left;
13      TreeNode * rigth;
14
15  public:
16      TreeNode(string data){hash = data;};
17      ~TreeNode(){};
18
19      string get_hash(){return hash;};
20      TreeNode * get_left(){return left;};
21      TreeNode * get_rigth(){return rigth;};
22      void set_left(TreeNode * left_){left = left_;};
23      void set_rigth(TreeNode * rigth_){rigth = rigth_;};
24
25  };
26
27  #endif
```

5.16. cmdline.h

```
1   #ifndef _CMDLINE_H_INCLUDED_
2   #define _CMDLINE_H_INCLUDED_
3
4   #include <string>
5   #include <cstring>
6   #include <string.h>
7   #include <iostream>
8
9   using namespace std;
10
11  #define OPT_DEFAULT   0
12  #define OPT_SEEN      1
13  #define OPT_MANDATORY 2
```

```
14
15  struct option_t {
16      int has_arg;
17      const char *short_name;
18      const char *long_name;
19      const char *def_value;
20      void (*parse)(string const &); // Puntero a funcion de opciones
21      int flags;
22  };
23
24  class cmdline {
25      /* Este atributo apunta a la tabla que describe todas
26         las opciones a procesar. Por el momento, solo puede
27         ser modificado mediante contructor, y debe finalizar
28         con un elemento nulo.
29      */
30      option_t *option_table;
31
32      /* El constructor por defecto cmdline::cmdline(), es
33         privado, para evitar construir "parsers" (analizador
34         sintactico, recibe una palabra y lo interpreta en
35         una accion dependiendo su significado para el programa)
36         sin opciones. Es decir, objetos de esta clase sin opciones.
37      */
38
39      cmdline();
40      int do_long_opt(const char *, const char *);
41      int do_short_opt(const char *, const char *);
42  public:
43      cmdline(option_t *);
44      void parse(int, char * const []);
45  };
46
47  #endif
```

5.17. cmdline.cc

```
1   #include <string>
2   #include <cstdlib>
3   #include <iostream>
4   #include <cstring>
5
6   #include "cmdline.h"
7
8   using namespace std;
9
10  cmdline::cmdline()
11  {
12  }
13
14  cmdline::cmdline(option_t *table) : option_table(table)
15  {
16  }
17
18  void
19  cmdline::parse(int argc, char * const argv[])
20  {
21  #define END_OF_OPTIONS(p) \
22      ((p)->short_name == 0   \
23       && (p)->long_name == 0 \
24       && (p)->parse == 0)
```

```
25      /* Primer pasada por la secuencia de opciones: marcamos
26         todas las opciones, como no procesadas. Ver codigo de
27         abajo.
28      */
29      for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op)
30          op->flags &= ~OPT_SEEN;
31
32      /* Recorremos el arreglo argv. En cada paso, vemos
33         si se trata de una opcion corta, o larga. Luego,
34         llamamos a la funcion de parseo correspondiente.
35      */
36      for (int i = 1; i < argc; ++i) {
37          /* Todos los parametros de este programa deben
38             pasarse en forma de opciones. Encontrar un
39             parametro no-opcion es un error.
40          */
41          if (argv[i][0] != '-') {
42              cerr << "Invalid non-option argument: "
43                   << argv[i]
44                   << endl;
45              exit(1);
46          }
47
48          /* Usamos "--" para marcar el fin de las
49             opciones; todo los argumentos que puedan
50             estar a continuacion no son interpretados
51             como opciones.
52          */
53          if (argv[i][1] == '-'
54              && argv[i][2] == 0)
55              break;
56
57          /* Finalmente, vemos si se trata o no de una
58             opcion larga; y llamamos al metodo que se
59             encarga de cada caso.
60          */
61          if (argv[i][1] == '-')
62              i += do_long_opt(&argv[i][2], argv[i + 1]);
63          else
64              i += do_short_opt(&argv[i][1], argv[i + 1]);
65      }
66
67      /* Segunda pasada: procesamos aquellas opciones que,
68         (1) no hayan figurado explicitamente en la linea
69         de comandos, y (2) tengan valor por defecto.
70      */
71      for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op) {
72  #define OPTION_NAME(op) \
73      (op->short_name ? op->short_name : op->long_name)
74          if (op->flags & OPT_SEEN)
75              continue;
76          if (op->flags & OPT_MANDATORY) {
77              cerr << "Option "
78                   << "-"
79                   << OPTION_NAME(op)
80                   << " is mandatory."
81                   << "\n";
82              exit(1);
83          }
84          if (op->def_value == 0)
85              continue;
86          op->parse(string(op->def_value));
87      }
```

```
88  }
89
90  int
91  cmdline::do_long_opt(const char *opt, const char *arg)
92  {
93      /* Recorremos la tabla de opciones, y buscamos la
94         entrada larga que se corresponda con la opcion de
95         linea de comandos. De no encontrarse, indicamos el
96         error.
97      */
98      for (option_t *op = option_table; op->long_name != 0; ++op) {
99          if (string(opt) == string(op->long_name)) {
100             /* Marcamos esta opcion como usada en
101                forma explicita, para evitar tener
102                que inicializarla con el valor por
103                defecto.
104             */
105             op->flags |= OPT_SEEN;
106
107             if (op->has_arg) {
108                 /* Como se trada de una opcion
109                    con argumento, verificamos que
110                    el mismo haya sido provisto.
111                 */
112                 if (arg == 0) {
113                     cerr << "Option requires argument: "
114                          << "--"
115                          << opt
116                          << "\n";
117                     exit(1);
118                 }
119                 op->parse(string(arg));
120                 return 1;
121             } else {
122                 // Opcion sin argumento.
123                 op->parse(string(""));
124                 return 0;
125             }
126         }
127     }
128
129     // Error: opcion no reconocida. Imprimimos un mensaje
130     // de error, y finalizamos la ejecucion del programa.
131     cerr << "Unknown option: "
132          << "--"
133          << opt
134          << "."
135          << endl;
136     exit(1);
137
138     /* Algunos compiladores se quejan con funciones que
139        logicamente no pueden terminar, y que no devuelven
140        un valor en esta ultima parte.
141     */
142     return -1;
143 }
144
145 int
146 cmdline::do_short_opt(const char *opt, const char *arg)
147 {
148     option_t *op;
149
150     /* Recorremos la tabla de opciones, y buscamos la
```

```
151         entrada corta que se corresponda con la opcion de
152         linea de comandos. De no encontrarse, indicamos el
153         error.
154      */
155      for (op = option_table; op->short_name != 0; ++op) {
156          if (string(opt) == string(op->short_name)) {
157              /* Marcamos esta opcion como usada en
158                 forma explicita, para evitar tener
159                 que inicializarla con el valor por
160                 defecto.
161              */
162              op->flags |= OPT_SEEN;
163
164              if (op->has_arg) {
165                  /* Como se trata de una opcion
166                     con argumento, verificamos que
167                     el mismo haya sido provisto.
168                  */
169                  if (arg == 0) {
170                      cerr << "Option requires argument: "
171                           << "-"
172                           << opt
173                           << "\n";
174                      exit(1);
175                  }
176                  op->parse(string(arg));
177                  return 1;
178              } else {
179                  // Opcion sin argumento.
180                  op->parse(string(""));
181                  return 0;
182              }
183          }
184      }
185
186      /* Error: opcion no reconocida. Imprimimos un mensaje
187         de error, y finalizamos la ejecucion del programa.
188      */
189      cerr << "Unknown option: "
190           << "-"
191           << opt
192           << "."
193           << endl;
194      exit(1);
195
196      /* Algunos compiladores se quejan con funciones que
197         logicamente no pueden terminar, y que no devuelven
198         un valor en esta ultima parte.
199      */
200      return -1;
201  }
```

5.18. sha256.h

```
1  #ifndef SHA256_H_INCLUDED_
2  #define SHA256_H_INCLUDED_
3  #include <string.h>
4  #include <string>
5  #include <cstring>
6
7  class SHA256
```

```
{
protected:
    typedef unsigned char uint8;
    typedef unsigned int uint32;
    typedef unsigned long long uint64;

    const static uint32 sha256_k[];
    static const unsigned int SHA224_256_BLOCK_SIZE = (512/8);
public:
    void init();
    void update(const unsigned char *message, unsigned int len);
    void final(unsigned char *digest);
    static const unsigned int DIGEST_SIZE = ( 256 / 8);

protected:
    void transform(const unsigned char *message, unsigned int block_nb);
    unsigned int m_tot_len;
    unsigned int m_len;
    unsigned char m_block[2*SHA224_256_BLOCK_SIZE];
    uint32 m_h[8];
};

std::string sha256(std::string input);

#define SHA2_SHFR(x, n)    (x >> n)
#define SHA2_ROTR(x, n)   ((x >> n) | (x << ((sizeof(x) << 3) - n)))
#define SHA2_ROTL(x, n)   ((x << n) | (x >> ((sizeof(x) << 3) - n)))
#define SHA2_CH(x, y, z)  ((x & y) ^ (~x & z))
#define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
#define SHA256_F1(x) (SHA2_ROTR(x,  2) ^ SHA2_ROTR(x, 13) ^ SHA2_ROTR(x, 22))
#define SHA256_F2(x) (SHA2_ROTR(x,  6) ^ SHA2_ROTR(x, 11) ^ SHA2_ROTR(x, 25))
#define SHA256_F3(x) (SHA2_ROTR(x,  7) ^ SHA2_ROTR(x, 18) ^ SHA2_SHFR(x,  3))
#define SHA256_F4(x) (SHA2_ROTR(x, 17) ^ SHA2_ROTR(x, 19) ^ SHA2_SHFR(x, 10))
#define SHA2_UNPACK32(x, str)                 \
{                                             \
    *((str) + 3) = (uint8) ((x)      );       \
    *((str) + 2) = (uint8) ((x) >>  8);       \
    *((str) + 1) = (uint8) ((x) >> 16);       \
    *((str) + 0) = (uint8) ((x) >> 24);       \
}
#define SHA2_PACK32(str, x)                   \
{                                             \
    *(x) =   ((uint32) *((str) + 3)      )    \
           | ((uint32) *((str) + 2) <<  8)    \
           | ((uint32) *((str) + 1) << 16)    \
           | ((uint32) *((str) + 0) << 24);   \
}
#endif
```

5.19. sha256.cc

```
#include <cstring>
#include <fstream>
#include "sha256.h"

const unsigned int SHA256::sha256_k[64] = //UL = uint32
            {0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
             0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
             0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
             0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
             0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
```

```
11              0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
12              0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
13              0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
14              0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
15              0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
16              0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
17              0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
18              0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
19              0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
20              0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
21              0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2};
22
23  void SHA256::transform(const unsigned char *message, unsigned int block_nb)
24  {
25      uint32 w[64];
26      uint32 wv[8];
27      uint32 t1, t2;
28      const unsigned char *sub_block;
29      int i;
30      int j;
31      for (i = 0; i < (int) block_nb; i++) {
32          sub_block = message + (i << 6);
33          for (j = 0; j < 16; j++) {
34              SHA2_PACK32(&sub_block[j << 2], &w[j]);
35          }
36          for (j = 16; j < 64; j++) {
37              w[j] =  SHA256_F4(w[j -  2]) + w[j -  7] + SHA256_F3(w[j - 15]) + w
                    [j - 16];
38          }
39          for (j = 0; j < 8; j++) {
40              wv[j] = m_h[j];
41          }
42          for (j = 0; j < 64; j++) {
43              t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
44                   + sha256_k[j] + w[j];
45              t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
46              wv[7] = wv[6];
47              wv[6] = wv[5];
48              wv[5] = wv[4];
49              wv[4] = wv[3] + t1;
50              wv[3] = wv[2];
51              wv[2] = wv[1];
52              wv[1] = wv[0];
53              wv[0] = t1 + t2;
54          }
55          for (j = 0; j < 8; j++) {
56              m_h[j] += wv[j];
57          }
58      }
59  }
60
61  void SHA256::init()
62  {
63      m_h[0] = 0x6a09e667;
64      m_h[1] = 0xbb67ae85;
65      m_h[2] = 0x3c6ef372;
66      m_h[3] = 0xa54ff53a;
67      m_h[4] = 0x510e527f;
68      m_h[5] = 0x9b05688c;
69      m_h[6] = 0x1f83d9ab;
70      m_h[7] = 0x5be0cd19;
71      m_len = 0;
72      m_tot_len = 0;
```

```
73  }
74
75  void SHA256::update(const unsigned char *message, unsigned int len)
76  {
77      unsigned int block_nb;
78      unsigned int new_len, rem_len, tmp_len;
79      const unsigned char *shifted_message;
80      tmp_len = SHA224_256_BLOCK_SIZE - m_len;
81      rem_len = len < tmp_len ? len : tmp_len;
82      memcpy(&m_block[m_len], message, rem_len);
83      if (m_len + len < SHA224_256_BLOCK_SIZE) {
84          m_len += len;
85          return;
86      }
87      new_len = len - rem_len;
88      block_nb = new_len / SHA224_256_BLOCK_SIZE;
89      shifted_message = message + rem_len;
90      transform(m_block, 1);
91      transform(shifted_message, block_nb);
92      rem_len = new_len % SHA224_256_BLOCK_SIZE;
93      memcpy(m_block, &shifted_message[block_nb << 6], rem_len);
94      m_len = rem_len;
95      m_tot_len += (block_nb + 1) << 6;
96  }
97
98  void SHA256::final(unsigned char *digest)
99  {
100     unsigned int block_nb;
101     unsigned int pm_len;
102     unsigned int len_b;
103     int i;
104     block_nb = (1 + ((SHA224_256_BLOCK_SIZE - 9)
105                      < (m_len % SHA224_256_BLOCK_SIZE)));
106     len_b = (m_tot_len + m_len) << 3;
107     pm_len = block_nb << 6;
108     memset(m_block + m_len, 0, pm_len - m_len);
109     m_block[m_len] = 0x80;
110     SHA2_UNPACK32(len_b, m_block + pm_len - 4);
111     transform(m_block, block_nb);
112     for (i = 0 ; i < 8; i++) {
113         SHA2_UNPACK32(m_h[i], &digest[i << 2]);
114     }
115 }
116
117 std::string sha256(std::string input)
118 {
119     unsigned char digest[SHA256::DIGEST_SIZE];
120     memset(digest,0,SHA256::DIGEST_SIZE);
121
122     SHA256 ctx = SHA256();
123     ctx.init();
124     ctx.update( (unsigned char*)input.c_str(), input.length());
125     ctx.final(digest);
126
127     char buf[2*SHA256::DIGEST_SIZE+1];
128     buf[2*SHA256::DIGEST_SIZE] = 0;
129     for (unsigned int i = 0; i < SHA256::DIGEST_SIZE; i++)
130         sprintf(buf+i*2, "%02x", digest[i]);
131     return std::string(buf);
132 }
```

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:

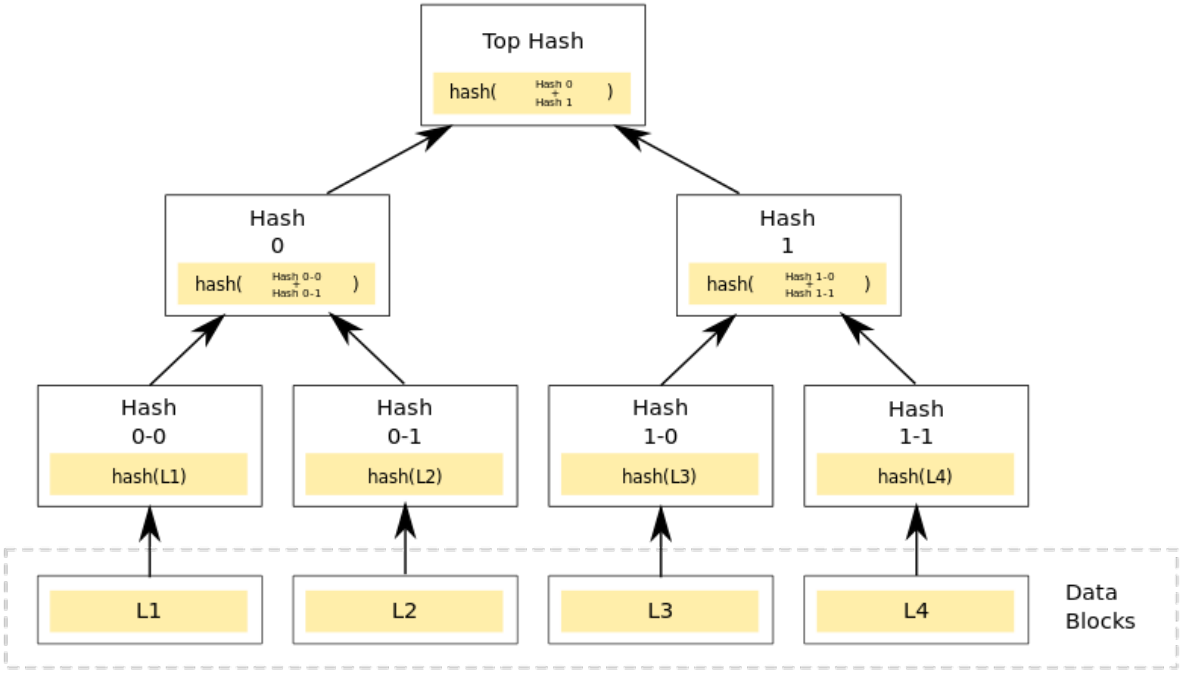

Figura 1: Esquema de un árbol de Merkle (cortesía Wikipedia). El operador +, en este contexto, indica concatenación de hashes y no suma numérica.

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/`.